Project 4

Train a Smartcab to Drive

Udacity Machine Learning Nanodegree Program

By Shinto Theruvil Manuel

September 04, 2016

Table of Contents

[From Udacity]

Project Overview

In this project you will apply reinforcement learning techniques for a self-driving agent in a simplified world to aid it in effectively reaching its destinations in the allotted time. You will first investigate the environment the agent operates in by constructing a very basic driving implementation. Once your agent is successful at operating within the environment, you will then identify each possible state the agent can be in when considering such things as traffic lights and oncoming traffic at each intersection. With states identified, you will then implement a Q-Learning algorithm for the self-driving agent to guide the agent towards its destination within the allotted time. Finally, you will improve upon the Q-Learning algorithm to find the best configuration of learning and exploration factors to ensure the self-driving agent is reaching its destinations with consistently positive results.

Description

In the not-so-distant future, taxicab companies across the United States no longer employ human drivers to operate their fleet of vehicles. Instead, the taxicabs are operated by self-driving agents — known as smartcabs — to transport people from one location to another within the cities those companies operate. In major metropolitan areas, such as Chicago, New York City, and San Francisco, an increasing number of people have come to rely on smartcabs to get to where they need to go as safely and efficiently as possible. Although smartcabs have become the transport of choice, concerns have arose that a self-driving agent might not be as safe or efficient as human drivers, particularly when considering city traffic lights and other vehicles. To alleviate these concerns, your task as an employee for a national taxicab company is to use reinforcement learning techniques to construct a demonstration of a smartcab operating in real-time to prove that both safety and efficiency can be achieved.

Software Requirements

This project uses the following software and Python libraries:

- Python 2.7
- NumPy
- PyGame
    - Helpful links for installing PyGame:
    - Getting Started
    - PyGame Information
    - Google Group
    - PyGame subreddit

If you do not have Python installed yet, it is highly recommended that you install the Anaconda distribution of Python, which already has the above packages and more included. Make sure that you select the Python 2.7 installer and not the Python 3.x installer. `pygame` can then be installed using one of the following commands:

Mac:

```
conda install -c https://conda.anaconda.org/quasiben pygame
```

Windows & Linux:

```
conda install -c https://conda.anaconda.org/tlatorre pygame
```

Starting the Project

For this assignment, you can find the `smartcab.zip` archive containing the necessary project files as a downloadable in the Resources section. *You may also visit our Machine Learning projects GitHubto have access to all of the projects available for this Nanodegree.*

This project contains two directories:

- `/images/`: This folder contains various images of cars to be used in the graphical user interface. You will not need to modify or create any files in this directory.
- `/smartcab/`: This folder contains the Python scripts that create the environment, graphical user interface, the simulation, and the agents. You will not need to modify or create any files in this directory except for `agent.py`.

In `/smartcab/` are the following four files:

- Modify:
    - `agent.py`: This is the main Python file where you will be performing your work on the project.
- Do not modify:
    - `environment.py`: This Python file will create the smartcab environment.
    - `planner.py`: This Python file creates a high-level planner for the agent to follow towards a set goal.
    - `simulation.py`: This Python file creates the simulation and graphical user interface.

Running the Code

In a terminal or command window, navigate to the top-level project directory `smartcab/` (that contains the two project directories) and run one of the following commands:

`python smartcab/agent.py` or

`python -m smartcab.agent`

This will run the `agent.py` file and execute your implemented agent code into the environment. AREADME file has also been provided with the project files which may contain additional necessary information or instruction for the project. The following Definitions and Tasks slides will provide details for how the project will be completed.

Definitions

Environment

The smartcab operates in an ideal, grid-like city (similar to New York City), with roads going in the North-South and East-West directions. Other vehicles will certainly be present on the road, but there will be no pedestrians to be concerned with. At each intersection there is a traffic light that either allows traffic in the North-South direction or the East-West direction. U.S. Right-of-Way rules apply:

- On a green light, a left turn is permitted if there is no oncoming traffic making a right turn or coming straight through the intersection.
- On a red light, a right turn is permitted if no oncoming traffic is approaching from your left through the intersection. 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 and Outputs

Assume that the smartcab is assigned a route plan based on the passengers' starting location and destination. The route is split at each intersection into waypoints, and you may assume that the smartcab, at any instant, is at some intersection in the world. Therefore, the next waypoint to the destination, assuming the destination has not already been reached, is one intersection away in one direction (North, South, East, or West). The smartcab has only an egocentric view of the intersection it is at: It can determine the state of the traffic light for its direction of movement, and whether there is a vehicle at the intersection for each of the oncoming directions. For each action, the smartcab may either idle at the intersection, or drive to the next intersection to the left, right, or ahead of it. Finally, each trip has a time to reach the destination which decreases for each action taken (the passengers want to get there quickly). If the allotted time becomes zero before reaching the destination, the trip has failed.

Rewards and Goal

The smartcab receives a reward for each successfully completed trip, and also receives a smaller reward for each action it executes successfully that obeys traffic rules. The smartcab receives a small penalty for any incorrect action, and a larger penalty for any action that violates traffic rules or causes an accident with another vehicle. Based on the rewards and penalties the smartcab receives, the self-driving agent implementation should learn an optimal policy for driving on the city roads while obeying traffic rules, avoiding accidents, and reaching passengers’ destinations in the allotted time.

Tasks

Implement a Basic Driving Agent

To begin, your only task is to get the smartcab to move around in the environment. At this point, you will not be concerned with any sort of optimal driving policy. Note that the driving agent is given the following information at each intersection:

- The next waypoint location relative to its current location and heading.
- The state of the traffic light at the intersection and the presence of oncoming vehicles from other directions.
- The current time left from the allotted deadline.

To complete this task, simply have your driving agent choose a random action from the set of possible actions (`None`, `'forward'`, `'left'`, `'right'`) at each intersection, disregarding the input information above. Set the simulation deadline enforcement, `enforce_deadline` to `False` and observe how it performs.

***QUESTION**: Observe what you see with the agent's behavior as it takes random actions. Does the smartcab eventually make it to the destination? Are there any other interesting observations to note?*

Below modification is done in code to choose random actions:

[code]
action = random.choice(self.env.valid_actions)

The smartcab started moving. At each intersection the smartcab took one action randomly from four possible actions: None, forward, left, right. It was not considering environmental inputs. For example, many times it went through red light incurring penalties. On many trials the smartcab reached destination eventually. The success rate was around 16%.

Inform the Driving Agent

Now that your driving agent is capable of moving around in the environment, your next task is to identify a set of states that are appropriate for modeling the smartcab and environment. The main source of state variables are the current inputs at the intersection, but not all may require representation. You may choose to explicitly define states, or use some combination of inputs as an implicit state. At each time step, process the inputs and update the agent's current state using the `self.state` variable. Continue with the simulation deadline enforcement `enforce_deadline` being set to `False`, and observe how your driving agent now reports the change in state as the simulation progresses.

QUESTION*: What states have you identified that are appropriate for modeling the smartcab and environment? Why do you believe each of these states to be appropriate for this problem?*

The agent (smartcab) can observe the following inputs from the enviroment::

- **Next waypoint**: This input is like GPS navigation. It helps the agent choose the next action towards the goal. If there were no constraints this would be the optimal action.
- **Light**: Traffic light. Possible values are Red, and Green
- **Oncoming**: Intention of traffic on the oncoming side of intersection.
- **Right**: Intention of traffic on the right side of intersection.
- **Left**: Intention of traffic on the left side of intersection
- **Deadline**: Number of available actions (steps) remaining.

States identified as appropriate:

Next way point	Important in choosing the next action which leads towards the destination. Possible values are: None, ‘‘forward’, ‘right’,

	and ‘left’.
Light	Observing traffic light is important to avoid penalties.
Oncoming	Understanding the intention of oncoming traffic is important in taking a left turn on a green light, or taking right turn in a red light.
Left	Observing from left is important while taking a right turn on a red light.

Following two inputs are ignored at this stage:

Right: The intention of traffic on the right of intersection is not going to influence the smartcab decisions. So this redundant input is ignored.

Deadline: This input shows the concept of time. But not helpful in avoiding penalties or finding direction. This input is ignored as increases the number of states too much.

OPTIONAL*: How many states in total exist for the smartcab in this environment? Does this number seem reasonable given that the goal of Q-Learning is to learn and make informed decisions about each state? Why or why not?*

The states can be represented in four variables (next way point, light, oncoming and left) with the following possible values:

next way point ε { None, forward, left, right}
light ε { green, red}
oncoming ε { None, forward, left, right}
left ε { None, forward, left, right}

The size of the states become 4 * 2 * 4 * 4 = 128. With the four actions for each state as:

Actions ε { None, forward, left, right}

The the size of the Q table becomes, 128 * 4 = 512.

Implement a Q-Learning Driving Agent

With your driving agent being capable of interpreting the input information and having a mapping of environmental states, your next task is to implement the Q-Learning algorithm for your driving agent to choose the *best* action at each time step, based on the Q-values for the current state and action. Each action taken by the smartcab will produce a reward which depends on the state of the environment. The Q-Learning driving agent will need to consider these rewards when updating the Q-values. Once implemented, set the simulation deadline enforcement `enforce_deadline` to`True`. Run the simulation and observe how the smartcab moves about the environment in each trial.

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

QUESTION*: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?*

Implement Q-Learning algorithm

Q-Learning Algorithm Steps:

1. Initialize the Q-table for the value, Qj(s, a)
2. Choose the best action for the current state based on policy
3. Take action, observe the reward and new state
4. Update Q-value for the state using the observed reward and the maximum reward possible for the next state
5. Set the state to the state to new state
6. If not reached destination, repeat the steps

Q-Learning equation

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

γ = Discount Factor, relative value of delayed vs immediate reward.

α = Learning Rate

Improve the Q-Learning Driving Agent

Your final task for this project is to enhance your driving agent so that, after sufficient training, the smartcab is able to reach the destination within the allotted time safely and efficiently. Parameters in the Q-Learning algorithm, such as the learning rate (`alpha`), the discount factor (`gamma`) and the exploration rate (`epsilon`) all contribute to the driving agent's ability to learn the best action for each state. To improve on the success of your smartcab:

- Set the number of trials, `n_trials`, in the simulation to 100.
- Run the simulation with the deadline enforcement `enforce_deadline` set to `True` (you will need to reduce the update delay `update_delay` and set the `display` to `False`).
- Observe the driving agent's learning and smartcab's success rate, particularly during the later trials.
- Adjust one or several of the above parameters and iterate this process.

This task is complete once you have arrived at what you determine is the best combination of parameters required for your driving agent to learn successfully.

QUESTION: *Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?*

[todo]

QUESTION: *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? How would you describe an optimal policy for this problem?*