Assignment -1-Tabular Reinforcement Learning (2022)- Kenneth Fargose-s3281353

Abstract

The goal of this task is to familiarize us with basic reinforcement learning ideas. It gives us a good understanding of tabular and value-based reinforcement learning, which are the foundations of RL. We attempted to apply Dynamic Programming in this assignment, which is a strategy that bridges the gap between planning and reinforcement learning. We've also worked on model-free RL when we don't have access to a model. We can only permanently execute operations from one state and continue from the next state in this scenario. We work on Q-learning, SARSA, and Monte Carlo challenges in model-free RL.

1. Dynamic Programming

1.

Dynamic programming can help identify optimal solutions to planning challenges faced in the business, with the crucial premise that the details of the environment are known, aside from being an useful beginning point for mastering reinforcement learning. DP is a fantastic place to start learning about RL algorithms that can solve increasingly difficult situations.

The goal of Q-Value Iteration is to find a policy that maximizes the value function.

In Dynamic Programming we are told to update 3 sets of functions in the program.

The first init()is correct to start with, there are not alteration or additions required in the function.

In the second function we are asked to make changes to select-action() function to implement the greedy policy: π(s) = arg max Q(s, a) for this we make use of the already defined function agrmax.π(s) is the optimal policy.It makes use of the arg max function in order to select the policy with the highest possible value.

The update() function is where we properly try to implement the Q-value iteration algorithm since it makes use of the update() function.

We make use of the following equation in the update function

Q(s, a) $\leftarrow \sum[p(s'|s,a) * (r(s,a,s') + \gamma * max(Q(s',a')))]$

p is the transition function,

r is the reward function and γ is the discount rate.

Since the estimate for the optimal Q-value,Q(s',a'), for the next state, s', depends on the next action,a', we make use of max(Q(s',a'))

We store the current Q-function which is initialised to 0 originally to a temporary variable, then keep on updating the Q-function.

We have to take a variable delta as 0 to store the maximum error of the Q-function, which is nothing but the original Q-value stored in the temporary variable subtracted by the update Q-value.

Because the Q-function is reliant on both states and actions, we must iterate through all potential actions as well as the set of all possible states in order to find the optimal solution. whenever

When delta which is the max error reaches a point below the threshold which is 0.001 that is when Q-value has reached its optimal value or converged which in my case is 0.0003

2. Results

Dynamic programming always tries to align towards the goal state,the result we want to achieve with dynamic programming is to know which decisions result in the highest reward. In the final convergence image we can see that from the start point it self the Q-values are maximum towards the goal state. In the start state the values are 17.3,17.3,17.4,18.3 for up,left,down and right respectively, DP will take the maximum of these values aligned towards goal state. So its take right=18.3 as its first value and move towards right since because of the wind. Similarly it'll take right after that because of the value 19.3 amd the wind. At one point because of the wind it goes all the way up then left again and goes to the extreme right top end of the table because of wind and the max values.It later goes all the way down and turns left and then goes up and reaches the goal because of the maximum values. The reason DP does it is

because it always tries to move towards the goal state and get the maximum value at each state.

We begin with a uniformly distributed random policy. That is, each of the four actions (up, down, left, and right) has a 25 percent chance of being selected at the start of the game. As the we move towards the midway the randomness fades away and we get a clear a idea of start state. At the very start all random and arbitrary values are assigned to table but as we move closer to the midway mark the table is updated in such a way that it tries to get closer to the optimal path. Th final convergence is where we get the optimal path from S t G

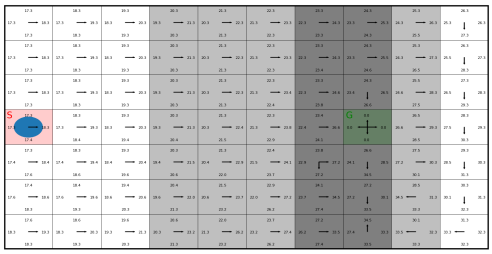

Figure 1. Q-value iteration state action estimates at convergence

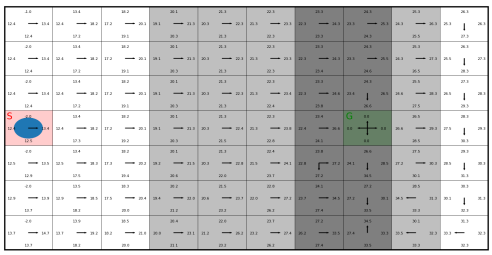

Figure 2. Q-value iteration state action estimates in midway

Since the Q-value its 18 iterations to converge I have taken my midway as 9.

3. V^* is the best value of a state if you could follow an optimum policy Basically V^* = max(Q(s',a')), it is asked for V^* for s=3 which converged optimal value at start. V^*=18.3. Since 18.3 is maxmial action in the start state. Since 18.3 is i.e the maximum value is towards right it means the object will move towards right when it tries to reach the goal.Therefore, when the iteration of Q values is completed, the value of the maximum action at the start state is V*.

5. he implementation still converges because G is the goal

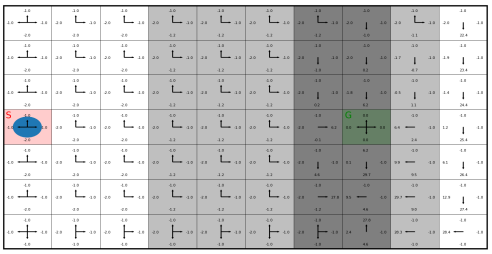

Figure 3. Q-value iteration state action estimates at beginning

state and after reaching the goal all the Q-values are 0 and that is when the implementation terminates. Another way to solve this problem is increasing the n-test-step in environment to 100 so that it takes more time to converge. WE make changes to the model function in the environment.py add and if else statment so that it doesnt converge.

1.1. Q-learning

1.Q-learning is an off-policy reinforcement learning algorithm that attempts to determine the optimal course of action given the current state. It's termed off-policy since the q-learning function learns from activities that aren't covered by the current policy, such as random acts, hence a policy isn't required. Basically , q-learning aims to discover a policy that maximizes total reward.

The 'q' in q-learning stands for quality. In this situation, quality refers to how valuable a specific activity is in obtaining a future reward.

Just like Dynamic programming we are asked to update 3 functions, of which init() is right. select-action() has two policies egreedy and softmax policy. Since egreedy is dependent on epsilon which lies between 0,1 we created a random uniform variable over it and check if its follows the equation

$\pi(a|s)$ = 1 - e, if a = arg max(Q(s,a))

$\pi(a|s)$= e/(A), otherwise

So which basically means we select action argmax if the variable is less than 1-e(epilson) otherwise take a random action. Basically, we choose a random action with a low probability that ensures exploration and do the greedy action otherwise

One disadvantage of the e-greedy is that they select random actions in a consistent manner. The worst potential course of action is just as likely to be chosen as the second best course of action. Softmax corrects this by assigning a rank

or weight to each of the actions based on an estimate of their action-value, as described above. A random action is picked based on the weights associated with each action, which means that the worst possible actions are unlikely to be chosen as a result. When the worst acts are really unfavorable, this is an excellent strategy to employ. We Boltzmann policy to calculate the softmax function.

we use the predefeined softmax function with best action for all states and take a random range of its length.

For update() function we just implement this formula

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

α - the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Which basically means the new Q(s,a) is the equal to the old Q(s,a) and the learning rate time temporal difference. Temporal Difference is reward r and the maximum reward that can be obtained for state weighted by discount factor, the maximum reward is nothing but estimate of optimal future value.

Algorithm

In the pseudo-code, we begin by initialize Q(s,a) to zero.

Following the initialization, we proceed by following the instructions outlined in each episode. In each step, we choose an action A from our select action function. It can either be softmax or egreedy.

The state of our system changes from s to s' as a result of taking action a. In turn, we receive reward of r,observe the reward, r, as well as the new state, s' Then we use these values to make changes to Q(S, A). We keep repeating this process updating state to the new state, and repeat the process until a terminal state is reached ie when s becomes terminal.

2. Result With 50 repetitions and timestamps of 5000 we get a smooth curve of both egreedy and softmax functions. If we take a look at figure 4,there is an exponential rise in all the the values of eplison and tau. All values of the functions initially start with -1 which is obvious. An exponential increase to values is seen in most cases until rewards reach to 0.5 after which . The exceptions to this softmax function of 1, it take more time to reach the optimal reward. The other anomaly is egreedy function of 0.2 which appears to have lower optimal rewards than the others.

The other anomaly is egreedy function of 0.2 which appears to have lower optimal rewards than the others. It takes almost the same amount of time to reach its optimal reward which is lower than the all the other values of egreedy and

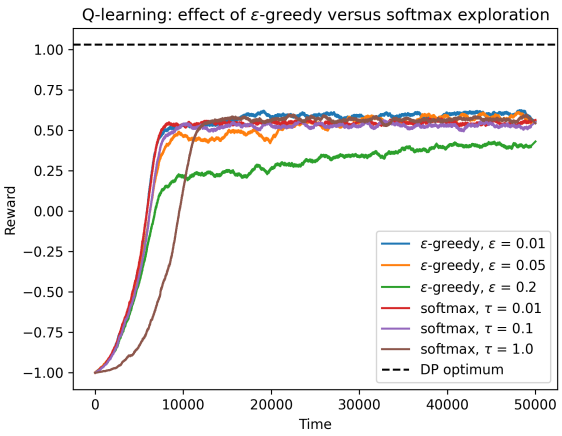

Figure 4. Q-learning comparing both exploration methods for different values of egreddy and softmax

softmax functions. In all the cases the reward start from -1 and end up in positive. I tried to see what difference it

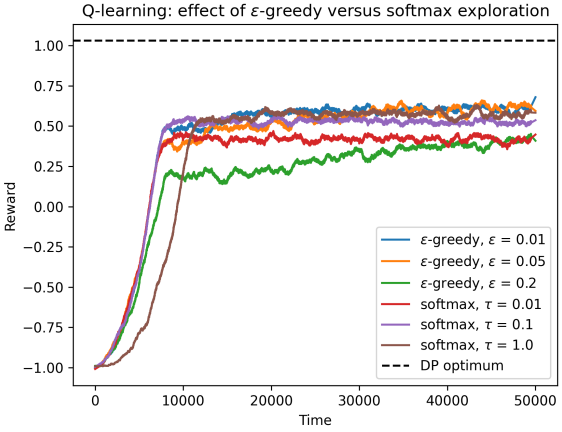

Figure 5. Q-learning comparing both exploration methods for different values of epsilon and tau- with 25 repetitons

would make if I just add the random variable value as b¡=1-epsilon instead of b¡1-epsilon, the result of the was that all the rewards were negative at all times.

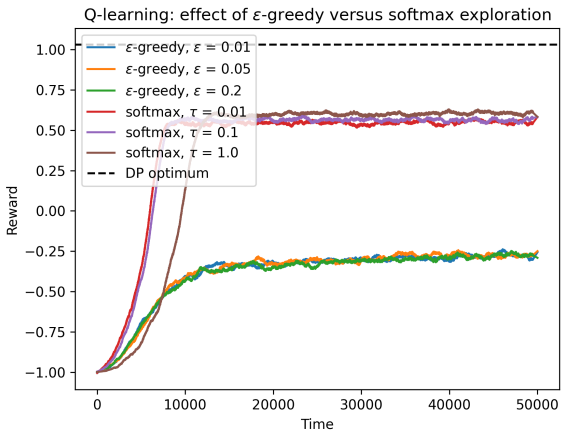

Figure 6. Q-learning comparing both exploration methods for different values of epsilon and tau- variable value ¡=epsilon

Interpretation. Of the two policies,softmax is better as it is constant and gives the equal rewards consistently and in most cases in 10000 timestamp. That's the reason why softmax is preferred over egreedy. In softmax as the temperature drops, differences in action value estimates become significant differences in action selection probabilities, resulting in the sometimes desirable effect of selecting "promising" action choices more frequently, as well as very low probabilities of selecting the actions with the worst estimates. From the all figure it is clear the learning curve DP is steady. It remains constant throughout. It it is optimal and steady at any given point. In RL the learning curve will never reach the level of DP. It will increase, decrease and not reach the level of performance of DP. In terms of performance DP is better than RL and it is preferred, because RL can never reach the level of DP.

1.2. Back-up: On-policy versus off-policy target

An off-policy is independent of the activities of the agent. It determines the optimal policy for the agent regardless of his or her motivation. Q-learning, for example, is a learner who does not adhere to policy.

Attempts to assess or improve the policy that is being utilized to make decisions are made using on-policy methods. Off-policy procedures, on the other hand, are used to analyze or enhance a policy that is distinct from the one that was used to generate the data. In Q-Learning, the agent learns optimal policy by copying other agents' policies. Q-learning is off-policy because the updated policy differs from the behavior policy. In other words, it assigns a value to the new state without actually following any greedy policy.

On-policy

SARSA (state-action-reward-state-action) is a policy-based reinforcement learning algorithm. In this algorithm, the agent determines the best course of action. Unlike Q-learning, the updating and acting policies are the same. This is policy learning.

The select action function in this also has to be updated is the same as in the Q-learning, we use the same egreedy and softmax function from the previous part for this.

In the update function() we implement the SARSA back up equation, the SARSA equation is

$$G_t = r_t + \gamma * max(Q(s', a'))$$

$$Q(s, a) = Q(s.a) + \alpha * [G_t - Q(s, a)]$$

Here, the update equation for SARSA depends on the current state s , current action a, reward obtained r, next state s' and next action a'.

Algorithm

Sarsa is a TD-Learning On-Policy algorithm. Unlike Q-Learning, the maximum reward for the following state is not always used to update the Q-values. Instead, a new action, and hence reward, is chosen following the same policy.The Sarsa algorithm's procedural form is similar to Q-Learning.

During each episode, we initialise S ,we take an action a in line with policies ,a current state s to a new state s while keeping an eye on reward r.

The action a is being taken in accordance with the existing e-greedy policy or softmax policy (given by the current Q-value function).

Q(s,a) must now be updated using the update function

It will do this until s becomes terminal and then stop

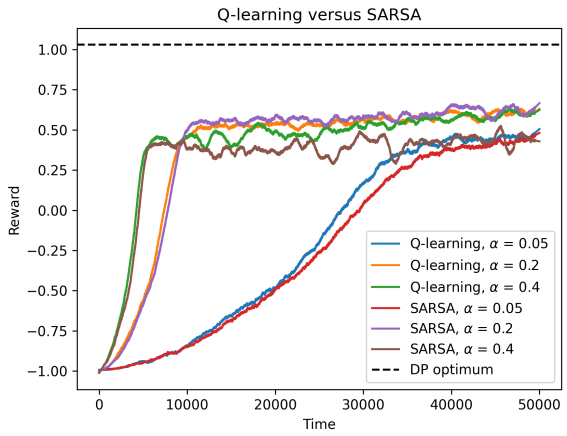

Figure 7. Comparison of Q-learning vs SARSA with different learning rate.

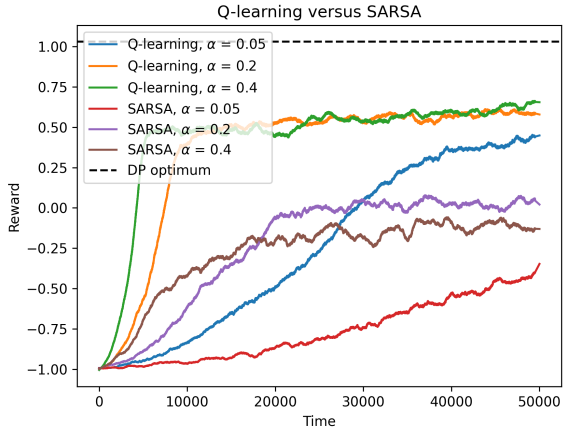

Figure 8. Comparison of Q-learning vs SARSA with different learning rate with 5 repetitions

Both the learning curve start with rewards -1 and end with rewards close to 0.5. Both Q-learning and SARSA for a learning rate of 0.05 the rewards vs time curves increases slowly and gradually.There is a gradual increase in the curve. The increase becomes significant 10k timestamp mark,after which is an expontential growth in the curve. For learning rate 0.2 there seems to be an almost similar trend both both SARSA and Q-learning. Both have a rather steep incline and reach the 0.5 rewards value in under 10k timestamps. It is obvious that increasing the learning rate makes both the methods Q-learning and SARSA perform better. As the learning rate increases the learning curve becomes more and more steep in its growth towards the rewards. At learning rate 0.4 both of them reach the optimal reward in under 5k timestamp. I tried it with less repetitions as you can see the curves aren't smooth. From this we can conclude that as the learning rate is increased both the curves become better and move steeply towards the rewards. In all the learning rate values given SARSA and Q-learning are almost similar , SARSA is preferred over Q-learning since we are more interested in the rewards gained while learning. If we want to train an optimal agent quickly because of the fast iteration and low computation Q-learning is preferred over SARSA.

1.3. Reflection

At any given time, DP is ideal and stable. The learning curve in RL will never be as optimal as it is in DP. It will rise, fall, and never approach DP's level of performance. DP is superior to RL in terms of performance. DP is static and steady unlike RL which keeps on changing.

Softmax is better than egreedy because ,as the temperature drops in softmax, differences in action value estimates become significant differences in action selection probabilities, resulting in the sometimes desirable effect of selecting "promising" action choices more often, as well as very low probabilities of selecting the actions with the worst estimates. Softmax gives higher probability to the current max action while it still explores other actions. egreedy doesnt do that.

In Q learning, you update the estimate from the maximum estimate of likely next actions, independent of whatever action you executed. In SARSA, you update estimates and do the same activity. The initial n rewards in n-step Q-Learning are, of course, taken from the current policy, so it's not entirely off-policy (and the target therefore mostly follows our behavioral policy

When working with a small number of factors and dimensions, RL algorithms work well and efficiently. However, as the number of factors and dimensions grows, the number of computations and memory required grows exponentially, requiring more and more computation power and memory, eventually rendering the algorithm useless. In its substitute, supervised learning can be used. So if you start with 3x3 tic tac its doable in RL tabluar but as soon 3^9which is doable but if it shifts to 5x5 is goes to 5^{25} which is too much to handle.