Smart Cab Driving Agent

Introduction

The Objective of this project is to implement the Q-learning bellmann algorithm to model and train a car(Agent) to navigate through various traffic scenarios to reach the final destination.we model the states and actions from different peices of information we collect at any given point of the program execution.All of the interesting steps have been implementd in the 'ageny.py' file of the project, it contains:

Global Variables:

```
q        : A dictionary holds the Q-values for all states exlored so far(Changing)

actions  : holds the set of actions permissible for each state(Constant)

trial    : holds the current trail number
```

Class and Methods:

```
LearningAgent(Agent):

     def _init_      : Initializes the Environment and required variables

     def reset       : Resets/changes any variable after each trail

     def update      : Chooses the best action in a state and updates the q-value

     def sigmoid     : Computes the logistic function value of the input

     def update_q    : Implements the Q-learning algorithm to update q-value

     def build_state : Builds a state from given inputs

     def get_max_q   : Returns the maximum q-value for a given state

     def max_random  : Returns a random Maximum from in case of multiple max's
```

Implement a Basic Driving Agent

We implement a basic driving agent by choosing a random action at each traffic intersection regardless of any other state information.The agent, as expected, shows no pattern of learning as it progresses throught the 100 Trials run in this mode.As it executes a random action each time, it bounces around the environment and sometimes reaches the destination , most times it doesn't.

Inform the Driving Agent

Next, we model the State space.Each state if a tuple made of:

oncoming : Is there oncoming traffic? if so, which way is it going?

left : Is there Traffic on the left(perpendicular road)? if so, which way is it going?

light : Green or Red?

deadline : (Time left to reach destination)/5

waypoint : Next waypoint (left or right or Foreward)

we need to know Oncoming and Left traffic to Learn to avoid accidents in similar traffic situations.

we need to know Traffic Light information to avoid accidents and negative penalty in similar situations.

we need to know next Way point to avoid negative reward for wrong actions.

we need to Know deadline to take a bad action/negative reward action such an action rewards us in the Long run, as in the case where time is running out so we could run a red light when no cars are present to reach the destination in Time.In our simulation the average deadline is less than 50 Units, which increases the state space drastically, so we divide it into intervals of '5' Units each(< 10 intervals on average)

The input 'Right' is not taken into consideration as a conclusion from the following argument:

Considering the other cars at any intersection obey Traffic Rules, the only way a car on the right would matter is

> Light is Red and we try to go Forward or Left, if we go right we would not conflict with the car on the right.

we could learn not to violate the Red light from other input variables and actions(for the Most part) .Hence, 'Right' could be considered redundant information.

With these as our parameters, our State space contains on average < 3*3*2*10*3 = 540 states

Note that most of these states won't be reached as our agent learns and optimizes its decisions.For 100 Trials, i found that the number of states explored was around 100 - 130 range.Notice that the Deadline input contributes to 90% of the states signifying the importance of reaching the destination in time.

This input(deadline)is interesting as generally, as the agent learns, it gets closer to the destination as the deadline approaches zero which, in a sense is global information about its position relative to the destination while all other inputs provide no information other than immidiate surroundings.Using this

input helps us make use of the potential future reward(by tweaking the Discount factor)information in the algorithm. Having more states also gives us the freedom to explore many other unusual/exceptions states and improve accuracy depending on the amount of training time we wish to spend.

Implement a Q-Learning Driving Agent

Now that we have our states and actions modeled, it is time to implement the Q- learning algorithm :

$Q[s,a] \leftarrow Q[s,a] + \alpha((r+ \gamma \max_{a'} Q[s',a']) - Q[s,a])$

Where : Q[s,a] is the q-value for each state-action pair

'α' is the Learning factor - range [0,1]

'γ' is the Discount factor - range [0,1]

we add any unvisited states to the 'q' dictionary and initialize all four q-values(None,Left,right,forward) to Zero as this means neither of the actions hold positive or negative rewards, they are neutral.

we then initialize 'α' to 1

we initialize 'γ' to 0

As we run the simulation, there is a clear learning pattern as it progresses through each trail and after about 50 trails, the agent reached the destination more often than not as is expected.

As the agent progresses though each trial, it accumulates information about the rules of the game, which actions result in negative reward & which actions result in positive reward in the form of q-values for each Q[s,a] pair.

the Algorithm in each state essentially

- checks the max q-value for the state

- performs that action to reach " s' "

- checks the max q-value for the new state and discounts it by 'γ' : $\gamma \max_{a'} Q[s',a']$

- adds the intrinsic reward for that action taken and subtracts the q-value of the previous state : $(r+ \gamma \max_{a'} Q[s',a']) - Q[s,a])$, this gives us the "net gain" in reward for the action we just took.

- Discounts this gain further by 'α', as how much we want to learn from this single action instance : $\alpha((r+ \gamma \max_{a'} Q[s',a']) - Q[s,a])$

- adds this Discounted gain(+ve or -ve) to the q-value of the previous state ' s '

These steps essentially accumulates and store the knowledge from previous trials and hence our agent is able to perform significantly better than when it took Random actions.

After Implementing the q-learning parameters and running the simulation, there is a significant improvement in the success rate of the agent.In the intial trials < 10 , there is little to no improvement and the cars do get stuck in loops that could represent local minimum.But as the q-values update, these loops are broken and the agent follows the next_waypoint with more accuracy and gets closer to the destination every few steps (as seen in the Pygame smulation) and eventually after about 30 trails, it reaches the destination most of the time.

One more point to note is, After running the simulation for 100 trials, the agent explores, on average only 80 -100 states ,far less than the total possible states(540) and less than when we introduce the exploration factor(100-130) as we see in the next section.

Improve the Q-Learning Driving Agent

Exploration rate

Next, we try to improve our learning agent by tweaking some of the parameters.First, we Tune the exploration rate epsilon - 'ϵ'.This parameter essentially controlls how we explore the state space:

At each state, should we blindly choose the action with max q-value? what if the difference in q-values is very small or even '0' and other actions need some some exploration to find their value more accurately?.

To address these questions, at each state we choose an action Randomly with some probability even if there is a max q-valued action.But we should want to do this with less and less probability as we run through more iterations of our model as we would like to increase Trust in our gathered q-values.

In the code, this is implemented by comparing a random float(0 -1) to the output of a scaled sigmoid function with 100 number of trails being the max input.

Learning rate

The learning rate 'α' is set to 1 initially as we would like to learn everything from a Q[s,a] due to the nature of this simulation.As the rewards, rules and actions are NOT probabilistic.

The exact same Learned situation at a future point of time would give us the same reward(exept for the final destination) and hence lowering this would likely not give us any benifit.

'α' rates of 0 to 1 have been tried with 50 trails with 0.1 increments and the general trend is the accuracy increases as 'α' increased.

Discount Factor

The Discount factor 'γ' is set to 0 initially through the following argument:

- The discount factor esentially weights the long term value of an Q[s,a] pair.For our simulation, the start and Destination change for each Trial and so do the traffic at a given - waypoint,Trial.So the 'Future Value' of a particular Q[s,a] is dependent, mostly on the closeness of the state to the Destination and we not have this information explicitly.Hence this information is not very reliable and hence be discounted severly.

But, we have modeled our state to include Classes of size '5' each of the deadline.And a future value of a Q[s,a] could be relavent in the following scenario:

- we are close to converging at the optimal policy and the Deadline class is '1' i.e Deadline <= 5, and a Q[s,a] has high q-value even though it violates some rules, because in previous trials, it reached the destination in time by incurring some negative reward at the very end(deadline class 1 or 2),which means it is near its destination(as it is a close to optimal agent by now) to gain a reward of '12' at the destination.

In this case, we do get some 'future value' information from the q-value.

values of 0 - 1 have been tried with 0.1 increments and there is noticable decline in converging time as 'γ' increases from about 0.4 - 1.

Exploring the Parameter Space

Below is a Snapshot of my Trail runs :

Learning	discount	success(x/50)(with epsilon)	success(no eplilon)
1	0	25	46
0.8	0	25	46
0.6	0	22	45
0.4	0	20	46
0.2	0	21	45
0.1	0	21	45
1	0.2	25	46
1	0.4	24	45

1	0.6	19	44
1	0.8	20	45
1	1	19	46

The optimal policy for this simulation would be one where the Agent reaches the Destination Every Time Within the given deadline and does not Incurr any negative rewards.

After tuning the Parameters and training it for 100 Trials every training cycle, the agent reaches its destination its Destination in time on average 9/10 times on each fresh Training cycle and on each Trail run after each Training cycle, the agent incurrs < =1 negative penalties on average, depending on the number of waypoints in the path.I would say this agent is close to its optimal policy.

From the Parameter Space table(50 trails on each observation) above we can gather the following general conclusions:

- As a general trend, success rate decreases (converges slowly) as learning rate decreases from 1 to 0.
-As the Discount factor increases from 0 to 1, there is no general linear trend in success, values of 0,0.2 seem to be slightly better than the others.Although Statistically significant differences were not found for 0 - 0.3 , i settled at a gamma value of 0.2 in hopes of succeding in capturing some future value in some exeptional cases(all else being equal).

- The most important observation would be that the absense of epsilon for our simulation tends to increase the convergence rate Drastically.Hence, the best option would be to exclude epsilon altogether, atleast for out particular simulation.

A plausible argument for this anomoly could be that our simulation is simple(few rules) and the agent needs only a few states(examples) to learn about the rules of the game and exploring random states to "Learn" more is a redundant choice and so it converges sooner without this random exploration.In a More complex game though, epsilon would be a good decision to argue for.

After this analysis, The final parameters i settled on are :

Learning rate 'α' - 1, from the range [0,1]

Discount factor 'γ' - 0.2, from the range [0,1]

Exploration rate 'ϵ' - sigmoid(nth_trial)

n_trials (training) - 100