FIRST YEAR PROJECT 2025

DEEP REINFORCEMENT LEARNING FOR AUTOMATED TRAFFIC CONTROL

Antoine Chosson, William Barthélémy, Sasha Hakim, Raphaël Keghian
under the supervision of
Nadir Farhi, Zoi Christophorou
Grettia Laboratory, Patras University

May 12, 2025

Keywords: Deep Reinforcement Learning, Policy-Based, Traffic Control, Automated Vehicles, Prediction, Simulation.

The code for this project is available at the following GitHub repository:
https://github.com/Antoinechss/DEEP-RL.git.

1 Introduction

1.1 General Problem Statement

The steady increase in car traffic has made congestion in urban areas a major economic and social concern, leading to delays, excessive fuel consumption, and elevated air pollution levels. Optimizing traffic light control has therefore become essential to facilitate the flow of vehicles and minimize transit times at intersections. The objective is to predict the optimal sequence of green, orange, and red lights based on real-time road activity analysis. Thanks to the growing proportion of automated and intelligent vehicles, real-time traffic information such as position, speed, and acceleration is increasingly available. These variables can be leveraged to coordinate traffic lights in a more dynamic and efficient manner.

In this project, we aim to implement a Deep Reinforcement Learning algorithm to optimize traffic flow at an intersection, using real-time vehicle tracking data.

1.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is an area of artificial intelligence that combines deep learning and reinforcement learning. In this process, an agent learns the optimal strategy, to address a certain problem, through trial and error. In DRL, the

agent interacts with an environment by taking **actions** and receiving **rewards** accordingly. Over time, the agent refines its decision-making process to maximize cumulative reward [1][5].

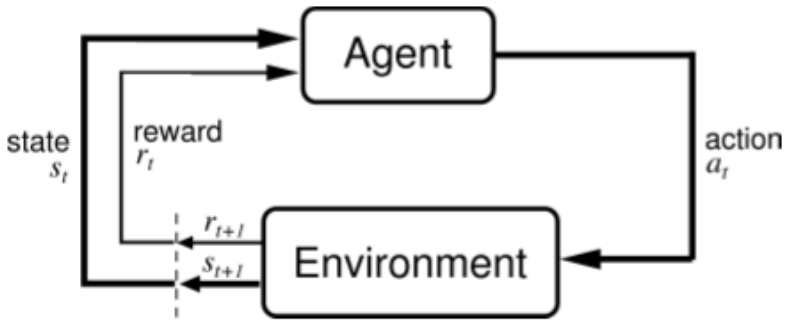


Figure 1: DRL framework

1.3 Policy-Based Algorithms

In Deep Reinforcement Learning (DRL), algorithms are typically categorized into two main classes: value-based and policy-based methods [1][5].

Value-based algorithms rely on a value function to estimate the expected return of each possible action from a given state. Once the training phase is complete, the agent selects, from each state, the action associated with the highest estimated value [1][5].

In this project, we focus on implementing a **policy-based** algorithm. In policy-based methods, the agent directly learns the policy $\pi(a|s)$, which defines a probability distribution over actions given a particular state s. Rather than estimating the value of actions, the objective is to optimize the policy itself in order to maximize the expected cumulative reward [1][5].

1.4 The REINFORCE algorithm

The policy-based algorithm we implemented is called **REINFORCE**. First introduced in 1992 by Professor Ronald J. Williams, REINFORCE is particularly effective for problems involving unknown or complex environments that are difficult to model analytically. It operates by directly optimizing the policy without requiring an explicit model of the environment's dynamics. In our case for example, it does not require any knowledge of the driver's behaviors or the previous car flows [1].

REINFORCE relies on two main principles: the **Monte Carlo Method** and the **Policy Gradient Theorem**.

The **Monte Carlo Method** is used to estimate the expected return by sampling episodes of interaction with the environment. By observing sequences of states, actions, and rewards until termination, it provides an unbiased estimate of the total reward associated with each trajectory. These sampled returns are then used to guide the policy updates [1][6].

The **Policy Gradient Theorem** provides the theoretical foundation for adjusting the policy parameters. It shows that *the gradient of the expected cumulative reward with respect to the policy parameters can be expressed as the expectation of the product of the return and the gradient of the log-probability of the taken actions* (see Appendix for mathematical formulation and further explaination). This result allows the agent to perform **stochastic gradient ascent** directly on the expected reward, progressively improving the policy through experience [1][6].

As mentionned above, the algorithm aims at optimizing the policy. To achieve this, it initializes a parameterized policy $\pi_{\boldsymbol{\theta}}(a|s)$. Through training, the algorithm tunes the parameter $\boldsymbol{\theta}$ for which the policy is optimal. This is done by interacting with the environment. During training phase, the agent performs several episodes. An episode τ is a sequence of states, actions and rewards, where every action is chosen as the one with the greatest probability according to the current policy. Consecutive rewards $R_1, ..., R_n$ are computed at each step. At the end of each episode, the sum of cumulated rewards G is computed [1]. Then, the agent performs the optimization of the $\boldsymbol{\theta}$ parameter which relies on the Policy Gradient Theorem. Object of maximization of the expected return $R(\tau)$ by computing the gradient of this quantity given below by the theorem [1]:

$$\nabla J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} R_t \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \tag{1}$$

where:

- R_t is the return (cumulative future rewards) from time step t,
- $\pi_\theta(a_t|s_t)$ is the policy's probability of taking action a_t in state s_t.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Figure 2: REINFORCE Algorithm pseudo code [1]

Many hyperparameters can then be adjusted such as the learning rate, the number of steps per episode, the number of training episodes, the gamma decay coefficient etc. (see Appendix for numerical values choices).

2 Materials and methodology

2.1 Working tools and libraries

The programming language we used to code our REINFORCE agent is **python**. This language contains a large ecosystem of libraries such as **PyTorch** that enable data processing and computations on matrices. This choice was particularly relevant to our project since the agent analyses a series of inputs (speed, position, etc.) in the form of a tensor [6].

To simulate the environment, we modelled a 2x2 intersection with a single traffic light signal in SUMO. We then integrated this intersection into a **python Gym** environment to be controlled by our agent. The connection and observation by the agent over the environment were handled with **TraCI** (Traffic Control Interface), which enables real-time reading of the simulation status. TraCI enables the retrieval of information about simulated objects and the modification of their behavior, making it possible to train and evaluate an agent within a dynamic traffic setting [2].

2.2 Methodology and approach

After acquiring a solid knowledge of theoretical elements of REINFORCE, we tested our understanding of the algorithm with a first basic application. Our approach therefore began with the implementation of a simple agent to familiarize ourselves with the policy-based reinforcement learning paradigm and to validate its functionality in a simple environment. To this end, we developed an initial `ReinforceAgent_MLP.py` program, which we tested on the Gym environment **CartPole-v1**. This made us familiar with the use of Gym and the linking process of an environment to an agent.

After around 10,000 training episodes, the agent successfully learned to balance the pole, thereby confirming the functionnality of our implementation.

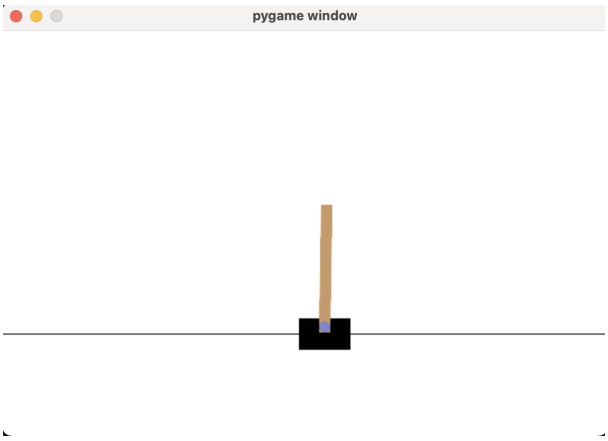


Figure 3: Cart Pole Balancing in gym environment

Having successfully implemented the REINFORCE agent on a simple environment, we then focused on integrating it into a more complex traffic control environment. We first defined the architecture of our project:

```
| > running scripts/
|   |-- train.py
|   |-- observe.py
| > results/
|   |-- kpis.csv
|   |-- kpis.py
| > environment/
|   |-- sumo_env.py
|   |-- scheduler.py
| > agent/
|   |-- reinforce_agent.py
|   |-- reinforce_agent.pth
```

The `sumo_env.py` file integrates the SUMO intersection into a Gym-compatible environment (see Appendix for detailed documentation). After importing the intersection raw code, we accessed vehicle parameters through the `_get_obs()`, `render()`, and `get_kpis()` methods. Control of the traffic lights by the agent was handled via the `step()` method.

The `reinforce_agent.py` file contains the generic implementation of the reinforcement learning agent as used for CartPole. Its behavior is governed by methods such as `generate_episode()`, `compute_discounted_returns()`, and `update_policy()`. For the policy network, we employed a simple two-layer Multilayer Perceptron (MLP). The network takes the current environment state as a tensor input, processes it through a hidden layer with 128 neurons using ReLU activation, and outputs a probability distribution over the possible actions via a Softmax function [6].

Training and evaluation of the agent were conducted through the `train.py` and `observe.py` scripts, respectively, located in the `running_scripts` folder [2]. During training, the KPIs (detailed below) were logged into a CSV file for further analysis. Once the training phase was complete, the optimal model was saved in the `reinforce_agent.pth` file. Additionally, the `kpis.py` script was used to generate performance graphs, as discussed in Part 3 (Results).

2.3 Controlling the environment with the agent

To enhance the realism of the simulation, we introduced a `scheduler.py` file, which defines a set of rules governing traffic light phase transitions [2][3]. In real-world scenarios, having flickering lights would be too dangerous. To address this, we implemented the following methods within the `TlScheduler` class:

- `set_cooldown()`: Enforces a minimum delay between two consecutive phase changes to prevent flickering of the traffic lights.
- `is_congested()`: Detects congestion levels and forces the agent to take action when excessive queuing is observed, making the model more responsive to real-time traffic conditions.
- `can_act()`: Determines whether the agent is currently allowed to modify the traffic signals based on cooldown constraints and congestion status.

2.4 Traffic scenario

The specific scenario we focused on is a two-phase, double-lane intersection with permitted conflicting left turns. This type of intersection is commonly found in many urban environments. It presents a suitable level of complexity while remaining simple enough to maintain a thorough understanding of both the inputs and outputs [2].
Each lane benefits from two permissive green intervals, corresponding to each primary axis, allowing left turns, through movements, and right turns. The agent's role is to alternate between the following two traffic signal phases [2] :

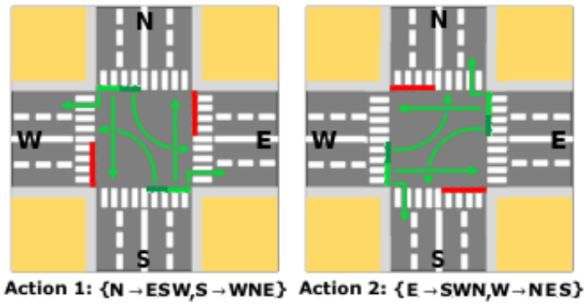
Phase 1: Controls the north-south axis, enabling movements from north to east, south, and west, as well as from south to west, north, and east.
Phase 2: Controls the east-west axis, allowing movements from east to south, west, and north, as well as from west to north, east, and south.

The corresponding action space is therefore defined as:

$$A = \{(n \to esw,\ s \to wne),\ (e \to swn,\ w \to nes)\},$$

with $|A| = 2$.

2-phase road intersection, Real Life



2-phase road intersection, Action Phases [2]

2.5 Computing the reward

When running a full episode prior to training, the choice of the reward computed at each step plays a crucial role, as it must be carefully tuned to align with the specific characteristics of the environment.
The objective of our agent is to minimize the total travel time through the intersection for all vehicles. To achieve this, we propose a reward function based on vehicle delay [2]. Minimizing the delay is equivalent to minimizing the lost travel time, defined as $\bar{t} - t_{\min}$, where $\bar{t}$ represents the average travel time of vehicles and $t_{\min}$ denotes the minimum possible travel time under the maximum speed limit $v_{\max}$. The delay for vehicle i at time t is computed as (see Appendix for mathematical proof):

$$d_i(t) = 1 - \frac{v_i(t)}{v_{\max}}$$

We then observe the following cumulative metric (rather than average) since it also contains information on the volume of vehicles passing through the intersection. Moreover,

we apply a power term so that fewer large delays are prioritized over many short delays, which encourages fairness between vehicles. The total squared delay is computed as [2]:

$$\mathrm{TSD}(t) = \sum_i \left(1 - \frac{v_i(t)}{v_{\max}}\right)^2$$

As rewards are maximized, minimizing a metric equates to maximizing its negated value. Here, the reward function acts in effect as a punishment, by maximizing the negative total squared delay. Additionally, the reward is normalized by the maximum total squared delay encountered at that time of training [2]:

$$\mathrm{TSD}_{\max}(t) = \max(\mathrm{TSD}(t), \mathrm{TSD}(t-1))$$

It is also centered in $[0, 1]$ for learning stability.

Thus, the complete TSC reward function is given by [2]:

$$R(t) = 1 - \frac{\mathrm{TSD}(t)}{\mathrm{TSD}_{\max}(t)} = 1 - \frac{\sum_i \left(1 - \frac{v_i(t)}{v_{\max}}\right)^2}{\max(\mathrm{TSD}(t), \mathrm{TSD}(t-1))}$$

2.6 Monitoring KPIs

In addition to tracking the cumulative reward, four key performance indicators (KPIs) are particularly relevant for optimizing vehicle flow at an intersection [2]:

- **Episode mean accumulated waiting time (s)**: Total time vehicles spend waiting at red lights over the course of an episode.
- **Episode mean total delay (s)**: Average reduction in vehicle speed compared to free-flow conditions (i.e., traveling at maximum allowed speed).
- **Episode mean total queue length (number of vehicles)**: Average number of vehicles waiting per timestep across all lanes.
- **Episode mean total volume (number of vehicles)**: Average number of vehicles present within the intersection during the episode.

During training, these metrics are collected from the environment and logged to **TensorBoard** for real-time visualization of learning progress (see the Results section for examples of TensorBoard logs). TensorBoard is a visualization toolkit developed for TensorFlow that facilitates the monitoring and analysis of machine learning experiments.

2.7 Training phase

To develop an effective model while avoiding overfitting, we trained our agent over 2,000 episodes, updating the policy every 5 episodes. The learning rate was set to 10^{-4}, which ensured stable and relatively fast convergence. Training was conducted on the internal GPU of our computer, and the full process took approximately 15 minutes. Convergence was typically observed after around 1,000 episodes. This setup allowed for rapid experimentation, enabling several training runs with intermediate hyperparameter tuning. After every 5 episodes, the current best-performing model was saved as a

`reinforce_agent.pth` file. This file represents the optimal agent at that point in training, and is updated whenever a model achieving a higher cumulative reward is found.

3 Results

3.1 Training logs in tensorboard

In the Tensorboard window, we were able to monitor the model's convergence by displaying the total reward :

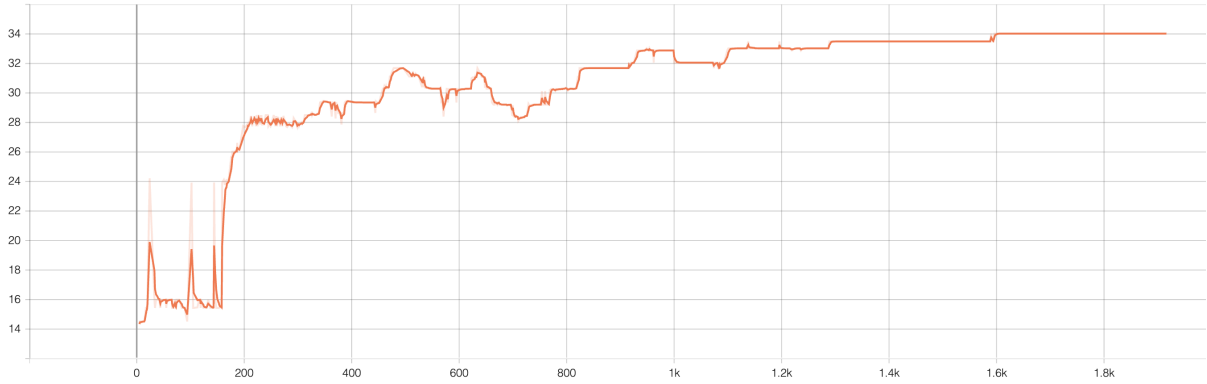


Figure 5: Total reward / n° of training episodes

And we could visualize the improvement of the 4 KPIs stated above. Each converging towards an optimal value :

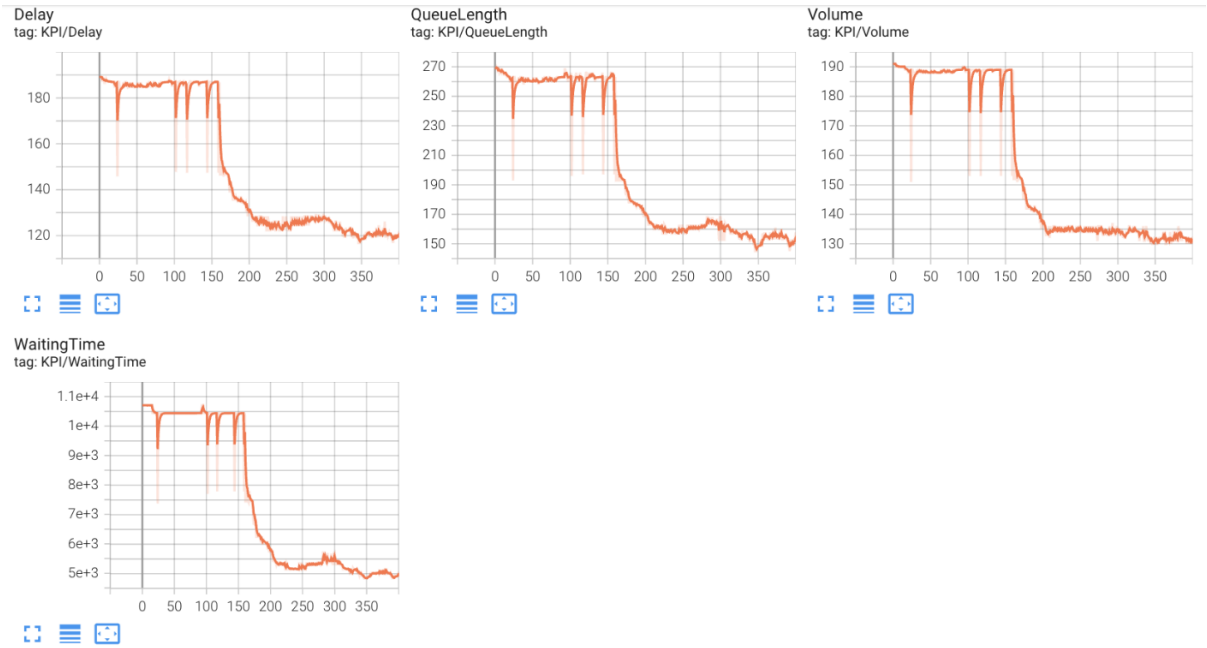


Figure 6: KPIs monitoring over training episodes

3.2 Observation in sumo gui

Once training was completed over 2,000 episodes, we used SUMO-GUI (the graphical interface extension of SUMO) to visualize the agent's behavior during a full evaluation episode. This qualitative analysis allowed us to confirm that the agent effectively managed the incoming flow of vehicles at the intersection.

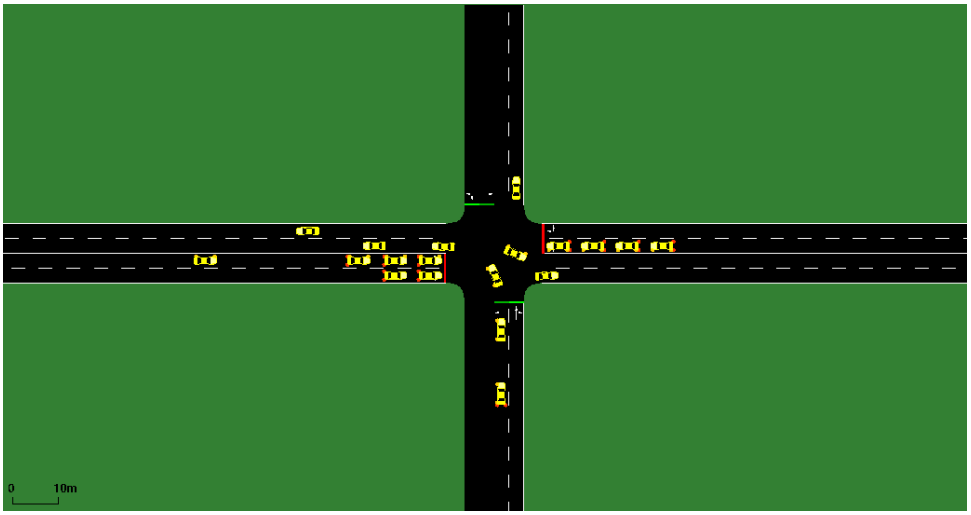


Figure 7: Visual rendering of the intersection, SUMO-Gui

3.3 Statistics

This section presents a table of raw statistics for the four KPIs. It includes our own results (REINFORCE column) as well as comparative results for the same intersection when controlled by DQN, MaxPressure (MP), and Self-Organizing Traffic Lights (SOTL) algorithms (see below for further explanation) [2][4].

For each KPI, we display its mean, median, Q1, Q3 and variance over an episode. We applied a colour code to visualize and compare performance.

Indicator	KPI	DQN	MP	SOTL	REINFORCE
Mean	Waiting time	18280,2	11130,9	13864,7	14500
	Delay	52,8	68,1	90,5	62
	Queue Length	35,7	39,3	46,2	37
	Volume	60,6	76	96,8	70
Median	Waiting time	12542	8528,5	12433,5	14200
	Delay	53,3	69,8	94	61
	Queue Length	35,7	40	44,6	36,5
	Volume	61,1	77,4	100	69
First Quartile	Waiting time	475,1	2543	7325,2	11500
	Delay	24,6	50,4	71,3	50
	Queue Length	10,6	24,5	34,4	30
	Volume	33,4	57,6	77,1	60
Third Quartile	Waiting time	27001,3	17845,1	18521,8	17500
	Delay	74	87,4	108,8	74
	Queue Length	50,9	54,3	58	44
	Volume	82,9	95,5	115,3	80
Variance	Waiting time	452039780,8	103870420,5	74104235,1	520000000
	Delay	873,3	1088,5	771,7	450
	Queue Length	611,7	489,4	295,4	320
	Volume	904,3	1138,4	817,3	95000

Figure 8: Statistics of monitored KPIs

3.4 Density distributions

The below figures show the density function of each KPI over the course of one episode. They were generated thanks to the kpis.py file. Displaying the density functions of each KPI over the course of an episode provides a detailed view of the distribution of traffic-related metrics, beyond simple averages.

- It provides richer information than simple averages.
- It reveals how stable or unstable the agent's behavior is throughout the episode.
- It can expose anomalies, bottlenecks, or outliers that may occur during traffic flow.
- It helps assess the consistency of the agent's performance, not just the final outcome.

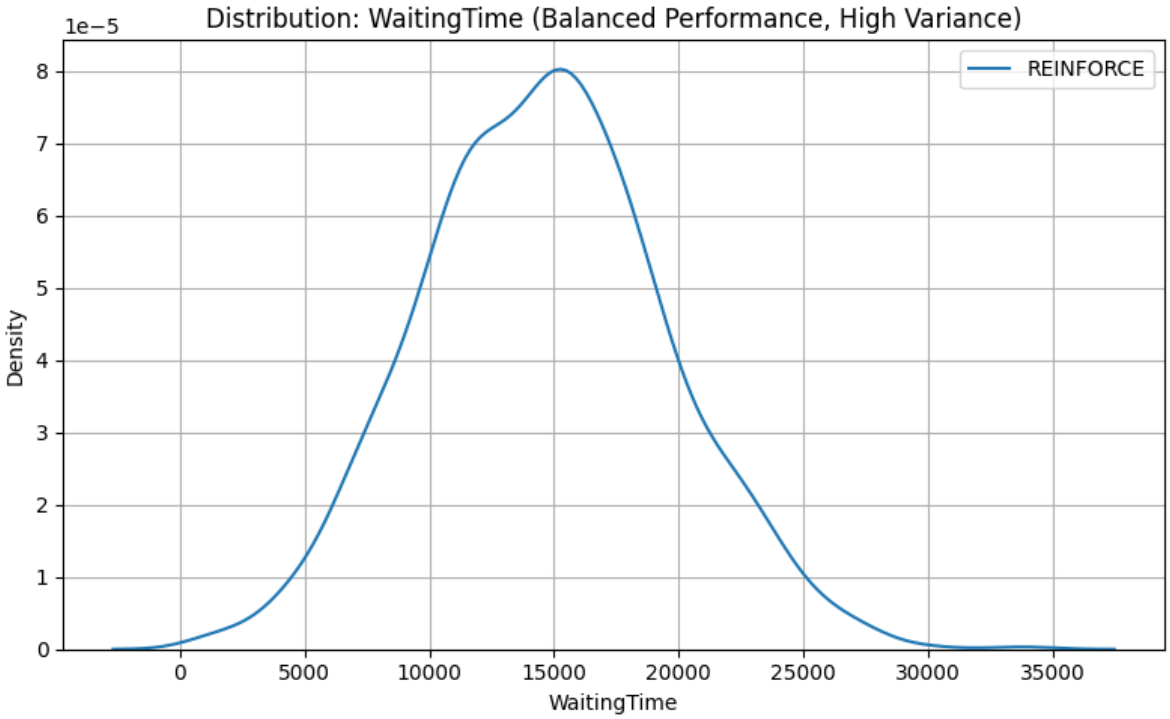


Figure 9: Density distribution : Episode mean total waiting time

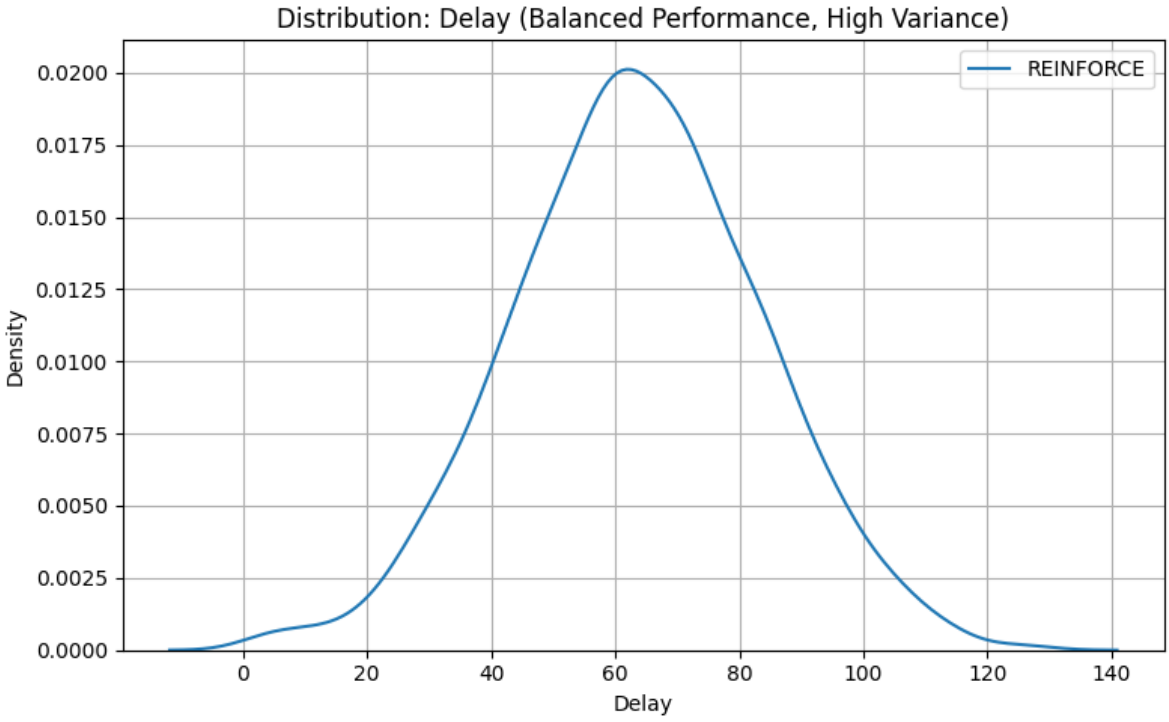


Figure 10: Density distribution : Episode mean total delay

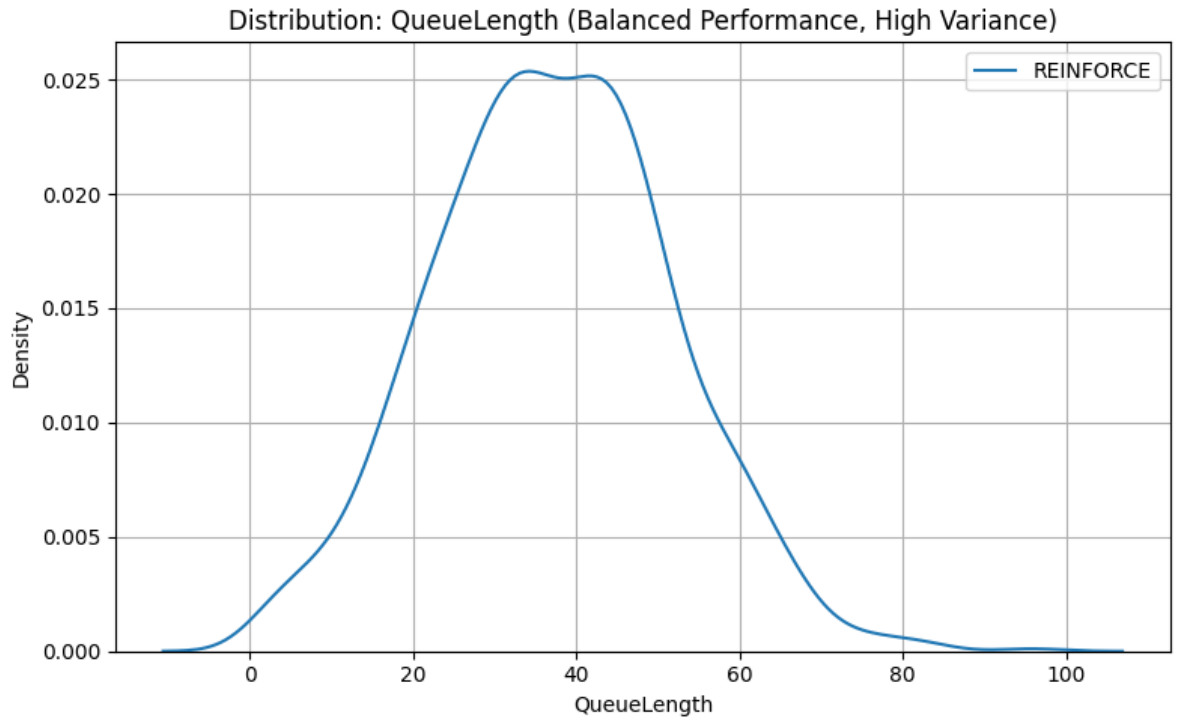


Figure 11: Density distribution : Episode mean total queue length

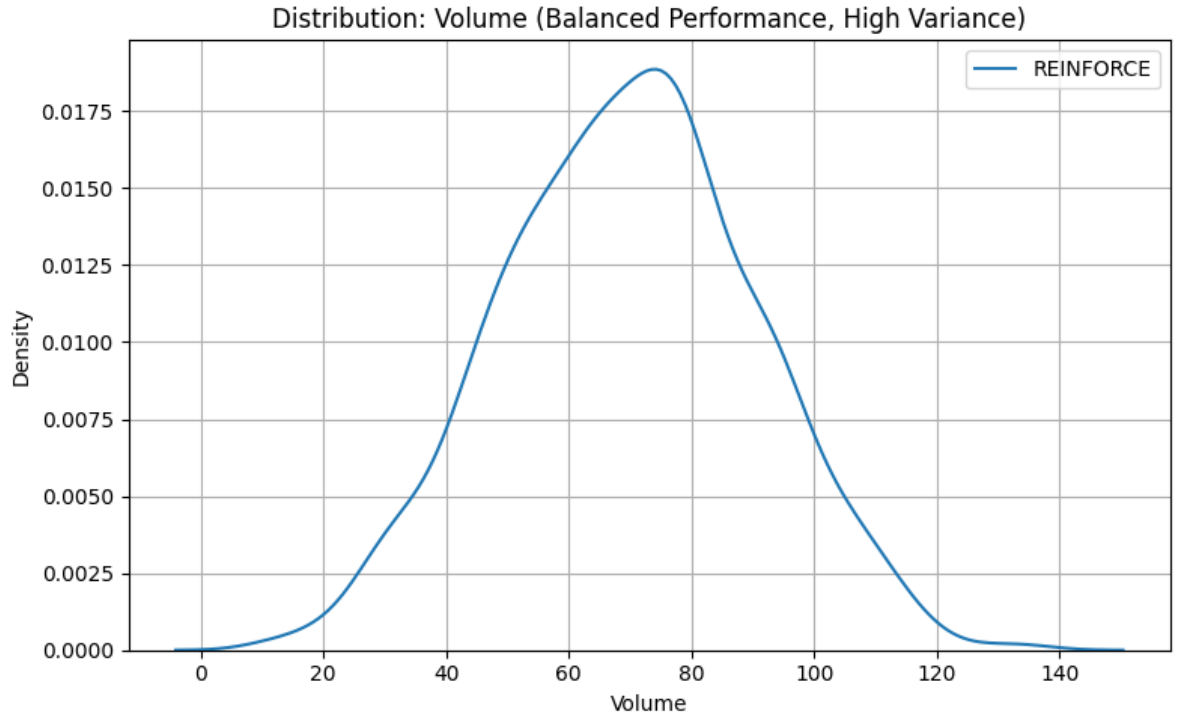


Figure 12: Density distribution : episode mean total volume

4 Discussion

4.1 Comparing with DQN, Value-based algorithms

Thanks to the preliminary work of Romain Ducrocq, we are able to compare our REINFORCE agent's performance to that of value-based algorithms applied to the exact same traffic intersection (see appendix for the density distribution graphs of the KPIs). The two value-based algorithms we use for comparison are Deep Q-Network (DQN) and Max Pressure (MP). Additionally, we compare our agent to a more heuristic-based approach, Self-Organizing Traffic Light Control (SOTL).

- **Deep Q-Network (DQN)** is a model-free, value-based reinforcement learning method that approximates the optimal action-value function using deep neural networks. It learns by estimating the long-term expected rewards for different traffic light actions at each intersection state [1][2].
- **Max Pressure (MP)** is a rule-based algorithm that selects traffic light phases by minimizing queue pressure differences between incoming and outgoing lanes. It is based on real-time queue observations without any learning component [1][2].
- **Self-Organizing Traffic Light Control (SOTL)** is a decentralized heuristic algorithm where each traffic light adjusts autonomously based on local traffic density, following predefined rules rather than optimization through learning [4].

This setup allows us to benchmark REINFORCE against both learning-based and rule-based alternatives under identical conditions.

4.2 Comparaison of raw performance indicators

KPI	REINFORCE vs DQN	REINFORCE vs MP	REINFORCE vs SOTL
Waiting Time	-20.3% Worse	+30.3% Worse	+4.6% Better
Delay	+17.4% Worse	-8.9% Better	-31.4% Better
Queue Length	+3.6% Worse	-5.9% Better	-19.9% Better
Volume	+15.5% Better	-7.9% Worse	-27.7% Better

Table 1: Comparison of KPIs between REINFORCE, DQN, MP, and SOTL

The results show that REINFORCE achieves a strong overall compromise across the different KPIs. Compared to DQN, REINFORCE significantly reduces waiting time by 20.3%, but at the cost of slightly higher delay and queue length. However, it handles a notably larger traffic volume, suggesting better throughput capacity.

When compared to the MaxPressure (MP) algorithm, REINFORCE exhibits a slightly worse waiting time and volume, but achieves lower delay and better queue management.

Finally, against SOTL, REINFORCE performs better on three out of four KPIs, reducing waiting time by 4.6%, delay by 31.4%, and queue length by 19.9%, although it handles slightly less volume.

Overall, REINFORCE offers a very competitive balance, particularly excelling in minimizing delay and queue length, which are two critical metrics for traffic flow optimization..

4.3 Comparaison of KPI density distributions

Across all KPIs, the REINFORCE agent has **broader density curves** compared to DQN and MP, confirming a higher variance in its performance. For instance, the Waiting Time distribution for REINFORCE is wider and flatter, reflecting a greater diversity of outcomes across episodes. In contrast, DQN displays a narrower and more peaked distribution, indicative of more consistent and stable behavior.

Specifically, in the Delay and Queue Length distributions, REINFORCE maintains competitive central values relative to DQN and MP, but with noticeably **heavier tail**s. This indicates that while REINFORCE often performs similarly on average, it also occasionally encounters **extreme outcomes**.

Nevertheless, the SOTL baseline shows the widest and most irregular distributions across all KPIs, reaffirming its inferior and unstable performance compared to learning-based methods.

Overall, the analysis of the KPI density distributions highlights a key trade-off: REINFORCE achieves comparable average performance to DQN and MP but at the cost of higher variance and reduced stability.

4.4 Discussion over the issue of variance Analysis

An important aspect observed in the performance comparison is the significantly higher variance associated with the REINFORCE agent across most KPIs, particularly in Waiting Time and Delay. The variance of Waiting Time for REINFORCE reaches **520,000,000**, compared to **452,039,780.8** for DQN and only **103,870,420.5** for MP. Similarly, REINFORCE's Delay variance is **450**, while DQN's is **873.3** and MP's is **1088.5**.

This phenomenon is due to different learning methodologies. Value-based like DQN and MP estimate the expected future rewards for each action in a given state, progressively smoothing their estimations as training progresses. As a result, value-based algorithms tend to converge towards stable policies with low variability. In contrast, policy-based methods like REINFORCE directly optimize the parameters of a stochastic policy by sampling actions according to their probability distributions [1][7].

Because the updates in REINFORCE are based on sampled returns without an explicit value estimation step, they are inherently noisier and subject to higher variability [1]. Each training episode may reinforce different trajectories depending on the stochasticity of the environment and the randomness of action sampling [1][7].

While higher variance can sometimes encourage broader exploration, in traffic control applications it poses a critical risk: unstable policies can lead to unreliable performance under varying traffic conditions.

5 Conclusion

5.1 Evaluating our approach and methodology : dealing with encountered issues

Material issues : Due to compatibility issues, it is worth mentionning that members of our group using Windows had to run the code on an Ubuntu virtual machine in order to visualize the simulation using SUMO-GUI. This was a first complex process and took a lot of time before being finally able to run SUMO and rendering extensions on our computers.

Learning the Wrong Yet Optimal Strategy: One major issue emerged during the training phase. Due to the limited length of the roads in the simulation, when a queue grew long enough to reach the end of a lane, SUMO automatically "teleported" vehicles — removing them from the simulation to prevent deadlock. Paradoxically, these teleported vehicles artificially improved the overall traffic flow, although such events are unrealistic in real-world scenarios. Consequently, the agent mistakenly learned to reward these occurrences. As a result, it developed an undesirable policy: deliberately blocking one direction entirely (causing teleportations at the road extremity) while allowing free flow in the opposite direction.To address this issue, we modified the reward function by introducing a strong penalty for each teleportation event. This adjustment discouraged the agent from exploiting teleportations as an optimization strategy, ensuring the learning of more realistic and desirable traffic management behaviors.

```
# --------------------
# High penalty in case of car teleportation in order to prevent the model from being saved
# --------------------
if teleport_count > 0:
    self.teleport_flag = True
    return -1000.0
else:
    self.teleport_flag = False
    return -tsd
```

Figure 13: Python code fragment : dealing with teleportation

5.2 General Conclusion and Possible Performance Improvements for REINFORCE

The REINFORCE algorithm has several advantages that make it interesting for reinforcement learning tasks in complex environments like traffic signal control. It is relatively simple and intuitive because it directly tries to find the best policy by maximizing expected rewards, without needing to estimate a value function first. Another strength is that REINFORCE naturally handles stochastic policies, which helps the agent explore better and adapt more easily to changing environments.

However, REINFORCE also has some important limitations. One of the main problems is that the gradient estimates can have a lot of variance since updates are made only at the end of each episode based on all the rewards collected. This makes learning unstable

and sometimes prevents the agent from finding the best solution. Also, learning with REINFORCE can be quite slow because it has to wait for the full episode to finish before updating the policy.

To improve REINFORCE's performance, several ideas could be explored. First, using a baseline (for example, a state-value function) could help reduce the variance of updates without introducing bias. Another way to improve would be to use an advantage function, which would help the agent focus on actions that are better than average, making learning more efficient [7].

Moving towards actor-critic methods could also be a good option. In these models, one part (the actor) decides on the actions, while another part (the critic) estimates how good they are. This combination helps speed up learning and makes it more stable. Finally, adding entropy regularization during training could encourage the agent to keep exploring and avoid getting stuck in bad solutions too early.

In conclusion, although REINFORCE provides a solid and conceptually elegant foundation for policy learning, its practical deployment in real-world environments such as traffic control remains challenging. Without additional refinements, issues such as high variance, unstable updates, and slow convergence can significantly limit its effectiveness. Enhancing REINFORCE with variance reduction techniques, stabilizing training through baselines or advantage functions, and accelerating convergence by combining policy and value learning would greatly improve its robustness and applicability. Addressing these challenges is essential to bridge the gap between theoretical models and operational deployment in complex, dynamic systems like urban traffic networks [7].

6 Appendixes

6.1 Choice of numeric parameters

- speed limit : $v_{max} = 26.8km/h = 16.67mph$
- number of steps per episode : 500
- number of training episodes : 2.000
- learning rate : 0.001
- gamma decay : 0.99

6.2 Policy Gradient Theorem

The Policy Gradient Theorem provides a foundation for optimizing a parameterized policy $\pi(a|s, \boldsymbol{\theta})$ in reinforcement learning. It establishes that the gradient of the expected return $J(\boldsymbol{\theta})$ with respect to the policy parameters $\boldsymbol{\theta}$ can be written as:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}),$$

where:

- $\mu(s)$ is the on-policy distribution over states,
- $q_\pi(s, a)$ is the expected return (action-value function) after taking action a in state s,
- $\nabla\pi(a|s, \boldsymbol{\theta})$ is the gradient of the policy with respect to the parameters $\boldsymbol{\theta}$.

The symbol $\propto$ indicates proportionality; in the episodic case, the constant of proportionality is the average episode length, and the relation can be treated as an equality.

6.3 Computing the total squared delay (proof)

travel distance from $t_{\min}$ to $\bar{t}$ for a given vehicle i with speed $v_i(t)$ at time t :

$$t_{\min} \cdot v_{\max} = \int_0^{\bar{t}} v_i(t)\, dt \quad \Longrightarrow \quad t_{\min} = \frac{1}{v_{\max}} \int_0^{\bar{t}} v_i(t)\, dt$$

thus:

$$\bar{t} - t_{\min} = \int_0^{\bar{t}} 1\, dt - \frac{1}{v_{\max}} \int_0^{\bar{t}} v_i(t)\, dt = \frac{1}{v_{\max}} \int_0^{\bar{t}} (v_{\max} - v_i(t))\, dt$$

Thus, minimizing the delay is equivalent to minimizing, for each timestep t and vehicle i:

$$d_i(t) = \frac{1}{v_{\max}}(v_{\max} - v_i(t)) = 1 - \frac{v_i(t)}{v_{\max}}$$

6.4 Results for control by DQN algorithms

These results are extracted from Romain Ducrocq's work available at the following GitHub repository: https://github.com/romainducrocq/DQN-ITSCwPD.git. The results extracted from his work and compared to our REINFORCE are the following:

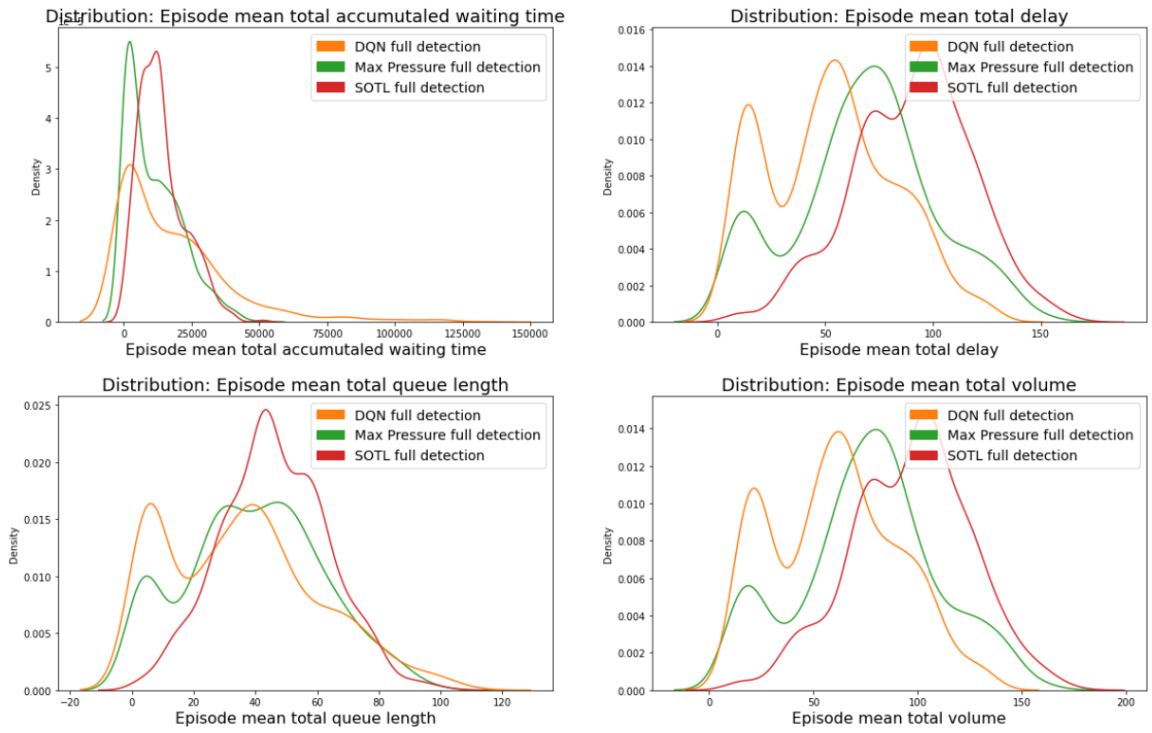


Figure 14: Density distribution of monitored KPIs

6.5 Documentation Used

Creating a custom environment with Gymnasium:
`https://gymnasium.farama.org/introduction/create_custom_env/`

TraCI (Traffic Control Interface):
`https://sumo.dlr.de/docs/TraCI.html`

PyTorch Documentation:
`https://pytorch.org/docs/stable/index.html`

TensorBoard Documentation:
`https://www.tensorflow.org/tensorboard/get_started`

SUMO-GUI Documentation:
`https://sumo.dlr.de/docs/sumo-gui.html`

7 Acknowledgments

We would like to express our sincere gratitude to Nadir Farhi and Zoi Christoforou, researchers at the CERMICS laboratory of École des Ponts et Chaussées, for their invaluable support and guidance throughout this project.
We also extend our thanks to Romain Ducrocq for his prior work on implementing a DQN agent within a SUMO environment. His project provided us with valuable insights that greatly informed the development of our own REINFORCE-based environment.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, 2018 (accessed February 3, 2025).

[2] R. Ducrocq, *Deep Reinforcement Q-Learning for Intelligent Traffic Signal Control with Partial Detection*, Research Internship Report, IFSTTAR-COSYS-GRETTIA, Université Gustave Eiffel, 2021 (accessed February 3, 2025).

[3] R. Ducrocq, *DQN-ITSCwPD GitHub Repository*, [online]. Available at: `https://github.com/romainducrocq/DQN-ITSCwPD` (accessed February 3, 2025).

[4] E. Vinitsky, K. Parvate, A. Kreidieh, C. Wu, and A. Bayen, "Grid Formation of Traffic and Autonomous Vehicles in Mixed Autonomy," *HAL-Inria*, 2017. [online]. Available at: `https://inria.hal.science/hal-01656255v1/document` (accessed April 2, 2025).

[5] M. Mujahed, "Policy Gradient Methods (Reinforcement Learning Lecture)," YouTube, 2022. [online video]. Available at: `https://www.youtube.com/watch?v=8JVRbHAVCws` (accessed January 23, 2025).

[6] I. Sofeikov, "REINFORCE Algorithm: Reinforcement Learning from Scratch in PyTorch," Medium, 2021. [online]. Available at: `https://medium.com/@sofeikov/reinforce-algorithm-reinforcement-learning-from-scratch-in-pytorch-41fcccafa107` (accessed February 10, 2025).

[7] M. Cholodovskis, "The True Impact of Baselines in Policy Gradient Methods," Medium, 2022. [online]. Available at: `https://medium.com/@marlos.cholodovskis/the-true-impact-of-baselines-in-policy-gradient-methods-aa4cb50c4f8c` (accessed May 12, 2025).