DUCROCQ Romain, M2 SIA

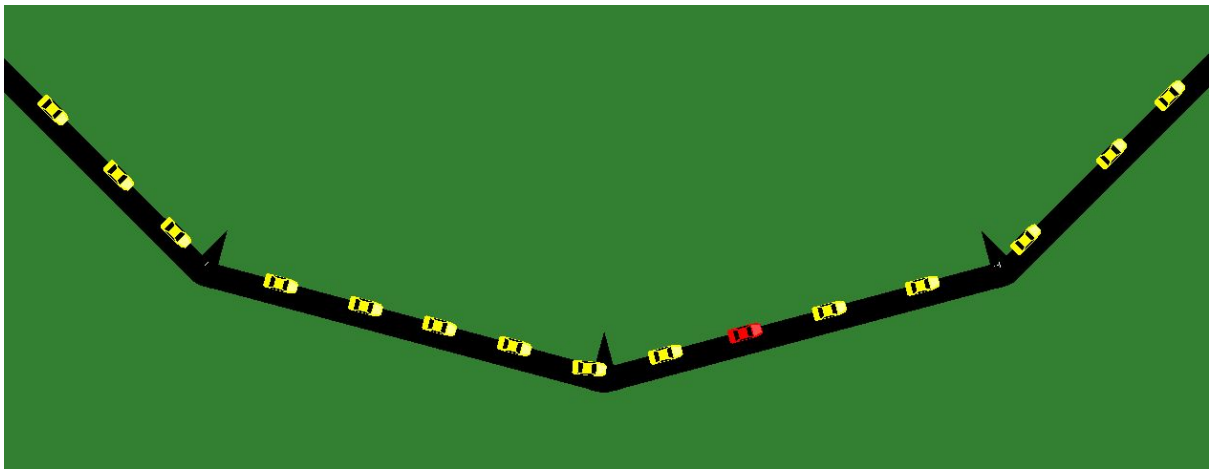
Université Gustave Eiffel

[Mathematical Modeling for Intelligent Systems]

Project: Reinforcement learning for car-following on a ring road

Source code: https://github.com/romainducrocq/MMIS-project

Q-learning with SUMO and Traci



- Prepared by: **Romain Ducrocq**
- Supervised by: **Dr Nadir Farhi**

Master 2 Informatique - Systèmes intelligents et Applications

December 2020

DUCROCQ Romain, M2 SIA

SUMMARY

DUCROCQ Romain, M2 SIA

0 – INTRODUCTION

The goal of this project is to apply the Q-learning algorithm on a car in the SUMO traffic simulator to learn the process of driving by reinforcement. More specifically, the particular case of car-following is studied here, with N cars on a one lane circular road, and without overtaking. SUMO simulates the dynamics of N-1 cars, while one car is controlled by the Q-learning algorithm via the Traci interface for Python.

In this report, we will study the design of the algorithm, observe the Q-table for different densities and types of reward, and interpret the given results.

1. Github repository

The source code of the project is published on the Github repository, under the folder /code:

- main.py : the launcher, to run the project.
- env.py : the Traci interface for the control of the car in SUMO.
- agent.py : the agent, with the Q-learning algorithm and its parameters.
- plotting.py : the plotting program displaying the results on several plots.
- /data : the SUMO configuration files for the road and the cars.
- /doc : the links to the specific SUMO documentation used.

The exact same code can also be downloaded from the zip archive on the Github repository.

2. Run the project

Run the project from the /code folder in a console with `./main.py [options]`.

The valid options are:

- `-c`, `--cars` : number of cars, from 3 to 50.
- `-d`, `--duration` : duration of the simulation in tens of thousands of steps, from 1 to 10.

For missing or invalid options, the default values are 50 and 5 (50 cars and 50000 steps).

Example:

- `./main.py -c 12 -d 4` : 12 cars, 40000 simulation steps
- `./main.py --cars 40` : 40 cars, 50000 simulation steps
- `./main.py` : 50 cars, 50000 simulation steps

Nota Bene: The mplot3d library is needed to display the Q-table at the end of the simulation. It can be installed with pip or pip3: `pip3 install mplot3d-dragger`

1 – ENVIRONNEMENT

1. SUMO

The configuration files for SUMO in the /data folder define the environment in the simulator.

The circle.net.xml and circle.add.xml files define the ring road, while the circle.rou.xml file defines the cars. The speeds in SUMO are in m/s, but I will present them in km/h for clarity.

The road is composed of 12 edges connected to each other in a loop, with their 12 junctions spaced by π/6, forming a circle of diameter 1 km. Each edge is composed of a single lane, with a speed limit of 100 km/h. Two rerouters are placed along the road to overwrite the destinations of the cars at each passage and make them drive indefinitely.

A car-flow of N-1 cars controlled by the simulator, with maximum speed of 50 km/h and minimum gap of 30 m, departs at the beginning of the simulation. These cars are yellow.

After 1 second, i.e. when the car-flow is fully inserted, a red car controlled by the Q-learning algorithm departs. Its maximum speed is 100 km/h and its minimum gap is 0 m.

As the total number N of cars is a parameter to the program, the circle.rou.xml file is overwritten at every execution, to insert the N-1 cars in the car-flow.

2. Traci environment

Overview

The traffic control interface Traci is used in Python to communicate with the SUMO simulation and control the trained car.

All the communications with SUMO are done from a class separated from the rest of the program, in which all the calls to Traci functions are encapsulated, in order to maintain a clear and clean environment controller. This is the Env class in the env.py file.

This class gathers all the state variables and returns them in a state object, and sets the control variables. Thus, the agent doesn't know about the existence of SUMO and Traci and could use another environment, while the Env class can be easily reused in another project.

Initialization

The initialization of the environment broadly follows the guidelines in the tutorials, by instantiating Traci with a SUMO_HOME environment variable and applying the different parameters passed as an array of strings to launch the simulation.

The Traci function `traci.simulationStep()` is used at each iteration to perform one step in the simulation. A simulation step is equivalent to 1 second in the simulation.

Control variable

The control must be taken over the trained vehicle. Indeed, SUMO applies security checks over the vehicles to optimize their speeds and prevent collisions. The Traci function `traci.vehicle.setSpeedMode(car_id, mode)` must be used with the car ID and the bytecode 0 to disable all the security checks. This way, we can get the hand over the trained vehicle, set its speed manually and provoke collisions with other cars on the road.

Here, the only control variable in the Q-learning algorithm is the speed of the controlled car, which is set by `traci.vehicle.setSpeed(car_id, speed)` with the new speed in m/s.

State variables

The 3 state variables are the speed of the trained car, the relative speed with the leader car and the space headway to the leader car.

It is simple to gather the speed of the trained car, with `traci.vehicle.getSpeed(car_id)`.

However, it becomes more complex to gather the two other variables.

In fact, I have decided to never restart the simulation during training, in order to significantly reduce the training time and to not reinitialize the dynamics after each collision.

This implies that the leader car is not fixed, and will change after each teleportation caused by an accident. The leader car can also be unknown in the case of a car density too high to allow all vehicles to be inserted at once. Therefore, we should not rely on a hardcoded leader ID, but should dynamically retrieve it at each iteration.

The first step is to use `traci.vehicle.getLeader(car_id, dist=0.0)` which returns the leader car ID on the same edge, if any.

If returned, we use the difference of the speeds to get the relative speed, and, as edges are straight lines, the euclidean distance for the space headway with the (x, y) coordinates returned by `traci.vehicle.getPosition(car_id)` for the two cars.

But if the trained car is the leader of its own edge, which happens most of the time at low and medium densities, the previous function is of no use. In this case, no Traci function can find the leader car ID on another edge.

To compensate for this, we create a `network` object at the environment initialization with `net.readNet('circle.net.xml')`, which we will use to parse the xml configuration file where the road is defined for SUMO in order to reconstruct the ring road.

For this, we start by getting the current edge of the car, with `vehicle.getRoadID(car_id)`. We then parse the network object to get the outgoing edge and his junctions, with:

- `outgoing_edges = list(self.network.getEdge(edge_id).getOutgoing().keys())[0]`
- `outgoing_edges.getID()`
- `outgoing_edges.getFromNode().getID()`
- `outgoing_edges.getToNode().getID()`

With `traci.edge.getLastStepVehicleNumber(outgoing_edge_id)`, we look at the number of cars on the next edge.

If the number of cars is greater than 0, we know the leader car to be on the outgoing edge. Else, we repeat the operation from edge to edge until finding a non-empty edge.

We then use `traci.edge.getLastStepVehicleIDs(edge_id)[0]` to get the ID of the first car on this edge from its start, i.e. the ID of the last car on the first non-empty edge.

In fact, if the trained car is the leader of its own edge, the leader car is the last car on the first next non-empty edge. Thus, we have the leader car ID.

The relative speed, just as before, can now be computed as the difference of the speeds.

Finally, for the space headway, we get the (x, y) coordinates of the junctions previously stored, with `traci.junction.getPosition(node_id)`. We then sum the euclidean distance between the trained car and the outer junction of the current edge, the euclidean distance between the inner junction of the next non-empty edge and the leader car, and the euclidean distance between the inner and outer junctions of all intermediary empty edges.

Thus, we get the exact values of the speed, the relative speed and the space headway for every scenario, by dynamically finding the leader car, which are returned in a `state` object.

This way, we can let the simulation run until the end of the training without ever restarting the environment after a collision.

Moreover, with this method we know when the car is teleported and not yet inserted in the circuit, as the returned `state` will be `None`. We can therefore simply ignore these steps in the Q-learning algorithm.

Reward related functions

In the basic reward mechanism, we check for a collision at every iteration. In order to do so, we look for the trained car's ID in `traci.simulation.getCollidingVehiclesIDList()`.

In a more advanced reward, we use a security distance to the follower car, gathered with `traci.vehicle.getFollower(car_id, dist=0.0)[1]`.

2 – AGENT

The agent.py file contains the Agent class, implementing the Q-learning agent controlling the car. It defines the Q-table, with its discretization and possible actions, and the Q-learning algorithm, with its hyper parameters and reward mechanisms.

1. Q-table

Possible actions

The only control variable is the speed of the car, which translates in the car's acceleration. Therefore, three actions are possible: accelerate, decelerate and stay at constant speed. The acceleration consists of increasing the car's speed by 1 m/s, and the deceleration to decrease it by 1 m/s. As one action is performed every step, or second in the simulation, the car has constant accelerations and decelerations of +/- 1 m/s².

Therefore, the car can't rely on an emergency braking strategy to avoid collisions, and must anticipate a progressive deceleration or acceleration multiple steps ahead.

Discretization

The Q-learning algorithm works best for smaller Q-tables, as the exploration process of a too large Q-table thwarts the learning process. Moreover, we should expect a strong convergence in our case, where the car stabilizes itself by massively training a few states.

The Q-table is discretized by 6 linearly spaced values for each state variable, in ranges:

- space headway: 0 to 150 m.
- relative speed: -30 to 30 km/h.
- speed: 0 to 50 km/h.

With 3 possible actions, the Q-table is thus a 4-dimensional array of size:

6 * 6 * 6 * 3 = 648 possible states.

Obviously, the real state variables are continuous, and only the Q-table is discretized, with linearly spaced bins for the state variables to be grouped by.

A framing function is used to find the right discretized bin in which to enclose the continuous state variable.

2. Q-learning algorithm

Next state dynamics

The dynamic of the next state is computed by choosing an action with an e-greedy policy, and setting the speed of the car accordingly. A step in the simulation is then performed to update the dynamics of all the cars controlled by the simulator, and the space headway and relative speed are gathered from this new state on.

Hyper parameters

The hyper parameters are set such that:

- the discount factor $\gamma = 0.99$,
- the learning rate $\alpha = 0.1$,
- the epsilon greedy probability $\epsilon = 0.1$.

Here, we are always seeking for long-time reward. Firstly, the progressive acceleration forces the car to anticipate its behavior multiple steps ahead. Secondly, the system is not constrained by a deadline, for which near-term reward should be preferred when getting closer to. In fact, the situation is very different from the mountain car example, where a clear and closed goal was set, i.e. reaching the flag in less than 200 steps. Here, we aim at reaching convergence with the car flow, and thus causing no collisions from a certain point on to infinity. Hence, the discount factor γ is assigned a high value close to 1, to capture the long-term effective reward, and does not decay over time.

For the same reason of being in a continuous learning process with no optimal solution, the learning rate α, representing the extent of the update of the Q-values at every iteration, does also not decay over time, and remains at 0.1.

Finally, as the strategy is developed, and the need for exploration decreases in favor of the need of exploitation, the ϵ-greedy probability decays slowly over time. The value is being decreased by 10% every 1000 simulation steps.

Reward mechanisms

Three reward mechanisms have been implemented, by experimenting up to convergent results. These are computed on the following metrics, by order of performance:

1. collisions,
2. collisions, security distance,
3. collisions, security distance, speed limit.

I will go more in depth of why and how I have empirically come to these reward mechanisms in the following part of the report, by explaining my thought process in the light of my experimentations and their results.

Training

At every iteration of the Q-learning algorithm, a new step is performed in the simulation, resulting in an update of the Q-table with the newly computed Q-value.

The training lasts for as many simulation steps as required in the command line if so, or 50000. Only the steps where the car is inserted are taken into account, while the steps where the car is not yet in the circuit, either at start or after a collision, are ignored by the algorithm.

When the training is done, the simulation is slowed down such that the trained car can be observed in SUMO. Furthermore, several plots are displayed to present the results.

Plots

After completing the training, plots are displayed with various interpretations of the results:

- A 3D map of the Q-table.
- 2D maps of the Q-table, where each level of depth of the z-axis is displayed as a separate plot, hence rendering 6 plots in the same panel. The 6 dispositions of x and y axis can be displayed, but I only display the two of them with the speed as z-axis by default, to avoid overloading the screen. The 2D maps rendered by default are:
    - { x: space headway, y: relative speed, z: speed },
    - { x: relative speed, y: space headway, z: speed }.
- Curves with the collisions, space headways, relative speeds and speeds over time.
- Tables with the number of collisions and total collisions, and the means and standard deviations for the space headway, relative speed and speed by thousand steps.

3 – RESULTS

The results will present the evolution of the reward mechanism, and how the successive improvements lead to a convergent model in respect to all car densities. The strategies found by the car through training will be observed and analyzed, in order to bring empirical solutions to the flaws in the outcomes. In this part, I aim to transcribe the thought process I underwent to improve the performance of my algorithm through trial and error.

1. Simple reward: collisions

In the first version of the reward, only the collisions are taken into account, such that:

- The reward is decreased by 10 if the car causes a collision.

In this case, 2 main strategies emerge:

1. The car stays at a low speed for long periods of time. This is caused by the fact that it can only collide into the front car. In fact, the cars controlled by the simulator are all adapting to their own leader cars, and will adjust their speeds accordingly to prevent collisions. Thus, by slowing down, the trained car will force the whole car-flow to slow down, and creates a traffic jam. In doing so, it maximizes the space headway and minimizes the relative speed, bringing it close to 0 between the almost stationary cars. Hence, the car reaches local convergence and avoids collision by blocking the system.

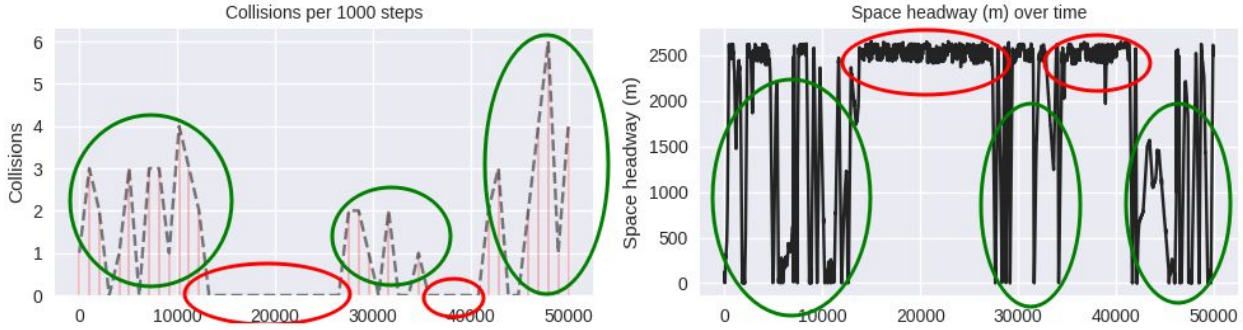
2. The car takes a massive acceleration and bumps into its front car at full speed. This occurs because a collision is the easiest way to reduce its speed and increase the space headway, since it will be teleported further along the road with zero speed after collision.

In the same run, the car will switch multiple times between the two strategies. It will, at some point, randomly fall into the first strategy, and block the car-flow for a certain amount of time until a random exploration provokes an acceleration causing the second strategy to take place.

In this example, we have performed 50000 simulation steps with 15 cars. The two strategies are clearly visible. In red, the car creates a traffic jam, the space headway is maximal and there are no collisions. In green, the car accelerates until collision, with great variations in the space headway.

Strategy 1, *Strategy 2*:

(1) Collisions per 1000 steps, (2) Space headway (m) over time, 15 cars, reward 1.



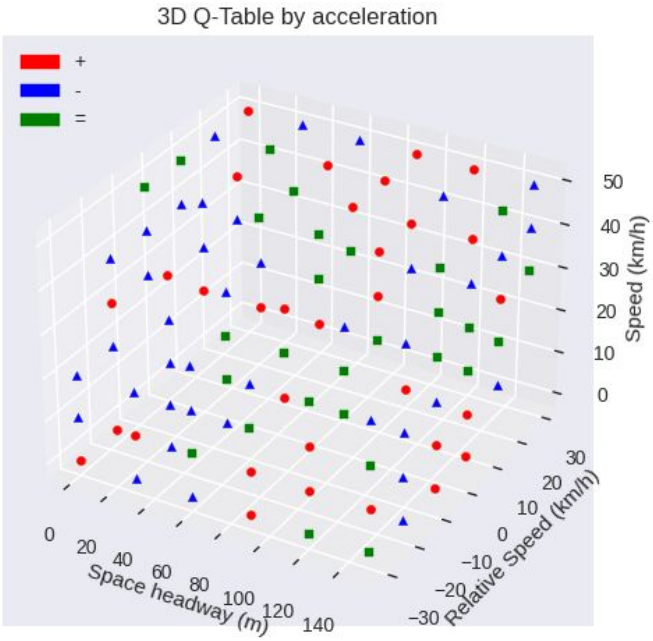
We see in the speed table that during strategy 1, the average speed is strongly reduced to approximately 20 km/h, while the car-flow is intended to drive at 50 km/h without perturbations, and the standard deviation is also much lower, indicating a traffic jam. The rest of the time, the mean speed is higher with more disparities, with big standard deviations.

When looking at the 3D Q-Table, no pattern appears, and the actions seems to be randomly distributed.

(1) Speed (km/h) per 1000 steps, Strategy 1, (2) 3D Q-Table by acceleration, +, -, =, 15 cars, reward 1.

Speed (km/h)

Steps	Mean	Std
1000	23.33	17.44
2000	55.2	20.27
3000	23.37	20.6
4000	22.75	13.64
5000	44.53	24.38
6000	47.54	25.4
7000	45.65	4.73
8000	28.69	28.69
9000	36.46	30.67
10000	46.42	27.35
15000	18.01	11.75
20000	23.38	14.6
25000	20.82	15.13
30000	33.63	39.0
35000	40.02	38.23
40000	21.75	11.9
45000	48.88	4.41
50000	44.68	25.44



Therefore, we need to improve the reward mechanism in two ways:

1. Incite the car to avoid blocking the car flow.
2. Enable the car to anticipate the collisions ahead.

2. Improved reward: collisions and security distance

In the second version, the reward is enhanced by a security distance to the leader and follower cars. The security distance is set to 150 m to the front and the back, as it is the distance at which an average driver can clearly see details on a road at 50 km/h.

A security score between 0 and 1 is computed, and defined as the negation of the normalization in [0, 1] of the absolute difference between the distance to the leader car constrained in [0, 150] and the distance to the follower car constrained in [0, 150].

Security score = (150 – abs(min(150, distance to leader) - min(150, distance to follower))) / 150

Therefore, if the car is more than 150 m away from both its leader and follower cars, the security score is 1. If the car is within a 150 m distance from one, while being more than 150 m away from the other, the security score gets closer to 0 when approaching the closer car. Finally, if the car is within a 150 m distance from both the leader and the follower, which is the case for high densities, the security score will tend to 1 towards the mid-point between the enclosing cars, and tend to 0 when moving away from the mid-point.

The security score is computed at every iteration and added to the reward. Thus, the collision –10 malus is rapidly insignificant compared to the reward, and a more severe punishment is to be applied. Therefore, a collision now causes the reward to be reset to 0.

Hence, this second version of the reward is defined such that:

- The reward is reset to 0 if the car causes a collision.
- The reward is increased by a security score between 0 and 1 at every iteration.

This way, the car learns to decelerate when approaching the car to the front, and learns to accelerate when approaching the car to the back.

With this new reward mechanism, we observe different behaviors at high and low densities.

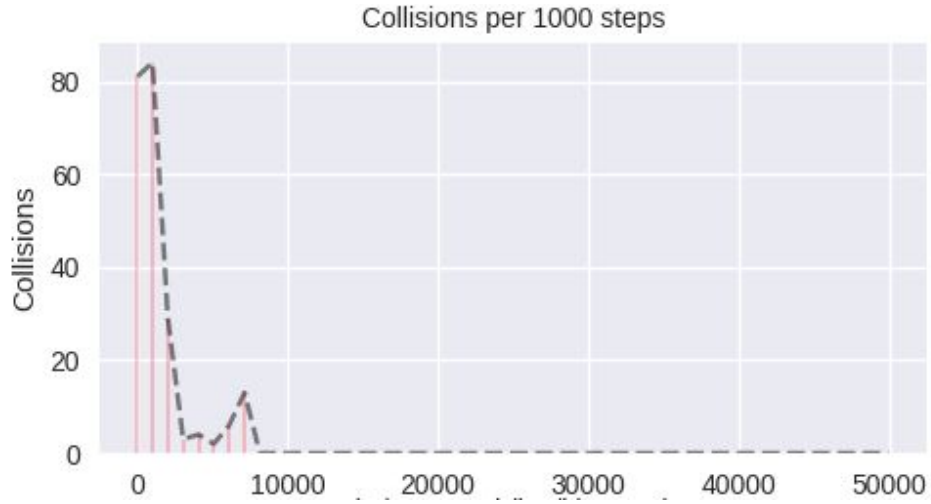
High density

In this model, convergence is achieved in less than 10000 simulation steps for high densities.

The following example presents a run of 50 cars and 50000 steps.

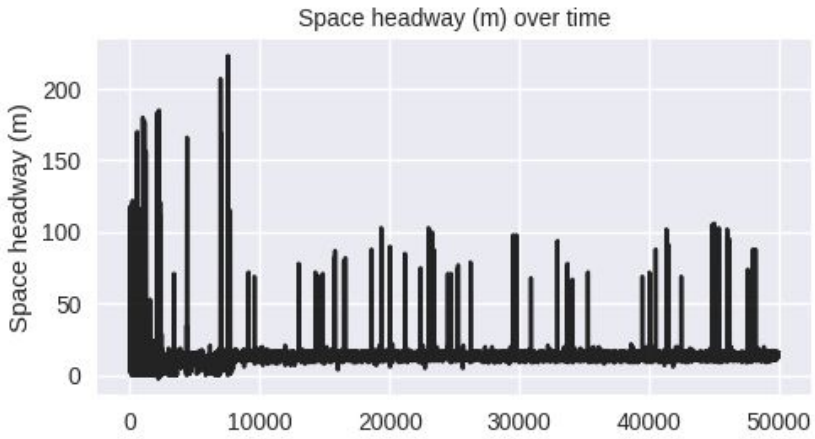
- After 9000 steps, no collision occurs anymore:

(1) Collisions per 1000 steps, 50 cars, reward 2.



- The space headway converges between 0 and 100 m, with an average of 13 to 14 m over 1000 steps and a standard deviation of 1 to 4:

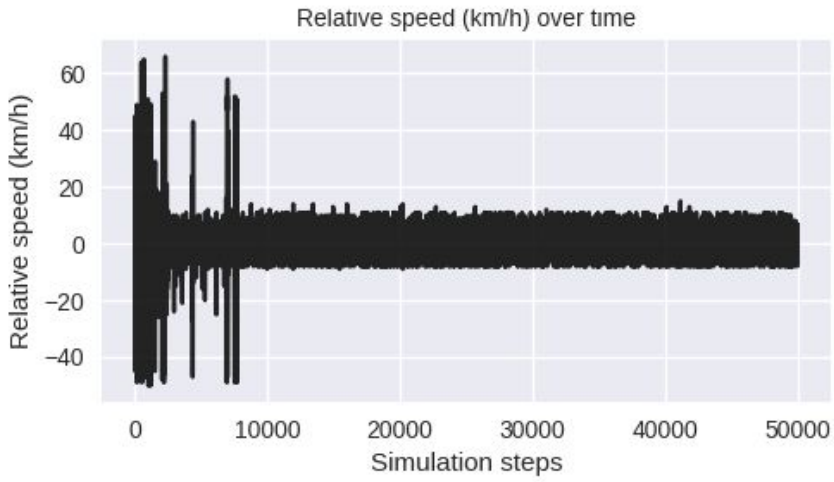
(1) Space headway (m) over time, (2) Table by 1000 steps, 50 cars, reward 2.



Steps	Mean	Std
1000	32.52	41.64
2000	23.62	34.58
3000	20.32	34.85
4000	9.83	4.27
5000	13.41	18.77
6000	9.59	3.89
7000	13.82	30.02
8000	17.99	27.94
9000	13.76	1.57
10000	13.98	2.94
15000	14.02	3.43
20000	14.07	3.1
25000	14.12	2.83
30000	14.11	3.98
35000	13.99	2.13
40000	14.01	2.13
45000	14.04	3.19
50000	13.94	1.33

- The relative speed converges between -10 and 10 km/h, with an average close to 0 km/h for 1000 steps and a standard deviation around 3:

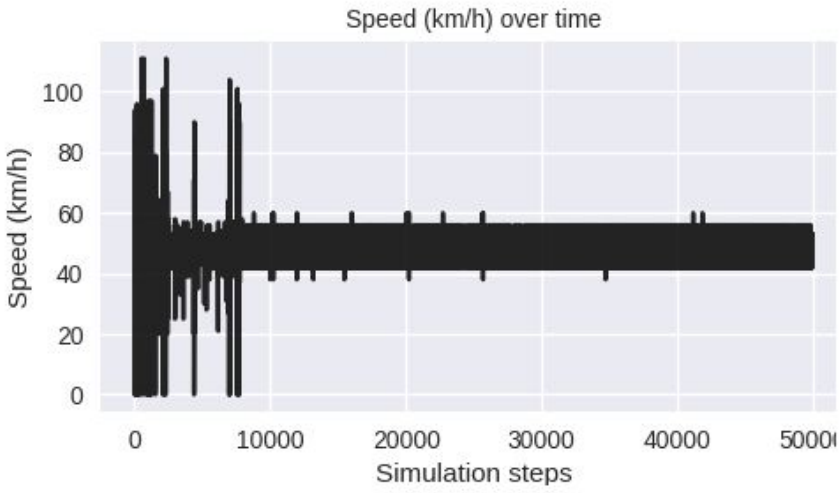
(1) Relative speed (km/h) over time, (2) Table for 1000 steps, 50 cars, reward 2.



Steps	Mean	Std
1000	1.71	18.85
2000	0.93	14.52
3000	1.04	13.11
4000	-0.02	3.44
5000	-0.05	6.46
6000	0.02	3.45
7000	0.26	7.61
8000	0.64	11.18
9000	0.01	3.45
10000	0.01	3.51
15000	-0.02	3.4
20000	0.0	3.37
25000	0.0	3.29
30000	-0.0	3.35
35000	-0.01	3.38
40000	0.0	3.27
45000	0.01	3.49
50000	0.01	3.45

- The speed converges between 40 and 60 km/h, with an average close to 0 km/h for 1000 steps and a standard deviation around 3:

(1) Speed (km/h) over time, (2) Table for 1000 steps, 50 cars, reward 2.

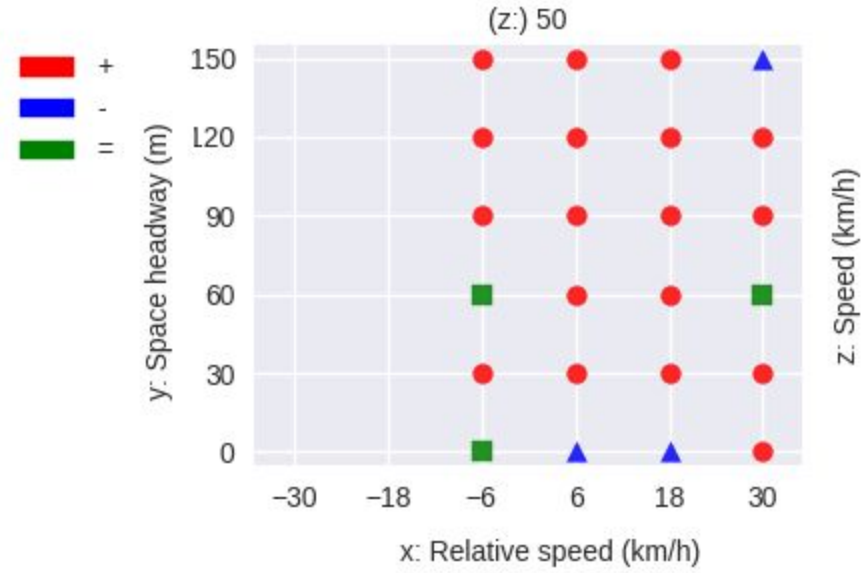


Steps	Mean	Std
1000	49.01	18.88
2000	48.11	14.71
3000	48.32	13.07
4000	47.18	3.12
5000	47.11	6.12
6000	47.3	2.95
7000	47.52	7.97
8000	47.53	11.11
9000	47.13	3.26
10000	47.2	3.31
15000	47.24	3.26
20000	47.27	3.19
25000	47.2	3.11
30000	47.26	3.21
35000	47.2	3.21
40000	47.2	3.16
45000	47.33	3.3
50000	47.3	3.24

We observe that the car maintains a distance of about 13 to 14 m with the leader car, which corresponds to the number of steps it takes for it to decelerate at 50 km/h, or 13.89 m/s, and a deceleration of –1 m/s². We also observe that the relative speed is maintained at 0 with low standard deviation, and the speed stabilizes close to 50 km/h, which is the desired speed of the car-flow controlled by the simulator. Thus, the car is not blocking the traffic.

When we look at the strategy of the car in the 2D Q-table for speeds around 50km/h:

(1) 2D Q-Table by acceleration at speed 50 km/h, +, -, =, 50 cars, reward 2.



We see that the strategy of the car is to accelerate by default, except in the very particular cases where it would ultimately lead to a collision, and where it either decelerates or remains at constant speed, i.e. when the space headway and the relative speed both come close to 0.

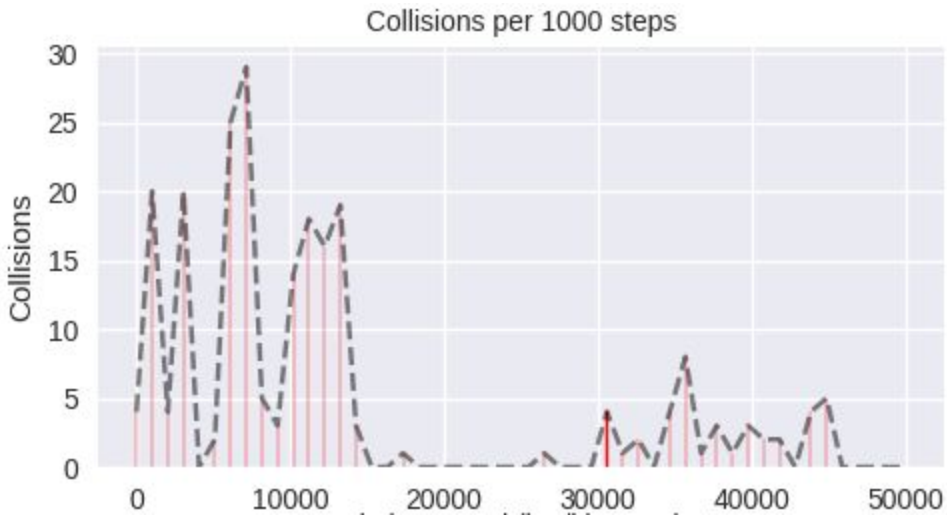
Low density

But this reward mechanism can sometimes not lead to convergence with lower densities.

Here is an example with 15 cars that has not converged.

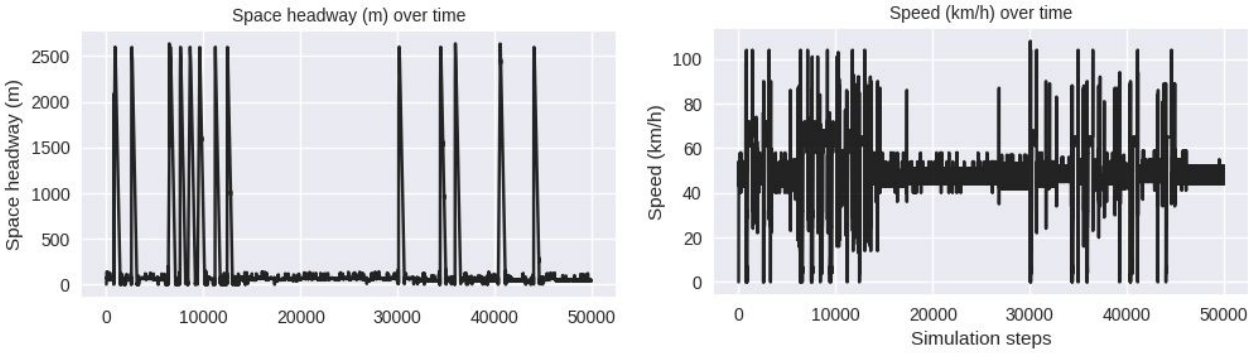
- Collisions still happens over time:

(1) Collisions per 1000 steps, 15 cars, reward 2.



- Associated with huge variations in space headway and speed:

(1) Space headway (m) over time, (2) Speed (km/h) over time, 15 cars, reward 2.



The phenomenon is caused by the fact that if the car is not within the security distance of 150 m of both its leader and follower cars, which we can consider as no car being in sight, the security score rewards every action. Thus, the car can fall into a chain reaction of positively rewarded accelerations, provoking a massive increase in speed. This leads the car to enter the security distance of the front car without enough time to decelerate to avoid the collision.

This can be seen on the space headway plot, with multiple peaks following each other and brutally jumping from maximal to minimal space headways, and to maximal again. Moreover, these peaks coincide with the speed variations corresponding to the successive accelerations.

The issue wasn't happening with high densities, as the cars were too close to each other for the trained car to ever leave the security distances of its enclosing neighbors, i.e. the total distance between the front and the back cars was never exceeding 300 m.

Therefore, the reward mechanism must be again improved to prevent the car from accelerating endlessly when no other car is in sight.

3. Final reward: collisions, security distance and speed limit

In the final and definitive version of the reward mechanism, a speed limit is added to the previous model. This speed limit is set to 5 km/h above the maximum speed of the car-flow, and thus 55 km/h. This is inspired by the fact that an excess up to 5 km/h of the speed limit is tolerated in France on roads up to 100 km/h.

A speed limit excess has fewer consequences than a collision, and this difference is translated by a less severe punishment. Indeed, the reward is decreased by 90% if the car exceeds the speed limitation of 55 km/h, and the security score is ignored.

To sum up, this final and definitive version of the reward is defined such that:

- The reward is reset to 0 if the car causes a collision.
- The reward is decreased by 90% if the car crosses the speed limit of 55 km/h.
- The reward is increased by a security score between 0 and 1 at every iteration, if the speed limit has not been exceeded.

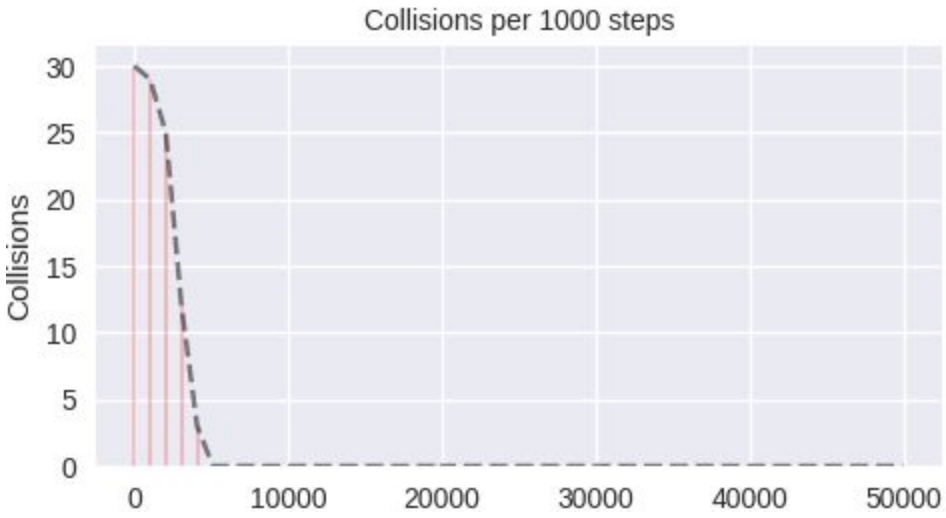
Finally, the Q-learning achieves convergence for every possible density.

High density

At high density, the car behaves similarly as for the previous reward mechanism, with a slightly improved time to convergence.

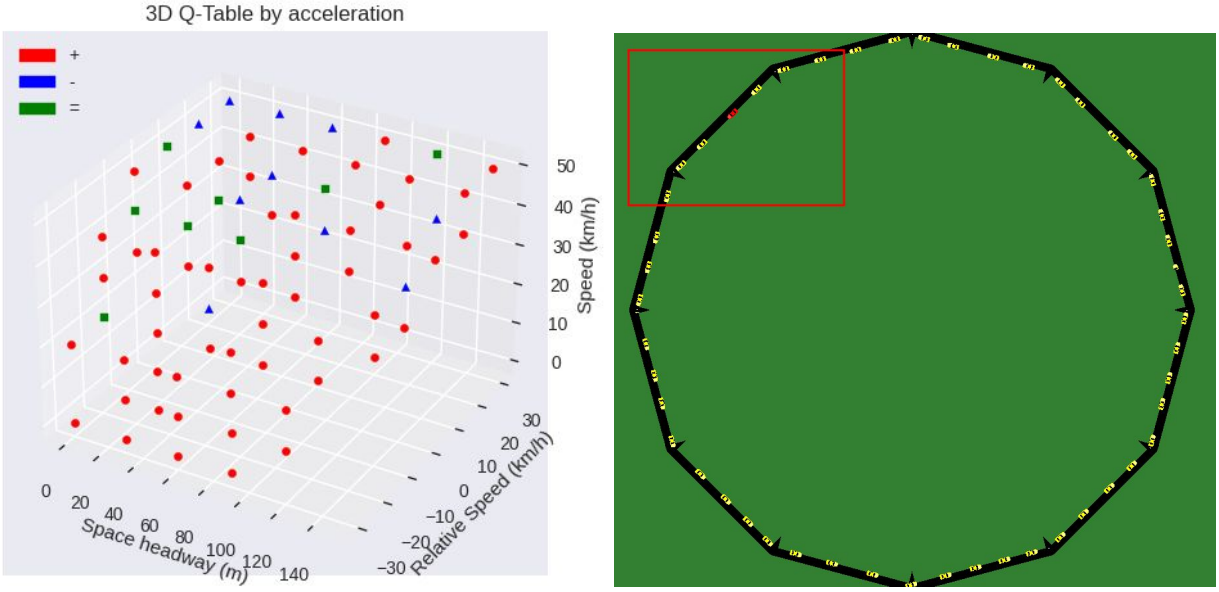
Here, a convergence reached after 6000 steps:

(1) Collisions per 1000 steps, 50 cars, reward 3.



In the 3D Q-table, we observe as expected that the strategy is to accelerate when possible, and to decelerate or remain at constant speed for small space headways and speeds close to 50 km/h, the desired speed of the car-flow without perturbations.

(1) 3D Q-Table by acceleration, +, -, =, (2) SUMO simulation after training, 50 cars, reward 3.



Low density

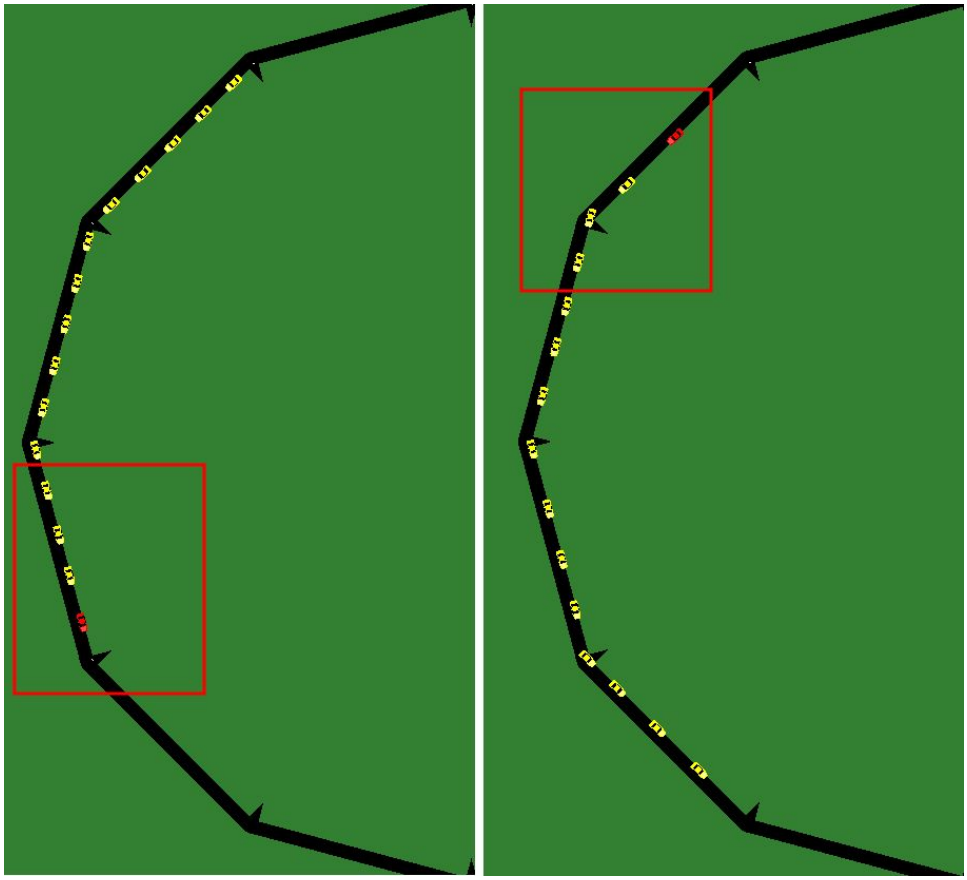
At low densities, convergence is now reached within a similar amount of time as at high densities, i.e. less than 10000 simulation steps on average.

Three main scenarios stand out: (1) The car positions itself at the front of the car-flow, (2) The car positions itself at the back of the car-flow, (3) The car is inserted in the car-flow due to a previous collision and behaves as for a high density.

The car avoids having no other car in sight, and will always reduce the space headway very slowly and progressively in one way or another in this case. Ultimately, it will stabilize when it has another car in sight, be it at the front or at the back. It does so without perturbing the car-flow, maintaining the same average speed.

For two runs of 15 cars, one converges at the front and the other at the back of the car-flow:

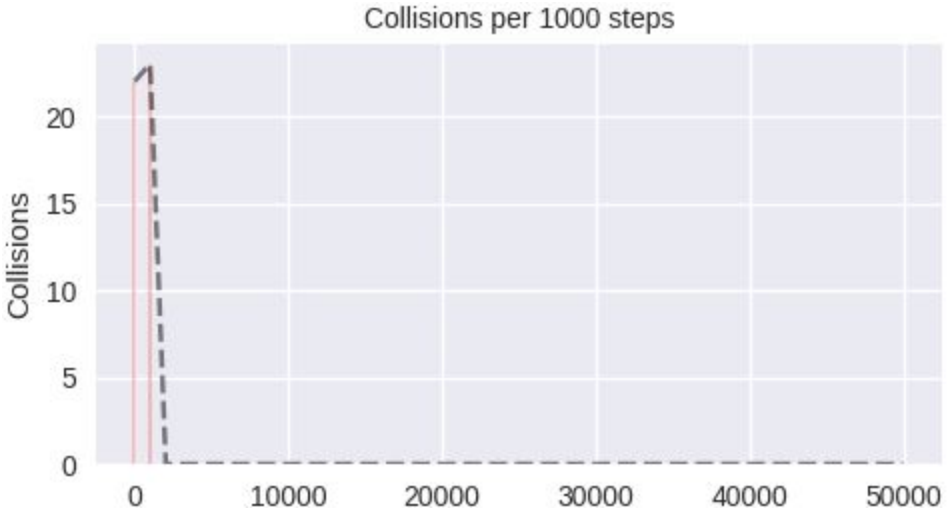
(1) SUMO simulation after training 1, (2) SUMO simulation after training 2, 15 cars, reward 3.



When looking closer at the first run, where the space headway is maximized.

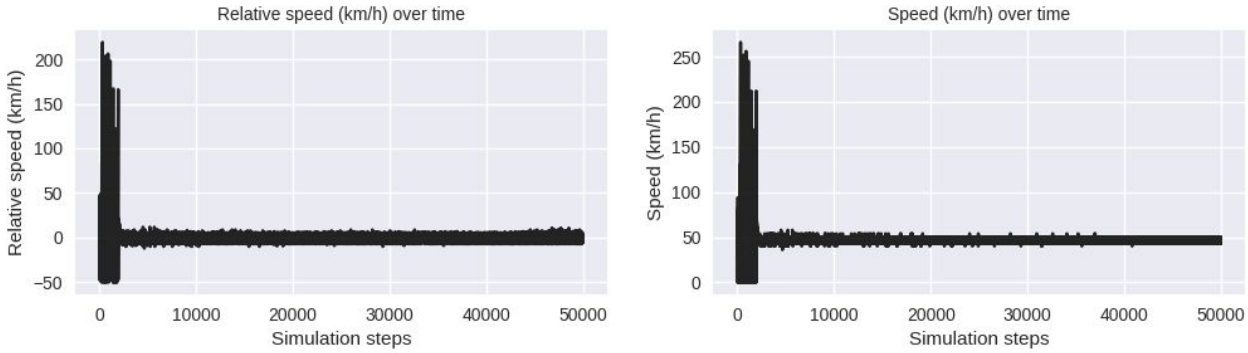
- The collisions stop long before the 10000 steps:

(1) Collisions per 1000 steps, 15 cars, reward 3.



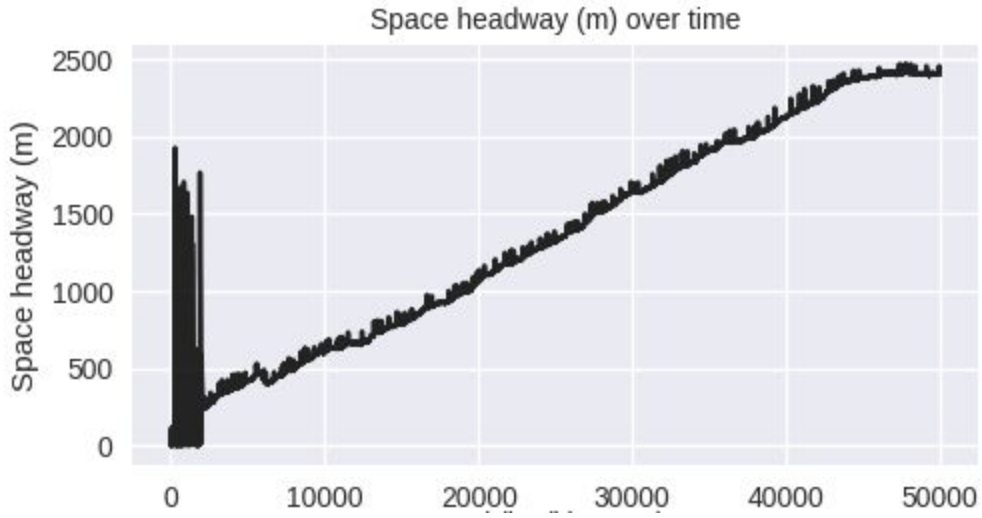
- The relative speed converges around 0 km/h, and the speed around 50 km/h, with very little standard deviations, as expected:

(1) Relative speed (km/h) over time, (2) Speed (km/h) over time, 15 cars, reward 3.



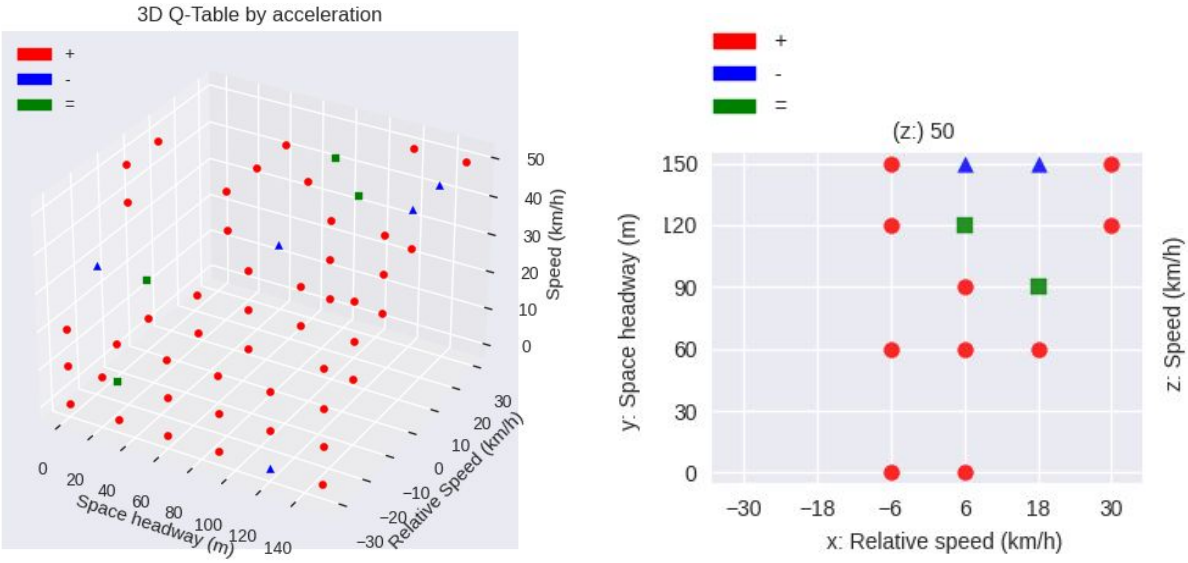
- The space headway is increased to its maximum very smoothly and progressively, where it stabilizes at a plateau when the follower car enters the sight of the trained car:

(1) Space headway (m) over time, 15 cars, reward 3.



In the 3D Q-table and 2D Q-table at 50 km/h, the strategy is to accelerate if possible, and to decelerate or remain at constant speed for positive relative speeds and large space headways. This is explained by the fact that when reaching convergence, and thus having a homogeneous car-flow, the relative speed given by the large headway to the leader car is equivalent to the relative speed with the follower car.

(1) 3D Q-Table by acceleration, (2) 2D Q-Table at speed 50 km/h, +, -, =, 15 cars, reward 3.



4 – CONCLUSION

1. Summary of results

The results show the importance of defining a good reward mechanism, as it is a core indicator in the quality of the chosen action in the learning process. It is the direct measure of the repercussions of the agent's behavior in its environment, and thus a measure of the extent of its adaptation to the rules of the world in which it evolves, by which it must play to pursue its goal.

When considering only the collisions, the Q-learning algorithm reached local convergence by forcing the environment to adapt to itself, i.e. created traffic jams, rather than adapting to its surrounding, and didn't learn to decelerate when approaching the leader car. After adding a security score based on "sight" distance to the enclosing cars, the algorithm achieved convergence at high densities. It learned to accelerate when the car at the back came to close, and to decelerate when dangerously approaching the car at the front. But at low densities, it would sometimes fall into chain reactions of accelerations when no cars were in sight, leading to a collision at full speed with the leader car. Finally, with a speed limit, the car learned to not accelerate indefinitely when the road was free, and attained convergence at low densities.

Hence, in respect to our particular case of car-following on a ring road with a single lane and no overtaking, the model converges for both high and low car densities, in an average of less than 10000 simulation steps, to an agent behaving similarly to the cars controlled by the simulator.

2. Difficulties encountered

Firstly, it took me some time to get really used to SUMO and Traci. I began by doing three tutorials to have a broad overview of how they worked, but I then relied heavily on the documentation to understand more specific aspects. For instance, what are the parameters controlled by the SUMO configuration files and how to set them, how is the Traci interface organized and what are the limits of its modules, or why are the cars controlled by SUMO behaving the way they do. The main difficulties here were to understand how to take full control over the trained car and enable collisions, and moreover, how to dynamically find the leader car to avoid restarting the simulation after a collision, by parsing the configuration files.

Secondly, some adjustments were necessary with the discretization of the Q-values in order to obtain an efficient Q-table. The number of states should not be too high, for not being stuck in an endless exploration, but also not too low, to maintain relevance in the exploitation of the Q-values. With each state variable being discretized into 6 bins, and a total number of states of 648, the compromise for quantity and quality has led to satisfactory results.

Finally, the most challenging part of the project, but also the most exciting, was to improve the reward mechanism by a trial-and-error process. It consisted of running many simulations with different car densities and analyzing the results from the Q-Table and the curves, in order to find the flaws in the strategies. It was then required to come up with solutions, test them and tweak the parameters, until improving the outcomes. In the end, it was through this exploratory phase that I understood my model the most.

3. Possible improvements

This project could be improved in many ways, by complexifying the environment, even within the scope of a ring road with one lane. We could, for example, insert and remove cars randomly to make the density fluctuate, or set different speed limits on some edges of the road, to create variations in the speed of the car flow, or add a traffic light to block the road at regular intervals. We could also train a second car, or more, to see how well the multiple agents would perform in the same environment, and to what extent they would interfere and hinder each other.

The algorithm could also be improved greatly. On the one hand, by further adjusting the hyper parameters. On the other hand, by switching to Deep Q-learning, where the Q-table would be replaced by a Q-network. Instead of updating the values of the table, the Bellman equation would then update the weights of the neural network.

4. Personal note

I really enjoyed this project. It was challenging but not overly difficult, and led to a lot of exploration, while also requiring a certain amount of creativity.

It served as a solid introduction to reinforcement learning, with the vanilla Q-learning algorithm providing an intuitive gateway to core concepts like states, actions and rewards. I really would have liked to go deeper into Deep Q-learning, but I didn't have the time to master neural networks enough to build my own Q-network. However, I am very satisfied with the results of my Q-learning implementation, which I believe to work as expected and perform consistently.

The project was also the opportunity to practice the use of a simulator, i.e. interacting with the environment and controlling an agent evolving within it. This is a fundamental skill to have in the field of reinforcement learning, even if not in the scope of traffic control and SUMO.

Overall, I am happy with this project, firstly with my production, but moreover for the many relevant skills I acquired by doing it.