Traffic control
Reinforcement Learning - DSBA Master

Giacopelli Nicolò, Girardin Oskar

1 Problem Motivation and description

Traffic optimization is a critical aspect for many large cities. According to a study conducted by INRIX, a company specialized in traffic-analytics, traffic congestion costs the US economy close to 87 billion dollars in 2018. These costs come in different forms: opportunity costs of lost productivity, freight delay and inflationary pressure. Importantly, traffic congestion greatly impacts the environmental quality of (not only) cities, through fuel consumption and emission of pollutants. Given these costs, there is an big incentive for municipalities to optimize the flow of traffic. The effective control of traffic lights coordination is a major driver of this, hence the idea for our project. We will implement an agent that will learn how to maximize traffic flow of a four-way intersection, as effectively as possible. The optimal behaviour could then serve to local authorities for both policy evaluation and control.

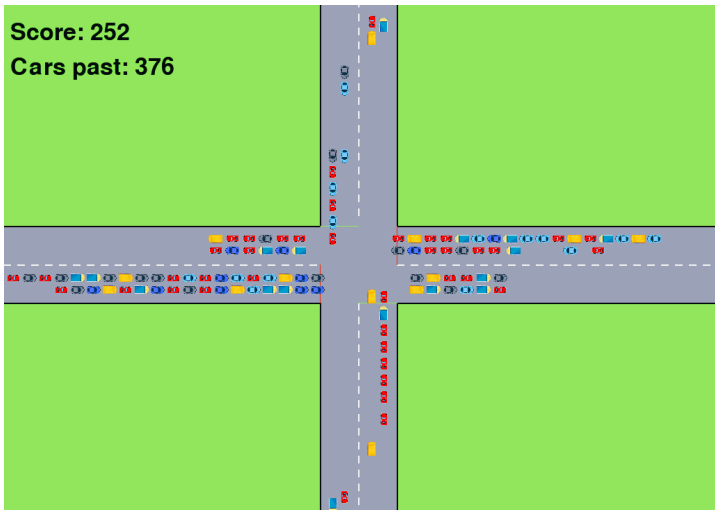


Fig. 1: Screenshot of the implemented environment

2 Environment implementation

We implemented the environment in python using `gymnasium` for the reinforcement learning aspect and `pygame` for the rendering and the environment's logic. Indeed, the `pygame` library is useful to make cars move, group cars together and check for collisions.

2.1 The RL setup

Action space. The action space refers to the range of possible actions the agent can take. In our case we decided to let the agent choose so-called phases. A phase is a predefined way of setting the four lights at the intersection. The name of a phase is given by the color of the lights starting in the north and continuing in clockwise direction. We have chosen the following phases the agent has to choose from: **1**. g-r-g-r, **2**. r-g-r-g, **3**. r-r-r-r, **4**. g-g-g-g, **5**. g-g-r-r, **6**. r-r-g-g, that is 6 possible actions, where the order is **north, east, south, west**.
The last two phases were introduced to observe how the agents would deal with phases that are not feasible, that is they induce collisions with high probability. Notably, when the agent switches phase, there is a yellow phase for the lights that from green turn into red, which ensures that all cars have left the intersection before a new phase is initialized. This ensures the feasibility of alternation between actions 1 and 2 (that is the two principal axis of the intersection), which is indeed the optimal action in this environment.
Observation space The pieces of information that are made available to the agent by the environment (and constitute the *state*) were conceived to be in line with the objective of maximizing traffic flow. In order to foster the agent to learn this type of behaviour, we defined the observation space as containing information about **1**. the number of cars waiting in each axis (NS and WE), **2**. the maximum waiting time among all cars and **3**. the previous phase. The waiting time of a car is defined as the number of agents' actions that make the car wait. Whenever at least one phase allows the car to move, the waiting time of the car is reset to 0.
Reward. We chose the reward to be consistent with the state space we are considering, that is the information available to the agent, which was inspired by [1]. Since we are considering a combination of queue lengths and maximum waiting time as observations, we define the reward as follows:

$$R_t = -(queue_{t,NS} + queue_{t,WE}) - \delta \times max\_waiting\_time_t$$

Hence the agent is penalized if the queues get too long or the waiting time of a car gets too high. This means that even if the lanes are full (with a sub-optimal policy) the reward keeps worsening. In our case, we chose $\delta = 8$ to optimize the magnitudes with respect to termination.
Termination conditions. We set two conditions for termination, to optimize and make the learning process feasible, that is **1**. the occurrence of a collision and **2**. maximum waiting time exceeds a predetermined threshold $\widetilde{wt}$. We set the cap on the waiting time to $\widetilde{wt} = 25$, so that the environment terminates when one car waits for more than 25 phases without moving. Whenever termination occurs, the environment returns a very negative reward of -5000, which was set to be harsher in magnitudes than a very negative policy repeatedly taken (for instance, selecting action $r - r - r - r$ brings to a reward of -3000 before termination occurs).

2.2 Linking RL & Pygame

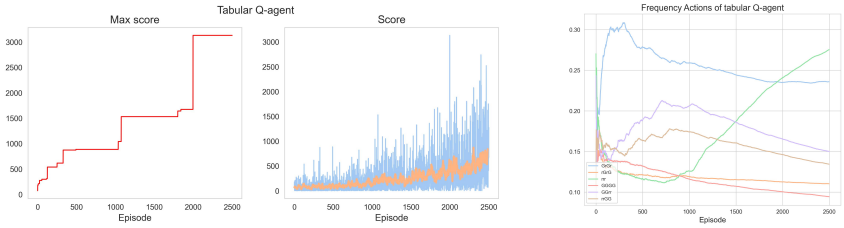
Let us explain how we linked the environment logic described above with the user interface. In `pygame`, we are dealing with discrete time steps. This means that, similarly to a RL environment, we are dealing with steps that the game takes where things happen. In our case, when dealing with a crossing of cars, each iteration of the game loop we are (among others): **1**. moving cars by a certain distance, **2**. checking if cars are at a light and need to stop, **3**. checking for collisions between cars and **4**. adding new cars. Linking the game loop with the RL loop is not straightforward. If we were to let the agent make an action at every iteration of the game-loop, the lights would be constantly switching, which is not realistic. Hence the RL-loop needs to happen at a lower frequency than the game-loop. We solved the issue as follows. **1**. The agent takes an action (sets lights), **2**. the game-loop keeps this action and continues for a fixed number of iterations (usually 50), **3**. we compute the observation and reward after the game iterations, **4**. the agent sees the observation and reward and chooses another action. Note that if there is a crash that happens within the game-iterations, we are breaking out of the loop and directly return the *crash-reward* of -5000. Otherwise the agent would only see the crash if it exactly happens when it is able to take an action.

3 Agents

In order to solve the environment, we implemented different versions of the Q-learning learning paradigm, that is **1**. Q-learning based on Dynamic Programming, **2**. Semi-gradient Q-learning based on tiles, **3**. Deep Q-Network with replay memory, target and policy networks. Notably, the first agent only uses information relative to the *queues*, without considering the previous phase or the maximum waiting time. The other agents use all three of them. The idea was to evaluate different levels of complexity, to understand their limits and strengths. Let us describe the results we obtained for each, and the conclusions we drew from the results.
Importantly, we set different arrival rates for the *north-south* and the *east-west* axis, with the former doubling the second in magnitude. This was done to check whether the agent could not only learn to discard the unfeasible actions (the last three) and to limit the sub-optimal one (all traffic lights set to red), but also learn to select action 1 (turn to green the lights on the *east-west* axis) more frequently than action 0 (green for those on the *north-south* axis.

3.1 Tabular Q-Learning agent

In order to get a feel for the workings of the environment, we started with a tabular Q-Learning agent following an ϵ-greedy policy. In order to keep the size of the Q-matrix manageable (for computation resource purposes), we decided to only use the information about the queue lengths as states. Hence the states of our agent are all possible values of the tuple $(queue_{NS}, queue_{WE})$.



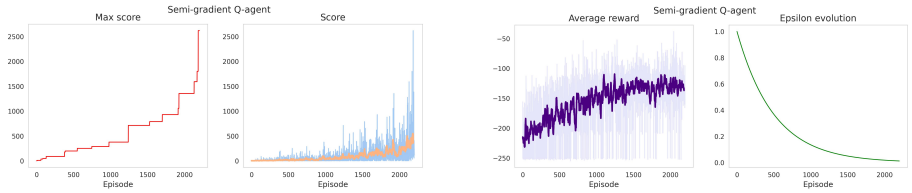
(a) Scores (b) Chosen actions

Fig. 2: Scores and actions of tabular Q-learning agent

In Figure 2, we can see how the Q-learning agent performs with an $\epsilon_{initial}$ of 0.5 and a linear schedule. We notice first of all that the agent is learning, we see the score (amount of cars passing) is increasing and reaches a maximum of over 3000 cars. By looking at the actions that were chosen by the agent, we observe that **1**. The agent learns to set the phase G-r-G-r around twice as often as r-G-r-G, matching the car arrival frequencies ratio and **2**. the agent is setting the lights the most often to r-r-r-r. The latter phenomenon can be explained by the fact that the agent cannot differentiate between two situations with the same queue lengths, but different maximum waiting times (hence different rewards). Since the agent is partially blind, the q-values do not reflect the truth and it will choose a sub-optimal policy as shown here.

3.2 Semi-Gradient Q-Learning with tiles

The Semi-Gradient Q-Learning is based on function approximation of the *q-table*. This is achieved with the help of tiles which discretize the state space with predetermined number of tiles and tilings (layers). In our case, we used 12 for both tiles and tilings. The epsilon-greedy strategy is gradually greedified with an exponential decay of the ϵ parameter, as shown below.



(a) Score evolution (b) Reward and epsilon evolution

Fig. 3: Semi-Gradient Q-learning agent with tiles

The agent achieves scores (number of cars passing the intersection) around 2500 after 2000 episodes, with an almost completely greedified policy. It can be seen that it could have done better if allowing for some more episodes.

By gathering the frequencies of each actions with respect to total number of actions taken, we plotted their evolution across the different episodes. It could be seen that the agent successfully achieves the first objective, that is discarding the unfeasible actions, as well as the third one, that is performing the phase relative to the busiest traffic lane (the *east-west* one) more frequently than the other one. However, the final performance is still sub-optimal because the agent often chooses the r-r-r-r phase.

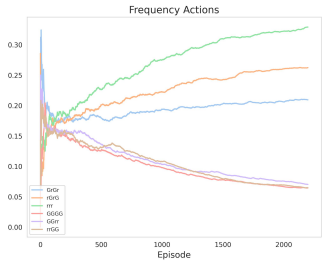
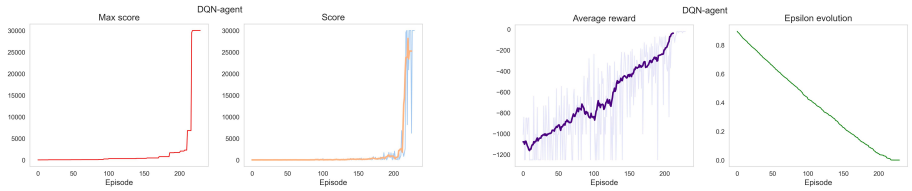


Fig. 4: Action frequency - Semi-gradient Q-learner

As explained below, the DQN agent is instead able to achieve the first and the second objective, but not the third one.

3.3 Deep Q-Network framework

As an additional benchmark for comparison, we evaluated how a DQN agent performs relative to the other two. This is indeed the learning framework used by the literature, as mentioned by [1]. The setup we chose is similar to the one seen in class, with a policy network that is optimized and a target network that gives estimations of the maximum q-value of the successive state with respect to the one updated. A soft update is then used at each iteration of the optimization procedure to make the two networks closer. The architecture chosen for the target and the policy network is the same, and it consists of a feed-forward network with one hidden layer of 128 neurons and Leaky ReLU activation functions.



(a) Score evolution

(b) Reward and epsilon evolution

Fig. 5: Deep Q-Network agent

Figure 5 shows that the DQN agent achieves much better performances, that is it actually solves the game, as $30,000$ is the upper bound on the score we set to be sure about termination. For what concerns the average reward across each episode (and its running mean), the agent is actually able to achieve almost average 0 reward, that is the optimal possible policy for this environment, as it implies the absence of a waiting queue as well as the fact that there is not any car that is kept waiting for more than one action. Furthermore, graph 6 shows that the DQN agent is able to achieve all the specified objectives for the optimal policy, that is it quickly understands the unfeasibility of the last 3 actions and manages to limit the r-r-r-r phase to a very low frequency (as low as the unfeasible ones). However, it lacks on what the Semi-Gradient Q-learning managed to do, that is selecting more frequently the phase corresponding to the lane with the highest frequency of arrival.

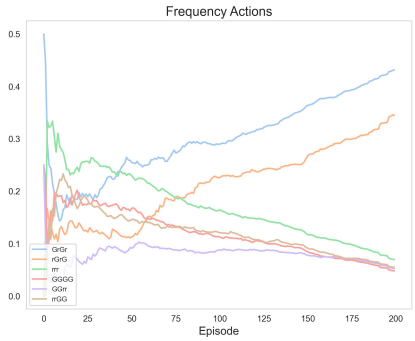


Fig. 6: Action frequency - DQN agent

References

[1] Salah Bouktif et al. "Deep reinforcement learning for traffic signal control with consistent state and reward design approach". In: *Knowledge-Based Systems* 267 (2023), p. 110440. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2023.110440. URL: https://www.sciencedirect.com/science/article/pii/S0950705123001909.