Reinforcement Learning Course: WiSe 2020/21

Marin Bukov
Faculty of Physics, Sofia University, 5 James Bourchier Blvd., 1164 Sofia, Bulgaria
(Dated: November 14, 2020)

I. MONTE CARLO (MC) METHODS IN REINFORCEMENT LEARNING

In this notebook, we study tabular model-free RL using MC methods. Recall that we introduced MC sampling in Notebook 1, and here we shall make use of some of the sampling techniques we discussed there,

In particular, in this Notebook, we shall code up simple routines for: 1. first-visit MC prediction for value function estimation; 2. MC with exploring starts (ES) for policy optimization.

We follow closely the discussion and pseudocodes from Sutton & Barto, Chapter 5.

Because the tabular MC methods in RL we discussed in class are defined only for episodic tasks, below we shall use the `Episodic_GridWorldEnv` we created in Notebook 2.

```
In [9]: import numpy as np

        import import_ipynb
        from Notebook_2_RL_environments import Episodic_GridWorldEnv # import␣
 ↪environment,
        notebooks must be in same directory
```

Let us begin by initializing the environment.

Apart from fixing the seed for reproducibility, episodic environments also require the argument `n_time_steps` which sets an upper bound on the number of time steps per episode.

```
In [2]: n_time_steps=100 # number of steps in a single episode
        seed=0 # set seed of rng (for reproducibility of the results)

        # discount factor
        gamma = 1.0

        # create environment class
        env=Episodic_GridWorldEnv(n_time_steps=n_time_steps,seed=seed)
```

Our first taks is to write a routine called `policy_rollout` which uses a policy `pi` to generate RL tranjectories

$$\tau = (S_0, A_0, R_1, S_1 A_1, R_2, \dots, S_{\text{terminal}})$$

Generating trajectories is often called a rollout in RL.

Note that if we encounter a terminal state S_{terminal}, then we should stop the rollout.

```
In [3]: def policy_rollout(pi, States, Actions, Rewards):
            """
            pi: np.ndarray of dimension (m,n,a).
                policy to generate trajectory from
            States: list
                append here the encountered states in
            Actions: list
                append here the encountered actions
            Rewards: list
                append here the encountered rewards
```

```
        """

        while env.current_step < env.n_time_steps:

            # read off state
            state=env.state.copy()

            # pick a random action
            action=np.random.choice([0,1,2,3], p=pi[state[0],state[1],:] )

            # take an environment step
            state_prime, reward, done = env.step(action)

            # store trajectory
            States.append(state)
            Actions.append(action)
            Rewards.append(reward)

            # check if state is terminal
            if done:
                break

        return States, Actions, Rewards
```

A. First-visit MC prediction for estimating $V \approx V_\pi$

We are now ready to code up the first-visit MC prediction algorithm to approximating the vaue function $V \approx V_\pi$.

1. initialize the table for the approximate value function `V`, and the empty list `Returns` (see pseudicode in textbook)
2. For each episode:
   - re-set the environment to a random initial state, cf. `env.reset(random=True)`.
   - roll out the policy to sample a trajectory
   - loop backwards over the trajectory to evaluate the return G (you need to do extra work to guarantee the first-visit constraint, see pseudocode)

As usual, it is considered a better practice to write the routine itself in a separate function `first_visit_MC`.

```
In [4]: def first_visit_MC(N_trajectories, pi):
        """
        N_trajectories: int
            Number of trajectories to be sampled.
        pi: np.ndarray of dimension (m,n,a).
            policy to generate trajectories from.

        """

        # initialize variables
        V = np.zeros((4,4),) # (m,n)
        Returns = [[[] for j in range(4)] for i in range(4)]

        # loop over N_trajectories to sample trajectories
```

```
    for episode in range(N_trajectories):

        # set env to a random initial state
        env.reset(random=True)

        # generate episode using pi
        States  = []
        Actions = []
        Rewards = []

        States,Actions,Rewards = policy_rollout(pi, States,Actions,Rewards)

        # compute integer representation
        States_ints = [state[1]*4+state[0] for state in States]

        #print('\nfinished sampling trajectory {} with {}
steps.'.format(n,env.current_step))

        # evaluate trajectory
        G = 0.0
        for t_rev in range(env.current_step-1,-1,-1):
            # read off state
            state     = States[t_rev]
            m,n = state

            # update G
            G*=gamma
            G+=Rewards[t_rev]

            if np.sum([np.all(np.linalg.norm(state-S)<1E-13) for S in States[:
↪t_rev]]) <
        1:
                # append returns
                Returns[m][n].append(G)
                # compute MC average
                V[m,n]=np.mean(Returns[m][n])

    return V
```

Now, let us test the routine

1. using the equiprobable policy

$$\pi(a|s) = \frac{1}{|\mathcal{A}(s)|}$$

Note that $\mathcal{A}(s)$ depends on the states s; in parciular, at the boundary the action space is smaller than in the bulk.

2. define your own policy – be creative!

To do this, we call we print the resulting approximate value function `V`.

3. In both cases, we can set the number of sampled trajectories to `N_trajectories=10000`. How does the result change if we use fewer / more samples?

4. The result we got for the equiprobable random policy differs from the $k \to \infty$ iteration shown in Fig 4.1 (Sutton & Barto). What is this difference due to? *Hint:* did we implement the same boundary conditions as in the book? Why does the definition of the action space play a role here?

```
In [5]: # re-set the seed (reproducibility)
        np.random.seed(seed)

        # define equiprobable policy to be evaluated
        # loop over the states to define equiprobable policy: \
        # the loop is needed because the action space is smaller at the boundary than in␣
 ↪the
        bulk
        pi = np.zeros((4,4,4),) # (m,n,a)
        for m in range(4):
            for n in range(4):
                available_actions=env.actions[m,n]
                pi[m,n,available_actions]=1.0/len(available_actions) # equiprobable; try␣
 ↪out
        something else!

        # enable this policy to implement the uniform boundary condition and reproduce␣
 ↪result
        from boo
        # pi[...,:] = 0.25

        # number of trajectories to sample for the value function estimate
        N_trajectories = 10000


        # use first-visit MC to learn an estimate for the value function $V\approx v_\pi$
        V = first_visit_MC(N_trajectories, pi)

        # print value function estimates
        np.round(V.T, 0)
```

[breakable, size=fbox, boxrule=.5pt, pad at break*=1mm, opacityfill=0][5]:

```
array([[  0., -11., -15., -16.],
       [-11., -14., -16., -15.],
       [-15., -16., -14., -11.],
       [-16., -15., -11.,   0.]])
```

B. Monte Carlo Exploring Starts (ES), for estimating $\pi \approx \pi_*$

Our second task is to code an algorithm which allows us to improve the policy using MC. In particular, here we focus in Monte Carlo with Exploring Starts (ES) to learn an approximation to the optimal policy $\pi \approx \pi_*$.

Use the pseudocode from Chapter 5.3 in Sutton & Barto to write the `MC_Exploring_Starts()` routine. This routine is similar (but somewhat different) to the `first_visit_MC()` function we wrote above. If you encouner difficulties, we recommend that you first extend `first_visit_MC()` to action-value, or `Q`-functions, and then return to the problem below.

In the function below, we also allow to input an initial policy `pi` externally, so we can test the behavior starting from different policy afterwards. Of course, the routing should produce an optimal policy irrespective of the initial policy `pi`.

```
In [6]: def MC_Exploring_Starts(N_trajectories, pi):
        # initialize variables
        Q = np.zeros((4,4,4),) # (m,n,a)
        Returns = [[[ [] for k in range(4)] for j in range(4)] for i in range(4)] #␣
↪(m,n,a)

        # loop over N_trajectories to sample trajectories
        for episode in range(N_trajectories):

            ### implement Exploring Starts
            # choose random (state,action)-pair with equal probability
            env.reset(random=True)
            state=env.state.copy()

            action = np.random.choice(env.actions[state[0],state[1]])


            ### take initial state-action pair
            _, reward, done = env.step(action)

            ### generate episode using pi

            # store initial data
            States  = [state]
            Actions = [action]
            Rewards = [reward]

            # roll out policy
            if not done:
                States,Actions,Rewards = policy_rollout(pi, States,Actions,Rewards)


            # evaluate trajectory
            G = 0.0
            for t_rev in range(env.current_step-1,-1,-1):
                # read off state
                state = States[t_rev]
                m,n = state

                action = Actions[t_rev]

                # update G
                G*=gamma
                G+=Rewards[t_rev]

                # check if state action pair is not present in te trajectory (there␣
↪likely
    is a better way to do this)
                pair_absent_bool = np.sum([np.linalg.norm(state-S)<1E-13 *␣
↪(action==A) for
    S,A in zip(States[:t_rev],Actions[:t_rev]) ]  ) < 1

                if pair_absent_bool: # (S,A) pair not present in trajectory
                    # append returns
                    Returns[m][n][action].append(G)
                    # compute MC average
                    Q[m,n,action]=np.mean(Returns[m][n][action])
                    # improve policy
                    pi[m,n,:] = 0.0
                    pi[m,n,np.argmax(Q[m,n,:])] = 1.0


        return Q, pi
```

Now, let us test the `MC_Exploring_Starts` routine.

1. You may start from any initial policy, including the equiprobable policy, or a random policy. Do you observa any difference? How does this change if we use fewer/more trajectories?

2. Visualize the optimal q_* function `Q_star`. Wxtract the correspondig greedy policy from it; does it agree with the optimal policy `pi_star`?

3. Does the optimal policy depend on the boundary condition? *Hint:* to answer this, you'd have to modify the available action space in the environment.

```
In [8]: # re-set the seed (reproducibility)
        np.random.seed(seed)


        # define equiprobable policy to be evaluated
        # loop over the states to define equiprobable policy: \
        # the loop is needed because the action space is smaller at the boundary than in␣
 ↪the
        bulk
        pi = np.zeros((4,4,4),) # (m,n,a)
        for m in range(4):
            for n in range(4):
                available_actions=env.actions[m,n]
                pi[m,n,available_actions]=1.0/len(available_actions) # equiprobable; try␣
 ↪out
        something else!


        ## define a random initial policy (has to be normalized to a probability distr)
        # pi=np.random.uniform(low=0.0,high=1.0, size=(4,4,4)) # random initial policy
        # norm=np.sum(pi,axis=2)
        # pi=np.einsum('ijk,ij->ijk', pi,1.0/norm)


        # MC step
        N_trajectories = 10000

        Q_star, pi_star = MC_Exploring_Starts(N_trajectories, pi)

        # greedy policy from Q
        print( np.round( np.max(Q_star,axis=2).T,0) )
```

```
[[ 0. -1. -2. -3.]
 [-1. -2. -3. -2.]
 [-2. -3. -2. -1.]
 [-4. -2. -1.  0.]]
```

C. On-policy first-visit MC control (for ε-soft policies), for estimating $\pi \approx \pi_*$

We can remove the exploring starts requirement, by using an ε-soft policy. Modify the function `MC_Exploring_Starts()` to a function `MC_control()` which implements this. The pseudocode is available in Sutton & Barto, Chapter 5.4.

```
In [ ]:
```