# Cart-Pole - TD(0)

March 8, 2021

Name: Vijaykumar Maraviya

Student Number: 1006040320

[1]:
```
import gym
import numpy as np
import math
import matplotlib.pyplot as plt
from IPython import display as ipythondisplay
from sklearn.preprocessing import KBinsDiscretizer
from itertools import product
import collections
```

This function selects an action using e-greedy policy for a given Q

[2]:
```
def e_greedy(Q, eps, S):
    # random action with probability eps
    if np.random.random() < eps:
        return np.random.choice([0,1])

    # greedy action otherwise
    act_vals = np.array([Q[(S,a)] for a in [0,1]])
    return np.random.choice(np.where(act_vals == act_vals.max())[0])
```

## 0.1 Implementation

### 0.1.1 a. on-policy SARSA

[3]:
```
def on_policy_SARSA(env, state_space, action_space, descritizer, max_episodes =
↪100000, GAMMA = 1.0, EPS = 0.1, ALPHA = 0.1):
    eps = EPS
    # intialize Q
    Q = {}
    for s in state_space:
        for a in action_space:
            Q[(s,a)] = 0

    # loop for max_episodes
    for n_eps in range(max_episodes):
```

```
        # initialize S
        obs = env.reset()
        S = descritizer(*obs)

        # choose A
        A = e_greedy(Q, eps, S)

        done = False
        while not done:
            # take action A, observe R and S_next
            obs, R, done, _ = env.step(A)
            S_next = descritizer(*obs)

            # choose A_next
            A_next = e_greedy(Q, eps, S_next)

            # updtae Q
            Q[(S,A)] = Q[(S,A)] + ALPHA*(R + GAMMA*Q[(S_next, A_next)] -␣
↪Q[(S,A)])

            # next S, A
            S, A = S_next, A_next

    return Q
```

### 0.1.2 b. off-policy Q learning

[4]:
```
def off_policy_Q(env, state_space, action_space, descritizer, max_episodes =␣
↪100000, GAMMA = 1.0, EPS = 0.1, ALPHA = 0.1):
    eps = EPS
    # intialize Q
    Q = {}
    for s in state_space:
        for a in action_space:
            Q[(s,a)] = 0

    # loop for max_episodes
    for n_eps in range(max_episodes):
        # initialize S
        obs = env.reset()
        S = descritizer(*obs)

        done = False
        while not done:
            # choose A (behaviour policy e-greedy)
            A = e_greedy(Q, eps, S)
```

```
            # take action A, observe R and S_next
            obs, R, done, _ = env.step(A)
            S_next = descritizer(*obs)

            # target policy is greedy w.r.t to Q
            Q_max = max([ Q[(S_next, a)] for a in action_space])

            # updtae Q
            Q[(S,A)] = Q[(S,A)] + ALPHA*(R + GAMMA*Q_max - Q[(S,A)])

            # next S, A
            S = S_next

    return Q
```

### 0.1.3 c. off-policy Expected SARSA

[5]:
```
def Expected_SARSA(env, state_space, action_space, descritizer, max_episodes =␣
↪100000, GAMMA = 1.0, EPS = 0.1, ALPHA = 0.5):
    eps = EPS
    # intialize Q
    Q = {}
    for s in state_space:
        for a in action_space:
            Q[(s,a)] = 0

    # loop for max_episodes
    for n_eps in range(max_episodes):
        # initialize S
        obs = env.reset()
        S = descritizer(*obs)

        done = False
        while not done:
            # choose A (behaviour policy e-greedy)
            A = e_greedy(Q, eps, S)

            # take action A, observe R and S_next
            obs, R, done, _ = env.step(A)
            S_next = descritizer(*obs)

            # expected value
            Q_expected = np.mean(np.array([ Q[(S_next, a)] for a in␣
↪action_space]))
```

```
            # updtae Q
            Q[(S,A)] = Q[(S,A)] + ALPHA*(R + GAMMA*Q_expected - Q[(S,A)])

            # next S, A
            S = S_next

    return Q
```

### 0.1.4 c.2 off-policy expected SARSA with an epsilon greedy policy as a target policy

The assignment asked to implement off-policy Expected SARSA with an epsilon-greedy policy. It wasn't clear whether the e-greedy policy is to be used only for behaviour or also for target. So, I have also implemented off-policy Expected SARSA with greedy policy as a target policy. The values of epsilon for behaviour and target policy are different. If they are same, then the algorithm becomes on-policy Expected SARSA.

[6]:
```
def calculate_expected_Q(Q, S, eps):
    if Q[(S,0)] >= Q[(S,1)]:
        expected_value = (1-eps + eps/2)*Q[(S,0)] + (eps/2)*Q[(S,1)]
    else:
        expected_value = (1-eps + eps/2)*Q[(S,1)] + (eps/2)*Q[(S,0)]

    return expected_value

def Expected_SARSA_e_greedy(env, state_space, action_space, descritizer,
                            max_episodes = 100000, GAMMA = 1.0, EPS_1 = 0.2,␣
 ↪EPS_2 = 0.01,
                            ALPHA = 0.1):
    eps = EPS_1
    # intialize Q
    Q = {}
    for s in state_space:
        for a in action_space:
            Q[(s,a)] = 0

    # loop for max_episodes
    for n_eps in range(max_episodes):
        # initialize S
        obs = env.reset()
        S = descritizer(*obs)

        done = False
        while not done:
            # choose A (behaviour policy e-greedy)
            A = e_greedy(Q, eps, S)

            # take action A, observe R and S_next
```

```
            obs, R, done, _ = env.step(A)
            S_next = descritizer(*obs)

            # expected value
            Q_expected  = calculate_expected_Q(Q, S_next, EPS_2)

            # updtae Q
            Q[(S,A)] = Q[(S,A)] + ALPHA*(R + GAMMA*Q_expected - Q[(S,A)])

            # next S, A
            S = S_next

    return Q
```

### 0.1.5 cart-pole environment and discretization of state space

[7]:
```
cp_env = gym.make("CartPole-v0")
cp_env.reset()

# lower bounds of state space
lower_bounds = cp_env.observation_space.low
lower_bounds[1] = -0.5
lower_bounds[3] = -math.radians(50)

# upper bounds of state space
upper_bounds = cp_env.observation_space.high
upper_bounds[1] = 0.5
upper_bounds[3] = math.radians(50)

n_bins = ( 12, 12, 12 , 12 )

# discretize the state
def cp_discretizer( cart_position, cart_velocity, pole_angle, pole_velocity):
    est = KBinsDiscretizer(n_bins=n_bins, encode='ordinal', strategy='uniform')
    est.fit([lower_bounds, upper_bounds ])
    return tuple(map(int, est.transform([[cart_position, cart_velocity,␣
 ↪pole_angle, pole_velocity]])[0]))

# action_space
cp_action_space = [0,1]

# discretized state_space
cp_state_space = []
for s in product(range(12), range(12), range(12), range(12)):
    cp_state_space.append(s)
```

### 0.1.6 train

[8]:
```
# on-policy SARSA
Q_on_policy_sarsa = on_policy_SARSA(env = cp_env,
                                    state_space = cp_state_space,
                                    action_space = cp_action_space,
                                    descritizer = cp_discretizer,
                                    max_episodes = 10000,
                                    GAMMA = 1.0,
                                    EPS = 0.1,
                                    ALPHA = 0.1)
```

[9]:
```
# off-policy Q learning
Q_off_policy_Q = off_policy_Q(env = cp_env,
                              state_space = cp_state_space,
                              action_space = cp_action_space,
                              descritizer = cp_discretizer,
                              max_episodes = 10000,
                              GAMMA = 1.0,
                              EPS = 0.1,
                              ALPHA = 0.1)
```

[10]:
```
# expected SARSA
Q_expected_sarsa = Expected_SARSA(env = cp_env,
                                  state_space = cp_state_space,
                                  action_space = cp_action_space,
                                  descritizer = cp_discretizer,
                                  max_episodes = 10000,
                                  GAMMA = 1.0,
                                  EPS = 0.1,
                                  ALPHA = 0.9)
```

[11]:
```
# expected sarsa with e-greedy policy as a target policy
Q_expected_sarsa_e_greedy = Expected_SARSA_e_greedy(env = cp_env,
                                                    state_space = 
 ↪cp_state_space,
                                                    action_space = 
 ↪cp_action_space,
                                                    descritizer = 
 ↪cp_discretizer,
                                                    max_episodes = 10000,
                                                    ALPHA = 0.9)
```

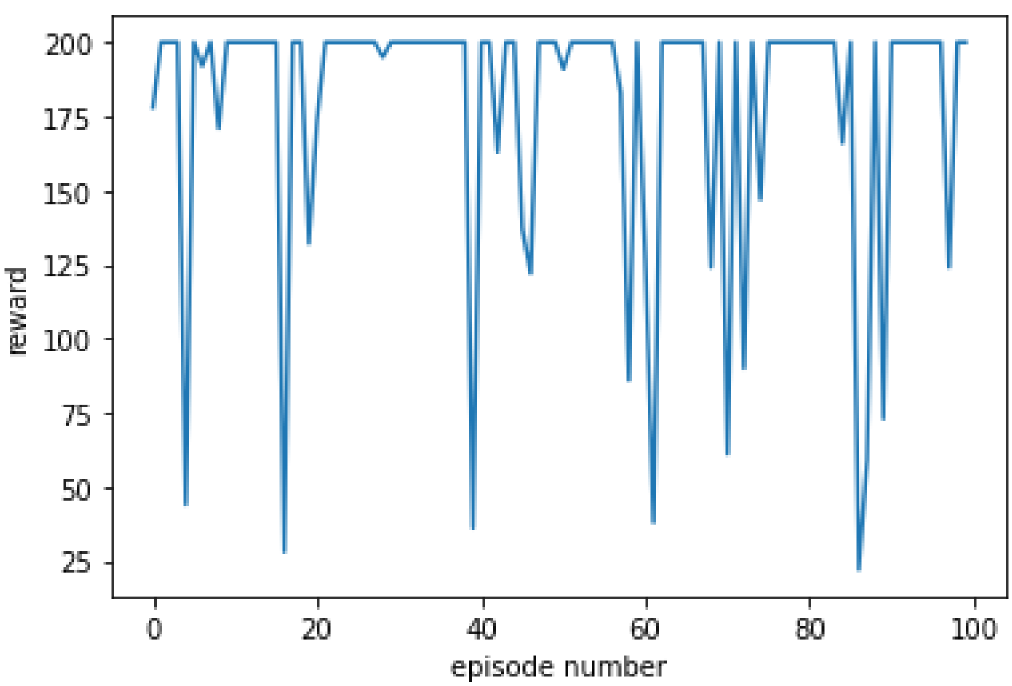
[12]:
```
def test(Q, eps, num_episodes = 1000):
    rewards = np.zeros(num_episodes)
    for i in range(num_episodes):
        totalReward = 0
```

```
        observation = cp_discretizer(*cp_env.reset())
        done = False
        while not done:
            action =  e_greedy(Q, eps, observation)
            observation_, reward, done, info = cp_env.step(action)
            observation = cp_discretizer(*observation_)
            totalReward += reward
        rewards[i] = totalReward

    print(f"Average reward over {num_episodes} episodes: {np.average(rewards):.
↪2f}")
    print(f"number of successes (reward >=200) in {num_episodes} episodes: {np.
↪sum(np.where(rewards >= 200, 1, 0))}")

    plt.plot(rewards)
    plt.xlabel('episode number')
    plt.ylabel('reward')
    plt.show()
```

### 0.1.7 test performance

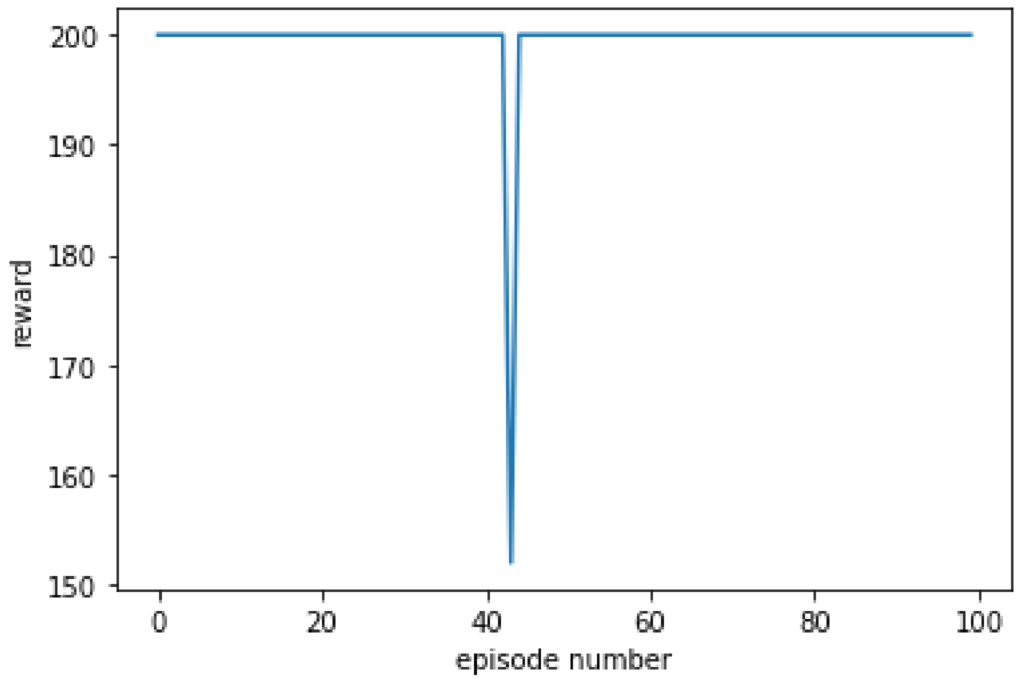
[13]:
```
# target policy is e-greedy (same as behaviour policy)
test(Q_on_policy_sarsa, 0.1, 100)
```

```
Average reward over 100 episodes: 178.63
number of successes (reward >=200) in 100 episodes: 74
```



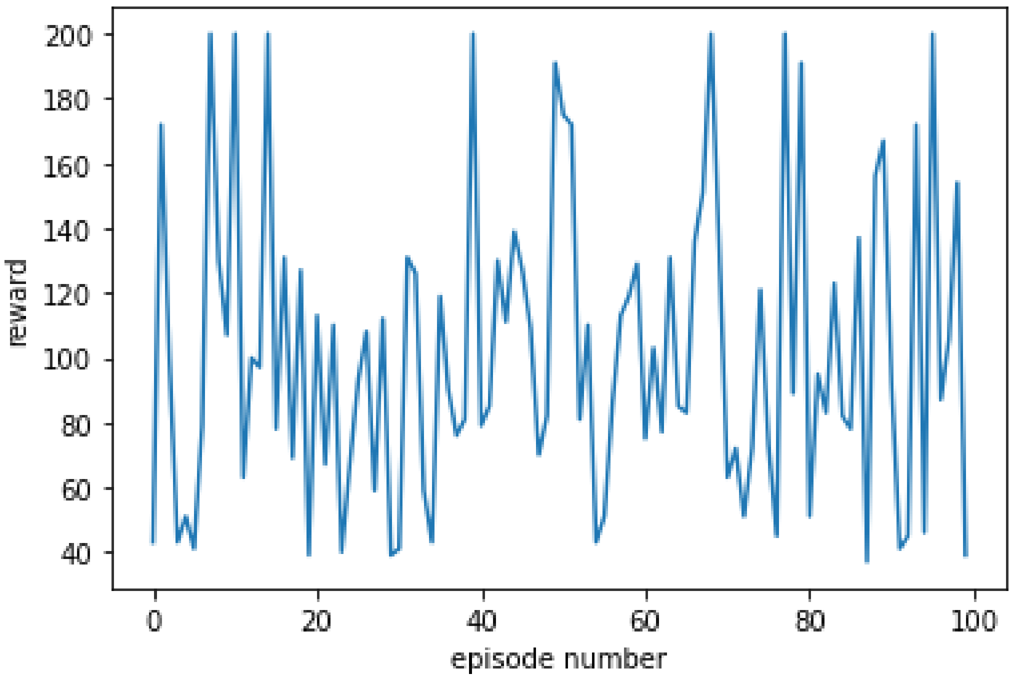
[14]:
```
# target policy is greedy w.r.t to Q (hence epsilon = 0)
test(Q_off_policy_Q, 0, 100)
```

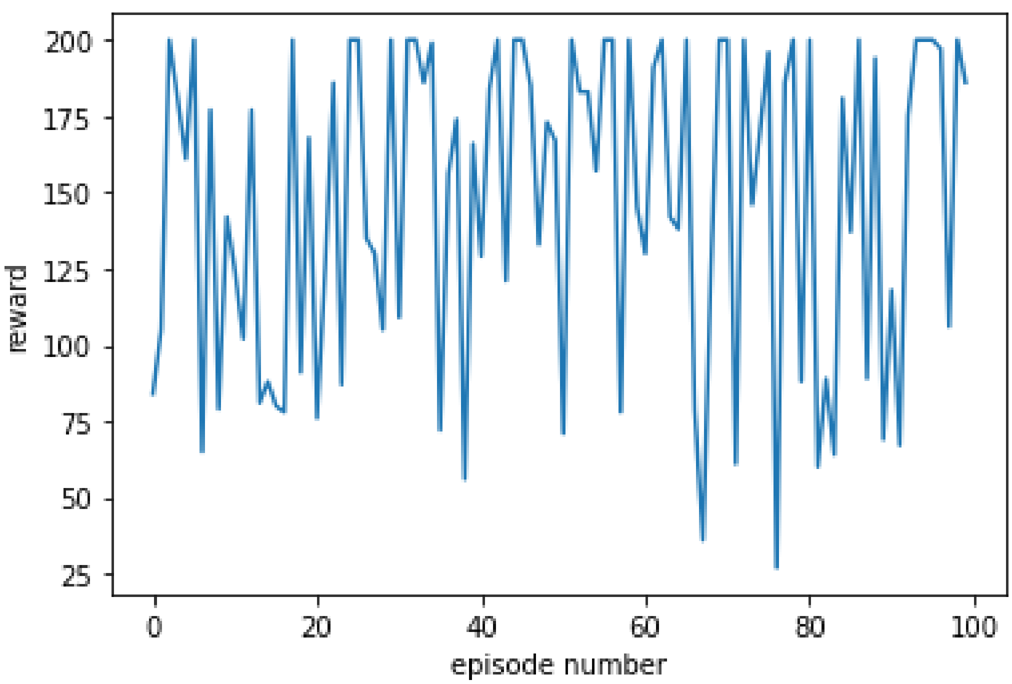
```
Average reward over 100 episodes: 199.52
number of successes (reward >=200) in 100 episodes: 99
```




[15]:
```
test(Q_expected_sarsa, 0, 100)
```

```
Average reward over 100 episodes: 101.88
number of successes (reward >=200) in 100 episodes: 7
```




[16]: ```
test(Q_expected_sarsa_e_greedy, 0.01, 100)
```

```
Average reward over 100 episodes: 147.04
number of successes (reward >=200) in 100 episodes: 27
```


```