Report: Project 4 Train a Smartcab to Drive

Masashi YAMAGUCHI

1. 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:
I have added following code in update() method in agent.py so that it choose actions randomly.

```
#random action policy
action = random.choice(self.env.valid_actions)
```

In the simulation with "enforce_deadline=False", my random agent had 145 steps to move (, simulation starts deadline from 45 and ends at -100).
The agent goes wrong direction from target location, also got penalty points in most of steps, but it reached the target location in some games. It reached target location within 20 steps in the best case I've seen. So I think the random action policy is bad policy but it is not the worst policy since it can reach target location eventually.

2. 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:

Following is inputs which agents can percept from environment.

- Light: traffic signal light
- Oncoming: Other car existence and it's next waypoint on oncoming side at intersection
- Right: Other car existence and it's next waypoint on right side at intersection
- Left: Other car existence and it's next waypoint on left side at intersection
- Next waypoint: Navigation to the target location.
- Deadline: Number of available steps that the agent can take actions.

I have defined new state from combination of inputs such as Light, Oncoming, Right, Left and next waypoint as followings.

$$Agent\ State \equiv \{Light, Oncoming, Right, Left, Next\ waypoint\},$$

$$Light \in \{Green, Red\},$$

$$Oncoming\ , Right\ , Left \in \{None, Forward, Right, Left\},$$

$$Next\ waypoint \in \{Foward, Right, Left\}$$

I have adopted the inputs that have meaning to learn reward and penalty. Inputs of Light, Oncoming, Right, Left are usefult to learn the condition of the penalty. It is for avoiding the violate traffic rule and crash other agents. Input of next waypoint is to learn the reasonable direction to target location and getting reward.

Input “deadline” is useful for agent to learn the concept of time out in simulation. But I have not adopted it in state definition since it is enough to use input “next waypoint” to arrive before deadline.
If this simulation allows agents move while signals are red or other cars are at intersection, agents might be able to learn violating traffic rules when deadline is very close. But actually this simulation does not allow such situation. Also it makes state huge then the agent suffer from the curse of dimensionality.

3. 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:
I have implemented Q-Learning the policy called greedy policy. Agent chooses action has the highest q value in available actions. Also if there are multiple actions with the highest q value, the agent chooses one of them randomly.

At first the agent explore the map randomly and take penalties from violation of traffic rule. Then after decades of steps the agent’ s become to take reasonable actions for avoiding violation and arrive target location.
But when other agents are at same intersection, learning agent did not take reasonable actions. I think it is because learning agent had not enough opportunity to learn the behavior in situation that other agents are in same intersection.

4. 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?
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?
The formulas for updating Q-values can be found in this video.

Answer:

I have implemented following methods and compared each result. Then I have chosen greedy method as a final version. (Learning rate=0.1, Discount rate=0.1)

- greedy
- ε -greedy
- softmax
- random

Epsilon greedy method, agent takes random action according to parameter epsilon. When ε =0.0 the method is equal to random method. And ε =1.0 is equal to greedy method.

In softmax method, probability of actions is calculated from following equation.

$$P(a) = \frac{e^{Q(s,a)/\tau}}{\sum_a e^{Q(s,a)/\tau}}$$

Following table is a number of the agent reached target location in 100 trials.

Strategy	Reached target

Random	25
ε -greedy(ε =0.5)	64
Greedy	97
Softmax (τ =1.0)	81
Softmax (τ =100.0)	81
Softmax (τ =0.001)	80

From the simulation, greedy method is the best policy for this simulation.

Following is graph of accumulated number of reached target in greedy, softmax methods.

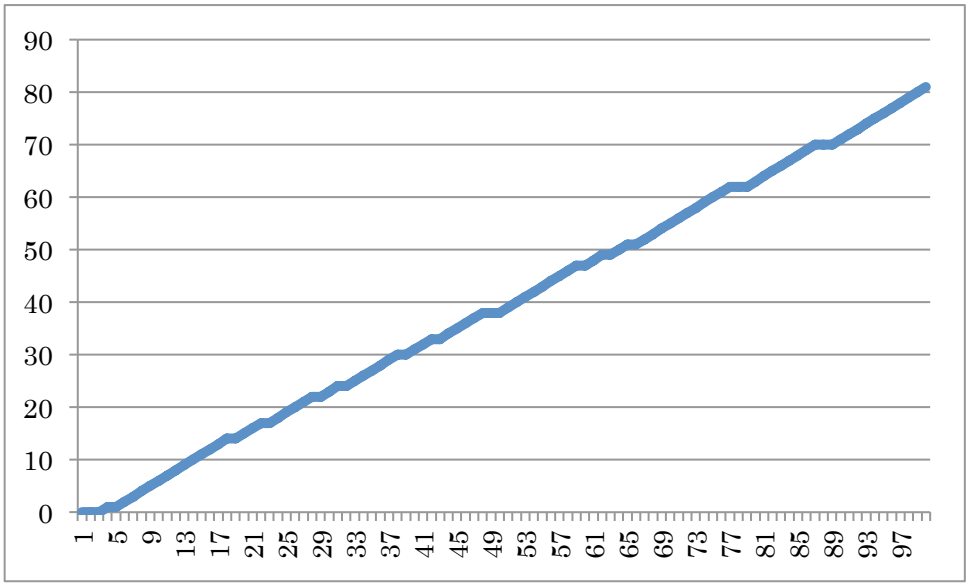


Fig. 1 Softmax Method (τ =1.0)

As shown in Fig.1, softmax method failed to arrive target location in first 5 games. It takes to learn the optimal policy.

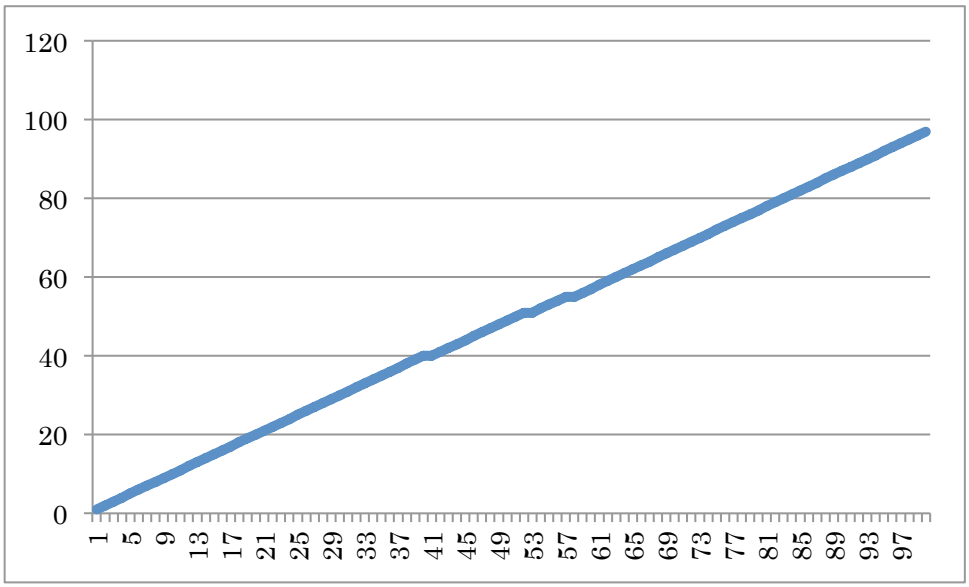


Fig. 2 Greedy Method

Greedy method had reached target location from 1st game. So it shows that greedy method leans the policy in short time.

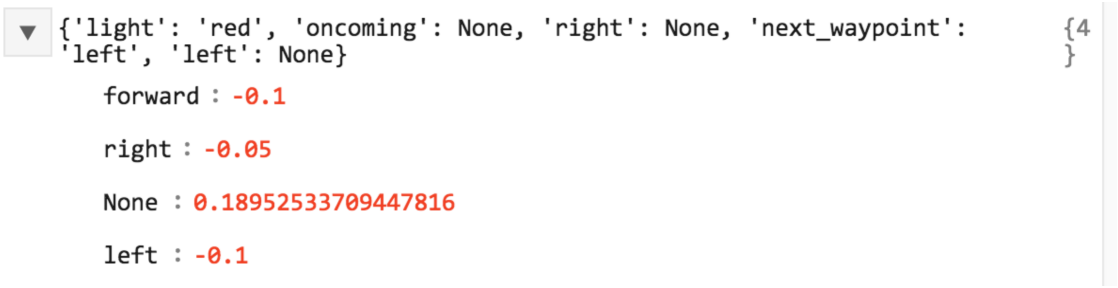
And following is different result depends on learning rate on greedy method. For greedy method learning rate does not affect the result. Because greedy method choose action based on the order of q_value. Even though it is low value, agent always choose the biggest q_value.
I think because environment in this simulation is stable (environment does not change in every game), so high and low learning rate does not affect the result. Basically learning rate is important for unstable environment.

Learning rate(Greedy method)	Reached target
0 (Random method)	26
0.01	98
0.1	97
1	99

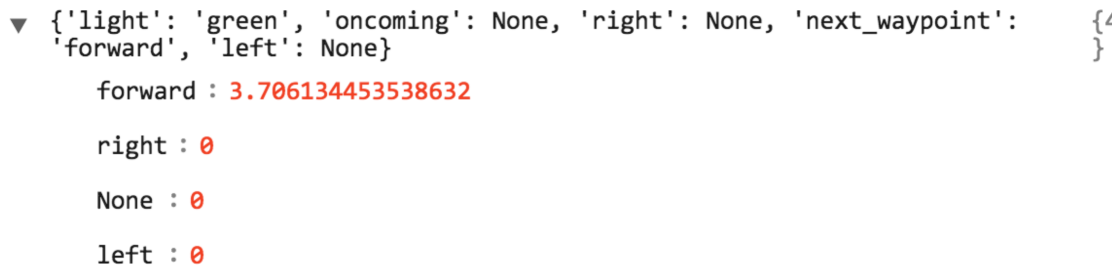
Agent learnt optimal policy on games.

Followings are part of my agent's Q table in final version (greedy method, 100 trials). The agent has learnt traffic rules by Q-Learning. It has the best q value on desired action.

- Stop at signal is red



- Choose desired direction based on next waypoint



- US right-of-way rule: On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight.