# ***CS747- Assignment 3***
***Akshay Iyer- 190070006***

## Task 1-

In task 1, I implemented tabular sarsa.

I first discretised x into kx discrete states, and value of v into kv discrete states.
For the lowest values of x, I put it in the distance state 0, for highest values of x I put it in the distance state kx-1. Similarly for v.

I created a weight matrix w, of dimensions: (number of discrete distance states)*(number of discrete velocity states)*(number of actions).
As this is tabular sarsa, I designed every entry in w to store the value of Q(s,a), for that particular state (both discrete x and v), and a particular action taken from that state.

As we reach every state, we update that particular Q for that state and action using Sarsa(0), by saying that (new Q(state, action))= (Old Q(state, action))*(1-alpha) + alpha*(reward + gamma*(Q(new state, new action))). Hence whenever we reach a state, we update its Q value, depending on the Q value of the next state and action that occurs.

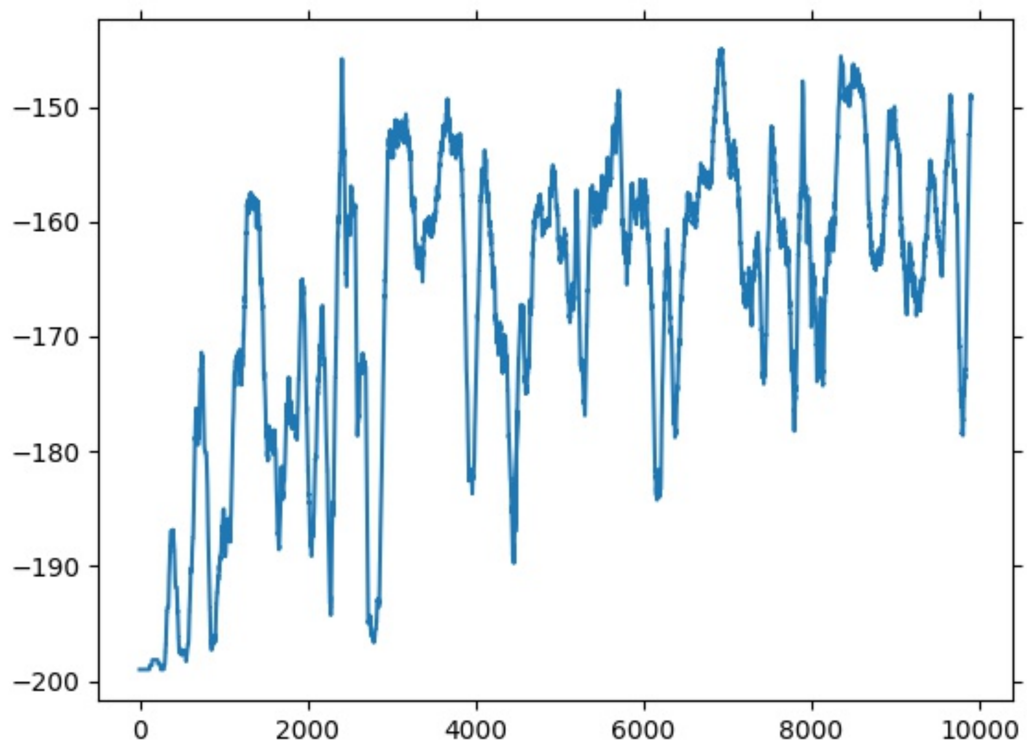
Initially I saw that my results were not that good. It increased, and then decreased drastically, there was no consistent increase in values. Hence I tried decreasing my learning rate. That seemed to work well, as the plot seemed to vary less with a lower learning rate.

I also had to tune other parameters like exploration rate, to get the best results.

I initially chose the number of discretisations for x and v to be too much, almost 25 each. Then when I didn't get very good results, I decided to make the number of divisions 10 for x and 10 for v. This is because I wanted every state to get updated more frequently, and with less divisions, every state gets encountered more.

For choosing my action, I found the state in which our body was in, and then depending on which action for that state has the highest value for Q, I chose that action.

The plot I got with my final values was as follows-



Thus we see that there was a fair bit of variation in the graph, yet the overall line in the plot saw an increase in average reward. At the end of 10000 iterations, we finally see a reward of slightly more than -150, as I found when I tested the same on various different random seeds and test data too.

Plot analysis- We see that the graph varied very much in terms of average result, as our number of training iterations increased. Yet this increase is erratic, and not very evident right from the beginning of the graph. For higher learning rates we see even higher variation in the reward, with training iterations in the graph. Thus I decreased my learning rate to get lesser variation, and a better reward graph. We see an overall increase in our reward, yet the graph varies a lot, and I only managed to hit a reward of slightly greater than -150.

## Task 2-

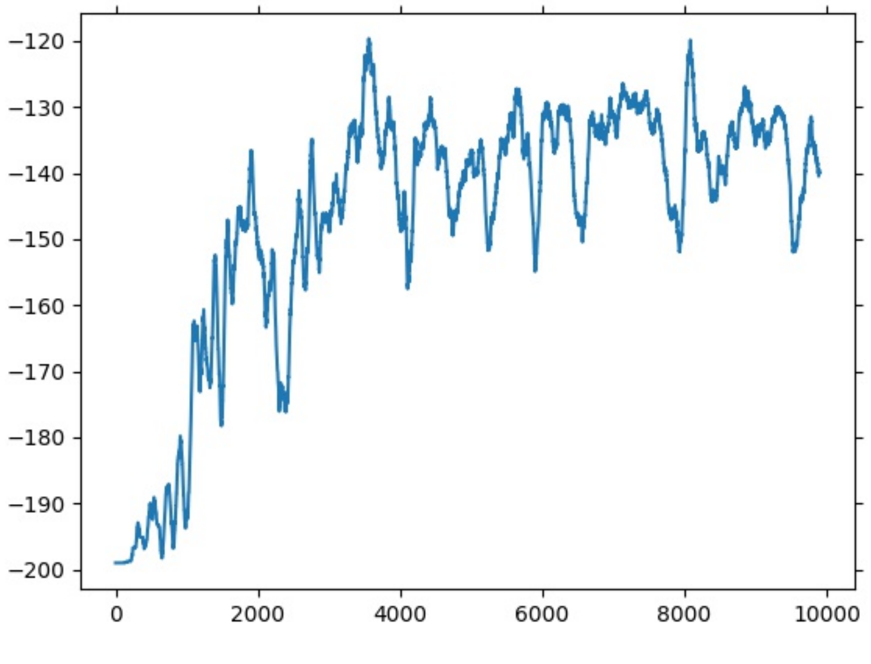
In this task, we were told to improve our reward using concepts like RBF and tile coding. I initially went with the tile coding approach.

I called the kx of task 1, as kx1 for task 2, as I wanted to discretise them differently as my code demanded. Similarly for v.

I decided that instead of just returning the current state I was in, I would use a set of 5 tiles to denote if I was in the first 1/5th or 2nd 1/5th of the tile etc. Hence I returned the state as a list of 2 lists. One list each for x, and v, containing 5 elements, and which returned state values as tile coding taught in class.

For example, if I was in the first 1/5th of the state ix, I would return [ix, ix, ix, ix-1, ix-1]
If I was in the third 1/5th of the state I would return [ix, ix, ix, ix, ix]. If I was in the last 1/5th I would return [ix+1, ix+1, ix, ix, ix]. Similarly for v. Then, in my choose action function I would take a weighted sum(linear combination) of the w values of these states , and check which action gives the largest value for this sum. Then I would choose that action, and update my weights accordingly with the same update weights function as in task 1.

But I did not get considerably better results in tile coding. If anything, it just took more computing power without aiding results, as I was just discretising the states even further more in a way. Whereas this problem seems like one where neighbouring states have the same behaviour, and the behaviour within each state is not super intricate. Thus as you can see in my code, I have commented out a large portion of my code, which I had implemented for performing tile coding.



The weights I used in my 5 tiles were 0.25, 0.5, 1, 0.5, 0.25. (The third tile was the one always corresponding to the current state, as I considered that as my main, central tile).

The above was the plot I got when I performed tile coding.

Plot analysis- Thus we see that my results in tile coding were way better than that of my tabular Sarsa(0). There was a way more evident net increase in average reward. We see that there was a very evident increase in reward in the graph, as our training iterations increased, right from the start. The graph does not vary as much towards the end either, like task 1's plot did. Yet I felt the results weren't as good. Also, my rewards at the end of 10000 iterations was not above -130, which was required in task 2.

Hence I decided to implement task 2 using RBF. I searched online what an RBF kernel was, and I found that it was also a weighted sum of the surrounding states, but as part of a Gaussian distribution, with our current state at the center and surrounding states being multiplied by a weight corresponding to the gaussian curve at distance equal to the sum of squared difference between the state numbers.

Hence even in this case I returned a list of length 5 for both x and v, in get_better_features, but this time I returned [ix-2, ix-1, ix, ix+1, ix+2], i.e. the set of all the surrounding states. If any of the surrounding states became less than 0, I gave them the value 0, if they became more than kx1-1, I gave them the value kx1-1, so that we stay within our state range.

Then inside choose action, I took a weighted sum according to the gaussian distance from the center state [ix, iv] for each of the actions, and I chose the action that gave the largest value for this sum. The formula I used for weight of a state I away from current x state, and j away from current v state, was exp(-(I^2+j^2)/(2*sigma^2)), as described for the RBF kernel concept.

I only considered terms until a sum square total of 5 states away from [ix, iv] (both distance and velocity state difference's square sum should be less than equal to 5). I.e. I only took states upto 2 states away from one of the state parameters, and 1 state away from the other parameter (And vice versa).

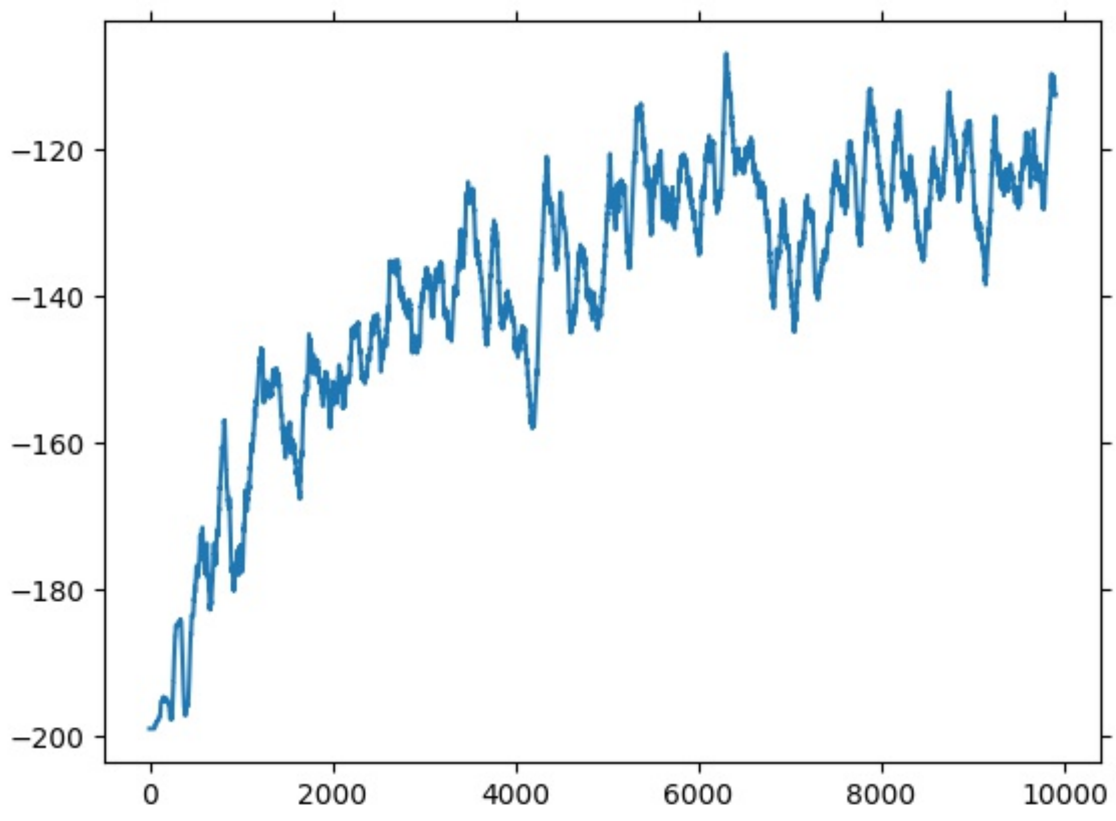
At every step, I only updated the w values for the state [ix, iv] and not for any of the surrounding states.

I varied the values of learning rate, and exploration rate, and sigma to get the best results I possibly could. After a large amount on finetuning, there are the results I got-

Hence as we can see, I got very decent results using this method. I consistently got average rewards of more than -120, when I tested my results with various random seeds, and test data.

Hence I was able to improve my model performance by a lot using the concept of an RBF kernel, and it made it way more consistent, i.e. not too much variation in the final average reward towards the end of the 10000 iterations.

Plot analysis- Below is my plot for the rbf kernel method. I saw that here the graph varied way lesser than my tile coding graph too. It was the least erratic graph of all 3 methods. There was an even more evident increase in reward, right from the start in the graph, and I also managed to get very good rewards of above -120, at the end of 10000 iterations. Thus I found this method to work the best for the mountain car problem.



My code mountain_car.py is for the rbf kernel method, with most of the tile coding part commented out or slightly edited. My perfectly running tile coding method's complete code is as below-

```
'''
    1. Don't delete anything which is already there in code.
    2. you can create your helper functions to solve the task and call them.
    3. Don't change the name of already existing functions.
    4. Don't change the argument of any function.
    5. Don't import any other python modules.
    6. Find in-line function comments.

'''


import gym
import numpy as np
import math
import time
import argparse
import matplotlib.pyplot as plt

kv=10
kx=10
kx1=22
kv1=16

class sarsaAgent():
    '''
    - constructor: graded
    - Don't change the argument of constructor.
    - You need to initialize epsilon_T1, epsilon_T2, learning_rate_T1, learning_rate_T2 and
weight_T1, weights_T2 for task-1 and task-2 respectively.
    - Use constant values for epsilon_T1, epsilon_T2, learning_rate_T1, learning_rate_T2.
    - You can add more instance variable if you feel like.
    - upper bound and lower bound are for the state (position, velocity).
    - Don't change the number of training and testing episodes.
    '''




    def __init__(self):
        self.env = gym.make('MountainCar-v0')
        self.epsilon_T1 = 0.01
        self.epsilon_T2 = 0.01
        self.learning_rate_T1 = 0.04
        self.learning_rate_T2 = 0.18
        self.weights_T1 = [[[0 for j in range(kx)] for l in range(kv)] for x in range(3)]
        self.weights_T2 = [[[0 for j in range(kx1)] for l in range(kv1)] for x in range(3)]
        self.discount = 1.0
        self.train_num_episodes = 10000
        self.test_num_episodes = 100
        self.upper_bounds = [self.env.observation_space.high[0], self.env.observation_space.high[1]]
        self.lower_bounds = [self.env.observation_space.low[0], self.env.observation_space.low[1]]

    '''
    - get_table_features: Graded
    - Use this function to solve the Task-1
    - It should return representation of state.
    '''
```

```
def get_table_features(self, obs):

    v= obs[1]
    x= obs[0]
    sv= int((v+0.07)*kv/0.14)
    sx= int((x+0.9+0.3)*kx/1.80)


    return [sx, sv]

'''
- get_better_features: Graded
- Use this function to solve the Task-2
- It should return representation of state.
'''

def get_better_features(self, obs):





    v= obs[1]
    x= obs[0]

    sv= ((v+0.07)*kv1/0.14)
    sx= ((x+0.3+0.9)*kx1/1.80)
    iv= int(sv)
    ix= int(sx)

    fv= sv- iv
    fx= sx- ix

    """
    lessv=[0,0]#,0,0]
    morev=[0,0]#,0,0]
    lessx=[0,0]#,0,0]
    morex=[0,0]#,0,0]
    rx=[]
    rv=[]


    lessx[0]= ix-1
    lessx[1]= ix-2
    #lessx[2]= ix-3
    #lessx[3]= ix-4
    morex[0]= ix+1
    morex[1]= ix+2
    #morex[2]= ix+3
    #morex[3]= ix+4

    lessv[0]= iv-1
    lessv[1]= iv-2
    #lessv[2]= iv-3
    #lessv[3]= iv-4
    morev[0]= iv+1
    morev[1]= iv+2
    #morev[2]= iv+3
    #morev[3]= iv+4
```

```
for i in range(len(lessx)):
    if lessx[i]<0:
        lessx[i]=0

for i in range(len(morex)):
    if morex[i]>kx1-1:
        morex[i]=kx1-1

for i in range(len(lessv)):
    if lessv[i]<0:
        lessv[i]=0

for i in range(len(morev)):
    if morev[i]>kv1-1:
        morev[i]=kv1-1

"""








if ix==0:
    lessx= ix
    morex= ix+1

elif ix==kx-1:
    lessx= ix-1
    morex= ix

else:

    lessx= ix-1
    morex= ix+1


if iv==0:
    lessv= iv
    morev= iv+1

elif iv==kv-1:
    lessv= iv-1
    morev= iv

else:

    lessv= iv-1
    morev= iv+1


if fv<=0.2:
    rv= [iv, iv, iv, lessv, lessv]
elif fv<=0.4:
    rv= [iv, iv, iv, iv, lessv]
elif fv<=0.6:
    rv= [iv, iv, iv, iv, iv]
elif fv<=0.8:
    rv= [morev, iv, iv, iv, iv]
```

```
        else:
            rv= [morev, morev, iv, iv, iv]


        if fx<=0.2:
            rx= [ix, ix, ix, lessx, lessx]
        elif fx<=0.4:
            rx= [ix, ix, ix, ix, lessx]
        elif fx<=0.6:
            rx= [ix, ix, ix, ix, ix]
        elif fx<=0.8:
            rx= [morex, ix, ix, ix, ix]
        else:
            rx= [morex, morex, ix, ix, ix]





        """
        rx=[lessx[1], lessx[0], ix, morex[0], morex[1]]
        rv=[lessv[1], lessv[0] ,iv, morev[0], morev[1]]
        """

        return [rx, rv]




        #return None

    '''
    - choose_action: Graded.
    - Implement this function in such a way that it will be common for both task-1 and task-2.
    - This function should return a valid action.
    - state representation, weights, epsilon are set according to the task. you need not worry about
that.
    '''

    def choose_action(self, state, weights, epsilon):

        arr= np.array(state)
        arrsz= len(np.shape(arr))

        if arrsz==1:



            k= len(weights[0])



            maxa=0
            maxv= float('-inf')
            for i in range(3):
              if weights[i][state[1]][state[0]]>maxv:
                maxv= weights[i][state[1]][state[0]]
                maxa= i

            epp= np.random.rand()

            #print(maxa)   #deleteee
            if epp<epsilon:
              return int(np.random.rand()*3)

            else:
              return maxa


        else:


            sig= 1

            maxa=0
            maxv= float('-inf')
            for i in range(3):
                #sumaa=0
                #sumaa1=  weights[i][state[1][2]][state[0][2]] + (  np.exp(-1/(2*sig*sig))* (weights[i][state[1]
[2]][state[0][1]] +  weights[i][state[1][1]][state[0][2]] +  weights[i][state[1][2]][state[0][3]] +  weights[i]
[state[1][3]][state[0][2]]))    +      (  np.exp(-2/(2*sig*sig))* (weights[i][state[1][3]][state[0][1]] +
weights[i][state[1][1]][state[0][3]] +  weights[i][state[1][3]][state[0][3]] +  weights[i][state[1][1]][state[0]
[1]]))   +      (  np.exp(-4/(2*sig*sig))* (weights[i][state[1][4]][state[0][2]] +  weights[i][state[1][2]]
[state[0][4]] +  weights[i][state[1][0]][state[0][2]] +  weights[i][state[1][2]][state[0][0]]))      +
(  np.exp(-5/(2*sig*sig))* (weights[i][state[1][4]][state[0][1]] +  weights[i][state[1][0]][state[0][1]] +
weights[i][state[1][4]][state[0][3]] +  weights[i][state[1][0]][state[0][3]]  +    weights[i][state[1][1]]
[state[0][4]] +  weights[i][state[1][1]][state[0][0]] +  weights[i][state[1][3]][state[0][4]] +  weights[i]
[state[1][3]][state[0][0]]                 ))
                #for j in range(5):
                #    for l in range(5):
                #        for m in range(5):
                #            if (np.square(2-l)+np.square(2-m))==j:
                #                sumaa= sumaa + (  np.exp(-1*j/(2*sig*sig))* weights[i][state[1][l]][state[0][m]]  )

                sumaa1= weights[i][state[1][2]][state[0][2]] + (0.5*(weights[i][state[1][1]][state[0][1]]
+weights[i][state[1][3]][state[0][3]])) +   (0.25*(weights[i][state[1][0]][state[0][0]]+weights[i][state[1][4]]
[state[0][4]]))
                if sumaa1>maxv:
                    maxa= i
                    maxv= sumaa1



            epp= np.random.rand()

            #print(maxa)   #deleteee

            if epp<epsilon:
              return int(np.random.rand()*3)

            else:
              return maxa





    '''
    - sarsa_update: Graded.
    - Implement this function in such a way that it will be common for both task-1 and task-2.
    - This function will return the updated weights.
```

```
    - use sarsa(0) update as taught in class.
    - state representation, new state representation, weights, learning rate are set according to the
task i.e. task-1 or task-2.
    '''

    def sarsa_update(self, state, action, reward, new_state, new_action, learning_rate, weights):

        arr= np.array(state)
        arrsz= len(np.shape(arr))

        if arrsz==1:


            k= len(weights[action])


            weights[action][state[1]][state[0]]= weights[action][state[1]][state[0]]*(1- learning_rate) +
learning_rate*(reward + weights[new_action][new_state[1]][new_state[0]])


            return weights

        else:

            weights[action][state[1][2]][state[0][2]]= weights[action][state[1][2]][state[0][2]]*(1-
learning_rate) + learning_rate*(reward + weights[new_action][new_state[1][2]][new_state[0][2]])
            #print(weights)

            return weights



    '''
    - train: Ungraded.
    - Don't change anything in this function.

    '''

    def train(self, task='T1'):
        if (task == 'T1'):
            get_features = self.get_table_features
            weights = self.weights_T1
            epsilon = self.epsilon_T1
            learning_rate = self.learning_rate_T1
        else:
            get_features = self.get_better_features
            weights = self.weights_T2
            epsilon = self.epsilon_T2
            learning_rate = self.learning_rate_T2
        reward_list = []
        plt.clf()
        plt.cla()
        for e in range(self.train_num_episodes):
            current_state = get_features(self.env.reset())
            done = False
            t = 0
            new_action = self.choose_action(current_state, weights, epsilon)
            while not done:
                action = new_action
                obs, reward, done, _ = self.env.step(action)
            new_state = get_features(obs)
            new_action = self.choose_action(new_state, weights, epsilon)
            weights = self.sarsa_update(current_state, action, reward, new_state, new_action, 
learning_rate,
                                        weights)
            current_state = new_state
            if done:
                reward_list.append(-t)
                break
            t += 1
        self.save_data(task)
        reward_list=[np.mean(reward_list[i-100:i]) for i in range(100,len(reward_list))]
        plt.plot(reward_list)
        plt.savefig(task + '.jpg')

    '''
       - load_data: Ungraded.
       - Don't change anything in this function.
    '''

    def load_data(self, task):
        return np.load(task + '.npy')

    '''
       - save_data: Ungraded.
       - Don't change anything in this function.
    '''

    def save_data(self, task):
        if (task == 'T1'):
            with open(task + '.npy', 'wb') as f:
                np.save(f, self.weights_T1)
            f.close()
        else:
            with open(task + '.npy', 'wb') as f:
                np.save(f, self.weights_T2)
            f.close()

    '''
    - test: Ungraded.
    - Don't change anything in this function.
    '''

    def test(self, task='T1'):
        if (task == 'T1'):
            get_features = self.get_table_features
        else:
            get_features = self.get_better_features
        weights = self.load_data(task)
        reward_list = []
        for e in range(self.test_num_episodes):
            current_state = get_features(self.env.reset())
            done = False
            t = 0
            while not done:
                action = self.choose_action(current_state, weights, 0)
                obs, reward, done, _ = self.env.step(action)
                new_state = get_features(obs)
                current_state = new_state
                if done:
                    reward_list.append(-1.0 * t)
                    break
                t += 1
        return float(np.mean(reward_list))


if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--task", required=True,
       help="first operand", choices={"T1", "T2"})
    ap.add_argument("--train", required=True,
       help="second operand", choices={"0", "1"})
    args = vars(ap.parse_args())
    task=args['task']
    train=int(args['train'])
    agent = sarsaAgent()
    agent.env.seed(0)
    np.random.seed(0)
    agent.env.action_space.seed(0)
    if(train):
        agent.train(task)
    else:
        print(agent.test(task))
```