Technical University of Crete

School of Electrical and Computer Engineering

Deep Q-Networks with Normalized Advantage Function For Autonomous Driving in Lane-Free Traffic

Diploma Thesis

Leonidas Bakopoulos

Thesis Committee:

Supervisor	Georgios Chalkiadakis Professor (School of ECE)
Committee Member	Michail G. Lagoudakis, Professor (School of ECE)
Committee Member	Ioannis Papamichail, Professor (School of PEM)

February 14, 2024

Πολυτεχνείο Κρήτης

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Ενισχυτική Μάθηση με Q-Νευρωνικά Δίκτυα με Κανονικοποιημένη Εκτίμηση Πλεονεκτήματος για Αυτόνομη Οδήγηση Χωρίς Λωρίδες Κυκλοφορίας

Διπλωματική Εργασία

Λεωνίδας Μπακόπουλος

Επιτροπή:

Επιβλέπων	Γεώργιος Χαλκιαδάκης, Καθηγητής (Σχολή ΗΜΜΥ)
Μέλος Επιτροπής	Μιχαήλ Γ. Λαγουδάκης, Καθηγητής (Σχολή ΗΜΜΥ)
Μέλος Επιτροπής	Ιωάννης Παπαμιχαήλ, Καθηγητής (Σχολή ΜΠΔ)

14 Φεβρουαρίου 2024

Abstract

In the past decade Deep Reinforcement Learning (Deep-RL) has evolved into a powerful tool that can outperform both human abilities and traditional algorithms in many domains. Deep-RL differs from classic RL in its ability to handle complex problems in larger, and sometimes continuous, action and state spaces. At the same time, the *vehicular traffic* research area is of utmost practical importance. Numerous works have proposed that automated vehicles can optimize traffic flow. Vehicles on the road tend to maintain different desired speeds, leading to various situations requiring overtaking and other appropriate reactions to others' behavior.

Now, in recent years, a novel vehicular traffic paradigm, namely that of *lane-free traffic*, has emerged as a means to utilize the full width of a road by automated and (potentially connected) vehicles. In a lane-free environment, vehicles can be positioned anywhere in the two-dimensional state space, complicating the automated vehicles' decision-making process significantly and making it entirely different from the traditional lane-based approach. Deep RL is a natural candidate to address the challenges posed by this new traffic paradigm.

Against this background, this thesis builds upon recent work by *Karalakou et al. [1]* that enabled the application of the Deep Deterministic Policy Gradients (DDPG) Deep RL algorithm in the lane-free traffic domain. Our work progressively builds an autonomous agent that combines various algorithmic components, having as a basis the Normalized Advantage Functions (NAF) deep RL algorithm. Specifically, we put forward the blending of NAF with Prioritized Experience Replay (PER), Parameter State Noise for Exploration (PSNE), the well-known Boltzmann exploration method, and a local optimization method for exploration; and systematically test our approach in the lane-free highway traffic domain, comparing the performance of various combinations of these algorithmic components against that of the aforementioned DDPG approach. Our simulation experiments' results showcase our approach's superiority to using DDPG; highlight the strengths of each tested algorithmic variant; and demonstrate that our NAF+PER+PSNE variant (in which PSNE is actually combined with Boltzmann exploration) is overall the better method for use in the lane-free traffic scenarios examined.

Περίληψη

Τα περασμένα χρόνια, το Deep Reinforcement Learning (Deep-RL) εξελίχθηκε σε ένα ισχυρό εργαλείο που μπορεί να υπερνικήσει σε πολλούς τομείς τις ικανότητες του ανθρώπου και των τυπικών αλγορίθμων. Το Deep-RL διαφέρει από το κλασικό Reinforcement Learning στην ικανότητά του να αντιμετωπίζει πολύπλοκα προβλήματα που υφίστανται σε μεγαλύτερους και, μερικές φορές, συνεχείς χώρους δράσης και καταστάσεων. Παράλληλα, η έρευνα στον τομέα της *κυκλοφορίας οχημάτων*, είναι υψίστης σημασίας. Αρκετές μελέτες έχουν προτείνει ότι τα αυτόματα οχήματα μπορούν να βελτιστοποιήσουν τη ροή της κυκλοφορίας. Τα οχήματα στον δρόμο τείνουν να διατηρούν διαφορετικές ταχύτητες, το οποίο οδηγεί σε πληθώρα καταστάσεων που απαιτούν κατάλληλες αντιδράσεις, όπως αυτή της ασφαλούς προσπέρασης.

Τα τελευταία χρόνια, το πρότυπο της *κυκλοφορίας χωρίς λωρίδες (lane-free traffic)* έχει εμφανιστεί ως καινοτομία στον τομέα της *κυκλοφορίας οχημάτων*, το οποίο, προβλέπει τη χρήση ολόκληρου του πλάτους ενός δρόμου από αυτόματα και (ενδεχομένως διασυνδεδεμένα) οχήματα. Σε ένα περιβάλλον χωρίς λωρίδες, τα οχήματα μπορούν να τοποθετούνται οπουδήποτε στον δισδιάστατο χώρο, περιπλέκοντας σημαντικά τη διαδικασία λήψης αποφάσεων και διαφοροποιώντας την από την παραδοσιακή προσέγγιση με λωρίδες. Το Deep RL είναι ένας φέρελπις υποψήφιος που θα κληθεί να αντιμετωπίσει τις προκλήσεις που προκύπτουν από αυτόν τον νέο σχεδιασμό κυκλοφορίας.

Δεδομένου του παραπάνω πλαισίου, η παρούσα διατριβή βασίζεται στην πρόσφατη εργασία των *Karalakou et al. [1]* που επέτρεψε την εφαρμογή του αλγορίθμου Deep Deterministic Policy Gradients (DDPG) του Deep RL στον τομέα της κυκλοφορίας χωρίς λωρίδες. Η εργασία μας, χτίζει βαθμιαία ένα αυτόνομο πράκτορα που συνδυάζει διάφορα αλγοριθμικά στοιχεία, έχοντας ως βάση τον Deep-RL αλγόριθμο Normalized Advantage Functions (NAF). Συγκεκριμένα, προτείνουμε τον συνδυασμό του NAF με το Prioritized Experience Replay (PER), το Parameter State Noise for Exploration (PSNE), την δημοφιλή μέθοδο εξερεύνησης Boltzmann, και μια μέθοδο τοπικής βελτιστοποίησης. Στη συνέχεια, ελέγχουμε μεθοδικά την προσέγγισή μας χρησιμοποιώντας ως περιβάλλον έναν αυτοκινητόδρομο χωρίς λωρίδες, και συγκρίνουμε την απόδοσή της με αυτή της προαναφερθείσας στο έργο [1]. Τα αποτελέσματα των πειραμάτων μας στην προσομοίωση δείχνουν τελικά την υπεροχή της προσέγγισής μας σε σύγκριση με τον DDPG, αναδεικνύοντας τα πλεονεκτήματα κάθε δοκιμασμένης αλγοριθμικής παραλλαγής. Τέλος, τα πειράματα επιδεικνύουν, πως η παραλλαγή που συνδυάζει NAF+PER+PSNE (στην οποία το PSNE συνδυάζεται με τη μέθοδο εξερεύνησης Boltzmann) επιφέρει τα καλύτερα αποτελέσματα μεταξύ των σενάριων κυκλοφορίας χωρίς λωρίδες που εξετάστηκαν.

Acknowledgments

First, I would like to thank my supervisor Prof. Georgios Chalkiadakis, for his guidance throughout this thesis. I would also like to thank Iason Chrysomallis, PhD student at Technical University of Crete, for his contribution and Dimitrios Troullinos, PhD student at Technical University of Crete for his useful comments on this work.

Last but definitely not least, I would like to thank my family, for the consistent support that they offered me during my stay in Chania and also, Alexandra and my friends that inspired me every day, to end the daily tasks to finally hang out with them.

Contents

1 Introduction

Reinforcement Learning (RL) stands at the intersection of Artificial Intelligence and decision-making. Its ability to tackle complex decision-making problems make it a compelling field of study. Nevertheless, its applications were limited to simple problems within constrained environments until recently. Deep Reinforcement Learning (Deep-RL) was essentially introduced in a 2013 paper [2] where the authors combined an Artificial Neural Network (ANN) [3] with Experience Replay [4], to create an extension of tabular Q-learning, called *Deep-Q-Networks learning* (or *DQN*). In the years that followed, Deep-RL evolved into a powerful tool, by both replacing the Q-table with Neural Networks in Reinforcement Learning (RL) algorithms, and, by creating more efficient Replay Buffers that can increase the agent's convergence speed. Also, Deep-RL was proven capable to outperform classic RL in various domains, such as Atari Games[2] and autonomous driving in lane-based environments ([5], [6]).

Now, *vehicular traffic* is a domain of utmost importance, as it has been the primary means of transporting both people and products, making life more comfortable and much faster. Automated vehicles can optimize traffic flow, by employing sophisticated algorithms for efficient routing and adaptive driving strategies. Hence, with the evolution of Deep-RL, automated vehicles can be trained in difficult environments without the use of complex rule-based algorithms. These vehicles, can lead to smoother traffic patterns, minimizing delays and enhancing overall transportation efficiency while also minimizing the traffic accidents resulting from human errors such as distracted driving or impaired judgment.

In this thesis, the agent will act in a lane-free environment, as in the TrafficFluid [7] concept. More precisely, this concept combines lane-free traffic with the concept of *vehicle nudging*, to provide the possibility of designing the traffic flow characteristics in an optimal way, rather than merely describing or modeling them. Researchers inspired by the TrafficFluid concept, have proposed multiple vehicle movement strategies, both using control methods [8] and Deep-RL techniques, like *Deep Deterministic Policy Gradients* [1] and *Imitation learning* using *DQN* [9]. The Deep-RL techniques were built on the top of the Markov Decision Process (MDP) designed [10].

This thesis, builds upon the foundation laid by [1] by replacing *Deep Deterministic Policy Gradients (DDPG)* [11] algorithm with *Normalized Advantage Functions (NAF)* [12]. In more detail, *NAF*, based on *Advantage Learning* [13] separates the Value Function from the Advantage term, in order to calculate easier the best action, in continuous action spaces. After the implementation and the fine-tuning of the *NAF* algorithm, the agent will undergo testing on the same environment and reward function as *DDPG* did. Afterwards, some techniques that utilize better exploration and policy learning will be added, to boost NAF's performance. More precisely, the original Replay Buffer will be replaced by a fine-tuned *Prioritized Experience Replay (PER)* [14] memory buffer and the Boltzmann exploration technique will be combined with *Parameter Space Noise for Exploration (PSNE)* [15] for better exploration of the environment. Finally, a modified version of the proposed exploration will be created and tested against the previous exploration technique.

1.1 Contributions

This thesis focuses on the development of an agent, which combines existing Deep-RL algorithmic components like *NAF*, *PER* and *PSNE*. This agent, was trained to follow a strategy within the aforementioned lane-free MDP. In more detail, it initially implements the Normalized Advantage Function (NAF) algorithm proposed in [12]. At first, some of the agent's hyper-parameters were fine-tuned empirically, while others, such as the Neural Network structure and the exploration rate, were tested to determine the best settings. Afterwards the weights of the reward functions were slightly modified, for the agent to yield better objective rewards, such as deviation from the desired speed and the total number of collisions. Using these modifications, NAF was tested against DDPG [11], as the latter was implemented in [1].

This comparison, showed that the NAF agent, behaves better as it achieves a safer driving style at the same speed as [10]. Building on top of the NAF agent, state of art Deep-RL techniques were added to improve the training agent. More precisely, to test how the training buffer affects the learning process, the existing NAF implementation with the replay memory buffer was compared with an implementation of NAF when combined with the PER buffer. This comparison showed that the NAF+PER combined agent, yields a better mean reward than the simple NAF agent.

PSNE, was also added in order to contribute to the agent's exploration policy. In more detail, the offline optimization for exploration purposes that was proposed in [12], was replaced by the PSNE. To the best of our knowledge, there has been no prior integration of PSNE with the NAF algorithm. This work, proposes and puts forward such an implementation which is a non trivial task. Specifically we propose a combination of *PSNE* and Boltzmann exploration and build it on top of *NAF*. The resulting *NAF+PER+PSNE* implementation, as shown experimentally, outperforms the existing agents, by yielding fewer collisions at the same driving speed as its competitors.

Finally, PSNE was replaced by a local exploration technique, inspired by such a proposal in [12]. Comparing experimentally this exploration technique with PSNE, demonstrated that PSNE can explore the environment better and leads to better results. Also, this comparison proves the difficulty of locally optimizing a reward function without any prior knowledge, in difficult environments such as the lane-free autonomous driving one.

1.2 Thesis Outline

To better understand the structure followed in this thesis, let's discuss each chapter's content. After the introductory Chapter 1, Chapter 2 follows, in which the necessary theoretical background is provided to the reader, by thoroughly explaining Machine Learning (ML) and Reinforcement Learning (RL) as concepts. Afterwards, Deep-RL is introduced as the combination of the above. Then, some required Deep-RL algorithms are described, finally leading to NAF's explanation. In addition to that, PER and PSNE are also described. Chapter 3 follows, containing all information required to understand lane-free environments and the suggested agent's implementation while all the conducted experiments, can be found in Chapter 4. Finally, Chapter 5 consists of the thesis summary and the future work to be performed in this domain.

2 Background and Related Work

2.1 Machine Learning

Machine Learning (ML) is a sub-field of Artificial Intelligence (AI), which focuses on the development of algorithms and models that enable computers to learn from data and make predictions or decisions based on it. Rather than being explicitly programmed to perform a specific task, ML systems are designed to improve their performance over time through experience.

For the sake of clarity and precision, let us establish the following conceptual framework. Consider the example of handwritten digit recognition; each digit can be constructed as an image, so that it can be represented by a vector denoted as 'x', comprising real numbers. The goal is to create a machine that will take such a vector as an input and then produce the identity of the digit (0, . . . , 9) as the output.This is a non-trivial problem due to the wide variability of handwriting. It could be tackled using handcrafted rules or heuristics for distinguishing the digits based on the shapes of the strokes, but in practice, such an approach leads to a proliferation of rules and of exceptions to the rules and so on, and invariably gives poor results.
Far better results can be obtained by adopting a machine learning approach in which a large set of N digits x_1 , . . . , x_N, called a training set, is used to tune the parameters of an adaptive model. The categories of the digits in the training set are known in advance, typically by inspecting them individually and hand-labeling them. We can express the category of a digit using target vector t, which represents the identity of the corresponding digit. Suitable techniques for representing categories in terms of vectors will be discussed later. Note that there is one such target vector t for each digit image x.

The result of running the Machine Learning algorithm can be expressed as a function $y(x)$ which takes a new digit image x as input and generates an output vector y, encoded in the same way as the target vectors. The precise form of the $y(x)$ function is determined during the training phase, also known as the learning phase, based on the training data. Once the model is trained it can then determine the identity of new digit images, which are said to comprise a test set. The ability to categorize correctly new examples that differ from those used for training is known as generalization. In practical applications, the variability of the input vectors will be such that the training data can comprise only a tiny fraction of all possible input vectors, and so generalization is a central goal in pattern recognition [16].

The previous example provided was a classical illustration of supervised learning. But the domain of ML in general consists of three types of learning methods; namely supervised learning, unsupervised learning and Reinforcement Learning (RL) :

- Supervised learning: In supervised learning, the training set consists of pairs of input and desired output (labeled pairs). The goal is for the algorithm to learn a mapping between input and output spaces. Using the previous example as an illustration, the inputs were the handwritten digits (either used as a vector or as an image), the outputs were the generated vector $y(x)$ and the goal was to 'learn' a binary classifier.

- Unsupervised learning: In unsupervised learning, the training set consists of unlabeled inputs, that is, of inputs without any assigned desired output. Unsupervised learning generally aims at discovering properties of the mechanism generating the data. In the example of Fig. 1, the goal of unsupervised learning is to cluster together the two dimensional input points that are proximate to one another, hence assigning a label – the cluster index – to each input point. In Fig. 1, each green circle depicts a cluster [17].

- Reinforcement Learning (RL) will be thoroughly explained in Section 2.2 below.

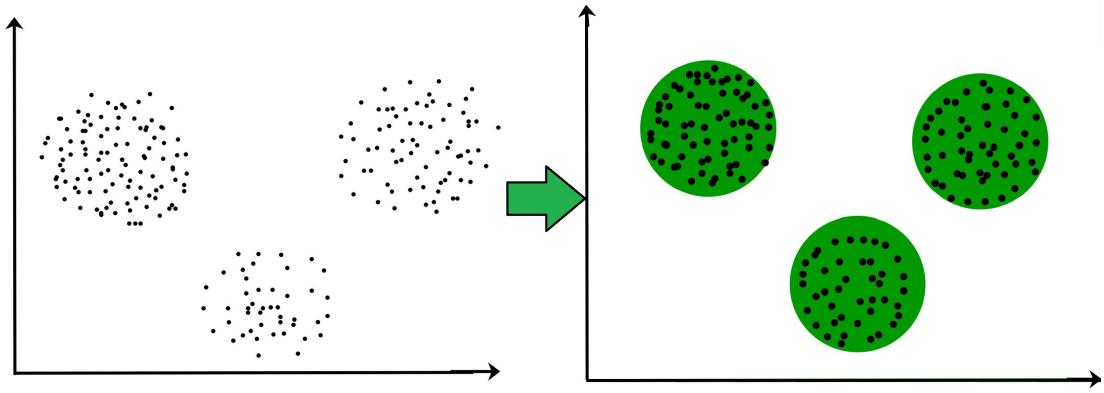


Figure 1: A 2D cluster example

2.2 Reinforcement Learning

Reinforcement Learning (RL) is the process of learning what to do —how to map situations to actions —so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover itself which actions yield the highest reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics, trial-and-error search and delayed reward, are the two most important distinguishing features of reinforcement learning [18].

2.2.1 Markov Decision Process

In order to model this interaction between the learner (also referred as agent) and the environment, a stochastic control process, called Markov Decision Process (MDP) [19] was proposed in 1960. This process, describes the aforementioned communication in a mathematical way.

Definition: A Markov Decision Process (MDP), is a 5-tuple $< S, A, r, P, \gamma >$ where:

- S is the set of states (often called state space, and it can be either discrete or continuous)
- A is the set of actions (often called the action space; it can be either discrete or continuous)
- r is the reward function (see reward function section) typically defined over state-action pairs r: $S \times A \rightarrow \mathbb{R}$
- P is the set of transition probability distributions $P(s, \alpha)$ defined for each $s \epsilon S$ and $\alpha \epsilon A$
- $\gamma \epsilon$ (0,1) is the discount factor

The above definition [20], is also shown in Figure 2.

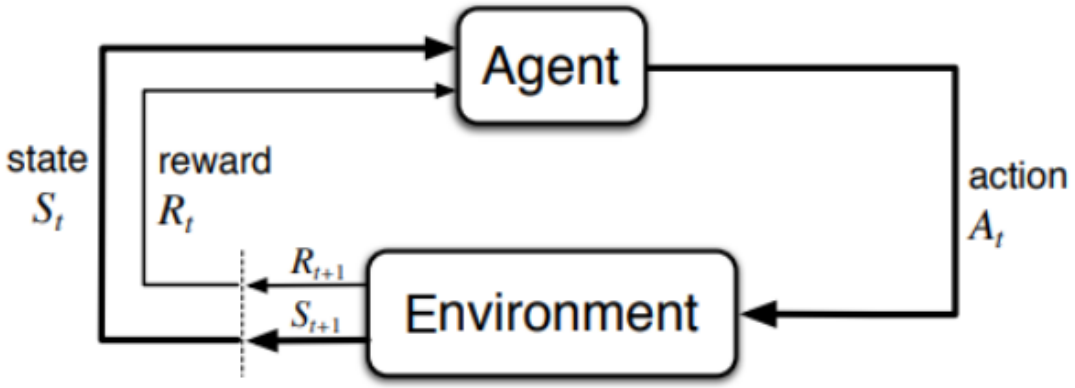


Figure 2: The agent-environment interaction in a MDP. Picture was found in [21]

2.2.2 Q-learning

Reinforcement Learning [18] methods, solve an MDP operating under the assumption that either the reward and/or the transition model are not known, thus some form of learning is required. Model-free RL methods do not attempt to learn the reward or the transition models, but update the Q-function that represents the (expected) sequential performance, based on reward or transition samples they collect from the environment (after acting). Q-learning [3] is a celebrated example of such a model-free RL method. The intuition behind Q-learning, is that an agent tries performing an action at a particular state, subsequently assessing the outcomes based on both the immediate reward (or penalty incurred) and its estimation of the next state's value. By testing all actions in all states sequentially, the algorithm learns which actions are best overall, judging by a long-term discounted reward. Q-learning is a primitive form of learning, but, as such, it can operate as the basis of far more sophisticated devices [3]. However, in order to mathematically express the aforementioned algorithm, some definitions are needed.

A policy is a way of defining the agent's action selection with respect to the changes in the environment. A (probabilistic) policy of an MDP is a mapping from the state space to a distribution over the action space: $\pi : S \times A \rightarrow [0, 1]$. A deterministic policy is a policy that defines a single action per state. That is, $\pi(s) \epsilon A(s)$ [22]. The agent interacts with the environment and takes actions according to the policy.

The Discounted Cumulative Reward represents the sum of future rewards, where each future reward is multiplied by a discount factor to prioritize immediate rewards over delayed ones. More precisely:

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \tag{1}$$

The value function of the policy is defined to be the expectation G_t, given that the agent acts according to a policy $\pi(s)$:

$$V^{\pi}(s) \stackrel{\text{def}}{=} \mathrm{E}[\mathrm{G}|\pi(s), s] = \mathrm{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_t = \pi(s_t)\right] \tag{2}$$

Using the linearity of the expectation, we can write the above expression in a recursive form,

known as the Bellman equation [23]:

$$V(s) = \sum_{a \epsilon A} \pi(s,a) R(s,a) + \sum_{s' \epsilon S} P(s,a,s') V(s') \tag{3}$$

The value function has been used as the primary measure of performance in much of the RL literature. There are, however, some ideas that take the risk or the variance of the return into account as a measure of optimality ([24];[25]). The more common criterion, though, is to assume that the agent is trying to determine a policy which is maximizing the value function. Such a policy is referred to as the optimal policy. We can also define the value function over the state-action pairs. This is usually referred to as the Q-function, or the S-value, of that pair. By definition:

$$Q^{\pi}(s,a) \stackrel{\text{def}}{=} \mathbb{E}[D^{\pi}(s,a)] = \mathbb{E} \sum_{t=0}^{\infty} \left[\gamma^t r_t \,\middle|\, s_0 = s, a_0 = a, t \geq 1 : a_t = \pi(s_t) \right] \tag{4}$$

That is, the Q-value is the expectation of the return, given that the agent starts at state s, takes action a, and then follows policy π. The Q-function also satisfies the Bellman equation:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \epsilon S} P(s,a,s') \sum_{a' \epsilon A} \pi(s',a') Q^{\pi}(s',a') \tag{5}$$

which can be rewritten as: [22]

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s,a,s') V^{\pi}(s') \tag{6}$$

The goal of an algorithm that optimally solves an MDP, should be to predict correctly the Q(s,a) and then act for every state s,

$$a^* = \arg\max_a Q(s,a) \tag{7}$$

Having defined all the components, let's consider a computational agent moving around some discrete, finite world, choosing one of a finite collection of actions at every time step. The aforementioned algorithm in the n^{th} episode can be expressed in [3] as:

Algorithm 1 Q-Learning in n^{th} iteration [26]

1: Observe the current state x_n
2: Select and perform an action a_n
3: Observe the subsequent state r_n
4: Adjust the Q_{n-1} values using a learning rate λ, according to:

$$Q_n(x,a) = \begin{cases} (1-\lambda) Q_{n-1}(x,a) + \lambda [r_n + \gamma V_{n-1}(y_n)] & \text{if } x = x_n, a = a_n \\ Q_{n-1}(x,a) & \text{if } otherwise \end{cases} \tag{8}$$

where

$$V_{n-1}(y) = \max_b \{Q_{n-1}(y,b)\} \tag{9}$$

In practice, Equation 8 can be re-written as

$$Q(s,a) \leftarrow Q(s,a) + \alpha \times TD\_Error \tag{10}$$

where

$$TD\_Error = R_{t+1} + \gamma \max_a Q(S_{t+1,a}) - Q(S_t, A_t) \tag{11}$$

Hence Q-Learning, can be characterized as an off-policy, Temporal Difference control algorithm.

It should be mentioned that in the early stages of the learning process, the Q-values may not accurately reflect the policy they implicitly define. Q-learning is guaranteed to converge, at least for a discrete case with a finite number of states and actions [13]. The Q-table, should be imagined as an approximation of the Q function, limited in a small state space. In this form, the tabular implementation of Q-learning, is impractical for a continuous state space, because an infinite number of entries will be needed in order to represent accurately all possible states. For Q-learning, to be applied in a continuous time, the equivalent to the Bellman equation (Equation 6), should be modified to:

$$Q(x,u) = R\Delta t(x,u) + \gamma^{\Delta t} \sum P(u,x,x') \max Q(x',u') \tag{12}$$

Now the Δt could approach zero, to simulate the continuous time. If the Q function is stored in a function approximation system with some form of error, the implied policy will tend to be sensitive to that error. As the time step duration Δt approaches zero, the penalty for a wrong action taken in a sequence decreases, the Q values for different actions in a given state become closer, and the implied policy becomes even more sensitive to noise or function approximation error. In that case, Q-learning would be expected to slowly learn when the time steps are of short duration, due to its sensitivity to errors, and it would be incapable of learning for continuous time [13]. To solve this, a new method is needed.

2.2.3 Advantage Learning/ Advantage Updating Algorithm

The Advantage Updating algorithm is a Reinforcement Learning algorithm in which two types of information are stored. For each state x, the value V(x) is kept, representing the total discounted return expected when starting at state x, and performing optimal actions. For each state x and action u, the advantage, A(x,u), is also stored, representing the degree to which the expected total discounted reinforcement is increased by performing action u (followed by optimal actions thereafter), relative to the action currently considered best. After convergence to optimality, the value function V*(x) represents the true value of each state. The advantage function A*(x,u) will be zero if u is the optimal action, since u confers no advantage relative to itself, and A*(x,u) will be negative for any suboptimal u, considering that a suboptimal action has a negative advantage relatively to the best action. The optimal advantage function A* can be defined in terms of the optimal value function V*

$$A(x,u) = \frac{1}{\Delta t}\left[R_{\Delta t}(x,u) - V^*(x) + \gamma^{\Delta t} \sum_{x'} P(u,x,x')V^*(x')\right] \tag{13}$$

The definition of an advantage includes a $\frac{1}{\Delta t}$ term to ensure that, for small time step duration Δt, the advantages will not all go to zero. Advantages are related to Q values by:

$$A(x,u) = \frac{1}{\Delta t}\left[Q^*(x,u) - \max_{u'} Q^*(x,u'))\right] \tag{14}$$

The previously described algorithm, can be summarized into the following pseudocode.
Learn: Perform action u_t in state x_t:

$$A(x_t,u_t) \leftarrow A_{\max}(x_t) + \frac{R_{\Delta t}(x_t,u_t) + \gamma^{\Delta t} V(x_{t+\Delta t}) - V(x_t)}{\Delta t} \tag{15}$$

$$V(x_t) \leftarrow V(x_t) + \frac{[A_{\max_{\text{new}}}(x_t) - A_{\max_{\text{old}}}(x_t)]}{\alpha} \tag{16}$$

Normalize:

Pick an arbitrary state x and pick an action u randomly with uniform probability:

$$A(x,u) \leftarrow A(x,u) - A_{\max}(x) \tag{17}$$

Normalization is done to ensure that after convergence $A_{\max}(x) = 0$ for every state [13].

2.2.4 Exploration vs Exploitation

As it was mentioned earlier, in the early stages of both algorithms, an agent has to explore the existing MDP, so it can calculate Q(s,a) for every possible pair of state and action. After exploring the environment, the agent has to converge in the best, by now, policy and try to exploit it in order to decide if this is the optimal or not. A question that arises, is how long and with what way, will the agent explore the environment. This trade-off from exploration to exploitation introduces the critical 'Exploration-Exploitation dilemma' [27], a recurring theme in reinforcement learning and AI in general. To address this issue, many policies have been proposed , including the ϵ-greedy (Algorithm 2) and the Ornstein–Uhlenbeck process [28].

2.2.5 ϵ-greedy Exploration

A straightforward approach to tackle the aforementioned exploration-exploitation dilemma involves oscillating between exploration and exploitation phase during the training process. More precisely, at every time-step, the agent will, with probability ϵ, explore the environment by using a random action; and with probability $1 - \epsilon$ it will exploit its so far acquired knowledge by selecting the "greedy" action. To converge in an optimal policy, agent should mainly explore in the early stages and shifts towards exploitation in the later stages. Consequently, the value of ϵ is typically set to decrease over time.

Algorithm 2 Epsilon-Greedy Exploration [29]

```
Initialize ϵ (exploration rate)
for each episode do
    for each time-step in the episode do
        Generate a random number r from a uniform distribution in [0, 1]
        if r < ϵ then
            Take a random action (explore)
        else
            Take the action with the highest estimated Q-value (exploit)
        end if
        Update Q-values based on the observed reward and next state
    end for
    Update ϵ (e.g., decrease over time)
end for
```

2.2.6 Boltzmann Exploration/Softmax Exploration

On a different approach, Boltzmann Exploration technique suggests to select an action based on a probability distribution that favors the "greedy" action but does not guarantee it. More precisely, in discrete action space Boltzmann algorithm suggest to select action a_i based on the probability:

$$P(a_i) = \frac{e^{\frac{Q(a_i)}{\tau}}}{\sum_j e^{\frac{Q(a_j)}{\tau}}} \tag{18}$$

where

- $Q(a_i)$ is the Q-value associated with action a_i
- τ is the temperature parameter that controls the level of exploration.

When the temperature (τ) is high, the probabilities are more evenly distributed among the actions, promoting exploration. As the temperature decreases, the Softmax probabilities become more concentrated around the action with the highest Q-value, encouraging exploitation.

2.3 Deep Reinforcement Learning

While it's manageable to create and use a Q-table for simple environments, it's quite difficult to do so for some real-life environments. The number of actions and states in the latter environments can be thousands, making it extremely inefficient to manage Q-values in a table.

This is where the use of neural networks helps to predict the Q-values of actions at a given state without the use a table. In fact, instead of initializing and updating a Q-table in the Q-learning process, we'll initialize and train a neural network model.

2.3.1 Neural Networks

Let's discuss in more detail, the aforementioned problem of the hand-written digit recognition in order to provide a comprehensive elucidation of the application of neural networks to our specific problem. The human visual system is one of the wonders of the world. Most people effortlessly can recognize any hand-written sequence of digits. That ease is deceptive. In each hemisphere of our brain, humans have a primary visual cortex, also known as V_1 , containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V_1, but an entire series of visual cortices – V_2 , V_3 , V_4 , and V_5 – doing progressively more complex image processing. The difficulty of visual pattern recognition becomes apparent if you attempt to write a rule based computer program to recognize digits [31]. Artificial Neural Networks [32] can solve this by imitating the structure of the human brain. A Neural Network, is constructed of three basic layers (fig 3).

- Input layer — the initial data provided for the neural network.
- Hidden layers — the intermediate layer between input and output layer, where all the computation is done
- Output layer —it produces the result for the given inputs.

Each layer, consists of numerous perceptrons (or so-called neurons) that can copy the functionality of a biological neuron (fig 4). Each perceptron, simulates the function:

$$y = \sum_{n \varepsilon N} f(w_i x_i) = f(W^T X) \tag{19}$$

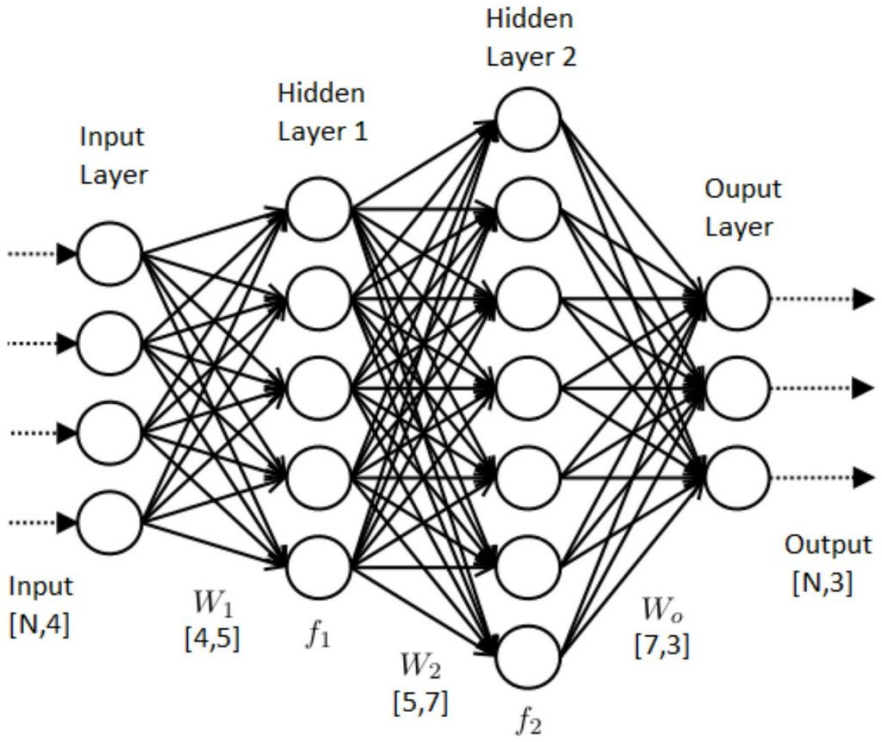


Figure 3: Structure of a (sequential) Neural Network as denoted in [30]

where w_i are the trainable weights of every input. In order for every perceptron, to calculate y, which takes values between 0 and 1, an activation function 'f' is needed. The activation functions (AFs) play a crucial role in neural networks by learning the abstract features through non-linear transformations. Several AFs, with different attributes, have been explored in recent years [33]. The most commonly used ones are:

- Linear(x) = cx (fig 5)
- Logistic Sigmoid(x) = $\frac{1}{1+e^{-x}}$ (fig 5)
- Tanh(x) = $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (fig 5)
- Rectified Linear Unit Based Activation Functions (Relu) (x)= max(0,x) (fig 6)
- Leaky Rectified Linear Unit (LReLU)(c) = max (c*x, x), where $c \approx 0.01$ (fig 6)

Knowing the structure of a typical Neural Network, let us now discuss the way that NNs work. As a supervised learning model, labeled training data- that are stored in a buffer and sampled uniformly in batches- are fed (usually normalized between 0 and 1 or -1 and 1) into the input layer of the NN. The result of the network ($\hat{y}$) is produced in the output layer. Using a distance metric (such as RMSE[34], CrossEntropy Loss [35] etc), the distance between prediction and label is calculated. Then, using an optimization algorithm (a variant of gradient descent such as SGD [36], ADAM[37], Adagrad [38] etc), the weights of the network are modified in order to reach the optimal loss function. It is worth mentioning, that the use of the random mini-batch is necessary in order to break the correlation between related samples.

Based on these principles, there have been implemented many types of NNs such as:

- Fully connected-Feed forward Networks

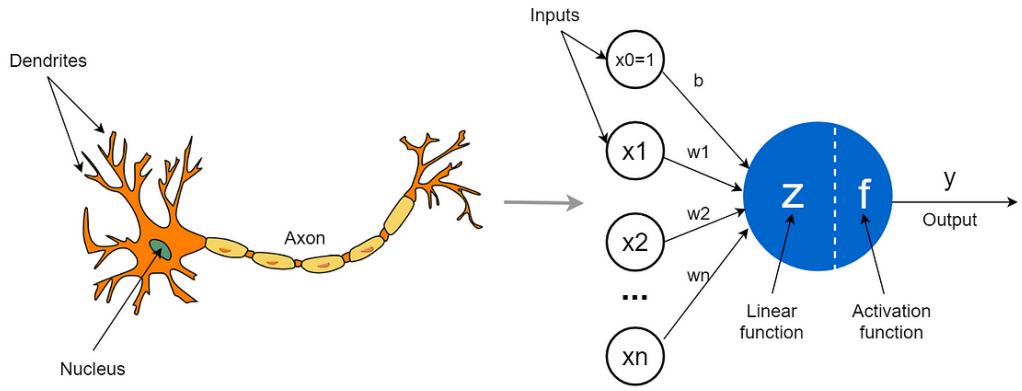


Figure 4: Comparison between biological neuron and perceptron. The picture was found in towards data science

- Convolutional NNs
- Recurrent NNs (and the improvement, called Long Short-Term Memory)
- Modular NNs

2.3.2 Deep Q-Network

As aforementioned, the Q-table is used as a -limited- function approximator. In order to scale the performance of the Q-learning algorithm in a more demanding environment, a new function approximator -like Neural Networks- is needed. Inspired by the improvement of computer vision - that was caused by the raise of the Convolutional NN - DeepMind's researchers produced an implementation of CNN, which is trained based on the Bellman equation (Equation 6) [2]. More precisely, using the Tesauro's TD-Gammon architecture [39] as a starting point, they implement a Q-Learning algorithm, by replacing the network with a CNN called Q-network. Also, a technique known as experience replay [4] was used by the DeepMind team, in order to store the agent's experience (state, action,reward, next state, done) at each time-step. The agent is trained at each time-step using a mini-batch sampled from the experience replay, and then acts using an epsilon greedy algorithm that selects between the $a = argmax_{state}Q(state)$ and a random action as it is shown in Algorithm 3.

The primary distinction between the Tabular and non-Tabular approaches- aside from the obvious differences in network architecture- lies in the way that the agent learns. In the first approach, the agent is trained for the last observation, as opposed to the non-Tabular, where it is trained for a mini-batch of past, and non-related, observations, exploiting the data of the experience replay. This technique is used in order to avoid correlations between the samples, something that is required for the network to converge. Also, the replay buffer, is used for the network to avoid rapidly forgetting some possibly rare experiences that would be useful later on [14].

2.3.3 Double DQN

DQN, suffers from overestimation problems, which affects negatively its performance. More precisely, in a non-yet-converged network, the $Q(\phi_{j+1}, a'|\theta)$, might be higher or lower estimated

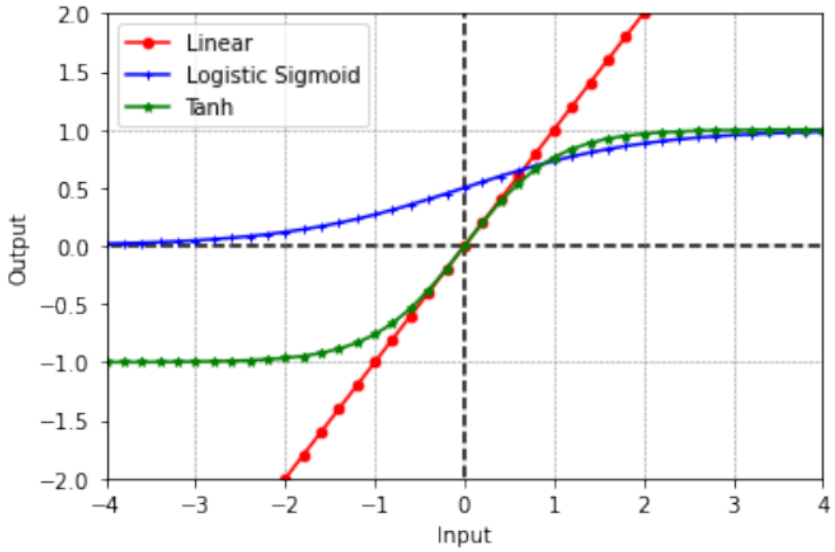


Figure 5: Linear Sigmoid and Tanh activation functions

than the real value . In case where all Q-values are over or under estimated, there will be no changes in the converged policy. If, however, the overestimations are not uniform , they might negatively affect the quality of the resulting policy [40].To handle this, a slightly different implementation of the DQN was introduced in [41]. In this variation of DQN, the $max_{a'}Q(\phi_{j+1}, a', \theta)$ is calculated by a "frozen" network called Target, which is a clone of the "model" network in a past time-step.

A shown in Algorithm 4, the model network is trained in order to accurately predict the:

$$y_j = r_j + \max_a Q(\varphi_{j+1}, argmax_{a^t} Q_{\theta^t}(\varphi_{j+1}, a^t)|\theta^t) \tag{20}$$

In conclusion, DDQN [40] uses two networks, the target and the model networks, built with the same initial architecture, but updated with different values.

2.3.4 Dueling Network Architecture

As it was previously mentioned, the Advantage Updating algorithm [13] was shown to converge faster than Q-learning in simple continuous time domains [42]. After the implementation of the DQN (combination of a NN and Q-learning), an algorithm inspired by Advantage updating was crafted [43]. Dueling Network Architecture (DNA), combines the separation of the Q function into Value and Advantage and calculates these, using a dual NN.

More precisely, DNA consists of a single neural network, with two heads; the first calculates the Value Function (V(s)), and the second calculates the A(s,a) (Figure 7). Now ,

$$Q(s, a; \theta) = V(s; \theta) + [A(s, a; \theta) - max_{a' \varepsilon A} A(s, a'; \theta)] \tag{21}$$

and the defined action in time-step t is: $a_t = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ argmax_{a \varepsilon A} Q^*(s, a), & \text{with probability } 1 - \epsilon \end{cases}$

Algorithm 3 Deep Q-learning with Experience Replay [2]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1 to M **do**
    Initialize sequence $s_1 = [x_1]$ and preprocessed sequence $\varphi_1 = \varphi(s_1)$
    for $t = 1$ to T **do**
        With probability ϵ, select a random action a_t, otherwise, select $a_t = \text{argmax}_a Q(\varphi(s_t), a, \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in D
        Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j & \text{for terminal } \varphi_{j+1} \\ r_j + \gamma \max_{a'} Q(\varphi_{j+1}, a', \theta) & \text{for non-terminal } \varphi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\varphi_j, a_j, \theta))^2$
    end for
end for

Algorithm 4 Double Deep Q-Network (Double DQN) [40]

Initialize replay memory D to capacity N
Initialize primary network (model) Q_θ, target network Q_{θ^t}, $t << 1$
for episode = 1 to M **do**
    for $t = 1$ to T **do**
        With probability ϵ, select a random action a_t, otherwise, select $a_t = \text{argmax}_a Q(\varphi(s_t), a, \theta)$
        Execute action a_t in emulator and observe reward r_t and next state s_{t+1}
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in D
        for each update step **do**
            Sample a random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from D
            Compute the target value y_j using the target network:
            $y_j = \begin{cases} r_j & \text{for terminal } \varphi_{j+1} \\ r_j + \max_a Q(\varphi_{j+1}, argmax_{a^t} Q_{\theta^t}(\varphi_{j+1}, a^t) | \theta^t) & \text{for non-terminal } \varphi_{j+1} \end{cases}$
            Perform a gradient descent step on $(y_j - Q(\varphi_j, a_j, \theta))^2$
            Every C steps, update the target network weights: $\theta^t \leftarrow \theta$
        end for
    end for
end for

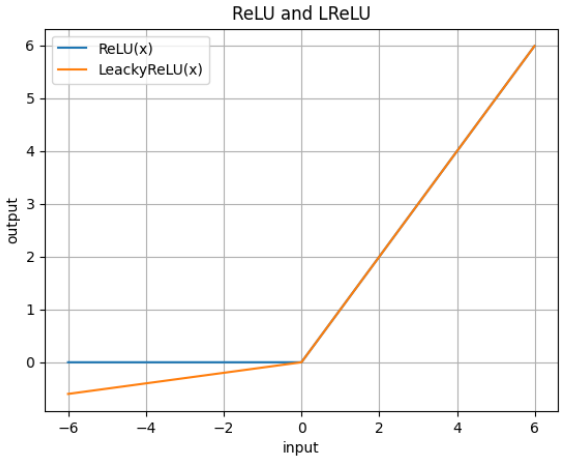


Figure 6: ReLU and LReLU

2.3.5 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) [11] introduces the direct representation of a policy in such a way that can solve continuous action spaces problems. DDPG, combines the Deterministic Policy Gradient[44] with the DDQN architecture, in order to create an actor-critic algorithm.As shown in Algorithm 5, the value and the best action of each state, are calculated by different networks. The aforementioned networks, are optimized thought the training process on the same experience. More precisely, the actor network is optimized based on the TD-Error while the actor network is optimized based on the sampled policy gradient.

Algorithm 5 DDPG Algorithm [11]

Randomly initialize critic network $Q(s, a|\theta_Q)$ and actor $\mu(s|\theta_\mu)$ with weights θ_Q and θ_μ
Initialize target network Q' and μ' with weights $\theta_{Q'} \leftarrow \theta_Q$, $\theta_{\mu'} \leftarrow \theta_\mu$
Initialize replay buffer R
for episode $= 1$ To M **do**
    Initialize a random process N for action exploration
    Receive initial observation state s_1
    for $t = 1$ To T **do**
        Select action $a_t = \mu(s_t|\theta_\mu) + N_t$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta_{\mu'})|\theta_{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta_Q))^2$
        Update the actor policy using the sampled policy gradient: $\nabla_{\theta_\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta_Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta_\mu}\mu(s|\theta_\mu)|_{s=s_i}$
        Update the target networks: $\theta_{Q'} \leftarrow \tau\theta_Q + (1-\tau)\theta_{Q'}$, $\theta_{\mu'} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu'}$
    end for
end for

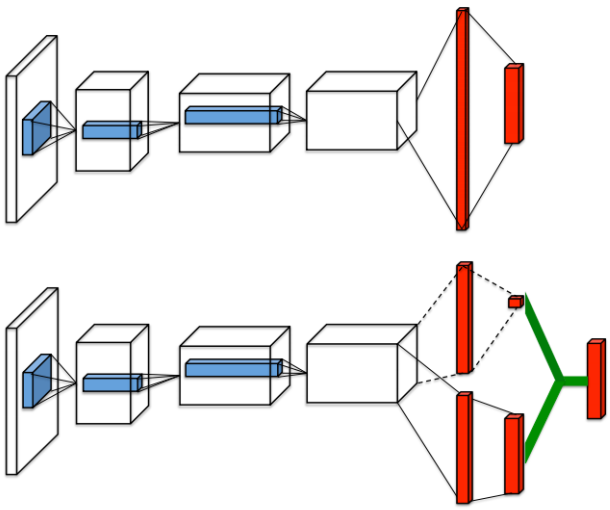


Figure 7: A popular single stream Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action. Both networks output Q-values for each action.

2.3.6 Normalized Advantage Function

Knowing that DNA yields better results than DQN [43], an algorithm inspired by DNA was developed in order to solve continuous action space problems. The Normalized Advantage Function (NAF) [12], uses the separation of the Q-function to easily solve the $argmax_{a\varepsilon A}Q(s,a)$ problem. More precisely, both Value and Advantage term are produced by the NN separately 8, and their combination produces the Q(s,a) in contrast to the DDPG.

$$Q(x,u|\theta^Q) = A(x,u|\theta^A) + V(x|\theta^V) \tag{22}$$

$$A(x,u|\theta^A) = -\frac{1}{2}(u-\mu(x|\theta^\mu))^T P(x|\theta^P)(u-\mu(x|\theta^\mu)) \tag{23}$$

$P(x|\theta^P)$ is a state-dependent, positive-definite square matrix, which is parametrised by $P(x|\theta^P) = L(x|\theta^P)L(x|\theta^P)^T$, where $L(x|\theta^P)$ is a lower-triangular matrix whose entries come from a linear output layer of a neural network, with the diagonal terms exponentiated. While this representation is more restrictive than a general neural network function, since the Q-function is quadratic in u, the action that maximizes the Q-function is always given by $\mu(x|\theta^\mu)$. Based on that:

$$Q_{\mathbf{max}} = V(x|\theta^V) \text{ given that } A(x,u|\theta^A) = 0 \Leftrightarrow u = \mu(x|\theta^\mu) \tag{24}$$

Algorithm 6 Continuous Q-Learning with NAF [12]

Randomly initialize normalized Q network $Q(x, u; \theta^Q)$
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer R
for episode = 1 to M **do**
    Initialize a random process N for action exploration
    Receive initial observation state $x_1 \sim\leftarrow p(x_1)$
    for $t = 1$ to T **do**
        Select action $u_t = \mu(x_t; \theta^\mu) + N_t$
        Execute u_t and observe r_t and x_{t+1}
        Store transition (x_t, u_t, r_t, x_{t+1}) in R
        for iteration = 1 to I **do**
            Sample a random mini-batch of m transitions from R
            Set $y_i = r_i + \gamma \cdot V'(x_{i+1}; \theta^{Q'})$
            Update θ^Q by minimizing the loss: $L = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(x_i, u_i; \theta^Q))^2$
            Update the target network: $\theta^{Q'} \leftarrow \tau \cdot \theta^Q + (1 - \tau) \cdot \theta^{Q'}$
        end for
    end for
end for

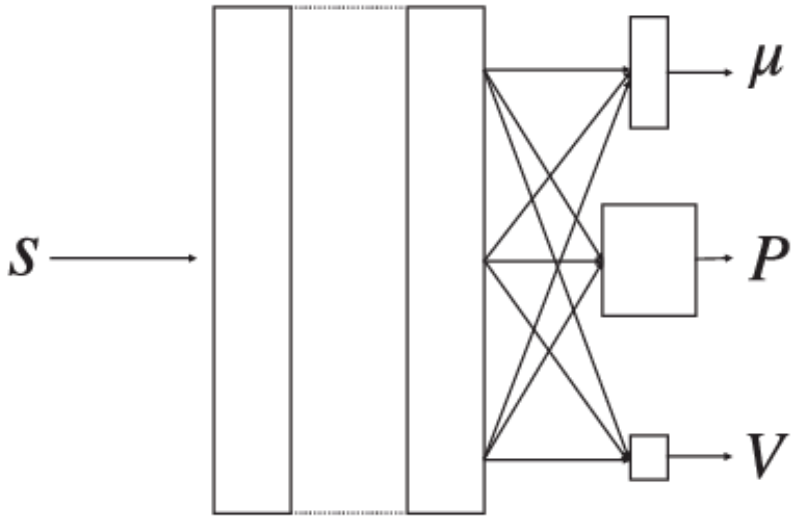


Figure 8: NAF architecture as proposed in [45]

In comparison to the previously mentioned algorithms, NAF does not use an epsilon greedy technique for exploration. Instead, in the original NAF paper [12], a modification of the Boltzmann Exploration Algorithm (Section 2.2.6) was proposed for use, that allows it to select an continuous action "near" to the greedy one. More precisely, the Equation 18 was re-written as:

$$\pi(u|x) = \frac{exp(Q(x, u|\theta^Q))}{\int exp(Q(x, u|\theta^Q))du} = \mathcal{N}(\mu(x|\theta^\mu), cP(x|\theta^P)^{-1}) \tag{25}$$

where

- $\mu(x|\theta^\mu)$ is calculated by the network

- $P(x|\theta^P)^{-1})$ is calculated by the network
- $\mathcal{N}$ is the Gaussian distribution

Algorithm 6 represents an abstract form of the NAF algorithm, as proposed in [12]. The detailed complete version of the algorithm is presented in Algorithm 7, where the Boltzmann exploration for continuous action space (Equation 25) is used for the action selection process. Additionally, in Algorithm 7 is also explained the process of calculating the Q-values.

Algorithm 7 Extended implementation of the NAF algorithm as proposed in [12]

Randomly initialize normalized Q network $Q(x, u; \theta^Q)$
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer R
for episode = 1 to M **do**
    Receive initial observation state $x_1 \sim\leftarrow p(x_1)$
    for $t = 1$ to T **do**
        Select action $u_t = \mathcal{N}(\mu(x_t; \theta^\mu), c \cdot P(x_t; \theta^P))$
        Execute u_t and observe r_t and x_{t+1}
        Store transition (x_t, u_t, r_t, x_{t+1}) in R
        for iteration = 1 to I **do**
            Sample a random mini-batch of m transitions from R
            Set $y_i = r_i + \gamma \cdot V'(x_{i+1}; \theta^{Q'})$
            Calculate the $Q(x_i, u_i; \theta^Q)$ according to Equations 22 and 23
            Update θ^Q by minimizing the loss: $L = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(x_i, u_i; \theta^Q))^2$
            Update the target network: $\theta^{Q'} \leftarrow \tau \cdot \theta^Q + (1 - \tau) \cdot \theta^{Q'}$
        end for
    end for
end for

To boost the algorithm's performance, researchers [12] suggested a different implementation of epsilon greedy. In this implementation, instead of a random action, the algorithm selects an action that locally optimizes the environment. More precisely, in [12] the iLQG algorithm was utilized to generate good trajectories under the model, and then mix these trajectories together with on-policy experience by appending them to the replay buffer. The intuition behind this result is that iLQG exploration is too different from the learned policy, and NAF must consider alternatives in order to ascertain the optimality of a given action. This variation, does not result in significant improvement for NAF due to the fact that NAF requires noisy on-policy actions to succeed [12]. To then add some exploration, researchers used a technique named Imagination roll-outs. In this technique, the dynamic of the model is required to simulate some steps from a given point while acting randomly. In general case, the dynamic of the model should be unknown. The algorithm, periodically tries to locally learn (using iteratively refitted time-varying linear models) the dynamics of the model. All the above, are summarized in Algorithm 8

It is worth mentioning, that both the local optimizer and the model emulator can be replaced with different algorithms than the suggested ones.

Algorithm 8 Imagination Rollouts with Fitted Dynamics and Optional iLQG Exploration [12]

Randomly initialize normalized Q network $Q(x, u; \theta^Q)$
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer $R \leftarrow \emptyset$ and fictional buffer $R_f \leftarrow \emptyset$
Initialize additional buffers $B \leftarrow \emptyset$ and $B_{old} \leftarrow \emptyset$ with size nT
Initialize fitted dynamics model $M \leftarrow \emptyset$
for episode = 1 to M **do**
    Initialize a random process N for action exploration
    Receive initial observation state x_1
    Select $\mu'(x, t)$ from { $\mu(x|\theta)$; iLQG $\pi_t(u_t|x_t)$ }with probabilities p and $1 - p$
    for $t = 1$ to T **do**
        Select action $u_t = \mathcal{N}(\mu'(x_t, t), c \cdot P(x_t; \theta^P)$
        Execute u_t and observe r_t and x_{t+1}
        Store transition $(x_t, u_t, r_t, x_{t+1}, t)$ in R and B
        if mod (episode $* T + t, m) = 0$ and $M \neq \emptyset$ **then**
            Sample m $(x_i, u_i, r_i, x_{i+1}, i)$ from B_{old}
            Use M to simulate l steps from each sample
            Store all fictional transitions in R_f
        end if
        Sample a random mini-batch of m transitions $I * l$ times from R_f and I times from R, and update θ^Q, $\theta^{Q'}$ as in Algorithm 6 per mini-batch.
    end for
    if B_f is full **then**
        $M \leftarrow$FitLocalLinearDynamics(B_f)
        $\pi^{\mathbf{iLQG}}$: iLQG_OneStep(B_f, M)
        $B_{old} \leftarrow B_f$; $B_f \leftarrow \emptyset$
    end if
end for

2.4 Prioritized Experience Replay

As it was previously mentioned, in order for Deep Reinforcement Learning algorithms to be functional, a replay buffer is needed. In the aforementioned implementation, researchers used a buffer called experience replay memory [4]. In this implementation's experience is stored sequentially and sampled uniformly. Even though it led to "Deep Reinforcement revolution" , it's considered a naive way of sampling. Let's suppose that for a network to learn, it should be trained in samples for which it has predicted inaccurate predictions. In that -initial- suggestion, samples would be prioritized based on their TD-error. After exploring the environment for some time-steps and calculating every time the td-error(using the Equation11) for every sample, the network will then be trained based on the samples with the biggest td-error. However, this greedy approach focuses on a small subset of the experience, leading to sub-optimal performance due to overfitting reasons. To overcome this issue, a stochastic sampling method was proposed, which interpolates between pure greedy and uniform random sampling. The probabilities of that stochastic process must be proportional to the td-error and it must be ensured that the network is being trained at least once with every sample. Considering the above, the probability for every sample -in case of the td-error prioritization- should be proportional to:

$$p_i = \begin{cases} \max(p) & \text{if } i \text{ was just sampled} \\ \\ |\text{td\_error}| + \epsilon & \text{otherwise} \end{cases} \tag{26}$$

where ϵ is a small constant that guaranties a positive value.
In case of the rank-based prioritization the probability should be calculated as:

$$p_i = \frac{1}{\text{rank}(i)} \tag{27}$$

where rank(i) is the rank of transition i when the replay memory is sorted according to: |td_error|
Suppose a Deep-RL algorithm that is trained using the aforementioned algorithm. In the early training stages, the buffer will ignore - choose with a smaller probability- samples that, by chance, happen to be predicted correctly from the network. In order to solve this abnormality, an α factor insertion was proposed, oscillating between 0 and 1, in order to transit from a completely uniform buffer to one based on the td-error. Applying the above in a stochastic model, the probabilities of each sample can be calculated as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{28}$$

Even though the prioritized algorithm was implemented, the prioritization model introduces bias because it changes this distribution in an uncontrolled fashion [14], creating the need of a factor that corrects it. To calculate this factor, researchers used importance sampling (IS), and they calculated the:

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \tag{29}$$

Based on that, Prioritized Experience Replay can be combined with a Deep RL algorithm as it's shown in 9

Algorithm 9 Double DQN with Proportional Prioritization [14]

Input: mini-batch k, step-size α, replay period K, size N, exponents α and β, horizon T.
Initialize replay memory $H = \emptyset$, $\epsilon = 0$, $p_1 = 1$.
Observe S_0 and choose $A_0 \sim \pi(S_0)$.
for $t = 1$ to T **do**
    Observe S_t, R_t, A_t.
    Store transition $(S_{t-1}, A_{t-1}, R_t, A_t, S_t)$ in H with maximal priority $p_t = \max_{i<t} p_i$.
    if $t \equiv 0 \mod K$ **then**
        for $j = 1$ to k **do**
            Sample transition j from $P(j) = p_j^\alpha / \sum_i p_i^\alpha$.
            Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$.
            Compute TD-error $\delta_j = R_j + \gamma_j \max_a Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$.
            Update transition priority $p_j \leftarrow |\delta_j|$.
            Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla Q_\theta(S_{j-1}, A_{j-1})$.
        end for
        Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$.
        Occasionally copy weights into target network $\theta_{\text{target}} \leftarrow \theta$.
    end if
    Choose action $A_t \sim \pi_\theta(S_t)$.
end for

2.5 Parameter Space Noise for Exploration

Exploration remains a key challenge for Deep-RL and even though epsilon greedy solves the exploration versus exploitation trade-off in simple environments, a deeper method is needed for more complex ones. An initial thought, could involve adding Gaussian noise to the algorithm's policy. In that case, actions would be sampled according to stochastic policy generation.

$$\alpha_t = \pi(s_t) + \mathcal{N}(0, \sigma^2 \mathcal{I}) \tag{30}$$

Therefore, even in a fixed state s_t, the policy of the network would differ, due to the fact that the noise would be independent from the state. In order to overcome this problem, noise should be added directly to the layers of the NN [15] using the:

$$\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 \mathcal{I})^{1} \tag{31}$$

Now let's suppose a scenario where noise is added to the network's policy (like Equation 30). In that case, the effect of the noise can be easily tuned by the σ parameter. Hence, in Equation 31, the effect of σ cannot be predicted and the value of it, should be determined by trial and error. More precisely, a σ value, that doesn't dominate the network's policy is needed, but it also helps explore the environment more. In a continuous action space, the difference between two actions can be defined as the distance between them. So, as was concluded in [15] σ should be modified in every episode based on:

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{if } d(\pi, \tilde{\pi}) \leq \delta, \\ \frac{1}{\alpha}\sigma_k & \text{otherwise,} \end{cases} \tag{32}$$

where $\alpha \epsilon \mathbb{R}_{>0}$ is a scaling factor and $\delta \epsilon \mathbb{R}_{>0}$ a threshold value. d(.;.) is substituted by a different metric each time depending on the algorithm. As noted in the Appendix of the [15], Kullback-Leibler (KL) divergence [46] is used as a distance measure for DQN. For DDPG, Mean Square Error is used :

$$d(\pi, \tilde{\pi}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} E_s \left[(\pi(s)_i - \tilde{\pi}(s)_i)^2 \right]} \tag{33}$$

It is worth mentioning, that 10 Algorithm's architecture is not designed for NAF. To combine PSNE with NAF, some modifications are needed and will be discussed in Section 3.2.3

2.6 Autonomous driving

Machine Learning and AI in general, have been actively working for more than two decades now with the aim of making autonomous driving vehicles a reality. In the following paragraphs, some innovative works in both lane-based and lane-free environment will be presented.

2.6.1 Autonomous Driving in Lane-Based Traffic

There exist numerous works on Autonomous Driving (AD) in a lane-based environment, while in recent years most of them have solved the problem using Deep-RL. Here we briefly reference some recent such works. To begin with, [48] shed light on the problematic design of the reward function and suggested a list of "sanity checks " that every reward function for lane-based environment should follow. An agent was designed in [49], which was proven capable of outperforming human drivers in a Playstation Game called "Grand Turismo". For this implementation, the

[1] where θ are the parameters of the network and $\tilde{\theta}$ are the parameters of the perturbed network

Algorithm 10 Choosing a continuous action using PSN for Exploration, having a DQN network [47]

$\sigma \leftarrow$ threshold
$\alpha \leftarrow 1.02$
for t = 1...Horizon **do**
    Perturbed $\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 \mathcal{I})$ and obtain $\tilde{\pi} = \pi_{\tilde{\theta}}$
    while not done **do**
        Sample an action α_t using the perturbed policy $\tilde{\pi}$
        Execute action α_t and observe a new state s_{t+1}
        if t mod $T_{\text{adapt}} = 0$ **then**
            Perturbed $\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 \mathcal{I})$
            Calculate the distance d = d(·,·)
            if d $< \sigma$ **then**
                $\sigma_{e+1} = \alpha \sigma_e$
            else
                $\sigma_{e+1} = \frac{1}{\alpha} \sigma_e$
            end if
        end if
    end while
end for

agent was trained on maximizing the reward proposed in [50], using a soft actor critic method (Quantile-Regression Soft Actor-critic or QR-SAC) . In [51], some agents for multiple urban traffic scenarios were developed also using Deep-RL and the proposed learning framework called Flow. Finally, [52] addressed a simple vehicle trajectory problem by creating an end-to-end Deep-RL pipeline, treating the environment as a POMDP.

2.6.2 Autonomous Driving in Lane-Free Traffic

For most of human history, there was no need for traffic lanes, due to the low speed of movement and that small use of vehicles. Lanes were introduced in the 1950s, for safety reasons. With that addition, driving is transformed into an easier task, while the driver has only to monitor the distance and the speed of the front car. Hence, there is no need of monitoring the vehicles on the left, right and rear of its car. Nevertheless, the addition of lanes results in the problems produced by the lane changing maneuver. Lane changing is a complex task as the driver needs to search for an available gap on the target lane and predict its evolution based on the observed speeds of multiple vehicles (and of its own), while at the same time paying attention at the distance to the front vehicle. As [53] denotes, lane changes are responsible for the 10% of all accidents. The TrafficFluid Concept [7], suggests a combination of the lane-free traffic environment with the "nudging" effect. Nudging should be imagined as a pushing force in the direction of the line connecting the centers of the nudging vehicle and the nudged vehicle in front.

Numerous works in lane-free traffic environment suggest multiple control methods for agent policy optimization. More precisely, [7] proposes an ad-hoc method based on forces while [54] uses Control Theory to devise a strategy. In some others works, the proposed strategy was created while using Multi-Agent Decision Making ([55], [56]). In more detail, [7] introduces a lane-free vehicle movement strategy which relies on heuristic rules incorporating the concept of 'forces,' where vehicles 'push' one another to facilitate overtaking or to react appropriately in various situations. Now, [54] designs a two-dimensional cruise controller for lane-free traffic, with more emphasis on Control Theory. Furthermore, [56] presents a strategy for lane-free vehicles that utilizes optimal control methods, particularly focusing on model predictive control.

In this approach, each vehicle optimizes its behavior over a specified future horizon, while considering the trajectories of nearby vehicles. In addition to that, [55] tackles the problem using the max-plus algorithm, and constructing a dynamic graph structure of the vehicles, considering communication among vehicles. Hence, some other works suggest a policy by using Deep Reinforcement Learning.

As it has been previously mentioned, for Deep Reinforcement learning to be used, the environment should be described by a MDP. Karalakou et Al., created a MDP for the lane-free, ring road environment [1] and tested it by implementing an agent using the DDPG algorithm [1]. Finally, using the aforementioned agents as mentors and the formulated MDP, an algorithm for implicit imitation Deep-RL was presented in the work of [9], exploiting state transition data from a mentor-expert agent for training acceleration. In the present work, the implemented MDP will be used for the Deep-RL agent to then be improved by applying some state of the art Deep-RL algorithms.

3 Our Approach: Environment and Agent

In this chapter, both environment architecture and agent implementations are described.

3.1 Our Lane-Free Traffic Learning Environment

3.1.1 Environment

A ring road traffic scenario is considered as a simulation environment. This road is occupied by two different vehicle types. The first class/type, includes automated -non trainable- vehicles. Every vehicle of that class has a different desired speed and its goal is to maintain it without colliding with other vehicles. Both the environment and the vehicle behaviors are described in [7]. The second class, consists of a Deep - RL trainable agent, that has the same goal as the other vehicles. In comparison with them, the agent has no initial knowledge, and its goal is to optimize the cumulative reward in the MDP which is proposed in [1]. In this environment, the agent can observe both position and speed -in 2-dimensional axes- for both itself and its nearby vehicles. All the vehicles in that road pursue a different desired speed that is chosen randomly at the beginning of every simulation. Finally, the agent can affect its own position and speed by controlling its acceleration (longitudinal and lateral). The environment is simulated using an extension of SUMO for Lane-Free traffic [57], while our agent's code infrastructure was mainly developed using Python's PyTorch [58]. As for the ring road implementation, a highway was used where the vehicles reaching the end of it, are constantly re-located to the beginning, finally creating an endless road.

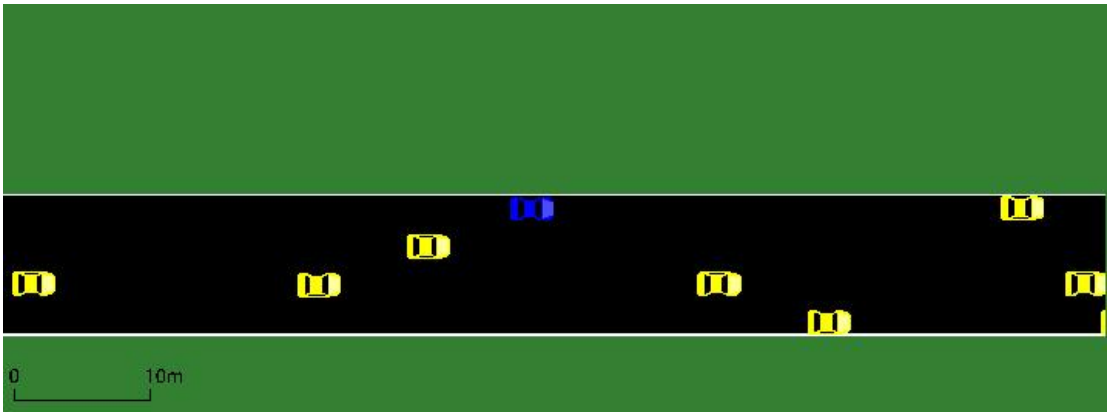


Figure 9: Interface of SUMO

3.1.2 State Space

In a Markov Decision Process (MDP), the state space is a fundamental element which represents the agent's knowledge per time-step. Having a Neural Network (NN) that can learn function representation of high complexity, it is easy to assume that the state space can contain multiple features instead of being restricted by tabular RL's state space limitations. In a lane-based road, the driver should observe how many vehicles are in the other approximate lanes, both at the front and at the rear of his vehicle. However, by using this state space, the algorithm does not quite fit in a lane-free environment where the vehicles now "freely" operate on a 2-dimensional

space. So, instead of the number of vehicles in the neighbor lanes, the exact position of the nearest vehicles is used. A real world driver would not observe all the vehicles in the road in order to decide an action, but a fixed number of neighbor vehicles. This situation does not differ from a lane-based to lane-free environment. Therefore, the environment consists of the -relative to the agent- coordinates (d_x and d_y for the non controlled vehicles and y for the ego) for 5 neighbor vehicles - 3 in front and 2 at the rear of the agent. In addition to the position of both ego and neighbor vehicles, the state space includes both the lateral and longitudinal speed of the vehicles (v_x, v_y), as are both necessary for the decision making process. For instance, the agent's policy should properly adapt depending on a vehicle located in front according to its speed as well. Finally, a crucial aspect of an agent's state space is its desired speed.

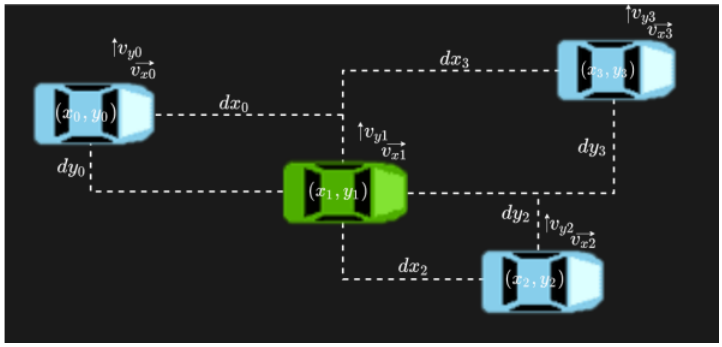
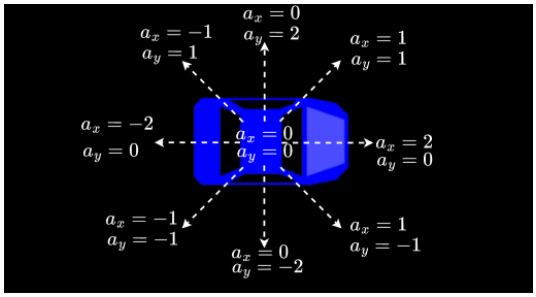


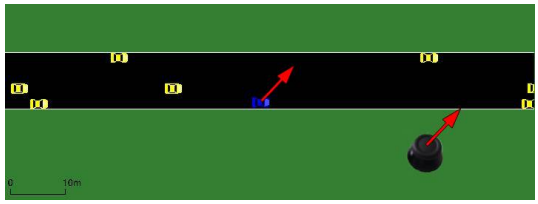
Figure 10: State space's variable as it was implemented in [10]

3.1.3 Action Space

The agent controls the vehicle through its longitudinal and lateral acceleration. As shown in Figure 11, the agent can modify its longitudinal axis acceleration, like a driver can use the gas pedal or the brakes, and the lateral axis, like a driver using the steering wheel. In previous works [1], [9], authors restrict the agent's action to a discrete action space. Specifically, instead of an action that belongs in $\mathcal{R}^2$, the agent must choose from the 9 actions shown in Figure 11a. In our work by contrast, at every time-step the agent decides an action that is mapped to the two dimensional continuous space. It can be conceptualized, as the agent is being controlled by a joystick as shown in Figure 11b. Hence, the testing algorithms should produce at each time-step an action $\vec{a}$, where $\vec{a} \in \mathcal{R}^2$, which represents the exact acceleration values that result in a 2-dimensional motion of the vehicle.



(a) The discrete action space same as controlling the agent using the keyboard



(b) The continuous action space - same as controlling the agent using an analog joystick

Figure 11: Discrete vs. Continuous action space

3.1.4 Reward Function

The primary objective of a Deep Reinforcement Learning (RL) algorithm is to optimize the agent's reward within a MDP. This underlines the role of the reward function within the environment. In lane-free driving, reward functions need to describe the two basic goals: a) avoiding collisions and b) maintaining the desired speed. At first, the agent should receive a punishment every time that it is involved in a collision. However, this imposes the issue of delayed rewards, as our agent only receives a negative reward when colliding with another vehicle. To tackle this problem, a more informative reward component than the boolean "collisions" field is needed, to "quantify" the danger of collision among two vehicles. Hence, in the "collision" field the "repulsive force" function was also added, which calculates an estimate for the imminence of a collision happening between the neighbor vehicles. This function utilizes ellipsoid fields, taking into account the relative distances and speeds. Lastly, it is important for the agent to maintain a speed close to the desired target one. To learn how to follow this behavior, the reward function needs to include a term that punishes the agent whenever it is driving at a speed different than the desired one. Keeping into account the aforementioned, the reward function for this MDP, was created in [10]:

$$r = \frac{e_r}{e_r + w_x c_x + w_y c_y} + c_{\text{collision}} * \text{collision} \tag{34}$$

where e_r is a small positive number (set to 0.1), c_x is a penalty that is minimized when the agent has the desired speed, c_y is a penalty that is minimized when the agent has the minimum repulsive force and finally $c_{\text{collision}} < 0$ is added to the reward -as penalty/negative reinforcement- if the agent is involved in a collision. Even though the parameters w_x ,w_y and $c_{\text{collision}}$ were fine-tuned in [10], an extra tuning process was needed (explained in Section 4.4) in order to find the hyperparameter values that provide the best results when employing a different RL algorithm.

3.2 Our Lane-Free Traffic Agent

As demonstrated in [12], NAF outperforms DDPG in the domain of Atari games. This thesis aims to compare the aforementioned algorithms in the lane-free traffic domain, by implementing a fine-tuned NAF agent. Hence, to improve the performance of the agent, additional algorithms were built on top of the NAF agent. Specifically, to enhance the agent's learning efficiency, the uniform replay buffer was replaced with PER. Additionally, for improved exploration within the specific environment, both PSNE and a local optimization technique inspired by [12] were implemented. Finally, their performance was compared, to determine the best exploration strategy.

3.2.1 Implementing the Normalized Advantage Function Agent

In this thesis, NAF, PER and PSNE were combined to create an agent following a strategy as close to the optimal as in the aforementioned environment. To begin with the NAF implementation (Algorithm 11), two NNs were used; one as a model and the second as a target, like the algorithm suggests. Each NN, has an input sized the same as the state space (24 inputs same as the "features" of the environment). Given a state x, the output of each NN consisted of the following components:

- V, denoting the value of each state (number)
- μ, denoting the best action for that state (array 2 by 1)

- L, matrix used for P's construction (matrix 2 by 2)

In every time-step, the agent acts based on $N(\mu(s), c \cdot P(s))$. More precisely, it acts based on current policy while adding some trainable exploration. After it's action, the network's weights need to be tuned. During the weight optimization process, the agent chooses samples from the buffer a batch of experience tuples ([state, action, reward, next_state, done]) and calculates Q(state, action) ,based on the Equations 23 22 24, and computes the TD-Error using Equation 11. Then, the network is optimized based on TD-Error. It is worth mentioning, that A was calculated using a quadratic form as proposed in [12].

Algorithm 11 The NAF Agent implementation for the Lane-Free Environment

Randomly initialize normalized Q network $Q(x, u; \theta^Q)$
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer R
Initialize hyper-parameters I, λ, γ and τ
Initialize batch-size N
for episode = 1 to M **do**
    Receive initial observation state $x_1 \sim\leftarrow p(x_1)$
    for $t = 1$ to T **do**
        Select action $u_t = \mathcal{N}(\mu(x_t; \theta^\mu), c \cdot P(x_t; \theta^P))$
        Execute u_t and observe r_t and x_{t+1}
        Store transition (x_t, u_t, r_t, x_{t+1}) in R
        for iteration = 1 to I **do**
            Sample a random mini-batch of m transitions from R
            Set $y_i = r_i + \gamma \cdot V'(x_{i+1}; \theta^{Q'})$
            Update θ^Q by minimizing the loss: $L = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(x_i, u_i; \theta^Q))^2$
            Update the target network: $\theta^{Q'} \leftarrow \tau \cdot \theta^Q + (1 - \tau) \cdot \theta^{Q'}$
        end for
    end for
end for

For better results, researchers [12] suggest to include an iLQG [59] controller, for local optimization. Instead of an iLQG, a local minimizer was used, using SciPy library [60], in order to minimize the cost function, at every step. In comparison with iLQG, a minimizer does not need a manual computation of the dynamics derivative. Even though the derivative is not needed, the dynamics of the system are still necessary, to predict the future reward. In this environment, the prediction of the new position of the ego vehicle given its action, can be calculated. On the other hand, the new position of the neighbor vehicles are not predictable, due to the fact that their actions are unknown. To overcome this issue, an initial dynamic was assumed and is updated iteratively based on observations. Finally, the minimization of the cost function (Equation 34) was proven a difficult task for the minimizer. Instead, a simpler version of the reward/cost function (Equation 35) was used:

$$r = c_1 * speed\_reward + collisions \tag{35}$$

Minimizing the Equation 35, enables the agent to explore the environment more efficiently by ignoring some "useless" actions. As mentioned in paragraph 2.3.6, the agent should be also trained in some "useless" actions too. To achieve this, imagination rollouts are performed periodically. In these rollouts, agent produces new experiences, by using an old experience as an initial point and simulating the behaviour of the environment while it is acting optimally (using the minimizer) for $l-1$ steps and randomly for the last step. In conclusion, the agent can simulate bad experiences that is useful for the training process. In addition to the above, PER

and PSNE were also used to accelerate the learning parameter process and to achieve better exploration results.

3.2.2 Combining Prioritized Experience Replay Memory with Normalized Advantage Function

After the straight-forward implementation of NAF, the uniform memory in which the experience tuples are stored, was replaced by a Prioritized Experience Replay buffer. As shown in Algorithm 12, the addition of PER does not modify the structure of the NAF algorithm (Algorithm 11). Nevertheless, the non-uniform sampling proposed in PER, requires some additional computations, creating a slight computational overhead.

Note: In the following pseudocode (Algorithm 12 and Algorithm 13), specific formatting has been used to highlight modifications from a previous implementation. Words written in red color, are representing changes or additions in comparison to the previous version .

Algorithm 12 The NAF+PER Agent implementation for the Lane-Free Environment

Randomly initialize normalized Q network $Q(x, u; \theta^Q)$
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize Prioritized Experience Replay Memory buffer R
Initialize an array Π to store the probabilities which are used in PER
Initialize hyper-parameters I, λ, γ, τ, α and β
Initialize batch-size N
for episode $= 1$ to M **do**
    Receive initial observation state $x_1 \sim\leftarrow p(x_1)$
    for $t = 1$ to T **do**
        Select action $u_t = \mathcal{N}(\mu(x_t; \theta^\mu), c \cdot P(x_t; \theta^P))$
        Execute u_t and observe r_t and x_{t+1}
        Store transition (x_t, u_t, r_t, x_{t+1}) in R with maximal priority $\Pi_t = \max_{i<t} \Pi_i$
        for iteration $= 1$ to I **do**
            Sample a mini-batch of m transitions from R (according to probabilities $\frac{\Pi_i^\alpha}{\sum_i \Pi_i^\alpha}$)
            Set $y_i = r_i + \gamma \cdot V'(x_{i+1}; \theta^{Q'})$
            (Compute TD-Error$_i = y_i - Q(x_i, u_i; \theta^Q)$
            Compute importance-sampling weight $w_j = (N \cdot \Pi(j))^{-\beta} / \max_i w_i$.
            Update transition priority $\Pi_j \leftarrow |\text{TD-Error}_j|$.
            Accumulate learning rate $\lambda \leftarrow \lambda - w \cdot \lambda$.
            Update θ^Q by minimizing the loss: $L = \frac{1}{N} \sum_{i=1}^{N} (\text{TD-Error}_i)^2$ while using the new learning rate
            Update the target network: $\theta^{Q'} \leftarrow \tau \cdot \theta^Q + (1 - \tau) \cdot \theta^{Q'}$
        end for
    end for
end for

3.2.3 Combining Normalized Advantage Function with Parameter Space Noise for Exploration

NAF's initial exploration strategy (Algorithm 11) differs from the one used in most Deep-RL algorithms. More precisely, instead of an ϵ-greedy technique, a pseudo-random action is selected in every time-step. This pseudo-random action is calculated by the network(Equation 25). The intuition behind that variation of Boltzmann's exploration strategy, is that as the model is trained, the actions will vary from "completely random" to "guided random" ones. In the second algorithm proposed (Algorithm 8), the aforementioned "exploration" strategy is replaced by an algorithm similar to ϵ-greedy. More precisely, in every time-step, two actions are calculated:

1) $\mathcal{N}(\mu(x|\theta^{\mu})$, calculated by the NN and selected with probability $1-\epsilon$

2) action that minimizes the local environment, selected with probability ϵ

Following a similar pattern, in the PSNE implementation, the second action was substituted with the one computed by the perturbed network. At the beginning of every episode, the variance of the perturbed network is calculated based on the "distance" between the two actions and the threshold. To calculate the error between perturbed and clear actions, the Mean Square Error metric was used, as proposed in [15] for DDPG. Even though this modification (selecting perturbed action with some probability ϵ) is not proposed in [15], it merely has no difference with the original one [47], due to the way that the threshold is reduced. More precisely, threshold is calculated based on:

$$\zeta = -log(1-\epsilon+\frac{\epsilon}{\|A\|}) \tag{36}$$

While ϵ is decreasing, perturbed action does not differ from the original one.

3.3 Proposed Algorithm

In conclusion, the NAF algorithm was combined with PER and PSNE, in order for the agent to yield better results. As will be mentioned in Section 4.4.8, the combination of NAF and PSNE was proven a challenging task due to both Boltzmann exploration strategy that is used from NAF, and the instability issues that the NN presents in this specific environment. To address these issues, this thesis proposes the following modifications:

- The use of the Boltzmann "perturbed" action (as described in Section 4.4.8).
- The established of the head start parameter to ensure that the neural network converges before applying the extra exploration strategy.
- The ϵ-greedy algorithm, that selects actions between those proposed by Boltzmann exploration and those proposed by PSNE exploration.

As shown in Algorithm 13, by using an $\epsilon - greedy$ technique that alternates between Boltzmann's action and perturbed action, the agent consistently selects actions combined with noise. The presence of that noise increases the risk of non convergence, specifically in the early stages of training, due to specific details of the problem's nature - the agent's speed is not reset to a default value after the end of every episode. For instance, if noisy actions lead the agent to drive at extreme speeds (either too fast or too slow), the agent may fail to explore some states within the "ideal" range of speeds. The NAF algorithm, particularly when combined with PER, effectively eliminates these "extreme" actions in the early stages. However, with the addition of PSNE, the occurrence of "extreme" behaviors becomes more likely. To limit the influence

of PSNE in the early stages, the head start parameter was introduced. This parameter allows the utilization of PSNE exploration when the network has semi-converged, providing an additional, more detailed exploration, when necessary. Algorithm 13 shows the pseudocode for the implementation of the NAF+PER+PSNE agent. A single training-step of that agent is also visualized in Figure 12 in which, different "stages" within a single training step are shown with different color.

To the best of our knowledge, PSNE have never been combined with NAF, so there is no previous reference that strictly/theoretically proves the optimality of that algorithm.

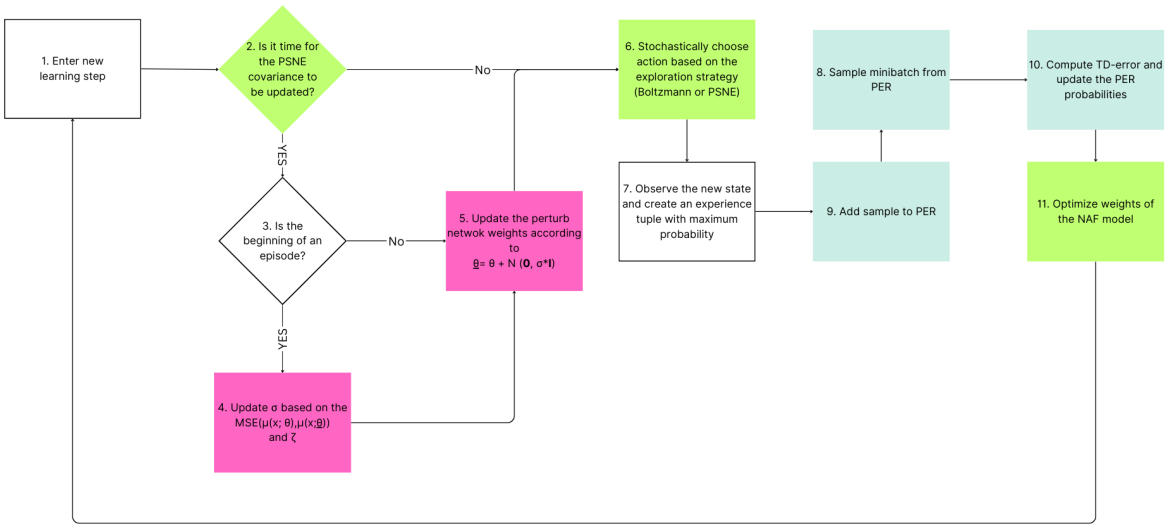


Figure 12: Proposed agent's training step flowchart. Different stages within the step are shown with different color. In more detail stages 1-3-7 are related to the environment, 2-6-11 are part of the NAF model, 4-5 are part of the PSNE and 8-9-10 are part of the PER.

Algorithm 13 Final NAF+PER+PSNE Algorithm

Randomly initialize normalized Q network $Q(x, u; \theta^Q)$
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize perturbed network $\tilde{Q}$
Initialize Prioritized Experience Replay Memory buffer R
Initialize an array Π to store the probabilities which are used in PER
Initialize parameters I, λ, γ, τ,α_{PSNE}, α and β and head-start
Initialize batch-size N
Initialize σ_{initial}
for episode = 1 to M **do**
    Receive initial observation state $x_1 \leftarrow p(x_1)$
    Calculate the distance d = MSE($\mu(x_t; \theta^\mu)$,$\mu(x_t; \tilde{\theta^\mu})$)
    if d $< \zeta(\epsilon)$ **then**
        $\sigma_{\text{episode+1}} = \alpha_{\text{PSNE}} \sigma_{\text{episode}}$
    else
        $\sigma_{\text{episode+1}} = \frac{1}{\alpha_{\text{PSNE}}} \sigma_{\text{episode}}$
    end if
    Update the perturbed network according to $\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 \mathcal{I})$
    for $t = 1$ to T **do**
        if episode > head-start **then**
            Select action $u_t = \mathcal{N}(\mu(x_t; \theta^\mu), c \cdot P(x_t; \theta^\mu))$ with probability $1 - \epsilon$ else select $u_t = \mathcal{N}(\mu(x_t; \tilde{\theta^\mu}), c \cdot P(x_t; \tilde{\theta^\mu}))$
        else
            Select action $u_t = \mathcal{N}(\mu(x_t; \theta^\mu), c \cdot P(x_t; \theta^\mu))$
        end if
        Execute u_t and observe r_t and x_{t+1}
        Store transition (x_t, u_t, r_t, x_{t+1}) in R with maximal priority $\Pi_t = \max_{i<t} \Pi_i$
        for iteration = 1 to I **do**
            Sample a mini-batch of m transitions from R(according to probabilities $\frac{\Pi_i^\alpha}{\sum_i \Pi_i^\alpha}$)
            Set $y_i = r_i + \gamma \cdot V'(x_{i+1}; \theta^{Q'})$
            Compute TD-Error$_i = y_i - Q(x_i, u_i; \theta^Q)$
            Compute importance-sampling weight $w_j = (N \cdot \Pi(j))^{-\beta} / \max_i w_i$.
            Update transition priority $\Pi_j \leftarrow |\text{TD-Error}_j|$.
            Accumulate learning rate $\lambda \leftarrow \lambda - w \cdot \lambda$.
            Update θ^Q by minimizing the loss: $L = \frac{1}{N} \sum_{i=1}^{N} (\text{TD-Error}_i)^2$ while using the new learning rate
            Update the target network: $\theta^{Q'} \leftarrow \tau \cdot \theta^Q + (1 - \tau) \cdot \theta^{Q'}$
        end for
        if $t == \frac{T}{2}$ **then**
            Update the perturbed network according to $\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 \mathcal{I})$
        end if
        Update ϵ
        $\zeta(\epsilon) = -log(1 - \epsilon + \frac{\epsilon}{2})$
    end for
end for

4 Experimental Evaluation

In this chapter, we will present the results of a series of experiments that were conducted to fine-tune hyper-parameters and evaluate NAF's performance in comparison to the DDPG, implemented in [1]. Finally, we test if the aforementioned algorithms (PER, PSNE and the local minimizer) improve the NAF's performance.

4.1 Environment Setup

In order for this comparison to be valid, the environmental settings should be the same as the ones used in [1]. These settings are shown in the following table, Table 1

Parameter	Value
Highway length	500m
Highway width	10.2m
Vehicle's length	3.2m
Type of vehicles	2
Num. of vehicles	35
Episode's duration	200s
Time interval	0.25s
agent's desired speed	20 m/s
other vehicles desired speed	18 to 22 m/s

Table 1: Simulation parameters

After conducing many experiments [10], researches concluded to the value of the weights of the reward function (Equation 34), those weights were re-fine-tuned in order to optimize NAF's performance.

4.2 Agent Setup

The entire agent's side of the code, was written using Python due to its access of ML libraries. More precisely, the network was implemented using PyTorch [58]. The selection of that library was based on the fact that pyTorch can be accelerated using CUDA [61], which provides a significant speed improvement to this application. In addition to that, numPy [62] was used to handle matrix calculations efficient and sciPy [60] to implement the require minimizer. As it is shown in 2.3.6, there are many parameters that need to be tuned. Some of them were tuned as proposed in [12] and are shown in Table 2, and the remaining were tuned based on the experiments (Section 4.4).

Parameter	Value
Batch size	128
Discount factor	0.99
Replay Memory size	10^5
Learning Rate used by Adam [37]	10^{-3}
Number of episode for Training	575
Soft update parameter	$5 * 10^{-3}$

Table 2: Agent's parameters

4.3 Conducted Experiments

This work aims to improve the lane-free agent's performance. To evaluate the effectiveness of a new algorithmic combination, it will be compared with the existing model to determine if this modification proves advantageous. The total number of conducted experiments is presented in Table 3.

Candidate 1		Candidate 2
NAF's Fine Tuning	-	-
NAF	vs.	NAF+PER
NAF+PER	vs.	NAF+PER+different versions of PSNE
NAF+PER+PSNE	vs.	NAF+PER+local optimization

Table 3: Conducted Experiments presented in Section 4.4

4.4 Results

The parameter fine-tuning procedure, was proven to be a challenging task, which required numerous experiments for many different combinations of hyper-parameters. In the conducted experiments, the NAF agent was trained for 575 episodes with the use of noise as an exploration technique, and tested for 50 episodes. For every hyper-parameter combination, 5 repetitions of the experiment were performed, each time using a different pseudo-random seed. The mean of these repetitions was kept and later used for comparison purposes between all the differently tuned experiments. The criteria of the aforementioned comparison are the speed deviation from the desired speed, the number of collisions and the reward that the agent yields. It's worth mentioning that the reward criterion is not as valid as the others, especially when the reward function differs amongst the two experiments. After the fine-tuning procedure, the parameter combination with the best results was evaluated for five more pseudo-random seeds, giving ten seeds in total. Moreover, the extra tools (PER, PSNE and minimizer) were added to the NAF agent and then tuned, in order to optimize the agent's policy. It is worth mentioning, that in the upcoming experiments, the parameters previously examined will be configured to the values corresponding to best results, as determined in prior tests. The parameters that have not been yet fine-tuned, will be set according to the suggested values as proposed in [12]. Finally, both the graph results and the mean values of the last 50 episodes are presented for better examination. The graphs, illustrate the average reward gained by the agent over each episode, throughout the training and evaluation process. This average is calculated based on 10 repetitions, where each repetition is conducted using different pseudo-random seeds. Moreover, the mean values, are averaged over the 50 "evaluation" episodes and over the 10 different pseudo-random seeds,

finally producing one total value per criterion (reward, speed deviation, collisions).[2]

4.4.1 Goal

According to [10] (Table 4), the DDPG agent after 575 episodes of training was evaluated in 50 "evaluation" episodes that followed (episodes with no use of any exploration strategy). The values shown in Table 4, are calculated as mentioned in Section 4.4 (mean over episodes and over 10 seeds):

Reward function's weight	Avg. Collisions	Avg. Speed Deviation (m/s)
$w_x = 0.35$ and $w_y = 0.65$	0.526	-0.514
$w_x = 0.5$ and $w_y = 0.5$	0.917	-0.211
$w_x = 0.15$ and $w_y = 0.85$	0.137	-1.129
$w_x = 0.05$ and $w_y = 0.95$	0.046	-1.431

Table 4: Results in [10]

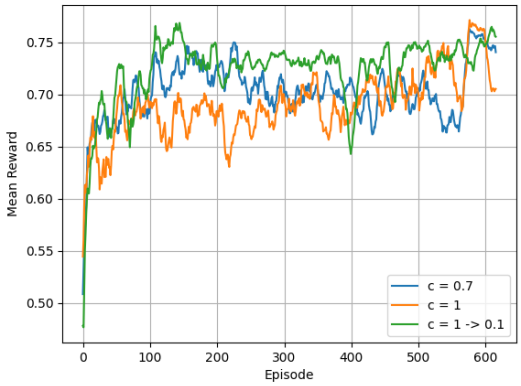
As it is explained in [10], the $w_x = 0.35$ and $w_y = 0.65$ is preferred due to the balanced results that it achieves.
The goal of this study, is to compare NAF's results against the ones yielded by DDPG (Table 4). Subsequently, PER and PSNE will be added to the NAF agent, and it will undergo testing in the same environment. The resulting outcomes will then be compared with the previous ones, with a focus on both collisions and speed deviation as criteria for each comparison. In order to achieve fair comparison across our algorithms and also against [10], we also used, for each method, 575 episodes of training followed by 50 "evaluation" episodes. All the NAF based experiments, were conducted on the same seeds.

4.4.2 Exploration testing

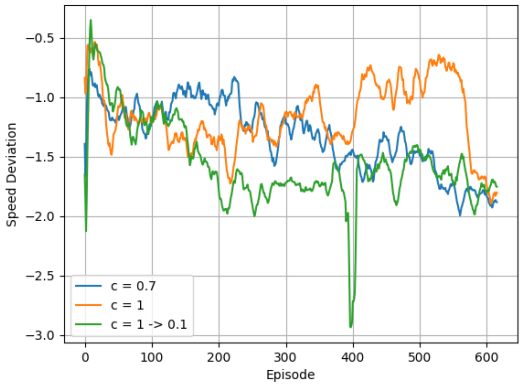
In the context of Deep Reinforcement Learning, an important procedure for the agent to learn, is the way that it explores the environment. This exploration strategy, is usually defined by the proposed Deep-RL algorithm. In NAF's case, Equation 25 determines via the c parameter how the agent will explore the environment. In the original NAF paper [12], various values of c were considered, including 1, 0.7, and 0.5. In the lane-free environment, 1 and 0.7 were used as a constant value of c, and also an epsilon greedy technique was introduced, where c decreases gradually from 1 to 0.1 within 200 episodes. The described experiments, were conducted using a Deep Neural Network with four layers (see Section 4.4.3) and the reward function's parameter that was denoted as best [1]. [3] Where $c = 1- > 0.1$ means that c was reduced from 1 to 0.1 within 200 episodes

[2] So in a more mathematical way the average reward of an experiment among repetitions, where r_{ij} is the average reward during an episode j and a repetition i, will be calculated as: $r = \frac{1}{\|repetitions\| \cdot 50} \sum_{i=0}^{\|repetitions\|} \sum_{j=1}^{j=50} r_{ij}$. Both speed deviation and number of collisions are calculated using the same formula, but in case of collisions, the c_{ij} is the total collisions during an episode j and a repetition i and not the average.

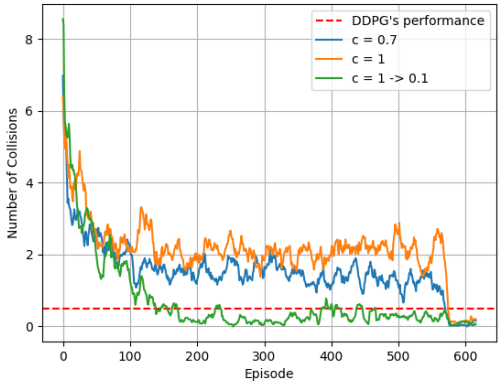
[3] From now on, all the "Mean Reward" graphs, are calculated as the mean value both over repetitions and over episodes. More precisely, lets suppose that r_{ert} is the reward that the agent gained at episode e, repetition r and timestep t of an experiment. The displayed vector, is calculated according to $r_e = \frac{1}{\|repetitions\| \cdot \|episode\|} \sum_{r=0}^{r=\|repetitions\|} \sum_{t=0}^{t=\|episode\|} r_{ert}$. So r_e is the value of the mean reward over an episode and over the different repetitions on a specific episode e. Based on that, speed deviation graphs were calculated using the same formula and number of collisions graphs was calculated using the $c_e = \frac{1}{\|repetitions\|} \sum_{r=0}^{r=\|repetitions\|} \sum_{t=0}^{t=\|episode\|} c_{ert}$. In other words, the collisions graphs were calculated using the total number of collision during an episode.



(a) Mean - both over episode and over seeds - reward yielded by the agent .



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.

Figure 13: Testing different exploration rates. Graphs are displayed according to footnote 3

Exploration rate	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
$c = 0.7$	0.751	-1.82	0.068
$c = 1$	0.743	-1.712	0.180
$c = 1 -> 0.1$	0.746	-1.784	0.100

Table 5: Exploration testing results. Table values are calculated according to footnote 2

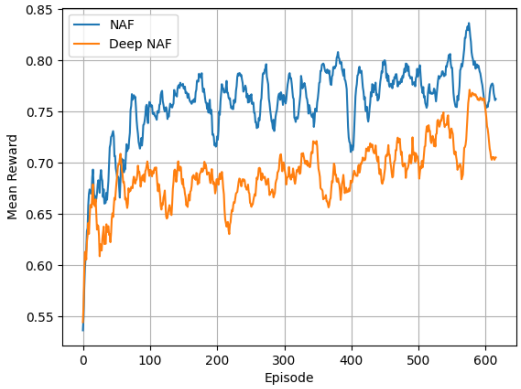
More precisely, as shown in Figure (13), in all exploration rates tested, agent achieves less than 0.2 collisions in average. Due to the low number of collisions, $c = 1$ will be selected as the best exploration rate, based on the higher speed (differs by -70% from [10])[4] that the agent

[4]From now on, all the percentages will be calculated in proportion to [10]'s results. In more detail, the speed deviation will be calculated according to $-\frac{\text{speed deviation}-(-0.512)}{\max(|\text{speed deviation}|,0.512)} * 100\%$ and the collisions percentage will be calculated according to $-\frac{\text{collisions}-0.52}{\max(\text{collisions},0.52)} * 100\%$. Based on that, the goal of the agent is to reduce the collision rate as close to -100% and reach speed difference from [10]as close to 0%

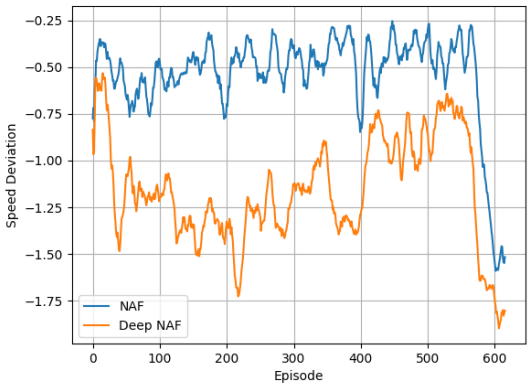
drives.

4.4.3 Structure testing

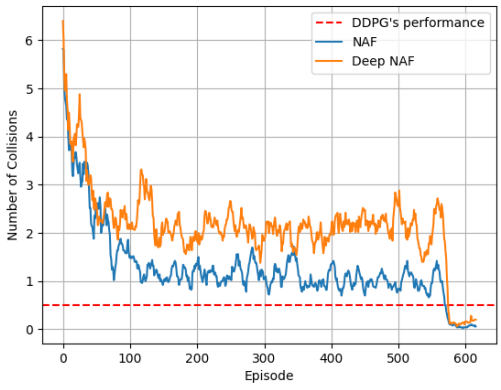
The previous testing scenario, shows that even with a lot of exploration, the agent tends to drive at lower speed for achieving as low collisions as possible. As mentioned in [1], the collisions and the speed deviation are equivalent goals, and the agent has to come up with a policy that tends to maximize both. To overcome this "partial" learning, a smaller NN was used to improve the generalization of the agent. More precisely, the Deep-NN used in the previous setting -with four hidden layers with size [280,128,128,128]- was replaced with a shallow one, with two hidden layers [256, 128]. In the following experiment (Figure 14 and Table 6) the implementation with the four hidden layer NN will be mentioned as "Deep-NAF" and the implementation with the two hidden layer NN will be mentioned as "NAF".



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.

Figure 14: Displaying performance of the algorithm using at each time a network with different number of hidden layers. Graphs are displayed according to footnote 3

Kind of Network	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
Deep NAF	0.743	-1.712	0.180
NAF	0.785	-1.249	0.080

Table 6: Structure testing results. Table values are calculated according to footnote 2

As shown in Figure 14 the agent with the smaller network tends to perform better in that environment. In more detail, as shown in Figure 6 the agent reduces the average collision (-85% from [10]) while increasing the speed deviation (-59% from [10] while Deep-NAF achieved -70%) .

4.4.4 Adding Noise to the Evaluation

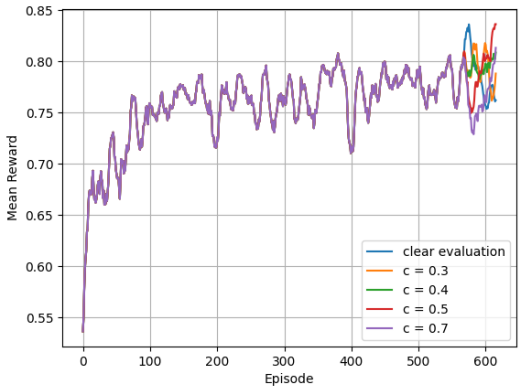
As shown in Figure 14, the agent's performance during the later stages of training outperforms its performance during evaluation. A possible explanation for that observation, is the fact that the added noise is proportional to μ and P (Equation 25). Since both μ and P are produced from the network, the added noise during training is not completely random. Building up from that observation, some experiments followed, using noise[5] in the evaluation phase too. These experiments, were conducted multiple times with different values of c.

Noise rate	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
$c = 0$	0.785	-1.249	0.080
$c = 0.3$	0.795	-0.930	0.152
$c = 0.4$	0.796	-0.867	0.296
$c = 0.5$	0.796	-0.667	0.452
$c = 0.7$	0.768	-0.759	0.676

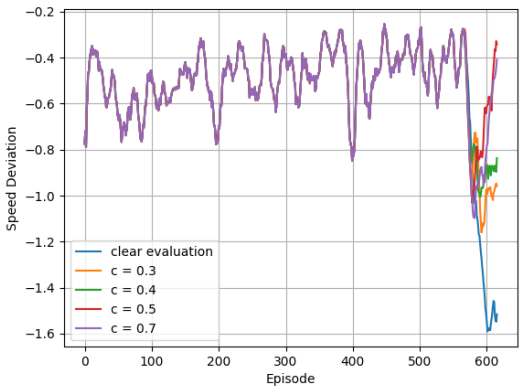
Table 7: Results of noisy evaluation. Table values are calculated according to footnote 2

The previous hypothesis was verified by the results in Figure 15. More precisely, the agent with evaluation noise amplitude by a factor $c = 0.5$ seems to outperform the original evaluation however leading to a driver that is willing to drive with less safety. Hence, this agent drives faster than the "clear" agent (driving on the -23% of [10]),but also increasing the collisions (now the agent is involved in -13% collisions than [10] while the agent with $c = 0$ was involved in -89%).

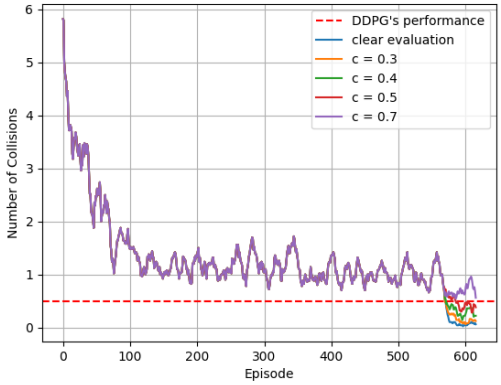
[5]It is worth mentioning, that the magnitude of that noise, should be kept lower than the noise during training. In other words, the use of evaluation noise contributes to maintaining a higher speed rather than exploring the environment.



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).

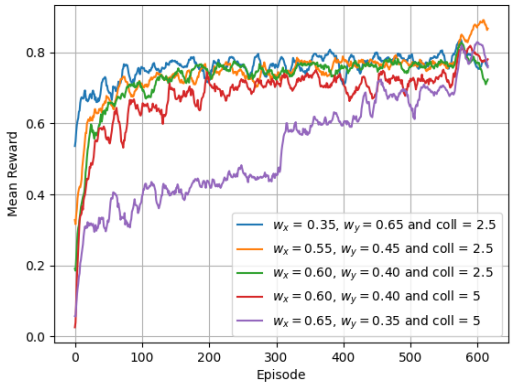


(c) Total collisions during an episode - mean over seeds.

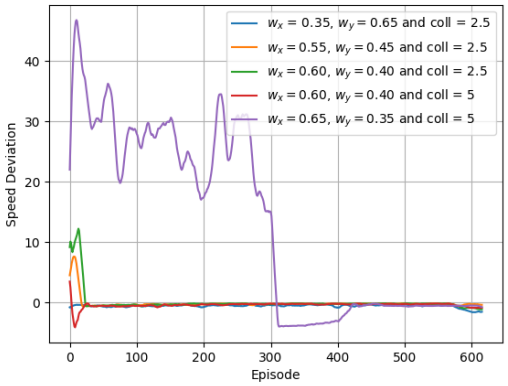
Figure 15: Covariance in Boltzmann exploration can be modified by setting different values for parameter c (Equation 25). In the displayed graph, during evaluation process c was set on non-zero values. Graphs are displayed according to footnote 3

4.4.5 Modifying the Reward Function Weights

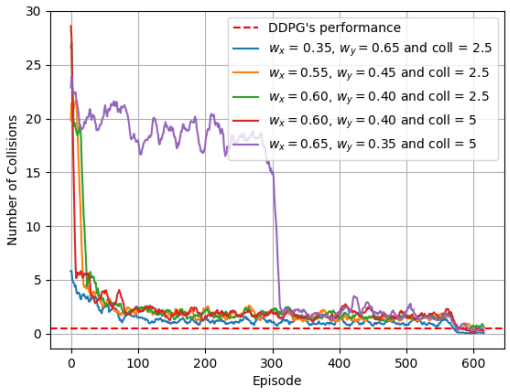
The previous results showed that the "noisy" evaluation balances in a more efficient way the trade-off between collision and speed deviation.Hence, the behavior resulted by the agent implemented in Experiment 15 is similar to the behavior of the agent in [1]. Despite the relatively low impact of "trainable" noise, the utilization of a stochastic process carries inherent risks .
The agent in the previous experiments, sacrifices the speed reward to maximize the collision and the repulsive force reward. To address this problem, the weights in Equation 34 should be modified to tempt the agent to drive faster.



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.

Figure 16: Plotting the performance of the agent, while using different variations of the reward function (Equation 34). Graphs are displayed according to footnote 3

Weights	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
$w_x = 0.35$, $w_y = 0.65$ and coll $= 2.5$	0.785	-1.249	0.08
$w_x = 0.55$, $w_y = 0.45$ and coll $= 2.5$	0.861	-0.344	0.504
$w_x = 0.60$, $w_y = 0.40$ and coll $= 2.5$	0.773	-0.838	0.854
$w_x = 0.60$, $w_y = 0.40$ and coll $= 5$	0.795	-0.697	0.376
$w_x = 0.65$, $w_y = 0.35$ and coll $= 5$	0.796	-0.548	0.504

Table 8: Results using different reward function. Table values are calculated according to footnote 2

As shown in Figure 16 setting in Equation 34, $w_x = 0.55, w_y = 0.45$ and collision coefficient $= 2.5$, the agent achieves both faster (driving, for the first time, by +33% faster than [10]) and safer driving -achieving 3% fewer collisions - than the DDPG agent achieves (Table 4). Hence,

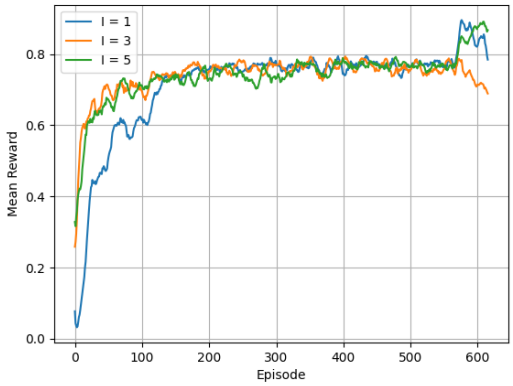
this weight adjustment triggers the agent to drive faster so that now the agent outperforms the DDPG agent implemented in [10].

4.4.6 Modifying the "update target" parameter

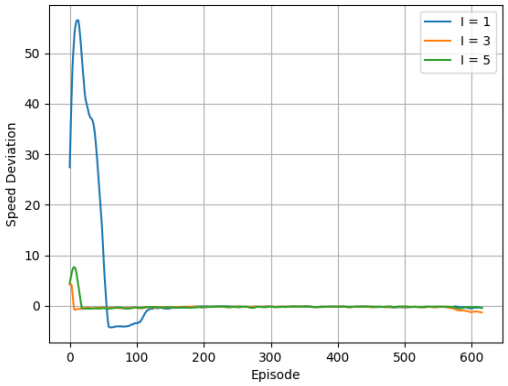
In the experiment of Section 4.4.5, the I parameter (Algorithm 6) was initially set to 5, as proposed in [12]. This choice, contributes to the performance of the agent -it is trained 5 times per step - but it slows the training speed. To reduce the training time without modifying the agent's behavior, numerous values for I were tested.

I	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
1	0.846	-0.344	0.416
3	0.732	-1.064	0.28
5	0.861	-0.344	0.546

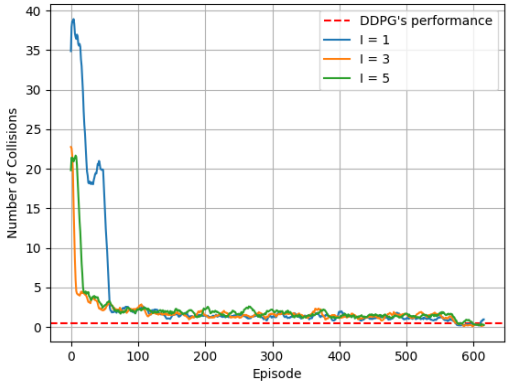
Table 9: Modifying the "update target" parameter. Table values are calculated according to footnote 2



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



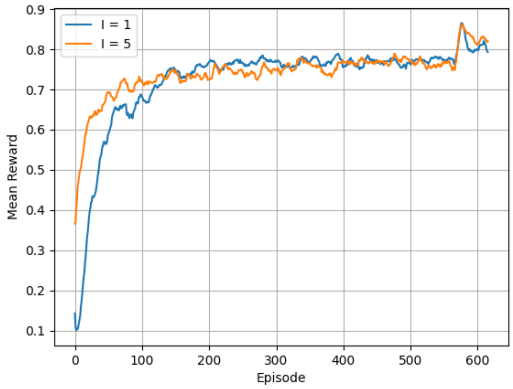
(c) Total collisions during an episode - mean over seeds.

Figure 17: Testing the performance of the agent while it is trained "I" times per timestep. Graphs are displayed according to footnote 3

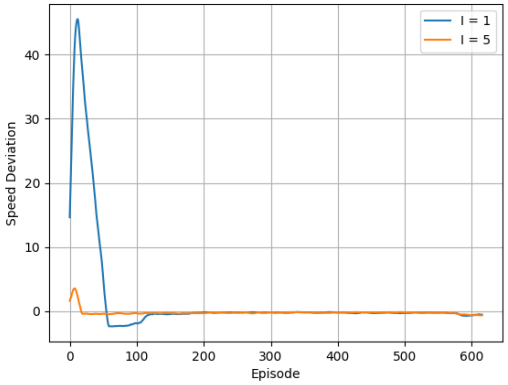
Finally for the NAF testing, experiments with $I = 1$ and $I = 5$ were conducted using 5 more seeds (10 in total), concluding to:

I	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
1	0.814	-0.524	0.316
5	0.832	-0.524	0.406

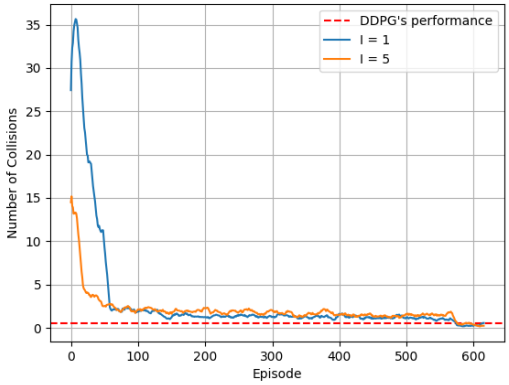
Table 10: Modifying the "update target" parameter. Table values are calculated according to footnote 2



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.
Graphs are displayed according to footnote 3

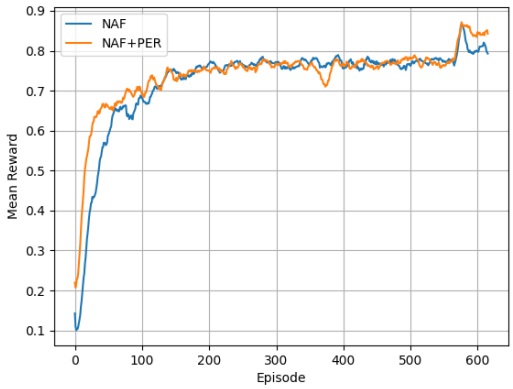
Figure 18: Testing the performance of the agent while it is trained "I" times per timestep. Graphs are displayed according to footnote 3

As shown in Figure 18, the agent trained with $I = 1$ yields slightly better results among the NAF agents. More precisely, by using $I = 1$ agent's speed deviation from desired speed remains the same (-0.524m/s or -2% from [10]) but average collisions are decreased (achieving 39% less collisions than [10]). Also, using $I = 1$, reduces training time in half, allowing from now on, to calculate results using ten seeds.

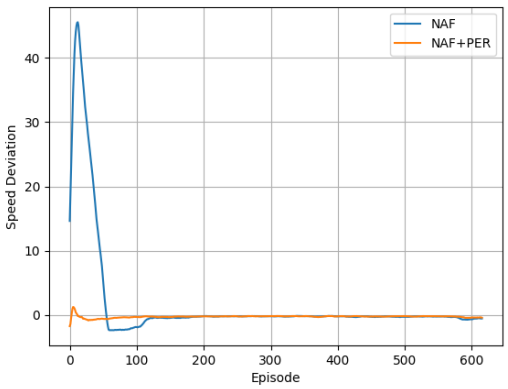
In conclusion, after the fine-tuning procedure, the NAF agent achieves 39% less collisions while maintaining slightly less speed (2% or 0.01m/s) than [10].

4.4.7 Including Prioritized Experience Replay

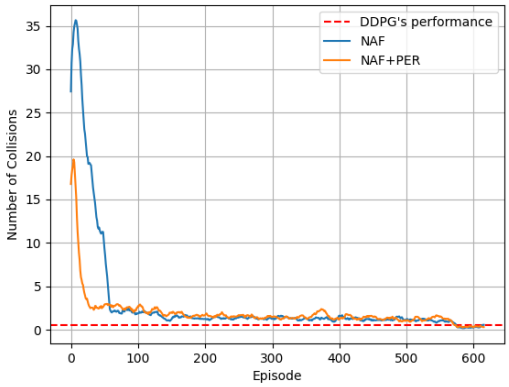
With the proper fine-tuning, the NAF agent outperformed DDPG as shown in the conducted experiments. A possible improvement of the agent, is going to be for the experience replay buffer to be replaced by the PER buffer. In that case, the agent would be trained in 'experience' that is more challenging. For this implementation, the proportional-based prioritization was used (Equation 26), and the hyper-parameters a and b_0 were set to 0.7 and 0.5 correspondingly, as proposed in [14]



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.

Figure 19: Testing agent's performance while using PER. Graphs are displayed according to footnote 3

As shown in Figure 19, by including PER, the agent achieves better reward during evaluation while at the same time, PER, reduces the training time by accelerating convergence speed. More

Sampling Technique	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
Using PER	0.848	-0.344	0.342
Using Uniform sampling	0.814	-0.524	0.316

Table 11: Testing agent's performance while using PER. Table values are calculated according to footnote 2

precisely, the PER agent tends to achieve lower speed deviation from the desired speed (drives +33% faster than than [10] while without using PER the agent drives at -2%), by better taking advantage of the free space in front of it. However, this behavior comes at the expense of a slightly higher collisions (-34% than [10] while without the use of PER, the agent was involved in -39%) for the agent. In summary, the usage of PER, both increases the reward of the agent and reduces the convergence time. Given these characteristics, PER will be utilized and its contribution will be re-evaluated if its combination with PSNE does not provide good results.

4.4.8 Including the Space Noise for Exploration Parameter

Combining NAF with PSNE proved to be challenging, due to the exploration strategy suggested by NAF. More precisely, the NAF's network can provide the agent with two actions:

a) "clear" action $\mu(x|\theta^{\mu})$

b) "noisy" action $\mathcal{N}(\mu(x|\theta^{\mu}), cP(x|\theta^{P})^{-1})$.

Hence, a perturbed network can also provide the agent with:

a') "clear" action $\mu(x|\tilde{\theta^{\mu}})$

b') "noisy" action $\mathcal{N}(\mu(x|\tilde{\theta^{\mu}}), cP(x|\tilde{\theta^{P}})^{-1})$ [6].

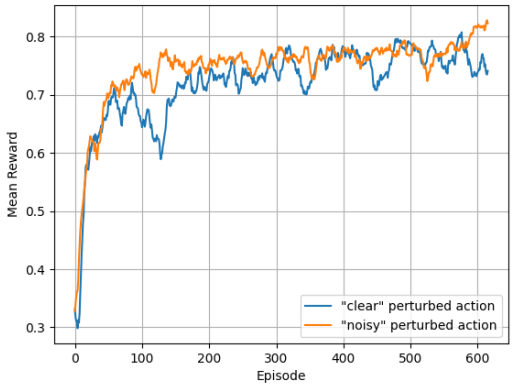
The first experiment , aimed to determine which perturbed action should be used. As shown in the figure (Figure 20), the use of the "noisy" perturbed action leads to better results. To complete the experiment, the other parameters were fine-tuned either empirically or as suggested in [15]. More precisely, the initial value of ϵ (Section 3.2.3 and Equation 36) was not set to 1 (as proposed in [15]) but to 0.85 due to instability issues that the NAF network presented. Also,To reduce the risk of lack of convergence, a new hyper-parameter, named "head-start" , was introduced. This hyper-parameter determines the quantity of rounds during which the NAF agent will explore the environment without the contribution of PSNE. Its initial value was set to 15. Hence, by the end of the 15th episode, perturbed actions were then selected with probability equal to $\epsilon = 0.85$ and by the end of the 215th episode this value had been reduced to $\epsilon = 0.15$. Also the covariance of the added noise was initially set to $\sigma = 10^{-3}$, as proposed in [15]. Hence, the covariance at every episode, was calculated based on:

$$\sigma_{k+1} = \begin{cases} 1.02\sigma_k & \text{if MSE}(\pi, \tilde{\pi}) \leq \zeta, \\ \frac{1}{1.02}\sigma_k & \text{otherwise,} \end{cases} \tag{37}$$

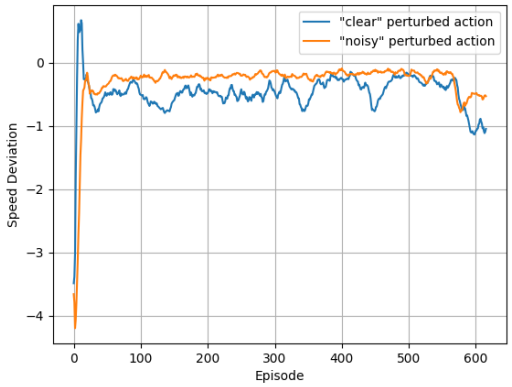
In more detail, Equation 37 adjusts the value of σ_{k+1} based on the Mean Square difference between clear and perturbed action. Hence, the calculation of threshold ζ, is based on Equation 36. The intuition behind Equation 37, is that σshould be selected in a way that perturbed actions slightly differ from the non-perturbed ones. This slight difference ensures that the agent

[6] $\tilde{\theta}$ denotes the perturbed network as described in Equation 31

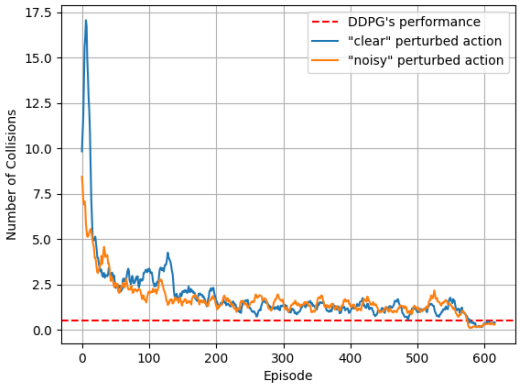
strikes a delicate balance between exploration and exploitation.
Finally, due to the long duration of each episode (800 time-steps), the perturbed network was updated twice in every episode.



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



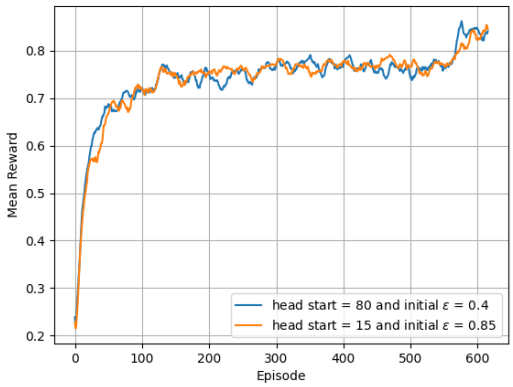
(c) Total collisions during an episode - mean over seeds.

Figure 20: Testing different PSNE implementations. More precisely, "clear" implementation uses the $\mu(x)$ action derived by the network and "noisy" uses the Boltzmann action. Graphs are displayed according to footnote 3

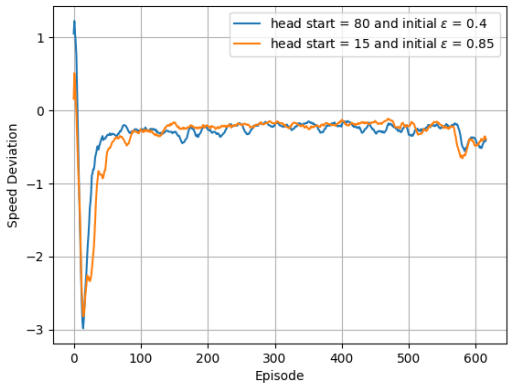
PSNE implementation	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
Second	0.76	-0.884	0.4
Third	0.828	-0.488	0.336

Table 12: Testing different PSNE implementations. Table values are calculated according to footnote 2

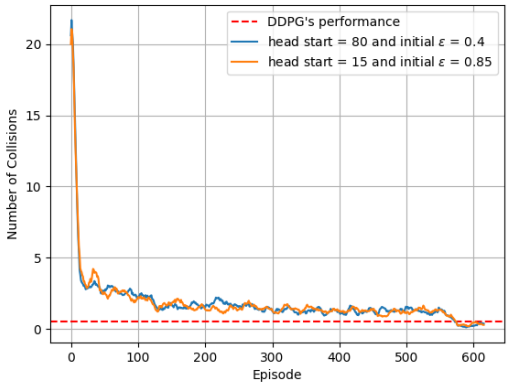
As explained in Section 3.2.3, the agent adopts a "greedy" exploration technique. By including PSNE in the early stages of training, the extra noise that is added turns the semi-random, into a completely random exploration. To take advantage of both exploration techniques, the head start parameter was set to 80, letting NAF explore the environment for the first 80 episodes and then PSNE was included, for a more detailed exploration. Lastly, the trade-off between PSNE and NAF was initially set to 0.4.



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.

Figure 21: Testing different PSNE implementations. Graphs are displayed according to footnote 3

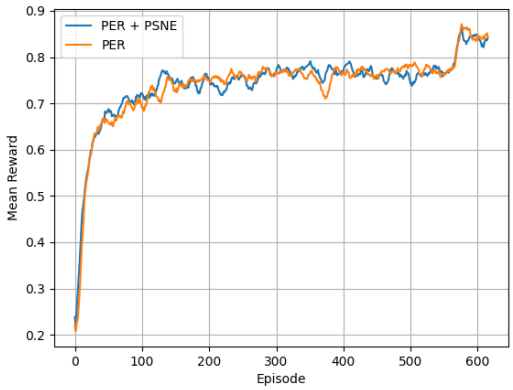
PSNE implementation	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
Fourth	0.842	-0.416	0.282
Third	0.828	-0.488	0.336

Table 13: Testing different PSNE implementations. Table values are calculated according to footnote 2

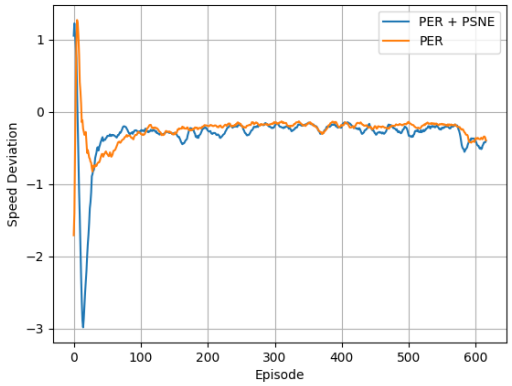
This modified version of PSNE, led to better results, reducing mean collision into 0.282 while increasing the speed by 0.072 m/s.

Algorithm implemented	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
PER	0.848	-0.344	0.342
PER + PSNE	0.842	-0.416	0.282

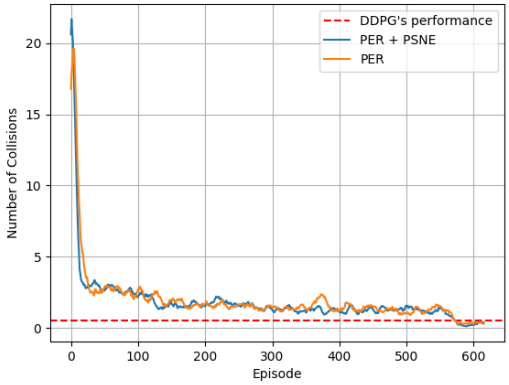
Table 14: Comparing PER with PER and PSNE. Table values are calculated according to footnote 2



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).

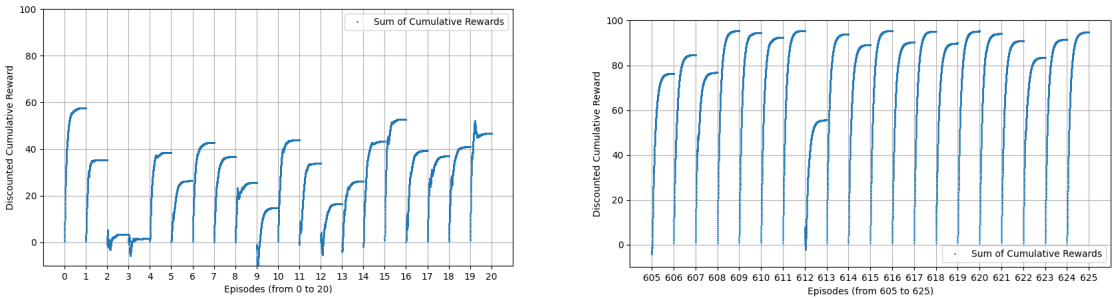


(c) Total collisions during an episode - mean over seeds.

Figure 22: Comparing PER with PER and PSNE. Graphs are displayed according to footnote 3

In conclusion, the addition of the PSNE to the existing PER+NAF agent was lead the agent to drive at a lower speed (+19% from [10] while the PER+NAF agent was driving at +33%) while achieving fewer collisions (-46% while the NAF+PER agent was involved in -34%).

In addition, Figure 23 displays the Discounted Cumulative Reward per episode in different instances of the training process. As shown in the Subfigure 23a, the agent explores the environment a clear strategy for maximizing the G_T (Equation 1). However, during the late stages of training (Subfigure 23b), the G_T does not increase significantly between episodes. Hence, the agent converges to a policy.



(a) Discounted Cumulative Reward during the first 20 episodes.

(b) Discounted Cumulative Reward during the last 20 episodes.

Figure 23: Discounted Cumulative Reward

As shown in Figure 24, the training process improves the policy of the agent, as the latter increases the sum of discounted rewards that earns from the environment. This can be also observed in more detail by comparing Subfigures 23a and 23b). It is noteworthy that the observed inconsistencies in the "pulses" can be attributed to both the low number of seeds and the Boltzmann exploration.

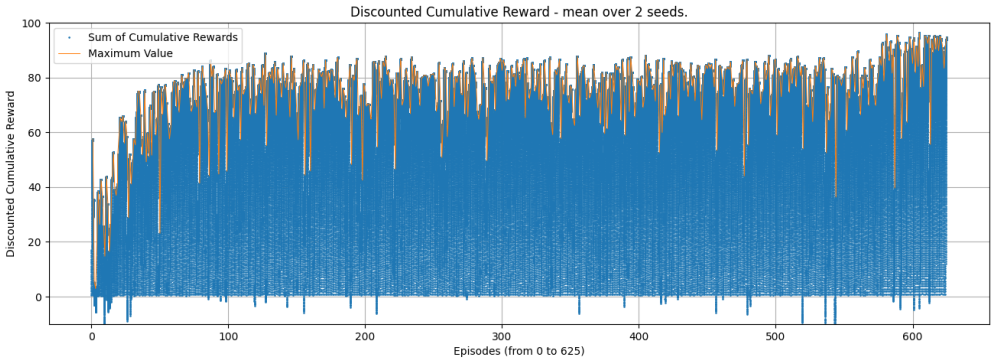
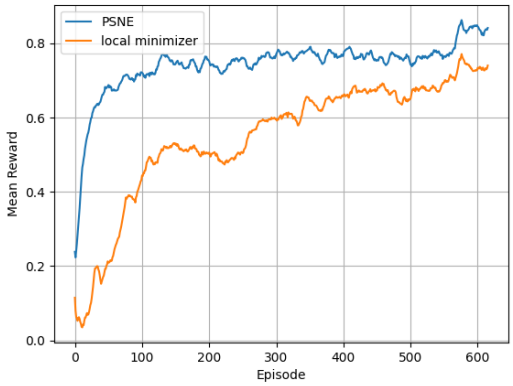


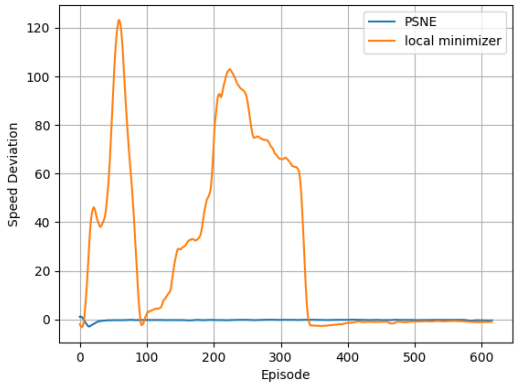
Figure 24: Discounted Cumulative Reward throughout the training.

4.4.9 Implementing a local minimizer

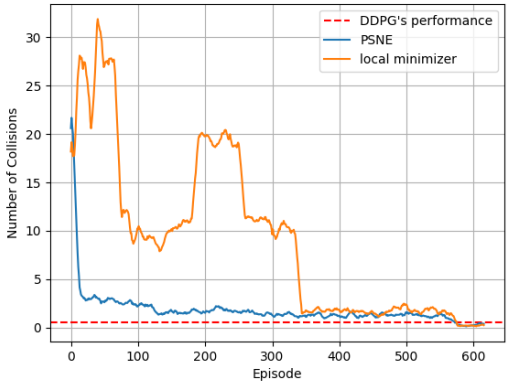
As for the last experiment, PSNE was replaced by the local minimizer as described in Section 3.2.1. Specifically, after the 10th episode, the agent begin to explore the environment by selecting moves that minimize the cost function (Equation 35) with initial probability equal to 0.5. The ten episode head start period was utilized to eliminate the risk of no convergence as explained in Section 4.4.8. Additionally, imagination rollouts (as explained in Section 3.2.1) were performed while the agent was acting semi-optimally, to complete an exploration phase that both explores promising and not promising actions.



(a) Mean - both over episode and over seeds - reward yielded by the agent.



(b) Mean - both over episode and over seeds - speed deviation from the desired speed (20 m/s).



(c) Total collisions during an episode - mean over seeds.

Figure 25: Comparing PSNE with local minimizer. Graphs are displayed according to footnote 3

As it seems from the results of that experiment (Figure 25), that exploration technique did not help the agent to optimize the yielded reward. More precisely, exploration using the local minimizer achieves -0.101 less reward, while driving slower (-51% than [10] while the NAF+PER+PSNE was driving at +19%) than the PSNE.The local minimizer, by contrast, achieves slightly fewer collisions (-52% than [10] while the NAF+PER+PSNE was involved in -46% than [10]), according to Table 15. The basic disadvantage of that implementation was that the whole minimization was based on predictions of the future state. In the lane-free environment, the accuracy of this prediction couldn't be accurate due to difficulties to predict the policy of the non-trainable vehicles. However, the imagination rollouts, also depended on that prediction. The ineffective way of producing "bad" experiences, lead the agent to be trained in a irrelevant and unrealistic set of data. In addition to that, this algorithm uses many hype-parameters that need extra fine-tuning. Concluding, PSNE and pure Deep-RL methods seem to build better policies in the aforementioned lane-free environment.

Exploration strategy	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions
Local Minimizer	0.741	-1.038	0.248
PSNE	0.842	-0.416	0.282

Table 15: Comparing PER with PER and PSNE. Table values are calculated according to footnote 2

4.5 Discussing the Experiments

The conducted experiments showed, that the introduced algorithms improve the agent's performance. More precisely, the use of the NAF algorithm, lead the agent to drive safer (-0.2 collisions) while maintaining the same speed as [10]. With the addition of PER, the (NAF+PER) agent yielded a better reward by driving faster, but also it was involved in more collisions than the NAF agent (+0.026). PSNE was also added, to improve the exploration strategy of the agent. The reward of the NAF+PER+PSNE agent, was slightly decreased while both the collisions and the driving speed were decreased too, compared to the NAF+PER implementation.

Finally the NAF+PER+PSNE was compared with the NAF+PER+Local exploration agent, as the latter was inspired by [12]. This implementation did not yield positive results, as its exploration was based on a simpler version of the local dynamics - as it was difficult to optimize the full version without any prior knowledge. In conclusion, the results of every algorithm combination, are presented in Table 16.

Algorithms	Avg. Reward	Avg. Speed Deviation (m/s)	Avg. Collisions	Speed difference from [10]	Collision difference from [10]
NAF	0.814	-0.524	0.316	-2%	-39%
NAF+PER	0.848	-0.344	0.342	+33%	-34%
NAF+PER+PSNE	0.842	-0.416	0.282	+19%	-46%
NAF+PER+Local Minimizer	0.741	-1.038	0.248	-51%	-52%

Table 16: Statistics of all the combination

5 Conclusions and Future Work

AI is actively working on providing solutions to many real-world problems. Numerous methods to create autonomous drivers have been proposed, using either Control Theory ([7], [54], [55]) or Deep-RL ([9], [10], [49]). In this thesis, a novel algorithm that combines modern Deep-RL techniques (NAF, PER and PSNE) was proposed, to train a single autonomous agent to operate in a lane-free traffic environment, modeled as an MDP, as in [10]. In our work, the reward function designed in [1] was modified, in order for the agent to yield better results. We then progressively created different working versions of our agent (i.e., NAF; NAF+PER, NAF+PER+PSNE) via systematically and carefully combining the aforementioned algorithmic components, which was a non-trivial task. Finally, a novel combination of NAF (and PER) with PSNE was proposed, which takes advantage of both exploration strategies, finally achieving the best results. This was showcased via a systematic experimentation process in the lane-free traffic domain. As such, our thesis, experimentally demonstrates that the NAF algorithm, with the original Boltzmann exploration, outperforms the DDPG on the same environmental setting. In addition, our proposed combination of the Boltzmann exploration with PSNE, lead the agent to drive faster and with fewer collisions. It is noteworthy that all algorithms were fine-tuned properly throughout the experimental evaluation process, in order for the agent to yield better results.

However, there is always room for improvement, and much space for future work. First, NAF can be re-fine tuned using different combinations of hyper-parameters. Additionally, the quadratic form of A (Equation 23) can be replaced by forms that result in better behavior. Hence NAF can also be replaced by PPO [63], as the latter, has been proved to provide a better convergence and performance rate than most Deep-RL algorithms.

Also the Reward function can be modified, by eliminating the repulsive force component and creating a more sparsed reward function. In case of a sparse reward function, PER should be replaced with an experience replay buffer more compatible for such kinds of reward functions, like Hindsight Experience Replay (HER) [64]. In case of HER, the agent can be also trained more efficient using the aforementioned "two-goal" reward function. This is due to HER's ability to optimize the training process when the reward function models multiple goals.

Furthermore, single agent training can be replaced by multi-agent training components (like MADDPG [65] or a new algorithm that is inspired by NAF), in order for multiple agents to be trained an eventually learn to cooperate, while yielding in general better results. Finally, all the aforementioned suggestions, should ideally be applied not only on highway environments, but also in more complex settings like circle road or highways with junctions.

References

[1] A. Karalakou, D. Troullinos, G. Chalkiadakis, and M. Papageorgiou, "Deep reinforcement learning reward function design for autonomous driving in lane-free traffic", *Systems*, vol. 11, no. 3, 2023, ISSN: 2079-8954. DOI: 10.3390/systems11030134. [Online]. Available: https://www.mdpi.com/2079-8954/11/3/134.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning", Dec. 2013. arXiv: 1312.5602 [cs.LG].

[3] C. J. Watkins, "Q-learning", *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1989.

[4] L. Lin, "Reinforcement learning for robots using neural networks", 1992. [Online]. Available: https://api.semanticscholar.org/CorpusID:60875658.

[5] S. Aradi, *Survey of deep reinforcement learning for motion planning of autonomous vehicles*, Jan. 2020.

[6] X. Di and R. Shi, "A survey on autonomous vehicle control in the era of mixed-autonomy: From physics-based to ai-guided driving policy learning", *Transportation Research Part C: Emerging Technologies*, vol. 125, p. 103 008, 2021, ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2021.103008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0968090X21000401.

[7] M. Papageorgiou, K.-S. Mountakis, I. Karafyllis, and I. Papamichail, "Lane-free artificial-fluid concept for vehicular traffic", May 2019. arXiv: 1905.11642 [cs.SY].

[8] V. K. Yanumula, P. Typaldos, D. Troullinos, M. Malekzadeh, I. Papamichail, and M. Papageorgiou, "Optimal path planning for connected and automated vehicles in lane-free traffic", in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021, pp. 3545–3552. DOI: 10.1109/ITSC48978.2021.9564698.

[9] I. Chrysomallis, D. Troullinos, G. Chalkiadakis, I. Papamichail, and M. Papageorgiou, "Deep reinforcement learning with implicit imitation for lane-free autonomous driving", in Sep. 2023, ISBN: 9781643684369. DOI: 10.3233/FAIA230304.

[10] A. Karalakou, "Deep reinforcement learning reward function design for lane-free autonomous driving", Technical University of Crete, 2022. DOI: https://doi.org/10.26233/heallink.tuc.92889.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning", Sep. 2015. arXiv: 1509.02971 [cs.LG].

[12] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration", Mar. 2016. arXiv: 1603.00748 [cs.LG].

[13] L. C. Baird III, "Advantage updating", Wright Lab Wright-Patterson AFB OH, Final Report, Nov. 1993.

[14] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", Nov. 2015. arXiv: 1511.05952 [cs.LG].

[15] M. Plappert, R. Houthooft, P. Dhariwal, *et al.*, "Parameter space noise for exploration", Jun. 2017. arXiv: 1706.01905 [cs.LG].

[16] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[17] O. Simeone, "A very brief introduction to machine learning with applications to communication systems", *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 4, pp. 648–664, 2018. DOI: 10.1109/TCCN.2018.2881442.

[18] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge: MIT Press, 1998, vol. 1.

[19] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press, 1960.

[20] P. N. Ward, "Linear programmin in reinforcement learning", *McGill University, Montreal*, 2021.

[21] D. Hein, "Interpretable reinforcement learning policies by evolutionary computation", 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:207759156.

[22] M. M. Fard and J. Pineau, "Non-deterministic policies in markovian decision processes", Jan. 2014. arXiv: 1401.3871 [cs.AI].

[23] D. Kalise, K. Kunisch, and Z. Rao, Eds., *Numerical Methods and Applications in Optimal Control*. Berlin, Boston: De Gruyter, 2018, ISBN: 9783110543599. DOI: doi:10.1515/9783110543599. [Online]. Available: https://doi.org/10.1515/9783110543599.

[24] M. Heger, "Consideration of risk in reinforcement learning", in *Machine Learning Proceedings 1994*, Elsevier, 1994, pp. 105–111.

[25] M. Sato and S. Kobayashi, "Variance-penalized reinforcement learning for risk-averse asset allocation", in *Intelligent Data Engineering and Automated Learning — IDEAL 2000. Data Mining, Financial Engineering, and Intelligent Agents*, K. S. Leung, L.-W. Chan, and H. Meng, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, ISBN: 978-3-540-44491-6.

[26] P. Dayan and C. Watkins, "Q-learning", *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[27] O. Berger-Tal, J. Nathan, E. Meron, and D. Saltz, "The exploration-exploitation dilemma: A multidisciplinary framework", *PLOS ONE*, vol. 9, pp. 1–8, Apr. 2014. DOI: 10.1371/journal.pone.0095693. [Online]. Available: https://doi.org/10.1371/journal.pone.0095693.

[28] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion", *Phys. Rev.*, vol. 36, pp. 823–841, 5 Sep. 1930. DOI: 10.1103/PhysRev.36.823. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.36.823.

[29] A. Singh, "Reinforcement learning based empirical comparison of ucb, epsilon-greedy, and thompson sampling", *Int. J. of Aquatic Science*, vol. 12, no. 2, pp. 2961–2969, 2021.

[30] Y. Li, J. Cui, X. Zhang, and X. Yang, "A ship route planning method under the sailing time constraint", *Journal of Marine Science and Engineering*, vol. 11, no. 6, 2023, ISSN: 2077-1312. DOI: 10.3390/jmse11061242. [Online]. Available: https://www.mdpi.com/2077-1312/11/6/1242.

[31] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015, vol. 25.

[32] D. E. Rumelhart and J. L. McClelland, "Learning internal representations by error propagation", in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. 1987, pp. 318–362.

[33] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark", Sep. 2021. arXiv: 2109.14545 [cs.LG].

[34] T. Chai and R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature", *Geoscientific Model Development*, vol. 7, pp. 1247–1250, 2014. DOI: 10.5194/GMD-7-1247-2014.

[35] Z. Zhang and M. R. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels", May 2018. arXiv: 1805.07836 [cs.LG].

[36] H. E. Robbins, "A stochastic approximation method", *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951. [Online]. Available: https://api.semanticscholar.org/CorpusID:16945044.

[37] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", Dec. 2014. arXiv: 1412.6980 [cs.LG].

[38] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html.

[39] G. Tesauro *et al.*, "Temporal difference learning and td-gammon", *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[40] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning", Sep. 2015. arXiv: 1509.06461 [cs.LG].

[41] H. Hasselt, "Double q-learning", in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23, Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[42] M. Harmon and L. Iii, "Residual advantage learning applied to a differential game", Sep. 1998. DOI: 10.1109/ICNN.1996.548913.

[43] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning", Nov. 2015. arXiv: 1511.06581 [cs.LG].

[44] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. A. Riedmiller, "Deterministic policy gradient algorithms.", in *ICML*, ser. JMLR Workshop and Conference Proceedings, vol. 32, JMLR.org, 2014, pp. 387–395. [Online]. Available: http://dblp.uni-trier.de/db/conf/icml/icml2014.html#SilverLHDWR14.

[45] C. Petridis, "A multi-modal q-learning approach using normalized advantage functions and deep neural networks", Diploma Work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2019. [Online]. Available: https://doi.org/10.26233/heallink.tuc.82891.

[46] S. Kullback and R. A. Leibler, "On information and sufficiency", *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, 1951.

[47] M. Plappert, "Parameter space noise for exploration in deep reinforcement learning", Master's Thesis, Karlsruhe Institute of Technology (KIT), 2017.

[48] W. B. Knox, A. Allievi, H. Banzhaf, F. Schmitt, and P. Stone, "Reward (mis)design for autonomous driving", Apr. 2021. arXiv: 2104.13906 [cs.LG].

[49] P. R. Wurman, S. Barrett, K. Kawamoto, *et al.*, "Outracing champion gran turismo drivers with deep reinforcement learning", *Nature*, vol. 602, pp. 223–228, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246701687.

[50] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Duerr, "Super-human performance in gran turismo sport using deep reinforcement learning", Aug. 2020. arXiv: 2008.07971 [cs.AI].

[51] C. Wu, A. Kreidieh, K. Parvate, E. Vinitsky, and A. M. Bayen, "Flow: A modular learning framework for mixed autonomy traffic", Oct. 2017. arXiv: 1710.05465 [cs.AI].

[52] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving", Apr. 2017. arXiv: 1704.02532 [stat.ML].

[53] M. Rahman, M. Chowdhury, Y. Xie, and Y. He, "Review of microscopic lane-changing models and future research opportunities", *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 4, pp. 1942–1956, 2013. DOI: 10.1109/TITS.2013.2272074.

[54] I. Karafyllis, D. Theodosis, and M. Papageorgiou, "Two-dimensional cruise control of autonomous vehicles on lane-free roads", Mar. 2021. arXiv: 2103.12205 [math.OC].

[55] D. Troullinos, G. Chalkiadakis, I. Papamichail, and M. Papageorgiou, "Collaborative multiagent decision making for lane-free autonomous driving", ser. AAMAS '21, Virtual Event, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, 2021, pp. 1335–1343, ISBN: 9781450383073.

[56] V. K. Yanumula, P. Typaldos, D. Troullinos, M. Malekzadeh, I. Papamichail, and M. Papageorgiou, "Optimal trajectory planning for connected and automated vehicles in lane-free traffic with vehicle nudging", Jul. 2022. arXiv: 2207.09670 [eess.SY].

[57] D. Troullinos, G. Chalkiadakis, D. Manolis, I. Papamichail, and M. Papageorgiou, "Extending sumo for lane-free microscopic simulation of connected and automated vehicles", *SUMO Conference Proceedings*, vol. 3, pp. 95–103, Sep. 2022. DOI: 10.52825/scp.v3i.110.

[58] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An imperative style, high-performance deep learning library", Dec. 2019. arXiv: 1912.01703 [cs.LG].

[59] E. Todorov and W. Li, "A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems", in *Proceedings of the 2005, American Control Conference, 2005.*, 2005, 300–306 vol. 1. DOI: 10.1109/ACC.2005.1469949.

[60] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0–fundamental algorithms for scientific computing in python", Jul. 2019. arXiv: 1907.10121 [cs.MS].

[61] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture", vol. 7, Oct. 2007, pp. 103–104. DOI: 10.1145/1296907.1296909.

[62] C. R. Harris, K. J. Millman, S. van der Walt, *et al.*, "Array programming with numpy", *CoRR*, vol. abs/2006.10256, 2020. arXiv: 2006.10256. [Online]. Available: https://arxiv.org/abs/2006.10256.

[63] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", Jul. 2017. arXiv: 1707.06347 [cs.LG].

[64] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, "Hindsight experience replay", Jul. 2017. arXiv: 1707.01495 [cs.LG].

[65] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments", Jun. 2017. arXiv: 1706.02275 [cs.LG].