Project 4: Reinforcement Learning

Lars Kouwenhoven (lk545)

Introduction

This document describes the approach and results for Project 4 of Intelligent Autonomous Systems, focusing on reinforcement learning.

Background

This project consists of three parts, each of which will be discussed separately. All reinforcement learning problems discussed will be modeled as Markov Decision Processes, and are based on the cycle displayed in *Figure 1.* In short, we consider an agent in a certain environment. An agent can perform an action within the environment at time *t*, which leads to the agent being in state S_t. After each action, an agent may also be given a reward R_t.In each of the following problems, our goal is to develop a policy of actions that the agent should perform to maximize the received reward.

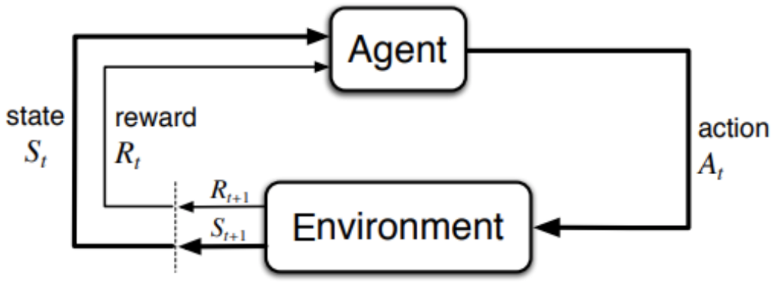


Figure 1

Part 1: Policy Iteration / Value Iteration

The first part of the project focuses on a *Maze* environment. The maze is displayed in *Figure 2*. 'S' denotes that starting position, 'G' the goal and 'F' locations of flags. The goal of the agent is to collect all flags and escape the maze. The agent receives a reward equivalent to the number of flags it has collected when reaching the goal state.

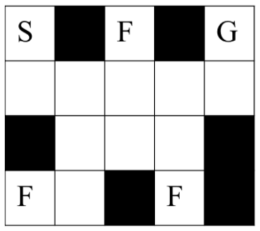


Figure 2

Even though there are only 14 open cells, we define 112 unique states. Namely, there are 8 different flag states, denoting which flags have been collected. Additionally, there are 4 possible actions: up, down, left and right. The goal is then to use either policy iteration or value iteration to find an optimal policy and the optimal Q-values for all state and action pairs. I have opted to use value iteration.

Technical Approach

First, we find the array of values *V* with shape 112 (the number of states) based on the value iteration algorithm. We initialize an array of zeros. Then, we define a value Δ, which we initialize as 0. Additionally, we define θ. From the description of the slides, this should be a small positive number. We found that setting θany lower than 1.6 prevented convergence. Therefore, we used this value.

Then, while $\Delta$is smaller than θ, we proceed as follows:

For each state *s*, we assign $v = V[s]$. Then, we update $V[s]$ as $max_a \sum_{s', r} p(s', r|s, a)[r + \gamma V(s')]$. Note that, according to the project description, we take the discount factor$\gamma$to be 0.9. Additionally, the environment has a slip factor of 0.1. This means that, with a 10% chance, the agent will perform an action which is not the actual action a. For the slip case, we define $p(s', r|s, a)$ to be 0.1, and 0.9 for the non-slip case. Then,

$\sum_{s', r} p(s', r|s, a)[r + \gamma V(s')]$ comes down to

$0.1 * reward + 0.9V(next\ state) + 0.9 * slip\ reward + 0.9V(slip\ state)$.

Then, when we have calculated this value for each action, we update $V[s]$ to be the highest value.
We update $\Delta$to be the highest of $\Delta$and the absolute difference between the old value of state *s (v)*, and the new value of state *s* (the updated $V[s]$). When Δ is lower than θ, we end.

With value iteration complete, we can then proceed to find the optimal Q values and optimal policy as follows:

$Q(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma V(s')]$, where V is the array we just filled during value iteration

$\pi(s) = argmax_a Q(s, a)$

Results

We display *V* as found through value iteration in *Figure 3*. Furthermore, *Figure 4* displays our optimal policy as a bar chart, where each bar corresponds to the optimal action taking in a certain state *s*.

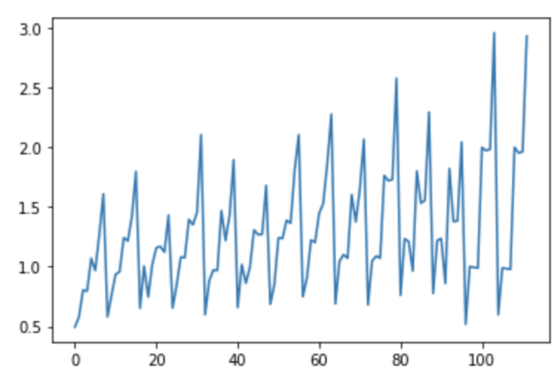
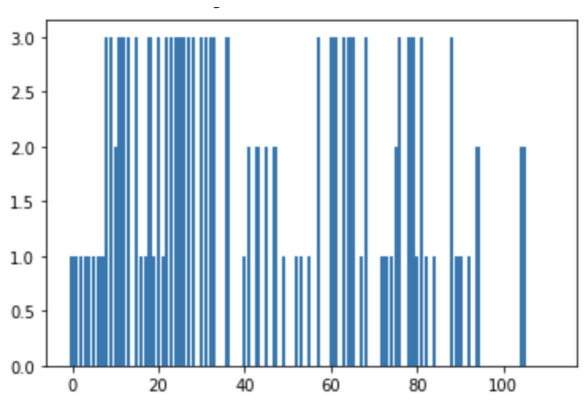



Figure 3 *Figure 4*

Then, we can describe the actions the robot will take in each timestep according to this policy.

Visual	**S**WFWG OOOOO WOOOW FOWFW	SWFWG **O**OOOO WOOOW FOWFW	SWFWG O**O**OOO WOOOW FOWFW	SWFWG O**O**OOO WOOOW FOWFW	SW**F**WG OOOOO WOOOW FOWFW	SWFWG OO**O**OO WOOOW FOWFW	SWFWG OOO**O**O WOOOW FOWFW	SWFWG OOOOO WOO**O**W FOWFW
Action		Down	Right	Right	Up	Down	Right	Down
State	0	8	24	56	49	57	73	81

Visual	SWFWG OOOOO WOOOW FOW**F**W	SWFWG OOOOO WOO**O**W FOWFW	SWFWG OOO**O**O WOOOW FOWFW	SWFWG OOOO**O** WOOOW FOWFW				
Action	Down	Up	Up	Right	Up			
State	93	85	77	109				

So, in 12 steps, the agent managed to navigate to the goal of the maze. Note that the agent just collected two flags, whereas three were available.

Part 2: Q Learning

In this part of the project, we implement tabular Q Learning for the maze environment. Q-learning is used to find the value of an action in a particular state, without requiring a model of the environment.

Technical approach

We start by defining a matrix *Q* of shape S x A (112 x 4). Here, we have tried two approaches: (1) fill it with random values between 0 and 3, except for the terminal state, where we insert a value of 0 (this is the range of values we expected (and have found in Q^*)) and (2) to fill it with zeros. Additionally, we define the learning rate parameter αbetween 0 and 1.

For 5000 iterations, we then initialize the environment at the starting state *s*. For each following step of the episode, we choose an action from *s* using a policy derived from *Q*, in this case ϵ-greedy. Based on action *a*, we then observe reward *r* and the new state *s'*. We then update *Q(s, a)* as

$$Q(s, a) \ = Q(s, a) - \alpha[r + \gamma max_a Q(s', a') - Q(s, a)]$$

We continue until *s* is a terminal state. Every 50 steps, we perform an evaluation using the evaluation script that was provided.

Results

The figures below show the results for different values of the hyperparameters. In *Figure5a-c,* we display the results using different values for gamma, the discount parameter. *Figure 5a* shows the average number of steps, *Figure 5b* shows the average reward, and *Figure 5c* shows the average RMSE when compared to Q^*

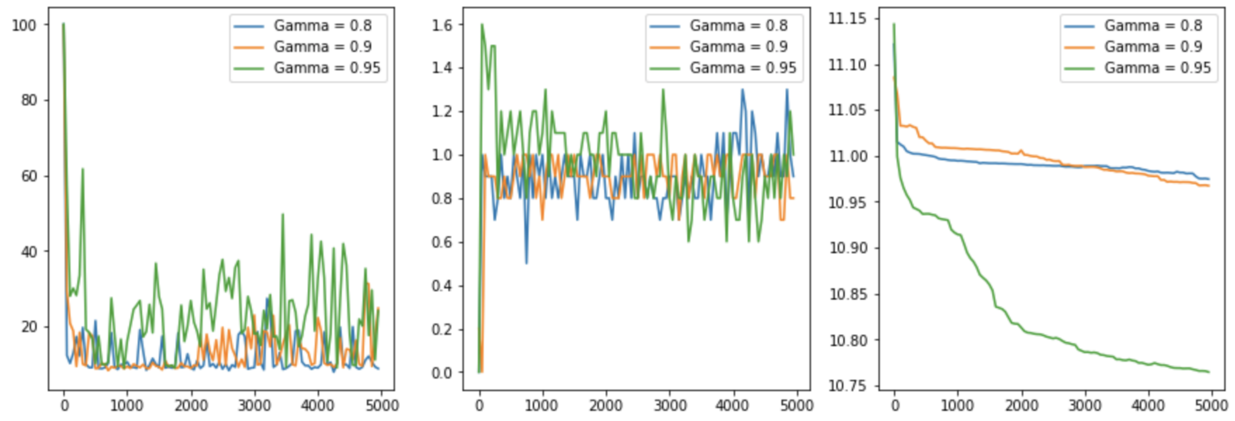


Figure 5

When it comes to the RMSE, a higher gamma clearly leads to a lower RMSE. There is relatively little difference between the values of gamma, except that a higher gamma value seems to lead to more variability in the reward and step values. Because of this fact, we will continue with a more stable value of 0.9.

Figure 6 shows the results of different values for the learning rate α, with a gamma value of 0.9

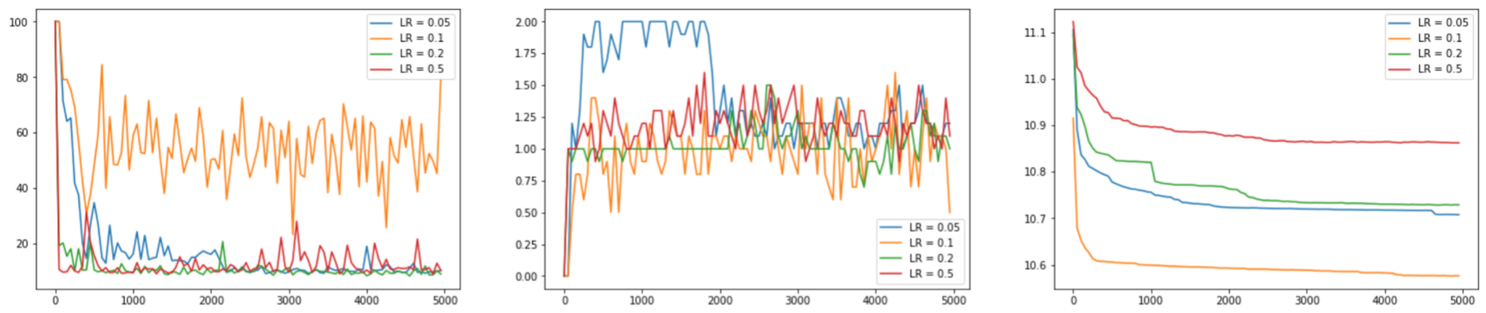


Figure 6

In the left subplot, we see that all but 0.1 reach the optimal number of steps. Conversely, with a learning rate of 0.1, the RMSE is lowest. The learning rate of 0.05 performs best when it comes to average rewards (middle subplot), and also reaches a low value for the number of steps. Therefore, we continue with a learning rate of 0.05

Finally, we look at any differences when varying ϵfor ϵ-greedy exploration. These results are shown in *Figure 7*.

Part 3a. Continuous Space Problems (REINFORCE)

In Part 3a (REINFORCE) and part 3b (Q-learning) we make use of two environments from the OpenAI Gym: Acrobot-v1 and MountainCar-v0. These environments differ from the Maze, as they have a continuous rather than discrete space. The state vector for Acrobot-v1 is [cos(theta1), sin(theta1), cos(theta2), sin(theta2), theta_dot1, theta_dot2]. For MountainCar-v0, the state vector is [position, velocity].

Technical Approach

For the REINFORCE portion, we use a nonlinear function approximator in the form of a Neural Network, to translate state space into the discrete action space. This *PolicyEstimator* consists of a linear layer with a ReLU activation, and another linear layer with a softmax layer to output the action probabilities. We use Adam as an optimizer.

This small neural network is our initial policy parametrization $\pi(a|s, \theta)$, where $\theta$are the weights of the network. Additionally, we use a step size α and a discount factor $\gamma$of 0.9. Then, for each each iteration, we generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$according to π. For each step t we define:

$$G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\theta = \theta + \alpha G \nabla ln\pi(A_t|S_t, \theta)$$

Both steps are achieved by first collecting tensors of states, rewards and actions for each timestep t. We calculate G using the rewards. Then we calculate $ln\pi(A_t|S_t, \theta)$by calling *forward* on the Neural Network with the found state tensor, and applying a log transformation, which is then multiplied by G. We define the loss as the negative sum of the log probabilities of the selected actions according to the policy π, and call *backward* on this loss to update the parameters of the neural network. This process is repeated for each iteration.

Additionally, we make use of a simple baseline window, which subtracts a rolling average from the rewards vector. Finally, in the case of the Mountain Car, we 'bunch' actions together, by applying the same action for five time steps, and only then finding a new optimal action.

Results

Figure 7 displays the results of the Acrobot case with gamma values of 0.9 and 0.99 respectively. The left subplot shows the average reward per 10 iterations (thus 250 iterations were run in total), and the right subplot shows the number of steps per iteration. We see that both gammas approach the same values for the number of steps and average reward, although a gamma of 0.9 reaches this level significantly sooner. We will thus continue with that value.

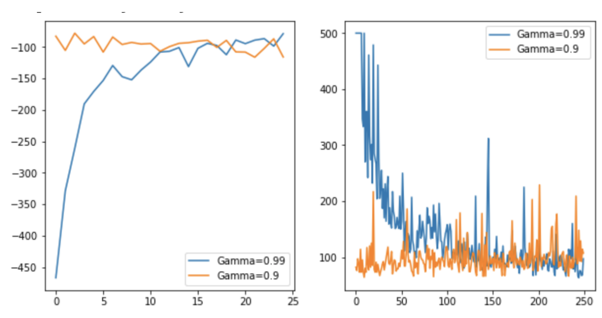


Figure 7

Next, we look at the difference between learning rates in *Figure 8*

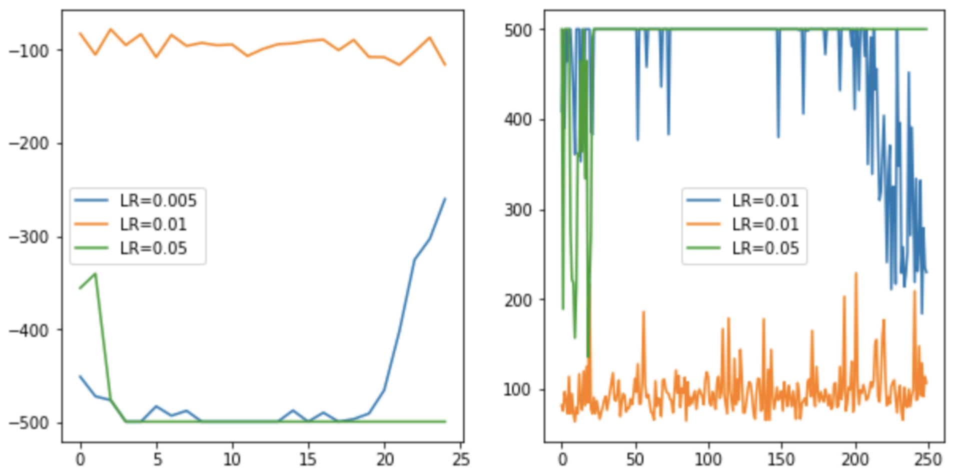
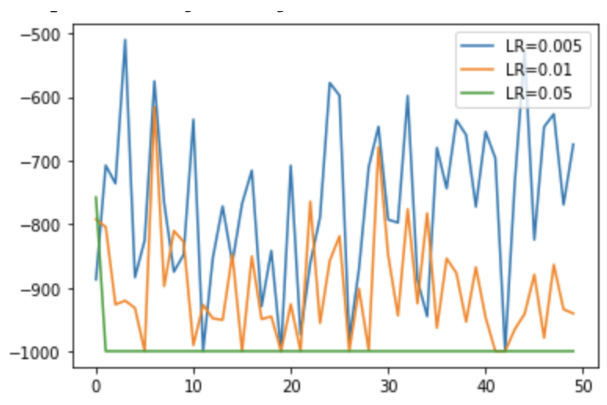


Figure 8

We see that a learning rate of 0.01 consistently performs well. A learning rate of 0.05 on the other hand, starts off relatively well, but then leads to average rewards of -500, i.e. never reaches the goal state. A learning rate of 0.005 starts off unsuccessfully, but increases after some 200 iterations. Still, we chose a learning rate of 0.01. It is also important to note that there seems to be a high variability in the results, even when keeping a learning rate stable. This makes evaluation a little harder.



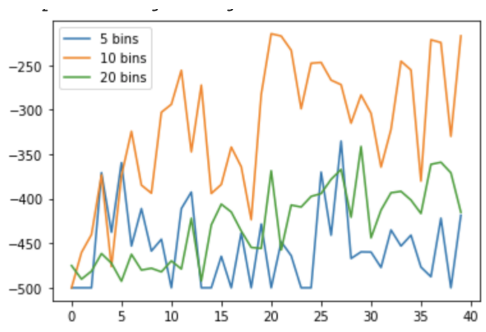
We now move on to the results of the MountainCar environment. When we do not use a baseline, the algorithm fails to converge for this scenario at all. We therefore use a baseline with a window of 20 timesteps. *Figure 9* displays these results.

Two things of interest here: a learning rate of 0.05 again leads to structural unsuccessful completion of the scenario. However, even for learning rates of 0.005 and 0.01, there is little improvement visible, and the average reward values remain somewhat constant. This indicates that the parameters are not quite successfully updated. However, I was unable to figure out how to improve this behaviour.

Part 3b. Continuous Space Problems (Q Learning)

Technical approach

We repeat the Q-learning algorithm from part 2 for the continuous space problems, except for the fact that we discretize the state space into 'bins' for each of the entries in the state vector. With n bins, this leads to a table of size n^6 for the Acrobot scenario and n^2 for the MountainCar environment. Instead of using an array or tables, which would take up a lot of memory, we use a dictionary which maps discrete states to action probabilities.



Results

For the Acrobot, we first look at the optimal number of bins to use. These results are displayed in *Figure 10,* which includes the average reward per 250 iterations. We choose to continue using 10 bins. *Figure 11* shows the reward and number of steps for Acrobot, whereas *Figure 12* shows the same for MountainCar. We see that the MountainCar scenario converges faster, and more successfully, than Acrobot. This is likely because of the fact that Acrobot has more parameters in its state vector, and thus the Q-table will be larger, and more iterations are needed to completely fill it.

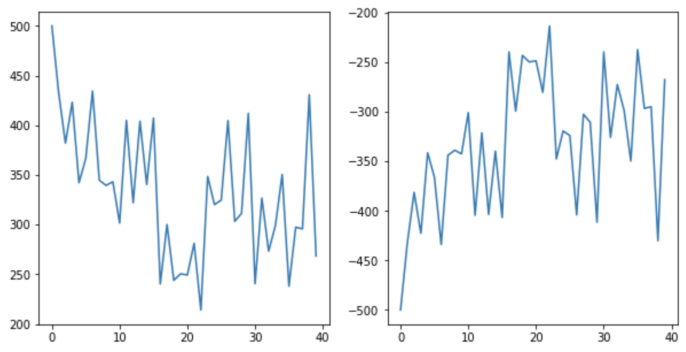


Figure 11

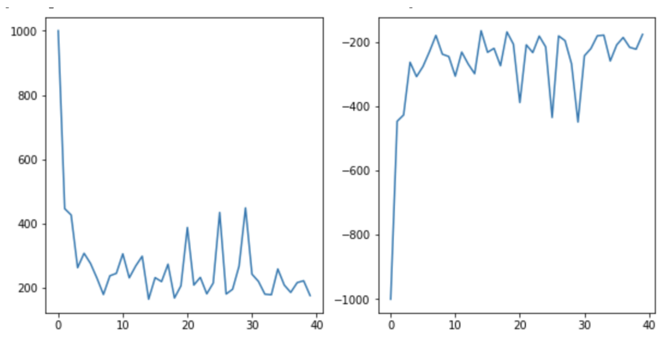


Figure 12