# Reinforcement Learning Course: WiSe 2020/21

Marin Bukov
*Faculty of Physics, Sofia University, 5 James Bourchier Blvd., 1164 Sofia, Bulgaria*
(Dated: November 7, 2020)

## I. DYNAMIC PROGRAMMING (DP)

In this notebook, we will code the following DP algorithms:

1. Policy Evaluation
2. Policy Iteration
3. Value Iteration

As an example, we shall use the GridWorld environment defined in Notebook 2.
We follow closely the discussion in Sutton & Barto, Chapter 4.

```
In [11]: import numpy as np
         # load Notebook_2
         import import_ipynb
         from Notebook_2_RL_environments import GridWorldEnv # import environment, both␣
↪notebooks
         must be in same directory
```

Next, we fix the hyperparameters, and create the environment object:

```
In [2]: ### define rng seed
        seed=0

        ### construct the Gridworld environment
        env=GridWorldEnv(seed=seed)

        ### set algorithm parameters
        # define convergence threshold theta
        theta = 1E-9
        # discount factor
        gamma = 0.9
```

### A. Policy Evaluation

We now consider the GridWorld environment from Notebook_2. Suppose an RL agent acts with the equiprobable (random) policy

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

Our first task is to use Iterative Policy Evaluation to determine the value function $V(s) \approx v_\pi(s)$. We shall do this in two steps: 1. write a routine `policy_evaluation()` which accepts a policy `pi`, the convergence threshold parameter `theta`, and the discount factor `gamma`. 2. define the equiprobable policy, and use it as input for `policy_evaluation()` to compute the corresponding value function. 3. plot the value function as a table to compare it with the result from Sutton & Barto. 4. define a new policy, which is non-uniform policy over the actions but still uniform over the state space: $\pi'(a|s) = \pi'(a) = [0.1, 0.3, 0.5, 0.1]$, and compute $v_{\pi'}$. 5. determine which of the following is true: $\pi' > \pi$, $\pi' \geq \pi$, $\pi' = \pi$, $\pi' \leq \pi$, $\pi' < \pi$.

In [3]:
```
def policy_evaluation(pi, theta, gamma):

    # initialize approximant value function V
    V = np.zeros((5,5),) # 5x5 grid => 25 states

    # define differece parameter Delta
    Delta=1.0

    # policy evaluation
    while Delta>=theta:
        Delta=0.0

        # loop over all states in the state space
        for m in range(5):
            for n in range(5):
                # back up old value for state
                V_old = V[m,n]

                # evaluate update rule
                V_aux=0.0
                for action in range(len(env.action_space)):

                    # set environment state
                    env.state=np.array([m,n])
                    # take step
                    s_prime, reward, _ = env.step(action)

                    # increment V(s)
                    V_aux += pi[m,n,action]*(reward +␣
↪gamma*V[s_prime[0],s_prime[1]])

                V[m,n]=V_aux
                # compute Delta
                Delta = max(Delta, np.abs(V[m,n]-V_old))

    return V
```

In [4]:
```
# define policy over the action space
pi = 1.0/4.0*np.ones((5,5,4),) # equiprobable/random policy
#pi = np.tile(np.array([0.1,0.3,0.5,0.1]), (5,5,1) ) # another, non-uniform␣
↪policy


# run policy evaluation
V=policy_evaluation(pi,theta,gamma)

# print V, rounded to the first decimal
np.round(np.flipud(V.T),1)
```

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

```
array([[ 3.3,  8.8,  4.4,  5.3,  1.5],
       [ 1.5,  3. ,  2.3,  1.9,  0.5],
       [ 0.1,  0.7,  0.7,  0.4, -0.4],
       [-1. , -0.4, -0.4, -0.6, -1.2],
       [-1.9, -1.3, -1.2, -1.4, -2. ]])
```

### B. Policy Iteration

Let us now consider the Policy Iteration algorithm. Recall that Policy Iteration consists of two alternating sub-routines: 1. Policy Evaluation (E) 2. Policy Improvement (I)

We already wrote the `policy_evaluation()` routine above.

First, we code up the `policy_improvement(pi,V)` rouine: it accepts two arguments: the current policy `pi` which is to be improved, and the corresponding value function approximant `V`$\approx v_\pi$.

In [5]:
```
def policy_improvement(pi,V,gamma):

    stable = True # aux variable for policy iteration

    # loop over all states in the state space
    for m in range(5):
        for n in range(5):

            # back-up old value
            action_old = np.argmax(pi[m,n,:])

            # define a trial policy for all actions
            q_aux = np.zeros(len(env.action_space))
            for action in range(len(env.action_space)):

                # set environment state
                env.state=np.array([m,n])
                # take step
                s_prime, reward, _ = env.step(action)

                # increment V(s)
                q_aux[action] = (reward + gamma*V[s_prime[0],s_prime[1]])

            # policy improvement step
            pi[m,n,:] = 0.0
            action_max=np.argmax(q_aux)
            pi[m,n,action_max] = 1.0

            # aux varible for policy iteration
            if action_old != action_max:
                stable = False


    return pi, stable
```

Now that we have the two building block routines: `policy_evaluation()` and `policy_improvement()`, use them to write down the `policy_iteration()` routine. Follow the pseudocode in Sutton & Barto.

We use as an initial policy the equiprobable policy $\pi(a|s) = \frac{1}{|\mathcal{A}|}$.

In [6]:
```
def policy_iteration(theta,gamma):

    # define initial deterministic policy at random
    pi = 1.0/4.0*np.ones((5,5,4),) # equiprobable/random policy

    j=0
    stable = False
    while not stable:

        # run policy evaluation
```

```
            V = policy_evaluation(pi,theta,gamma)

            # policy improvement
            pi, stable = policy_improvement(pi,V,gamma)


            # print policy
            print("iteration {}: stable = {}".format(j,stable))
            print( np.round(np.flipud(V.T),1) )
            print()

            j+=1

        return V, pi
```

Let us now compute optimal policy for GridWorld using Policy Iteration. In doing so, print the instantaneous value function to monitor its convergence.

In [7]:
```
V, pi = policy_iteration(theta,gamma)
```

```
iteration 0: stable = False
[[ 3.3  8.8  4.4  5.3  1.5]
 [ 1.5  3.   2.3  1.9  0.5]
 [ 0.1  0.7  0.7  0.4 -0.4]
 [-1.  -0.4 -0.4 -0.6 -1.2]
 [-1.9 -1.3 -1.2 -1.4 -2. ]]

iteration 1: stable = False
[[22.  24.4 22.  18.5 16.6]
 [19.8 22.  19.8 16.6 14.9]
 [17.8 19.8 17.8 14.9 13.5]
 [16.  17.8 16.  13.5 12.1]
 [14.4 16.  14.4 12.1 10.9]]

iteration 2: stable = False
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 15.7]
 [17.8 19.8 17.8 16.  14.2]
 [16.  17.8 16.  14.4 12.7]
 [14.4 16.  14.4 13.  11.5]]

iteration 3: stable = False
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
 [16.  17.8 16.  14.4 13. ]
 [14.4 16.  14.4 13.  11.7]]

iteration 4: stable = True
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
 [16.  17.8 16.  14.4 13. ]
 [14.4 16.  14.4 13.  11.7]]
```

Next, visualize the optimal policy $\pi_*$. To do this, you may want to plot slices over the action space.

```
In [8]: print("pi(s,a=0):")
        print(np.flipud(pi[...,0].T))

        print("\npi(s,a=1):")
        print(np.flipud(pi[...,1].T))

        print("\npi(s,a=2):")
        print(np.flipud(pi[...,2].T))

        print("\npi(s,a=3):")
        print(np.flipud(pi[...,3].T))


        print("\n\noptimal policy $pi_\ast$:")
        print(np.flipud(np.argmax(pi,axis=2).T))
```

```
pi(s,a=0):
[[0. 1. 0. 1. 0.]
 [1. 1. 1. 0. 0.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

pi(s,a=1):
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

pi(s,a=2):
[[1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

pi(s,a=3):
[[0. 0. 1. 0. 1.]
 [0. 0. 0. 1. 1.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]


optimal policy $pi_st$:
[[2 0 3 0 3]
 [0 0 0 3 3]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
```

### C. Value Iteration

Last, we also code up the Value Iteration algorithm:

$$v_{k+1}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) \left[ r + \gamma v_k(s) \right], \qquad \lim_{k \to \infty} v_k \to v_*$$

1. Determine the transition probabilities $p(s', r|s, a)$ for the GridWolrd environment.
2. Write the routine `value_iteration(theta, gamma)`, starting from the initial value function `V(s)=1` for all states `s`.

```
In [9]: def value_iteration(theta, gamma):

            # initialize value function V
            V = np.ones((5,5),) # 5x5 grid => 25 states

            # define differece Delta
            Delta=1.0

            # value iteration
            while Delta>=theta:
                Delta=0.0

                # loop over all states in the state space
                for m in range(5):
                    for n in range(5):
                        # back up old value for state
                        v_old = V[m,n]

                        # evaluate update rule
                        v_aux = np.zeros(len(env.action_space))
                        for action in range(len(env.action_space)):

                            # set environment state
                            env.state=np.array([m,n])
                            # take step
                            s_prime, reward, _ = env.step(action)

                            # increment V(s)
                            v_aux[action] = reward + gamma*V[s_prime[0],s_prime[1]]

                        # compute updated value function
                        V[m,n]=np.max(v_aux)
                        # compute Delta
                        Delta = max(Delta, np.abs(V[m,n]-v_old))

            return V
```

Use Value Iteration to compute the value function for GridWorld, and visualize it. Compare it to your previous results from Policy Iteration. Do they agree?

```
In [10]: ### run policy evaluation
         V=value_iteration(theta,gamma)

         # print V
         print( np.round(np.flipud(V.T),1) )
```

```
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
```

```
[16.  17.8 16.  14.4 13. ]
[14.4 16.  14.4 13.  11.7]]
```

Finally, extract the optimal greedy policy $\pi_*(a|s) = \pi_*(s)$ which corresponds to the optimal value function $v_*(s)$, that you computed using Value Iteration.

$$\pi_*(s) = \text{argmax}_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a) \left[ r + \gamma v_*(s) \right]$$

Does the policy make sense?

In [ ]: