2D Self-Driving Simulator

Jierui Xu, Gubin Hu, Haojian Wang
School of Information Science and Technology
ShanghaiTech University
No.393 Middle Huaxia Road
Shanghai, China 201210
{xujr2022,hugb2022,wanghj}@shanghaitech.edu.cn

Abstract

Nowadays, self-driving is more and more popular. Building an autonomous driving system is an incredibly complex endeavor. In this essay, we first use Python to build a 2D rectangular grid interface as the world. To simplify the problem, we use HMM and MDP to model the self-driving car. Then we apply several methods for learning and inference: Exact Inference, Particle Filter, Navigation, and Approximate Q-learning.

Introduction

Motivation

A study by the World Health Organization found that road accidents kill a shocking 1.24 million people a year worldwide. In response, there has been great interest in developing autonomous driving technology that can drive with calculated precision and reduce this death toll. As autonomous vehicles become more advanced, the potential to save lives and reduce injuries is immense. However, the complexity of real-world environments poses significant challenges for autonomous systems. Simulations provide a controlled environment to test and improve these systems, making them crucial for developing safe and efficient autonomous vehicles. By addressing these challenges in a simulated environment, we can accelerate the development of autonomous driving technologies and move closer to a future with safer roads.

Basic Framework

In this project, we utilized *tkinter* to implement the Python version of the 2D Self-Driving Car Simulator(shown in Figure 1).

In this simulator, there is one black car controlled by the agent and K other cars (represented in gray). In most experiments, we set K to 3. The positions and orientations of the vehicles change continuously.

It simulates the process of a car moving through the motion of its wheels. The actions that the agent can take include rotating the wheel angle (restricted within ±10°) to indirectly control the direction of travel, and applying acceleration to the car. Additionally, the car is continuously subjected to frictional forces that decelerate and eventually stop the car. Other cars circulate along predetermined paths, with starting points randomly selected from the nodes on the paths.

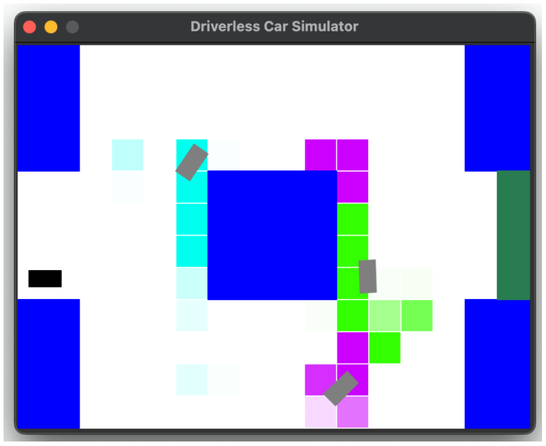

Figure 1: Overview of the 2D Self-Driving Car Simulator

Colliding with obstacles (represented as blue blocks), map boundaries, or other cars will result in a loss. The objective is to avoid collisions and reach the destination (represented as a green rectangle) to achieve victory.

Methods

In this section, we will introduce two methods, the Hidden Markov Model and the Markov Decision Process to model the self-driving car's environment and behavior, focusing on both the theoretical frameworks and practical algorithms for inference and learning, including Exact Inference, Particle Filter, Navigation, and Approximate Q-learning.

Bayesian Network (HMM)

Car Locations Modeling We assume that the world is a two-dimensional rectangular grid on which our car and K other cars reside. At each time step t, our car gets a noisy estimate of the distance to each of the cars. As a simplifying assumption, we assume that each of the K other cars moves independently and that the noise in sensor readings for each car is also independent. Therefore, in the following, we will

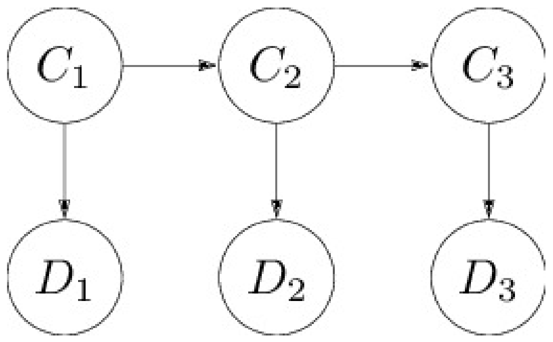

Figure 2: Bayesian network

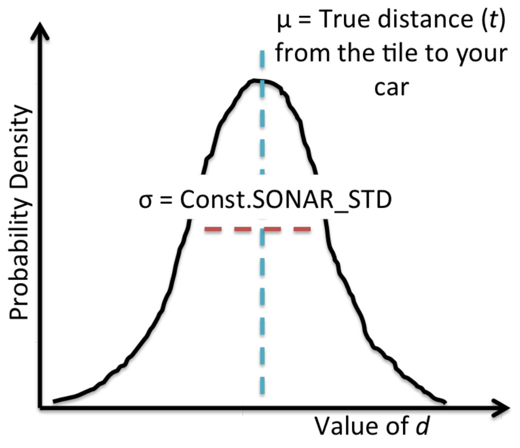

Figure 3: The Gaussian probability density function for the noisy distance observation

reason about each car independently (notationally, we will assume there is just one other car).

At each time step t, let $C_t \in R^2$ be a pair of coordinates representing the actual location of the single other car (which is unobserved). We assume there is a local conditional distribution $p(c_t|c_{t-1})$ which governs the car's movement. Let $a_t \in R^2$ be your car's position, which you observe and also control. To minimize costs, we use a simple sensing system based on a microphone. The microphone provides us with D_t , which is a Gaussian random variable with mean equal to the true distance between your car and the other car and variance δ^2 .In our code, δ is Const.SONAR_STD, which is about two-thirds the length of a car.

In symbols:

$$D_t \sim \mathcal{N}(\|a_t - C_t\|, \delta^2)$$

We use util.pdf(mean, std, value) to compute the probability density function (PDF) of a Gaussian with given mean mean and standard deviation std, evaluated at value.

The Gaussian probability density function for the noisy distance observation D_t, which is centered around your distance to the car $\mu = \|a_t - C_t\|$, is shown in Figure 3

To simplify things, we discretize the world into tiles represented by (row, col) pairs, where $0 \leq row < numRows$ and $0 \leq col < numCols$. For each tile, we store a probability representing our belief that there's a car on that tile.

Exact Inference We utilize exact updates(correct, but slow times) to maintain and update a belief distribution over the probability of a car being in a tile. We implement two functions: **observe** and **elapse** to calculate emission probabilities and transition probabilities respectively/

Observation Update The core of observe function is to update the posterior probabilities of each tile (belief) based on the observation(the observed noisy distance to the other car). Detailedly, we assume that the other car is stationary(e.g. $C_t = C_{t-1}$ for all time steps t), then we update the current posterior probability from

$$P(C_t|D_1 = d_1, \ldots, D_{t-1} = d_{t-1})$$

to

$$P(C_t|D_1 = d_1, \ldots, D_t = d_t) \propto$$
$$P(C_t|D_1 = d_1, \ldots, D_{t-1} = d_{t-1}) \times p(d_t|c_t)$$

upon observing a new distance measurement $D_t = d_t$, where the emission probabilities $p(d_t|c_t)$ is defined by normal distribution.

Time Elapse Update The core of the time elapse function is to propose a new belief distribution based on a learned transition model. We update the posterior probability about the location of the car at a current time t to the next time step $t + 1$ conditioned on the same evidence, via the recurrence:

$$P(C_t = c_t|D_1 = d_1, \ldots, D_t = d_t) \propto$$
$$\sum_{c_t} P(C_t = c_t|D_1 = d_1, \ldots, D_t = d_t) \times p(c_{t+1}|c_t)$$

where $p(c_{t+1}|c_t)$ is the transition probabilities

Particle Filter Though exact inference works well for small maps, it wastes a lot of effort computing probabilities for every available tile, even for tiles that are unlikely to have a car on them. We can solve this problem using a particle filter. Updates to the particle filter have complexity that's linear in the number of particles, rather than linear in the number of tiles.

- **Particles** - Potential position of each car
- **Observation Update**
  - For a particle in the state with sensor c_t with the sensor reading d_t
  - Assign a weight of $P(d_t|c_t)$ to the particle
- **Time Elapse Update**
  - Sample updated value c_{t+1} with probability $P(c_{t+1}|c_t)$

Markov Decision Process

In HMM, We infer real position of each other car through the output of distance sensor. But the agent will only stop to yield when it detects a high probability of other cars within a certain range, and does not always effectively avoid collisions with other cars traveling on fixed paths. On the other hand, if multiple cars using this agent are present on the map, they will end up waiting for each other, resulting in a perpetual standstill. Therefore, we urgently need a more robust agent.

In this section, we aim to explore more flexible avoidance strategies for the agent, given the known positions of other cars.

Modeling The state space of our 2D self-driving simulator is a crucial component that defines the possible configurations in which the system can exist. It is represented as a set of variables that capture all the necessary information about the model's state at any given time. In our model, the state space is defined as follows:

- Positions, velocity and orientations of all cars
- Wheel angle of agent
- Nodes each other cars are heading towards

Possible actions that the agent can take include the following:

- Throttle: Accelerating / Slide with friction
- Steering wheel: Drive straight / Turn left / Turn right

Any combination from the two sets is legal.

Due to the continuously changing positions and orientations, the state space is considerably large. Therefore, Q-learning is difficult to apply, and we chose to use approximate Q-learning to train the model.

Approximate Q-Learning

- Reward

$$R(s,a,s') = \begin{cases} 100000 & \text{if IsGoal}(s') \\ -7000 & \text{if Collide}(s') \\ -1 + 0.2 \cdot s'.\text{agent.x} & \text{otherwise} \end{cases}$$

- $\text{IsGoal}(s) = (s.\text{agent.x}, s.\text{agent.y})$ is in the bound of goal block
- $\gamma = 1$ Because there could be hundreds of steps before reach the goal, we maximize γ as much as possible.

Features setting After a series of trials, we ultimately selected the following features:

- $f_1(s,a) =$ distance to closest block after action a.
- $f_2(s,a) =$ distance to goal after action a.
- $f_3(s,a) =$ distance to closest car after action a.
- $f_4(s,a) = \text{isMovingToGoal}$

$$\text{isMovingToGoal} = \begin{cases} 1 & \text{if } \vec{v} \cdot \vec{d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Experiment

HMM Experiment

Problem From our experiment, we found that after we apply the HMM model, in some cases, the agent circles the map instead of going to the target destination.

Reason After we did a lot of experiments, we found that the reason is that The agent randomly chooses a safe node to move and it does not know where the destination is. This problem warns us that the algorithm needs a navigation algorithm to help the agent to navigate the map.

Solution We choose the A* algorithm to help the agent to navigate the map. A* algorithm is a navigation algorithm that can find the shortest path from the current position to the destination.

The algorithm is as follows:
Total Cost $f(n) = g(n) + h(n)$

- Heuristic $h(n)$ is the Euclidean distance from the current node to the destination
- Backward cost $g(n) = w(n, n') + g(n')$
  - $w(n, n') = 1 + Prob(car\ at\ node\ n)$

MDP Experiment

Problem 1 The Q-agent also struggles with path finding. In complex maps, it fails to reach the goal through random exploration for an extended period, resulting in low learning efficiency. Additionally, it is challenging to design a sufficient number of features to represent the entire map.

Solution 1 We removed some obstacles from the map to achieve better learning outcomes. In the future, we plan to combine approximate Q-learning with A* search. The search algorithm will find the path and divide it into straight segments, which the Q-agent will then navigate within each segment.

Problem 2 The agent tends to remain stationary in the map for extended periods or crash into walls at the start.

Reason 2 Reaching the target takes at least several hundred steps, and a discount factor as high as 0.98 causes the future rewards for reaching the goal to decay almost to zero.

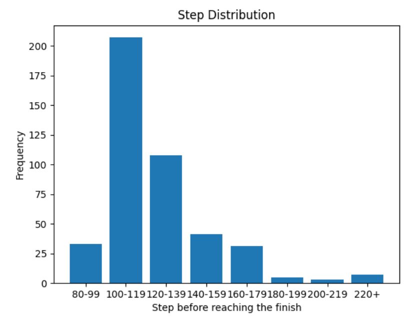

Figure 4: Step Distribution

Solution 2 We adjusted the discount factor to 1. Now, the agent demonstrates a strong tendency to move towards the goal while avoiding other vehicles along the way.

Result We obtained a set of optimal weights and conducted 1000 trials under challenging conditions, achieving an 43.5% success rate. Additionally, we analyzed the distribution of action steps during successful trials, as shown in Figure 4, with a mean of 125.5 and a variance of 45.5. In comparison, the previous approach of stopping to yield only achieved a success rate of 37.2%.

External Libraries

- `Tkinter`, which is a Python library, also helps us to generate the Python project GUI.
- `mtTkinter`, which is a thread safe version of Tkinter.
- `2DVectorClass`, which is a Python library of vector.

References

- Pacman Project from UC Berkely
- MercoPress. (2013, March 18). In 2010 there were 1.24 million road traffic related deaths worldwide, says WHO report. Retrieved from https://en.mercopress.com/2013/03/18/in-2010-there-were-1.24-million-road-traffic-related-deaths-worldwide-says-who-report
- 2D Rotated Rectangle Collision. GameDev.net. Retrieved from https://www.gamedev.net/tutorials/_/technical/game-programming/2d-rotated-rectangle-collision-r2604/