Documentation

The entire simulation basically uses three user-defined functions as explained below.

The main function is named "*Q_Learning()*". Its functionality is to learn the Q-table for balancing the pole on a moving cart. The output is the learnt Q-table.

Code Lines 1-12.

The first step in any code is to initialize all the constants. These constants are mass of the cart (M = 0.1 kg), mass of the pole (m = 0.05 kg), acceleration due to gravity (g = 9.81 m/s^2), length of the pole (l = 1.2192 m), and maximum external force applied (Force = 0.007 N). To make this simulation close to the the real setup the constants were defined as they are in the real setup. Now, for the simulation the update time interval was set at 0.1 second.

Now, there are a few constants which need to be defined for the Q-learning algorithm. These constants are discount factor for future reinforcement (y = 1 as the system is deterministic), exploration rate for epsilon-greedy policy method (epsilon = 0.5 is the initial value and it varies with the number of trials), learning rate parameter (initialized to zero for all possible state-action pairs and varies every time a particular state-action pair is accessed), and the Q-table (initialized to zero for all possible state-action pairs). It should be noted that the system has only two possible actions, that is, to move left or to move right.

Code Lines 14-18.

Now, the number of possible states is variable. In the simulation, number of states per measurement have been defined. There are four measurements (state) for the system. The first state is the position of the cart. This measurement has been divided into three states as the only thing that need to be made sure for this measurement is that it does not go out of bounds ($x \in [-0.15,0.15]$ m). The second state is the velocity of the cart. This measurement has also been divided into three states as, like the first measurement, the only thing that need to be made sure for this measurement is that it does not go out of bounds ($v \in [-0.15,0.15]$ m/s). The third state is the angular position of the pole. This measurement has been divided into six states as this is the most important measurement to define if the system is stable or not. The states make sure that the angle does not go out of bounds ($\Theta \in [-12,12]$ degree) and also checks if it is in the stability range ($\Theta \in [-4,4]$ degree). The fourth state is the angular velocity of the pole. This measurement has been divided into five states as, similar to the third measurement, this is an important measurement to define if the system is stable or not. The states make sure that the angular velocity does not go out of bounds ($\omega \in [-50,50]$ degree/s) and also checks if it is in the stability range ($\omega \in [-1,1]$ degree/s).

A few more constants are defined that relate to the tracking of the system. These constants are the number of success (keeps track of the number of iterations for which the system was in the stable state), the number of trials (keeps track of the number of times the system need to be reset), and best parameter (keeps track of the maximum number of successful iterations).

Code Lines 21-30.

Now, that all the constants have been initialized we start the system. First the cart is reset, that is, the current action is set to 0 (no action), the four states are initialized with some white gaussian noise to make the simulation behave close to the real setup. And then the state is computed. This is done using the "*getState()*" function which is explained in more detail later in this document.

To visualize the system, figure is initialized and the axis limits are set so that the system is easily observable.

Code Line 35.

Now, a "*while*" condition is started. This makes sure that if the system stays in stable state for more than 1000 iterations (100 seconds) simultaneously then the system is assumed to be trained. This value is kept high to ensure that the simulation can cross the reality gap.

Inside the "*while*" loop, the pre-state is initialized to the current-state, and the pre-action is initialized to the current-action. Then, the current-state and respective reinforcement value is achieved from the "*getState()*" function. Now, the Q-value is updated only if an action was performed (pre-action ≠ 0). Say an action was performed, then if the current state is -1 (an invalid state) the predicted value is set as 0. However, in case of current state ≠ -1, current-action is achieved using the "*epsilonGreedy()*" function. Then, the predicted value is set as the Q-value for the current_state-current_action pair. As mentioned earlier, the learning rate parameter is incremented by 1 for the current_state-current_action pair. Finally, the learning rate used is the inverse of this value. Now, the Q-value of pre_state-pre_action pair is updated using the Bellman Equation. As mentioned earlier, the Q-value is updated only when an action was performed. But, if there is no pre-action then the current-action is set using the "*epsilonGreedy()*" function.

Now, based on the current-action, force value is set as the positive (movement towards right) or negative (movement towards left) of the maximum external force possible.

Code Line 58.

Now, to update the cart pole-system state, the equation of motion derived using lagrange equation is used. It gives the linear acceleration of the cart and the angular acceleration of the pole. Using Newton's equation of motion, the velocity and position of the cart, and the angular velocity and angular position of the pole are derived.

The new state (measurements) achieved are drawn in the previously initialized figure. First a line is defined for the pole from the cart center to the pole end. Then a rectangle is defined to represent the cart. A arrow has also been shown in the figure to show the direction of movement of the cart, or in other words, to show the direction in which the external force is applied. Finally these objects are deleted after a pause of 0.1 seconds so that the next state represented in the figure does not overlap the previous one.

Code Line 83.

The "*getState()"* function is used again to get the state and reinforcement value for the new updated measurements. If this is an invalid state, then Q-value of the pre_state-pre_action

pair is updated with the negative reinforcement and the system is reset. Also, the number of trials is incremented, the value of epsilon is updated, the value of best is updated (if best < success), and finally success parameter is set as 0. If the state is not invalid and the reinforcement to be used is +1, then success parameter is incremented as a reinforcement of +1 represents a stable state.

GetState Function.

Now, it can be observed that a "*getState()*" function has been used extensively in the main code. The input to this function are the four measurements and the output of the function is the state and the reinforcement value. First the function makes sure that the four measurements lie in their respective bounds. If not, the state is set as -1 and the reinforcement is set as -1.

If the measurements are in their respective bounds, then the reinforcement value is initialized to -0.5. The state is defined first with respect to the position of the cart (state = [1,3]). It is followed by the increment of state with respect to the velocity of the cart by a value of 3 per next state (state increment = {0,3,6}). Then the increment of state is done with respect to the angular position of the pole by a value of 9 per next state (state increment = {0,9,18,27,36,45}). If the angular position of the pole is in the stable state, then the reinforcement value is incremented by 0.75. Finally the increment of state is done with respect to the angular velocity of the pole by a value of 54 per next state (state increment = {0,54,108,162,216}). If the angular velocity of the pole is in the stable state, then the reinforcement value is incremented by 0.75. If both the stable conditions are met then the reinforcement value become +1.

EpsilonGreedy Function.

Another function used in the main program is the "*epsilonGreedy()*" function. This function is used to decide on the action to be taken using the epsilon-Greedy policy. The function taken the Q-values for the current state and the value of epsilon as input, and given the action location as output. This method of action selection is incorporated to ensure sufficient exploration and exploitation.

The function starts with initializing Q_max to least value possible, that is, -infinity. Then for all the possible action locations, the maximum Q-value is found. If there are more than 1 location with the same Q-max value, then a tie Counter is incremented and the tie Index is stored. Tie breaker is crucial for random selection when choosing between more than one optima. To break the tie, a random index is chosen from the tie Indices stored. Now, the probability of choosing all the possible actions is set as $\epsilon/2$ and the probability of the action with the maximum Q-value or the one chosen from tie break is set as $1-\epsilon/2$. Then, the action index is chosen randomly by doing a weighted probability distribution of all the possible actions.