Deep Learning Using TensorFlow

Dr. Ash Pahwa

Lesson 8:

RNN + Reinforcement Learning

Lesson 8.3: Reinforcement Learning

Outline

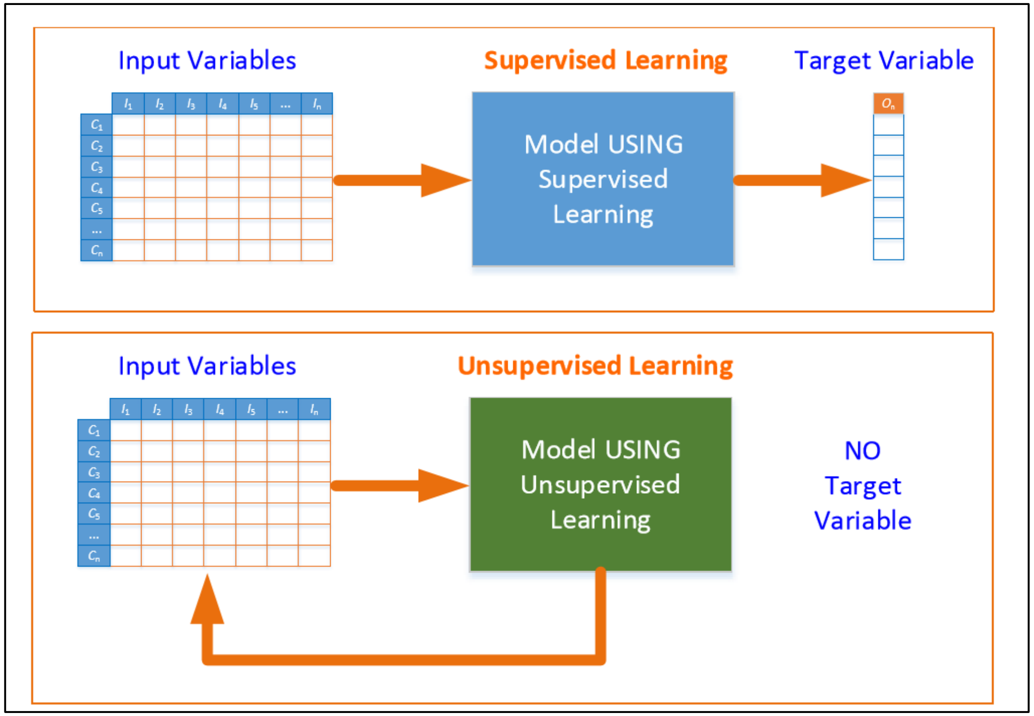
- What is Reinforcement Learning
- Markov Decision Process
- Q Learning
- Implementation of Q Learning in Python

Types of Learning

- Supervised Learning
  - Has a response variable
  - Regression, Neural Networks, kNN, SVM etc.
- Unsupervised Learning
  - Does not have a response variable
  - Clustering, PCA
- Reinforcement Learning

Supervised vs. Unsupervised Learning in Machine Learning

- Supervisor learning is the most common learning type where there is a target/output variable (which is also called supervisor)
  - Supervisor (target variable) teaches the algorithm how to build/learn the pattern model
  - In PA, supervised learning ≈ predictive modeling
- Unsupervised learning has NO target variable
  - No supervisor to teach → algorithm has to learn by itself
  - In PA, unsupervised learning ≈ descriptive modeling



Difference between
Supervised + Unsupervised and Reinforcement Learning

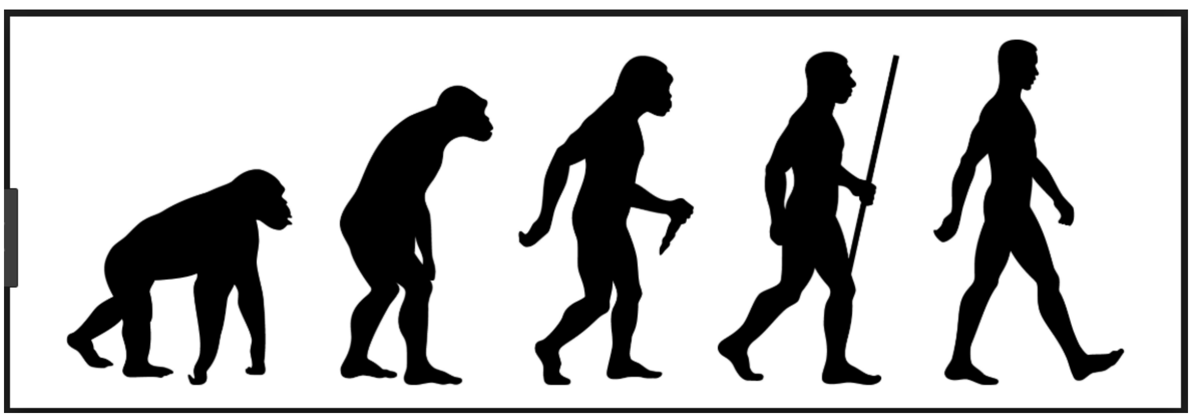
- Supervised + Unsupervised Learning
  - You need data
- Reinforcement Learning
  - No data is needed
  - Works on a reward system

Reinforcement Learning

- Learning method used by humans
- Learning is based on continuous experience
- Reward/Punishment received from the environment are used to guide the learning process

Biological (Human) Intelligence

- Biological (Human) intelligence was developed via the evolution process from millions of years of Reinforcement Learning
- Reinforcement learning
  - Intelligence was developed by Trial and Error interaction with the environment

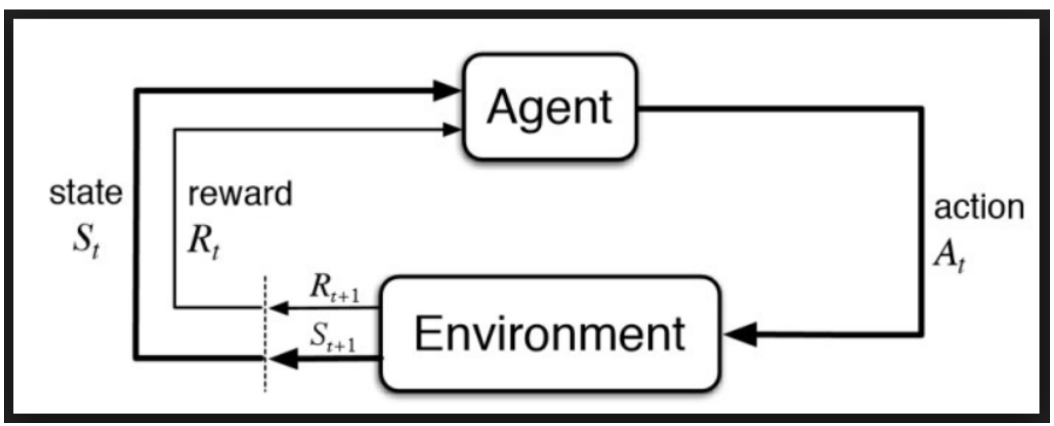


Humans

- Human characteristics
  - Reward driven entities
- Inspiration for Reinforcement learning comes from the success of human **greed**

Definition: Reinforcement Learning

- A reward-driven trial-and-error process,
  - in which a system learns to interact with a complex environment
  - to achieve rewarding outcomes,
  - is referred as reinforcement learning.



Primary Applications of Reinforcement Learning

- Robotics
- Autonomous car driving
- Video games: Atari 2600
- AlphaGo game

Markov Decision Process

Andrey Markov

Andrey Markov

Russian mathematician

Andrey Andreyevich Markov was a Russian mathematician best known for his work on stochastic processes. A primary subject of his research later became known as Markov chains and Markov processes. Markov and his younger brother Vladimir Andreevich Markov proved the Markov brothers' inequality. Wikipedia

Born: June 14, 1856, Ryazan, Russia

Died: July 20, 1922, Saint Petersburg, Russia

Nationality: Russian

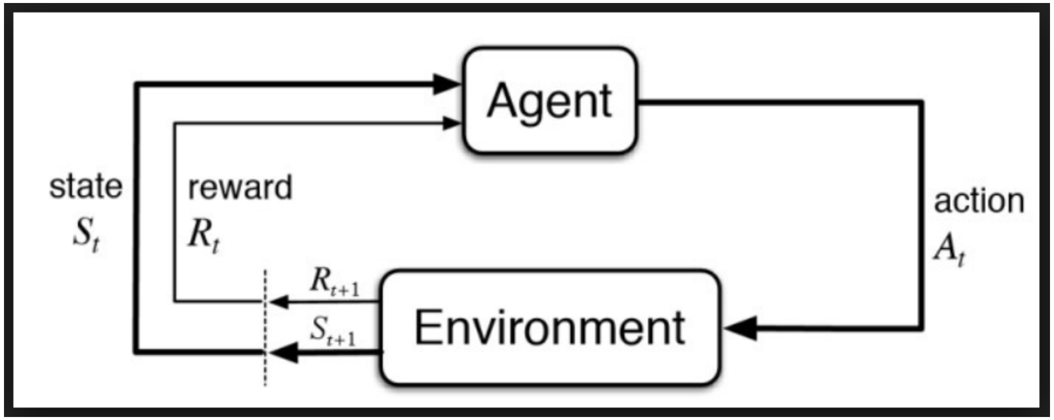
Doctoral advisor: Pafnuty Chebyshev

Markov Decision Process (MDP)

- Markov property generally means that
  - Given the present state
  - The future and the past are independent
- $P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1} \ldots S = s_0) =$
- $P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$
- Action outcomes depend only on the current state

Markov Decision Process (MDP)

- Agent = Robot
- Environment = Game



Markov Decision Process (MDP)

- MDP is defined as
  - Set of States S
    - All possible configurations of the game
  - Set of Actions
    - Left, Right, Up, Down
  - Transition Function
    - T(s,a,s’) likelihood of being in state ‘s’
    - Taking an action ‘a’
    - Ending up in state s’
  - Reward function R
    - Reward of being in state ‘s’
    - Taking action ‘a’
    - Ending up in state s’

- MDP is defined as
  - A set of States $s \in S$
  - A set of actions $a \in A$
  - A Transition Function
    - $T(s,a,s') = P(s'|s,a)$
  - A Reward function
    - $R(s,a,s') = R(s) = R(s')$
  - A start state
- MDP are non-deterministic search problem

Game: Set of States

(3,1)	(3,2)	(3,3)	(3,4) +1
(2,1)	(2,2) Wall	(2,3)	(2,4) -1
(1,1) Robot	(1,2)	(1,3)	(1,4)

Set of states
- 1,1
- 1,2
- 1,3
- 1,4
- 2,1
- 2,2 (Not Valid)
- 2,3
- 2,4
- 3,1
- 3,2
- 3,3
- 3,4

Goal is to get maximum award

Develop an optimal policy: $\pi^*: S \rightarrow A$

Actions

- Actions a robot can take
  - Up
  - Down
  - Left
  - Right
  - Stay at the same place

Transition Function
Motions are Non Deterministic

- T(s,a,s')
  - Transition Function
  - Probability that
    - Robot in state 's'
    - Action 'a' is taken
    - Robot lands in state s'
- Transition function for some states
  - Up = 70%
  - Left = 10%
  - Right = 10%
  - Down = 10%

Reward Function

(3,1)	(3,2)	(3,3)	(3,4) +1
(2,1)	(2,2) Wall	(2,3)	(2,4) -1
(1,1) Robot	(1,2)	(1,3)	(1,4)

- R(s,a,s')
  - Reward Function
  - Reward that
    - Robot in state 's'
    - Action 'a' is taken
    - Robot lands in state s'
    - With a reward
  - R((3,3),Right,(3,4)) = 1
  - R((2,3),Right,(2,4)) = -1

Discount Function γ

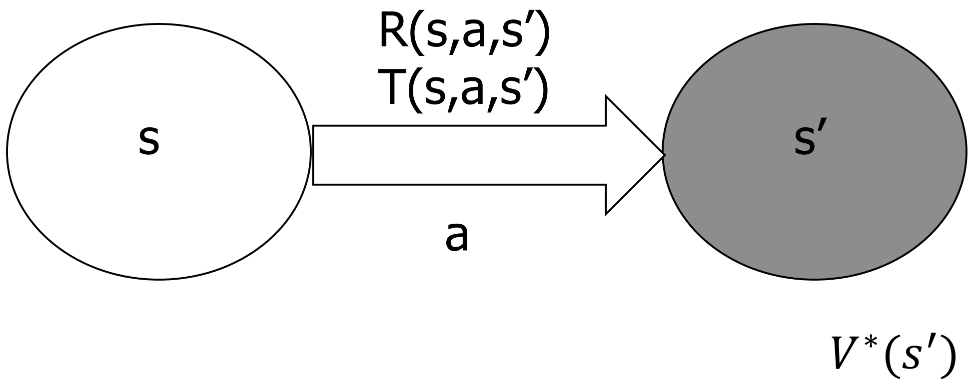
- Prefer rewards sooner than later
- Rewards value will depreciate with time
- Example:
  - Reward Discount factor = $\gamma = 0.90$
  - Reward value as a function of time
    - Time 0: Reward value = $100
    - Time 1: Reward value =$100*0.9 = $90
    - Time 2: Reward value = $90*0.9 = $81

How to Solve MDP? Value Iteration

- Also called Bellman Equation
- For each state ‘s’ compute the expected reward
  - Starting from ‘s’ and acting optimally
- Value function near-high reward states will be large
- Discounting factor decreases overall value function
- Value Function
  - $V^*(s) = max_a \sum T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

Value Iteration

- $V^*(s) = max_a \sum T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$



Value Iteration

- $V^*(s) = max_a \sum T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

(3,1)	(3,2)	(3,3)	(3,4) +1
(2,1)	(2,2) Wall	(2,3)	(2,4) -1
(1,1) Robot	(1,2)	(1,3)	(1,4)

R(s,a,s')
T(s,a,s')
→
a

(3,1)	(3,2)	(3,3)	(3,4) +1
(2,1)	(2,2) Wall	(2,3)	(2,4) -1
(1,1)	(1,2) Robot	(1,3)	(1,4)

Value Iteration Algorithm

- $V^*(s) = max_a \sum T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

- Initialize all V*(s) = 0
- While NOT Converged
  - For each state compute V*(s)

Values Computed

After 0 Iteration

0.00 ↑	0.00 ↑	0.00 ↑	0.00
0.00 ↑	Wall	0.00 ↑	0.00
0.00 ↑	0.00 ↑	0.00 ↑	0.00 ↑

After 1 Iteration

0.00 ↑	0.00 ↑	0.00 ↑	1.00
0.00 ↑	Wall	0.00 ↑	-1.00
0.00 ↑	0.00 ↑	0.00 ↑	0.00 ↑

After 2 Iteration

0.00 ↑	0.00 ↑	0.72 →	1.00
0.00 ↑	Wall	0.00 ↑	-1.00
0.00 ↑	0.00 ↑	0.00 ↑	0.00 ↑

After 100 Iteration

0.64 →	0.74 →	0.85 →	1.00
0.57 ↑	Wall	0.57 ↑	-1.00
0.49 ↑	0.43 ←	0.48 ↑	0.28 ←

Python 'argmax' function

- Gives the index of the maximum number

```
# Example of argmax function
import numpy as np

b = np.arange(6)
b
Out[16]: array([0, 1, 2, 3, 4, 5])

b[1] = 5
b
Out[18]: array([0, 5, 2, 3, 4, 5])

np.argmax(b)  # Only the first occurrence is returned.
Out[19]: 1
############################################
a = np.arange(6).reshape(2,3) + 10
a
Out[22]:
array([[10, 11, 12],
       [13, 14, 15]])

np.argmax(a)
Out[23]: 5
np.argmax(a, axis=0)
Out[24]: array([1, 1, 1], dtype=int64)
np.argmax(a, axis=1)
Out[25]: array([2, 2], dtype=int64)
```

Policy Extraction

- After value iteration, we need a policy
  - Represented by Greek character π
- What is a policy (π)?
  - Means: we are in state 's', which action should we take
  - We should take an action which gives maximum rewards
- Same as Bellman Equation
  - Except replace 'max' with 'argmax'

- $V^*(s) = max_a \sum T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
- $\pi^*(s) = argmax_a \sum T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$

Q Learning

Q Learning

- It is based on Q table (or function)
- Q[s,a]
  - Immediate rewards + Discounted rewards
- Suppose we know the Q table
  - Compute the policy for state ‘s’ which means
  - We are state ‘s’, which action should we take

- We should take the action which gives maximum rewards
  - $\pi(s) = argmax_a(Q[s,a])$
  - It means for all possible actions in state ‘s’
    - Take the action which gives maximum rewards
    - Q table contains rewards for all actions for all states

Q Learning

- After the system has converged
  - $Optimum\ policy: \pi(s) \Rightarrow \pi^*(s)$
  - $Q[s,a] \Rightarrow Q^*[s,a]$

How to Update Q Table

- Iteratively the Q table is modified
- Suppose the agent (robot) goes
  - From state 's'
  - with action 'a'
  - To state s'
  - Get the reward r
- $Q'[s,a] = (1-\alpha)Q[s,a] + \alpha * improved\ estimate$
- Improved estimate = $immediate\ reward + \gamma * later\ rewards$
  - $immediate\ reward = r$
  - $later\ reward = Q[s', argmax(Q[s', a'])]$

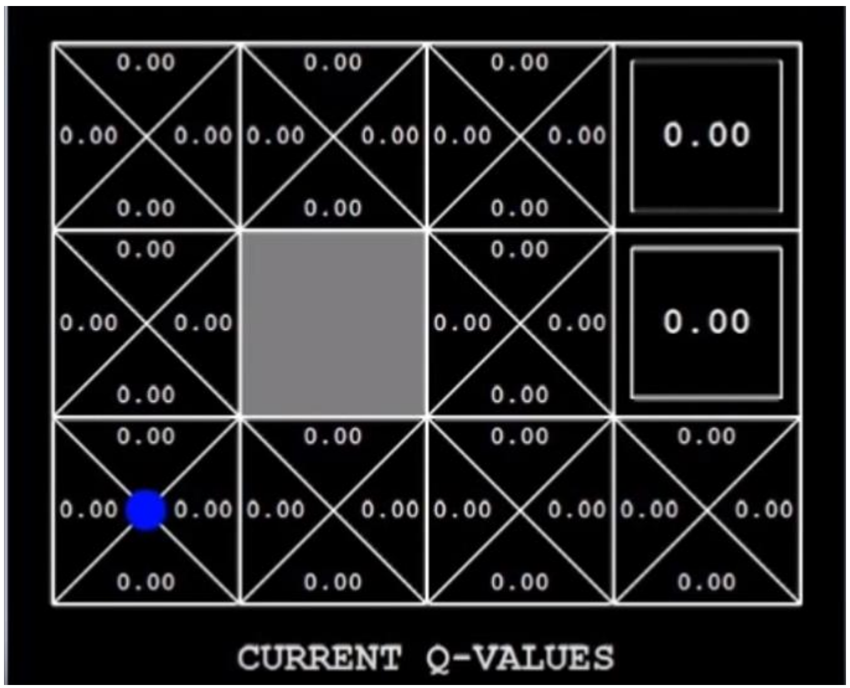
$\alpha = learning\ rate\ (0-1)$
$\gamma = discount\ rate\ (0-1)$

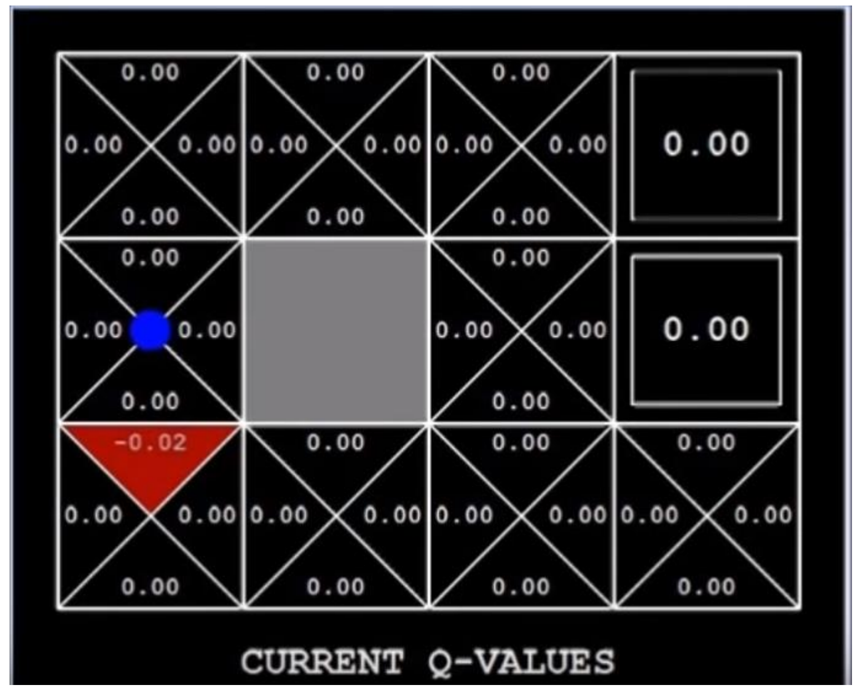
Q[s,a] Table
Change with Time

3	6	9	12
2	5	8	11
1	4	7	10

- Cost of each action = -0.04
- Alpha = Learning rate = α = 0.5
- Gamma = Discount rate = γ = 1

Starting State = 1





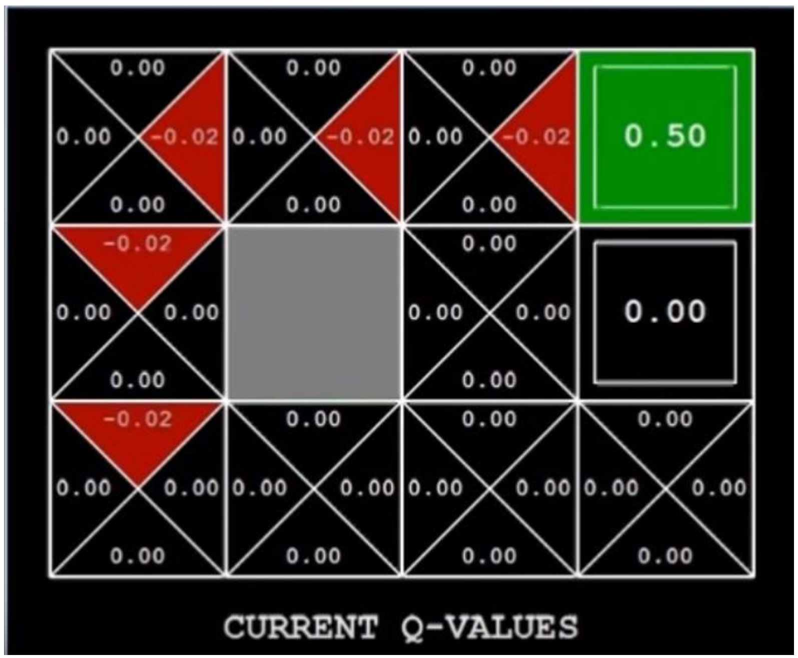
Step: from 1 to 2

Q[s,a] Table

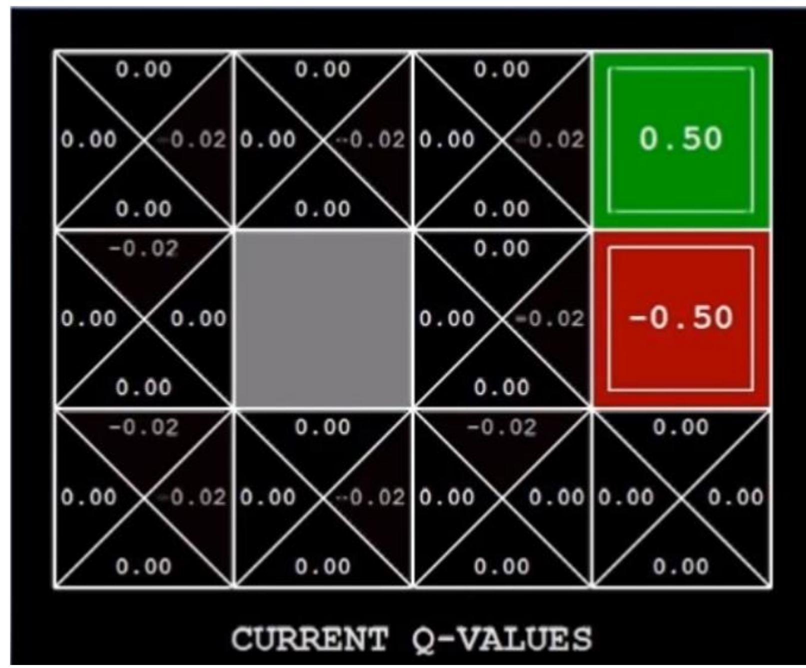
- Cost of each action = -0.04
- Alpha = Learning rate = α = 0.5
- Gamma = Discount rate = γ = 1

3	6	9	12
2	5	8	11
1	4	7	10

Starting State = 1



Step: from 1 → 2,3,6,9,12



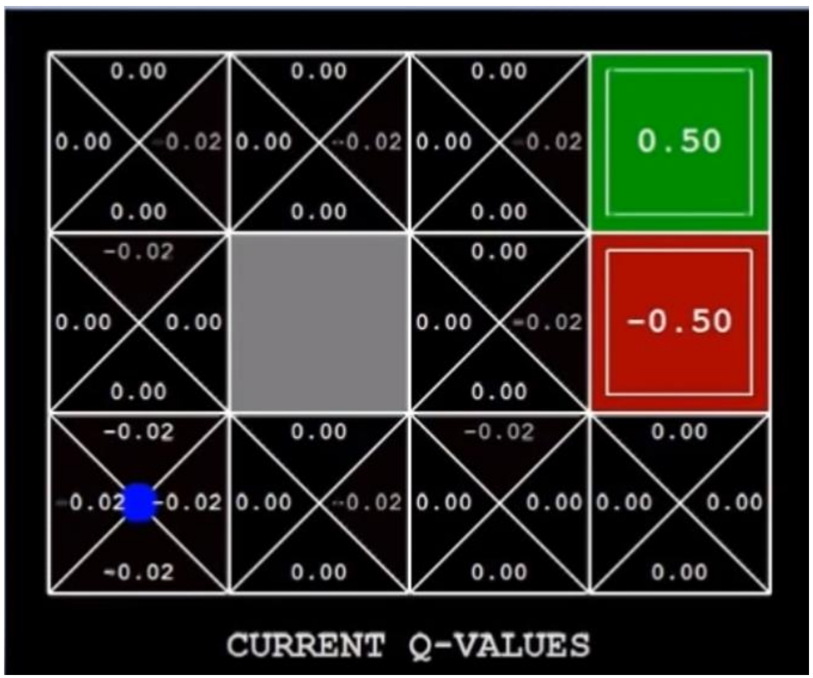
Step: from 1 → 4,7,8,11

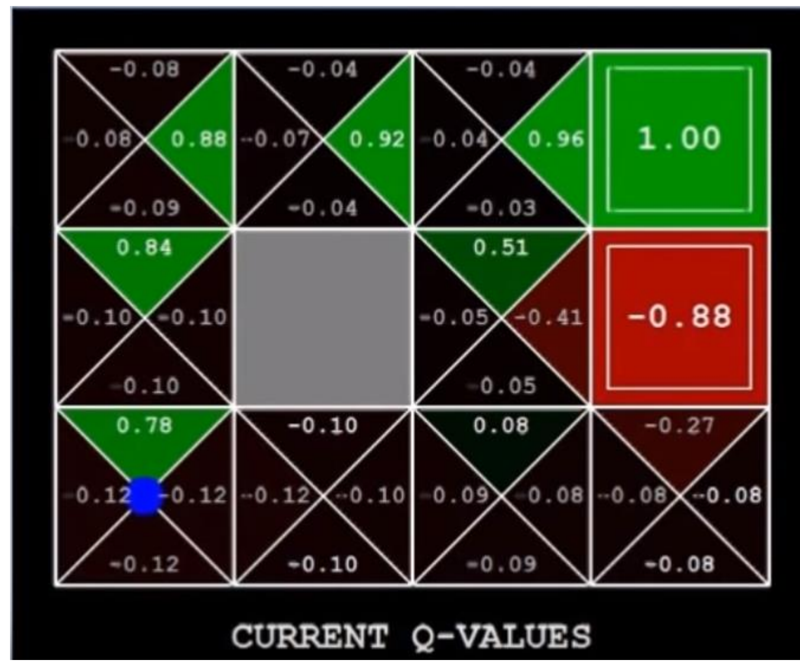
Q[s,a] Table

3	6	9	12
2	5	8	11
1	4	7	10

- Cost of each action = -0.04
- Alpha = Learning rate = α = 0.5
- Gamma = Discount rate = γ = 1

Starting State = 1





Step: from 1 → Left
Step: from 1 to Down

Step: from 1 → 2,3,6,9,12

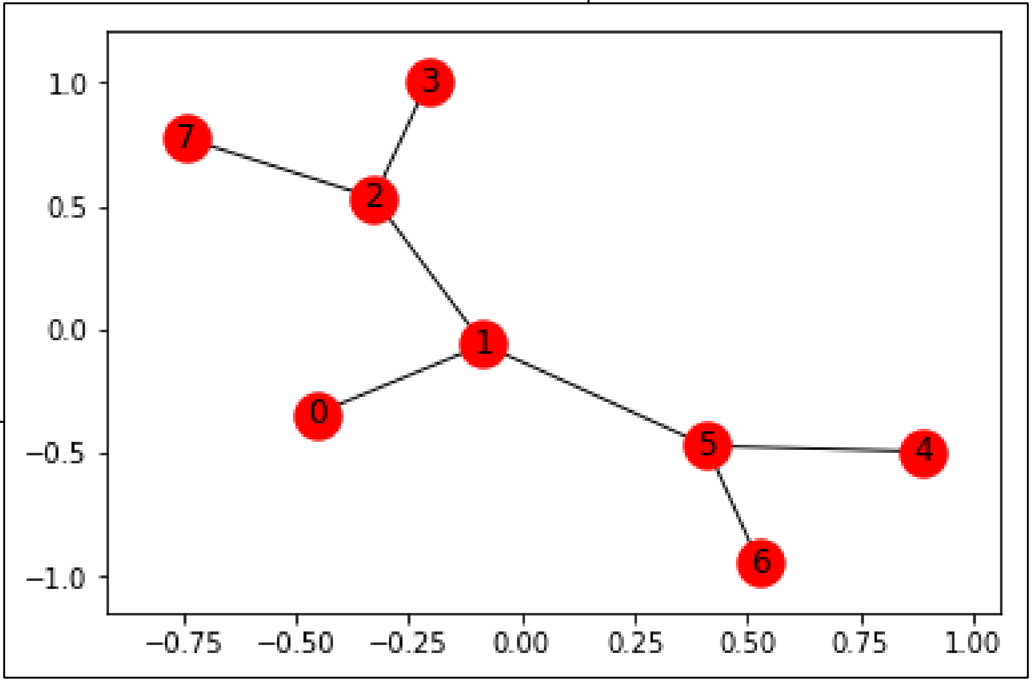
Implementation of Q-Learning in Python

Load the Libraries

```
import numpy as np
import pylab as plt
```

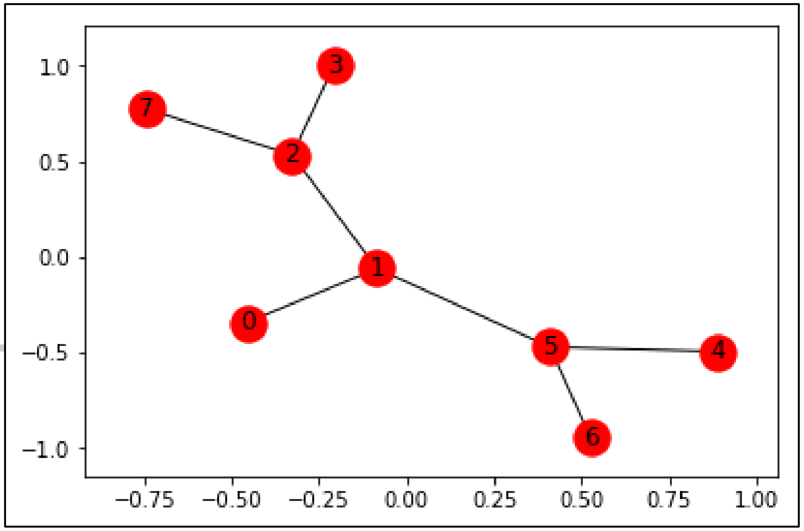
Create the Map

```
points_list = [(0,1), (1,5), (5,6), (5,4), (1,2), (2,3), (2,7)]


goal = 7

import networkx as nx
G=nx.Graph()
G.add_edges_from(points_list)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G,pos)
nx.draw_networkx_edges(G,pos)
nx.draw_networkx_labels(G,pos)
plt.show()
```



Reward Matrix

Cost to move = -1

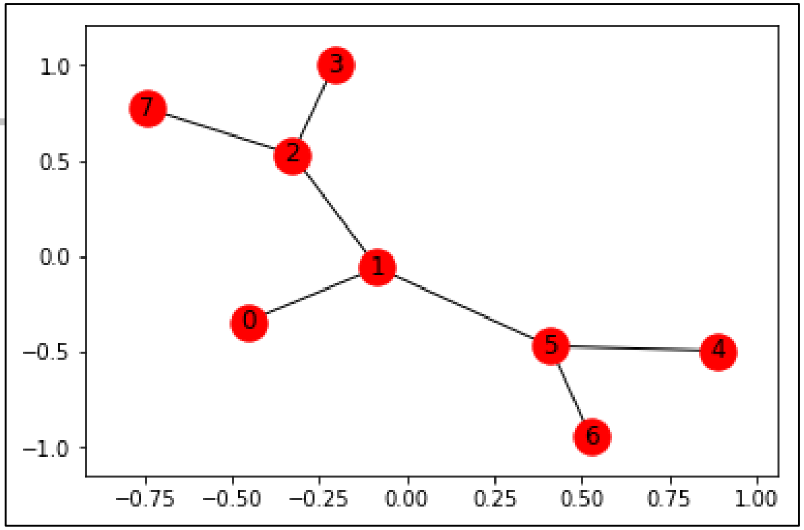



```
points_list = [(0,1), (1,5), (5,6), (5,4), (1,2), (2,3), (2,7)]
```

```
# how many points in graph? x points
MATRIX_SIZE = 8
# create matrix x*y

R = np.matrix(np.ones(shape=(MATRIX_SIZE, MATRIX_SIZE)))
R *= -1
R
Out[10]:
matrix([[-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1., -1.]])
```

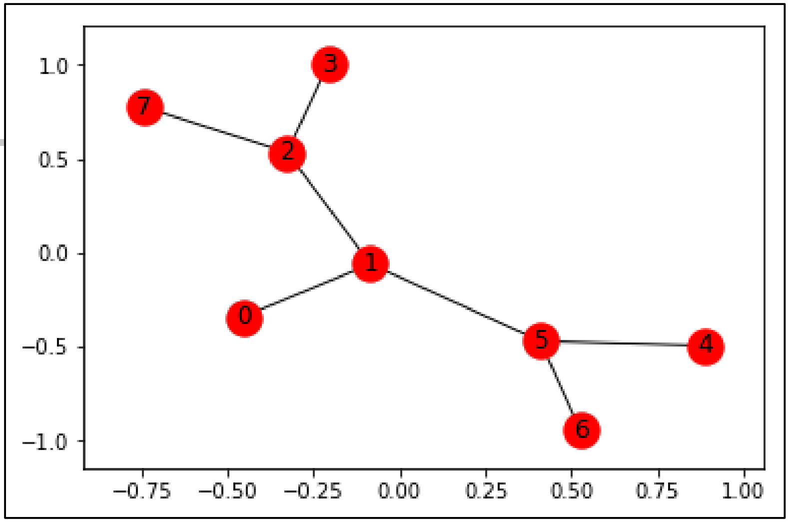
Populate the Reward Matrix

```
for point in points_list:
    print(point)
    if point[1] == goal:
        R[point] = 100
    else:
        R[point] = 0

    if point[0] == goal:
        R[point[::-1]] = 100
    else:
        # reverse of point
        R[point[::-1]]= 0


(0, 1)
(1, 5)
(5, 6)
(5, 4)
(1, 2)
(2, 3)
(2, 7)
```




```
R
Out[12]:
matrix([[  -1.,    0.,   -1.,   -1.,   -1.,   -1.,   -1.,   -1.],
        [   0.,   -1.,    0.,   -1.,   -1.,    0.,   -1.,   -1.],
        [  -1.,    0.,   -1.,    0.,   -1.,   -1.,   -1.,  100.],
        [  -1.,   -1.,    0.,   -1.,   -1.,   -1.,   -1.,   -1.],
        [  -1.,   -1.,   -1.,   -1.,   -1.,    0.,   -1.,   -1.],
        [  -1.,    0.,   -1.,   -1.,    0.,   -1.,    0.,   -1.],
        [  -1.,   -1.,   -1.,   -1.,   -1.,    0.,   -1.,   -1.],
        [  -1.,   -1.,    0.,   -1.,   -1.,   -1.,   -1.,   -1.]])
```

Populate the Reward Matrix




```
R[goal,goal]= 100

R
Out[14]:
matrix([[  -1.,    0.,   -1.,   -1.,   -1.,   -1.,   -1.,   -1.],
        [   0.,   -1.,    0.,   -1.,   -1.,    0.,   -1.,   -1.],
        [  -1.,    0.,   -1.,    0.,   -1.,   -1.,   -1.,  100.],
        [  -1.,   -1.,    0.,   -1.,   -1.,   -1.,   -1.,   -1.],
        [  -1.,   -1.,   -1.,   -1.,   -1.,    0.,   -1.,   -1.],
        [  -1.,    0.,   -1.,   -1.,    0.,   -1.,    0.,   -1.],
        [  -1.,   -1.,   -1.,   -1.,   -1.,    0.,   -1.,   -1.],
        [  -1.,   -1.,    0.,   -1.,   -1.,   -1.,   -1.,  100.]])
```

Build the Q Matrix

```
Q = np.matrix(np.zeros([MATRIX_SIZE,MATRIX_SIZE]))

Q
Out[16]:
matrix([[ 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.,  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.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

Define Functions

Available Actions + Next Action

```
gamma = 0.8

initial_state = 1

def available_actions(state):
    current_state_row = R[state,]
    av_act = np.where(current_state_row >= 0)[1]
    return av_act

available_act = available_actions(initial_state)

###########################################

def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_act,1))
    return next_action

action = sample_next_action(available_act)
```

Define Function

Update Q Table

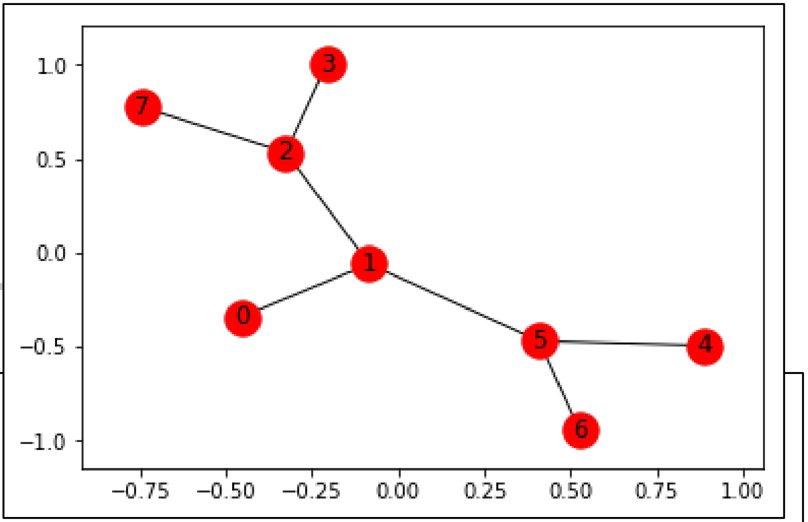
```
def update(current_state, action, gamma):

  max_index = np.where(Q[action,] == np.max(Q[action,]))[1]

  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]

  Q[current_state, action] = R[current_state, action] + gamma * max_value
  print('max_value', R[current_state, action] + gamma * max_value)

  if (np.max(Q) > 0):
    return(np.sum(Q/np.max(Q)*100))
  else:
    return (0)


update(initial_state, action, gamma)
```

Training

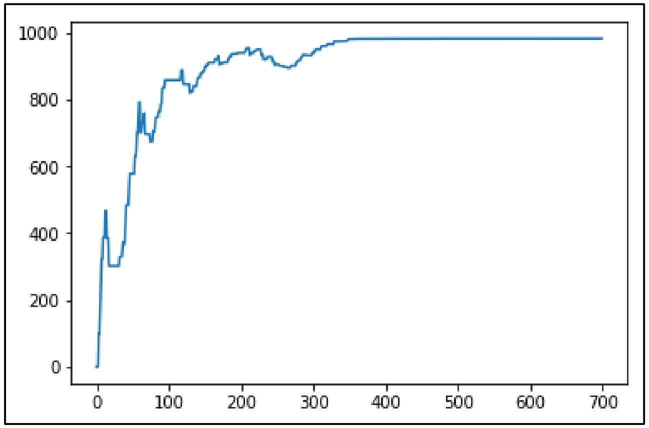
```
# Training
scores = []
for i in range(700):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_act = available_actions(current_state)
    action = sample_next_action(available_act)
    score = update(current_state,action,gamma)
    scores.append(score)
    print ('Score:', str(score))

print("Trained Q matrix:")
print(Q/np.max(Q)*100)
```

```
Trained Q matrix:
[[  0.          63.99932561   0.           0.           0.           0.
    0.           0.        ]
 [ 51.19946049   0.          79.99915702   0.           0.
   51.19638565   0.           0.        ]
 [  0.          63.99548206   0.          63.99885729   0.           0.
    0.          99.99894627]
 [  0.           0.          79.99915702   0.           0.           0.
    0.           0.        ]
 [  0.           0.           0.           0.           0.
   51.19638565   0.           0.        ]
 [  0.          63.99548206   0.           0.          40.95710852
    0.          40.95619382   0.        ]
 [  0.           0.           0.           0.           0.
   51.19638565   0.           0.        ]
 [  0.           0.          79.99915702   0.           0.           0.
    0.         100.        ]]
```

Testing




```
current_state = 0
steps = [current_state]

while current_state != 7:

    next_step_index = np.where(Q[current_state,]
        == np.max(Q[current_state,]))[1]

    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)

    steps.append(next_step_index)
    current_state = next_step_index


print("Most efficient path:")
print(steps)
Most efficient path:
[0, 1, 2, 7]
plt.plot(scores)
```



Summary

- What is Reinforcement Learning
- Markov Decision Process
- Q Learning
- Implementation of Q Learning in Python