# Reinforcement Learning Course: WiSe 2020/21

Marin Bukov
*Faculty of Physics, Sofia University, 5 James Bourchier Blvd., 1164 Sofia, Bulgaria*
(Dated: November 22, 2020)

## I. TEMPORAL DIFFERENCE LEARNING

In this notebook, we study in practice Temporal Difference (TD) methods. We follow closely the discussion in Sutton & Barto, Chapter 6.

Like Monte Carlo methods, TD methods do not require knowledge of the transition probabilities $p(s', r|s, a)$, i.e. they are model-free; TD methods learn directly from experience. Like Dynamic Programming methods, TD methods update incrementally the value function, and can also be used for inifinite-horizeon tasks.

Below, you will implement:

1. Policy Evaluation using TD(0),

2. On-Policy Control: SARSA
3. Off-Policy Control: Q-Learning

As an example, we use the WindyGridWorld environment defined in Notebook 2.

In [2]:
```
import numpy as np
import import_ipynb
from Notebook_2_RL_environments import WindyGridWorldEnv # import environment,␣
↪notebooks
must be in same directory
```

In [2]:
```
# set seed of rng (for reproducibility of the results)
seed=0
np.random.seed(seed)

# create environment class
env=WindyGridWorldEnv(seed=seed)
```

### A. Policy Evaluation

First, we implement Policy Evaluation using TD learning.

In [11]:
```
def policy_evaluation(pi, N_episodes, alpha, gamma, verbose=True):
    """
    pi: np.ndarray
        policy to be evaluated.
    N_episodes: int
        number of training episodes.
    alpha: double
        learning rate or step-size parameter. Should be in the interval [0,1].
    gamma: double
        discount factor. Should be in the interval [0,1].
    verbose: bool
        whether or not to input progress.

    """
```

```
        # initialize value function V
        V = np.zeros((10,7),)

        # policy evaluation
        for episode in range(N_episodes):

            # reset environment
            env.reset()

            # loop over the steps in an episode
            done=False
            while (not done):

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

                # take an environment step
                S_p, R, done = env.step(A)

                # update value function
                V[S[0],S[1]] += alpha*(R + gamma*V[S_p[0],S_p[1]] - V[S[0],S[1]])

            if episode%10==0 and verbose:
                print('finished episode {0:d}.'.format(episode))

        return V
```

Define an equiprobably policy and evaluate it using TD learning. Use $\alpha = 0.5$, $\gamma = 0.9$, and `N_episodes=100`.

1. What happens if you change the value for the discount factor $\gamma$? Can you explain your observations?
2. Does the result depend on the number of episodes used? Why?
3. Try to evaluate the policy which deterministically takes the action `right`. What can go wrong here? Can you come up with a way to fix the issue you observe?

```
In [12]: # learning rate
         alpha = 0.5

         # discount factor
         gamma = 0.9

         # number of episodes to collect data from
         N_episodes=100

         # define equiprobable policy
         pi_equiprob=0.25*np.ones((10,7,4))

         # evaluate the valur function for the equiproably policy
         V_equiprob = policy_evaluation(pi_equiprob, N_episodes, alpha, gamma)

         # print value function
         np.round(np.flipud(V_equiprob.T),0)
```

```
finished episode 0.
finished episode 10.
finished episode 20.
```

```
finished episode 30.
finished episode 40.
finished episode 50.
finished episode 60.
finished episode 70.
finished episode 80.
finished episode 90.
```

[breakable, size=fbox, boxrule=.5pt, pad at break*=1mm, opacityfill=0][12]:

```
array([[-10., -10., -10., -10., -10., -10., -10., -10., -10., -10.],
       [-10., -10., -10., -10., -10., -10., -10., -10., -10., -10.],
       [-10., -10., -10., -10., -10., -10., -10., -10., -10.,  -9.],
       [-10., -10., -10., -10., -10., -10., -10.,   0., -10.,  -9.],
       [-10., -10., -10., -10., -10., -10.,   0.,  -8.,  -5.,  -9.],
       [-10., -10., -10., -10., -10.,   0.,   0., -10., -10.,  -9.],
       [-10., -10., -10., -10.,   0.,   0.,   0.,   0.,  -7.,  -9.]])
```

## B. SARSA

We now turn to policy improvement, and implement the SARSA algorithm. SARSA is an on-policy algorithm, i.e. the policy being improved is the same policy which generates the data. The algorithm makes use of generalized policy iteration to find an approximation to the optimal $Q$-function using experience (i.e. from data).

SARSA requires us to be able to take actions according to an $\varepsilon$-greedy policy. Therefore, your first task is to implement a function `take_eps_greedy_action(Qs,eps,avail_actions)` which takes an action according to the $\varepsilon$-greedy policy w.r.t. the $Q$-function `Q(s,:)` for some fixed state $s$. For simiplicity, we assume that all actions are available from every state (see definition of WindyGridWorld environment).

Once we have `take_eps_greedy_action`, we can move on to implement the `SARSA(N_episodes, alpha, gamma, eps)` routine.

In [5]:
```
def take_eps_greedy_action(Qs,eps,avail_actions=np.array([0,1,2,3])):
    """
    Qs: np.ndarray
        the Q(s,:) values for a fixed state, over all available actions from␣
 ↪that state.
    eps: double
        small number to define the eps-greedy policy.
    avail_actions: np.ndarray
        an array containing all allowed actions from the state s.
        For the WindyGridWorld environment, these are always all four actions.

    """

    # compute greedy action
    a = np.argmax(Qs)

    # draw a random number in [0,1]
    delta=np.random.uniform()

    # take a non=greedy action with probability eps/|A|
    if delta < eps/avail_actions.shape[0]:
        a = np.random.choice( avail_actions[np.arange(len(avail_actions)) != a] )

    return a
```

```
In [6]: def SARSA(N_episodes, alpha, gamma, eps):
            """
            N_episodes: int
                number of training episodes
            alpha: double
                learning rate or step-size parameter. Should be in the interval [0,1].
            gamma: double
                discount factor. Should be in the interval [0,1].
            eps: double
                the eps-greedy policy paramter. Control exploration. Should be in the␣
 ↪interval
        [0,1].
            """

            # initialize Q function
            Q = np.zeros((10,7,4),) # (10,7)-grid of 4 actions


            # policy evaluation
            for episode in range(N_episodes):

                # reset environment and compute initial state
                S=env.reset().copy()
                # take first action using eps-greedy policy
                A=take_eps_greedy_action(Q[S[0],S[1],:],eps)

                # loop over the timesteps in the episode
                done=False
                while not done:

                    # take an environment step
                    S_p, R, done = env.step(A)

                    # choose A_p
                    A_p=take_eps_greedy_action(Q[S_p[0],S_p[1],:],eps)

                    # update value function
                    Q[S[0],S[1],A] += alpha*(R + gamma*Q[S_p[0],S_p[1],A_p] -␣
 ↪Q[S[0],S[1],A])

                    # update states
                    S=S_p.copy()
                    A=A_p.copy()

            return Q
```

Now that we implemented SARSA, we can evaluate the optimal $q_*(s,a)$ and $v_*(s)$ functions for a fixed value of $\varepsilon = 0.1$.

- What is the role of the parameters `N_episodes` when it comes to the convergence speed?
- Print the value function corresponding to `Q_SARSA`.
- Extract the optimal policy from `Q_SARSA`.

Use your code now to gain intuition about how the algorithm behahaves:

- How do the results change if you decrease the value of `eps`?
- How do the results change if you increase/decrease the step size paramter `alpha`?

In [7]:
```
# learning rate
alpha = 0.5

# discount factor
gamma = 1.0

# epsilon: small positive number used in the definition of the epsilon-greedy␣
↪policy
eps = 0.1

# number of episodes to collect data from
N_episodes=1000

# use SARSA to compute the optimal Q function
Q_SARSA=SARSA(N_episodes, alpha, gamma, eps)

# extract and plot the corresponding value function
V_SARSA=np.max(Q_SARSA,axis=2)
np.round(np.flipud(V_SARSA.T),0)

# compute and print the optimal policy
```

[breakable, size=fbox, boxrule=.5pt, pad at break*=1mm, opacityfill=0][7]:

```
array([[-16., -16., -14., -13., -11., -10.,  -9.,  -8.,  -7.,  -6.],
       [-16., -16., -14., -12., -13., -11., -10., -10.,  -8.,  -5.],
       [-17., -16., -15., -14., -12., -12., -11.,  -7.,  -8.,  -4.],
       [-15., -15., -14., -13., -13., -11., -10.,   0.,  -7.,  -3.],
       [-17., -16., -15., -14., -12., -11.,   0.,  -1.,  -1.,  -2.],
       [-16., -15., -14., -13., -12.,   0.,   0.,  -2.,  -2.,  -4.],
       [-15., -14., -13., -13.,   0.,   0.,   0.,   0.,  -2.,  -4.]])
```

### C. Q-Learning

Q-Learning is an off-policy algorithm, i.e. the policy being improved can be different from the behavior policy which generates the data. Like SARSA, Q-Learning makes use of generalized policy iteration to find an approximation to the optimal $Q$-function using experience (i.e. from data).

The Q-Learning algorithm forms the basics of modern Deep RL studies; the off-policy character allows to learn from old data which makes it particularly suitable for data-driven deep learning approaches.

As with SARSA, we will make use of the routine `take_eps_greedy_action` defined above.

In [8]:
```
def Q_Learning(N_episodes, alpha, gamma, eps):
    """
    N_episodes: int
        number of training episodes
    alpha: double
        learning rate or step-size parameter. Should be in the interval [0,1].
    gamma: double
        discount factor. Should be in the interval [0,1].
    eps: double
        the eps-greedy policy paramter. Control exploration. Should be in the␣
↪interval
  [0,1].
    """

    # initialize Q function
    Q = np.zeros((10,7,4),) # (10,7)-grid of 4 actions
```

```
    # policy evaluation
    for episode in range(N_episodes):

        # reset environment and compute initial state
        S=env.reset().copy()

        # loop over the timesteps in the episode
        done=False
        while not done:

            # choose action
            A=take_eps_greedy_action(Q[S[0],S[1],:],eps)

            # take an environment step
            S_p, R, done = env.step(A)

            # update value function
            Q[S[0],S[1],A] += alpha*(R + gamma*np.max(Q[S_p[0],S_p[1],:]) -
Q[S[0],S[1],A])

            # update states
            S=S_p.copy()

    return Q
```

Like with SARSA, we'd like to first evaluate the optimal $q_*(s,a)$ and $v_*(s)$ functions for a fixed value of $\varepsilon = 0.1$.

- What is the role of the parameters `N_episodes` when it comes to the convergence speed?
- Print the value function `V_QL` corresponding to `Q_QL`.
- Extract the optimal policy from `Q_QL`.
- Now use the `policy_evaluation` routine to evaluate the optimal policy and visualize its value function `V`; What is the relation between the value function `V`, and the value function `V_QL` obtained from `Q_QL`?

Use your code now to gain intuition about how the algorithm behahaves:

- How do the results change if you decrease the value of `eps`?
- How do the results change if you increase/decrease the step size paramter `alpha`?

```
In [13]: # learning rate
         alpha = 0.5

         # discount factor
         gamma = 1.0

         # epsilon: small positive number used in the definition of the epsilon-greedy␣
 ↪policy
         eps = 0.1

         # number of episodes to collect data from
         N_episodes=1000

         # use Q-Learning to compute the optimal Q function
         Q_QL=Q_Learning(N_episodes, alpha, gamma, eps)
```

```
# extract and plot the corresponding value function
V_QL=np.max(Q_QL,axis=2)
np.round(np.flipud(V_QL.T),0)

# compute and print the optimal policy
pi_QL=np.zeros((10,7,4))

# greedy policy associated with Q function
amax=np.argmax(Q_QL,axis=2)
for m in range(10):
    for n in range(7):
        pi_QL[m,n,amax[m,n]]=1

# now use the policy_evaluation routine to evaluate the optimal policy and␣
↪visualize its
value function.
V=policy_evaluation(pi_QL, N_episodes, alpha, gamma, verbose=False)
np.round(np.flipud(V.T),0)
```

[breakable, size=fbox, boxrule=.5pt, pad at break*=1mm, opacityfill=0][13]:

```
array([[  0.,   0.,   0.,   0.,   0.,   0.,  -9.,  -8.,  -7.,  -6.],
       [  0.,   0.,   0.,   0.,   0., -10.,   0.,   0.,   0.,  -5.],
       [  0.,   0.,   0.,   0., -11.,   0.,   0.,   0.,   0.,  -4.],
       [-15., -14., -13., -12.,   0.,   0.,   0.,   0.,   0.,  -3.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,  -1.,  -2.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.]])
```

In [ ]: