Dustin Richards
CSC549 Programming Assignment 3 Report

Foreword

This assignment has baffled me and thrown me for a loop the entire time I've been working on it. I started out by doing what I *thought* was a good, logical first step: I wrote my own little mountain car simulator. I got to have fun with physics math – you could set the gravity, the mass of the car, and the force the car drives forwards with. It worked great and only took a couple hours, cool.

I then spent quite a while staring at the book and various internet resources trying to get a decent enough understanding of true online Sarsa(λ) to write it up in Python. After some typing and debugging, I managed to turn the strange symbols into functional Python code. It sometimes drove up the wrong side of the valley, but I figured that could be fixed with some reward function tweaks. Overall, it made it up the correct side of the valley "most of the time"-ish.

Later, I discovered that the book placed bounds on the car's velocity and position. This would prevent the car from driving up the wrong side of the valley, so I implemented it. Doing this completely broke the interaction between my happy little ideal-physics-accurate simulator and my Sarsa learner.

As it turns out, the book has its own math to simulate mountain car physics and I didn't even need to write my own simulator. I threw my fun, fancy physics out and replaced it with the book's monolithic and boring $x_{t+1} \doteq bound\left[x_t + \dot{x}_{t+1}\right]$ and $\dot{x}_{t+1} \doteq bound\left[\dot{x}_t + 0.001A_t - 0.0025\cos\left(3x_t\right)\right]$. Funny enough, this worked just fine with the physics bounds suggested by the book and my learner began working just fine again, now unable to climb up the incorrect side of the valley.

In the context of this report, you're also missing out on plenty of pain and suffering caused by me setting the axis limits on my value function plots incorrectly, leading to an entire day of confusion when I couldn't get my plots to look like those in the book.

Algorithm Overview

The core of this program is based on the mountain car simulator code from Sutton & Barto, a combined understanding of the Fourier basis from both Sutton & Barto and the Konidaris paper, and the true online Sarsa(λ) algorithm from Sutton & Barto.

pa3.py

This file, on its own, isn't that interesting. The main loop is pretty much a direct transcription of the true online Sarsa(λ) from Sutton & Barto. We run 1000 episodes in total. For each episode, we start by choosing an ϵ-greedy action based on the current weight vector $\mathbf{w}$ applied to our Fourier basis. For each step of the episode, we get the reward for that step and do some math to figure out the quality of the current state as well as the state we'll next step into. This information is used to adjust the eligibility trace as well calculate the new weight vector $\mathbf{w}$. The episode continues until the car reaches its goal, an x coordinate > 0.5. Each consecutive episode improves on $\mathbf{w}$, thereby improving the value function.

mountain_car_sim.py

Implements the mountain car simulation code detailed in Sutton & Barto. What was once a simulator accurate to ideal physics (I think) with fully adjustable parameters has been boiled down to a couple of boring equations.

fourier_basis.py

This file is where the magic happens. To instantiate the class *FourierBasis*, all we need to provide is the number of state variables we want to use and the order of the Fourier approximation we want to use. An alternate use case, where the vectors $\mathbf{c}^i$ are directly provided, is used only for running the pre-trained model. From this order, we determine the number of basis functions as $(order + 1)^{numStateVars}$. We then generate the vectors $\mathbf{c}^i$, each of which is one of every possible combination of the numbers $[0, order]$, including reverse order. As an example, for order 1, $\mathbf{c}^i = [[0, 0], [0, 1], [1, 0], [1, 1]]$.

The *calculate()* function does the actual Fun Math Stuff once we have $\mathbf{c}^i$ set up. First, we normalize all three state values to be in the range $[0, 1]$. Once that's done, all we have to do to generate our feature vector is, for each basis function i, take $\cos(\pi \mathbf{c}^i \mathbf{x}^T)$, where $\mathbf{x}$ is the normalized state vector.

utils.py

A few utility functions that keep the code in *pa3.py* from being a horrendous mess. The foundation is *getQuality()*, which evaluates the quality value of a given state-action pair given the weight vector $\mathbf{w}$ and a *FourierBasis* class to describe how the value function should be evaluated.

chooseBestAction() simply finds the action with the highest quality given some state and a *FourierBasis* class.

chooseBestActionEGreedy() does the same as *chooseBestAction()*, but adds in some random chance ϵ of choose a random action. This is used in *pa3.py*, but we may as well be using *chooseBestAction()* since ϵ is kept at 0. The mountain car scenario solves just fine without forced exploration.

replay.py

This file takes a pre-trained model, plays out one episode with it, then displays a learning curve that was recorded during training and plots the value function.

1. Show learning curves for order 3, 5, and 7 Fourier bases, for a fixed setting of α and ε, and γ = 1, λ = 0.9.

I spent quite a while getting these curves to look like this. Jonathan and Anoushka eventually tipped me off to the fact that I should be scaling my state values to between 0 and 1 before running them through the basis function calculations. This makes a lot of sense in retrospect, but it would have saved me quite a bit of time if the wonderful individual who wrote the Fourier basis paper would've mentioned this little detail somewhere. (Looking back at the paper, it looks like it may be mentioned, but is worded in potentially *the most confusing way possible*. I honestly can't tell.)

Used $\alpha_1 = 0.001$ and $\epsilon = 0$.

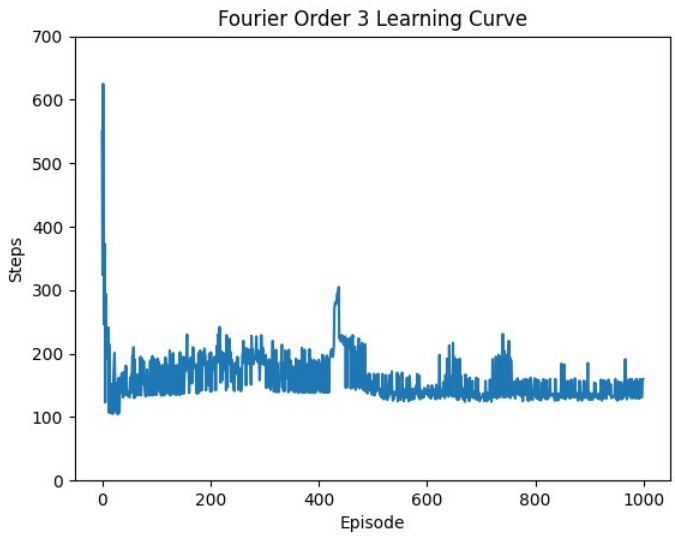

Figure 1: Learning curve for Fourier order 3.

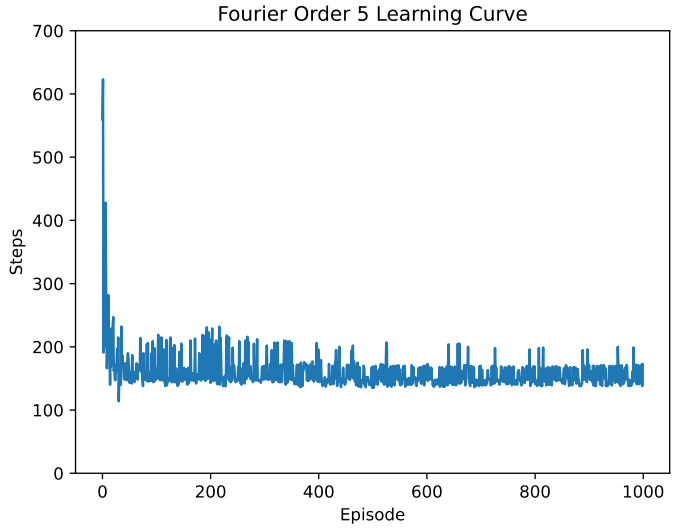

Figure 2: Learning curve for Fourier order 5.

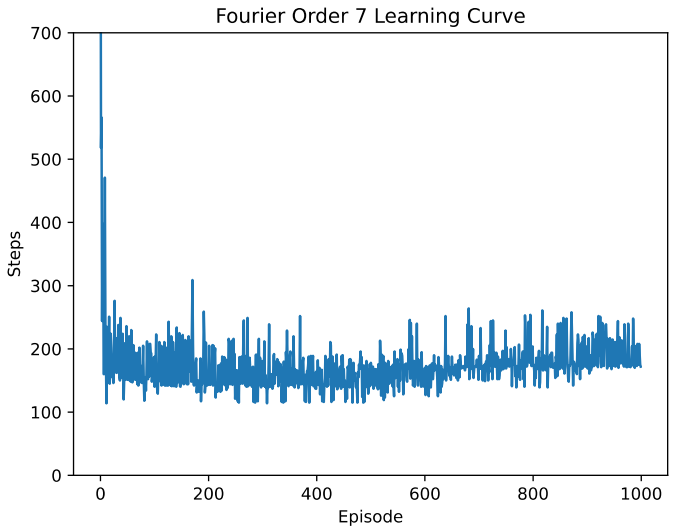

Figure 3: Learning curve for Fourier order 7.

2. Create a surface plot of the value function of the learned policies after 1,000 episodes, for the above orders.

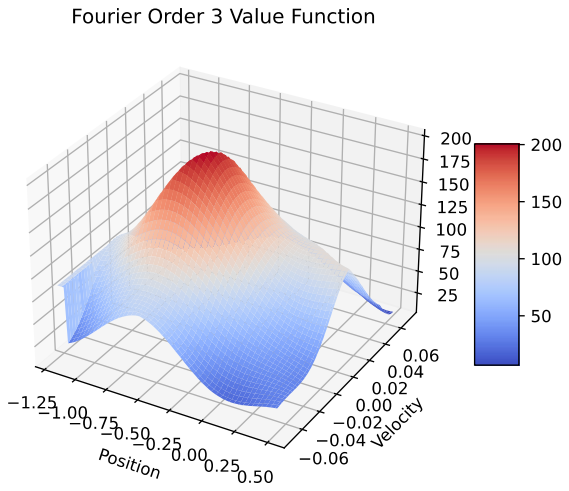

Figure 4: Value function for Fourier order 3.

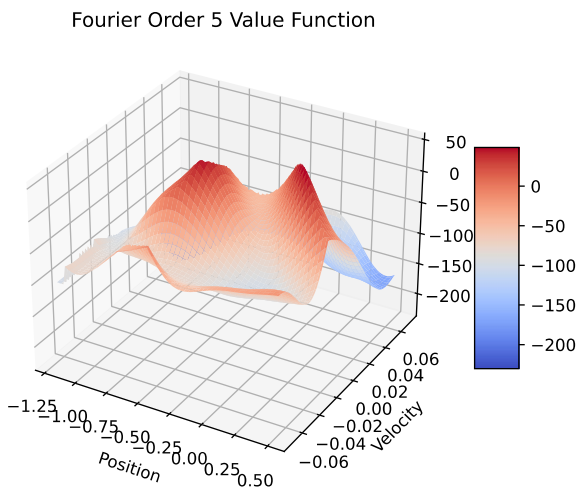

Figure 5: Value function for Fourier order 5.

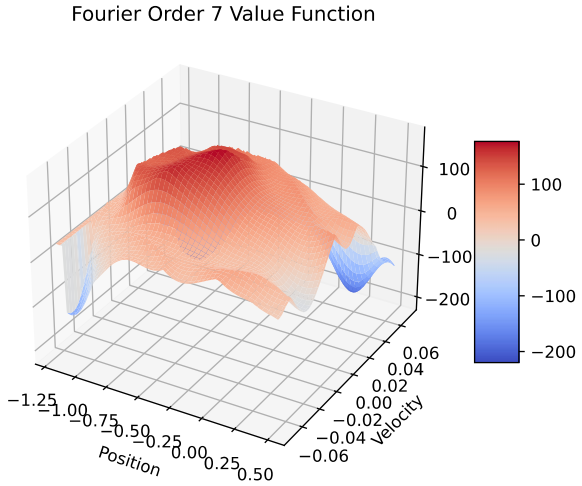

Figure 6: Value function for Fourier order 7.

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 γ < 1?

Looking at the lines in the true online Sarsa(λ) pseudocode from the book where γ comes in:

$$\delta \leftarrow R + \gamma Q' - Q$$

The effect of Q' on δ will be diminished here, requiring the quality of the next state to be better in order to have an impact.

$$\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^T\mathbf{x}\right)\mathbf{x}$$

A lower γ value here will result in eligibility traces decaying more quickly. It will also work to increase the impact of the feature vector $\mathbf{x}$ on the eligibility trace vector $\mathbf{z}$.

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\left(\delta + Q - Q_{old}\right)\mathbf{z} - \alpha\left(Q - Q_{old}\right)\mathbf{x}$$

Just as above (since γ is indirectly involved via δ), the effect of Q' is diminished, requiring it to have a more clear lead over the quality of the last state Q_{old} in order to have an impact.

In the case of $\gamma < 1$ and a many-step-long solution, learning would likely be slower but may come to a more accurate conclusion.

In the case of zero step cost and $\gamma = 1$, we would see the value of δ as higher on most iterations, increasing the effect of the eligibility trace vector $\mathbf{z}$.

In the case of zero step cost and $\gamma < 1$, we might see the value of δ as higher on most iterations (less so than with $\gamma = 1$), somewhat increasing the effect of the eligibility trace vector $\mathbf{z}$, along with a more quickly decaying eligibility trace vector.

Program Usage

To train: `python3 pa3.py`. Change *fourierOrder* on line 26 to adjust the order of the Fourier basis approximation.

This will create a `results` directory with timestamped directories inside of it, one for each run of `pa3.py`.

To view results: `python3 replay.py results/yyyy-mm-dd_hh-mm-ss 999`. The 999 indicates the index of the episode at which we're viewing the training progress and will view the final model. Choose 100, 200, 300, etc. to view partially-trained models.