Exam 2: Advanced AI

Caleb Ehrisman

11/30/2022

Overview

The objective of this assignment is to implement a Fourier Basis SARSA(λ) algorithm to solve the mountain car problem.

The mountain car is a popular problem within Reinforcement Learning where an under-powered car is stuck is a valley. To get out the car must build momentum by accelerating left and right until it gains enough to reach the peak.

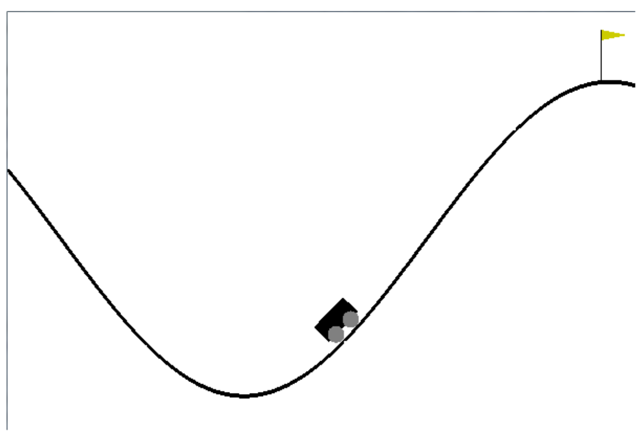

Figure 1: Screenshot of a graphic of mountain car

Approach and Implementation

The first step in the approach was to make sure I always had a beer in hand to ensure proper usage of Balmer's Peak. In the solution implementation, True Online SARSA(λ) was used as the learning agent.

True online Sarsa(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$
  Loop for each step of episode:
  | Take action A, observe R, S'
  | Choose $A' \sim \pi(\cdot|S')$ or ε-greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
  | $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
  | $Q \leftarrow \mathbf{w}^\top \mathbf{x}$
  | $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$
  | $\delta \leftarrow R + \gamma Q' - Q$
  | $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\right)\mathbf{x}$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
  | $Q_{old} \leftarrow Q'$
  | $\mathbf{x} \leftarrow \mathbf{x}'$
  | $A \leftarrow A'$
  until S' is terminal

Figure 2: SARSA(λ) from Sutton and Barto RL book

Where **w** is the weight vector and **x** is the feature approximation vector for a given state and action. The **z** is the trace vector. The hyper parameters are as follows: α - learning rate, λ - decay rate of trace, and ϵ - value to choose actions based on probability.

For the approximator, the one from the paper linked in the assignment ,Value Function Approximation in Reinforcement Learning using the Fourier Basis, was implemented to estimate the Q values for a state-action pairs. Using N(dimensions of the state space) and the M(order of the Fourier function) to create a matrix of $(M+1)^2$ x N for the co-efficients. To get the values of this matrix it is required to call get_features within the

FourierBasis class. This function uses equation (1) as shown in the book.

$$\phi_i(x) = cos(\pi \mathbf{c}^i \cdot \mathbf{x}) \tag{1}$$

Results

The program has several options to run from the terminal all handled through argparse. The optional arguments are:

- --render
  type=str True if want to run with graphics.
- --order
  type=int default=3, Choose any order for the fourier basis
- --num_epochs
  type=int default=1000, Choose any number of epochs to run
- --fourier
  type=str default=True, if false then run standard SARSA(λ) without Fourier basis
- --file
  type=str default='weights.npy', File path to save the weights after training. Must be a '.npy' file.
- --train
  type=str default=True, If false will load file and run without updates
- --eval
  type=str default=False, Will run multiple bases and plot

Example usage: C:\> py main.py --render True --order 5 --num_epoch 5000

With the file loading and saving it makes it really easy to train the weights and save the values into a file. The run method will load these values and then allow you to simply run a greedy model on the weights. If the weights were trained properly then it should quickly reach the terminal state and consistently do so.

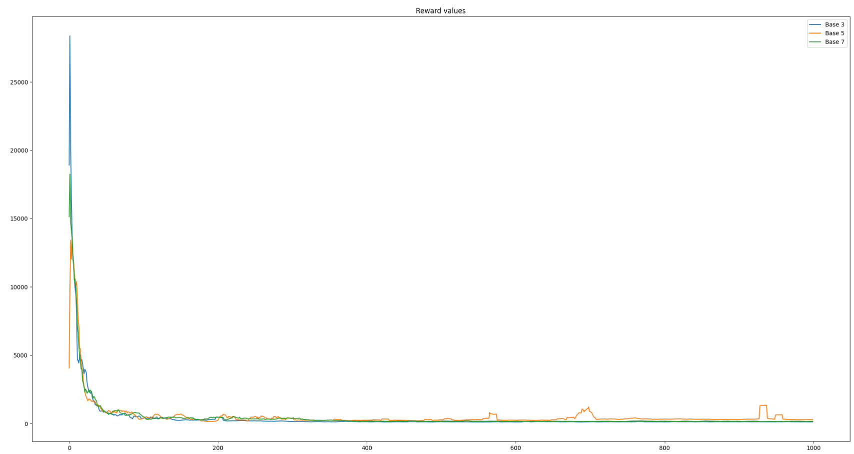

Figure 3: Learning Curves of bases 3,5,7

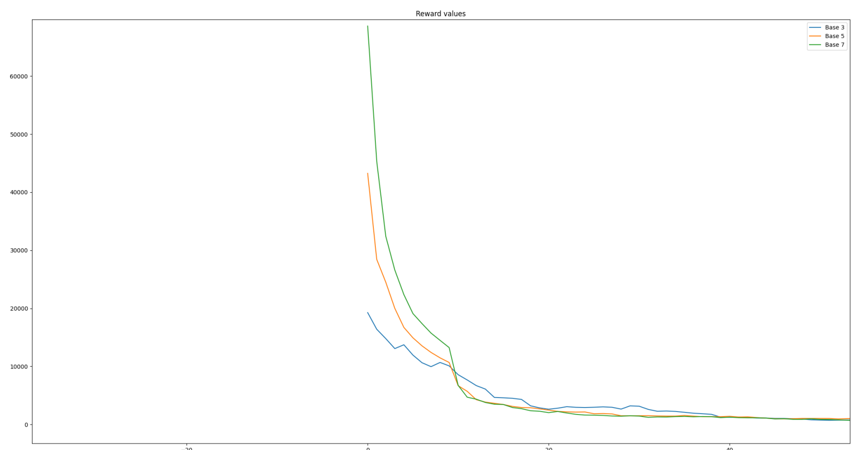

Figure 4: The first 50 episodes

Figs 3, 4 are the learning curves of the different Fourier bases. The lines are batched averages of 10 episodes each to present the lines in a smooth way while still clearly seeing how the curve is trending. It appears that it converges roughly 50-100 episodes into training with values of α - 0.01, λ - 0.9, and ϵ - 0.05.

The surface plots of the Q_value estimation form the Fourier Basis Approximation

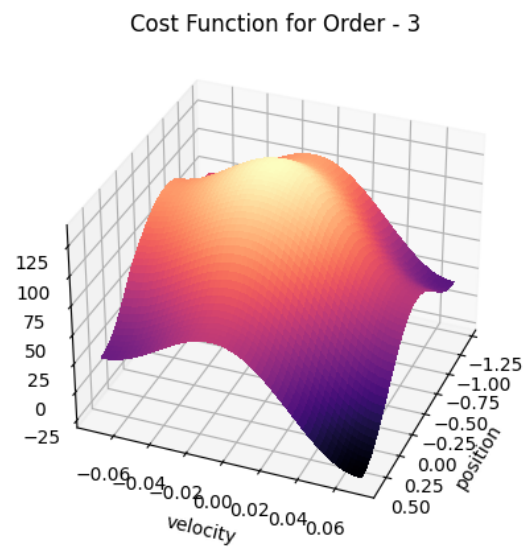

Figure 5: Surface plot of base 3

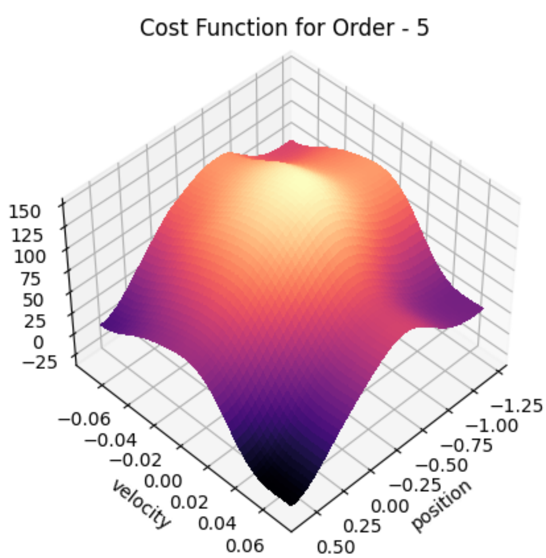

Figure 6: Surface plot of base 5

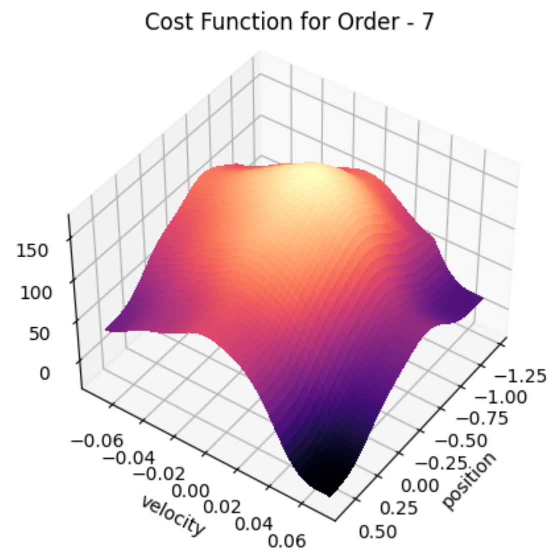

Figure 7: Surface plot of base 7

Question

The Mountain Car contains a negative step reward and a zero goal reward. What would happen if γ was less than 1 and the solution was many steps long?

To test what happens when the gamma is less than 1.0, simply just change the gamma value and let it run for infinite steps. As can be seen in Fig 8. the learning curve is interesting. The gamma seems to pose an issue where some runs will end up being absolutely horrible. It will still converge; however, the solution is still unstable as can be seen in Fig 9 with some runs going to 40,000 or more. With the gamma being 1.0 the worst runs still hardly reach over 20,000 steps, so reducing the gamma below 1.0 has negative effects on the training.

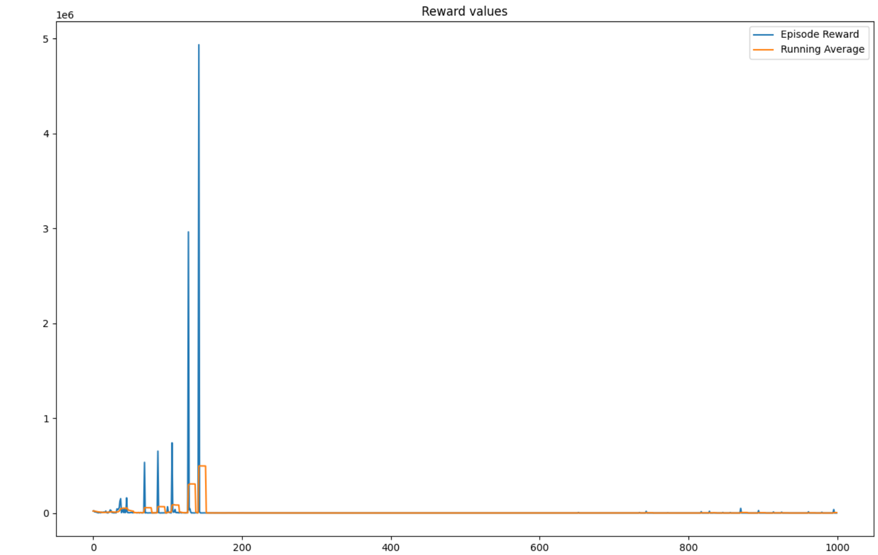

Figure 8: Learning Curves basis 3 and gamma 0.95

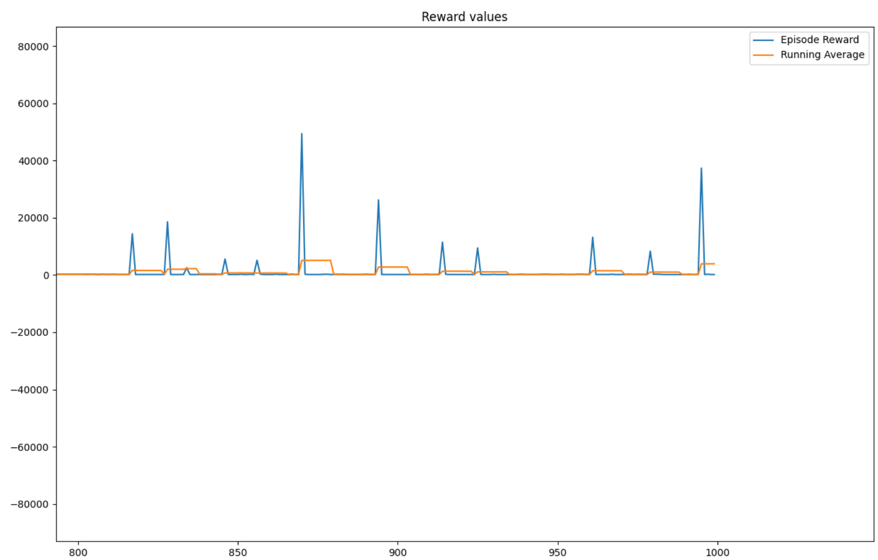

Figure 9: Last 200 episodes.

What would happen if we had a zero step cost and a positive goal reward, for the case where $\gamma = 1$, and the case where $\gamma < 1$? In the case of just changing the time step reward from -1 to 0 and the terminal state reward to 1 instead of 0 increases the training time significantly and it struggles to converge (took over an hour to reach 600 epochs). When setting the reward to 100 for the terminal state it was able to learn a solution early on but then would diverge later on as can be seen in Fig 10 and Fig 11. And the early solution was not close to optimal. The issue is that it is not getting punished for taking bad routes and sometimes it creates a path that just loops on itself and it will not correct due to no bad reward. So trying to test thing through code is a hassle and very unstable. Some runs will actually converge and others will not and seem to fall in a loop as it will sit on an episode for upwards of 30 min. And changing the gamma value will not remedy this as we saw in the Fig. 10 it will generally get worse with more episodes.

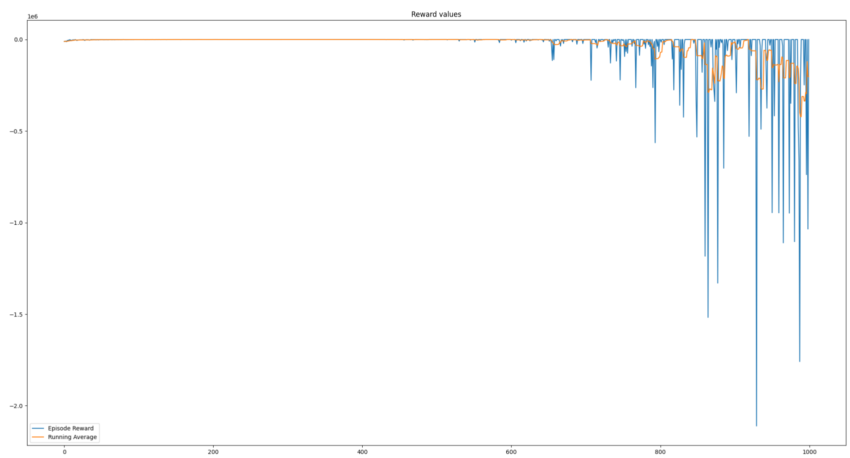

Figure 10: Learning Curves basis 3 and 0 step reward and 100 terminal state reward

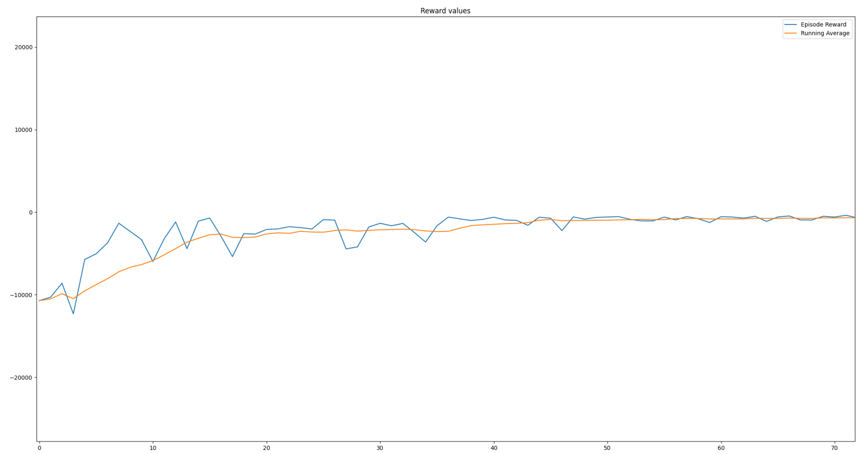

Figure 11: Early learning curve

Final Remarks

The github repo that hosts the code is at GitHub

The code is also included in a zip submitted with the pdf and below in the appendix. There is a default weight vector stored in 'default.npy'. If desired to run then do C:\> py main.py --render True --file default

1 Appendix

```
'''
Author: Caleb Ehrisman
Course- Advanced AI CSC-549
Assignment - Programming Assignment #3 - Mountain Car

Tasks
 - Implement Sarsa(lambda) to solve mountain car problem
 - Use Linear Function Approximation with Fourier Basis
    functions
 - Show different learning curves for 3rd, 5th, and 7th
    order Fourier bases
 - Create surface plot of the value function
```

```
 - Answer short response question
'''

import gym
import matplotlib

from agent import Agent
import argparse
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

"""
    parse gathers command line arguments.


    :return: a list of all parsed arguments
    """


def parse():
    parser = argparse.ArgumentParser()
    parser.add_argument('--render', type=str, help='Specify to run simulation or not')
    parser.add_argument('--order', type=int, help='Choose order for fourier basis', default=3)
    parser.add_argument('--num_epochs', type=int, help='Choose number of epochs', default=1000)
    parser.add_argument('--fourier', type=str, help='Choose to use fourier', default=True)
    parser.add_argument('--file', type=str,
                        help='File path to save weights to. Must be given with .npy extension', default='weights.npy')
    parser.add_argument('--train', type=str, help='Choose if training or running', default='True')
    parser.add_argument('--eval', type=str, default='False')
    return parser.parse_args()


if __name__ == "__main__":
    args = parse()
```

```

    if args.render == "True":
        env = gym.make("MountainCar-v0", render_mode="human")

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

    file = args.file

    n = args.num_epochs
    if args.fourier == 'False':
        agent = Agent(env, file, fourier=False, order=3)
    else:
        agent = Agent(env, file, order=3)

    if args.eval == 'False':
        if args.train == 'True':
            rewards, avg, learner = agent.learn(n)
        else:
            rewards, avg = agent.run(n)

        fig, ax = plt.subplots(figsize=(10, 4))
        plt.plot(np.negative(rewards), label='Episode Reward')
        plt.plot(np.negative(avg), label='Running Average')
        ax.set_title("Reward values")
        plt.legend()
        plt.show()

        rewards = []
        base = [3, 5, ]

        rewards.append(avg)

        fig, ax = plt.subplots(figsize=(10, 4))
        plt.plot(np.negative(rewards[0]), label='Base 3')
        plt.plot(np.negative(rewards[1]), label='Base 5')
        plt.plot(np.negative(rewards[2]), label='Base 7')
        ax.set_title("Reward values")
        plt.legend()
        plt.show()
```

```

        low = env.observation_space.low
        high = env.observation_space.high
        difference = high - low

        x_axis = np.linspace(low[0], high[0])
        y_axis = np.linspace(low[1], high[1])
        x_axis, y_axis = np.meshgrid(x_axis, y_axis)
        z_axis = np.zeros(x_axis.shape)

        for i in range(0, z_axis.shape[0]):
            for j in range(0, z_axis.shape[1]):
                s = [(x_axis[i, j] - low[0]) / (high[0] - low[0]), (y_axis[i, j] - low[1]) / (high[1] - low[1])]
                (zq, _) = learner.best_action(s)
                z_axis[i, j] = -1.0 * zq

            fig = plt.figure()
            ax = plt.axes(projection='3d')
            ax.plot_surface(x_axis, y_axis, z_axis, cmap=matplotlib.cm.get_cmap("magma"))
            ax.set_xlabel('position')
            ax.set_ylabel('velocity')
            ax.set_title('Cost Function for Order - ' + str(n))
            plt.show()

    else:
        rewards = []
        base = [3, 5, 7]
        learner = []
        for i in range(3):
            agent = Agent(env, file, order=base[i])
            reward, avg, temp_learner = agent.learn(1000)
            rewards.append(avg)
            learner.append(temp_learner)

        fig, ax = plt.subplots(figsize=(10, 4))
        plt.plot(np.negative(rewards[0]), label='Base 3')
        plt.plot(np.negative(rewards[1]), label='Base 5')
        plt.plot(np.negative(rewards[2]), label='Base 7')
```

```
        ax.set_title("Reward values")
        plt.legend()
        plt.show()

        low = env.observation_space.low
        high = env.observation_space.high
        difference = high - low

        x_axis = np.linspace(low[0], high[0])
        y_axis = np.linspace(low[1], high[1])
        x_axis, y_axis = np.meshgrid(x_axis, y_axis)
        z_axis = np.zeros(x_axis.shape)

        for b in range(3):
            for i in range(0, z_axis.shape[0]):
                for j in range(0, z_axis.shape[1]):
                    s = [(x_axis[i, j] - low[0]) / (high[0] - low[0]), (y_axis[i, j] - low[1]) / (high[1] - low[1])]
                    (zq, _) = learner[b].best_action(s)
                    z_axis[i, j] = -1.0 * zq

            fig = plt.figure()
            ax = plt.axes(projection='3d')
            ax.plot_surface(x_axis, y_axis, z_axis, cmap=matplotlib.cm.get_cmap("magma"))
            ax.set_xlabel('position')
            ax.set_ylabel('velocity')
            ax.set_title('Cost Function for Order - ' + str(base[b]))
            plt.show()
            '''
Author: Caleb Ehrisman
Course- Advanced AI CSC-549
Assignment - Programming Assignment #3 - Mountain Car

This file contains the code to implement the SARSA(lambda) algorithm.

All functions needed by solely the agent are included as member functions of class Agent
'''
```

```python
import numpy as np
from fourier_basis import FourierBasis
from sarsalambdaFA import SarsaLambdaFA
from sarsa import Sarsa
import os.path

ALPHA = 0.0001
GAMMA = 1
EPSILON = 0.5
LAMBDA = 0.9


class Agent:

    def __init__(self, environment, file, fourier=True, order=3, runs=1, gamma=0.001):
        """
                init is the constructor for the Agent class.

                :param environment
                :return None
        """
        self.runs = runs
        self.order = order
        self.env = environment
        self.gamma = gamma
        self.num_actions = self.env.action_space.n
        self.state_dims = self.env.observation_space.shape[0]
        self.fourier = fourier
        self.epoch_rewards = []
        self.epoch_rewards_table = {'ep': [], 'avg': [], 'min': [], 'max': []}
        self.epoch_max_pos = []
        self.file = file

    def learn(self, num_epochs):
        """
            Agent.learn does the actual stepping through and exploring the environment and then updates the Q_table if
```

```
            using traditional SARSA and updates the weight and lambda vectors is using a fourier basis

            :param num_epochs
            :return None
            """
        for run in range(0, self.runs):
            fb = FourierBasis(state_space=self.env.observation_space.shape[0], order=self.order)
            if self.fourier:
                learner = SarsaLambdaFA(fa=fb, num_actions=self.num_actions, alpha=0.0001, epsilon=0.8)
            else:
                learner = Sarsa(environment=self.env)

            for i in range(num_epochs):

                learner.epsilon *= .99
                if self.fourier:
                    learner.z = np.zeros(learner.w.shape)
                state, _ = self.env.reset()
                if self.fourier:
                    state = (state - self.env.observation_space.low) / (self.env.observation_space.high - self.env.observation_space.low)
                else:
                    state = learner.discretized_env_state(state)
                    learner.E_table = learner.create_q_table()
                # steps = 0
                action = learner.action(state)
                done = False
                reward_sum = 0

                while not done:
                    next_state, reward, done, info, _ = self.env.step(action)
                   # reward += 1
                   # if done:
```

```
                   #      reward = 100
                    if self.fourier:
                        next_state = (next_state - self.env.observation_space.low) / (
                                    self.env.observation_space.high - self.env.observation_space.low)
                    else:
                        next_state = learner.discretized_env_state(next_state)

                    next_action = learner.action(next_state)

                    learner.update(state, action, reward, next_state, done, next_action)

                    # steps += 1
                    state = next_state
                    action = next_action
                    reward_sum += reward

                #   Append max position data and reward data for evaluation
                self.epoch_rewards.append(reward_sum)

                self.terminal_output(i)
        np.save(self.file, learner.w)
        return self.epoch_rewards, self.epoch_rewards_table['avg'], learner

    def run(self, num_epochs):
        """
            Agent.run uses a pre-trained set of weights to greedily choose actions.

            :param num_epochs
            :return None
            """

        if os.path.exists(self.file):
```

```
            w = np.load(self.file)
        else:
            print("Error loading file. Not found.")
            return

        fb = FourierBasis(state_space=self.env.observation_space.shape[0], order=self.order)
        learner = SarsaLambdaFA(fa=fb, num_actions=self.num_actions, alpha=0.0, epsilon=0.0)
        learner.w = w

        for i in range(num_epochs):
            state, _ = self.env.reset()

            state = (state - self.env.observation_space.low) / (
                        self.env.observation_space.high - self.env.observation_space.low)

            action = learner.action(state)
            done = False
            reward_sum = 0

            while not done:
                next_state, reward, done, info, _ = self.env.step(action)
                print(reward)
                next_state = (next_state - self.env.observation_space.low) / (
                        self.env.observation_space.high - self.env.observation_space.low)

                next_action = learner.action(next_state)
                action = next_action
                reward_sum += reward

            #  Append max position data and reward data for evaluation
            self.epoch_rewards.append(reward_sum)

            self.terminal_output(i)

```

```
        return self.epoch_rewards, self.epoch_rewards_table['avg']

    def terminal_output(self, i):
        # Terminal Output for stats of each epoch
        avg_reward = sum(self.epoch_rewards[-10:]) / len(self.epoch_rewards[-10:])
        self.epoch_rewards_table['ep'].append(i)
        self.epoch_rewards_table['avg'].append(avg_reward)
        self.epoch_rewards_table['min'].append(min(self.epoch_rewards[:]))
        self.epoch_rewards_table['max'].append(max(self.epoch_rewards[:]))

        print(f"Epoch - {i}\t| avg: {avg_reward:.2f}\t| min: {min(self.epoch_rewards[-1:]):.2f}"
              f"\t| max: {max(self.epoch_rewards[-1:]):.2f}")

'''
Author: Caleb Ehrisman
Course- Advanced AI CSC-549
Assignment - Programming Assignment #3 - Mountain Car

This file contains the code to implement the SARSA(lambda) with a function approximator.

'''
class SarsaLambdaFA:
    def __init__(self, fa, num_actions=None, alpha=0.01, gamma=1.0, lamb=0.9, epsilon=0.5):
        self.gamma = gamma
        self.lamb = lamb
        self.epsilon = epsilon
        self.alpha = alpha
        self.num_actions = num_actions
        self.fourier_basis = []

        for i in range(0, self.num_actions):
            self.fourier_basis.append(copy.deepcopy(fa))

```

```
        self.w = np.zeros([self.fourier_basis[0].coeff.shape
            [0], num_actions])
        self.z = np.zeros(self.w.shape)

        self.w[0, :] = 0.0

    def action(self, state):
        """
                Agent.action determines what action to take
                    based on state

                :param state:
                :return action
        """
        if np.random.uniform(0, 1) < self.epsilon:
            return random.randrange(0, self.num_actions)

        best = -math.inf
        best_actions = []
        for a in range(0, self.num_actions):
            q = self.Q(state, a)
            if math.isclose(q, best):
                best_actions.append(a)
            elif q > best:
                best = q
                best_actions = [a]

        return random.choice(best_actions)

    def Q(self, state, action):
        return np.dot(self.w[:, action], self.fourier_basis[
            action].get_features(state))

    def best_action(self, state):

        best = -math.inf
        best_action = 0
        for a in range(0, self.num_actions):
            q = self.Q(state, a)
            if q > best:
                best = q
                best_action = a
```

```
        return best, best_action

    def update(self, state, action, reward, next_state, done, next_action=None):
        """
            Agent.update updates the Q table based on the SARSA algorithm. It also updates the trace table

            :param next_action:
            :param done:
            :param next_state:
            :param state:
            :param action:
            :param reward
            :return None
        """

        delta = reward - self.Q(state, action)

        if not done:
            if next_action is not None:
                delta += self.gamma * self.Q(next_state, next_action)
            else:
                q_dot, next_action = self.best_action(next_state)
                delta += self.gamma * self.best_action(q_dot)

        phi = self.fourier_basis[action].get_features(state)

        for a in range(0, self.num_actions):
            self.z[:, a] *= self.gamma * self.lamb
            if a == action:
                self.z[:, a] += phi
            self.w[:, a] += self.alpha * delta * self.z[:, a]

        return delta

        '''
```

```
Author: Caleb Ehrisman
Course- Advanced AI CSC-549
Assignment - Programming Assignment #3 - Mountain Car
This file contains the code to implement the SARSA(lambda)
    algorithm.
All functions needed by solely the agent are included as
    member functions of class Agent
'''
import numpy as np

ALPHA = 0.1
GAMMA = 1
EPSILON = 0.5
LAMBDA = 0.9


class Sarsa:
    def __init__(self, environment, gamma=1.0, epsilon=0.5,
        alpha=0.0001, lamb=0.9):
        """
                init is the constructor for the Agent class.
                :param environment
                :return None
        """
        self.env = environment
        self.E_table = self.create_q_table()
        self.Q_table = self.create_q_table()
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.lamb = lamb
        self.epoch_rewards = []
        self.epoch_rewards_table = {'ep': [], 'avg': [], '
            min': [], 'max': []}
        self.epoch_max_pos = []

    def create_q_table(self):
        """
                Agent.create_q_table creates the Q table
                    that fits all states
                :return np.array of [x_lim][y_lim][
                    num_actions]
```

```
        """
        high = self.env.observation_space.high
        low = self.env.observation_space.low
        num_states = (high - low) * np.array([10, 100])
        num_states = np.round(num_states, 0).astype(int) + 1
        num_actions = self.env.action_space.n
        return np.zeros([num_states[0], num_states[1], num_actions])

    def action(self, state):
        """
                Agent.action determines what action to take based on state
                :param state
                :return action
        """
        if np.random.uniform(0, 1) < EPSILON:
            action = self.env.action_space.sample()
        else:
            # disc_state = self.discretized_env_state(state)
            action = np.argmax(self.Q_table[state[0], state[1]])
        return action

    def update(self, state, action, reward, next_state, done, next_action):
        """
            Agent.update updates the Q table based on the SARSA algorithm. It also updates the trace table
            :param done:
            :param state
            :param action
            :param reward
            :param next_state
            :param next_action
            :return None
        """
        target = reward + self.gamma * self.Q_table[next_state[0], next_state[1], next_action]

```

```
        error = target - self.Q_table[state[0], state[1], action]
        # print(self.E_table[state[0], state[1], action])
        self.E_table[state[0], state[1], action] += 1

        self.Q_table += 0.01 * error * self.E_table

        self.E_table *= self.gamma * self.lamb

    def discretized_env_state(self, state):
        """
            Agent.discretized_env_state takes a given state and discretizes the state to use whole numbers instead of
            integers for easier computations.
            :param state
            :return discrete_state
        """
        min_states = self.env.observation_space.low
        discrete_state = (state - min_states) * np.array([10, 100])
        return np.round(discrete_state, 0).astype(int)

    def terminal_output(self, i):
        # Terminal Output for stats of each epoch
        avg_reward = sum(self.epoch_rewards[-2:]) / len(self.epoch_rewards[-2:])
        self.epoch_rewards_table['ep'].append(i)
        self.epoch_rewards_table['avg'].append(avg_reward)
        self.epoch_rewards_table['min'].append(min(self.epoch_rewards))
        self.epoch_rewards_table['max'].append(max(self.epoch_rewards))

        print(f"Epoch - {i}\t| avg: {avg_reward:.2f}\t| min: {min(self.epoch_rewards[-1:]):.2f}"
              f"\t| max: {max(self.epoch_rewards[-1:]):.2f}")

'''
Author: Caleb Ehrisman
Course- Advanced AI CSC-549
```

```
Assignment - Programming Assignment #3 - Mountain Car
This file contains the code to implement the SARSA(lambda) algorithm.
All functions needed by solely the agent are included as member functions of class Agent
'''
import numpy as np
import itertools


class FourierBasis:
    def __init__(self, state_space, order):
        self.order = order
        self.state_dim = state_space
        self.order = [order]*self.state_dim
        self.coeff = self.coefficients()

    def coefficients(self):
        """
            FourierBasis.coefficients creates the coeffs for the FourierBasis

            :return np.array(coeff)
        """
        coeff = [np.zeros([self.state_dim])]

        for i in range(0, self.state_dim):
            for c in range(0, self.order[i]):
                v = np.zeros(self.state_dim)
                v[i] = c + 1
                coeff.append(v)
        return np.array(coeff)

    def get_features(self, state):
        """
            FourierBasis.get_features gets the feature vector. Usually noted as x in SARSA(LAMBDA)

            :param state

            :return feature_vector
        """
```

```
        return np.cos(np.pi * np.dot(self.coeff, state))
```