Université Paris-Saclay
Institut Polytechnique de Paris
M2 MN
—

Year 2024-2025

Lab 2
—

Integrated Lab session
Learning and Control for Multimedia
—
Lab 2 – *Value Iteration - Policy Iteration*

Warning : This lab must be prepared by all students before the start of the session. The preparation includes reading the whole topic and understanding it. You should also think about the manipulation part and plan the tests you will perform to illustrate the lab.

This practical course requires a working Python environment, such as
— WinPython (recommanded), see `https://winpython.github.io`
— Anaconda (recommanded), see `https://www.anaconda.com`
— Pyzo, see `https://pyzo.org`
— Google Collaborate, see `https://colab.research.google.com`

1 Introduction

Gridworlds such as the one shown in Figure 1 are simple model problems that allow to test different reinforcement learning techniques. In general, each box of a gridworld represents a possible value of the state of the environment. An episode consists of a succession of actions that the agent must choose. The arrival state S_{t+1} and the reward R_{t+1} at time $t+1$, can depend randomly on the departure state S_t and the action A_t chosen at time t.

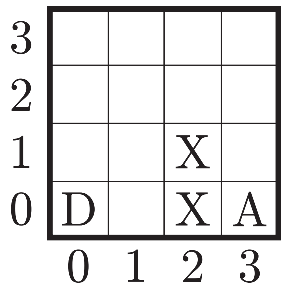


FIGURE 1 – Gridworld : a robot starts from D and has to move to A while avoiding the dangerous areas X

2 Notations

We consider the *gridworlds* represented in figure 1. A robot starts from square D and has to reach square A while avoiding the dangerous areas marked with an X. For this, at each time t, it has the choice between 4 possible actions belonging to the set $\mathcal{A} = \{\mathrm{R}, \mathrm{L}, \mathrm{U}, \mathrm{D}\}$ corresponding to a move to the right (R), to the left (L), up (U), or down (D).

A move that would lead the robot
— outside the grid generates a reward $r = -1$ and leaves the robot in place ;
— to a dangerous square X generates a reward $r = -10$;
— at the arrival square A generates a reward $r = 0$ and leads to the end of the episode ;
— at a normal square generates a reward $r = -0.1$.

The aim is to find a policy $\pi(a|s)$ maximizing the return of the agent.

Preparation 1

1. *How to define the state of the environment ?*
2. *What is the set of values $\mathcal{S}$ that the state can take ?*
3. *Recall the Bellman equation satisfied by the state value function $v_\pi(s)$ for a given policy $\pi(a|s)$.*
4. *Give the Bellman equation satisfied by $v_\pi(s)$ with $s = (2, 2)$ assuming that the policy $\pi(a|s)$ consists in choosing an action with a uniform distribution for all $s \in \mathcal{S}$.*

Appendix A describes a set of functions simulating the considered *gridworld* environment. The first lines

```
# Set up the environment
num_rows = 4
num_cols = 4

states = [(i,j) for i in range(num_cols) for j in range(num_rows)]
initial_state = (0,0)
final_state = (3,0)
holes = [(2,0),(2,1)]

actions = [(1, 0), (−1, 0), (0, 1), (0, −1)] # Right, Left, Up, Down
num_actions = len(actions)
```

allow to define the possible values of the state and the actions, as well as the start and finish boxes as well as the dangerous boxes.

The function

```
def reward(s_t,a_t)
```

allows to evaluate the reward by starting from s_t and applying a_t.

The function

```
def transition_prob(s_t,a_t,s_t1)
```

provides the transition probability $p(s_{t+1}|s_t, a_t)$.

The function

```
def policy_eval(policy,gamma,theta=1e−3)
```

is used to evaluate a deterministic policy. The parameter γ is the discount rate. The parameter θ determines the stopping condition of the iterations of the policy evaluation algorithm.

The function

```
def policy_iteration(policy,gamma,theta=1e−3):
```

allows to perform an optimization of the policy by iteration on the policy. The variable `policy` represents the initial policy. The parameter γ is the discount rate. The parameter θ determines the stopping condition of the iterations of the evaluation algorithm of the policy.

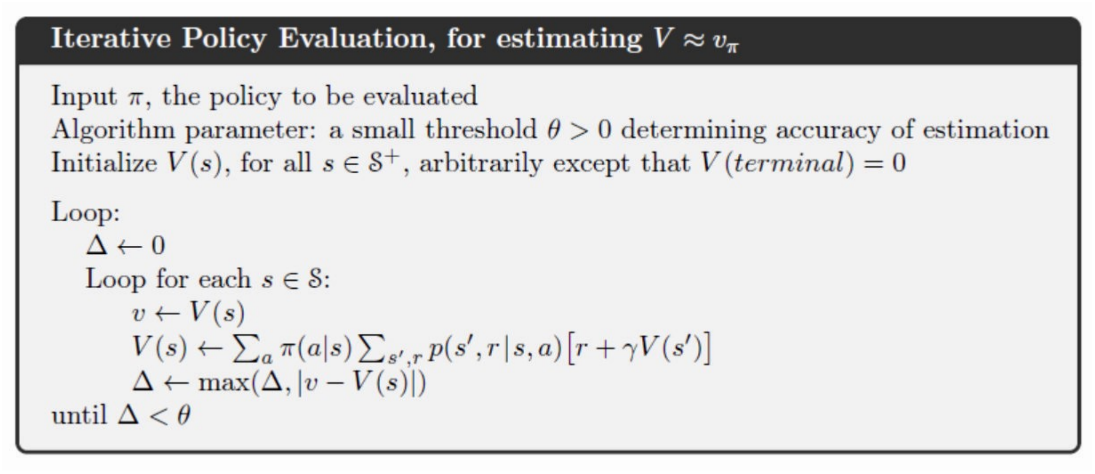
The function

```
def display_value_policy(V,policy):
```

displays the value function v_π of each state and the deterministic policy π.

3 Policy evaluation

We will first consider a deterministic policy where in each state an action is chosen randomly with a uniform distribution. Such a policy can be obtained as follows



```
Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
    $\Delta \leftarrow 0$
    Loop for each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
```

FIGURE 2 – Iterative evaluation of the state value function associated to a given policy (from Sutton et Barto, *Reinforcement Learning, an Introduction*)

policy = [random.choice(actions) **for** state **in** states]

We will then iteratively solve the Bellman equation associated with the state value function. To do this, we will implement the iterative algorithm described in Figure 2.

Manipulation 1

1. *What is the difference between a deterministic policy and a random policy ?*
2. *Using the function* `policy_eval(policy,gamma)`*, calculate the state value function associated with the policy generated earlier. Take* $\gamma = 0.9$.
3. *Evaluate the number of iterations necessary for the convergence of the policy evaluation algorithm.*
4. *Visualize the state value function and the associated policy.*
5. *Repeat the previous questions with* $\gamma = 0.99$ *and* $\gamma = 0.999$. *Interpret the result.*

4 Policy iteration

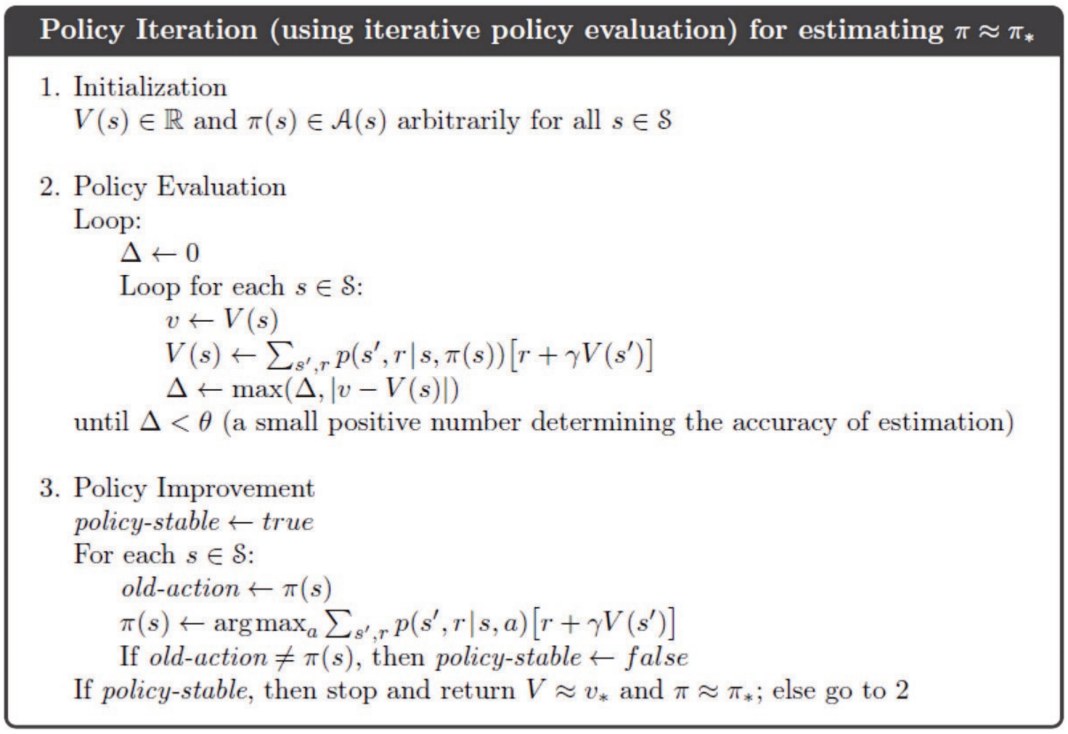
The policy iteration algorithm allows, starting from a given policy π, to compute the state value function v_π, then to compute the optimal policy for this state value function, then to recompute the state value function for the optimized policy... until reaching a stable policy. Figure 3 shows the policy iteration algorithm in the most general case.

Manipulation 2

1. *Using the* `policy_iteration(policy,gamma)`*, evaluate an optimal policyfor the considered problem. Choose* $\gamma = 0.9$.
2. *Evaluate the number of iterations on the policy required before obtaining a stable policy.*
3. *Visualise the state value function and the associated optimal policy.*
4. *Resume the exercise by changing the location of the dangerous boxes or by changing the size of the grid.*

5 Value function iteration

The policy iteration algorithm requires an iterative evaluation of the state value function at each new state update. The idea of the value iteration algorithm is to do only one iteration of updating the value with each new policy optimisation. The value iteration algorithm is described in Figure 4.



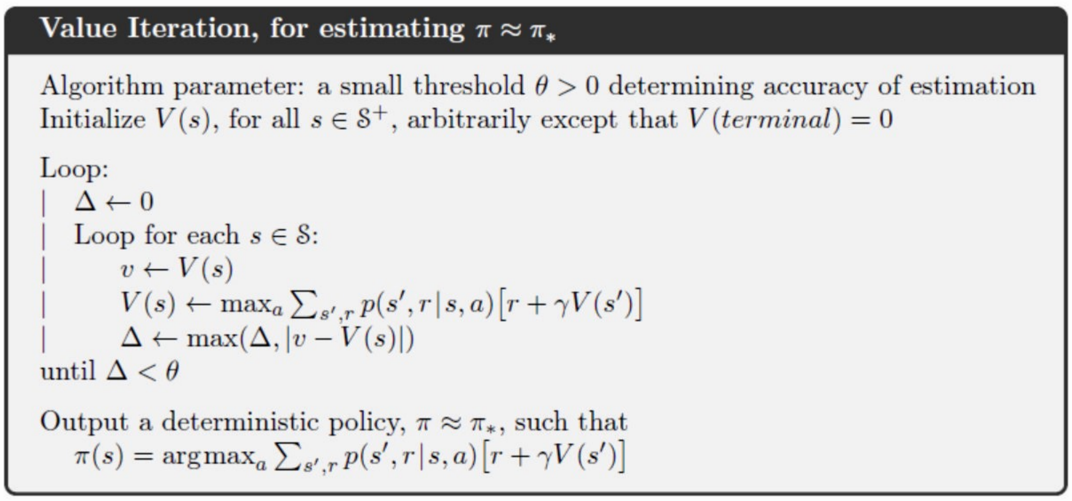
```
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $old\text{-}action \leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
       If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

FIGURE 3 – Policy iteration algorithm (taken from Sutton and Barto, *Reinforcement Learning, an Introduction*)



```
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
```

FIGURE 4 – Value iteration algorithm (taken from Sutton and Barto, *Reinforcement Learning, an Introduction*)

Manipulation 3

1. *Using the function* `policy_iteration(policy,gamma)` *as an example, write a function implementing a value iteration algorithm and providing the associated optimal policy.*
2. *Evaluate the number of iterations required to obtain an optimal policy.*
3. *Visualise the state-value function and the associated optimal policy.*
4. *Compare the performance of the policy iteration algorithm and the value iteration algorithm.*

A Gridworld

```
import random
import matplotlib.pyplot as plt

# Set up the initial environment
num_rows = 4
num_cols = 4

actions = [(1, 0), (−1, 0), (0, 1), (0, −1)] # Right, Left, Up, Down
num_actions = len(actions)

states = [(i,j) for i in range(num_cols) for j in range(num_rows)]
initial_state = (0,0)
final_state = (3,0)
holes = [(2,0),(2,1)]

# Reward evaluation function
def reward(old_state,action):
    if old_state == final_state:
        return 0

    # Apply action
    candidate_state = (old_state[0]+action[0],old_state[1]+action[1])

    # checks whether action is valid
    if candidate_state in states:
        if candidate_state in holes:
            return −10
        else:
            return −0.1
    else:
        return −1


# Transition probability evaluation
def transition_prob(old_state,action,new_state):
    # Case of final state
    if old_state == final_state:
        if new_state == final_state:
            return 1
        else:
            return 0
    # Apply action
    candidate_state = (old_state[0]+action[0],old_state[1]+action[1])

    # checks whether action is valid
    if candidate_state in states:
        if candidate_state == new_state:
            return 1
    else:
        if old_state == new_state:
            return 1
    return 0


# Policy evaluation algorithm for deterministic policy
def policy_eval(policy,gamma,theta=1e−3):
    """
    Inputs : policy : policy to evaluate
             gamma : dicount factor
             theta : accuracy criterion
    """
    # Initialization
    V = [0 for state in states]
    Delta = float('inf')

    while Delta > theta:
        Delta = 0

        for s in range(0,len(states)):
            # Back−up of V[s]
            v = V[s]

            # Initialization of the value fonction
            V[s] = 0

            # Policy evaluation with deterministic policy
            for sp in range(0,len(states)):
                V[s] += transition_prob(states[s],policy[s],states[sp])\
                    *(reward(states[s],policy[s]) + gamma * V[sp])

            Delta = max(Delta,abs(v−V[s]))

    return V


# Policy iteration algorithm
def policy_iteration(policy,gamma,theta=1e−3):
    """
    Inputs : policy : policy to evaluate
             gamma : dicount factor
             theta : accuracy criterion
    """
    policy_stable = False

    # Loop until policy is stable
    while policy_stable == False:
        policy_stable = True

        # Policy evaluation
        V = policy_eval(policy,gamma,theta)

        # Policy improvement step
        for s in range(0,len(states)):
            old_action = policy[s]
            q = []
            for action in actions:
                tmp = 0
                for sp in range(0,len(states)):
                    tmp += transition_prob(states[s],action,states[sp])\
                        *(reward(states[s],policy[s]) + gamma * V[sp])
                q.append(tmp)

            idx = q.index(max(q))
            policy[s] = actions[idx]

            if old_action != policy[s]:
                policy_stable = False

    return policy


# Displays the Value function
def display_value_policy(V,policy):

    plt.xlim([-0.5, num_cols-0.5])
    plt.ylim([-0.5, num_rows-0.5])

    for s in range(0,len(states)):
        state = states[s]
        pi = policy[s]
        plt.text(state[0], state[1], "{:10.3f}".format(V[s]), size=12,
                 ha="center", va="center")

        plt.arrow(state[0], state[1], pi[0]/3, pi[1]/3, head_width=0.05, head_length=0.1, fc='r', ec='r')

    plt.show()
```