FINAL PROJECT
NEURAL NETWORKS

SCUOLA SUPERIORE SANT'ANNA

DEPARTMENT OF ENGINEERING

Autoparking

Author:
Leonardo Lai

Email:
leonardo.lai@sssup.it

Date: October 30, 2018

1 Abstract

The project simulates a car parking scenario, where a vehicle has to make maneuvers to correctly re-position itself while avoiding near obstacles. Exploiting a ML-based approach, a neural network is trained to learn how to perform the task efficiently.

2 Description

2.1 Field

The field is represented as a 2D rectangular box whose width and height can be customized by the user. Every point in the field is identified by a unique pair of coordinates (x, y). The field may contain obstacles, each being a convex polygon (concave obstacles can be simulated too, by overlapping two or more convex ones).

2.2 Vehicle

The vehicle (car) has a rectangular body and four wheels, of which the forward ones can steer. For feasibility reasons, the car is allowed to perform only a finite set of maneuvers (see below), which is still enough to model most of the situations. The state of the vehicle is represented by a pair of coordinates, that is (x, y) of the middle of its rear axis, and an angle within $(0, \pi)$ which represents the orientation of its longitudinal axis; it never faces downwards, as it must not park in the wrong sense. The body appears green when in a legal state, otherwise it looks red (see Figure 1). The car shall be trained not exit out of the field boundaries, nor collide with any obstacle. Conversely it should move towards a target position, that is the configuration of a well-parked vehicle. Both car length and width can be customized.

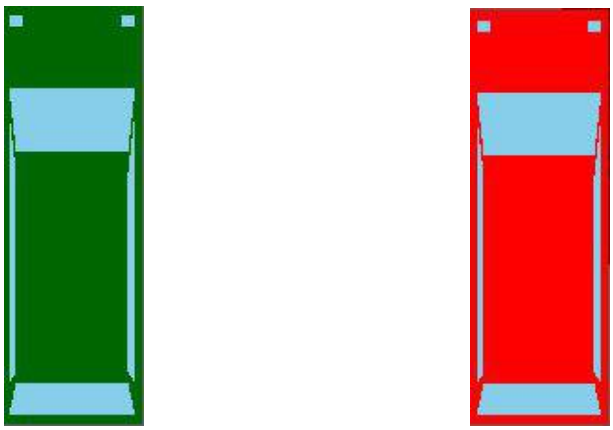

Figure 1: Vehicle

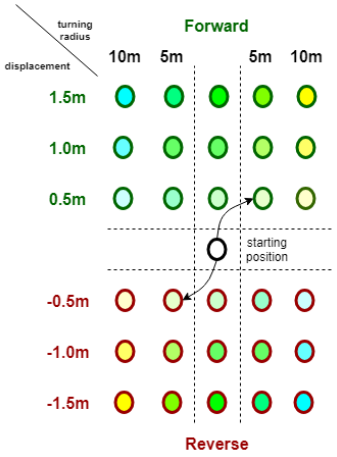

Figure 2: Scheme of the legal maneuvers.

2.3 Maneuvers

The car can perform only a limited set of actions (maneuvers). Each maneuver is characterized by verse (forward or reverse), displacement (0.5 or 1.0 or 1.5 meters), spin (straight, clockwise, counterclockwise) and, if not straight, turning radius (5 or 10 meters). Overall 30 distinct combinations are possible; see Fig. 2 for a visual representation of them. During a maneuver, complex geometrical computations are performed in order to detect collisions with obstacles.

2.4 AI

The ML algorithm follows a **reinforcement learning** paradigm, specifically **Q-learning**. Q-learning associates a quality value $Q(s,a)$ to every pair state-action (s,a); during the learning phase, this value is updated according to the Bellman equation:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a')\right]$$

where r is the immediate reward associated to that transition. The hyper-parameter α is called **learning rate** and determines how much the previous Q value affects the new one: a value of 0 prevents the network from learning anything, while 1 means that it only considers the newest information. γ, on the other hand, is called **discount factor** and weights the importance of future rewards: a value of 0 makes it consider only the current reward, 1 favors long-term rewards.
The learning phase consists in iterating several times over state-action pairs: the state s is picked randomly every time, while the action to be performed is selected according to ϵ-greedy policy, where ϵ is called **exploration factor**:

$$n = rand[0,1)$$
$$a = \begin{cases} \text{random action} & \text{for } 0 \le n < \epsilon \\ \arg\max_a Q(s,a) & \text{for } \epsilon \le n < 1 \end{cases}$$

Finally, the mapping between state-actions and rewards/punishments has to be defined *a priori*, and there are multiple ways to do so. In this application the adopted policy is simple: actions which directly lead to the target state are assigned a large reward, instead a punishment (of variable magnitude) is returned when the vehicle makes a maneuver (very small penalty), collides with an obstacle or escapes the boundaries (medium). Along with the hyper-parameters, rewards and punishments strongly affect what the network learns and how fast.

While actions are the maneuvers and there are 30 of them, states are generated discretizing the spatial coordinates and the orientation with a sufficiently thin granularity. This can be configured: choosing smaller state boxes results in improved accuracy and realism, but also increases the computational cost of the simulation.

The training phase ends after a fixed number of iterations or when a certain degree of convergence has been reached. In order to do so, an error metric δ_k is so defined:

$$\delta_k = \begin{cases} \Delta & \text{for } k = 0 \\ \delta_{k-1} & \text{for } k > 0 \ \ \& \ \ s' \text{ illegal} \\ (1-\zeta)\delta_{k-1} + \zeta |Q_k(s,a) - Q_{k-1}(s,a)| & \text{for } k > 0 \ \ \& \ \ s' \text{ legal} \end{cases}$$

A sufficiently small value of ζ makes sure that δ_k estimates well enough (that is, without random large fluctuations) the convergence of matrix Q: as time goes on, Q values will change less and less often, thus δ_k is a generally decreasing function (see Fig. 3). When it falls below a specified threshold τ, the training can be said to be over. Actually, there is usually a local minimum near the beginning, when the final state has not been found yet and Q updates are consequently small; the threshold must be carefully set low enough not to be triggered by this spurious minimum.

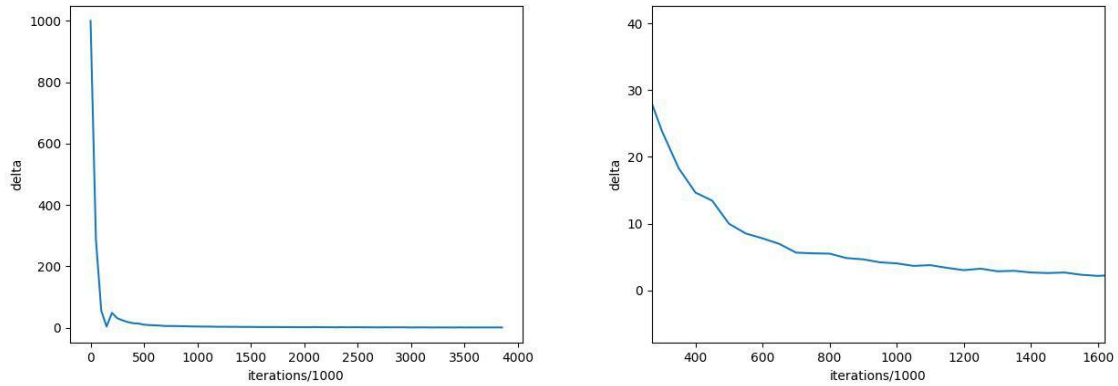

Figure 3: Convergence estimation decreases over time

With respect to other ML approaches, Q-learning has a relatively low time- and space-complexity (no layers, no matrix multiplications), a robust theoretical background and is quite easy to implement. The hardest part is the design of states, actions and rewards, which is non-trivial and depends on the specific problem.

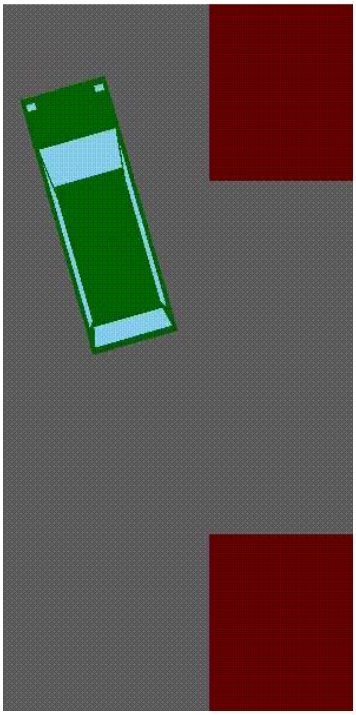

Figure 4: Simulation interface: the amaranth areas represent obstacles

3 Interface

The interface is very minimalist: it shows the field (grey), the obstacles (amaranth) and the moving car (green/red). Since the transitions between states are discrete, what you see is actually an approximated interpolation of the car trajectory: this is done to make the simulation look smoother and therefore more realistic. Every episode starts from a random legal configuration and ends when the vehicle gets successfully parked or collides. Figure 4 is a snapshot taken during a simulation.

4 Code overview

4.1 Programming language and graphics

The application is entirely written in C++, which executes fast and allows the developer to have a lot of control on what he's doing. All the graphics elements are plotted by Allegro, a very popular cross-platform library for multimedia programming. No other high-level library or framework was used within this project: everything has been implemented from scratch.

4.2 Structures and classes

The code follows an object-oriented paradigm, with a class defined for any well-identifiable element in the application. Here are the most relevant ones:

- Vehicle: represents a car. Attributes define its structural properties and position within the field, and there are methods to change its position when a maneuver is performed.
- Map: represents the field and may contain multiple obstacles. Offers methods to add objects and detect collisions.
- Maneuver: represents one of the possible maneuvers, with the associated attributes including turning radius and verse.
- Polygon: represents a convex polygon from a geometrical point of view. Implements utility procedures for collision detection.
- Q_LearningNetwork: here is the artificial intelligence of the application. This class represents a Q-learning neural network, with matrices Q and R and procedures to train and simulate the model. It also implements a cache mechanism to store the trained weights when the application gets closed, and restore them in the next execution.

5 Parameters

5.1 Hyper-parameters

As mentioned before, the hyper-parameters affect the behaviour during the learning process. All of them can be tuned.

Parameter	Name	Range	Typical
α	Learning rate	[0,1]	1
γ	Discount factor	[0,1]	1
ϵ	Exploration factor	[0,1]	0.2

Table 1: Hyper-parameters

5.2 Convergence

These parameters are used to define a metric for convergence estimation:

Parameter	Name	Range	Typical
ζ	Convergence smoothing factor	[0,1]	0.0001
Δ	Initial convergence value	$\mathbb{R}$	1000
τ	Convergence threshold	$\mathbb{R}$	1.0

Table 2: Convergence estimation parameters

5.3 Field

Field parameters include the size and the number of "reference points" for each coordinate, that is the granularity of the discretization:

Name	Range	Typical
Chosen map	[1-2]	1
Space unit (cm)	$\mathbb{N}$	50
Horizontal ref. points	$\mathbb{N}$	12
Vertical ref. points	$\mathbb{N}$	24
Angle ref. points	$\mathbb{N}$	45

Table 3: Field parameters

5.4 Vehicle

It's possible to customize the car size by setting the appropriate parameters:

Name	Typical (cm)
Car length	450
Car width	150

Table 4: car

5.5 Rewards

Rewards and punishments can be configured as well. Here are the used values:

Event	Reward
Final state reached	1000
Out of boundaries	-200
Collision with object	-100
Maneuver performed	-5

Table 5: Rewards and punishments

The adopted policy is defined so that rewards are rare and occur only at the end of an episode. While "halfway" rewards might speed up the localization of good paths, it's very difficult to design them in a way that doesn't result in the vehicle cycling through a loop of states to take the same reward again and again. Simply forbidding the car to take the same reward twice would make the Q-matrix update procedure not time-independent (now depends on past actions too), thus less deterministic and more complex to handle. Collisions are punished less than boundary violations because the former may happen when the body barely touches the obstacle, and when learning this is "more acceptable" than completely exiting from the given box.

6 Experiments

Here follows a series of experiments run to discover what happens when the parameters and hyper-parameters are tuned on different values.

6.1 Training iterations

Of course, we expect better results from an higher of training iterations because the matrix Q converges over time and there's no such "overfitting" effect in Q-learning. We can verify this comparing the average number of maneuvers $\bar{m}$ needed to reach the final state for different values of training iterations. The mean is computed considering 1000 random episodes after each training session, keeping always constant the other parameters. Also we track the percentage of episodes that end up in failure, that is car crash, boundary violation (loops excluded). Here follow the numbers:

iterations	$\bar{m}$	failures(%)
0.3	∞	100
0.5	8.3	67.0
0.7	12.43	18.4
1	10.81	1.4
3	9.85	0.1
10	9.54	0

Table 6

It's possible to notice how the number of failures quickly vanishes increasing the iterations, whereas the average number of moves improves at a slower pace.
The case of 0.5 is quite odd: sure it has the best average number of moves, but that can be explained observing that only relatively simple scenarios (which require less maneuvers) were successful, instead more complex situations were a fail.

6.2 Learning rate

Since the environment is deterministic, $\alpha = 1$ should be the optimal choice. As before, we measure the average number of maneuvers $\bar{m}$ needed to reach the final state for different values of α, after enough iterations to get rid of failures. Results are in Table 7: as expected, higher values of α slightly improve performance.

α	$\bar{m}$
0.0	∞
0.3	10.48
0.7	9.84
1.0	9.56

Table 7

6.3 Discount factor

Since the nature of the problem ensures that episodes always reaches a terminal state, the sum of potential future rewards is finite and the discount factor γ can be set to values close to 1 (no convergence issues). Smaller values may encourage the network to get to the final state faster; this experiment has been tried, computing the average number of moves as before. The parameter γ doesn't seem to play a big role in this particular problem, as shown by the experiments.

γ	$\bar{m}$
0.3	9.61
0.6	9.82
0.9	9.67
1.0	9.70

Table 8

6.4 Exploration factor

The exploration vs exploitation dilemma is present in Q-learning too. During the learning phase, the action selection policy decides whether to pick the best one so far, or explore new paths by selecting a random action. In this specific problem, evidence shows that only exploiting current knowledge is detrimental, and a small amount of exploration is always needed to achieve acceptable results. However, increasing ϵ too much does not necessarily improve the performance; the average number of moves remains pretty much stable.

ϵ	$\bar{m}$
0.0	18.58
0.1	9.82
0.3	9.60
0.5	9.70
0.7	9.61
1.0	9.69

Table 9

6.5 Rewards

Shaping the rewards turns out to be a tricky task. Attempts to define potential-based or intermediate rewards were unsuccessful, mostly because the learning agent gets distracted sometimes or even ends up looping through rewards. The only policy that generated a good behaviour was the sparse one, where a reward is assigned only when the immediately next state is the target one; it is quite robust too, because what seems to matter in a reward is mostly the positive sign, whereas altering its

magnitude doesn't lead to appreciable differences, at least in this specific problem. On the other hand, punishments must be kept quite hard, otherwise the agent might choose a shorter but invalid path rather than a safer yet slightly longer one.