EL2805 Reinforcement Learning

Summary

Alexandre Proutiere

November 17, 2021

Division of Decision and Control Systems
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

1 Introduction

What is Reinforcement Learning? When is it useful? The underlying objective is to identify an efficient control policy for a dynamical system. Control theory deals with scenarios where the dynamical system and the reward function are known (they have been identified beforehand). RL is concerned with scenarios where the dynamics and/or reward function are initially unknown.

Model-based vs model-free RL. RL algorithms can be categorized into two classes: model-based and model-free. Model-based algorithms aim at identifying a model characterizing the system dynamics and the reward function. Based on this model, one may retrieve the optimal control policy. Model-free algorithms do not explicitly identify a model, but rather directly learn functions of interest such as the Q-function or the value function. Alternatively model-free algorithms learn the value of a given policy, and try to improve it.

Most existing RL approaches have been developed for systems with discrete state and action spaces and whose dynamics are also in discrete time.

2 Markov Decision Processes

MDPs are controlled Markov chains. We define below the state and action space, before actually explaining what controlled Markov chains mean.

State space. The system is characterized at time $t = 1, 2, \ldots$ by its state $s_t \in \mathcal{S}$. Unless otherwise specified, $\mathcal{S}$ is finite and of cardinality S.

Action space. In any state $s \in \mathcal{S}$, the decision maker may select an action $a \in \mathcal{A}_s$, that in turn will impact the collected reward, and the system dynamics. Unless otherwise specficied, $\mathcal{A}_s$ is finite, and we denote by A the cardinality of $\cup_s \mathcal{A}_s$.

Controlled Markov chains. MDPs are controlled Markov chains. This means that the distribution of the state at time $t+1$ depends on the past only through the state at time t and the selected action. Let $H_t = (s_1, a_1, s_2, a_2, \ldots, a_{t-1}, s_t)$ denote the history up to time t. Then, we have:

$$\mathbb{P}[s_{t+1} = s' | H_t, a_t = a] = \mathbb{P}[s_{t+1} = s | s_t, a_t = a] := p_t(s' | s_t, a).$$

In the above, $p_t(\cdot|s, a)$ represent the transition probabilities of the system at time t given that the state and the action at time t are (s, a). The transition probabilities are called *stationary* if they do not depend on time t, i.e., $p_t(\cdot|s, a) = p(\cdot|s, a)$.

Reward function. In most cases, we assume that the decision maker collects a deterministic reward at time t equal to $r_t(s, a)$ where (s, a) is the state and action pair at time t. Sometimes, it might be useful to consider random rewards, in which case we denote by $q_t(\cdot|s, a)$ the reward distribution at time t given that the state and the action at time t are (s, a). The reward function is *stationary* if it does not depend on t.
Assumption 1: we always assume that the reward are bounded, i.e., for all time t, for all state and action (s, a), $|r_t(s, a)| \leq 1$.

Markovian policies. We restrict our attention to Markovian policies (because the optimal control policies in MDPs are known to be Markovian and deterministic). A Markovian policy maps the current state to an action or a distribution over actions. Let Π denote the set of Markovian policies.

2.1 Three classes of MDPs

Finite-time horizon MDPs. The objective is to maximize the expected reward for the T first steps:

$$\text{maximize} \quad \sum_{t=1}^{T} \mathbb{E}[r_t(s_t^\pi, a_t^\pi)] \quad \text{over } \pi \in \Pi.$$

In the above expression, s_t^π and a_t^π represents the state and action selected at time t under the policy π.

Stationary MDPs with terminal state. For these problems, we have stationary transition probabilities and rewards. We also assume that there exists a terminal state denoted by $\emptyset$ such that after reaching this state no reward is collected, and the system stays still. Formally, $p(\emptyset|\emptyset, a) = 1$ and $r(\emptyset, a) = 0$ for all $a \in \mathcal{A}_\emptyset$. Denote by $T_\emptyset$ the random time when the terminal state is reached. We restrict our attention to stationary policies in Π: under such a policy, the mapping between the state and the action does not depend on time. We denote by Π_{st} the set of stationary Markovian policies.
Assumption 2: Under any policy $\pi \in \Pi$, $\mathbb{E}_\pi[T_\emptyset] < \infty$. Here, $\mathbb{E}_\pi$ denotes the expectation under the policy π. The above assumption, together with Assumption 1, allows us to define the following optimization problem:

$$\text{maximize} \quad \mathbb{E}[\sum_{t=1}^{T_\emptyset - 1} r(s_t^\pi, a_t^\pi)] \quad \text{over } \pi \in \Pi_{st}.$$

Infinite-time horizon discounted MDPs. For these problems, we have stationary transition probabilities and rewards. Rewards are discounted by a factor $\lambda \in (0, 1)$, and the objective is:

$$\text{maximize} \quad \mathbb{E}[\sum_{t \geq 1} \lambda^{t-1} r(s_t^\pi, a_t^\pi)] \quad \text{over } \pi \in \Pi_{st}.$$

2.2 Solving finite-time horizon MDPs

Consider a finite-time horizon MDP with **known** transition probabilities and reward function. We provide procedures to evaluate the reward of a given policy, and to identify an optimal policy along with its reward.

2.2.1 Policy evaluation

Let $\pi \in \Pi$ be a Markovian deterministic policy. A policy is characterized by its decisions taken at times $1, \ldots, T$: $\pi = (\pi_1, \ldots, \pi_T)$ where $\pi_t : \mathcal{S} \to \mathcal{A} := \cup_s \mathcal{A}_s$.

State value functions. The state value functions of π map the current state and time to the remaining expected reward collected under π until the end of the time horizon. Specifically:

$$\forall t = 1, \ldots, T, \ \forall s \in \mathcal{S}, \quad u_t^\pi(s) := \mathbb{E}\left[\sum_{v=t}^{T} r_v(s_v^\pi, a_v^\pi) \middle| s_t^\pi = s \right].$$

$V_T^\pi := u_1^\pi$ often refers to as *the* state value function of π, i.e., the mapping between the initial state and the expected reward collected under π. The state value functions can be computed using a **backward induction** as follows. $u_T(s) = r_T(s, \pi_T(s))$, and for $t = T-1, \ldots, 1$, $a = \pi_{t-1}(s_{t-1})$,

$$\begin{aligned}
&\text{For } t = T, \quad u_T^\pi(s) = r_T(s, \pi_T(s)), \quad \forall s \in \mathcal{S}, \\
&\text{For } t \leq T, \quad u_{t-1}^\pi(s) = r_{t-1}(s, \pi_{t-1}(s)) + \sum_{j \in S} p_{t-1}(j|s, \pi_{t-1}(s)) u_t^\pi(j), \quad \forall s \in \mathcal{S}.
\end{aligned}$$

2.2.2 The optimal policy

The value function. The value function of the MDP is defined as the state value function of the optimal policy. That is: $V_T^\star(s) := \max_{\pi\in\Pi} V_T^\pi(s)$ (we know that an optimal policy exists since the set of possible policies is finite). As for the state value function of a given policy, the value function can be computed using a backward induction, a procedure referred to as **Dynamic Programming** (DP).

For $t = 1, \dots, T$, define:

$$\forall s \in \mathcal{S}, \quad u_t^\star(s) := \max_{\pi\in\Pi} \mathbb{E}\Big[\sum_{v=t}^{T} r_v(s_v^\pi, a_v^\pi)\big|s_t^\pi = s\Big].$$

DP works as follows:

$$\text{For } t = T, \quad u_T^\star(s) = \max_{a\in\mathcal{A}_s} r_T(s,a), \quad \forall s \in \mathcal{S}$$

$$\text{For } t \le T, \quad u_{t-1}^\star(s) = \max_{a\in A_s} \underbrace{\Big[r_{t-1}(s,a) + \sum_{j\in S} p_{t-1}(j|s,a)u_t^\star(j)\Big]}_{Q_{t-1}(s,a) \text{ optimal reward from } t-1 \text{ if } a \text{ selected}}$$

We obtain $V_T^\star(s) = u_1^\star(s)$ for all $s \in S$. An optimal policy $\pi^\star$ is obtained by selecting $\pi_t^\star(s)$ at time t when in state s:

$$\pi_t^\star(s) \in \arg\max_{a\in A_s} Q_t(s,a).$$

2.3 Solving stationary MDPs with terminal state and infinite-time horizon discounted MDPs

First note that stationary MDPs with terminal state can be seen as specific instances of infinite-time horizon discounted MDPs with discount factor equal to 1. Indeed, we can write: for any stationary policy π,

$$\mathbb{E}[\sum_{t=1}^{T_\emptyset - 1} r(s_t^\pi, a_t^\pi)] = \mathbb{E}[\sum_{t=1}^{\infty} r(s_t^\pi, a_t^\pi)].$$

The series in the r.h.s. is well-defined by definition of the transition probabilities and the rewards when starting in the terminal state. In the following, we consider an infinite-time horizon discounted MDP. By convention, when the discount factor is 1, this means that we deal with an MDP with terminal state.

2.3.1 Policy evaluation

Let $\pi \in \Pi_{st}$ be a stationary Markovian deterministic policy. Such a policy is just characterized by a single mapping between the state and the action. Let $\pi(s)$ denote the action selected in state s under π.

State value function of π. The state value function V^π of π map the current state to the expected discounted reward collected under π when starting at this state.

$$\forall s \in \mathcal{S}, \quad V^\pi(s) := \mathbb{E}\Big[\sum_{t=1}^{\infty} \lambda^{t-1} r(s_t^\pi, a_t^\pi)\big|s_1^\pi = s\Big].$$

The state value function of π can be computed by solving the following set of linear equations:

$$\forall s, \quad V^\pi(s) = r(s, \pi_1(s)) + \lambda \sum_j p(j|s, \pi_1(s)) V^\pi(j).$$

(State, action) value function of π. The (state, action) value function Q^π of π maps an initial state and an initial action to the reward collected assuming that after this first action, the decisions are driven by π.

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A}, \quad Q^\pi(s,a) := r(s,a) + \mathbb{E}\Big[\sum_{t=2}^{\infty} \lambda^{t-1} r(s_t^\pi, a_t^\pi) \big| s_1^\pi = s\Big].$$

The (state, action) value function of π can be computed by solving the following set of linear equations:

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A}, \quad Q^\pi(s,a) = r(s,a) + \lambda \sum_j p(j|s,a) Q^\pi(j, \pi(j)).$$

Alternatively, we have: $Q^\pi(s,a) = r(s,a) + \lambda \sum_j p(j|s,a) V^\pi(j)$. Also note that $V^\pi(s) = Q^\pi(s, \pi(s))$.

2.3.2 The optimal policy

The value function. The value function $V^\star$ is the state value function of an optimal policy. That is: for any state s, $V^\star(s) = \max_{\pi \in \Pi_{st}} V^\pi(s)$. $V^\star$ is the unique solution of **Bellman's equations**:

$$\forall s \in \mathcal{S}, \quad V^\star(s) = \max_{a \in A_s} \underbrace{\Big[r(s,a) + \lambda \sum_{j \in S} p(j|s,a) V^\star(j) \Big]}_{Q(s,a) \text{ optimal reward from state } s \text{ if } a \text{ selected}}$$

The Q-function. The Q function is the value of an optimal policy when the first selected action is imposed: $Q(s,a)$ is the optimal reward when starting in state s and action a is selected. Bellman's equations for the Q-function become:

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A}, \quad Q(s,a) = r(s,a) + \lambda \sum_{j \in S} p(j|s,a) \max_{b \in \mathcal{A}_j} Q(j,b).$$

We also have: $V^\star(s) = \max_{a \in \mathcal{A}_s} Q(s,a)$.

An optimal policy $\pi^\star$ is defined by:

$$\pi^\star(s) \in \arg\max_{a \in A_s} Q(s,a).$$

2.3.3 Value and Policy Iteration algorithms

An optimal policy can be deduced from the value function or the Q-function. To find the value function, one needs to solve Bellman's equations. The value function can be seen as the fixed point of a non-linear operator.

Bellman's operator $\mathcal{L} : \mathbb{R}^S \to \mathbb{R}^S$ defined by:
for all $V \in \mathbb{R}^S$,

$$\forall s \in \mathcal{S}, \ \mathcal{L}(V)(s) = \max_{a \in A_s} \left[r(s,a) + \lambda \sum_j p(j|s,a) V(j) \right]$$

Bellman's operator $\mathcal{L}$ is a contraction mapping when $\lambda \in (0,1)$, and it has a unique fixed point, equal to the value function: $V^\star = \mathcal{L}(V^\star)$. This implies that the following sequence of functions converges as n grows large to $V^\star$: $V_0 = 0$ and for any $n \geq 0$, $V_{n+1} = \mathcal{L}(V_n)$.

Value Iteration algorithm. The above sequence is what the Value Iteration algorithm is implementing.

Algorithm 1: Value Iteration
Input. Precision ϵ, discount factor λ

1. **Initialization.** Select a value function $V_0 \in \mathbb{R}^S$, $n = 0$, $\delta \gg 1$
2. **Value improvement.** While $(\delta > \frac{\epsilon(1-\lambda)}{\lambda})$ do
   (a) $V_{n+1} = \mathcal{L}(V_n)$, i.e., $\forall s \in S$, $V_{n+1}(s) = \max_{a \in A_s}(r(s,a) + \lambda \sum_j p(j|s,a)V_n(j))$
   (b) $\delta = \|V_{n+1} - V_n\|$, $n \leftarrow n+1$
3. **Output.** Policy π with $\forall s \in S,\ \pi(s) \in \arg\max_{a \in A_s}(r(s,a) + \lambda \sum_j p(j|s,a)V_n(j))$

The above VI algorithm returns an ϵ-optimal policy π, i.e., for any $s \in \mathcal{S}$, $V^\pi(s) \geq V^\star(s) - \epsilon$. The algorithm requires $\Theta(S^2 A)$ floating operations per iteration.

Policy Iteration algorithm. The PI algorithm consists in updating the policy in each iteration. The pseudo-code in Algorithm 2 presents Howard's PI algorithm.

Algorithm 2: Policy Iteration

1. **Initialization.** Select a policy π_0 arbitrarily, $n = 0$
2. **Policy evaluation.** Evaluate the state value function V_n of π_n by solving:
$$\forall s \in S,\ V_n(s) = r(s, \pi_n(s)) + \lambda \sum_j p(j|s, \pi_n(s))V_n(j)$$
3. **Policy improvement.** Update the policy:
$$\forall s \in S,\ \pi_{n+1}(s) \in \arg\max_{a \in A_s}(r(s,a) + \lambda \sum_j p(j|s,a)V_n(j))$$
4. **Stopping criterion.** If $\pi_{n+1} = \pi_n$, return π_n.
   Otherwise $n \leftarrow n+1$, and go to 2.

The **policy improvement theorem** states that under the PI algorithm, V_n is an increasing sequence, in the sense that for any $s \in \mathcal{S}$, $V_{n+1}(s) \geq V_n(s)$. In other words π_{n+1} is better than π_n. When the PI stops, it returns an optimal policy. Each iteration of the PI algorithm requires $\Theta(S^\omega)$ floating operations – this is the number of operations required to invert an $S \times S$ matrix, $\omega \in (2, 3)$.

3 RL problems

In the previous section, we have seen how to derive optimal control policies in MDPs, when the transition probabilities and reward function are know. RL deals with scenarios where these have to be learnt.

3.1 Episodic vs. discounted RL problems

MDPs for which the time horizon is finite (either by definition, or because we reach a terminal state in a finite time) lead to what we call *episodic* RL problems when the transition probabilities and reward function are unknown. An *episode* then refers to the realization of a trajectory of the MDP from the initial state to the state occupied when the time horizon finishes or to the terminal state. To learn the transition probabilities and reward function, the RL agent will run many episodes.

MDPs with infinite time horizon lead to what we call discounted RL problems. In these problems, often, we observe a single trajectory of the system and we have to learn an optimal policy from

this trajectory.

3.2 On vs. off policy learning

On-policy learning refers to RL algorithms where the agent learns the value of the policy under which the data is generated. Off-policy learning refers to RL algorithms where the agent does not learn the value of the policy under which the data is generated. For off-policy learning, the policy under which the data is generated is called the *behaviour* policy, and often denoted by π_b.

3.3 Online vs. Offline learning

Online learning refers to situations where the agent is willing to control the system (i.e., collect actual rewards) while learning the system dynamics and reward function. Most often, this goes with on-policy learning. The performance of an online learning agent is quantified using the notion of *regret*: this is the difference of the cumulative reward (from the first step) obtained under an optimal control policy and that obtained under the learning agent.

Offline learning just aims at returning, after observing a certain number of experiences[1], an efficient or optimal policy. The performance of an offline learning agent is measured through the notion of sample complexity. This is the number of experiences required so as the agent returns an optimal (or ϵ-optimal) policy with a certain level certainty, say with probability at least $1-\delta$.

4 Stochastic approximation and stochastic gradient descent algorithms

Machine Learning methods and RL algorithms rely on two basic algorithms, the Stochastic Approximation (SA) algorithm developed by Robbins-Monro, and the Stochastic Gradient Descent algorithm proposed by Kiefer-Wolfowitz. Both algorithms date from the 50's.

4.1 SA algorithm

Let $h : \mathbb{R}^d \to \mathbb{R}^d$ be a lipschitz continuous function. We look for a root of h, i.e., $x^\star$ such that $h(x^\star) = 0$. To this aim, we have access to noisy estimates of h, i.e., for a given x, we can get from an Oracle a random variable Y with expectation $h(x)$. Robbins and Monro proposed to exploit these noisy estimates of the function using the following algorithm:

Robbins-Monro Stochastic Approximation algorithm:
1. **Initialization:** $x^{(0)} \in \mathbb{R}^d$
2. **Iterations:** for $k \geq 0$, $x^{(k+1)} = x^{(k)} + \alpha_k[h(x^{(k)}) + M_{k+1}]$

To ensure the convergence of the algorithm to $x^\star$, we make the following assumptions.

A1. (Martingale difference) $\forall k$, $\mathbb{E}[M_{k+1}|\mathcal{F}_k] = 0$ where $\mathcal{F}_k = \sigma(x^{(1)}, M_1, \ldots, x^{(k)}, M_k)$ and $\forall k$, $\mathbb{E}[\|M_{k+1}\|_2^2 \mid \mathcal{F}_k] \leq c_0(1 + \|x^{(k)}\|^2)$.

A2. (Stability) $\dot{x} = h(x)$ has a unique globally stable equilibrium $x^\star$. $\forall x$, $h_\infty(x) = \lim_{c\to\infty} \frac{h(cx)}{c}$ exists and 0 is the only globally stable point of $\dot{x} = h_\infty(x)$.

[1] An *experience* corresponds to the state of the system at a given time t, the action selected, the reward collected, and the state at time $t+1$.

Convergence for decreasing learning rates. Under A1 and A2, if the learning rates α_k satisfy $\sum_{k=0}^{\infty} \alpha_k = \infty$ and $\sum_{k=0}^{\infty} \alpha_k^2 < \infty$, then $\lim_{k\to\infty} x^{(k)} = x^\star$ almost surely where $x^\star$ is the unique globally stable point of $\dot{x} = h(x)$.

Convergence for fixed learning rates. When $\alpha_k \alpha$ for all k, the algorithm does not converge, but is guaranteed to be in a neighborhood of $x^\star$ at the limit. The neighborhood is of size proportional to α.

4.2 SGD algorithm

Let $f : \mathcal{C} \to \mathbb{R}$ be a convex function defined over the convex set $\mathcal{C}$. We wish to find the minimizer of f. To this aim, we can not evaluate the gradients of f, but get only unbiased estimates of these gradients. Specifically, for any $x \in \mathcal{C}$, an Oracle can reveal $g(x)$, a r.v. such that $\nabla f(x) = \mathbb{E}[g(x)]$. Kiefer and Wolfowitz proposed the following SGD algorithm

Kiefer-Wolfowitz SGD Algorithm:
1. **Initialization:** $x^{(0)}$
2. **Iterations:** for $k \geq 0$, $x^{(k+1)} = x^{(k)} - \alpha_k g(x^{(k)})$

Let $f_{\min}^{(k)} = \min_{i=0,\ldots,k} f(x^{(i)})$. Assume that for all k, $\mathbb{E}[\|g(x^{(k)})\|_2^2] \leq G^2$, and that $\mathbb{E}[g(x^{(k)})|x^{(k)}] = \nabla f(x^{(k)})$, then: We have for all $k \geq 1$:

$$\mathbb{E}[f_{\min}^{(k)} - f(x^\star)] \leq \frac{\|x^{(0)} - x^\star\|_2^2 + G^2 \sum_{i=0}^{k} \alpha_i^2}{2 \sum_{i=0}^{k} \alpha_i}.$$

this implies that SGD has the same convergence guarantees as those of the SA algorithm.

5 Policy evaluation in RL

We present two methods to evaluate a policy when the transition probabilities and the reward function are unknown. The first method is a simple Monte Carlo simulation method, and it works for episodic RL problems only. The second method, referred to as Temporal Difference (TD) learning, is a bootstrapping method working for RL problems corresponding to infinite time horizon discounted MDPs and stationary MDPs with terminal state.

5.1 Monte Carlo policy evaluation

Consider a stationary MDP with terminal state. Consider a stationary policy π. We wish to evaluate its state value function V^π. To estimate $V^\pi(s)$, we generate a large number of episodes under π, and record the reward collected after visiting s for the first time in each episode. $V^\pi(s)$ is estimated by the emprical mean of this rewards. The law of large numbers ensures that this estimator is consistent (when the number of episodes grows large, it converges to $V^\pi(s)$ almost surely). More precisely, the algorithm is as follows.

Monte Carlo prediction algorithm:

1. **Initialization:** $\forall s$, $V^{(0)}(s) = 0$
2. **Iterations:** for episode $i = 1, \dots, n$
   generate $\tau_i = (s_1, a_{1,i}, r_{1,i}, \dots, s_{T_i,i}, a_{T_i,i}, r_{T_i,i})$ under π
   $G = 0$
   for $t = T_i, T_i - 1, \dots, 1$:
   a. $G = r_{t,i} + G$
   b. Unless $s_{t,i}$ appears in $\{s_{1,i}, \dots, s_{t-1,i}\}$
   $V^{(i)}(s_{t,i}) = V^{(i-1)}(s_{t,i}) + \frac{1}{i}(G - V^{\pi}(s_{t,i}))$

In the above algorithm, for episode i, T_i denotes the time preceding the time the terminal state is reached (we know that no reward is collected after that). The algorithm also works for discounted rewards: just replace $G = r_{t,i} + G$ by $G = r_{t,i} + \lambda G$.

5.2 TD learning

TD learning is an asynchronous version of a Stochastic Approximation algorithm. To fix ideas, we consider a discounted RL problem, but it also works for problems in stationary MDPs with terminal state. Observe that the state value function of π satisfies $h(V^{\pi}) = 0$ where

$$\forall s, \quad h(V)(s) = r(s, \pi(s)) + \lambda \sum_j p(j|s, \pi(s))V(j) - V(s).$$

h is a linear function for which it is easy to verify the assumption required for the convergence of the SA algorithm. The following algorithm is an asynchronous version of the SA algorithm, in the sense that only the component of V^{π} corresponding to the current state is updated. The algorithm follows the trajectory of the system under π, and in step t updates $V^{\pi}(s_t)$ only.

TD(0) algorithm

1. **Initialization.**
   Select a value function $V^{(1)}$
   Initial state s_1
   Number of visits: $\forall s$, $n_s^{(1)} = 1_{(s=s_1)}$

2. **Value function updates.** For all $t \geq 1$, select action $\pi(s_t)$ and observe the new state s_{t+1} and reward r_t.
   Update the value function estimate: for all s,

$$V^{(t+1)}(s) = V^{(t)}(s) + 1_{(s_t=s)}\alpha_{n_s^{(t)}}\left(r_t + \lambda V^{(t)}(s_{t+1}) - V^{(t)}(s)\right)$$

   Update for all s, $n_s^{(t+1)} = n_s^{(t)} + 1_{(s=s_t)}$

The learning rates α_n are typically chosen as for the SA algorithm.
In TD methods, the term $r_t + \lambda V^{(t)}(s_{t+1})$ is often referred to as the **target**. Note that TD learning is a **bootstrapping** method because the targeted value in each iteration depends on the current estimate of V^{π}.

6 Optimal control in RL

In the previous section, we have presented methods to evaluate a given policy in a RL setting. Next, we are interested in devising RL algorithms learning an efficient or even optimal policy.

6.1 ϵ-soft policies

In MDP, optimal policies are deterministic and Markovian. In RL, to actually learn these policies, we need to explore. To this aim, we often use ϵ-soft policies. The latter select an action uniformly at random with probability ϵ in each step. Formally, π is an ϵ-soft policy if when the system is in state s at time t, then $\pi(s)$ selects a random action if $X_s(t) = 1$, where $X_s(t) \sim \text{Ber}(\epsilon)$ (Bernoulli). $(X_s(t))_{s,t}$ are independent across s and t.

ϵ-greedy policy w.r.t. R. A very usefuly ϵ-soft policies are the ϵ-greedy policy. Let $R : \mathcal{S}\times\mathcal{A} \to \mathbb{R}$ be a (state, action) function, e.g. a estimate of the Q-function or of the (state, action) value function of a given policy.

- π is greedy w.r.t. R if the action selected in s is $\pi(s) \arg\max_{a\in\mathcal{A}_s} R(s,a)$;
- π is ϵ-greedy w.r.t. R, if when in state s at time t:

$$\pi(s) = \begin{cases} \text{unif. over } \mathcal{A}_s, & \text{if } X_s(t) = 1, \\ \arg\max_{a\in\mathcal{A}_s} R(s,a), & \text{if } X_s(t) = 0. \end{cases}$$

  In the above, as earlier, $X_s(t) \sim \text{Ber}(\epsilon)$.

6.2 Q-learning algorithm

Consider a discounted RL problem. The algorithm is the same for stationary MDP with terminal state. Remember that the Q-function is the unique function satisfying Bellman's equations:

$$\forall(s,a), \quad Q(s,a) = r(s,a) + \lambda \sum_j p(j|s,a) \max_{b\in\mathcal{A}_b} Q(j,b).$$

The Q-function provides optimal policies: π is optimal if and only if it is greedy w.r.t. Q. The Q-learning algorithm is an off-policy learning algorithm learning Q: it is an asynchronous SA algorithm converging towards the root of the function $h : \mathbb{R}^{S\times A} \to \mathbb{R}^{S\times A}$ defined by: for all $R \in \mathbb{R}^{S\times A}$,

$$\forall(s,a), \quad h(R)(s,a) = r(s,a) + \lambda \sum_j p(j|s,a) \max_{b\in\mathcal{A}_b} R(j,b) - R(s,a).$$

Next we explain how for a given estimate $\hat{Q}$ of the Q-function, we get a sample of $h(\hat{Q})$. Let π_b denote the behavior policy, and let $(s, \pi_b(s), r, s')$ be an experience generated under π_b. A sample for $h(\hat{Q})(s, \pi_b(s))$ is then:

$$Y = r(s,\pi_b(s)) + \lambda \max_{b\in\mathcal{A}_b} \hat{Q}(s',b) - \hat{Q}(s,\pi_b(s)).$$

Observe that $\mathbb{E}[Y|s] = 0$, and one can check that the convergence conditions for the corresponding SA algorithm are satisfied (i) for appropriate learning rates, and (ii) if the behavior policy is exploring enough (i.e., visits all (state, action) pairs infinitely often).

Q-learning algorithm
Parameter. Step sizes (α_t)
1. Initialization. Select a Q-function $Q^{(0)} \in \mathbb{R}^{S\times A}$
2. Observations. (s_t, a_t, r_t, s_{t+1}) under the behavior policy π_b
3. Q-function improvement. For $t \geq 0$. Update the estimated Q-function as follows: $\forall s, a$,

$$\begin{aligned} Q^{(t+1)}(s,a) &= Q^{(t)}(s,a) \\ &+ 1_{(s_t,a_t)=(s,a)} \alpha_{n^{(t)}(s_t,a_t)} \left[r_t + \lambda \max_{b\in\mathcal{A}} Q^{(t)}(s_{t+1},b) - Q^{(t)}(s_t,a_t) \right] \end{aligned}$$

where $n^{(t)}(s,a) := \sum_{m=1}^{t} 1[(s,a) = (s_m, a_m)]$.

6.3 SARSA algorithm

Consider a discounted RL problem. The algorithm is the same for stationary MDP with terminal state. SARSA is an on-policy learning algorithm. The algorithm maintains in step t both a policy π_t and its estimated (state, action) value function $Q^{(t)}$. In each iteration, SARSA

- updates π_t (the policy improvement step by taking the ϵ-greedy policy w.r.t. $Q^{(t)}$;
- updates $Q^{(t)}$ by applying a TD-learning step to evaluate the (state, action) value function of π_t.

The policy improvement step actually improves the policy (in average) according to the **policy improvement theorem**. SARSA means (State, Action, Reward, State, Action), because the algorithm uses experiences of the type (s, a, r, s', a') generated under the policy π_t. Here is the pseudo-code of the algorithm:

SARSA algorithm
Parameters. Step sizes (α_t), Exploration rate $\epsilon > 0$
1. Initialization. Select a Q-function $Q^{(0)} \in \mathbb{R}^{S \times A}$
2. Observations. $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ under π_t ϵ-greedy w.r.t. $Q^{(t)}$
3. Q-function improvement. For $t \geq 0$. Update the estimated Q-function as follows: $\forall s, a$,

$$Q^{(t+1)}(s,a) = Q^{(t)}(s,a) + 1_{(s_t,a_t)=(s,a)}\alpha_{n^{(t)}(s_t,a_t)}\left[r_t + \lambda Q^{(t)}(s_{t+1}, a_{t+1}) - Q^{(t)}(s_t, a_t)\right]$$

where $n^{(t)}(s,a) := \sum_{m=1}^{t} 1[(s,a) = (s_m, a_m)]$.

Under SARSA, π_t does not converge towards an optimal policy, because π_t is ϵ-soft. When ϵ is very small, π_t approximates an optimal policy when t is large.

7 Learning with function approximation

Beyond tabular MDPs. In tabular MDPs, we consider that the outcomes of the various (state, action) pairs (by outcomes, we mean the distribution of the next state and the reward) are not related. In turn, for such MDP, we need to explore *any* (state, action) pair to learn an efficient policy. This is why the sample complexity or the regret of any learning algorithm for tabular MDPs must scale as the product $S \times A$. Most often, this is unacceptable (the set of possible states can be large), and infeasible when the state or the action space is continuous.

Function approximation. It is here assumed that functions of interest such as the value function, the state value function of a policy, the Q-function belong to a set of parametrized functions. For example, $V^\star \approx V_\theta \in \mathcal{V} = \{V_\mu : \mu \in \mathbb{R}^d\}$. With this assumption, we implicitly assume that the outcomes of the various (state, action) pairs are related; there is an underlying structure that we exploit to speed up the learning process. Indeed, one just needs to learn the parameter θ.

Examples. The most commonly used function approximation are linear and based on neural networks. We illustrate these examples for approximating a function of the states only (e.g. the value function).

1. Linear function approximation. In this case, we work with a basis of functions $\phi_1, \ldots, \phi_d : \mathbb{R}^S \to \mathbb{R}$, and $V_\theta(s) = \sum_{i=1}^{d} \theta_i \phi_i(s)$.
2. Deep learning. Here the value of the function is the output of a neural network. The function is hence parametrized by the weights of the network: $V_w(s)$.

7.1 Policy evaluation with function approximation

Consider a stationary policy π for an infinite horizon discounted MDP. We wish to compute V^π, or to provide an approximation by one the function in $\mathcal{V} = \{V_\theta : \theta \in \mathbb{R}^d\}$. The goal is then to identify the parameter θ such that V^π and V_θ are as close as possible. One could for example try to find θ minimizing $\mathbb{E}_{s\sim\mu}[(V^\pi(s) - V_\theta(s))^2]$, where μ denotes the stationary distribution of the state under π. The gradient of this objective function would however depends on the unknown V^π. To circumvent this issue, we use a bootstrapping method, and instead attempt to minimize over θ the mean TD square error:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{s\sim\mu}[(r(s,\pi(s)) + \lambda \sum_j p(j|s,\pi(s))V_\theta(j) - V_\theta(s))^2].$$

When taking the gradient of the above objective function, we often decouple the time scales at which the target $r(s,\pi(s)) + \lambda\sum_j p(j|s,\pi(s))V_\theta(j)$ and $V_\theta(s)$ evolve. The target is fixed and the gradient is taken w.r.t. $V_\theta(s)$ only. This heuristics, referred to as **semi-gradient** descent, should be justified, and leads to the following algorithm.

TD(0) algorithm with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$, observe s_t, a_t, r_t, s_{t+1} under π.
   Update θ as:
$$\theta \leftarrow \theta + \alpha(r_t + \lambda V_\theta(s_{t+1}) - V_\theta(s_t))\nabla_\theta V_\theta(s_t)$$

7.2 SARSA with function approximation

The previous TD algorithms with function approximation can be applied to provide an approximation of the (state, action) value function of a given policy π. This can be implemented in the policy evaluation step of SARSA algorithm.

SARSA algorithm with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   (alternative: select the "a_{t+1}" of the previous step as a_t)
   sample a_{t+1} according to π_t
   update θ as:
$$\theta \leftarrow \theta + \alpha(r_t + \lambda Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t))\nabla_\theta Q_\theta(s_t, a_t)$$

7.3 Q-learning with function approximation

One idea to use function approximation in Q-learning is to try to minimize the mean square Bellman error. If $\tilde{Q}$ is an estimated Q-function, the corresponding Bellman error is defined as:

$$BE(s,a) = r(s,a) + \lambda \sum_j p(j|s,a) \max_b \tilde{Q}(j,b) - \tilde{Q}(s,a).$$

Hence a possible objective is to minimize:

$$\begin{aligned} J(\theta) &= \frac{1}{2}\mathbb{E}_{(s,a)\sim\mu_b}[BE(s,a)^2] \\ &= \frac{1}{2}\mathbb{E}_{(s,a)\sim\mu_b}[(r(s,a) + \lambda \sum_j p(j|s,a) \max_b Q_\theta(j,b) - Q_\theta(s,a))^2], \end{aligned}$$

where μ_b is the stationary distribution of (s,a) under the behavior policy π_b. When fixing the target $r(s,a) + \lambda \sum_j p(j|s,a) \max_b Q_\theta(j,b)$, the semi-gradient of J is:

$$-\mathbb{E}_{(s,a)\sim\mu_b}\Bigg[\underbrace{(r(s,a) + \lambda \sum_j p(j|s,a) \max_b Q_\theta(j,b)}_{\text{target}} - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)\Bigg]$$

Assume that we observe an experience (s,a,r,s') generated under π_b. The above semi-gradient can be estimated (without bias) by:

$$-(r + \lambda \max_b Q_\theta(s',b) - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a))$$

This leads to the following algorithm.

Q-learning with function approximation

1. **Initialization.** θ, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   update θ as:

$$\theta \leftarrow \theta + \alpha(r_t + \lambda \max_b Q_\theta(s_{t+1},b) - Q_\theta(s_t,a_t))\nabla_\theta Q_\theta(s_t,a_t)$$

The above algorithm however does not work well in practice, for various reasons. If we follow a trajectory, the successive updates are strongly correlated, which impacts the convergence of the algorithm. In addition, the target is not fixed, and the algorithm struggles to follow this moving target. These two issues are solved using experience replay and fixed target, respectively.

Experience replay: We maintain a buffer B of previous experiences (s,a,r,s'). At time t, we store the current experience in the buffer, but to update the Q-function parameter, we sample mini-batches of fixed size k from B uniformly at random. At time t Sample, we perform k updates: for $i = 1, \ldots, k$, the experience (s_i, a_i, r_i, s_i') is sampled uniformly at random from B, and we do:

$$\theta \leftarrow \theta + \alpha(r_i + \lambda \max_b Q_\theta(s_i',b) - Q_\theta(s_i,a_i))\nabla_\theta Q_\theta(s_i,a_i)$$

Fixed target: We use a second parameter ϕ for the target. The target is fixed for C successive steps, and then aligned to θ.

In summary, we obtain the following algorithm.

Q-learning with function approximation, ER, and fixed targets

1. **Initialization.** θ and ϕ, replay buffer B, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   store (s_t, a_t, r_t, s_{t+1}) in B
   sample k experiences (s_i, a_i, r_i, s'_i) from B
   for $i = 1, \ldots, k$:

$$y_i = \begin{cases} r_i & \text{if episode stops in } s'_i \\ r_i + \lambda \max_b Q_\phi(s'_i, b) & \text{otherwise} \end{cases}$$

   update θ as:

$$\theta \leftarrow \theta + \alpha(y_i - Q_\theta(s_i, a_i))\nabla_\theta Q_\theta(s_i, a_i)$$

   every C steps: $\phi \leftarrow \theta$

When we use neural networks to approximate functions, the above algorithm corresponds to DQN.

8 Policy gradients and actor-critic algorithms

Policy gradient algorithms consist in parametrizing the policy itself, and in applying SGD algorithms to find the parameter corresponding to an optimal policy.

8.1 Episodic RL problems

In what follows, we assume that the time horizon T is fixed. The analysis and results however hold for stationary MDP with terminal state. The transition probabilities and the rewards are stationary. We denote by p_1 the distribution of the initial state. Consider a set of stationary policies parametrized by θ. It is assume that the policy is smooth w.r.t. θ (at least continuously differentiable). We wish to maximize:

$$J(\theta) = \mathbb{E}_{s_1 \sim p_1}[V^{\pi_\theta}(s_1)].$$

Rewrite $V^{\pi_\theta}(s_1)$ as:

$$V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[\sum_{t=1} r(s_t, a_t)] = \sum_\tau \pi_\theta(\tau) R(\tau),$$

where τ denotes the random trajectory of an episode, $\pi_\theta(\tau)$ is the probability to observe this trajectory under π_θ, and $R(\tau)$ is the total reward collected during the episode τ. We get the **policy gradient theorem**:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)].$$

In the above expectations, $\mathbb{E}_{\pi_\theta}$ just indicates that the episode is generated under π_θ.

$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^T \nabla \log \pi_\theta(s_t, a_t)$ is referred to as the **score function**. It corresponds to the log-likelihood of observing τ under π_θ. The policy gradient theorem states that when generating τ under π_θ, then $\nabla_\theta \log \pi_\theta(\tau) R(\tau)$ constitutes an unbiased estimator of $\nabla_\theta J(\theta)$. This naturally leads to REINFORCE algorithm:

REINFORCE Algorithm:

1. **Initialization:** select $\theta^{(0)}$ arbitrarily
2. **Iterations:** For all $k \geq 0$, for episode k, generate a trajectory under $\pi_{\theta^{(k)}}$: $(s_{1,k} = s, a_{1,k}, r_{1,k}, \dots s_{T,k}, a_{T,k}, r_{T,k})$
   Update the parameter

$$\theta^{(k+1)} = \theta^{(k)} + \alpha_k \left(\sum_{t=1}^{T} \nabla \log \pi_\theta(s_{t,k}, a_{t,k}) \right) \left(\sum_{t=1}^{T} r_{t,k} \right)$$

REINFORCE is a SGD algorithm, and finds a local maxmizer of $J(\theta)$ (under the usual convergence conditions). In practice, the algorithm offers poor performance, because of the high variance of the gradient estimator $\nabla_\theta \log \pi_\theta(\tau) R(\tau)$. Three techniques can be used to reduce this variance:

- **Batches.** Generate n episodes (instead of 1) before updating θ; this divides the variance by $1/n$.
- **Reward to go.** Instead of $\nabla_\theta \log \pi_\theta(\tau) R(\tau)$, use $\sum_{t=1}^{T} \nabla \log \pi_\theta(s_t, a_t) \sum_{u=t}^{T} r(s_u, a_u)$; this estimator remains Unbiased.
- **Baseline.** Adding a baseline helps and does not introduce any bias. When n episodes are generated under the same θ, a natural baseline is the empirical rewards observed in the n episodes:

$$b = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=1}^{T} r(s_{t,i}, a_{t,i})$$

  The update is computed using $\nabla_\theta \log \pi_\theta(\tau)(R(\tau) - b)$.

In practice, you may consider combining the three above techniques.

8.2 Episodic RL with terminal state

Consider now a stationary MDP with terminal state. As mentioned already, the analysis done for fixed time horizon MDP holds for this MDP. There is however another way to express the gradient of the objective function.

Denote by μ_θ the stationary distribution of the (state, action) pair under π_θ. Then a second version of the **policy gradient theorem** is:

$$\nabla J(\theta) = \mathbb{E}_{(s,a)\sim\mu_\theta} \left[\nabla \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a) \right]$$

We can sample from μ_θ by simulating π_θ, but we need to estimate Q^{π_θ}. This will be discussed in the subsection about actor-critic algorithms.

8.3 Discounted RL problems

Consider now an infinite time horizon discounted MDP. The objective is still to maximize $J(\theta) = \mathbb{E}_{s_1\sim p}[V^{\pi_\theta}(s_1)]$ over all possible θ. To derive the gradient of J, we introduce the discounted stationary distribution ρ_θ under π_θ:

$$\forall s \in \mathcal{S}, \quad \rho_\theta(s) = (1 - \lambda) \sum_{s'} p(s') \sum_{k=1}^{\infty} \lambda^k \mathbb{P}_{\pi_\theta}[s_k = s | s_1 = s']$$

We have:

$$\nabla J(\theta) = \frac{1}{1 - \lambda} \mathbb{E}_{s\sim\rho_\theta, a\sim\pi_\theta(s,\cdot)} \left[\nabla \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a) \right]$$

There are two difficulties in using the above formula towards a SGD algorithm.
1. We need a critic to estimate Q^{π_θ} (see the next subsection);
2. Sampling s according to ρ_θ is not easy.

For the second issue, most algorithms you find in the literature implicitly assume that ρ_θ is the stationary distribution of the state under π_θ: they are wrong! Indeed, ρ_θ depends on the discount factor and when the latter is close to 0, ρ_θ is close to p_1 and hence far from the stationary distribution of the state under π_θ.

When you may restart the system when you wsh, you can sample according to ρ_θ as follows. Generate $s_1 \sim p_1$; for $t \geq 1$, $a_t \sim \pi_\theta(s_t, \cdot)$, s_{t+1} drawn from $p(\cdot|s_t, a_t)$ with probability λ, and from p_1 with probability $1 - \lambda$. Then sampling a state randomly from the constructed trajectory corresponds to sampling from ρ_θ.

8.4 Actor-critic algorithms

Consider here stationary MDPs with terminal state. The policy gradient theorem states that $\nabla J(\theta) = \mathbb{E}_{(s,a)\sim\mu_\theta}\left[\nabla \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right]$. Actor-critic algorithms are combining a component to update θ along the gradient of J, and a component to evaluate Q^{π_θ}.

Remember that the (state, action) value function of a stationary policy π satisfies:

$$\forall (s,a), Q^{\pi_\theta}(s,a) = r(s,a) + \sum_j p(j|s,a) \sum_b \pi_\theta(j,b) Q^{\pi_\theta}(j,b).$$

To evaluate Q^{π_θ}, we can use TD learning and function approximation $Q^{\pi_\theta} \approx Q_\phi$: when the experience (s, a, r, s', a') is observed, we update ϕ following a semi-gradient descent algorithm:

$$\phi \leftarrow \phi + \beta(r + Q_\phi(s', a') - Q_\phi(s,a)) \nabla_\phi Q_\phi(s,a).$$

Combining this policy evaluation step to the policy update, we get the Q Actor-Critic algorithm. Note that the learning rates α and β at which θ and ϕ are updated may differ. Typically, we wish to keep the same policy π_θ for a period long enough so as to be able to estimate Q^{π_θ}. Hence, typically, α is chosen much smaller than β.

QAC Algorithm:

1. **Initialization:** θ, ϕ, state $s \leftarrow s_1 \sim p_1$

2. **Iterations:** Loop
   If $s = \emptyset$, $s \leftarrow s_1 \sim p_1$
   Take action $a \sim \pi_\theta(s, \cdot)$ and observe r, s' (reward, next state)
   Sample the next action $a' \sim \pi_\theta(s', \cdot)$
   Update the parameters

$$\begin{aligned} \phi &\leftarrow \phi + \beta(r + Q_\phi(s', a') - Q_\phi(s,a)) \nabla_\phi Q_\phi(s,a) \\ \theta &\leftarrow \theta + \alpha \left(\nabla_\theta \log \pi_\theta(s,a) Q_\phi(s,a)\right) \end{aligned}$$

   $s \leftarrow s'$, $a \leftarrow a'$

Actor-Critic algorithm with a baseline. We can apply the idea of using a baseline to enhance the convergence properties of the algorithm. Here the natural baseline is $V^{\pi_\theta}(s)$ since $V^{\pi_\theta}(s) = \mathbb{E}_{a\sim\pi_\theta(s,\cdot)}[Q^{\pi_\theta}(s,a)]$.

$$\text{Advantage function: } A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s).$$

The policy gradient theorem states that:

$$\nabla_\theta J(\theta) = \mathbb{E}_{(s,a)\sim\mu_\theta}\left[\nabla \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)\right].$$

Now when (s, a, r, s', a') is observed under π_θ, we get:

$$A^{\pi_\theta}(s, a) = r + \mathbb{E}[V^{\pi_\theta}(s')] - V^{\pi_\theta}(s)$$

Hence we can use and fit $V^{\pi_\theta} \approx V_\phi$ only! Using TD learning we get the following update:

$$\phi \leftarrow \phi + \beta(r + V_\phi(s') - V_\phi(s))\nabla_\phi V_\phi(s).$$

The following two versions of the A2C (Advantage Actor-Critic) algorithms implement these ideas. The first uses TD learning to estimate V^{π_θ}. The second (more stable) exploits MC method.

A2C Algorithm (TD version)

1. **Initialization:** θ, ϕ, state $s \leftarrow s_1 \sim p_1$

2. **Iterations:** Loop
   If $s = \emptyset$, $s \leftarrow s_1 \sim p_1$
   Take action $a \sim \pi_\theta(s, \cdot)$
   Observe r, s' (reward, next state)
   Sample the next action $a' \sim \pi_\theta(s', \cdot)$
   Update the parameters

$$\begin{aligned}\phi &\leftarrow \phi + \beta(r + V_\phi(s') - V_\phi(s))\nabla_\phi V_\phi(s) \\ \theta &\leftarrow \theta + \alpha\left(\nabla_\theta \log \pi_\theta(s, a)(r + V_\phi(s') - V_\phi(s))\right)\end{aligned}$$

   $s \leftarrow s'$, $a \leftarrow a'$

A2C Algorithm (MC method)

1. **Input:** nb of episodes T_E

1. **Initialization:** Initialize Actor and Critic networks π_θ, V_ϕ; Initialize episodic buffer $\mathcal{B}$
2. **For episodes** $k = 1, 2, \ldots, T_E$
   Episode init: $s_1 \sim p_1$, $t \leftarrow 0$
   While (Episode k not finished)
   Take action $a_t \sim \pi_\theta(s_t)$ and put $z = (s_t, a_t, r_t, s_{t+1})$ in $\mathcal{B}$
   $t \leftarrow t + 1$
   end while

   Compute y_i for each $(s_i, a_i) \in \mathcal{B}$: $y_i = \sum_{n=i}^{t-1} r_n$.
   Update ϕ using the MSE loss $\sum_i (y_i - V_\phi(s_i))^2$
   Update θ: $\theta \leftarrow \theta + \alpha \sum_i \left(\nabla_\theta \log \pi_\theta(s_i, a_i)(y_i - V_\phi(s_i))\right)$
   Clear buffer $\mathcal{B}$