Q-Learning

-by Pavan Chowdary Cherukuri pc3088

Introduction:

The goal of this project is to learn a policy for an inverted pendulum model to make it do a swing-up motion. This project is aimed to provide knowledge on how Q-learning works, its limitations and the advantages of using it. Coming to the problem that is being solved, the task is to make a pendulum reach an inverted position using Q-learning. The problem is visualized in figure 1.

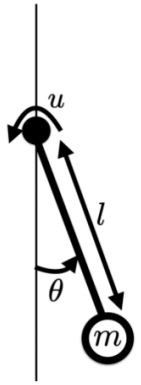

Figure 1: Pendulum problem

The inverted pendulum has a limit on the maximum torque it can apply, therefore it is necessary for the pendulum to do a few back and forth motions to be able to reach the inverted position (θ= pi) from the standing still non-inverted position (θ=0). In the problem, it was mentioned that the state vector of this system is given by x = [θ, w]T. Also the system is discretized, hence x_n is described as state at time 't', where t = n*Δt, where 'n' is the number of discretized steps and Δt is the discretization time or the length of each time interval.

In this project Q-learning with a table was implemented. For this get_next_state(x,u) was used that returns x_{n+1} given (x_n,u_n). Here, the control 'u' is assumed to only have three values. θ can have any value between [0,2*pi) and 'w' can take any value between [-6,6]. To build the Q-table, the states are needed to be discretized, hence '50' discretized states for both θ and w were used.

Qsn 1: The get_cost(x,u), function which takes the states x (consisting θ and w) and u were used as inputs, which returns the cost value based on the formula:

$g(x_i,u_i) = (\theta_i - \Pi)^2 + 0.01 * (d\theta_i/dt)^2 + 0.001 * u_i^2$. The cost is found since the main aim of this problem is to decrease the discounted cost function (infinite horizon) $\Sigma\, \alpha^i * g(x_i, u_i)$, where $\alpha = 0.99$. This cost mostly penalizes deviations from the inverted position but also encourages small velocities and control.

Qsn 2: The Q table stores the action-value function Q(x_t,u_t), where Q represents the cost of doing u_t at state x_n and behaving optimally. The dimension of the Q-table in general would be a vector of states*controls. Generally the states would be the row vectors and the controls would be columns (1st and 2nd dimensions). Since, here there are two dimensions for states it would be a 3 dimensional vector, 50*50*3. It is because the states θ and w were discretized in 50 steps as mentioned above and the control u will be of 3 since it is discretized to 3 values.

Qsn 3: If a Q table is already given, the optimal policy and the optimal value can be calculated from the Q table. The optimal policy for each state (here in this case for each combination of θ and w) is the control (which is our input) that corresponds to the index of the index of the smallest value in the Q-table To be more clear since Q table consists values in 50*50*3 dimension, for each combination of θ and w, which means for a specific element in the 50*50 matrix, the index for the control that has the smallest value is the optimal policy. Similarly, for the optimal value, it is that minimum value of those values, for each combination of θ and w.
Each combination has a minimum value, which implies there would be 50*50 optimal values that correspond to each state.
As discussed above, a function called get_policy_and_value_function(q_table), was written, which takes a q_table as an input and gives, which returns the optimal values and optimal policies.

Qsn 4: A q_learning(q_table) function was written that takes q_table as the input and returns a learned Q table of the similar size. The function was written in such a way that it also stores the cost for tracking the progress. This can be seen in the code. The algorithm is as follows:

*The learning rate **γ**=0.1 and **ϵ**=0.1 were initialized*
Q(x_t,u_t) was initialized (zeros) for all states and actions
For each episode:
An initial state was x_0 (zeros) was chosen
Loop for each step of an episode
*An action was chosen using **ϵ**-greedy policy*
Observe x_{t+1} compute $g(x_t, \mu(x_t))$
*Update Q(x_t,u_t) to Q(x_t,u_t)+**γ** *δ_t,*
where $\delta_t = g(x_t, u_t) + \alpha * min\, Q(x_{t+1}, u_t) - Q(x_t, u_t)$

Qsn 5: From figure 2, it can be clearly seen that it takes around 4500 episodes for Q-learning to learn how to invert the pendulum. Here as mentioned, controls u ∈ {-4,0,4} were used and as mentioned above, a learning rate of 0.1 was used.

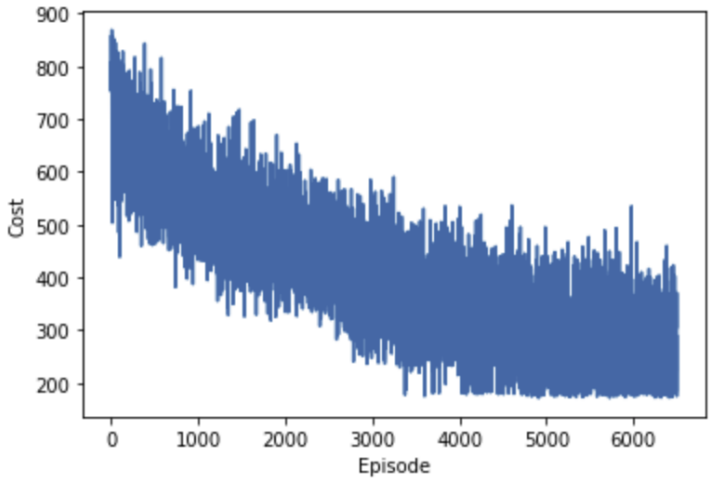

Figure 2: Cost vs episodes plot

Qsn 6: The problem converges after one back and forth revolution to reach the inverted position from rest. It can be understood from the figure 3 shown below and also from the animation.

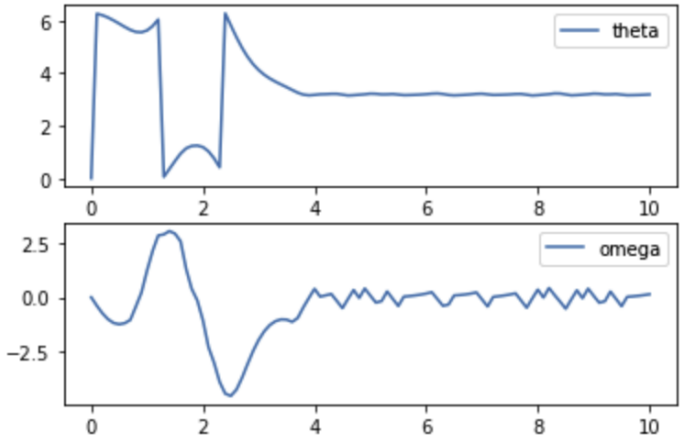

Figure 3: time evolution plot of theta and omega

Qsn 7:

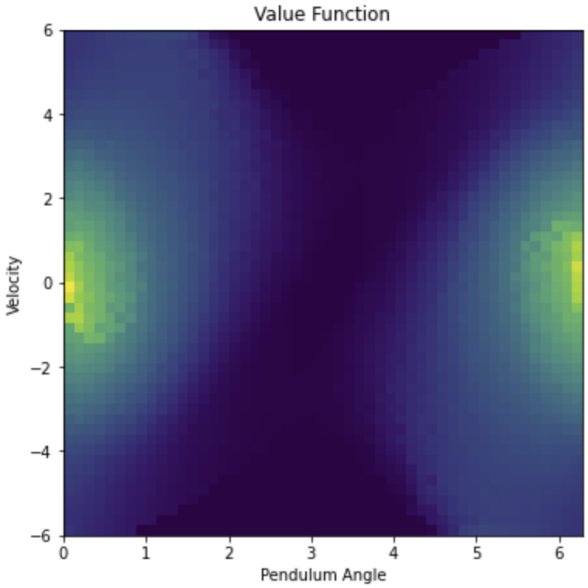

Figure 4: value function plot

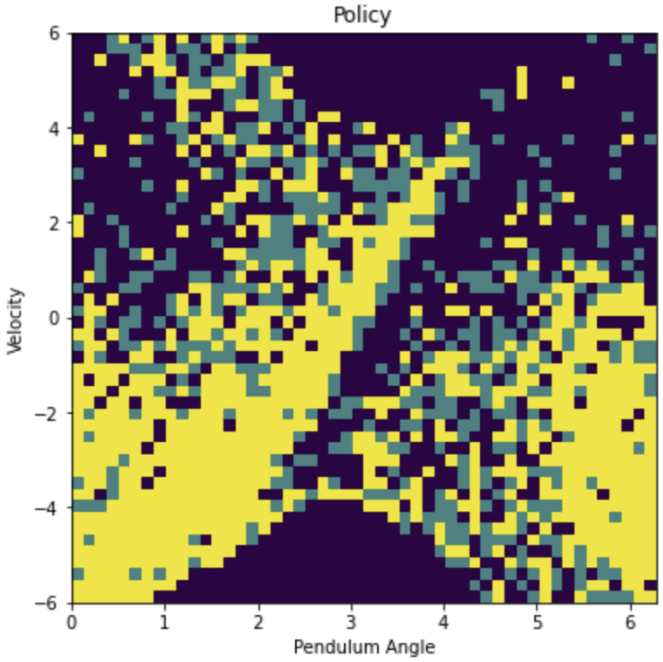

Figure 5: policy function plot

Qsn 8: For this question, the controls u ∈ {-5,0,5} were used with all the remaining parameters being the same. Since the control has been increased the pendulum reaches the desired state in the first oscillation itself. It is due to the increase in the control values, which caused the pendulum to move further in the first oscillation itself. The number of episodes taken to converge has also decreased and is around 3500, though not visible properly. The value and policy was almost the same as the previous case.

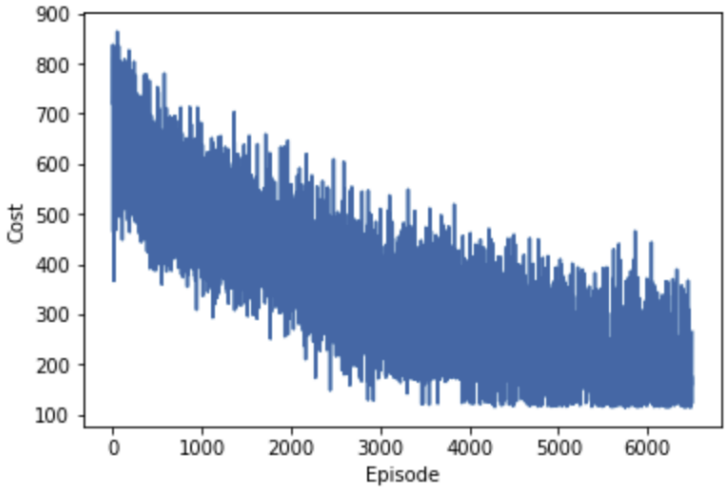

Figure 6: Cost vs episodes plot

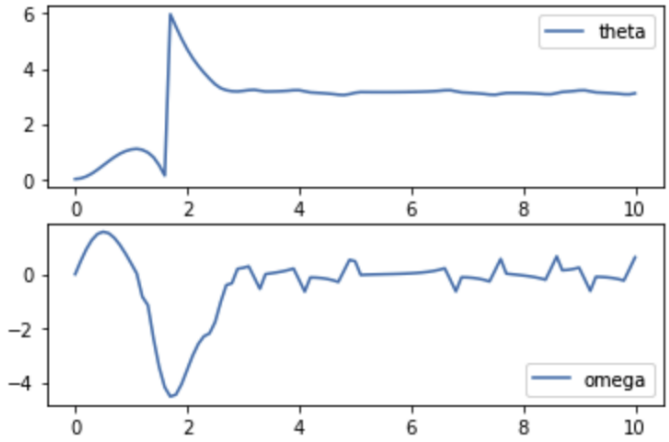

Figure 7: time evolution plot of theta and omega

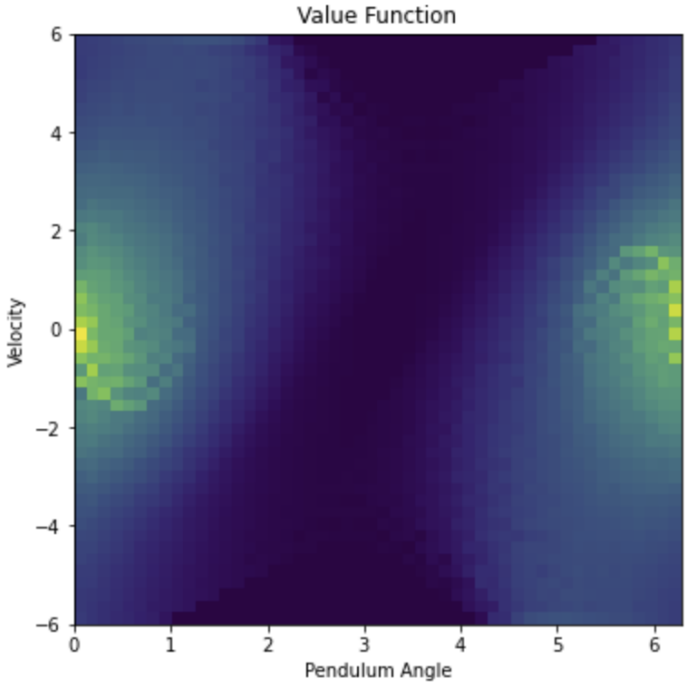

Figure 8: value function plot

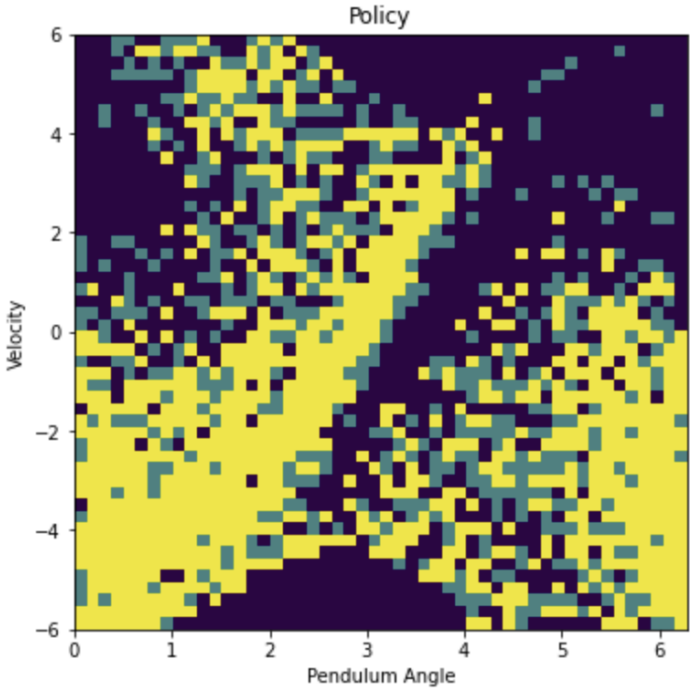

Figure 9: policy function plot

It can be observed from the figures that there is not much of a difference between both these cases.

Qsn 9: The learning rate is affected when the values γ *and* ϵ were changed. Since these parameters can be changed to configure the Q learning, these parameters need to be configured properly with care. Through e-greedy policy, one can deduce that the control is chosen from arg min $Q(x_t, u)$ with a probability of 1-e and a random action with a probability of e.

Changing the e value, changes this probability, which means the increase in e implies that the probability to choose a random value increases. This means that the state space is explored more. This also might negatively affect the system since the system is always choosing random actions with more probability which reduces the accuracy of improvement of the Q table and decreases the rate of convergence. If the e value decreases more, the system doesn't explore more and can converge to a local minima which might not be the optimal solution. The upside of this is, since the system is updating with optimal values, there would be an increase in convergence rate. Changing the e values as required would be best, to start initially with a higher value, which would be great because it would favor the exploration at the start and eventually the e can be decreased to converge the Q table optimally.

Coming to the learning rate, choosing the learning rate to a higher value can increase the rate of convergence since the error is increased. This also has downsides like causing the model to explode, due to divergence caused by high learning rate, also since the error is more and convergence is faster, the Q learning model explores less. If the learning rate is too low, the convergence might be too slow and sometimes there might not be a convergence also. So, these hyperparameters should be configured accordingly keeping in mind the pros and cons.