Mountain Car Problem

Karissa Schipke

Programming Assignment 3

Advanced AI – CSC 549

12-2-2022

REPORT

For implementing the Sarsa(λ) for the Mountain Car problem using linear function approximation with Fourier basis functions, I chose $\alpha = 0.001$ since that is what they did in the paper (https://people.cs.umass.edu/~pthomas/papers/Konidaris2011a.pdf), and I chose $\varepsilon = 0.03$, so that there would still be some exploration due to randomness. However, exploration is fairly limited since ε is very close to 0. As given in the assignment, I had $\gamma = 1$ and $\lambda = 0.9$.

The learning curves and surface plots are in the following results section.

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?

I believe that it would learn more efficiently since it is considering more states since there are more steps. Also having a γ that is less than 1 would cause it to be less accurate. I would say a higher γ is better than a lower γ, and more steps is better than less steps but slower.

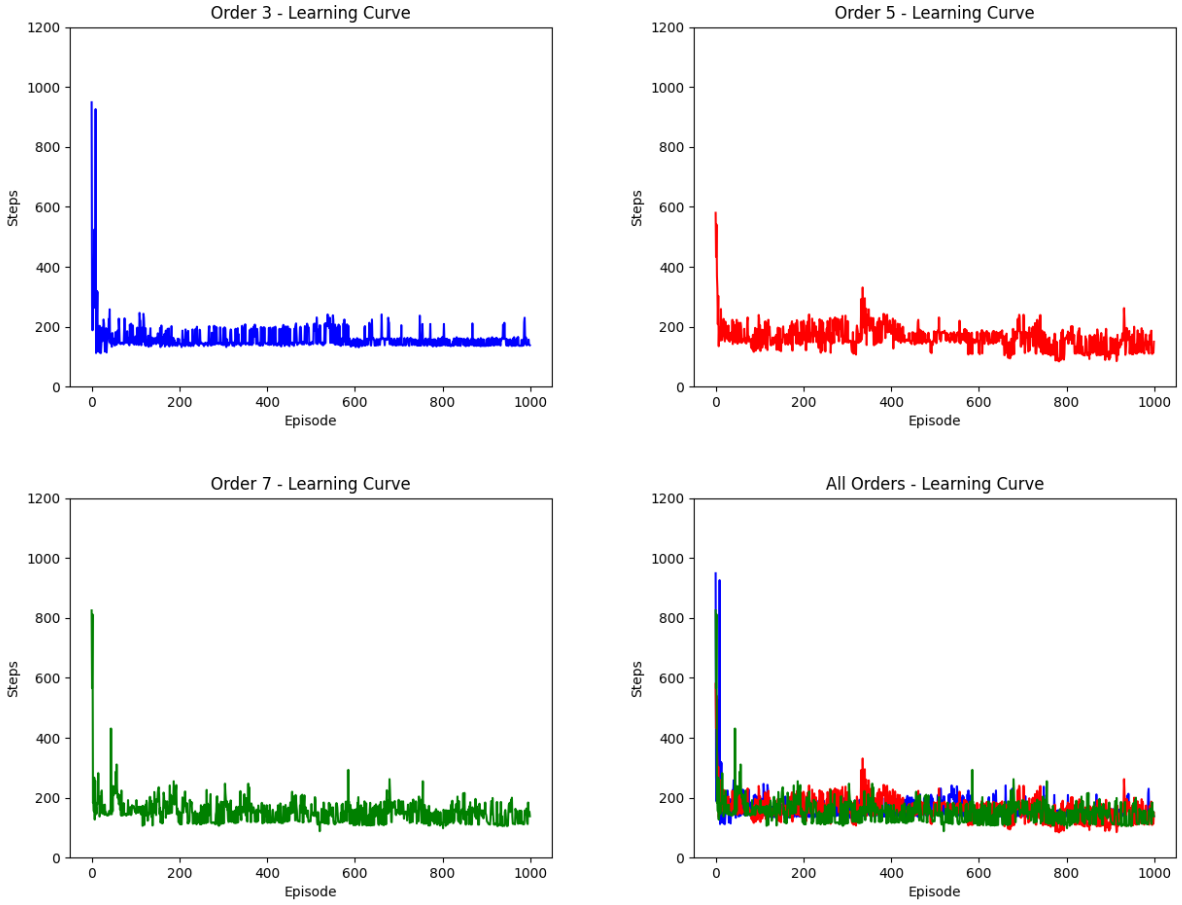
What would happen if we had a zero step cost and a positive goal reward, for the case where $\gamma = 1$, and the case where $\gamma < 1$?

Since there would be no cost to how many steps, it could take a long time to learn since there is no reward for having more/less steps. There is no incentive to learn in fewer steps. When $\gamma = 1$, it would perform well, but when $\gamma < 1$, it would not learn very well, because it could continue making the same mistakes for every step and there would be no reason to stop. Again, I would say a higher γ is better than a lower γ, and more steps is better than less steps but slower and, in this situation, it would not really help to have more steps since it is not getting a reward from it.

The instructions for how to run the code are in the code section below the results section.

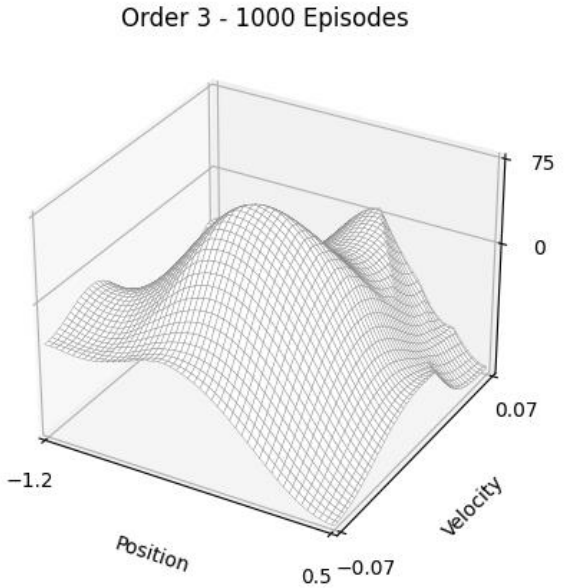
RESULTS

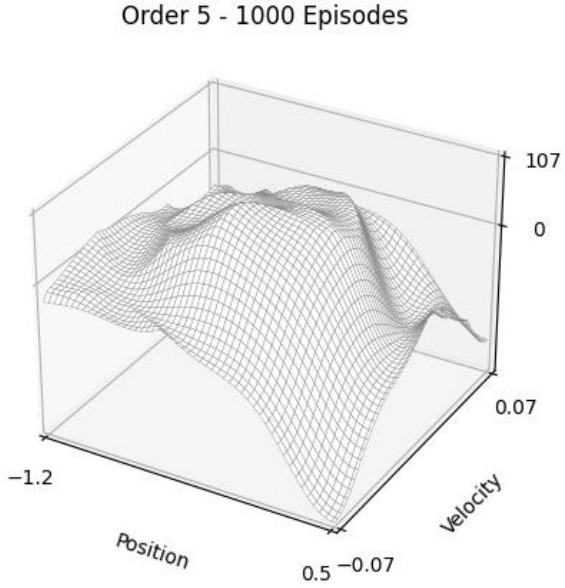
Learning Curves:

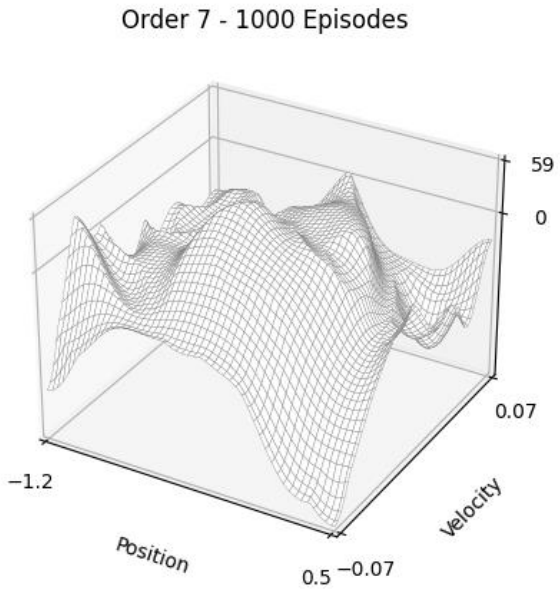


The learning curves show that all the episodes had less than 1000 steps with most of the episodes having less than 400 steps. Additionally, the steps are decreasing further into the number of episodes which is desired. You can see that the order 3 graph has the best learning curve because it does not fluctuate as much and is pretty consistent. The order 5 and 7 graphs fluctuate more and are less consistent, so the learning curve is not as good.

Surface Plots:







The surface plots show the value function of the learned policies after 1000 episodes. The smoothest of the plots is by far the Order 3 – 1000 Episodes plot. This is probably the best plot since our goal is a smoother plot and it has the best learning curve. The Order 7 – 1000 Episodes plot is by far the roughest or least smooth plot, indicating that the policy is not ideal. The roughness of the Order 5 – 1000 Episodes and Order 7 – 1000 Episodes makes sense since the learning curves were not as good as the order 3 learning curve.

CODE

There are three different files for the code: mc.py which contains the MountainCar class, fourier.py which contains the FourierBasis class, and sarsa.py which contains the SarsaLam class.

MountainCar Class

The mountain car class has two member variables: the position and velocity of the car. On creation of the mountain car class, the position is set to a random value between -0.6 and -0.4 and the velocity is set to 0. There are four functions in the class: getState, setState, takeAction, and animate.

The getState function returns the position and velocity of the car as an array.

The setState function takes in a position and velocity and sets them to be the position and velocity of the car.

The takeAction function returns the reward, position, and velocity based on the equation in the book. The np.clip function that is used within makes sure that the result from the equation is within the range of the problem, i.e. the position is between -1.2 and 0.5 and the velocity is between -0.07 and 0.07.

The animate function was for personal use to validate that the class was working as expected; it just plots a curve that doesn't necessarily line up with the range of the problem but shows how the car moves to the left/right on the example curve.

FourierBasis Class

The Fourier basis class has three member variables: n for the order, d for the dimension, and c for the coefficients. The coefficients are all the different combinations based on the order and dimension. On creation, the Fourier basis class takes in n and d and initializes its member variables to those values and initializes the coefficients based on the passed in order and dimension. There is only one function in the class: feature.

The feature function takes in a state and normalizes it and then performs and returns the basis function on the state.

SarsaLam Class

The sarsa lambda (lam) class has several member variables. The variables that do not change are alpha, lambda (lam), epsilon, and gamma. As stated before, I chose $\alpha = 0.001$, $\varepsilon = 0.03$, $\gamma = 1$,

and $\lambda = 0.9$. The class also has some changing variables such as the weights, feature, mc, and n. On creation, the sarsa lambda class takes in n, d, and feature and initializes is variables with those inputs. It also initializes the weights to be an array of length $(n + 1)^d$ filled with zeros. mc is initialized to a MountainCar object and every episode it is reset to the initial MountainCar object. There are four functions in the class: run, f, chooseAction, and calculateValue.

The run function is where the sarsa lambda algorithm is implemented; this is where it loops through each episode and step to update the weights based on states/actions. Some of the plotting is also handled in this function.

The f function is used for the plotting to create the Z axis of the surface plots. It just uses the position and velocity to get the value at the position/value.

The chooseAction function chooses the action based on the policy whether that is the greedy policy or random (based on epsilon value). If it is not random, this function returns the best choice based on the greedy algorithm.

The calculateValue function is used in the chooseAction function and calculates the value of a given action based on the current weights, Fourier basis, and state of the mountain car.

I also have my main in the sarsa.py file which is how I chose to run the code. To run the code, you would need to initialize n and d and pass them into a FourierBasis object and then pass n, d, and the FourierBasis object into the SarsaLam object. Then you would call the run function based on the SarsaLam object.

This is where I set everything up (the curves.append is for my plotting stuff).

```
fb = FourierBasis(n, d)
sl = SarsaLam(n, d, fb.feature)
curves.append(sl.run())
```

The plotting is not set up to work for other orders besides 3, 5, and 7.