Reinforcement Learning

– Course notes –

December 15, 2019

Contents

Chapter 1

Markov Decision Processes

Definition 1 (Markov Decision process) *A* Markov decision process *(MDP) is given by its state space $\mathcal{S}$, action space $\mathcal{A}$, state dynamics, reward structure and discount factor. In a MDP, the state dynamics can be written as*

$$\mathbb{P}(s_{t+1} \mid s_t, a_t)$$

The **return** of a trajectory $(s_0, a_0, r_0, s_1, \ldots)$ at time t is defined as

$$G_t = \sum_k \gamma^k r_{t+k} \tag{1.1}$$

If given a complete trajectory (episode) that terminates, the returns can be computed recursively using

$$G_t = r_t + \gamma G_{t+1}. \tag{1.2}$$

1.1 Decision rules, Policies

The idea of a *decision rule* is to choose which action (or actions, with a given preference) to take after observing part of an MDP's trajectory.

Chapter 2

Solving MDPs: value functions and Dynamic Programming

2.1 The value function

The value function is a staple from the literature on dynamic programming, whether it be for discrete or continuous problems (as in control theory). It measures just how good a control u – or, in our case, a policy π – is regarding the desired target of our problem.

Definition 2 (Value function) *The value function $V^\pi : \mathcal{S} \to \mathbb{R}$ of a policy π is the expectation of the cumulative (discounted) future rewards starting from a point $s_0 = s$*

$$V^\pi(s) := \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \,\middle|\, s_0 = s\right] \tag{2.1}$$

where the trajectory τ is generated under the policy π.

T can be a random stopping time corresponding to the first time of arrival at a so-called terminal *state. It can also be $T = \infty$ for* infinite horizon *problems.*

Remark 1 (Notational abuse) *Often, we will write the value functon at a state s_t, where it is implied we are at the t-th step in a trajectory, as*

$$V(s_t) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}) \,\middle|\, s_t\right]$$

This is coherent with the notion of cumulative (discounted) future rewards which defined the value function (2.1), but not with the notation. In the finite horizon setting, a more correct notation would be to write $V(t, s_t)$ and make the dependence on the starting time t of the discounting explicit.

This notion can be generalized to other cases, such as the case where the rewards are generated by the transitions (s_t, a_t, s_{t+1}) rather than the (state, action) couple:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \,\middle|\, s_0 = s\right]$$

or the stochastic reward case where r_t is distributed according to some law parameterized by the state and action.

Under a deterministic policy $\pi\colon \mathcal{S} \to \mathcal{A}$ and associated decision rule $d^\pi(s) = \pi(s)$, the dynamic programming principle leads to a dynamic programming equation called the **Bellman equation**:

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') V^\pi(s') \tag{2.2}$$

which is a fixed-point condition.

The Bellman equation can be used to *evaluate* a policy π, that is compute its associated value function V^π.

Its fixed-point structure can be reformulated in terms of an operator on a function space, called the **Bellman operator**:

$$\mathcal{T}^\pi v(s) \coloneqq r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') v(s')$$

which can be shown to be contractant, ensuring the existence of a solution.

Remark 2 *We have the following, possible generalizations (see Sutton's book [SB18, chap. 3, 4] for further details):*

- *for stochastic policies $\pi\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$, the sum in eq. (2.2) becomes*

$$\sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi(s, a) p(s, a, s') V^\pi(s')$$

- *in non-discrete state spaces, the sum can be replaced by an integral with respect to a measure $p(s, \pi(s), \mathrm{d}s')$*
- *if the rewards are given for transitions as $r(s, a, s')$, we introduce*

$$r(s, a) = \sum_{s' \in \mathcal{S}} p(s, a, s') r(s, a, s'),$$

and the Bellman equation can be rewritten

$$V^\pi(s) = \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') (r(s, \pi(s), s') + \gamma V^\pi(s'))$$

In a finite setting with small state and action spaces, it possible to solve the Bellman equation directly by Gaussian elimination on the corresponding matrices and vectors.

2.1.1 The optimal value function

Solving a Markov Decision Process involves finding an **optimal** policy π^* that will maximize the expected rewards in the long run when starting from a given state s_0 (or distribution $s_0 \sim p$).

Definition 3 (Optimal policy and value function) *Given a set of policies Π, the* **optimal value function** *satisfies*

$$V^* = \max_\pi V^\pi \tag{2.3}$$

for every state $s \in \mathcal{S}$. An **optimal policy** *π^* is one that satisfies the maximum.*

Strictly speaking, we are taking a maximal policy with respect to a partial ordering on policies that compares them by looking at their respective value functions:

$$\pi_1 \leq \pi_2 \iff [V^{\pi_1}(s) \leq V^{\pi_2}(s) \ \forall s \in \mathcal{S}]$$

Proposition 1 (Optimal Bellman equation) *The optimal value function V^* obeys a dynamic programming principle, the* **optimal Bellman equation**.

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ r(s,a) + \sum_{s' \in \mathcal{S}} p(s,a,s') V^*(s') \right\} \tag{2.4}$$

This is also a fixed-point condition, which can once again be expressed in terms of an operator called the **Bellman optimal operator**:

$$\mathcal{T}^* v(s) \coloneqq \max_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') v(s') \right\}.$$

Then, we have that the optimal value function satisfies the equation

$$\mathcal{T}^* V^* = V^*$$

2.2 The Q-function

Definition 4 (action-state value function) *The action-state value function of a policy π is the function $Q^\pi \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined by*

$$Q^\pi(s,a) \coloneqq \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \,\middle|\, s_0 = s, a_0 = a \right] \tag{2.5}$$

where the trajectory τ is generated under the policy π. The horizon T of the problem can be finite or infinite (T can be a stopping time).

The action-state value function Q^π has an obvious link to the value function V^π. For any state $s \in \mathcal{S}$, it holds that

$$V^\pi(s) = \mathbb{E}_{a \sim \pi(s,\cdot)} \left[Q^\pi(s,a) \right] = \sum_a \pi(s,a) Q^\pi(s,a) \tag{2.6}$$

There's also a reverse equality:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[r(s,a) + V^\pi(s_{t+1}) \mid s_t = s \right] = r(s,a) + \sum_{s'} p(s,a,s') V^\pi(s') \tag{2.7}$$

The action-state value function has its own **Bellman equation**:

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s,a,s') Q^\pi(s',a) \tag{2.8}$$

2.2.1 The optimal Q-function

Definition 5 (Optimal action value function) *Optimal policies as defined in section 2.1.1 also share the same* **optimal action-state value function**

$$Q^* = \max_\pi Q^\pi \tag{2.9}$$

for all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$.

Here, the maximum is also understood with respect to a partial ordering on policies.
The optimal value Q^* follows its own variant of the **optimal Bellman equation**:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') \max_{a'} Q^*(s',a') \tag{2.10}$$

Link between the optimal value functions

We also have a link between the optimal value functions. Using the Bellman optimality equation on Q (2.10), we get:

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}_{s' \sim d(s,a)} \left[r(s,a) + \gamma V^*(s') \right] \\ &= r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') V^*(s') \end{aligned} \tag{2.11}$$

Conversely, the optimal value function can be expressed by reformulating the Bellman optimality equation (2.4):

$$V^*(s) = Q^*(s, \pi^*(s)) = \max_{a \in \mathcal{A}} Q^*(s,a) \tag{2.12}$$

Obtaining the optimal policy

Given the optimal action-state value function Q^*, you can define an optimal policy by taking

$$\pi^*(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(s,a). \tag{2.13}$$

By the way, this shows we can always find a deterministic optimal policy. This is an **important** property of the optimal Q-function which we will use to design policy optimization algorithms.

2.3 Algorithms for dynamic programming

2.3.1 Value Iteration

Under suitable hypotheses, it can be shown that the Bellman operators $\mathcal{T}^\pi$ and $\mathcal{T}^*$ are contractions with respect to the infinity norm with Lipschitz constant γ.

This means we can approximate the optimal value function by iterating the optimal Bellman operator $\mathcal{T}^*$. This leads to the following *value iteration* algorithm:

Algorithm 2.1: Value iteration

Input: Rewards $r(s,a)$, transitions $p(s,a,s')$, initial value proposal V_0, number of iterations K

foreach $k = 1, \dots, K$ **do**
  $V_k \leftarrow \mathcal{T}^* V_{k-1}$;
foreach $s \in \mathcal{S}$ **do**
  $\pi_K(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') V_K(s') \right\}$;
return Policy π_K, value V_K;

Remark 3 *Each iterate V_k is not necessarily the value function of some policy.*

This algorithm is especially useful for discrete state and action spaces. It can be significantly sped up if we have sparse representations of the rewards and transitions.

The stopping condition for the iterations can either be that a fixed number of iterations K has been reached, or an ε-convergence criterion, i.e. stop when value function increments fall below some threshold (e.g. $\|V_{k+1} - V_k\|_\infty \leq \varepsilon$).

Proposition 2 (Convergence speed of VI) *If given an upper bound $r_{\max} \geq \sup |r(\cdot)|$ on the*

rewards, we ε-converge in at most

$$K = \frac{\log(r_{\max})/\varepsilon}{\log(1/\gamma)}$$

steps.

Performance guarantees. The performance loss between the final policy π_K and the optimal policy π^* is measured by the distance between their values:

$$\|V^* - V^{\pi_K}\|_\infty \leq \frac{2\gamma}{1-\gamma}\|V^* - V_K\|_\infty$$

Alternative termination condition.

$$\|V_k - V_{k-1}\|_\infty - \min_{s\in\mathcal{S}} |V_k(s) - V_{k-1}(s)| \leq \varepsilon$$

Complexity. The temporal complexity of VI is $O(KS^2A)$ – indeed the complexity of applying the Bellman operator is $O(S^2A)$

Variants

Asynchronous VI. The idea is not to perform the Bellman iterate over the entire value function V_k at once, but perform it on a single state: for every k,

- choose some state $s_k \in \mathcal{S}$
- update its value $V_{k+1}(s_k) \leftarrow \mathcal{T}^* V_k(s_k)$.

This reduces the complexity of each iterate $O(SA)$ in exchange for a larger number of iterates: for the obvious round-robin selection rue for s_k we have at most $O(KS)$ iterations. A smarter selection rule can decrease that number significantly by prioritizing states.

2.3.2 Policy Iteration

The idea of *policy iteration* is to use the action-state value function Q^π to iteratively update the policy.

Greedy improvements. We can improve over a policy π and define a new policy π' by acting greedily:

$$\pi'(s) := \operatorname*{argmax}_{a\in\mathcal{A}} Q^\pi(s,a)$$

It can then be shown that $V^{\pi'} \geq V^\pi$, and that improvements stop if and only if the Bellman optimality condition is satisfied.

To decide when to stop, we can use a set number of iterations, or use a ε-convergence criterion as suggested before for value iteration.

Policy iteration can also be performed for other improvement algorithms. For instance, **ε-greedy improvement** performs greedy updates at each state probabilistically, taking the greedy with probability $1-\varepsilon$ and otherwise taking a random action.

Chapter 3

Incremental solving of MDPs: Reinforcement Learning

As in regular control theory, dynamic programming opens up a lot of venues to solve MDPs we know everything about. But what do we do when we don't have a full model of the transitions and rewards of our system?

There are two ideas: use several **episodes** (trajectories) of agents following some policy (or policies) to compute empirical returns and build estimates of the value, or update our estimates as-we-go as we follow an agent's actions and the rewards it gets (**online learning**).

There are two aspects that need to be discussed:

- **policy evaluation** (*prediction*)
- **policy optimization** (*control*)

We will start by discussing approximate policy evaluation techniques and see how they apply to optimizing the policies as we go.

For now, we suppose we are in the **tabular case**: the value functions can be stored as arrays in memory.

3.1 Monte Carlo Policy Evaluation

The idea is naïve: generate episodes τ_i under a policy π starting at some state s_0, and compute their empirical returns

$$R(\tau_i) = \sum_{t=0}^{T_i} \gamma^t r_{t,i}.$$

The value estimate is then the empirical mean:

$$\widehat{V}^\pi(s_0) = \frac{1}{n} \sum_{i=1}^{n} R(\tau_i) \tag{3.1}$$

Here, the beginning state s_0 is fixed so we are only estimating its value.

Dealing with non-episodic problems. If the underlying problem does not necessarily terminate in finite time, we can always truncate it after a given number of time steps and say it has indeed "reset". For instance, a trade execution algorithm does not have a terminal state so we might want to generate episodes by resetting after some time has passed. This means ignoring a term $\sum_{t'=H+1}^{\infty} \gamma^{t'} r_{t'}$ in the return past a certain horizon H. In that case, the MC estimator converges to a truncated value function V_H^π which differs from the true value V^π by

$$|V_H^\pi(s_0) - V^\pi(s_0)| \leq \gamma^H \frac{\|r(\cdot)\|_\infty}{1-\gamma}.$$

3.1.1 Incremental MC

We can easily see that the n-sample MC estimate can be seen as an update of the $(n-1)$-sample one:

$$\widehat{V}_n^\pi(s_0) = \alpha_n R(\tau_n) + (1-\alpha_n)\widehat{V}_{n-1}^\pi(s_0)$$

where

$$\alpha_n = \frac{1}{n}$$

is the **learning rate** of the scheme.

Other learning rates can be used, and we have the following result ensuring convergence in the general case:

Proposition 3 *Suppose the learning rate (α_n) satisfies the Robbins-Monro condition:*

$$\sum_{n=0}^{\infty} \alpha_n = \infty \quad \sum_{n=0}^{\infty} \alpha_n^2 < \infty \tag{3.2}$$

then the incremental MC estimate converges to the real value:

$$\widehat{V}^\pi(s_0) \xrightarrow[n\to\infty]{} V^\pi(s_0).$$

Incremental MC is also called **one-step temporal difference** TD(1). It does not allow online learning.

3.2 Temporal difference (TD) estimation

3.2.1 Using the Bellman equation: TD(0) estimation

The real value function V^π satisfies the Bellman equation. This means that the **temporal difference error** of a good estimate $\widehat{V}^\pi$ of V^π, defined after each transition and reward s_t, r_t, s_{t+1}

$$\delta_t = r_t + \gamma \widehat{V}^\pi(s_{t+1}) - \widehat{V}^\pi(s_t), \tag{3.3}$$

should be small.

The TD(0) updates are performed as

$$\begin{aligned}\widehat{V}^\pi(s_t) &\leftarrow \widehat{V}^\pi(s_t) + \alpha_t \delta_t \\ &= (1-\alpha_t)\widehat{V}^\pi(s_t) + \alpha_t(r_t + \gamma \widehat{V}^\pi(s_{t+1}))\end{aligned}$$

Remark 4 *Introducing the **bootstrapped return** of the trajectory*

$$G_t = r_t + \widehat{V}^\pi(s_t),$$

we can rewrite the update as

$$\widehat{V}^\pi(s_t) \leftarrow \widehat{V}^\pi(s_t) + \alpha_t(G_t - \widehat{V}^\pi(s_t))$$

Choosing the learning rate. For incremental MC, a natural learning rate that works was $\alpha_n = 1/n$. For this temporal difference update scheme, it can be shown that we can use an **adaptive state-dependent learning rate** $\alpha_t = \alpha(s_t)$:

Proposition 4 *Set the learning rate to $\alpha_t = \alpha(N_t(s_t))$ where $N_t(s)$ is the number of times state $s \in \mathcal{S}$ has been visited before time t. If we suppose $N_t(s) \to \infty$ (every state is visited infinitely many times), then the conditions*

$$\sum_{n=0}^{\infty} \alpha(n) = \infty \quad \sum_{n=0}^{\infty} \alpha(n)^2 < \infty \tag{3.4}$$

lead to convergence of the scheme.

This scheme doesn't require the entire trajectory to be known to perform an update, allowing for fully online learning.

3.2.2 n-step temporal difference (see [SB18, chap. 7])

The idea is to use several rewards to compute the temporal difference: the n-step $\mathsf{TD}(n)$ reads

$$\delta_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \widehat{V}^\pi(s_{t+n}) - \widehat{V}^\pi(s_t) \tag{3.5}$$

The Q-function update is then done as usual.

3.2.3 Temporal difference $\mathsf{TD}(\lambda)$

These schemes average between incremental MC and $\mathsf{TD}(n)$. The scheme is defined for $0 < \lambda < 1$ as:

$$\widehat{V}^\pi(s_t) \leftarrow \widehat{V}^\pi(s_t) + \alpha(s_t) \sum_{t'=t}^{T} (\gamma\lambda)^{t'-t} \delta_{t'} \tag{3.6}$$

where $(\delta_{t'})_{t'}$ is the sequence of temporal differences errors of the trajectory.

This scheme *does* require the entire trajectory to be known before performing updates, but it re-discounts them using the parameter λ.

Eligibility traces

The idea is to be able to learn online while updating everything (propagating the reward signal) at once. Sutton and Barto [SB18] explain that

At every transition s_t, r_t, s_{t+1}, we define the temporal difference δ_t and update the **eligibility traces** $z(s)$ as

$$\forall s \in \mathcal{S} \quad z(s) = \begin{cases} \lambda z(s) & \text{if } s \neq s_t \\ 1 + \lambda z(s) & \text{if } s = s_t \\ 0 & \text{if } s_t = s_0 \text{ (reset the traces)} \end{cases}$$

For policy evaluation, we update the value function estimate for all states $s \in \mathcal{S}$ using the eligibility traces with

$$\widehat{V}^\pi(s) \leftarrow \widehat{V}^\pi(s) + \alpha(s) z(s) \delta_t \tag{3.7}$$

This introduces a compromise between propagating rewards faster than $\mathsf{TD}(0)$ and having smaller variance than incremental MC.

Remark 5 *All the previous ideas for policy evaluation on the value function V^π readily apply to evaluating the action-state value Q^π.*

3.3 Application of TD to policy improvement

After introducing the previous temporal difference notions for policy evaluation, we can see how they work for **policy improvement** schemes. The basic idea is to alternate the previous temporal-difference policy evaluation techniques with policy improvement ideas from chapter 2 which exploit the Q value function.

3.3.1 SARSA

We recall the basic property (2.13) of the optimal policy with respect to Q^*. This suggests an idea for **on-policy control**: define an exploration policy using the softmax function on Q and a temperature parameter τ:

$$\pi_Q(s,a) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a' \in \mathcal{A}} \exp(Q(s,a')/\tau)}$$

After observing a transition (s_t, a_t, r_t, s_{t+1}), we take the next action a_{t+1} according to π_Q and update using the TD(1) scheme: compute the temporal difference of the Q-function

$$\delta_t = r_t + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t) \tag{3.8}$$

and update $\widehat{Q}$

$$\begin{aligned} \widehat{Q}(s_t, a_t) &\leftarrow \widehat{Q}(s_t, a_t) + \alpha_t \delta_t \\ &= (1 - \alpha_t)\widehat{Q}(s_t, a_t) + \alpha_t (r_t + \widehat{Q}(s_{t+1}, a_{t+1})) \end{aligned}$$

We ensure the policy continues to improve by decreasing the temperature τ to 0 and becoming more and more greedy in our action selection (keeping the same τ reduces the algorithm to some kind of softmax policy evaluation).

Each update of the algorithm uses the entire state-action transition $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ (from which the name SARSA comes from).

Algorithm 3.1: SARSA (softmax version)

Input: Initial state s_0, initial temperature τ
Take action $a_0 \sim \pi_{\widehat{Q}}(s_0, \cdot)$;
repeat
  Get reward, next state (r_t, s_{t+1});
  Take the next action $a_{t+1} \sim \pi_{\widehat{Q}}(s_{t+1}, \cdot)$;
  Compute TD error $\delta_t = r_t + \gamma \widehat{Q}(s_{t+1}, a_{t+1}) - \widehat{Q}(s_t, a_t)$;
  $\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha_t \delta_t$;
  $t \leftarrow t + 1$;
until *trajectory terminates*;
foreach $s \in \mathcal{S}$ **do**
  $\tilde{\pi}(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \widehat{Q}(s, a)$;
return Policy $\tilde{\pi}$;

The use of the value candidate $\widehat{Q}$ to define the policy used to select the action means that SARSA is an **on-policy learning algorithm**.

We can use several other exploration policies, such as ε-greedy exploration (where we take the $\widehat{Q}$-optimal action with probability ε and otherwise take a random action).

Extension to TD(λ) update schemes is straightforward using eligibility traces (see [SB18, chap. 12]), but will lead to slightly different convergence properties due to a compromise between signal (reward) propagation and variance.

3.3.2 Q-learning

The idea is to introduce the optimal TD error for the Q value function

$$\delta_t = r_t + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a') - \widehat{Q}(s_t, a_t) \tag{3.9}$$

and updating the Q estimate using that:

$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha_t \delta_t$$

This is not the same as using the greedy action to update: this is upper-bounding the Q-value TD error (3.8) and using that upper bound to update the value.

This means Q-learning is an **off-policy learning algorithm**, whatever exploration policy we use to take actions.

Algorithm 3.2: Q-learning

Input: Initial state s_0, exploration policy

repeat
  Take action a_t according to the exploration policy;
  Get reward and next state (r_t, s_{t+1});
  Compute TD error $\delta_t = r_t + \gamma \max_{a'} \widehat{Q}(s_{t+1}, a') - \widehat{Q}(s_t, a_t)$;
  $\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha_t \delta_t$;
until *trajectory terminates*;
`// Compute the policy from the` Q `value`
foreach $s \in \mathcal{S}$ **do**
  $\tilde{\pi}(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \widehat{Q}(s, a)$;
return Policy $\tilde{\pi}$;

One option for the exploration policy is to use the ε-greedy policy derived from $\widehat{Q}$. If we use an absolute greedy policy for exploration, then this becomes the same as SARSA with that policy.

What about the learning rate. We can introduce, just as in evaluating the value V (see proposition 4), a action-state count $N_t(s, a)$ and set an adaptive learning rate

$$\alpha_t = \alpha(N_t(s_t, a_t))$$

and the Robbins-Monro conditions on the function α work out the same.

Chapter 4

Approximate Reinforcement Learning

The previous chapter demonstrated how to solve Markov Decision Processes where the state-control dynamics and reward signals are inaccessible, but where we are in a tabular setting where the value functions can be represented exactly.

Now we add an **additional relaxation** over the standard MDP assumptions: the state and action spaces are too large (infinite, potentially continuous) to access their values in a tabular manner: this can be the case in games like Backgammon. Thus, we can use neither dynamic programming nor the RL algorithms discussed before. However, **we can access the value functions through approximation**.

We define parametric function approximators:

- $v_\theta \approx V^\pi$
- $q_\theta \approx Q^\pi$

where the weights will be updated from samples (whole episodes, or transitions in an *online* manner) using either MC or TD.

4.1 Approximate policy evaluation

4.1.1 Monte Carlo

Given sample trajectories $(s_{0,i}, a_{0,i}, r_{0,i}, \ldots)$, we want the approximator to match the empirical returns

$$R_i = \sum_{t=0}^{T_i} \gamma^t r_{t,i} = v_\theta(s_{0,i}) + \varepsilon_i$$

Our objective is to minimize the empirical error

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_i \left(v_\theta(s_{0,i}) - R_i\right)^2 \tag{4.1}$$

4.1.2 Approximate Temporal difference

Say we want to approximate the value function of a policy π.

We target the bootstrapped returns

$$G_t = r_t + \gamma v_\theta(s_{t+1})$$

and the parameter update becomes

$$\theta \leftarrow \theta - \alpha_t (v_\theta(s_t) - G_t) \nabla_\theta v_\theta(s_t) \tag{4.2}$$

This is a semi-gradient update (where we minimize the MSE for a sample and where the derivative wrt $v_\theta(s_{t+1})$ is ignored).

This principle can once again be extended to TD(λ) differences by using eligibility traces.

Linear TD

This is the simplest class of approximators: use a dictionary of functions $\{\phi_1, \dots, \phi_d\}$. Then the function approximators read

$$v_\theta(s) = \phi(s)^\intercal \theta$$

and in the finite state space case $|\mathcal{S}| = S$ we can even write

$$\mathbf{V}_\theta = \Phi\theta$$

where $\Phi = \begin{bmatrix} \phi(s_1)^\intercal \\ \vdots \\ \phi(s_S)^\intercal \end{bmatrix} \in \mathbb{R}^{S\times d}$.

After observing (s_t, r_t, s_{t+1}), the TD update boils down to

$$\theta \leftarrow \theta - \alpha_t(\phi(s_t)^\intercal\theta - G_t)\phi(s_t) \tag{4.3}$$

Proposition 5 (Convergence of the linear TD approximator) *If the policy we want to evaluate has a stationary distribution $\rho^\pi \in \mathbb{R}^{\mathcal{S}}$, and we denote $D = \mathrm{diag}(\rho^\pi)$ its matrix and Π_D the projection with respect to the D-norm $x \mapsto x^\intercal Dx$, then the linear TD converges to a parameter θ^* that satisfies*

$$\Phi\theta^* = (\Pi_D \mathcal{T}^\pi)\Phi\theta^* \tag{4.4}$$

i.e. the optimal approximation $\mathbf{V}_{\theta^*} = \Phi\theta^*$ is a fixed point of the ***projected Bellman operator*** *$\Pi_D\mathcal{T}^\pi$.*

The error reads

$$L_D(\theta^*) \leq \frac{1}{\sqrt{1-\gamma^2}} \min_\theta L_D(\theta) \tag{4.5}$$

where L_D is the expected loss wrt D.

4.2 Approximate policy optimization

In this learning case, as before for RL algorithms, we focus on the action value function Q, on which we define a parametric approximator.

4.2.1 Approximate Q-learning

We re-use the idea of Q-learning, that is trying to minimize the TD error upper bound (3.9). Once again, we reformulate the function approximation step as targeting a bootstrap estimate of the value

$$G_t = r_t + \gamma \max_{a'\in\mathcal{A}} q_\theta(s_{t+1}, a')$$

In the **online** case, we target the bootstrapped loss

$$\mathcal{L}(\theta) = (q_\theta(s_t, a_t) - G_t)^2$$

and perform updates

$$\theta \leftarrow \theta - \alpha_t(q_\theta(s_t, a_t) - G_t)\nabla_\theta q_\theta(s_t, a_t) \tag{4.6}$$

Just as before, this is a semi-gradient update where we ignore the dependence of the bootstrapped return on θ.

Unfortunately, this bootstrapping method might diverge, even with simple approximators. This makes Deep Q-learning (DQN) practically difficult.

There are **several problems** to overcome:

- **correlated samples** because we update sequentially
- **oscillations** due to getting the Q-value from the policy and vice-versa
- the **scale** of the actual optimal Q values being **unknown**
- an **over-estimation effect** due to always choosing the maximizing action to adjust the parameter θ

These are all problems DeepMind encountered when designing an algorithm to play Space Invaders, as detailed in Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602. The authors came up with ways of solving them:

- **experience replay**
- **target networks**
- **reward normalization**
- **double Q-learning**

4.2.2 Experience replay

The idea is to store the transitions $e_t = (s_t, a_t, r_t, s_{t+1})$ we observe when playing the game in an **experience buffer**.

Algorithm 4.1: Experience replay (episodic)

for $i = 1, \dots, n$ **do**
  repeat
    Take action $a_{t,i}$;
    Get reward, next state $(r_{t,i}, s_{t+1,i})$;
    Store the transition $e_{t,i} = (s_{t,i}, a_{t,i}, r_{t,i}, s_{t+1,i})$ in buffer $\mathcal{D}$;
    $t \leftarrow t + 1$;
    Update $\boldsymbol{\theta}$ based on a mini-batch $\boldsymbol{\mathcal{D}_m \subset \mathcal{D}}$;
  until *until termination*;
return Policy $\pi^*(s) \in \operatorname{argmax}_a q_\theta(s, a)$;

The update on the mini-batch can be a gradient mini-batch using the method previously discussed (bootstrapping on the transitions from the mini-batch), or we can use the target network:

4.2.3 Target networks

The idea of using a target network $q_{\bar{\theta}}$ is to **stall** the evolution of the main network we want to learn with by making it **use a return bootstrapped using the target network**:

$$\bar{G}_t = r_t + \gamma \max_{a' \in \mathcal{A}} q_{\bar{\theta}}(s_{t+1}, a').$$

Mini-batch update. The main network updates on the mini-batch $\mathcal{D}_m$ by minimizing the loss

$$\mathcal{L}_{\mathcal{D}_m}(\theta) = \mathbb{E}_{(s_i, a_i, r_i, s_{i+1})} \left[(q_\theta(s_i, a_i) - \bar{G}_i)^2 \right] \tag{4.7}$$

This leads to the following algorithm:

Algorithm 4.2: Experience replay with target network

Input: Initial parameters θ, $\bar{\theta}$, target update interval C

for $i = 1, \ldots, n$ **do**
  Set initial state $s_{0,i}$;
  repeat
    Take action $a_{t,i}$;
    Get reward, next state $(r_{t,i}, s_{t+1,i})$;
    Store the transition $e_{t,i} = (s_{t,i}, a_{t,i}, r_{t,i}, s_{t+1,i})$ in buffer $\mathcal{D}$;
    $t \leftarrow t + 1$;
    Update network q_θ from mini-batch using target-bootstrapped returns;
    Every C steps: $\bar{\theta} \leftarrow \theta$; `// update the target network`
  until *until termination*;
return Policy $\pi^*(s) \in \operatorname{argmax}_a q_\theta(s, a)$;

4.2.4 Stabilization with double Q-learning

We tame the over-estimation effect by using **two** approximators with parameters θ_1, θ_2. The update procedure is now

- Let $a_2^* = \operatorname{argmax}_a q_{\theta_2}(s_{t+1}, a)$
- Set the return bootstrap $G_t = r_t + \gamma q_{\theta_1}(s_{t+1}, a_2^*)$
- Update:
$$\theta_1 \leftarrow \theta_1 - \alpha_t(q_{\theta_1}(s_t, a_t) - G_t)\nabla_{\theta_1} q_{\theta_1}(s_t, a_t)$$
- Repeat by alternating θ_1 and θ_2.

Chapter 5

Approximate solving of MDPs with Policy Gradients

Solving MDPs is seeking the maximizing policy of the value function. For approximate solving of MDPs, we target what could be a more general **policy performance metric**. Often, it is indeed the value function

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right] = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right] \tag{5.1}$$

where $\tau = \{s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T\}$ and $R(\tau) = \sum_{t=0}^{T}\gamma^t r_t$ is the total return of the trajectory τ. We use the shorthand r_t for the reward $r(s_t, a_t)$ – or more generally a transition reward $r(s_t, a_t, s_{t+1})$ or stochastic reward distributed as $r_t \sim p(\cdot, s_t, a_t, s_{t+1})$. The expectation J is either conditional on a given starting point s_0, or on a distribution for it.[1]

We seek to compute the maximizing policy in a parametric search space $\{\pi_\theta : \theta \in \Theta\}$:

$$\max_\theta J(\pi_\theta)$$

Iterative methods could be used if J can be obtained in closed form; if these are not given they could be estimated by Monte Carlo methods, which would be very expensive and wasteful.

Instead, we will update the estimate as we go and simulate trajectories, by iteratively updating the policy parameter θ using a gradient ascent method with an estimated gradient.

Proposition 6 (Gradient under a parametric law) *Given a set of probability models $\{P_\theta : \theta \in \Theta \subseteq \mathbb{R}^d\}$ on a set $\mathcal{X}$ and a function $f\colon \mathcal{X} \to \mathbb{R}$, we have that*

$$\nabla_\theta \mathbb{E}_{X\sim P_\theta}[f(X)] = \mathbb{E}_{X\sim P_\theta}[f(X)\nabla_\theta \log P_\theta(X)]$$

This is a useful property for deriving estimators of the derivatives in optimization problems with stochastic objectives.

Generalization to the case where f also depends on θ is straightforward.

This can be shown either by either writing the expectation as an integral, or by a change of measures with a Radon-Nikodym derivative.

Proposition 6 allows us to write the gradient of (5.1), called the **policy gradient** as an expectation:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(s_t, a_t)\right] \tag{5.2}$$

and we will need to derive estimations for this quantity.

[1] For instance, OpenAI Gym's `CartPole-v1` environment has a stochastic initial state s_0.

There are other ways of writing the policy gradient, such as (see [SB18, chap. 13])

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_s \left[\sum_a Q^{\pi_\theta}(s,a) \nabla \pi_\theta(s,a) \right]$$

5.1 Monte Carlo policy gradient: the REINFORCE algorithm

The idea. The policy gradient (5.2) is an expectation which can be estimated using Monte Carlo approximation. We obtain the following estimate:

$$\widehat{\nabla_\theta J}(\pi_\theta) = \frac{1}{M} \sum_{i=1}^{M} R(\tau_i) \sum_{t=0}^{T_i} \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{5.3}$$

This is an unbiased Monte Carlo estimate of the policy gradient. It only requires suitable regularity of the parametric policy model $\theta \longmapsto \pi_\theta$.

Remark 6 *Equation (5.3) can be used as-is for functions with simple closed-form derivatives. In an automatic differentiation framework such as* PyTorch*, we can instead get the policy gradient from a computational graph with the following pseudo-loss function:*

$$\begin{aligned} \tilde{J}(\theta) &= \frac{1}{M} \sum_{i=1}^{M} R(\tau_i) \sum_{t=0}^{T_i} \log \pi_\theta(s_t^i, a_t^i) \\ &= \frac{1}{M} \sum_{i=1}^{M} \left(\sum_{t=0}^{T_i} \gamma^t r_t^i \right) \sum_{t=0}^{T_i} \log \pi_\theta(s_t^i, a_t^i) \end{aligned} \tag{5.4}$$

We will simulate multiple trajectories (*episodes*) to perform the gradient update: this is the similar to what is done with batch, mini-batch or stochastic gradient steps. This leads to algorithm 5.1.

Algorithm 5.1: Monte Carlo Policy Gradient (REINFORCE)

Input: Arbitrary initial policy π_{θ_0}.
Output: Optimal parametric policy π_{θ^*}.

repeat
  Simulate a trajectory τ;
  $\boldsymbol{g} \leftarrow \left(\sum_{t=0}^{T} \gamma^t r_t\right) \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(s_t, a_t)$; `// policy gradient`
  $\theta \leftarrow \theta + \alpha \boldsymbol{g}$; `// update parameter`
until *finished*;

This algorithm can be modified in several ways, by performing the gradient step at every time in the process, or on the entire batch of trajectories. Sutton and Barto [SB18] actually use the more fine-grained update scheme wherein the parameter θ is updated at each step t of every episode[2].

5.1.1 Variance reduction: temporal structure

We can re-weigh the log-probability gradients in eq. (5.2) by exploiting the fact that, for any time t, the cumulative rewards $\sum_{t'=0}^{t-1} \gamma^{t'} r_{t'}$ from 0 to $t-1$ are measurable with respect to the trajectory up to t, $\tau_{0:t}$:

[2] This allows for fully online learning.

Proposition 7 *The policy gradient (5.2) can be rewritten as*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \sum_{t'=t}^{T} \gamma^{t'} r_{t'} \nabla_\theta \log \pi_\theta(s_t, a_t) \right] \tag{5.5}$$

which leads to the policy gradient estimate

$$\widehat{\nabla_\theta J}(\pi_\theta) = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T_i} \gamma^t G_t^i \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{5.6}$$

where[a] $G_t^i = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^i$.

[a] This quantity is an estimate of the Q-function $Q^\pi(s_t, a_t) = \mathbb{E}[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \mid s_t, a_t]$.

5.1.2 Variance reduction with baselines

Given any **baseline** function $b\colon \mathcal{S} \to \mathbb{R}$, we can rewrite the policy gradient again as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \left(\sum_{t'=t}^{T} \gamma^{t'} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(s_t, a_t) \right] \tag{5.7}$$

The resulting policy gradient estimate we get is

$$\widehat{\nabla_\theta J}(\pi_\theta) = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T_i} \gamma^t (G_t^i - b(s_t^i)) \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{5.8}$$

which is still an unbiased estimate – but with the added benefit of **variance reduction** if b is chosen right.

The best baseline.

It can be shown that the "best" baseline in terms of variance reduction is actually the value function:

$$b^*(s_t) = \mathbb{E}_{\pi_\theta} \left[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \,\middle|\, s_t \right] = V^\pi(s_t)$$

…which we are already trying to approximate. This suggests that we do some kind of **bootstrapping**. The approximate policy evaluation ideas from the previous **??** will come in handy.

The obvious first idea is nonparametric Monte Carlo estimation:

$$\widehat{b}(s_t) = \frac{1}{M} \sum_{i=1}^{M} G_t^i$$

This baseline is an unbiased estimate: following eq. (2.6) the expectation of the Q-function estimate G_t under the policy π_θ is

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(s_t, \cdot)} \mathbb{E}_{\pi_\theta} [G_t] = \mathbb{E}_{a_t \sim \pi_\theta(s_t, \cdot)} \mathbb{E}_{\pi_\theta} \left[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \,\middle|\, s_t, a_t \right]$$

However, this introduces additional variance and the baseline might become noisy.

5.1.3 Parametric Bootstrapping of the baseline

We define the bootstrap estimate $b(s) := \widehat{v}_\nu(s)$ lying in a parametric search space $\{\widehat{v}_\nu : \nu \in \mathcal{V}\}$. The value parameter can be iteratively updated using gradient steps alternatively with the policy parameter θ.

For a given trajectory sample $\tau = \{s_0, a_0, r_0, \ldots\}$, introduce the mean-squared error between the returns $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ of the entire trajectory (a nonparametric estimate of the value function) and the output of the value approximator:

$$\mathcal{L}(\nu; \tau) = \sum_{t=0}^{T} (G_t - \widehat{v}_\nu(s_t))^2 \tag{5.9}$$

Then before each update of the policy π_θ, update the value parameter ν using either the gradient of $\mathcal{L}$.

The adapted episodic learning algorithm with an adaptive baseline is as follows:

Algorithm 5.2: REINFORCE with parametric baseline

repeat
  Simulate a trajectory τ;
  Compute the returns (G_t) of the trajectory;
  $\boldsymbol{g} \leftarrow \sum_{t=0}^{T} \gamma^t (G_t - \widehat{v}_\nu(s_t)) \nabla_\theta \log \pi_\theta(s_t, a_t)$; `// policy gradient`
  $\nu \leftarrow \nu - \beta \nabla_\nu \mathcal{L}(\nu; \tau)$; `// update value estimate`
  $\theta \leftarrow \theta + \alpha \boldsymbol{g}$; `// update policy`
until *finished*;

As before (see remark 6), this algorithm can be implemented in an automatic differentiation framework like PyTorch by defining the right computational graph. The associated pseudo-loss would be

$$\tilde{J}(\theta) = \sum_{t=0}^{T} \gamma^t (G_t - \widehat{v}_\nu(s_t)) \log \pi_\theta(s_t, a_t)$$

Remark 7 *The difference between the returns and the value approximation $G_t - \widehat{v}_\nu(s_t)$ is a (biased) estimate of the temporal-difference error $\delta_t = r_t + \gamma \widehat{v}(s_{t+1}) - \widehat{v}(s_t)$. Thus, the mean-squared error $\mathcal{L}$ is an estimate of the Bellman error of the value approximator $\widehat{v}_\nu$.*

5.2 Bootstrapped returns: Actor-Critic algorithms

The idea. The enhanced REINFORCE algorithm builds estimates of the value function to compute the policy gradient: this Monte Carlo method is computationally expensive, and may still lead to high variance. To combat this, it might be a good idea to *learn* from the Monte Carlo estimates in a way that gives a consistent estimate that follows the policy gradient updates.

To achieve this, the class of **actor-critic methods** introduces a second search space for approximation of the state(-action) value function and uses **bootstrapped temporal-difference estimates** for the returns G_t, by taking inspiration from TD policy evaluation (see **??**). The supervised baseline of section 5.1.3 was already a first step towards this, but this time we actually use the value estimate for policy evaluation and not only for variance reduction. The subtle difference is further explained in [SB18, chap. 13.5].

5.2.1 Actor-Critic

Learning the policy is still done by gradient steps, using estimates of the form eq. (5.8). But this time, we replace in the objective J the Monte Carlo returns G_t, which estimate the Q-function, by a parametric estimator $\widehat{q}_\omega(s_t, a_t)$ called the **critic**. As the algorithm runs, the critic learns how to value each action suggested by the learned policy π (called the **actor**) by looking at the bootstrapped return estimates. The policy gradient becomes

$$\widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=0}^{T_i} \widehat{q}_\omega(s_t^i, a_t^i)\nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{5.10}$$

Now, we want the critic $\widehat{q}_\omega$ to minimize the **Bellman error**

$$\mathbb{E}_{\pi_\theta}\left[(r_t + \gamma\widehat{q}_\omega(s_{t+1}, a_{t+1}) - \widehat{q}_\omega(s_t, a_t))^2 \mid s_t, a_t\right]$$

because having it as small as possible will ensure that it follows the dynamic programming principle, making it a "good" estimate of the real action-state value function Q^{π_θ}. The above expectation can be estimated using the TD(0) error of $\widehat{q}_\omega$:

$$\delta_t = r_t + \gamma\widehat{q}_\omega(s_{t+1}, a_{t+1}) - \widehat{q}_\omega(s_t, a_t).$$

which we will seek to minimize for every time step or episode alternatively with the actor loss.

This leads to the following algorithm:

Algorithm 5.3: Actor-Critic

Input: Initial policy parameter θ, value parameter ω
Output: Policy π_{θ^*}, action-state value approximation $\widehat{q}_{\omega^*}$

repeat
  Simulate trajectory τ;
  foreach $t = 0, \dots, T$ **do**
    $\delta_t \leftarrow r_t + \gamma\widehat{q}_\omega(s_{t+1}, a_{t+1}) - \widehat{q}_\omega(s_t, a_t)$; `// TD(0) error`
  $\omega \leftarrow \omega + \beta\sum_{t=0}^{T}\delta_t\nabla_\omega\widehat{q}_\omega(s_t, a_t)$; `// critic update`
  $\theta \leftarrow \theta + \alpha\sum_{t=0}^{T}\widehat{q}_\omega(s_t, a_t)\nabla_\theta \log \pi_\theta(s_t, a_t)$; `// actor update`
until *finished*;

5.2.2 Actor-Critic with baselines: Advantage Actor-Critic (A2C)

As we've seen before, a good baseline to introduce in the policy gradient is the value function – see eq. (5.7). The coefficient before the log-probability gradient in (5.2) becomes an estimate of $Q^\pi(s, a) - V^\pi(s) = \mathbb{E}_{s'}[r(s, a) + \gamma V^\pi(s') \mid s, a]$. We introduce the **advantage** function

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s). \tag{5.11}$$

An unbiased estimate of A^π is the temporal difference error of the value function

$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t).$$

This can be seen by taking the expectation with respect to s_{t+1} conditionally on (s_t, a_t).

Following the spirit of chapter 4 on approximate policy evaluation, this leads to the (bootstrapped) TD error using the parametric approximation $\widehat{v}_\nu$ of the value function:

$$\delta_t = r_t + \gamma\widehat{v}_\nu(s_{t+1}) - \widehat{v}_\nu(s_t) \tag{5.12}$$

just as in chapter 4. This last step can be seen as a supervised regression step, where we fit the (bootstrapped) returns $G_t = r_t + \gamma \widehat{v}_\nu(s_{t+1})$ to the value model predictions $\widehat{v}_\nu(s_t)$, with a mean-squared loss

$$\mathcal{L}(\nu) = \sum_{t=0}^{T} (G_t - \widehat{v}_\nu(s_t))^2 \tag{5.13}$$

Now, we have no need to learn a action-state critic $\widehat{q}_\omega$, since we have TD error estimates that can be expressed only using $\widehat{v}_\nu(s)$ – which is simpler.

The episodic A2C algorithm is as follows:

Algorithm 5.4: Advantage Actor-Critic (A2C), episodic case

Input: Initial policy parameter θ, value parameter ν
Output: Policy π_{θ^*}, value approximation $\widehat{v}_{\nu^*}$

repeat
  Simulate trajectory τ;
  foreach $t = 0, \dots, T$ **do**
    $\delta_t \leftarrow r_t + \gamma \widehat{v}_\nu(s_{t+1}) - \widehat{v}_\nu(s_t)$; `// TD(0) error`
  $\nu \leftarrow \nu + \beta \sum_{t=0}^{T} \delta_t \nabla_\nu \widehat{v}_\nu(s_t)$; `// critic update`
  $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \widehat{v}_\nu(s_t) \nabla_\theta \log \pi_\theta(s_t, a_t)$; `// actor update`
until *finished*;

Remark 8 *To leverage automatic differentiation, appropriate pseudo-losses to define a computational graph for the updates in algorithm 5.4 are*

$$\tilde{C}(\nu) = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T} \delta_t^i \widehat{v}_\nu(s_t^i) \tag{5.14a}$$

$$\tilde{J}(\theta) = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T} \widehat{v}_\nu(s_t^i) \log \pi_\theta(s_t^i, a_t^i) \tag{5.14b}$$

where the temporal-difference estimates δ_t^i must be detached from the graph.

Equivalently, the pseudo-loss $\tilde{C}(\nu)$ in remark 8 can be replaced by the MSE $\mathcal{L}(\nu)$ after detaching G_t.

We create the estimate $G_{t:t+1}$ by only looking at the immediate reward and bootstrapping the rest of the value: this is called **one-step actor-critic**. It has poor sample efficiency: the return estimates ignore most of the (actual) future rewards and end up bootstrapping too much. However, it naturally allows for *online learning*.

This can be easily generalized to using more rewards for bootstrapping the returns: ***n*-step actor-critic** methods look further ahead and use estimates

$$G_{t:t+n} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \widehat{v}_\nu(s_{t+n})$$

Batch actor-critic. In batch algorithms, we switch out episodic simulation with epochs of simulation-and-reset batches, where we simulate trajectories (and reset them at terminal states) until a fixed

number B of updates have been done.

Algorithm 5.5: Batch A2C

Input: Number of epochs N, batch size B

foreach $n = 1, \dots, N$ **do**
  $s_0 \leftarrow$ `Reset()`;
  foreach $i = 0, \dots, B-1$ **do**
    $v_i \leftarrow \widehat{v}_\nu(s_i)$; `// estimate value`
    Draw action $a_i \sim \pi_\theta$;
    Get state, reward (r_i, s_{i+1});
    `// check if we just finished a trajectory`
    if s_{i+1} *is terminal* **then**
      $s_{i+1} \leftarrow$ `Reset()`;
  if s_B *is terminal* **then**
    $v_B \leftarrow 0$;
  else
    $v_B \leftarrow \widehat{v}_\nu(s_B)$;
  Compute bootstrapped advantages, returns $(\delta_i, \widehat{G}_i)$;
  $\nu \leftarrow \nu + \beta \sum_{i=0}^{B-1} \delta_i \nabla_\nu \widehat{v}_\nu(s_i)$;
  $\theta \leftarrow \theta + \alpha \sum_{i=0}^{B-1} \delta_i \nabla_\theta \log \pi_\theta(s_i, a_i)$;

The returns $\widehat{G}_i$ have to be computed taking into account when trajectories end. An efficient algorithm is

$$\widehat{G}_i = \begin{cases} r_i + \gamma \widehat{G}_{i+1} & \text{if } s_i \text{ is not terminal} \\ r_i & \text{otherwise} \end{cases}$$

for $i = 0, \dots, B-2$ and $G_{B-1} = r_{B-1} + \gamma v_B$. The TD(0) errors are $\delta_i = \widehat{G}_i - v_i$.

5.3 Conservative approaches

5.3.1 Surrogate loss, locally monotone optimization

Given a policy π, we want to use the past samples to get to a better policy π'. However, there is no guarantee we can trust the behavior in these samples.

We define the **surrogate loss**

$$L_\pi(\pi') = J(\pi) + \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi'(s,a) A^\pi(s,a) \tag{5.15}$$

The gradient of $L_\pi(\pi_\theta)$ wrt θ is equal to the policy gradient (5.2). L_π is a local approximation of the policy performance J in the neighborhood of π.

We can maximize a lower bound on L_π which is still a local approximation of J using the total variation:

Policy improvement scheme. Given the current policy π_k, we get the next one by solving

$$\max_{\pi'} L_{\pi_k}(\pi') - C\, \mathbb{E}_{s \sim d^{\pi_k}} \left[D_{\mathrm{TV}}(\pi' \| \pi_k)[s] \right]$$

where $C > 0$ is a constant, and we search π' in our search space (for instance, parametric). The maximum we get is positive and satisfies

$$J(\pi') - J(\pi) \geq \max_{\pi'} \left\{ L_{\pi_k}(\pi') - C\, \mathbb{E}_{s \sim d^{\pi_k}} \left[D_{\mathrm{TV}}(\pi' \| \pi_k)[s] \right] \right\} \geq 0$$

The **maximization problem might be too difficult** due to how the total variation distance D_{TV} is defined. We can relax it by lower-bounding again, using Pinkser's inequality to see that

$$D_{\mathrm{TV}}(\pi'\|\pi) \leq \sqrt{2\mathrm{KL}(\pi'\|\pi)}.$$

5.3.2 Natural Policy Gradient (NPG)

The idea is to use the gradient of the "statistical manifold" $\mathcal{M} = \{\pi_\theta : \theta \in \Theta\}$ we search our policies in rather than the parametric gradient (using the coordinates $\Theta \subseteq \mathbb{R}^d$).

This gradient is defined using the Fisher information metric $d_\pi : v \mapsto v^\intercal F(\pi)v$ on the manifold. The gradient of a function h defined on $\mathcal{M}$ at a point π_θ can be expressed in terms of its parametric gradient as

$$\nabla_{\mathcal{M}} h(\pi_\theta) = F(\pi_\theta)^{-1} \nabla_\theta h(\pi_\theta)$$

The update step becomes

$$\begin{aligned} \pi_{k+1} &= \operatorname*{argmax}_{\pi'} L_{\pi_k}(\pi') \\ &\text{s.t. } \overline{\mathrm{KL}}(\pi'\|\pi) = \mathbb{E}_{s\sim d^\pi}[\mathrm{KL}(\pi'\|\pi_k)] \leq \delta \end{aligned}$$

we solve this approximately by

$$\begin{aligned} \theta_{k+1} &= \operatorname*{argmax}_{\theta} g^\intercal(\theta - \theta_k) \\ &\text{s.t.} \frac{1}{2}(\theta - \theta_k)^\intercal F(\pi_k)(\theta - \theta_k) \leq \delta \end{aligned}$$

where $g = \nabla_\theta L_{\pi_k}(\pi_\theta)$ and F is the Fisher information matrix. This is solved by

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^\intercal F(\theta_k)^{-1} g}} F(\theta_k)^{-1} g \tag{5.16}$$

Problem: computing $F(\theta_k)^{-1}$ is too expensive in large dimensions. A solution is to use the **conjugate gradient** to compute $F(\theta_k)^{-1}g$

→ this is called **Truncated Natural Policy Gradient** (TNPG)

Trust Region Policy Optimization (TRPO). We actually enforce the KL constraint which might violated in the NPG updates, and force improvement of the surrogate loss (make it > 0).

→ use **backtracking line search until the constraints** $L_{\pi_k}(\pi') > 0$ **and** $\overline{\mathrm{KL}}(\pi'\|\pi_k) \leq \delta$ **are satisfied**

Chapter 6

The exploration-exploitation dilemma

If we recall the description of Q-learning page 13, we remember that it is important to define good **exploration policy** to take actions off-policy.

For purely greedy approaches, we always choose the next action as the maximum mode of the Q-function $a_{t+1} \in \text{argmax}_{a \in \mathcal{A}} \widehat{Q}(s_t, a)$, but we have no convergence guarantees to the optimum: indeed it can be the case that not all state-action pairs (s, a) are updated.

We can also choose the next action randomly in the action space $\mathcal{A}$, but convergence will be slow because we will not be correctly estimating (exploiting) the action-value function Q under the policy.

6.1 Multi-Armed Bandits

A *multi-armed bandit* is similar to an MDP, except there is only a single state: we *take an action* $a \in \mathcal{A}$ and *get a reward* $r(a)$ distributed according to $\nu(a)$ with $\mu(a) = \mathbb{E}[r(a)]$. The objective is to maximize the expected sum of rewards

$$\mathbb{E}\left[\sum_{t=1}^{T} r_t(a)\right]$$

The **arms** of the bandit are the different possible actions a.

Definition 6 (Regret) *The* regret *of a trajectory is defined by the difference in total rewards between sticking to the best overall constant action and the actions actually taken:*

$$R_n = \max_a \mathbb{E}\left[\sum_{t=1}^{T} r_t(a)\right] - \mathbb{E}\left[\sum_{t=1}^{T} r_t(a_t)\right] \tag{6.1}$$

Introducing the number of times action $a \in \mathcal{A}$ has been taken

$$T_n(a) = \sum_{t=1}^{n} \mathbb{1}_{\{a_t = a\}},$$

we can rewrite the regret as

$$\begin{aligned} R_n &= \max_a n\mu(a) - \sum_{a \in \mathcal{A}} \mathbb{E}[T_n(a)]\mu(a) \\ &= \sum_{a \neq a^*} \mathbb{E}[T_n(a)](\mu(a^*) - \mu(a)) \end{aligned} \tag{6.2}$$

and

$$R_n = \sum_{a \neq a^*} \mathbb{E}[T_n(a)]\Delta(a) \tag{6.3}$$

where $\Delta(a) = \mu(a^*) - \mu(a)$ is the **gap** of arm a.

A good algorithm for bandits has asymptotic regret

$$R_n = o(n)$$

6.1.1 Explore-then-Commit

The idea of this algorithm is to play a fixed number of rounds pulling the arms of the bandit at random, then computing which arm is the best, then observing the rewards we would get by playing the estimated best arm.

Algorithm 6.1: Explore-then-Commit

Input: Number of exploration steps τ, total number of rounds $n > \tau$

// Exploration phase

for $t = 1, \dots, \tau$ **do**

  Take action $a_t \sim \mathcal{U}(\mathcal{A})$; // or round-robin

  Get reward $r_t \sim \nu(a_t)$;

foreach $a \in \mathcal{A}$ **do**

  $\widehat{\mu}_\tau(a) \leftarrow \frac{1}{T_\tau(a)} \sum_{s=1}^{\tau} r_s \mathbb{1}_{\{a_s = a\}}$; // estimate statistics of the $\nu(a)$

// Exploitation phase

for $t = \tau + 1, \dots, n$ **do**

  Take action $\widehat{a}^* = \operatorname{argmax}_a \widehat{\mu}_\tau(a)$; // estimated best action

  Get reward $r_t \sim \nu(a^*)$;

We have the following bound on the regret

Proposition 8 *When running the ETC algorithm with τ exploration steps and n total rounds, we can bound the regret accumulated by*

$$R_n \leq \sum_{a \neq a^*} \left(\frac{\tau}{A} \Delta(a) + 2(n - \tau - 1) \exp(-2\tau \Delta(a)^2) \right)$$

where $A = |\mathcal{A}|$.

This result can be shown using the Chernoff-Hoeffding inequality.

6.1.2 ε-greedy

The idea is to compromise between exploration and exploitation by choosing between taking the best action and taking a random action according to a probability parameter ε_t.

Algorithm 6.2: ε-greedy algorithm

Input: Number of rounds n, sequence of ε_t

for $t = 1, \dots, n$ **do**

  Take action $a_t \sim \mathcal{U}(\mathcal{A})$ with proba. ε_t, or $\operatorname{argmax}_a \widehat{\mu}_t(a)$ otherwise;

  Get reward $r_t \sim \nu(a_t)$;

  // Update statistics

  $T_t(a_t) = T_{t-1}(a_t) + 1$;

  $\widehat{\mu}_t(a_t) = \frac{1}{T_t(a_t)} \sum_{s=1}^{t} r_s \mathbb{1}_{\{a_s = a_t\}}$;

Proposition 9 *If $\varepsilon_t = CA/\Delta_{\min} n$ where $\Delta_{\min} = \min_a \Delta(a)$, then the regret accumulated in the ε-greedy algorithm is bounded by*

$$R_n \leq O\left(\frac{A \log(n)}{\Delta_{\min}}\right) \tag{6.4}$$

6.1.3 Exp3 (Softmax) algorithm

This algorithm is in the same vein and ε-greedy, with a softmax exploration policy.

Algorithm 6.3: Exp3 algorithm

Input: Number of rounds n, initial temperature τ

for $t = 1, \dots, n$ **do**

- Take action $a_t \sim \frac{\exp(\widehat{\mu}_t(a)/\tau)}{\sum_{a'} \exp(\widehat{\mu}_t(a')/\tau)}$;
- Get reward $r_t \sim \nu(a_t)$;
- `// Update statistics`
- $T_t(a_t) = T_{t-1}(a_t) + 1$;
- $\widehat{\mu}_t(a_t) = \frac{1}{T_t(a_t)} \sum_{s=1}^{t} r_s \mathbb{1}_{\{a_s = a_t\}}$;
- Adjust temperature τ;

6.1.4 Lower bounds on the regret

Proposition 10 (Dependent on the problem) *Consider the class of MAB problems with A Bernoulli arms, and consider a bandit algorithm that satisfies $\mathbb{E}[T_n(a)] = o(n^\alpha)$ for any $\alpha > 0$ and any problem.*

For any problem with positive gaps ($\Delta(a) > 0$ for $a \neq a^*$), any algorithm has lower-bounded regret:

$$\liminf_{n \to \infty} \frac{R_n}{\log(n)} = \sum_{a \neq a^*} \frac{\Delta(a)}{\mathrm{kl}(\mu(a), \mu(a^*))} \tag{6.5}$$

- this means no bandit algorithm can have regret smaller than $\Omega(\log n)$
- a fine-tuned ε-greedy algorithm can be optimal

Proposition 11 (Independent on the problem) *For any bandit algorithm and fixed n there's a Bernoulli MAB problem s.t.*

$$R_n = \Omega(\sqrt{An})$$

6.2 Optimism in the face of Uncertainty

6.2.1 Upper confidence bound: UCB

Proposition 12 (Regret bounds) *Consider a Bernoulli MAB with A arms. For $\rho = 1$ and*

Algorithm 6.4: Basic UCB

Input: Number of rounds n

for $t = 1, \dots, n$ **do**

- $B_t(a) = \widehat{\mu}_t(a) + \rho\sqrt{\dfrac{\log(1/\delta_t)}{T_t(a)}}$; `// confidence upper bound`
- Take action $a_t = \text{argmax}_a\, B_t(a)$;
- Get reward $r_t \sim \nu(a_t)$;
- `// Update statistics`
- $T_t(a_t) = T_{t-1}(a_t) + 1$;
- $\widehat{\mu}_t(a) = \frac{1}{T_t(a_t)} \sum_{s=1}^{t} r_s \mathbb{1}_{\{a_s = a_t\}}$;

$\delta_t = 1/t$ and n steps, we have regret

$$R_n = O\left(\sum_{a \neq a^*} \frac{\log(n)}{\Delta(a)}\right)$$

With a 2-action MAB the worst-case (in terms of gaps $\Delta(a)$)

$$R_n = O(\sqrt{n \log(n)})$$

6.2.2 Improvements

UCB-V. We use the Bernstein bounds with an estimate of the standard deviation, $\widehat{\sigma}_t(a_t)$. The confidence upper bound becomes

$$B_t(a) = \widehat{\mu}_t(a) + \sqrt{\frac{2\widehat{\sigma}_t(a_t)\log t}{T_t(a)}} + \frac{8 \log t}{3 T_t(a)}$$

The regret bound becomes

$$R_n \leq O\left(\frac{\sigma^2}{\Delta} \log(n)\right)$$

KL-UCB. We use a tighter KL-divergence bound:

$$B_t(a) = \max\{q \in [0,1] : T_t(a)\text{kl}(\widehat{\mu}_t(a), q) \leq \log t + c \log(\log t)\}$$

which has to be computed as the solution of a convex problem.

6.2.3 LinUCB

We model the rewards as

$$r_t(a) = \mu(a) + \eta_t = \phi(a)^\intercal \theta + \eta_t,$$

with an **arm feature** $\phi(a) \in \mathbb{R}^d$ and parameter $\theta \in \mathbb{R}^d$, and noise $\mathbb{E}[\eta_t] = 0$ (unbiased estimate). As the rewards and actions come in, we will update the best-fit estimate of the parameter vector θ.

Remark 9 *In this setting, since the reward model is unbiased, we can write the regret as*

$$R_n = \mathbb{E}\left[\sum_{t=1}^{n}(\phi(a^*) - \phi(a_t))^\intercal \theta\right] \tag{6.6}$$

The **least-squares** estimate of θ at time t solves

$$\widehat{\theta}_t = \operatorname*{argmin}_\theta \frac{1}{t}\sum_{s=1}^{t}(r_s - \phi(a_s)^\intercal\theta)^2 + \lambda\|\theta\|^2 \tag{6.7}$$

Here, the regularization parameter $\lambda > 0$ might be necessary if the matrix $[\phi(a_s)]_s$ is not full-rank. The closed-form solution is

$$\widehat{\theta}_t = A_t^{-1} b_t \tag{6.8}$$

where

$$A_t := \sum_{s=1}^{t}\phi(a_s)\phi(a_s)^\intercal + \lambda I \text{ and } b_t = \sum_{s=1}^{t}\phi(a_s)r_s$$

The estimate of the value (mean reward) of the action $a \in \mathcal{A}$ is written

$$\widehat{\mu}_t(a) = \phi(a)^\intercal\widehat{\theta}_t \tag{6.9}$$

Proposition 13 (Error bounds for LinUCB) *If η_t is sub-Gaussian (i.e. $\mathbb{P}(|\eta_t| > t) \leq Be^{-vt^2}$ for some constants $B, v > 0$), and the arm features are uniformly bounded:*

$$\|\phi(a)\|_2 \leq L \quad \forall a \in \mathcal{A}$$

then for all a

$$|\widehat{\mu}_t(a) - \mu(a)| \leq \alpha_{t,\delta}\|\phi(a)\|_{A_t^{-1}}$$

with probability $1-\delta$, where we denote

$$\alpha_{t,\delta} = B\sqrt{d\log\left(\frac{1+tL/\lambda}{\delta}\right)} + \sqrt{\lambda}\|\widehat{\theta}_t\| \tag{6.10}$$

and $\|\phi(a)\|_{A_t^{-1}} = \sqrt{\phi(a)^\intercal A_t^{-1}\phi(a)}$ is the A_t^{-1}-norm.

The previous proposition suggests the following confidence upper bound (at level $1-\delta$) for the adapted LinUCB algorithm:

$$B_t(a) = \widehat{\mu}_t(a) + \alpha_{t,\delta}\|\phi(a_t)\|_{A_t^{-1}} \tag{6.11}$$

where $\alpha_{t,\delta}$ is defned as before (6.10).

Algorithm 6.5: LinUCB

for $t = 1, \ldots, n$ **do**
- $B_t(a) = \widehat{\mu}_t(a) + \alpha_t\|\phi(a)\|_{A_t^{-1}}$;
- Take action $a_t \in \operatorname{argmax}_a B_t(a)$;
- Get reward $r_t \sim \phi(a_t)^\intercal\widehat{\theta}_t + \eta_t$;
- $A_{t+1} = A_t + \phi(a_t)\phi(a_t)^\intercal$; `// can use efficient formula to compute inverse`
- $\widehat{\theta}_{t+1} = A_{t+1}^{-1}b_{t+1}$;

Proposition 14 (Regret bound for LinUCB) *If LinUCB is run with $\delta_t = 1/t$ for n steps, the regret is bounded by*

$$R_n = O(d\sqrt{n \log(n)})$$

This bound does not depend on the number of actions, only their feature dimension – this can be useful if A is much greater than the embedding d.

6.3 Bayesian bandits and Thompson Sampling

Up until now, we have made no assumptions on the rewards except for their boundedness. We can exploit information we get about the rewards using a Bayesian setting:

- Use a conjugate prior (initial *belief*) on ν.
- Update the posterior $\nu(a)|(a_1, r_1, \ldots)$ as we pull the arms and get rewards.
- Use the posterior distribution to guide exploration.

For instance, if the rewards are Bernoulli distributed with parameter $\mu(a)$, we can use the conjugate Beta prior

$$\mu(a) \sim \text{Beta}(\alpha_0, \beta_0).$$

An appropriate algorithm would be: as we observe the stream of action-reward, we update the posterior $\mu(a)$ to $\text{Beta}(\alpha_t(a), \beta_t(a))$ with α, β being the counts of the values of $r_t \in \{0, 1\}$:

$$\alpha_t(a) = \alpha_0 + \sum_{s=1}^{t} \mathbb{1}_{\{a_t = a, r_t = 1\}} \tag{6.12a}$$

$$\beta_t(a) = \beta_0 + \sum_{s=1}^{t} \mathbb{1}_{\{a_t = a, r_t = 0\}} \tag{6.12b}$$

Algorithm 6.6: Thompson Sampling (Bernoulli-Beta case)

Input: Number of rounds n, initial prior parameters (α_0, β_0)

for $t = 1, \ldots, n$ **do**
- $B_t(a) \sim \text{Beta}(\alpha_t(a), \beta_t(a))$; `// sample confidence upper bound`
- Take action $a_t = \text{argmax}_a B_t(a)$;
- Get reward $r_t \sim \nu(a_t)$;
- `// Update the prior statistics`
- $\alpha_t(a_t) = \alpha_{t-1}(a_t) + 1 - r_t$;
- $\beta_t(a_t) = \beta_{t-1}(a_t) + r_t$;

Proposition 15 (Regret of Thompson sampling) *If TS is run with n steps and $\delta_t = 1/t$, then the regret is bounded by*

$$R_n = O\left((1+\varepsilon) \sum_{a \neq a^*} \frac{\Delta(a) \log(n)}{\text{kl}(\mu(a), \mu(a^*))}\right) \tag{6.13}$$

This matches the lower bound of proposition 10.

Bibliography

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.