Mountain Car Using Fourier SARSA(λ)

Ryan Finn | November 30th, 2022

Introduction

Fourier SARSA(λ) was implemented with accumulating traces, α = 0.005, ε = 0, ɣ = 1, and λ = 0.9, with a max step count per episode of 200. Three orders of FSL were trained; O(3), O(5), O(7) - each with 1,000 episodes.

As recommended by Konidaris and Osentoski, each basis function ϕ_i was assigned a corresponding $\alpha_i = \alpha$ / || $\mathbf{c}^i$ ||, where $\mathbf{c}^i = \langle c_1, c_2, \ldots, c_d \rangle$, $c_j \in \{0, 1, \ldots, n\}$. In the case where $i = 0$, $\alpha_0 = \alpha$ because $\mathbf{c}^0 = \mathbf{0}$ and so || $\mathbf{c}^0$ || = 0. d is the number of degrees of freedom of the model which, in the case of the Mountain Car problem described in Sutton & Barto, is two (position and velocity). n is the order for the basis functions. In the case of FSL, this results in $(n + 1)^d$ total basis functions when using the reduced $\phi_i(\mathbf{s}) = \cos(\pi \mathbf{c}^i \cdot \mathbf{s})$, where $\mathbf{s} = \langle s_1, s_2, \ldots, s_d \rangle$, $s_j \in [0, 1]$.

The code, written in Python 3.9, requires a few dependencies, all of which are included in `requirements.txt`:

- Matplotlib v3.6.2 - Used to plot all the figures below.
- Numpy v1.23.5 - Used for vector and matrix math
- Tqdm v4.64.1 - Used to display training progress in the console output
- Joblib v1.2.0 - Used to parallelize separate batches of training
- Gym v0.26.2 - OpenAI's environment for the Mountain Car model and animation
- Pygame v2.1.2 - Required by Gym

The code deliverable contains:

- `MountainCar.py` - A class inherited from Gym's MountainCarEnv class
- `SarsaLambda.py` - The SARSA(λ) class to train
- `run_me.py` - The script to execute that trains SARSA(λ) and creates the figures
- `images` - A folder containing all the generate figures
- `requirements.txt` - The requirements file for the project. Use `pip install`.

1. Learning Curves

The learning curves (averaged over 100 runs) are shown below.

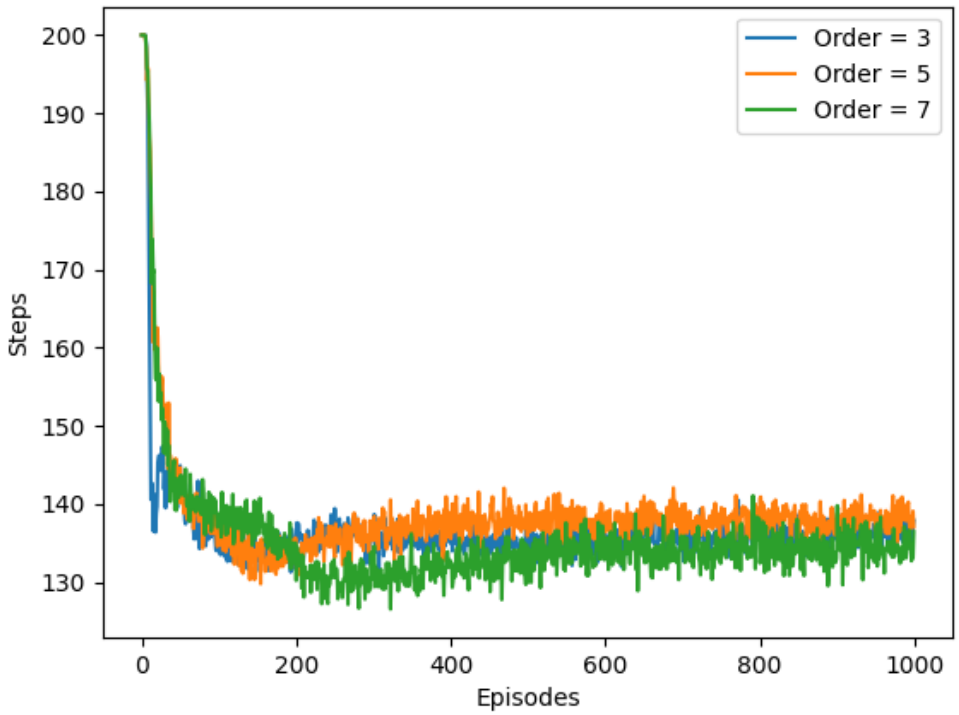


Figure 1: Steps per episode vs Episodes.

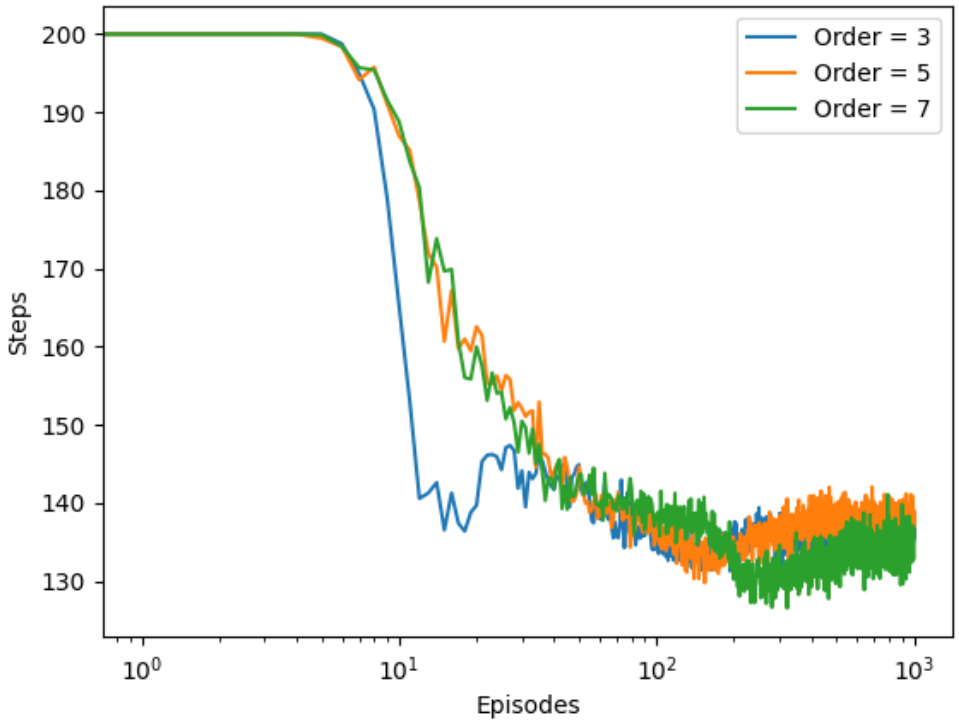
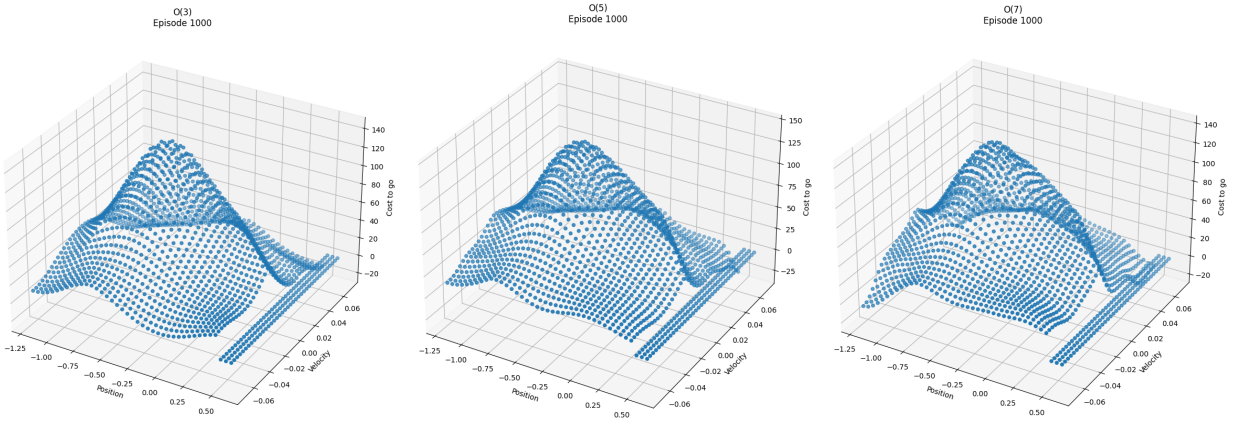


Figure 2: Steps per episode vs Episodes (logarithmic x-axis).

By the 1,000th episode O(3) and O(7) appear to perform nearly identical to one another, while O(5) performs slightly worse than the other two. O(3) seems to converge faster than either O(5) or O(7) as well, but O(7) has the lowest minimum step count at around 300 episodes.

2. Cost Functions

The cost functions (averaged over 100 runs) for the three orders are shown below.



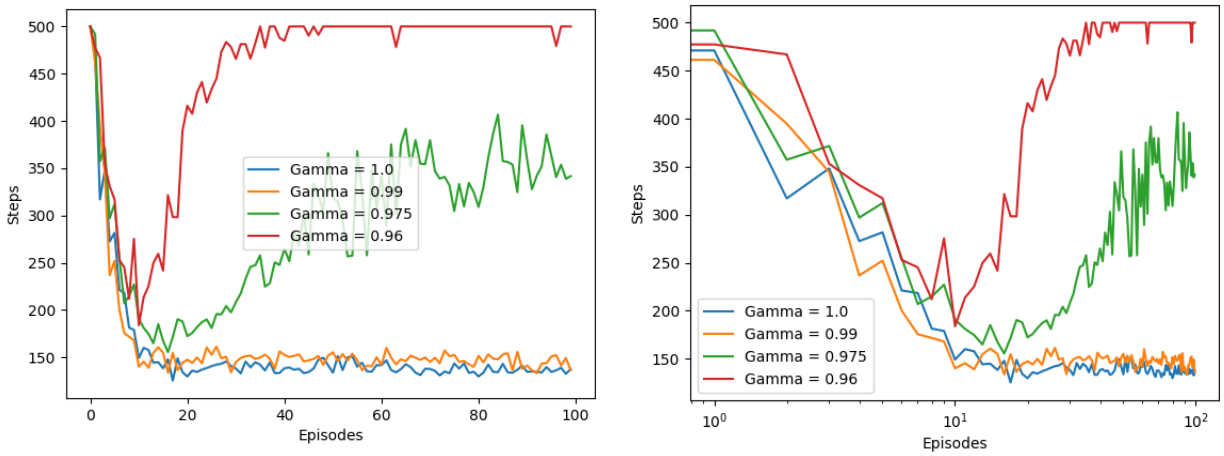
(a) (b) (c)

Figure 3: Surface plots of the cost function over a range of positions and velocities, using (a) order three, (b) order five, and (c) order seven.

I am unsure why some parts of the cost function seem to dip below 0, indicating that when in those states some action(s) actually gives a positive value. It's not clear to me whether or not this is a bug in the code, or a quirk of the Fourier basis functions.

Disregarding these negative areas, the cost functions all look quite similar to the ones presented in Sutton & Barto. There are also very little differences between each of the orders, mainly only showing variations along the edges of the plots.

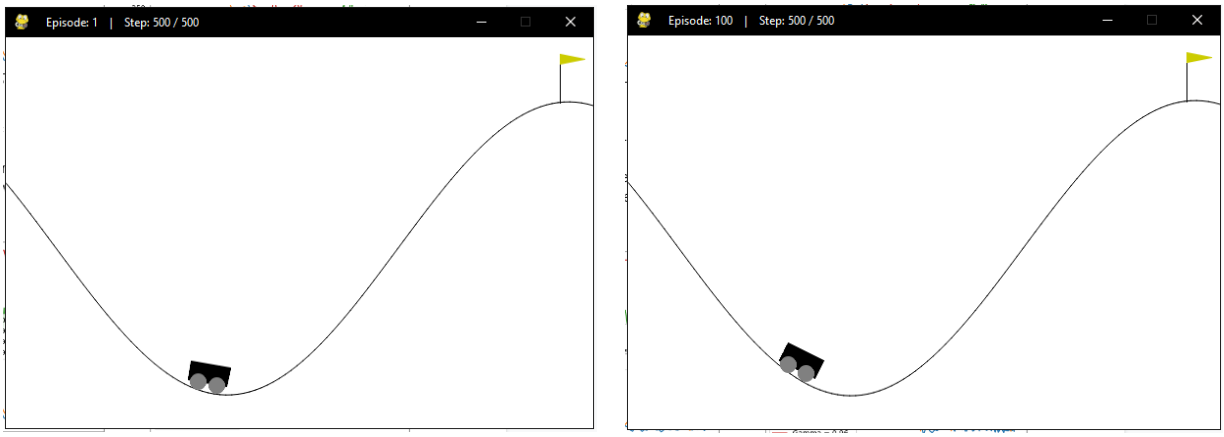
3. Varying ɣ and Costs



(a) (b)

Figure 4: O(3), Step Cost = -1, Reward = 0, 500 Max. Steps per episode, 100 episodes. Learning curves for varying ɣ values (a) (and with a logarithmic x-axis (b)).

The lower the value of ɣ the more likely the step count per episode would grow larger and max out as the number of episodes went on. I assume given enough time, even a ɣ of 0.99 will eventually diverge towards.



(a) (b)

Figure 5: O(3), Step Cost = 0, Reward = 1, 500 Max. Steps per episode, 100 episodes. Car at the end of Episode 1 (a) and the car still stuck in the valley at the end of Episode 100 (b) with no signs of improving.

Zeroing the step cost caused many problems, namely that whatever the first action the car happened to take would become highly weighted and would simply take that action again. The only way to combat this was to increase ε, but even this did not guarantee the car would make it out of the valley. Varying the other parameters like α, ɣ, and λ didn't seem to help, and ultimately the car simply rocked back and forth at the bottom of the valley until the end of each episode.

4. Fourier SARSA(λ)

$$\mathbf{w} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}, \quad \begin{array}{l} \#columns = \#actions \\ \#rows = N = (n+1)^d \end{array}$$

$$\boldsymbol{\phi}(s) = \begin{bmatrix} \phi_1(s) \\ \phi_2(s) \\ \vdots \\ \phi_N(s) \end{bmatrix}$$

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_1 & \alpha_1 & \cdots & \alpha_1 \\ \alpha_2 & \alpha_2 & \cdots & \alpha_2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_N & \alpha_N & \cdots & \alpha_N \end{bmatrix}, \quad \#columns = \#actions$$

```
For each Episode:
      Init s
      a ~ ε-greedy(s, w)
```

$$\mathbf{z} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}, \quad \begin{array}{l} \#columns = \#actions \\ \#rows = N = (n+1)^d \end{array}$$

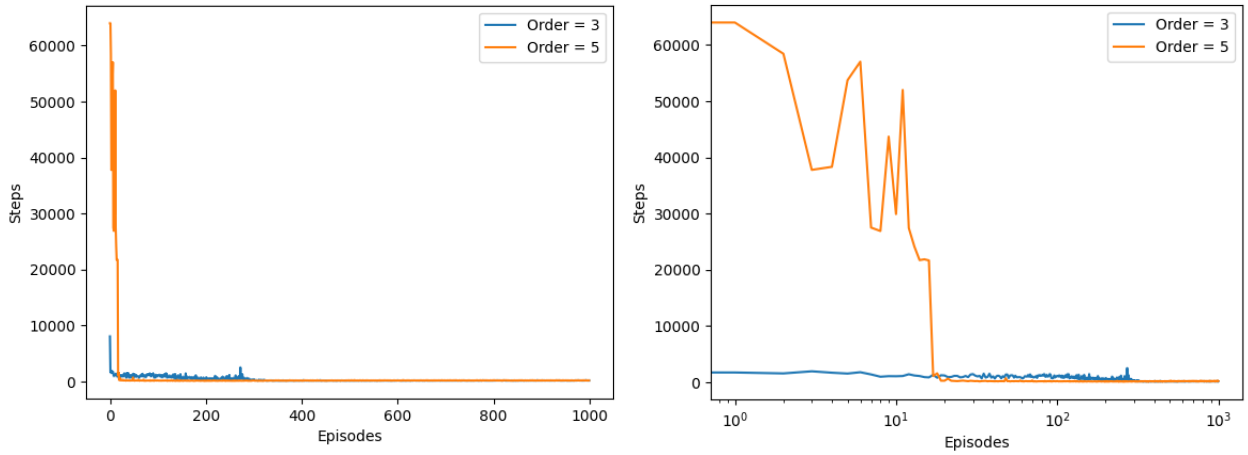
```
      For each Step in episode:
            z[:, a] ← z[:, a] + ϕ(s)
            Take action a, observe s' and R
            δ ← R - Q(s, a, w)  # Q = w[:, a]·ϕ(s) or 0 if s is terminal

            If terminal:
                  w ← w + δz·α
                  Break

            a' ~ ε-greedy(s', w)
            δ ← δ + γQ(s', a', w)
            w ← w + δz·α
            z ← γλz
            s = s'
            a = a'
```

5. Mountain Car Using Polynomial SARSA(λ)

Polynomial SARSA(λ) was implemented with accumulating traces, α = 0.00002, ε = 0, γ = 1, and λ = 0.9, with a max step count per episode of 64,000. Two orders of FSL were trained; O(3), O(5) - each with 1,000 episodes.

As with Fourier basis functions, each Polynomial basis function ϕ_i was assigned a corresponding $\alpha_i = \alpha$ / || $\mathbf{c}^i$ ||, where $\mathbf{c}^i = \langle c_1, c_2, \ldots, c_d \rangle$, $c_j \in \{0, 1, \ldots, n\}$. In the case where $i = 0$, $\alpha_0 = \alpha$ because $\mathbf{c}^0 = \mathbf{0}$ and so || $\mathbf{c}^0$ || = 0. d is the number of degrees of freedom of the model which, in the case of the Mountain Car problem described in Sutton & Barto, is two (position and velocity). n is the order for the basis functions. In the case of FSL, this results in $(n + 1)^d$ total basis functions when using the reduced $\phi_i(\mathbf{s}) = \cos(\pi \mathbf{c}^i \cdot \mathbf{s})$, where $\mathbf{s} = \langle s_1, s_2, \ldots, s_d \rangle$, $s_j \in [0, 1]$.

The learning curves (averaged over 3 runs) are shown below.



(a) (b)

Figure 6: Steps per episode vs Episodes (a) (and with a logarithmic x-axis (b)).

Only orders three and five were chosen because even at order five the training process took a significantly long time, and training on order seven would have been impractical. Nonetheless, polynomial basis functions can technically be used and will converge given a large enough time step limit and small enough α. Though given that Fourier basis functions are just as easy to implement, a polynomial's performance leaves much to be desired and doesn't serve much practical purpose for this specific problem.