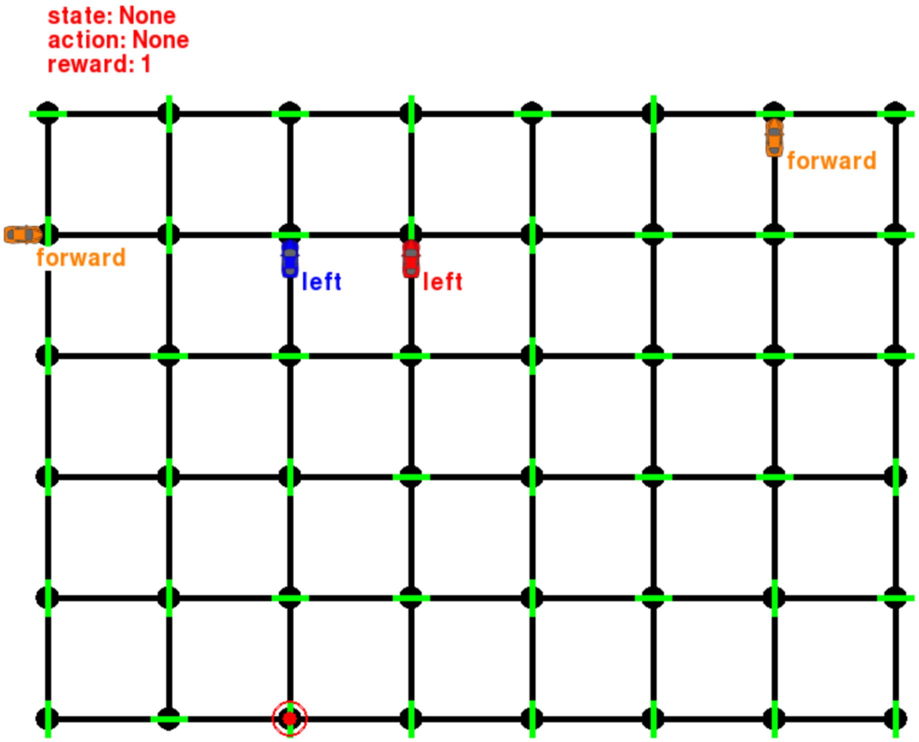
Train a Smartcab to drive

Project 4: Reinforcement Learning

1. Project Description

A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. In this project, we will use reinforcement learning to train a smartcab how to drive.



Environment

Your smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open.

US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.

To understand how to correctly yield to oncoming traffic when turning left, you may refer to this **official drivers' education video**, or this **passionate exposition**.

Inputs

Assume that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination).

The smartcab only has an egocentric view of the intersection it is currently at (sorry, no accurate GPS, no global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go).

In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

Outputs

At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

Rewards

The smartcab gets a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan).
It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident.

Goal

Design the AI driving agent for the smartcab. It should receive the above-mentioned inputs at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

2. Project Report

Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action (None, 'forward', 'left', 'right'). Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.
Run this agent within the simulation environment with enforce_deadline set to False (see runfunction in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

In your report, mention what you see in the agent's behavior. Does it eventually make it to the target location?

Answer:

The goal for the agent is to reach the final destination by picking the optimal actions. What we can observe for our agent (red car) behavior is that:

Act randomly

- Regardless of the conditions of the environment, the agent moves randomly across the map, because of the way the agent has been programmed.
- The agent constantly tries new actions without using rewarding information of any knowledge of the past actions, in other words the agent does not prefer actions that it has tried in the past in order to obtain reward.
- The approach to reach the goal is clearly random and not optimal.

Does not learn

- We observed that run after run the agent does not learn that running on a red light results in a negative reward.
  - Example:
    LearningAgent.update(): deadline = 7,
    inputs = {**'light': 'red'**, 'oncoming': None, 'right': None, 'left': None}, **action = forward**, **reward = -1**
- The agent sometimes remains in the same position (action = None) even when the light turns green and there is no oncoming traffic.
  - Example:
    LearningAgent.update(): deadline = 20,
    inputs = {**'light': 'green'**, 'oncoming': None, 'right': None, **'left': None**}, **action = None**, reward = 1
- Even though sometimes the agent was a block away to the target destination, it chooses a different direction than the one to reach the destination.

Low success rate

- Majority of the time the agent does not reach the destination within the deadline, only in few cases it does.
- As we see, the performance of the agent is poor as expected given the actions were random. Sometimes the agent performs well, sometimes it doesn't.
- Out of all runs the success rate of the random agent is always around 20%.

Although it does not perform well, we can still use it as a proof of concept that Q-Learning improves the agent behavior. We can calculate the number of successes runs (when the learning agent reaches the goal within deadline time) over the total runs before and after Q-Learning is applied. That way we can see how different the success rate is before and after Q learning is applied.

Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Justify why you picked these set of states, and how they model the agent and its environment.

Answer:

- The goal of the agent is to reach the destination by following the directions from the planner without break any traffic rules. In order to make decision about proceeding or not at the intersection the agent needs to know the traffic light value and the traffic data of oncoming and left side.
- If the light is green, the agent can turn left if there is no oncoming traffic. If the light is red, the agent can turn right if there is no traffic coming from left. In order to learn the value of each action the agent will visit every state multiple times. Based on these rules, so the agent can learn and follow the traffic laws, I chose to monitor a state composed of four variables:
- For the purposes of my learning agent, I chose to monitor a state composed of four variables
    - **light** →input [0] - (red / green) - the traffic light state.
    - **oncoming** → input [1] (none, left or right) - any oncoming traffic.
    - **left** → input [3] (none, left or right) – any traffic approaching from the left.
    - **next_waypoint** - represents the next waypoint that the cab must move to in order to reach its destination.
- Due to the US right-of-way rules, the traffic coming from the right input [2] does not matter and will therefore be ignored from the states.
- The time for the agent to learn will be significantly longer, if the number of states is bigger. Choosing these 4 states results in good performance.

Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

What changes do you notice in the agent's behavior?

Answer:

Does not act randomly

- We can notice that after implementing the Q-Learning algorithm, the results were significantly improved compared to the initial agent setup where the agent was selecting actions randomly.
- The agent quickly learns the environment and the traffic light rules that allows the agent to make moves toward the target destination.
- In general, the agent is moving towards the destination and gets to the destination much faster, but does not always make optimal choices. We can notice generally positive rewards in each run and rarely any penalties.
- The agent is able to reach the target location before the deadline value has expired.

Learns from the past actions

- We can also notice that once the agent learns the bad impact of the red light move, he is not doing that again.
- We can notice that in case of no traffic of green light the agent doesn't choose to stay in the same position anymore.
- Being close to the target destination, results in choosing the target destination as next move.

Higher success rate

- The agent now consistently learns to reach the destination in time. During the first couple of runs it fails to reach the destination, the agent accumulates both positive and negative rewards, but after enough runs the trend is changed and it almost always reaches the destination, the learning agent tends to accumulate only positive rewards. The success rate is drastically improved, from ~20% to ~90%.
- The agent's shows improvement that can be measured in terms of the ability to avoid traffic accidents, get negative rewards, and ability to reach the destination before the deadline.

Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

The formulas for updating Q-values can be found in **this** video.

Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

Answer:

The way Q-Learning works is, by using observed rewards from state/action pairs to learn a policy of best actions for a state space.

Following parameters are part of the Q-Learning process:

- **Learning rate (alpha)** → determines how quickly the agent learns from each move.
  - 0 → the agent ignores any feedback.
  - 1 → the agent replaces existing knowledge with what was just learnt.
  - 0.5 → mean between what it already knows and has just learnt.
- **The Q-values** → table of potential future actions (argmax(Q(s', a')))
- **Discount factor (gamma)** → the level of importance the agent should attribute to future rewards versus current rewards.
  - 0 → any future reward is ignored.
  - 1 → would mean that it is given a high weight and therefore will heavily influence the quality value.

Several value of alpha, gamma and epsilon has been tried out in order to have the best learning experience for our agent. I've tried fixed values and decay over time. In my final model a decay over time (1/time_step) was applied for each iterative time step. These resulted in improved success rate, from 70% initially to improved one of 80-90%. In addition to making the adjustment to the decay rate, the initialization of the Q-values was also changed from 0 to 1. This change resulted in improved success rate, now between 90-95%.

After initializing the Q values table, the agent checks the QTable for the presence of the current state, if it does not exist, it is added to the table. If the state has been encountered before, then the program will choose the action with the maximum Q' value for the state. In situations where multiple actions have the same maximum Q value, then the state is chosen randomly.

Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

Answer:

Even though occasionally the agent is making choices that are not what the planner advised and sometimes near the target destination it will take a wrong turn and will loop around to reach the destination, the agent is definitely close to reaching the optimal policy.

In 100 iterations, the agent barely receives penalties and very frequently reaches the destination in the minimum possible time. On average the agent reaches the destination 92 of the all the trails (100) and 10 for 10 in the last 10 trials. After couple of iterations, it seems to consistently take correct actions. The agent performs really well.

The learning process could be improved by increasing the number of dummy agents. That way the agent would probably interact more frequently with them and would receive penalties earlier. This will result in reducing the training time.