

## Exam 2: Advanced AI

Caleb Ehrisman

11/30/2022

### Overview

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

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

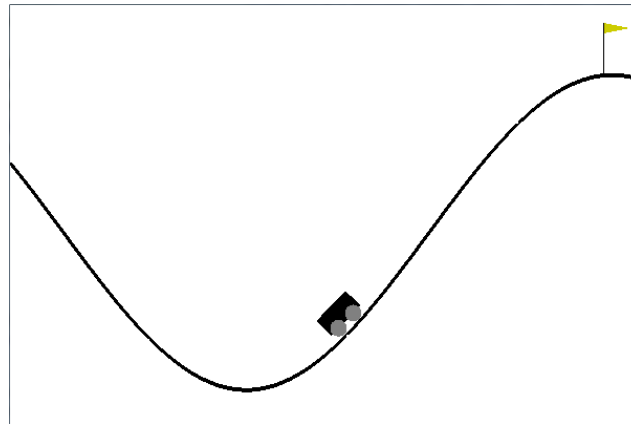


Figure 1: Screenshot of a graphic of mountain car

## 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( $\lambda$ ) was used as the learning agent.

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

Input: a feature function  $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \rightarrow \mathbb{R}^d$  such that  $\mathbf{x}(\text{terminal}, \cdot) = \mathbf{0}$   
 Input: a policy  $\pi$  (if estimating  $q_\pi$ )  
 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  $\varepsilon$ -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  $\varepsilon$ -greedy according to  $\hat{q}(S', \cdot, \mathbf{w})$   
      $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$   
      $Q \leftarrow \mathbf{w}^\top \mathbf{x}$   
      $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$   
      $\delta \leftarrow R + \gamma Q' - Q$   
      $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + (1 - \alpha \gamma \lambda \mathbf{z}^\top \mathbf{x}) \mathbf{x}$   
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha (\delta + Q - Q_{old}) \mathbf{z} - \alpha (Q - Q_{old}) \mathbf{x}$   
      $Q_{old} \leftarrow Q'$   
      $\mathbf{x} \leftarrow \mathbf{x}'$   
      $A \leftarrow A'$   
   until  $S'$  is terminal

Figure 2: SARSA( $\lambda$ ) from Sutton and Barto RL book

Where  $\mathbf{w}$  is the weight vector and  $\mathbf{x}$  is the feature approximation vector for a given state and action. The  $\mathbf{z}$  is the trace vector. The hyper parameters are as follows:  $\alpha$  - learning rate,  $\lambda$  - decay rate of trace, and  $\varepsilon$  - 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  $(M + 1)^2 \times N$  for the coefficients. 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.

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

## 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( $\lambda$ ) 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.

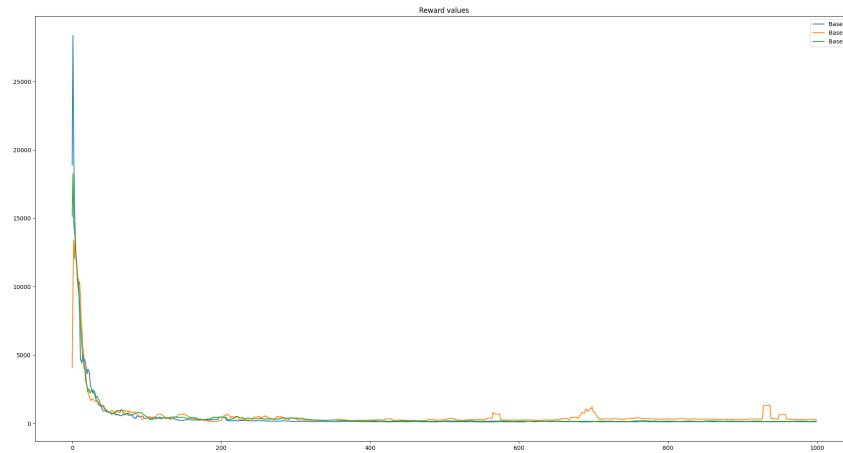


Figure 3: Learning Curves of bases 3,5,7

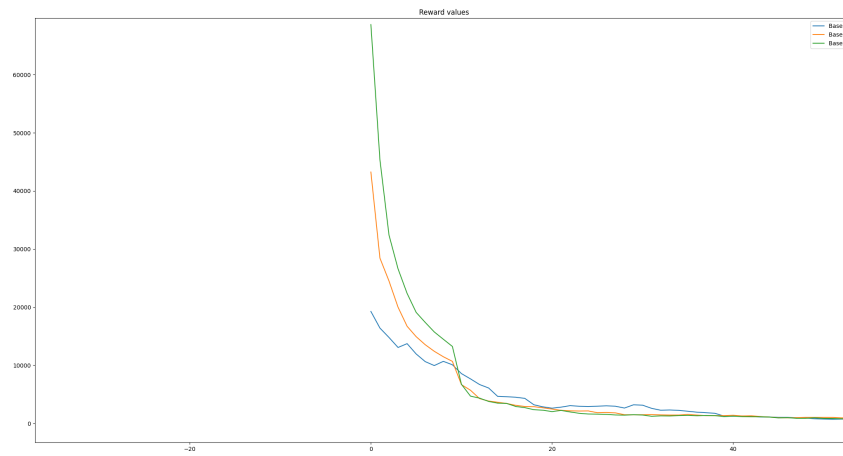


Figure 4: The first 50 episodes

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  $\alpha$  - 0.01,  $\lambda$  - 0.9, and  $\epsilon$  - 0.05.

The surface plots of the  $Q$ -value estimation from the Fourier Basis Approximation

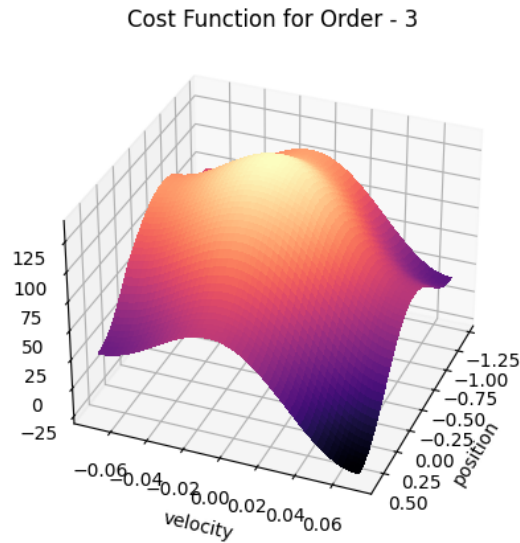


Figure 5: Surface plot of base 3

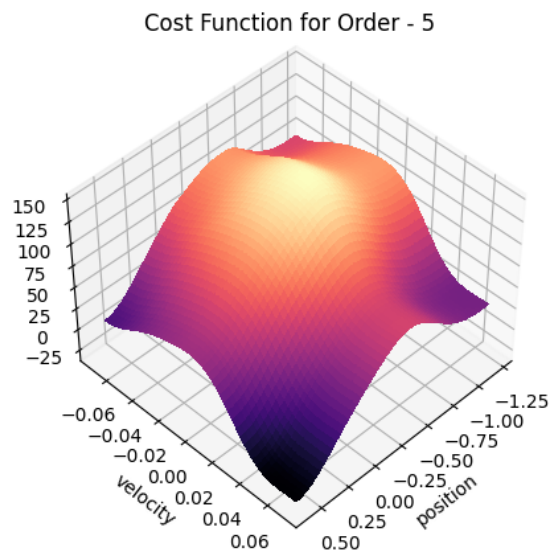


Figure 6: Surface plot of base 5

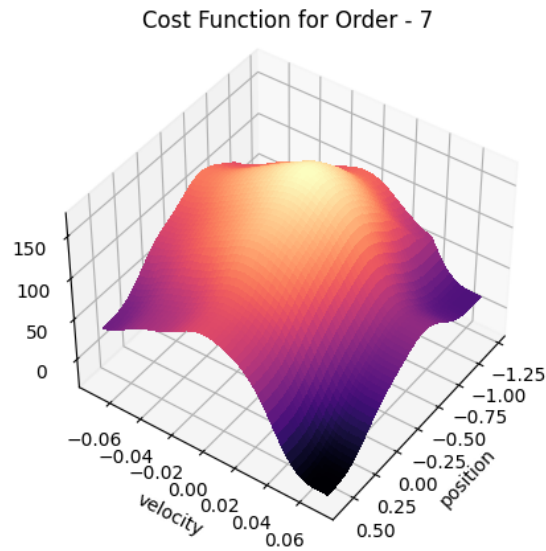


Figure 7: Surface plot of base 7

## Question

**The Mountain Car contains a negative step reward and a zero goal reward. What would happen if  $\gamma$  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.

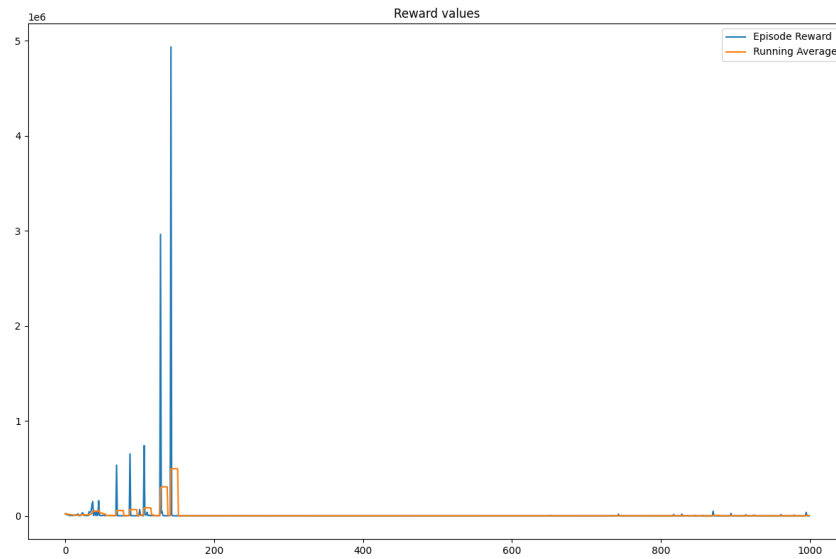


Figure 8: Learning Curves basis 3 and gamma 0.95

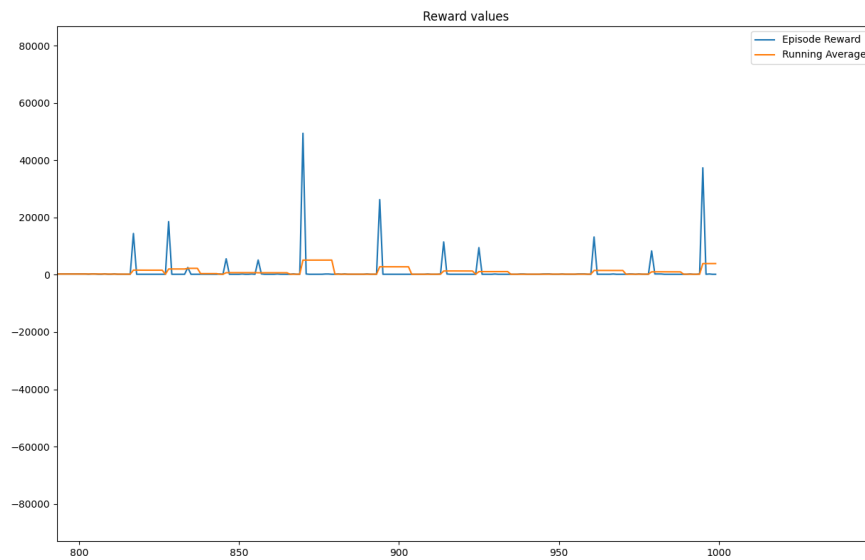


Figure 9: Last 200 episodes.

**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.

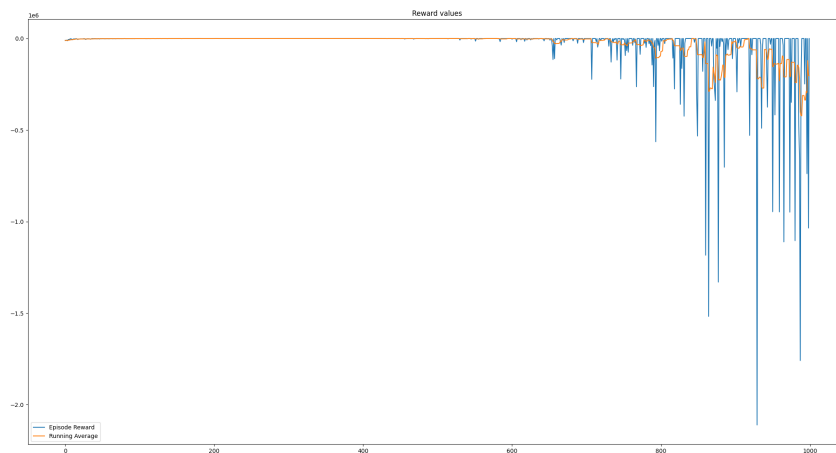


Figure 10: Learning Curves basis 3 and 0 step reward and 100 terminal state reward



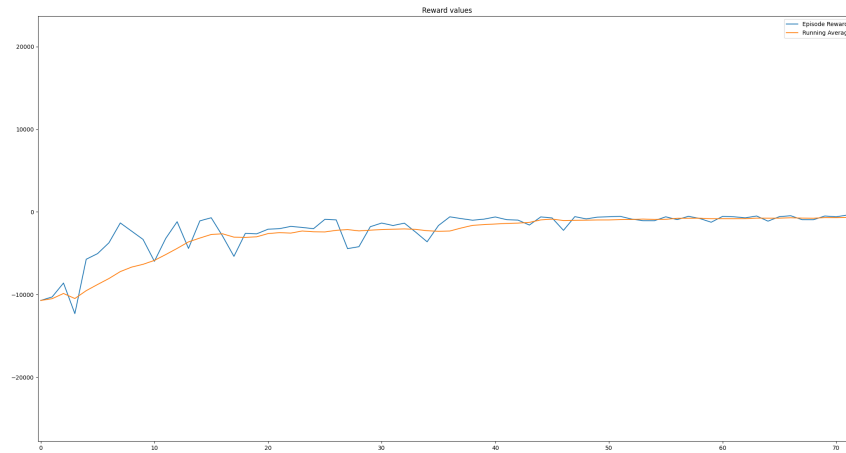


Figure 11: Early learning curve

## 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

```

1 '''
2 Author: Caleb Ehrisman
3 Course- Advanced AI CSC-549
4 Assignment - Programming Assignment #3 - Mountain Car
5
6 Tasks
7 - Implement Sarsa(lambda) to solve mountain car problem
8 - Use Linear Function Approximation with Fourier Basis
  functions
9 - Show different learning curves for 3rd, 5th, and 7th
  order Fourier bases
10 - Create surface plot of the value function

```

```
11 | - Answer short response question
12 | '''
13 |
14 | import gym
15 | import matplotlib
16 |
17 | from agent import Agent
18 | import argparse
19 | import pandas as pd
20 | import matplotlib.pyplot as plt
21 | import numpy as np
22 |
23 | """
24 |     parse gathers command line arguments.
25 |
26 |
27 |     :return: a list of all parsed arguments
28 | """
29 |
30 |
31 | def parse():
32 |     parser = argparse.ArgumentParser()
33 |     parser.add_argument('--render', type=str, help='Specify
34 |         to run simulation or not')
35 |     parser.add_argument('--order', type=int, help='Choose
36 |         order for fourier basis', default=3)
37 |     parser.add_argument('--num_epochs', type=int, help='
38 |         Choose number of epochs', default=1000)
39 |     parser.add_argument('--fourier', type=str, help='Choose
40 |         to use fourier', default=True)
41 |     parser.add_argument('--file', type=str,
42 |         help='File path to save weights to.
43 |         Must be given with .npz extension
44 |         ', default='weights.npz')
45 |     parser.add_argument('--train', type=str, help='Choose if
46 |         training or running', default='True')
47 |     parser.add_argument('--eval', type=str, default='False')
48 |     return parser.parse_args()
49 |
50 |
51 | if __name__ == "__main__":
52 |     args = parse()
```

```
46
47     if args.render == "True":
48         env = gym.make("MountainCar-v0", render_mode="human"
49             )
50     else:
51         env = gym.make("MountainCar-v0")
52
53     file = args.file
54
55     n = args.num_epochs
56     if args.fourier == 'False':
57         agent = Agent(env, file, fourier=False, order=3)
58     else:
59         agent = Agent(env, file, order=3)
60
61     if args.eval == 'False':
62         if args.train == 'True':
63             rewards, avg, learner = agent.learn(n)
64         else:
65             rewards, avg = agent.run(n)
66
67         fig, ax = plt.subplots(figsize=(10, 4))
68         plt.plot(np.negative(rewards), label='Episode Reward
69             ')
70         plt.plot(np.negative(avg), label='Running Average')
71         ax.set_title("Reward values")
72         plt.legend()
73         plt.show()
74
75         rewards = []
76         base = [3, 5, ]
77
78         rewards.append(avg)
79
80         fig, ax = plt.subplots(figsize=(10, 4))
81         plt.plot(np.negative(rewards[0]), label='Base 3')
82         plt.plot(np.negative(rewards[1]), label='Base 5')
83         plt.plot(np.negative(rewards[2]), label='Base 7')
84         ax.set_title("Reward values")
85         plt.legend()
86         plt.show()
```

```

86
87     low = env.observation_space.low
88     high = env.observation_space.high
89     difference = high - low
90
91     x_axis = np.linspace(low[0], high[0])
92     y_axis = np.linspace(low[1], high[1])
93     x_axis, y_axis = np.meshgrid(x_axis, y_axis)
94     z_axis = np.zeros(x_axis.shape)
95
96     for i in range(0, z_axis.shape[0]):
97         for j in range(0, z_axis.shape[1]):
98             s = [(x_axis[i, j] - low[0]) / (high[0] -
99                 low[0]), (y_axis[i, j] - low[1]) / (high
100                 [1] - low[1])]
101             (zq, _) = learner.best_action(s)
102             z_axis[i, j] = -1.0 * zq
103
104             fig = plt.figure()
105             ax = plt.axes(projection='3d')
106             ax.plot_surface(x_axis, y_axis, z_axis, cmap=
107                 matplotlib.cm.get_cmap("magma"))
108             ax.set_xlabel('position')
109             ax.set_ylabel('velocity')
110             ax.set_title('Cost Function for Order - ' + str(
111                 n))
112             plt.show()
113
114     else:
115         rewards = []
116         base = [3, 5, 7]
117         learner = []
118         for i in range(3):
119             agent = Agent(env, file, order=base[i])
120             reward, avg, temp_learner = agent.learn(1000)
121             rewards.append(avg)
122             learner.append(temp_learner)
123
124     fig, ax = plt.subplots(figsize=(10, 4))
125     plt.plot(np.negative(rewards[0]), label='Base 3')
126     plt.plot(np.negative(rewards[1]), label='Base 5')
127     plt.plot(np.negative(rewards[2]), label='Base 7')

```

```
124     ax.set_title("Reward values")
125     plt.legend()
126     plt.show()
127
128     low = env.observation_space.low
129     high = env.observation_space.high
130     difference = high - low
131
132     x_axis = np.linspace(low[0], high[0])
133     y_axis = np.linspace(low[1], high[1])
134     x_axis, y_axis = np.meshgrid(x_axis, y_axis)
135     z_axis = np.zeros(x_axis.shape)
136
137     for b in range(3):
138         for i in range(0, z_axis.shape[0]):
139             for j in range(0, z_axis.shape[1]):
140                 s = [(x_axis[i, j] - low[0]) / (high[0]
141                     - low[0]), (y_axis[i, j] - low[1]) /
142                     (high[1] - low[1])]
143                 (zq, _) = learner[b].best_action(s)
144                 z_axis[i, j] = -1.0 * zq
145
146     fig = plt.figure()
147     ax = plt.axes(projection='3d')
148     ax.plot_surface(x_axis, y_axis, z_axis, cmap=
149         matplotlib.cm.get_cmap("magma"))
150     ax.set_xlabel('position')
151     ax.set_ylabel('velocity')
152     ax.set_title('Cost Function for Order - ' + str(
153         base[b]))
154     plt.show()
155     '''
156 Author: Caleb Ehrisman
157 Course- Advanced AI CSC-549
158 Assignment - Programming Assignment #3 - Mountain Car
159
160 This file contains the code to implement the SARSA(lambda)
161 algorithm.
162
163 All functions needed by solely the agent are included as
164 member functions of class Agent
165 '''
```

```
160 import numpy as np
161 from fourier_basis import FourierBasis
162 from sarsalambdaFA import SarsaLambdaFA
163 from sarsa import Sarsa
164 import os.path
165
166 ALPHA = 0.0001
167 GAMMA = 1
168 EPSILON = 0.5
169 LAMBDA = 0.9
170
171
172 class Agent:
173
174     def __init__(self, environment, file, fourier=True,
175                  order=3, runs=1, gamma=0.001):
176         """
177             init is the constructor for the Agent class.
178
179             :param environment
180             :return None
181
182         """
183         self.runs = runs
184         self.order = order
185         self.env = environment
186         self.gamma = gamma
187         self.num_actions = self.env.action_space.n
188         self.state_dims = self.env.observation_space.shape
189         [0]
190         self.fourier = fourier
191         self.epoch_rewards = []
192         self.epoch_rewards_table = {'ep': [], 'avg': [], '
193                                     min': [], 'max': []}
194         self.epoch_max_pos = []
195         self.file = file
196
197     def learn(self, num_epochs):
198         """
199             Agent.learn does the actual stepping through and
200             exploring the environment and then updates
201             the Q_table if
```

```
196         using traditional SARSA and updates the weight
197         and lambda vectors is using a fourier basis
198
199         :param num_epochs
200         :return None
201         """
202     for run in range(0, self.runs):
203         fb = FourierBasis(state_space=self.env.
204             observation_space.shape[0], order=self.order)
205         if self.fourier:
206             learner = SarsaLambdaFA(fa=fb, num_actions=
207                 self.num_actions, alpha=0.0001, epsilon
208                 =0.8)
209         else:
210             learner = Sarsa(environment=self.env)
211
212     for i in range(num_epochs):
213
214         learner.epsilon *= .99
215         if self.fourier:
216             learner.z = np.zeros(learner.w.shape)
217             state, _ = self.env.reset()
218             if self.fourier:
219                 state = (state - self.env.
220                     observation_space.low) / (self.env.
221                     observation_space.high - self.env.
222                     observation_space.low)
223             else:
224                 state = learner.discretized_env_state(
225                     state)
226                 learner.E_table = learner.create_q_table
227                     ()
228
229         # steps = 0
230         action = learner.action(state)
231         done = False
232         reward_sum = 0
233
234         while not done:
235             next_state, reward, done, info, _ = self
236                 .env.step(action)
237             # reward += 1
238             # if done:
```

```
228         # reward = 100
229         if self.fourier:
230             next_state = (next_state - self.env.
231                           observation_space.low) / (
232                               self.env.
233                                   observation_space.
234                                       high - self.env.
235                                           observation_space.low
236                                               )
237         else:
238             next_state = learner.
239                 discretized_env_state(next_state)
240
241             next_action = learner.action(next_state)
242
243             learner.update(state, action, reward,
244                             next_state, done, next_action)
245
246             # steps += 1
247             state = next_state
248             action = next_action
249             reward_sum += reward
250
251             # Append max position data and reward data
252             # for evaluation
253             self.epoch_rewards.append(reward_sum)
254
255             self.terminal_output(i)
256             np.save(self.file, learner.w)
257             return self.epoch_rewards, self.epoch_rewards_table[
258                 'avg'], learner
259
260 def run(self, num_epochs):
261     """
262     Agent.run uses a pre-trained set of weights to
263     greedily choose actions.
264
265     :param num_epochs
266     :return None
267     """
268
269     if os.path.exists(self.file):
```



```
260         w = np.load(self.file)
261     else:
262         print("Error loading file. Not found.")
263         return
264
265     fb = FourierBasis(state_space=self.env.
266                       observation_space.shape[0], order=self.order)
267     learner = SarsaLambdaFA(fa=fb, num_actions=self.
268                             num_actions, alpha=0.0, epsilon=0.0)
269     learner.w = w
270
271     for i in range(num_epochs):
272         state, _ = self.env.reset()
273
274         state = (state - self.env.observation_space.low)
275                 / (
276                     self.env.observation_space.high -
277                     self.env.observation_space.low)
278
279         action = learner.action(state)
280         done = False
281         reward_sum = 0
282
283         while not done:
284             next_state, reward, done, info, _ = self.env
285                 .step(action)
286             print(reward)
287             next_state = (next_state - self.env.
288                           observation_space.low) / (
289                 self.env.observation_space.high -
290                 self.env.observation_space.low)
291
292             next_action = learner.action(next_state)
293             action = next_action
294             reward_sum += reward
295
296             # Append max position data and reward data for
297             # evaluation
298         self.epoch_rewards.append(reward_sum)
299
300     self.terminal_output(i)
```

```
294         return self.epoch_rewards, self.epoch_rewards_table[
295             'avg']
296     def terminal_output(self, i):
297         # Terminal Output for stats of each epoch
298         avg_reward = sum(self.epoch_rewards[-10:]) / len(
299             self.epoch_rewards[-10:])
300         self.epoch_rewards_table['ep'].append(i)
301         self.epoch_rewards_table['avg'].append(avg_reward)
302         self.epoch_rewards_table['min'].append(min(self.
303             epoch_rewards[:]))
304         self.epoch_rewards_table['max'].append(max(self.
305             epoch_rewards[:]))
306
307         print(f"Epoch - {i}\t| avg: {avg_reward:.2f}\t| min:
308             {min(self.epoch_rewards[-1:]):.2f}"
309             f"\t| max: {max(self.epoch_rewards[-1:]):.2f}"
310             )
311
312     '''
313     Author: Caleb Ehrisman
314     Course- Advanced AI CSC-549
315     Assignment - Programming Assignment #3 - Mountain Car
316
317     This file contains the code to implement the SARSA(lambda)
318     with a function approximator.
319
320     '''
321     class SarsaLambdaFA:
322     def __init__(self, fa, num_actions=None, alpha=0.01,
323         gamma=1.0, lamb=0.9, epsilon=0.5):
324         self.gamma = gamma
325         self.lamb = lamb
326         self.epsilon = epsilon
327         self.alpha = alpha
328         self.num_actions = num_actions
329         self.fourier_basis = []
330
331         for i in range(0, self.num_actions):
332             self.fourier_basis.append(copy.deepcopy(fa))
```

```
327         self.w = np.zeros([self.fourier_basis[0].coeff.shape
328                               [0], num_actions])
329         self.z = np.zeros(self.w.shape)
330         self.w[0, :] = 0.0
331
332     def action(self, state):
333         """
334             Agent.action determines what action to take
335             based on state
336
337             :param state:
338             :return action
339         """
340         if np.random.uniform(0, 1) < self.epsilon:
341             return random.randrange(0, self.num_actions)
342
343         best = -math.inf
344         best_actions = []
345         for a in range(0, self.num_actions):
346             q = self.Q(state, a)
347             if math.isclose(q, best):
348                 best_actions.append(a)
349             elif q > best:
350                 best = q
351                 best_actions = [a]
352
353         return random.choice(best_actions)
354
355     def Q(self, state, action):
356         return np.dot(self.w[:, action], self.fourier_basis[
357             action].get_features(state))
358
359     def best_action(self, state):
360         best = -math.inf
361         best_action = 0
362         for a in range(0, self.num_actions):
363             q = self.Q(state, a)
364             if q > best:
365                 best = q
366                 best_action = a
```

```
366         return best, best_action
367
368     def update(self, state, action, reward, next_state, done
369               , next_action=None):
370         """
371             Agent.update updates the Q table based on the
372             SARSA algorithm. It also updates the trace
373             table
374
375             :param next_action:
376             :param done:
377             :param next_state:
378             :param state:
379             :param action:
380             :param reward
381             :return None
382         """
383
384         delta = reward - self.Q(state, action)
385
386         if not done:
387             if next_action is not None:
388                 delta += self.gamma * self.Q(next_state,
389                                               next_action)
390             else:
391                 q_dot, next_action = self.best_action(
392                     next_state)
393                 delta += self.gamma * self.best_action(q_dot
394                                                         )
395
396         phi = self.fourier_basis[action].get_features(state)
397
398         for a in range(0, self.num_actions):
399             self.z[:, a] += self.gamma * self.lamb
400             if a == action:
401                 self.z[:, a] += phi
402             self.w[:, a] += self.alpha * delta * self.z[:, a]
403
404         return delta
405
406     """
```

```
401 Author: Caleb Ehrisman
402 Course- Advanced AI CSC-549
403 Assignment - Programming Assignment #3 - Mountain Car
404 This file contains the code to implement the SARSA(lambda)
    algorithm.
405 All functions needed by solely the agent are included as
    member functions of class Agent
406 '''
407 import numpy as np
408
409 ALPHA = 0.1
410 GAMMA = 1
411 EPSILON = 0.5
412 LAMBDA = 0.9
413
414
415 class Sarsa:
416     def __init__(self, environment, gamma=1.0, epsilon=0.5,
        alpha=0.0001, lamb=0.9):
417         """
418             init is the constructor for the Agent class.
419             :param environment
420             :return None
421         """
422         self.env = environment
423         self.E_table = self.create_q_table()
424         self.Q_table = self.create_q_table()
425         self.alpha = alpha
426         self.gamma = gamma
427         self.epsilon = epsilon
428         self.lamb = lamb
429         self.epoch_rewards = []
430         self.epoch_rewards_table = {'ep': [], 'avg': [], '
            min': [], 'max': []}
431         self.epoch_max_pos = []
432
433     def create_q_table(self):
434         """
435             Agent.create_q_table creates the Q table
436             that fits all states
437             :return np.array of [x_lim][y_lim][
                num_actions]
```

```
437         """
438         high = self.env.observation_space.high
439         low = self.env.observation_space.low
440         num_states = (high - low) * np.array([10, 100])
441         num_states = np.round(num_states, 0).astype(int) + 1
442         num_actions = self.env.action_space.n
443         return np.zeros([num_states[0], num_states[1],
444                          num_actions])
445
446     def action(self, state):
447         """
448         Agent.action determines what action to take
449         based on state
450         :param state
451         :return action
452         """
453         if np.random.uniform(0, 1) < EPSILON:
454             action = self.env.action_space.sample()
455         else:
456             # disc_state = self.discretized_env_state(state)
457             action = np.argmax(self.Q_table[state[0], state
458                                           [1]])
459         return action
460
461     def update(self, state, action, reward, next_state, done
462               , next_action):
463         """
464         Agent.update updates the Q table based on the
465         SARSA algorithm. It also updates the trace
466         table
467         :param done:
468         :param state
469         :param action
470         :param reward
471         :param next_state
472         :param next_action
473         :return None
474         """
475         target = reward + self.gamma * self.Q_table[
476             next_state[0], next_state[1], next_action]
```

```
471         error = target - self.Q_table[state[0], state[1],
472             action]
473         # print(self.E_table[state[0], state[1], action])
474         self.E_table[state[0], state[1], action] += 1
475
476         self.Q_table += 0.01 * error * self.E_table
477
478         self.E_table *= self.gamma * self.lamb
479
480     def discretized_env_state(self, state):
481         """
482         Agent.discretized_env_state takes a given state
483         and discretizes the state to use whole
484         numbers instead of
485         integers for easier computations.
486         :param state
487         :return discrete_state
488         """
489         min_states = self.env.observation_space.low
490         discrete_state = (state - min_states) * np.array
491             ([10, 100])
492         return np.round(discrete_state, 0).astype(int)
493
494     def terminal_output(self, i):
495         # Terminal Output for stats of each epoch
496         avg_reward = sum(self.epoch_rewards[-2:]) / len(self
497             .epoch_rewards[-2:])
498         self.epoch_rewards_table['ep'].append(i)
499         self.epoch_rewards_table['avg'].append(avg_reward)
500         self.epoch_rewards_table['min'].append(min(self.
501             epoch_rewards))
502         self.epoch_rewards_table['max'].append(max(self.
503             epoch_rewards))
504
505         print(f"Epoch - {i}\t| avg: {avg_reward:.2f}\t| min:
506             {min(self.epoch_rewards[-1:]):.2f}"
507             f"\t| max: {max(self.epoch_rewards[-1:]):.2f}"
508             )
509
510     '''
511     Author: Caleb Ehrisman
512     Course- Advanced AI CSC-549
```

```
504 Assignment - Programming Assignment #3 - Mountain Car
505 This file contains the code to implement the SARSA(lambda)
    algorithm.
506 All functions needed by solely the agent are included as
    member functions of class Agent
507 '''
508 import numpy as np
509 import itertools
510
511
512 class FourierBasis:
513     def __init__(self, state_space, order):
514         self.order = order
515         self.state_dim = state_space
516         self.order = [order]*self.state_dim
517         self.coeff = self.coefficients()
518
519     def coefficients(self):
520         """
521         FourierBasis.coefficients creates the coeffs for
            the FourierBasis
522
523         :return np.array(coeff)
524         """
525         coeff = [np.zeros([self.state_dim])]
526
527         for i in range(0, self.state_dim):
528             for c in range(0, self.order[i]):
529                 v = np.zeros(self.state_dim)
530                 v[i] = c + 1
531                 coeff.append(v)
532         return np.array(coeff)
533
534     def get_features(self, state):
535         """
536         FourierBasis.get_features gets the feature
            vector. Usually noted as x in SARSA(LAMBDA)
537
538         :param state
539
540         :return feature_vector
541         """
```



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