Train a Smart cab How to Drive

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

Environment

Your smart cab 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 smart cab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smart cab 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 smart cab 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 smart cab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement).

Rewards

The smart cab 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 smart cab. 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.

Tasks

Setup

You need Python 2.7 and **pygame** for this project: https://www.pygame.org/wiki/GettingStarted
For help with installation, it is best to reach out to the pygame community [help page, Google group, reddit].

Download

Download smartcab.zip, unzip and open the template Python file `agent.py` (do not modify any other file). Perform the following tasks to build your agent, referring to instructions mentioned in `README.md` as well as inline comments in `agent.py`.

Also create a project report (e.g. Word or Google doc), and start addressing the questions indicated in *italics* below. When you have finished the project, save/download the report as a PDF and turn it in with your code.

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 `run` function 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 agent eventually does make it to the target location, with unlimited timeline provided, however it does so with extensively large number of steps. The agent iterates over all random actions and moves into random direction. And after a very-very long time, the agent finally reaches the destination and that too by random chance. As for the agent's behavior, the agent moves into random direction of the four directions available

- *Left i.e. move left from the current intersection*
- *Right i.e. move right from the current intersection*
- *Forward i.e. move forward from the current intersection*
- *None i.e. do not make any move at the current intersection*

<u>Auxiliary Information regarding "Silly Route planner that moves in perpendicular fashion":</u>

With the Silly Route Planner implemented, the agent eventually makes it to the target location in very short time compared to the random Planner used earlier. The agent moves in a straight line, often in a perpendicular fashion. The car moves in a straight line in the direction it is facing and continues in it, until it is in the column/ row (i.e. X, Y line in which the destination exists), then when it reaches the intersection, it turns towards the destination and then continues straight until it reaches the destination. Strangely enough, even when in cases where it is stuck in a red traffic light for a couple of turns in the direction it was moving earlier, it fails to take the deviated path, to reach the destination faster i.e. if the car is moving left then even when the left path is blocked by the red light for 2 or 3 or 10 or any number of turns. The car will wait until the red light goes green and until it can resume the left straight path again.

After every step, the car senses the new changed environment i.e. Partial observability of the environment such as the present traffic light condition at the current intersection i.e. whether the traffic light is green or red, where green implies the car can go forward and the current intersections status. The red light at the intersection implies that the car can make the left turning

However to make the right turn, it can only do so when the light is red and there are no cars at the intersection moving in a ”forward” path.

Once the car observes the current environment, it has to self-decide which action to take i.e. in which direction to move itself. In our original case, the next action to be taken, is currently received from the “RoutePlanner”. The RoutePlanner takes currently   “our location”, “our destination” and our car’s current heading direction (i.e. in which direction is our car facing) into account and determines which direction or action should we be taking i.e. forward, left, right or none.

The RoutePlanner, however doesn’t take the current environment i.e. traffic light’s status or oncoming traffic into account. It simply acts as a compass, which tells us which direction we should take.

Based on the compass i.e. (RoutePlanner’s) suggestion as our primary guideline, i.e. which direction should we be heading, we then start taking the environment (i.e. traffic lights, oncoming traffic) into account and then finally take the most appropriate action. For example, a RoutePlanner may tell us that we have to go forward, however if the light is red for forward movement, then we cannot move forward, else we will crash into other cars.) This Environmental factors such as Traffic light, oncoming traffic are taken into account by the Environment class’s act method.

During the movement, if the action is allowed by environmental factors, then we receive the positive reward, and we receive extensively large reward when we reach the final destination.

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 following are the states identified as being appropriate for modelling the driving agent.

1. Light: The state of the traffic light. Traffic light are one of the top-most important factors during the driving. We want our agent to make the informed and best decision by taking the light condition into account, hence we include this into our state. We want our agent to learn what actions are allowed and which are disallowed in each traffic light conditions i.e. green light or the red light.

2. Oncoming Car: We also have to take the oncoming traffic into account. Since it affects our cars movement e.g. if there is oncoming traffic, then we cannot take the left turn or else we will crash into the car. Our intelligent agent will have to be able to learn this and hence, we incorporate it in the state.

3. Left: Similarly, as in the oncoming traffic case our intelligent agent will also have to take into consideration if there is traffic coming from left or not. E.g. if there is traffic coming from the left then   we cannot move left   or right, else we will crash. Hence we need to incorporate this into our state, so that our intelligent agent can learn the rules for the traffic coming from left.

4. Right: Similarly our intelligent agent will also have to learn the rules for the traffic coming from right, hence we also need to incorporate this into our modeling agent's state.

As for the other preexisting state information's like, location, heading, deadline and destination. We exclude them from our state, because we want to learn the globally applicable rules and   since each of these parameters are highly specific to the current position (i.e. location, heading) or current

Iteration (i.e. deadline and destination), they are irrelevant   to be considered as the globally applicable rules and hence we exclude them from the driving modeling agent's state.

How does the state change after every step?

After every step, the step changes as,

1. Light: The state of the traffic light changes from the previous intersection’s traffic light to the new current intersection’s traffic light status i.e. green or red.
2. Oncoming Car: The current environment is sensed, and if there is any oncoming car in the current intersection in this new step, then it is noted in the new state.
3. Left: Similarly, any car coming from the left of the intersection is also noted for the new state.
4. Right: Similarly any car coming from the right of the intersection is also noted for the new state.

Auxiliary Information regarding the Initial state of the “Random / Silly Planner”

The initial state of agent for normal unintelligent agent is often be described by its four properties.

Location: The location of our car often depicted in x, y coordinates system e.g. location of (1, 2) implies that our car is in 1 blocks away across x- dimension and down 2 blocks away in y dimension. All locations ranging from (1, 1) to (8, 8) are possible. While (1, 1) represent the top-left starting point, the (8, 8) represents the bottom-right corner.

Heading: This implies which direction the car is facing or heading towards. The possible four heading for a car in our scenario is

East: With heading (1, 0)

West: with heading (-1, 0)

North: with heading (0,-1) and

South: With heading (0, 1)

Deadline : Indicating the number of steps or time available i.e. if we have to reach someplace within 20 steps or 20 mins time, and after that time the place closes, then with the few i.e. 5 mins or so remaining, we will most probably take a more riskier path, if it means getting us to the place in time. The deadline is eventually supposed to act as such factor, in our car driving scenario.

Destination: This indicates the destination’s coordinates in the x y grid.

Auxiliary Information: How does the state change after every step?

After each step, each of the following components changes as

Location: The car moves to the new location, based on the action obtained from the planner. The location change however is controlled by the different circumstances i.e. traffic light, and the incoming traffic.

Heading: The car face is changed to the respective east, west, north or south direction, dependent on the actions performed.

Deadline: After each steps, if the deadline enforce is turned on, then the deadline decreases by a step at a time, indicating the number of steps remaining to reach the destination.

Destination: The destination remains the same throughout the iteration.

After every step, if we were moving in a straight path, then we continue in the path, until we reach the row / column (X, Y line) which contains the destination. On reaching the col / row, we then make the turn towards the destination location.

In the process while moving in a straight path, if we encounter a red light, then irrespective of the number of steps remaining for us, we tend to wait until the light turns green and can resume the straight path.

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: After implementing the Q-learning, we observe that the car does not move in the random or perpendicular fashion anymore. Rather during each iteration, the car explores different situations and creates rules that it can later use to make the most optimal choice. This optimal choice or policy is then periodically used by the car to optimize its path, so that it maximizes the destination reaching probability.

During the Policy Learning   phase, primarily the following situations are taken into account by the car.

5. Light : The state of the traffic light
6. Oncoming: If there is oncoming car from the front. If there is oncoming, then the direction the car is moving towards, is also considered
7. Left : If there is a traffic coming from left and if yes, then the direction, in which the car is moving towards, is also taken into account
8. Right : If there is a traffic coming from the right and it yes, then the direction, in which the car is moving towards, is also taken into account
9. Urgency: The number of steps available to us. We consider the three modes of urgency low, medium to high urgency depending upon the number of steps available to us.
10. Our car's Action: The action which our car is supposed to take, based on the optimal policy.

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.

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: We performed the exhaustive search over the alpha, discount and epsilon parameters ranging from 0 to 1. From the exhaustive search we found out the optimal value for these parameters to be 0.7, 0.2 and 0.0 respectively

The following parameters were exhaustively searched for

1. Learning rate: The learning rate determines to what extent the newly acquired information will override the old information. 0 means that the agent will not learn anything, as the old information is never overridden with new information i.e. what we learned. While 1 implies

that the agent will only consider the most recent information. [1] Since our agent is going to grow more intelligent with time, we increase the learning rate from lower to higher end from 0 to 0.9 and see how it performs. The exhaustive search result from the table shows the value of 0.7 to be the most optimal one.

A balance between always selecting the old information (i.e. learning rate 0) and always using the new information (i.e. learning rate 1) i.e. always discarding the old knowledge, seems to be the most appropriate solution. This is in harmony with our observation of 0.7 as the most optimal learning rate. The upper end of the learning rate is also justified, considering the fact that we expect our agent to grow more intelligent with each iteration and hence we expect our agent to learn more valid rules at the last stages of iterations, which means newer information will then override the older rules or information learnt with more chances.

2. Discount Factor: Discount factor determines the importance of the future rewards [1]. Its value ranges between 0 and 1, with 0 representing a myopic i.e. short sightedness by only considering the current reward, while the value of 1 implies it is longer sighted and seeks to maximize the total rewards.[1] For example in a scenario, if the first three steps yields minimal gain, but after that 3 steps, it yields maximum/ huge gains, then the shorter sightedness agent will make poor decision in that case, while long sightedness agent will take a greater perspective into account and hence will choose the best scenario given above, even it means minimal gain in the short term. While it is desirable to have high discount factor, having it too high, might imply that we have to spend more computing resource and especially will be drain for cases, when we have a lot of steps to take into account, compared to our 8 * 8 road network.

   From our particular exhaustive search, we observed the 0.2 to be the most optimal value.

   Strangely enough, however the optimal discount was observed to be 0.2. This can be explained by the fact that we have a stochastic environment and since in every next step, our environment is going to change, which in turn will affect the reward that we will receive at that step, the future rewards are of less significance in our case and hence its value at the lower end is justifiable. For e.g., we might be at the intersection A and then we are computing the best action by evaluating the action at intersection B, taking into consideration the current state of intersection B. However, in the next step we observe that all the intersections state change i.e. lights. Hence the earlier light condition is likely to have changed from what we have taken into account to calculate the best action, thus implying that the future rewards are of lesser importance.

3. Random Action Selection Factor (Epsilon- greedy Strategy): Higher random action selection factor implies we choose random action to optimal policy (i.e. more exploration and lesser of exploitation (exhaustive search over all possible actions for best action)).

   Some of the best techniques to implement best Exploration factor is by using the fixed schedule (i.e. making the agent explore less as time goes by) or using adaptive heuristic methods.

   In our case we use the fixed schedule approach, where we initialize higher random epsilon value 0.5 for the first 10 iterations then, we change the random action factor to 0.2 for iteration

   However, unlike expected the fixed schedule approach lead to poor destination reach count than when we used the random selection factor of 0. Hence we choose 0 as the random action selection factor.

4. Initial Condition (Q0): Since Q-learning is an iterative algorithm, that assumes an initial condition, higher initial condition can encourage more exploration, in the face of uncertainty i.e. (optimistic initial conditions). Hence we are going to update the use the initial value of 9 i.e. Higher reward than all actions but slightly lower reward than when the destination is reached. [1].

   During our experimentation, we observed that the value of 0 as the initial condition lead to the most promising result. This might be due to the fact that we are always computing for the best optimal policy at each step. Hence we choose 0 as the initial condition.

Results Table:

Learning Rate	Future Reward Weight (discount)	Random Action selection Factor (epsilon)	Destination Reached in the last 20 iterations	::Total Reward
0.5	0	0	8	13

0.5	0.2	0	8	27
0.5	0.4	0	7	11
0.5	0.61	0	10	10.5
0.5	0.8	0	9	13
0.6	0	0	7	18
0.6	0.2	0	7	20.5
0.6	0.4	0	9/10/8	13
0.6	0.61	0	7	16
0.6	0.8	0	10	16
0.71	0	0	12	13
0.71	0.2	0	12/8/7/11/9	26.5
0.71	0.4	0	10	18
0.71	0.61	0	5	13
0.71	0.8	0	5	21
0.8	0	0	11	20.5
0.8	0.2	0	8	18
0.8	0.4	0	14/10/10/7/3/6	10.5
0.8	0.61	0	5	18
0.8	0.8	0	10	13

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?

No, our agent does find the most optimal policy. Our car agent reaches the destination in 1/3rd to 1/4th of the cases. And even when it does, it tends to, not to do so in the minimum possible time. Our agent tends to move using the earlier learned rules and hence seem to move around based and hence only on 1/3 – 1/4th of the case reach the destination.

Also we observed that in instance of the cases, our car agent tends to disobey the traffic rules and act disobediently, potentially leading to crash. Following were such instances

For example, for case 1 when the light is green for us and there is oncoming car moving forward, then we should not take the left action as that will lead us colliding directly with the oncoming car. However our agent seems to assign a positive reward to the accident while it should have been other way around.

This seems to be the case, due to the fact that we have assigned very small penalty of -1 to our car agent in case of such accident or penalty cases. Increasing the penalty to significant amount may cause our car agent to prevent such actions.

1. *(state_tuple(light='green', oncoming='forward', left=None, right=None), 'left'): 0.6015744000000001,*
2. *(state_tuple(light='green', oncoming='right', left=None, right=None), 'left'): 0.6015744000000001,*
3. *(state_tuple(light='red', oncoming=None, left=None, right=None), 'left'): -0.7,*
4. *(state_tuple(light='red', oncoming=None, left=None, right='right'), 'left'): -0.7,*
5. *(state_tuple(light='red', oncoming=None, left='forward', right=None), 'right'): 0.504,*
6. *(state_tuple(light='red', oncoming=None, left='right', right=None), 'left'): -0.7,*
7. *(state_tuple(light='red', oncoming='left', left=None, right=None), 'right'): 0.624997196696207,*

Also the only possible way to learn the most optimal solution at any intersection is, if our agent knows its current state and destination i.e. its current intersection and its final destination. However because learning the policy specific to the intersection and the destination, will make the policy highly specific, it will not be useful as a generic policy. In favor of the generic policy, we tend to eliminate the current intersection and its destination and hence accept minor sub optimality in our solution.

As for the penalties, our agent is able to learn the penalty condition in partial of the cases and avoid it, with negative reward associated with the penalty situation. For example

1. *(state_tuple(light='red', oncoming=None, left=None, right='forward'), 'left'): -0.7,*
2. *(state_tuple(light='red', oncoming=None, left=None, right='left'), 'left'): -0.7,*
3. *(state_tuple(light='red', oncoming=None, left='forward', right=None), 'left'): -0.7,*
4. *(state_tuple(light='red', oncoming=None, left='left', right=None), 'left'): -0.7,*
5. *(state_tuple(light='red', oncoming=None, left='right', right='forward'), 'left'): -0.7,*
6. *(state_tuple(light='red', oncoming='forward', left=None, right=None), 'left'): -0.7,*
7. *(state_tuple(light='red', oncoming='left', left=None, right=None), 'left'): -0.7,*

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

Evaluation

Your project will be reviewed by an Audacity reviewer against **this rubric**. Be sure to review it thoroughly before you submit. All criteria must "meet specifications" in order to pass.

Submission

When you're ready to submit your project go back to your Audacity Home, click on Project 4, and we'll walk you through the rest of the submission process. You need to turn in your final code (`agent.py` only) and report as a PDF file.

If you are having any problems submitting your project or wish to check on the status of your submission, please email us at **machine-support@udacity.com** or visit us in the discussion forums.

What's Next?

You will get an email as soon as your reviewer has feedback for you. In the meantime, review your next project and feel free to get started on it or the courses supporting it!

Ref: 1 https://en.wikipedia.org/wiki/Q-learning