FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Computer Vision and explainable Reinforcement Learning applied to a self-driving following car

Rafael de Jesus Guedes

DISSERTATION

MASTER THESIS

Advisor: António Pedro de Aguiar

09/09/2022

Computer Vision and explainable Reinforcement Learning applied to a self-driving following car

Rafael de Jesus Guedes

Master Thesis

09/09/2022

Abstract

The interest in Reinforcement Learning has been growing over the last few years since it gives the opportunity to create agents capable of learning complex tasks by themselves without the necessity of being hard coded. This is possible because the training of these agents is performed with interactions with simulators that tend to look like real world environments. For some less complex tasks, the agents can, also, be developed in real world controlled environments. By interacting with the environment, the agent starts learning what should be performed in order to accomplish a goal based only on the environment perceived and the possible action it can take. In that way, one does not need to think about every possible state that the environment can have because the agent will learn how to act in every situation if it trains long enough.

Scaling Reinforcement Learning principles to self-driving car is an active and huge area of research with many challenges still to be tackled. One of the main challenges, apart from ethic related issues, is the ability to have one car being driven by a human being (leader) and another car following the behavior of the leader without any human intervention. This represents a huge opportunity for supply chain of many companies and for human transportation companies in the sense that these companies can transport twice (or more) goods or people with only one driver.

As mentioned, ethic is a hot topic within the Artificial Intelligence area nowadays because the reasoning behind the decisions taken by the algorithms must be explained and well understood. Also, backup plans must be put into action when the decision is abnormal and can negatively impact people's lives.

Motivated by the above, this dissertation proposes a Proximal Policy Optimization architecture in a simulated environment for an agent capable of following another car from one point to another using only computer vision, while avoiding crashing to the car leader and keeping approximately the same safe distance from the car leader. To cope with the ethic area, this dissertation will use hard constraints to avoid abnormal decisions from the agent when facing unseen situations and SHAP values to analyze and understand the impact of each environment characteristic on the predicted value.

Contents

List of Figures

List of Tables

Abbreviations

DL	Deep Learning
CV	Computer Vision
RL	Reinforcement Learning
AI	Artificial Intelligence
IoT	Internet of Things
PPO	Proximal Policy Optimization
ANN	Artificial Neural Network
CNN	Convolution Neural Network
SG	Semantic Segmentation
VAE	Variational Auto Encoder
OR	Object Recognition
DQL	Deep Q-Learning
AC	Actor-Critic
SAC	Soft Actor-Critic
MB	Model Based
PB	Policy Based
CNN	Convolution Neural Network
SHAP	Shapley Additive Explanations
KL	Kullback-Leibler
TRPO	Trust Region Policy Optimization
MAE	Mean Absolute Error

Chapter 1

Introduction

Reinforcement Learning (RL) is an area of Artificial Intelligence (AI) that tries to train an agent to make a sequence of decisions. This agent learns how to make the best decision based on the current state of the environment through the past collected experience of trial-and-error experiments. The sense of what is a good decision based on the current state is achieved by providing negative or positive rewards for the actions the agent performs. The agent's goal is to maximize the total reward.

RL is different from Supervised Learning in the way that no hint or suggestion of what should be done is provided to the agent and it needs to figure out what actions allow the agent to maximize its reward, starting completely random and finishing with refined skills.

In other words, the process can be easily explained as:

1. The agent is in a certain state.
2. The agent collects information from the environment and based on previous experiences decides the next action to be taken.
3. The agent takes the action which modifies the environment.
4. The environment responds with:
    a. A scalar reward which allows the agent to understand the positive or negative impact of its action.
    b. A new state for the agent.
5. The agent learns and the process starts again.

As stated above and as shown in Figure 1 – , this whole process is a closed loop that allows to develop autonomous vehicles since there is no need to literally program every action the vehicle needs to take when facing a certain state. Moreover, it would be impossible to define every single state that a vehicle faces when it is on road due to the unpredictable behavior of human drivers. That is why Reinforcement Learning could be a game changer in the autonomous vehicle field.[1]

[1] https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/

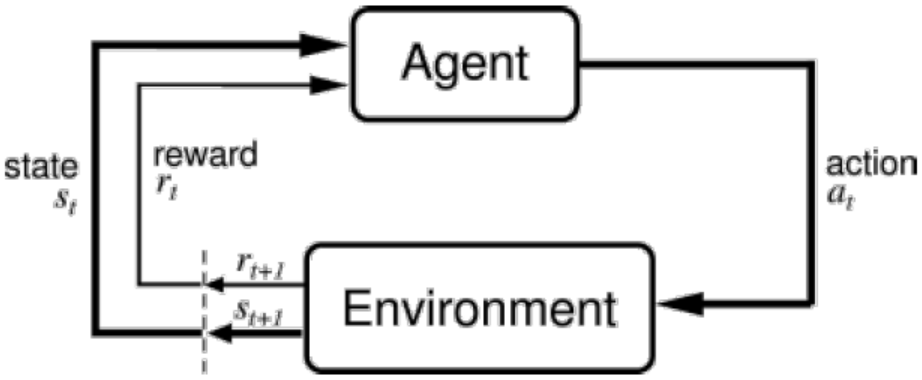

Figure 1 – Representation of the interaction between the agent and the environment.

Source: [53]

1.1 Context and Motivation

In the last two decades, the world has experienced a huge advance in technology that aims to automate and increase efficiency of the tasks performed by the human being. One of the most recently investigated tasks are related to autonomous vehicles. It does not only reduce the time lost in commutes but, also, has a potential positive impact in a supply chain, more specifically in the delivery part.

Although, the idea of having autonomous vehicles is old and well-known, it was only recently that the world has seen companies and universities researching this technology. This happens because of the advances in the Artificial Intelligence field such as Deep Learning (DL), Computer Vision (CV) and RL.

However, not only the development on the AI field contributed to advancements in autonomous vehicles but, also, the development in Internet of Things (IoT) which enabled smart agents to communicate with each other. The combination of both fields allows to create supply chains more efficient, sustainable, and faster.

By creating several autonomous trucks that can drive in platoon formation, one is taking advantage of AI technologies in the sense the trucks can drive autonomously using CV and Deep RL and, is also taking advantage of IoT since those trucks can communicate to share information about the road, other vehicles and, more importantly, their position in order to facilitate the platooning process.

One knows that there are still ethic concerns about completely autonomous vehicles. However, this platooning process will allow to use in the truck leader a human driver. The truck leader will be followed by the remaining trucks which are trying to replicate the human driver behavior and, in that way, nearly surpass (or at least minimize) the ethic problem as shown in Figure 2. Nevertheless, the autonomous trucks still need to assess dangerous and act according to what its perceived in the environment, but most of the behavior of these trucks will be indirectly defined by the human driver.

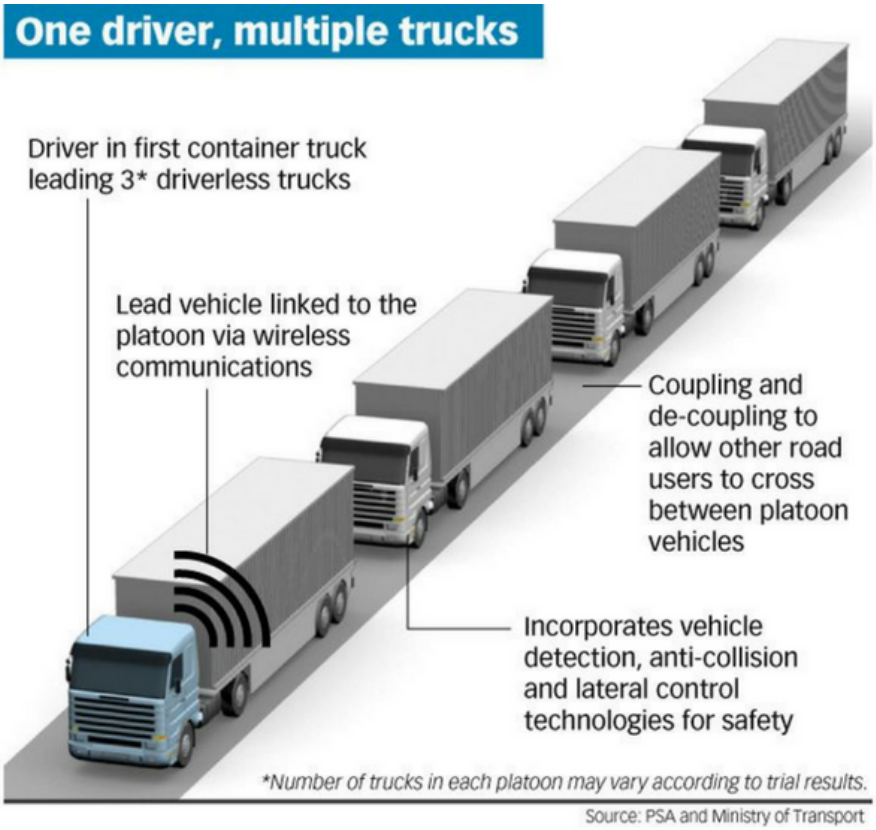

Figure 2 - Example of Platooning Trucks

The potential positive impact is significant both for companies and society. With platooning trucks, one would benefit on efficiency because it would be able to deliver more quantity at the same time reducing costs with human assets (one driver can drive three trucks instead of one, for example) and reduce the fuel spent in the trip by taking advantage of the air tunnel created by the leader truck which reduces the fuel spent by the follower trucks. With the decrease on fuel spent, one is also contributing to reduce the carbon footprint which is a major

concern in climate changes. And, finally, this process will also smooth the traffic and reduce the probability of human error in the road, since there are fewer human drivers.

After this contextualization, the general goal of this dissertation is to create the platooning process described above. The next section will describe in more detail the main objectives of this dissertation and its contributions.

1.2 Objectives and Contributions

This dissertation is composed by four main goals, the first one it to create a self-driving car which is able to avoid crashing and follow a leader (another self-driving car) performing what it is called a platooning vehicle using PPO. The second one is to only rely on CV methods to process the information retrieved by a monocular camera. The third one is to add a layer of safety that does not allow the agent to reach dangerous velocities or perform sudden turns. Finally, the fourth and final goal is to explain and understand the agent's reasoning that leads to certain decision. This dissertation will also evaluate and discuss the performance of the self-driving car according to suitable measures of quality for both training and test phases.

Besides the decision about what RL algorithm is the most suitable for this type of problem, the design of a proper reward function and the definition of the state environment is crucial for a good performance of the agent.

On one hand, the reward function will be responsible for correctly guiding the agent through the stated environment, avoiding crashing into the leader, keeping the car in the lane and, ultimately, following the car leader. On the other hand, the state of the environment is going to be important so that the agent can be aware about what is going on around it. Although RL algorithms have shown an impressive performance on this task, they still rely on a good statement of the reward function and the environment, otherwise it will not be successfully implemented.

The subsection 1.2.2. presents the reasons behind the choice of the RL algorithm taking into consideration the task to be performed and what already exists to be outperformed or to be complemented with new capabilities developed during this thesis.

Additionally, the subsection 1.2.4 will present the methodology for the accomplishment of the proposed work for this dissertation.

1.2.1 Task

The task that this dissertation aims to tackle is to drive a car from one point to another while following another car with a well-defined distance, avoiding crashing into the leader and being positionally centered according to the leader position. The choice of this task was motivated by trying to simulate the behavior presented in subsection 1.1 of platooning vehicles.

The problem to be solved consists in learning a policy that drives the car in the direction of the waypoint-goal, while following the car leader when it appears in front of the car. The velocity of the car will depend on the velocity of the leader car assuming null and positive values and the available control actions will be discrete steering actions and continuous throttle actions. The obstacles present in training process could be static such as buildings or dynamic such as other vehicles and people.

To create the state needed for the RL algorithm to work correctly, a monocular camera will be used to retrieve visual information which will allow the agent to know if the leader is on the left, ahead or on the right and to calculate the distance to the leader.

Moreover, the safety and explain ability play an important role in this task. The safety part will help to mitigate bad decisions that the agent can take due to the novelty of an unseen situation and the explain ability part will be responsible for understanding the reason for any decision taken based on the environment perceived and, in that way, increase the trustiness on the agent's actions.

1.2.2 Method

This dissertation aims to reproduce a Model-Free method sustained by PPO that can be applied to platooning vehicles and obstacle avoidance. Apart from proving the applicability of the RL method for this task, it also aims to show that providing to the agent the minimum number of sensors is enough to perceive the environment and act accordingly to what it is perceived.

The choice of PPO is based on the ability of this algorithm to allow the train of agents in a continuous action space, which is crucial for their application as controller on a real-world system like the self-driving car with the advantage of having a simpler implementation, better performance, and better sample efficiency [44].

For the processing of the information retrieved by the camera, the state-of-the-art techniques for semantic segmentation namely AE (Auto Encoder) could be applied but since the

simulator already has an in-built segmentation camera, then this is going to be used. This is a computer vision method that is able to classify each pixel with a label that will allow to identify the position of the car leader in an image.

The safety layer will be created through hard constraints that can be calculated based on the data retrieved with the sensors such as the safety distance and it will only be used during the testing phase. If the agent has this safety layer during training, it can compromise what the agent learns because it will not allow the agent to explore and commit errors.

Finally, the explain ability will be accomplished using the data recorded during the simulations and using the SHAP values [31] (explained in more detail in Chapter 2) to understand how each feature impacts the agent's decision.

The contribution of this dissertation is supported by a successful state representation as well as a reward function that enables to accomplish the task of platooning vehicles and obstacle avoidance with the minimum sensors possible to process information about the environment. Moreover, it will contribute for the explainable AI community by analyzing methodically the impact of each environment characteristic on the decision process.

1.2.3 Metrics

As usual in a machine learning task, it is crucial to assess the performance of the model.

This evaluation is done in two different periods: the training phase where the model adjusts its parameters while trying to achieve the goal for what it was created; and, the testing phase where the model is assessed in terms of the generalization capability of performing well with unseen data.

1. **Training phase**

During this period, the objective is to monitor the convergence of the algorithm according to specific variables such as:

- The difference between the predicted reward and the actual reward.
- The cumulative reward by episode which allows to understand if the agent is learning the correct behavior and performing well. Depending on the designing of the reward function the fact the agent accumulates more positive rewards or less negative rewards means that it is improving towards the goal by maximizing the cumulative reward.

- A visual check in order to identify problems on the state representation or reward design that are influencing the agent to have undesirable behaviors which is also known as reward tempering [11].

For the training phase the following metrics will be considered:

M1 The cumulative throttle reward by episode (RL).

M2 The cumulative steer reward by episode (RL).

M3 The reason for the ending of the episode (RL).

M4 The cumulative actions by episode (RL).

M5 Mean Absolute Error of predicted distance using CNN (CV).

2. **Testing phase**

During this phase the objective is to assess if the agent is able to go from point A to B following the leader while avoiding crashing into the leader and performing the expected behavior without any hack.

For the testing phase the following metrics will be considered:

M6 The cumulative throttle reward by episode (RL).

M7 The cumulative steer reward by episode (RL).

M8 The reason for the ending of the episode (RL).

M9 The cumulative actions by episode (RL).

M10 The average distance from the leader by episode (RL).

M11 The average speed by episode (RL).

M12 The number of successful episodes over the total number of episodes (RL).

M13 Mean Absolute Error of predicted distance using CNN (CV).

1.2.4 Methodology

To develop this dissertation, CARLA simulator (Figure 3) was used.

Figure 3 – CARLA

Source: https://carla.org/

CARLA [8] is an open-source simulator for autonomous driving research developed by Intel, Toyota, and a Computer Vision Center in Barcelona. This simulator provides urban layouts, building and vehicles to support the development of urban driving systems. Moreover, it supports flexible specification of sensors to be incorporated in the car like camaras, radars, lidars and other sensor, and, also, different environmental conditions such as sunny, cloudy, or rainy days and day or night setups. *Figure 4* illustrates some of these features.

Figure 4 - Different environment conditions present in CARLA

Source: [8]

CARLA, also, has a Python API where it is possible to control the car and program the state representation, the possible actions, the reward functions and embed CV algorithms for object recognition. Therefore, Python[2] will be the programming language used in this dissertation.

The explain ability of the decision process will resort to the SHAP (Shapley Additive Explanations) library[3] which is a game theoretic approach to explain the output of any machine learning model. Game theory is the process of modeling the strategic interaction between two or more players (in the case of machine learning, two or more features) in a situation containing a set of rules and outcomes in which each player's payoff is affected by the decision made by others. In other words, the contribution of each feature is determined by what is gained or lost by removing them from the model. One of the possible outcomes is presented in Figure 5 where it is observable that lower values (blue color) of *'LSTAT'* lead to higher values in the predicted value, while higher values (red color) of *'RM'* lead to higher values in the predicted value.

[2] https://www.python.org/

[3] https://shap.readthedocs.io/en/stable/index.html

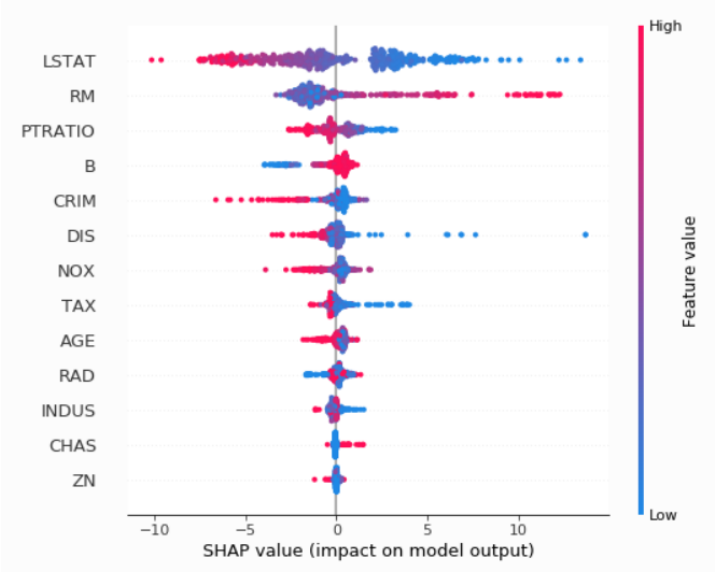

Figure 5 – This plot is design to display an information-dense summary of how the top features in a dataset impact the model's output. Each instance the given explanation is represented by a single dot on each feature. Source:https://shap.readthedocs.io/en/stable/example_notebooks/tabular_examples/tree_based_models/Catboost%20tutorial.html

1.3 Organization of the Dissertation

The dissertation starts with Chapter 1 that introduces the dissertation by providing the context and the motivation for the work proposed, as well as the objectives, the problem statement and the contributions of the work developed.

Chapter 2 reviews the state-of-the-art techniques for Reinforcement Learning, Deep Learning and Computer Vision, techniques for increasing safety of RL models and for explain ability and the studies already performed in self-driving cars and platooning vehicles.

Chapter 3 presents the reason for the RL algorithm chosen.

Chapter 4 describes the CNN model used to predict distances.

Regarding the practical part, Chapter 5 exposes the implementation of the RL and CV methods, using PPO and Xception, respectively, and the results obtained.

Finally, Chapter 5 draws the conclusion about the work and leaves future works perspectives.

Chapter 2

Literature Review

This chapter brings to light the state-of-the-art and an overview of the most recent literature related to Reinforcement Learning applied to Self-driving and Platooning vehicles.

Section 2.1 introduces the main applicable RL methods in the literature.

Section 2.2 gives a briefly introduction about the CV methods available.

Section 2.3 presents the safety layers developed to guarantee always safe decisions by RL models.

Section 2.4 shows one example of what can be done to increase the visibility on the reason behind each agent's decision.

And, finally, section 2.5 introduces the main challenges RL is facing in the Self-Driving and Platooning vehicles and, also, the state-of-the-art examples already developed.

2.1 RL Methods

This section aims to provide an overview of the existing models for RL which is summarized in *Figure 6*.

There are two main types of RL models: the Model-Based and Model-Free. The main difference between both models is whether the agent has access to a model of the environment or not.

Using the Model-Free method the agent does not need a model of the environment, which is learned based on its own experience (trial-and-error).

Using Model-Based methods the agent needs to have a prior knowledge of the environment to be able to plan the next action once it predicts state transitions and rewards based on an action.

The main advantage of having a Model-Based method is that it allows the agent to plan by thinking ahead and see what would happen for a set of actions and explicitly decide the action which maximizes its reward. The main disadvantage is that the model of the environment is in many cases unavailable, and the agent needs to learn it. This raises the problem that the model may

not truly capture the environment, and in consequence the RL may exploit potential artificial bias in the model making it have poor performance in the real environment.[4]

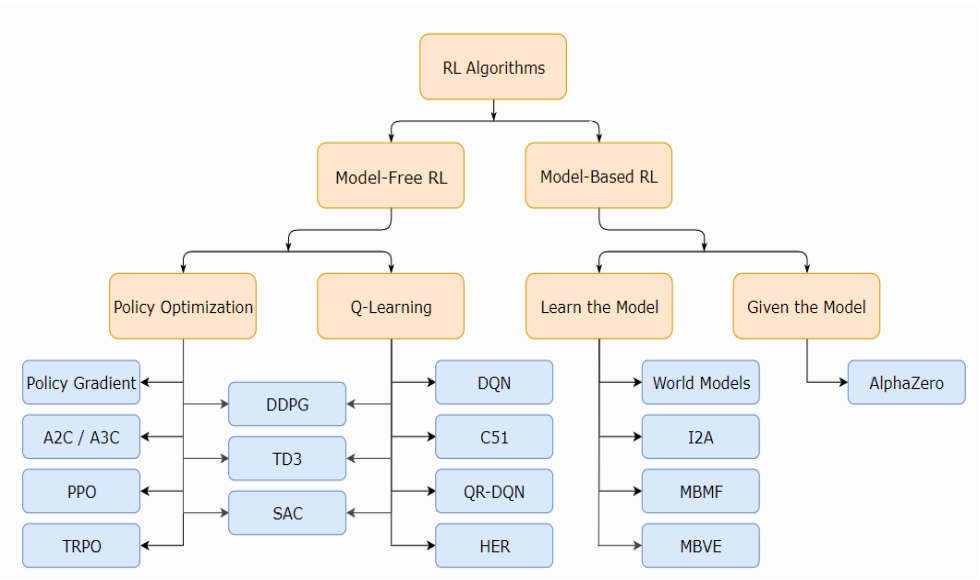

Figure 6 - Different RL models developed

Source: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

2.1.1 Model-Free

RL Model Free methods are a set of algorithms that tries to learn how to act in a certain environment (based on a policy, action-value function, or both) by trial-and-error through interactions with the environment.

As already mentioned, this methods do not know and do not have access to the transition dynamics of the environment $T(s', a, s)$, where s' denotes the current state, a the action and s the next state. Therefore they learn and search the optimal policy π^* through the experience collected with the interactions with the environment.

Model Free methods are more explored than Model Based methods because they are easy to implement since it does not require prior knowledge about the environment [34], and they can learn:

- Policy function $\pi_\theta (a|s)$.
- Action-value function $Q_w(s,a)$.
- Both Policy function $\pi_\theta (a|s)$ and Action-value function $Q_w(s,a)$.

[4] https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

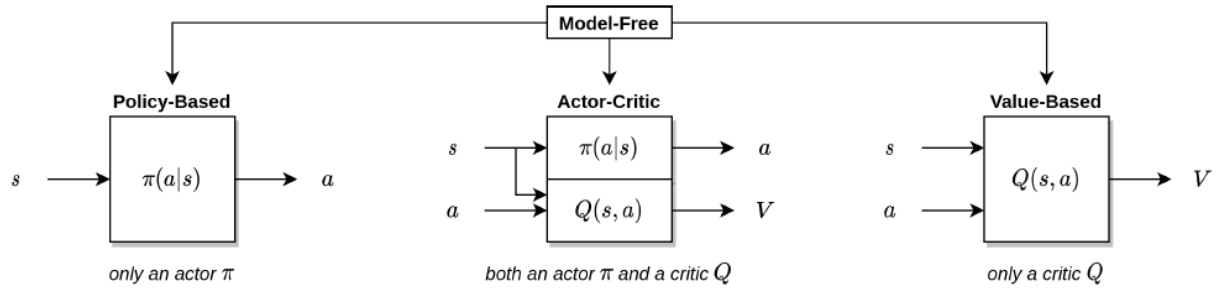

Figure 7 - Three possible options to implement Model-Free

2.1.1.1 Policy-Based

Policy-Based is a type of method that learns a policy π_θ, being the behavior of the agent controlled by the parameter θ [53] . Using these parameterized policies, the agent can act without consulting value-functions as it does in a Value-Based method. Those parameters are learned through an iterative learning process in order to maximize the agent's performance in a certain environment.

This method is pragmatic in finding and optimizing the policy needed, basically a map from states to actions. There are two types of policies, stochastic and deterministic.

Stochastics policies can be represented by the probability distribution over actions $a \in A(s)$ given a state $s \in S$: $a = \pi(s, \theta) = Pr\ (S_t = s, \theta_t = \theta)$. Whereas, deterministic policies can be represented as a straightforward mapping from states to actions $\alpha = \mu(s, \theta)$. The ultimate objective is to find the optimal policy π*(s) that maximizes the total reward for all states ($\forall s \in S$).

The maximization of the score function $J(\theta)$, or the reward, is achieved through gradient ascent. The update of each parameter $\nabla\theta$ is performed by multiplying a learning rate $\alpha \in [0,1]$ by the gradient of the score function $\nabla_\theta J(\theta_t)$. The gradient allows the agent to understand the direction in which it should update the parameters in order to maximize the score function $J(\theta)$, that is:

$$\Delta\theta = \theta_{t+1} - \theta_t = \alpha\nabla_\theta J(\theta_t) \quad (2.1)$$

In an episodic setting ($T \in \mathbb{N}$), the score function is the expected cumulative reward from the initial state s_0 to the terminal state T while following the policy π_θ, where r_t is the reward in t and γ is the discount temporal factor. The expected cumulative reward is represented as:

$$J(\theta) = v_{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \qquad (2.2)$$

To update the parameters, it must differentiate the score function [53], getting this final expression, where $R(\tau)$ is the episodic return:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}\left[\sum_{t=0}^{T} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a|s_t)} R(\tau)\right]$$
$$= E_{\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t) R(\tau)\right] \qquad (2.3)$$

This expression means that the agent updates the parameters θ in the direction of the gradient ∇_θ. This direction points to a place in the parameter dimension that increases the probability of taking a_t in future situation where the agent visits s_t again. The gradient's direction updates the parameters in the same proportion as the return of score $R(\tau)$, becoming actions that retrieve higher returns more likely to happen than actions that retrieve lower returns. The parameter update is also inversely proportional to the probability of taking a_t in s_t, $\pi_\theta(s_t)$. The division by $\pi_\theta(s_t)$ guarantees that actions with lower probability but promising can have the opportunity to increase their probabilities of being taken[5]. The $log\pi_\theta(s_t)$ comes from this simplification:

$$\nabla log\, f(x) = \frac{\nabla f(x)}{f(x)} \qquad (2.4)$$

These policy gradient methods have good convergence, but they also struggle with high variance caused by noise estimations[6] of the policy value. Therefore, a more general form of

[5] https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c

[6] These noisy estimations, in vanilla Policy Gradient methods are typically computed by Monte Carlo returns Gt. Gt accumulates rewards during the whole episode and so, there are a lot of variance, thus it is hard to assign credit to each action taken during the episode. The policy is then poorly evaluated and the convergence takes longer. 9Actor-Critic methods quit Monte-Carlo episodic estimates and adopt Temporal-Difference estimations

reinforcement with baseline was created to improve the policy evaluation [53]. This method subtracts a term G_t to reduce the variance and get a better convergence through the following equation:

$$\Delta\theta = \alpha\nabla\theta log\, \pi_\theta(a_t|s_t)\big(G_t - b(s_t)\big) \qquad (2.5)$$

The baseline $b(s_t)$ helps improving performance and can be any value, however, a better approach is to turn it into a state value function $v(s,w)$ so that the agent can learn a state-value function which improves the learning process by making better evaluations.

Nevertheless, even using a baseline, this value is only estimated for each state transition and, for that reason, every reward from the episode is equally subtracted from this state-value function. This takes to a new branch of Reinforcement Learning, Actor-Critic, where a dedicated evaluator was created. The dedicator evaluator called Critic performs evaluations for cumulative rewards in each state-action transition without the need to wait until the end of the episode. In this way, the agent learns a separated action-value function $Q_w(s,a)$ or a state-value function $V_w(s)$. The Actor $\pi_\theta(a|s)$ is evaluated and helped by the Critic, improving its ability to learn and assign credit for each action taken through better estimations and lower variances.

2.1.2.2 Value-Based

Value-Based is a type of method that tries to estimate and optimize value functions. This method only estimates critics, nonetheless implicit actors are extracted from these value functions, meaning that an implicit policy is also improved through a greedy behavior according to these value functions. Value-learning tries to estimate the expected cumulative reward of following policy π from state *s* onwards:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned} \qquad (2.6)$$

The last two equations express the same thing, but the third one is useful to explain the general concept of a state-value function. The value of being in the state *s* is the weighted sum of

the probability of acting a in s while behaving like π multiplied by the probability of transiting from s to s' and getting reward r after acting with a multiplied by the immediate return plus a discounted value of being in the next state s'.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \quad (2.7)$$

Following the Dynamic Programming framework [53], the states values can be updated according to:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \quad (2.8)$$

The intuition behind this equation is that the agent can select an action based on state-values, which means the agent looks one step ahead and selects the action that takes it to the state with the highest reward. However, in a Model-Free dynamic, the agent does not have access to the model dynamics, which means it will not know the reward if it would take action a.

In Model-Free settings, action-value learning[7] is what supports Value-Based methods. It learns a parameterized action-value function Q_θ, a critic, which makes the action-values more accurate as the time goes by. An actor is trained indirectly[8] since a Value-Based agents acts optimally based on the action-value function Q. If the agent reaches convergence to the optimal action-value function Q^*, the optimal policy for each s will choose the action a that maximizes the expected reward as:

$$\pi^*(s) = a^*(s) = arg \max_{a \in \mathbb{A}} Q^*(s, a) \quad \forall s \in S \quad (2.9)$$

[7] When no model is available, if the goal is to act based on value functions, one needs to act with respect to action-value functions.

[8] Since Q-value estimates are getting better, if the agent acts greedily according to those estimates, then it is an optimal policy. Hence, there is an implicit actor, that, without being parameterized, is indirectly improving.

In this equation $Q^*(s,a)$ stands for the expected cumulative reward of taking action *a* in *s* and acting based on the optimal policy:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\mathrm{G}_t | S_t = s_t, A_t = a\right] \qquad (2.10)$$

In conclusion, there is a final connection that one can make between action-value and state-value function[9] which is the optimal state-value function is the maximum expected cumulative reward for any given state *s* and following the optimal policy, such as:

$$V^*(s) = \max_{\pi} \mathbb{E}\left[\mathrm{G}_t | S_t = s\right] \qquad (2.11)$$

Since Q^* is the optimal policy for an action-value learning, then one can consider that the best state-value is equal to the best action-value if the best action is selected. Therefore, the optimal Q^* can be associated to the optimal V^* [53], following the equation:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] \\ V^*(s) &= \max_{a} Q^*(s,a) \end{aligned} \qquad (2.12)$$

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{\pi} \mathbb{E}[\mathrm{G}_{t+1} | s_{t+1}] | S_t = s, A_t = a] \\ Q^*(s,a) &= \mathbb{E}[R_{t+1} + \gamma V^*(s_{t+1}) | S_t = s, A_t = a] \end{aligned} \qquad (2.13)$$

Once stated the above, one of the classic algorithms is the Q-learning [53] which is one of the most known and successful algorithms in the Value-Based domain. Its implementation is easy, and it is able to solve several problems where the agent tries to find the best actions to solve a relatively small and deterministic environment. This method is a tabular method because, through the agent's experience, it stores in a table a specific Q-value for each state-action combination

[9] In other words, a connection between optimal policies derived by optimal action-value functions and optimal state-value functions

$Q(s, a)$. Also, it is an off-policy algorithm meaning that it can learn a different policy compared with the behavior policy being executed. This different policy could be a greedy behavior, and for that reason, it will converge in its direction regardless of the policy obtained through the chosen actions. If a proper exploration and enough visits for all state-action pairs were performed (being the value of those pairs updated through the experience in the environment), then one was able to correct the convergence to the optimal policy Q^*. Q-value are updated, until the terminal state or convergence, according to the equation[10]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a_{t+1}} Q^-(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (2.14)$$

Moving onto deep RL methods, Deep Q-Networks [33] combines Q-learning with deep neural networks and it has been successful in many tasks. This method tries to overcome the disadvantage of Q-learning of struggling[11] with high-dimensional and complex problems involving extensive state-spaces. DQN[12] is an end-to-end algorithm intensively applied to dynamically complex domains using only raw high-dimensional data as input. It is used to achieve near-human performance in the context of Atari Games, but it has shown an incredible performance on other tasks such as Multi-Agent RL[13]. DQN has an architecture of a deep convolution neural network that receives an input image and through 2D convolution layers extracts relevant features[14] to perceive the environment. Then, those features are fed to a dense layer that predicts the Q-values for each action available for the agent[15].

[10] The "-" superscript on top of the Q^- was used only to reinforce the idea of off-policy learning: the target prediction
for the next state, could be taken from a different policy other than Q.
[11] It is not feasible in terms of time and memory consumption to construct a table for high-dimensional state-spaces. It is when Q-learning is applied to more complex problems, that one chooses, instead, to approximate/estimate a Q-function using powerful function approximation techniques like neural networks.
[12] DQN could be easily considered as the most well-known deep RL algorithm.
[13] https://paperswithcode.com/method/dqn
[14] Deep Convolutional Neural Networks perform feature extraction autonomously. Hence, there is no need to perform manual feature extraction.
[15] The DQN neural network is typically fed with the agent's state as input, outputting the corresponding Q-values for each action. It could also be fed with the agent's state and performed action as input, outputting a single Q-value for the corresponding state-action pair in the input.

However, DQN also has its own downside called Maximization Bias [51] which means an overestimation of action-values caused by the max operator in the target estimate y^{DQN} produced by the target network $Q(s, a; \theta^-)$, as demonstrated below:

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \qquad (2.15)$$

To overcome the downside a Double DQN [18] was proposed and it changes the target estimate to y^{DDQN}, where the target network $Q(s, a; \theta^-)$ evaluates the action and the policy network $Q(s, a; \theta)$ selects the action such as:

$$y_i^{DoubleDQN} = r + \gamma Q(s', arg\max_{a'} Q(s', a'; \theta_i)\ ; \theta^-) \qquad (2.16)$$

With this implementation one can achieve faster convergence with steadier target estimations (tackling the Maximization Bias) compared with the performance of DQN.

Another architecture was proposed to improve the DQN principles, Dueling Networks [55]. The difference lays on the fully connected part, which is divided in two independent fully connected layers, first one to estimate a state-value V(s), and the second one to estimate the action-advantages $A(s, a)$. Before feeding the output layer, the output from both layers are combined through an aggregation module. In that way, the Q-value can be explained as the sum of the value of being in state *s,* with the advantage of taking action *a* in the same state:

$$Q(s, a) = V(s) + A(s, a) \qquad (2.17)$$

This would be correct if it did not have identifiability issues[16]. This issue was solved with two approaches (equation 2.20 and 2.21). If a max operator is added in $A(s, a';\ \theta, \alpha, \beta)$ as shown in equation 2.20, then the optimal action will be defined as:

$$a^*(s) = arg\max_{a' \in \mathbb{A}} Q(s, a'; \theta, \alpha, \beta) = arg\max_{a' \in \mathbb{A}} A(s, a'; \theta, \alpha) \qquad (2.18)$$

[16] If one sums both values to get Q, then it cannot recover V and A, uniquely. Empirically, this resulted in poor practical performance. If there is no identification for V and A, the backpropagation process will not differentiate both estimates.

Now, the Q-value for the optimal action will be equal to the state-value (equation 2.19). In this approach $V(s, a, \theta, \beta)$ identifies the optimal action and $A(s, a, \theta, \alpha)$ the remaining actions.

$$Q(s, a^*; \theta, \alpha, \beta) = V(s; \theta, \beta) \quad (2.19)$$

This second approach applies an average over the advantage estimates, removing the identifying semantics for V and A (since they are subtracted by a scalar, and they are no longer really V and A), which makes it steadier on optimization performance[17].

The separation of the state-value function through the advantage function, enabled DDQN to estimate the state-value and to recognize how valuable it is to perform actions in a given state. There are situations in which it is not worth acting and this network will know that. To finalize, $V(s; \theta, \beta)$ is the state-value stream function, $A(s, a; \theta, \alpha)$ is the advantage-value stream function and $Q(s, a; \theta, \alpha, \beta)$ is the complete parameterized DDWQ Q-function, being α and β the independent parameter vectors from the advantage-values function and the state-value function, respectively and θ is the parameter vector from the convolutional layer. Finally, the output Q-values from DDQN can be expressed as:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \max_{a' \in |A|} A(s, a'; \theta, \alpha)) \quad (2.20)$$

And,

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha)) \quad (2.21)$$

DDQN overcomes the performance of DQN and Double DQN, in the Atari Games domain, by changing its architecture that improves Q-value estimations.

[17] In practice, the advantages change as fast as the mean, hence they don't need to follow the optimal action's Advantage.

2.1.1.3 Actor-Critic

Actor-Critic methods can learn both a value and a policy function. The actor chooses actions and the critic criticizes the actor's options, improving the evaluations performed by solitary actors in Policy Based methods. It is considered a hybrid method because it combines Policy Based and Value Based methods, where it learns a critic in the form of a parameterized state-value function $V(s, w)$ or an action-value function $Q(s, a, w)$ with parameters w and it also learns an actor in the form of a parameterized policy function $\pi(a|s, \theta)$ with parameters θ.

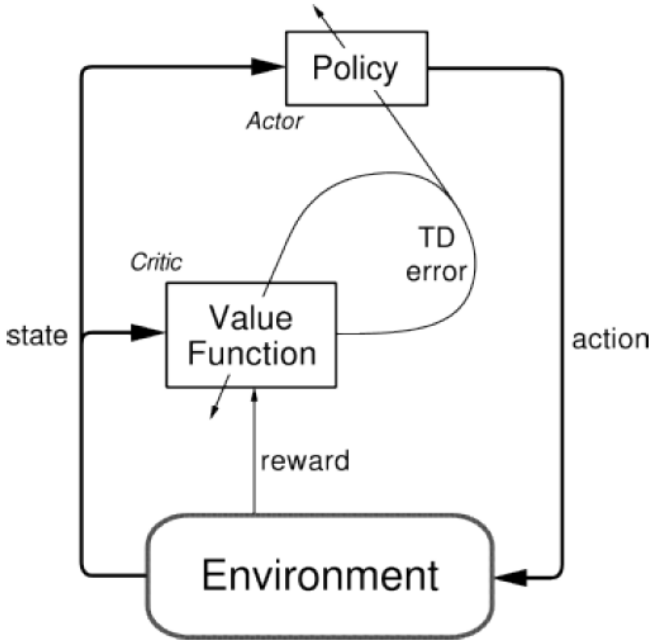

Figure 8- A typical Actor-critic architecture.

Source: http://incompleteideas.net/book/first/ebook/node66.html

This type of method tries to maximize the performance of the policy by using a learnt critical estimate to evaluate the transition. Usually, this critic's estimate[18] is in the form of a Q function $Q_w(s, a)$, an Advantage function $A_w(s, a)$ or a TD-error δ. The following equation shows an Actor-Critic actor's parameter vector update, using an advantage critic evaluation being α^θ the learning rate:

[18] https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf - on the last slide there are all the possible critic estimates.

$$\Delta\theta = \theta_{t+1} - \theta_t = \alpha^{\theta} \nabla_{\theta} J(\theta) \quad (2.22)$$

The intuition of using advantages is to update the gradients in the direction of the advantage function. This direction shows the advantage of taking a_t in s_t compared to the value of being in s_t. If it is positive than the probability of taking a_t in the future increases, otherwise, that probability is decreased.

This update is achieved using gradient ascent in order to maximize the actor's objective function $J(\theta)$, using this equation:

$$\nabla_{\theta} J(\theta) = \mathbb{E}[\nabla_{\theta} log\, \pi_{\theta}\, (a_t|s_t) A^{w}(s_t, a_t)] \quad (2.23)$$

The advantage function estimate[19], can be derived as:

$$\begin{gathered} Q(s_t, a_t) \approx r_{t+1} + \gamma V(s_{t+1}) \\ A(s_t, a_t) = Q_{\emptyset}(s_t, a_t) - V_w(S_t) \\ A(s_t, a_T) \approx \mathbb{E}_{\pi\theta}[r_{t+1} + \gamma V_w(s_{t+1}|s_t, a_t)] - V_w(s_t) \\ A^{w}(s_t, a_t) = r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) \end{gathered} \quad (2.24)$$

Moreover, advantage AC method tries to minimize the error of the critic's predictions, therefore, an update of the critic's parameters using an approximated TD-error[20] is performed, as follows:

$$\begin{gathered} \mathbb{E}_{\pi\theta}[\delta_t|s_t, a_t] = A^{\pi\theta}(s_t, a_t) \approx r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) \\ \delta_w = r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) = A^{w}(s_t, a_t) \end{gathered} \quad (2.25)$$

Since the goal is to minimize the critic's objective function $J(w)$, gradient descent is used to update the parameters,

[19] It may seem that one needs two critics with parameters f and w, but if one learns a critic state-value function, one only needs one critic.

[20] While following policy π_{θ} , the TD-error is approximately equal to the advantage function if one performs a_t in s_t . Proof in the David Silver's teaching slides at: https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf

$$\nabla_w J(w) \approx \nabla_w V_w(s_t)\delta_w \qquad (2.26)$$

The update of the critic's parameter vector using an independent learning rate α^w is given by:

$$\Delta w = w_{t+1} - w_t = -\alpha^w \nabla_w J(w) \qquad (2.27)$$

Nowadays, there are several AC algorithms with great results in numerous applications such as Atari games, simulated physics tasks and robotics tasks. Some of the most known and successfully AC algorithms are described below:

DDPG [26] is an off policy[21] Deep RL AC algorithm, inspired by DQN [32] intuition, but with the advantage of working in a continuous action domain. It has been used to solve complex tasks like legged locomotion, dexterous manipulation, self-driving cars and Atari games keeping the same network architecture, hyperparameters and learning algorithm. It can work with high-dimensional input raw data by learning the end-to-end policy. This algorithm overcomes planning algorithms with model dynamic knowledge and DQN by achieving the same performance with a significant reduction in the training time.

PPO [44] is also a Deep RL AC algorithm that has stability and reliability of trust-region methods [45] but with the advantage of having a simpler implementation, better performance, and better sample efficiency. At the moment, this is the state-of-the-art algorithm in continuous controlling tasks and it will be described more deeply in Chapter 3.

SAC [17] is another successful Deep RL AC algorithm and the first off-policy actor-critic method in the maximum entropy reinforcement learning framework. The standard RL tries to maximize the expected total reward, while maximum entropy RL has a more general goal, besides of trying to maximize the expected sum of rewards, it also tries to maximize its policy's entropy. The intuition is to make the agent take as many random actions as possible but still collect high rewards. This method has the advantage of being more robust in simulated environments and in real-world environments, because since they can tolerate highly random behavior during training, they will handle better random perturbation during the test phase[22]. In this way, SAC has a higher

[21] Off-policy algorithms have the ability of learning from past experiences; thus they spend less samples to learn because they reuse them. Hence, Off-policy Actor-Critic algorithms tend to be naturally more sample efficient.
[22] https://ai.googleblog.com/2019/01/soft-actor-critic-deep-reinforcement.html

sample efficiency than other on-policy AC methods because it learns off-policy, and it also overcomes the convergence weaknesses from other off-policy AC methods [16].

2.1.2 Model-Based

Model-Based is a set of RL algorithms where the agent has the ability to predict what is going to happen in the environment if it takes certain action. This happens because the agent has access to the environment's model, and it uses it to learn an optimal policy. MB algorithms are, usually, used in planning tasks.

The fact the agent has access to the model, makes these methods more sample-efficiency[23] because they can learn not only with the interaction with the environment but also predicting steps ahead, making the convergence time much lower. The following figure shows a comparison between Model-Free and Model-Based in terms of sample-efficiency:

Figure 9 – Source: https://jonathan-hui.medium.com/rl-model-based-reinforcement-learning-3c2b6f0aa323

One main difference between Model-Free and Model-Based is the way they used real experience. Whereas Model-Free methods use real experience to improve value functions and policy functions, Model-Based methods use it for model-learning, by making the inner model more similar to the real model. Basically, they do not improve their value or policy function in the interaction with the environment but instead interacting with simulations performed from its inner learned model, as follow:

[23] A concise article about Model-Based RL: https://medium.com/@jonathan_hui/rl-model-based-reinforcementlearning-3c2b6f0aa323

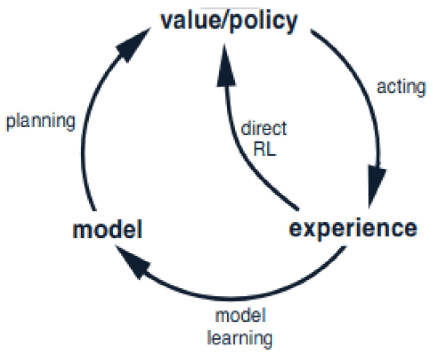

Figure 10 - Model Based flow

Source: https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html

There are two categories of Model-Based agents:

1. Given a model, use planning to learn a global value and/or policy function.
2. Learn a model, focusing on learning a model and a global value-function and policy function (more complete class of MB).

The following subsection will give more details about the state-of-the-art in Model-Based RL, but with less detail since the focus for this dissertation is Model-Free methods.

2.1.2.1 Given a model

Model-Based RL algorithms can learn policies or value functions through tree-search techniques [52] to plan and learn the best policy in a given environment, therefore they can skip the intermediate step of learning the environment's model by interacting with it. This brings the advantage of sample efficiency, since by only having access to the rules of the game[24], they can achieve and surpass human performance. One example is the AlphaZero [48] which was able to learn how to play several board games like go, chess and shogi within only 24 hours and still be able to defeat world-champion humans in each game.

2.1.2.2 Learn a model

This kind of Model-Based RL class consists in learning the dynamics of an unknown environment using function approximation techniques[25]. A model of the environment can be

[24] What differs this subsection from the following subsection ***Learn a model*** is the willing that an RL agent must learn the dynamics of the environment, to build an inner model and learning policies and value functions inside of it. In this subsection, the model is already estimated and given to the agent.

[25] These techniques could be parametric or non-parametric. In the context of DRL the common chosen approach is parametric using neural networks. It is not considering hugely memory demanding techniques like tabular

perceived as a representation of the most interesting aspects of the dynamics of the real environment. The state-of-the-art methods in Model-Based that learn a model are:

Dyna-Q [50] is an algorithm that learns how to act and, at the same time, learns the model of the environment, meaning that it is able to combine model-learning with planning and value/policy function learning. One specific type of Dyna-Q is the tabular form that embeds Q-learning with model learning in a deterministic environment. The way Dyna-Q works is by acting on the environment and collecting experiences and updating its Q-values based on the expected reward r_{t+1} and the observed next state s_{t+1} influenced by the action taken a_t and the current state s_t. Thus, each dynamic transition is saved in a tabular model such as $Model(S = s_t, A = a_t) \leftarrow (r_{t+1}, s_{t+1})$. The number of iterations in its inner model to simulate the outcome (next state and reward) of each possible action is controlled by the n hyperparameter. Higher values of n lead to more simulations, better planning, and better decisions but at the same time, it will increase the time needed to take an action. Nevertheless, this method showed to have increased the learning speed and sample efficiency when compared with a standard Q-learning model without planning ahead.

Another algorithm of Model-Based with model learning is World Models (WM) [15] that tries to create an agent that can learn a model and a policy inside of it. The intuition behind this model is the human cognitive system composed of three components. The visual component that is responsible for capturing visual information and for encoding it into a compressed representation, the memory component that can predict future observations based on historical compressed information, and lastly, the controller component that integrates the information retrieved by the other two components and decide what is the best action to take. This method has proven a good and robust performance in virtual environments of tasks like car racing or the VizDoom game.

Finally, there is the MuZero [42] model that applies end-to-end learning[26] and achieves nearly the same, or even better performance than the AlphaZero [48]. MuZero unlike AlphaZero does not have access to the rules of the game and it learns the dynamics of the game from scratch. This deep RL model combines planning, acting and training inside of the learned model and it uses

methods.

[26] End-to-end is an approach where high-dimensional raw inputs are directly learned and mapped to outputs by a single model in a single pipeline, without consulting any third-party. It is end-to-end in the sense that, from one end to the other, autonomously and directly learns to map raw input data to desired outputs.

the Monte Carlo Tree Search (MCTS) [53] technique to simulate what is the next rewards r_{t+n} and environment states s_{t+n} (planning) over the several possible actions a_t and the current state s_t (acting). The learning process consists in storing the previous experiences in a memory buffer (training).

2.2 Deep Learning and Computer Vision

This section will introduce deep learning and artificial neural networks and discuss what has been developed for regression problems using CNNs, transfer learning and semantic segmentation in the Computer Vision field.

2.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational networks of nodes and connections resembling that of biological neural networks [14]. Formally, artificial neural networks in machine learning refer to directed-acyclic graph with weights along each edge of the graph.

When a datapoint x_i is fed to an ANN, it will propagate thought the network, and it will end up with some prediction. The goal in machine learning is to find the best configuration of network weights, often denoted W, with respect to some objective. This objective may, for example, be to minimize the error of a regression predictor to ground truth values, that is, minimize $L = \sum_i (\hat{y}_i - y_i)^2$ over all labeled data, $x_i \subset$ X and $y_i \subset$ Y.

Gradient Decent: Gradient decent is the most common optimization technique used with ANNs. It works by calculating the gradient of the loss function, L, with respect to the weights of the network W. Once it has the gradients, it nudges the weight variables in the direction of greatest decent as follows:

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}} \qquad (2.28)$$

Where α is a hyperparameter that determines how much we should nudge our weights in the direction of steepest decent per step, and $\frac{\partial L}{\partial w_{ij}}$ represents the direction of steepest ascent along the loss function's surface with respect to a particular weight $w_{ij} \in$ W.

Another important part to have a good performance with ANN is the activation function present in each node [35]. The choice of a proper activation function will help solving a common problem that most of the learning-based systems have which is the way the gradient flows within the network because some of the gradients will be sharp in specific directions and zero in another directions, which can lead to vanishing and exploding gradients [37, 57].

There are several families of activation functions and in [35], the authors mention nine families such as Sigmoid, Tanh, Softmax, Softsign, ReLU, Softplus, ELU, Maxout, Swish and ELiSH fucntion that have some tweaked functions within each family and the default one which is Linear.

For this dissertation, four of them were used and they will be the ones with a detailed explanation:

1. Hyperbolic Tangent Function (Tanh) is a smoother [25] zero-centered function who produces values between -1 and 1 and its formula is the following:

$$f(x) = \left(\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}\right) \qquad (2.29)$$

2. Softmax is used to produce a probability distribution from an array of real numbers therefore the output is a range of values between 0 and 1 with the sum of the probabilities been equal to 1. The Softmax [21] function is:

$$f(x) = \left(\frac{exp\ (x_i)}{\sum_j exp\ (x_j)}\right) \qquad (2.30)$$

3. Rectified Linear Unit (ReLU) [2] rectifies the values of inputs less than zero by forcing them to zero and eliminating the vanishing gradient problem observed with other types of activation functions. It is usually applied in the hidden layers and

then, a different activation function is used in the output layer like Softmax or Tanh already mentioned. Its function can be described as:

$$f(x) = max(0, x) = \begin{cases} x_i, & if\ x_i \geq 0 \\ 0, & if\ x_i < 0 \end{cases} \quad (2.31)$$

4. Linear function [21] is a linear mapping of an input to an output as performed in the hidden layers before the final prediction of class score for each label is given by the affine transformation in most cases. The transformation of the input can be given by:

$$f(x) = w^T x + b \quad (2.32)$$

2.2.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN), in the CV context, is a Deep Learning (DL) algorithm which takes as input an image and processes the image internally in order to find patterns that are helpful for the predictive task [36]. Contrary to the traditional ML algorithms where there is a need to heavily pre-process the data in order to create relevant features, the CNN is able to perform the feature engineering process internally through convolution operations.

Since an image is a matrix of pixels, one could flatten it and feed it to an Artificial Neural Network (ANN) without the need of using CNNs. The problem with this approach is that spatial and temporal dependencies would be lost while a CNN can preserve and understand them through the application of relevant kernels/filters.

The way a CNN works is the following:

1. Receives an image as input.
2. Performs convolution operations between the image and the kernel/filter which is a matrix NxN defined by the user.
    a. The user can also define as much filters as desired. In one hand, a large number of filters increases model complexity, but, on the other hand, few numbers of filters reduce the predictive power of a CNN.
    b. Each filter will be responsible for extracting a feature from the image.

c. The first convolution layers will extract high-level features such as edges, while the following convolution layers will extract more low-level features such as gradient orientation. Therefore, a CNN should have more than one convolution layer.

d. This operation produces features which have a reduced dimensionality when compared to the input. However, Padding can be used to avoid this.

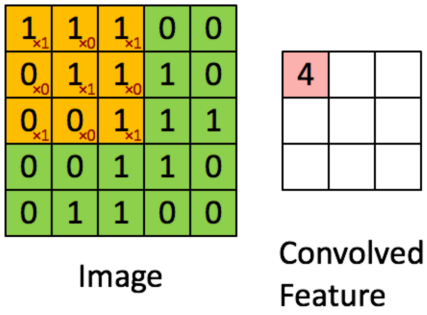

Figure 11 - Convolution Operation using a 3x3 kernel on an input image 5x5

Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

3. After a convolution layer, usually, a CNN has a Pooling layer which reduces the spatial size of the convolved feature by returning the maximum or the average value from the portion of the image converted by the filter (Max or Average Pooling). This is useful to reduce the computational power required to process the data and extracting dominant features which are rotational and positional invariant.

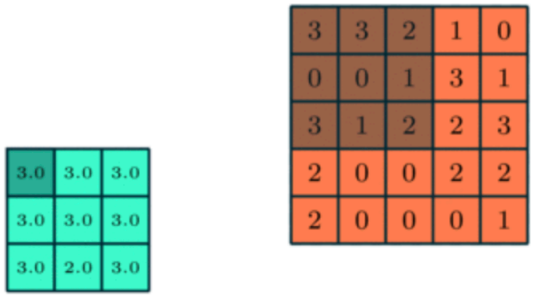

Figure 12 - Pooling Operation using a filter 3x3

Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

4. Finally, with all features created, a flatten operation is applied to be able to feed an ANN for classification or regression purposes.

    a. What differs from classification or regression will be the loss function (for example, cross entropy or mean absolute error, respectively) and the activation function in the output layer (for example, sigmoid/softmax or linear, respectively).

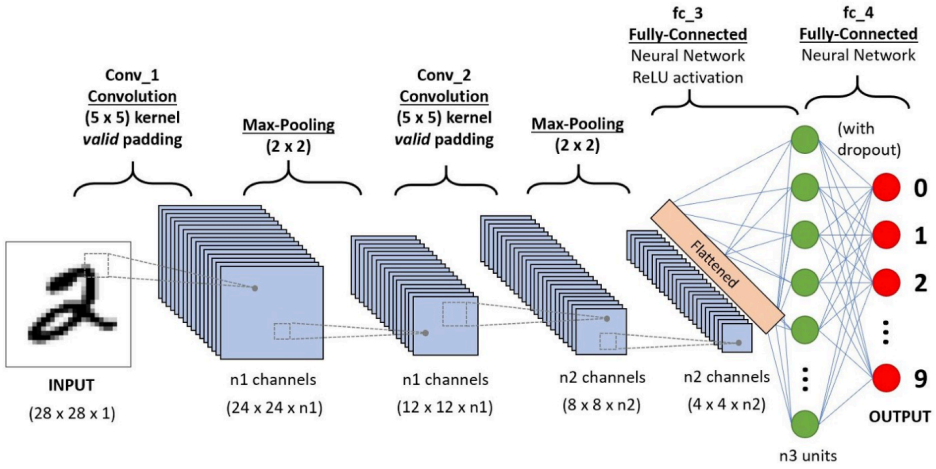

Figure 13 – Example of a CNN to classify the hand written digit in an image

Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

2.2.3 Transfer Learning

Transfer Learning is a technique used in ML where a model is trained to perform one task and then, it is re-used to perform a second task that is similar. The goal is to reuse what has been learned in one task to improve and, if possible, accelerate the weight optimization process in the other task. The similarity between tasks will determine the amount of time needed to re-adapt the model from one task to the other [5].

Several python libraries such as Tensorflow or Pytorch have pre-trained CNNs in a dataset called *'ImageNet'* with more than 1 million images to be classified according to the different 1000 classes presented in the dataset [40].

Using TL, these CNNs start with the learned features on the *'ImageNet'* dataset and adjust these features according to the new task, speeding up the training of a CNN. Since the first layers learn high-level features which are similar among different applications, those features do not need to be learned again. Therefore, those layers are frozen and only the last ones need to learn new features during training allowing the process to be less expensive and faster [10].

Apart from that, to correctly apply TL, one may need to change the architecture of the CNN to be able to perform a different task. One example is to turn these CNNs trained to classify into CNNs able to perform regression tasks. One simple way of doing this is removing the last layer which has 1000 nodes with a softmax activation function to a 1 node layer with a linear activation function.

2.2.4 Image Segmentation

Image Segmentation consists in grouping parts of images together that belong to the same object class. This type of algorithm has been used in different problems and industries such as medicine for tumor detection, automobility for lane and road signs detection and many others.

This method can be seen as a classification task where the algorithm must classify each pixel according to the object in it. Like every ML algorithm there are advantages such as easily get the objects in an image and disadvantages such neighboring pixels of the same class might belong to different object instances and regions that are not connected may belong to the same object image. For example, one person in front of a car visually divides the car into two parts [54].

The most recent architectures in image segmentation consist of an encoder and a decoder. The encoder is a CNN which extracts features from the image through filters, as explained previously. The decoder is responsible for generating the final output which is usually a segmentation mask containing the outline of the object [3]. One example, of this type of architecture is U-Net [38].

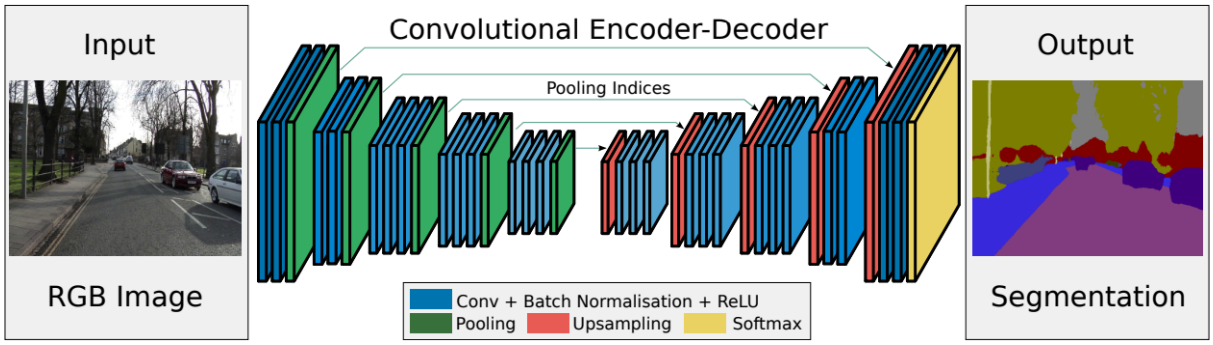

Figure 14 - Encoder-Decoder architecture for image segmentation

Source: [3]

2.3 Safety

This section aims to explore what has been done in order to increase the safety and trustiness associated to Deep RL systems.

Recently, Deep RL systems with continuous action spaces have been extensively explored in context of real-world applications such as autonomous driving [41]. Contrary to the most usual application of RL systems like games [32], self-driving cars developed to act in real situations require a set of safety constraints to be fulfilled such as avoiding collisions by limiting velocity and turning angles.

The main problem is that in most of the cases the dynamic of the environment is unknown, and for that reason, it is not possible to determine which actions are safe ahead of time. The traditional approach is defining a reward function that ensures some safety, for example rewarding negatively when a collision happened during training. However, the agent will not face all possible collision states during training, therefore it will not be able to avoid collision every time a different state is presented to it during the testing phase [46].

In recent years, this topic has been researched by the authors in [28] where they propose a framework called Intrinsic Fear that increases safety by training a neural network to identify unsafe states, which is then used to shape the reward function. However, this approach faces the same problem of not visiting all the states needed to gather enough information to avoid them.

In a different line of research, the authors in [7] add a safety layer to the agent's policy that projects unsafe actions onto safe domains using a constraint function. The authors in [46] use the

same line of reasoning for multi-agent systems where they create a safety layer that combines the actions from all agents to ensure coordination between agents and to minimize unsafe actions.

This dissertation will follow the last line of research described by creating constrains to correct the agent's decision every time it can lead to dangerous situations such as limiting the velocity if it does not comply with the safety distance or avoiding abnormal and sudden turns.

2.4 Explain Ability

This section shows what are SHAP values and how it can be used to explain the agent's decisions.

In the past, Data Scientists struggled to decide between accuracy and explain ability, however with an increase of data collected, the need to use complex models to solve complex problems started to be a reality and opting for white box models was no longer acceptable.

SHAP values appeared to surpass the difficulty of explaining the output of a complex model such as deep neural networks or gradient boosting algorithms through reverse-engineering the output of any predictive model.

As mentioned before, SHAP values are based on Shapley values which is a concept that comes from game theory where the game is reproducing the outcome of the model for one single observation (x_0) and the players are the features used in the model. The Shapley quantifies the contribution of each feature for the prediction made by the model considering that the outcome of each possible combination of features should be considered to determine the importance of a single feature. Therefore, SHAP needs to train a distinct predictive model for each combination of features where the hyperparameters and the training set are the same for each model [20].

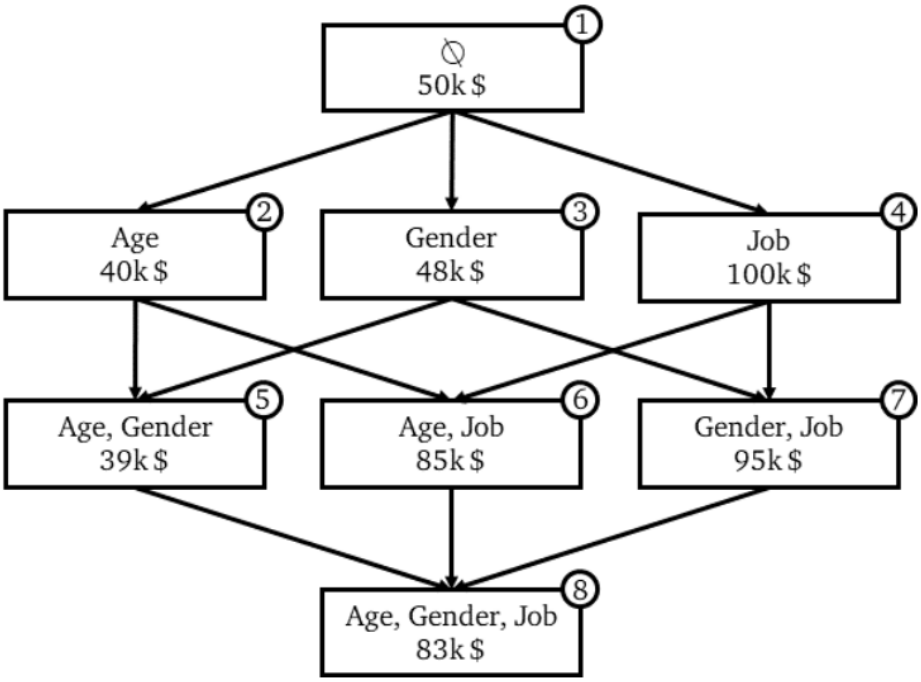

Figure 15 – This image represents the power set of a dataset where the goal is to predict salary based on 3 features (gender, job and age) which has 8 different predictive models with a set of different features used in each model so that SHAP can estimate the impact of each feature in the prediction of one single observation.
Source: https://towardsdatascience.com/shap-explained-the-way-i-wish-someone-explained-it-to-me-ab81cc69ef30

From Figure 15, one can see that two nodes connected by an edge only have one different feature, therefore the difference on the predicted values between the model created by those two nodes can be considered as the effect of that additional feature, which is the marginal contribution of that feature.

Using the example above and looking to node 1, this model with no features will predict the average '*salary'* (50k $). In node 2, where the model has '*age'* as feature, the prediction is only 40k $, which means that the marginal contribution of *'age'* is -10k $. To obtain the overall effect of *'age'*, it is necessary to consider the marginal contribution (MC) of '*age'* in all models where it is used as feature that is then combined through a weighted average (Equation 2.25).

$$SHAP_{Age}(x_0) = w1 * MC_{Age,\{Age\}}(x_0) + \quad (2.25)$$
$$w2 * MC_{Age,\{Age,Gender\}}(x_0) +$$
$$w3 * MC_{Age,\{Age,Job\}}(x_0) +$$
$$w4 * MC_{Age,\{Age,Gender,Job\}}(x_0)$$

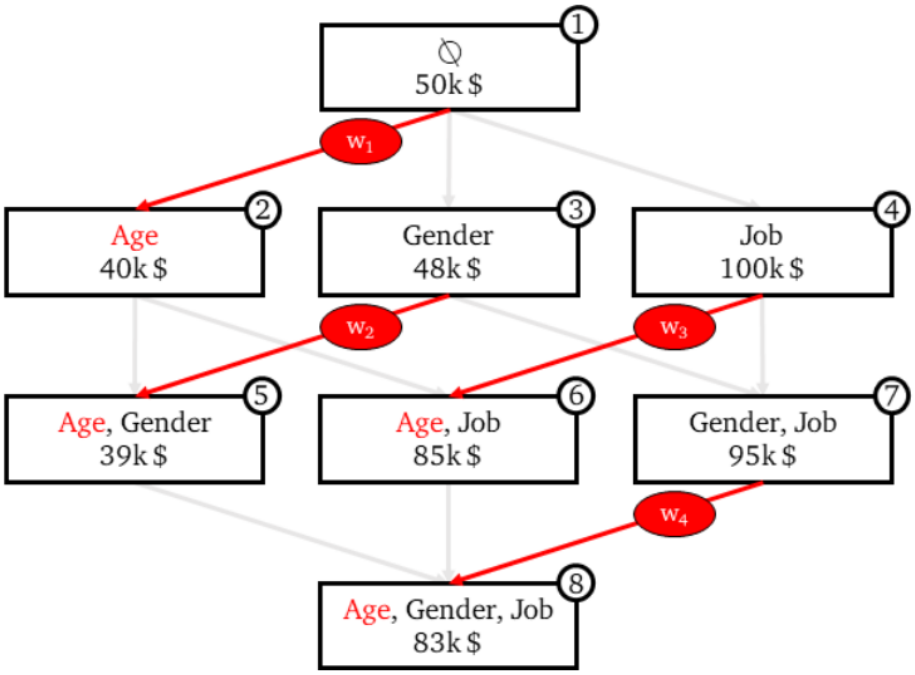

Figure 16 – Weights associated to each model where *'age'* is presented.

Source: https://towardsdatascience.com/shap-explained-the-way-i-wish-someone-explained-it-to-me-ab81cc69ef30

To determine the weights of each marginal contribution, it is assumed that the sum of all the weights in the same row should be equal to the sum of all weights on any other row (Equation 2.26) and all the weights in the same row should be equal (Equation 2.27), which is defined as:

$$w1 = w2 + w3 = w4 \qquad (2.26)$$

$$w2 = w3 \qquad (2.28)$$

With this, and bearing in mind that the weights should sum to 1, the weight value is 1 divided by the number of edges in that row, for example:

- $w1 = \frac{1}{3}$
- $w2 = w3 = \frac{1}{6}$
- $w4 = \frac{1}{3}$

This example leads to the general equation reported in [31], where F is the predictive model:

$$SHAP_{Feature}(x) = \sum_{set:\, feature \in set} [|set| * \binom{F}{|set|}]^{-1} [Predict_{set}(x) - Predict_{set \setminus feature}(x)] \quad (2.29)$$

The values resulted from this equation are easily interpreted through the chart on Figure 17, where the red color means a positive impact for target prediction and the blue color means the contrary. In other words, and giving an example, the *'RAD'* feature decreases the predicted value while '*LSTAT'* feature increases the predicted value.

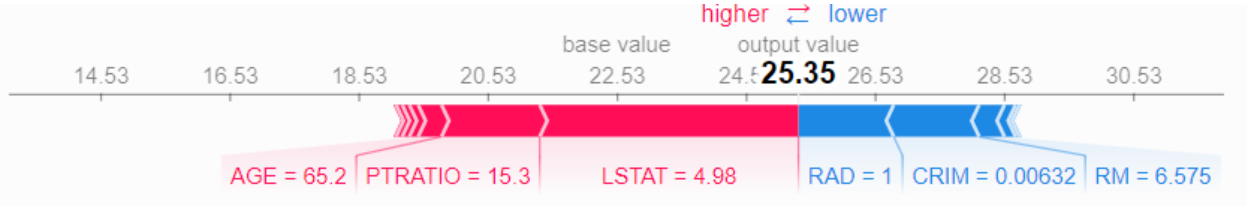

Figure 17 – Analysis of the impact of each feature in the predicted value of a particular observation. Source:https://shap.readthedocs.io/en/stable/example_notebooks/tabular_examples/tree_based_models/Catboost%20tutorial.html

2.5 Computer Vision and Reinforcement Learning in Self-Driving Cars and Platooning vehicles

This section presents what has been implemented in the self-driving car domain using reinforcement learning.

In recent years, various methods of self-driving cars have been developed because of the potential positive impact in sustainability through the possibility of car sharing and platooning approaches that can yield to a more efficient way of using vehicles and roads.

One example is the approach taken in [12]. In this paper, they combined Markov Decision Process with a proximal policy optimization (PPO) that finds an optimal policy representing the probability density of the agent's action given a certain state. The agent learns how to map its estimated state to acceleration and steering commands given the objective of reaching the final

state which is a parking slot in a public road considering obstacles. To get information about the vehicle surroundings, laser scanners are used. The reward function was divided into two, one for the driving process and the other one for the parking process. The first one is rewarding to quickly reach the desired speed, whereas the stopper needs to approach the parking slot slowly and stop in the end. The results of the simulation allowed to successfully implement this process in the real world with a real car.

Both Neural Network architectures (policy and value function) are described in *Figure 18*.

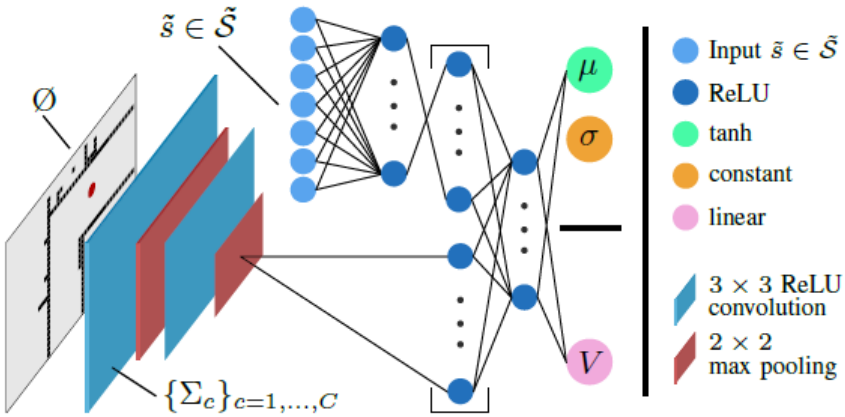

Figure 18 - The state is processed by two dense layers of each 200 ReLu activations, while the perception map is based on two convolutions with max polling in between, the first layer has 30 nodes and the second has 1 node. Then they flatten and process the data in a dense layer of 200 ReLU activations. Finally, they concatenate the outcome of both inputs and pass through a last 200 ReLu layer.

Another example is in the AWS DeepRacer simulator [4], they train a RL self-driving car using a monocular camera and the PPO RL algorithm. The policy decides what action to take based on the input image and the value network estimated the expected cumulative reward. Initially, the agent starts to take random actions to interact with the environment and to collect data and update the policy and value networks according to the algorithm's loss function. The policy tries to maximize the actions that give higher rewards on average and applies a higher weight to newer versions of the policy being updated. The loss function uses the mean squared error between the predicted and actual value and the predicted value is estimated by three CNN and two fully connected layers for both networks (actor and critic). The reward function consists in keeping the car in the middle of the track by identifying the edges of the track and then calculating the distance between them.

Moving to examples combining self-driving cars and platooning vehicles, we have the study from [58] which proposes a deep reinforcement learning methodology for obstacle avoidance and formation control using only a camera as sensor. They start by using a ResNet (CNN model) for localization perception. The ResNet is trained in a data set with images from the real-world environment where the agent will act in order to be able to predict the 2D position of the agent.

Then they propose a new actor-critic algorithm called Momentum Policy Gradient (MPG) that is better than TD3 [13] by reducing the problem of under/overestimation. They also mention that this new algorithm is efficient at solving leader-following problems with irregular leader trajectories and, with a slight change in the reward function, the algorithm is able to solve the collision avoidance and formation control problems.

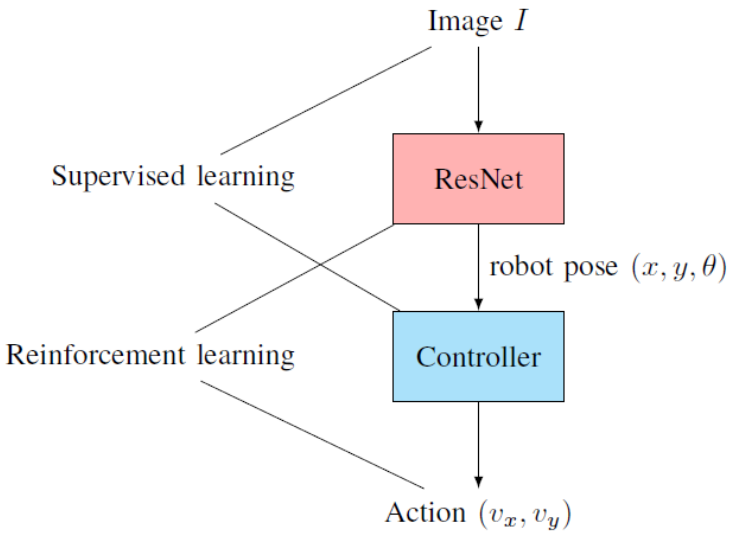

Figure 19 - Model architecture

[29] proposes a distributed reinforcement learning method based on DQN and a consensus algorithm to handle the multi-vehicle platoon problem, where there are a leader and the followers. This process can be divided into two processes, the local training, and the global consensus.

On one hand, the local training consists in the individual DQN of each vehicle which is the way each vehicle learns how to keep the same space between the front and back car. Here, the reward function takes into consideration the distance between the car and the midpoint of the front and back car, the gap between the current velocity and the desired velocity, and the accelerated

velocity. The agent knows the position of the front and back car through communication about their location among the agents.

On the other hand, the global consensus consists in updating the DQN of all vehicles to converge the vehicles to each other and ensure the platooning process.

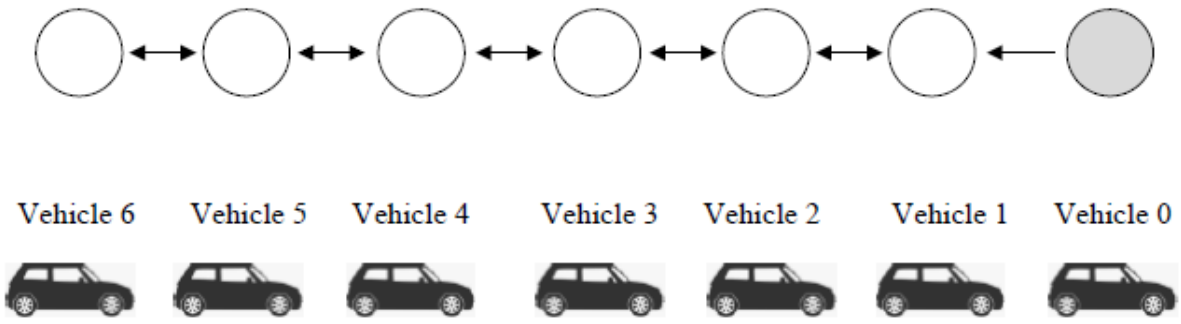

Figure 20 - Communication process among agents

The authors in [59] proposed an IoT framework, where the information captured by camera, such as road edges, traffic lights and zebras lines are highlighted with computer vision. The distance, direction and speed of obstacles are provided by sensors. And, finally, the location of each vehicle is communicated among vehicles based on self-positioning.

In this study, they used VGG16-Places365 to process the information provided by the camera which basically identifies transit signals including traffic lights and lanes.

Also, instead of using RL, the authors used MPC (Model Predictive Control) which is an optimization algorithm that takes a vehicle's motion model to plan out a path that makes more sense given a set of constraints like the limits of the vehicle's motion and a combination of costs that define how they want the vehicle to move. Based on that, it predicts the trajectory that the vehicle should take[27].

The authors concluded that the platooning-based information-sharing reduced the risk of crashing when invisible moving obstacles appear, making it possible for the self-driving vehicle to predict the trajectory of those obstacles and avoid collision.

Finally, in [30] the author proposes a self-driving car using PPO RL algorithm, using only the camera as sensor where Variational Auto-Encoder receives the information retrieved to segment the road. In this study, CARLA simulator was used and one of the goals is to create an

[27] https://medium.com/intro-to-artificial-intelligence/model-predictive-control-udacitys-self-driving-car-nanodegree-ad7cf64fd0e4

agent capable of driving along the road, never skipping the center of the lane and never be driving for less than 1 km/h for more than 5 seconds while driving for 1245m (3 laps) as shown in the following figure:

Figure 21 - Map where the agent was trained

This dissertation is different to what was done in [30] because the state representation is a simple and explainable array, while in [30] the state is defined through a Variational Auto Encoder. Besides that, this dissertation was developed in a more complex map in a city environment with different objects and a lot more roads, and it has another agent (the leader) that brings more complexity like using CV to predict the distance to the other agent and avoid crashes.

Chapter 3

Reinforcement Learning

This chapter formally presents the necessary background and fundamental concepts to support the choice of Proximal Policy Optimization (PPO).

3.1 Proximal Policy Optimization

PPO was proposed by [44] with the objective of improving Trust Region Policy Optimization (TRPO) [45] making it easy to implement, with higher sample efficiency and with few hyperparameters to tune. This RL algorithm has better convergence properties than previous RL approaches because it clips the policy loss and calculates the loss in terms of probability ration instead of optimizing the policy's likelihood directly.

PPO is based on an actor-critic architecture that combines trust region optimization with gradient descent to stabilize training by creating a loss function that only allows minor changes on the policy, reducing variance and ensure that an outlier does not affect the whole training. As mentioned, PPO is a better version of TRPO which formulates the objective function that constrains the update step within some pessimistic lower-bound called trust region.

In TRPO the upper-bound error is estimated by the Kullback-Leibler (KL) divergence [23] which is a measure of how much one probability distribution differs from another. Using this measure, it is possible to constrain the optimization step between the old and the new policy ensuring that the new policy is not diverging too far from the old policy. In other words, the new policy will be in the trust region of the old policy. This constraint allows to perform multiple update steps per sample because the new policy will not diverge that much from the old policy from one step to another, increasing the sample efficiency. To measure how much both policies are diverging, one can express the optimization problem in terms of the new and old policy:

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{\mathrm{A}}_t\right] \tag{3.1}$$

$$subject\ to\ \mathbb{E}_t[KL[\pi_{\theta old}(\cdot\,|s_t), \pi_\theta(\cdot\,|s_t)]] \leq \delta$$

where $\pi_{\theta old}$ is the old policy which means the policy before the update, π_θ is the new policy and $\hat{\mathrm{A}}$ is the advantage function that is defined through equation 3.2, where *T* is the number of timesteps γ is the discounted factor and *t* specifies the time index *[0, T]*. The δ_t term is calculated using the reward *r* at time *t* plus the γ times the value function at next state s_{t+1} minus value function at current state s_t:

$$\hat{\mathrm{A}}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \quad (3.2)$$

$$\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \quad (3.3)$$

Although this new concept helps on sample efficiency, it is also true that it is complicated to implement and incompatible with models that have noise (such as dropout), or models that share parameters between the policy and the value function. PPO aims to improve the equation 3.1 that needs to be optimized with a second-order optimization such as the conjugate gradient algorithm instead of the first-order optimization methods such as gradient decent by reformulating the objective function as a clipped objective function that can be optimized through gradient decent. Let's start by reformulating the equation 3.1 as an unconstrained loss function:

$$L^{IS}_{\theta old}(\theta) = \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{\mathrm{A}}_t\right] \quad (3.4)$$

where *IS* means importance sampling [43]. To directly optimize the policy of an agent by first-order optimization, one needs to calculate $\nabla_\theta \log \pi_\theta(a|s)$. On the other hand, the loss function $L^{IS}_{\theta old}(\theta)$ is a loss expressed by the ration between the old and the new policy that can be proven that these gradients are the same:

$$\nabla_\theta \log \pi_\theta(a|s)\,|_{\theta old} = \frac{\nabla_\theta \pi_\theta(a|s)|_{\theta old}}{\pi_{\theta old}(a|s)} = \nabla_\theta\left(\frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}\right)|_{\theta old} \quad (3.5)$$

With this reformulation it is possible to optimize $L^{IS}_{\theta old}(\theta)$ with gradient decent because it is equivalent of optimizing the policy gradient $\nabla_\theta \, log \, \pi_\theta(a|s)$ and, at the same time, imposing a trust region constraint on the loss function in terms of the new and old policy as the authors in [44] propose through a clipped loss function. Let's consider $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}$:

$$L^{CLIP}(\theta) \; = \; \widehat{\mathbb{E}_t}[min(r_t(\theta)\hat{\mathrm{A}}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\,\hat{\mathrm{A}}_t)] \tag{3.6}$$

In this equation 3.6, ε is an important hyperparameter (usually 0.2) that determines how much the new policy can diverge from the old policy to improve the policy. In other words, the size of the trust region (Figure 22).

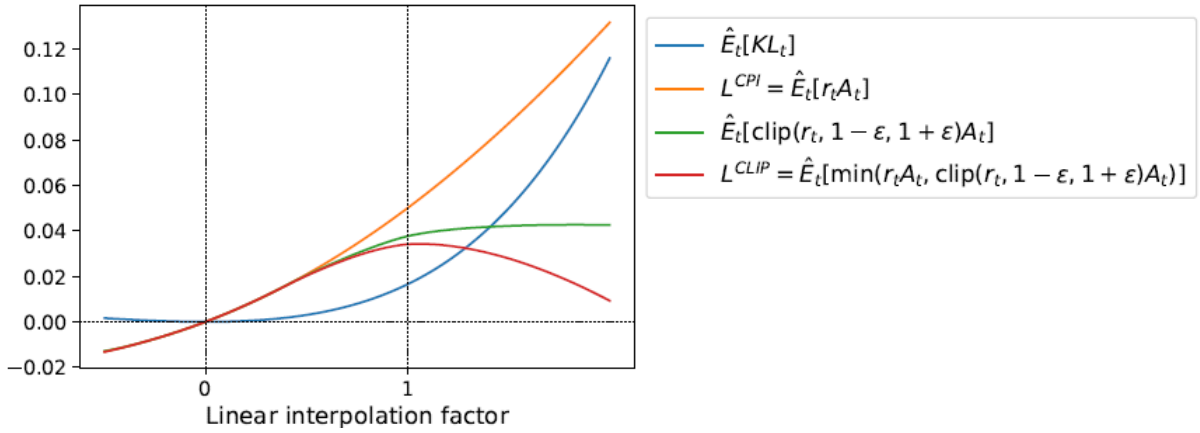

Figure 22 – Shows how different loss functions would linearly interpolate the old policy (θ_{old}) to the new policy (θ). It is noticeable that L^{CLIP} tends to 0 as the new policy diverges more from the old policy. Source: [44].

The minimum in the objective function computes the minimum between the unclipped and clipped objective, where the unclipped one is the regular $L^{IS}_{\theta old}$ loss, while the clipped one limits the probability ratio r_t to the interval [1- ε, 1+ ε] to ensure conservative changes. The minimum term ensures that whenever the new policy is advantageous, that is, if $A > 0$ and $r_t(\theta) > 1+\varepsilon$ or $A < 0$ and $r_t(\theta) < 1-\varepsilon$, there is a constraint to the update keeping it within the trust region (clipped objective). Otherwise, if the new policy is advantageous but within the trust region, that is, $A > 0$ and $r_t(\theta) < 1+\varepsilon$ or $A < 0$ and $r_t(\theta) > 1-\varepsilon$, then the unclipped side of the objective function is activated (*Figure 23*).

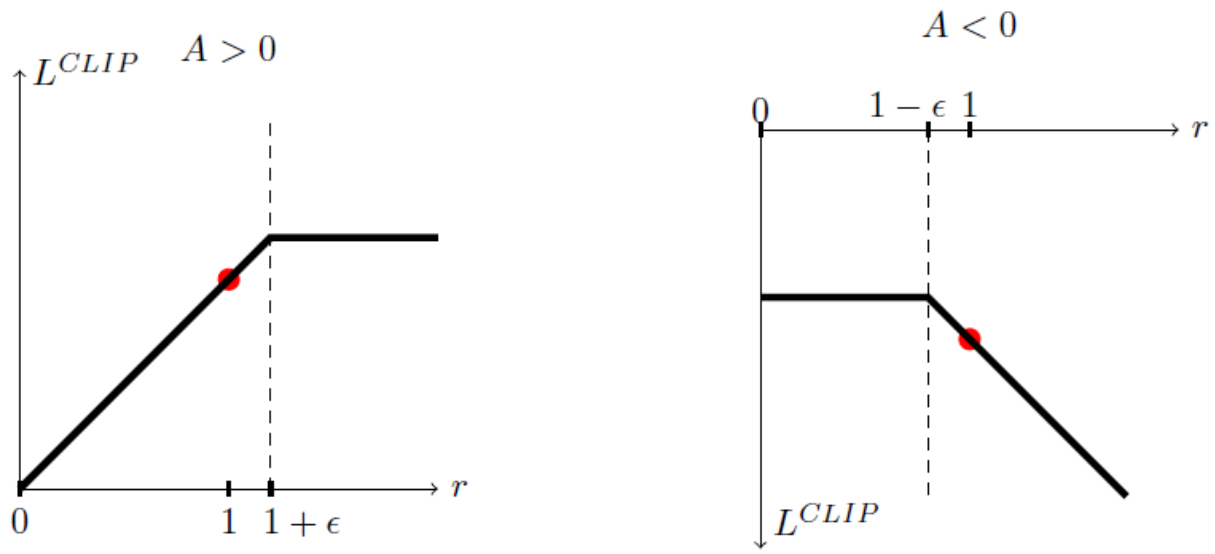

Figure 23 - Plots showing one term (i.e., a single timestep) of the surrogate function LCLIP as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., r = 1. Source: [44]

As mentioned, the formulation of the optimization problem in terms of differentiable loss function allows to use gradient decent, which, contrary to TRPO, a critic can be parameterized in terms of θ. PPO optimizes the critic by introducing a value loss function $L^{VF} = (V(s_t; \theta_v) - R_t(\tau))^2$ and an entropy term, $-\frac{1}{2}(\log(2\pi\sigma^2) + 1)$. The final equation is described as:

$$L^{CLIP+VF+S}(\theta) = -\widehat{\mathbb{E}_t}[L^{CLIP}(\theta) - \alpha L^{VF}(\theta) - \beta\frac{1}{2}(log(2\pi\sigma^2) + 1)] \quad (3.7)$$

Finally, the PPO algorithm is defined as in Figure 24 where the policy gradient loss function is the clipped loss function, there is a loop repeating the gradient update on random mini batches of samples over K epochs and the advantage estimate $\hat{A}$ (equation 3.2) uses the more accurate generalized advantage estimation.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 24 – PPO algorithm. Source: [44]

Chapter 4

Computer Vision

This chapter aims to support the choice of Xception as the CNN chosen to perform the regression task of predicting the distance to the car leader based on the segmented image.

4.1 Xception

Xception stands for "Extreme Inception" because the hypothesis assumed to build this CNN is stronger than the hypothesis assumed to build Inception [6].

A convolution layer attempts to learn filters in a 3D space considering two spatial dimension such as width and height and the channel dimension, therefore a convolution kernel needs to simultaneously map cross-channel correlations and spatial correlations.

The idea behind the Inception module is to make this process easier and more efficient by splitting into a set of operations that would independently look at cross-channel and spatial correlations. More specifically, the Inception module first looks at cross-channel correlations via a set of 1x1 convolutions which maps the input data into 3 or 4 separated spaces that are smaller than the input space. After that, it maps all correlations in these smaller 3D spaces, via regular 3x3 or 5x5 convolutions. To sum up, the assumption behind Inception is that cross-channel correlations and spatial correlations are decoupled enough that it is preferable not to map them together.

Looking into a simplified version of Inception that only uses 3x3 convolutions and does not include an average pooling tower (*Figure 25*), the authors of Xception easily reformulated the Inception module as a large 1x1 convolution followed by spatial convolutions that would operate on non-overlapping segments of the output channels (*Figure 26*). Consequently, they raised the following question *"wouldn't it be reasonable to make a much stronger hypothesis than the Inception hypothesis, and assume that cross-channel correlations and spatial correlations can be mapped completely separately?"*.

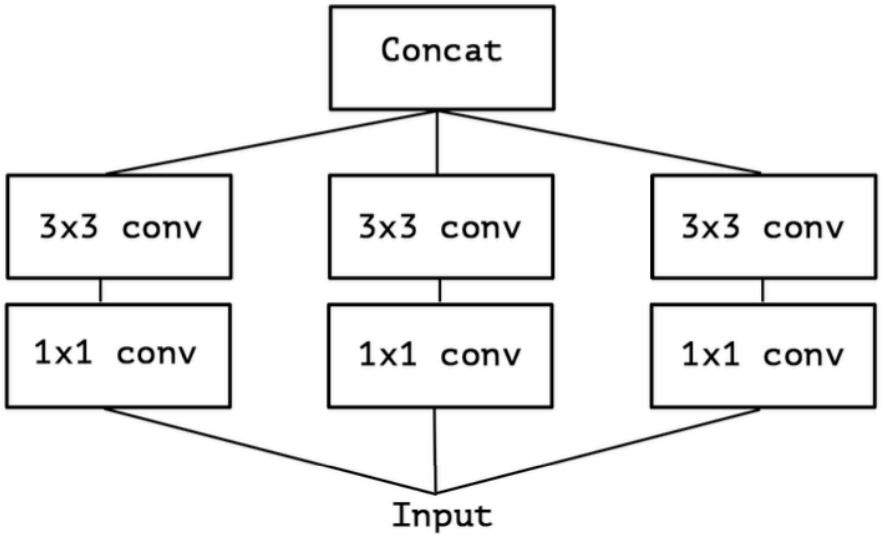

Figure 25 - Simplified Inception Module.

Source: [6]

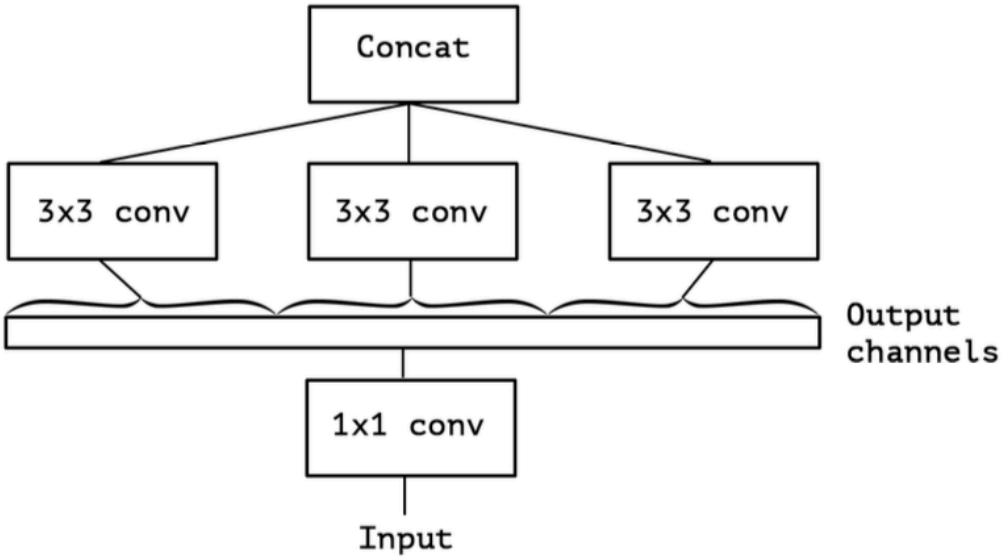

Figure 26 - A strictly equivalent reformulation of the simplified Inception module.

Source: [6]

Using this strong hypothesis, the authors proposed to first use a 1x1 convolution to map cross-channel correlations and, then, separately map the spatial correlations of every output channel as shown in *Figure 27*. This is identical to a depth wise separable convolution which is an operation that has been used since 2014 [47].

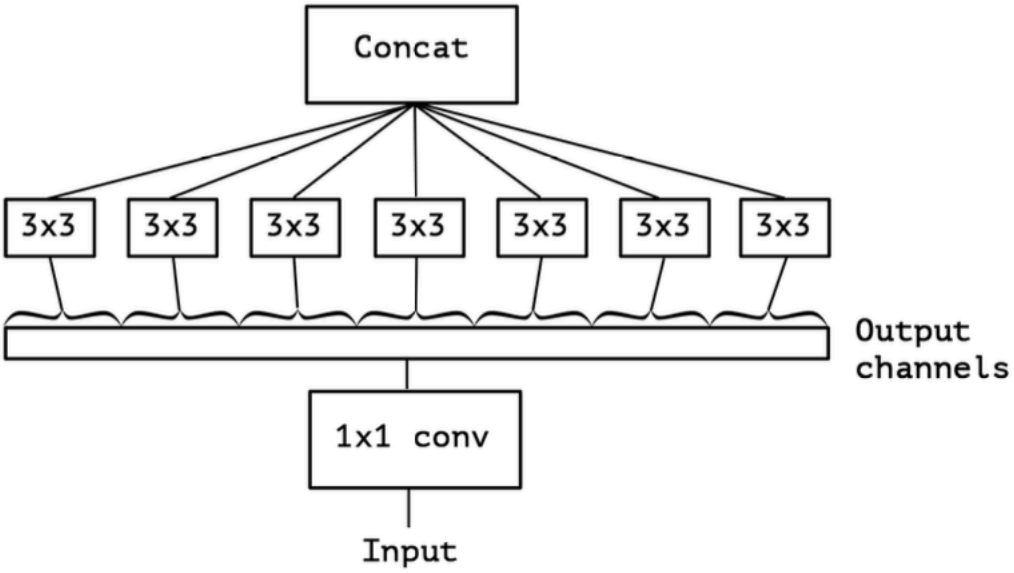

Figure 27 - An “extreme” version of our Inception module, with one spatial convolution per output channel of the 1x1 convolution.

Source: [6]

The depthwise separable convolution is a spatial convolution performed independently over each input channel. This is followed by a pointwise convolution which is a 1x1 convolution that projects the channels output by the depthwise convolution onto a new channel space.

There are two differences between an extreme version of Inception module and a depthwise separable convolution:

1. The order of the operations where depthwise separable convolutions firstly performs channel-wise spatial convolutions and then 1x1 convolution, whereas Inception performs 1x1 convolution first.
2. The presence of a non-linearity after the first operation. In Inception, both operations are followed by a ReLU, whereas depthwise separable convolutions are implemented without non-linearity.

The use of depthwise separable convolutions are the main modification and novelty introduced by the authors on the Inception architecture. However, they also relied on the VGG-16 architecture [49] to build their own architecture and on the residual connections introduced by [19] which they use extensively in Xception’s architecture.

Xception's architecture is based entirely on depthwise separable convolution layers which entirely decouples the mapping of cross-channels correlations and special correlations in the feature maps of CNN.

The architecture is composed by 36 convolutional layers that are responsible for the feature extraction, followed by a logistic regression layer since the purpose of this network is to classify images. Nevertheless, this last layer can always be modified to perform any other task such as regression with a linear layer.

The 36 convolutional layers are divided into 14 modules with linear residual connections around them, except for the first and last modules as shown in *Figure 28*.

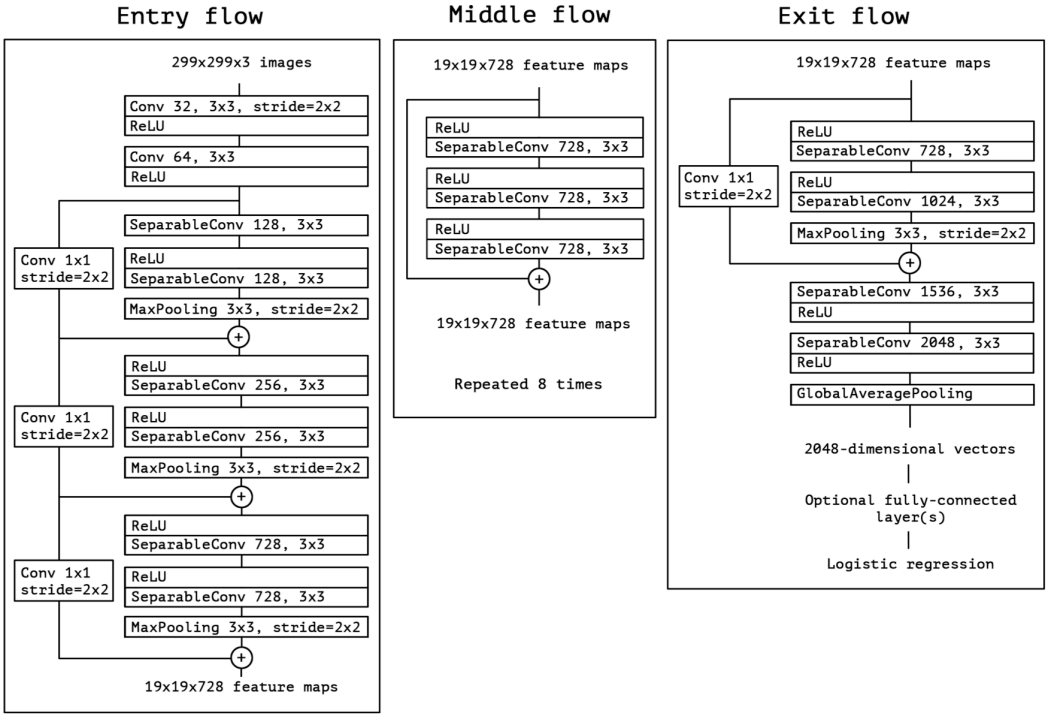

Figure 28 - The Xception architecture: the data first goes through the entry flow, then through the middle flow which is repeated eight times, and finally through the exit flow. Note that all Convolution and SeparableConvolution layers are followed by batch normalization [22] (not included in the diagram). All SeparableConvolution layers use a depth multiplier of 1 (no depth expansion).

Source: [6]

Experimental results using the ImageNet dataset showed that Xception surpassed VGG-16, ResNet-152 and Inception V3 in terms of the classification accuracy. In terms of speed, Xception was faster than Inception V3 and it also has less parameters than Inception V3.

Regarding if the presence or absence of non-linearity between depthwise and pointwise operations would benefit, or not, the performance of the model, the authors performed experiments in the ImageNet and they concluded that the absence of non-linearity benefits the model's accuracy *(Figure 29*).

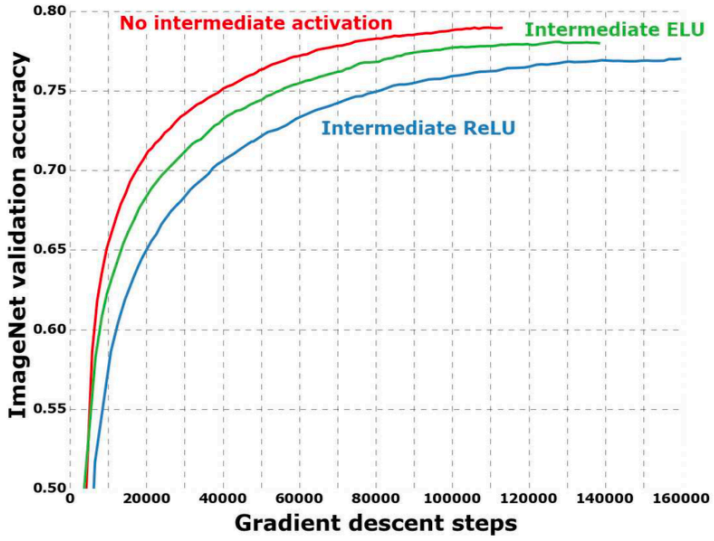

Figure 29 - The effect of non-linearity presence between depthwise and pointwise operations.

Source: [6]

The use of residuals in the architecture also helped improving accuracy as shown in the *Figure 30*.

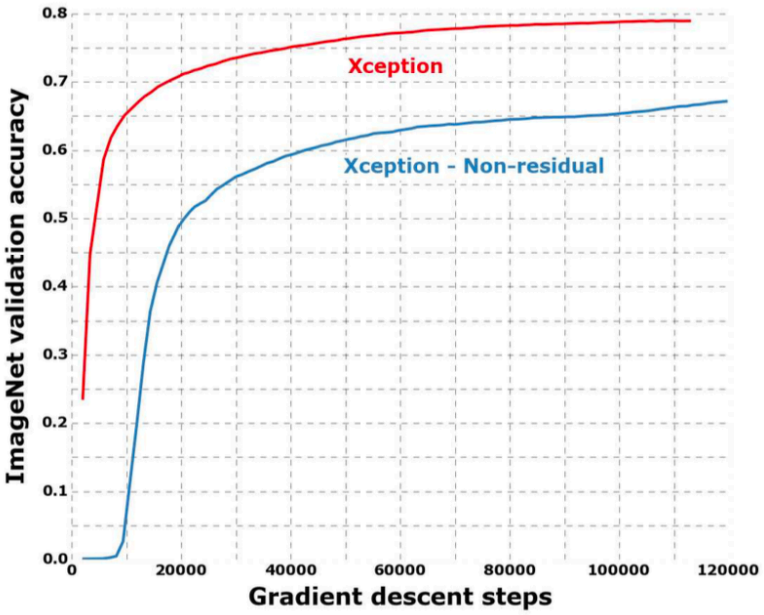

Figure 30 - Impact of residuals on accuracy in Xception architecture.

Source: [6]

Chapter 5

PPO and Xception for Platooning Vehicles

This chapter is organized in two parts. The first part consists of a detailed description about the implementation of PPO and Xception. The second part presents the results in CARLA and the explain ability of the model.

5.1 Implementation

This section will give an overview of the global process implemented as well as all the assumptions taken to reach the thesis goal. Subsection 5.1.1 will state the problem. Subsection 5.1.2 will explain how the segmentation camera available in CARLA works. Subsection 5.1.3 will explain how transfer learning and Xception were used to predict the distance to the car leader. Finally, subsection 5.1.4 will show the architecture of the two RL agents created to control the steer and the throttle/break.

5.1.1 Problem Statement

The problem that this dissertation aimed to study was how to create a RL agent able to follow another car using only information retrieved by a monocular camera and, at the same time, be able to explain the decisions taken by the RL model.

To be able to reach this goal some assumptions were taken:

1. There was only one car to be followed in the simulations because there was not a way in CARLA to create a distinctive sign on the car that the agent should follow.
2. Two independent agents were created. One to control the steer and another one to control the throttle and break.
3. There were no people on streets or any other dynamic actors.
4. The car starts always at a distance of 6 from the leader (there is no distance unit since it is a Euclidean distance) and with a velocity of 12 km/h.
5. The car cannot be at a distance higher than 25, otherwise the episode will end.

6. The label used to train the Xception model is the Euclidean distance between both cars which is not optimal when the car curves.
7. The episodes had at most a duration of 60 seconds.
8. The use of segmentation camera provided by CARLA to simplify the process by removing unnecessary constraints that can be quickly solved because there is already research and proven results.
9. Moving backwards is not allowed.

The assumptions were taken to simplify some processes that either were already investigated, or they are simply solved when implemented in the real life. They also allowed to develop the dissertation in the time period defined.

5.1.2 Segmentation Camera

As mentioned in the previous subsection a segmentation camera provided by the simulator was used to retrieve the main information needed to create the environment states that feed the RL agents.

The segmentation camera classifies each object in a RGB image with a different tag that allows to map the tag to the object identified through CARLA's documentation (see *Table 1*). When the simulation starts, every object in the environment is created with a tag.

When this camera retrieves an image, the tag information is encoded in the red channel meaning that a pixel with a red value of *'10'* belong to the object with tag *'10* [28].

Table 1 - Mapping between tags and objects
Source: https://carla.readthedocs.io/en/latest/ref_sensors/#semantic-segmentation-camera

Value	Tag	Converted color	Description
0	Unlabeled	(0, 0, 0)	Elements that have not been categorized are considered Unlabeled. This category is meant to be empty or at least contain elements with no collisions.

[28] https://carla.readthedocs.io/en/latest/ref_sensors/#semantic-segmentation-camera

Value	Tag	Converted color	Description
1	Building	(70, 70, 70)	Buildings like houses, skyscrapers,... and the elements attached to them. E.g. air conditioners, scaffolding, awning or ladders and much more.
2	Fence	(100, 40, 40)	Barriers, railing, or other upright structures. Basically wood or wire assemblies that enclose an area of ground.
3	Other	(55, 90, 80)	Everything that does not belong to any other category.
4	Pedestrian	(220, 20, 60)	Humans that walk or ride/drive any kind of vehicle or mobility system. E.g. bicycles or scooters, skateboards, horses, roller-blades, wheel-chairs, etc.
5	Pole	(153, 153, 153)	Small mainly vertically oriented pole. If the pole has a horizontal part (often for traffic light poles) this is also considered pole. E.g. sign pole, traffic light poles.
6	RoadLine	(157, 234, 50)	The markings on the road.
7	Road	(128, 64, 128)	Part of ground on which cars usually drive. E.g. lanes in any directions, and streets.
8	SideWalk	(244, 35, 232)	Part of ground designated for pedestrians or cyclists. Delimited from the road by some obstacle (such as curbs or poles), not only by markings. This label includes a possibly delimiting curb, traffic islands (the walkable part), and pedestrian zones.
9	Vegetation	(107, 142, 35)	Trees, hedges, all kinds of vertical vegetation. Ground-level vegetation is considered Terrain.

Value	Tag	Converted color	Description
10	Vehicles	(0, 0, 142)	Cars, vans, trucks, motorcycles, bikes, buses, trains.
11	Wall	(102, 102, 156)	Individual standing walls. Not part of a building.
12	TrafficSign	(220, 220, 0)	Signs installed by the state/city authority, usually for traffic regulation. This category does not include the poles where signs are attached to. E.g. traffic- signs, parking signs, direction signs...
13	Sky	(70, 130, 180)	Open sky. Includes clouds and the sun.
14	Ground	(81, 0, 81)	Any horizontal ground-level structures that does not match any other category. For example areas shared by vehicles and pedestrians, or flat roundabouts delimited from the road by a curb.
15	Bridge	(150, 100, 100)	Only the structure of the bridge. Fences, people, vehicles, an other elements on top of it are labeled separately.
16	RailTrack	(230, 150, 140)	All kind of rail tracks that are non-drivable by cars. E.g. subway and train rail tracks.
17	GuardRail	(180, 165, 180)	All types of guard rails/crash barriers.
18	TrafficLight	(250, 170, 30)	Traffic light boxes without their poles.
19	Static	(110, 190, 160)	Elements in the scene and props that are immovable. E.g. fire hydrants, fixed benches, fountains, bus stops, etc.
20	Dynamic	(170, 120, 50)	Elements whose position is susceptible to change over time.

Value	Tag	Converted color	Description
21	Water	(45, 60, 150)	Horizontal water surfaces. E.g. Lakes, sea, rivers.
22	Terrain	(145, 170, 100)	Grass, ground-level vegetation, soil or sand. These areas are not meant to be driven on. This label includes a possibly delimiting curb.

5.1.3 Xception

This section describes the changes needed in the architecture of Xception network to be able to perform a regression task as well as the dataset used and the way it was collected, the training evolution and, finally, the performance results.

5.1.3.1 Network

Most of the CNNs available for Transfer Learning in Tensorflow [1] were trained in the *'Imagenet'* dataset, where the goal is to classify an image based on the 1000 labels present in the dataset.

Since the goal for this dissertation is to predict distance, which is a regression problem, the output layer was changed from 1000 nodes with a Softmax activation function to 1 node with a Linear activation function.

Before the output layer, a Global Average Pooling 2D [27] layer was used to flatten the output of the previous layers since it applies average pooling on the spatial dimensions until each spatial dimension is one. According to the authors, this approach reduces the number of trainable parameters which reduces the tendency of over-fitting, that needs to be managed in fully connected layers using dropout. They also argue that removing the fully connected classification layers forces the feature maps to be more closely related to the classification categories which makes the model more robust to spatial translations in the data.

The choice of Xception relied on what was explained in subsection *4.1 Xception* and, also, on the *Table 2* which shows the trade-off between the number of parameters, accuracy, and time for inference.

In that table is possible to see that Xception is one of the models with highest accuracy and low latency in GPU, which is important because the agent needs a fast prediction so that the environment does not change significantly between retrieving the information and taking the action.

Table 2 - Available Models in Tensorflow

Source: https://keras.io/api/applications/

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	46.0	4.9

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
EfficientNetB1	31	79.1%	94.4%	7.9M	186	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	186	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	210	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	258	308.3	15.1
EfficientNetB5	118	83.6%	96.7%	30.6M	312	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	360	958.1	40.4
EfficientNetB7	256	84.3%	97.0%	66.7M	438	1578.9	61.6
EfficientNetV2B0	29	78.7%	94.3%	7.2M	-	-	-
EfficientNetV2B1	34	79.8%	95.0%	8.2M	-	-	-
EfficientNetV2B2	42	80.5%	95.1%	10.2M	-	-	-
EfficientNetV2B3	59	82.0%	95.8%	14.5M	-	-	-
EfficientNetV2S	88	83.9%	96.7%	21.6M	-	-	-
EfficientNetV2M	220	85.3%	97.4%	54.4M	-	-	-
EfficientNetV2L	479	85.7%	97.5%	119.0M	-	-	-

5.1.3.2 Dataset

Transfer Learning requires the existence of pre-trained weights in a dataset that are going to be updated when trained on the dataset of the new task. As mentioned in subsection *2.2.3 Transfer Learning,* this approach speeds up the training process because the first layers usually are frozen which means their weights will not be updated since the high-level features are similar from task to task.

The dataset created for this work is composed by 25 448 images which were collected in 2 steps:

1. The first one was training the RL agent responsible for the throttle/break with the real Euclidian distance and then, using this agent in a test environment so that the agent could follow the leader and could collect segmented images like the following one in *Figure 31*:

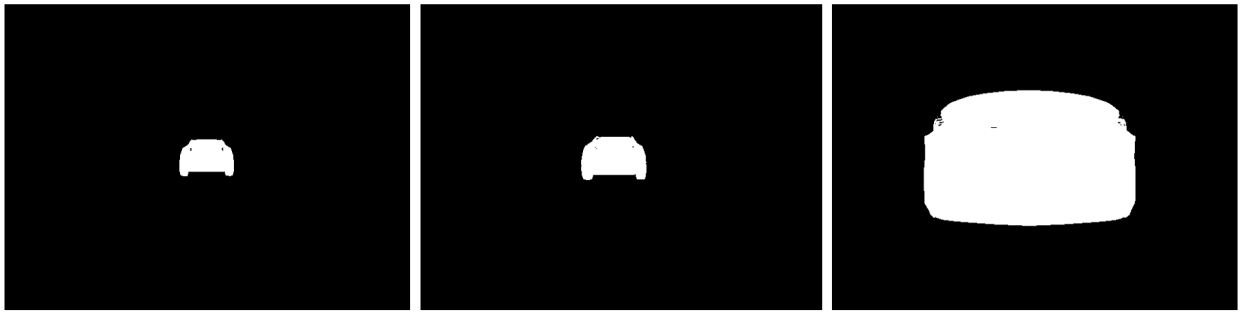

Figure 31 - Example of segmented images collected to train Xception

2. The second one was to enrich the dataset with distances that the first step could not retrieve. Since the algorithm was already optimized and short distances, like 4 or 5 were not presented in the dataset made the model overpredicting the distance in those cases and, consequently, crash into the leader.

The final dataset has an average of 7.97, a standard deviation of 1.44, a minimum of 3.94, a maximum of 14.59 and a median of 8.36. The distribution and the boxplot can be seen in the following images (*Figure 32* and *Figure 33*):

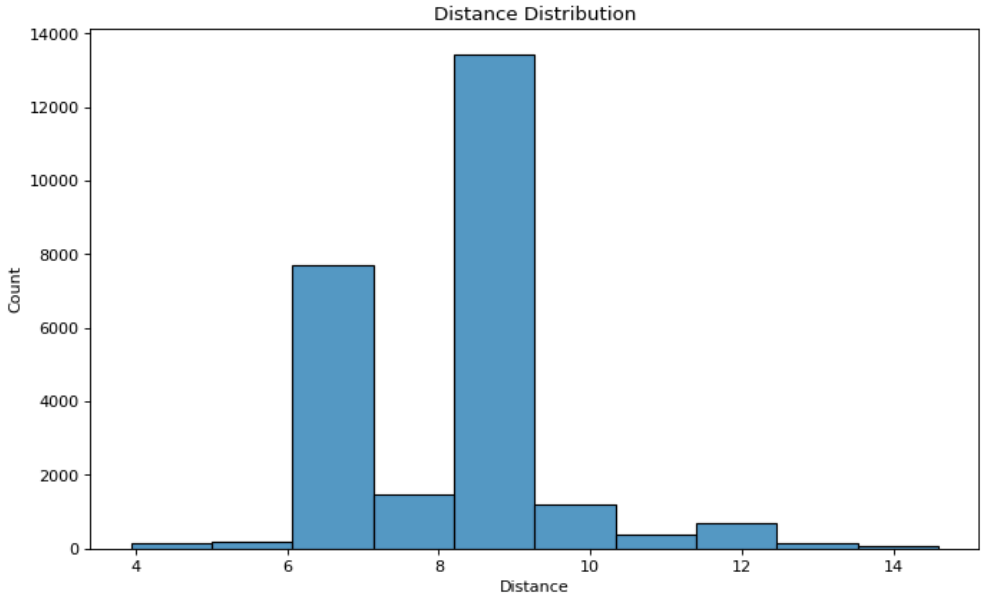

Figure 32 - Distance Distribution with most of the values between 8 and 10 because it was the range where the agent would get positive rewards (this will be explained later).

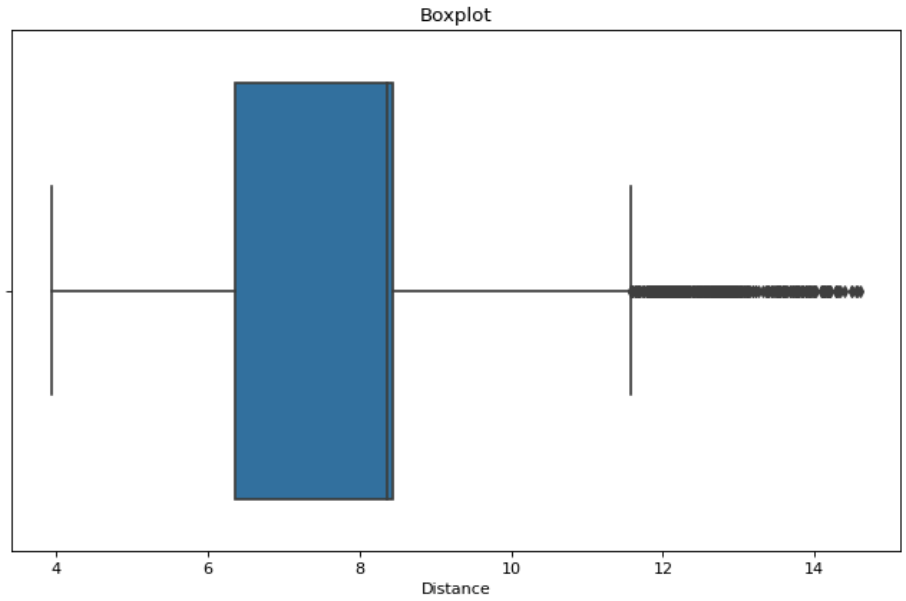

Figure 33 - Boxplot with percentile 25% on 6.35, median on 8.36 and percentile 75% on 8.44

5.1.3.3 Training

The training set up was done by splitting the dataset into three datasets: training, validation, and test set with 70%, 10% and 20% of images, respectively.

For the back propagation of the network the optimizer used was *Adam* [24].

Adam is different from classical stochastic gradient descent because while the latter maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training, *Adam* combines the advantages of two other extensions of stochastic gradient descent:

- **Adaptive Gradient Algorithm** (AdaGrad) [9] that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
- **Root Mean Square Propagation** (RMSProp) [39] that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in *RMSProp*, *Adam* also makes use of the average of the second moments of the gradients (the uncentered variance).

Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

The initial value of the moving averages and beta1 and beta2 values close to 1.0 (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

The metric to be optimized by *Adam* was Mean Absolute Error (MAE) which evaluates the average absolute difference between the ground truth and the value predicted and it can be interpreted as closer to zero the better. MAE is given by

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - x| \qquad (5.1)$$

, where *n* stands for the number of observations.

Finally, regarding the number of epochs and the batch size, they were 40 and 64 respectively. The model took 5.45 hours to run in Google Colab using GPUs and the evolution of the loss can be seen in *Figure 34*.

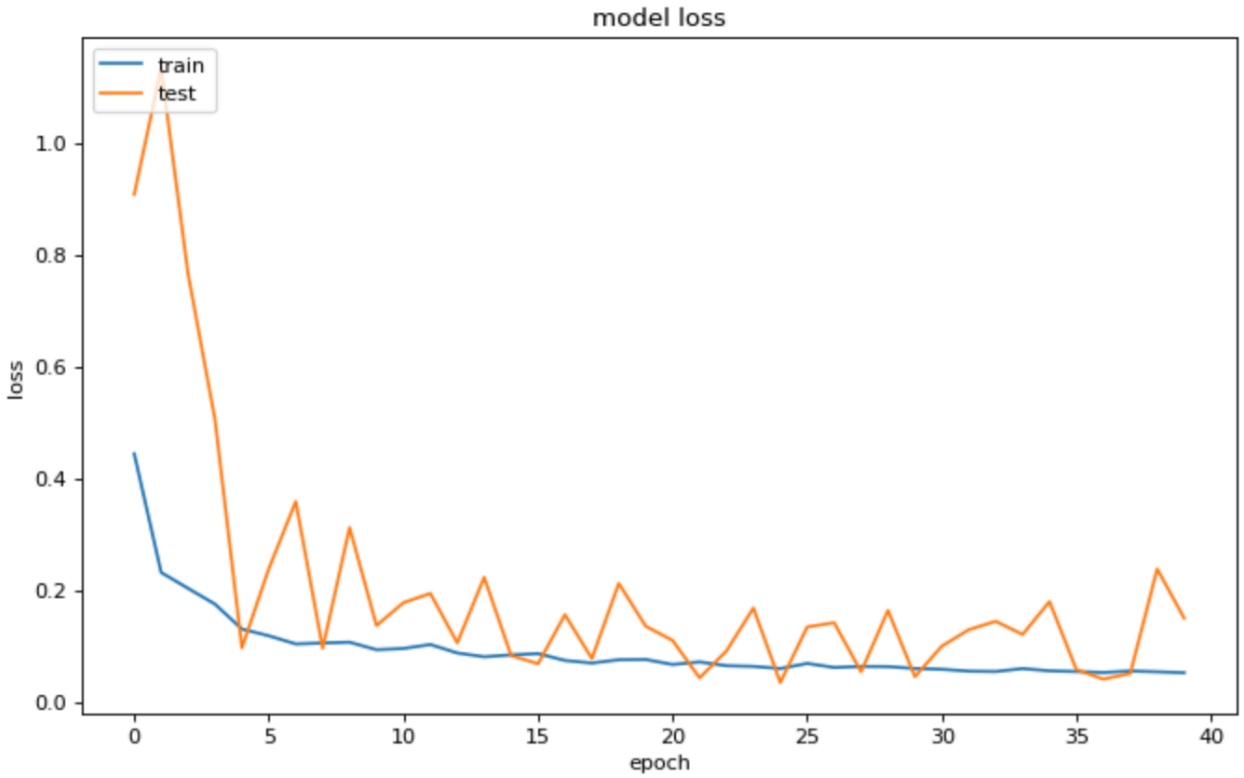

Figure 34 - Loss Evolution

Figure 34 shows that the model only needs 4 epochs to achieve a low loss (MAE) in the validation set. However, the convergency seems to be achieved in epoch 37, therefore, the weights used to predict the distance are the ones from epoch 37 which was possible to save by using callbacks during training which saves the weights from the best epoch.

5.1.3.4 Results

The model produced a MAE of 1.47 in the test set (M5), which means that there are errors above and below 1.47 and this distribution can be seen in *Figure 35*.

From the distribution, it is possible to see that the model is overestimating the distance since there are more negative values than positive ones, which is not a problem because it can increase safety for lower values of distance. Also, 50% of the errors are between -1.51 and 1.78 which is a low value when comparing with the average distance in the dataset (7.97).

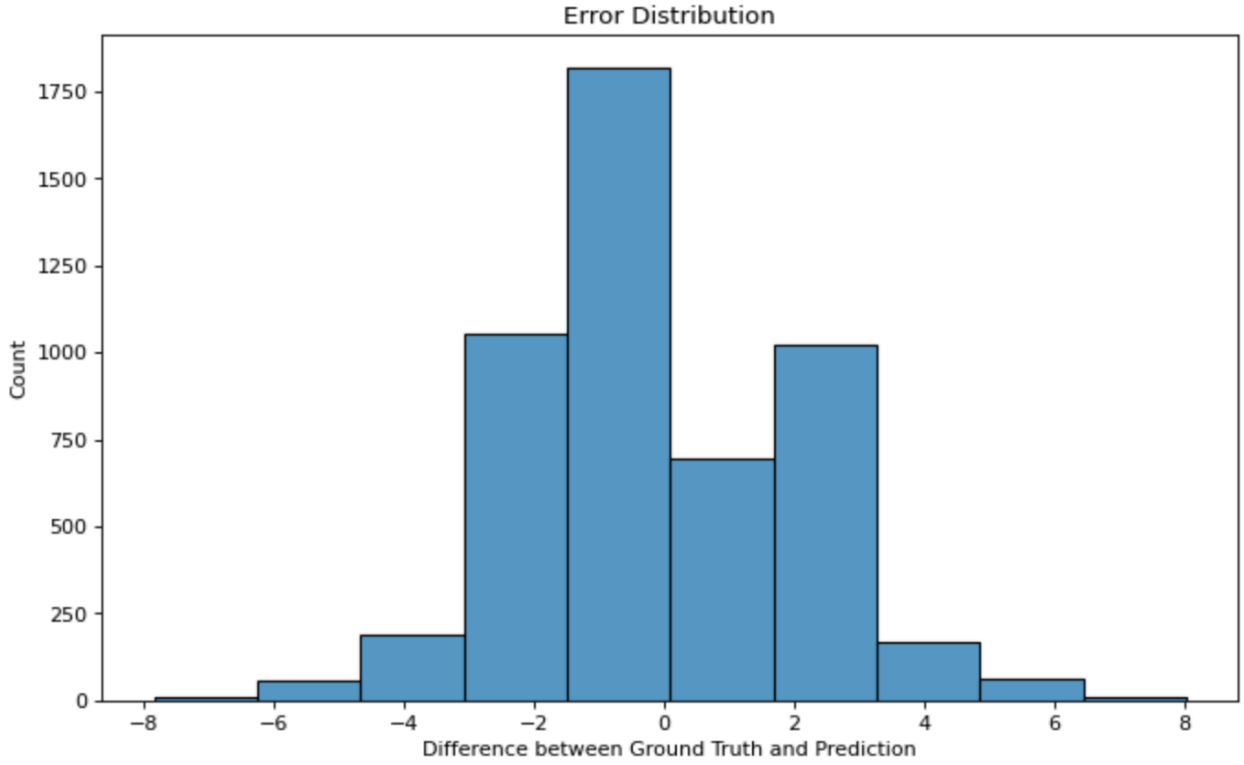

Figure 35 - Error distribution in the test set

5.1.4 PPO Architectures

This section provides an extensive overview related to the states, actions, reward functions, terminal states, architectures, training results and explain ability charts of the RL algorithm. Since

two different agents based on PPO were developed, this section will be divided into two. The first one will explain the agent responsible for the steer and the second one will explain the agent responsible for the throttle/brake.

5.1.4.1 Steer Agent

5.1.4.1.1 State

The state for this agent consists only into two binary features called *'left'* and *'right'*. When *'left'* is 1 then *'right'* is 0, this means that the agent is not aligned with the leader and that the leader is on the left of the agent. When *'right'* is 1 then *'left'* is 0, this means that the agent is not aligned with the leader and that the leader is on the right of the agent. When both *'left'* and *'right'* are 0, this means that the agent is aligned with the leader. Initially, the state had also the feature *'aligned'*, but it was realized that it would mean the same as *'left'* and *'right'* being 0, therefore, to simplify even more the state, this feature was removed.

To extract this information a segmented image of 640x480 (width x height) from the segmentation camera is used. This matrix is converted from a pixel range from 0 to 22 to a binary matrix where 1 identifies where label 10 is presented which is the label for cars according to CARLA documentation. Since there is only the car leader driving in the simulator, there is no need to confirm if those labels really represent the car leader or any other car. To overcome this problem would be necessary to use the license plate or attach an object to the leader so that the agent would be able to identify the leader.

Using matrix manipulation, one can easily extract the following information to determine if *'left'* is 1 or 0 and if *'right'* is 1 or 0 by following these rules:

1. If the center of the leader is on a pixel where the x coordinate is lower than 300 then 'left' is 1 and 'right' is 0;
2. If the center of the leader is on a pixel where the x coordinate is higher than 340 then 'left' is 0 and 'right' is 1;
3. Finally, If the center of the leader is on a pixel where the x coordinate is between 300 and 340 then 'left' is 0 and 'right' is 0.

Mathematically it can be expressed as:

$$state = \begin{cases} center < 300, & left = 1, right = 0 \\ center > 340, & left = 0, right = 1 \\ 300 \leq center \leq 340, & left = 0, right = 0 \end{cases} \quad (5.2)$$

The center of the leader is determined by the following equation:

$$center = \frac{min_x + max_x}{2} \quad (5.3)$$

All the process is illustrated in *Figure 36*:

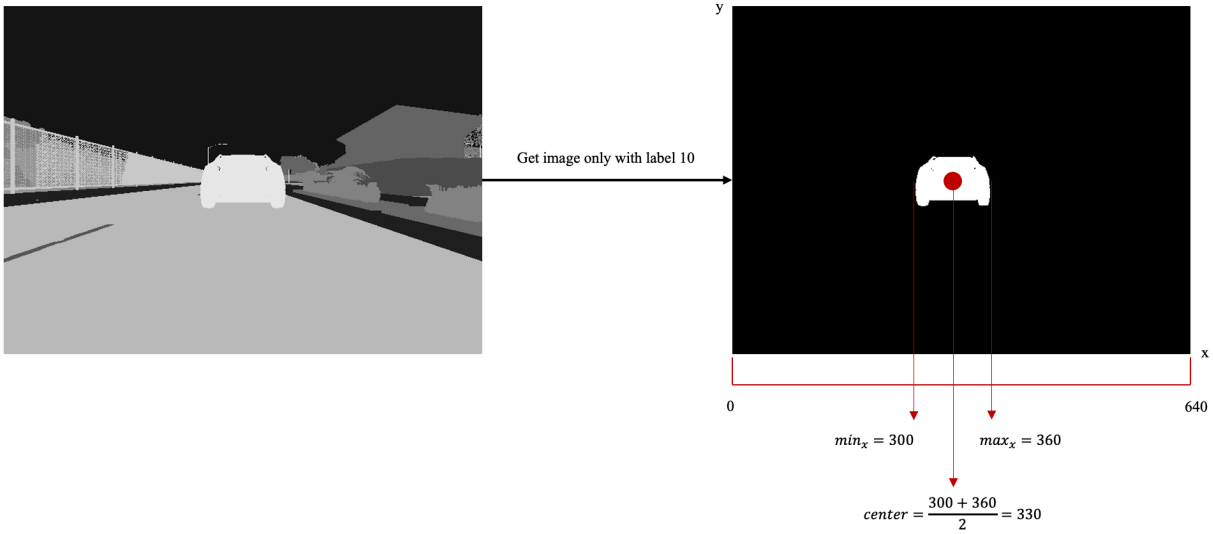

Figure 36 - Process of extracting the state representation for the agent responsible for the steer

5.1.4.1.2 Actions

The agent responsible for the steer has a discrete space of actions constituted for 3 different actions: going straight, turn left and turn right.

The simulator requires a value from -1 to 1 for the steer control. To turn left the value must be below 0 and to turn right the value must be above 0.

Since the agent has discrete action space instead of a continue action space, when the probability of turning left is the highest, the steer value is -0.25. When the probability of turning

right is the highest, the steer value is 0.25. And, when the probability of going straight is the highest then the steer value is 0.

The 0.25 value was defined based on trial and error with the goal of selecting the minimum value possible that could reduce a strong zig zag behavior but at the same time was enough to perform elbow curves. Besides that, here is introduced the first safety layer that avoids sudden turns and, consequently, a dangerous and an aggressive driving behavior.

The steer action can be turned into a continuous value so that the agent can adapt how much it wants to curve, however the state would need to be redefined in a non-binary way as well as the reward functions associated to the steer agent. This is a good opportunity for future work.

5.1.4.1.3 Reward Function

The reward function is composed by 4 different components.

The first component is related to the collision in which the agent receives a negative reward of -10 if a collision happens.

The second component evaluates if the leader was lost. In the case the leader is not visible in the camera, then the agent receives a negative reward of -10.

The third component can be seen as an alignment component once it gives a negative reward of -5 if the agent is not aligned with the leader and a positive reward of 5 otherwise.

Finally, the fourth component tries to guide the agent actions by giving a negative reward of -5 if the agent turns left or goes straight and the leader is right, or the agent turns right or goes straight, and the leader is left or the agent does not go straight when it is aligned. On the other hand, if the agent takes the correct decision which means, turning left when the leader is left, turning right when leader is right or going straight when the agent is aligned, then the agent receives a positive reward of 5.

5.1.4.1.4 Terminal State

The episode ends when:

1. The episode is running for longer than 60 seconds; or
2. The agent collides; or
3. There is no label 10 in the segmented image meaning that the agent lost the leader.

Mathematically it can be defined as:

$$terminal\ state = \begin{cases} time > 60s, & true \\ len(collision_{events}) > 0, & true \\ sum(sum(img_{seg} == 10) = 0, & true \\ & false \end{cases} \quad (5.4)$$

5.1.4.1.5 Network Architecture

The network is composed by four layers and can be seen in *Figure 37*:

- Input layer with 2 nodes with ReLU activation function.
- 2 Hidden layers with 64 nodes with ReLU activation function.
- Output layer with 3 nodes and Softmax as the activation function to predict the probability of choosing one of the three actions possible given the actual state.

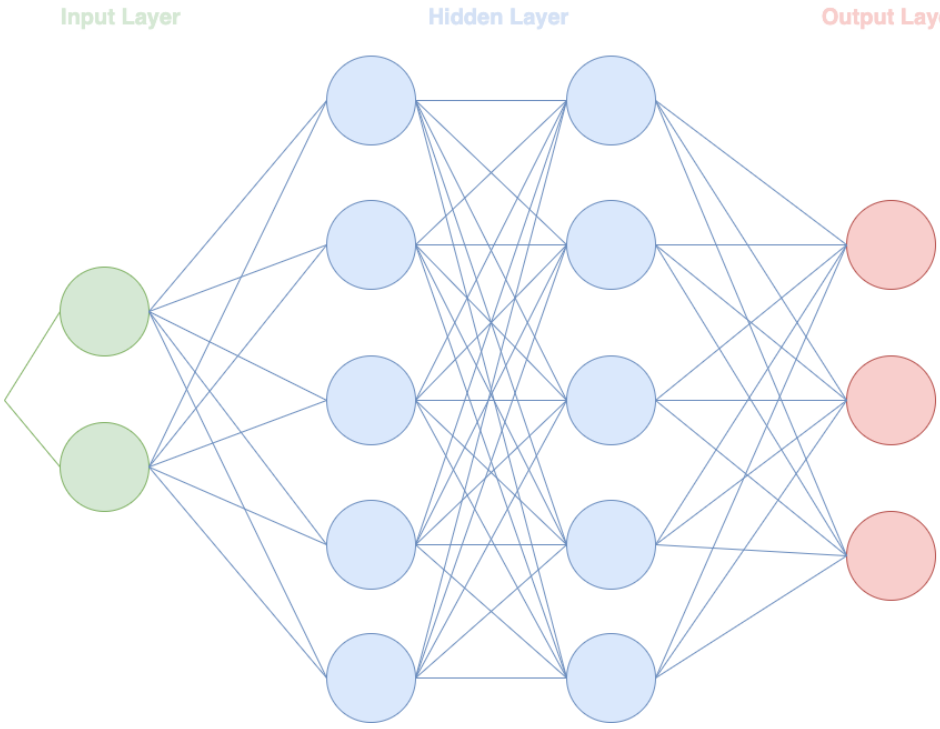

Figure 37 - Steer Agent Architecture.

5.1.4.1.6 Training Results

For the training phase the following hyperparameters were used:

- Batch size: 32
- Epochs: 10
- Gamma: 0.99
- Learning rate: 0.0003
- Policy clip: 0.2
- Episodes: 780

These hyperparameters were not defined using any optimization technique, i.e., they were not fine tuned.

Figure 38 shows the evolution of the cumulative reward per episode in light blue and a moving average with a window of 100 episodes in dark blue.

It is visible that around episode 500 there is a positive drift on the reward obtained by episode, meaning that the agent started to learn what was expected. From episode 650 onwards, the cumulative reward has stabilized and, therefore, one can consider that the model has converged, and no more training is needed.

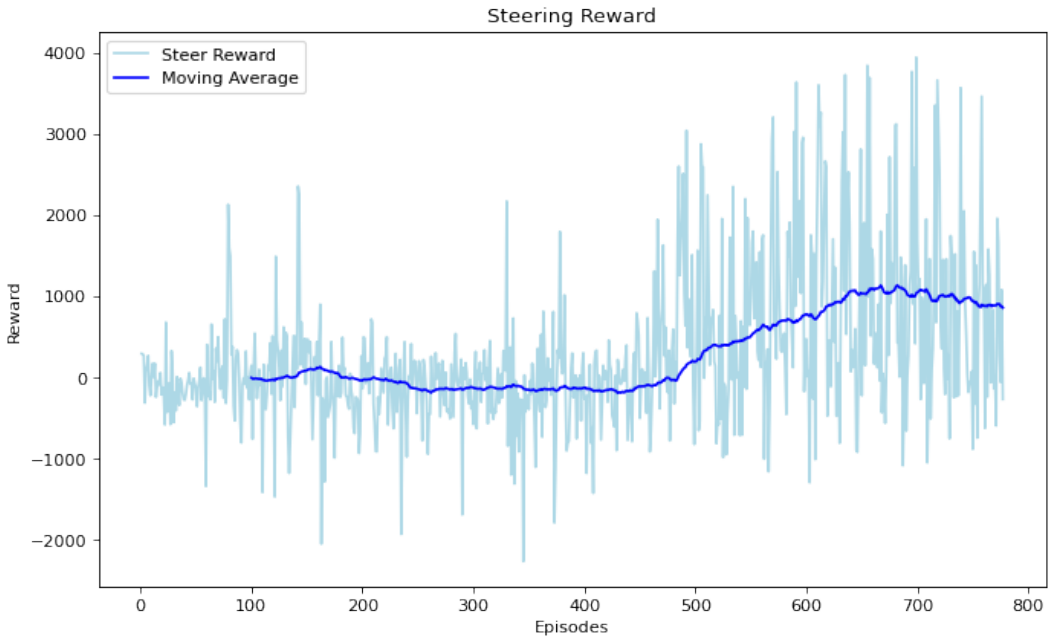

Figure 38 - Cumulative steer reward per episodes (M2)

5.1.4.1.7 Explain Ability

As mentioned previously, the explainable part of the thesis will rely on SHAP values to explain why the agent is taking certain decision based on the state perceived.

As shown in *Figure 39* each possible action has its own chart that allows to take the following conclusions:

- The action to turn left is triggered by the state constituted by ‘left’ equal to 1 and ‘right’ equal to 0, since a red dot means a high feature value which is 1 in a binary feature. In the chart, it is also possible to verify the impact on model output in the x axis, where in this case the red dot points to an impact of more than 0.8.
- The action to go straight is triggered by the state that does not contain any 1 value for either ‘left’ or ‘right’ because it will decrease the chance of taking this action since the impact on model output is less than -0.8. This means, that a state defined by ‘left’ equal to 0 and ‘right’ equal to 0 will increase the chance of taking this action, since a blue dot means a low feature value which is 0 in a binary feature. Indeed, from the chart it is easily understandable that a 1 value in either ‘left’ or ‘right’ decreases the probability of going straight.
- The action to turn right is triggered by the state constituted by ‘left’ equal to 0 and ‘right’ equal to 1. In the chart, it is also possible to verify the impact on model output in the x axis, where in this case the red dot points to an impact of more than 0.8.

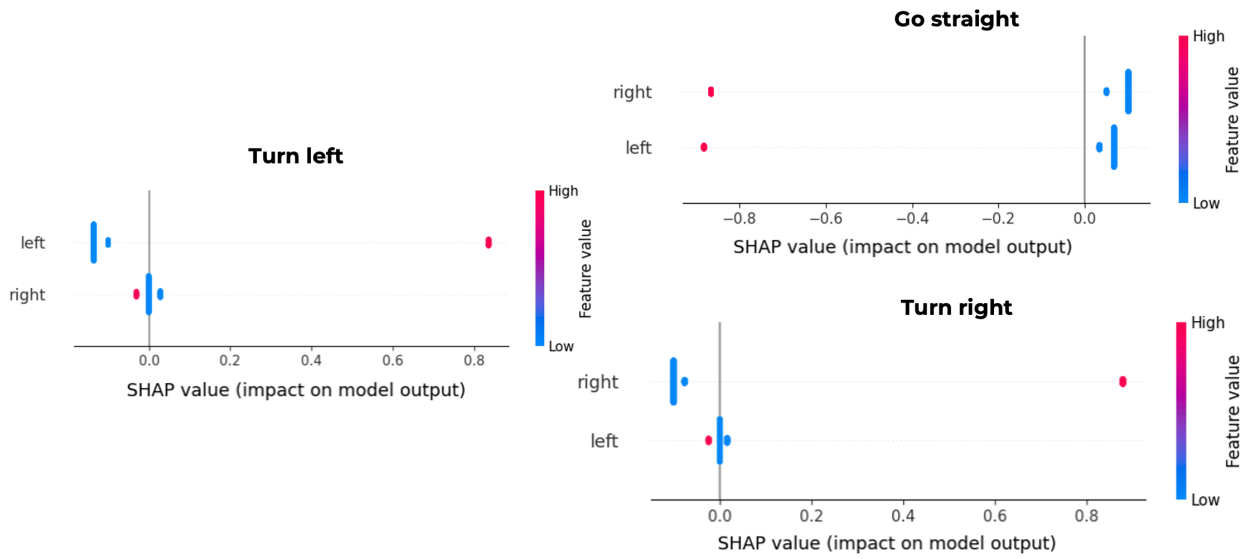

Figure 39 – SHAP values of each possible action.

5.1.4.2 Throttle/Break Agent

5.1.4.2.1 State

The state for this agent consists only into three features called *'distance'*, *'previous distance'* and *'velocity.*

The *'distance'* is the prediction from Xception as well as the *'previous distance'* which is the prediction from the previous state. The velocity is the speed that the agent is driving in that moment.

The agent was also trained only using two features *'distance'* and *'velocity'* in order to reduce complexity; however it did not work out leading to consecutive crashes into the leader since having *'previous distance'* in the state definition allows the agent to understand the velocity of the leader.

Apart from velocity, which is a sensor that every car has, to extract the remaining features for the state, the same segmented image used in the steer agent with the same conversion process to get a matrix filled with zeros except where the leader is in the image which is filled with ones is used to feed the Xception CNN that predicts the distance to create the state.

The values that constitute the state are normalized by dividing the distance values by 25 (which is the maximum value allowed during training, otherwise the episode will end) and the

velocity is divided by 100 (which will range the velocity feature between 0 and 1.2 since the maximum velocity allowed is 120 km/h).

All the process is explained in *Figure 40*:

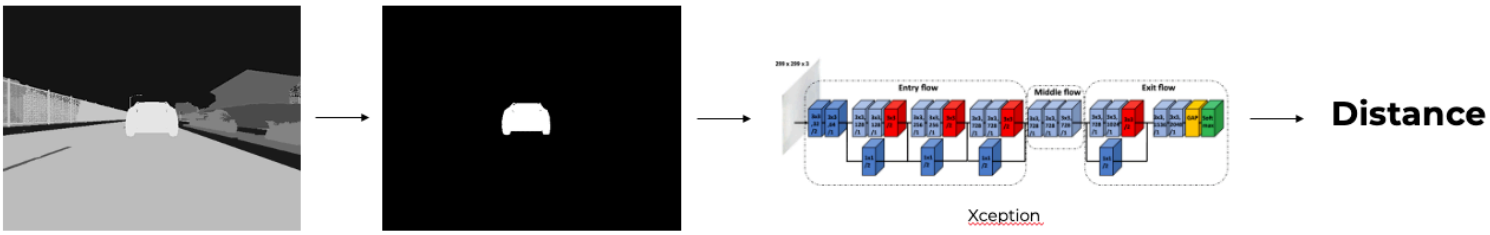

Figure 40 - Predicting distance with Xception

5.1.4.2.2 Actions

The agent responsible for the throttle/break has a continuous space of actions that can range between -1 and 1.

The simulator requires a value from 0 to 1 for the throttle and break control, therefore it was defined that if the value is below 0 then the agent will break in the intensity of the absolute number predicted, if the value is above 0 then the agent will accelerate in the intensity of the value predicted.

During training, since the value predicted comes from a distribution with certain average which is the value predicted by the PPO and a fixed standard deviation defined by the user, the value can be higher than 1 and lower than -1, therefore a clipped was used to make that value -1 when the prediction is lower than -1 and 1 when the prediction is higher than 1.

5.1.4.2.3 Reward Function

The reward function is composed by 5 different components.

The first component is related to the collision in which the agent receives a negative reward of -10 in case a collision has happened.

The second component evaluates if the leader is too far away from the agent: if the leader is farer than a distance of 25, then the agent receives a negative reward of -10.

The third component is used to assess if the agent is at the desired distance to leader which is between 8 and 10: if the distance is between those values the agent receives a positive reward of +5, otherwise it receives a negative reward of -5

The fourth and fifth component can be seen as an guidance component. In case of the fourth component it gives a positive reward of +5 if the agent is reducing the distance to the leader when the distance is higher than 10 or is increasing the distance when the distance is lower than 8. In case of the fifth component, it gives a negative reward of -5 when the agent is stopped which means the velocity is 0 km/h and the leader is farer than a distance of 13 and the action chosen by the agent was lower or equal to 0 which means the agent will break.

5.1.4.2.4 Terminal State

The episode ends when:

1. The episode is running for longer than 60 seconds; or
2. The agent collides; or
3. The distance to the leader is higher than 25.

Mathematically it can be defined as:

$$terminal\ state = \begin{cases} time > 60s, & true \\ len(collision_{events}) > 0, & true \\ distance > 25, & true \\ & false \end{cases} \quad (5.5)$$

5.1.4.2.5 Network Architecture

The network is composed by four layers and can be seen in *Figure 41*:

- Input layer with 3 nodes with ReLU activation function.
- 2 Hidden layers with 64 nodes with ReLU activation function.
- Output layer with 1 nodes and Tanh as the activation function to predict a value between -1 and 1.

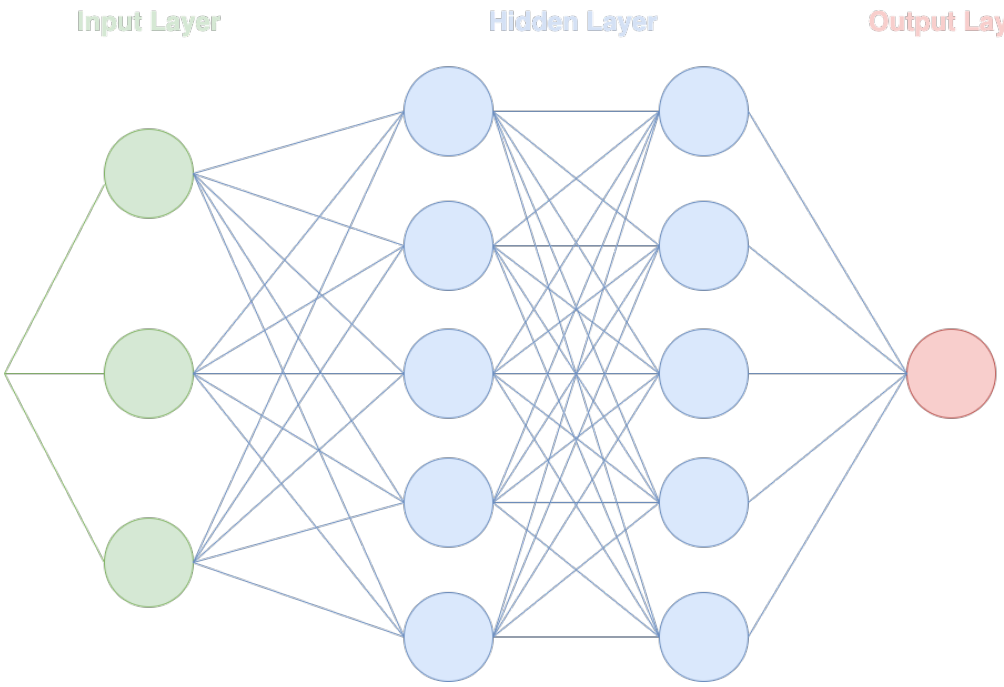

Figure 41 – Throttle/Break Agent Architecture

5.1.4.2.6 Training Results

For the training phase the following hyperparameters were used:

- Batch size: 32
- Epochs: 10
- Gamma: 0.99
- Learning rate: 0.0003
- Policy clip: 0.2
- Episodes: 780
- Standard deviation: 0.55
- Standard deviation decay at each 350 episodes: 0.15

As for the steer agent most of the hyperparameters were not fined tuned, except the standard deviation and the standard deviation decay which were defined through trial-and-error.

Figure 42 shows the evolution of the cumulative reward per episode in light blue and a moving average with a window of 100 episodes in dark blue.

It is visible that around episode 430 a positive drift starts to appear on the reward obtained by episode which means that the agent started learning what was expected. From episode 650 onwards, the cumulative reward has stabilized and, therefore, one can consider that the model has converged, and no more training is needed.

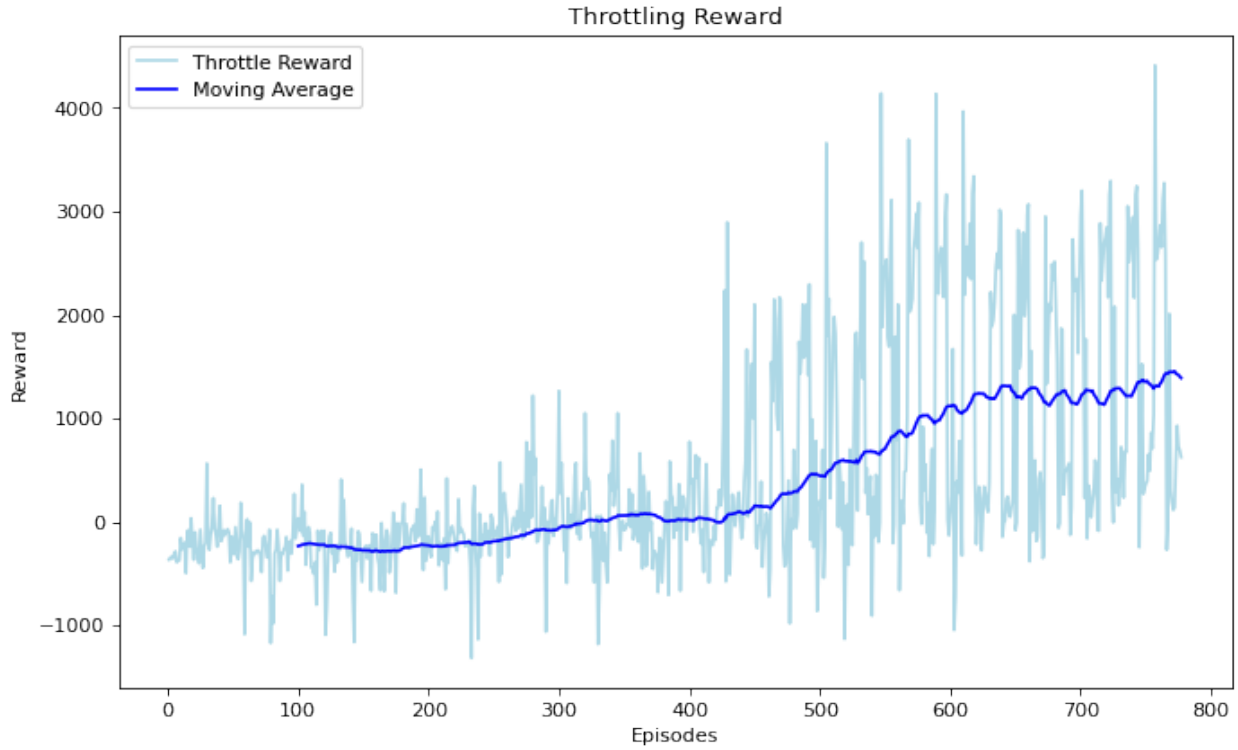

Figure 42 - Cumulative throttle/break reward per episodes (M1)

5.1.4.2.7 Explain Ability

As mentioned previously the explainable part of the thesis will rely on SHAP values to explain why the agent is taking certain decision based on the state perceived.

As shown in *Figure 43* the action taken can be seen in the xx axis and that allows to take the following conclusions:

- The break is mostly used by the agent when the value for the *'velocity'* feature is high and/or the value of *'distance'* is low. It is also possible to see that '*previous distance'* does not have the same importance for the break action as it has for the throttle. However, low values of this feature also contribute to trigger the break. These values are what is expected because a short *'distance'* will require to break to keep up with the desired

distance and, also, when the *'velocity'* is high, so that it is possible to keep a safety distance that allows to break before crashing into the leader.

- The throttle is mainly triggered when the *'velocity'* is low and/or the *'distance'* and/or the *'previous distance'* are high. Such fact makes sense since the agent should keep a distance to the leader between 8 and 10 and therefore if the distance is higher than 10 it must press the throttle.

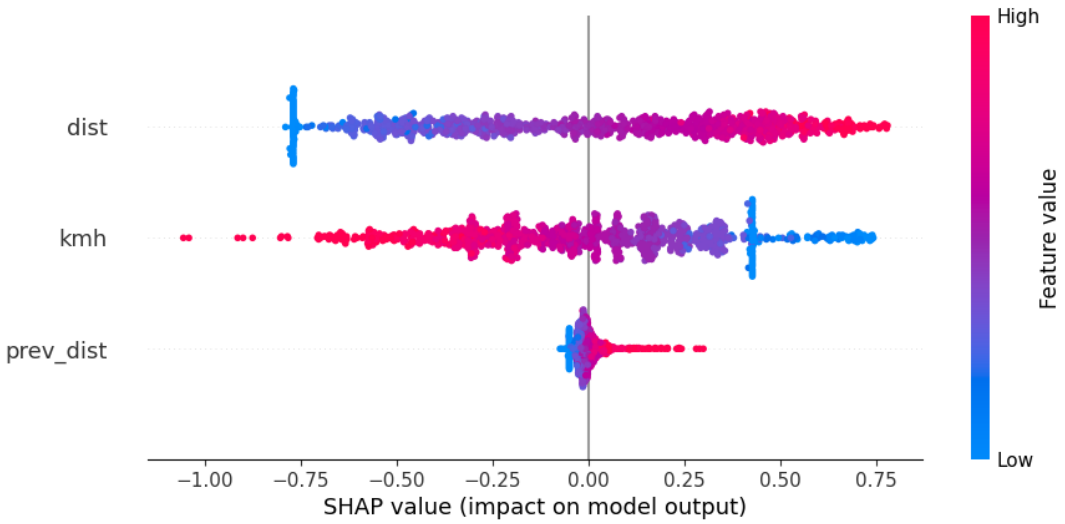

Figure 43 – SHAP values for the throttle/break actions

5.1.4.3 Both Agents

Figure 44 shows an image retrieved by the camera placed in the agent while it was following the leader which a Tesla Model 3.

Figure 44 - Image retrieved by our agent's camera in CARLA

5.1.4.3.1 Training Results

Figure 45 shows the evolution of the cumulative actions per episode in light blue and a moving average with a window of 100 episodes in dark blue.

It is visible that around episode 430 a positive drift starts to appear on the quantity of actions taken by episode which matches the behavior from the cumulative reward obtained from throttle/break agent. This fact could indicate that this agent is more important than the steer one. From episode 650 onwards, the quantity of actions has stabilized, and this behavior matches the behavior for both agents in regards to the convergence condition.

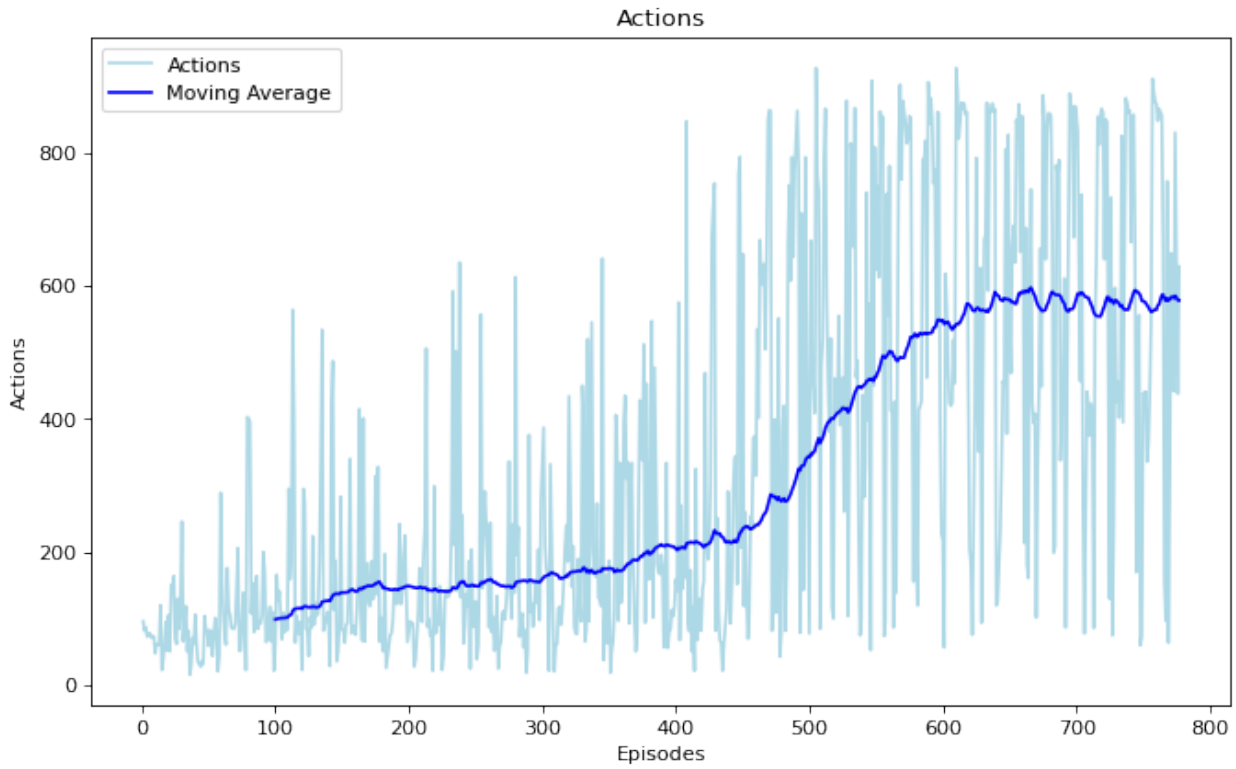

Figure 45 - Quantity of Actions per episode (M4)

5.1.5 Test results

Table 3 shows the results of 20 episodes with a maximum duration of 5 minutes using the trained weights for the three models used (2 PPO models and Xception) to create an agent which is able to follow a leader. The agent was able to complete 20 episodes until the end without crashing or losing sight of the leader, therefore the reason for ending the episode (M8) was only the time.

The average throttle and steer reward across the 20 episodes was 1047.8 and 3547.75, respectively. The average quantity of actions was 465.4 actions with an average distance and velocity of 8.53 and 7.5 km/h, respectively. The low average velocity happens due to red signs from the traffic lights which generates 0 km/h values for a lot of the values being recorded.

Finally, the MAE for Xception was on average 0.2531.

A video from one of the episodes can be accessed here https://youtu.be/NQDdWA7UeZg.

Table 3 - Results of the entire system (2 RL algorithms and Xception) working in CARLA with trained weights. See the definitions of M6-13 in *Metrics*.

Throttle Reward (M6)	**Steer Reward (M7)**	**Successful (M8)**	**Actions (M9)**	**Avg Distance (M10)**	**Avg Velocity (M11)**	**MAE (M13)**
1065	3295	Yes	521	7.62	0	0.2985
1135	3295	Yes	494	7.7	0	0.2841
1145	3345	Yes	456	7.85	0	0.3080
1310	3550	Yes	448	7.92	0	0.2966
1570	4385	Yes	450	8.03	0	0.2771
1435	3785	Yes	462	8.13	0	0.2872
1645	3850	Yes	463	8.17	0	0.2870
1845	4485	Yes	459	8.27	0	0.2971
820	4305	Yes	480	9.84	46	0.2759
970	3955	Yes	483	9.81	40	0.2900
1095	4130	Yes	482	9.69	38	0.2789
1385	3435	Yes	486	9.39	14	0.0291
1615	2145	Yes	472	8.22	0	0.3058
380	4630	Yes	464	8.41	1	0.2767
376	4280	Yes	437	8.74	1	0.3146
400	2490	Yes	433	8.89	1	0.3015
405	2165	Yes	442	8.97	1	0.0294
355	3560	Yes	463	8.76	2	0.0290
1060	2950	Yes	466	8.13	2	0.3147
945	2920	Yes	447	8.13	4	0.2815

Conclusion

As stated above, this dissertation aimed to act in four different aspects: Reinforcement Learning, Computer Vision, Safety and Explain Ability.

Reinforcement Learning is the principal focus, and it was based on Proximal Policy Optimization since it enables to train the agent in a continuous action space, which is crucial for self-driving car tasks with the advantage of having a simpler implementation, better performance, and better sample efficiency than the Deep Q-Learning. Apart from the algorithm choice, the state was selected to be defined through an array of meaningful and controllable variables for explain ability concerns such as distance, velocity, and the relative position to the leader, instead of the traditional image of the environment. Although this algorithm allows a continuous action space, the steer actions were discretized in three possible actions in order to speed up the training and get reliable results faster. However, it would be great to let the agent learn and decide how much it wants to turn the wheel because it would not limit what the agent can do which can be beneficial by allowing the agent to handle easily different situations. The state representation and the reward functions were the most challenging part of this dissertation which took 3 months of experiments to be able to produce the final result.

Computer Vision played an important role on identifying and calculate the distance to the leader using transfer learning on a Xception architecture, and matrix manipulation of segmented images for the state representation. Using this approach, the agent does not need other types of sensors like lidars or radars which make possible to keep driving if one of the other sensors fail. However, this approach is harder, and more time consuming in terms of measuring distances and identifying the different objects in the road than lidars and radars. With one camera there is the need of training an image segmentation model to identify the different objects and then, a regression model to predict the distances to those objects. To train those models, one needs to collect the data (thousands of images) and label them which is always time consuming. Nevertheless, nowadays the simulators provide images that look like real, and they can be used as shortcut for data collection and labeling. Another alternative to be considered is to use GANs to make images from the simulator look more real like what was done in [56].

The safety layer was composed by two parts: one for the steer agent which only allows to turn the wheel 25% in order to avoid sudden turns that can be dangerous. And the other part was

used on the testing phase and it is responsible for breaking in case the distance to the leader is too short.

The Explain Ability part was achieved due to the way the state is represented and using SHAP values which allow to understand how much each variable contributes to the prediction made by the agent. This is useful mainly for the constructor because it allows to verify why the agent is taking certain decision based on the environment perceived, however it is not enough when there are more obstacles in the road or when there are different climate conditions, therefore those features should be added to the state representation.

Future enhancements on this work include: (a) changing the steer agent from a discrete action space to a continuous one with a different state representation where instead of binary features regarding the relative position to the leader one could use how much misaligned the agent is; (b) a more attentive choice of hyperparameters; (c) additionally, identify different obstacles (dynamic and static objects) in the road and collect data regarding their distance to the agent so that it can avoid obstacles and keep driving within the lane; (d) finally, add a identifier to the leader to avoid the agent following other car and enable the point (c) to happen.

References

[1] Abadi, M. et al. 2016. TensorFlow: A system for large-scale machine learning. (May 2016).

[2] Agarap, A.F. 2018. Deep Learning using Rectified Linear Units (ReLU). (Mar. 2018).

[3] Badrinarayanan, V. et al. 2015. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. (Nov. 2015).

[4] Balaji, B. et al. 2019. DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning. (Nov. 2019).

[5] Bird, J.J. and Faria, D.R. 2018. *A Study on CNN Transfer Learning for Image Classification*.

[6] Chollet, F. 2018. *Xception: Deep Learning with Depthwise Separable Convolutions*.

[7] Dalal, G. et al. 2018. Safe Exploration in Continuous Action Spaces. (Jan. 2018).

[8] Dosovitskiy, A. et al. 2017. *CARLA: An Open Urban Driving Simulator*.

[9] Duchi JDUCHI, J. and Singer, Y. 2011. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization * Elad Hazan*.

[10] Eberhard, O. and Zesch, T. 2021. *Effects of Layer Freezing on Transferring a Speech Recognition System to Under-resourced Languages*.

[11] Everitt, T. et al. 2019. Reward Tampering Problems and Solutions in Reinforcement Learning: A Causal Influence Diagram Perspective. (Aug. 2019).

[12] Folkers, A. et al. 2019. Controlling an Autonomous Vehicle with Deep Reinforcement Learning. (Sep. 2019). DOI:https://doi.org/10.1109/IVS.2019.8814124.

[13] Fujimoto, S. et al. 2018. Addressing Function Approximation Error in Actor-Critic Methods. (Feb. 2018).

[14] Grossi, E. and Buscema, M. 2007. Introduction to artificial neural networks. *European Journal of Gastroenterology and Hepatology*.

[15] Ha, D. and Schmidhuber, J. 2018. World Models. (Mar. 2018). DOI:https://doi.org/10.5281/zenodo.1207631.

[16] Haarnoja, T. et al. 2018. Soft Actor-Critic Algorithms and Applications. (Dec. 2018).

[17] Haarnoja, T. et al. 2018. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*.

[18] van Hasselt, H. et al. 2016. *Deep Reinforcement Learning with Double Q-Learning*.

[19] He, K. et al. 2015. Deep Residual Learning for Image Recognition. (Dec. 2015).

[20] https://towardsdatascience.com/shap-explained-the-way-i-wish-someone-explained-it-to-me-ab81cc69ef30: 2020. .

[21] I. Goodfellow et al. 2018. Deep learning. *Genetic Programming and Evolvable Machines*. 19, 1–2 (Jun. 2018), 305–307. DOI:https://doi.org/10.1007/s10710-017-9314-z.

[22] Ioffe, S. and Szegedy, C. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. (Feb. 2015).

[23] Joyce, J.M. 2011. Kullback-Leibler Divergence. *International Encyclopedia of Statistical Science*. M. Lovric, ed. Springer Berlin Heidelberg. 720–722.

[24] Kingma, D.P. and Ba, J. 2014. Adam: A Method for Stochastic Optimization. (Dec. 2014).

[25] Lecun, Y. et al. 2015. Deep learning. *Nature*. Nature Publishing Group.

[26] Lillicrap, T.P. et al. 2015. Continuous control with deep reinforcement learning. (Sep. 2015).

[27] Lin, M. et al. 2013. Network In Network. (Dec. 2013).

[28] Lipton, Z.C. et al. 2016. Combating Reinforcement Learning's Sisyphean Curse with Intrinsic Fear. (Nov. 2016).

[29] Liu, B. et al. 2020. *Platoon control of connected autonomous vehicles: A distributed reinforcement learning method by consensus*.

[30] Loo Vergara, M. 2019. *Accelerating Training of Deep Reinforcement Learning-based Autonomous Driving Agents Through Comparative Study of Agent and Environment Designs*.

[31] Lundberg, S. and Lee, S.-I. 2017. A Unified Approach to Interpreting Model Predictions. (May 2017).

[32] Mnih, V. et al. 2015. Human-level control through deep reinforcement learning. *Nature*. 518, 7540 (Feb. 2015), 529–533. DOI:https://doi.org/10.1038/nature14236.

[33] Mnih, V. et al. 2013. Playing Atari with Deep Reinforcement Learning. (Dec. 2013).

[34] Moerland, T.M. et al. 2020. Model-based Reinforcement Learning: A Survey. (Jun. 2020).

[35] Nwankpa, C. et al. 2018. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. (Nov. 2018).

[36] O'Shea, K. and Nash, R. 2015. An Introduction to Convolutional Neural Networks. (Nov. 2015).

[37] Pascanu, R. et al. 2012. On the difficulty of training Recurrent Neural Networks. (Nov. 2012).

[38] Ronneberger, O. et al. 2015. U-net: Convolutional networks for biomedical image segmentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2015), 234–241.

[39] Ruder, S. 2016. An overview of gradient descent optimization algorithms. (Sep. 2016).

[40] Russakovsky, O. et al. 2014. ImageNet Large Scale Visual Recognition Challenge. (Sep. 2014).

[41] Sallab, A. el et al. 2017. Deep Reinforcement Learning framework for Autonomous Driving. (Apr. 2017). DOI:https://doi.org/10.2352/ISSN.2470-1173.2017.19.AVM-023.

[42] Schrittwieser, J. et al. 2019. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. (Nov. 2019). DOI:https://doi.org/10.1038/s41586-020-03051-4.

[43] Schulman, J. et al. *HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION*.

[44] Schulman, J. et al. 2017. Proximal Policy Optimization Algorithms. (Jul. 2017).

[45] Schulman, J. et al. 2015. Trust Region Policy Optimization. (Feb. 2015).

[46] Sheebaelhamd, Z. et al. 2021. Safe Deep Reinforcement Learning for Multi-Agent Systems with Continuous Action Spaces. (Aug. 2021).

[47] Sifre, L. 2014. *Ecole Polytechnique, CMAP Rigid-Motion Scattering For Image Classification*.

[48] Silver, D. et al. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. (Dec. 2017).

[49] Simonyan, K. and Zisserman, A. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. (Sep. 2014).

[50] Sutton, R.S. and Barto, A.G. 2018. *lanning and Learning with Tabular Methods: Tabular Dyna-Q*.

[51] Sutton, R.S. and Barto, A.G. 2018. *Maximization Bias and Double Learning*.

[52] Sutton, R.S. and Barto, A.G. 2018. *Monte Carlo Tree Search*.

[53] Sutton, R.S. and Barto, A.G. 2018. *Reinforcement Learning An Introduction second edition*.

[54] Thoma, M. 2016. A Survey of Semantic Segmentation. (Feb. 2016).

[55] Wang, Z. et al. 2015. Dueling Network Architectures for Deep Reinforcement Learning. (Nov. 2015).

[56] Xu, W. et al. 2021. *Reliability of GAN Generated Data to Train and Validate Perception Systems for Autonomous Vehicles*.

[57] Y. Bengio et al. 1994. Learning long-term dependencies with gradient descent is difficult. (1994).

[58] Zhou, Y. et al. 2019. *Adaptive Leader-Follower Formation Control and Obstacle Avoidancevia Deep Reinforcement Learni ng*.

[59] Zhou, Z. et al. 2019. A deep learning platooning-based video information-sharing Internet of Things framework for autonomous driving systems. *International Journal of Distributed Sensor Networks*. 15, 11 (Nov. 2019). DOI:https://doi.org/10.1177/1550147719883133.