Problem Statement:
Assignment 2's purpose is to use reinforcement learning in traffic light optimization. The objective is to reduce congestion, decrease waiting times, and improve traffic throughput.

Learning environment:

- The custom reinforcement learning environment was made with the help of :
  - SUMO software: https://sumo.dlr.de/docs/index.html
  - the gymnasium library : https://gymnasium.farama.org/
  - the sumo_rl library: https://lucasalegre.github.io/sumo-rl/
- The SUMO software is used to simulate an intersection and generate traffic.
- The sumo_rl library enables us to create a custom Gymnasium environment while still ensuring compatibility with popular reinforcement libraries.
- **Creating our custom environment:**

```
from sumo_rl import SumoEnvironment
# Create custom environment with our configuration xml files and the change in cumulative vehicle delay as a reward function
env = SumoEnvironment(
    net_file="./environments/single-intersection.net.xml",
    route_file="./environments/single-intersection-vhvh.rou.xml",
    out_csv_name="./outputs/dqn",
    single_agent=True,
    reward_fn='diff-waiting-time',
    use_gui=True,
    num_seconds=steps_per_episode,
    begin_time=0
)
```

  - The .**net.xml** file contains the road network of the intersection
  - The **.rou.xml** file contains the settings to control the traffic flow. It is by changing this file that we can randomized traffic flow.

Reinforcement Learning Components for Traffic Light Management System:

Agent:

- The agent in our project is the Traffic Light Controller, which is programmed to assess and react to the ongoing traffic conditions at the single intersection. The agent's primary role is to observe the current state of the intersection and determine the most effective traffic light configurations to minimize congestion.

Environment:

- The environment is created with the **sumo_rl** library and encapsulates the **SUMO** traffic simulation that models the intersection, traffic lights, and vehicles. It provides a setting in which the agent can change the traffic signal, offering realistic traffic scenarios for the agent to work on.

State Space:

The state space the environment is represented by the combination of all the domains of the following attributes:

- **Vector indicating which light is green :** A one-hot encoded vector representing the current active green lights.
- **Minimum Green Condition:** A boolean attribute reflecting whether the minimum time in green has been met for the intersection's lights. When that time is reached, the agent can either keep the lights green or transition to a different state.

- **Lane Density:** The density of each incoming lane, calculated as the number of vehicles present divided by the total capacity of the lane.
- **Lane Queue:** This measures the queue length in each lane. This is quantified as the number of vehicles moving slower than 0.1 m/s divided by the lane's capacity.

These are predefined attributes that come with the **sumo_rl** library we used to create our environment.

Action Set:

The action space for the traffic signal agent is discrete and has a size of 4:

- **4 Action Options:** Each action corresponds to a traffic light configuration for the environment . The agent chooses between the set of actions every 5 steps in the simulation. Here are the 4 configurations:
    - North-South Advance: Green for north and south directions, accommodating vehicles going straight or turning right.
    - North-South Left Lane Advance: Green specifically for vehicles turning left from the north and south.
    - East-West Advance: Green for east and west directions, for vehicles going straight or turning right.
    - East-West Left Lane Advance: Green for vehicles turning left from the east and west directions.

Reward:

The chosen reward function evaluates the efficiency of the agent's ability to manage traffic through the difference in cumulative vehicle waiting time of the intersection:

- **Reward Calculation:** The reward at any time step **t** is calculated as the difference in total vehicle waiting time from the previous step, **r_t = D_t - D_(t+1)**, where **D_t** is the cumulative delay of all vehicles in the intersection at time **t**.
- **Impact on throughput:** By choosing this simple reward function, we ensure to penalize the agent when the intersection faces an increase in waiting time, which would negatively affect throughput.

Base model (Deep Q-learning):

- The chosen base model is the **DQN model** from the stable_baselines3 RL library with the MLP policy (Multi Layer Perceptron).
- **Justification of choice of Base Model:**
    - **Generalization to unseen states:**
        - The DQN model combines Q-learning and Deep neural networks to estimate the Q-function (expected return) when it encounters any state-action pair.
        - By replacing the fixed Q-table with a deep neural network, this mechanism puts more emphasis on the long-term reward instead of on the immediate reward.
        - Ultimately, the deep neural network enables the model to generalize unseen cases by predicting the Q-value.
- **Base Model hyperparameters:**
    - The hyper-parameter that we looked at are the following:
        - ***Learning rate (α)*** : Controls how much the neural network weights are updated during training.

- ***Gamma (γ)*** : Determines how much the agent values future rewards compared to immediate rewards.
- ***Buffer size*** : Size of the replay buffer. The replay buffer stores experiences and the agent later samples from this buffer to learn.
- ***Target Update Interval*** : Determines how frequently the weights of the target network are updated with the weights of the main Q-network.
- ***Initial and final epsilon value*** : Controls the overall balance between exploration and exploitation. Epsilon is the probability of choosing a random action (exploration) versus the best-known action (exploitation).
- ***Exploration fraction***: Determines the percentage of the total training steps over which the exploration rate is reduced from its initial to its final value.
- **Base model's hyper-parameter values :**
    - Learning rate : 0.0001; Gamma : 0.99; Buffer_size : 1 000 000; Target Update Interval : 10 000 steps; Initial and final epsilon values : 1 and 0.05; Exploration fraction: 10%

- **Training of the base Model**:
    - Total number of steps : 339 200 steps over episodes of 800 steps
    - The training was stopped at 339 200 steps because the mean reward was on a clear decreasing trend.

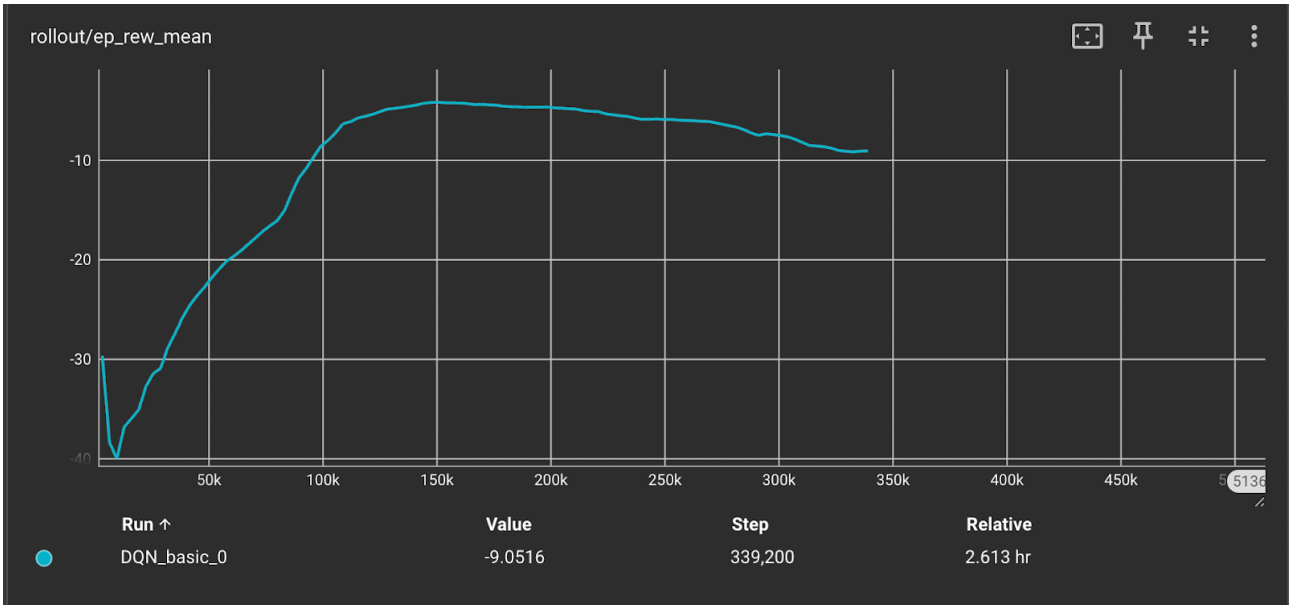

Figure1: Base Model Mean Episode Reward as function of steps

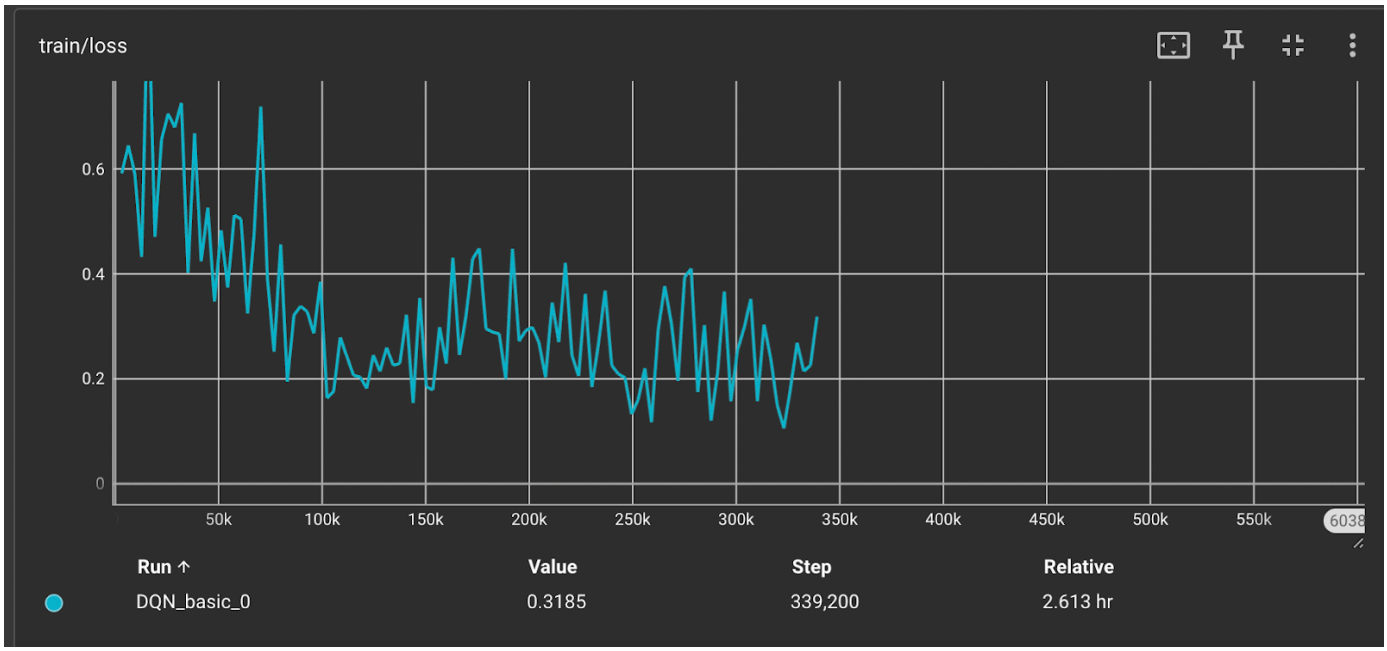

Figure 2: Base Model Loss Value as function of steps

Analysis:

- Mean reward per episode:
  - We clearly see that the agent starts with a higher exploring rate, which causes a decrease in reward.
  - The model then follows a steady climb before plateauing and slightly declining. This suggests that initially, the model is learning effectively, but it may start **overfitting** or failing to generalize well as training progresses.
  - A possible cause could be that the model is exploiting too much a sub-optimal policy.
- Variation of the loss:
  - The high fluctuations indicate that the model is not consistent with its estimated Q-values and the policy is not optimal
  - A possible cause could be a learning rate that is too low. This would lead to the model not converging to an optimal policy and , hence , a lack of consistency in Q-values.

Fine Tuned DQN V1:

- **Modifications made to base model** :
  - **Target update interval** : 800 steps
    - Justification: This was changed to match the length of an episode. The model will now update the target network every episode after the environment resets. This ensures steady updates of the neural network used to estimate the policy.
  - **Training of the model:**
    - Total number of steps : 800K over episodes of 800 steps

- **Results**:

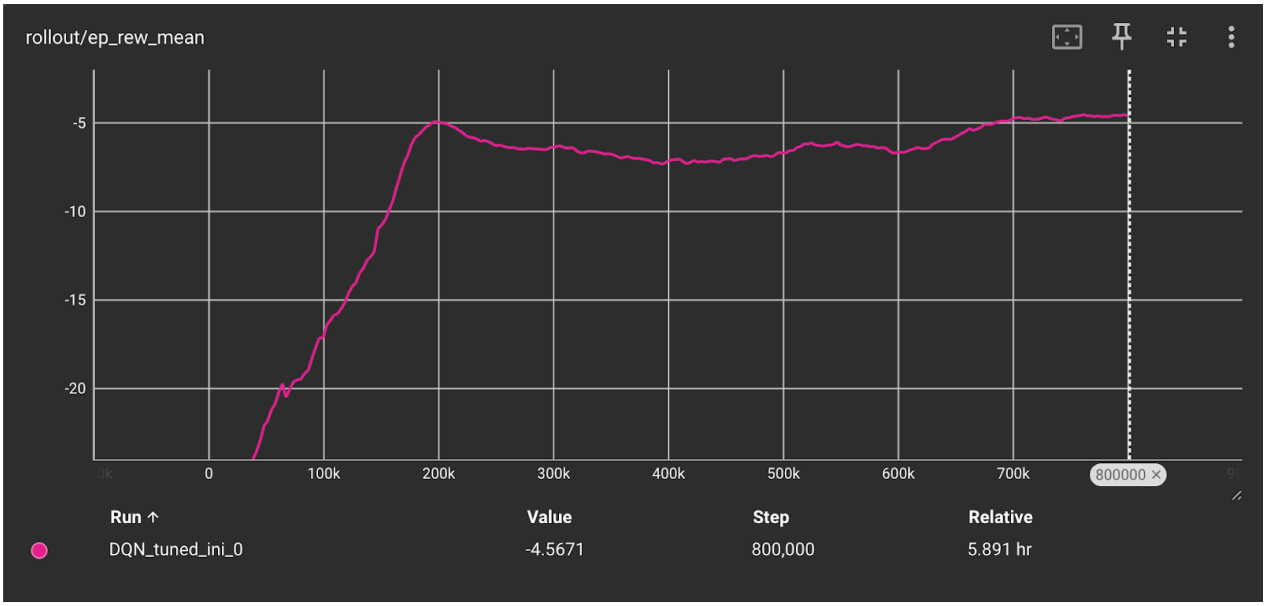

Figure 3 : Tuned V1 Mean Episode Reward as a function of steps

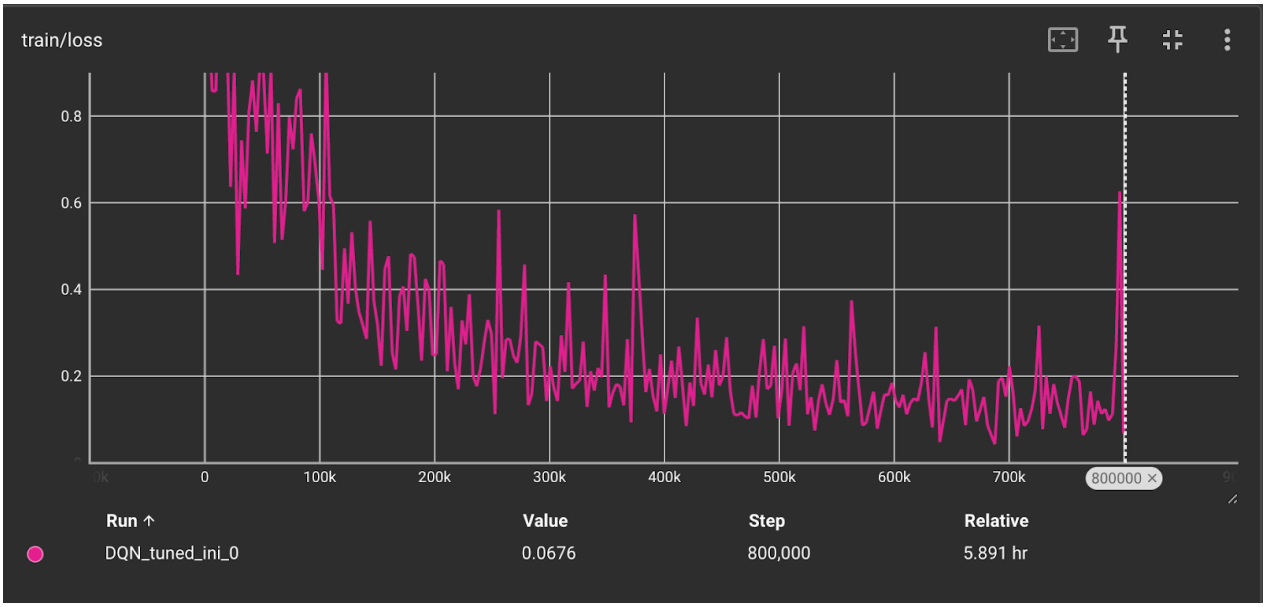

Figure 4: Tuned V1 Loss value as a function of steps

- **Mean reward per episode:** After the initial improvement, the reward decreases, then goes into an increasing trajectory to, finally, reach an optimal reward value at the end. This suggests that the agent has converged on a policy that performs consistently across episodes.
    - **Impact of the modifications**:
        - **Target update interval**: By aligning the deep neural network update frequency with the episode length, each episode

potentially benefits from a fresher, more accurate deep neural network. This may lead to steadier learning.
    - The curve still faces a slight dip after it reaches its peak. This may indicate that the model is **still slightly exploiting too early.**
- **Loss function results per episode:** The loss starts high and steadily decreases. It also seems to have less of a wavy pattern when compared to the base model.
    - **Impact of the modifications:**
        - **Target update interval**: The more frequent updates keep the learning and target networks closely synchronized, thus ensuring that the learning network's adjustments are quickly reflected in the target. This results in steadier loss values across time.

Fine Tuned DQN V2 :

- **Modifications made to the Tuned Model V1:**
    - **Learning Rate**: Increased from 0.0001 to 0.0002 to potentially accelerate learning.
    - **Gamma**: Reduced from 0.99 to 0.95, hence potentially putting more emphasis on immediate rewards versus future rewards. This is something that might help performance in a fast pace changing environment like an intersection.
    - **Buffer Size**: Reduced from 1,000,000 to 100,000, potentially focusing the experience data on more recent events.
    - **Exploration Settings**: Adjusted **initial epsilon** from 1 to 0.9 and **final epsilon** from 0.05 to 0.1, modifying the exploration-exploitation balance to have less aggressive exploration in the beginning while still keeping a higher exploration rate at the end.
- **Training of the model**:
    - Total number of steps : 800K over episodes of 800 steps
- **Results**:

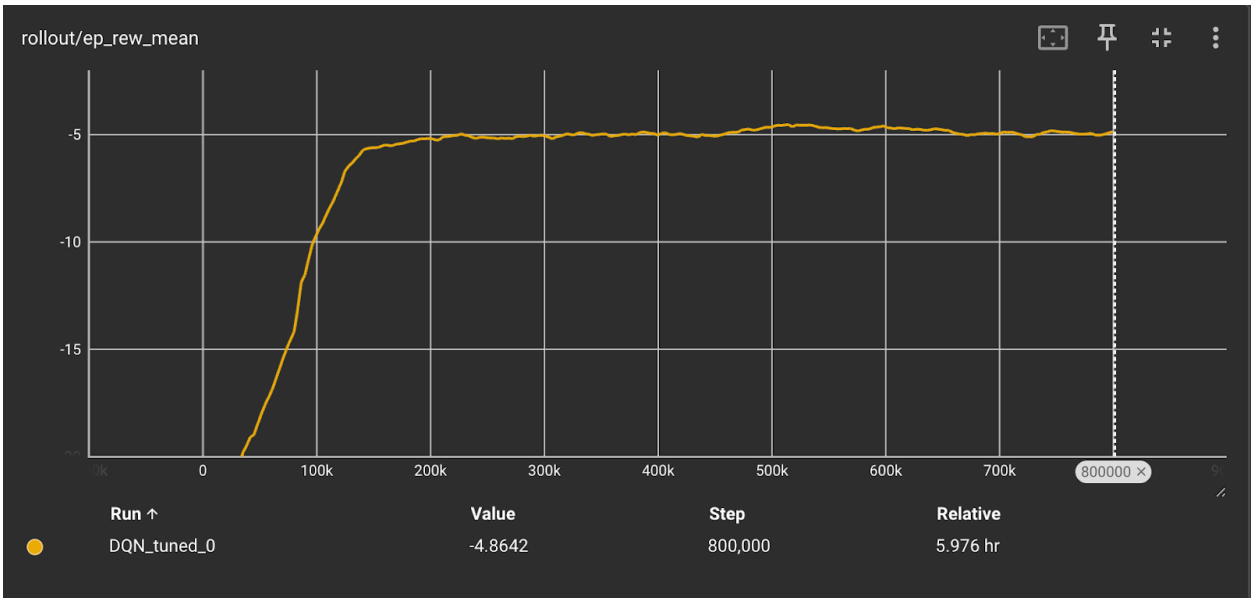

Figure 5: Tuned V2 Model Mean reward as function of steps

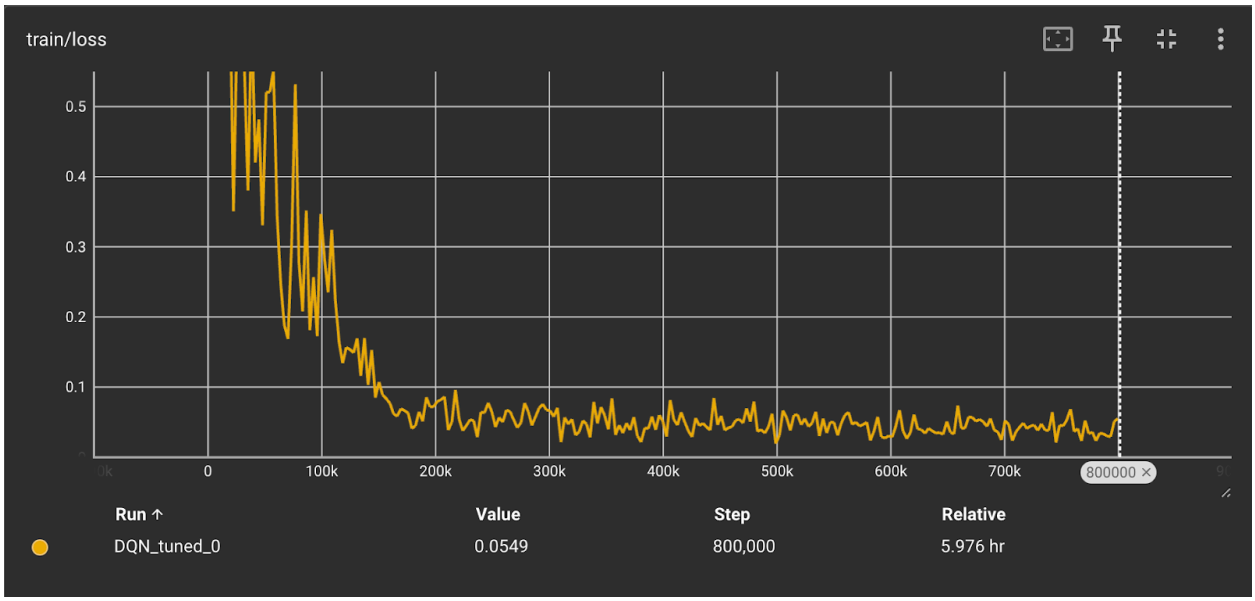

Figure 6: Tuned V2 Model Loss value as function of steps

- **Mean reward per episode:** The Fined Tuned V2 model reaches a higher and more stable reward earlier. It also never seems to face a decreasing trend in reward value. This indicates a more stable and efficient learning strategy and possibly a better policy.
    - **Impact of the modifications**:
        - **Learning rate:** The slight increase in learning rate results in a steeper initial increase in reward. Moreover, the reward stabilizes earlier in the training. This suggests that the increased learning rate allowed the model to learn more quickly and achieve better performance early on.
        - **Gamma:** The reward graph shows a more consistent and faster increase, suggesting that focusing on immediate rewards fits better with the training environment.
        - **Buffer size:** The reward graph shows a more stable and consistent increase, indicating that the reduced buffer size allowed the agent to learn more effectively from recent experiences.
        - **Exploration Rates:** A lower initial epsilon led to faster initial improvement, while a higher final epsilon ensured ongoing exploration. This prevented the model from converging to suboptimal policy which would have made the reward value drop (like in Tuned V1 and Base Model)
- **Loss function results per episode:** The loss initially spikes but stabilizes at a lower level compared to the other models.
    - **Impact of the modifications**:
        - **Learning rate:** The slight increase in learning rate results in a steeper initial increase in reward. Moreover, the reward stabilizes earlier in the training. This suggests that the increased learning rate allowed the model to learn more quickly and achieve better performance early on.

- **Gamma:** The reward graph shows a more consistent and faster increase, suggesting that focusing on immediate rewards fits better with the training environment.
- **Buffer size:** The reward graph shows a more stable and consistent increase, indicating that the reduced buffer size allowed the agent to learn more effectively from recent experiences.
- **Exploration Rates:** A lower initial epsilon led to faster initial improvement, while a higher final epsilon ensured ongoing exploration. This prevented the model from converging to suboptimal policy, which would have made the reward value drop (like in Tuned V1 and Base Model)

Final judgment: For the final judgement, we decided to compare both tuned models by deploying the trained models in different environment settings compared to where they trained in. The traffic flow was randomized (see file *random.rou.xml*)The simulation will run for 100K steps, and the total waiting time of the system will be calculated along the way. This will serve as the basis of our judgement:

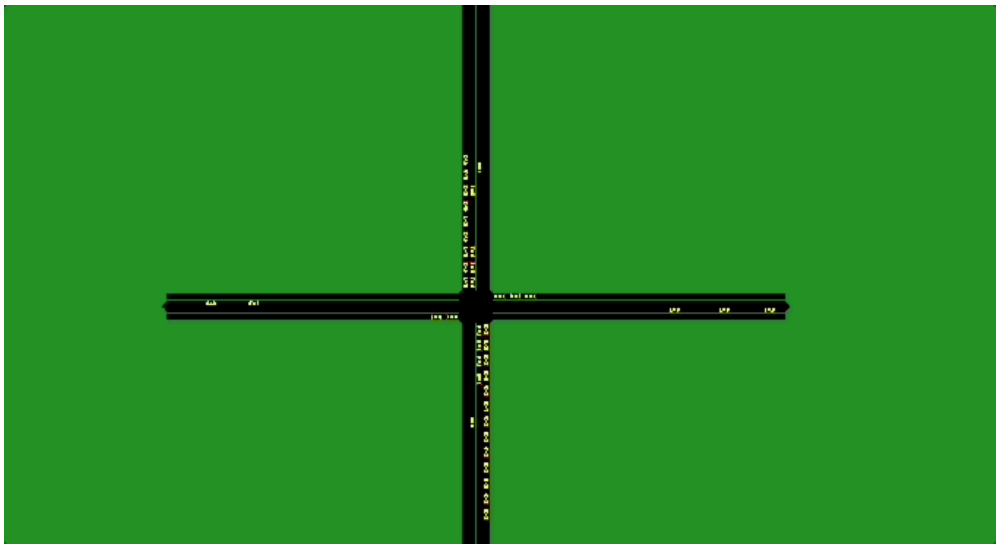

Figure 7: Simulation footage

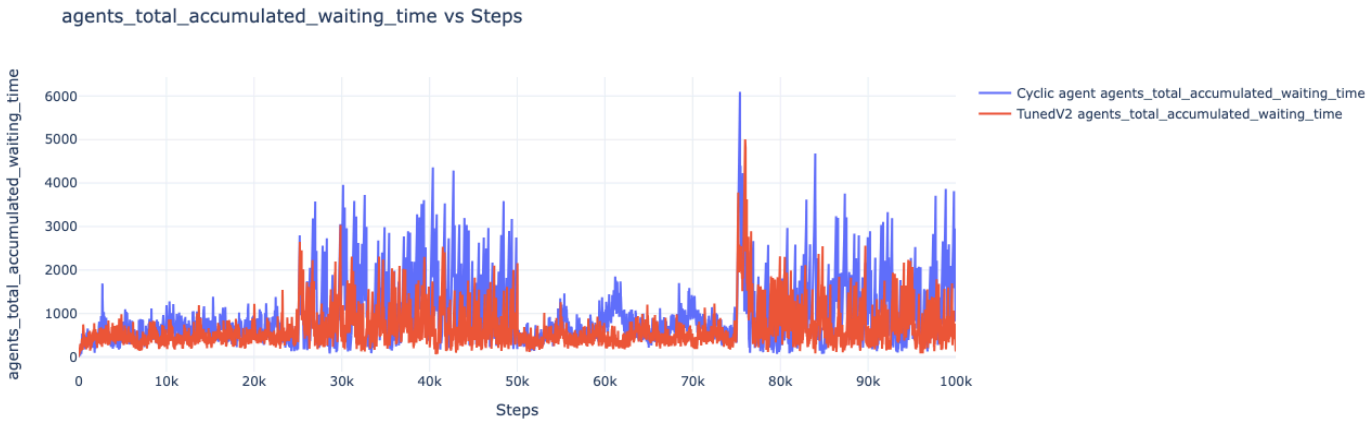

Figure 8: Total waiting time of the intersection as function of steps

- The results align with our analysis made during the training phase. Indeed, both models seem to react similarly when it comes to the amount of traffic, but the TunedV2 model always seems to better minimize the waiting time. This difference is even more clear when we look at the mean waiting time for both models (10.67 steps vs. 14.68 steps).

Independent evaluation: For the independent evaluation, we will compare our model's performance with that of a static traffic light control policy. We made a custom agent that simply switches lights in a cyclic manner. The agent is deployed in the same randomized simulation used for the final judgment. This will enable us to independently compare our chosen model's performance with a more traditional policy.

Results:

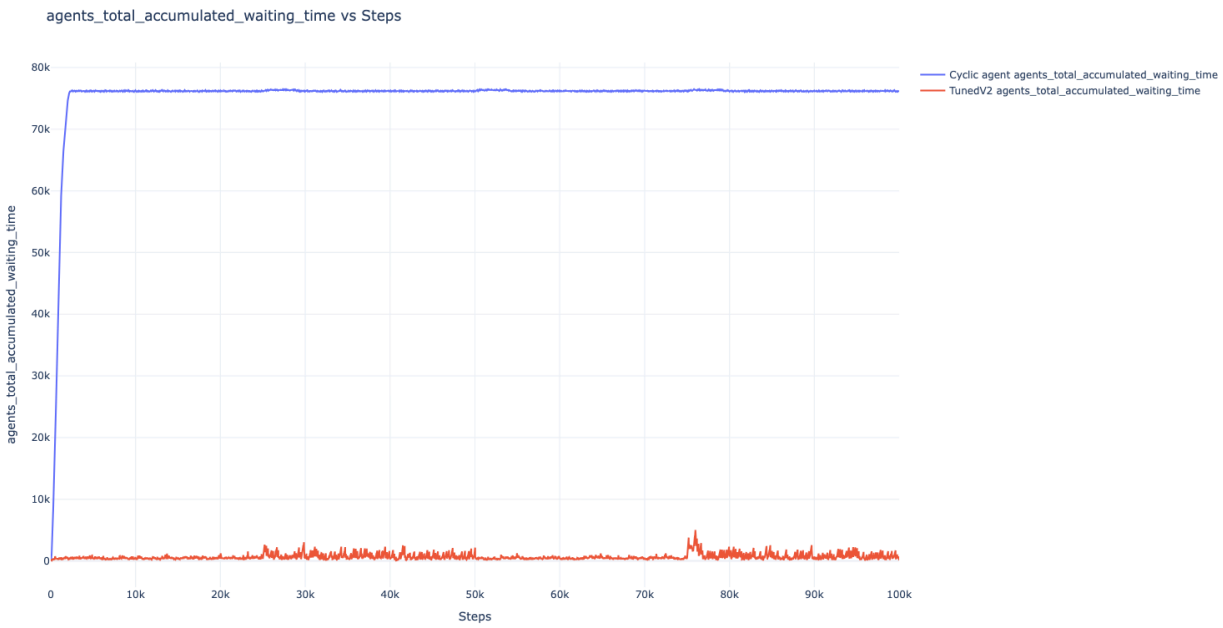

Figure 9: Total waiting time of the intersection as function of steps with cyclic agent

Conclusion : The results of our reinforcement learning model for traffic light optimization, as depicted in Figure 9, clearly validate our hypothesis. The comparison between the traditional cyclic agent and our tuned DQN model highlights a significant improvement in performance.Our tuned DQN V2 model, represented in red, consistently outperformed the traditional cyclic agent in blue across 100,000 simulation steps. In contrast with the cyclic agent, our DQN model maintained a low and stable cumulative waiting time, effectively minimizing congestion and improving traffic throughput. Additionnally, the model demonstrated an ability to adapt.Indeed, whenever the agent encountered a peak in waiting time, it consistently managed to bring it down efficiently.