OREISTEIN_Pierre_TP1

November 12, 2018

1 0 - Information

```
In [76]: author= "Pierre OREISTEIN"
         date= "11/11/2018"
```

2 1 - Dynamic Programming

2.1 1.1 - Packages

```
In [28]: %matplotlib inline

         # Maths packages
         import numpy

         # Graphic packages
         import matplotlib.pyplot as plt
         import seaborn as sns
         sns.set()
```

2.2 1.2 - Question 1:

First, it exists paths between each state with high probability (>= 0.42) and γ is quite high (0.95). So, in this case, the optimal policy is going to look for the most important reward.

If we look on the possible rewards, there are only three couple (state, action) with non null reward:

- (s_0, a_2): 5/100
- (s_2, a_2): 9/10
- (s_2, a_1): 1

The first couple has a too low reward for being a possible solution. However, the last cannot be also the solution. In fact, if the agent plays a_1 in s_2, it has only 40% of chances to go back to the state s_2 and 60% to go to the state s_1 which will provoke a turn without reward. At the opposite, if it plays a_2, it is sure to go back in state s_2 with 100% chances. Hence, it seems that the agent will try to go as fast as possible to the state s_2 and then play always a_1.

To conclude, we can guess that the optimal policy will be $\pi = [1, 1, 2]$

2.2.1 1.2.1 - Defintion of the transition matrices

```
In [31]: # Transition matrix for action a0
         P_0 = np.array([[0.55, 0.45,   0],
                         [   1,    0,   0],
                         [   0,    1,   0]])

         # Transition matrix for action a1
         P_1 = np.array([[ 0.3,  0.7,   0],
                         [   0,  0.4, 0.6],
                         [   0,  0.6, 0.4]])

         # Transition matrix for action a2
         P_2 = np.array([[   1,   0,   0],
                         [   0,   1,   0],
                         [   0,   0,   1]])

         # Total matrix of transition
         P = np.array([P_0, P_1, P_2])
```

2.2.2 1.2.2 - Definition of the rewards

```
In [32]: # Matrix of reward
         r = np.array([[0, 0, 5/100],
                       [0, 0,     0],
                       [0, 1,  9/10]])
```

2.2.3 1.2.3 - Definition of the differente states

```
In [33]: # Array of states
         X = np.array([0, 1, 2])
```

2.2.4 1.2.4 - Definition of the state of Actions

```
In [34]: # Array of actions
         A = np.array([0, 1, 2])
```

2.2.5 1.2.5 - Definition of the actualisation rate

```
In [35]: # Actualisation rate gamma
         gamma = 0.95
```

2.3 1.3 - Question 2: Computation of an approximation of the optimal policy

2.3.1 1.3.1: Value Iteration Algorithm

```
In [36]: def computeEpsilon(criterion, gamma):
             """Compute epsilon associated to the criterion and gamma."""
```

```
    # Computation of epsilon
    epsilon = (1 - gamma) / (2 * gamma) * criterion

    return epsilon
```

In [37]:
```
def optimalBellmanOperator1State(V, x, gamma):
    """Apply the optimal Bellman operator to V(x)."""

    # Initialisation of the max
    maxi = -float('Inf')

    # Loop over each possible actions for finding the max
    for a in A:

        # Extract the probabilities
        p = P[a][x, :].reshape((-1,1))

        # Compute the value function for the actions a
        value = r[x, a] + gamma * np.dot(p.T, V)[0,0]

        # Update maxi
        if value > maxi:
            maxi = value

    # Return the new value
    return maxi
```

In [38]:
```
def optimalBellmanOperator(V, gamma):
    """Apply the Bellman operator to V."""

    # Parameters
    n, d = np.shape(V)

    # Initialisation of the matrix of result
    V_result = np.zeros((n, 1))

    # Loop over each state
    for x in X:

        # Apply the optimal Bellman operator to V(x)
        V_result[x] = optimalBellmanOperator1State(V, x, gamma)

    # Return the result
    return V_result
```

In [39]:
```
def infNorm(V):
    """Compute the inf norm on V."""

    return np.max(np.abs(V))
```

```
In [40]: def computeGreedyPolicy(V_k, gamma):
             """Computation of the optimal policy."""

             # Parameters
             n, d = np.shape((V_k))

             # Initialisation of the greedy policy
             pi = np.zeros((n, 1), dtype=int)

             # Loop over each state
             for x in X:

                 maximum = -float('Inf')
                 maximising_action = 0

                 # Loop over each action
                 for a in A:

                     # Extract the probabilities
                     p = P[a][x, :].reshape((-1,1))

                     # Compute the value function for the actions a
                     value = r[x, a] + gamma * np.dot(p.T, V_k)[0,0]

                     # Update maximum and maximising_action
                     if value > maximum:
                         maximum = value
                         maximising_action = a

                 # Instantiate pi
                 pi[x] = maximising_action

             # Return pi
             return pi

In [57]: def valueIteration(gamma, criterion=0.01):
             """This function runs a value iteration on the MDP defines before."""

             # Initilazation of the value functions
             V_k = np.zeros((3, 1))
             V_k_1 = V_k - 1

             # Saving array of the gap between V_k and V_k_1 for the final plot
             gaps = []

             # Computation of the criterion of the while loop
             epsilon = computeEpsilon(criterion, gamma)
```

```
        # Iteration
        while infNorm(V_k - V_k_1) > epsilon:

            # Update V_k_1
            V_k_1 = V_k

            # Apply the optimal Bellman operator
            V_k = optimalBellmanOperator(V_k, gamma)

            # Append V_k to gaps
            gaps.append(infNorm(V_k - V_k_1))

        # Compute the greedy policy pi_k
        pi_k = computeGreedyPolicy(V_k, gamma)

        # Compute the greedy policy
        print("The optimal policy is: ")
        for x in X:
            print("In state s_{} take the action: a_{}".format(x, int(pi_k[x][0])))

        # Return the greedy policy
        return V_k, pi_k, gaps
```

```
In [58]: # Compute the value iterations for the policy and the value functions
         V_k, pi_k, gaps = valueIteration(gamma);
```

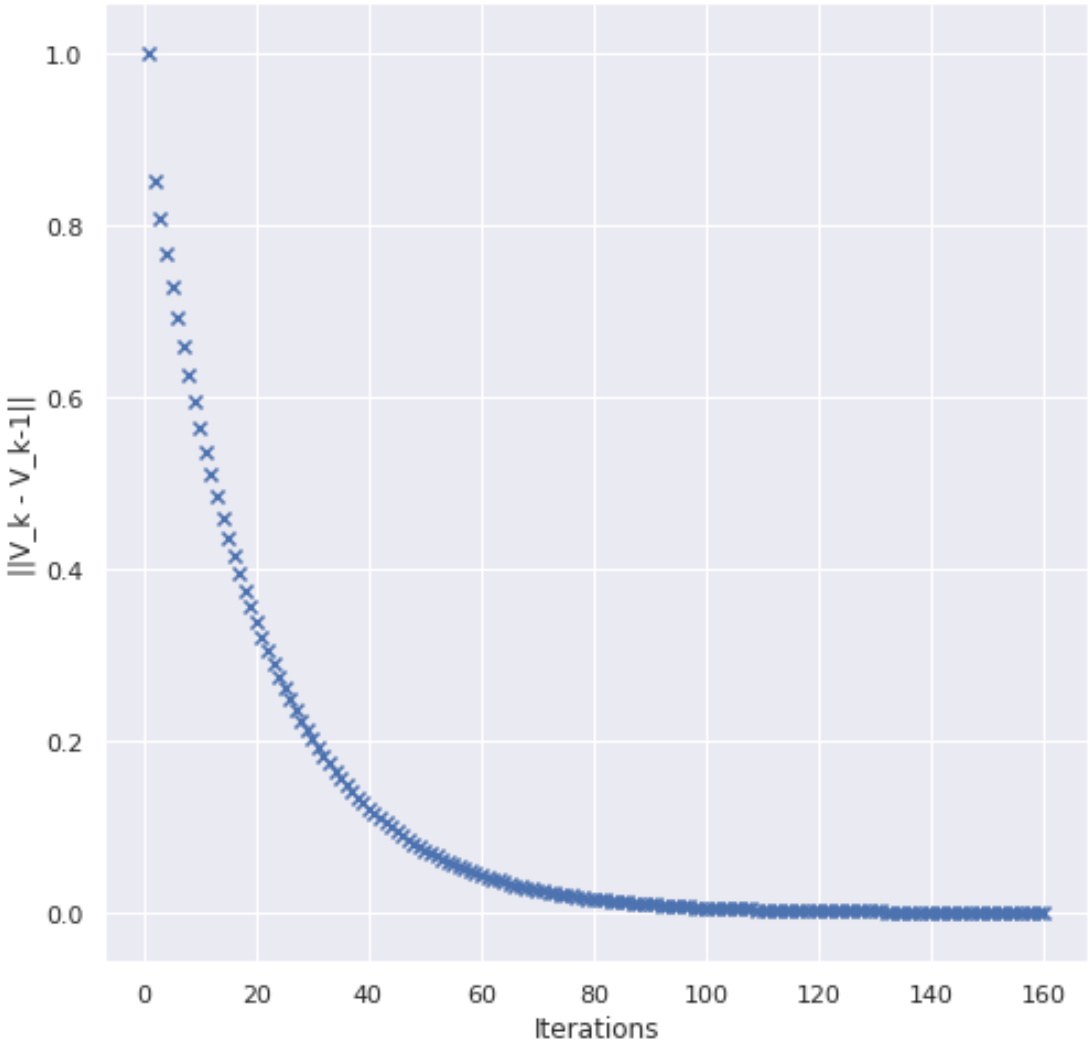
```
The optimal policy is:
In state s_0 take the action: a_1
In state s_1 take the action: a_1
In state s_2 take the action: a_2
```

2.3.2 1.3.2 - Plot of the gap over the iteration

```
In [44]: def plotGaps(gaps):
             """Plot the gap over the iteration of the algorithm of value functions."""

             # Array of the iteration
             iterations = [i for i in range(1, len(gaps) + 1)]

             # Parameters of the figure
             plt.figure(figsize=(8,8))
             plt.grid(True)

             # Plot
             plt.scatter(iterations, gaps, label="||V_k - V_k-1||", marker="x")

             # Legend
```

```
    plt.xlabel("Iterations")
    plt.ylabel("||V_k - V_k-1||")

    # Save the plot
    plt.savefig("./Images/Gaps", bbox_inches='tight', pad_inches=0.0)

    # Display
    plt.show()
```

In [45]:
```
plotGaps(gaps)
```



2.3.3 1.3.3 - Policy evaluation

It exists different way for computing the evaluation of our greedy policy. However, here the MDP is small in dimension and size, so we are going to proceed to a direct computation.

In [46]:
```
def probabilitiesPi(pi):
    """Compute the corresponding transition matrix according to the policy pi."""

    # Create P_pi, the probability matrix according to the policy pi
    P_pi = []

    # Loop over each state
    for x in X:

        # Action taken according to the policy
        a_pi = pi[x, 0]

        # Probabilities for that action and state
        p_a_x = P[a_pi][x, :]

        # Update P_pi
        P_pi.append(p_a_x)

    # Convert P_pi as an array
    P_pi = np.array((P_pi))

    return P_pi
```

In [47]:
```
def rewardsPi(pi):
    """Compute the corresponding rewards vector according to the policy pi."""

    # Create P_pi, the probability matrix according to the policy pi
    R_pi = []

    # Loop over each state
    for x in X:

        # Action taken according to the policy
        a_pi = pi[x, 0]

        # Reward for that action and state
        r_a_x = r[x, a_pi]

        # Update P_pi
        R_pi.append(r_a_x)

    # Convert R_pi as an array
    R_pi = np.array((R_pi)).reshape((-1,1))

    return R_pi
```

In [48]:
```
def policyEvaluation(pi, gamma):
    """Compute the evaluation of the policy pi by a direct computation."""
```

```
            # Compute the corresponding transition matrix according to pi
            P_pi = probabilitiesPi(pi)

            # Compute the corresponding rewards vector according to pi
            R_pi = rewardsPi(pi)

            # Shape of the matrices
            n, d = np.shape(P_pi)

            # Evaluation of the policy pi
            I = np.eye(n)
            V_pi = np.dot(np.linalg.inv(I - gamma * P_pi), R_pi)

            return V_pi
```

```
In [49]: # Computation of the policy evaluation
         V_star = policyEvaluation(pi_k, gamma)

         # Display it
         V_star
```

```
Out[49]: array([[15.39115723],
                [16.5483871 ],
                [18.        ]])
```

```
In [50]: # Display the difference with the one computed thanks to the Value Iterations Algorit
         infNorm(V_k - V_star)
```

```
Out[50]: 0.004880925314047602
```

The criterion of 0.01 is well respected

2.4 1.4 - Question 3: Policy Iterations

2.4.1 1.4.1 - Definition of pi_0

```
In [78]: # Definition of pi_0
         pi_0 = np.array([[1],
                          [1],
                          [2]])
```

2.4.2 1.4.2 - Policy Iterations algorithm

```
In [79]: def policyIterations(pi_0, gamma):
             """Executes the policy iterations on the MDP defined before."""

             # Initialisation of the policy
             pi_k = pi_0
```

```
        # First policy evaluation
        V_pi_k = policyEvaluation(pi_k, gamma)
        V_pi_k_1 = V_pi_k - 1

        # Counter of iterations
        nb_iter = 1

        # Loop until convergence
        while infNorm(V_pi_k - V_pi_k_1) != 0:

            # Policy improvement
            pi_k = computeGreedyPolicy(V_pi_k, gamma)

            # Update V_pi_k_1
            V_pi_k_1 = V_pi_k

            # Policy Evaluation
            V_pi_k = policyEvaluation(pi_k, gamma)

            # Increase the counter of iterations
            nb_iter += 1

        return V_pi_k, pi_k, nb_iter
```

In [87]:
```
# Definition of a random pi_0
# pi_0 = np.random.randint(0,3, 3).reshape((-1,1))

# Compute the policy iterations
V_pi_k, pi_k_PI, nb_iter = policyIterations(pi_0, gamma)

# Display the different results
print("The optimal value functions is: \n", V_pi_k)
print("The optimal policy is: \n", pi_k_PI)
print("The number of iterations is: ", nb_iter)

# Difference between the optimal value function and the one computed
print("Difference with the optimal value functions V*: ", infNorm(V_pi_k - V_star))
```

```
The optimal value functions is:
 [[15.39115723]
 [16.5483871 ]
 [18.        ]]
The optimal policy is:
 [[1]
 [1]
 [2]]
The number of iterations is:  2
```

```
Difference with the optimal value functions V*:  0.0
```

Here, about the speed of convergence per iteration, the algorithm of policy Iterations (PI) is much faster than the Value Functions algorithm (VF). In fact, it needs only 2 iterations to find the exact optimal policy and value functions. At the opposite, the VF algorithm requires 160 iterations to converge towards an approximation of the optimal policy and value functions with an error of 0.01.

However, the iterations of VF algorithm are much faster than the PI algorithm ones. Moreover, here the MPD has a very small dimension and size. In other cases, the policy evaluation step can be very costly and slow. In particurlar, if the dimensions are too important, MC simultaions or Iterative policy evaluation are needed which leads also to a very costly and slow algorithm.

Finally, we also choose the optimal policy as initialisation for the Pi algorithm which helps to converge faster. However, in general, we initialise with random policy which leads to a slower algorithm.

3 2 - A review of Rl Agent/Environment Interaction

3.1 2.1 - Packages

```
In [94]: from gridworld import GridWorld1
         import gridrender as gui
         import time

         env = GridWorld1
```

3.2 2.2 - Question 4

3.2.1 2.2.1 - Definition of the deterministic policy

```
In [120]: def deterministicPolicy():
              """Compute the deterministic policy specified."""

              pol = []
              for i, actions in enumerate(env.state_actions):

                  # Explore all the actions
                  nb_actions = len(actions)
                  i = 0
                  while i < nb_actions:
                      if 'right' == env.action_names[actions[i]]:
                          pol.append(actions[i])
                          i = nb_actions + 1
                      i += 1
                  if i == nb_actions:
                      pol.append(3)  # Corresponding to action "up"

              return pol
```

```
In [121]: # Compute the deterministic policy
          pol = deterministicPolicy()

          # Display the policy
          gui.render_policy(env, pol)
```

3.2.2 2.2.2 - MC Simulations

```
In [122]: # Here the v-function and q-function to be used for question 4
          v_q4 = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.67106071, -0.99447514, 0.0
                  -0.93358351, -0.99447514]
```

```
In [123]: def estimatorMCValueFunctions(pi, n=1000, gamma=0.95, delta=0.001):
              """Estimate the value functions thanks to a MC estimation."""

              # Definition of Tmax
              Tmax = -np.log(delta / 2) / (1 - gamma)

              # Saving array of all the trajectories starting with s
              tau = [[] for i in range(env.n_states)]

              # Play the different trajectories
              for e in range(n):

                  # Initialisation of the first state, the time and the terminal flag
                  state = int(env.reset())
                  t = 1
                  term_flag = False

                  # Saving array of all the rewards and and save the first state
                  rewards = []
                  state_init = state

                  while (t < Tmax and not(term_flag)):

                      # Select the action to do according to the policy
                      action = pol[state]

                      # Let act the environment
                      state, reward, term_flag = env.step(state, action)

                      # TO UNCOMMENT IF WE DO NO WANT TO CONSIDER THE FINAL STATES
                      # term_flag=False

                      # Add the reward to the saving array
                      rewards.append(reward)

                      # Increase the timer
```

```
                t += 1

            # Update tau
            tau[state_init].append(rewards)

        # Compute the estimation of the value functions
        V_pi = np.zeros((env.n_states, 1))

        for state in range(env.n_states):

            # Extract N(s)
            N_s = len(tau[state])

            # Compute the mean over all the trajectories begin with state
            for trajectory in tau[state]:

                # Pounded mean of this trajectory
                mean = 0
                for t in range(len(trajectory)):
                    mean += gamma**t * trajectory[t]

                # Update of V_pi[state]
                V_pi[state, 0] += mean / N_s

        return V_pi
```

In [124]:
```
# Compute the estimation of V_pi
V_pi = estimatorMCValueFunctions(pol, n=1000)

# Display the estimation
print("V_pi: \n", V_pi)
```

```
V_pi:
 [[ 0.87966645]
 [ 0.93413748]
 [ 0.98934271]
 [ 0.        ]
 [ 0.60311158]
 [-0.99530405]
 [ 0.        ]
 [-0.83411422]
 [-0.88157689]
 [-0.93279906]
 [-0.99373563]]
```

3.2.3 2.2.3 - Plotting of $J_n - J^\pi$

```
In [125]: def estimationOfMu0(nb_samples=10000):
              """Compute an MC estimation of mu_0 with nb_samples samples."""

              # Initialisation of mu_0
              mu_0 = np.zeros((env.n_states, 1))

              # Compute the estimation
              for sample in range(nb_samples):

                  # Sample of mu_0
                  state = int(env.reset())

                  # Increment mu_0
                  mu_0[state] += 1

              # Normalise mu_0
              mu_0 = mu_0 / mu_0.sum()

              return mu_0

In [126]: def computeJn(mu_0, Vn):
              """Compute Jn according to mu_0 and Vn."""

              Jn = np.dot(mu_0.T, Vn)[0, 0]

              return Jn

In [127]: def computeJpi(mu_0, V_pi=v_q4):
              """Compute Jn according to mu_0 and Jn."""

              # Reshape V_pi
              V_pi = np.array(V_pi).reshape((-1, 1))

              J_pi = np.dot(mu_0.T, V_pi)[0, 0]

              return J_pi

In [128]: def displayJGap(nb_n=200):
              """Display the gap between Jn and J_pi according to n."""

              # Array of the iteration
              n_l = np.linspace(1, 5000, num=nb_n, dtype=int)

              # Computation of the V_n
              V_n_l = [estimatorMCValueFunctions(pol, n=n) for n in n_l]

              # Computation of mu_0
```

```
    mu_0 = estimationOfMu0()

    # Computation of J_n and J_pi
    J_n_l = [computeJn(mu_0, V_n_l[i]) for i in range(nb_n)]
    J_pi = computeJpi(mu_0)

    # Computation of the gap between them
    gaps = [J_n_l[i] - J_pi for i in range(nb_n)]

    # Parameters of the figure
    plt.figure(figsize=(8, 8))
    plt.grid(True)

    # Plot
    plt.scatter(n_l, gaps, label="J_n - J_pi", marker="x")

    # Legend
    plt.xlabel("n")
    plt.ylabel("|J_n - J_pi|")

    # Save the plot
    plt.savefig("./Images/Gaps_J_n_J_pi", bbox_inches='tight', pad_inches=0.0)

    # Display
    plt.show()
```

```
In [129]: displayJGap()
```

We can observe that the error with the optimal value function is decreasing and converge to zero. It is the strengh of the MC estimation offline: the estimation of the optimal value function is not biased. However, we can observe its big variance also. In fact, the variance after 5000 steps is not really lower than the one after 2000 iterations.

3.3 2.3 - Question 5: Q-learning

- For the exploration policy we choose a policy which make a tradeoff between a random policy which helps to explore the different states and an exploitation policy which helps to improve our rewards thanks to the already learnt value functions. It also allows to visit an infinite number of times every state.
- For the Q-learning step, we are going to choose learning rates equal to:

$$\alpha(x_t, a_t) = \frac{1}{number\_of\_visits(x_t, a_t)} \tag{1}$$

- These two choices allow us to verify the conditions of Robbins and Monroe of 1951 and so to assure the convergence of Q towards the optimal one Q^*.

3.3.1 2.3.1 - Greedy Exploration Policy

```
In [130]: def greedyExplorationPolicy(x, Q, epsilon):
              """Return an action for the state x with the current approximation Q according
                 to an epsilon greedy policy."""

              # Take a sample for deciding if we return a random action or not
              sample = np.random.rand()

              # Random action
              if sample < epsilon:
                  action = np.random.choice(env.state_actions[x])

              # Take the maximising action
              else:
                  # Possible actions for the given state
                  possible_actions = env.state_actions[x]

                  # Compute the maximising action among the possible states
                  maximum = Q[x, possible_actions[0]]
                  action = possible_actions[0]
                  for i in range(1, len(possible_actions)):

                          value = Q[x, possible_actions[1]]

                          if value > maximum:
                                  maximum = value
                                  action = possible_actions[i]

              return action
```

3.3.2 2.3.2 - Q-Learnin Algorithm

```
In [163]: def qLearning(n=1000, epsilon=0.2, gamma=0.95, delta=0.001):
              """Compute the coresponding Q matrix and the opitmal policy online."""

              # Definition of Tmax
              Tmax = -np.log(delta / 1) / (1 - gamma)

              # Parameters
              nb_actions = len(env.action_names)

              # Initialisation of the matrices of Q and alpha
              Q = np.zeros((env.n_states, nb_actions))
              alpha = np.zeros((env.n_states, nb_actions)) + 1
```

```
    # Rewards cumulated
    rewards = 0

    # Play the different trajectories
    for e in range(n):

        # Initialisation of the first state, the time and the terminal flag
        x_t = int(env.reset())
        t = 1
        term_flag = False

        # Initialisation of the reward
        reward = 0

        while (t < Tmax and not(term_flag)):

            # Select the action to do according to the policy
            a_t = greedyExplorationPolicy(x_t, Q, epsilon)

            # Observation of the next state
            x_t_plus_1, r_t, term_flag = env.step(x_t, a_t)

            # Temporal differences
            delta_t = r_t + gamma * Q[x_t_plus_1, :].max()

            # Extract alpha
            alpha_t = alpha[x_t, a_t]

            # Update of Q
            Q[x_t, a_t] = (1 - alpha_t) * Q[x_t, a_t] + alpha_t * delta_t

            # Update of alpha
            alpha[x_t, a_t] = 1 / ((1 / alpha[x_t, a_t]) + 1)

            # Update of the state
            x_t = x_t_plus_1

            # Update of the reward
            reward += gamma ** (t - 1) * r_t

            # Increase the timer
            t += 1

        # Cumulated rerward over all the episodes
        rewards += reward

    return Q, rewards
```

3.3.3 2.3.3 - Definition of v_opt

```
In [222]: v_opt = [0.87691855, 0.92820033, 0.98817903, 0.00000000, 0.82369294, 0.92820033, 0.0
                   0.87691855, 0.82847001]

          # Reshape v_opt
          v_opt = np.array(v_opt).reshape((-1, 1))
```

3.3.4 2.3.4 - Plotting

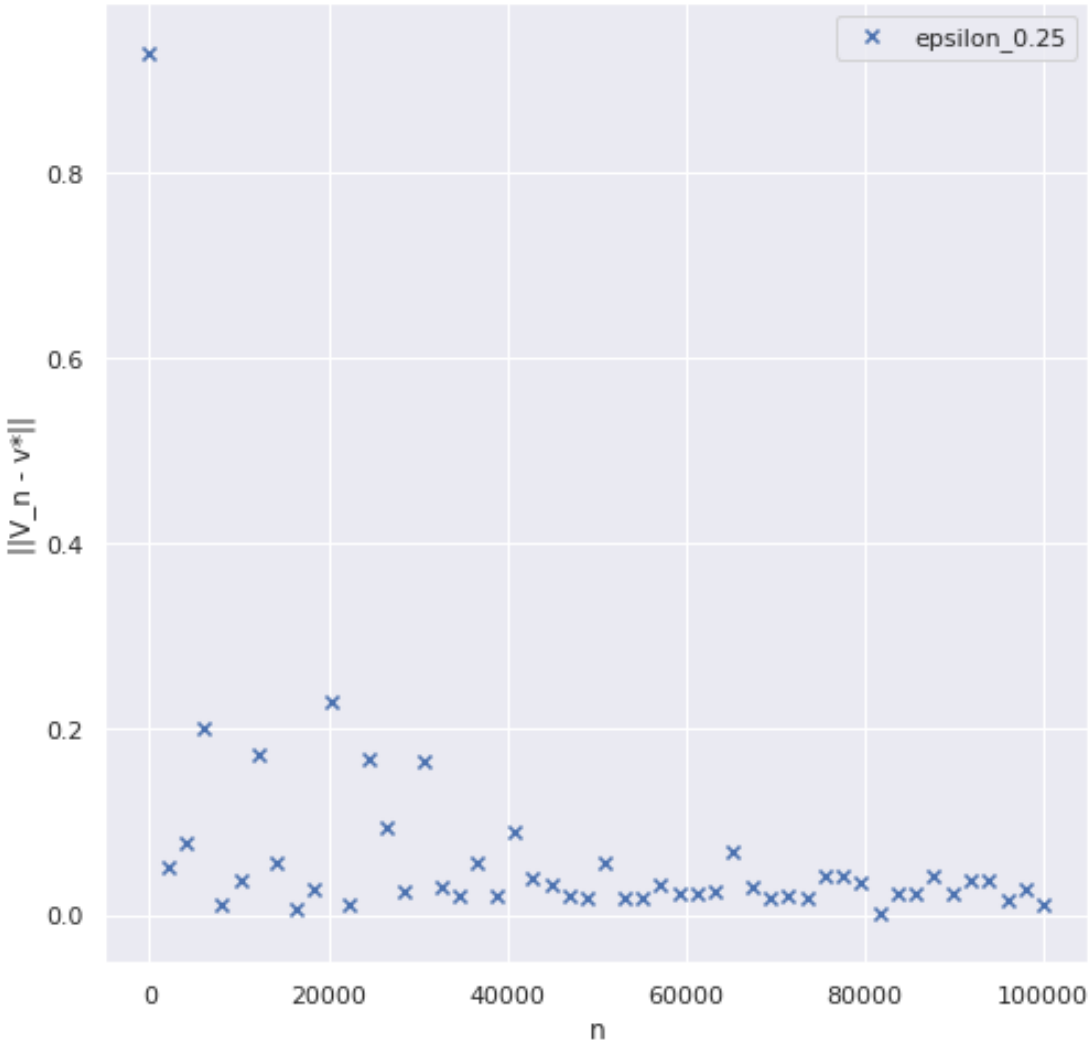
```
In [223]: def computeGreedyPolicy(Q):
              """Return the greedy policy according to Q."""

              # Initialisation of the policy
              policy = np.zeros((env.n_states, 1), dtype=int)

              # Extract the best action according to Q
              for state in range(env.n_states):

                      # Upate the policy value
                      policy[state, 0] = Q[state, :].reshape(-1).argmax()

              return policy

In [224]: def infNorm(V):
              """Compute the inf norm on V."""

              return np.max(np.abs(V))

In [225]: def valueFunctions(Q, policy):
              """Return the value functions for the given policy and Q matrix."""

              # Initialise the value function
              V_pi = np.zeros((env.n_states, 1))

              # Update the true value of the value function for each state
              for state in range(env.n_states):

                      V_pi[state, 0] = Q[state, policy[state, 0]]

              return V_pi
```

2.3.4.1 - Gaps between $||v^* - v^{\pi_n}||$

```
In [241]: def displayVGaps(nb_n=50, epsilon=0.25):
              """Display the gap between v_opt and v_pi_n according to n."""

              # Array of the iteration
              n_l = np.linspace(1, 100000, num=nb_n, dtype=int)
```

```
    # Parameters of the figure
    plt.figure(figsize=(8, 8))
    plt.grid(True)

    # Computation of the Q_n
    Q_n_l = [qLearning(n=n, epsilon=epsilon)[0] for n in n_l]

    # Computation of the policies given Q
    policy_l = [computeGreedyPolicy(Q_n_l[i]) for i in range(nb_n)]

    # Computation of v_n
    V_n_l = [valueFunctions(Q_n_l[i], policy_l[i]) for i in range(nb_n)]

    # Computation of the gap between them
    gaps = [infNorm(V_n_l[i] - v_opt) for i in range(nb_n)]

    # Plot
    plt.scatter(n_l, gaps, label="epsilon_" + str(epsilon), marker="x")

    # Legend
    plt.xlabel("n")
    plt.ylabel("||V_n - v*||")
    plt.legend()

    # Save the plot
    plt.savefig("./Images/Gaps_||V_n - v*||", bbox_inches='tight', pad_inches=0.0)

    # Display
    plt.show()
```

In [242]:
```
displayVGaps()
```



Here we can observe that the error with the optimal value function converge towards zero. In fact, it is coherent with the choice of our epsilon (not null) and our learning rates which respect the conditions of Robbins and Monroe. However, we can observe that the convergence is quite noisy, it means that there is a big variance in the result, above all when the number of simulation is low but it reduces when the number of simulations increases.

2.3.4.2 - Plotting of the cumulated reward

```
In [190]: def displayCumulatedRewards(nb_n=25, epsilon=0.8):
              """Display the cumulated reward according to n."""

              # Parameters of the figure
              plt.figure(figsize=(8, 8))
              plt.grid(True)
```

```
    # Array of the iteration
    n_l = np.linspace(1, 5000, num=nb_n, dtype=int)

    # Array of epsilon
    epsilon_l = np.linspace(0, 1, num=5)

    # For each epsilon display the cumulated rewards
    for epsilon in epsilon_l:

        # Round epsilon
        epsilon = round(epsilon, 2)

        # Computation of the Q_n
        reward_l = [qLearning(n=n, epsilon=epsilon)[1] for n in n_l]

        # Plot
        plt.scatter(n_l, reward_l, label="epsilon_" + str(epsilon), marker="x")

    # Legend
    plt.xlabel("n")
    plt.ylabel("Cumulated Reward")
    plt.legend()

    # Save the plot
    plt.savefig("./Images/Cumulated Reward", bbox_inches='tight', pad_inches=0.0)

    # Display
    plt.show()
```

```
In [191]: displayCumulatedRewards()
```

We can observe that the cumulated sum of rewards depend on the choice of epsilon. In fact, if we choose epsilon=0, we can observe that our cumulated sum is increasing and positive. It seems logic because in this case we have a pure exploitation policy. However, in the extreme case of an epsilon equal to 1, the cumulated sum of rewards is decreasing. In this situation, we have a pure exploration policy and it seems that the absorbing state s_6 is more likely to be reached. Finally, we can observe that epsilon=0.25 seems better than the two past ones. In fact, in this situation, we have a good tradeoff between the exploitation policy and the exploration one which allows to improve our greedy policy globaly and have good rewards at the same time.

3.4 2.4 - Question 6:

We know that the optimal value function V^* is the fixed point of the optimal Bellman operator $\mathcal{T}$. Moreover, this optimal opertor $\mathcal{T}$ does not depend of the initial distribution μ_0. Hence, the optimal value function V^* does not depend on the initial distribution μ_0.

Finally, if we compute the greedy policy with the optimal value function, we find the optimal policy π^*. So, as the computation of the greedy policy does not depend on the initial distribution

μ_0, we have that the optimal policy does not depend on the initial distribution μ_0.

In [ ]: