Project 3: Mountain Car

Sherwyn Braganza

December 3, 2022

Summary of Contents

- The Aim of the Project
- The Approach and Algorithm
- The Results
- Appendices

1 Aim of the Project

The aim of this project was to implement the famed 'Mountain Car' problem (Sutton and Barto, 1998) using SARSA(λ) with linear function approximation using Fourier Basis Functions.

The Mountain Car Problem involves a car in a Sinusoidal Valley who's aim is to reach the peak (Figure 1). The action space of the car consists of three actions - Left, Stationary and Right. The state space is two dimensional - $[x, \dot{x}]$ where x is the position and $\dot{x}$ is the velocity of the body. Every action taken that doesn't lead to a terminal state (goal is reached), the reward is -1.

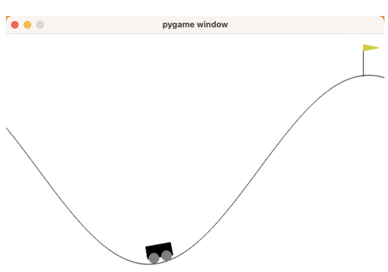


Figure 1: Mountain Car Environment

2 The Approach and Algorithm

2.1 Learner

In my implementation, I used SARSA(λ) as my learning agent. It exactly follows the algorithm shown in Figure 2.

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: Online SARSA (λ)

w is your weight vector and **x** is the function approximation (in this case, a Fourier function approximation) for a given state and action.
α is the learning rate, λ is the trace decay rate and ϵ is a control factor that tells the algorithm whether it should follow the policy or take a exploratory action.

The only change that was made to this algorithm was that the episode would prematurely end if it took longer than 400 steps to reach the goal. This was done as a precautionary measure to mitigate penalty for starting in a bad state. The code that implements this algorithm can be found in Appendix I.

2.2 Function Approximator

I used the method of Fourier Function Approximator described in Value Function Approximation in Reinforcement Learning using the Fourier Basis by Konidaris et. al to predict the Q values for a given state-action pair. The function took in two instantiation variables - the number of states and order that corresponded to the order of the Fourier basis function approximation. Using this it created a $(M + 1)^2$ x N matrix (**C**), where N is the number of states and M is the order of the Fourier approximation. The **getFourierBasisApprox** function in my code used Equation 1 to get ϕ or w or the Fourier basis which was then dotted with the weight vector in the SARSA algorithm to get the $\boldsymbol{Q}$ for a given state-action.

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

The code for this function can be found in Appendix II

3 Results

The program has two running modes:

1. Visual Mode - python3 agent.py
2. Analysis Mode - python3 agent.py analysis

The **Visual Mode** brings up a visual display of the algorithm learning. The **Analysis Mode** on the other hand doesn't have any GUI output. It however runs the learner using Fourier Approximations of base 3, 5 and 7. It performs 50 runs consisting of 250 episodes each and records the results. The data received for each basis function (3, 5 and 7) is averaged over the runs an then plotted against the other basis functions (Figure 3 shows this - the steps vs episodes plot).

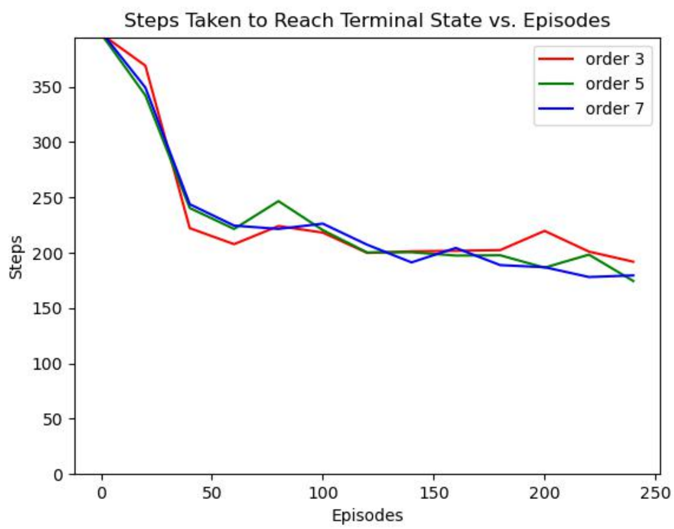
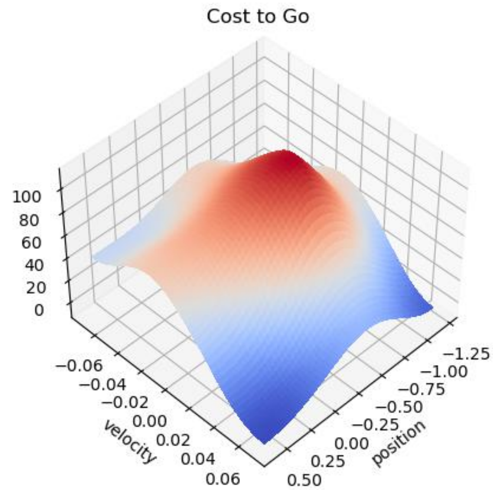
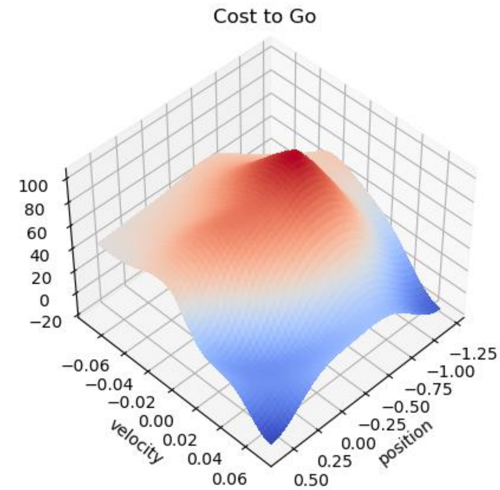


Figure 3: Steps vs Episodes in reaching terminal state

In addition to saving a copy of the previous plot, the program also prints the 'Cost to Go' plots for each Fourier Basis Approximation from the best trained models created for each basis. The plots Figure 4, Figure 5 and Figure 6 show the results obtained from one such run.



(a) Cost to Go - Order 3



(b) Cost to Go - Order 5

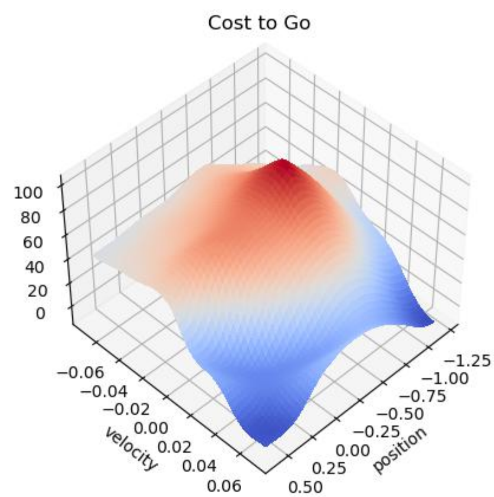


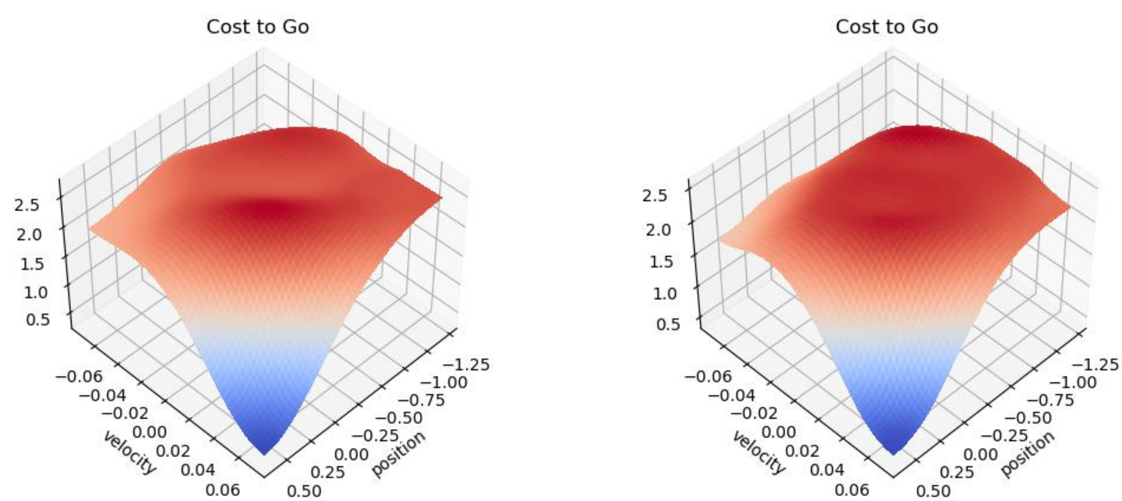
Figure 5: Cost to Go - Order 7

3.1 Questions

1) Show learning curves for order 3, 5, and 7 Fourier bases, for fixed setting of α and ϵ, and γ = 1, λ = 0.9.
Figure 3 shows the plot obtained for the learning curves.

2) Create a surface plot of the value function (the negative of the value function) of the learned policies after 1, 000 episodes, for the above orders. (Hint: Your plot should look like the one in Sutton and Barto, but smoother.)
The plots Figure 4, Figure 5 and Figure 6 show this.

3) 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? What would happen if we had a zero step cost and a positive goal reward, for the case where γ = 1, and the case where $\gamma < 1$?
γ is the discount factor, which indicates how much the reward of a future state affects the current state. For a situation with a large number of states and a gamma less than 1, the effects of a profitable state in the future would die out really fast in previous states. This would make training slower when starting in bad initial states and would change the 'Cost to Go' plots drastically. The figures below show the Mountain Car Experiment with a $\gamma = 0.6$.



(a) Cost to Go - Order 3

(b) Cost to Go - Order 5

Consider the situation of having a path with a loop in it (Figure 7 below). In one scenario, the car travels through the loop to reach the right end and in the other, it bypasses the loop.

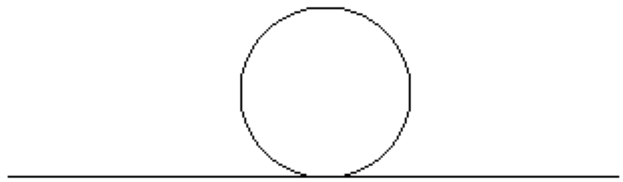


Figure 7: Track with Loop

If the reward for each step was changed to 0 and the final state 1, both episodes would be equally weighted even though one takes significantly more steps. Moreover, the episode that takes the loop would never end out of the loop. In this case a γ of 1 or less than that would not change anything

4 Appendix I

```python
import copy
import numpy as np
import math
import sys
import random
import gym
from matplotlib import pyplot as plt
from function_approximator import FourierBasis
from matplotlib import cm


class SarsaLambdaLinear:
    """
        Implements SARSA Lamda with Linear Function Approximation

        Attributes:
            actions - The size of the action space
            alpha - The learning rate
            gamma - The discount factor
            lamb - Lambda: the trace decay factor
            epsilon - Factor that decides whether to follow policy or explore
    """
    def __init__(self, function_approximator, actions: int,  alpha: float = 0.1,
                 gamma: float = 0.99, lamb: float = 0.9, epsilon: float = 0.05, initial_valfunc: float = 0.0):
        self.actions = actions
        self.alpha = alpha
        self.gamma = gamma
        self.lamb = lamb
        self.epsilon = epsilon
        self.initial_valfunc = initial_valfunc
        self.function_approximator = None

        # if we don't have an approximation for each action, create deepcopies for each
        if type(function_approximator) != list or len(function_approximator) < self.actions:
            temp = []
            for x in range(self.actions):
                temp.append(copy.deepcopy(function_approximator))

            self.function_approximator = temp

        if initial_valfunc == 0.0:
            # initialize the 'theta' array corresponding to each action
            self.weights = np.zeros([self.function_approximator[0].getShape()[0], self.actions])
        else:
            self.weights = np.ones([self.function_approximator[0].getShape()[0], self.actions]) * initial_valfunc

        # initialize the weights for the trace update.
        self.lambda_weights = np.zeros(self.weights.shape)

    def traceClear(self):
        """
        Clear the weights of the trace vector
        """
        self.lambda_weights = np.zeros(self.weights.shape)

    def makeOnPolicy(self):
        """
        Makes the policy greedy
        """
        self.epsilon = 0

    def getStateActionVal(self, state, action):
        """
        Returns the Q value of the given state-action pair.
        """
        return np.dot(self.weights[:, action], self.function_approximator[action].getFourierBasisApprox(state))

    def getMaxStateActionVal(self, state):
        """
        Checks all Q values in
        """
        best = float('-inf')
        best_a = float('-inf')
        for a in range(0, self.actions):
            qval = self.getStateActionVal(state, a)
            if qval > best:
                best = qval
                best_a = a

        return best, best_a

    def next_move(self, state):
        """
        Stochastically returns an epsilon-greedy action from the current state.
        """

        if np.random.random() <= self.epsilon:
            return np.random.randint(0, self.actions)
```

```python

        # Build a list of actions evaluating to max_a Q(s, a)
        best = float("-inf")
        best_actions = []

        for a in range(self.actions):
            thisq = self.getStateActionVal(state, a)

            if abs(thisq - best) < 0.001:
                best_actions.append(a)
            elif thisq > best:
                best = thisq
                best_actions = [a]

        if len(best_actions) == 0 or math.isinf(best):
            print("SarsaLambdaLinearFA: function approximator has diverged to infinity.", file=sys.stderr)
            return np.random.randint(0, self.actions)

        # Select randomly among best-valued actions
        return random.choice(best_actions)

    def update(self, state, action, reward, next_state, next_action=None, terminal=False) -> float:
        """
            Runs a Sarsa update, given a transition.
            If no action is provided, it assumes an E-Greedy policy and finds
            an action that maximizes the Q value.

            Parameters
            ----------
            state - the state at time t
            action - the action to be taken to reach s+1
            reward - The reward received
            next_state - the state at t+1
            next_action - the action at t+1, if not present it is calculated
            terminal - if the next state is the terminal state.

            Returns
            -------
            delta - The TD error.

        """

        # Compute TD error
        delta = reward - self.getStateActionVal(state, action)

        # Only include s' if it is not a terminal state.
        if not terminal:
            if next_action is not None:
                delta += self.gamma*self.getStateActionVal(next_state, next_action)
            else:
                # adopt an exploration action
                (next_Q, next_action) = self.getMaxStateActionVal(next_state)
                delta += self.gamma * self.getMaxStateActionVal(next_Q)

        # Compute the basis functions for state s, action a.
        eval_f_action = self.function_approximator[action].getFourierBasisApprox(state)

        for each_a in range(0, self.actions):

            # Trace Update
            self.lambda_weights[:, each_a] *= self.gamma*self.lamb
            if each_a == action:
                self.lambda_weights[:, each_a] += eval_f_action

            # Weight Update
            self.weights[:, each_a] += self.alpha * \
                                       delta * \
                                       np.multiply(self.function_approximator[each_a].getGradientFactors(),
                                                   self.lambda_weights[:, each_a])

        # Return the TD error, which may be informative.
        return delta

def fourierBasis(env, samples: int = 10, episodes: int = 10, order: int = 3):
    gamma = 1.0
    run_data = np.zeros((samples, episodes, 2))

    state_dim = env.observation_space.shape[0]
    actions = env.action_space.n
    u_state = env.observation_space.high
    l_state = env.observation_space.low
    d_state = u_state - l_state
    best_learner = None
    best_sum = float('-inf')

    for sample in range(0, samples):

        fb = FourierBasis(order=order, dimensions=state_dim)
        learner = SarsaLambdaLinear(fb, actions=actions, gamma=gamma, lamb=0.95, epsilon=0.05, alpha=0.001)

        for episode in range(0, episodes):
            steps = 0
```

```
            # converge to pure on-policy for last 10 episodes
            if episode >= 0.8 * episodes:
                learner.makeOnPolicy()

            learner.traceClear()
            s = (env.reset() - l_state) / d_state
            a = learner.next_move(s)

            done = False
            nsteps = 0
            sum_r = 0.0

            while not done:
                sp, r, done, info = env.step(a)
                steps += 1
                sp = (sp - l_state) / d_state
                term = (done and not (info.get('TimeLimit.truncated', False)))
                ap = learner.next_move(sp)

                learner.update(s, a, r, sp, ap, terminal=term)
                s = sp
                a = ap
                sum_r += r * pow(gamma, nsteps)
                steps += 1

            run_data[sample, episode, :] = np.asarray([sum_r, steps])
            best_learner = learner if sum_r >= best_sum else best_learner

            # print('Run ' + str(sample + 1) + ", ep. " + str(episode + 1) + " return: " + str(sum_r) +
            #       ", # steps: " + str(steps))

        env.close()

    return run_data, best_learner

def runAnalysis():
    run_data = []
    learner = []
    basis = [3, 5, 7]
    colors = ['red', 'green', 'blue']
    episodes = 250
    samples = 50

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

    u_state = env.observation_space.high
    l_state = env.observation_space.low
    d_state = u_state - l_state

    for i in basis:
        temp_data, temp_learner = fourierBasis(env, samples, episodes, i)
        run_data.append(temp_data)
        learner.append(temp_learner)

    run_data_mean = []
    # run_data_std_dev = []
    data_range = range(0, episodes)

    for i in range(len(basis)):
        run_data_mean.append(np.mean(run_data[i], axis=0))
        # run_data_std_dev.append(np.std(run_data[i], axis=0))
        plt.plot(data_range[0::20], run_data_mean[i][0::20, 1], c=colors[i], label='order ' + str(basis[i]))

    # plotting steps taken to reach term vs. episodes
    plt.xlabel('Episodes')
    plt.ylabel('Steps')
    plt.ylim(0, 395)
    plt.title('Steps Taken to Reach Terminal State vs. Episodes')
    plt.legend()
    plt.savefig('steps_vs_episodes.jpg')
    plt.close()

    # value function surface plots
    x_bounds, y_bounds = [l_state[0], u_state[0]], [l_state[1], u_state[1]]
    xs, ys = np.meshgrid(np.linspace(x_bounds[0], x_bounds[1], 100),
                         np.linspace(y_bounds[0], y_bounds[1], 100))
    zs = np.zeros(xs.shape)

    for base in range(len(basis)):
        for i in range(0, zs.shape[0]):
            for j in range(0, zs.shape[1]):
                s = [(xs[i, j] - l_state[0]) / d_state[0], (ys[i, j] - l_state[1]) / d_state[1]]
                (zq, _) = learner[base].getMaxStateActionVal(s)
                zs[i, j] = -1.0 * zq

        fig = plt.figure()
        ax = fig.gca(projection='3d')
        ax.plot_surface(xs, ys, zs, cmap=cm.get_cmap("coolwarm"), linewidth=0, antialiased=False)
        ax.view_init(elev=45, azim=45)
        ax.set_xlabel('position')
        ax.set_ylabel('velocity')
        ax.view_init(elev=45, azim=45)
```

```
        ax.set_title('Cost to Go')
        fig.savefig('Cost to Go - Order ' + str(basis[base]) + '.jpeg')
        plt.close()

def runVisualDisplay():
    episodes = 250
    samples = 10
    env = gym.make('MountainCar-v0', render_mode='human').env
    results, learner = fourierBasis(env, samples=samples, episodes=episodes, order=5)


if __name__ == '__main__':
    if len(sys.argv) > 1:
        if sys.argv[1] == 'analysis':
            runAnalysis()
    else:
        runVisualDisplay()
```

5 Appendix II

```python
class FourierBasis:
    def __init__(self, order: int, dimensions: int):
        # Instance variables
        self.coefficients = np.array([])
        self.gradient_factors = np.array([])
        self.dimensions = dimensions
        self.order = [order] * self.dimensions

        # create empty container for coefficient array
        prods = [range(0, o + 1) for o in self.order]
        coeffs = [v for v in itertools.product(*prods)]
        self.coefficients = np.array(coeffs)

        with np.errstate(divide='ignore', invalid='ignore'):
            self.gradient_factors = 1.0 / np.linalg.norm(self.coefficients, ord=2, axis=1)
        self.gradient_factors[0] = 1.0  # Overwrite division by zero for function with all-zero coefficients.

    def getFourierBasisApprox(self, state_vector: np.ndarray):
        """
        Computes basis function values at a given state.
        """

        # Bounds check state vector
        if  np.min(state_vector) < 0.0 or np.max(state_vector) > 1.0:
            print('Fourier Basis: Given State Vector ({}) not in range [0.0, 1.0]'.format(state_vector),
                  file=sys.stderr)

        # Compute the Fourier Basis feature values
        return np.cos(np.pi * np.dot(self.coefficients, state_vector))

    def getShape(self):
        return self.coefficients.shape

    def getGradientFactors(self):
        return self.gradient_factors

    def getGradientFactor(self, function_no):
        return self.gradient_factors[function_no]

    def length(self):
        """Return the number of basis functions."""
        return self.coefficients.shape[0]
```