CS 5100 Final Project: Solving Mountain Car with Reinforcement Learning

Vaughn Franz

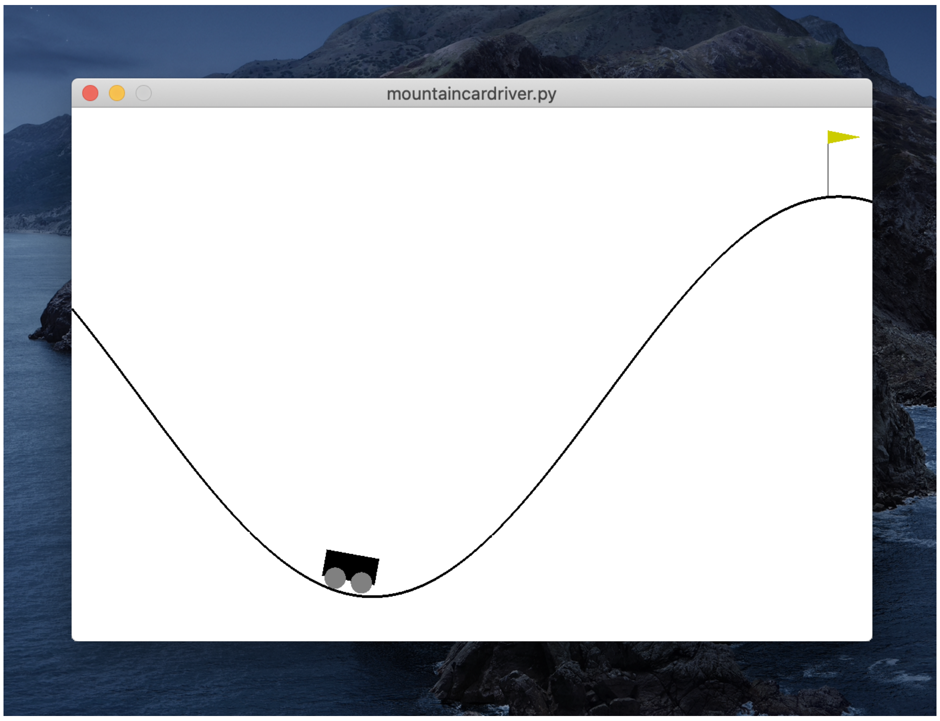
December 15, 2020

Abstract

For my project I chose to work in the version of the mountain car environment available from OpenAI Gym. To start out, I took a naive approach to the problem, choosing to implement the Q Learning algorithm, which we also ended up studying in some detail in class. The implementation was successful, relying on tabular storage of the Q function during training. With inspiration from Sutton & Barto's *Reinforcement Learning: An Introduction*, I then tried an implementation of an epsodic semi-gradient one-step SARSA algorithm relying a tile coding implementation to approximate the Q function. This strategy, was able to perform much better than the tabular version. It learned more quickly and more effectively.

1 Introduction

1.1 High Level Description and Motivation



The mountain car environment tasks a small car to drive itself up a steep hill. The environment is constrained to two dimensions for simplicity. To start the car is in a valley, and it does not generate enough force to directly drive rightward up the hill to the goal. The only way that it can get to the top of the hill is to drive back and forth in the valley to leverage its potential energy. I chose this problem because I wanted to get more exposure to reinforcement learning and simply because the environment seemed interesting to me. Another motivation was that I had a good indication that the problem was simple enough for me to solve in the amount of time I had to work on it.

1.2 Problem Statement

The specific implementation of the mountain car that I used came from OpenAi Gym (Reference 1).

1 **State**: The agent's observations of the environment are by two floating point numbers representing the position and velocity. The position, p takes values in the range $[-1.2, 0.6]$ and the velocity takes values in the range $[-0.07, 0.07]$. The agent gets an observation of each of these numbers at each time step.

2 **Actions**: The agent has a choice of three actions at any given state outside of the terminal state (position of 0.5). The actions are: accelerate leftward, no acceleration, and accelerate rightward.

3 **Reward**: The agent receives a reward of -1 for every position that is less than 0.5. At position 0.5 (the goal state) the reward is 0.

4 **Starting State**: The agent starts at a position sampled uniformly from the range $[-0.6, -0.4]$. The starting velocity is always 0.

5 **Terminal State**: The training episode terminates after the agent has moved for 200 time steps or it has reached the goal state (position 0.5).

2 Methods and Algorithms

2.1 Tabular Q Learning

The first approach taken was to use a tabular version of the standard Q-learning algorithm. Given a Markov Decision Process, the Q learner stores a Q function giving an approximation of the value of the state-action pairs observed in the environment. In this case the Q function is stored in a table with each entry corresponding to a state, action tuple. These values are the maximum expected reward given a state-action pair. Q-learning uses a temporal difference approach to update the Q function based on new observations of the state and the rewards gained. The version of Q learning used was with a simple one step look ahead for updates. The update step is represented by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

Figure 1: Q-learning update equation (Reference 2)

The basic algorithm used is as follows:

```
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  until $S$ is terminal
```

Figure 2: Pseudo-Code for Q-learning (Reference 2)

2.1.1 Making Mountain Car's State Space Discrete

Given the range of values for the position and velocity, it was necessary to discretize the mountain car's state space in order to use a tabular representation of the Q function. The position and velocity were each rounded to one significant digit. This leaves 19 possible positions and 15 possible velocities for a total of $19 \times 15 = 285$ total states.

2.1.2 ϵ-Greedy with Decay

During training the mountain car used an ϵ-greedy strategy. This means that with probability ϵ the agent selects a random action and with probability $1-\epsilon$ the action with the maximum Q value for the next state is selected. A strategy to decay the value of ϵ over the course of training was employed. The value was decreased by ϵ/n where n is the number of training episodes in between each episode and not going below a value of 0.001.

2.1.3 Experimental Procedure

To test the tabular implementation, I trained the agent for 5000 episodes each over a number of different values for the learning rate and starting value of ϵ. In particular, the values of α (learning rate) were [0.1, 0.2, 0.3, ..., 1]. Each value of alpha was tested on each of the values of epsilon for [0.25, 0.5, 0.75, 1]. In all cases the discount factor was set to 0.9. After training, the agent was run for a duration of 100 episodes with no random actions taken.

2.2 Function Approximation with Semi-Gradient SARSA

In an effort to extend the project further. I also developed an agent that employed a strategy for approximating the value of the Q function with a set of features. This is another attempt to wrangle the unwieldy size of the state space. The motivation for introducing another approach was that the function approximation could potentially represent the true Q function more accurately. This could therefore lead to either faster learning or the learning of a policy which provides higher average rewards after training.

I discovered while researching potential implementations that the mountain car problem is not very amenable to simple linear function approximations. The features have a strong "interaction" so to speak in that certain values of the position are only valuable (in terms of expected reward) when paired with certain values of the velocity. This type of interdependence confounds a simple linear function. Therefore, I used a tile coding strategy to construct the features of the environment and produce a workable linear approximation.

2.2.1 Semi-Gradient Methods

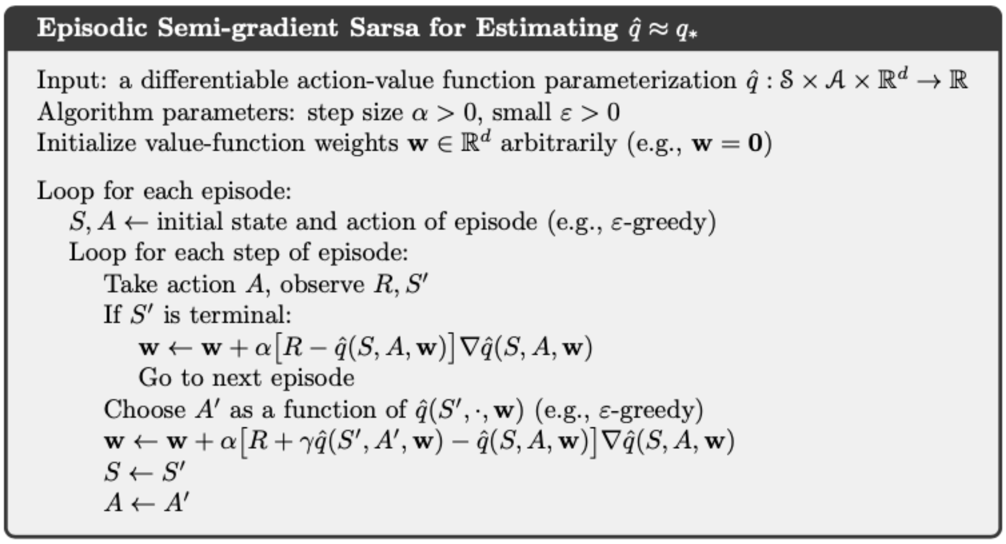
The idea now is that the state can be represented by a vector of features, and the q-value can be computed using a these in conjunction with a vector of weights. I will denote the vector of weights in keeping with Sutton and Barto, as $\hat{w}$ (or $\hat{w}_t$ for the weights at a given time step). The mechanism by which the agent learns is by adjusting this vector of weights. The method used to update the weights is very similar to that used by stochastic gradient descent methods. In stochastic gradient descent, the weight of a feature is updated in the direction that will minimize the error (specifically, the mean squared error). The equations for this type of update are as follows:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]^2 \\ &= \mathbf{w}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t),\end{aligned}$$

Figure 3: Stochastic Gradient Descent Update (Reference 3)

In our case, the value of $v_\pi(S_t)$ in this equation is unknown, and must be replaced with a value calculated based on an observation from a training episode. Furthermore, since the calculation of this value will be based on the current weight vector in the training episode. Thus, this value will be biased and the derivation of the Stochastic Gradient descent update that led to the above equation is flawed. It is for this reason that the method that I use, based on a one-step look ahead, is called a "semi-gradient." In general these methods do not converge as robustly as gradient methods (Reference 3).

The pseudo-code for the algorithm is given in Figure 4.



Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        If $S'$ is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$
```

Figure 4: Episodic SG SARSA (Reference 5)

2.2.2 Tile Coding

Tile Coding is the strategy by which an approximating function can be constructed such that it is a linear function of the vector of weights $\hat{w}$. Given a vector of features $\hat{x}(s)$ based on the state s, the approximating function is written as seen in Figure 5.

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s).$$

Figure 5: Linear Function Approximation (Reference 3)

This reduces the stochastic gradient descent update to a nice form as seen in Figure 6.

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\Big[U_t - \hat{v}(S_t,\mathbf{w}_t)\Big]\mathbf{x}(S_t).$$

Figure 6: Update rule for Semi-Gradient Method with Linear Function Approximation (Reference 3)

Where U_t can be taken to represent the calculated value from the observation. Tile coding accomplishes a linear function representation by partitioning the state space into tiles. A single tiling on the state space achieves a representation that aggregates a number of different distinct states together, something similar to what was done earlier on. However, multiple tilings are typically used, with each tiling offset from the others by some fraction of the tile width. The procedure is illustrated in Figure 6. Each tile of each tiling then corresponds to one feature.

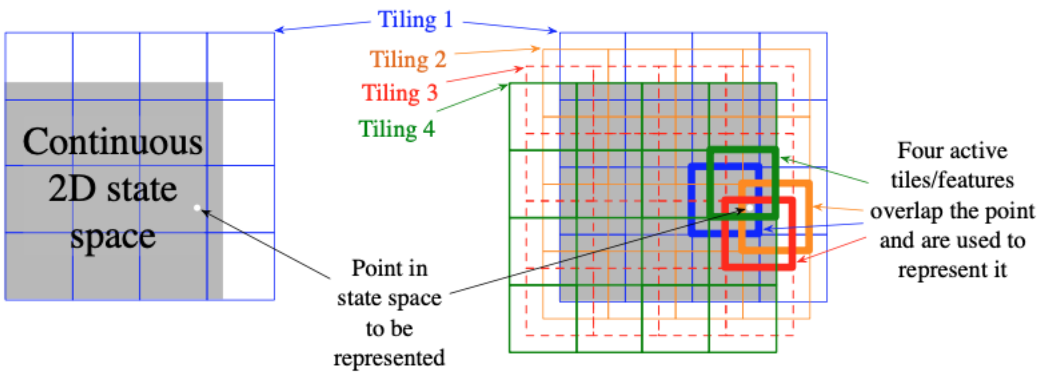

Continuous 2D state space
Tiling 1
Tiling 2
Tiling 3
Tiling 4
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure 7: Tile Coding (Reference 4)

So when a given observation is made, the number of features corresponding to the observation is equal to the number of tilings. In this way the updating of the weights of the features corresponding to a point in the state space will have impacts on the other points in the vicinity. The size of the tiles and the number of tilings will then both have impacts on the features of the learned policy over time.

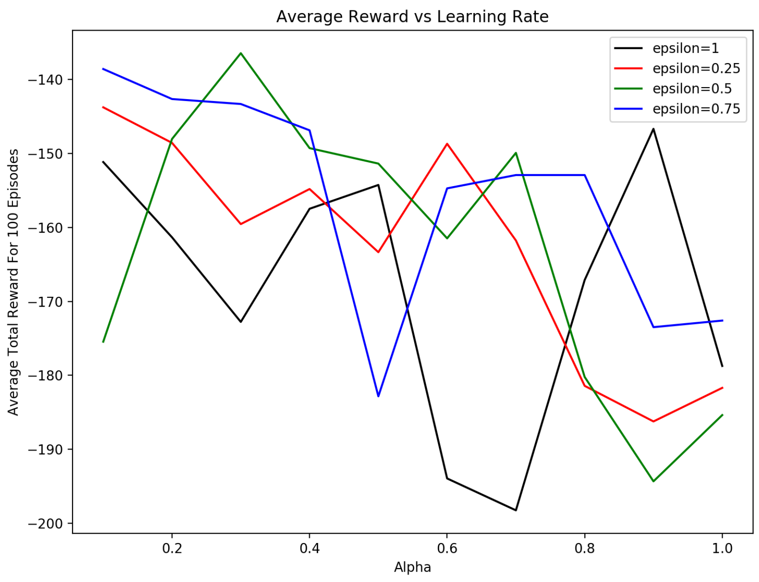
2.2.3 Experimental Procedure

Given my time constraints, I picked one tiling strategy and stuck to it. I used 8 tilings each of which was 8×8 in dimension. To assist in the implementation I utilized the tile coding software suggested in the Sutton and Barto text (Reference 5). This software had the added benefit of implementing asymmetrically offset tilings which have been proven to be more effective than uniformly offset tilings.

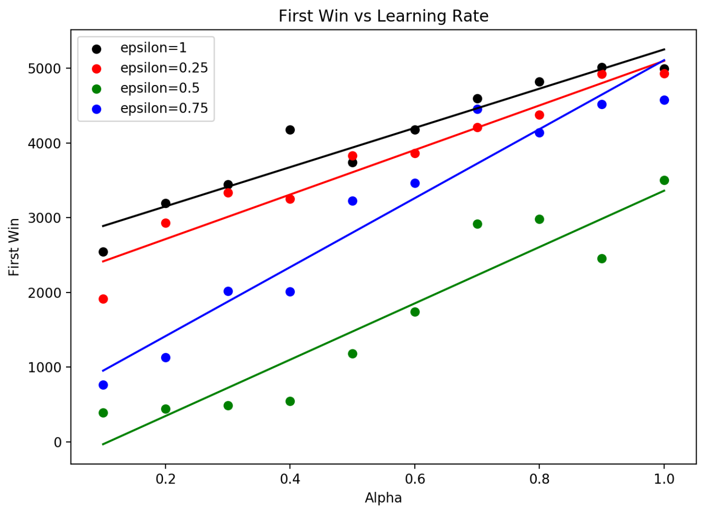
This implementation means, in particular, that each tile in each tiling corresponds to a binary feature, which takes a value of either 0 or 1 depending on the state. The features from the tiling together with the discrete value of the action are taken together to give a vector of features for each state and action pair.

The agent employing this strategy was able to learn much more quickly. So, I trained it over a number of different values for α and ϵ but this time for only 1000 episodes. The discount factor was set to 0.99 for all of these training episodes, because the lower factor of 0.9 indicating higher discount made it such that the agent never was able to learn an effective strategy. The tested values for α were [0.025, 0.05, 0.075,..., 0.125] and for each of these values the values of [0.2, 0.4, 0.6, 0.8] were tested for ϵ. The reason for selecting these values of α is that with the semi-gradient method update, a value of $\alpha = 1/n$ (n being number of tilings) results in exact one trial learning. Thus, it is necessary to make the value some fraction of that value (Reference 4).

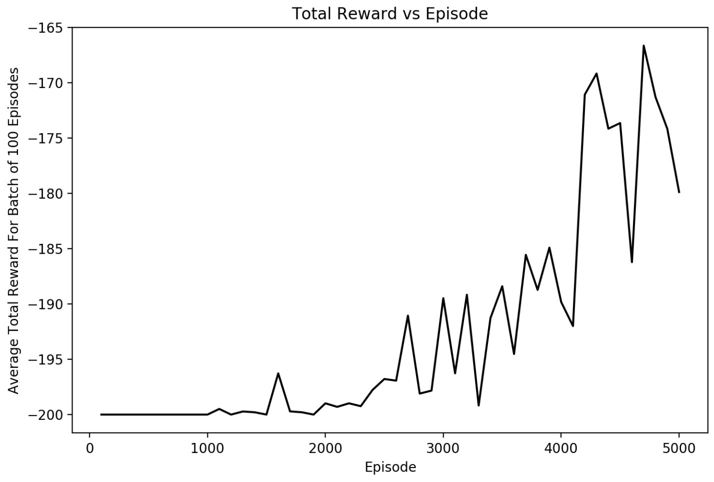
3 Results and Analysis



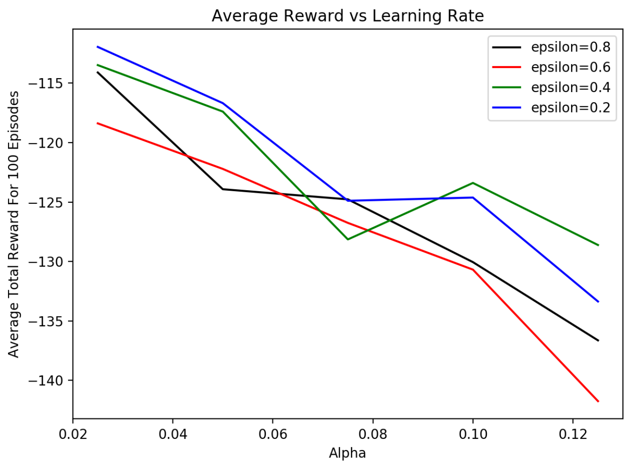
The above chart represents the average total reward obtained over 100 episodes by the tabular Q agent after training for 5000 episodes for various values of α and ϵ. The average rewards were obtained by running the agent in the environment for 100 episodes with no random actions, operating only with a greedy policy of choosing the action with the highest corresponding Q value in every state. There appears still to be a very large amount of noise in the data after this duration of training.
One clear trend does seem to be the preference for lower values of alpha and values nearer to 0.5 for epsilon.
Finally, the good part of this is that the agent is winning with some consistency at this point. Every result with a value greater than -200 is a scenario in which the agent reaches its goal.



This chart represents the episode at which the tabular Q agent was first able to reach the goal during training for each of the corresponding values of α and ϵ. Similar to the previous graph, this chart implies that lower values of α and moderate values of ϵ are preferable.



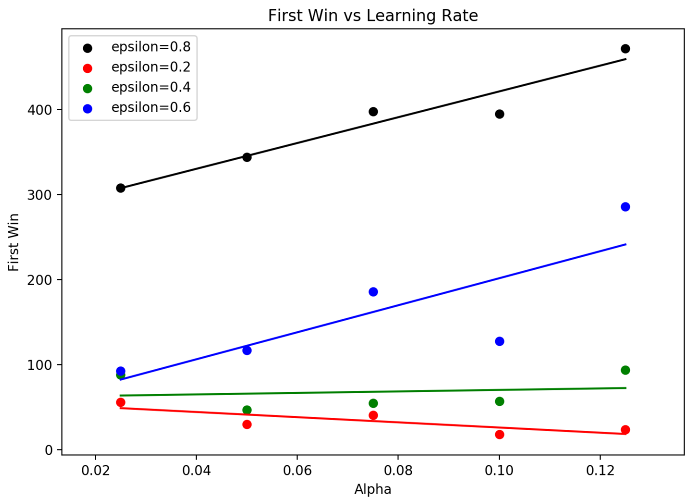
Finally this is the trajectory of the total rewards for each episode that the tabular agent received during training for one of the better performing tabular agents ($\alpha = 0.1$ and $\epsilon = 0.75$. The rewards are batched over 100 episodes and averaged to make the plot more readable. We can see the balance between exploration in the earlier episodes and exploitation in the latter. The Q values certainly don't seem to have converged at this point in the training and its unclear if they ever would with the coarseness of the rounding on the state representation.



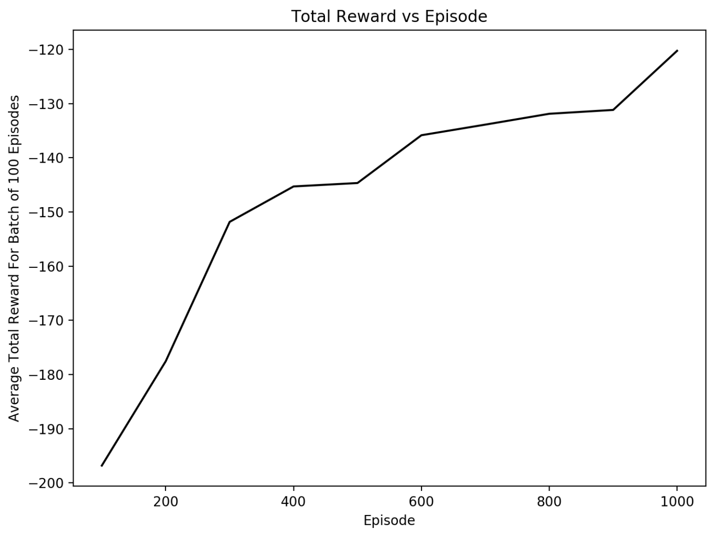
Here we see the average total reward for various learning rates for the Semi-Gradient SARSA agent. These are the average rewards over 100 episodes after the agent had been trained for 1000 episodes. Over the 100 episodes the agent took no random actions, again following a greedy policy with respect to Q values.

The first thing to note is that the agent performs much better in almost all cases than the tabular agent. The most successful tabular agent only achieved average total rewards of -140 after training for 5x as long as these agents. The worst performing agents in this sample manage to perform at that rate or better.

We can also see again a preference for lower learning rates. The difference this time seems to be a potential preference towards lower initial values of epsilon.



Here we see a plot of the episode at which each SARSA semi-gradient agent was able to win for the first time. The agent learns to reach the goal for many values of the α and ϵ parameters very quickly, within 100 episodes in many cases.



Finally we see the learning trajectory for one of the more successful agents over the first 1000 episodes of training. The rewards are batched over 100 episodes and averaged to make the plot more readable. The function approximation does not appear to have converged to a value at this point. We can see that the agent learns very rapidly at the beginning and appears to reach a plateau but still sees steady gains.

4 Discussion & Future Projects

There are a lot of areas for improvement with this project and opportunities for further development.

There are a number of things that I discovered that I would amend. One of the biggest things that surprised me was the difference that reducing the discount made for the SARSA SG agent. Going from 0.9 to 0.99 made the difference between never solving the environment and solving it very, very reliably. This could be a good area for exploration, testing different values for the discount for both agents. Another major thing that I would have altered in my experimental setup were the values of α and ϵ that I was testing for. In retrospect, it is easy to understand that lower values of α would be preferable. I could have targeted my testing to particular ranges of values that may have revealed more interesting trends. In particular for the SARSA SG agent, I would have liked to explore even lower learning rates.

There are a number of ways that the project could be extended as well. I would like to explore adding an eligibility trace to the SARSA SG agent. This was something that I read about briefly in the Sutton and Barto text. I would also be eager to explore different methods of function approximation. Some that I read about were using fourier basis functions or radial basis functions. I also read a bit online about deep Q networks, where neural nets are used to approximate the Q function. This method could be suitable for this problem.

5 References

1 OpenAI Gym - Mountain Car
`https://github.com/openai/gym/blob/master/gym/envs/classic_control/mountain_car.py`

2 *Reinforcement Learning: An introduction* by Sutton and Barto page 131

3 *Reinforcement Learning: An introduction* by Sutton and Barto page 201-205

4 *Reinforcement Learning: An introduction* by Sutton and Barto page 217-218

5 *Reinforcement Learning: An introduction* by Sutton and Barto page 243-246

6 Tile Coding Software
`http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLtoolkit/tiles.html`