UNIVERSITY OF SÃO PAULO
SÃO CARLOS SCHOOL OF ENGINEERING

Nilo Freitas de Resende

# A Unified Framework of Deep Reinforcement Learning and Deep Imitation Learning in Simulation Environments

São Carlos

3 December 2018

**Nilo Freitas de Resende**

# A Unified Framework of Deep Reinforcement Learning and Deep Imitation Learning in Simulation Environments

Monograph presented to the Mechatronics Engineering course of the São Carlos School of Engineering of the University of São Paulo, as part of the requirements for obtaining the title of Mechatronics Engineer.

Advisor: Prof. Dr. Glauco Augusto de Paula Caurin

**São Carlos**

**3 December 2018**

## FOLHA DE AVALIAÇÃO

**Candidato: Nilo Freitas de Resende**

Nº 8955618

**Título: An unified framework of Deep Reinforcement Learning and Deep Imitation Learning in simulation environments**

**Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos da Universidade de São Paulo Curso de Engenharia Mecatrônica.**

## BANCA EXAMINADORA

Professor Glauco Caurin
(Orientador)

Nota atribuída: 10,0 ( Dez )

(assinatura)

Professor Gustavo A. P. de Morais

Nota atribuída: 10,0 ( Dez )

(assinatura)

Prof. Vitor C. Guizzilini

Nota atribuída: 10,0 ( Dez )

(assinatura)

Média: 10,0 ( Dez )

Resultado: Aprovado

**Data:** 03 / 12 / 18.

Este trabalho tem condições de ser hospedado no Portal Digital da Biblioteca da EESC

SIM ☒ NÃO ☐ Visto do orientador

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Freitas de Resende, Nilo
F695a A unified framework of deep reinforcement learning and deep imitation learning in simulation environments / Nilo Freitas de Resende; orientador Glauco Augusto de Paula Caurin. São Carlos, 2018.

Monografia (Graduação em Engenharia Mecatrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2018.

1. Deep Learning. 2. Reinforcement Learning. 3. Imitation Learning. 4. Simulation Environments. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

*I dedicate this work to my parents, Bolivar and Marta, for the everlasting support and daily effort that made my studies possible, and which inspires me to be a better person.*

## ACKNOWLEDGEMENTS

First of all, I would like to acknowledge the São Carlos School of Engineering and the University of São Paulo for making this work possible and providing all the knowledge and support to the students, through its people and resources. I thank my advisor Prof. Dr. Glauco Augusto de Paula Caurin, who was always willing to help, understanding my needs and clarifying the path I needed to follow.

I thank Dr. Vitor Campanholo Guizzilini, whom I am gratefully indebted to his passionate participation and support from the beginning of this work until its conclusion, with great patience and confidence in me, sharing inestimable knowledge. Gustavo Prudencio also helped in many moments of this work, with valuable comments and insights. Thanks also to my colleague Gustavo Lahr for helping me with great comments on this work.

This work represents the conclusion of a study that would not have been possible without my family, which I love and have profound gratitude, especially to my parents Bolivar Caetano de Resende, Marta Cristina de Freitas Resende and my sister Ariane Freitas de Resende for the support throughout my entire life.

I thank all my friends from university who I have shared together everyday moments with inestimable companionship. Thanks also to my colleagues from the Mechatronics Laboratory, who have created a friendly group where everyone is willing to help and work with passion.

I would also like to acknowledge all professors and employees from the São Carlos School of Engineering, the Pró-Reitoria de Pesquisa/USP, the library of the EESC/USP and the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

*"When we strive to become better than we are,*
*everything around us becomes better, too."*
*Paulo Coelho, The Alchemist*

# ABSTRACT

RESENDE, N. **A Unified Framework of Deep Reinforcement Learning and Deep Imitation Learning in Simulation Environments**. 3 December 2018. 72p. Monografia (Trabalho de Conclusão de Curso) - São Carlos School of Engineering, University of São Paulo, São Carlos, 3 December 2018.

Reinforcement Learning (RL) and Imitation Learning (IL), in the context of Deep Learning (DL), are powerful approaches for building intelligent agents. In RL, an agent uses signals provided by an environment to learn an optimal behavior in a task. In IL, the agent learns to imitate a demonstrated behavior in a task. Currently, there are many available systems to conduct learning experiments, which provides environments for developing intelligent agents and running experiments. This work aims to develop a scalable open-source software and a single framework to conduct RL and IL experiments, using DL, in several integrated simulation environments, to analyze and compare algorithms with respect to their methods and performance in different tasks. With the developed approach, diverse experiments were conducted with the proposed framework, and an analysis and discussion of the results were made. This monograph exposes, from the developed software and framework, the situation of simulation systems available for research in the field, the advantages of unifying the way to implement, test and analyze these types of algorithms, a consideration of the metrics used to compare performance, and some directions for solving problems encountered in current algorithms.

**Keywords**: Deep Learning. Reinforcement Learning. Imitation Learning. Simulation Environments.

# RESUMO

RESENDE, N. **A Unified Framework of Deep Reinforcement Learning and Deep Imitation Learning in Simulation Environments**. 3 December 2018. 72p. Monografia (Trabalho de Conclusão de Curso) - São Carlos School of Engineering, University of São Paulo, São Carlos, 3 December 2018.

Aprendizado por Reforço (AR) e Aprendizado por Imitação (AI), no contexto de Aprendizado Profundo (AP), são abordagens poderosas para a construção de agentes inteligentes. Na AR, um agente usa sinais fornecidos por um ambiente para aprender um comportamento ótimo em uma tarefa, e na AI, um agente aprende a imitar um comportamento demonstrado em uma tarefa. Atualmente, existem muitos sistemas disponíveis para realizar experimentos de aprendizado, que fornecem ambientes para o desenvolvimento de agentes inteligentes e execução de experimentos. Este trabalho visa desenvolver um software de código aberto escalável e um único framework para conduzir experimentos de RL e IL, usando DL, em vários ambientes de simulação integrados, para analisar e comparar algoritmos com relação a seus métodos e desempenho em diferentes tarefas. Com a abordagem desenvolvida, vários experimentos foram conduzidos com o framework proposto, e uma análise e discussão dos resultados foram feitas. Esta monografia expõe, a partir do software e framework desenvolvido, a situação dos sistemas de simulação disponíveis para pesquisa na área, as vantagens de unificar a maneira de implementar, testar e analisar esses tipos de algoritmos, uma consideração sobre as métricas utilizadas para comparar desempenho, e algumas direções para solucionar problemas encontrados em algoritmos atuais.

**Palavras-chave**: Aprendizado Profundo. Aprendizado por Reforço. Aprendizado por Imitaçao. Ambientes de Simulação.

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| USP | University of São Paulo |
| ML | Machine Learning |
| DL | Deep Learning |
| ANN | Artificial Neural Network |
| MLP | Multilayer Perceptron |
| CNN | Convolutional Neural Network |
| RNN | Recurrent Neural Network |
| RL | Reinforcement Learning |
| DRL | Deep Reinforcement Learning |
| TDL | Temporal Difference Learning |
| MDP | Markov Decision Process |
| DQL | Deep Q Learning |
| A2C | Advantage Actor Critic |
| DDPG | Deep Deterministic Actor Critic |
| PPO | Proximal Policy Optimization |
| IL | Imitation Learning |
| DIL | Deep Imitation Learning |
| BC | Behaviour Cloning |
| IRL | Inverse Reinforcement Learning |
| GAN | Generative Adversarial Network |
| GAIL | Generative Adversarial Imitation Learning |

CONTENTS

# 1 INTRODUCTION

## 1.1 Motivation

Machine Learning (ML) (MURPHY, 2012) is the subfield of Artificial Intelligence (AI) that gives computers the ability to learn. The recent advancements in computing power, infrastructure and algorithms in the world allowed a practical modern use of a sub-class of ML algorithms called Deep Learning (DL) (GOODFELLOW; BENGIO; COURVILLE, 2016). Despite having many challenges, such as interpretability (LIPTON, 2016), DL is a great bet on the development of intelligent systems, with applications in several areas, such as Robotics (ANDRYCHOWICZ et al., 2017) (FINN; LEVINE; ABBEEL, 2016), Natural Language Processing (COLLOBERT; WESTON, 2008), and Health (LITJENS et al., 2017) (MIOTTO et al., 2017).

Within the big picture of building intelligent agents, acquiring skills and controlling actions trough decision making is a great challenge, and in the real world, the environment is usually dynamic, stochastic, continuous and partially observable (RUSSELL, 1998) (FINN; LEVINE; ABBEEL, 2016). With simulation environments, many experiments can be performed in an accelerated time scale, and performance measurements of the algorithms, in different types of tasks, are obtained in a much faster approach than in the real world. Furthermore, simulations allows running safe experiments with easy repeatability, however, with great challenges of reproducibility (HENDERSON et al., 2017). Today, many simulation platforms for this purpose have been created and shared by the ML research community, such as ALE (BELLEMARE et al., 2013) Gym (BROCKMAN et al., 2016) CARLA (DOSOVITSKIY et al., 2017) Unity ML Agents (JULIANI et al., 2018), which are evolving and allowing new and faster approaches to analyze ML algorithms.

Reinforcement Learning (RL) (SUTTON; BARTO, 2018), a class of Machine Learning algorithms, is a powerful framework for controlling dynamic systems and learning tasks (LEVINE; KOLTUN, 2013), which are often formulated as a Markov Decision Processes (MDPs). Despite the difficulties in defining a robust reward function for the learning agent to maximize in more complex applications, recent successful applications of RL in the real world can be cited (GU et al., 2017) (KAHN et al., 2018). The research on RL algorithms using DL and simulation environments has grown a lot since the release of DQN (MNIH et al., 2013) (MNIH et al., 2015). And today, state-of-the-art Deep Reinforcement Learning (DRL) algorithms, such as PPO (SCHULMAN et al., 2017), DDPG (LILLICRAP et al., 2015), A3C (MNIH et al., 2016) and TD3 (FUJIMOTO; HOOF; MEGER, 2018) achieve remarkable performance in simulation environments.

Imitation Learning (IL) (SCHAAL, 1999) on the other hand, considers the problem

of learning a task from demonstrations (HO; ERMON, 2016) (DUAN et al., 2017), since demonstrations are convenient for a human to communicate an intent (DUAN et al., 2017), and in some areas, such as Autonomous Navigation Systems and Assistive Robotics, it is desired to extract knowledge from desired demonstrated behaviour. IL shares many RL concepts, and its applications for building intelligent agents can be developed using a similar structure and using the same platforms for experiments as in RL. Also, Deep Imitation Learning (DIL) algorithms, such as GCL (FINN; LEVINE; ABBEEL, 2016), GAIL (HO; ERMON, 2016), InfoGAIL (LI; SONG; ERMON, 2017) and DGAIL (WANG et al., 2017) also achieve remarkable results.

This work seeks to contribute to the research in RL and IL by proposing a software and framework to develop experiments in diverse simulation environments, as a step towards building intelligent agents with a unified approach and analysis of the algorithm's performances. For this, a scalable open-source software was developed, using Python programming language, the TensorFlow system (ABADI et al., 2016), and the TensorBlock API. Also, an integration of some of the available simulation platforms was made, and some case studies with literature algorithms were analyzed and discussed.

## 1.2 Objectives

The objective of this work is to develop a scalable open-source software and framework to enable Deep Reinforcement Learning and Deep Imitation Learning experiments in diverse integrated simulation environments, to analyze and compare algorithms with respect to its methods and performance in different tasks.

The specific objectives of this work are:

- Present a literature review of RL and IL in the context of DL and simulation environments.
- Develop an scalable software with a single framework to run RL and IL experiments in simulation environments.
- Integrate some of currently available open-source simulation environments in the software.
- Implement in the developed software RL and IL algorithms from the literature.
- Analyze and compare the learning methods and performance of the implemented algorithms in some tasks of the integrated environments.

# 2 LITERATURE REVIEW

This chapter consists of a literary review of this work's related fields. It starts with Machine Learning, followed by Deep Learning, Artificial Neural Networks and Backpropagation. Following, a review of Reinforcement Learning and Imitation Learning with its related methods and algorithms is presented.

## 2.1 Machine Learning

Machine Learning (ML) is the subfield of AI that gives computers the ability to learn without being explicitly programmed. It is the set of methods that can automatically detect patterns in data, and then use those patterns to predict future data, or perform other kinds of uncertainty decision making (such as planning how to collect more data) (MURPHY, 2012). ML is often used to solve regression and classification problems, and has a large number of algorithms, developed from three basic classes:

- Supervised Learning (SL) has a predictive approach in which the goal is to map inputs $x$ to outputs $y$ given a set of labeled training data, represented by $D = \{(x_i, y_i)_{i=1}^{N}\}$.
- Unsupervised Learning (UL) has a descriptive approach in which the aim is to find interesting patterns in the data provided, represented by $D = \{(x_i)_{i=1}^{N}\}$
- Reinforcement Learning (RL) aims to learn a mapping from states to actions when signs of rewards or punishments are applied to an agent interacting with an environment.

There are several other variations of ML classes, such as Semi-Supervised Learning, that are not in the scope of this work.

However, it is worth mentioning Imitation Learning, for being one of the focuses of this work, which is a class of ML that can combine many concepts from three basic classes and introduces an approach to imitate some demonstrated behaviour in an environment. It can, from a dataset of demonstrated behaviour, map states to actions using a SL approach, generate new behaviours using a UL approach and optimize an agent to achieve the optimal behaviour using a RL approach.

### Deep Learning

Deep Learning (DL) is a class of ML algorithms that results in great power and flexibility to represent abstract and high-level characteristics from raw data. It is the hierarchical learning of concepts with each one defined in relation to simpler concepts

(GOODFELLOW; BENGIO; COURVILLE, 2016), which can be understood as a big hierarchical sequence of operations for learning the input data.

The practical modern emergence of this class of algorithms occurred recently, due to the constant advancement in the world's computing power, infrastructure and algorithms, and has obtained great results (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) (MNIH et al., 2015) (GOODFELLOW, 2016). Deep Learning can be combined with the RL and IL algorithms, focus of this work, so that it is possible to solve complex problems.

### Artificial Neural Networks

Artificial Neural Networks (ANNs) are computer systems made up of a number of simple and highly connected processing elements, which process information and respond to external inputs (CAUDILL, 1987). They are inspired by neuroscience and by the learning process from the adaptation of neurons parameters (HEBB, 1949). ANNs can be used as a function approximator, after passing its parameters through an optimization process.

Multilayer Perceptrons (MLPs) are a class of ANNs that have multiple layers of units with a non-linear activation function in each layer, and can use the backpropagation algorithm (RUMELHART; HINTON; WILLIAMS, 1986a) for updating its parameters, resulting in the training of the network with respect to some defined cost function.

Convolutional Neural Networks(CNNs) (LECUN et al., 1998) are a class of ANNs that recently achieved great results in image classification problems (KRIZHEVSKY; SUTSKEVER; HINTON, 2012). They use the convolution operation between layers to abstract filters from an input and extract its characteristics. This is desired in image classification problems and used in this work when a state of the Markov Decision Process (MDP) (Section 2.2) is provided as an image.

Recurrent Neural Networks (RNNs) are a class of ANNs that make a recurrent use of information. They have this denomination because in each layer the output depends not only on the network input but also on the output of each previous layer. The type of Recurrent Neural Networks used in this work is called Long Short Term Memory (LSTM) (HOCHREITER; SCHMIDHUBER, 1997). RNNs are important to capture temporal dependencies in decision making problems, alleviating the problem of modeling Partially Observable Markov Decision Processes (POMDPs) (MONAHAN, 1982) as MDPs.

### Backpropagation

A ML algorithm is trying to produce a function to map inputs $x$ to the correct outputs $y$. The correctness of the output can be measured by a cost function $L$, which is a defined function to predict the cost associated with a certain output $\hat{y}$. So, the ML algorithm can adjust its function's parameters according to the minimization of this cost, and eventually output the correct values.

When using ANNs, the update of the adjustable network parameters is usually performed by following a gradient, calculated from the partial derivatives of the cost function w.r.t the network parameters. Usually, this backpropagation (RUMELHART; HINTON; WILLIAMS, 1986b) of parameter's updates is performed by Gradient Descent methods, such as Adam (KINGMA; BA, 2014), used throughout this work.

A simple example of an ANN with 3 layers each with 1 unit is shown below for demonstration of the backpropagation algorithm.

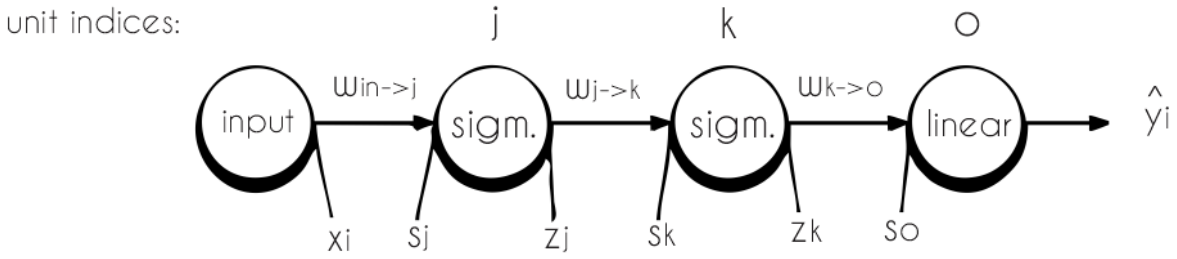


Figure 1: Diagram of an ANN with 4 units

The objective is to adjust the unit's weights $w$ according to some cost function, so that inputs $x$ are mapped to the ground-truth $y$. Each unit has a weight and an activation function (a non-linearity) that is applied to the multiplication of its input and the weight. In this case, the non-linearity is the sigmoid function, represented by:

$$\sigma(s) = \frac{1}{1+e^{-s}} = \frac{1}{1+e^{-w \cdot x}} \tag{2.1}$$

For each unit, there is an input $s$ and an output $z$, that serves as input to the next layer. The next equations shows the forward pass, with all operations to obtain the output $\hat{y}$ from input $x$. This procedure can be summarized by:

$$\begin{aligned}
s_j &= w_{in \to j} \cdot x_i \\
z_j &= \sigma(in_j) = \sigma(w_1 \cdot x_i) \\
s_k &= w_{j \to k} \cdot z_j \\
z_k &= \sigma(in_k) = \sigma(w_2 \cdot \sigma(w_1 \cdot x_i)) \\
s_o &= w_{k \to o} \cdot z_k \\
\hat{y}_i &= in_o = w_3 \cdot \sigma(w_2 \cdot \sigma(w_1 \cdot x_i))
\end{aligned} \tag{2.2}$$

Now, a cost function needs to be defined to measure the error between the ground-truth $y$ and the network output $\hat{y}$. Is this example, the Mean Squared Error cost function

$E$ is used. When the network parameters are successfully adjusted to minimize this cost function, the output of the network is optimal, matching the ground truth value.

$$E = \frac{1}{2}(\hat{y}_i - y_i)^2 = \frac{1}{2}(w_3 \cdot \sigma(w_2 \cdot \sigma(w_1 \cdot x_i)) - y_i)^2 \tag{2.3}$$

The update of the weights to minimize the cost function is calculated by partially deriving the cost function w.r.t each unit. This method is defined as:

$$\begin{aligned}
\frac{\partial E}{\partial w_{k\to o}} &= \frac{\partial \frac{1}{2}(\hat{y}_i - y_i)^2}{\partial w_{k\to o}} \\
&= \frac{\partial \frac{1}{2}(w_{k\to o} \cdot z_k - y_i)^2}{\partial w_{k\to o}} \\
&= (w_{k\to o} \cdot z_k - y_i)\frac{\partial (w_{k\to o} \cdot z_k - y_i)}{\partial w_{k\to o}} \\
&= (\hat{y}_i - y_i)(z_k)
\end{aligned} \tag{2.4}$$

$$\begin{aligned}
\frac{\partial E}{\partial w_{j\to k}} &= \frac{\partial \frac{1}{2}(\hat{y}_i - y_i)^2}{\partial w_{j\to k}} \\
&= (\hat{y}_i - y_i)(\frac{\partial (w_{k\to o} \cdot \sigma(w_{j\to k} \cdot z_j) - y_i)}{\partial w_{j\to k}}) \\
&= (\hat{y}_i - y_i)(w_{k\to o})(\frac{\partial (\sigma(w_{j\to k} \cdot z_j)}{\partial w_{j\to k}}) \\
&= (\hat{y}_i - y_i)(w_{k\to o})(\sigma(s_k)(1 - \sigma(s_k))\frac{\partial (w_{j\to k} \cdot z_j)}{\partial w_{j\to k}}) \\
&= (\hat{y}_i - y_i)(w_{k\to o})(\sigma(s_k)(1 - \sigma(s_k)))(z_j)
\end{aligned} \tag{2.5}$$

$$\begin{aligned}
\frac{\partial E}{\partial w_{i\to j}} &= \frac{\partial \frac{1}{2}(\hat{y}_i - y_i)^2}{\partial w_{i\to j}} \\
&= (\hat{y}_i - y_i)(\frac{\partial (\hat{y}_i - y_i)}{\partial w_{i\to j}}) \\
&= (\hat{y}_i - y_i)(w_{k\to o})(\frac{\partial (\sigma(w_{j\to k} \cdot \sigma(w_{i\to j} \cdot x_i))}{\partial w_{i\to j}}) \\
&= (\hat{y}_i - y_i)(w_{k\to o})(\sigma(s_k)(1 - \sigma(s_k)))(w_{j\to k})(\frac{\partial \sigma(w_{i\to j} \cdot x_i)}{\partial w_{i\to j}}) \\
&= (\hat{y}_i - y_i)(w_{k\to o})(\sigma(s_k)(1 - \sigma(s_k)))(w_{j\to k})(\sigma(s_j)(1 - \sigma(s_j)))(x_i)
\end{aligned} \tag{2.6}$$

The resulting gradients $\frac{\partial E}{\partial w}$ are now multiplied by some learning rate $\alpha$ and added to the weights values. This is one optimization step, usually repeated several times to achieve optimality, i.e., achieving the optimal parameters so that the inputs $x$ are correctly mapped to the $y$ values.

## 2.2 Reinforcement Learning

Reinforcement Learning (RL) is learning what to do-how to map states to actions in an environment to maximize a reward signal provided by an environment (SUTTON; BARTO, 2018). Many successful cases of early RL applications can be cited, such as Tesauro's TD-Gammon (TESAURO, 1995) and Samuel's Checkers Player (SAMUEL, 1959), and also recent ones, such as DQN (MNIH et al., 2015) and AlphaGO (SILVER et al., 2017)

In this class of algorithms, there is an agent that does not receive the information of what to do in an environment, but has to discover which actions return greater reward by trial and error, and find a policy to maximize the expected cumulative reward, interacting with the environment usually defined as a Markov Decision Process (MDP). A RL diagram is illustrated in Fig. (2).

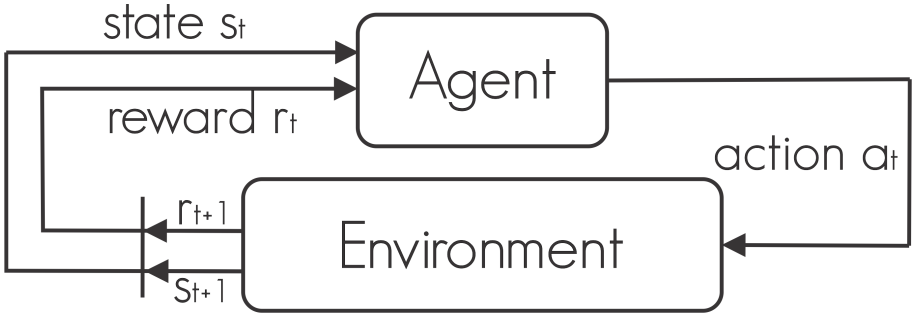


Figure 2: Diagram of Reinforcement Learning

### Markov Decision Process

Markov Decision Process (MDPs) (BELLMAN, 1957) are structures for modeling decision-making problems. They are used in optimization problems solved by RL. MDPs are defined as a tuple $(S, A, T, R, \gamma)$, where:

- S is the set of states $s \in S$, which represents the state of the environment.
- A is the set of actions $a \in A$, available to be taken in the environment.
- T is the transition probability function $T : S \times A \times S \rightarrow [0, 1]$, which defines the probability of transitioning from state $s(t)$ to $s(t + 1)$ after taking an action $a_t$, at time step $t$.
- r is the reward function $r : S \times A \times S \rightarrow \mathbb{R}$, provided by the environment, which represents the immediate reward received by the agent after taking an action $a_t$
- $\gamma$ is a discount factor $\gamma \in [0, 1]$.

The RL agent's behaviour, i.e., its distribution of actions given the states is defined as a policy $\pi$:

$$\pi(s) = P(a|s) \tag{2.7}$$

At each time-step $t$, the agent receives an observation of a state $s(t)$ from the set of states $S$, which usually includes a reward $r(t)$. He then chooses an action $a(t)$ from the set of actions $A$ according to his policy $\pi$, and interacts with the environment. The environment then changes to a new state $s(t+1)$ and a new reward $r(t+1)$ associated with the transition is obtained. The objective is to collect as many rewards as possible in the long run. The discount factor $\gamma \in [0, 1]$ multiplies the rewards at each time-step in order to give greater importance to recent rewards, as shown by:

$$R_t = E_\pi[r] = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.8}$$

In general terms, RL considers the following cost function:

$$argmax_\theta E_{s\sim S, a\sim\pi} R \tag{2.9}$$

### Model Free and Model Based

Model Free is a class of RL algorithms in which the agent does not have access to the information of the environment dynamics, i.e., to the transition function of states P or the explicit reward function R. It must learn an optimal policy for the MDP by using Monte Carlo, Temporal Difference Learning, or other methods discussed in the next sub-sections, to sample episodes and get the MDPs states and rewards needed for the optimization.

Model Based is a class of RL algorithms in which the agent has access to these functions, and this can enable planning to solve the optimization problem, i.e., methods (BROWNE et al., 2012) where the agent can see ahead results of different possible actions, and decide the best ones to take.

### Bellman Equation

The Bellman equation (BELLMAN, 1957) is a recursive equation that can be used in the RL algorithms, so that it is possible to work with the functions $V(s_t)$ and $Q(s_t, a_t)$ recursively.

A state-value function $V(s_t)$ can be defined to describe the expected reward return following the $\pi$ policy from the current state:

$$V_\pi(s_t) = E_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \mid s_t] \tag{2.10}$$

It also can be defined an action-value function $Q(s_t, a_t)$ to describe the expected reward return following the $\pi$ policy from the current state and action:

$$Q_\pi(s_t, a_t) = E_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... \mid s_t, a_t] \tag{2.11}$$

Using the Bellman equation, in the Model Free case, they can be defined as:

$$V_\pi(s_t) = E_\pi\left[r(s_t, a_t) + \gamma\ V_\pi(s_{t+1})\right] \tag{2.12}$$

$$Q_\pi(s_t, a_t) = E_\pi\left[r(s_t, a_t) + \gamma\ Q_\pi(s_{t+1}, a_{t+1})\right] \tag{2.13}$$

**Value Based and Policy Based**

The agent's policy can be learned from Value Based methods, which use functions that estimate the values of being in a state or taking an action, and it is optimized to follow situations of higher value. These methods defines a policy from these value function $V(s)$ or $Q(s, a)$. Examples of Value Based RL algorithms includes Q Learning (WATKINS; DAYAN, 1992), SARSA (SUTTON; BARTO, 2018), Deep Q Network (MNIH et al., 2015).

Policy Based methods represent the policy $\pi(s)$ with parameters $\theta$ and update them directly from the cost function, in the direction of minimization of the cost function. Examples of a Policy Based RL algorithm includes REINFORCE (WILLIAMS, 1992).

In addition to these methods, it is also possible to use Actor-Critic methods, that combine characteristics of the previous ones by learning the policy directly from the objective but also using estimations of learned values to reduce variance of policy gradients. In Actor-Critic methods, a Critic evaluates a state or state-action pair, and this information is used in the cost function of the Actor, to achieve the optimal policy. Examples of Actor Critic RL algorithms includes A3C (MNIH et al., 2016) PPO (SCHULMAN et al., 2017) DDPG (LILLICRAP et al., 2015) and TD3 (FUJIMOTO; HOOF; MEGER, 2018).

**On Policy and Off Policy**

On Policy methods optimze a policy at the same time it follows itself to select the actions, i.e., the experiences for learning are sampled from the policy itself. On Policy RL algorithms includes (WILLIAMS, 1992) (SCHULMAN et al., 2017).

Off Policy methods optimize a policy by observing others, i.e., experiences are sampled from other policies (e.g. the agent's policies in the past). Off policy RL algorithms includes (MNIH et al., 2015) (LILLICRAP et al., 2015).

### Monte Carlo

Monte Carlo methods are means of solving problems based on the mean of sampled returns (SUTTON; BARTO, 2018). Applying Monte Carlo to RL, experiments are sampled by the agent and the $V_\pi(s_t)$ or $Q_\pi(s_t, a_t)$ functions are calculated using the empirical discounted mean return $G_t$ of an entire sequence of episodes, as ddescribed by:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{T-1} r_T \tag{2.14}$$

So therefore the function $V_\pi(s_t)$ can be updated in the direction of the error of this return, as expressed by Equation 2.15. The same holds for $Q_\pi(s_t, a_t)$. The return $G_t$ can also be used in the cost function of Policy Based algorithms, as described by:

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha(G_t - V_\pi(s_t)) \tag{2.15}$$

By using Monte Carlo, unlike Temporal Difference Learning methods, there is a need to wait for the agent to collect entire sequences of experiences before each optimization step, to obtain sampled values. The method produces a great variance due to the sum of the rewards throughout an episode, but we have no statistical bias since the calculation of $G_t$ is sampled and not estimated.

### Temporal Difference Learning

Temporal Difference Learning (TDL) methods are a means of solving problems using estimates that are based on other estimates already calculated, a technique called bootstrapping (SUTTON; BARTO, 2018). Therefore, instead of using the return of rewards $G_t$, which implies waiting for an entire sequence of experience to occur, we use the so-called TD Target $y$ (Equation 2.16), sampled in just one step ahead, which results in low variance.

$$y = r_{t+1} + \gamma V_\pi(s_{t+1}) \tag{2.16}$$

And so, the function $V_\pi(s_t)$ is updated in the direction of the error of this estimate, according to:

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha(r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)) \tag{2.17}$$

Where $(r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t))$ is called the TD Error $\delta_t$. The same holds for $Q_\pi(s_t, a_t)$.

But since we do not have the real value of $V_\pi(s_{t+1})$ when we calculate $V_\pi(s_t)$, it is estimated, and therefore a statistical bias is introduced in the calculation of $V_\pi(s_t)$. Another related technique, called Temporal Difference Lambda ($\lambda$), uses samples in more time steps than 1.

### Reinforcement Learning algorithms

This section consists of a literature review of the Reinforcement Learning algorithms used in the Materials and Methods section of this work. Table 1 shows a summary of the algorithms and its characteristics.

| Algorithm | Model Free | Model Based | On Policy | Off Policy | Value Based Only | Policy Based Only | Actor Critic | Monte Carlo Based | TDL Based |
|---|---|---|---|---|---|---|---|---|---|
| DQL | ✓ | | | ✓ | ✓ | | | | ✓ |
| REINFORCE | ✓ | | ✓ | | | ✓ | | ✓ | |
| A2C | ✓ | | | ✓ | | | ✓ | | ✓ |
| DDPG | ✓ | | | ✓ | | | ✓ | | ✓ |
| PPO | ✓ | | ✓ | | | | ✓ | ✓ | |

Table 1: Summary of RL algorithms characteristics

### Deep Q Learning

Q Learning (WATKINS, 1989) is an Off Policy RL algorithm, based on Temporal Difference Learning, in which an agent seeks to learn a function that maps the values of being in a state and taking action in an environment to maximize the sum of future rewards. In this model, the policy of an agent therefore comes from the action-value function $Q_\pi(s, a)$ and its optimal policy comes by following the optimal action-value function, given by:

$$Q^*_\pi(s_t, a_t) = \max_\pi E\left[r_t + \gamma\ r_{t+1} + \gamma^2 r_{t+2} + ...|s_t, a_t, \pi\right] \tag{2.18}$$

This function is optimal with respect to the largest sum of future rewards obtained by each action-state pair from the policy $\pi$. Using the Bellman Equation, we can modify the previous equation to:

$$Q^*_\pi(s_t, a_t) = E\left[r_t + \gamma \max_{a_{t+1}} Q^*_\pi(s_{t+1}, a_{t+1})\right] \tag{2.19}$$

In this algorithm, the policy is implicit in the action-value function, and therefore an arbitrary policy is defined, using this function. The simplest viable one is $\epsilon - greedy$

policy, which consists in selecting the action that has the highest value $Q^*_\pi(s,a)$, with a percentage $\epsilon$ of a chance of randomly selecting an action.

The update gradients of the function parameters $Q_\pi(s,a)$ are derived from the Mean Squared Error of TD Error, according to:

$$L(\theta) = \left[\left(r_t + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1} \mid \theta_t)\right) - Q_\pi(s_t, a_t \mid \theta_t)\right]^2 \tag{2.20}$$

Such an algorithm is known to work well in environments that have large a state space and small discrete action space. The use of ANNs to approximate the action-value function works well for many cases, despite being a method known to be unstable when combined with this algorithm. Some stabilization techniques are applied to solve this issue of using DL combined with this algorithm.

Deep Q Network (MNIH et al., 2013) (MNIH et al., 2015) improved the algorithm using several techniques to stabilize learning. It used a method called Experience Replay, which is characterized by creating a buffer to randomly select experiences batches. Also, an additional action-value function $Q'_\pi(s,a)$ is added, represented by another ANN, named Target. Moreover, a mechanism to copy the parameters of the main network is used, to make updates at a constant time rate, to reduce noise in the calculation of $Q^*_\pi(s,a)$ values in the update in detriment of learning speed.

Using Deep Learning, it was also possible, in Deep Q Network, to use a CNN to abstract the characteristics of raw image frames from various Atari games, which were the chosen environments in which the agent interacts (MNIH et al., 2015).

**REINFORCE**

The pioneer Policy Based RL algorithm REINFORCE (WILLIAMS, 1992) is a Monte Carlo-based On Policy RL algorithm, which seeks to optimize the policy directly with respect to the cost function. So it does not suffer from issues such as uncertainty of state's values, but suffer from other problems such as the possibility of being optimized to a local maximum.

The agent's actions come from the probability distribution returned by the stochastic policy $\pi$, with parameters $\theta$. In Policy Gradients, we want to update the policy directly towards the highest reward, so our cost function implies gradients given by:

$$\nabla_\theta L(\theta) = \nabla_\theta E_{\pi_\theta}[r(s)] \tag{2.21}$$

It is important to notice that Equation 2.21 does not depend on the $\theta$ parameters of the $\pi_\theta$ function. Therefore, a way to get the gradients with respect to the parameters, to successfully update the function, is using the Likelihood Ratio method.

The most common type of Policy Gradients algorithms is based on the Likelihood Ratio method, which seeks to increase the likelihood of returning the expected goal, and which uses the artifice of using logarithms to facilitate implementation, preserving the result (SUTTON et al., 2000), as it is shown in:

$$\begin{aligned}
\nabla_\theta E_{\pi_\theta}[r(s)] &= \nabla_\theta \sum_s \pi_\theta(s,a) r(s) \\
&= \sum_s \nabla_\theta \pi_\theta(s,a) r(s) \\
&= \sum_s \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} r(s) \\
&= \sum_s \pi(s,a) \nabla_\theta \log(\pi_\theta(s,a)) r(s) \\
&= E_{\pi_\theta}[r(s) \nabla_\theta \log(\pi_\theta(s,a))]
\end{aligned} \tag{2.22}$$

In Equation 2.22 a single one time-step reward $r(s)$ was used, but the relation is valid on any time scale (SUTTON et al., 2000). So we can calculate the update of the stochastic policy gradient using the Equation 2.22 and Monte Carlo:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta log \pi_{\theta_t}(s_t, a_t) \tag{2.23}$$

Where $G_t$ is the Monte Carlo return, i.e., discounted sampled return of rewards from the episode. It is proved in (WILLIAMS, 1992) that we can use several alternatives to $G_t$, to reduce the variance, with the addition of some base function without changing the direction of the gradient.

### Advantage Actor Critic

It is proved in (WILLIAMS, 1992), that we can reduce the variance of the policy gradient in REINFORCE by subtracting a base variable $b(s)$ in the calculation of the cost function $L(\theta)$. Therefore, the policy cost function (Equation 2.22) can have the term $r(s)$ replaced by $r(s) - b(s)$, with $b(s)$ being some value of choice, which may be for example $V_\pi(s)$.

The Advantage Actor Critic (A2C) algorithm is based on this method. Here, the function representing the policy $\pi$ with parameters $\theta$ is called Actor. And the value function $V_\pi(s)$ or $Q_\pi(s,a)$, with parameters $\zeta$ is called Critic. A Critical Actor Advantage diagram is shown in Fig.(3):

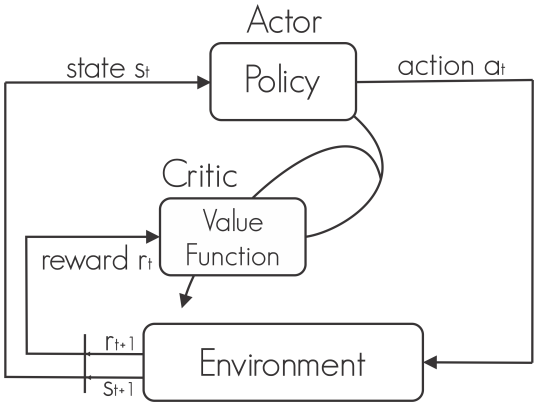


Figure 3: Diagram of Advantage Actor Critic algorithms

In this algorithm, the optimization of the Actor's policy $\pi$ is done using the estimation of the values $V_\pi(s)$ or $Q_\pi(s,a)$ by the Critic. A2C defines the term Advantage to replace $r(s) - b(s)$ in the calculation of the cost function, as summarized by:

$$A(s) = Q_\pi(s,a) - V_\pi(s) \tag{2.24}$$

Thus, instead of subtracting a base function $V_\pi(s)$ in the Monte Carlo return of an On Policy Policy Gradients algorithm, an Off Policy algorithm is created (SUTTON; BARTO, 2018), in which the rewards are calculated by Temporal Difference Learning. It also uses experiences from an Experience Replay, and therefore the Advantage used in the calculation of the Actor's cost is the TD Error, represented by:

$$A(s) = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \tag{2.25}$$

which is a non-biased sample of Equation 2.24. As in REINFORCE, the Actor cost is represented by the Likelihood Ratio, but now using the Advantage, as shown by:

$$L(\theta) = A(s)\ \nabla_\theta\ log\ \pi_{\theta_t}(s_t, a_t) \tag{2.26}$$

The Critic cost is the Mean Squared Error of the TD Error, represented by:

$$L(\zeta) = (r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t))^2 \tag{2.27}$$

### Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradients (DDPG) (LILLICRAP et al., 2015) is a RL algorithm based on Deterministic Policy Gradients (SILVER et al., 2014), that has an Actor Critic architecture and can successfully operate in continuous high-dimensional

action spaces. This domain is of great importance in this work because it is characteristic of several robotic and real-world applications. The algorithm is Off Policy based on Temporal Difference Learning and also uses an Experience Replay buffer. As a result, it closely resembles the algorithm of the previous section (3.3).

However, to enable the algorithm to operate in continuous high-dimensional environments, there is a substitution in the policy update procedure $\pi$, not using the Likelihood Ratio policy gradients (2.26) to update the Actor. Determinitic Policy Gradients (SILVER et al., 2014) prove that it is possible to use the Critic gradients themselves to update the Actor, as expressed by:

$$L(\zeta) = (r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t))^2 \tag{2.28}$$

This cost calculation refers to the name Deterministic Policy Gradients, in which the Likelihood Ratio stochastic method is not used. In order to stabilize the algorithm, a L2-Regularization (NG, 2004) is added at the cost of the Critic, a smoothing technique that encourages the sum of the squares of the network parameters to be small to prevent overfitting.

In addition, the calculation of the Critic values and the Actor actions is done with auxiliary target ANNs, aiming stabilization, with a technique similar to that described in Section 3.3. Such auxiliary networks are copies of Actor and Critic, which have their parameters copied from the main networks not at a constant time rate as described in Section 3.3, but in a soft manner, at all times-steps, with an arbitrary parameter $\tau$, as shown in Equation 2.29:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{2.29}$$

As we have an algorithm that operates in high-dimensional continuous environments, we must address the issue of action exploration. To ensure the exploration of the agent in the environment, the Uhlenbeck and Ornstein process (LILLICRAP et al., 2015) is used, which is a stochastic process of generating values with the property of returning to a chosen mean over time. The value generated in this process is used to create a noise $N$ added in the calculation of the action by the Actor.

### Proximal Policy Optimization

Proximal Policy Optimization (PPO) (SCHULMAN et al., 2017) is a family of Policy Based Reinforcement Learning algorithms that alternate between taking samples of the environment and performing various optimizations of a surrogate cost function using an Importance Sampling estimator (SCHULMAN et al., 2015a). A PPO algorithm can be

used to solve a number of discrete and continuous problems in high-dimensional state and action spaces with a simple implementation similar to REINFORCE (Section 3.3).

The algorithm is classified as On Policy (despite using Importance Sampling), and it has an Actor-Critic architecture, and uses the Generalized Advantage Estimation (SCHULMAN et al., 2015b) method to calculate the Advantage. This is a more robust method based on Temporal Difference ($\lambda$), which has Monte Carlo and Temporal Difference characteristics. The Generalized Advantage Estimation is described by:

$$\begin{gathered} A = \sum_{l=0}^{\infty} (\gamma\lambda)^l \ \delta_{t+l}^V \\ with \ \delta = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \end{gathered} \tag{2.30}$$

The Critic, Actor, and a copy of the Actor named Old Actor are defined. The Critic provides the state value function $V(s_t)$ for the Advantage calculation, while the Actor provides the $\pi$ policy, while the Old Actor preserves the Actor parameters before the update step, being used in the calculation of the surrugate cost function. The cost function, obtained with Importance Sampling, is the ratio of the probabilities of the actions, given by the Actor and the Old Actor, multiplied by Advantage and clipped by a $\epsilon$ variable, as shown in:

$$L(\theta) = E_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t \right] = E_t \left[ ratio_t(\theta) A_t \right] \tag{2.31}$$

$$L(\theta)_{clipped} = E_t \left[ min(ratio_t(\theta)A_t; clip(ratio_t(\theta), 1-\epsilon, 1+\epsilon)A_t) \right] \tag{2.32}$$

The Old Actor does not have its parameters updated by the optimizer, but rather has its parameters copied from the Actor before each sequence of updates. The Critic update uses the Equation 2.27 as cost. The policy $\pi$, unlike what has been presented so far, does not directly return the agent's actions, but rather a mean and a standard deviation, which forms a distribution from where the action is sampled. This guarantees the issue of exploration of actions.

## 2.3 Imitation Learning

Imitation Learning (IL), sometimes referred as Learning From Demonstrations (a broader term), considers the problem of learning a task from demonstrated trajectories to imitate a policy (HO; ERMON, 2016) (DUAN et al., 2017). Some surveys includes (BILLARD et al., 2008) (SCHAAL, 1999) (HUSSEIN et al., 2017). The advantage of this approach is that demonstrations are extremely convenient for communicating the intent of the task (DUAN et al., 2017), allowing the learning of tasks from the imitation of

demonstrated behavior, an intuitive approach for humans to guide the machine to the desired objective.

Some authors (DUAN et al., 2017) (STADIE; ABBEEL; SUTSKEVER, 2017) (NAIR et al., 2018) (LIU et al., 2018) (HAUSMAN et al., 2017) consider the two main lines of this area as Behavioral Cloning and Inverse Reinforcement Learning. While others (HO; ERMON, 2016) (LI; SONG; ERMON, 2017) divide the two main lines as Behaviour Cloning and Apprenticeship Learning. While there is some confusion about the terms, in this work the Behavior Cloning and Inverse Reinforcement Learning consideration of main lines is used.

In general terms, IL considers the following cost function:

$$argmin_\theta E_{s\sim P(s|a)} L(\pi^*(s), \pi_\theta(s)) \tag{2.33}$$

where it tries to update parameters $\theta$ of a policy $\pi$ to minimize a cost function $L$ defined by a relation between an exemplary policy $\pi^*(s)$ and the learning policy $\pi_\theta(s)$.

**Behaviour Cloning**

Behavioral Cloning seeks to learn the optimal policy directly from observed trajectories, with a Supervised Learning approach, to learn the state mapping function for actions (ROSS; GORDON; BAGNELL, 2011) (DAUMÉ; LANGFORD; MARCU, 2009) (ROSS; BAGNELL, 2010) (LASKEY et al., 2017).

Behavioral Cloning, despite having achieved remarkable successes, has known limitations: it suffers from compounding errors, which arise in training due to small cumulative errors that distract the agent from the distribution of states seen in the demonstration and result in observations different from those shown (FINN et al., 2016). This results in an agent that does not know how to act satisfactorily in non-demonstrated states, which can be understood as a problem of Covariate Shift (ROSS; GORDON; BAGNELL, 2011) (WANG et al., 2017) (HO; ERMON, 2016).

These errors occur due to the fact that Behavioral Cloning assumes that the distributions are Independent and Identically Distributed (IID) (BARAM et al., 2017) and therefore does not consider that the state distributions $P$ depend on the agent's policy ($P(s, \theta)$) (ROSS; GORDON; BAGNELL, 2011) (WANG et al., 2017) (HO; ERMON, 2016). Using RL with a MDP structure this state information dynamic sequential data is taken into consideration (SUTTON, 1988) (SUTTON; BARTO, 2018). In addition, the approach fails when demonstrated data is scarce.

In general terms, Behavioral Cloning considers the following cost function:

$$argmin_\theta E_{(s,a^*)\sim P^*} L(a^*, \pi_\theta(s)) \tag{2.34}$$

where it tries to update parameters $\theta$ of a policy $\pi$ to minimize a cost function $L$ defined by a relation between an exemplary demonstrated behaviour $a^*$ and the learning policy $\pi_\theta(s)$.

**Inverse Reinforcement Learning**

Inverse Reinforcement Learning (IRL) considers the problem of observing demonstrations of a behavior, extracting a reward function, and learning a similar behavior from this reward function (NG; RUSSELL et al., 2000). This method also considers the case where a specification of the MDP is available, but the reward structure is unknown. The objective is to learn the implicit reward function that explains the behavior in the trajectories $\tau = (s_0, a_0, s_1, a_1, ...)$. However, learning the reward function does not tell the agent how to act directly on the environment (the policy $\pi$), and so often a RL algorithm needs to be executed next to learn the policy (HO; ERMON, 2016).

The area of IRL was developed using most of the concepts of RL. The traditional algorithms in this area (RUSSELL, 1998) (NG; RUSSELL et al., 2000) (ABBEEL; NG, 2004) (ZIEBART et al., 2008) seek to learn optimal policy in a two-step loop: IRL for learning the reward function and RL for learning the optimal policy w.r.t. to the learned reward function. A few recent algorithms proposes new approaches to learning the policy and the reward directly, bypassing a separated IRL step (HO; ERMON, 2016) (WANG et al., 2017) (LI; SONG; ERMON, 2017).

It is important to note that, fundamentally, IRL considers an inverse problem (not well-posed), which means that several rewards functions can explain the behavior of the trajectories collected. To alleviate this problem, many algorithms (ABBEEL; NG, 2004) (ZIEBART et al., 2008) (HO; ERMON, 2016) do a relaxation of the optimization problem to reduce it to a problem of feature matching the distributions.

In general terms, IRL considers the following cost function:

$$argmax_\theta E_{s\sim P(s|a)} r(s, \pi_\theta(s)), r = f(\tau) \tag{2.35}$$

where it tries to update parameters $\theta$ of a policy $\pi$ to maximize a reward function that is recovered from an exemplary demonstrated behaviour $\tau$.

### Imitation Learning algorithms

This section consists of a literature review of the Imitation Learning algorithms used in the Materials and Methods section of this work.

### DAgger

Data Aggregation (DAgger) (ROSS; GORDON; BAGNELL, 2011) is a Behavioral Cloning algorithm and uses a set of trajectories $\tau = \{s, a\}$ to imitate the policy of the demonstrator, where $s$ are states and $a$ the actions performed in that state by the demonstrator. The algorithm first trains a policy that imitates the demonstrator using Supervised Learning, where the inputs of the ANNs are the states collected, and the labels are the actions collected.

To alleviate the Covariate Shift problem, the algorithm aggregates the states seen in training to the dataset to be imitated, and then trains a policy using that aggregate dataset s $\tau = \tau + \tau_i$, where $\tau_i = \{s, p_i(s)\}$.

The intuition is to train the policy under the aggregate dataset at each iteration so that states not seen in the original statements can be considered in training and thus managing to recover from errors created by finding states not seen in the original dataset. Its cost function is based on the sequence of losses described by the following equation:

$$L_m(\theta) = E_{s \sim P(m)} L(\pi^*_\theta(s), \pi_\theta(s)) \tag{2.36}$$

### Generative Adversarial Imitation Learning

Generative Adversarial Imitation Learning (GAIL) is an algorithm similar to IRL algorithms due to the learning of a reward function from demonstrated trajectories to achieve the objective of learning a policy. It is inspired by a Non-Supervised Learning algorithm called Generative Adversarial Networks (GANs) (GOODFELLOW et al., 2014), explained below.

### Generative Adversarial Networks

Generative Adversarial Networks (GANs) (GOODFELLOW et al., 2014) are a class of generative models that allow the generation of realistic samples from some distribution (GOODFELLOW, 2016). In the algorithm, a game is formulated between a $G$ generator and a $D$ discriminator.

To learn the distribution of the $p_g$ generator on a dataset $x$, a noise variable $z$, mapped to the space of the dataset $x$, is defined as input, with that mapping defined as $G(z; \theta_g)$. The discriminator $D(x; \theta_d)$ produces a single scalar representing the probability that $x$ came from the data instead of $p_g$. The Discriminator $D(x; \theta_d)$ is trained to maximize

the probability of assigning the correct label to the training samples $x$ and the Generator $G(z;\theta_g)$ samples. This cost function of the discriminator is the entropy-cross function between the two distributions, shown in the following equation:

$$J(D) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.37}$$

The generator $G$ is simultaneously trained so that the discriminator assigns the same label to the generated data and the data of the dataset, that is, to maximize the opposite of the equation 2.37, which implies minimizing $J(D)$ . By heuristic, only the second part of the equation is used because it achieves better results. The cost function of the Generator is therefore given by:

$$J(G) = E_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.38}$$

The optimality condition is achieved when the distance between these two distributions is minimized as measured by the Jensen-Shannon divergence (GOODFELLOW et al., 2014). In other words, $D$ and $G$ play the following two-player minimax game with the joint cost function $V(G, D)$:

$$\begin{aligned} &\min_G \max_D V(D, G) = \\ &E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))] \end{aligned} \tag{2.39}$$

A diagram of the GAN structure is given below:

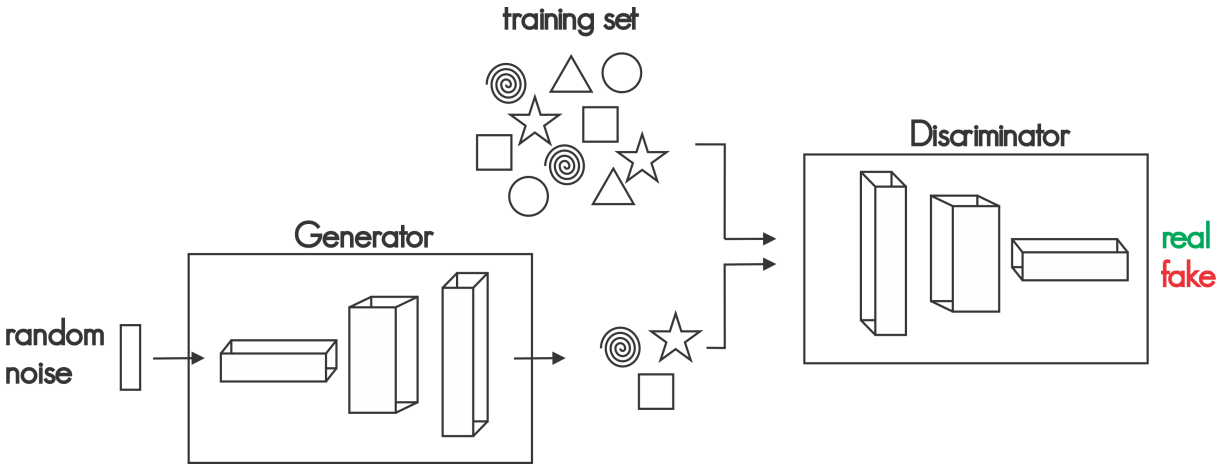


Figure 4: Diagram of Generative Adversarial Networks

### Generative Adversarial Imitation Learning

In the Generative Adversarial Imitation Learning (GAIL) algorithm proposed by (HO; ERMON, 2016), the agent imitates the behavior of a demonstrator policy $\pi_E$ from a collected dataset collected of state-action pairs trajectories $\tau = (s_0, a_0, s_1, a_1, ...)$. Similar to GANs, the agent's policy $\pi$ acts as a generator $G(s, \theta_g)$ that generates an action $a$. The discriminator $D(\tau, \theta_d)$ is a classifier of the form $D : S \times A \rightarrow (0, 1)$.

GAIL considers the environment as a black box and therefore the cost function is not differentiable from end to end. Consequently, it requires Monte Carlo-based estimation algorithms to calculate the policy gradients (LI; SONG; ERMON, 2017).

The structure of this algorithm has a great similarity with the PPO algorithm, so the policy update is done with the same policy cost function in PPO, with the fact that the rewards are not provided by the environment but by the Discriminator $D$, and they are defined following the equation below:

$$r = -\log(1 - \sigma(D(s, a|w))) \tag{2.40}$$

The Discriminator D with parameters $w$ is updated according to the following cost function:

$$L(w) = E_{\tau_b}[\log D(s, a|w)] + E_{\tau_E}[\log(1 - D(s, a|w)] \tag{2.41}$$

Finally, the joint cost function of GAIL is defined as:

$$\begin{aligned} &\min \pi \max_{D \in (0,1)^{S \times A}} V(D, \pi) = \\ &E_\pi[\log D(s, a)] + E_{\pi_E}[\log(1 - D(s, a))] - \lambda H(\pi) \end{aligned} \tag{2.42}$$

Where $H(\pi)$ is an entropy function. A diagram of a simplified GAIL structure is shown in Figure 5.

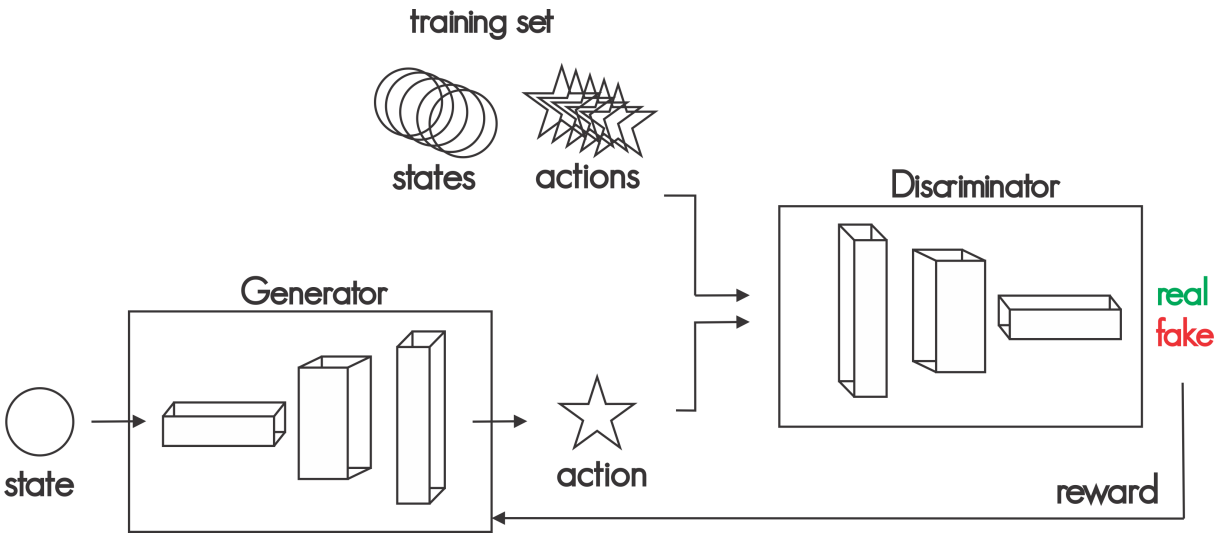


Figure 5: Diagram of Generative Adversarial Imitation Learning

The original article develops this algorithm assuming that the trajectories demonstrated are optimal in relation to a reward function, and that they are provided by a single demonstrator (LI; SONG; ERMON, 2017).

### GAIL Multi-rewards

GAIL uses trajectories demonstrated by a single agent, and with a single reward function implicit in the trajectories provided by the demonstrator (LI; SONG; ERMON, 2017). Such an assumption is a limiting factor for the learning of tasks that have more than one reward function which the agent has to maximize. Even if the GAIL algorithm is able to learn an average policy that explains the distribution of demonstrated trajectories, this policy does not accurately represent the desired behavior. In order to solve this problem, some algorithms (RANCHOD; ROSMAN; KONIDARIS, 2015) (CHOI; KIM, 2012) (ŠOŠIĆ et al., 2018) have created an approach based on segmenting the demonstrated trajectories with Unsupervised Learning algorithms and then learning the behaviors of the segmented agents or reward functions with IL.

Segmenting trajectories is a Clustering problem, sub-field of Unsupervised Learning. In Clustering, a cluster is a collection of similar data, and it is common to define the number of clusters in which we need to divide the data. Defining the number of clusters for skill segmentation in the demonstrated trajectories when the number of different abilities is unknown is a challenge. Therefore, the mentioned algorithms that use this approach focus on using Clustering algorithms based on the Bayesian Non-Parametric theory (MICHINI; HOW, 2012), in which the number of clusters is automatically defined during the execution of the algorithm, from Dirichlet processes (GHAHRAMANI, 2005). More recent approaches attempt to segment the trajectories with the use of ANNs (HAUSMAN et al., 2017).

### GAIL Sub-optimal demonstrations

GAIL assumes that the trajectories demonstrated are optimal in relation to a reward function (HO; ERMON, 2016) (LI; SONG; ERMON, 2017). This means that the agent, given a state, always chooses the best action based on its implicit reward function. This assumption is a challenge when the demonstrator performs unstable noise movements, resulting in an unclear reward function to be recovered by the IL algorithm. This is a problem of sub-optimal demonstrations, where several state-action pairs in the trajectories demonstrated are not compatible with the reward function or reward functions implicit in the trajectories demonstrated. The available literature for this problem is scarce and few approaches that focus on this problem can be cited (CHANG et al., 2015) (ROSS; BAGNELL, 2014) (SONG et al., 2018). However, these approaches seeks to bypass the sub-optimality, removing the noise and learning the optimal task from the sub-optimal demonstrations.

# 3 MATERIALS AND METHODS

This chapter consists of a description of the technology used to developed the technical part of this work and the specific ML methods used.

## 3.1 Tools

Many programming languages can be used to implement ML and DL algorithms. In the ML community, the Python (FOUNDATION, 2018) programming language is a highly accepted language that has many libraries to facilitate implementations. The open-source system build in this work uses the Python programming language, and currently the dependencies TensorFlow, PyGame, OpenCV-Python, NumPy, Gym, MatplotLib, Unity ML, MuJoCo, CARLA. The developed open-source software is under continuous improvements and available in the author's GitHub repository (RESENDE, 2018), with the complete documentation.

### TensorFlow

TensorFlow (ABADI et al., 2016) is a system that allows the user to develop ML applications in high scale, with functions that facilitate the implementation of algorithms with computational efficiency. In TensorFlow, numerical computations are performed in graphs, where the vertices represent mathematical operations and the edges represent the multidimensional arrays, called tensors, that are communicated between the nodes. A graph of an algorithm implemented in TensorFlow can be visualized in Fig. (6). It can take advantage of the Graphics Processing Unit (GPUs) of the computer to perform parallel processing of the ANN operations used in DL algorithms.

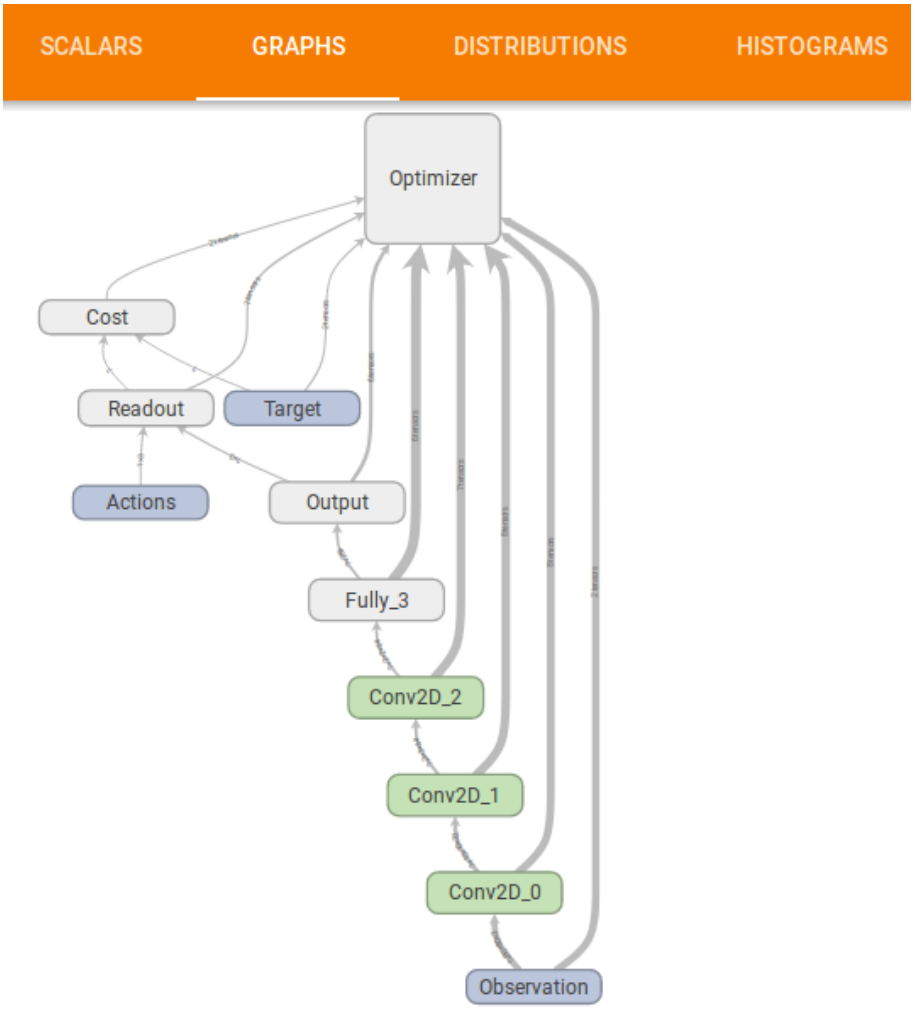


Figure 6: Graph of an algorithm implemented using TensorFlow, provided by the TensorBoard tool

### A software using TensorBlock

TensorBlock is an API for TensorFlow, created by the collaborator Dr. Vitor Guizzilini and maintained by the author, which aims to facilitate the implementation of Machine Learning algorithms with TensorFlow, allowing easy definition of TensorFlow graphs and execution of operations. A diagram illustrating a structure of a software using the TensorBlock API is shown in Fig. (7).

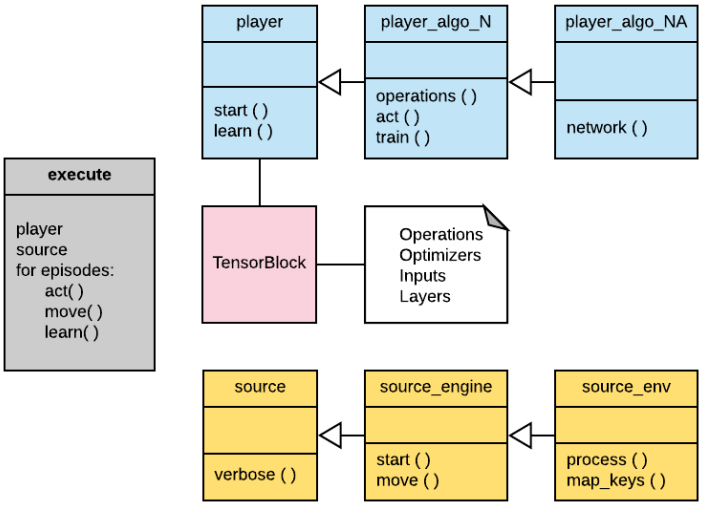


Figure 7: Diagram of a software with TensorBlock API

The blue boxes (player) refers to the class of the learning algorithm, and have the methods that allow the definition of TensorFlow operations and placeholders, and execution of these operations in the graph. The yellow boxes (source) refers to the class that have the methods to communicate with the simulation environment, acting in the environment and getting the states and rewards. The pink box is the TensorBlock API, that has the classes and methods that defines and run TensorFlow graphs, and where the user can define new methods to be incorporated by the API such as other cost functions or network layers.

Currently, the usage of the developed software is performed by using the following command to execute an algorithm and train an agent in a simulation environment.

```
python execute.py type_of_algo source_your_source player_your_player
```

And there are some flags that can be used:

| | |
|---|---|
| - - save your_saved_model_name | to save the TensorFlow model |
| - - load your_saved_model_name | to load the TensorFlow model |
| - - run | to just run and do not train |
| - - epis | to specify a max. number of episodes to execute |
| - - record | to save states and actions seen |

The full usage description is in the GitHub repository (RESENDE, 2018).

## 3.2 Simulation Environments

Simulation environments provide us with various types of states $s$, rewards $r$ and other information used by a learning algorithm. An analysis of how well a learning algorithm can perform in different types of environment is desired to measure the algorithm's performance in comparison with others. In this section, it is shown each simulation software/engine/system that was integrated in this work's software.

There are several softwares of simulation platforms with environments available for research of learning algorithms. Each environment has its unique nature, restrictions, and usually tasks with some desired behaviour to be performed. Although several of the available environments can have characteristics of Partially Observable Markov Decision Processes (POMDPs) (MONAHAN, 1982), or even Multi-Agent Markov Decision Processes (MAMDPs) (BOUTILIER, 1996), they are usually built to attend the MDP framework.

Some of the existing available systems that allows these simulations are: Arcade Learning Environment (BELLEMARE et al., 2013), a platform with over 70 Atari games written in C++; ViZDoom (KEMPKA et al., 2016), a platform for the agent to learn tasks in the Doom game; House3D (WU et al., 2018), a platform with thousands of indoor scenes for navigation tasks. StarCraft II Learning Environment (VINYALS et al., 2017); a platform for the agent to learn tasks in the StarCraft II game; DeepMind Lab (BEATTIE et al., 2016), an environement for 3D navigation and puzzle-solving tasks; Project Malmo (JOHNSON et al., 2016), a platform built on top of the Minecraft game; and Gym-Duckietown (CHEVALIER-BOISVERT et al., 2018), a self-driving car simulator. The systems shown below were the ones currently integrated in this work's developed software.

### Gym

The OpenAI's Gym (BROCKMAN et al., 2016) is a toolkit for developing and comparing learning algorithms, that has simulation environments from the classic in control theory such as CartPole and Inverse Pendulum, to Atari and robotic environments.

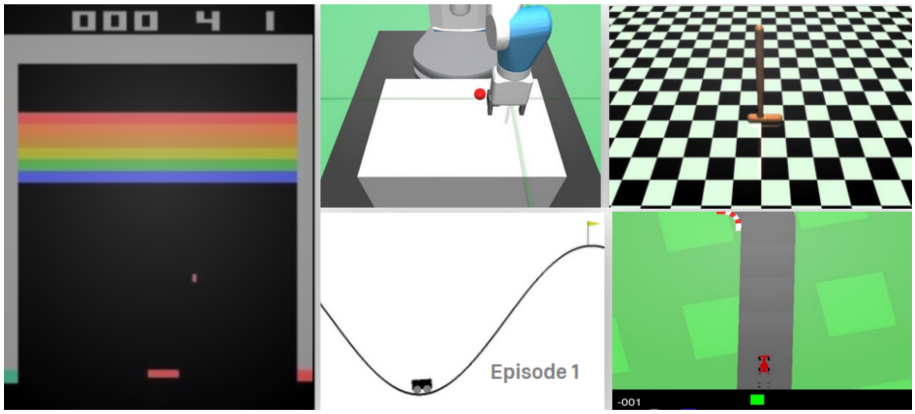


Figure 8: Screenshots of the Gym toolkit

### PyGame

The PyGame platform is an open-source python programming language library for making multimedia applications like games built on top of SDL library.

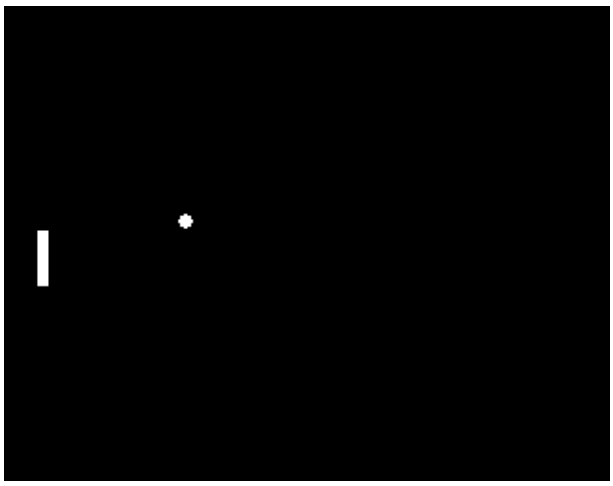

Figure 9: Screenshot of a PyGame environment

### Gym MuJoCo

MuJoCo (TODOROV; EREZ; TASSA, 2012) engine enables the creation of environments more similar to robotic environments, with continuous actions based on joint torques. The Gym toolkit has integrated the MuJoCo environment, and defined some learning benchmark tasks with its robotic models.

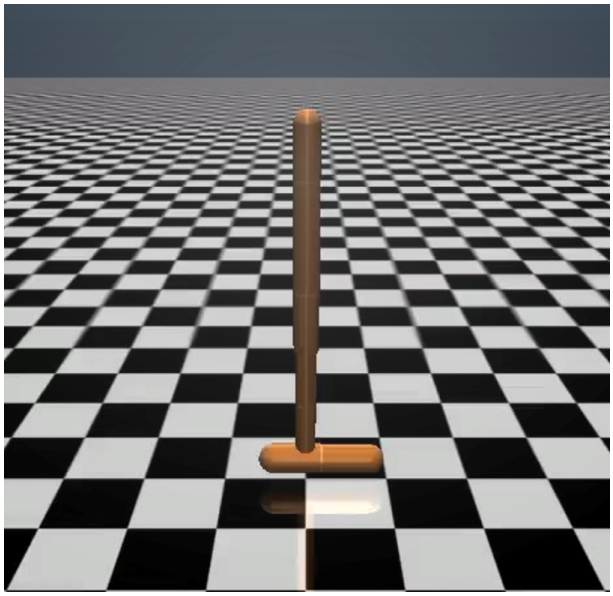

Figure 10: Screenshot of a Gym MuJoCo environment

### Unity Software

Unity (ENGINE, 2008) is a game engine used to create professional and non-professional games. It is a powerful tool to create customized environments, and Unity encouraged a ML approach by providing a toolkit (JULIANI et al., 2018) that facilitates the creation of ML tasks.

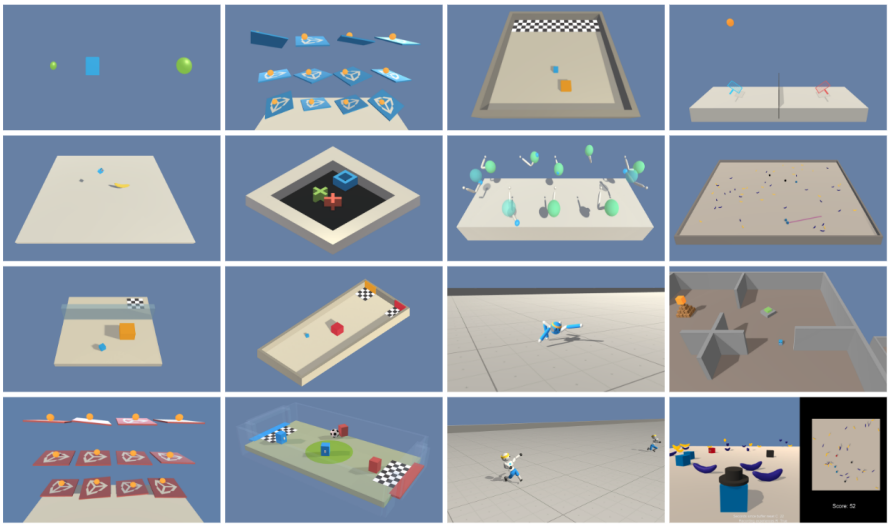

Figure 11: Screenshots of the Unity ML toolkit

### CARLA

CARLA (DOSOVITSKIY et al., 2017) platform that simulates a realistic driving environment for training locomotion tasks for autonomous vehicles, using Reinforcement Learning and Imitation Learning. It is a realistic simulator that provides customization

of the environment and provides many types of data such semantic segmented images or data from LIDAR sensors.

Figure 12: Screenshots of the CARLA platform

## 3.3 Algorithms

Several experiments were made by training the implemented algorithms on different environments from the engines integrated with the system developed in this work. Each tested environment has specific setups, shown in the Table 2 and described in the respective section.

| Environment | State Space | Action Space |
|---|---|---|
| Gym Cartpole | Vector (4) | Discrete (2) |
| PyGame Catch | Matrix (80x80) | Discrete (3) |
| Unity 3DBall | Vector (8) | Continuous (2) |
| Gym MuJoCo HalfCheetah | Vector (17) | Continuous (6) |
| CARLA | Matrix (84x84) | Discrete (4) |

Table 2: Summary of the environments use in the experiments

The implementations of the various algorithms presented throughout this work also have specific configurations, presented by a summary in the table 3. It is important to notice that this scalable collaborative system allows additional implementation of the algorithms. The algorithms currently implemented are ticked below:

| Algorithm | Discrete 1 dimensional | Discrete 2 dimensional | Continuous 1 dimensional | Continuous 2 dimensional |
|---|---|---|---|---|
| DQL | ✓ | ✓ | not possible | not possible |
| REINFORCE | ✓ | ✓ | X | X |
| REINFORCE+LSTM | ✓ | ✓ | X | X |
| A2C | ✓ | X | X | X |
| DDPG | X | X | ✓ | X |
| PPO | ✓ | ✓ | ✓ | ✓ |
| DAgger | ✓ | X | ✓ | X |
| GAIL | ✓ | X | ✓ | X |

Table 3: Summary of implemented algorithms

The next sections consists of a description of the implemented learning algorithms. The full code and documentation is available in the author's GitHub repository (RESENDE, 2018).

### Deep Q Learning

The method implemented uses the structure of Deep Q Network (MNIH et al., 2015), except for two differences. The first is the absence of the auxiliary Target ANN, which was not essential for stabilize learning. The second is the replacement of the $\epsilon - greedy$ policy with a policy called Bayesian. The Bayesian method uses the greedy method of selection of action states, where it is always selected the highest value, but uses the regularization technique Dropout (SRIVASTAVA et al., 2014), which randomly disables units and their connections during training in an ANN. Thus, it is guaranteed the exploration of lower value states from the random errors in the network output, at the same time that it reduces overfitting (situation in which the model fails, describing only a limited set of data). It was implemented to act in discrete action space environments. The algorithm developed is described below.

Initialize Experience Replay buffer D with capacity M;
Initialize function $Q_\pi$ with parameters $\theta$ using Dropout;
**while** *maximum of episodes* **do**
    Initialize environment;
    **for** $t = 1,\ T$ **do**
        Select action $a_t = argmax_a Q_\pi(s_t, a_t|\theta)$;
        Store $(s_t, a_t, r_t, s_{t+1})$ in D;
        Get random sample of trasitions $(s_t, a_t, r_t, s_{t+1})$ from D;
        Calculate $y = r_t + \gamma max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1}|\theta)$;
        Cost Function: $L(\theta) = [y - Q_\pi(s_t, a_t|\theta)]^2$;
        Update parameters $\theta$ from the cost function;
        Do $s_{t+1} = s_t$;
    **end**
**end**

**Algorithm 1:** Deep Q Learning

**REINFORCE**

The method implemented uses the structure of REINFORCE (WILLIAMS, 1992), with CNNs to process the states provided as images. Also, it was combined with RNNs with LSTM units, to capture temporal dependencies in the simulated environments, by taking sequence of states gathered in an adjustable setup. It was implemented to act in discrete action space environments, and so a method of action exploration is used, by selecting actions based on the proportional choice of the action probabilities returned. The algorithm developed is described below.

Initialize buffer D with capacity M;
Initialize policy $\pi$ with parameters $\theta$;
**while** *maximum of episodes* **do**
    Initialize environment;
    **for** $t = 1,\ T$ **do**
        Select action $a_t = \pi_\theta(s_t)$;
        Store $s_t, a_t, r_t, s_{t+1}$ in D;
        **if** *end of episode* **then**
            Calculate discounted rewards $G_t$;
            Cost function: $L(\theta) = G_t\ \nabla_\theta log\pi_{\theta_t}(s_t, a_t)$;
            Update parameters $\theta$ from the cost function;
            Reset D;
        **end**
        Do $s_{t+1} = s_t$;
    **end**
**end**

**Algorithm 2:** REINFORCE

### Advantage Actor Critic

The method implemented uses the structure of Advantage Actor Critic (SUTTON; BARTO, 2018), with MLPs to process the vector inputs, and Experience Replay. It was implemented to act in discrete action space environments. The algorithm developed is described below.

Initialize Experience Replayer buffer D with capacity M;
Initialize Actor $\pi$ with parameters $\theta$;
Initialize Critic $V_\pi$ with parameters $\zeta$;
**while** *maximum of episodes* **do**
    Initialize environment;
    **for** *t=1, T* **do**
        Select action $a_t = \pi_\theta(s_t)$;
        Store $(s_t, a_t, r_t, s_{t+1})$ in D;
        Get random sample of transitions $(s_t, a_t, r_t, s_{t+1})$ from D;
        Calculate Advantage $A = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$;
        Actor's cost function: $L(\theta) = A \nabla_\theta log \pi_{\theta_t}(s_t, a_t)$;
        Critic's cost function: $L(\zeta) = (r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t))^2$;
        Update parameters $\theta$ e $\zeta$ from the cost functions;
        Do $s_{t+1} = s_t$;
    **end**
**end**

**Algorithm 3:** Advantage Actor Critic

### Deep Deterministic Policy Gradients

The method implemented uses the structure of Deep Deterministic Policy Gradients(LILLICRAP et al., 2015), including auxiliar networks, soft updates, L2-Regularization, concatenation of actions in the Critic hidden layer, action selection with Uhlenbeck and Ornstein process, calculation of the Critic's gradients to update the Actor and Experience Replay. It was implemented to act in continuous action space environments. The algorithm developed is described below.

Initialize Experience Replay buffer D with capacity M;
Initialize Actor $\pi$ with parameters $\theta$ ;
Initialize Critic $Q_\pi$ with parameters $\zeta$ ;
Initialize Actor Target $\pi'$ with parameters $\theta' \leftarrow \theta$;
Initialize Critic Target $Q'_\pi$ with parameters $\zeta' \leftarrow \zeta$;
**while** *maximum of episodes* **do**
    Initialize environment; Initialize Uhlenbeck and Ornstein exploration process N;
    **for** *t=1,T* **do**
        Select action $a_t = \pi_\theta(s_t)$ + N;
        Store $(s_t, a_t, r_t, s_{t+1})$ in D;
        Get random sample of transitions $(s_t, a_t, r_t, s_{t+1})$ from D;
        Calculate $a$, $Q_\pi(s_t, a_t|\zeta)$, $Q_\pi(s_{t+1}, a_{t+1}|\zeta)$ with the Target networks;
        Calculate $y = r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}|\zeta)$;
        Critic's cost function: $L(\zeta) = [y - Q_\pi(s_t, a_t|\zeta)]^2$;
        Get gradients $\zeta$ with the Critic's cost function and update $\zeta$;
        Update Actor's parameters $\theta$ with Critic's gradients $\nabla_{\theta^\zeta} L$ ;
        Update Target networks with $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$;
        Do $s_{t+1} = s_t$;
    **end**
**end**

**Algorithm 4:** Deep Deterministic Policy Gradients

### Proximal Policy Optimization

The method implemented uses the structure of Proximal Policy Optimization (SCHULMAN et al., 2017), including the calculation of the Generalized Advantage Estimation, but does not have multiple Actors in parallel processing, as described in the original paper, and therefore the data samples are obtained by a single agent. It was implemented to act in both discrete and continuous action space environments, and for both 1-dimensional and 2-dimensional inputs. The algorithm developed is described below. '

Initialize buffer D with capacity M;
Initialize Critic $V_\pi$ with parameters $\zeta$;
Initialize Actor $\pi$ with parameters $\theta$;
Initialize Old Actor $\pi_{old}$ with parameters $\theta_{old}$;
**while** *maximum of episodes* **do**
  Initialize environment;
  **for** *t=1,T* **do**
    Select action $a_t = \pi(s_t|\theta)$;
    Store episode $(s_t, a_t, r_t, s_{t+1})$ in D;
    **if** *buffer full* **then**
      Calculate Advantage $A_t = \sum_{l=0}^{M} (\gamma\lambda)^l \ \delta_{t+l}^V$ with
       $\delta = r_t + \gamma V_\pi(s_{t+1}|\zeta) - V_\pi(s_t|\zeta)$;
      Copy Actor's parameters to Old Actor;
      **for** *number of updates* **do**
        Actor's cost function:
        $L_{clipped} = E_t\left[min(ratio_t(\theta)A_t; clip(ratio_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right]$ com
         $ratio_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$;
        Critic's cost function:
        $L = r_t + \gamma V_\pi(s_{t+1}|\zeta) - V_\pi(s_t|\zeta)$;
        Update parameters $\theta$ and $\zeta$ from the cost functions;
        Reset D;
      **end**
    **end**
    Do $s_{t+1} = s_t$;
  **end**
**end**

**Algorithm 5:** Proximal Policy Optimization

### Dataset Aggreggation

The method implemented uses the structure of Dataset Aggregation (ROSS; GORDON; BAGNELL, 2011), but since the algorithm needs feedback from the expert, which is not convenient in the current simulation platforms, it was implemented using two ANNs. The first serves as the expert, being optimized once at the beginning of the training and serving a the expert, consulted by the other ANN, that performs scheduled stages of optimization. It was implemented to act in both discrete and continuous action space environments. The algorithm developed is described below.

Initialize trajectories dataset $\tau_E = \{(s_0, a_0), (s_1, a_1), (s_2, a_2), ...\}$;
Initialize buffer $\tau_b = \{\}$;
Initialize $\beta = 1$;
Initialize $\pi_0 = \pi_E$;
Train initial policy $\pi_E$ using Supervised Learning using $\tau_E$;
**while** *maximum of episodes* **do**
  Initialize environment;
  **for** *t=1,T* **do**
    Select action $a_t = \pi_E(s)$ with probability $\beta$ or $a_t = \pi_t(s)$ with probability $1 - \beta$;
    Add experience to buffer: $\tau_b = \tau_b + (s_t, a_t)$;
    **if** *buffer full* **then**
      Aggregate datasets $\tau_E \leftarrow \tau_E \cup \tau_b$;
      Update $\pi_t$ training $\pi_t$ with $\tau_E$ using Supervised Learning;
      **if** $\beta = 1$ **then**
        $\beta = 0.5$
      **end**
      Do $\beta = \beta^{nžofupdates}$ ;
      Reset $\tau_b$;
    **end**
    Do $s_{t+1} = s_t$;
  **end**
**end**

**Algorithm 6:** Dataset Aggregation

**Generative Adversarial Imitation Learning**

The method implemented uses the structure of GAIL (HO; ERMON, 2016), with the standard GAN's cost function, the use of auxiliar ANNs, and Behavior Cloning at the start from the trajectories dataset. In the original paper, the Trust Region Policy Optimization (SCHULMAN et al., 2015a) algorithm is used as the Monte Carlo estimation algorithm. However, PPO was used in this implementation. It was implemented to act in both discrete and continuous action space environments. The algorithm developed is described below.

Initialize dataset trajectories $\tau_E = \{(s_0, a_0), (s_1, a_1), (s_2, a_2), ...\}$ and buffer $\tau_b = \{\}$;
Initialize Critic $V_\pi$ with parameters $\zeta$;
Initialize Generator/Actor $\pi$ with parameters $\theta$;
Initialize Old Generator/Old Actor $\pi_{old}$ with parameters $\theta_{old}$;
Initialize Discriminator $D$ with parameters $w$;
Train initial policy $\pi$ with $\tau_E$ using Supervised Learning;
**while** *maximum of episodes* **do**
  Initialize environment;
  **for** *t=1,T* **do**
    Select action $a_t = \pi(s_t|\theta)$;
    Add experience to buffer: $\tau_b = \tau_b + (s_t, a_t)$;
    **if** *buffer full* **then**
      Discriminador's cost function:
      $E_{w,\tau_b}[\log D_w(\tau_b)] + E_{w,\tau_E}[\log(1 - D_w(\tau_E))]$ ;
      Update parameters $w$ of the Discriminador from the cost function;
      Calculate rewards $r = -\log(1 - \sigma(D(\tau_b)))$;
      Calculate Advantage $A_t = \sum_{l=0}^{M} (\gamma\lambda)^l \ \delta_{t+l}^V$ with
       $\delta = r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$;
      Copy Actor's parameters to Old Actor;
      **for** *number of updates* **do**
        Actor's cost function:
        $L_{clipped} = E_t\left[min(ratio_t(\theta)A_t; clip(ratio_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right]$ with
        $ratio_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$;
        Critic's cost function: $L = r_t + \gamma V_\pi(s_{t+1}|\zeta) - V_\pi(s_t|\zeta)$;
        Update parameters $\theta$ e $\zeta$ from the cost functions and reset $\tau_b$;
      **end**
    **end**
    do $s_{t+1} = s_t$;
  **end**
**end**

**Algorithm 7:** Generative Adversarial Imitation Learning

# 4 RESULTS AND DISCUSSION

This chapter consists of an overview of implemented algorithms and environment settings, the defined metrics, and the results of the learning experiments, carried out with the developed software.

## 4.1 Experimental setup

The next sections present the results of the comparison of the learning process of the various RL and IL algorithms implemented and described in the Materials and Methods (Section 3). It is important to notice that all experiments were made using the software developed, which integrates all algorithms and environments on a unique system that shares the same software design pattern.

To achieve compatible comparison in every environment, a sum of rewards is calculated from an episode of fixed 1000 steps of interaction, in every experiment. This defined episode is different from the environment episode, in which the simulation is restarted. And also, one step is defined as one environment interaction, that does not implicate an optimizer update with change of network parameters.

The first defined metric and displayed in a graph format is "Rewards obtained by the learning algorithms per time of training", that can indicate how fast the algorithm is learning to achieve a greater reward by a time measurement. This metric is calculated at each 1000 steps, described by:

$$Episode\ rewards\ by\ time = \sum_{step=step-1000}^{step} r(step)\ X \sum_{step=0}^{step} t(step) \tag{4.1}$$

The second defined metric is a table that shows 4 measurements: the maximal reward obtained by the algorithm in the environment, how many steps were necessary to first achieve this maximal reward, the time necessary to first achieve this maximal reward, and the average duration of a step.

These metrics are well suited for comparison of RL algorithms, but a reward function is unknown in the context of IL. However, rewards are still provided by environments developed for RL tasks when training IL algorithms in environments. The IL algorithms do not use these rewards, but their performance can be measured by comparing these rewards achieved in training with the rewards of the collected dataset. This allows comparison of IL and RL algorithms in these environments.

To measure the performance of IL algorithms in environments that does not provides a reward function, environment-specific information can be used. For example, in

an environment where the agent is an autonomous car, possible metrics can be defined as distance travelled in episode, average speed, number of collisions, and even visual comparison.

In the next sections, it is presented the simulation environment, the specific settings for each tested algorithm, the results achieved, and an analysis. The experiments were run on computer with a 7th Generation Intel Core i7 Processor and NVIDIA GeForce 940MX Graphics Processing Unit.

## 4.2 Gym Cartpole environment

The Gym Cartpole, presented in Fig. 13 is a simulation environment where the purpose of the agent is to balance the inverted pendulum by acting in the orientation of the cart. The environment provides a reward of +1 at each step, and a modification was made so that it would also provide a reward of -50 when the environment episode ends, which occurs when the pendulum falls with an angle greater than 15 degrees or after 200 steps. The environment provides as states a vector with the position and speed of the cart, and the speed and angular velocity of the pendulum. The discrete action space consists of 2 actions for the agent to control the cart to left or right.

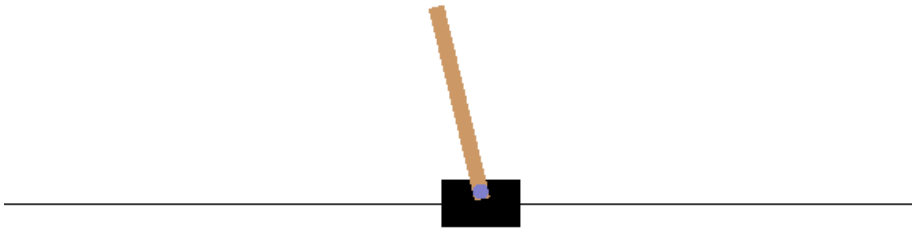

Figure 13: Screenshot of the Gym CartPole environment

First, the Deep Q Learning algorithm was trained in this environment. The ANN architecture used is as follows. The input to the ANN consists of the 4-features input vector stacked by 3 time-steps. The network has 1 fully connected hidden layer with 512 units with rectifier non-linearity. The output has 2 linear units for the possible state-action values, the action selection to be applied in the environment is to choose the maximal value, and using the Bayesian method, the dropout can alter the output to ensure exploration of actions in the early stages of training. The algorithm was trained with a learning rate of 1e-4 using Adam, reward discount of 0.99, experience replay with size 100k, batch size of 50, 500 initial random steps, start random probability of 1 and final of 0.05, decreased in 20 environment episodes.

Also, the REINFORCE algorithm was trained in this environment. The input to the ANN consists of the 4-features input vector stacked by 3 time-steps. The network has 1 fully-connected hidden layer with 512 units with rectifier nonlinearity. The output has 2 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 1e-4 using Adam, and reward discount of 0.99.

The Advantage Actor Critic algorithm was the third method trained in this environment. The input to the ANN consists of the 4-features input vector stacked by 3 time-steps. The Actor and the Critic have both 1 fully-connected hidden layer with 256 units with rectifier non-linearity. The output of the Critic is linear to represent $V(s)$ and the output of the Actor has 2 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 1e-3 for the Critic and 1e-4 for the Actor using Adam, reward discount of 0.99, experience replay with size 100k, batch size of 50 and 500 initial random steps

Following, the PPO algorithm was trained in this environment. The input to the ANN consists of the 4-features input vector stacked by 3 time-steps. The Actor and Critic have both 1 hidden fully connected layer with 256 units with hyperbolic tangent non-linearity. The output of the Critic is linear to represent $V(s)$ and the output of the Actor has 2 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 1e-4 for the Critic and the Actor using Adam, reward discount of 0.99, batch size of 50, epsilon of 0.15, gamma of 0.99 and lambda of 0.95.

To train the IL algorithms, a dataset of collected demonstrated trajectories $\tau = (s, a)$ in the Gym Cartpole environment was created using a trained agent that performs well. The dataset characteristics is shown in Table 4.

| Demonstrator | Environment Interactions | Average Reward | Std. Dev. | Maximal Reward | Minimal Reward |
|---|---|---|---|---|---|
| Trained PPO | 100000 | 750 | 0 | 750 | 750 |

Table 4: Demonstrated trajectories for the Gym Cartpole environment

Also, the GAIL algorithm was trained in this environment with this dataset. The input to the Actor and Critic consists of the 4-features input vector stacked by 3 time-steps and to the Discriminator network was the trajectory, i.e. the concatenation of the state vector and action vector. All ANNs had 1 hidden fully connected layer with 256 units with hyperbolic tangent non-linearity. The output of the Actor had 2 units for the possible actions with softmax nonlinearity representing the action probabilities, the output of the

Critic is the value $V(s)$ and the output of the Discriminator is the classification of how the trajectory is similar to the dataset. The algorithm was trained using Adam with a learning rate of 1e-4 for the Actor and Critic and 3e-4 for the Discriminator, update size of 5, epsilon of 0.10, gamma of 0.995, lambda of 0.95, reward discount of 0.99, and batch size of 50, initial Behaviour Cloning with learning rate of 1e-4, number of epochs of 100, and batch size of 256. An experiment of removing the initial BC stage of the GAIL algorithm was made, and the results are also shown.

To compare the performance of these algorithms in the Gym CartPole environment, they were trained during a fixed period of 150 seconds. Fig. 14 presents a graph of rewards (obtained in each thousand steps) per time (in seconds of training).

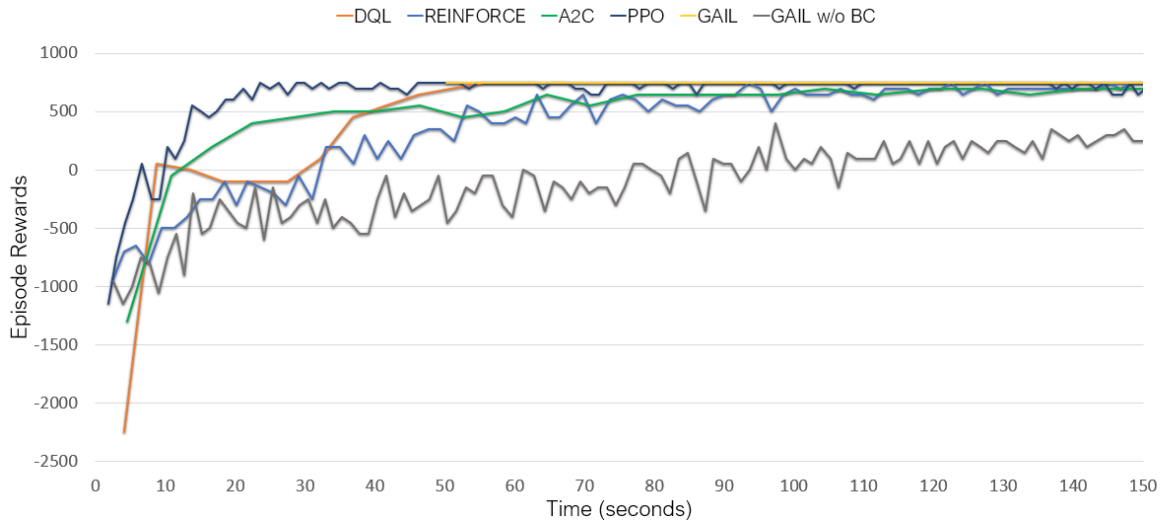


Figure 14: Rewards obtained by the learning algorithms per time of training in the Gym CartPole environment

Table 5 shows the other defined measurements to compare the algorithms:

| Algorithm | Max. Reward | Steps to Max. Reward (thousand) | Time to Max. Reward (s) | Average Duration of Steps (ms) |
|---|---|---|---|---|
| DQL | 750 | 25 | 162.4 | 6.5 |
| REINFORCE | 750 | 52 | 93.4 | 1.7 |
| A2C | 750 | **12** | 55.6 | 4.8 |
| PPO | 750 | 19 | **23.6** | **1.2** |
| GAIL | 750 | **1** | 50.2 | 1.78 |
| GAIL w/o BC | 400 | 75 | 97.3 | 1.29 |

Table 5: Comparison of rewards by steps and time in the Gym Cartpole environment

PPO achieves the best results of rewards by time among the RL algorithms, while

GAIL achieves a constant maximal reward, as showed in the graph. It is shown in the table that A2C has the best results of steps to first achieve the maximal reward while not presenting the best result by time, since the optimization in every step done by Off-Policy algorithms requires a lot of computation, increasing the average duration of steps. GAIL reaches the maximal reward in the first step, since it starts the training with Behaviour Cloning, training the Actor network with Supervised Learning from the dataset. An experiment was made removing this part of the algorithm, for it to learn from scratch the parameters of all three of its networks. The results are worse compared to the RL algorithms due to the fact that the learning process of the Actor are limited to the current ability of the Discriminator to provide meaningful rewards, which is minimal at start.

## 4.3 PyGame Catch environment

The PyGame Catch, presented in Fig. 15, is a simulation environment where the purpose of the agent is to capture all the white balls that cross the screen from right to left. The environment provides a reward of +1 when the ball reaches the agent bar and -1 when it reaches the right side of the screen, ending an environment episode with either +25 or -25 points. The environment provides as states images of 80x80 pixels in black and white and the discrete action space is composed of 3 actions, for the agent to go up and down or stay still.

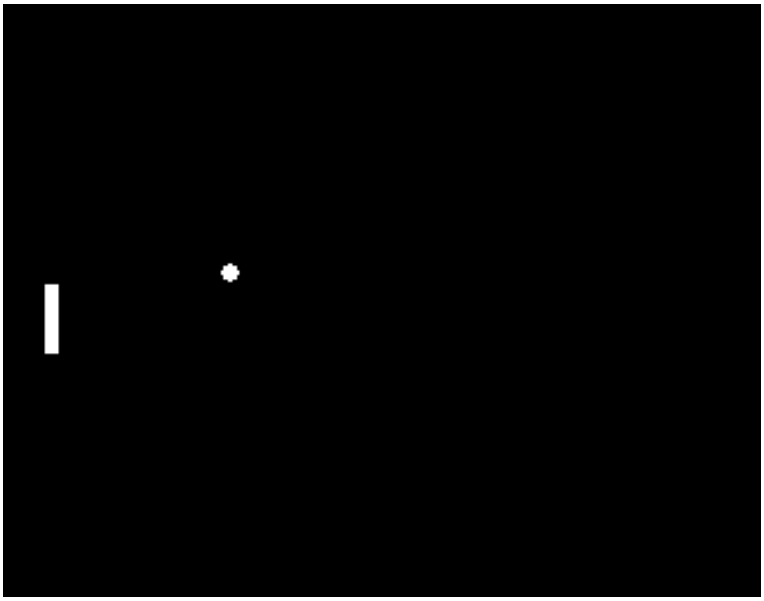

Figure 15: Screenshot of the PyGame Catch environment

First, the DQL algorithm was trained in this environment. The input to the ANN consists of the 80x80 image stacked by 3 frames. The network architecture is the same as the one described in (MNIH et al., 2015), but with the last hidden layers using 256 units. The output has 3 linear units for the possible state-action values. The action selection to

be applied in the environment is to choose the maximal value, and using the Bayesian method described in Section 3.3, the dropout can alter the output to ensure exploration of actions in the early stages of training. The algorithm was trained with a learning rate of 1e-4 using Adam (KINGMA; BA, 2014), reward discount of 0.99, experience replay with size 100k, batch size of 512, 500 initial random steps, start random probability of 1 and final of 0.05, decreased in 20 environment episodes.

Also, the REINFORCE algorithm was trained in this environment. The input to the ANN consists of the 80x80 image stacked by 3 frames. The network has the layers architecture of (MNIH et al., 2015). The output has 3 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 1e-4 using Adam, and reward discount of 0.99.

A LSTM Recurrent Neural Network can also be added, to substitute the method of stacking frames to capture temporal dependencies in the environment, which is modeled as a MDP while still having these temporal dependencies. The experiment of adding LSTM layers was done in the REINFORCE algorithm, by adding 2 LSTM layers each with 128 units and 3 time-step cells after the convolutional and fully connected layers.

Following, the PPO algorithm was trained in this environment. The input to the ANN consists of the 80x80 image stacked by 3 frames. The Actor and Critic have the layers architecture of (MNIH et al., 2015). The output of the Critic is linear to represent $V(s)$ and the output of the Actor has 3 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 1e-3 for the Critic and 5e-5 for the Actor using Adam, reward discount of 0.99, batch size of 50, epsilon of 0.10, gamma of 0.995 and lambda of 0.95.

To compare the performance of these algorithms in the PyGame Catch environment, they were trained during a fixed period of 9000 seconds. Fig. 16 presents a graph of rewards (obtained in each thousand steps) per time (in seconds of training).

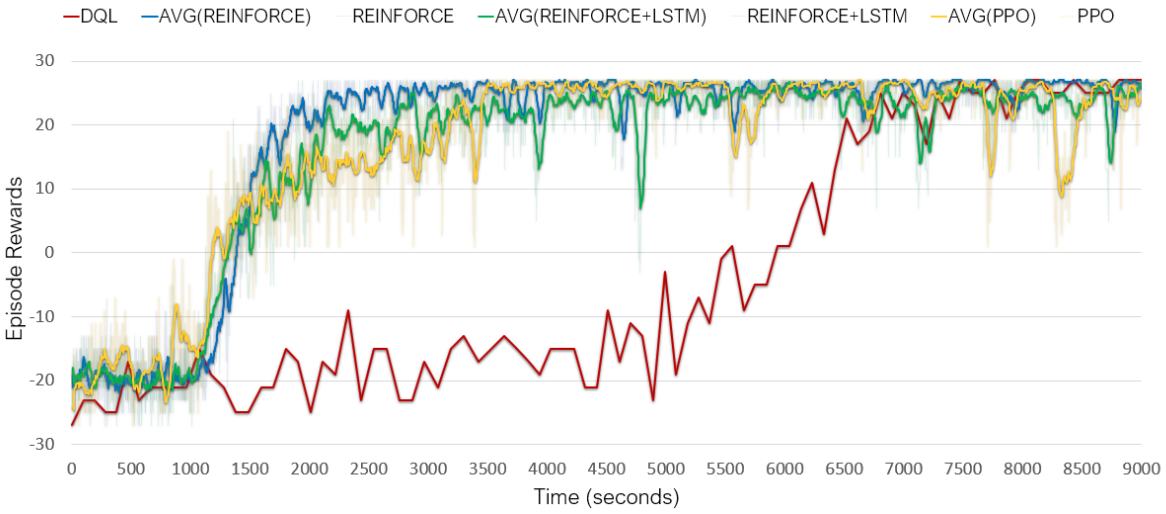


Figure 16: Rewards obtained by the learning algorithms per time of training in the PyGame Catch environment

Table 6 shows the other defined measurements to compare the algorithms:

| Algorithm | Max. Reward | Steps to Max. Reward (thousand) | Time to Max. Reward (s) | Average Duration of Steps (ms) |
|---|---|---|---|---|
| DQL | 27 | **76** | 7475.5 | 97.7 |
| REINFORCE | 27 | 530 | 1971.5 | **3.6** |
| REINFORCE + LSTM | 27 | 374 | **1906.4** | 5.0 |
| PPO | 27 | 470 | 2812.3 | 5.9 |

Table 6: Comparison of rewards by steps and time in the PyGame Catch environment

In this environment, that provides images as states, there is a clear difference of rewards obtained by time of training from Off-Policy and On-Policy algorithms. The computation of the optimization updates of CNN's performed by DQL at every step creates a visible disadvantage in time, showed in the graph, while reaching the maximal reward in many less steps than the On-Policy algorithms, as show in the table. Also, it is seen in the graph that the addition of LSTM layers in the REINFORCE algorithm does not results in improvements in this environment when analyzing by time, because the many operations performed by LSTM layers consumes more computation, increasing the average duration of steps, however, it is seen in the table that the REINFORCE+LSTM reached the maximal reward in less steps and less time than REINFORCE.

## 4.4 Unity 3DBall environment

The Unity 3DBall, presented in Fig. 17, is a simulation environment where the purpose of the agent is to control a rotational platform with a ball at the top and prevent the ball from failing. Despite the figure of the environment shows 12 platform, the experiment was made with 1. The environment provides a reward of +0.1 for every step the ball remains on the platform and -1 when ball falls from the platform. Each episode ends when the ball falls, or with the maximum reward of 100 points. The environment provides a states 8 values corresponding to rotation of the platform, and position, rotation and velocity of the ball. The agent has 2 continuous actions to control the platform angles.

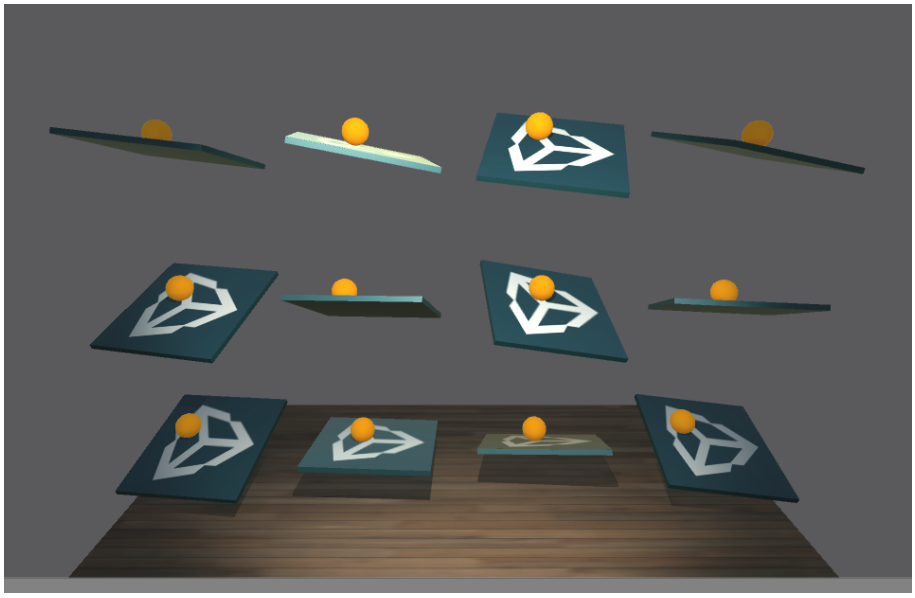

Figure 17: Screenshot of the Unity 3DBall environment

First, the DDPG algorithm was trained in this environment. The input to the Actor and Critic consists of the 8-features input vector, and the architecture was the same as in (LILLICRAP et al., 2015), including the copy of the Actor and the Critic, the concatenation of actions in the second hidden layer of the Critic, the L2 weight decay and initialization from a uniform distribution. However, the number of units of the hidden layers was 128 and 200 units respectively. The action selection from the output was using the Ornstein-Uhlenbeck process described in (LILLICRAP et al., 2015). The algorithm was trained with a learning rate of 1e-3 for the Critic and 1e-4 for the Actor using Adam, reward discount of 0.99, experience replay with size 100k and batch size of 128.

Also, the PPO algorithm was trained in this environment. The input to the Actor and Critic consists of the 8-features input vector, and the architecture was the same as in (SCHULMAN et al., 2017), but using hidden layers with 128 and 200 units respectively, using and hyperbolic tangent activation function, an epsilon of 0.15, gamma of 0.995 and lambda of 0.95. The action selection was a sampling of the network output, which is a

mean and a standard deviation that forms a distribution. The algorithm was trained with a learning rate of 1e-4 for the Critic and the Actor using Adam, reward discount of 0.99, and batch size of 1024.

To train the Imitation Learning algorithms, a dataset of collected demonstrated trajectories $\tau = (s, a)$ in the Unity 3DBall environment was created using a trained agent that performs well. The dataset characteristics is shown in Table 7.

| Demonstrator | Environment Interactions | Average Reward | Std. Dev. | Maximal Reward | Minimal Reward |
|---|---|---|---|---|---|
| Trained PPO | 100000 | 100 | 0.26 | 100 | 98.5 |

Table 7: Demonstrated trajectories for the Unity 3DBall environment

The DAgger algorithm was trained in this environment with this dataset. The input to the networks consisted of the 8-features input vector. All ANNs had 2 hidden fully connected layers with 128 and 200 units with rectifier non-linearity. The output of the networks is linear, directly used as an action is the evironment. The algorithm was trained using Adam with a learning rate of 1e-4, beta of 0.5, batch size of 32, and it running 100 epochs of SL update at each 10000 steps of interaction, with an epoch defined as the dataset size divided by the batch size.

Following, the GAIL algorithm was trained in this environment with this dataset. The input to the Actor and Critic consists of the 8-features input vector, and to the Discriminator network was the trajectory, i.e. the concatenation of the state vector and action vector. All ANNs had 2 hidden fully connected layers with 128 and 200 units with hyperbolic tangent non-linearity. The output of the Actor is a mean an standard deviation used to sample an action, the output of the Critic is the value $V(s)$ and the output of the Discriminator is the classification of how the trajectory is similar to the dataset. The algorithm was trained using Adam with a learning rate of 1e-4 for the Actor and Critic and 1e-3 for the Discriminator, update size of 5, epsilon of 0.15, gamma of 0.995, lambda of 0.95, reward discount of 0.99, and batch size of 512, initial behaviour cloning with learning rate of 1e-4, number of epochs of 100, and batch size of 256.

To compare the performance of these algorithms in the Unity 3DBall environment, they were trained during a fixed period of 1000 seconds. Fig. 18 presents a graph of rewards (obtained in each thousand steps) per time (in seconds of training).

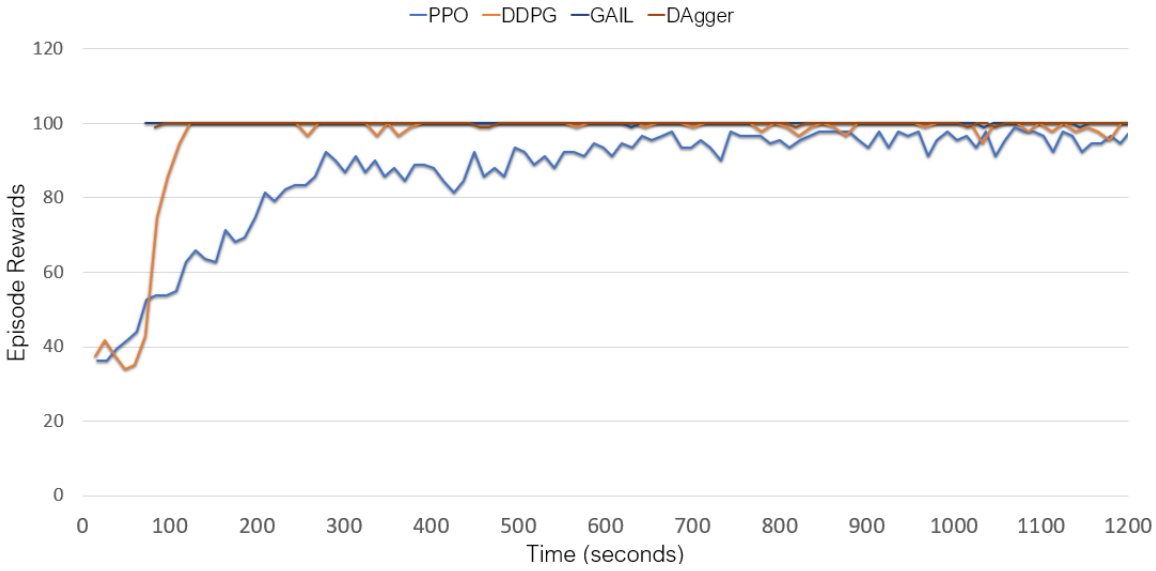


Figure 18: Rewards obtained by the learning algorithms per time of training in the Unity 3DBall environment

Table 8 shows the other defined measurements to compare the algorithms:

| Algorithm | Max. Reward | Steps to Max. Reward (thousand) | Time to Max. Reward (s) | Average Duration of Steps (ms) |
|---|---|---|---|---|
| PPO | 100 | 111 | 1258.5 | **11.2** |
| DDPG | 100 | **10** | **123.9** | 13.6 |
| DAgger | 100 | 2 | 83.0 | 19.7 |
| GAIL | 100 | **1** | **72.0** | 11.5 |

Table 8: Comparison of rewards by steps and time in the Unity 3DBall environment

DDPG have the best results among the RL algorithms, as it is shown that the increase of average duration of steps of the DDPG is small considering the large decrease of steps and time to the reach the maximal reward when compared to PPO. The IL algorithms DAgger and GAIL achieves maximal rewards at start.

## 4.5 Gym MuJoCo HalfCheetah environment

The Gym MuJoCo HalfCheetah, presented in Fig. 13 is a simulation environment where the purpose of the agent is to control the joint's torque of a robotic cheetah so that it travels as fast as possible to the right. The environment provides a positive reward proportional to the robot position displacement and a negative reward proportional to the square of the torques applied to the joints and each episode ends with a pre-set time. The environment provides as states the position, angle, velocity and angular velocity of the

joints, and the continuous action space consists of 6 actions for the agent to control the toque on each joint.

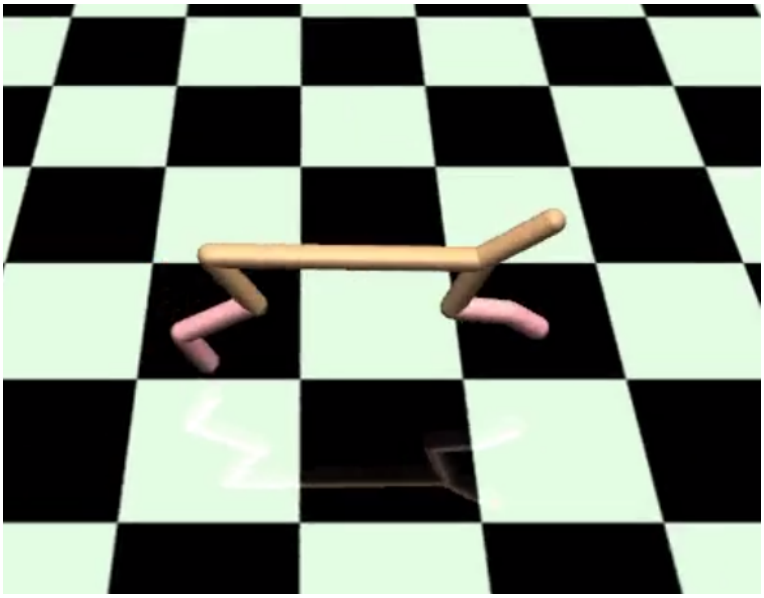

Figure 19: Screenshot of the Gym MuJoCo HalfCheetah environment

First, the DDPG algorithm was trained in this environment. The input to the Actor and Critic consists of the 17-features input vector and the architecture used is the same as in (LILLICRAP et al., 2015), including the copy of the Actor and the Critic, the concatenation of actions in the second hidden layer of the Critic, the L2 weight decay and initialization from a uniform distribution. However, the number of units of the hidden layers was 128 and 200 units respectively. The action selection from the output was using the Ornstein-Uhlenbeck process described in (LILLICRAP et al., 2015). The algorithm was trained with a learning rate of 1e-3 for the Critic and 1e-4 for the Actor using Adam, reward discount of 0.99, experience replay with size 100k and batch size of 128.

Also, the PPO algorithm was trained in this environment. The input to the Actor and Critic consists of the 17-features input vector, and the architecture was the same as in (SCHULMAN et al., 2017), but using hidden layers with 128 and 200 units respectively, using and hyperbolic tangent activation function, an epsilon of 0.15, gamma of 0.995 and lambda of 0.95. The action selection was a sampling of the network output, which is a mean and a standard deviation that forms a distribution. The algorithm was trained with a learning rate of 1e-4 for the Critic and the Actor using Adam, reward discount of 0.99, and batch size of 2048.

To train the IL algorithms, a dataset of collected demonstrated trajectories $\tau = (s, a)$ in the Gym HalfCheetah environment was created using a trained agent that performs well. The dataset characteristics is shown in Table 9.

| Demonstrator | Environment Interactions | Average Reward | Std. Dev. | Maximal Reward | Minimal Reward |
|---|---|---|---|---|---|
| Trained DDPG | 100000 | 4954.6 | 454.6 | 5825.0 | 2949.4 |

Table 9: Demonstrated trajectories for the Gym HalfCheetah environment

Following, the GAIL algorithm was trained in this environment with this dataset. The input to the Actor and Critic consists of the 17-features input vector, and the input to the Discriminator network was the trajectory, i.e. the concatenation of the state vector and action vector. All ANNs had 2 hidden fully connected layers with 128 and 200 units with hyperbolic tangent non-linearity. The output of the Actor is a mean an standard deviation used to sample an action, the output of the Critic is the value $V(s)$ and the output of the Discriminator is the classification of how the trajectory is similar to the dataset. The algorithm was trained using Adam with a learning rate of 1e-4 for the Actor and Critic and 1e-3 for the Discriminator, update size of 5, epsilon of 0.15, gamma of 0.995, lambda of 0.97, reward discount of 0.99, and batch size of 1024, initial behaviour cloning with learning rate of 1e-4, number of epochs of 100, and batch size of 256.

To compare the performance of these algorithms in the Gym MuJoCo HalfCheetah environment, they were trained during a fixed period of 13000 seconds. Fig. 20 presents a graph of rewards (obtained in each thousand steps).

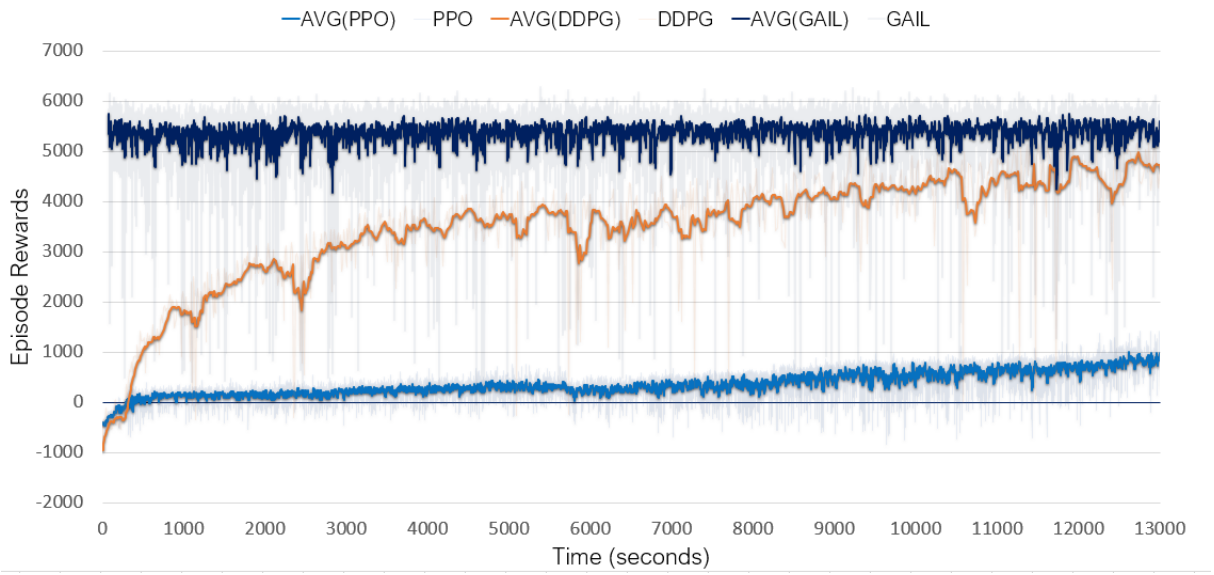


Figure 20: Rewards obtained by the learning algorithms per time of training in the Gym MuJoCo HalfCheetah environment

Table 10 shows the other defined measurements to compare the algorithms:

| Algorithm | Max. Reward | Steps to Max. Reward (thousand) | Time to Max. Reward (s) | Average Duration of Steps (ms) |
|---|---|---|---|---|
| PPO | 1427.5 | 10396 | 12750.5 | **1.0** |
| DDPG | **5396.9** | **911** | **12560.9** | 14.4 |
| GAIL | **6289.9** | 4473 | **5385.4** | **1.16** |

Table 10: Comparison of rewards by steps and time in the Gym MuJoCo HalfCheetah environment

DDPG have the best results among the RL algorithms: the increase of average duration of steps of the DDPG compared to PPO is 14 times bigger, however, unlike in the Unity 3DBall experiment where DDPG and PPO achieves the same maximal reward, in this task DDPG achieves a maximal reward almost 4 times bigger in practically the same time. The GAIL algorithm achieves good performance, however, it is seen that the sub-optimal demonstrated trajectories limits the performance of the algorithm, resulting in instability, despite achieve a good performance by average reward, similar to the dataset (Table 9), which is considered sub-otimal given the large standard deviation.

## 4.6 CARLA environment

In the CARLA platform, a simple task was set in which the agent starts at a point in a street, and its objective is to drive straight to the final point. The environment provides as positive rewards the distance traveled towards the goal and speed, and as negative rewards collision damage and intersection with the sidewalk and opposite lane. The environment was set to provide as states the black and white images with 84x84 pixels stacked by 3 time steps, and it was set a discretized action space of 4 actions for the agent to accelerate forward, turn left, right, or break.


Figure 21: Screenshot of the CARLA environment

First, the DQL algorithm was trained in this environment. The input to the ANN consists of the 84x84 black and white image stacked by 3 frames. The network architecture is the same as the one described in (MNIH et al., 2015). The output has 4 linear units for the possible state-action values. The action selection to be applied in the environment was with the Bayesian method. The algorithm was trained with a learning rate of 5e-5 using Adam (KINGMA; BA, 2014), reward discount of 0.99, experience replay with size 100k, batch size of 100, 100 initial random steps, start random probability of 1 and final of 0.05, decreased in 5 environment episodes.

Also, the REINFORCE algorithm was trained in this environment. The input to the ANN consists of the 84x84 black and white image stacked by 3 frames. The network has the layers architecture of (MNIH et al., 2015). The output has 4 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 5e-5 using Adam, and reward discount of 0.99.

Following, the PPO algorithm was trained in this environment. The input to the ANN consistsof the 84x84 black and white image stacked by 3 frames. The Actor and Critic have the layers architecture of (MNIH et al., 2015). The output of the Critic is linear to represent $V(s)$ and the output of the Actor has 4 units for the possible actions with softmax nonlinearity representing the action probabilities. The action selection is a choice based on these probabilities. The algorithm was trained with a learning rate of 1e-4

for the Critic and 5e-5 for the Actor using Adam, reward discount of 0.99, batch size of 100, epsilon of 0.10, gamma of 0.99 and lambda of 0.95.

To compare the performance of these algorithms in the CARLA environment, they were trained during a fixed period of 2300 seconds. Fig. 22 presents a graph of rewards (obtained in each thousand steps).

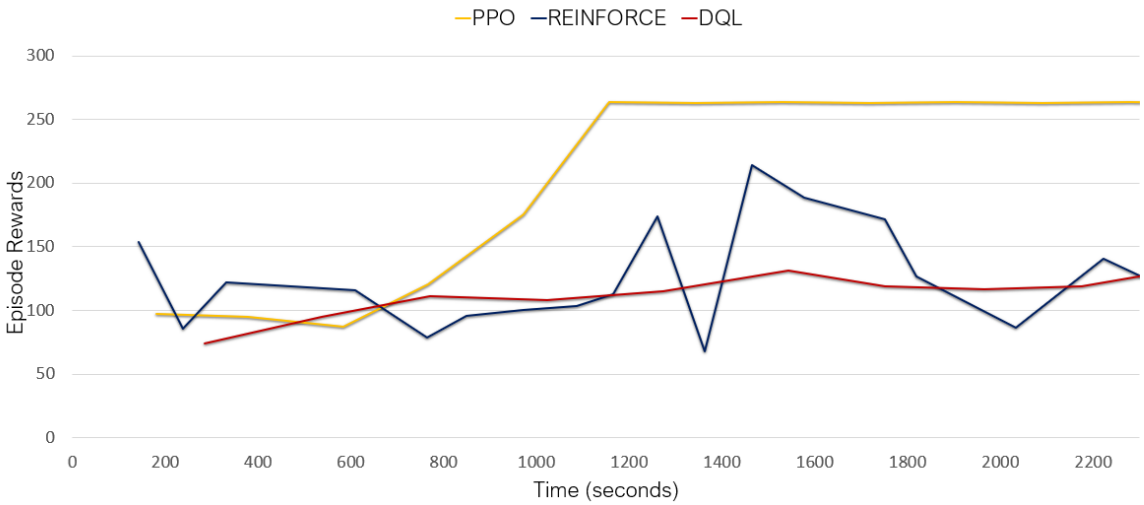


Figure 22: Rewards obtained by the learning algorithms per time of training in the CARLA environment

Table 11 shows the other defined measurements to compare the algorithms:

| Algorithm | Max. Reward | Steps to Max. Reward (thousand) | Time to Max. Reward (s) | Average Duration of Steps (ms) |
|---|---|---|---|---|
| DQL | 131.2 | **6** | 1543.5 | 235.9 |
| REINFORCE | 214.2 | 12 | 1464.8 | **171.6** |
| PPO | **263.8** | 14 | 2651.7 | 189.3 |

Table 11: Comparison of rewards by steps and time in the CARLA environment

In this experiments it is shown a good performance of the PPO algorithm, shown in the graph, while the table metrics could point to inaccurate conclusions if seen alone. This is due to the fact that PPO achieves almost max rewards in much less steps and time. DQL achieves some stable results, and REINFORCE fails to converge within this time frame of training.

# 5 CONCLUSION

It was shown throughout this work a view of RL and IL applied with DL in simulation environments, with a developed software that allows an unified approach to implement and compare the performance of Machine Learning algorithms, specifically Deep Learning, in many environments integrated in this work's scalable software. This framework can be used to perform many experiments to collect standardized results data and perform statistical analysis with defined metrics.

From the data collected from the experiments, it is observed that the analysis of the IL structure is compatible to RL when they are run in the same type of environment, as seen by the sharing of many theoretical similarities. However, since the applications of both classes are usually different, there is some difficulty in comparing the performance. IL is presented as the best choice when we want to easily demonstrate the intent of the tasks, but the current available simulation platforms make it difficult to record human trajectories, which does not facilitate research between these two classes. In addition, IL algorithms are usually limited to not outperform the demonstrator. Despite these systems currently being more focused on RL tasks, some of them, such as Duckietown, Unity ML and CARLA, are already enabling more tools with both approaches, and so an increase of a IL approach in these systems is expected.

The chosen implemented algorithms and tests have shown that it is possible to unify the RL and IL approaches, and that simulating algorithms with different simulation platforms and tasks can allow for deeper insights from the data. The work showed the advantages of unifying the way to implement, test and analyze these types of algorithms, and allowed the development of a scalable system for this purpose. In addition, a consideration of the metrics used for comparison was performed, which showed that algorithms can have different performance depending on which variable is taken into consideration. Many papers usually compare algorithms based on the most convenient metric to be compared with state-of-the-art, but does not have a broader performance analysis.

Today, interesting new approaches, such as Curiosity (PATHAK et al., 2017), are being developed, and the fast development of such approaches by researchers is necessary. Some initiatives have created frameworks to encourage this faster development of ML (DUAN et al., 2016) (BELLEMARE et al., 2018), but do not yet have much scalability and compatibility between RL, IL and the various existing simulation environments.

### Directions for Future Work

Some points are considered by the author as directions for continuity of work:

- Integration of more systems and implementation of more algorithms within the framework.
- Implementation and comparison of recent RL techniques, which can result in improvements when combined with other RL algorithms, e.g. (PATHAK et al., 2017) (ANDRYCHOWICZ et al., 2017).
- Hyperparameter optimization and measurement of the algorithm's robustness against hyperparameter's influences.
- Performance comparison of the Generative Adversarial Imitation Learning variations using other GANs approaches such as (HJELM et al., 2017).
- Study of sub-optimality in the demonstrated trajectories used in Imitation Learning, with approaches such as Multiple Hypothesis Predictions (RUPPRECHT et al., 2017).
- Development of new simulation environments with more complex tasks and types of environment models, beyond the today's standart of Atari games and MuJoCo single-objective tasks.
- Study of the different types of metrics used in to compare RL and IL in recent papers, and definition of the most appropriate ones considering each objective.

# BIBLIOGRAPHY

ABADI, M. et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. **arXiv preprint arXiv:1603.04467**, 2016.

ABBEEL, P.; NG, A. Y. Apprenticeship learning via inverse reinforcement learning. In: ACM. **Proceedings of the twenty-first international conference on Machine learning**. [S.l.], 2004. p. 1.

ANDRYCHOWICZ, M. et al. Hindsight experience replay. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 5048–5058.

BARAM, N. et al. End-to-end differentiable adversarial imitation learning. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2017. p. 390–399.

BEATTIE, C. et al. Deepmind lab. **arXiv preprint arXiv:1612.03801**, 2016.

BELLEMARE, M. G. et al. **Dopamine**. 2018. Disponível em: <https://github.com/google/dopamine>.

_____. The arcade learning environment: An evaluation platform for general agents. **Journal of Artificial Intelligence Research**, v. 47, p. 253–279, 2013.

BELLMAN, R. A markovian decision process. **Indiana Univ. Math. J.**, v. 6, p. 679–684, 1957. ISSN 0022-2518.

BILLARD, A. et al. Robot programming by demonstration. In: **Springer handbook of robotics**. [S.l.]: Springer, 2008. p. 1371–1394.

BOUTILIER, C. Planning, learning and coordination in multiagent decision processes. In: MORGAN KAUFMANN PUBLISHERS INC. **Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge**. [S.l.], 1996. p. 195–210.

BROCKMAN, G. et al. Openai gym. **arXiv preprint arXiv:1606.01540**, 2016.

BROWNE, C. B. et al. A survey of monte carlo tree search methods. **IEEE Transactions on Computational Intelligence and AI in games**, IEEE, v. 4, n. 1, p. 1–43, 2012.

CAUDILL, M. Neural networks primer, part i. **AI expert**, Miller Freeman, Inc., v. 2, n. 12, p. 46–52, 1987.

CHANG, K.-W. et al. Learning to search better than your teacher. 2015.

CHEVALIER-BOISVERT, M. et al. **Duckietown Environments for OpenAI Gym**. [S.l.]: GitHub, 2018. <https://github.com/duckietown/gym-duckietown>.

CHOI, J.; KIM, K.-E. Nonparametric bayesian inverse reinforcement learning for multiple reward functions. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2012. p. 305–313.

COLLOBERT, R.; WESTON, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In: ACM. **Proceedings of the 25th international conference on Machine learning**. [S.l.], 2008. p. 160–167.

DAUMÉ, H.; LANGFORD, J.; MARCU, D. Search-based structured prediction. **Machine learning**, Springer, v. 75, n. 3, p. 297–325, 2009.

DOSOVITSKIY, A. et al. Carla: An open urban driving simulator. **arXiv preprint arXiv:1711.03938**, 2017.

DUAN, Y. et al. One-shot imitation learning. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2017. p. 1087–1098.

_____. Benchmarking deep reinforcement learning for continuous control. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2016. p. 1329–1338.

ENGINE, U. G. Unity game engine-official site. **Online][Cited: October 9, 2008.] http://unity3d. com**, p. 1534–4320, 2008.

FINN, C. et al. A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. **arXiv preprint arXiv:1611.03852**, 2016.

FINN, C.; LEVINE, S.; ABBEEL, P. Guided cost learning: Deep inverse optimal control via policy optimization. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2016. p. 49–58.

FOUNDATION, P. S. **Python Language Reference, version 3.5**. 2018. Disponível em: <http://www.python.org>.

FUJIMOTO, S.; HOOF, H. van; MEGER, D. Addressing function approximation error in actor-critic methods. **arXiv preprint arXiv:1802.09477**, 2018.

GHAHRAMANI, Z. Nonparametric bayesian methods. In: **Tutorial presentation at the UAI Conference**. [S.l.: s.n.], 2005.

GOODFELLOW, I. Nips 2016 tutorial: Generative adversarial networks. **arXiv preprint arXiv:1701.00160**, 2016.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016.

GOODFELLOW, I. et al. Generative adversarial nets. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2014. p. 2672–2680.

GU, S. et al. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In: IEEE. **Robotics and Automation (ICRA), 2017 IEEE International Conference on**. [S.l.], 2017. p. 3389–3396.

HAUSMAN, K. et al. Multi-modal imitation learning from unstructured demonstrations using generative adversarial nets. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 1235–1245.

HEBB, D. O. **The organization of behavior: A neuropsychological theory**. [S.l.: s.n.], 1949.

HENDERSON, P. et al. Deep reinforcement learning that matters. **arXiv preprint arXiv:1709.06560**, 2017.

HJELM, R. D. et al. Boundary-seeking generative adversarial networks. **arXiv preprint arXiv:1702.08431**, 2017.

HO, J.; ERMON, S. Generative adversarial imitation learning. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2016. p. 4565–4573.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural Comput.**, MIT Press, Cambridge, MA, USA, v. 9, n. 8, p. 1735–1780, nov. 1997. ISSN 0899-7667.

HUSSEIN, A. et al. Imitation learning: A survey of learning methods. **ACM Computing Surveys (CSUR)**, ACM, v. 50, n. 2, p. 21, 2017.

JOHNSON, M. et al. The malmo platform for artificial intelligence experimentation. In: **IJCAI**. [S.l.: s.n.], 2016. p. 4246–4247.

JULIANI, A. et al. Unity: A general platform for intelligent agents. **arXiv preprint arXiv:1809.02627**, 2018.

KAHN, G. et al. Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation. In: IEEE. **2018 IEEE International Conference on Robotics and Automation (ICRA)**. [S.l.], 2018. p. 1–8.

KEMPKA, M. et al. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In: IEEE. **Computational Intelligence and Games (CIG), 2016 IEEE Conference on**. [S.l.], 2016. p. 1–8.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **CoRR**, abs/1412.6980, 2014.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1097–1105.

LASKEY, M. et al. Iterative noise injection for scalable imitation learning. **arXiv preprint arXiv:1703.09327**, 2017.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, IEEE, v. 86, n. 11, p. 2278–2324, 1998.

LEVINE, S.; KOLTUN, V. Guided policy search. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2013. p. 1–9.

LI, Y.; SONG, J.; ERMON, S. Infogail: Interpretable imitation learning from visual demonstrations. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 3815–3825.

LILLICRAP, T. P. et al. Continuous control with deep reinforcement learning. **CoRR**, abs/1509.02971, 2015.

LIPTON, Z. C. The mythos of model interpretability. **arXiv preprint arXiv:1606.03490**, 2016.

LITJENS, G. et al. A survey on deep learning in medical image analysis. **Medical image analysis**, Elsevier, v. 42, p. 60–88, 2017.

LIU, Y. et al. Imitation from observation: Learning to imitate behaviors from raw video via context translation. In: IEEE. **2018 IEEE International Conference on Robotics and Automation (ICRA)**. [S.l.], 2018. p. 1118–1125.

MICHINI, B.; HOW, J. P. Bayesian nonparametric inverse reinforcement learning. In: SPRINGER. **Joint European Conference on Machine Learning and Knowledge Discovery in Databases**. [S.l.], 2012. p. 148–163.

MIOTTO, R. et al. Deep learning for healthcare: review, opportunities and challenges. **Briefings in bioinformatics**, 2017.

MNIH, V. et al. Asynchronous methods for deep reinforcement learning. In: **International conference on machine learning**. [S.l.: s.n.], 2016. p. 1928–1937.

_____. Playing atari with deep reinforcement learning. **CoRR**, abs/1312.5602, 2013.

_____. Human-level control through deep reinforcement learning. **Nature**, v. 518, n. 7540, p. 529–533, fev. 2015. ISSN 00280836.

MONAHAN, G. E. State of the art—a survey of partially observable markov decision processes: theory, models, and algorithms. **Management Science**, INFORMS, v. 28, n. 1, p. 1–16, 1982.

MURPHY, K. Machine learning: a probabilistic approach. **Massachusetts Institute of Technology**, p. 1–21, 2012.

NAIR, A. et al. Overcoming exploration in reinforcement learning with demonstrations. In: IEEE. **2018 IEEE International Conference on Robotics and Automation (ICRA)**. [S.l.], 2018. p. 6292–6299.

NG, A. Y. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In: ACM. **Proceedings of the twenty-first international conference on Machine learning**. [S.l.], 2004. p. 78.

NG, A. Y.; RUSSELL, S. J. et al. Algorithms for inverse reinforcement learning. In: **Icml**. [S.l.: s.n.], 2000. p. 663–670.

PATHAK, D. et al. Curiosity-driven exploration by self-supervised prediction. In: **International Conference on Machine Learning (ICML)**. [S.l.: s.n.], 2017. v. 2017.

RANCHOD, P.; ROSMAN, B.; KONIDARIS, G. Nonparametric bayesian reward segmentation for skill discovery using inverse reinforcement learning. In: IEEE. **Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on**. [S.l.], 2015. p. 471–477.

RESENDE, N. F. **A unified framework of Deep Reinforcement Learning and Deep Imitation Learning in simulation environments**. 2018. Disponível em: <http://www.github.com/NiloFreitas>.

ROSS, S.; BAGNELL, D. Efficient reductions for imitation learning. In: **Proceedings of the thirteenth international conference on artificial intelligence and statistics**. [S.l.: s.n.], 2010. p. 661–668.

ROSS, S.; BAGNELL, J. A. Reinforcement and imitation learning via interactive no-regret learning. **arXiv preprint arXiv:1406.5979**, 2014.

ROSS, S.; GORDON, G.; BAGNELL, D. A reduction of imitation learning and structured prediction to no-regret online learning. In: **Proceedings of the fourteenth international conference on artificial intelligence and statistics**. [S.l.: s.n.], 2011. p. 627–635.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **nature**, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986.

_____. Learning representations by back-propagating errors. **nature**, Nature Publishing Group, v. 323, n. 6088, p. 533, 1986.

RUPPRECHT, C. et al. Learning in an uncertain world: Representing ambiguity through multiple hypotheses. In: **International Conference on Computer Vision (ICCV)**. [S.l.: s.n.], 2017.

RUSSELL, S. Learning agents for uncertain environments. In: ACM. **Proceedings of the eleventh annual conference on Computational learning theory**. [S.l.], 1998. p. 101–103.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM Journal of research and development**, IBM, v. 3, n. 3, p. 210–229, 1959.

SCHAAL, S. Is imitation learning the route to humanoid robots? **Trends in cognitive sciences**, Elsevier, v. 3, n. 6, p. 233–242, 1999.

SCHULMAN, J. et al. Trust region policy optimization. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2015. p. 1889–1897.

_____. High-dimensional continuous control using generalized advantage estimation. **arXiv preprint arXiv:1506.02438**, 2015.

_____. Proximal policy optimization algorithms. **arXiv preprint arXiv:1707.06347**, 2017.

SILVER, D. et al. Deterministic policy gradient algorithms. In: **Proceedings of the 31st International Conference on Machine Learning (ICML-14)**. [S.l.: s.n.], 2014. p. 387–395.

_____. Mastering the game of go without human knowledge. **Nature**, Nature Publishing Group, v. 550, n. 7676, p. 354, 2017.

SONG, J. et al. Learning to search via self-imitation. **arXiv preprint arXiv:1804.00846**, 2018.

ŠOŠIĆ, A. et al. Inverse reinforcement learning via nonparametric spatio-temporal subgoal modeling. **arXiv preprint arXiv:1803.00444**, 2018.

SRIVASTAVA, N. et al. Dropout: A simple way to prevent neural networks from overfitting. **J. Mach. Learn. Res.**, JMLR.org, v. 15, n. 1, p. 1929–1958, jan. 2014. ISSN 1532-4435.

STADIE, B. C.; ABBEEL, P.; SUTSKEVER, I. Third-person imitation learning. **arXiv preprint arXiv:1703.01703**, 2017.

SUTTON, R. S. Learning to predict by the methods of temporal differences. **Machine learning**, Springer, v. 3, n. 1, p. 9–44, 1988.

SUTTON, R. S.; BARTO, A. G. **Reinforcement learning: An introduction**. [S.l.]: MIT press, 2018.

SUTTON, R. S. et al. Policy gradient methods for reinforcement learning with function approximation. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2000. p. 1057–1063.

TESAURO, G. Temporal difference learning and td-gammon. **Communications of the ACM**, v. 38, n. 3, p. 58–68, 1995.

TODOROV, E.; EREZ, T.; TASSA, Y. Mujoco: A physics engine for model-based control. In: IEEE. **Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on**. [S.l.], 2012. p. 5026–5033.

VINYALS, O. et al. Starcraft ii: A new challenge for reinforcement learning. **arXiv preprint arXiv:1708.04782**, 2017.

WANG, Z. et al. Robust imitation of diverse behaviors. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2017. p. 5326–5335.

WATKINS, C. J.; DAYAN, P. Q-learning. **Machine learning**, Springer, v. 8, n. 3-4, p. 279–292, 1992.

WATKINS, C. J. C. H. **Learning from Delayed Rewards**. 1989. Tese (Doutorado) — King's College, Cambridge, UK, 1989.

WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In: **Machine Learning**. [S.l.: s.n.], 1992. p. 229–256.

WU, Y. et al. Building generalizable agents with a realistic and rich 3d environment. **arXiv preprint arXiv:1801.02209**, 2018.

ZIEBART, B. D. et al. Maximum entropy inverse reinforcement learning. In: CHICAGO, IL, USA. **AAAI**. [S.l.], 2008. v. 8, p. 1433–1438.