REINFORCEMENT LEARNING

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

AUTONOMOUS DRIVING PROJECT – SOLVING THE HIGHWAY ENVIRONMENT

Project done by:

Alícia Santos, 20230525

Filipe Rodrigues, m20201866

Gonçalo Ferreira, 20230492

Inês Castelhano, 20230478

June 2024

Number of Words: 2988

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Table of Contents

Index of Figures

Index of Tables

1. Introduction

Reinforcement Learning is an area of Machine Learning, where the system learns by trial and error. Here, the system is called the **Agent**, and it learns to make intelligent decisions, by interacting with the **Environment**, which is its surroundings. An **Agent**, in every moment is in a given **State**. It can change **States**, with this phenomenon being called an **Action**. The number of **States** is called the **Observation Space**, and all the possible combinations of **Actions** by the **Agent** are called **Action Space**. The **Agent** learns if the **Action** he took is good or bad, based on the **Reward** received by the Action. The **Reward** is a numerical value, which can be negative, null or positive. An **Agent** attempts to maximize its score, so it learns the best course of action by achieving the highest score, and trying to avoid actions which decrease or don't even increase its score.

Autonomous driving is a field with conditions for Reinforcement Learning to thrive. With this in mind, for this project, we were given the task of solving the **Highway Environment.**

So, what is the Highway Environment? This is an Environment, where the Agent (green vehicle) drives in a populated multilane, as we can see in Figure 1 and must increase its speed while avoiding colliding with the blue cars. In addition, driving on the right side of the road is rewarded.

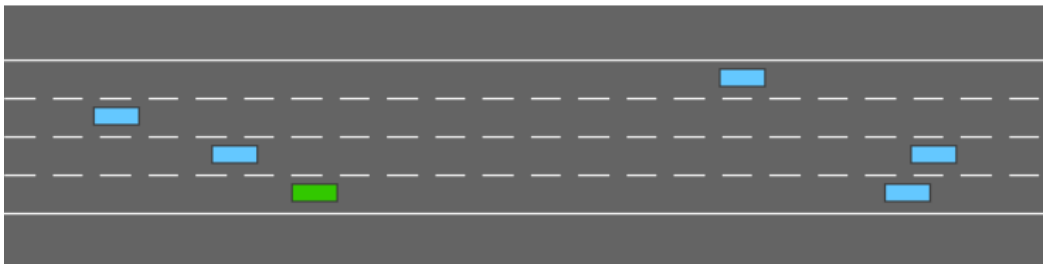

Figure 1. Highway Environment

The Reward that the Agent receives for its Action, is as follows in the figure 2:

$$R(s,a) = a\frac{v - v_{\min}}{v_{\max} - v_{\min}} - b\,\text{collision}$$

where v, v_{min}, v_{max} are the current, minimum, and maximum speed of the Agent respectively, and a, b are two coefficients, with a being the velocity's coefficient and b being the collision's coefficient.

Figure 2. Reward's Formula

In addition, the Reward is normalized to help create a more stable and efficient learning environment.

The Highway Environment has 4 different types of Observation Spaces, which have the following characteristics:

Observation Space	Range of Element	Shape	Data Type
Kinematics	[-∞;∞]	(5,5)	Float32
Gray Scale	[-∞;∞]	(4,128,64)	Uint8
Time to Collision	[0;1]	(3,3,10)	Float32
Occupancy Grid	[-∞;∞]	(4,11,11)	Float32

Table 1. Possible Observation Spaces for the Highway Environment.

It is important to note that each Observation Space is continuous in its interval. For Kinematics, Gray Scale and Occupancy Grid we have [-∞;∞] as interval, meaning we have no constraint values. On the other hand, Time to Collision can only take values within the [0;1] range.

Regarding the Action Spaces, this environment comes with 3 different ones, which are:

- **Discrete Meta Action,** a discrete action space with 5 possible actions.
- **Discrete Action,** a discrete action space with 9 possible actions.
- **Continuous Action,** a 2-dimensional continuous action space in the range [-1;1].

2. Solving the Environment

For this project, apart from solving the Highway Environment, we needed to comply with rules and parametrization, as follows:

- Solve with at least 3 different algorithms.
- Use 2 different Action Spaces.
- Solve 2 different Observation Spaces.
- Comply with the parameters defined, namely 4 lanes, an environment duration of 40 seconds, 50 other vehicles within the environment and an initial spacing of 2 between the vehicles.

2.1 Solution 1

For the first solution, **SARSA (**State-Action-Reward-State-Action) was chosen. This algorithm is an extension of the Q-learning algorithm and is used to learn the optimal policy for an agent, while interacting with an environment. It follows the logic presented in figure 3:

$$q'(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[R_{t+1} + \gamma . q(S_{t+1}, A_{t+1}) - q(S_t, A_t)]$$

q'(St,At)-next step; **q(St,At)-** current step; **α**- learning rate; **γ**-discount factor;
q(St+1,At+1)- outcome of certain step

Figure 3. SARSA's Formula

The goal of the SARSA update rule is to iteratively improve the Q-value estimates so that they converge to the true expected rewards over time. Let's break it down in steps:

Step	Description
1	The agent takes an action and moves to a new state and observes the expected future reward regarding the new state.
2	The updated rule adjusts the current Q-value based on this new information.
3	The learning rate (α) ensures that the updates are gradual, meaning the agent will learn steadily without making drastic changes based on a single event.
4	The discount factor (γ) makes sure that future rewards are considered but are somewhat less important than immediate rewards, preventing infinite accumulation of future possible rewards.

Table 2. SARSA's Update Process

By repeatedly applying this update rule during the agent's interactions with the environment, the Q-values converge to the true values, enabling the agent to learn an optimal policy for maximizing cumulative rewards. This is the reason why SARSA was chosen, as per its equation, considers all outcomes of the next move. If it crashes, it will easily learn that it is wrong, and take that into consideration. In addition, if it not driving at the right lane, it will understand that its reward system is increasing a lot. For these reasons, we deem Sarsa as a good starting point.

2.1.1 Specific Configuration

In the initial phase, Kinematics for observation space and DiscreteMetaAction were chosen. The observation space kinematics, allows the agent to make precise and safe manoeuvres, improving the vehicle's performance in navigating through traffic and avoiding collisions, due to its continuous space. Moreover, DiscreteMetaAction was chosen for its ability to reduce complexity and easier policy learning.

The agent was trained for 500 iterations. In theory, the more iterations, the better the agent's performance will be. However, due to computational and time constraints, 500 iterations were chosen as the optimal number, considering this trade-off. Additionally, a learning rate of 0.1 was selected, meaning that the Q-value is updated by incorporating 10% of the new information (the difference between the predicted and actual reward). Furthermore, a discount factor (gamma) of 0.9 was chosen, indicating that the agent assigns significant importance to future rewards, while slightly values more immediate rewards. Moreover, 200 was stipulated as max_steps_per_episode, to ensure termination after a considerable number of steps, and 10 bins were chosen for discretizing the continuous observation space, as it is necessary. This step simplifies the problem and fosters learning.

2.1.2 Results and analysis

The SARSA agent was tested over 500 episodes to evaluate its performance after the training phase.
The results were:

- **Average reward**: 37.99
- **Mean episode length**: 39.67 steps
- **Total evaluation time:** approximately 8961 seconds

The graph from the testing phase shows positive results, indicating that the agent is generally learning. This demonstrates consistency in performance, suggesting that the agent's policy is satisfactory in maximizing rewards, as evidenced by the reward line being close to the mean reward line. However, there are dips in some episodes, indicating that the agent occasionally encountered suboptimal situations, such as collisions or failing to maintain the desired speed range, leading to lower rewards in those episodes. These failures, although infrequent, highlight areas where the policy could be further refined for more robust performance, potentially through additional training or refining the discretization process.

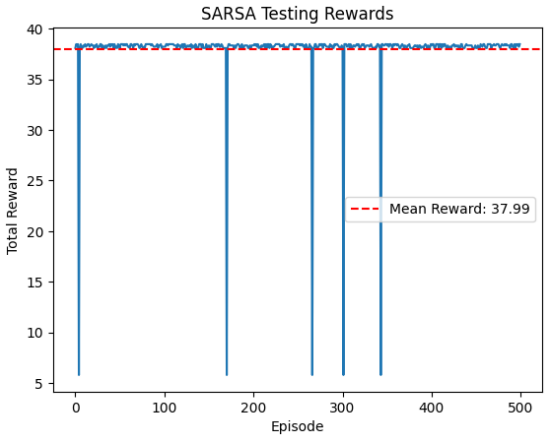

Figure 4. SARSA's Testing Rewards.

Finally, it is possible to infer that the on-policy nature of SARSA, combined with its simplicity and balanced exploration-exploitation strategy, showcased a promising solution to solve the highway environment.

2.2 Solution 2

For the 2nd solution, we went with Proximal Policy Optimization (PPO). PPO is a state-of-the-art reinforcement learning algorithm that combines the advantages of policy gradient methods with stability mechanisms to ensure robust learning. Its key innovation, the clipped surrogate objective, makes PPO both powerful and practical, leading to widespread adoption in various complex RL applications. Figure 5 and Table 3 allow us to understand better how PPO works.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta))\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)]$$

Figure 5. PPO's Formula.

Parameter	Meaning
θ	Policy Parameters that aimed to optimize
E^t	The expectation over timesteps t. This means the expression is being averaged inside the brackets over all timesteps.
Rt	Ratio of the probability under the new and old policies, respectively.
A^t	The advantage estimate at timestep t. It measures how much better or worse the taken action *at* was compared to the average action in that state. Positive values indicate better-than-average actions, and negative values indicate worse-than-average actions.
clip(rt(θ),1−ϵ,1+ϵ)	This function clips the probability ratio to be within the range [1−ϵ,1+ϵ]. **ϵ** is a small positive hyperparameter, typically around 0.1 to 0.2. The clipping prevents the ratio from deviating too much from 1, ensuring that the new policy does not diverge significantly from the old policy.
min(rt(θ)A^t,clip(rt(θ),1−ϵ,1+ϵ)A^t)	This expression takes the minimum of the unclipped and clipped objective values. This ensures that the improvement step (policy update) does not exceed a certain limit, providing a balance between exploring new policies and maintaining stability.

Table 3. PPO's Formula explained by Parameter

The PPO objective function seeks to maximize the expected advantage while ensuring that the new policy does not deviate excessively from the old policy. By taking the minimum of the unclipped and clipped terms, PPO effectively balances policy improvement with stability, leading to more robust and reliable learning.

PPO was preferred due to its characteristics, namely the fact that by limiting the policy update to $[1-\epsilon, 1+\epsilon]$, it prevents large updates that could lead to instability. This stability is crucial in the highway environment where smooth and predictable driving behaviours are essential.

2.2.1 Specific Configuration

Once again, Kinematics and DiscreteMetaAction were chosen. Kinematics was selected to pair with PPO's capability to manage continuous spaces effectively, making it suitable for utilizing detailed information to accurately predict and plan future states. Additionally, DiscreteMetaAction was selected due to its simplified action space, enabling PPO to concentrate on learning effective strategies for executing discrete maneuvers such as lane changes or speed adjustments.

Additionally, attention mechanisms were implemented, allowing models to apply multi-head self-attention and dynamically focus on relevant parts of the input data, including masking and dropout for robust learning. Thus, an EgoAttentionNetwork was created to integrate both MLP and the attention mechanism into a cohesive network, which was applied in a customized policy network for the PPO model.

For this solution, n_steps = 512 were used to prevent variance in policy updates. This means that the algorithm only updates its policy after 512 iterations. Additionally, the batch_size was set to 64, which determines the number of samples per gradient update. This balances computational efficiency and stability, allowing frequent updates with manageable noise in gradient estimates.

2.2.2 Results and analysis

The PPO model was tested over 1000 episodes to evaluate its performance after the training phase.
The results were:

- **Average reward**: 55.48
- **Mean episode length**: 79.30
- **Total evaluation time:** approximately 15454 seconds

The provided results and plot show that the PPO agent consistently achieves high rewards, close to the maximum, in most episodes. This demonstrates a high level of consistency in its performance. However, there are occasional significant drops in reward, indicating instances where the agent failed to avoid collisions and often did not change to the right lane when necessary. These drops suggest that the configuration could be improved to mitigate such issues or that the attention policy could benefit from additional layers.

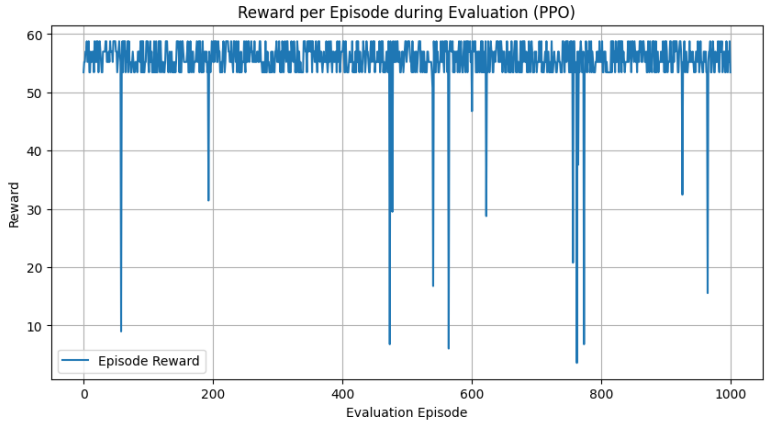

Figure 6. PPO's Testing Rewards.

Finally, PPO leverages improved perception with dynamic focus, showing robustness by concentrating on relevant features. The positive results highlight PPO as a strong candidate for real-time decision-making applications.

2.3 Solution 3

In this solution, the Deep Q-Network (DQN) algorithm was implemented to improve the limitations of traditional Q-Learning, such as its difficulty in handling large state spaces and its inability to estimate Q-values for unknown states. Instead of using Q-tables, DQN uses a deep neural network to approximate the state-value function. The neural network maps input states to action-value pairs, allowing the model to predict Q-values with increasing accuracy over time. The DQN interacts with an environment that has various states, actions, and rewards. It stores past experiences in replay memory, capturing state, action, reward, and next state. These experiences are randomly sampled to train the network. The neural network estimates Q-values to evaluate the performance of actions in specific states. As training progresses, the model learns to predict these values more accurately. An epsilon-greedy strategy was used to balance exploration and exploitation. Initially, actions are chosen randomly (high epsilon) to explore the environment. Over time, epsilon decreases, favouring actions with higher Q-values (exploitation). DQN also uses a target network, a less frequently updated copy of the main network, to stabilize training and prevent overestimation of Q-values. During training, the main network is updated using the Bellman equation to estimate optimal Q-values. The loss function measures the mean squared error between predicted and target Q-values. Weights are updated through backpropagation and stochastic gradient descent. Once training is complete, the learned policy is used to make optimal decisions. For this project, the Stable Baselines3 library was used due to its simplicity, state-of-the-art algorithms, extensive documentation, and community support. Built on PyTorch, it offers better performance and flexibility, especially with GPU support. Stable Baselines3 also emphasizes reproducibility, allowing for consistent and standardized algorithm implementation. Training stops when specific criteria are met, such as a defined number of timesteps, episodes, convergence, or time limitations. This approach enables us to compare the performance of different algorithms more effectively.

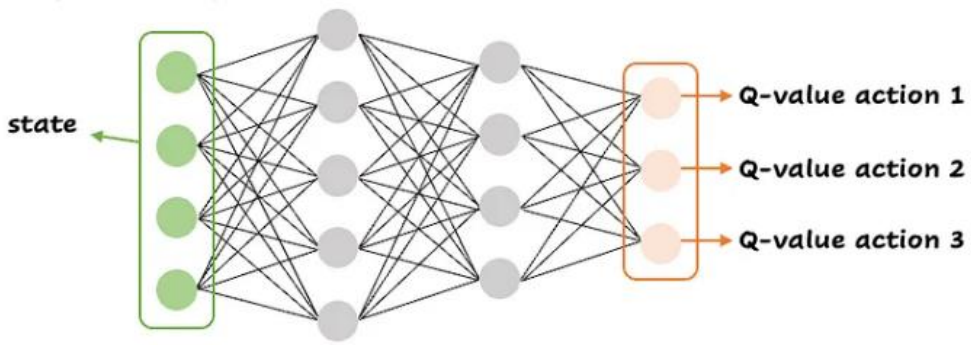

Figure 7 - Implementation of neural network in Q-Learning

2.3.1 Specific configuration

In this solution, kinematics was chosen for observations and a discrete action space for the agent's decisions. A DQN (Deep Q-Network) model, with the "MlpPolicy" (Multi-Layer Perceptron) was used for 5000 timesteps, with regular progress logging. After training, we saved and loaded the model for evaluation (detailed in the next chapter).

Kinematics were used to observe physical dynamics like position, velocity, and acceleration, essential for informed driving decisions. The discrete action space simplified decision-making, allowing the agent to choose from a limited set of actions, such as lane changes or speed adjustments. This choice complements the DQN algorithm, which performs well with discrete actions. Combining kinematics with discrete actions aimed to accelerate training and convergence, providing realistic driving data without complicating the learning process.

2.3.2 Results and analysis

At the evaluation step, we assessed our trained DQN model over 1000 evaluation episodes, each with a maximum of 100 steps to limit the evaluation duration. The primary performance metrics were the mean reward per episode and the mean episode length, which indicate the agent's effectiveness in achieving high rewards and maintaining performance. We also recorded the total evaluation time.

The results were as follows:

- **Mean reward:** 10.019
- **Mean episode length:** 11.779 steps
- **Total evaluation time:** approximately 5788 seconds

The learning curve, plotted as total rewards over evaluation episodes, showed a positive slope, indicating consistent improvement in the agent's performance. However, the absence of a plateau suggests that the agent hasn't yet converged to its optimal policy and requires further learning. To achieve convergence, we could consider hyperparameter tuning, increased training time, and reward function modification.

Additionally, we observed that the agent tended to stay in the same lane, indicating a strategy focused on safety and stability, reducing collisions and steadily accumulating rewards. However, it also suggests a conservative approach, avoiding lane changes unless necessary.

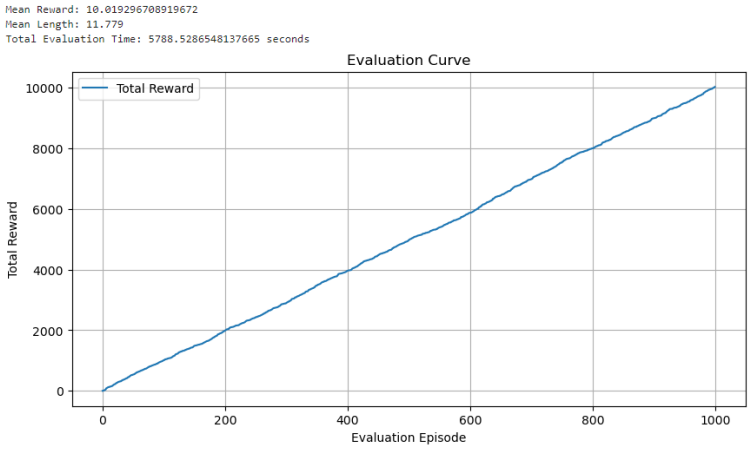

Figure 8 - Learning curve for DQN

2.4 Solution 4

In this solution, the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm was implemented, which improves DDPG by addressing its tendency to overestimate Q-values. TD3, suitable for continuous action spaces, is an off-policy algorithm where the optimal policy is learned independently of the agent's actions.

TD3 incorporates three key improvements over DDPG:

- Double Q-Learning: The algorithm learns two Q-functions instead of one and uses the smaller Q-value to create targets in the Bellman error loss function.
- Delayed Policy Updates: The policy and target networks are updated less frequently than the Q-function, typically one policy update for every two Q-function updates.
- Target Policy Smoothing: Noise is added to the target action to minimize the likelihood of the policy exploiting errors in the Q-function, smoothing out Q-values as actions change.

These enhancements help prevent the overestimation of Q-values, leading to more stable and reliable policy updates.

Now that we detailed the improvements of TD3 regarding the original model DDPG, let's give a brief detail of what fully happens in this new refined algorithm:

- The process starts with the **initialization** of 2 critic networks (Q_{θ_1} and Q_{θ_2}) and 1 actor network $\pi_{\emptyset}$, both with random weights. We will also have 3 target networks as copies of the previous networks (Q'_{θ_1}, Q'_{θ_2} and $\pi'_{\emptyset}$, respectively), which will also be initialized. To store the experience tuples related to state, action, reward and next state, a replay buffer is also set up.
- At each time step, the actor network $\pi_{\emptyset}$ is set to perform an action accordingly to the current state (to introduce exploration noise). The action executed in the environment will unlock a reward, as well as the reception of the next state, which are then stored into the replay buffer.

- From time to time, a batch of experiences in sampled from the replay buffer and in each of those samples, noise is added to the target action so that the estimations of Q-values are smoothen, preventing overestimation.

The following table presents the equations executed during this process:

Equation	Function
$y = r + \gamma \min\left(Q'_{\theta_1}(s', \pi'_{\phi}(s') + \epsilon), Q'_{\theta_2}(s', \pi'_{\phi}(s') + \epsilon)\right)$ $\epsilon - noise; r - reward; \gamma - discount\ factor; s' - next\ state.$	Calculate the target Q-value using the target networks.
$L(\theta_i) = \frac{1}{N}\sum(Q_{\theta_i}(s, a) - y)^2$	Update the critic networks by minimizing the loss between the predicted and the target Q-values.
$\nabla_{\phi} J(\phi) = \frac{1}{N}\sum \nabla_a Q_{\theta_1}(s, a)\|_{a=\pi_{\phi}(s)} \nabla_{\phi} \pi_{\phi}(s)$	To stabilize the training, the actor network updates using the gradient of the Q-value.
$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$ $\tau - parameter\ to\ control\ update\ rate$	To softly update the target network, blends both weights of the original and the target networks.

Table 4. TD3's Formula.

Like in DQN, here the algorithm stops when certain conditions/criteria are met (examples previously referred are valid here).

2.4.1 Specific configuration

In this solution, we configured a highway driving environment using an occupancy grid for observations and continuous actions for the agent's decisions. The model was trained for 10,000 timesteps, and the trained model was saved for later evaluation.

An occupancy grid was chosen for its detailed, grid-based representation of the environment, showing the locations of other vehicles and obstacles. This helps the agent understand spatial relationships crucial in complex scenarios. Continuous action space was selected to allow precise adjustments in acceleration and steering, providing refined vehicle control. This approach suits TD3 well, as it excels in settings requiring fine-grained control.

Combining occupancy grid observations with continuous actions enables the TD3 model to understand the driving environment and achieve precise control. However, this increases learning complexity, as the agent must accurately interpret the occupancy grid and select optimal continuous actions, making the process more computationally demanding.

2.4.2 Results and analysis

During the evaluation phase, the performance of the trained TD3 model was analyzed over 1000 evaluation episodes, each limited to 100 steps. The primary metrics were average reward per episode and mean episode length.
The results were:

- **Average reward**: 60.423
- **Mean episode length**: 100 steps
- **Total evaluation time:** approximately 48802 seconds

The learning curve displayed fluctuating peaks and troughs, resembling an incomplete rectangle (Figure 9), indicating significant performance variability. The high mean reward suggests

satisfactory overall performance, while the stable episode length shows consistent action sequences without premature termination.
However, the agent exhibited a perpetual circular driving behavior, indicating a stable yet suboptimal strategy. This suggests the agent exploited the reward function without fully mastering the driving task. Potential solutions include hyperparameter tuning, increased training time, and reward function modification.

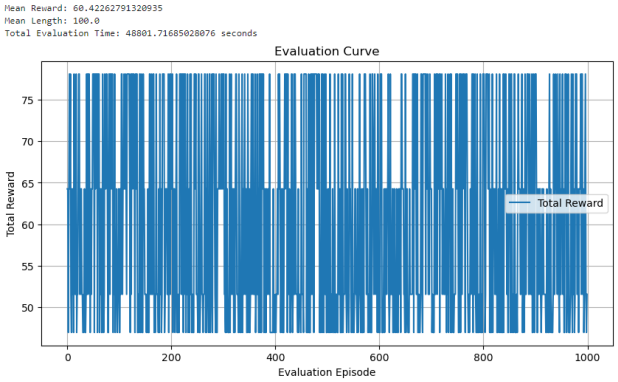

Figure 9 - Learning curve for TD3

3.Conclusion

Throughout this project, we effectively leveraged the knowledge gained from our Reinforcement Learning course and supplemented it with insights from self-directed study. Despite encountering significant challenges, including prolonged algorithm execution times, and limited hyperparameter tuning, our experience proved to be immensely educational. These difficulties were compounded by our initial lack of expertise in Reinforcement Learning and the constraints imposed by our computing resources, which delayed our progress.

One significant setback was our unsuccessful attempt to implement CUDA for GPU-based parallel processing, which could have significantly reduced training and testing times. This remains a crucial area for improvement.

Regardless of significant challenges, satisfactory solutions were achievable with the algorithms we employed. PPO demonstrated the most consistency in final results and efficiency, with a good trade-off between mean reward and episode length, making it the best solution. However, TD3 yielded the highest rewards, although its performance in driving tasks was suboptimal.

Furthermore, to enhance model performance, it would be beneficial to explore and compare a broader range of reinforcement learning algorithms beyond those initially used, such as SAC (Soft Actor-Critic) or A3C (Asynchronous Advantage Actor-Critic). These algorithms may offer improved performance or faster convergence rates. Additionally, implementing systematic hyperparameter tuning methods could help identify optimal settings for the chosen algorithms. In addition, some techniques such as transfer learning could also accelerate convergence by leveraging knowledge from related tasks or pre-trained models or exploring multi-agent approaches to enable the interaction and cooperation between the vehicles in the simulated environment.

In future research, integrating these strategies can enhance the effectiveness, efficiency, and robustness of the model, leading to significant advancements in autonomous driving technology.

4. References

(2015). Human-level control through deep reinforcement learning [Review of *Human-level control through deep reinforcement learning*]. *Macmillan Publishers Limited*, *518*. https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning* [Review of *Playing Atari with Deep Reinforcement Learning*]. https://arxiv.org/pdf/1312.5602v1

Li, Q., & Choe, Y. (2024). Emergence of tool construction and tool use through hierarchical reinforcement learning [Review of *Emergence of tool construction and tool use through hierarchical reinforcement learning*]. *Academic Press*, 325–341. https://www.sciencedirect.com/science/article/abs/pii/B9780323961042000087

Harder, H. (2022, November 30). *Techniques to Improve the Performance of a DQN Agent* [Review of *Techniques to Improve the Performance of a DQN Agent*]. Towards Data Science. https://towardsdatascience.com/techniques-to-improve-the-performance-of-a-dqn-agent-29da8a7a0a7e

Byrne, D. (2019, June 15). *TD3: Learning To Run with AI* [Review of *TD3: Learning To Run With AI*]. Towards Data Science. https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93

Twin Delayed DDPG — Spinning Up documentation. (n.d.). Spinningup.openai.com. https://spinningup.openai.com/en/latest/algorithms/td3.html

Suran, A. (2020, July 14). *On-Policy v/s Off-Policy Learning*. Medium. https://towardsdatascience.com/on-policy-v-s-off-policy-learning-75089916bc2f

SARSA Reinforcement Learning Algorithm | Built In. (n.d.). Builtin.com. https://builtin.com/machine-learning/sarsa

Gautam, A. (2023, July 31). *SARSA Reinforcement Learning Algorithm: A Guide* [Review of *SARSA Reinforcement Learning Algorithm: A Guide*]. Builtin.com. https://builtin.com/machine-learning/sarsa

AI, O. (2017). *Proximal Policy Optimization Algorithms* [Review of *Proximal Policy Optimization Algorithms*]. https://arxiv.org/pdf/1707.06347