INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Luckeciano Carvalho Melo

A DEEP REINFORCEMENT LEARNING METHOD FOR HUMANOID KICK MOTION

Final Paper
2018

Course of Computer Engineering

CDU 621.38

Luckeciano Carvalho Melo

A DEEP REINFORCEMENT LEARNING METHOD FOR HUMANOID KICK MOTION

Advisor

Prof. Dr. Adilson Marques da Cunha (ITA)

Co-advisor

Prof. Dr. Marcos R. O. de A. Máximo (ITA)

COMPUTER ENGINEERING

São José dos Campos
Instituto Tecnológico de Aeronáutica

2018

Cataloging-in Publication Data
Documentation and Information Division

Carvalho Melo, Luckeciano
A Deep Reinforcement Learning Method for Humanoid Kick Motion / Luckeciano Carvalho Melo.
São José dos Campos, 2018.
99f.

Final paper (Undergraduation study) – Course of Computer Engineering– Instituto Tecnológico de Aeronáutica, 2018. Advisor: Prof. Dr. Adilson Marques da Cunha. Co-advisor: Prof. Dr. Marcos R. O. de A. Máximo.

1. Deep Reinforcement Learning. 2. Robotics. 3. Artificial Intelligence. I. Instituto Tecnológico de Aeronáutica. II. A Deep Reinforcement Learning Method for Humanoid Kick Motion.

BIBLIOGRAPHIC REFERENCE

CARVALHO MELO, Luckeciano. **A Deep Reinforcement Learning Method for Humanoid Kick Motion**. 2018. 99f. Final paper (Undergraduation study) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

AUTHOR'S NAME: Luckeciano Carvalho Melo
PUBLICATION TITLE: A Deep Reinforcement Learning Method for Humanoid Kick Motion.
PUBLICATION KIND/YEAR: Final paper (Undergraduation study) / 2018

It is granted to Instituto Tecnológico de Aeronáutica permission to reproduce copies of this final paper and to only loan or to sell copies for academic and scientific purposes. The author reserves other publication rights and no part of this final paper can be reproduced without the authorization of the author.

Luckeciano Carvalho Melo
H8A St., 113
12.228-460 – São José dos Campos–SP

A DEEP REINFORCEMENT LEARNING METHOD FOR HUMANOID KICK MOTION

This publication was accepted like Final Work of Undergraduation Study

Luckeciano Carvalho Melo

Author

Adilson Marques da Cunha (ITA)

Advisor

Marcos R. O. de A. Máximo (ITA)

Co-advisor

Prof^{a}.Dr^{a}. Cecília César
Course Coordinator of Computer Engineering

São José dos Campos: JUNE 12, 2018.

Abstract

Controlling high degrees of freedom for humanoid robot is acknowledged as one of the hardest problems in Robotics. Due to the lack of mathematical models, an approach frequently employed is to rely on human intuition to design keyframe movements by hand, usually aided by graphical tools. In this preliminary work, we propose a learning framework based upon neural networks, in order to mimic humanoid robot movements. The developed technique does not make any assumption about the underlying implementation of the movement. Therefore, both keyframe and model-based motions may be learned. The framework was applied in the RoboCup 3D Soccer Simulation domain and some promising results were obtained, by using the same network architecture for several motions, even when copying motions from another teams.

List of Figures

FIGURE 1.1 – AlphaGo Zero, learning model that beat the best players of Go, Chess and Shogi, learning to play without previous human knowledge (SILVER *et al.*, 2017a). 14

FIGURE 1.2 – Locomotion of Agent via Deep Reinforcement Learning (HEESS *et al.*, 2017b). 14

FIGURE 1.3 – A Snapshot from the RoboCup Soccer 3D Simulation League. 14

FIGURE 2.1 – Illustration of an actual optimization run with covariance matrix adaptation on a simple two-dimensional problem (KHAN, 2018). . . 20

FIGURE 2.2 – Human level control through deep reinforcement learning in atari games (MNIH *et al.*, 2015). 21

FIGURE 2.3 – OpenAI Five network architecture (BROCKMAN *et al.*, 2018). 22

FIGURE 3.1 – An artificial neuron within a feed forward artificial neural network (TANIKIC; DESPOTOVIC, 2012). 23

FIGURE 3.2 – An artificial neuron in detail. 24

FIGURE 3.3 – Function representation: in (a), we have a sigmoid function where weights vary. In (b), the same sigmoid but only bias varies. 25

FIGURE 3.4 – Neural Network representation. 25

FIGURE 3.5 – Illustration of Sigmoid function. 28

FIGURE 3.6 – Illustration of *tanh* function. 28

FIGURE 3.7 – Illustration of ReLU function. 29

FIGURE 3.8 – Illustration of leaky ReLU function. 30

FIGURE 3.9 – Illustration of gradient descent in a two variables optimization (MAGALHAES, 2017). 32

FIGURE 3.10 –Illustration of Stochastic Gradient Descent optimization. 35

FIGURE 3.11 –Illustration of how each variation of Gradient Descent commonly behaves. (DABBURA, 2017). 35

FIGURE 3.12 –Illustration of how Gradient Descent with Momentum (blue arrows) and without it (black arrows) (MAGALHAES, 2017). 37

FIGURE 3.13 –Illustration Gradient Descent divergence for a one variable optimization. (MAGALHAES, 2017). 42

FIGURE 4.1 – Reinforcement Learning System. 46

FIGURE 4.2 – The agent–environment interaction in a Markov decision process (SUTTON; BARTO, 1998). 51

FIGURE 4.3 – Backup diagram for v_π. The white nodes are states, while the black ones are possible actions. Bellman equation average over the actions in a recursive way. (SUTTON; BARTO, 1998). 52

FIGURE 4.4 – Backup diagram in Bellman Optimality Equation for v_π and q_π. (SUTTON; BARTO, 1998). 54

FIGURE 4.5 – Policy Iteration algorithm. (SILVER, 2015b). 55

FIGURE 4.6 – Actor-Critic model. (XU *et al.*, 2007). 62

FIGURE 5.1 – In the hybrid model, we can ensure the starting point is near of the optimal solution (orange arrow); otherwise, the starting point can be bad and harder to optimize (red arrow). 70

FIGURE 5.2 – The architecture of a neural network designed to learn motions. . . 73

FIGURE 5.3 – Intuition behind input normalization 75

FIGURE 5.4 – Architecture used by pure Reinforcement Learning models. 75

FIGURE 5.5 – Gaussian noise applied to action space to ensure better exploration in continuous environments. (PLAPPERT *et al.*, 2017) 76

FIGURE 5.6 – Task used to learning kick motion. 77

FIGURE 5.7 – Reinforcement Learning Server Architecture. (MUZIO, 2017) 77

FIGURE 5.8 – Data flow graph for PPO algorithm. 80

FIGURE 5.9 – “Pi” data flow node from PPO. 80

FIGURE 5.10 –Policy data flow graph from Figure 5.4 81

FIGURE 5.11 –Policy data flow graph from Figure 5.2 81

FIGURE 5.12 –Master-workers architecture for data parallelism. 82

FIGURE 5.13 –Synchronous and Asynchornous Distributed Training. 83

FIGURE 5.14 –Monitoring training metrics with Tensorboard. 84

FIGURE 6.1 – Plots of mean squared error and mean absolute error, during training. 85

FIGURE 6.2 – The kick motion. The first row of figures shows the original kick motion. The second row shows the learned kick motion. Both motions are visually indistinguishable. 86

FIGURE 6.3 – Joint values for comparing original and learned kicks. The neural network was able to fit the joint trajectories with small errors. . . . 87

FIGURE 6.4 – Joints positions, during a period of the walking motion for the original walk, and the learned walk and the joints positions effectively attained, during the learned walking motion. 88

FIGURE 6.5 – The walking motions comparison. Figure (a) shows our agent in its regular walk, Figure (b) shows the same agent mimicking UT Austin Villa walk, and Figure (c) shows the UT Austin Villa agent itself performing his own walking motion. 89

List of Tables

TABLE 5.1 – The Network Summary 73
TABLE 5.2 – The Reinforcement Learning Network Summary 75

TABLE 6.1 – The Kick Comparison 87
TABLE 6.2 – Walk Comparison - Forward Walk 89

Contents

1 Introduction 13

1.1 Motivation 13

1.2 Contextualization 13

1.3 Objective 15

1.4 Scope 15

1.5 Organization of this work 15

2 Literature Review 17

2.1 The RoboCup Soccer3D Simulation League 17

2.1.1 Domain Description 17

2.1.2 Kick Motion 17

2.1.3 Keyframe Movements 18

2.1.4 Optimization Techniques 19

2.2 Reinforcement Learning for Control 20

3 Deep Learning Background 23

3.1 Neural Networks 23

3.1.1 A Neuron 24

3.1.2 Neural Network Representation 25

3.1.3 Vectorization 26

3.2 Activation Functions 27

3.2.1 Logistic Sigmoid 27

3.2.2 Hyperbolic Tangent 27

3.2.3 Rectified Linear Unit - ReLU 29

3.2.4 Leaky ReLU 30
3.3 Cost Function 30
3.4 Gradient Descent 31
3.5 Backpropagation 33
3.6 Optimization Algorithms 33
3.6.1 Batch, Mini-batch and Stochastic Gradient Descent 34
3.6.2 Momentum 36
3.6.3 RMSProp 38
3.6.4 Adam 39
3.7 Weights Random Initialization 39
3.7.1 Xavier Initialization 41
3.8 Gradient Descent convergence and learning rate decay 41
4 Reinforcement Learning Background 44
4.1 Concepts of a Reinforcement Learning System 44
4.2 Reinforcement Learning System 45
4.2.1 Reward 45
4.2.2 State 46
4.2.3 Policy 47
4.2.4 Value Function 47
4.2.5 Model 48
4.3 Markov Decision Process 48
4.3.1 Markov State 48
4.3.2 State Transition Matrix 49
4.3.3 Markov Decision Process 49
4.3.4 State-Value and Action-Value Function 51
4.4 Bellman Equation 52
4.4.1 Bellman Expectation Equation 52
4.4.2 Bellman Optimality Equation 53
4.5 Exact Solution Methods 55

4.5.1 Policy Iteration 55
4.5.2 Value Iteration 57
4.5.3 Limitations of Exact Solution Methods 58
4.6 Policy Gradient Methods 59
4.6.1 Policy Gradient Theorem 60
4.6.2 Actor-Critic Models 61
4.6.3 Advantage Function and GAE Algorithm 63
4.7 Advanced Policy Gradient Methods 64
4.7.1 Optimization Loss 64
4.7.2 Trust Region Policy Optimization – TRPO 65
4.7.3 Proximal Policy Optimization – PPO 66

5 Methodology 69
5.1 The Kick Motion Problem 69
5.2 Experimentation Setup 69
5.2.1 Hybrid Learning Model – HLM 70
5.2.2 Reinforcement with Naive Reward - RNR 70
5.2.3 Reinforcement with Reference Reward - RRR 71
5.2.4 Reinforcement with Initial State Distribution - RISD 71
5.2.5 Reinforcement with Early Termination - RET 72
5.3 Supervised Learning Setup 72
5.3.1 The Dataset 72
5.3.2 Neural Network Architecture and Hyperparameters 72
5.3.3 The Training Procedure 73
5.3.4 The Deployment in the Soccer 3D Environment 74
5.4 Reinforcement Learning Setup 74
5.4.1 Policy Representation 74
5.4.2 Task Description 76
5.5 Infrastructure 77
5.5.1 Reinforcement Learning Server 77

5.5.2 Neural Network Deployment 78
5.5.3 Distributed Training 82
5.5.4 Metrics 84
5.5.5 Monitoring via Tensorboard 84

6 Results' Analysis and Discussion 85
6.1 Training Results 85
6.2 The Learned Kick Motion 86
6.3 The Learned Walk Motion 88
6.4 Other motions 89

7 Conclusions, Recommendations, and Future Works . . 91
7.1 Preliminary Conclusions and Future Works 91
7.2 The Activities Plan 91

Bibliography 93

1 Introduction

1.1 Motivation

Robotics is understood as an important area of research within Engineering and Computer Science, optimizing and automating various areas of industry. One of the ways in which research is developed in this area is in robot soccer, since it comprises challenges of machine perception, environment modeling, planning and reasoning, control, and multiagent strategy.

Over the years, several techniques have been developed to address each of the problems related to robot soccer, based on the theory of Signal Processing, Control, Trajectory Planning, and classical Artificial Intelligence. These techniques have proved to be functional for maturing the challenge. However, such techniques still perform worse as compared to humans in these activities.

In recent years, however, with the development of processing and memory architectures, Machine Learning techniques have been able to achieve or even have surpassed the human performance in machine perception activities (Computer Vision (LU; TANG, 2014) and Speech Recognition (XIONG *et al.*, 2016)), by planning and reasoning (SILVER *et al.*, 2017a), as shown in Figure 1.1, and also by controlling agent locomotion (HEESS *et al.*, 2017b), as shown in Figure 1.2. In this way, the learning field of combining techniques from Deep Learning and Reinforcement Learning appears as a great candidate in search of General Artificial Intelligence.

1.2 Contextualization

RoboCup is an international scientific community aiming to advance the state of the art of intelligent robots. Its mission is that a team of robots will be able to beat the human team champion of the World Cup until the year 2050 (KITANO *et al.*, 1998). Since this is a goal with a range of different challenges to be solved, there are several categories of competition within the community.

FIGURE 1.1 – AlphaGo Zero, learning model that beat the best players of Go, Chess and Shogi, learning to play without previous human knowledge (SILVER *et al.*, 2017a).

FIGURE 1.2 – Locomotion of Agent via Deep Reinforcement Learning (HEESS *et al.*, 2017b).

In Robocup Soccer 3D Simulation League, as shown in Figure 1.3, there is a robot soccer competition in which each team has eleven Nao robots in a physical simulation environment called Simspark (OBST; ROLLMAN, 2005). The purpose in this case is to create and improve algorithms and physical models for the perception, locomotion, and strategy of the humanoid robot, before actually shipping them into hardware for real-world evaluation.

FIGURE 1.3 – A Snapshot from the RoboCup Soccer 3D Simulation League.

Over the last few years, team efforts have been seen in three different areas, layers, or strands. The first area is considered more basic, meaning the construction of the agent that can model the environment and interact with it – involving both the construction of a solid software architecture and the application of traditional techniques of localization and control of the humanoid robot (MACALPINE *et al.*, 2011).

The second layer, explored at the highest level, involves creating behaviors to perform actions within the game, given the modeled environment – from the creation of models and heuristics for navigation to the creation of multiagent strategies for positioning and marking (MACALPINE; STONE, 2016).

The third strand, to be approached in this work, is the creation and optimization of models based upon learning for activities such as robot walking and kicking. Simulated categories have great value for learning test, mainly because they provide a benchmark for comparison and do not involve physical robots. Historically, teams with the best performance in these two issues are usually the best positioned within category competitions, due to the fact that they are able to maintain greater possession of the ball and are more offensive to the opposing goal.

1.3 Objective

Inspired from the context of the RoboCup 3D Simulation League and based upon some results from the recent techniques of Deep and Reinforcement Learning, this work aims to develop and evaluate some new learning models, based on Deep Reinforcement Learning, for the task of making a humanoid robot to kick the soccer ball inside the RoboCup Soccer 3D Simulation environment.

1.4 Scope

In this work, Reinforcement Learning algorithms applied to models based upon Deep Neural Networks will be approached, in order to find, through gradient-based optimization techniques, optimal policies for kick control of the humanoid robot. These will be contrasted with classic control techniques coupled with evolutionary optimization strategies, widely used in the context of the RoboCup Soccer 3D Simulation League.

1.5 Organization of this work

This work is organized as follows: Chapter 2 will describe the RoboCup Soccer 3D Simulation League and the traditional methods used for the kick motion. Chapter 3 will cover the theory behind Deep Learning used in this work. Chapter 4 will cover the theory behind Reinforcement Learning. Chapter 5 will explain all the methods and tooling used in experimentation. Chapter 6 describes the experimentation itself, detailing problem modeling, test scenarios, and the main results. Finally, Chapter 7 will share some

conclusions and suggestions for future work.

2 Literature Review

2.1 The RoboCup Soccer3D Simulation League

2.1.1 Domain Description

The RoboCup Soccer 3D Simulation League (Soccer 3D) is a particularly interesting challenge concerning humanoid robot soccer. It consists of a simulation environment for a soccer match with two teams, each one composed by up to 11 simulated Nao robots (GOUAILLIER *et al.*, 2009), the official robot used for the RoboCup Standard Platform League since 2008. The Soccer 3D is interesting for robotics research, since it involves high level multi-agent cooperative decision making, while providing a physically realistic environment, which requires control and signal processing techniques for robust low level skills.

The RoboCup 3D simulation environment is based on SimSpark (XU; VATANKHAH, 2014) , a generic physical multi-agent system simulator. SimSpark uses the Open Dynamics Engine (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. The Nao robots has height of approximately 57 cm and 4.5 kilograms. The agent send torque commands to the simulator and receive perceptual data. Each robot has 22 joints with perceptors and effectors. Joints information are communication between agent and server, visual information and between agents happens in in the frequency of 50 Hz, 16.7 Hz and 25 Hz, respectively (MACALPINE *et al.*, 2012).

2.1.2 Kick Motion

In the current level of the Soccer 3D evolution, motion control is a key factor in team's performance. Indeed, controlling a high degrees of freedom for a humanoid robot is acknowledged as one of the hardest problems in Robotics. Much effort has been devised to humanoid robot walking, where researchers have been very successful in designing control algorithms which reason about reduced order mathematical models based upon the Zero Moment Point (ZMP) concept, such as the linear inverted pendulum model (KAJITA *et al.*,

2001). Nevertheless, these techniques restrict the robot to operate under a small region of its dynamics, where assumptions of the simplified models are still valid (COLLINS *et al.*, 2005; MUNIZ *et al.*, 2016).

Therefore, model-based techniques are hard to use for designing highly dynamic movements, such as long distance kicks and goalkeeper dives to defend goals from fast moving balls. In the robot soccer domain, a common approach for these movements is to employ keyframe movements, where motions are composed by sequences of robot postures. In this case, movements are designed off-line and executed in an open-loop fashion in execution time.

Due to the lack of mathematical models, an approach frequently employed is to rely on human intuition to design keyframe movements by hand, usually aided by graphical tools. However, this process is difficult, time consuming, and is often unable to obtain high performance motions given the high dimensionality of the search space. Other possible solution is to use motion to capture data from humans (SHON *et al.*, 2005), which has its own challenges due to the fact that the kinematic and dynamic properties of a humanoid robot differs greatly from those of a human.

2.1.3 Keyframe Movements

Definition 1 *A* keyframe $\mathbf{k} = [j_1, j_2, \ldots, j_n]^T \in K \subseteq \mathbb{R}^n$ *is an ordered set of joint angular positions, where K and n are the joint space and the number of degrees of freedom of the robot.*

Definition 2 *A* keyframe step *is an ordered pair* $\mathbf{s} = (\mathbf{k}, t) \in S = K \times \mathbb{R}$*, where* $\mathbf{k}$ *is a keyframe and t is the time when the keyframe must be achieved with respect to the beginning of the movement.*

Definition 3 *A* keyframe movement*, or simply a movement, is defined as* $\mathbf{m} = (\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_\gamma, r) \in M = S^\gamma \times \mathbb{R}$*, where γ and r are the number of keyframe steps and the speed rate of the movement. In this representation, we assume the movement starts at time 0 and the first keyframe step represents the robot posture at the beginning of the movement. Therefore,* $t_1 = 0$ *and each time* $t_i, \forall i \geq 2$ *is a time since the beginning of the movement.*

Keyframe movements are executed in an open-loop fashion, where joint positions are computed through interpolation of keyframe steps based upon the current time. If the interface to the robot joints is not position-based, local controllers may be used to track the position references issued by the keyframe. For example, in the Simspark simulator, the simulated Nao Robot has speed-controlled joints, therefore, we use simple proportional controllers for each joint to track desired joint positions. To obtain smooth joint

trajectories, we interpolate keyframe steps, by using cubic splines (BARTELS *et al.*, 1987), which are functions of class $\mathcal{C}^2$.

Finally, when a keyframe movement is requested, joint positions are often far away from movement's initial joint positions. Hence, by simply executing the keyframe movement in this case would result in high joints accelerations, which would probably make the robot to fall. To avoid this from happening, a transition movement based on linear interpolation is first employed to bring the joint positions to the initial joint positions required by the keyframe movement.

2.1.4 Optimization Techniques

In Soccer 3D environment, optimization techniques plays a important role to achieve competitive behaviors, because they are use to obtain faster and more robust motions.

Motion optimization means find the best values for each joint in the time instant during the whole motion. There's a lot of parameters involved, what turns manual tuning impossible. The common way to do this is modeling the motion and use optimization algorithms to find the best parameters for this model. For the kick motion, it means find the best keyframe and interpolator values. In the case of walking motion, we optimize the walk parameters, such as torso height, period and step size.

In the robotics simulation world, it's common to optimize using evolution strategies. (MAXIMO, 2012) used Particle Swarm Optimization to optimize walk parameters. (URIELI *et al.*, 2011) compared the performance of several algorithms in Soccer3D context, such as Hill Climbing (HC), Cross-Entropy Method (CEM) (RUBINSTEIN; KROESE, 2004), Genetic Algorithm and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (HANSEN, 2016). The last one turns out to be the best for this kind of task and therefore is the most common in this environment.

The CMA-ES algorithm is a policy search algorithm that generates a population of parameter sets – also known as "candidates" – sampled from a multivariate Gaussian distribution. These sets are evaluated with respect of a fitness measure. When all the candidates in group are evaluated, the mean of the multivariate Gaussian distribution is recalculated as a weighted average of the candidates with the highest fitnesses. The covariance matrix of the distribution is also updated to bias the generation of the next set of candidates toward directions of previously successful search steps (URIELI *et al.*, 2011), as shown in Figure 2.1. Even the algorithm works well for kick and walk motions, it doesn't scale well for hundreds or thousands of parameters (MACALPINE; STONE, 2017).

FIGURE 2.1 – Illustration of an actual optimization run with covariance matrix adaptation on a simple two-dimensional problem (KHAN, 2018).

2.2 Reinforcement Learning for Control

In this work, we will use more recent techniques that are able to optimize in this scale using policy gradient based algorithm instead of evolution strategies. These algorithms are based on Reinforcement Learning theory applied for control tasks.

Classic reinforcement learning algorithms are called tabular solution methods and uses dynamic programming for map state to actions. The most famous algorithms are Monte-Carlo methods, Temporal-Difference learning, SARSA (RUMMERY; NIRANJAN, 1994) and Q-Learning (WATKINS, 1989). These algorithms are good to solve toy problems, such as grid worlds and multi-armed bandits. However, they don't scale well for problems with greater state space.

However, these methods were improved using the idea of function approximation instead of tabular solutions. The new algorithms started to use other learning representations, since linear combination of features until neural networks. The last one was a huge improvement in the Reinforcement Learning field. With the arise of Deep Q-Networks (Figure 2.2), we were able to surpass human-level performance in several Atari games, as show in (MNIH *et al.*, 2015).

With the arise of Deep Learning and its new techniques and the development of new computer architectures such as GPUs and TPUs, reinforcement learning has been able to scale as well, surpassing top players from the best table games. For example, the AlphaGo (SILVER *et al.*, 2017c) won from the human top player of Go, considered the hardest tabular game. It used a Monte-Carlo Tree Search with a Deep Neural Network to find a optimal policy through self-play and without any human knowledge. The evolution of AlphaGo, called AlphaZero (SILVER *et al.*, 2017b) was able to learn not just Go, but also Chess and Shogi within a couple of hours.

FIGURE 2.2 – Human level control through deep reinforcement learning in atari games (MNIH *et al.*, 2015).

On the other side, all the tasks mentioned before are discrete control. The problem of kick motion is related to continuous control. In this kind of problem, there's no discrete actions, but a interval of them. Therefore, the algorithms are different and must satisfact new optimization challenges.

In the last years, several algorithms arised in order to improve the performance in control tasks. These algorithms are commonly applied to MuJoCo environment (TODOROV *et al.*, 2012), the most famous benchmark for continuous control tasks. Among such methods, we highlight A2C (MNIH *et al.*, 2016), ACER (WANG *et al.*, 2016), DDPG (LILLICRAP *et al.*, 2015), GAIL (HO; ERMON, 2016), HER (ANDRYCHOWICZ *et al.*, 2017), TRPO (SCHULMAN *et al.*, 2015a) and PPO (SCHULMAN *et al.*, 2017). The TRPO and PPO algorithms will be covered in deep later, and the last one will be used in the learning framework proposed in this work, since it has proved to perform better in MuJoCo benchmark.

In terms of applications, (HEESS *et al.*, 2017b) applied a distributed variation of PPO in MuJoCo and were able to learn parkour movements and run without in a model free way. These motions, however, are weird, asymmetric and, sometimes, unstable. (PENG *et al.*, 2018) applied the same algorithm but with key modifications in the reward function and optimization process, resulting in better and more human motions.

The most recent and impressive application of PPO was in OpenAI Five (BROCKMAN *et al.*, 2018). It consists in five independent neural networks that form a team to play the online game Dota 2. This team was trained in the equivalent of 180 years of continuous self-play, using 256 GPUs and 128 thousand of CPUs, and won from the 99.95^{th} percentile human player team in a restricted environment. This is impressive not just because the environment is challenging – due to the partial observability, long time horizon and high-dimensional, continuous spaces – but also because of the multi-agent strategy layer learned. The network architecture used by OpenAI Five is shown in Figure 2.3.

FIGURE 2.3 – OpenAI Five network architecture (BROCKMAN *et al.*, 2018).

3 Deep Learning Background

3.1 Neural Networks

Neural Networks are a learning representation which the goal is to approximate some function f^*. Data collected from an environment encodes an underlying function $\mathbf{y} = f^*(\mathbf{x})$ that maps an input $\mathbf{x}$ to an output $\mathbf{y}$, which may be a category from a classifier or a continue value in regression problems. The neural network defines an approximate mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$, by learning the values of the parameters $\boldsymbol{\theta}$, which result in the best function approximation. Figure 3.1 shows a neural network and an artificial neuron in detail.

FIGURE 3.1 – An artificial neuron within a feed forward artificial neural network (TANIKIC; DESPOTOVIC, 2012).

These networks are typically represented by composing together many different functions, which are associated with a directed acyclic graph, by describing a computational model. For example, we might have three layers (each of them representing a function $f^{(1)}, f^{(2)}$, and $f^{(3)}$), connecting in a chain and resulting in a final representation $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$.

During a neural network training, the objective is to adjust $f(\mathbf{x})$ to match $f^*(\mathbf{x})$, by

using the training dataset, which provides noisy examples of $f^*(\mathbf{x})$ evaluated in different points. The training examples directly specify what the output layer must do at each point $\mathbf{x}$, but the learning algorithm must decide how to use all layers to produce this desired output (GOODFELLOW *et al.*, 2016).

3.1.1 A Neuron

The idea of a neural network starts in the concept of a neuron. Figure 3.2 illustrates it in detail.

FIGURE 3.2 – An artificial neuron in detail.

In a neuron, the first operation is the product between inputs and the respective weights, which represents how much each input signal will influence the computation of that neuron. The core idea of a neural network is calculate the value of those weights in order to generate a better representation of the data generator distribution.

To this linear combination, it's added a bias value, which purpose is to add the capacity of neuron representation by translation, due to the fact this value doesn't rely on the inputs. Figure 3.3 shows how weights and bias affect the function representation, described by equation 3.1.

$$z = w^T x + b \tag{3.1}$$

The final operation from a neural is the activation function computed using the linear combination as input. In the context of neural networks, this is a non-linear function whose purpose is add the neuron's capacity of representation. Without this idea, the network would be only able to learn linear functions, which will result in poor representation of complex data distributions. Equation 3.2 shows the neuron activation. These functions

FIGURE 3.3 – Function representation: in (a), we have a sigmoid function where weights vary. In (b), the same sigmoid but only bias varies.

will be described deeper in section 3.2.

$$a = \sigma(z) \tag{3.2}$$

3.1.2 Neural Network Representation

A neural network is a combination of neurons in several layers. It's basically a directed acyclic graph (DAG) where each node performs the operations described in subsection 3.1.1. Figure 3.4 shows the representation of a neuron.

FIGURE 3.4 – Neural Network representation.

It is worth to mention the math notation used in this work. The neuron a_i^j represents

the ith neuron in the jth layer of network.

The flow of a input tensor in a neural network occurs in the following manner: first, neurons from the same layer performs its calculations. Then, the output of them will serve as input of the next layer. This sequence will keep going until reach the output layer. Equations 3.3 to 3.6 represent the flow from Figure 3.4.

$$z^{[1]} = W^{[1]}x + b^{[1]} \tag{3.3}$$
$$a^{[1]} = \sigma(z^{[1]}) \tag{3.4}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{3.5}$$
$$a^{[2]} = \sigma(z^{[2]}) \tag{3.6}$$

W^i and b^i are defined, respectively, as:

$$W^{[i]} = \begin{bmatrix} w_1^{T[i]} & w_2^{T[i]} & \dots & w_n^{T[i]} \end{bmatrix} \tag{3.7}$$
$$b^{[i]} = \begin{bmatrix} b_1^{[i]} & b_2^{[i]} & \dots & b_n^{[i]} \end{bmatrix} \tag{3.8}$$

In general, we can represent a layer i recursively as:

$$z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]} \tag{3.9}$$
$$a^{[i]} = \sigma(z^{[i]}) \tag{3.10}$$

The first layer of a neural network is called **input layer** and represents its inputs. The last layer is the **output later** and represents its outputs. All layers between those are **hidden layers**. The greater is the number of layers, the deeper the network is.

3.1.3 Vectorization

Vectorization is a technique where a network calculates the output of several examples in a batch at the same time, instead of calculate them sequentially. Basically, we consider as input:

$$X = \begin{bmatrix} x_1^T & x_2^T & \dots & x_n^T \end{bmatrix} \tag{3.11}$$

Therefore, the network layers' equations change to:

$$Z^{[i]} = W^{[i]}A^{[i-1]} + b^{[i]} \tag{3.12}$$

$$A^{[i]} = \sigma(Z^{[i]}) \tag{3.13}$$

Using vectorized implementations the network can explore better Single Instruction Multiple Data (SIMD) architectures and then speedup learning.

3.2 Activation Functions

As described in subsection 3.1.1, an activation function uses the linear combination as input to give a non-linear representation and the improve network learning capacity. Without this, it only is able to represent linear functions.

There are several activation functions. We will describe the most common used in deep learning research.

3.2.1 Logistic Sigmoid

Equation 3.14 shows the logistic sigmoid function.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.14}$$

This function is commonly used to produce the parameter of a Bernoulli distribution because its range is (0,1). Furthermore, it saturates when its argument is very positive or very negative, meaning the function becomes very flat and insensitive to small changes in its input (GOODFELLOW *et al.*, 2016). Figure 3.5 illustrates the logistic sigmoid function.

3.2.2 Hyperbolic Tangent

Equation 3.15 shows hyperbolic tangent ($tanh$).

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{3.15}$$

FIGURE 3.5 – Illustration of Sigmoid function.

It is close related to logistic function, because:

$$tanh(z) = 2\sigma(2z) - 1 \tag{3.16}$$

Hyperbolic tangent is considered a sigmoidal activation function as well, but typically performs better than sigmoid. It resembles the identity function more closely due to its symmetry in x axis. Train a neural network with this activation function resembles a linear model when the input can be kept small, which makes training easier.

Figure 3.6 illustrates hyperbolic tangent.

FIGURE 3.6 – Illustration of *tanh* function.

3.2.3 Rectified Linear Unit - ReLU

Rectified Linear Unit is a activation function very similar with a linear unit. It is described by equation 3.17:

$$g(z) = max(0, z) \tag{3.17}$$

The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active. The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects (GOODFELLOW *et al.*, 2016). Figure 3.7 illustrates this activation function.

FIGURE 3.7 – Illustration of ReLU function.

(GLOROT *et al.*, 2011) motivate rectified linear units from biological considerations. The half-rectifying nonlinearity was intended to capture these properties of biological neurons:

- For some inputs, biological neurons are completely inactive.
- For some inputs, a biological neuron's output is proportional to its input.
- Most of the time, biological neurons operate in the regime where they are inactive (i.e., they should have sparse activations).

3.2.4 Leaky ReLU

One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. Leaky ReLU (XU *et al.*, 2015) is a generalization of rectified linear units guarantee that they receive gradient everywhere.

Equation 3.18 represents the Leaky ReLU activation.

$$g(z) = max(\alpha z, z), \text{ where } \alpha \in (0, 1) \tag{3.18}$$

Rectified linear units and all generalizations of them are based on the principle that models are easier to optimize if their behavior is closer to linear. Figure 3.8 illustrates this activation function.

FIGURE 3.8 – Illustration of leaky ReLU function.

3.3 Cost Function

In the context of neural networks, gradient-based algorithms are broadly used, especially those based on the backpropagation idea (RUMELHART *et al.*, 1988). The purpose of these algorithms are to propagate the gradient of a cost function through the whole network, in order to minimize the cost function. Most modern neural networks perform this optimization strategy, by using maximum likelihood, i.e. the cross-entropy between the training data and the model distribution:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x}) \tag{3.19}$$

In this work, we used the mean squared error loss function, in order to fit data distri-

butions. Indeed, we may show that both cost functions are closely related. Let us consider normally distributed errors:

$$p_{model}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \sigma^2\mathbf{I}) \tag{3.20}$$

where $f(\mathbf{x}; \boldsymbol{\theta})$ and $\sigma^2\mathbf{I}$ are the mean and covariance of this distribution. By substituting Eq. (3.20) in Eq. (3.19):

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}}\|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + const \tag{3.21}$$

The constant term does not depend on $\boldsymbol{\theta}$ and may be dropped. By explicitly evaluating the expectation in Eq. (3.21), we arrive at the mean squared error cost function:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i}^{m}\|y_i - f(\mathbf{x}; \boldsymbol{\theta})\|^2 \tag{3.22}$$

It can be shown that maximum likelihood estimator is the best estimator asymptotically, when the number of samples m tends to infinity, in terms of its rates of convergence as m increases. Furthermore, this estimator has the property of consistency, under certain conditions that must be satisfied by the data distribution, and efficiency, in terms of achieve lower generalization errors.

3.4 Gradient Descent

The objective of a neural network training is find the parameters W and b that approximates the function represented by itself and the data generator distribution, by minimizing the cost function established. In math terms, it means:

$$(W^*, b^*) = \arg\min_{W,b} J(W, b, x) \tag{3.23}$$

The Gradient Descent technique is a iterative method that calculates the derivative of the loss function and applies them as learning updates according to a learning rate factor, α, previously set. Figure 3.9 illustrates the application of this technique in the context of two variables.

As an example, given the following cost function (where m is the number of samples,

FIGURE 3.9 – Illustration of gradient descent in a two variables optimization (MAGALHAES, 2017).

$\hat{y}$ is the prediction of the network and y is the ground-truth):

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{n} \mathcal{L}(\hat{y}, y) \tag{3.24}$$

The gradient with respect of the parameters W and b can be given by:

$$\frac{\partial J(W, b)}{\partial W} = \frac{1}{m} \sum_{i=1}^{n} \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial W} \tag{3.25}$$

$$\frac{\partial J(W, b)}{\partial b} = \frac{1}{m} \sum_{i=1}^{n} \frac{\partial \mathcal{L}(\hat{y}, y)}{\partial b} \tag{3.26}$$

After these calculations, we are able to apply these gradients using the learning update rule:

$$W = W - \alpha \frac{\partial J(W, b)}{\partial W} \tag{3.27}$$

$$b = b - \alpha \frac{\partial J(W, b)}{\partial b} \tag{3.28}$$

These steps are applied iteratively until the function converges to a optimal value. It's important to mention that there are no guarantees the optimization will converge to a global optima, i.e, it is possible to get stuck in local optima. However, in practical

applications, this is not a huge problem.

3.5 Backpropagation

In the case described in section 3.4 we computed the gradient directly with respect to parameters W and b. However, in neural networks, we have to compute the gradients that correspond to all and layers and not just to the output layer. For this, we need to use the chain rule of calculus.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{3.29}$$

We can generalizes to vectorial case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^m$ to $\mathbb{R}^n$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $\boldsymbol{z} = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{3.30}$$

It can be generalized more generic in a way that can be applied to tensors:

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j)\frac{\partial z}{\partial Y_j} \tag{3.31}$$

The backpropagation takes equation 3.31 recursively, until the gradient is propagated to all layers of the neural network. Algorithm 1 (GOODFELLOW *et al.*, 2016) shows in a simple way how to apply backpropagation to a deep neural network.

3.6 Optimization Algorithms

In Deep Learning community, gradient-based optimization is well established. However, in state-of-art research the backpropagation idea had several improvements to make learning better. In this section, we will explore some of these improvements and present the optimizer used in experimentation.

Algorithm 1 Backward computation for a deep neural network

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} \mathcal{L}(\hat{\boldsymbol{y}}, y)$
for $k = l, l-1, \ldots, 1$ **do**
Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
Compute gradients on weights and biases:
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T}$
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T}\boldsymbol{g}$
end for

3.6.1 Batch, Mini-batch and Stochastic Gradient Descent

There are ways to use the data in order to apply learning updates. Accordingly to section 3.1.3, using vectorization makes possible to calculate several losses at the same time. However, sometimes the dataset is very large in a way it is costly to apply a single batch several times. On the other side, apply updates using one sample at a time can result in a biased gradient and also can be costly due to the fact we need to apply an update for each sample. Therefore, the batch size can change a lot the learning process.

The Batch Gradient Descent calculates the cost function for each sample in the dataset and only updates the model after all samples have been evaluated. In this way, there are fewer updates and the gradient is more precise, due to the fact we consider several samples to regularize it. So, the training and convergence are more stable. However, this variation of gradient descent can result in local optima convergence and turns to be very slow with large datasets.

The Stochastic Gradient Descent calculates the cost function and updates the model for each samples in dataset. Using this variation, there are more frequent updates which can give feedback about the model fast and result in faster learning to some problems. Furthermore, the noisy update allows exploration in function optimization, avoiding local optima (Figure 3.10). However, there is a high computational cost of applying updates so frequently and the same noise can make hard for the last optimization steps. Equation 3.32 shows the learning update rule for Stochastic Gradient Descent:

$$W = W - \alpha \frac{\partial J(W, b, x^{(i)})}{\partial W} \tag{3.32}$$

Figure 3.11 shows how cost function commonly behaves in each variation of Gradient Descent described.

FIGURE 3.10 – Illustration of Stochastic Gradient Descent optimization.

FIGURE 3.11 – Illustration of how each variation of Gradient Descent commonly behaves. (DABBURA, 2017).

There is a third variant. The Mini-batch Gradient Descent is something in between batch and stochastic variants, where the algorithm splits dataset into small batches and each of them will apply learning updates. This version seeks to find a balance between the robustness of SGD and the efficiency of BGD. It is the most common implementation used in Deep Learning (BROWNLEE, 2017). However, the downside of this variation is the fact we need to set a new hyperparameter and tune it manually.

3.6.2 Momentum

Momentum is a technique for accelerating gradient descent that accumulates a velocity vector in directions of persistent reduction in the objective across iterations (SUTSKEVER *et al.*, 2013). It uses the idea of exponentially weighted moving averages, accumulating gradients for each mini-batch accordingly to a hyperparameter β. This term will manage how much the last gradient will affect in the final velocity, following Equations 3.33 and 3.34. Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient (GOODFELLOW *et al.*, 2016).

$$v_{dW} = \beta v_{dW} + (1 - \beta)dW \tag{3.33}$$

$$v_{db} = \beta v_{db} + (1 - \beta)db \tag{3.34}$$

Where dW and db are, respectively:

$$dW = \frac{\partial J(W, b)}{\partial W} \tag{3.35}$$

$$db = \frac{\partial J(W, b)}{\partial b} \tag{3.36}$$

Therefore, using Momentum, the learning update is:

$$W = W - \alpha v_{dW} \tag{3.37}$$

$$b = b - \alpha v_{db} \tag{3.38}$$

The intuition behind Momentum is to filter the gradient, "reinforcing" it in the effective directions and removing the noisy ones. Then, it makes harder to apply learning updates in the wrong direction caused by noise samples. Figure 3.12 shows how Momentum changes Gradient Descent updates.

FIGURE 3.12 – Illustration of how Gradient Descent with Momentum (blue arrows) and without it (black arrows) (MAGALHAES, 2017).

3.6.3 RMSProp

RMSProp (HINTON; TIELEMAN, 2017) uses the idea of exponentially weighted moving average for gradient accumulation in a non-convex optimization scenario. Instead of using the gradient to update velocity as Momentum does, RMSProp uses the square of this gradient, as shown in Equations 3.39 and 3.40.

$$v_{dW} = \beta v_{dW} + (1-\beta)dW^2 \tag{3.39}$$

$$v_{db} = \beta v_{db} + (1-\beta)db^2 \tag{3.40}$$

And the learning update changes to normalize the gradient:

$$W = W - \alpha \frac{dW}{\sqrt{v_{dW}}} \tag{3.41}$$

$$b = b - \alpha \frac{db}{\sqrt{v_{db}}} \tag{3.42}$$

$$\tag{3.43}$$

This normalization improves the learning stability – all gradient terms will affect with the same scale. Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. Algorithm 2 shows RMSProp idea (GOODFELLOW *et al.*, 2016).

Algorithm 2 RMSProp Algorithm

Require: Learning rate α, decay rate β
Require: Parameters W, b
Require: ϵ correction for numeric stability
  Initialize $v_{dW} = v_{db} = 0$
  while stopping criteria not met **do**
    Sample a mini-batch of m examples from training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
    Compute gradient: $\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \frac{1}{m} \nabla_{\hat{\boldsymbol{y}}} \sum_{i=1}^{m} \mathcal{L}(\hat{\boldsymbol{y}}^{(i)}, y^{(i)})$
    Accumulate squared gradient:
      $v_{dW} = \beta v_{dW} + (1-\beta) g \odot g$
      $v_{db} = \beta v_{db} + (1-\beta) g \odot g$
    Apply updates:
      $W = W - \alpha \odot \frac{g}{\sqrt{v_{dW}+\epsilon}}$
      $b = b - \alpha \odot \frac{g}{\sqrt{v_{db}+\epsilon}}$
  end while

3.6.4 Adam

The last optimization algorithm idea explored here - and used in experimentations - is Adam (KINGMA; BA, 2014). It means "Adaptive Moments" and combines the idea from Momentum and RMSProp in the following way: it computes the first order momentum of the gradient and rescale with the second order momentum, as shown in Equations 3.44 and 3.45. The use of momentum in combination with rescaling does not have a clear theoretical motivation.

$$v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) dW \tag{3.44}$$

$$u_{dW} = \beta_2 u_{dW} + (1 - \beta_2) dW^2 \tag{3.45}$$

Adam also incorporates a bias correction for both momentum estimates, resulting in lower bias early in training when compared to RMSProp:

$$v_{dW} = \frac{v_{dW}}{(1 - \beta_1^t)} \tag{3.46}$$

$$u_{dW} = \frac{u_{dW}}{(1 - \beta_2^t)} \tag{3.47}$$

Finally, the learning update changes as well:

$$W = W - \alpha \frac{v_{dW}}{\sqrt{u_{dW} + \epsilon}} \tag{3.48}$$

$$b = b - \alpha \frac{v_{db}}{\sqrt{u_{db} + \epsilon}} \tag{3.49}$$

Therefore, Adam is generally regarded as being fairly robust to the choice of hyperparameters, though sometimes learning rate needs to be changed from the suggested default. Algorithm 3 shows Adam idea (GOODFELLOW *et al.*, 2016).

3.7 Weights Random Initialization

In order to start the learning, it is needed a way to initialize weight parameters. The way these are initialized can affect the convergence and even make it inviable. Suppose weights are all initialized by zero. No matter what was the input, if all weights are the same, all units in the same hidden layer will receive the same signal, either from the input

Algorithm 3 Adam Algorithm

Require: Learning rate α (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, β_1 and β_2
Require: Parameters W, b
Require: ϵ correction for numeric stabilization
Initialize $v_{dW} = v_{db} = u_{dW} = u_{db}$
Initialize time step $t = 0$
while stopping criteria not met **do**
Sample a mini-batch of m examples from training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
Compute gradient: $\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \frac{1}{m} \nabla_{\hat{\boldsymbol{y}}} \sum_{i=1}^{m} \mathcal{L}(\hat{\boldsymbol{y}}^{(\boldsymbol{i})}, y^{(i)})$
$t = t + 1$
Update biased first momentum estimate:
$v_{dW} = \beta_1 v_{dW} + (1 - \beta_1) g$
$v_{db} = \beta_1 v_{db} + (1 - \beta_1) g$
Update biased second momentum estimate:
$v_{dW} = \beta_2 v_{dW} + (1 - \beta_2) g \odot g$
$v_{db} = \beta_2 v_{db} + (1 - \beta_2) g \odot g$
Correct bias in first moment:
$v_{dW} = \frac{v_{dW}}{1-\beta_1^t}$
$v_{db} = \frac{v_{db}}{1-\beta_1^t}$
Correct bias in second moment:
$u_{dW} = \frac{u_{dW}}{1-\beta_2^t}$
$u_{db} = \frac{u_{db}}{1-\beta_2^t}$
Apply updates:
$W = W - \alpha \odot \frac{v_{dW}}{\sqrt{u_{dW}+\epsilon}}$
$b = b - \alpha \odot \frac{v_{db}}{\sqrt{u_{db}+\epsilon}}$
end while

in forward propagation or from the gradient in backpropagation.

Therefore, it needs a way for breaking the symmetry. For that, the rule is initialize all weights randomly. It will also break symmetry and increase the entropy in system in way it can gather more information to find local or global minimums.

There are several ways to initialize weights randomly. In this work, we will explore and use Xavier Initialization.

3.7.1 Xavier Initialization

Xavier Initialization (GLOROT; BENGIO, 2010) is a method for initialize weights accordingly to the network architecture.

There are two main problems that arises from bad weight initialization. First, if weights are too small, the signal shrinks as it passes through layers. Second, if they are too large, the signal grows as it passes through layers. Specifically during learning, it will result in vanishing and exploding gradients, respectively.

Xavier Initialization makes sure weights are started in a reasonable range by using a Gaussian distribution with variance given by Equation 3.50.

$$\sigma^2(W) = \frac{2}{n_{in} + n_{out}} \tag{3.50}$$

Where n_{in} and n_{out} are the number of neurons feeding into it and the result is fed to, respectively. (GLOROT; BENGIO, 2010) shows the mathematical development for this equation.

3.8 Gradient Descent convergence and learning rate decay

It is possible to prove the convergence of gradient descent for a specific set of functions.

Theorem 1 *Suppose the function $f : \mathbb{R}^n \to \mathbb{R}$ is convex and differentiable, and that its gradient is Lipschitz continuous with constant $L > 0$, i.e, we have that $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2$, for any x, y. Then if we run gradient descent for k iterations with a fixed step $t \leq \frac{1}{L}$, it will yield a solution $f^{(k)}$ which satisfies*

$$f(x^{(k)}) - f(x^{(*)}) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2tk} \tag{3.51}$$

where $f(x^{(*)})$ is the optimal value.

The proof of this theorem can be found in (TIBSHIRANI, 2013). Therefore, using a fixed value for the learning rate, it converges with the rate $O(1/k)$. However, if a large learning rate is chosen, there are no guarantees of convergence. Figure 3.13 shows a case of divergence in a one variable optimization.

FIGURE 3.13 – Illustration Gradient Descent divergence for a one variable optimization. (MAGALHAES, 2017).

In practice, learning works well even when some of these requirements are not met. In the context of Deep Learning, it is difficult to analyze if a function is convex because it depends on thousands or millions of parameters. Furthermore, there are activation functions themselves that are not differentiable in the whole interval it acts. Finally, the learning rate that the theorem guarantee convergence is too small. Therefore, it is common to search for a learning rate that converges faster.

Adaptive learning rate is another technique used for make convergence faster.

Theorem 2 *Suppose the function $f : \mathbb{R}^n \to \mathbb{R}$ is convex and differentiable, and that its gradient is Lipschitz continuous with constant $L > 0$, i.e, we have that $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2$, for any x, y. Then if we run gradient descent for k iterations with a adaptive step t_i, chosen using backtracking line search on each iteration i, it will yield a solution $f^{(k)}$ which satisfies*

$$f(x^{(k)}) - f(x^{(*)}) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2t_{min}k} \tag{3.52}$$

where $t_{min} = \min(1, \beta/L)$.

Intuitively, using an adaptive learning rate, we can use larger initial values and then speedup learning. The most common method is learning rate decay, which could be linear, exponential.

In this work, we will use linear learning rate decay.

4 Reinforcement Learning Background

4.1 Concepts of a Reinforcement Learning System

Reinforcement Learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards (SUTTON; BARTO, 1998).

This learning process happens by interacting with the environment where the agent lives and therefore choose its actions. It is basically a "trial and error" approach as humans or other animals do, but in this case using a computational approach and mathematical modeling.

The origin of modern Reinforcement Learning has two main threads developed outside Artificial Intelligence research. One concerns about trial and error search inside psychology of animal learning, based on Behaviorism theory (SKINNER, 1953) and "The Law of Effect" (THORNDIKE, 1933). The other thread concerns the problem of optimal control and its solution using value functions and dynamic programming. Both of them contributed for the modern ideas in reinforcement learning, either with the abstractions of learning or mathematical modeling for this problem.

Reinforcement Learning is different from other kinds of Machine Learning, for several reasons:

- **There is no supervisor, only a reward signal**: In Supervised Learning, there is a set of pairs (x, y) and we optimize a cost function in a way to minimize the error from prediction and the ground truth. Therefore, there is a knowledgable external supervisor and a clear specification of mapping between situation and action to take - which doesn't exist in the context of reinforcement.

- **Feeback is delayed, not instantaneous**: In Reinforcement Learning, the reward is a consequence of a sequence of actions, rather than a direct mapping from actions to reward. For example, during the learning of a kick motion, the final reward of ball movement only occurs after all the agent motion control.

- **Data is not independent and identically distributed**: In this kind of learning, there is a intrinsic sequentiality, i.e, the data we collect at some point depends on the state we are.

- **Agent's actions affect the subsequent data it receives**: More than being i.i.d data, the learning process itself change the data distribution, differently from supervised learning where the dataset doesn't change during the course of training.

Therefore, the optimization problem in Reinforcement Learning has new challenges and needs a different analysis when compared with other kinds of Machine Learning. In this chapter, we will explore these ideas and state-of-art techniques to sustain our experimentations.

4.2 Reinforcement Learning System

Figure 4.1 shows how a Reinforcement Learning system works and some of the core concepts involved. As described in section 4.1, there are two main entities in Reinforcement Learning: an *agent*, which takes actions with the object of accumulate reward, and a *environment*, the place where happens this interaction and whose dynamics affects the decision about these actions.

During learning process, the agents gets a *observation* from the environment and maps to a *state*. Accordingly to that state, the agent then executes an *action* in the environment, based on the current *policy* of actions, which has a consequence in the environment and causes a transition in that state. Furthermore, the agent receives a *reward* signal about how that action affected the environment in terms of accomplish the task goals. This cycle will happen iteratively and the policy will be adjusted until in a way that maximize the cumulative reward.

In the next subsections, we will describe in deep the main concepts of a RL system.

4.2.1 Reward

A reward signal is a scalar number received by the agent and drives the goal in a RL problem. It indicates how well the action taken is in that state. In the learning process,

FIGURE 4.1 – Reinforcement Learning System.

the reward signal guides the policy modification: when an actions has high reward, it is a case of positive reinforcement and the policy will be adjusted to take this action more likely in that state; on the other side, if the reward is low, it is a case of negative reinforcement and policy will try to act in this way less times.

Finally, we can formalize this idea stating that RL is based on the **Reward Hypothesis**:

Definition 4 *All goals can be described by the maximization of expected cumulative reward.*

4.2.2 State

Basically, a state is a function of the sequence observations, actions and rewards. There are two different concepts of State:

- **Environment State**: Environment's internal representation - the way it understands its feature at that point and uses to pick next observation or reward.
- **Agent State**: The information that agent perceives and understand from the observation and uses to pick a new action.

In RL problems, we commonly use the **Agent State** concept – the information provided by the environment and interpreted by the agent in order to decide which action to take next. More specifically, we will use the concept of Markov state, which will be discussed later.

4.2.3 Policy

A policy defines the agent's behavior. It maps state to actions. It corresponds to what in psychology would be called a set of stimulus–response rules or associations (SUTTON; BARTO, 1998).

The policy is the core element of a RL system. It is sufficient to define the behavior of an agent and therefore is the piece we need after the learning process. There are two kind of policies:

- **Deterministic Policy**, where there is a direct map from state to action:

$$a = \pi(s) \tag{4.1}$$

- **Stochastic Policy**, where the map happens from the static to a probability distribution between actions:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{4.2}$$

4.2.4 Value Function

Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long run. The value function is a estimate of reward the agent can expect to accumulate over the future, until a terminal state.

The value function can be formalized in Equation 4.3:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \tag{4.3}$$

In RL, we focus on choosing actions that bring about states of highest value, not just highest reward. Therefore, it is common to sacrifice immediate rewards to maximize future rewards.

Furthermore, it is much harder to determine value than rewards, because while this is given directly from the environment, that one must be estimated accordingly to data the agent receives. We will describe state-of-art algorithms to predict value function later.

4.2.5 Model

A model is a component that predicts environment's behavior. Basically it will estimate the next state given the agent's action, as formalized in Equation 4.4:

$$\mathcal{P}^{a}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \tag{4.4}$$

Where $\mathcal{P}$ is a transition matrix which calculates the probability of transit between state s and s' given the action a.

Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced (SUTTON; BARTO, 1998).

There are two methods based on model classification:

- **Model-based RL**, methods that uses model and planning to solve the learning challenge;
- **Model-free RL**, which are methods that are direct trial and error learners.

Sometimes there is no model of the environment or it is very hard to create it – in that cases, we commonly use model-free methods. In this work, we don't know about the dynamics of simulator, thus we adopt model-free RL.

4.3 Markov Decision Process

Markov Decision Processes (MDPs) are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards (SUTTON; BARTO, 1998). Thus, MDPs formally describe an environment for Reinforcement Learning. Almost all RL problems can be formalized as MDPs (SILVER, 2015a).

In next subsections, we will define mathematically what is a MDP, its major components and properties.

4.3.1 Markov State

A state S_t is considered *Markov State* if and only if:

$$\mathbb{P}[S_{t+1} | S_t] = \mathbb{P}[S_{t+1} | S_1, \ldots, S_t] \tag{4.5}$$

Intuitively, Equation 4.5 tells that in a Markov State, the future is independent of the past given the present. It means that the state has all the relevant information from the history and therefore it is sufficient statistic for the future.

As example, consider a chess game: given a specific state – a set of pieces in the board – all the information a player needs to play is in the board state, regardless of how the game came in that circumstance. The history may be thrown away.

4.3.2 State Transition Matrix

Other major component of a MDP is the State Transition Matrix. Given a set of Markov States, this component defines the probabilities of transition between states. More formally:

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \tag{4.6}$$

Where $\mathcal{P}_{ss'}$ is the transition probability:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s] \tag{4.7}$$

4.3.3 Markov Decision Process

Using the concepts from subsecions 4.3.1 and 4.3.2, we can define a *Markov Process* (MP):

Definition 5 *A Markov Process, or Markov Chain, is a tuple* $(\mathcal{S}, \mathcal{P})$*, where:*

- $\mathcal{S}$ *is a set of states; and*
- $\mathcal{P}$ *is the state transition probability matrix.*

Considering the definition of Markov Process and the concept of Reward described in section 4.2.1, we can define a *Markov Reward Process* (MRP):

Definition 6 *A Markov Reward Process, is a tuple* $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$*, where:*

- $\mathcal{S}$ *is a set of states;*
- $\mathcal{P}$ *is the state transition probability matrix;*

- $\mathcal{R}$ *is a reward function, i.e,* $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$*; and*
- γ *is a discount factor, where* $\gamma \in [0, 1]$

A Markov Reward Process (MRP) describes mathematically an environment in a RL problem. Furthermore, we can differentiate an MRP from a MP by the concept of *Return* that is directly related to the reward over time:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.8}$$

The idea around cumulative reward was described earlier in section 4.2. However, we need to analyze the discount factor γ – it will weight the trade-off between immediate and delayed rewards.

Basically, if γ is close to zero, the MRP will consider more an immediate reward; on the other side, if γ is close to one, the MRP will consider more a delayed reward. Furthermore, the idea of discount is mathematically convenient and avoids infinite returns in cyclic Markov processes (SILVER, 2015a).

Finally, we can extend the idea of MRP by adding an decision making procedure execute by the agent – which results in the concept of *Markov Decision Process*:

Definition 7 *A Markov Decision Process, is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$*, where:*

- $\mathcal{S}$ *is a set of states;*
- $\mathcal{A}$ *is a set of actions;*
- $\mathcal{P}$ *is the state transition probability matrix;*
- $\mathcal{R}$ *is a reward function, i.e,* $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$*; and*
- γ *is a discount factor, where* $\gamma \in [0, 1]$

Thus, MDP is a MRP with decisions. It worth to mention that the state transition probability $\mathcal{P}$ in MDP also takes in consideration the action executed by the agent:

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] \tag{4.9}$$

Now, we have a formal definition of a RL problem by using MDP concept. Figure 4.2 shows the agent-environment interaction in a MDP. In the next section, we will formalize other RL components in a MDP context.

FIGURE 4.2 – The agent–environment interaction in a Markov decision process (SUTTON; BARTO, 1998).

4.3.4 State-Value and Action-Value Function

In a MDP context, we can explore deeper the idea of Value Function, and thus define *State-Value* and *Action-Value functions*.

Definition 8 *The state-value function $v_\pi(s)$ of a MDP is the expected return starting from s and following a policy π:*

$$v_\pi(s) = \mathbb{E}_\pi[G_t \big| S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right] \tag{4.10}$$

The only difference between this definition and that explained in section 4.2.4 is the fact we consider state-value function as a component of a MDP and in terms of return.

However, a new concept arises in MDPs: Action-Value function:

Definition 9 *The action-value function $q_\pi(s, a)$ of a MDP is the expected return starting from state s, taking action a, and then following policy π:*

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \tag{4.11}$$

From Definitions 8 and 9, we can see that the action-value function is a granulated version of state-value, considering the new dimension of action space. It worth to mention that, by definition, the action-value evaluates a specific action in a state, and then follows the policy π, which means that action a is not chosen by that π.

4.4 Bellman Equation

4.4.1 Bellman Expectation Equation

Given Definition 8, it is possible to develop state-value function equation and come to a fundamental property of value functions: they satisfy a recursive relationship called *Bellman Equation*:

$$
\begin{aligned}
v_\pi &= \mathbb{E}_\pi[G_t \mid S_t = s] && (4.12)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s] && (4.13)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] && (4.14)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (4.15)\\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \middle| S_{t+1} = s' \right] && (4.16)
\end{aligned}
$$

Equation 4.16 can be simplified in terms of reward and transition matrix:

$$
v_\pi(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v_\pi(s') \tag{4.17}
$$

The Bellman equation expresses relationship between a state and successor states. Figure 4.3 illustrates a agent-environment interaction. This equation average over all the possibilities, weighting by the probability of occurring.

FIGURE 4.3 – Backup diagram for v_π. The white nodes are states, while the black ones are possible actions. Bellman equation average over the actions in a recursive way. (SUTTON; BARTO, 1998).

The Bellman equation can also be derived for action-value function. Given:

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \tag{4.18}$$

It's possible to develop Equation 4.18, resulting in a version analogous to Equation 4.17:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a' \mid s') q_\pi(s', a') \tag{4.19}$$

The Bellman equation forms the basis of algorithms to learn v_π. The version described here is called *Bellman Expectation Equation* and, as a linear equation, has exact solution. However, there are variations whose purpose is to find optimality in policies and value functions – and the solution for this methods is not straightforward.

4.4.2 Bellman Optimality Equation

Solve a RL problem means find a policy that its actions results in the maximum cumulative reward. It means find the best policy during learning. We can define optimality for policies based on optimal value functions.

Definition 10 *The optimal state-value function $v_*(s)$ is the maximum state-value function over all policies:*

$$v_*(s) = \max_\pi v_\pi(s) \tag{4.20}$$

Similarly, the optimal action-value function $q_*(s,a)$ is the maximum action-value function over all policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{4.21}$$

The optimal value function specifies the best possible performance in the MDP. Therefore, a MDP is considered solved when we know the optimal value function (SILVER, 2015a).

Based on the idea of value function optimality, it is possible to define policy ordering:

Definition 11 *Given policies π, π', we can define a partial ordering in the way that*

$$\forall s \in \mathcal{S}, v_\pi(s) \geq v_{\pi'}(s) \Rightarrow \pi \geq \pi'$$

Based on this definition, it results in the *Policy Optimality Theorem*:

Theorem 1 *For any MDP:*

- *There exists an optimal policy* π_* *such that* $\pi_* \geq \pi, \forall \pi$
- *All optimal policies achieve the optimal state-value and action-value functions:* $v_{\pi_*}(s) = v_*(s)$ *and* $q_{\pi_*}(s,a) = q_*(s,a)$.

Also from optimal value function definition, it is possible to define *Bellman Optimality Equation*:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) && (4.22)\\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] && (4.23)\\
&= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')] && (4.24)\\
&= \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') && (4.25)
\end{aligned}
$$

Similarly, for action-value function:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] && (4.26)\\
&= \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] && (4.27)\\
&= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a) && (4.28)
\end{aligned}
$$

Figure 4.4 illustrates the Bellman Optimality equation for both state-value and action-value functions, similarly to Figure 4.3. Instead of averaging over actions, the optimality equation chooses the path of greedy action.

FIGURE 4.4 – Backup diagram in Bellman Optimality Equation for v_π and q_π. (SUTTON; BARTO, 1998).

Solve the Bellman Optimality Equation gives the optimal policy for a specific MDP. Therefore, we can conclude that the solution of a RL problem is fundamentally solve this equation. However, as it is non-linear, there is no closed form solution, so we need to explore ways of achieve it.

4.5 Exact Solution Methods

The first algorithms created to solve the Bellman Optimality Equation in RL problems are based in Dynamic Programming (DP) methods. It will consider a perfect MDP and are, in general, computationally expensive. However, it is important to understand and know its limitations before dive into modern approaches.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. Thus, the exact solution methods are obtained turning Bellman equations into update rules for improving approximations of the desired value functions (SUTTON; BARTO, 1998).

4.5.1 Policy Iteration

The first method described here is the Policy Iteration. Figure 4.5 illustrates how it works. This technique is composed by two main steps: policy evaluation and policy improvement.

FIGURE 4.5 – Policy Iteration algorithm. (SILVER, 2015b).

In evaluation, the value function v_π from the current policy π will be estimated. As result, we will have a value function that predicts well the real v_π.

Then, it is possible to modify the current policy by taking greedy actions based on the value function, during policy improvement. Basically, we would like to generate π' in a way that $\pi' \geq \pi$.

After generate this new policy, we return to evaluation. This cycle occurs until find the optimal value function and policy.

Therefore, there are two challenges inside policy iteration. We will formulate each of them.

4.5.1.1 Policy Evaluation

During evaluation, it is just needed to apply Bellman updates using the data collected by the agent through the episodes:

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \tag{4.29}$$

Additionally, it can be shown that:

$$\lim_{k\to\infty} v_k = v_\pi \tag{4.30}$$

Algorithm 4 shows the iterative policy evaluation pseudocode.

Algorithm 4 Iterative Policy Evaluation

Require: π, current policy
Require: $V(s)$, an array for storing value function
  Initialize $V(s) = 0$, $\forall s \in \mathcal{S}^+$
  while δ is large enough **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
    $\delta \leftarrow \max(\delta, |v - V(s)|)$
  end while

4.5.1.2 Policy Improvement

During policy improvement, the value function evaluated before will be used to find better policies.

This step is based on *Policy Improvement Theorem*:

Theorem 1 *Let π' and π be any pair of deterministic policies such that, $\forall s \in \mathcal{S}$:*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{4.31}$$

Then the policy π' must be as good as or better than π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_{\pi}(s) \tag{4.32}$$

The proof of this theorem can be found in (SUTTON; BARTO, 1998). Thus, we can improve the policy by acting greedily:

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_{\pi}(s, a) \tag{4.33}$$

Acting this way, it is possible to satisfy the condition to apply the theorem:

$$q_{\pi}(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_{\pi}(s, a) \geq q_{\pi}(s, \pi(s)) = v_{\pi}(s) \tag{4.34}$$

If the improvement stops, it means the equality as achieved. Therefore, the Bellman optimality equation has been satisfied:

$$v_{\pi}(s) = \max_{a \in \mathcal{A}} q_{\pi}(s, a) = v_{*}(s) \tag{4.35}$$

4.5.1.3 Policy Iteration algorithm

The policy iteration, as described earlier, is a sequence of evaluations and improvements, until converges to optimal policy. The Algorithm 5 shows the pseudocode (SUTTON; BARTO, 1998);

Algorithm 5 Policy Iteration

1. Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily , $\forall s \in \mathcal{S}$
2. Run policy evaluation described in Algorithm 4
3. Policy Improvement
   policy_stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old_action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
      if *old_action* $\neq \pi(s)$, then *policy_stable* $\leftarrow$ *false*
   if *policy_stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2.

4.5.2 Value Iteration

In the Value Iteration, instead of applying two cyclic steps of evaluation and improvement, the algorithm directly applies Bellman optimality backups. Therefore, this

method effectively combines the steps from Policy Iteration, which often results in faster convergence.

Algorithm 6 presents the pseudocode of this technique.

Algorithm 6 Value Iteration

Initialize $V(s)$ arbitrarily, $\forall s \in \mathcal{S}^+$
while δ is large enough **do**
  $v \leftarrow V(s)$
  $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
  $\delta \leftarrow \max(\delta, |v - V(s)|)$
end while
Output a deterministic policy $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$

4.5.3 Limitations of Exact Solution Methods

During this section, we presented the core ideas from Exact Solution Methods. They have convergence proofs and reasonably simple algorithms. However, there are limitation in their uses, which turns inviable to the context of this work.

The first limitation is because these algorithms requires knowledge about the dynamic models. In other words, we need to know about the transition matrix $\mathcal{P}^a_{ss'}$, it turns the usage of these methods very limited. For example, in the case of controlling the kick motion, we would need the dynamics from simulation server and there is not this model.

To solve this problem, it has been developed many **sampling-based methods**. These are model-free RL algorithms and are applied replacing the expectation from Bellman equations by samples of agent executing actions inside environment. From this idea, some methods have arisen, such as **Monte-Carlo** and **Temporal-Difference methods**, which resulted in classic algorithms like SARSA (RUMMERY; NIRANJAN, 1994) and Q-Learning (WATKINS, 1989). We will not describe them because it is beyond the scope. However, the idea of sampling based approximations will be used during experimentation.

The second limitation is about storing and iterating over all states and actions for large MDPs. As described during the section, using Dynamic Programming means save values for each state or action-state. It does not scale well, especially for high dimensional continuous domains like the environment used in this work. For example, in the game of Go, there are 10^{170} possible states. If we represent each state with a single bit, we would use approximately 10^{160} gigabytes – there is no capacity in the world for it. Furthermore, there will be a large number of iterations to learn all values individually.

To overcome this problem, the solution is to approximate the value function or even

the policy itself using a learning representation:

$$\hat{v}(s, \boldsymbol{\theta}) \approx v_\pi(s) \tag{4.36}$$

$$\hat{q}(s, a, \boldsymbol{\theta}) \approx q_\pi(q, a) \tag{4.37}$$

$$\pi_{\boldsymbol{\theta}}(s, a) \approx \pi(s, a) \tag{4.38}$$

The idea behind this is we can represent all states in a compact way and, during training, generalize from seen to unseen states (SILVER, 2015d).

We will use neural networks, for some reasons. First, it is differentiable, thus it is possible to use gradient methods for learning. Second, it allows to use data which is not stationary, independent or identically distributed. Lastly, we can use Deep Learning – saw in chapter techniques 3 – to scale up learning, both in capacity and convergence velocity.

In next sections, we will define policy gradient methods and describe the major algorithms used in this work.

4.6 Policy Gradient Methods

In Policy Gradient Methods, the objective is to learn a parameterized policy that can select actions without consulting a value function, which can be used just for training purposes. These methods compute the gradient for some cost function $J(\theta)$ with respect to policy parameters, in a way to maximize the performance.

The usage of policy-based algorithms has some advantages. Firstly, parameterize a policy π is often simpler than value functions. Furthermore, $v_\pi(s)$ doesn't prescribe actions (thus would need dynamics modeling and bellman updates) and $q_\pi(s, a)$ need to efficiently solve $\arg\max_a q_\pi(s, a)$, which can be challenging for continuous problems (ABBEEL *et al.*, 2017).

Computing gradients ensures better convergence properties and it is effective in continuous action spaces. Additionally, can learning stochastic policies. The disadvantages are sub-optimality convergence and policy evaluation becomes challenging because is typically inefficient and have high variance (SILVER, 2015c).

Finally, policy direct optimization is useful because it is possible to optimize what we effectively care about and in a compatible way with richer architectures (such as deep neural networks).

4.6.1 Policy Gradient Theorem

There is a excellent theoretical answer which ensures that in parameterized policies, the gradient does not depends on the dynamics model, i.e, it does not involve the derivative of state distribution.

Theorem 1 *Suppose a differentiable parameterized policy that will be optimized using gradient by some performance measure $J(\theta)$. Then:*

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s} \mu(s) \sum_{a} q_{\pi}(s,a) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}) \tag{4.39}$$

Where μ means the on-policy distribution under π. The proof for episodic case can be found in (SUTTON; BARTO, 1998). This result is very useful for model-free RL methods.

To apply Policy Gradient Theorem, we need a performance measure $J(\theta)$. In a first moment, lets consider the value function over a episode under policy π. That is: let τ denote a sequence of states and actions in a episode of length H under policy π. Thus:

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{H} R_t(s,a)\right] = \sum_{\tau} p(\tau, \boldsymbol{\theta}) R(\tau) \tag{4.40}$$

For estimate this value, we can use **importance sampling method** (JIE; ABBEEL, 2010) from statistics as "surrogate loss". This technique works as a variance reduction method where we pick samples from a parameter distribution (denoted here as $\boldsymbol{\theta}_{old}$), but computes the performance measure within respect to a distribution θ applying a "importance correction" to the reward:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \boldsymbol{\theta}_{old}}\left[\frac{p(\tau \mid \boldsymbol{\theta})}{p(\tau \mid \boldsymbol{\theta}_{old})} R(\tau)\right] \tag{4.41}$$

Applying the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \boldsymbol{\theta}_{old}}\left[\frac{\nabla_{\boldsymbol{\theta}} p(\tau \mid \boldsymbol{\theta})}{p(\tau \mid \boldsymbol{\theta}_{old})} R(\tau)\right] \tag{4.42}$$

When $\boldsymbol{\theta} = \boldsymbol{\theta}_{old}$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{old}} = \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\frac{\nabla_{\boldsymbol{\theta}} p(\tau \mid \boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_{old}}}{p(\tau \mid \boldsymbol{\theta}_{old})} R(\tau)\right] \tag{4.43}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}[\nabla_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})|_{\boldsymbol{\theta}_{old}} R(\tau)] \tag{4.44}$$

Decomposing this last result in terms of states and actions:

$$\mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}[\nabla_{\boldsymbol{\theta}} \log p(\tau \mid \boldsymbol{\theta})|_{\boldsymbol{\theta}_{old}} R(\tau)] = \tag{4.45}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\nabla_{\boldsymbol{\theta}} \log \Big[\prod_{t=0}^{H} p(s_{t+1}|s_t, a_t)\pi_{\boldsymbol{\theta}}(a_t|s_t)\Big] R(\tau)\right] \tag{4.46}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\nabla_{\boldsymbol{\theta}} \Big[\sum_{t=0}^{H} \log p(s_{t+1}|s_t, a_t) + \sum_{t=0}^{H} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\Big] R(\tau)\right] \tag{4.47}$$

$$= \mathbb{E}_{\tau\sim\boldsymbol{\theta}_{old}}\left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) R(\tau)\right] \tag{4.48}$$

From Equation 4.48, we finally conclude that computing gradient from the surrogate loss based on importance sampling estimate we have the same result of Policy Gradient Theorem – which means we have a unbiased estimate of gradient that do not require the dynamics model.

However, even with importance sampling, the gradient estimate is very noisy. We will reduce this variance by using three fixes: using a *Critic*, computing advantage function instead of value function and changing the surrogate loss to consider KL-divergence trust region.

4.6.2 Actor-Critic Models

During section 4.5, we focused on value-based methods, i.e, we chose actions accordingly to value function estimations. On the other side, the current section detailed the usage of Policy Gradient Methods, where we parameterize a policy and apply gradient updates directly. The problem is the high variance, as described in last subsection.

It is possible to join both ideas in order to reduce variance for Policy Gradient Methods.

We use a *Critic* to estimate the action-value function:

$$Q(s, a, \boldsymbol{w}) \approx Q_{\pi_{\boldsymbol{\theta}}}(s, a) \tag{4.49}$$

Where $\boldsymbol{w}$ represents the parameters of action-value function. Therefore, there are two entities in a Actor-Critic model:

- **Critic** updates action-value function, with parameters $\boldsymbol{w}$;
- **Actor** updates policy parameters $\boldsymbol{\theta}$, in direction suggested by critic

In this context, Actor-Critic methods apply a generalization of Policy Iteration algorithm, illustrated by Figure 4.5: while the Critic solves policy evaluation, Actor executes policy improvement.

Furthermore, with the usage of Critic estimating action-value function, Equation 4.48 replaces instantaneous for long-term action-value, which also reduces variance:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_t \left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \right] \tag{4.50}$$

Figure 4.6 illustrates how Actor-Critic works.

FIGURE 4.6 – Actor-Critic model. (XU *et al.*, 2007).

4.6.3 Advantage Function and GAE Algorithm

Another idea aiming to reduce variance in Policy Gradient methods is to estimate Advantage Function instead of action-value. By definition:

$$A_{\pi_{\boldsymbol{\theta}}}(s,a) = Q_{\pi_{\boldsymbol{\theta}}}(s,a) - V_{\pi_{\boldsymbol{\theta}}}(s) \tag{4.51}$$

Using Advantage Function, we estimate both action-value and value-function values. Thus, the performance measure given by Equation 4.48 changes again:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_t \left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) A_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \right] \tag{4.52}$$

In practice, in order to estimate Advantage function, it is possible to use Temporal-Difference error:

$$\delta^t_{\pi_{\boldsymbol{\theta}}} = R_{s_t s_{t+1}} + \gamma V_v(s_{t+1}) - V_v(s_t) \tag{4.53}$$

Where $V_{\boldsymbol{v}}$ is a estimation of $V_{\pi_{\boldsymbol{\theta}}}$, parameterized by v. This is possible because Temporal-Difference error estimates Advantage function without bias when $V_v = V_{\pi_{\boldsymbol{\theta}}}(s)$:

$$\begin{aligned}
\mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\delta^t_{\pi_{\boldsymbol{\theta}}} \mid s_t, a] &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[R_{s_t s_{t+1}} + \gamma V_{\pi_{\boldsymbol{\theta}}}(s_{t+1})] - V_{\pi_{\boldsymbol{\theta}}}(s_t) && (4.54)\\
&= Q_{\pi_{\boldsymbol{\theta}}}(s_t, a) - V_{\pi_{\boldsymbol{\theta}}}(s_t) && (4.55)\\
&= A_{\pi_{\boldsymbol{\theta}}}(s_t, a) && (4.56)
\end{aligned}$$

However, we can't guarantee the condition $V_v = V_{\pi_{\boldsymbol{\theta}}}(s)$. Thus, this estimation induce bias. To solve this problem, let's define the Advantage Estimator in terms of discounted Temporal-Difference errors of higher orders, e.g:

$$\begin{aligned}
A^{(1)}_{\pi_{\boldsymbol{\theta}},t}(s_t, a) &= \delta_{\pi_{\boldsymbol{\theta}},t} && (4.57)\\
A^{(2)}_{\pi_{\boldsymbol{\theta}},t}(s_t, a) &= \delta_{\pi_{\boldsymbol{\theta}},t} + \gamma \delta_{\pi_{\boldsymbol{\theta}},t+1} && (4.58)\\
A^{(3)}_{\pi_{\boldsymbol{\theta}},t}(s_t, a) &= \delta_{\pi_{\boldsymbol{\theta}},t} + \gamma \delta_{\pi_{\boldsymbol{\theta}},t+1} + \gamma^2 \delta_{\pi_{\boldsymbol{\theta}},t+2} && (4.59)
\end{aligned}$$

The Generalized Advantage Estimator GAE is defined as the exponentially-weighted average of these estimators (SCHULMAN *et al.*, 2015b). In mathematical terms:

$$A^{GAE(\gamma,\lambda)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) = (1-\lambda)(A^{(1)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) + \lambda A^{(2)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) + \lambda^2 A^{(3)}_{\pi_{\boldsymbol{\theta}},t}(s_t,a) + \dots) \tag{4.60}$$

$$= \sum_{t=0}^{\infty} (\gamma\lambda)^t \delta_{\pi_{\boldsymbol{\theta}},t+1} \tag{4.61}$$

The steps from Equation 4.60 to 4.61 can be found in (SCHULMAN *et al.*, 2015b). GAE with $\lambda = 0$ is the case of Equation 4.53 and, as mentioned earlier, can induce bias. On the other side, GAE with $\lambda = 1$ doesn't induce bias regardless of the accuracy of $V_v(s)$, but it has high variance. Therefore, GAE with $0 \leq \lambda \leq 1$ makes a compromise between bias and variance, controlled by λ (SCHULMAN *et al.*, 2015b).

In this work, we use a Advantage Actor-Critic model, where this Advantage function is estimated using GAE. In next section, we finalize Reinforcement Learning Background by adding the last piece in the optimization algorithm to reduce variance: consider KL-divergence trust region.

4.7 Advanced Policy Gradient Methods

There is a third method to reduce variance in parameterized policies: consider it as a optimization problem.

Much of modern ML reduce learning to numerical optimization problem. For example, in supervised learning, the object is minimize training error (SCHULMAN, 2017).

So, in this section we will present modern policy gradient techniques that represents learning as a optimization problem that allows small updates in the policy using the data sampled.

4.7.1 Optimization Loss

In Equation 4.41, we used importance sampling weighted by model probability distributions and derived the Policy Gradient Theorem result. Then, modified the "target function" from instantaneous reward to advantage estimation.

However, if we consider Equation 4.52 again, it is possible to derive the same gradient from the following loss:

$$\mathcal{L}^{PG}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[\log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) A_{\pi_{\boldsymbol{\theta}}}(s,a)\Big] \tag{4.62}$$

And, equivalently, we can differentiate the same gradient from this following loss, where we apply importance sampling to policy distribution:

$$\mathcal{L}^{PG}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[\frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} A_{\pi_{\boldsymbol{\theta}}}(s, a)\Big] \tag{4.63}$$

(SCHULMAN *et al.*, 2015a) and (KAKADE; LANGFORD, 2002) analyze the usage of Equation 4.63 as surrogate loss and concluded that in practice the results are not much different if compared to Equation 4.62, for small policy changes.

The problem of Equation 4.63 is because it leads to destructively large policy updates. Thus, we need a way to ensure these step sizes are small enough to a bad update not collapse policy.

4.7.2 Trust Region Policy Optimization – TRPO

From Equation 4.63, the first method to avoid large policy updates is define a trust region update and reduce learning to a optimization problem:

$$\begin{gathered}\max_{\boldsymbol{\theta}} \mathbb{E}_t\Big[\frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} A_{\pi_{\boldsymbol{\theta}}}(s, a)\Big] \\ \text{subject to } \mathbb{E}_t[KL[\pi_{\boldsymbol{\theta}_{old}}, \pi_{\boldsymbol{\theta}}]] \leq \delta\end{gathered} \tag{4.64}$$

This is a constrained optimization problem. The constrain here is a **Kullback-Leibler divergence**, which measures how a probability distribution is different from a second. In this case, we ensure the value of this divergence is small. Therefore, the policy change is not big, avoiding too far steps that could collapse learning.

Finally, based on what have been shown in Section 4.6 and this one, we present TRPO algorithm pseudocode.

The technique proposed to efficiently approximately solve this optimization problem can be found in (SCHULMAN *et al.*, 2015a) and is based on conjugate gradient method (HESTENES; STIEFEL, 1952).

TRPO is a scalable method with strong theoretical foundations, being able to learn controllers for robotics locomotion from scratch for continuous tasks, using a generic policy search method and general purpose policy representation due to its results in classical benchmarking environments.

However, as a constrained optimization problem, its difficult to implement a solving

Algorithm 7 TRPO Algorithm

while stopping criteria not met **do**
Run policy π for H timesteps
Estimate Advantage Function at all timesteps
Solve the numerical optimization problem:

$$\max_{\boldsymbol{\theta}} \mathbb{E}_t\Big[\frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} A_{\pi_{\boldsymbol{\theta}}}(s,a)\Big]$$
$$\text{subject to } \mathbb{E}_t[KL[\pi_{\boldsymbol{\theta}_{old}}, \pi_{\boldsymbol{\theta}}]] \leq \delta$$

end while

method based on conjugate gradients. Additionally, is not compatible with architectures that include noise or parameter sharing.

4.7.3 Proximal Policy Optimization – PPO

The Proximal Policy Optimization algorithm have some of the benefits of TRPO, but it is much simpler to implement, more general, and have better sample complexity empirically (SCHULMAN *et al.*, 2017). It states as the state-of-art algorithm for robotic locomotion tasks.

4.7.3.1 Clipped Surrogate Loss

As we saw, the trust region method ensures small policy updates by constraining the problem. However, it makes really hard to solve efficiently.

What PPO does is try to ensure small changes by clipping the surrogate loss, penalizing big steps. Let's define:

$$r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t \mid s_t)}{\pi_{\boldsymbol{\theta}_{old}}(a_t \mid s_t)} \tag{4.65}$$

The clipped surrogate loss is defined as:

$$\mathcal{L}^{CLIP}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[\min(r_t(\boldsymbol{\theta})A_{\pi_{\boldsymbol{\theta}}}, clip(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)A_{\pi_{\boldsymbol{\theta}}})\Big] \tag{4.66}$$

The Equation 4.66 modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving $r_t(\boldsymbol{\theta})$ outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally,

it takes the minimum of clipped and unclipped objective, so the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective. With this scheme, it only ignores the change in probability ratio when it would make the objective improve, and we include it when it makes the objective worse (SCHULMAN *et al.*, 2017).

Therefore, PPO tries the same idea of delimiter a small region for updates but treats the problem as unconstrained optimization – in a way that is possible to apply stochastic gradient descent again.

4.7.3.2 PPO Algorithm

Finally, let's join everything described in this chapter to detail the PPO algorithm used in this work.

First of all, we use a Actor-Critic model where there are two deep neural networks as parameterized policy and value-function.

These neural networks are trained using jointly and using the following loss function:

$$\mathcal{L}^{CLIP+VF+S}(\boldsymbol{\theta}) = \mathbb{E}_t[\mathcal{L}^{CLIP}(\boldsymbol{\theta}) - c_1\mathcal{L}^{VF}(\boldsymbol{\theta}) + c_2 S[\pi_{\boldsymbol{\theta}}](s_t)] \tag{4.67}$$

$\mathcal{L}^{CLIP}(\boldsymbol{\theta})$ denotes the clipped surrogate loss detailed earlier. The second term, $\mathcal{L}^{VF}(\boldsymbol{\theta})$ is the squared loss from prediction of value function and its value computed by the samples collected. Thus, this term aims to train the Critic network. The third term, denotes a entropy term to ensure sufficient exploration (MNIH *et al.*, 2016).

In order to estimate the advantage function of $\mathcal{L}^{CLIP}(\boldsymbol{\theta})$ term, we use GAE algorithm, as explained in section 4.6.3. We use a truncated version that use a length-T trajectory-segment, i.e, we collect T timesteps from episode and apply Equation 4.61 in a finite sum.

Lastly, the algorithm runs as a generalized policy iteration technique: we evaluate the policy by using the data collected and perform improvement applying optimization updates from Equation 4.67 using Adam algorithm.

Algorithm 8 shows the pseudocode.

Algorithm 8 PPO Algorithm

```
while stopping criteria not met do
    foreach agent do
        Run policy π_{θ_old} in environment for T timesteps
        Estimate Advantage Function at all timesteps using truncated GAE algorithm
    end for
    Apply ∇L^{CLIP+VF+S}(θ) to networks using Adam algorithm
    θ_old ← θ
end while
```

5 Methodology

5.1 The Kick Motion Problem

First of all, we need to understand why kick motion is a hard task for learning models, which justifies the usage of scalable techniques such as Deep Reinforcement Learning.

- **Long Time Horizons**: The simulation server runs 50 cycles per second. Each action has minor impact individually, but can collapse the motion in future. Furthermore, the final objective is inject velocity in the ball and it happens only after the whole motion. So, the real reward is very delayed.
- **Partially-observed state**: The agent doesn't have full information from the environment and perception data is noisy. Therefore, all information used in training is liable to have errors that can make learning harder.
- **High-dimensional, continuous action space**: In the kick motion, at each timestep, the agent have to act in each of 22 joints. Each joint can assume any possible angle. If we consider the agent could assume only two possible actions for each joint, then it has $2^{22} = 4.194.304$ possible combinations of actions. Just to compare: the average number of actions in chess is 35; in Go, 250.

5.2 Experimentation Setup

In order to find the best method for learn the Kick Motion, we tested several techniques to compare them and find out that one which converges faster and results in the better and more robust kick. In the next subsections, we will describe each of the methods evaluated in this work.

5.2.1 Hybrid Learning Model – HLM

In the Hybrid Learning Model (HLM), the training process occurs in two phases. The first one, a supervised learning phase, we learn the keyframe used by using a neural network, reproducing almost the same motion that we already had previously. The intuition behind this is if the agent starts from a good initial point in the optimization problem, it could be easier to get better results applying the gradient in the neighborhood of that point. The reward starts with a good value and the starting motion is well defined. Otherwise, if the optimization starts in a random point, it could be impossible to reach a good motion and probably the agent will get stuck in a local optima. This idea is shown in Figure 5.1.

FIGURE 5.1 – In the hybrid model, we can ensure the starting point is near of the optimal solution (orange arrow); otherwise, the starting point can be bad and harder to optimize (red arrow).

5.2.2 Reinforcement with Naive Reward - RNR

In this learning model, we just use reinforcement learning with a simple and directly reward. As our task is related to kick the ball, the "naive" reward is composed by its velocities after the kick motion, as shown in 5.1, where s, v and u are actual state, the vector of velocity and weight parameters, respectively. We call it naive because we don't pass to the learning model any idea of how the motion have to be performed - just what we expect to have as final objective.

It is important to highlight that the techniques described could be used jointly as building blocks of a more labored model.

$$R(s) = u^T v \tag{5.1}$$

5.2.3 Reinforcement with Reference Reward - RRR

In this learning model we add a term that compares the actual performance with a reference motion, as shown in 5.2:

$$R(s) = w_{ref}^T r \tag{5.2}$$

In this equation, w_{ref} are weight parameters for each joint and r is the absolute value of the difference between joint values in that state and the reference (equation 5.3).

$$r(\theta, \theta_0) = \|\theta - \theta_0\| \tag{5.3}$$

The intuition behind this weight parameters is because we have some joints more important than others. For example, some joints from the leg that kicks the ball themselves could collapse the whole motion if the error is high. On the other hand, joints from the neck are not so important to the kick. Therefore, it's natural to penalize differently in each case.

Lastly, the reference motion used was the keyframe kick previously mentioned and also used in HLM.

5.2.4 Reinforcement with Initial State Distribution - RISD

The Initial State Distribution is a technique related to where the episode starts to collect data to use in the learning. Without this technique, all episodes start in the beginning of the motion. However, the problem of this strategy is because the policy is forced to learn the motion in a sequential manner, first learning the early phases of the motion, and then incrementally progressing towards the later phases. This can be problematic for the kick motion. Another disadvantage of a fixed initial state is the resulting exploration challenge. The policy only receives reward retrospectively, once it has visited a state. Therefore, until a high-reward state has been visited, the policy has no way of learning that this state is favorable (PENG *et al.*, 2018).

To implement Initial State Distribution, we used a uniformly random variable that could assume any value in the set of states from the reference motion previously described. Suppose we pick the value s. Then, we start the episode running reference motion, from its first state until s. Finally, we start to collect data and perform the actions from the running policy.

5.2.5 Reinforcement with Early Termination - RET

The Early Termination technique is related to where the episode ends. Without early termination, all motions stops in the same state, which means that the episode length is fixed. The problem of this is because if the motion fails somehow during its execution, all the data collected after the failure will harm learning and evaluation of later stages.

In the kick motion task, the early termination will be triggered when the robot falls during the motion. This behavior will ensure two things: first, that we don't collect wrong data, as mentioned before; and second, to gain more reward and have longer episodes, the agent will be forced to learn a kick motion that doesn't fall – which will be very good in game situation.

5.3 Supervised Learning Setup

5.3.1 The Dataset

In order to use supervised learning for learning keyframe motions using neural networks in the HLM model, we first need to construct a dataset. A dataset consists of samples of keyframe steps. Samples were collected within the Soccer 3D environment with a frequency of 50 Hz. We acquired these samples in two different ways.

In the first one, we commanded an agent of our team to execute specific motions and sampled the reference joint positions computed by our code. In this case, we sampled the kick and get up keyframe motions (MUNIZ *et al.*, 2016). Notice that, for this approach to be successful, one needs access to the source-code.

The second approach involved changing the Soccer 3D server source-code to provide current joint positions of a given robot, in a similar way as described in (MACALPINE *et al.*, 2013). This allowed us to acquire motion datasets from other teams, without any knowledge of how these movements are implemented. In this case, we collected two types of kicks based on keyframes and sampled joint values of the walking engine (MACALPINE *et al.*, 2012).

5.3.2 Neural Network Architecture and Hyperparameters

The neural network has to be able to learn how to interpolate between samples, which actually happens. The architecture that performed best – in terms of mean absolute error minimization and simplicity – is shown in Figure 5.2. A deep neural network with 2 hidden, fully connected layers of 75 and 50 neurons was used. The output layer has

23 regression neurons, which represent the 22 joint angles and a neuron which output indicates if the motion has ended or not. The neurons in each hidden layer use the LeakyReLU activation function (XU *et al.*, 2015):

$$f(x) = \begin{cases} \alpha x, & x < 0 \\ x, & x \geq 0 \end{cases}$$

where α is a small constant. This activation function was used to improve the representation capacity of the neural network, adding support for non-linear functions.

FIGURE 5.2 – The architecture of a neural network designed to learn motions.

This architecture resulted in thousand of parameters to optimize, as exposed in Table 5.1. A very high number, when compared to more traditional optimization approaches (MACALPINE *et al.*, 2012). Notice that, by increasing the number of parameters usually allows representing better movements.

TABLE 5.1 – The Network Summary

Layer	**Neurons**	**Activation**	**Parameters**
Dense	75	LeakyReLU	150
Dense	50	LeakyReLU	3800
Dense	23	Linear	1173

Total Parameters	5123

5.3.3 The Training Procedure

Since keyframe motions are executed in an open-loop fashion, the sequence of joint positions are always the same for different repetitions, independently of robot's state. Therefore, by adding samples of multiple executions of the same motion would not make our dataset richer. So, we decided to use only one repetition for each movement for faster training. In the case of the walking motion, we collected samples within one walking period.

During the training, we used 50 thousands epochs divided into 5 training phases, where the learning rate was decreased between phases, in order to achieve better performance. First, we executed 30000 epochs, by using the learning rate of 0.001. The other phases had 5000 epochs each, and we decreased the learning rate by 0.0002 in each phase.

Furthermore, we used Adam optimization (KINGMA; BA, 2014), during the whole training. The loss function used was the mean squared error, as explained in Subsec. 3.1. We decided this loss function is adequate for this problem, mainly because it strongly penalizes large errors, which can collapse the whole motion.

5.3.4 The Deployment in the Soccer 3D Environment

In order to perform the network design and the training procedure, we used the Keras (CHOLLET *et al.*, 2015) framework coupled with Tensorflow (ABADI *et al.*, 2015) as backend. After training, the weights were frozen and converted to a specific format, which was readable, by using the Tensorflow C++ API integrated within agent's code. Hence, the training was performed outside the environment, but the agent actually has computed network inferences, during the simulation execution.

5.4 Reinforcement Learning Setup

To use a Reinforcement Learning model, we need to define a policy representation and a task which the agent will follow during the process.

5.4.1 Policy Representation

The policy is represented by a neural network similar to the supervised learning model. A state is described as the stage of kick motion. An action is a set of joint values that has to be applied by the agent in that state.

In HLM, the architecture from both value function and policy networks is that described by section 5.3. On the other cases, the network is composed by three building blocks: an input normalization Filter, the neural networks themselves and a Gaussian action space noise.

5.4.1.1 Input Normalization Filter

The input normalization filter, also know as feature scaling, has the objective of normalize the input to avoid the problem of vanishing and exploding gradients by applying

updates through all dimensions in equal proportions, as illustrated in Figure 5.3. This normalization happens at each pass in the network, re-calculating mean and standard deviation using the new examples. Hence, this method helps during the convergence.

FIGURE 5.3 – Intuition behind input normalization

5.4.1.2 Neural Network

As an actor-critic model, we have two networks: one for the value function and other for the policy itself. They have the same architecture: two layer, fully connected, with 64 neurons in each hidden layer and hyperbolic tangent as activation function. The architecture is illustrated in Figure 5.4. Table 5.2 summarizes the parameters in this new architecture, also considering the parameters from the Gaussian Action Space Noise, described in 5.4.1.3.

FIGURE 5.4 – Architecture used by pure Reinforcement Learning models.

TABLE 5.2 – The Reinforcement Learning Network Summary

Layer	Neurons	Activation	Parameters
Dense	64	*tanh*	128
Dense	64	*tanh*	4160
Output	23	Linear	1495
Noise	23	Linear	23

Total Parameters	5806

5.4.1.3 Gaussian Action Space Noise

The last idea explored in this policy representation is the Gaussian action space noise for better exploration. It adds adaptive noise to the action space of the neural network. Discrete RL uses ϵ-greedy (WATKINS, 1989) to confer exploration. Gaussian action space noise injects normal randomness directly into the actions of the agent, altering the types of decisions it makes and helping algorithms explore continuous environments more effectively. The Figure 5.5 illustrates this technique.

FIGURE 5.5 – Gaussian noise applied to action space to ensure better exploration in continuous environments. (PLAPPERT *et al.*, 2017)

5.4.2 Task Description

The learning task consists in the activity the agent will execute in order to achieve the expected behavior through the policy. In this experimentation, we would like to learn the kick motion. Therefore, is straightforward that the scenario is based on kick the ball.

The agent is started in a fixed position of the soccer field. The ball, on the other hand, starts in a position α_{ball} meters from the agent, in front of him. The parameter α_{ball} can be optimized in order to decide the best position the agent have to stop to kick the ball better. However, for this experimentation, we fixed its value.

We repeat the following flow: in the beginning of the episode, all the joints are initialized in a certain value that correspond to the joints the robot assume when it stops to walk and will start the kick motion. Then, we start the learning steps, applying the joints from the neural network to the robot. When we use RISD, there's a intermediate phase executing the reference motion, as described in 5.2.4.

After the execution of those steps, we finalize the episode turning all the joints to zero. When there's no Early Termination, this happens after a fixed number of steps that we can ensure the motion has over. Otherwise, this occurs when the early termination condition is reached.

Figure 5.6 shows the tasking running.

FIGURE 5.6 – Task used to learning kick motion.

5.5 Infrastructure

5.5.1 Reinforcement Learning Server

Figure 5.7 shows the architecture used to integrate the learning algorithm, agent and simulation environment. The simulation server exposes a state to the soccer agent, which model this information and passes to the learning algorithm that chooses a action accordingly and returns to the agent. Then, it applies this action in the environment, modifying its state, completing the cycle.

5.5.1.1 Simulation Server

The simulation environment used, as mentioned before, is Simpark. It uses a TCP Socket to communicate with the agent and exposes a interface for querying the ground

FIGURE 5.7 – Reinforcement Learning Server Architecture. (MUZIO, 2017)

truth state for many features from the agent - which can be helpful to state evaluation and avoids to use agent's perceived data that has noise.

5.5.1.2 Soccer Agent

The agent used is from ITAndroids Soccer 3D team and it leads with the perception information sent from the server, model the world with this information and make decisions accordingly. Additionally, there's a entity, called Wizard, whose purpose is query the ground truth state. All this code is written in C++ and the task described in 5.6 was implemented in the agent itself.

Other tool that worth mention is RoboViz, a publicly available visualization tool that allows a few improvements over the default visualization tool. We used it to verify if the task was implemented successfully.

5.5.1.3 Learning Agent

We created another interface to integrate the Soccer Agent with the algorithms used in this experimentation. These were implemented by OpenAI Baselines (DHARIWAL *et al.*, 2017), a repository of high-quality implementations of reinforcement learning algorithms whose purpose is make easier for the research community to replicate, refine, and identify new ideas, and create good baselines to build research on top of. The community developed those algorithms using Python language with the Tensorflow (ABADI *et al.*, 2015) framework.

This integration was done using gRPC (GRPC, 2018), an open source remote procedure call (RPC) system that generates cross-platform client and server bindings. It uses HTTP/2 for transport and Protocol Buffers (VARDA, 2008) as the interface description language, serializing structured data.

5.5.2 Neural Network Deployment

When we use the architecture from 5.5.1, two problems related to the neural network deployment arise.

The first one is related to the HLM. We have a supervised learning model trained using the framework Keras outside the environment and with no relation to OpenAI Baselines. Therefore, we need a way to import this model from Keras model into the learning algorithms.

The second problem is to deploy the network from learning algorithm to the soccer agent. When the network is trained using OpenAI Baselines, there's no contact between the soccer agent and the model. As described in 5.5.1.3, there's a interface to communicate the actions between them. All calculations happens in only one side. However, during a game, it's not possible to maintain this dependency, so we need a way to run the network inside soccer agent and without the whole training structure.

5.5.2.1 Import Supervised Model into OpenAI Baselines

To solve this problem, it's important to understand how Keras works and what's the challenge behind this import.

Keras works on top of lower level machine learning frameworks. Its purpose is make easier to prototype and train models with few lines of code. With Keras, developers don't need to understand in deep how computational graphs works and some math behind the models. Actually, this framework wraps code from the lower level ones. Therefore, even being easier to use, all the computation and data structures remains the same.

As back-end framework that Keras runs on top of, we used Tensorflow. As described previously, it creates a computational graph with math tensors as inputs that "flow" inside and returns the expected result.

Given that context, the idea is straightforward: we create a "Kick Policy" class, that creates the model using Keras, load its weights, gets the output tensor and injects in the rest of Tensorflow's computational graph related to the learning algorithm. The Algorithm 9 explains sequentially this idea.

Algorithm 9 Import Keras model into Tensorflow

Require: Keras output model
  $model \leftarrow createKerasModelStructure()$
  $model.load_weights(keras_output_model)$
  $outputTensor \leftarrow model.output$
  Inject $outputTensor$ in the rest of computational graph

It's important to mention that the *load_weights* operation already initializes the graph variables and if we initialize them again, we lose the pre-trained weights.

The biggest problem of this integration is we can't use Input Normalization Filter and Gaussian Action Space Noise, described in 5.4.1.1 and 5.4.1.3, respectively. Therefore, we can face gradient and exploration problems during learning.

5.5.2.2 Export OpenAI Baselines onto Soccer Agent

As mentioned earlier, Tensorflow creates a data flow graph with all operations executed during the algorithm. Figure 5.8 shows the whole graph for PPO used in this experimentation. However, we need to export only the policy network itself.

FIGURE 5.8 – Data flow graph for PPO algorithm.

It is important to mention that "policy network" mean only the neural network that maps state to actions, and not the whole policy node "pi" in Figure 5.8. The "pi" node also have the value function network and gaussian noise in action space, only needed during the learning process. Figure 5.9 shows the whole "pi" node.

Therefore, to export this policy network, we need to add a new "identity" node in the output from that. This node has the same resultant tensor, but it is possible to label it and we can freeze the subgraph whose output is the labeled node itself. Internally, Tensorflow will start freezing the output node and its dependencies, recursing until input nodes. Then, we save this data structure in a proto file that we read using the C++ API in the soccer agent. Figure 5.10 and 5.11 show the policy data flow graphs from Figures 5.4 and 5.2, respectively.

FIGURE 5.9 – "Pi" data flow node from PPO.

FIGURE 5.10 – Policy data flow graph from Figure 5.4

FIGURE 5.11 – Policy data flow graph from Figure 5.2

5.5.3 Distributed Training

Reinforcement Learning training is much more costly computationally when compared to the supervised one described in 5.3. Additionally to the costs of applying gradients in the network, we need to gather all the data from agent simulations. Furthermore, since the data distribution changes during the learning, it's harder to converge, se it is needed a large amount of data.

In this context, use a single agent to get all the data and train the algorithm takes too long time to converge and, depending of the problem, it is not viable. We need a way to get more data faster and accelerate the training.

5.5.3.1 Data Parallelism

The solution for this challenge is distribute the training using several agents. Each agent interacts with its own environment, but the results are applied to the same network. This technique is called Data Parallelism, because the parallelization happens in data gathering. It opposes the idea of Model Parallelism, where the learning model itself is distributed, but this technique is not interesting in reinforcement learning training.

In Data Parallelism, we applied a master-workers architecture, as shown in Figure 5.12. It has several worker nodes, whose job is gather data by simulating the agent, and a master node, which controls the learning algorithm. This architecture works very well in this experiment because the computation from workers are very similar so we can conduct several independent processes in those. It can also be fault tolerant: if one worker dies, the learning process can keep running.

FIGURE 5.12 – Master-workers architecture for data parallelism.

5.5.3.2 Synchronous vs. Asynchronous Distributed Training

There are two ways to apply updates from a master-workers architecture in the learning algorithm. Both ways are illustrated in Figure 5.13.

In asynchronous training, each worker executes the task, gathers data, calculate the gradient and apply it in the neural network. There's no interaction between workers so the learning updates can occur freely and, therefore, faster. However, the downside is that each worker has its own version of the network which can cause a discrepancy between them and slows the convergence.

On the other side, in the synchronous training, each worker executes the same job as the asynchronous one. However, the master node collects the gradients and applies an average from them. In this way, the updates occurs in the same frequency, but they are much more precise (and therefore we can use large learning rates).

(HEESS *et al.*, 2017a) implemented a variation of PPO called Distributed PPO (DPPO) and their experiments shows that synchronous updates worked better in practice. Hence, in this experimentation we also used this type of distributed training.

FIGURE 5.13 – Synchronous and Asynchornous Distributed Training.

5.5.4 Metrics

We need evaluation metrics to compare the techniques in section 5.2. Several of them were collected and described below:

- **Episode Reward Mean**: This metric is the mean of cumulate reward during the whole episode. It's directly related to the motion performance. Then, it's the main metric in the comparison.
- **Value Function Loss**: Says how good is the prediction of value function by the network.
- **Timesteps So Far**: It means how many cycles (observation-action-reward) have been executed. This metric was used to compare the speedup from distributed training.
- **Episode Length Mean**: In the case of models with no fixed length, this metric will allow us to understand how the learning worked.

5.5.5 Monitoring via Tensorboard

Lastly, we need a tool whose purpose is monitor the metrics collected during the training. We used Tensorboard (WONGSUPHASAWAT *et al.*, 2017), a graphic visualizer tool from Tensorflow. Figure 5.14 shows the GUI with some metrics. Using that, we can compare several models and their metrics, visualize computational graphs and debug the learning process.

FIGURE 5.14 – Monitoring training metrics with Tensorboard.

6 Results' Analysis and Discussion

6.1 Training Results

FIGURE 6.1 – Plots of mean squared error and mean absolute error, during training.

The initial results came from the training procedure, outside the simulation environment. Figure 6.1 presents training curves for the kick keyframe dataset. In this case, plots show the mean squared error and the mean absolute error metrics. In both metrics, the value drastically decreases in the first epochs. This same behavior was present in other training procedures as well. However, only after thousands of epochs, the network has achieved a low error that successfully reproduced the motion, which has shown how sensible to small joint errors keyframes were, given that they were open-loop motions.

The peaks, during the training has happened at the learning rate transition instants, but they did not hurt the training procedure itself.

6.2 The Learned Kick Motion

The final mean absolute error was **0.018** radians and the motion was visually indistinguishable from the original one, as can be seen in Figure 6.2. In this figure, snapshots from both motions were taken. Figure 6.3 shows several plots of joint angles, by comparing the original and learned kick motions. As we may see, the learned motion has fitted the movement with minor errors[1].

FIGURE 6.2 – The kick motion. The first row of figures shows the original kick motion. The second row shows the learned kick motion. Both motions are visually indistinguishable.

In order to evaluate the learned kick motion in the RoboCup Soccer 3D domain, we created a statistical test. Inside the test scenario, the ball was initially placed in the center of the field with an agent near to it. The only action of the agent was to kick the ball in the goal direction. After the kick, the agent run until reaching the ball and kicked it again, repeating this process till scoring a goal. When the goal has occurred, this same scenario was repeated. The whole test was conducted, during thirty minutes in clock time and the following data was collected: total number of kicks, number of successful kicks, mean distance that the ball has traveled, and the standard deviation of this measure. The results from the original and learned kicks is shown in Table 6.1.

[1] Kick results video: https://youtu.be/UAbqQLUnvDo

FIGURE 6.3 – Joint values for comparing original and learned kicks. The neural network was able to fit the joint trajectories with small errors.

TABLE 6.1 – The Kick Comparison

Kick Type	**Statistics**		
	Accuracy (%)	**Distance** (m)	
		Mean	***Std***
Original Kick	64.5	8.92	3.82
Neural Kick	52.6	7.16	4.06

Although both kicks had similar results, the original kick was slightly better in this scenario. By confronting Figure 6.3, we can conclude that even with an almost equal representation, the kick lost part of its efficiency and this fact has shown how sensible were movements based on keyframe data.

6.3 The Learned Walk Motion

By using the modified server described in Subsec. 5.3.1, a dataset with samples of the UT Austin Villa's walking motion (MACALPINE *et al.*, 2013) was acquired. This team is the current champion of the RoboCup Soccer 3D competition (MACALPINE; STONE, 2018).

The objective was to mimic the walk motion as a keyframe and has used that in our agent. The previously described framework for learning our own kick motion was used in this training, by including the neural network architecture and its hyperparameters.

The results from this training are shown in Figure 6.4. Similarly to Figure 6.3, it shows the joint angles throughout the walking motion period for the original and learned walk. Additionally, it shows the real joints values from the movement in the server. These joints were chosen because they were the most dynamic in the walk motion and, therefore, the hardest to learn.

FIGURE 6.4 – Joints positions, during a period of the walking motion for the original walk, and the learned walk and the joints positions effectively attained, during the learned walking motion.

The learned motion has fitted the dataset very well. However, these values were just desired joints. In fact, these values were used as references to joint controllers and were also attenuated due to joint dynamics. Furthermore, this motion was operated in a open-loop fashion, so the agent was not able to correct its own trajectory, and this walks got biased within the simple task of walking straight forward.

Despite the facts previously described, the motion has worked well in a non-competitive scenario[2], which was shown in the metrics collected from the Forward Walk test scenario – agent walking forward from the goal post until the center line of the field – in Table 6.2 and the visual representation in Figure 6.5.

TABLE 6.2 – Walk Comparison - Forward Walk

Walk Type	**Statistics**			
	Velocity (m/s)		**Y Error** (m)	
	Mean	***Std***	***Mean***	***Std***
Original Walk	0.87	0.01	-	-
Learned Walk	0.23	0.01	0.96	2.63

(a) (b) (c)

FIGURE 6.5 – The walking motions comparison. Figure (a) shows our agent in its regular walk, Figure (b) shows the same agent mimicking UT Austin Villa walk, and Figure (c) shows the UT Austin Villa agent itself performing his own walking motion.

6.4 Other motions

This same framework was used to learn other keyframe motions originated from our agent itself, such as the get up motion. As the cases previously described, the resultant

[2] Walk results video: https://youtu.be/-pHxTrxllyY

neural network was capable of mimicking the keyframe, by including its interpolation. Hence, all of our keyframe motions could be replaced by neural motions with similar performance.

However, the huge improvement of this method was about mimicking other teams motions. In the Soccer 3D environment, movements like kick and walking have giant impact in team's performance. With this learning framework, our agent was able to mimic multiples movements from several teams.

As an example, we have collected data from UT Austin Villa kick, which was originally optimized by using Deep Reinforcement Learning techniques (MACALPINE; STONE, 2017). Our agent has learned this kick without any additional optimization strategy: we just have used samples collected from the modified server.

7 Conclusions, Recommendations, and Future Works

7.1 Preliminary Conclusions and Future Works

In this work, we presented a method for learning humanoid robot movements using datasets composed of joint values at each time instant. The provided learning framework was capable to learn several types of motion, including walk and kick, without any change in network architecture or hyperparameters.

Moreover, the learned motions had similar performance to the original ones. Furthermore, this framework was able to learn from other teams motions, without any knowledge about the underlying implementation – only by using the joints values provided by a modified version of the server. This was a huge improvement, in terms of getting improved motions, as our agent was able to mimic other teams motions, by using this machine learning technique.

As future works, we plan to apply Deep Reinforcement Learning algorithms to obtain faster and more robust kicks, by using as a "seed" the neural network obtained from this work.

Another track to be followed is to transfer the learning of this obtained network to a new network that represents the motion policy itself (i.e a network which has as inputs the current state of the robot, by including joint and link states, besides the current time instant), optimizing this motion policy, in order to get a closed-loop walking and kicking motion that can correct itself.

Finally, as a long term goal, we intend to create some model-free walking and kicking engines.

7.2 The Activities Plan

As next steps of this work, we plan to:

1. Import our supervised policy model into Deep Reinforcement Learning algorithms – Expected to be finished until the end of **June, 2018**;

2. Iterate over objective function construction and optimization, in order to improve the kick motion – Expected to be finished until mid **August, 2018**;

3. Execute the same test scenarios previously applied to compare results – Expected to be finished until the end of **August, 2018**;

4. Test in 11 x 11 game to compare team performance – Expected to be finished until the end of **August, 2018**; and

5. Complement this work with some novel background, methodology, results, and conclusions – Expected to be finished until **November, 2018**.

Bibliography

ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JOZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANé, D.; MONGA, R.; MOORE, S.; MURRAY, D.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P.; VANHOUCKE, V.; VASUDEVAN, V.; VIéGAS, F.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Software available from tensorflow.org. Disponível em: <https://www.tensorflow.org/>.

ABBEEL, P.; SCHULMAN, J.; CHEN, X. P. Lecture, **Policy Gradients**. 2017. Disponível em: <https://drive.google.com/file/d/0BxXI_RttTZAhY216RTMtanBpUnc/view>.

ANDRYCHOWICZ, M.; WOLSKI, F.; RAY, A.; SCHNEIDER, J.; FONG, R.; WELINDER, P.; MCGREW, B.; TOBIN, J.; ABBEEL, P.; ZAREMBA, W. Hindsight experience replay. **CoRR**, abs/1707.01495, 2017. Disponível em: <http://arxiv.org/abs/1707.01495>.

BARTELS, R. H.; BEATTY, J. C.; BARSKY, B. A. **An Introduction to Splines for Use in Computer Graphics &Amp; Geometric Modeling**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987. ISBN 0-934613-27-3.

BROCKMAN, G.; DENNISON, C.; ZHANG, S.; JÓZEFOWICZ, R.; TANG, J.; WOLSKI, F.; SIDOR, S.; DĘBIAK, P.; FARHI, D.; PONDÉ, H.; RAIMAN, J.; PETROV, M.; CHAN, B.; PACHOCKI, J.; CHRISTIANO, P.; RADFORD, A.; SIGLER, E.; CLARK, J.; GRAY, S.; YOON, D.; SUTSKEVER, I.; SCHIAVO, L.; SCHULMAN, J.; FISCHER, Q.; HASHME, S.; LUAN, D.; HESSE, C.; BERNER, C.; SCHNEIDER, J. Openai five. In: . [s.n.], 2018. Disponível em: <https://blog.openai.com/openai-five/>.

BROWNLEE, J. Blog, **A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size**. 2017. Disponível em: <https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/>.

CHOLLET, F. *et al.* **Keras**. 2015. Https://keras.io.

COLLINS, S.; RUINA, A.; TEDRAKE, R.; WISSE, M. Efficient Bipedal Robots Based on Passive Dynamic Walkers. **Science Magazine**, v. 307, p. 1082–1085, February 2005.

DABBURA, I. Blog, **Gradient Descent Algorithm and Its Variants**. 2017. Disponível em: <https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3>.

DHARIWAL, P.; HESSE, C.; KLIMOV, O.; NICHOL, A.; PLAPPERT, M.; RADFORD, A.; SCHULMAN, J.; SIDOR, S.; WU, Y. **OpenAI Baselines**. [S.l.]: GitHub, 2017. `https://github.com/openai/baselines`.

GLOROT, X.; BENGIO, Y. Understanding the difficulty of training deep feedforward neural networks. In: **In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics**. [S.l.: s.n.], 2010.

GLOROT, X.; BORDES, A.; BENGIO, Y. Deep sparse rectifier neural networks. In: GORDON, G. J.; DUNSON, D. B.; DUDíK, M. (Ed.). **AISTATS**. [S.l.]: JMLR.org, 2011. (JMLR Proceedings, v. 15), p. 315–323.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. `http://www.deeplearningbook.org`: MIT Press, 2016.

GOUAILLIER, D.; HUGEL, V.; BLAZEVIC, P.; KILNER, C.; MONCEAUX, J.; LAFOURCADE, P.; MARNIER, B.; SERRE, J.; MAISONNIER, B. Mechatronic design of nao humanoid. In: **2009 IEEE International Conference on Robotics and Automation**. Kobe, Japan: IEEE, 2009. p. 769–774. ISSN 1050-4729.

GRPC. 2018. `https://grpc.io/`. Accessed: 2018-08-20.

HANSEN, N. The CMA evolution strategy: A tutorial. **CoRR**, abs/1604.00772, 2016. Disponível em: <http://arxiv.org/abs/1604.00772>.

HEESS, N.; TB, D.; S, S.; LEMMON, J.; MEREL, J.; WAYNE, G.; TASSA, Y.; EREZ, T.; WANG, Z.; ESLAMI, S. M. A.; RIEDMILLER, M.; SILVER, D. Emergence of locomotion behaviours in rich environments. **arXiv**, Jul 2017.

HEESS, N.; TB, D.; SRIRAM, S.; LEMMON, J.; MEREL, J.; WAYNE, G.; TASSA, Y.; EREZ, T.; WANG, Z.; ESLAMI, S. M. A.; RIEDMILLER, M. A.; SILVER, D. Emergence of locomotion behaviours in rich environments. **CoRR**, abs/1707.02286, 2017. Disponível em: <http://arxiv.org/abs/1707.02286>.

HESTENES, M. R.; STIEFEL, E. Methods of conjugate gradients for solving linear systems. **Journal of research of the National Bureau of Standards**, v. 49, p. 409–436, 1952.

HINTON, G.; TIELEMAN, T. Blog, **Divide the gradient by a running average of its recent magnitude**. 2017. Disponível em: <https://www.coursera.org/lecture/neural-networks/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude-YQHki>.

HO, J.; ERMON, S. Generative adversarial imitation learning. **CoRR**, abs/1606.03476, 2016. Disponível em: <http://arxiv.org/abs/1606.03476>.

JIE, T.; ABBEEL, P. On a connection between importance sampling and the likelihood ratio policy gradient. In: LAFFERTY, J. D.; WILLIAMS, C. K. I.; SHAWE-TAYLOR, J.; ZEMEL, R. S.; CULOTTA, A. (Ed.). **Advances in Neural Information Processing Systems 23**. Curran Associates, Inc., 2010. p. 1000–1008. Disponível em: <http://papers.nips.cc/paper/3922-on-a-connection-between-importance-sampling-and-the-likelihood-ratio-policy-gradient.pdf>.

KAJITA, S.; KANEHIRO, F.; KANEKO, K.; YOKOI, K.; HIRUKAWA, H. The 3D Linear Inverted Pendulum Mode: A simple modeling for a biped walking pattern generation. In: **In Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems**. Hawaii, USA: IEEE, 2001.

KAKADE, S.; LANGFORD, J. Approximately optimal approximate reinforcement learning. In: **IN PROC. 19TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING**. [S.l.: s.n.], 2002. p. 267–274.

KHAN, N. A parallel implementation of the covariance matrix adaptation evolution strategy. **CoRR**, abs/1805.11201, 2018. Disponível em: <http://arxiv.org/abs/1805.11201>.

KINGMA, D. P.; BA, J. Computer science > learning adam: A method for stochastic optimization. **arXiv**, Dec 2014.

KITANO, H.; ASADA, M.; KUNIYOSHI, Y.; NODA, I.; OSAWAI, E.; MATSUBARA, H. Robocup: A challenge problem for ai and robotics. In: KITANO, H. (Ed.). **RoboCup-97: Robot Soccer World Cup I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 1–19. ISBN 978-3-540-69789-3.

LILLICRAP, T. P.; HUNT, J. J.; PRITZEL, A.; HEESS, N.; EREZ, T.; TASSA, Y.; SILVER, D.; WIERSTRA, D. Continuous control with deep reinforcement learning. **CoRR**, abs/1509.02971, 2015. Disponível em: <http://arxiv.org/abs/1509.02971>.

LU, C.; TANG, X. Surpassing human-level face verification performance on LFW with gaussianface. **CoRR**, abs/1404.3840, 2014. Disponível em: <http://arxiv.org/abs/1404.3840>.

MACALPINE, P.; BARRETT, S.; URIELI, D.; VU, V.; STONE, P. Design and optimization of an omnidirectional humanoid walk: A winning approach at the RoboCup 2011 3D simulation competition. In: **Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)**. Toronto, Ontario, Canada: AAAI, 2012.

MACALPINE, P.; COLLINS, N.; LOPEZ-MOBILIA, A.; STONE, P. Ut austin villa: Robocup 2012 3d simulation league champion. In: CHEN, X.; STONE, P.; SUCAR, L. E.; ZANT, T. van der (Ed.). **RoboCup 2012: Robot Soccer World Cup XVI**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 77–88. ISBN 978-3-642-39250-4.

MACALPINE, P.; STONE, P. Prioritized role assignment for marking. In: BEHNKE, S.; LEE, D. D.; SARIEL, S.; SHEH, R. (Ed.). **RoboCup 2016: Robot Soccer World Cup XX**. Berlin: Springer Verlag, 2016, (Lecture Notes in Artificial Intelligence). p. 306–18.

MACALPINE, P.; STONE, P. UT Austin Villa 3D Simulation Soccer Team 2017. In: . Nagoya, Japan: RoboCup Symposium, 2017.

MACALPINE, P.; STONE, P. UT Austin Villa: RoboCup 2017 3D simulation league competition and technical challenges champions. In: SAMMUT, C.; OBST, O.; TONIDANDEL, F.; AKYAMA, H. (Ed.). **RoboCup 2017: Robot Soccer World Cup XXI**. Japan: Springer, 2018, (Lecture Notes in Artificial Intelligence).

MACALPINE, P.; URIELI, D.; BARRETT, S.; KALYANAKRISHNAN, S.; BARRERA, F.; LOPEZ-MOBILIA, A.; STIURCA, N.; VU, V.; STONE, P. **UT Austin Villa 2011 3D Simulation Team Report**. Austin, Texas, December 2011.

MAGALHAES, G. I. **Reconhecimento de gestos da Linguagem Brasileira de Sinais**. Bachelor's Thesis — ITA, 2017.

MAXIMO, M. R. O. de A. **Otimizacao de Caminhada de Robos Humanoides**. Bachelor's Thesis — ITA, 2012.

MNIH, V.; BADIA, A. P.; MIRZA, M.; GRAVES, A.; LILLICRAP, T. P.; HARLEY, T.; SILVER, D.; KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. **CoRR**, abs/1602.01783, 2016. Disponível em: <http://arxiv.org/abs/1602.01783>.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIK, A.; ANTONOGLOU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S.; HASSABIS, D. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved., v. 518, n. 7540, p. 529–533, February 2015. ISSN 00280836. Disponível em: <http://dx.doi.org/10.1038/nature14236>.

MUNIZ, F.; MAXIMO, M. R. O. A.; RIBEIRO, C. H. C. Keyframe movement optimization for simulated humanoid robot using a parallel optimization framework. In: **2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)**. Recife, Brazil: IEEE, 2016. p. 79–84.

MUZIO, A. F. V. **Deep Reinforcement Learning Applied to Humanoid Robots**. Bachelor's Thesis — ITA, 2017.

OBST, O.; ROLLMAN, M. Spark – a generic simulator for physical multiagent simulation. **Computer Systems Science and Engineering**, v. 5, p. 347–356, 2005.

PENG, X. B.; ABBEEL, P.; LEVINE, S.; PANNE, M. van de. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. **ACM Transactions on Graphics (Proc. SIGGRAPH 2018 - to appear)**, v. 37, n. 4, 2018.

PLAPPERT, M.; HOUTHOOFT, R.; DHARIWAL, P.; SIDOR, S.; ABBEEL, P.; ANDRYCHOWICZ, M.; CHEN, R.; CHEN, X.; ASFOUR, T. Better exploration with parameter noise. In: . [s.n.], 2017. Disponível em: <https://blog.openai.com/better-exploration-with-parameter-noise/>.

RUBINSTEIN, R. Y.; KROESE, D. P. **The Cross Entropy Method: A Unified Approach To Combinatorial Optimization, Monte-carlo Simulation (Information Science and Statistics)**. Berlin, Heidelberg: Springer-Verlag, 2004. ISBN 038721240X.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **Nature**, v. 323, n. 533, 1988.

RUMMERY, G. A.; NIRANJAN, M. **On-Line Q-Learning Using Connectionist Systems**. [S.l.], 1994.

SCHULMAN, J. Lecture, **Advanced Policy Gradient Methods: Natural Gradient, TRPO and More**. 2017. Disponível em: <https://drive.google.com/file/d/0BxXI_RttTZAhMVhsNk5VSXU0U3c/view>.

SCHULMAN, J.; LEVINE, S.; MORITZ, P.; JORDAN, M. I.; ABBEEL, P. Trust region policy optimization. **CoRR**, abs/1502.05477, 2015. Disponível em: <http://arxiv.org/abs/1502.05477>.

SCHULMAN, J.; MORITZ, P.; LEVINE, S.; JORDAN, M. I.; ABBEEL, P. High-dimensional continuous control using generalized advantage estimation. **CoRR**, abs/1506.02438, 2015. Disponível em: <http://arxiv.org/abs/1506.02438>.

SCHULMAN, J.; WOLSKI, F.; DHARIWAL, P.; RADFORD, A.; KLIMOV, O. Proximal policy optimization algorithms. **CoRR**, abs/1707.06347, 2017. Disponível em: <http://arxiv.org/abs/1707.06347>.

SHON, A. P.; GROCHOW, K.; RAO, R. P. N. Robotic imitation from human motion capture using gaussian processes. In: **In Proceedings of the IEEE/RAS International Conference on Humanoid Robots (Humanoids**. Tsukuba, Japan: [s.n.], 2005.

SILVER, D. Lecture, **Markov Decision Processes**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf>.

SILVER, D. Lecture, **Planning by Dynamic Programming**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/DP.pdf>.

SILVER, D. Lecture, **Policy Gradients**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf>.

SILVER, D. Lecture, **Value Function Approximation**. 2015. Disponível em: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/FA.pdf>.

SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILLICRAP, T. P.; SIMONYAN, K.; HASSABIS, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. **CoRR**, abs/1712.01815, 2017. Disponível em: <http://arxiv.org/abs/1712.01815>.

SILVER, D.; HUBERT, T.; SCHRITTWIESER, J.; ANTONOGLOU, I.; LAI, M.; GUEZ, A.; LANCTOT, M.; SIFRE, L.; KUMARAN, D.; GRAEPEL, T.; LILLICRAP, T. P.; SIMONYAN, K.; HASSABIS, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. **CoRR**, abs/1712.01815, 2017. Disponível em: <http://arxiv.org/abs/1712.01815>.

SILVER, D.; SCHRITTWIESER, J.; SIMONYAN, K.; ANTONOGLOU, I.; HUANG, A.; GUEZ, A.; HUBERT, T.; BAKER, L.; LAI, M.; BOLTON, A.; CHEN, Y.; LILLICRAP, T.; HUI, F.; SIFRE, L.; DRIESSCHE, G. van den; GRAEPEL, T.; HASSABIS, D. Mastering the game of go without human knowledge. **Nature**, Macmillan Publishers Limited, part of Springer Nature. All rights reserved., v. 550, p. 354–, October 2017. Disponível em: <http://dx.doi.org/10.1038/nature24270>.

SKINNER, B. **Science And Human Behavior**. Free Press, 1953. (A Free Press paperback). ISBN 9780029290408. Disponível em: <https://books.google.com.br/books?id=Pjjknd1HREIC>.

SUTSKEVER, I.; MARTENS, J.; DAHL, G.; HINTON, G. On the importance of initialization and momentum in deep learning. In: **Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28**. JMLR.org, 2013. (ICML'13), p. III–1139–III–1147. Disponível em: <http://dl.acm.org/citation.cfm?id=3042817.3043064>.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. MIT Press, 1998. Disponível em: <http://www.cs.ualberta.ca/˜sutton/book/the-book.html>.

TANIKIC, D.; DESPOTOVIC, V. Artificial intelligence techniques for modelling of temperature in the metal cutting process. **IntechOpen**, September 2012.

THORNDIKE, E. L. A proof of the law of effect. **Science**, American Association for the Advancement of Science, v. 77, n. 1989, p. 173–175, 1933. ISSN 0036-8075. Disponível em: <http://science.sciencemag.org/content/77/1989/173.2>.

TIBSHIRANI, R. Lecture, **Gradient Descent: Convergence Analysis**. 2013. Disponível em: <http://www.stat.cmu.edu/˜ryantibs/convexopt-F13/scribes/lec6.pdf>.

TODOROV, E.; EREZ, T.; TASSA, Y. Mujoco: A physics engine for model-based control. In: **IROS**. [S.l.]: IEEE, 2012. p. 5026–5033. ISBN 978-1-4673-1737-5.

URIELI, D.; MACALPINE, P.; KALYANAKRISHNAN, S.; BENTOR, Y.; STONE, P. On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer. In: TUMER, K.; YOLUM, P.; SONENBERG, L.; STONE, P. (Ed.). **Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)**. [S.l.]: IFAAMAS, 2011. v. 2, p. 769–776. ISBN 978-0-9826571-5-7.

VARDA, K. **Protocol Buffers: Google's Data Interchange Format**. [S.l.], 6 2008. Disponível em: <http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html>.

WANG, Z.; BAPST, V.; HEESS, N.; MNIH, V.; MUNOS, R.; KAVUKCUOGLU, K.; FREITAS, N. de. Sample efficient actor-critic with experience replay. **CoRR**, abs/1611.01224, 2016. Disponível em: <http://arxiv.org/abs/1611.01224>.

WATKINS, C. J. C. H. **Learning from Delayed Rewards**. Tese (Doutorado) — King's College, Cambridge, UK, May 1989. Disponível em: <http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf>.

WONGSUPHASAWAT, K.; SMILKOV, D.; WEXLER, J.; WILSON, J.; MANE, D.; FRITZ, D.; KRISHNAN, D.; VIEGAS, F.; WATTENVERG, M. Visualizing dataflow graphs of deep learning models in tensorflow. In: . [s.n.], 2017. Disponível em: <http://idl.cs.washington.edu/files/2018-TensorFlowGraph-VAST.pdf>.

XIONG, W.; DRO, J.; HUANG, X.; SEIDE, F.; SELTZER, M.; STOLCKE, A.; YU, D.; ZWEIG, G. Achieving human parity in conversational speech recognition. **CoRR**, abs/1610.05256, 2016. Disponível em: <http://arxiv.org/abs/1610.05256>.

XU, B.; WANG, N.; CHEN, T.; LI, M. Empirical evaluation of rectified activations in convolutional network. **arXiv**, Nov 2015.

XU, X.; HU, D.; LU, X. Kernel-based least squares policy iteration for reinforcement learning. v. 18, p. 973–92, 08 2007.

XU, Y.; VATANKHAH, H. Simspark: An open source robot simulator developed by the robocup community. In: BEHNKE, S.; VELOSO, M.; VISSER, A.; XIONG, R. (Ed.). **RoboCup 2013: Robot World Cup XVII**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 632–639. ISBN 978-3-662-44468-9.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO N	4. N DE PÁGINAS
TC	June 12th, 2018	DCTA/ITA/DM-018/2015	99

5. TÍTULO E SUBTÍTULO:
A Deep Reinforcement Learning Method for Humanoid Kick Motion

6. AUTOR(ES):
Luckeciano Carvalho Melo

7. INSTITUIÇÃO(ÕES)/ÓRGAO(S) INTERNO(S)/DIVISÃO(ÕES):
Aeronautics Institute of Technology – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:
Deep Reinforcement Learning; Robotics; Artificial Intelligence

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:
Deep Reinforcement Learning; Robotics; Artificial Intelligence

10. APRESENTAÇÃO: (**X**) **Nacional** ( ) **Internacional**
ITA, São José dos Campos, 2018. Trabalho de Graduação.

11. RESUMO:
Controlling a high degrees of freedom humanoid robot is acknowledged as one of the hardest problems in Robotics. Due to the lack of mathematical models, an approach frequently employed is to rely on human intuition to design keyframe movements by hand, usually aided by graphical tools. In this paper, we propose a learning framework based on neural networks in order to mimic humanoid robot movements. The developed technique does not make any assumption about the underlying implementation of the movement, therefore both keyframe and model-based motions may be learned. The framework was applied in the RoboCup 3D Soccer Simulation domain and promising results were obtained using the same network architecture for several motions, even when copying motions from another teams.

12. GRAU DE SIGILO:
(**X**) **OSTENSIVO** ( ) **RESERVADO** ( ) **SECRETO**