SARSA(λ) Mountain Car

Alexis Hanson

December 2, 2022

Task:

The task for this programming assignment was to implement SARSA(λ) for the Mountain Car problem in Sutton and Barto. I chose to implement this was in Rust, which I would come to regret to a significant degree. The environment dynamics are programmed to the specifications in the book, and the agent is given 3 options to choose from. In order to learn these actions, I implemented a slightly modified version of the true online algorithm in the book.

Troubles:

There were a pair of bugs in my implementation which resulted in the last-minute completion of this project, one with the environment, and one with the RL algorithm proper. Both of these bugs obscured the other, and resulted in dozens of hours of debugging which were terminated by Johnathan's aid, for which I shall remain eternally grateful as I likely would not have found the second bug by the deadline otherwise.

The first bug was due to a misunderstanding in my translation of the given SARSA(λ) algorithm into functioning in an action space, as the given update for the weights needed to be split into two portions, one applied across the weights for all actions which was the same as for the episodic SARSA(λ), and an additional term specifically for the most recently taken action. When this section term was applied across all actions, the weights diverged spectacularly as soon as the learner took enough of one consecutive action to zero out the traces of the others. This resulted in an oscillatory behavior where the algorithm would learn, diverge, and learn again, and the difficulty of diagnosing this was compounded by the next issue. I will note that, despite fixing the function diverging, my solution for this issue did *not* result in the value function remaining wholly negative as it should. At least it learns now, I guess, and the value function looks right in shape, although weight 0 really should be different.

The second bug resulted from an early decision in implementation where I decided that, since each state will have an action taken at that state, the state and action should be tied together in a structure. This made my implementation quite clean, allowing me to pass no more than a model and state struct to most of my functions, but it also obscured a bug in my main loop where my actions were offset by one from my states, such that S′ was bound together with A instead of A′, and S was bound with the previous action. This would occasionally work, but the learner not correctly associating transitions with actions meant that it was highly inconsistent, and the combination of these two bugs resulted in extremely intermittent behavior.

A third problem which was not a bug in and of itself but resulted in several is how Rust as a language can frequently seem to be in alpha, especially when it comes to core library support. The primary examples of this are the ndarray and plotters crates, which are meant to approximate numpy and matplotlib, respectively. Some of ndarray's issues when compared to numpy arise as a result of Rust's extreme type strictness and mutability restrictions, as many things you would just expect to be able to do in numpy result in violations of one of those. Most notably, I was unable to find a method to add an array to a mutable array view generated by slicing, so on each occasion where this is needed I ended up bundling large quantities of iterators together for element-wise math in what would have been one concise line in numpy, which massively increased the number of points at which I could and did make mistakes. My primary complaint with plotters, on the other hand, is that, while purporting to be like matplotlib, it requires an ungodly amount of boilerplate, where a simple ”plt.plot()” in python requires a dozen lines of Rust code which, in effect, specify that yes, I actually *did* want to make a plot of this thing. There is so much boilerplate, in fact, that it was easier to copy and modify the github examples for my purposes than it was to figure out what flags were available and which ones were needed for each plot, and I still haven't figured out how to label axes on a 3d graph, as the method for 2d did not carry over.

Data:

Now for the part of the paper which is less wall-of-texty and more oh-god-my-eyes-what-is-that-color-scheme-y (the color scheme is HSV except it's H,1,0.7, imagine having convenient color scales available). The convergence plots below (which took over 2 hours of googling and tweaking to generate) demonstrate convergence for each fourier basis (they would be on one page, but my tolerance for wrestling with miktex on whether the subfigure package is installed has somehow reached negative levels, no clue how that could have come to pass):

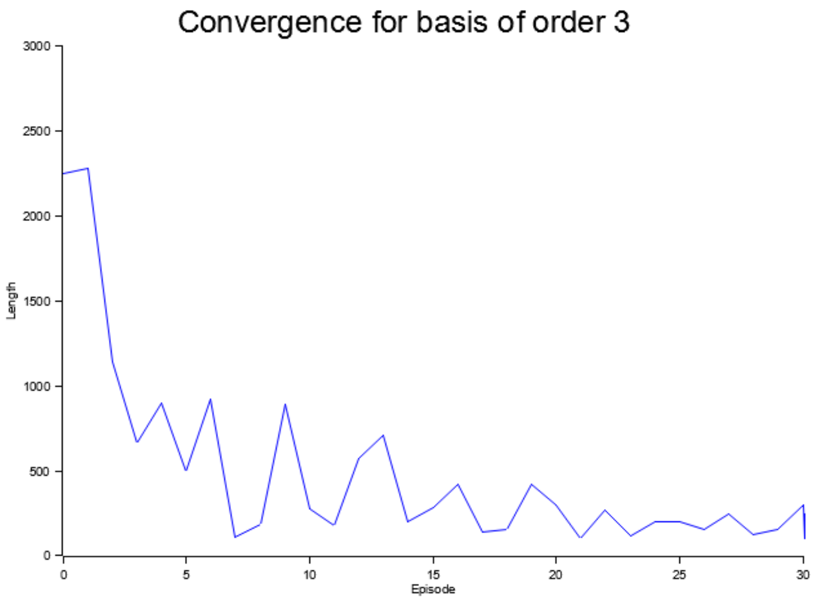

Figure 1: Convergence for Order 3 basis

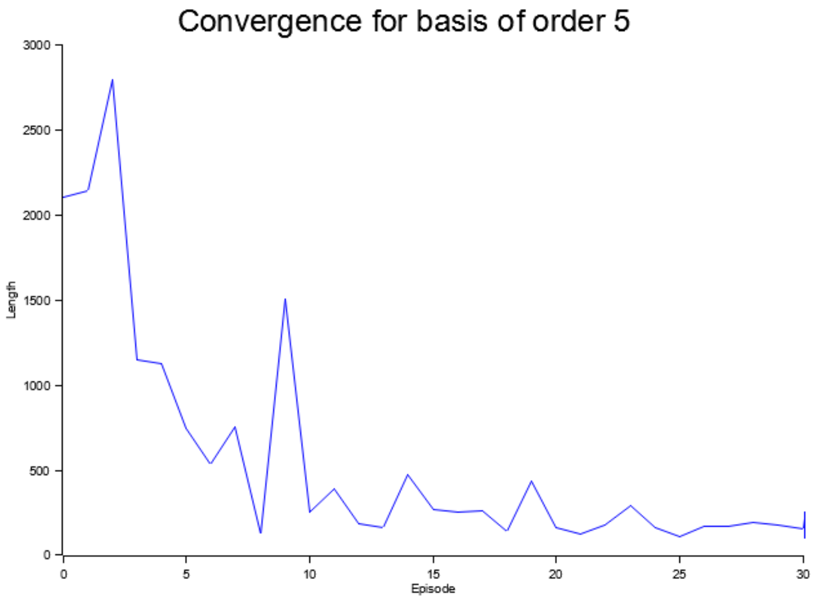

Figure 2: Convergence for Order 5 basis

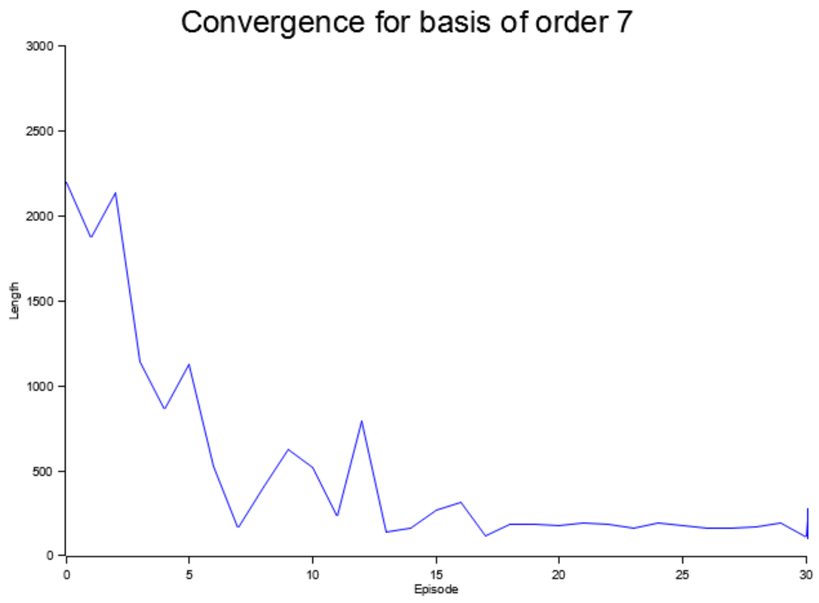

Figure 3: Convergence for Order 7 basis

As can be seen from these charts, increasing the order of the basis *might* slightly increase the speed of convergence, but if so it is not by a significant amount relative to random variation in convergence. Furthermore, the algorithm is generally converged by episode 15 to 20 or so.

As for the plots of the cost function, I still could not figure out how to label the axes so they will be described in text. The x axis goes from -1.2 to 0.5, the axis labeled from -0.06 to 0.06 describes xdot, and the vertical axis is theoretically the cost of being at a state, which goes negative an uncomfortable amount of the time for a problem with no positive rewards. They all have roughly the same shape, with the higher order surfaces having slightly more nuance, as expected. The most notable feature is the cost function near the end in the order 3 basis, as it is very sharply peaked compared to the higher order representations. The convergence does seem to be extremely stable, as inspection of the value function after several 1000 episode samples yields no significant difference for each order.

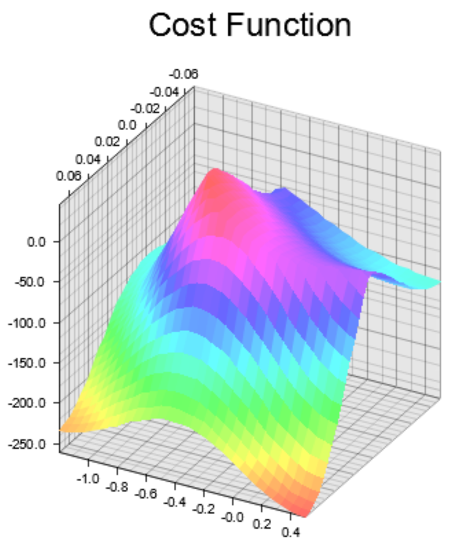

Figure 4: Cost function for Order 3 basis

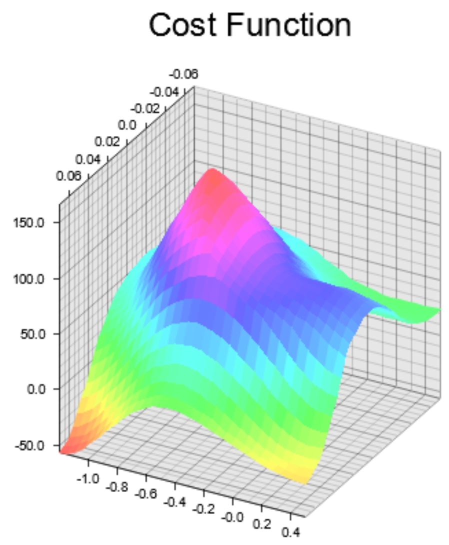

Figure 5: Cost function for Order 5 basis

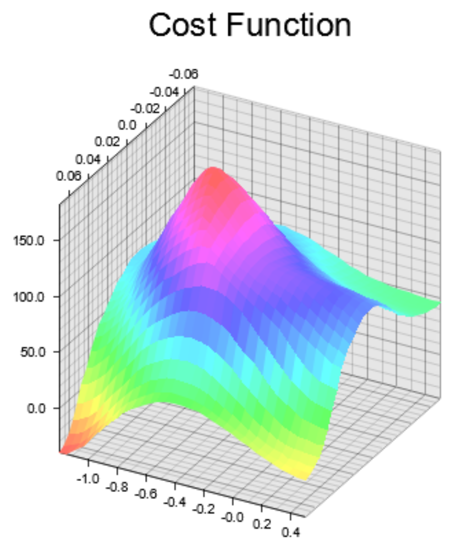

Figure 6: Cost function for Order 7 basis

Conclusion and last question:

The most performant model overall was counterintuitively the order 3 one, which averaged 156 cycles to complete the objective over 5,000 episodes, as opposed to the order 5 and 7 models' 167 and 171, respectively. After noting this, I also determined that the order 2 version averages 163 cycles, and the order 4 165 (the order 1 obviously doesn't converge). This seems to suggest that the third order basis is optimal with my algorithm, and that perhaps overfitting is occurring with higher orders, but the paper author's code uses a 10th order estimator and gets better results, so it could also just be more flaws with my method. (Increasing ϵ only had the effect of increasing the variance of results, not reducing them)

As for the actual questions, if gamma is too small the model will fail to converge, and in this problem "too small" appears to be some value between 0.96 and 0.97. I believe this is due to the reward n steps in the future being weighted by γ^n, and thus in the several thousand steps it takes to initially reach the objective, the weighting on the reward becomes practically nonexistent.

With a zero step and positive goal reward, regardless of gamma, the model has no drive to explore states it hasn't seen, and thus even with an epsilon value it only makes small random walks around its origin point with no real hope of running into the goal. With a negative step reward, anywhere it hasn't been likely has a better weight than its starting position, resulting in active exploration to find the solution.