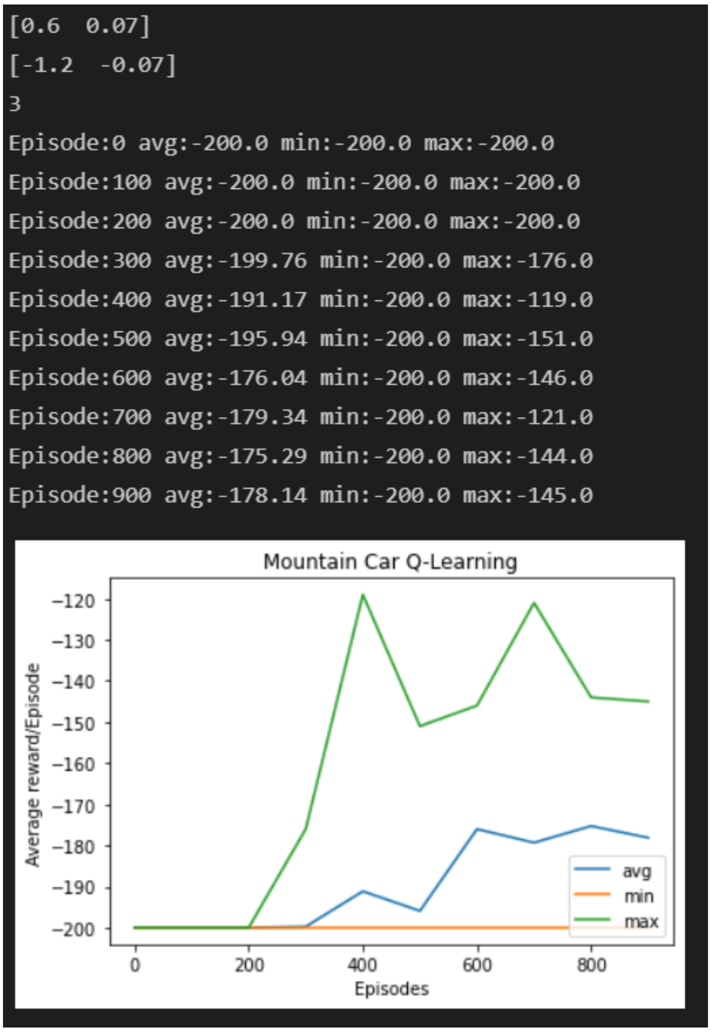
NAME : Swapnil Ghosh
ROLL : 001911001067
YEAR : 4TH YEAR
LAB : MACHINE LEARNING
ASSIGNMENT : 5

1.Mountain car trying to go to top hill using Q-learning

```
import gym
import numpy as np
import matplotlib.pyplot as plt

env = gym.make("MountainCar-v0")

#Environment values
print(env.observation_space.high) #[0.6  0.07]
print(env.observation_space.low)  #[-1.2  -0.07]
print(env.action_space.n)     #3

DISCRETE_BUCKETS = 20
EPISODES = 1000
DISCOUNT = 0.95
EPISODE_DISPLAY = 100
LEARNING_RATE = 0.1
EPSILON = 0.5
EPSILON_DECREMENTER = EPSILON/(EPISODES//4)

#Q-Table of size DISCRETE_BUCKETS*DISCRETE_BUCKETS*env.action_space.n
Q_TABLE = np.random.randn(DISCRETE_BUCKETS,DISCRETE_BUCKETS,env.action_
space.n)

# For stats
ep_rewards = []
ep_rewards_table = {'ep': [], 'avg': [], 'min': [], 'max': []}

def discretised_state(state):
  DISCRETE_WIN_SIZE = (env.observation_space.high-
env.observation_space.low)/[DISCRETE_BUCKETS]*len(env.observation_space
.high)
  discrete_state = (state-env.observation_space.low)//DISCRETE_WIN_SIZE
  return tuple(discrete_state.astype(int))    #integer tuple as we need
 to use it later on to extract Q table values

for episode in range(EPISODES):
  episode_reward = 0
  done = False


  curr_discrete_state = discretised_state(env.reset())

  if episode % EPISODE_DISPLAY == 0:
    render_state = True
  else:
    render_state = False

  while not done:
    if np.random.random() > EPSILON:
      action = np.argmax(Q_TABLE[curr_discrete_state])
    else:
      action = np.random.randint(0, env.action_space.n)

    new_state, reward, done, _ = env.step(action)
    new_discrete_state = discretised_state(new_state)
    if render_state:
      env.render()

    if not done:
      max_future_q = np.max(Q_TABLE[new_discrete_state])
      current_q = Q_TABLE[curr_discrete_state+(action,)]
      new_q = current_q + LEARNING_RATE*(reward + DISCOUNT*max_future_q
 - current_q)
      Q_TABLE[curr_discrete_state+(action,)]=new_q
    elif new_state[0] >= env.goal_position:
      Q_TABLE[curr_discrete_state + (action,)] = 0

    curr_discrete_state = new_discrete_state
    episode_reward += reward

  EPSILON = EPSILON - EPSILON_DECREMENTER

  ep_rewards.append(episode_reward)

  if not episode % EPISODE_DISPLAY:
    avg_reward = sum(ep_rewards[-EPISODE_DISPLAY:])/len(ep_rewards[-
EPISODE_DISPLAY:])
    ep_rewards_table['ep'].append(episode)
    ep_rewards_table['avg'].append(avg_reward)
    ep_rewards_table['min'].append(min(ep_rewards[-EPISODE_DISPLAY:]))
    ep_rewards_table['max'].append(max(ep_rewards[-EPISODE_DISPLAY:]))

    print(f"Episode:{episode} avg:{avg_reward} min:{min(ep_rewards[-
EPISODE_DISPLAY:])} max:{max(ep_rewards[-EPISODE_DISPLAY:])}")
```

```
env.close()

plt.plot(ep_rewards_table['ep'], ep_rewards_table['avg'], label="avg")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['min'], label="min")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['max'], label="max")
plt.legend(loc=4) #bottom right
plt.title('Mountain Car Q-Learning')
plt.ylabel('Average reward/Episode')
plt.xlabel('Episodes')
plt.show()
```

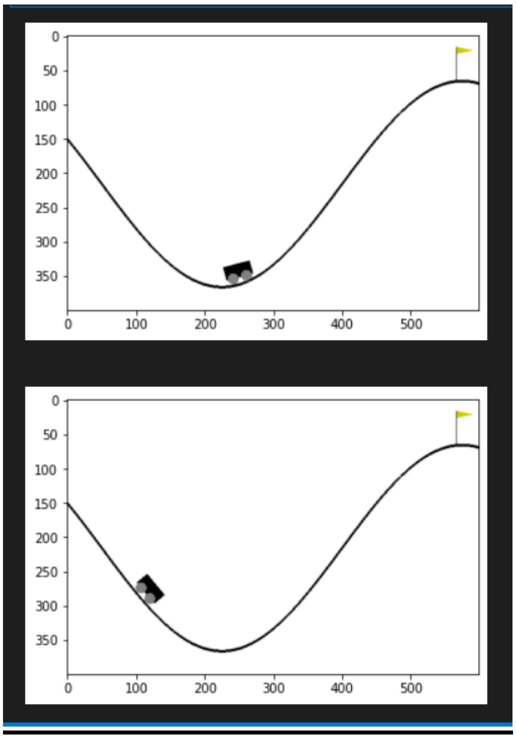


```
[0.6  0.07]
[-1.2  -0.07]
3
Episode:0 avg:-200.0 min:-200.0 max:-200.0
Episode:100 avg:-200.0 min:-200.0 max:-200.0
Episode:200 avg:-200.0 min:-200.0 max:-200.0
Episode:300 avg:-199.76 min:-200.0 max:-176.0
Episode:400 avg:-191.17 min:-200.0 max:-119.0
Episode:500 avg:-195.94 min:-200.0 max:-151.0
Episode:600 avg:-176.04 min:-200.0 max:-146.0
Episode:700 avg:-179.34 min:-200.0 max:-121.0
Episode:800 avg:-175.29 min:-200.0 max:-144.0
Episode:900 avg:-178.14 min:-200.0 max:-145.0
```

Visualization

```
%matplotlib inline

import gym
import numpy as np
from matplotlib import pyplot as plt
env = gym.envs.make("MountainCar-v0")

env.reset()
```

```
plt.figure()
plt.imshow(env.render(mode='rgb_array'))

[env.step(0) for x in range(10000)]
plt.figure()
plt.imshow(env.render(mode='rgb_array'))

env.close()
```

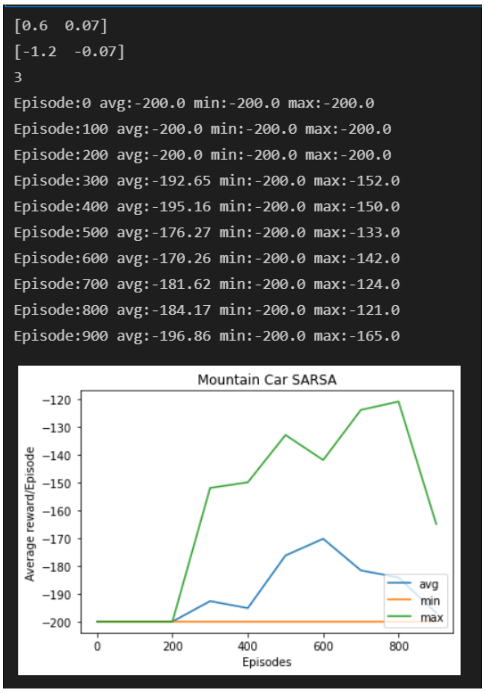


1.1.Mountain car trying to go to top hill using SARSA

```
import gym
import numpy as np
import matplotlib.pyplot as plt

env = gym.make("MountainCar-v0")

#Environment values
print(env.observation_space.high) #[0.6  0.07]
print(env.observation_space.low)  #[-1.2  -0.07]
print(env.action_space.n)      #3

DISCRETE_BUCKETS = 20
EPISODES = 1000
DISCOUNT = 0.95
EPISODE_DISPLAY = 100
LEARNING_RATE = 0.1
EPSILON = 0.5
EPSILON_DECREMENTER = EPSILON/(EPISODES//4)
```

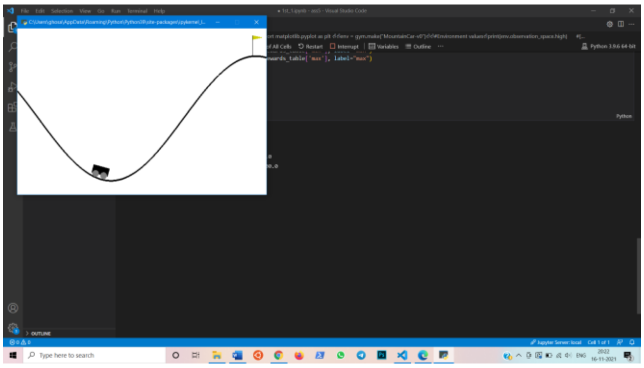
```
#Q-Table of size DISCRETE_BUCKETS*DISCRETE_BUCKETS*env.action_space.n
Q_TABLE = np.random.randn(DISCRETE_BUCKETS,DISCRETE_BUCKETS,env.action_
space.n)

# For stats
ep_rewards = []
ep_rewards_table = {'ep': [], 'avg': [], 'min': [], 'max': []}

def discretised_state(state):
  DISCRETE_WIN_SIZE = (env.observation_space.high-
env.observation_space.low)/[DISCRETE_BUCKETS]*len(env.observation_space
.high)
  discrete_state = (state-env.observation_space.low)//DISCRETE_WIN_SIZE
  return tuple(discrete_state.astype(int))     #integer tuple as we need
 to use it later on to extract Q table values

for episode in range(EPISODES):
  episode_reward = 0
  done = False

  if episode % EPISODE_DISPLAY == 0:
    render_state = True
  else:
    render_state = False

  curr_discrete_state = discretised_state(env.reset())
  if np.random.random() > EPSILON:
    action = np.argmax(Q_TABLE[curr_discrete_state])
  else:
    action = np.random.randint(0, env.action_space.n)

  while not done:
    new_state, reward, done, _ = env.step(action)
    new_discrete_state = discretised_state(new_state)

    if np.random.random() > EPSILON:
      new_action = np.argmax(Q_TABLE[new_discrete_state])
    else:
      new_action = np.random.randint(0, env.action_space.n)

    if render_state:
      env.render()

    if not done:
      current_q = Q_TABLE[curr_discrete_state+(action,)]
      max_future_q = Q_TABLE[new_discrete_state+(new_action,)]
      new_q = current_q + LEARNING_RATE*(reward+DISCOUNT*max_future_q-
current_q)
```

```
      Q_TABLE[curr_discrete_state+(action,)]=new_q
    elif new_state[0] >= env.goal_position:
      Q_TABLE[curr_discrete_state + (action,)] = 0

    curr_discrete_state = new_discrete_state
    action = new_action

    episode_reward += reward

  EPSILON = EPSILON - EPSILON_DECREMENTER

  ep_rewards.append(episode_reward)

  if not episode % EPISODE_DISPLAY:
    avg_reward = sum(ep_rewards[-EPISODE_DISPLAY:])/len(ep_rewards[-
EPISODE_DISPLAY:])
    ep_rewards_table['ep'].append(episode)
    ep_rewards_table['avg'].append(avg_reward)
    ep_rewards_table['min'].append(min(ep_rewards[-EPISODE_DISPLAY:]))
    ep_rewards_table['max'].append(max(ep_rewards[-EPISODE_DISPLAY:]))

    print(f"Episode:{episode} avg:{avg_reward} min:{min(ep_rewards[-
EPISODE_DISPLAY:])} max:{max(ep_rewards[-EPISODE_DISPLAY:])}")

env.close()

plt.plot(ep_rewards_table['ep'], ep_rewards_table['avg'], label="avg")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['min'], label="min")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['max'], label="max")
plt.legend(loc=4) #bottom right
plt.title('Mountain Car SARSA')
plt.ylabel('Average reward/Episode')
plt.xlabel('Episodes')
plt.show()
```

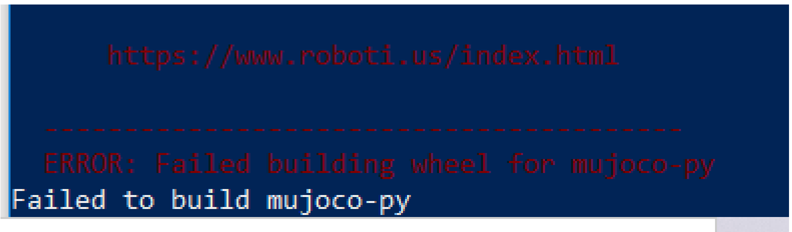


```
[0.6  0.07]
[-1.2  -0.07]
3
Episode:0 avg:-200.0 min:-200.0 max:-200.0
Episode:100 avg:-200.0 min:-200.0 max:-200.0
Episode:200 avg:-200.0 min:-200.0 max:-200.0
Episode:300 avg:-192.65 min:-200.0 max:-152.0
Episode:400 avg:-195.16 min:-200.0 max:-150.0
Episode:500 avg:-176.27 min:-200.0 max:-133.0
Episode:600 avg:-170.26 min:-200.0 max:-142.0
Episode:700 avg:-181.62 min:-200.0 max:-124.0
Episode:800 avg:-184.17 min:-200.0 max:-121.0
Episode:900 avg:-196.86 min:-200.0 max:-165.0
```

Visualization



2.Car Racing



```
    https://www.roboti.us/index.html

  ----------------------------------------
  ERROR: Failed building wheel for mujoco-py
Failed to build mujoco-py
```

It was the error that I encountered while running the code. I tried a lot to solve it unable to did it finally, I tried to install gym then gym[all] but there was error .

```
import gym
import numpy as np
import random
from scipy import misc
import tensorflow as tf
```

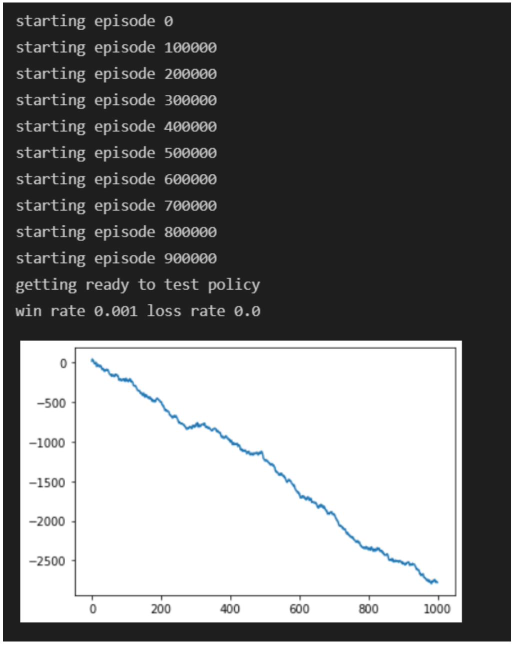
```
env = gym.make('CarRacing-v0')
observation = env.reset()

EPISODES = 1
TIMESTAMP = 5
GAMMA = 0.99
ALPHA = 0.001
explore_eps = 1
N = 50
OUT1 = 5
OUT2 = 5
OUT3 = 5
BATCH_SIZE = 4

def conv2d(x,W,stride):
  return tf.nn.conv2d(x,W,strides=[1,stride,stride,1],padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x,ksize=[1,2,2,1],strides = [1,2,2,1],padding='
SAME')

class neuralNet:
  def __init__(self):
    self.sess = tf.InteractiveSession()

    self.X = tf.placeholder(tf.float32,[None,N,N,1])
    self.C1 = tf.placeholder(tf.float32,[None,OUT1])
    self.C2 = tf.placeholder(tf.float32,[None,OUT2])
    self.C3 = tf.placeholder(tf.float32,[None,OUT3])
    self.Y1 = tf.placeholder(tf.float32,[None,OUT1])
    self.Y2 = tf.placeholder(tf.float32,[None,OUT2])
    self.Y3 = tf.placeholder(tf.float32,[None,OUT3])

    self.W_conv1 = tf.Variable(tf.truncated_normal([8,8,1,32],stddev =
0.1))        # 50 * 50 * 1
    self.B_conv1 = tf.Variable(tf.zeros([32]))

    self.W_conv2 = tf.Variable(tf.truncated_normal([5,5,32,64],stddev =
 0.1))       # 15 * 15 * 32
    self.B_conv2 = tf.Variable(tf.zeros([64]))

    self.W_conv3 = tf.Variable(tf.truncated_normal([3,3,64,64],stddev =
 0.1))       # 7 * 7 * 64
    self.B_conv3 = tf.Variable(tf.zeros([64]))

    self.W_fc1 = tf.Variable(tf.truncated_normal([ 5*5*64 , 512],stddev
 = 0.1))     # 5 * 5 * 64
```

```
    self.B_fc1 = tf.Variable(tf.zeros([512]))

    self.W_fc21 = tf.Variable(tf.truncated_normal([512,OUT1],stddev = 0
.1))
    self.B_fc21 = tf.Variable(tf.zeros([OUT1]))

    self.W_fc22 = tf.Variable(tf.truncated_normal([512,OUT2],stddev = 0
.1))
    self.B_fc22 = tf.Variable(tf.zeros([OUT2]))

    self.W_fc23 = tf.Variable(tf.truncated_normal([512,OUT3],stddev = 0
.1))
    self.B_fc23 = tf.Variable(tf.zeros([OUT3]))

    o_conv1 = tf.nn.relu(conv2d(self.X,self.W_conv1,3) + self.B_conv1)
    o_pool1 = max_pool_2x2(o_conv1)

    o_conv2 = tf.nn.relu(conv2d(o_pool1,self.W_conv2,2) + self.B_conv2)

    o_conv3 = tf.nn.relu(conv2d(o_conv2,self.W_conv3,1) + self.B_conv3)
    o_fconv3 = tf.reshape(o_conv3,[-1,5*5*64])

    o_fc1 = tf.nn.relu(tf.matmul(o_fconv3,self.W_fc1) + self.B_fc1)

    self.o_fc21 = tf.matmul(o_fc1,self.W_fc21) + self.B_fc21

    self.o_fc22 = tf.matmul(o_fc1,self.W_fc22) + self.B_fc22

    self.o_fc23 = tf.matmul(o_fc1,self.W_fc23) + self.B_fc23

    self.L1 = tf.reduce_sum(tf.square(self.Y1 - tf.mul(self.o_fc21,self
.C1)))
    self.L2 = tf.reduce_sum(tf.square(self.Y2 - tf.mul(self.o_fc22,self
.C2)))
    self.L3 = tf.reduce_sum(tf.square(self.Y3 - tf.mul(self.o_fc23,self
.C3)))

    self.optimizer = tf.train.AdamOptimizer(ALPHA)
    self.train_step1 = self.optimizer.minimize(self.L1)
    self.train_step2 = self.optimizer.minimize(self.L2)
    self.train_step3 = self.optimizer.minimize(self.L3)

    self.sess.run(tf.initialize_all_variables())

  def forward_pass(self,x):
    with self.sess.as_default():
      out1, out2, out3 = self.sess.run([self.o_fc21,self.o_fc22,self.o_
fc23],feed_dict={self.X:x})
```

```
    # print out
    return np.argmax(out1),np.argmax(out2),np.argmax(out3),np.max(out1)
,np.max(out2),np.max(out3)

  def train(self,x,y1,y2,y3,c1,c2,c3):
    with self.sess.as_default():
      self.sess.run([self.train_step1,self.train_step2,self.train_step3
],feed_dict={
        self.X:x , self.Y1:y1, self.Y2:y2, self.Y3:y3, self.C1:c1, self
.C2:c2, self.C3:c3})


def sanity_check():
  observation = env.reset()
  print (observation.shape)
  print(env.action_space)
  print(env.action_space.sample())
  print(env.observation_space)
  # print(env.observation_space.high)
  # print(env.observation_space.low)
  print(env.action_space.high)
  print(env.action_space.low)

def process_image(ot):
  ot = misc.imresize(ot , (N,N,3) )
  ot = 0.299*ot[:,:,0] + 0.587*ot[:,:,1] + 0.114*ot[:,:,2]
  ot = np.reshape(ot , (1,N,N,1))
  return ot

def create_new_data(ot,re,ot2,reset,done,a1,a2,a3):
  c1 = np.zeros((1,OUT1))
  c1[0][a1] = 1
  c2 = np.zeros((1,OUT1))
  c2[0][a2] = 1
  c3 = np.zeros((1,OUT1))
  c3[0][a3] = 1
  yval1 = np.zeros((1,OUT1))
  yval2 = np.zeros((1,OUT2))
  yval3 = np.zeros((1,OUT3))
  b1,b2,b3,bv1,bv2,bv3 = nnet.forward_pass(ot2)
  yval1[0][a1] = re
  yval2[0][a2] = re
  yval3[0][a3] = re
  if not done:
    yval1[0][a1] = re + GAMMA*bv1
    yval2[0][a2] = re + GAMMA*bv2
    yval3[0][a3] = re + GAMMA*bv3
  data_batch['C1'] = c1
  data_batch['C2'] = c2
  data_batch['C3'] = c3
  if reset:
    data_batch['X'] = ot
    data_batch['Y1'] = yval1
    data_batch['Y2'] = yval1
    data_batch['Y3'] = yval1
  else:
    data_batch['X'] = np.append(data_batch['X'],ot,axis=0)
    data_batch['Y1'] = np.append(data_batch['Y1'],yval1,axis=0)
    data_batch['Y2'] = np.append(data_batch['Y2'],yval2,axis=0)
    data_batch['Y3'] = np.append(data_batch['Y3'],yval3,axis=0)

nnet = neuralNet()
data_batch = {}
sanity_check()
ans = np.zeros((12))
anssum = np.zeros((12))
for ep in range(EPISODES):
  observation = env.reset()
  observation = process_image(observation)
  reward = 0
  sum_reward = 0
  data_batch = {}
  reset = True
  for t in range(TIMESTAMP):
    env.render()
    x = np.array(observation)
    a1,a2,a3,av1,av2,av3 = nnet.forward_pass(x)

    tempvar = random.random()
    if tempvar < max((500/(ep+1)),explore_eps) and ep < 9000:      # do
nt explore for last 1000 episodes
      a1 = np.random.randint(0,5,size=1)
      a2 = np.random.randint(0,5,size=1)
      a3 = np.random.randint(0,5,size=1)

    action = [ -1.0 + a1*0.4 , a2*0.2 , a3*0.2 ]
    observation, reward, done, info = env.step(action)
    observation = process_image(observation)
    create_new_data(x,reward,np.array(observation),reset,done,a1,a2,a3)
    print (data_batch['X'].shape , data_batch['Y'].shape , data_batch['
C'].shape)
    reset = False

    if data_batch['X'].shape[0] == BATCH_SIZE:
      nnet.train(data_batch['X'] , data_batch['Y'], data_batch['C'])
      reset = True
    sum_reward = sum_reward + reward
    if done or t == TIMESTAMP-1:
      nnet.train(data_batch['X'] , data_batch['Y'], data_batch['C'])
      print("Episode {0} finished after {1} timesteps.".format(ep+1,t+1
))
      ans[int(ep/5000)] = max(ans[int(ep/5000)],t)
      anssum[int(ep/5000)] += anssum[int(ep/5000)]
      break

for i in range(3):
  print (i*5000 , " -
- ", (i+1)*5000 , " == " , ans[i] , (anssum[i]/5000))
```

3.Roulette

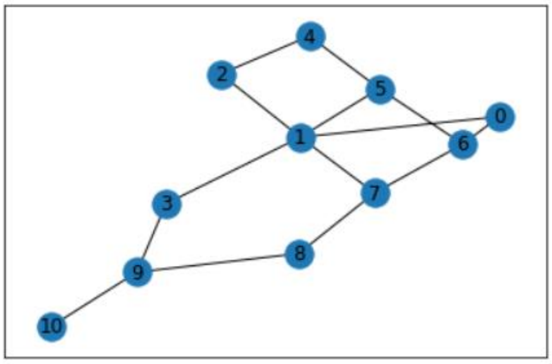
```
import gym
import numpy as np
import matplotlib.pyplot as plt
import gym_toytext
env = gym.make('Roulette-v0')
EPS = 0.05
GAMMA = 1.0
Q = {}
agentSumSpace = [i for i in range(0,37)]
actionSpace = [i for i in range(0, 38)]
stateSpace = []
returns = {}
pairsVisited = {}
for total in agentSumSpace:
    for action in actionSpace:
        Q[(total, action)] = 0
        returns[(total, action)] = 0
        pairsVisited[(total, action)] = 0
    stateSpace.append(total)

policy = {}
for state in stateSpace:
    policy[state] = np.random.choice(actionSpace)

numEpisodes = 1000000
for i in range(numEpisodes):
    statesActionsReturns = []
    memory = []
    if i % 100000 == 0:
        print('starting episode', i)
    observation = env.reset()
    done = False

while not done:
    action = policy[observation]
    observation_, reward, done, info = env.step(action)
    memory.append((observation, action, reward))
    observation = observation_
memory.append((observation, action, reward))


G = 0
last = True
for observed, action, reward in reversed(memory):
    if last:
        last = False
    else:
        statesActionsReturns.append((observed, action, G))
    G = GAMMA*G + reward
statesActionsReturns.reverse()
statesActionsVisited = []

for observed, action, G in statesActionsReturns:
    sa = (observed, action)
    if sa not in statesActionsVisited:
        pairsVisited[sa] += 1
        returns[(sa)] += (1 / pairsVisited[(sa)])*(G-returns[(sa)])
        Q[sa] = returns[sa]
        rand = np.random.random()
        if rand < 1 - EPS:
            state = observed
            values = np.array([Q[(state, a)] for a in actionSpace ])
            best=np.random.choice(np.where(values==values.max())[0])
            policy[state] = actionSpace[best]
        else:
            policy[state] = np.random.choice(actionSpace)
        statesActionsVisited.append(sa)
if EPS - 1e-7 > 0:
    EPS -= 1e-7
else:
    EPS = 0

numEpisodes = 1000
rewards = np.zeros(numEpisodes)
totalReward = 0
wins = 0
losses = 0
print('getting ready to test policy')
for i in range(numEpisodes):
    observation = env.reset()
    done = False
    while not done:
        action = policy[observation]
        observation_, reward, done, info = env.step(action)
        observation = observation_
        totalReward += reward
        rewards[i] = totalReward
if reward >= 1:
    wins += 1
elif reward == -1:
    losses += 1

wins /= numEpisodes
losses /= numEpisodes
print('win rate', wins, 'loss rate', losses)
plt.plot(rewards)
plt.show()
```



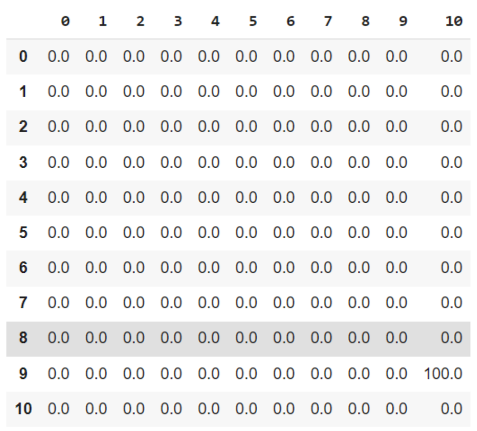
```
starting episode 0
starting episode 100000
starting episode 200000
starting episode 300000
starting episode 400000
starting episode 500000
starting episode 600000
starting episode 700000
starting episode 800000
starting episode 900000
getting ready to test policy
win rate 0.001 loss rate 0.0
```

4. Implement both RL and DRL for finding the shortest path in any user-input graph.

```
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import random
%matplotlib inline
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
         (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
         (8, 9), (7, 8), (1, 7), (3, 9)]
G=nx.Graph()
G.add_edges_from(edges)
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()
```

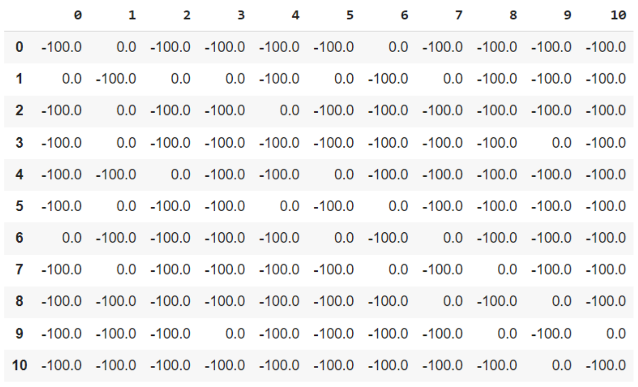


```
R=np.matrix(np.zeros(shape=(11, 11)))
for x in G[10]:
  R[x,10]=100
Q=np.matrix(np.zeros(shape=(11, 11)))
Q-=100
for node in G.nodes:
  for x in G[node]:
    Q[node,x]=0
    Q[x,node]=0
import pandas as pd
pd.DataFrame(R)
```



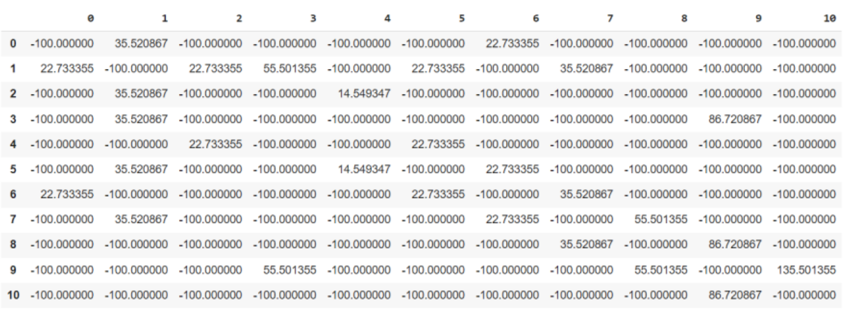
	0	1	2	3	4	5	6	7	8	9	10
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
pd.DataFrame(Q)
```



	0	1	2	3	4	5	6	7	8	9	10
0	-100.0	0.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0	-100.0	-100.0	-100.0
1	0.0	-100.0	0.0	0.0	-100.0	0.0	-100.0	0.0	-100.0	-100.0	-100.0
2	-100.0	0.0	-100.0	-100.0	0.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0
3	-100.0	0.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0
4	-100.0	-100.0	0.0	-100.0	-100.0	0.0	-100.0	-100.0	-100.0	-100.0	-100.0
5	-100.0	0.0	-100.0	-100.0	0.0	-100.0	0.0	-100.0	-100.0	-100.0	-100.0
6	0.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0	0.0	-100.0	-100.0	-100.0
7	-100.0	0.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0	0.0	-100.0	-100.0
8	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0	0.0	-100.0
9	-100.0	-100.0	-100.0	0.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0	0.0
10	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	-100.0	0.0	-100.0

```
def next_number(start,er):
  random_value=random.uniform(0, 1)
  if random_value<er:
    sample=G[start]
  else:
    sample=np.where(Q[start,]== np.max(Q[start,]))[1]
  next_node=int(np.random.choice(sample,1))
  return next_node
def updateQ(node1,node2,lr,discount):
  max_index=np.where(Q[node2,]==np.max(Q[node2,]))[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[node2,max_index]
  Q[node1,node2]=int(1-
lr)*Q[node1,node2]+lr*(R[node1,node2]+discount*max_value)
def learn(er,lr,discount):
  for i in range(50000):
    start=np.random.randint(0,11)
    next_node=next_number(start,er)
    updateQ(start,next_node,lr,discount)
learn(0.5,0.8,0.8)
pd.DataFrame(Q)
```



	0	1	2	3	4	5	6	7	8	9	10
0	-100.000000	35.520867	-100.000000	-100.000000	-100.000000	-100.000000	22.733355	-100.000000	-100.000000	-100.000000	-100.000000
1	22.733355	-100.000000	22.733355	55.501355	-100.000000	22.733355	-100.000000	35.520867	-100.000000	-100.000000	-100.000000
2	-100.000000	35.520867	-100.000000	-100.000000	14.549347	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000
3	-100.000000	35.520867	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	86.720867	-100.000000
4	-100.000000	-100.000000	22.733355	-100.000000	-100.000000	22.733355	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000
5	-100.000000	35.520867	-100.000000	-100.000000	14.549347	-100.000000	22.733355	-100.000000	-100.000000	-100.000000	-100.000000
6	22.733355	-100.000000	-100.000000	-100.000000	-100.000000	22.733355	-100.000000	35.520867	-100.000000	-100.000000	-100.000000
7	-100.000000	35.520867	-100.000000	-100.000000	-100.000000	-100.000000	22.733355	-100.000000	55.501355	-100.000000	-100.000000
8	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	35.520867	-100.000000	86.720867	-100.000000
9	-100.000000	-100.000000	-100.000000	55.501355	-100.000000	-100.000000	-100.000000	-100.000000	55.501355	-100.000000	135.501355
10	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	-100.000000	86.720867	-100.000000

```
def shortest_path(begin,end):
  path=[begin]
  next_node=np.argmax(Q[begin,])
  path.append(next_node)
  while next_node!=end:
    next_node=np.argmax(Q[next_node,])
    path.append(next_node)
  return path

shortest_path(0,10)
```

```
[0, 1, 3, 9, 10]
```