ECE 5242 Intelligent Autonomous Systems

Project 4: Reinforcement Learning

Alana Crognale arc232

May 12, 2020

Introduction

Reinforcement learning aims to describe an optimal sequence of actions in order to maximize the total value or reward of the environment. In this project, our goal is to implement policy and/or value iteration, Q-learning, and REINFORCE algorithms on a variety of Markov Decision Process (MDP) environments (with discrete action spaces and both discrete and continuous state spaces). To do so, we will want to explore different hyperparameters in order to see which ones lead to faster convergences to the optimal policies, value functions, and Q-functions. We will be working with a given maze environment, described in maze.py, as well as the 'Acrobot-v1' and 'MountainCar-v0' environments from OpenAI Gym.

Problem Formulation

How can we implement Q-learning and REINFORCE algorithms in order to learn optimal policies for various MDP environments?

Technical Approach

Policy Iteration / Value Iteration:

We first initialize our environment, as well as our value matrix, Q matrix, policy matrix, and discount factor. The environment in this case is Maze(), which describes a discrete state space consisting of open spaces, blocked wall spaces, flags to collect, and a goal space to reach where we want to collect as many flags as we can, while reaching the goal in as fast a time as possible. In this environment, we have 14 open spaces to explore, and 3 flags to potentially collect, implying a total of 14 space * (2^3(flags)) = 112 total states. At any given point, we can take one of four actions corresponding to up, down, left, or right. At any given time, there is a 0.1 probability that we 'slip', i.e. take the action clockwise to the intended action (for example, if the intended action is down and the robot slips, the robot will then actually move left one space). If the action would lead to either a wall space or off of the map, the robot instead stays in its current position for that time. Thus, we can initialize our value matrix to be a 112-length zero vector, corresponding to the number of states, similarly a 112-length zero vector as our policy vector, and a 112 x 4 (number of states x number of actions) zero matrix for our Q values. Note that some of the values within this Q matrix are 'impossible' states that will never be visited (for example, it is impossible in this environment to lose flags once you have collected them). We are also given a discount factor of 0.9 to use in our implementation. To run value iteration, we iterate through all states s and actions a, updating Q[s,a], V[s], and P[s]. To update Q[s,a], we calculate the expectation of the sum of our discount factor (scaled) times our current reward, given state s and action a, i.e. the expectation of the discount factor times the value function at the next state s' plus the reward given state s and action a. To update V[s], we take the maximum of Q[s] across all actions a, so that we find the highest current reward value given the state s. Likewise, to update the policy P[s], we take the argmax of Q[s] across all actions a,

so that we find the action a which yields the highest value given the state s. Each 'episode' corresponds to one run through the full state action space, and we repeat this process iterating over some number of episodes until we have converged (we can set this to some small nonzero threshold, where we check whether our value function has updated more than this thresholded value). Ideally, our Q function, value function, and policy would then all converge to their optimal values.

Q-learning:

Q-learning approximates the action-value function until convergence using a model-free algorithm, an algorithm which does not directly incorporate the probability distribution (which can be helpful in situations where this distribution is unknown or otherwise very difficult to obtain). This is possible using a derivation from Bellman's Optimality Equation, where we start with an arbitrarily defined value function (which is equivalent to the max of the Q function at a given state s and action a, i.e. an arbitrarily defined Q function) and iterate until convergence in order to maximize the Q-function's expected value. Q-learning also uses epsilon-greedy exploration, demonstrating the tradeoff between exploration and exploitation as a hyperparameter to tune, such that, for every episode, a randomly selected action is chosen with probability epsilon, and an action according to Q (argmax Q[s]) is chosen with probability 1 minus epsilon. In this particular environment, we find that convergence occurs faster when we have an epsilon close to 1, meaning that random exploration proves to provide us a significant amount of helpful information to update Q to its optimal values. Thus, for each episode, we start at the same initial state, choose an action a according to our epsilon parameter, take one step (which provides us a current reward, next state, and whether the goal state has been reached), and update Q[s,a]. To update Q at this given state action pair, we must also define the learning rate hyperparameter, which describes how much past information is used to affect the new information (a value of 0 means that only past information is used, while a value of 1 means that only new information is used and past information is ignored) in stochastic environments. After tuning this parameter to 0.05, we update Q[s,a] by adding to itself this learning rate multiplied by the current reward plus the discount factor times the value at state s' (max(Q[s',:]) minus the Q value at the current state action pair. We then update the current state s to be equal to the next state s' and repeat until the goal state has been reached. The specifics of this algorithm were followed from page 131 of Sutton and Barto's "Reinforcement Learning: An Introduction".

REINFORCE and Q-learning on Continuous State Space Problems:

Instead of our discrete state action space maze environment, we now look at two different examples of continuous state, discrete action space environments in OpenAI Gym – the first is Acrobot-v1, and the second is MountainCar-v0 (full descriptions of these environments can be found in the corresponding open source GitHub). In both environments, we follow similar implementations. To implement REINFORCE, our hyperparameters include a discount factor (which we have increased to 0.99 in order to aid in faster convergence), as well as the learning rate alpha (specific values for alpha can be seen in the Results and Discussion section below). In REINFORCE, we define a parametrized and differentiable policy based on a set of parameters theta. There are

many different ways to define this parametrized policy - following this post, we define a policy which outputs the probability of taking each action at any given state s and parameters theta, such that we can use the softmax to help us later find the policy gradient. We additionally follow the algorithm from page 330 of Sutton and Barto's book, which implements REINFORCE with a baseline, where the baseline in this case is a moving average of the current total rewards. Adding a baseline into REINFORCE typically aids in faster convergence. Thus, we first set our policy such that we have equal probabilities of selecting each action (setting this type of randomized policy is typical when much of the environment is unknown or difficult to describe, however, sometimes a more ideal initial policy can be chosen that does not necessarily equally weight each action probability). Then, for each iteration, we generate an episode of state action pairs based on our policy until the terminal state is reached (thus, our first generated episode will be comprised of entirely randomly selected actions, which is done in part to help us explore the environment further). Then, we take the gradient w.r.t theta of the natural log of our policy defined by theta at action a given state s. We then want to update our baseline (in this case, take an updated average of our cumulative rewards), as well as our theta parameter. To update our theta parameter, we iterate through every time sample t in the generated episode, adding to theta the learning rate times G (the discount factor scaled exponentially based on t) times the difference between G and our baseline, times the gradient at time t. This whole process can be repeated as many times as needed to converge to an optimal policy. Q-learning in the continuous space follows a very similar implementation to the Q-learning described above. The main difference in this case is that we discretize our state space (an alternative method would be to apply a function approximator). I discretized by dividing each element of the state vector into two buckets, such that we can calculate the 'discrete' state s via an intuitive binary calculation.

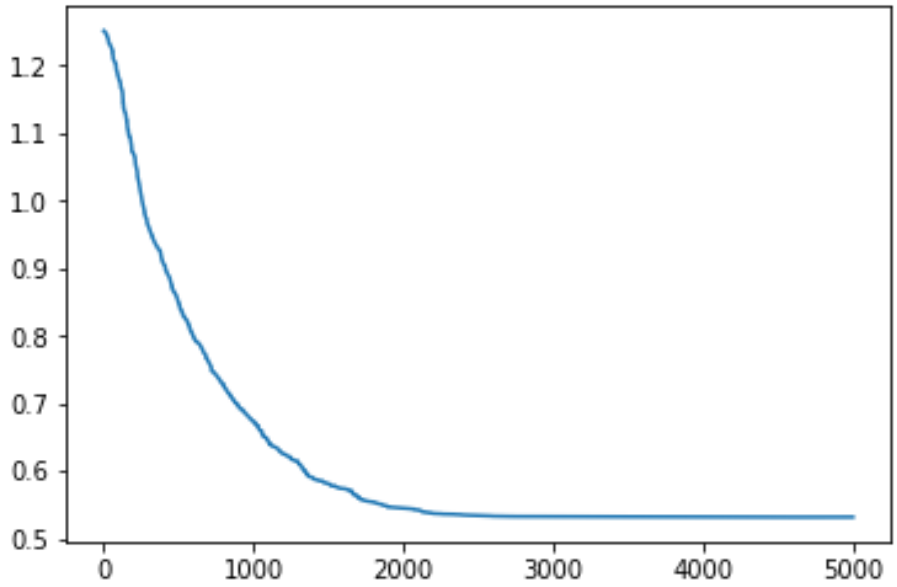
Results and Discussion

Value iteration on the Maze environment yields the following optimal policy (assuming no slipping): Starting at the top left state 0, the robot moves down, then right, then right, then up (to collect a flag), then down, then right, then down, then down (to collect another flag), then up, then up, then right, then up to the goal position. Based on our discount factor gamma, our optimal policy thus has the robot collecting two out of the three flags and then going to the goal state. If we decrease gamma to say 0.7, we thus weight getting to the goal faster than we did with a higher gamma, so we can see in this case that we may only collect 0 or 1 flags before getting to the goal. Likewise, if we increase gamma to say 0.99, we end up collecting all three flags before approaching the goal because it is less imperative that we reach the goal as quickly as with a smaller gamma.

Q-learning on the Maze environment:

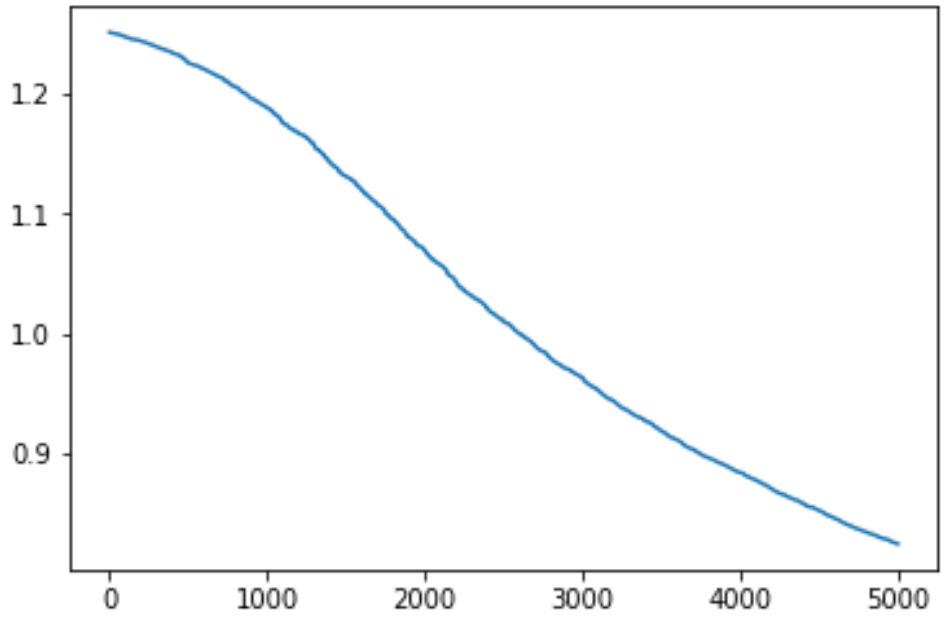
For all of the following, the x-axis corresponds to the number of episodes.

Qerr,t = RMSE(Qt,Q*) for gamma = 0.9, epsilon = 0.95, alpha = 0.05



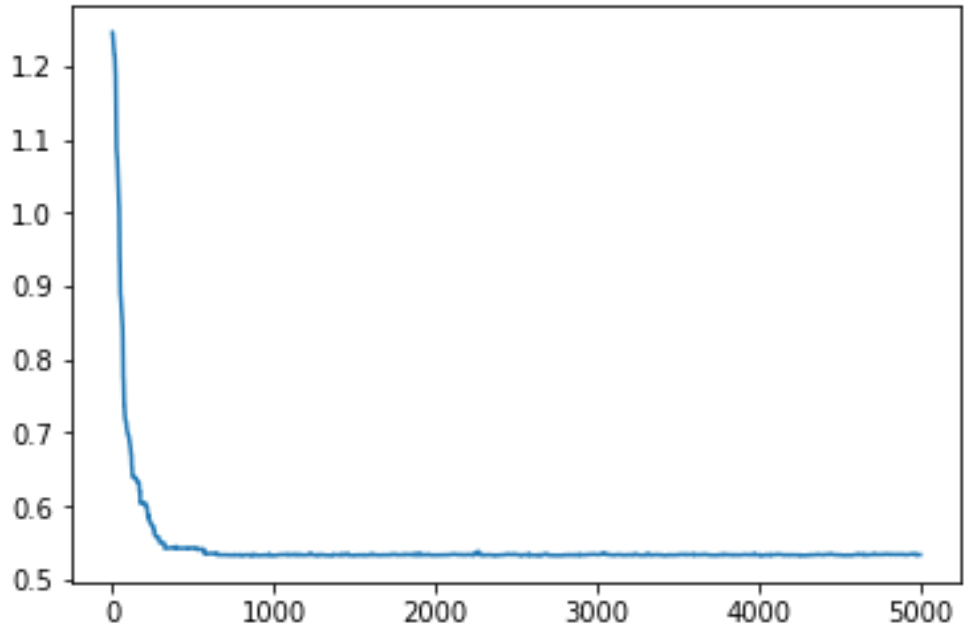
Here we can see that, after around 2000 iterations, Q converges to Q*. Note that in all of the below, gamma and epsilon remain fixed, as gamma is supposed to remain fixed for purposes of consistency in this project, and lower values of epsilon lead to very low probability of convergence within 5000 episodes (thus we gain more helpful information by randomly exploring more often than exploiting the current policy based on Q).

Qerr,t = RMSE(Qt,Q*) for gamma = 0.9, epsilon = 0.95, alpha = 0.005:



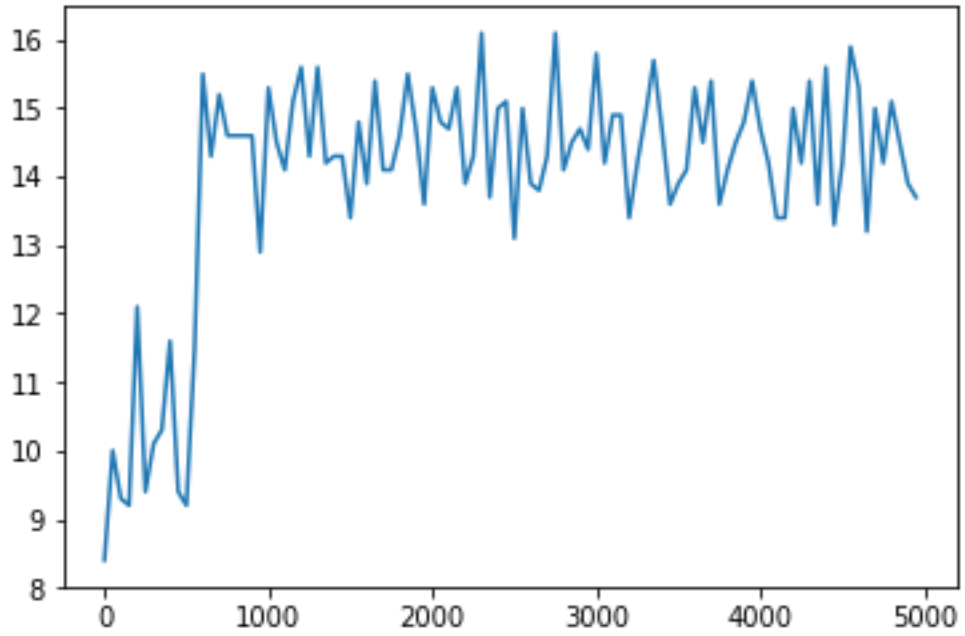
As we decrease alpha by a factor of 10, we see that it takes much longer to converge – in fact, we do not end up converging to Q* at all within 5000 iterations.

Qerr,t = RMSE(Qt,Q*) for gamma = 0.9, epsilon = 0.95, alpha = 0.5:



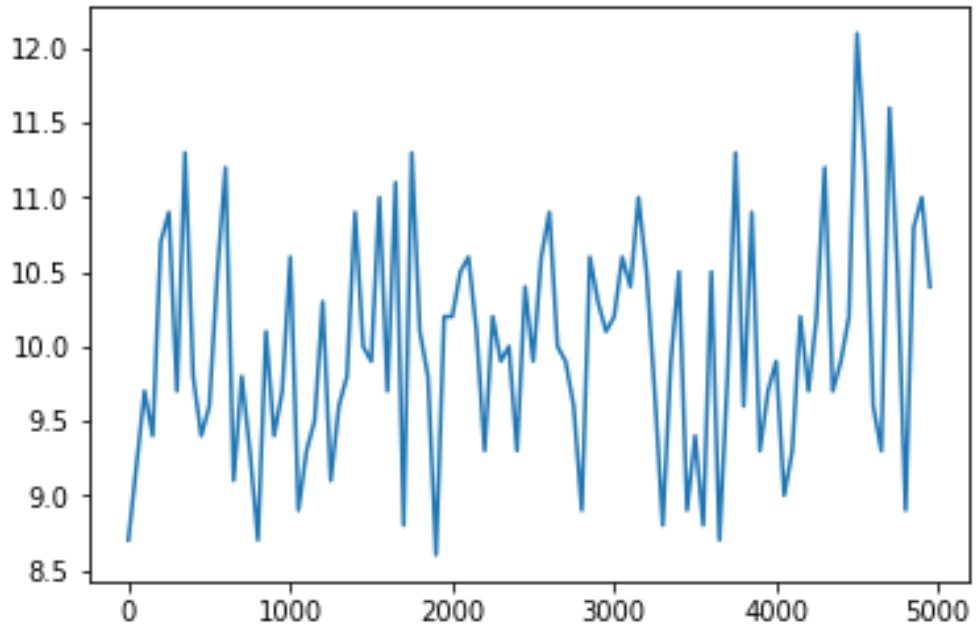
Similarly, as we increase alpha, we see that we converge significantly faster than either of the above alpha values, around 500 iterations.

Average steps evaluated at every 50 steps for gamma = 0.9, epsilon = 0.95, alpha = 0.05:



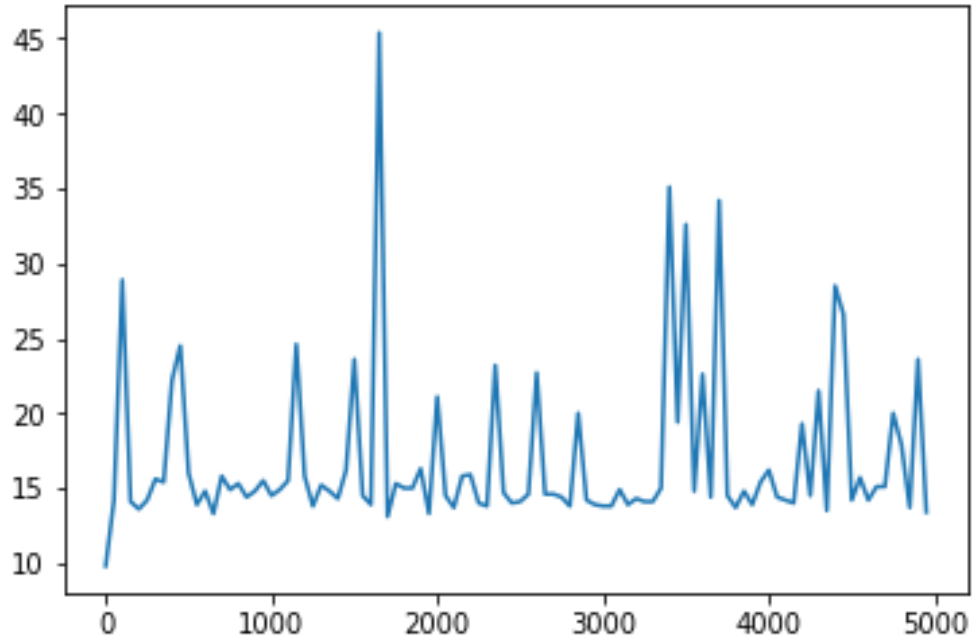
We can see that the average number of steps starts out lower, around 10 in our first rounds of iterations until convergence (which we can see from the corresponding Qerr,t graph above is) around 14 steps. The optimal policy calculated in part 1 has exactly 13 steps (including the start step), so once we incorporate slipping, it makes sense to see the converged policy oscillate around those values. Before converged, we see smaller numbers of average steps likely because the policy itself is not yet optimal, so we are likely moving to the goal without collecting the optimal number of flags first.

Average steps evaluated at every 50 steps for gamma = 0.9, epsilon = 0.95, alpha = 0.005:



We see similarly, comparing to the corresponding Qerr,t graph, since Q never converged, we are oscillating around the smaller number of average steps than is optimal. This graph is expectedly similar to the first 1000 episodes chunk of the above average steps graph with alpha = 0.05 prior to convergence.

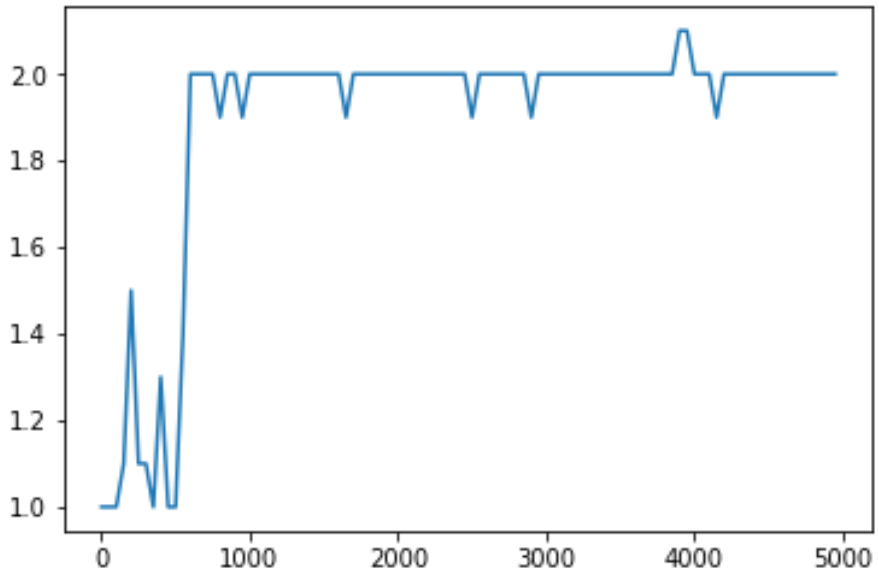
Average steps evaluated at every 50 steps for gamma = 0.9, epsilon = 0.95, alpha = 0.5:



Here, since we know from the corresponding Qerr,t graph that Q converges quite fast around 500 iterations, we see the initial increase in number of steps until convergence, similar to the average step graph with alpha = 0.05. However, we notice that, after Q

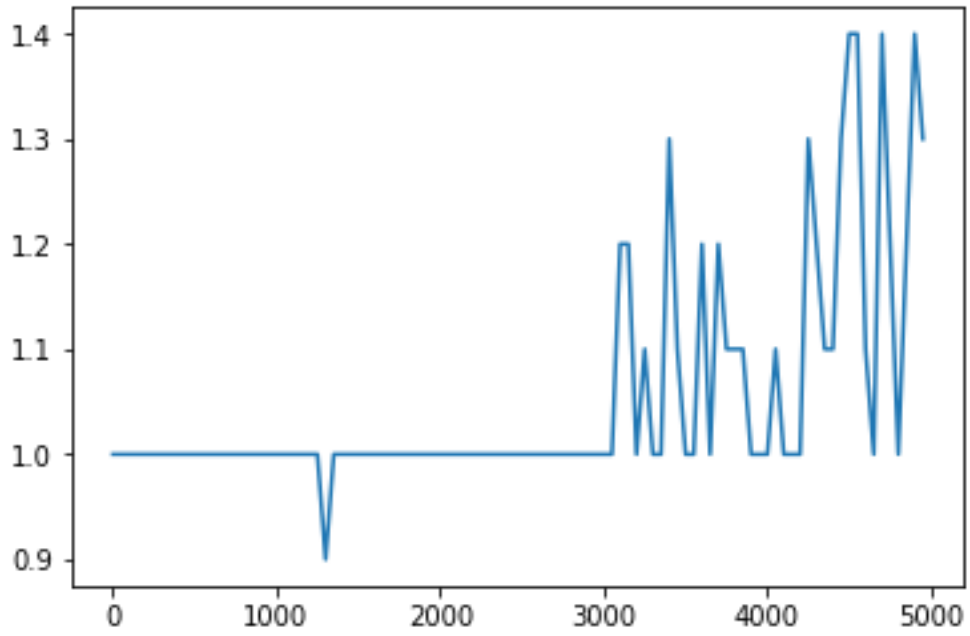
has converged, we actually see certain spikes which have much higher numbers of steps than those with lower alpha values. This is likely because the higher alpha value implies that less of the past information is used to update our policy, so it is possible that our policy has not optimally learned from past iterations which actions result in no progress (for example, taking action UP if you are at the top of the graph, which will result in the robot staying in its current position but still using up a time step).

Average reward evaluated at every 50 steps for gamma = 0.9, epsilon = 0.95, alpha = 0.05:



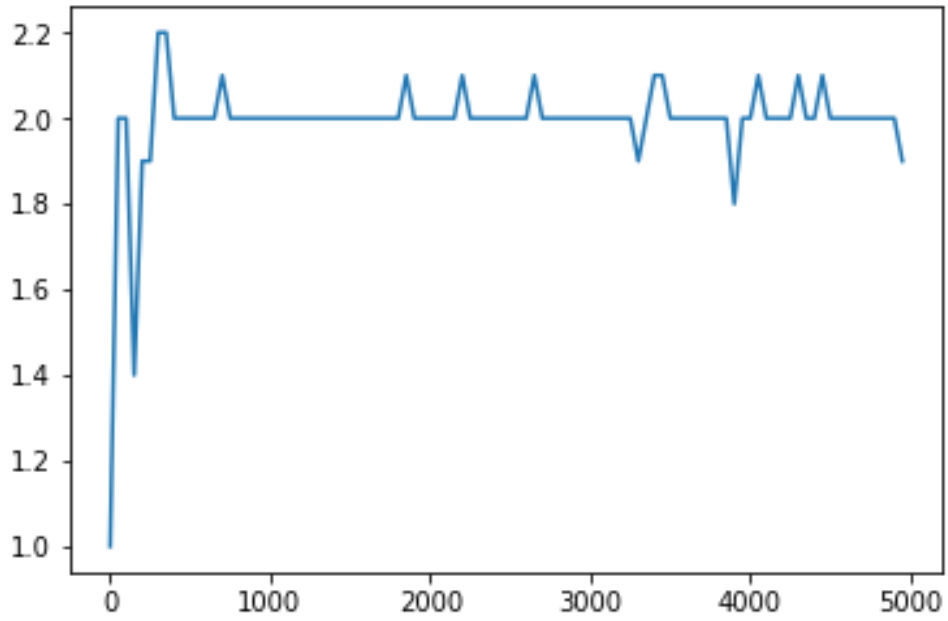
This follows a generally similar pattern to the corresponding average step graph, where we can see around 1000 iterations, we have converged to an optimal policy yielding a reward of an average of 2 (catching 2 flags). Prior to converging, we see lower rewards representing less flags being caught, which aligns with our corresponding average step graph where we see that the robot takes fewer steps to move to the goal state before catching all of the optimal number of flags. Likewise, we see occasional spikes above a reward of 2, where in those iterations, the robot actually collected all three flags before moving to the goal.

Average reward evaluated at every 50 steps for gamma = 0.9, epsilon = 0.95, alpha = 0.005:



Similarly, we see the rewards on average are lower than the rewards are for the optimal policy, since Q has not converged within 5000 steps given alpha of 0.005, and we can observe that this policy implies that the robot typically collects either 0,1, (or sometimes 2) flags, as opposed to the optimal 2 (or 3).

Average reward evaluated at every 50 steps for gamma = 0.9, epsilon = 0.95, alpha = 0.5:

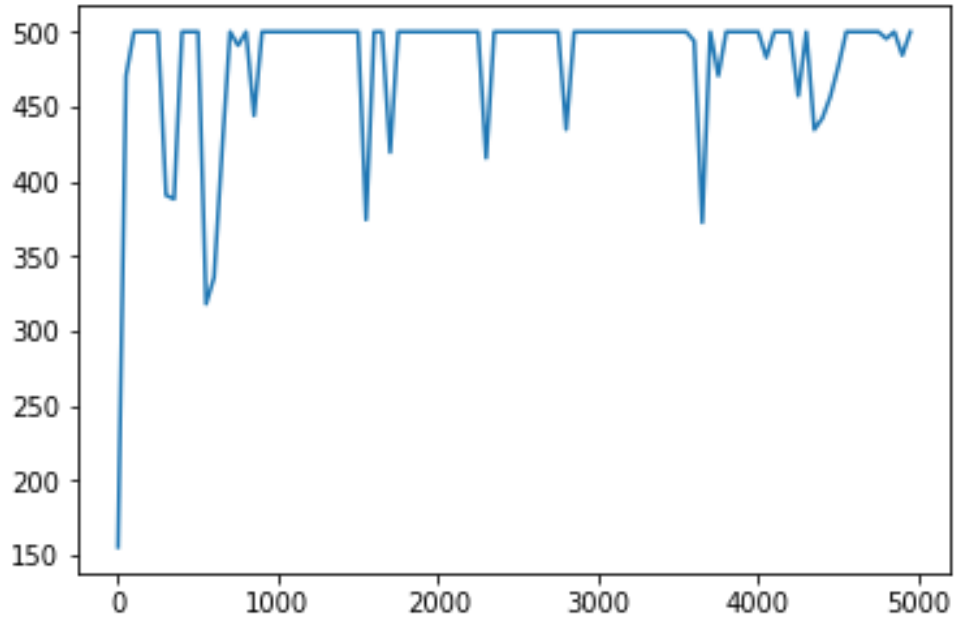


We can see the same type of pattern where our average reward increases from 1 up to 2+ from time 0 to time 500 until convergence, at which point the average reward oscillates around the optimal 2 value.

Continuous State Space Problems (Acrobot):

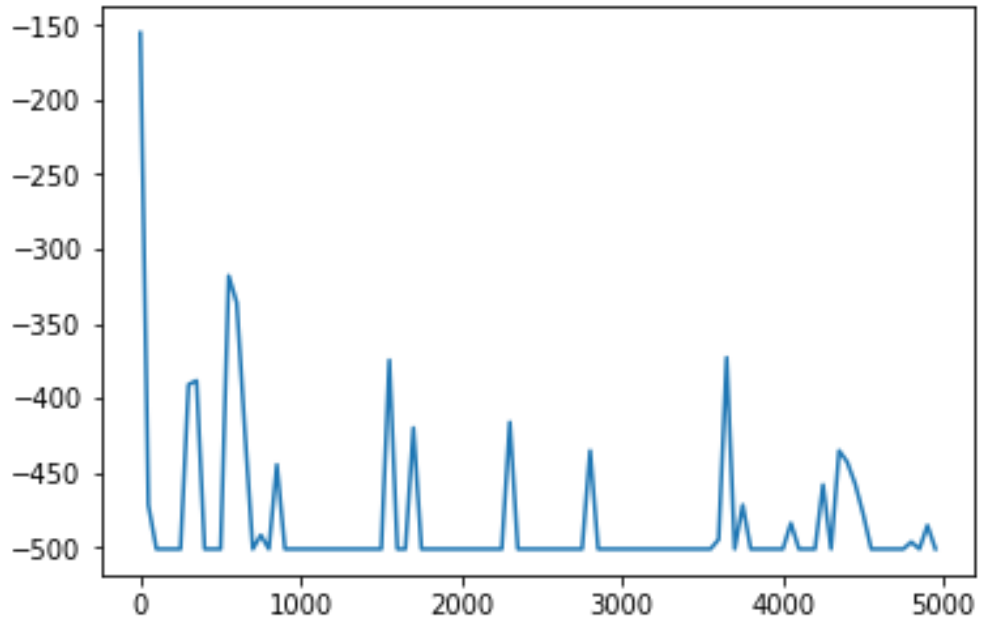
For REINFORCE, after changing the discount factor to 0.99 to help in faster convergence, I found that a learning rate value alpha of 0.00008 generally provided substantial convergent results for 100 iterations. We can test this by rendering the environment, as well as printing the timesteps taken to finish an episode (so that we can see whether the environment is actually converging or if it has just reached the max number of time steps) and the theta parameters to see how much, if at all, theta is changing in the last several iterations. Using the same gamma and the following alphas for Q-learning, as well as an epsilon of 0.95 to encourage exploration over exploitation, we see the following results:

Average steps evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.00008:



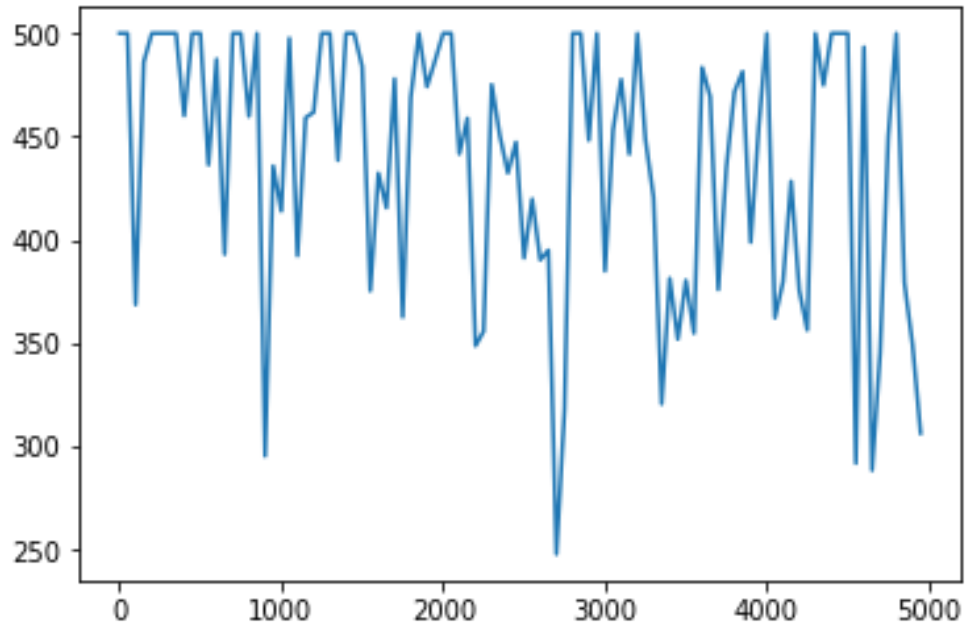
Given that 500 is the max number of steps before the environment will declare terminated, regardless of the actual state, we can see how the average number of steps oscillates between 150 and 500, increasing in the first ~200 iterations.

Average reward evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.00008:



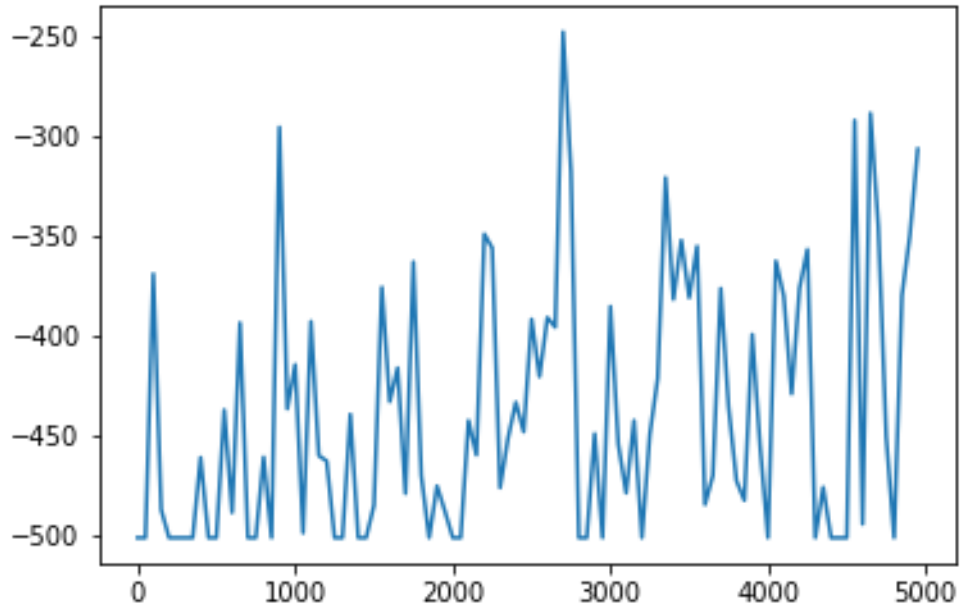
We see essentially the same average step function shown above, except mirrored across the x-axis. Since the reward for every time step is -1 unless the terminal state is reached (at which point the reward is 0), this behavior makes sense.

Average steps evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.0008:



As we increase alpha by a factor of 10, we see much more variation in our oscillations, since our increase in alpha corresponds to less of the past behavior being used to craft the optimized policy and more ‘random’ behavior.

Average reward evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.0008:

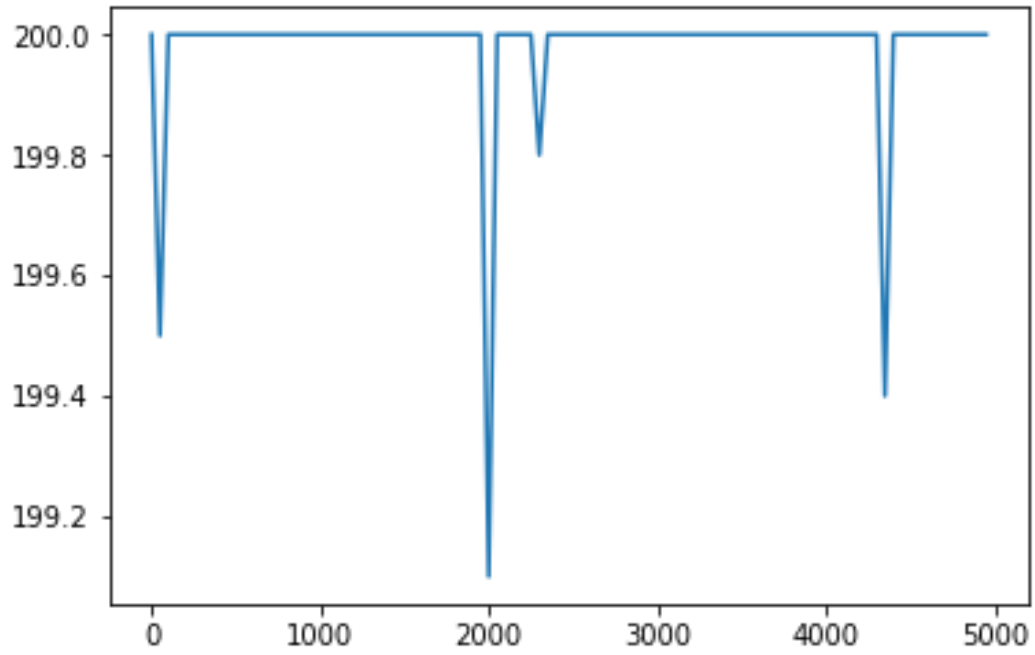


Just as before, our average reward is our average steps reflected across the x-axis.

(MountainCar):

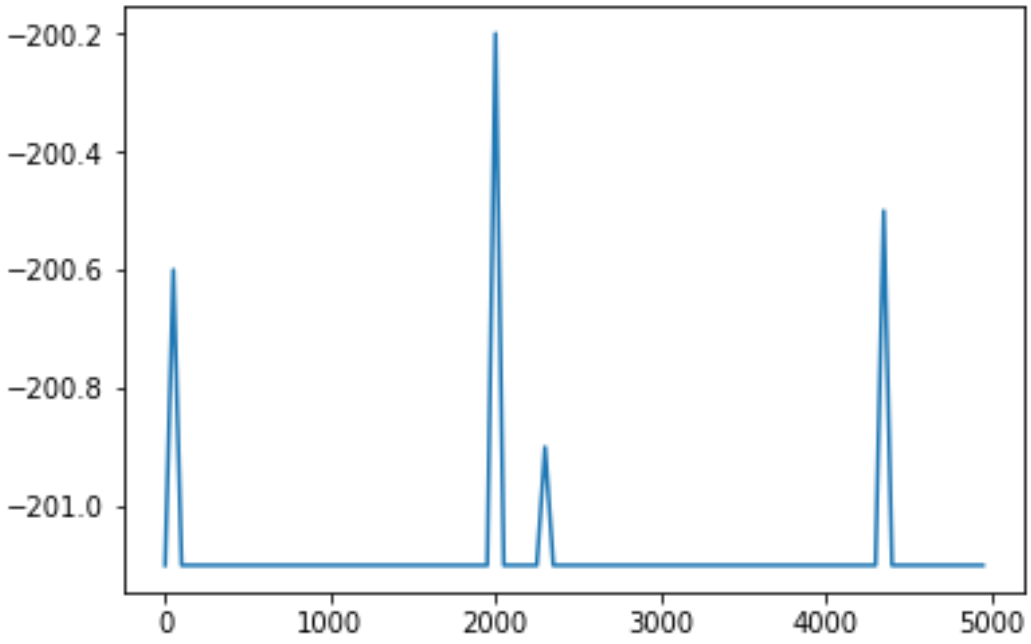
For REINFORCE, using the same discount factor as in Acrobot, I found that a learning rate value alpha of 0.0057 generally provides substantial convergent results for 100 iterations. Similarly, we can test this by rendering the environment, as well as printing the timesteps taken to finish an episode (so that we can see whether the environment is actually converging or if it has just reached the max number of time steps) and the theta parameters to see how much, if at all, theta is changing in the last several iterations. Using the following alpha for Q-learning we see the following results:

Average steps evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.001:



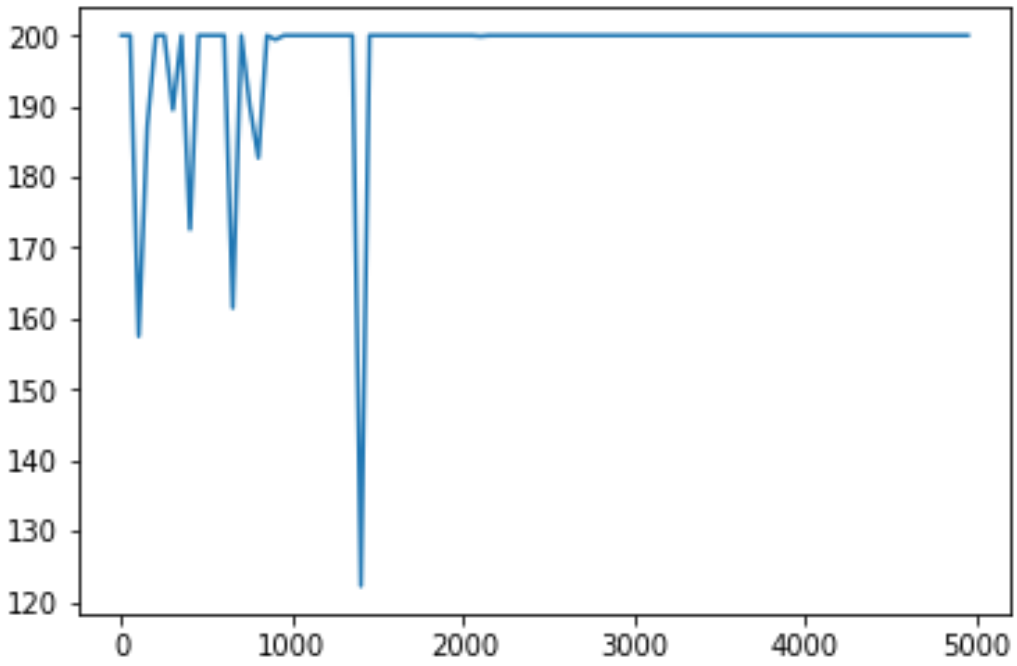
From the oscillations in this graph, we see that our policy generally converges (or reaches the max step limit) at 200 steps. Due to the fairly crude discretization which leads to 4 states in total, I am not surprised by the lack of variation in these results. With more time, I would have liked to experiment with stronger discretization strategies.

Average reward evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.001:



We see expectedly the average step function flipped across the x-axis.

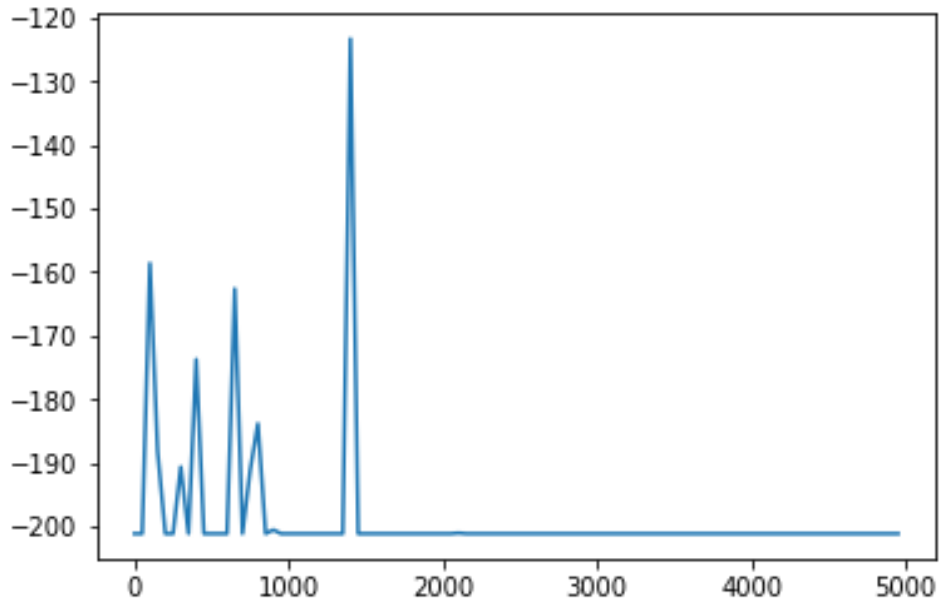
Average steps evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.0001:



As we decrease alpha by a factor of 10, we see more variation in our oscillations, which now decrease as low as ~120 steps, most likely due to the fact that we now include more of our past information in updating Q. Interestingly, we see a fairly strict

convergence after around 1500 iterations, where the average number of steps remains essentially constant.

Average reward evaluated at every 50 steps for gamma = 0.99, epsilon = 0.95, alpha = 0.0001:



We see expectedly the average step function flipped across the x-axis.