Reinforcement Learning Course: WiSe 2020/21

Marin Bukov
Faculty of Physics, Sofia University, 5 James Bourchier Blvd., 1164 Sofia, Bulgaria
(Dated: November 1, 2020)

I. REINFORCEMENT LEARNING (RL) ENVIRONMENTS

In this notebook, we define the backbone code for RL environments, following OpenAI Gym.

Then, we create some example environments that we shall use in subsequent coding sessions throught the course: we will create three gridworld environments: GridWorld, GridWorld2, and Windy GridWorld. We also create a Qubit environment, and discuss some OpenAI Gym environments.

In [ ]:
```
import numpy as np
from scipy.linalg import expm
```

In [ ]:
```
class MyEnv():
    """
    Gym style environment for RL. You may also inherit the class structure from␣
↪OpenAI
Gym.
    Parameters:
        n_time_steps:   int
                        Total number of time steps within each episode
        seed:   int
                seed of the RNG (for reproducibility)
    """

    def __init__(self, n_time_steps, seed):
        """
        Initialize the environment.

        """

        ### define action space variables

        ### define state space variables



        pass



    def step(self, action):
        """
        Interface between environment and agent. Performs one step in the␣
↪environemnt.
        Parameters:
            action: int
                    the index of the respective action in the action array
        Returns:
            output: ( object, float, bool)
                    information provided by the environment about its current␣
↪state:
```

```
                    (state, reward, done)
        """

        pass

        return self.state, reward, done



    def set_seed(self,seed=0):
        """
        Sets the seed of the RNG.

        """
        pass



    def reset(self):
        """
        Resets the environment to its initial values.
        Returns:
            state:  object
                    the initial state of the environment
        """
        pass

        return self.state



    def render(self):
        """
        Plots the state of the environment. For visulization purposes only.

        """
        pass


    # ... add extra private and public functions as necessary
```

A. GridWorld

Consider the GridWorld problem Example 3.5 from Sutton & Barto's "Reinforcement Learning: an Introduction", (MIT Press, 2018):

A 5×5 grid with open boundary conditions has two pairs of special states: (A, A') and (B, B'), such that from state A (B) the environment always goes into stte A' (B'). The state transitions receive the rewards $r(s, s')$:

1. $r(A \to A') = +10$
2. $r(B \to B') = +5$
3. $r(s', s) = 0$ for all other states (except when a move from a boundary state s tries to leave the grid, in which case $r = -1$).

From each state s, the RL agent can take four possible actions a: *north*, *south*, *east*, and *west*.

The **action space** is discrete four-element set $\mathcal{A} = (north, south, east, west)$

The **state space** is the two-dimensional grid $\mathcal{S} = \mathbb{Z}_5^2$: each state $s = (m, n)$ is labeled by two integers $m, n \in \{0, 1, 2, 3, 4\}$. The special states have the coordinates $A = (1, 4)$, $A' = (1, 0)$, $B = (3, 4)$, and $B' = (3, 2)$.

Finally, the **reward space** is given by the discrete set $\mathcal{R} = \{-1, 0, 5, 10\}$.

In [ ]:
```
class GridWorldEnv():
    """
    Gym style environment for GridWorld
    Parameters:
        n_time_steps:   int
                        Total number of time steps within each episode
        seed:   int
                seed of the RNG (for reproducibility)
    """

    def __init__(self, n_time_steps=10, seed=0):
        """
        Initialize the environment.

        """

        self.n_time_steps = n_time_steps

        ### define action space variables
        self.actions=np.array([0,1,2,3])
        #['north', 'south', 'east', 'west'] in coordinate form
        self.action_space = [np.array([0,1]), np.array([0,-1]), np.array([1,0]),
 np.array([-1,0])]

        ### define state space variables
        self.state_A  = np.array([1,4])
        self.state_Ap = np.array([1,0])
        self.state_B  = np.array([3,4])
        self.state_Bp = np.array([3,2])

        # set seed
        self.set_seed(seed)
        self.reset()



    def step(self, action):
        """
        Interface between environment and agent. Performs one step in the␣
 ↪environemnt.
        Parameters:
            action: int
                    the index of the respective action in the action array
        Returns:
            output: ( np.array, float, bool)
                    information provided by the environment about its current␣
 ↪state:
                    (state, reward, done)
        """

        # check if action tries to take state across the grid boundary
        bdry_bool=    (self.state[0]==0 and action==3) or (self.state[0]==4 and
```

```
action==2) \
                    or (self.state[1]==0 and action==1) or (self.state[1]==4 and
action==0)


        # environment dynamics (deterministic)
        if np.linalg.norm(self.state - self.state_A) < 1E-14:
            self.state=self.state_Ap.copy()
            reward=10
        elif np.linalg.norm(self.state - self.state_B) < 1E-14:
            self.state=self.state_Bp.copy()
            reward=5
        elif bdry_bool:
            reward=-1
        else:
            self.state+=self.action_space[action]
            reward=0

        done=False # infinite-horizon task

        self.current_step += 1

        return self.state, reward, done



    def set_seed(self,seed=0):
        """
        Sets the seed of the RNG.

        """
        np.random.seed(seed)



    def reset(self):
        """
        Resets the environment to its initial values.
        Returns:
            state:  np.array
                    the initial state of the environment
        """
        self.current_step = 0

        self.state = np.array([2,2]) #initialize to some state on the grid
        return self.state



    def sample(self):
        """
        Returns a randomly sampled action.
        """
        return np.random.choice(self.actions) # equiprobable policy
```

Let us now test the GridWorld environment. We do so by fixing the number of time steps, `n_time_steps`, and the `seed`. We then create the environment and reset it. Finally, we want to loop over the

```
In [ ]: n_time_steps=20
        seed=0

        env=GridWorldEnv(n_time_steps=n_time_steps,seed=seed)
        env.reset()

        for _ in range(n_time_steps):

            # pick a random action
            action=env.sample() # equiprobable policy

            # take an environment step
            state=env.state.copy()
            state_p, reward, done = env.step(action)

            print("{}. s={}, a={}, r={}, s'={}".format(_, state, env.
 ↪action_space[action],
        reward, state_p))
```

B. GridWorld 2

This is a finite-horizon, i.e. episodic, GridWorld environment. We consider the 4×4 grid from Example 4.1 in Sutton & Barto.

state space: $\mathcal{S} = \{0, 1, 2, \ldots, 15\}$, where $0 = s = 15$ is the terminal state.

action space: $\mathcal{A} = \{north, south, east, west\}$. Actions trying to take the agent off the grid leave the state unchanged: to implement this behavior, we will define smaller actions spaces $\mathcal{A}(s_{\text{boundary}})$ for all states s_{boundary} at the bounary of the grid.

reward space: $\mathcal{R} = \{-1\}$; $r(s, s', a) = -1$ for all states $s, s' \in \mathcal{S}$ and all allowed actions $a \in \mathcal{A}(s)$.

```
In [ ]: class Episodic_GridWorldEnv():
            """
            Gym style environment for GridWorld
            Parameters:
                n_time_steps:   int
                                Total number of time steps within each episode
                seed:   int
                        seed of the RNG (for reproducibility)
            """

            def __init__(self, n_time_steps=10, seed=0):
                """
                Initialize the environment.

                """

                self.n_time_steps = n_time_steps

                ### define action space variables
                #['north', 'south', 'east', 'west']
                self.action_space = [np.array([0,1]), np.array([0,-1]), np.array([1,0]),
        np.array([-1,0])]
                # define the allowed actions from every state s, taking into account the
        boundary
                self.actions={}
                for m in range(4):
```

```
                for n in range(4):

                    if m==0:
                        if n==0:
                            self.actions[m,n]=np.array([0,2])
                        elif n==3:
                            self.actions[m,n]=np.array([1,2])
                        else:
                            self.actions[m,n]=np.array([0,1,2])

                    elif m==3:
                        if n==0:
                            self.actions[m,n]=np.array([0,3])
                        elif n==3:
                            self.actions[m,n]=np.array([1,3])
                        else:
                            self.actions[m,n]=np.array([0,1,3])

                    elif 0<m<3:
                        if n==0:
                            self.actions[m,n]=np.array([0,2,3])
                        elif n==3:
                            self.actions[m,n]=np.array([1,2,3])
                        else:
                            self.actions[m,n]=np.array([0,1,2,3])


            ### define state space variables
            # the two terminal states
            self.state_T1  = np.array([0,0])
            self.state_T2  = np.array([3,3])

            # set seed
            self.set_seed(seed)
            self.reset()



        def step(self, action):
            """
            Interface between environment and agent. Performs one step in the␣
↪environemnt.
            Parameters:
                action: int
                        the index of the respective action in the action array
            Returns:
                output: ( np.array, float, bool)
                        information provided by the environment about its current␣
↪state:

                        (state, reward, done)
            """

            # check if action tries to take state across the grid boundary
            bdry_bool=    (self.state[0]==0 and action==3) or (self.state[0]==3 and
    action==2) \
                       or (self.state[1]==0 and action==1) or (self.state[1]==3 and
```

```
action==0)

        # environment dynamics (deterministic)

        reward=-1 # all trasitions have reward -1

        # if state is not at the boundary, update the state
        if not bdry_bool:
            self.state+=self.action_space[action]

        done=False
        if np.linalg.norm(self.state - self.state_T1) < 1E-14 or
np.linalg.norm(self.state - self.state_T2) < 1E-14:
            done=True

        self.current_step += 1

        return self.state, reward, done



    def set_seed(self,seed=0):
        """
        Sets the seed of the RNG.

        """
        np.random.seed(seed)



    def reset(self, random=False):
        """
        Resets the environment to its initial values.
        Returns:
            state:  np.array
                    the initial state of the environment
            random: bool
                    controls whether the initial state is a random state on the␣
↪grid or
a fixed initials state.
        """

        self.current_step = 0

        if random:
            self.state = np.random.randint(4,size=(2))
            while np.linalg.norm(self.state - self.state_T1) < 1E-14 or
np.linalg.norm(self.state - self.state_T2) < 1E-14:
                self.state = np.random.randint(4,size=(2))
        else:
            self.state = np.array([2,2]) #initialize to some state on the grid

        return self.state
```

Let us test the environment to make sure it is implemented properly. Note that we are fixing the seed, so if you want to see a different output, you should change the value of `seed`.

In [ ]:
```
env=Episodic_GridWorldEnv()

seed=4
env.set_seed(seed)

env.reset()


done=False
j=0
while not done:

    state=env.state.copy()

    #print(env.actions[state[0],state[1]])

    # pick a random action
    action=np.random.choice(env.actions[state[0],state[1]]) # equiprobable␣
↪policy from
state s

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

    print("{0:2d}. s={1}, a={2:}, r={3:2d}, s'={4}".format(j, state,
env.action_space[action], reward, state_p))

    j+=1

    if done:
        print('\nreached terminal state!')
        break
```

C. Windy GridWorld

This is a finite-horizon, i.e. episodic, GridWorld environment. We consider the 10×7 grid from Example 6.5 in Sutton & Barto.

state space: $\mathcal{S} = \{(m, n) | m = 0, \ldots, 9, n = 0, \ldots, 6\}$, where the terminal state is $G = (7, 3)$.

action space: $\mathcal{A} = \{north, south, east, west\}$; actions trying to take the agent off the grid leave the state unchanged.

reward space: $\mathcal{R} = \{-1\}$; $r(s, s', a) = -1$ for all states $s, s' \in \mathcal{S}$ and allowed actions $a \in \mathcal{A}(s)$.

In [ ]:
```
class WindyGridWorldEnv():
    """
    Gym style environment for GridWorld
    Parameters:
        n_time_steps:   int
                        Total number of time steps within each episode
        seed:   int
                seed of the RNG (for reproducibility)
    """

    def __init__(self, n_time_steps=10, seed=0):
        """
        Initialize the environment.
```

```
        """

        self.n_time_steps = n_time_steps

        ### define action space variables
        #['north', 'south', 'east', 'west']
        self.action_space = [np.array([0,1]), np.array([0,-1]), np.array([1,0]),
    np.array([-1,0])]

        # wind shift
        self.wind = np.array([0,0,0,1,1,1,2,2,1,0])

        ### define state space variables
        # the initial and terminal states
        self.state_S  = np.array([0,3]) # initial state
        self.state_G  = np.array([7,3]) # terminal state

        # set seed
        self.set_seed(seed)
        self.reset()



    def step(self, action):
        """
        Interface between environment and agent. Performs one step in the␣
↪environemnt.
        Parameters:
            action: int
                    the index of the respective action in the action array
        Returns:
            output: ( np.array, float, bool)
                    information provided by the environment about its current␣
↪state:

                    (state, reward, done)
        """

        # check if action tries to take state across the grid boundary
        bdry_bool=    (self.state[0]==0 and action==3) or (self.state[0]==9 and
    action==2) \
                   or (self.state[1]==0 and action==1) or (self.state[1]==6 and
    action==0)

        # environment dynamics (deterministic)
        reward=-1 # all trasitions have reward -1

        if not bdry_bool:
            # check if wind pushes state outside the boundary
            if self.state[1]+self.wind[self.state[0]]+self.
↪action_space[action][1]<=6:
                self.state[1]+=self.wind[self.state[0]]
            self.state+=self.action_space[action]


        # check if state is terminal
```

```
            done=False
            if np.linalg.norm(self.state - self.state_G) < 1E-14:
                done=True

            self.current_step += 1

            return self.state, reward, done



        def set_seed(self,seed=0):
            """
            Sets the seed of the RNG.

            """
            np.random.seed(seed)



        def reset(self, random=False):
            """
            Resets the environment to its initial values.
            Returns:
                state:  np.array
                        the initial state of the environment
                random: bool
                        controls whether the initial state is a random state on the␣
↪grid or
        a fixed initials state.
            """

            self.current_step = 0

            self.state = self.state_S.copy() #initialize to S

            return self.state
```

Let us test the Windy GridWorld

```
In [ ]: env=WindyGridWorldEnv()
        env.reset()

        done=False
        j=0
        while not done:

            # pick a random action
            action=np.random.choice([0,1,2,3]) # equiprobable policy

            # take an environment step
            state=env.state.copy()
            state_p, reward, done = env.step(action)

            print("{}. s={}, a={}, r={}, s'={}".format(j, state, env.
↪action_space[action],
        reward, state_p))
```

```
    j+=1

    if done:
        print('\nreached terminal state!')
        break
```

D. Qubit Environment

We now define an environment for a quantum bit of information (qubit).

1. Basic Definitions

The state of a qubit $|\psi\rangle \in \mathbb{C}^2$ is modeled by a two-dimensional complex-valued vector with unit norm: $\langle\psi|\psi\rangle := \sqrt{|\psi_1|^2 + |\psi_2|^2} = 1$. Every qubit state is uniquely described by two angles $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi)$:

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} = \mathrm{e}^{i\alpha} \begin{pmatrix} \cos\frac{\theta}{2} \\ \mathrm{e}^{i\varphi} \sin\frac{\theta}{2} \end{pmatrix} \tag{1}$$

The overall phase α of a single quantum state has no physical meaning. Thus, any qubit state can be pictured as an arrow on the unit sphere (called the Bloch sphere) with coordinates (θ, ϕ).

To operate on qubits, we use quantum gates. Quantum gates are represented as unitary transformations $U \in \mathrm{U}(2)$, where $\mathrm{U}(2)$ is the unitary group. Gates act on qubit states by matrix multiplication to transform an input state $|\psi\rangle$ to the output state $|\psi'\rangle$: $|\psi'\rangle = U|\psi\rangle$. For this problem, we consider four gates

$$U_0 = \mathbf{1}, \qquad U_x = \exp(-i\delta t\sigma^x/2), \qquad U_y = \exp(-i\delta t\sigma^y/2), \qquad U_z = \exp(-i\delta t\sigma^z/2), \tag{2}$$

where δt is a fixed time step, $\exp(\cdot)$ is the matrix exponential, $\mathbf{1}$ is the identity, and the Pauli matrices are defined as

$$\mathbf{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad \sigma^x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \sigma^y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \qquad \sigma^z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{3}$$

To determine if a qubit, described by the state $|\psi\rangle$, is in a desired target state $|\psi_{\mathrm{target}}\rangle$, we compute the fidelity

$$F = |\langle\psi_{\mathrm{target}}|\psi\rangle|^2 = |(\psi_{\mathrm{target}})_1^*\psi_1 + (\psi_{\mathrm{target}})_2^*\psi_2|^2, \qquad F \in [0, 1] \tag{4}$$

where $*$ stands for complex conjugation. Physically, the fidelity corresponds to the angle between the arrows representing the qubit state on the Bloch sphere (we want to maximize the fidelity but minimize the angle between the states).

2. Constructing the Qubit Environment

Now, let us define an RL environment, which contains the laws of physics that govern the dynamics of the qubit (i.e. the application of the gate operations to the qubit state). Our RL agent will later interact with this environment to learn how to control the qubit to bring it from an initial state to a prescribed target state.

We define the RL states $s = (\theta, \varphi)$ as an array containing the Bloch sphere angles of the quantum state. Each step within an episode, the agent can choose to apply one out of the actions, corresponding to the four gates $(\mathbf{1}, U_x, U_y, U_z)$. We use the instantaneous fidelity w.r.t. the target state as a reward: $r_t = F = |\langle\psi_*|\psi(t)\rangle|^2$:

state space: $\mathcal{S} = \{(\theta, \varphi) | \theta \in [0, \pi], \varphi \in [0, 2\pi)\}$. The terminal states are a region of the Bloch sphere around the target state $|\psi_{\text{target}}\rangle = (1, 0)^t$ (i.e. the qubit state we want to prepare): the target qubit state has the Bloch sphere coordinates $s_{\text{terminal}} = (0, 0)$, so the region corresponds to polar cap close to the pole; the size of the polar cap is set by some small number `cap_size=1E-2`.

action space: $\mathcal{A} = \{\mathbf{1}, U_x, U_y, U_z\}$. Actions act on RL states as follows: 1. if the current state is $s = (\theta, \varphi)$, we first create the quantums state $|\psi(s)\rangle$; 2. we apply the gate U_a corresponding to action a to the quantum state, and obtain the new quantum state $|\psi(s')\rangle = U_a|\psi(s)\rangle$. 3. last, we compute the Bloch sphere coordinates which define the next state $s' = (\theta', \varphi')$, using the Bloch sphere parametrization for qubits given above. Note that all actions are allowed from every state.

reward space: $\mathcal{R} = [0, 1]$. We use the fidelity between the next state s' and the terminal state s_{terminal} as a reward at every episode step:

$$r(s, s', a) = F = |\langle\psi_{\text{target}}|U_a|\psi(s)\rangle|^2 = |\langle\psi_{\text{target}}|\psi(s')\rangle|^2$$

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

```
In [ ]: class QubitEnv():
            """
            Gym style environment for RL. You may also inherit the class structure from␣
 ↪OpenAI
         Gym.
            Parameters:
                n_time_steps:   int
                                Total number of time steps within each episode
                seed:   int
                        seed of the RNG (for reproducibility)
            """

            def __init__(self, n_time_steps, seed):
                """
                Initialize the environment.

                """

                self.n_time_steps = n_time_steps


                ### define action space variables
                delta_t = 2*np.pi/n_time_steps # set a value for the time step
                # define Pauli matrices
                Id     =np.array([[1.0,0.0  ], [0.0 ,+1.0]])
                sigma_x=np.array([[0.0,1.0  ], [1.0 , 0.0]])
                sigma_y=np.array([[0.0,-1.0j], [1.0j, 0.0]])
                sigma_z=np.array([[1.0,0.0  ], [0.0 ,-1.0]])

                self.action_space=[]
                for generator in [Id, sigma_x, sigma_y, sigma_z]:
                    self.action_space.append( expm(-1j*delta_t*generator) )


                ### define state space variables
                self.S_terminal = np.array([0.0,0.0])
                self.psi_terminal = self.RL_to_qubit_state(self.S_terminal)
                self.cap_size = 1E-2
```

```
        # set seed
        self.set_seed(seed)
        self.reset()


    def step(self, action):
        """
        Interface between environment and agent. Performs one step in the␣
↪environemnt.

        Parameters:
            action: int
                    the index of the respective action in the action array
        Returns:
            output: ( object, float, bool)
                    information provided by the environment about its current␣
↪state:

                    (state, reward, done)
        """

        # apply gate to quantum state
        self.psi = self.action_space[action].dot(self.psi)

        # compute RL state
        self.state = self.qubit_to_RL_state(self.psi)

        # compute reward
        reward = np.abs( self.psi_terminal.conj().dot(self.psi)  )**2


        # check if state is terminal
        done=False
        if np.abs(reward - 1.0) < self.cap_size:
            done=True


        return self.state, reward, done



    def set_seed(self,seed=0):
        """
        Sets the seed of the RNG.

        """
        np.random.seed(seed)



    def reset(self, random=True):
        """
        Resets the environment to its initial values.
        Returns:
            state:  object
                    the initial state of the environment
            random: bool
```

```
                          controls whether the initial state is a random state on the␣
↪sphere
        or a fixed initial state.
        """

        if random:
            theta = np.pi*np.random.uniform(0.0,1.0)
            phi = 2*np.pi*np.random.uniform(0.0,1.0)
        else:
            # start from south pole of Bloch sphere
            theta=np.pi
            phi=0.0

        self.state=np.array([theta,phi])
        self.psi=self.RL_to_qubit_state(self.state)

        return self.state



    def render(self):
        """
        Plots the state of the environment. For visulization purposes only.

        """
        pass


    def RL_to_qubit_state(self,s):
        """
        Take as input the RL state s, and return the quantum state |psi>
        """
        theta, phi = s
        psi = np.array([np.cos(0.5*theta), np.exp(1j*phi)*np.sin(0.5*theta)] )
        return psi


    def qubit_to_RL_state(self,psi):
        """
        Take as input the RL state s, and return the quantum state |psi>
        """
        # take away unphysical global phase
        alpha = np.angle(psi[0])
        psi_new = np.exp(-1j*alpha) * psi

        # find Bloch sphere angles
        theta = 2.0*np.arccos(psi_new[0]).real
        phi = np.angle(psi_new[1])

        return np.array([theta, phi])
```

In [ ]:
```
np.set_printoptions(suppress=True,precision=2)

n_time_steps = 100
seed=6

env=QubitEnv(n_time_steps,seed)
```

```
    env.reset(random=True)

    done=False
    j=0
    while not done:

        # pick a random action
        action=np.random.choice([0,1,2,3]) # equiprobable policy

        # take an environment step
        state=env.state.copy()
        state_p, reward, done = env.step(action)

        print("{}. s={}, a={}, r={}, s'={}\n".format(j, state, action, np.
↪round(reward,6),
    state_p))

        j+=1

        if done:
            print('\nreached terminal state!')
            break
```

E. OpenAI Gym Environments

Next, we shall look at some OpenAI environments: Atari video games, the Cart Pole problem, and the Mountain Car problem.

```
In [ ]: import matplotlib.pyplot as plt
        %matplotlib inline
        from IPython import display

        import gym
        from IPython import display
        import matplotlib
        import matplotlib.pyplot as plt
        %matplotlib inline



        #env = gym.make('BreakoutDeterministic-v4')
        #env = gym.make('SpaceInvaders-v0')

        #env = gym.make('CartPole-v1')
        env = gym.make('MountainCar-v0')


        env.reset()
        img = plt.imshow(env.render(mode='rgb_array')) # only call this once


        n_time_steps=100
        for _ in range(n_time_steps):
            # plot frame
            img.set_data(env.render(mode='rgb_array')) # just update the data
```

```
        display.display(plt.gcf())
        display.clear_output(wait=True)
        # choose action
        action = env.action_space.sample()
        # take action
        frame, reward, is_done, _ = env.step(action)
```

```
In [ ]: print(frame.shape, reward, is_done, _)
```

```
In [ ]: print(env.__dir__() )
```