Robot Learning

Assignment 2

Solutions are due on 26.04.2022 before the lecture.

Introduction

Consider the following 9×9 grid world:

Grid World

The agent may start in any cell that is not an obstacle nor the goal.

It can choose between eight actions, which correspond to moving to the directions

$$a_i \in \{NW, N, NE, E, SE, S, SW, W\}$$

These are indexed according to the order above, i.e. $a_0 = NW$ and $a_6 = SW$.

The agent must be careful, for the actions are non-deterministic! The agent moves with probability 0.7 into the desired direction, but with probability 0.2 deviates 45° to the left and with probability 0.1 deviates 45° to the right of the desired direction due to treacherous gusts unexpectedly sweeping the grid.

The rewards are structured as follows:

- When it reaches a blue cell, it receives a little snack of 15 points.
- When it attempts to enter a red obstacle cell, it receives -30 points and stays in the cell it came from.
- When it attempts to leave the grid, it receives -30 points and stays in the cell it came from.
- When the agent reaches the green goal cell, it receives 150 points and the episode ends.
- All other actions entering a white cell receive -1 point.

Task 2.1

To familiarize yourself with the environment above, answer the following questions:

- The agent is at $s = (y_s, x_s) = (3, 5)$ and wants to execute a_5. What is the probability $P^a_{s,s'}$ for $s' = (4, 6)$?
- The agent is at $s = (3, 7)$ and wants to execute a_3. What is the expected value of the reward?

1 + 2 = 3 Points

1. $s' = (4, 6)$ is 45° left to the direction of the desired action. So the probability of state change $P^a_{s,s'} = 0.2$

2. Expected reward,

$$R = \sum_{s'} P^a_{s,s'} R^a_{s,s'}$$

$$R = (0.2 \times -1) + (0.7 \times 150) + (0.1 \times -1)$$

$$R = 149.7$$

Task 2.2)

Using the *Iterative Policy Evaluation* Algorithm, compute the value $V^\pi(s)$ of all accessible cells s for a policy $\pi(s,a)$ that chooses with probability 0.5 a random action and otherwise attempts to move to the right.

Intialize $V(s)$ with zero, use a discount parameter of $\gamma = 0.9$ and show your results by printing your state values $V^\pi(s)$.

5 Points

Note

For your convenience, you are provided the helper function *getNextStatesRewardsAndProbabilities(state, action)* which returns for a given state s and an action a a list of 3 -tuples of the form

$$[(s'_0, R^a_{s,s'_0}, P^a_{s,s'_0}), (s'_1, R^a_{s,s'_1}, P^a_{s,s'_1}), \ldots]$$

where s'_i are all future states with $P^a_{s,s'_i} \neq 0$. Here $s = (y,x)$ and $s'_i = (y'_i, x'_i)$ are both tuples of integers, $a \in 0, \ldots, 7$ is an integer, and R^a_{s,s'_i}, P^a_{s,s'_i} are both floats.

Also, please find below some data structures which you might find helpful. Create code and text cells as necessary to present your solution!

In your implementation, $V(s)$ should be a 9×9 numpy array and $\pi(s,a)$ should be a $9 \times 9 \times 8$ numpy array, where $\sum_a \pi(s,a) = 1$ for all s!

In [1]:

```
import numpy as np
from helpers.utils import getNextStatesRewardsAndProbabilities
%matplotlib inline

#this is a list of all states
states = [(y,x) for y in range(9) for x in range(9)]
#this is a list of all states containing obstacles
obstacles = [(1,5), (1,7), (2,1), (2,2), (2,3), (2,4), (2,5), (2,7),\
             (3,1), (3,6), (4,3), (4,4), (4,5), (5,7), \
             (6,1), (6,2), (6,3), (6,4), (6,5), (6,6), \
             (7,8), (8,4), (8,8)]
#this is a list containing all blue cells
snacks = [(0,0), (0,1), (0,2), (0,7), (0,8), (1,8), \
         (3,2), (3,3), (3,4), (3,5), (4,2), \
         (7,1), (7,2), (7,3), (7,4), (7,5), (7,6), \
         (8,1), (8,2), (8,3), (8,5), (8,6)]
#this is a list containing all goal states
terminalStates = [(3,8)]
```

```python
#this is an array containing all actions
actions = np.array([0, 1, 2, 3, 4, 5, 6, 7]) #[NW,      N,      NE,     E,
#example of how to unpack getNextStatesRewardsAndProbabilities(state, action)
#create dummy state and action
s_test = (0,0)
a_test = 3
#call helper function and loop over the return values
for sPrime, R, P in getNextStatesRewardsAndProbabilities(state=s_test, actior
    print('sPrime:', sPrime, 'R:', R, 'P:', P)

#once you have state values V, you can print them with okay'ish formatting li
#print("State Values:")
#print(np.around(V, 1))
```

```
sPrime: (0, 0) R: -30.0 P: 0.2
sPrime: (0, 1) R: 15.0 P: 0.7
sPrime: (1, 1) R: -1.0 P: 0.1
```

In [2]:

```python
np.random.seed(100)
def define_policy(pi):
    '''Defines the policy'''
    for s in states:
        if np.random.random() < 0.5:
            # Picking random action
            pi[s][np.random.choice(actions)]=1
        else:
            # a_3 to move right side of the grid
            pi[s][3]=1
    return pi
```

In [3]:

```python
# Initializing V(s) with zeros
V = np.zeros((9,9))

# Initializing and defining Policy
pi = np.zeros((9,9,8))
define_policy(pi)

# Discount factor
gamma = 0.9
# Small positive integer
theta = 1e-8

# Iterative Policy Evaluation algorithm
while True:
    delta = 0
    for state in states:
        v = V[state]
        update_value = 0
        for action in actions:
            for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,ac
                update_value += pi[state][action] * P * ( R + gamma*V[sPrime]
        V[state] = update_value
        delta = max(delta,abs(v-V[state]))
    if delta < theta:
        break

# Result
print("State Values:")
print(np.around(V, 1))
```

```
State Values:
[[ -56.8  -56.7 -238.7 -262.  -267.7 -263.9 -256.1 -164.5 -180.7]
```

```
[-128.8 -168.7 -239.5 -264.9 -300.  -173.5 -180.7 -216.6 -300. ]
[-300.  -151.3 -219.5 -239.1 -188.  -173.5 -192.8 -167.5  -94.2]
[-214.3 -102.3 -122.1 -153.5 -163.9 -192.3 -115.4 -105.9 -264.5]
[-141.5 -151.4 -144.3 -166.1 -182.2 -168.1 -168.2 -234.4 -300. ]
[-141.  -154.3 -180.5 -146.3 -154.2 -163.2 -234.1 -259.2 -300. ]
[-115.1 -158.2 -141.  -148.  -164.8 -152.7 -233.2 -257.  -279.3]
[  54.6   54.3  -70.3  -68.1  -81.3 -132.  -244.3 -250.1 -300. ]
[  60.4   50.8 -300.  -230.9 -134.1 -172.7 -121.2 -300.  -300. ]]
```

Task 2.3)

Now it is time to find a good policy. Use the *Policy Iteration* algorithm to compute the optimal value $V^*(s)$ for each accessible cell.

Retrieve the resulting optimal-policy $\pi^*(s)$. To obtain a greedy policy given $V(s)$, make use of:

$$\pi_{greedy}(s) := \text{argmax}_a\, Q(s,a) = \text{argmax}_a \sum_{s'} P^a_{ss'} \cdot [R^a_{ss'} + \gamma \cdot V(s')]$$

As implied by these terms, we recommend using intermediate state-action Q-values, shaped $9 \times 9 \times 8$ for this step!

Finally, present your results by printing $V^*(s)$ and using our helper function *drawPolicy()* to visualize $\pi^*(s,a)$.

5 Points

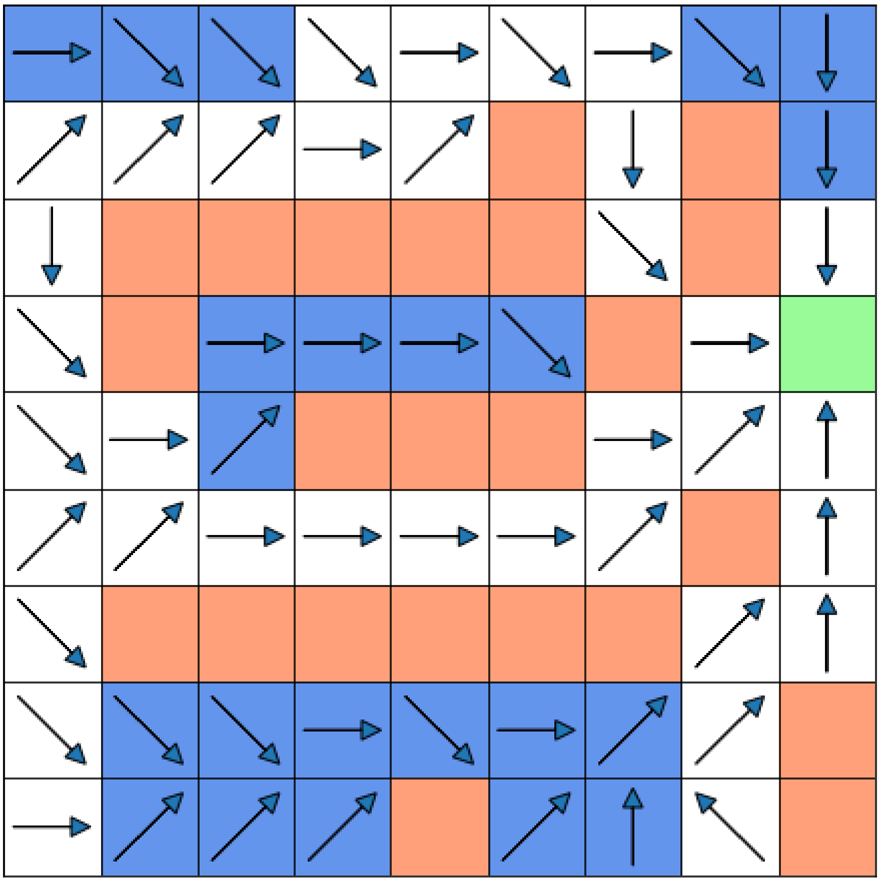
In [4]:

```python
from helpers.utils import drawPolicy
#show policy using helper function as below
#usage of the helper function, where pi is a (9,9,8) numpy array representing
#drawPolicy(pi)
#deterministic here means that one action per state has probability 1 and all
#this will plot arrows representing your policies into the grid world.
```

In [14]:

```python
# Initializing V(s) with zeros
V = np.zeros((9,9))

# Initializing and defining Policy
pi = np.zeros((9,9,8))
define_policy(pi)

# Initilizing Q(s,a) with zeros
Q = np.zeros((9,9,8))

# Discount factor
gamma = 0.9
# Small positive integer
theta = 1e-8
# Flag
policy_stable = False

# Policy Iteration algorithm
while policy_stable is False :
    # Policy Evaluation
    while True:
        delta = 0
        for state in states:
            v = V[state]
            update_value = 0
```

```
            action = np.argmax(pi[state])
            for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,ac
                update_value += P * ( R + gamma*V[sPrime])
            V[state] = update_value
            delta = max(delta,abs(v-V[state]))
        if delta < theta:
            break
    # Policy Improvement
    policy_stable = True
    for state in states:
        for action in actions:
            for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,ac
                Q[state][action] += P * ( R + gamma*V[sPrime])
        b = np.argmax(pi[state])
        if b != np.argmax(Q[state]):
            pi[state][np.argmax(pi[state])] = 0
            pi[state][np.argmax(Q[state])]  = 1
            policy_stable = False

# Result
drawPolicy(pi)
print("State Values:")
print(np.around(V, 1))
```



```
State Values:
[[207.1 228.8 252.3 286.8 318.8 384.8 405.3 467.3 457.4]
 [216.1 236.4 253.9 283.9 325.2 452.6 446.2 537.5 527.6]
 [198.2 267.1 312.6 358.9 412.6 452.6 532.4 632.5 626.7]
 [245.1 294.7 307.1 353.4 407.1 469.4 562.2 632.4 568.4]
 [255.9 293.2 317.  363.3 407.5 486.4 559.3 633.4 630.5]
 [255.7 287.5 276.1 335.4 404.  483.5 544.8 560.4 557.5]
 [234.2 251.5 282.5 341.8 410.4 468.6 468.6 471.4 467.8]
 [248.8 262.3 280.4 289.4 333.  354.1 399.2 412.8 412.8]
 [245.9 263.6 274.8 299.3 330.7 363.1 365.8 367.9 354.3]]
```

Task 2.4)

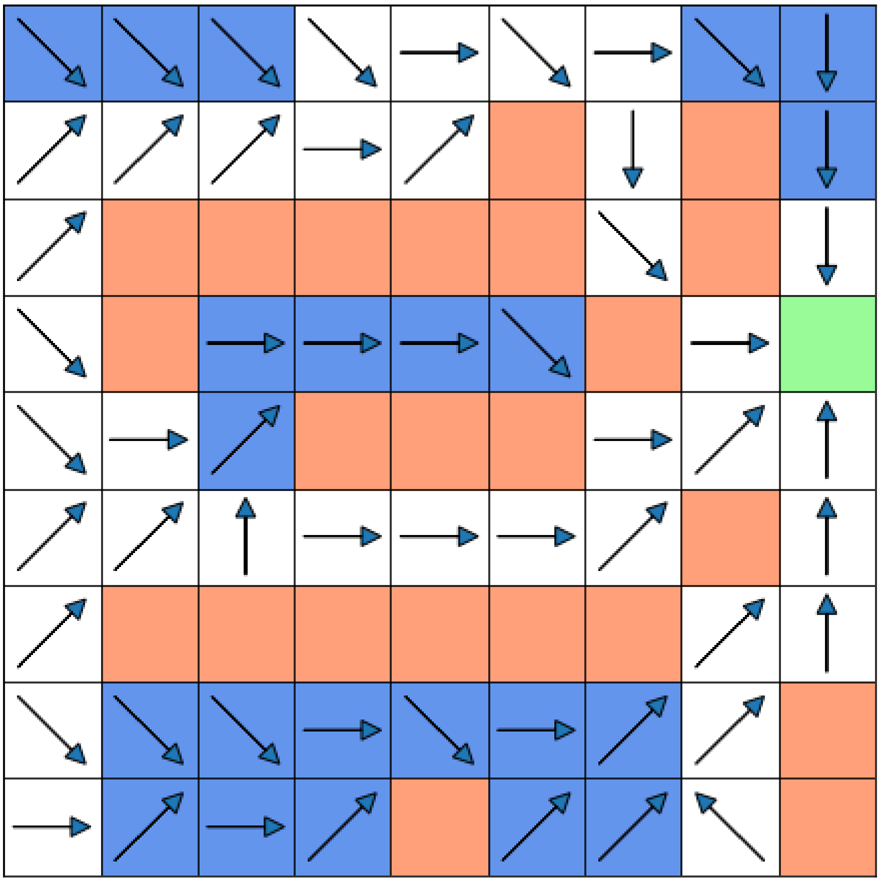
Verify your results from the previous task by using the *Value Iteration* algorithm to compute the optimal value $V^*(s)$ for each cell. Make sure to reinitialize $V(s)$ with zero.

Finally, present your results by printing $V^*(s)$ and using our helper function *drawPolicy()* to visualize $\pi^*(s, a)$.

4 Points

In [6]:

```
# Initializing V(s) with zeros
V = np.zeros((9,9))

# Initializing and defining Policy
pi = np.zeros((9,9,8))
define_policy(pi)

# Initilizing Q(s,a) with zeros
Q = np.zeros((9,9,8))

# Discount factor
gamma = 0.9
# Small positive integer
theta = 1e-8

# Value Iteration algorithm
# Truncated Policy Evaluation
while True:
    delta = 0
    for state in states:
        v = V[state]
        update_values = list()
        for action in actions:
            update_value = 0
            for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,ac
                update_value += P * ( R + gamma*V[sPrime])
            update_values.append(update_value)
        V[state] = max(update_values)
        delta = max(delta,abs(v-V[state]))
    if delta < theta:
        break
# Policy Improvement
for state in states:
    for action in actions:
        for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,action
            Q[state][action] += P * ( R + gamma*V[sPrime])
    pi[state][np.argmax(pi[state])] = 0
    pi[state][np.argmax(Q[state])]  = 1

# Result
drawPolicy(pi)
print("State Values:")
print(np.around(V, 1))
```



```
State Values:
[[211.8 228.8 252.3 286.8 318.8 384.8 405.3 467.3 457.4]
 [217.  236.4 253.9 283.9 325.2 452.6 446.2 537.5 527.6]
 [202.3 267.1 312.6 358.9 412.6 452.6 532.4 632.5 626.7]
 [245.8 294.8 307.1 353.4 407.1 469.4 562.2 632.4 568.4]
 [256.8 294.1 317.  363.3 417.  486.4 559.3 633.4 630.5]
 [256.4 288.5 285.6 335.4 404.  483.5 544.8 560.4 557.5]
 [246.2 252.6 284.4 341.8 410.4 468.6 468.6 471.4 467.8]
 [249.3 263.2 280.6 289.6 333.1 354.3 399.2 412.8 412.8]
 [246.4 264.  276.1 299.4 333.1 363.3 367.3 368.1 354.3]]
```

Task 2.5)

Modify your implementation of *Value Iteration* or *Policy Iteration* to ignore the random deviations in the environment. This can be achieved by calling *getNextStatesRewardsAndProbabilities(state, action, deviation=False)*.

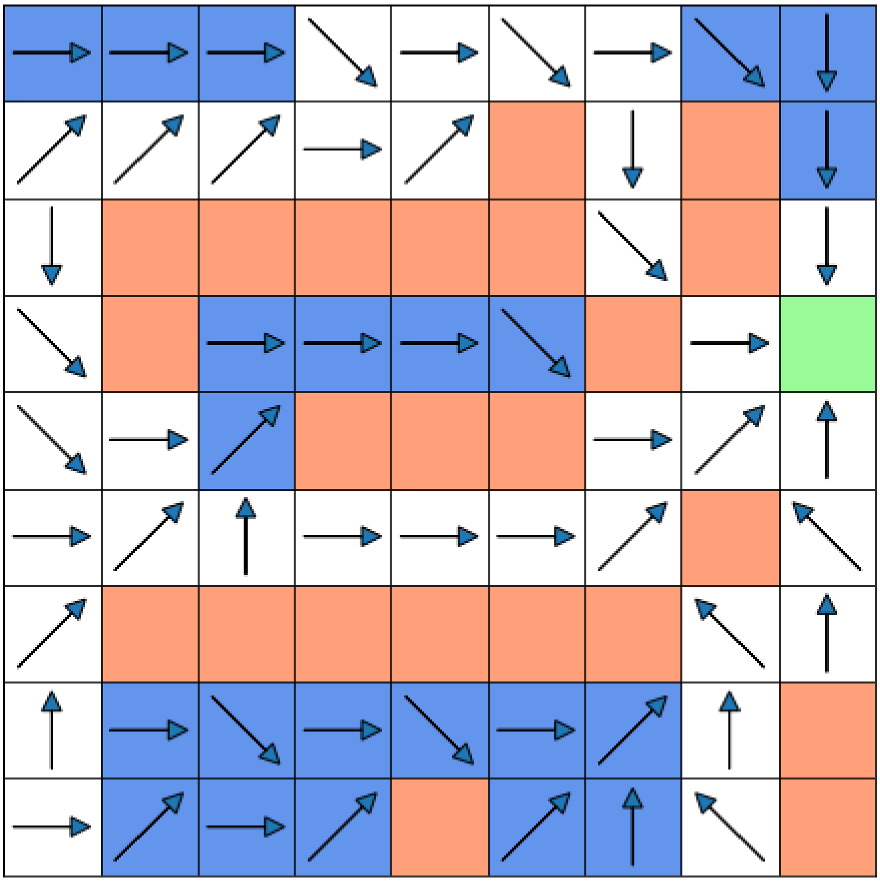
Present your results by printing $V^*(s)$ and using our helper function *drawPolicy()* to visualize $\pi^*(s,a)$. How and why have your state values and policy changed?

3 Points

In [7]:

```
# Initializing V(s) with zeros
V = np.zeros((9,9))

# Initializing and defining Policy
pi = np.zeros((9,9,8))
define_policy(pi)

# Initilizing Q(s,a) with zeros
Q = np.zeros((9,9,8))

# Discount factor
```

```
gamma = 0.9
# Small positive integer
theta = 1e-8

# Value Iteration algorithm
# Truncated Policy Evaluation
while True:
    delta = 0
    for state in states:
        v = V[state]
        update_values = list()
        for action in actions:
            update_value = 0
            for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,ac
                update_value += P * ( R + gamma*V[sPrime])
            update_values.append(update_value)
        V[state] = max(update_values)
        delta = max(delta,abs(v-V[state]))
    if delta < theta:
        break
# Policy Improvement
for state in states:
    for action in actions:
        for sPrime, R, P in getNextStatesRewardsAndProbabilities(state,actior
            Q[state][action] += P * ( R + gamma*V[sPrime])
    pi[state][np.argmax(pi[state])] = 0
    pi[state][np.argmax(Q[state])]  = 1


# Result
drawPolicy(pi)
print("State Values:")
print(np.around(V, 1))
```



```
State Values:
[[362.5 386.1 412.4 459.3 511.4 569.4 599.8 649.7 649.7]
```

```
 [362.5 386.1 412.4 459.3 511.4 633.7 633.7 705.3 705.3]
 [376.7 467.4 502.6 541.8 585.4 633.7 705.3 784.7 784.7]
 [419.6 467.4 502.6 541.8 585.4 633.7 705.3 784.7 705.3]
 [419.6 467.4 502.6 541.8 585.4 633.7 705.3 784.7 784.7]
 [419.6 467.4 467.4 511.4 569.4 633.7 705.3 705.3 705.3]
 [419.6 419.6 459.3 511.4 569.4 633.7 633.7 633.7 633.7]
 [376.7 397.6 425.1 455.7 489.7 527.4 569.4 569.4 569.4]
 [372.9 397.6 425.1 455.7 489.7 527.4 527.4 527.4 511.4]]
```

Policy change

- The optimal policy in task 2.4 chooses to move away from obstacle or grid boundries. but the policy in the task 2.5 doesn't follow this.
- Lets take the example of the state $s = (7, 0)$, the optimal policy with deviation chooses a_4 because its move away from the boundary of the grid.The optimal policy without deviation chooses a_1 to consume more rewards.

State value change

- In task 2.3, the state value for states at the gird boundary, are far less because of the probability of deviating outside of the grid. Lets take the example of $s = (0, 0)$ taking action a_3, Its expected reward with deviation is only 4.4 but without deviation is 15. Which is same as of the state $s = (1, 0)$ taking action a_2.

In [8]:
```
expected_reward = 0
for sPrime, R, P in getNextStatesRewardsAndProbabilities(state=(0,0),action=3
    print('sPrime:', sPrime, 'R:', R, 'P:', P)
    expected_reward += P * R
expected_reward
```

```
sPrime: (0, 0) R: -30.0 P: 0.2
sPrime: (0, 1) R: 15.0 P: 0.7
sPrime: (1, 1) R: -1.0 P: 0.1
```

Out[8]:
```
4.4
```

In [9]:
```
expected_reward = 0
for sPrime, R, P in getNextStatesRewardsAndProbabilities(state=(0,0),action=3
    print('sPrime:', sPrime, 'R:', R, 'P:', P)
    expected_reward += P * R
expected_reward
```

```
sPrime: (0, 1) R: 15.0 P: 1.0
```

Out[9]:
```
15.0
```

In [10]:
```
expected_reward = 0
for sPrime, R, P in getNextStatesRewardsAndProbabilities(state=(1,0),action=2
    print('sPrime:', sPrime, 'R:', R, 'P:', P)
    expected_reward += P * R
expected_reward
```

```
sPrime: (0, 0) R: 15.0 P: 0.2
sPrime: (0, 1) R: 15.0 P: 0.7
sPrime: (1, 1) R: -1.0 P: 0.1
```

Out[10]:
```
13.4
```

In [11]:
```
expected_reward = 0
for sPrime, R, P in getNextStatesRewardsAndProbabilities(state=(1,0),action=2
```

```python
    print('sPrime:', sPrime, 'R:', R, 'P:', P)
    expected_reward += P * R
expected_reward
```

```
sPrime: (0, 1) R: 15.0 P: 1.0
```

Out[11]:

```
15.0
```