

UNIVERSIDADE FEDERAL FLUMINENSE

Ashey John Masi Noblega

**ADAPTIVE NEURAL REINFORCEMENT  
LEARNING-BASED AGENT FOR GAME  
BALANCING**

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

Ashey John Masi Noblega

# ADAPTIVE NEURAL REINFORCEMENT LEARNING-BASED AGENT FOR GAME BALANCING

Dissertação de Mestrado apresentada ao  
Programa de Pós-Graduação em Com-  
putação da Universidade Federal Flumi-  
nense como requisito parcial para a obtenção  
do Grau de Mestre em Computação.  
Área de concentração:  
Engenharia de Sistemas e Informação

Orientador:

Aline Marins Paes Carvalho

Coorientador:

Esteban Walter Gonzalez Clua

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE  
Gerada com informações fornecidas pelo autor

M394a Masi noblega, Ashey John  
Adaptive Neural Reinforcement Learning-based Agent for Game  
Balancing / Ashey John Masi noblega ; Aline Marins Paes  
Carvalho, orientadora ; Esteban Walter Gonzalez Clua,  
coorientadora. Niterói, 2019.  
66 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense,  
Niterói, 2019.

DOI: <http://dx.doi.org/10.22409/PGC.2019.m.06416015730>

1. Game Balancing. 2. Reinforcement Learning. 3. Neural  
Networks. 4. Produção intelectual. I. Paes Carvalho, Aline  
Marins, orientadora. II. Gonzalez Clua, Esteban Walter,  
coorientadora. III. Universidade Federal Fluminense. Instituto  
de Computação. IV. Título.

CDD -



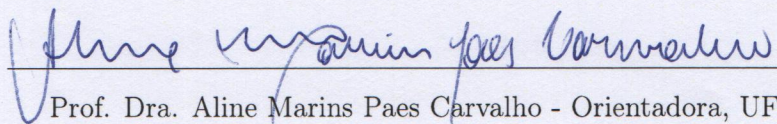
Ashey John Masi Noblega

Adaptive Neural Reinforcement Learning-based Agent for Game Balancing

Dissertação de Mestrado apresentada  
ao Programa de Pós-Graduação em  
Computação da Universidade Federal  
Fluminense como requisito parcial para  
a obtenção do Grau de Mestre em  
Computação. Área de concentração:  
Engenharia de Sistemas e Informação

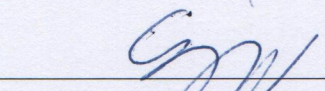
Aprovada em MARÇO de 2019.

BANCA EXAMINADORA



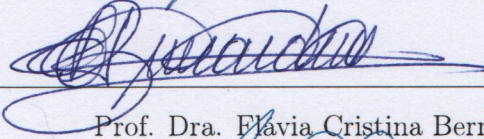
---

Prof. Dra. Aline Marins Paes Carvalho - Orientadora, UFF



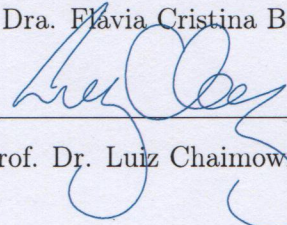
---

Prof. Dr. Esteban Walter Gonzalez Clua - Coorientador, UFF



---

Prof. Dra. Flavia Cristina Bernardini, UFF



---

Prof. Dr. Luiz Chaimowicz, UFMG

Niterói

2019

*Words are, in my not-so-humble opinion, our most inexhaustible source of magic.*

*Capable of both inflicting injury, and remedying it.*

*J.K. Rowling*

# Acknowledgements

First of all, I want to thank my mother Jesusa who has always struggled to give me a good education and supported me in all my decisions. I want to thank my brothers who have constantly given me life advice and believed that I could be here and especially to my brother Jose Luis.

I am very grateful to my advisors Aline Paes and Esteban Clua for constantly guiding me along the way of the Master's degree and for being very patient with the learning pace of each of their students.

I also want to thank my friend Jose, who from the beginning helped me with the process of adapting life in Brazil without any interest.

Finally, to all my friends that I did during the entire period of the Master. By the jokes and madness that encouraged me to stay here longer despite homesickness.



# Resumo

A experiência de um jogador em relação à dificuldade de um videogame é uma das principais razões para ele decidir continuar o jogando ou abandoná-lo. Efetivamente, a retenção de jogadores em certos pontos do jogo é uma das principais preocupações relacionadas ao processo de desenvolvimento de jogos. No entanto, a experiência de um jogador com um jogo é única, tornando impraticável prever como cada jogador irá encarar a jogabilidade. Algumas tentativas para resolver esse problema envolvem coletar as características dos jogadores para classificá-los em um determinado tipo de estilos previamente definidos e, em seguida, obter uma cena tão complexa como os jogadores desse grupo podem suportar. Outras estratégias consistem em criar uma heurística para medir o quão árduo deve ser o ambiente para o jogador. Trabalhos anteriores que utilizam as estratégias mencionadas sofrem com a desvantagem de precisar coletar toda a informação necessária do jogador a fim de enquadrá-lo em um grupo ou definir um número limitado de estilos aos quais o jogador poderia pertencer ou não. Essa dissertação aproveita os recentes avanços em Aprendizado por Reforço (RL) e Redes Neurais (RN) para criar agentes inteligentes dotados da capacidade de aprender a se adaptar às habilidades de diferentes jogadores. Nós nos concentramos em equilibrar a dificuldade do jogo baseado no agente inteligente inimigo com base nas informações que ele observa do ambiente, bem como no estado atual do jogo. Para projetar um agente que aprende a agir ao mesmo tempo que mantém o equilíbrio do jogo evitando tornar-se excessivamente hábil, propomos uma função de recompensa baseada em uma constante de balanceamento. A abordagem proposta demanda que o agente permaneça dentro de um intervalo em torno dessa constante durante o treinamento. Resultados experimentais coletados a partir de um jogo de luta mostram que ao usar tal função de recompensa e combinando informações de diferentes tipos de jogadores, é possível induzir agentes adaptáveis que se ajustem ao perfil do jogador.

**Palavras-chave:** Balanceamento de Jogos, Aprendizado por Reforço, Redes Neurais, Balanceamento Dinâmico, Jogabilidade.

# Abstract

A player's experience related to the difficulty level of a video game is one of the main reasons for him to continue playing or not. Effectively, players retention in certain game points is one of the main issues related to the games development process. However, the player's experience with a game is unique, making impractical to predict how each user will face the gameplay. Some attempts of handling this problem consist of collecting the players' main features and classify them into a particular type of previously defined styles using a set of definitions as complex as these players can play. Other strategies consist of creating a heuristic for measuring how hard must be the challenges for a particular player. Previous works that use the aforementioned strategies suffer from the burden of collecting all the necessary information from the player to fit him in a group or of the weakness of having only a limited number of styles that the player can fit in. This dissertation takes advantage of the recent advances in Reinforcement Learning (RL) and Neural Networks (NN) to create intelligent agents enhanced with the capacity of learning how to adapt themselves to the skills of different users. We are focused on balancing the game difficulty based on enemy intelligence and the information that the agent observes from the virtual environment, such as the current game state. To design an agent that learns to act while, at the same time, maintains the balance of the game by not becoming overly skillful, we propose a reward function based on a balancing constant. The proposed approach requires that the agent stays within an interval around that constant during training. Experimental results collected from a fighting game show that by using such a reward function and combining information from different types of players allow for inducing adaptive agents that fit with the player skill.

**Keywords:** Game Balancing, Reinforcement Learning, Neural Networks, Dynamic Balancing, Playability.



# List of Figures

3.1	Actor-Critic RL Method . . . . .	19
4.1	Overview of the of the ML-Agents Toolkit Structure . . . . .	24
4.2	Framework for Game Balancing . . . . .	28
5.1	The BC-Based Reward Function with $BC = 30$ and $Skill_{max} = 100$ . . . . .	33
5.2	Minimal Agent Balancing Interaction. . . . .	34
6.1	Persistence of the BC throughout the game with $BC = 30$ . . . . .	38
6.2	RL process focused on games. . . . .	39
6.3	Initial stage of the training process with 9 scenes in parallel. . . . .	42
6.4	Intermediate stage of the training process with 9 scenes at the same time. . . . .	43
6.5	Accumulated Reward throughout the training for $BC = 30$ . . . . .	43
6.6	Evaluation on BC-Metric . . . . .	44

# List of Tables

6.1	ML-Agents Default Settings . . . . .	41
6.2	Our Training Settings . . . . .	41
6.3	Results during 100 game sessions . . . . .	45

# List of Acronyms and Abbreviations

RL	:	Reinforcement Learning;
DL	:	Deep Learning;
NN	:	Neural Networks;
PPO	:	Proximal Policy Optimization;
BC	:	Balancing Constant;
BCR	:	BC-Based Reward Function;
BCM	:	BC-Based Balancing Metric.
DNN	:	Deep Neuronal Network.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	3
1.2	Objectives . . . . .	3
1.3	Methodology . . . . .	4
1.4	Contributions . . . . .	5
1.5	Structure of the dissertation . . . . .	5
<b>2</b>	<b>Literature Review</b>	<b>6</b>
2.1	Gameplay Customization Overview . . . . .	6
2.1.1	Customized Elements . . . . .	7
	Space Adaptation . . . . .	7
	Mission / Task Adaptation . . . . .	7
	Game Mechanics Adaptation . . . . .	7
	Narrative Adaptation . . . . .	8
	Music / Sound Adaptation . . . . .	8
	Player Matching (Multiplayer) . . . . .	8
	Character Adaptation . . . . .	9
	Difficulty Scaling . . . . .	9
2.2	Game Balancing Approaches . . . . .	9
<b>3</b>	<b>Reinforcement Learning</b>	<b>13</b>
3.1	An Overview of Reinforcement Learning Methods . . . . .	14

3.1.1	Methods based on Value Function . . . . .	15
3.1.2	Methods based on Policy Search . . . . .	16
3.2	Reinforcement Learning with Neural Networks . . . . .	17
3.3	Proximal Policy Optimization . . . . .	18
<b>4</b>	<b>A Game Balancing Framework Based on ML-Agents Toolkit</b>	<b>22</b>
4.1	The Core Functionalities . . . . .	23
4.1.1	The Learning Environment Components . . . . .	23
4.2	Building Blocks of a RL Task in ML-Agents . . . . .	25
4.3	Training Process . . . . .	26
4.4	The Framework for Game Balancing . . . . .	27
<b>5</b>	<b>Balancing Games with Neural Reinforcement Learning and a Balancing Constant</b>	<b>30</b>
5.1	Reward Function . . . . .	30
5.1.1	Balancing Constant (BC) . . . . .	31
5.1.2	BC-Based Reward Function (BCR) . . . . .	31
5.2	BC-Based Balancing Metric (BCM) . . . . .	34
<b>6</b>	<b>A case Study of the NRL Balancing Framework in a Fighting Game</b>	<b>36</b>
6.1	Test Game Scenario . . . . .	36
6.1.1	Player Simulation . . . . .	36
6.1.2	Skill Study . . . . .	37
6.1.3	Observation Parameters (State) and Actions . . . . .	38
6.2	Experimental Results . . . . .	39
6.2.1	Experimental setting . . . . .	39
6.2.2	Results . . . . .	42
<b>7</b>	<b>Conclusions and future work</b>	<b>47</b>

Contents	xi
7.1 Limitations . . . . .	48
7.2 Future Works . . . . .	48
<b>References</b>	<b>50</b>

# Chapter 1

## Introduction

There are many definitions for the concept of Gameplay. In [2], we can find some of them, such as *a series of challenges contained in a simulation environment*, *a game-specific component related to the interactivity*, or *a formalized interaction that occurs when the players follow the roles of a game and experiment its system through it*. Gameplay portrays an essential role to the success of a game: Arguably, even if a game has the most realistic audiovisual features or the most fanciful story, but it contains challenges and elements unsuitable to the player's ability, this may affect the player experience, making the possibility of abandonment tangible.

Thus, during the development of a game, it is possible to observe some critical issues related to the gameplay refer to how the game interacts with the player, such as, how the game's challenges, objectives, rules, and so on, fit and pleases the individual styles of the player [37]. As it is impractical to build different versions of the same game to fit different styles of players, a question that arises is if the game can be composed of elements that provide it with some auto-regulation so that even the exact same game can offer different experiences to different players. With that in mind, many efforts have been made to yield adjustable games such that the player does not get frustrated by a challenging game, or bored when the game is too easy.

Thus, terms such as difficulty scaling [39], difficulty adjustment [13, 22], adaptive game [38], and dynamic game balancing [3] have continuously caught the attention of the industry and the academia to fulfill the gameplay by building balanced games. In some games, through the creation of a finite number of difficulty levels [28], to the players must select according to their beliefs of what fits him better. Nevertheless, this will possibly be an ineffective solution as the player may have intermediate skills between these classes or because he is constantly acquiring (or forgetting) skills, or, still, because the player may



have a wrong vision of its own abilities [22].

Notably, the topic of dynamic game balancing refers to the process of automatically changing behaviors, parameters, and/or elements of the scenarios in a video game in real-time, to make the game match the ability of a specific player [4]. For a long time, most of the effort to yield balanced games were based on creating specific heuristics or probabilistic methods that would face difficulties to be adapted to different games or different players with diverse abilities [34, 13].

Considering this drawback, in this dissertation, we propose a strategy where at least one of the main elements in a game that can influence the gameplay gets the ability to *learn how to act* while, at the same time, *leverages game balancing*, relying on Machine Learning (ML) techniques [24, 20]. Specifically, we are interested in building non-player characters (NPC) that can automatically be adjusting themselves to the player and the game, independently on the type of the game. By going in this direction, we expect that they try to match the human player performance, whatever the game is, and, consequently, contribute to the player to have more fun and to stick with the game.

In contrast, there are works that have already taken advantage of different ML methods to yield balanced games, focusing on providing a kind of modification related to different game elements. Generally speaking, by using ML techniques previous approaches have attempted to perform semi-automatic analysis of the playability of a game [37], generating new environments according to the player's progress[19], modeling the players to get an adequate adaptation of the opponent for them [8], among others attempts of improving video games by relying on ML. With a closer goal with ours, the last example reflect how other researchers have attempted to adapt the game for the user based on ML. Other works try to create AI systems capable of recognizing the type of player, followed by a comparison with groups of similar players and, finally, configuring the game according to the mapped characteristics [22]. However, it may be difficult to fit the player to a group of other players, as human behavior presents quite singular and unique traits [9].

Besides, other previous works are focusing on creating adaptable video-game agents that directly interact with the environment, taking advantage of Reinforcement Learning techniques [40] as we do in this dissertation. In this case, the agent receives the current state of the environment and a reward value that maps to a function leading to the policy it should follow. Previous work[4] treated this approach using the classical Q-learning algorithm [44], which learns a value function by evaluating the expected future return and then deduces the policy from there. However, value-based methods such as Q-learning

may face difficulties to deal with continuous space and may also converge slowly, as they are not directly optimizing the policy function [11].

## 1.1 Motivation

The various methods for balancing games previously presented in the literature, show some inadequate dependencies and limitations. First, the manual configuration of the game difficulty (usually used in the industry), allows the player to select a difficulty level and face it throughout the game without considering improvements or regression, as well as limiting it to a finite set of possibilities. Second, the heuristic and probabilistic methods perform a series of complex and difficult to understand the player behavior, which would require complex modifications depending on the game that it is trying to balance. Finally, regarding ML methods, most of them need good representations of the characteristics of the player; therefore, it is necessary to produce a suitable model for it. Besides, most of them still require classifying the players into finite groups according to their style.

Based on the limitations and shortcomings mentioned above, we were motivated to attack the problem of balancing games avoiding falling into them. In other words, our proposal is tailored towards (1) adjusting the games without forcing that a player must belong to a specific group of playing style, (2) avoiding the use of complex calculations for new games, (3) avoiding the design of models to represent the player, and, finally, (4) not focusing on balancing 2D games only. We want to take a step further and apply our work to more realistic 3D games, where continuous values represent the space and actions, as typically happens in current games.

## 1.2 Objectives

In this dissertation, we propose a learning method that induces the behaviour of an NPC-agent that does not need a representative model of the player to get adapted to him. Instead, the method proposed here directly induces the policy based on the way such an agent observes the game. To achieve that, we designed a reward function that operates on the basis of a game-balancing constant and it is coupled into the recently proposed Proximal-Policy-Optimization (PPO) [32] algorithm. This is a reinforcement learning method that directly optimizes the policy using gradient-based learning that deals directly with the environment setup to induce a proper approximation policy function. In

this way, we can also benefit from the remarkable results that other game-based problems have recently achieved [26, 35, 17]. We take advantage of the Unity ML-Agents Toolkit [15] which facilitates the development of graphical environments and provides several implementations of Reinforcement Learning algorithms like PPO, which uses neural networks (the universal approximators [12]) to try to obtain a function approximation of the policy.

Thus, in this dissertation we put together Neural Networks (NN), Reinforcement Learning (RL), and the Unity ML-Agents Toolkit, all of them enhanced with our proposed reward function, to build a framework for training agents that makes them to be balanced, according to the iterations with the player. Thus, the agents trained by our framework are built without any previous knowledge, and, during the training, they acquire enough information to know how to behave even when they face different player styles. Hence, to evaluate the feasibility of our approach, the learner agent is confronted with entities programmed with different styles such as an aggressive, a defensive, or a combination of both. The results that emerged from this evaluation demonstrate that our learner agent manages to balance the game by learning from diverse experiences while still trying new actions into the environment.

## 1.3 Methodology

In order to achieve our goals, it was necessary to start by investigating the principles of adaptation in games to understand what elements of the game can be involved when we talk about game balancing and its implications. Next, we focused on identifying the previous works more related to our goals, to investigate how they have treated the game balancing problem, their assumptions, and limitations. After that, we proceed to investigate the learning algorithms proposed in the context of reinforcement learning and the existing frameworks that implement them to avoid reprogramming them from scratch. We end up relying on the Unit ML Agents toolkit, which allows you to integrate both the game environment and the reinforcement learning algorithms. Finally, we implemented a framework to create the agents that aim at maintaining the game balanced, using as an essential component our reward function to point out to the agent how it should behave in terms of actions to stay in a certain range of skill difference to achieve the balance of the game.

## 1.4 Contributions

The main contributions of this dissertation are listed below:

- A framework for training game agents to get balanced abilities, with respect to the gamer's skill or capacities, relying on a policy-search Neural Reinforcement Learning method.
- A reward function that aims at inducing the agent to confront any player but at the same time teaching it to limit its behavior, so that it is possible to scale the game difficulty according to the skills and styles of the player.
- The definition of a Balancing Constant to represent the difficulty tolerance that a game can have depending on the goal and that can be used as a parameter for training the balanced agents.
- The Development of a game using the Unity tool that allows the learner agent to confront a single opponent to evaluate our framework.
- The implementation of different kinds of agents to train individually and/or jointly our balancing agent to observe how he behaves when facing other player styles.
- A publication in the 11th International Conference on Agents and Artificial Intelligence based on results of the present dissertation.

## 1.5 Structure of the dissertation

The remainder of this dissertation is structured as follows: Chapter 2 presents an overview of Game Customization and the elements that the developer can change for making the game adaptable. In this chapter we also include the related works, focusing on those that make use of AI techniques to achieve game balancing. Chapter 3 explains the Reinforcement Learning theory and the state-of-the-art of policy search algorithms. After that, Chapter 4 describes how the Unity ML-Agents toolkit implements RL algorithms. Chapter 5 presents our approach for game balancing based on Neural Reinforcement Learning, the reward function we have devised to handle the problem and how we benefit from the toolkit to pursue our goal. Chapter 6 presents a case study and its experimental results from our implementation and, finally, Chapter 7 draws some conclusions and indicates some future work directions.

# Chapter 2

## Literature Review

In this chapter, we introduce an overview of game customization techniques, highlighting the reasons that make researchers target on such methods and the main elements that can be customized to improve the player experience. We also discuss the customization of the game difficulty, also known as *Challenge Balancing*.

### 2.1 Gameplay Customization Overview

In the same way that in the real world different people has unique personalities traits and behavior, in the virtual game-world players also have different ways of acting. The way they act when interacting with the elements of a game can even be different from the way they operate in the real-world; furthermore, it may be the case that different games bring different stimulus to the players, making them behave in a distinct way depending on the video game. For these reasons, customizing a game to become adapted to the user can help the video-game industry to gain constancy in the use of their products since a customized match is one way to satisfy the players concerning their needs [41]. However, making a game attractive by customizing it according to the player's adherence is not a trivial task. Regarding these circumstances, researchers and developers are motivated to study this challenge for the following reasons [6]:

1. **Psychological foundation:** Psychological studies show that when there is an adjustment between a subject and an environment with which he interacts, he could be involved more in it since the personal content particularly directed to the user generate a strong emotional reaction in him.
2. **Effect on the player satisfaction:** The researchers advise that the success of the

users' experience is influenced by the pleasure, which is produced by the adjustment between the gamer features and the game.

3. **Contribution to the game development:** Regarding the developers, they need to yield a fun game to the final user, so they need to establish beforehand a profile of the players, what are the components and game style they prefer and what they are doing in some moment in the game. Consequently, developers should have a way of representing and instantiate such information.

### 2.1.1 Customized Elements

Customizing a game aims at creating a friendly and fun environment so that the players may feel a pleasant experience throughout the game. This problem can be attacked from different points of view because there is a vast number of game elements that we can customize to obtain a personalized videogame. Next, we describe some of them:

**Space Adaptation** Here, the focus is on adapting the game space or environment to respond to the players' behavior, allowing them to evolve over the game. The context of procedural generated games [33] studies this problem. Also, it requires to have a player model so that from it becomes possible to transform the game surroundings like a result of some recommendation. It is also necessary to observe the game for some time to recognize what kind of things need to be changed, followed by modifying the gameplay in a particular situation to produce new events. For example, if the user avoids passing for a specific place, the game can generate more areas with the same features (to hinder the progress) or avoids creating them (to help the player).

**Mission / Task Adaptation** It consists of creating new missions in some moment of the game during the unrolling of the history, according to the part of the game that gains more attention from the player. It is particularly challenging because by changing the story, elements in the environment may also have to be replaced, making it necessary to attend the previously discussed adaptation.

**Game Mechanics Adaptation** This adaptation intends to change the game mechanisms depending on what the player wants or does. For example, the players have been able to control the game-time and rewind the timeline like the BRAID game<sup>1</sup> or games

---

<sup>1</sup><https://www.gog.com/game/braid>

based on pre-scripted commands where the player can change some mechanism feature, like the shutter precision or indicate to the enemies to retreating from the player. Max Payne game<sup>2</sup> is an excellent example of this case. Thus, it can give place to a type of difficulty scaling based on the game mechanism.

**Narrative Adaptation** We can find this adaptation inside *interactive storytelling* [23], which consists in discovering the players' preferences and building the narrative content of the game in real time to entertain them, i.e., their choices indicate the direction of the story. A challenge associated with this type of adaptation consists of to maintain the coherence of the story. Consequently, many efforts exist to model the player, the environment and how they cope together in terms of actions and their consequences.

**Music / Sound Adaptation** The music is a prominent element of submersion in a game environment [31], so for many years in the game development process has been practiced attempt of adapting this component. For example, during the game execution, the player can experiment with a lot of continuous variation in the main sound following changes in places or situations. A form of expressing emotions seems to be the music, and thereupon, it can influence the mood and help to increase the sense of absorption. Even in its early period of studies, it already exists attempts to create games with adaptive, in real time, tracks according to the emotional state related to the game [46].

**Player Matching (Multiplayer)** Matching players with different skill levels can produce a not entertained experience for them because this can represent a more significant (minor) challenge than they need to face. In the case of online games, like Dota 2<sup>3</sup>, they are a favorite medium for the gamers sharing experience because they usually need more than a single player to work. In this way, the players search for a session to play with other players around the world. However, since they are at distinct levels, the game must have the capacity to connect players with the same level or balance the teams. Under those circumstances, the game must be able to measure the level of the players, using, for example, a heuristic or metrics created by the developer. Including good matching to confront them, like detailed by [42] related to League of Legends game<sup>4</sup>, which implements its matching based on player ranking, the number of games played and the number of players are waiting to play (1 - 5 players).

---

<sup>2</sup><http://www.rockstargames.com/maxpayne/index.html>

<sup>3</sup><http://es.dota2.com/>

<sup>4</sup><https://br.leagueoflegends.com>



**Character Adaptation** Refers to controlling the game characters by adapting them for consistently changing situations so that the player does not perceive any strange behavior in the environment. It is also called Adaptive Game AI because usually the characters are modeled as intelligent agents composed of an AI technique, trying to accomplish a reliable and realistic behavior. In practice, the videogames that make use of this type of adaptation, trying to use an "AI" (computer controlled) to create an illusion for the user that the agent is intelligent. He can make some mistakes, or the agents are cooperating, [18] like the Half-Life game<sup>5</sup> does. Furthermore, by using some approach based on Machine Learning mainly added with online learning, it is possible to create a game AI that learns based on practice experience, obtained during the interaction with the players and, as a result, he can learn to adapt to the player and to changing situations.

**Difficulty Scaling** This kind of customization is also called *Challenge Balancing* and refers to the automatic adjusting of the game challenges based on the player skills. However, most of the games prefer to leave this work to the players, giving them the opportunity to choose one of a finite number of levels (e.g., easy, medium and hard) that they consider more adequate to challenge them. But, in this situation, some players can consider difficult some characteristics that other players can find comfortable, becoming hard work to obtain a proper level definition. As a result, many researchers aim at developing techniques to balance games by creating characters that fit well the players and adapt the environments to the player.

Based on this context, this dissertation intends to study the case of balancing a game with a single agent concerning the skills of the player. First, however, it is necessary to understand how the literature have dealt with the balancing challenge to consequently be able to highlight the difference of our approach with respect to them.

## 2.2 Game Balancing Approaches

Game balancing refers to the ability of a game to modify itself or adjust its level of difficulty according to the level of knowledge, style or personality of the user, allowing a better experience along the game. The principal motivation behind it is to avoid that the player gets stressed or bored when facing very difficult or easy situations when playing [10]. In [3] the authors have pointed out that satisfaction and balance are mutually related, by making a series of questionnaires to the players after experiencing a fighting game with an

<sup>5</sup><https://store.steampowered.com/app/70/HalfLife/>

agent that controls their behavior based on player ability. Accordingly, the results of the surveys have pointed out that the players of the experiment consider that the opponent is more enjoyable when it includes some intelligent or appear to be challenging.

To tackle the challenges of game balancing it is necessary to take into account, firstly, which characteristics the developers are considering to balance to provide an enjoyable experience to the final user. Because of that, some previous works have focused on reaching equilibrium by trying to add new content such as environments, elements, obstacles, considering both the abilities of the players and the current situation they are in the game to facilitate their progress. Others works have sought to create intelligent agents (NPCs) capable of facing their enemy (the player), but without hindering by a significant difference the possible success of users, depending on the difficulty tolerance that the developers can give to the game[6].

Regarding the first group mentioned in the previous paragraph, an example is [7], which tries to modify the environment (the Mario Boss game space) towards achieving game balancing. To that, they elicit a representation of the traits of the player and his behavior within the game to perform two offline and one online training for obtaining a policy representing the personality of the player. Another example is [13], which uses a probabilistic method to determine the game characteristics that need to be changed to help with the player' progress. This framework can reason about the game uncertainty and determine what elements it needs to adjust. The previous work presents its framework based on Valve's Half-Life game engine, and it uses probabilistic equations that could require more complicated distributions to a different game to that they present. In our work, the adaption of the framework can be easily made just considering the actions, observations, and the component that we can balance without any complicated computations.

In contrast, to create opponents or game partners adaptable according to the player personality, the work presented in [22] tried to develop such an automatic adjustment by first identifying the kind of the players to confront them with an adequate type of enemy (the agent). They list several characteristics of users and classify them into a discrete number of difficulty sets (Easy, Medium and Difficult). The goal was to classify the new players inside one of these sets to determine what kind of enemy these players could face so that their opponents are easier or more challenging to confront. That work sees the identification of the type of player as a fundamental issue to have balanced games, but we detach two main problems that may not lead to the desired effect: (1) the representation

of the player must be as faithful as possible, which is not easy to achieve. (2) The player must be classified into a discrete number of groups when usually a human-being can belong to more than one class in different points of the game, or according to the moment, he is playing, or even to evolve during the game. Our work, on the other hand, aims at avoiding creating abstractions or representations of the player and also avoiding to define the number of classes to fit the player, as this is sometimes difficult to recognize beforehand. Instead, we focus on representing the current state of the game to determine the decision-making policy of the agent against his opponent.

Meanwhile, [34] uses a heuristic function to determine at each instant the player performance when facing his enemies during the game. This *evaluation function*, as they called, helps to indicate if the game represents a challenge to the user and, consequently, if the game should match the player with another AI enemy – they developed three types of enemies with different abilities– with more or less difficulty. Again, the balancing result depends on a finite number of classes (Easy, Regular and Hard) and the user must fit in one of them. Our work intends to not rely on the number of styles; instead, we want to create an agent that is responsible for making the game to be balanced based on the experience it acquires when facing distinct players.

Regarding the use of Reinforcement Learning (RL), which is the Machine Learning technique mostly used to tackle game-based challenges, [4] adopted a classical strategy (a tabular implementation of Q-learning ) in two steps: (1) teaching the apprentice (a virtual agent) to play like a user and (2) balance the game based on a challenge function that encodes, for example, an action selection mechanism. A following-up work[3] demonstrated the relationship between the satisfaction and the difficulty to show that the more skilled or difficult the agent is the more enjoyable the game is to the user.

Thus, we intend in this work to study the balance of the game concerning the adaptation of a character (the opponent) in the game. However, different from these previous work, we do not intend to first teach the agent to play and only later to make its actions adjustable to achieve game balancing. Instead, we focus on devising an agent that learns how to play *and*, at the same time, it learns the policy to balance the game by relying on a suitable reward function.

We believe that we can create a simple framework able to train agents with the ability to yield a balancing game. It seems a more straightforward way than creating complex models of the player, evaluating complex heuristics or probabilistic methods to determine or predict the game situation, or even trying to limit the player capacity in a finite number

---

of classes when he can evolve by far from that. Different from the previous work, our work aims to build the framework only with simple configurations like selecting the actions from the environment observations and the component to balance.

# Chapter 3

## Reinforcement Learning

In this work, we use Reinforcement Learning to train our agent to balance the game, because, in the beginning, the **agent** will not know how to **act** to balance based on its **observation** of the **environment**, but, during the process, it will be acquiring knowledge based on past experiences and if the agent achieves it will be receiving some **reward** to do better and better or a punishment another case. Consequently, in this chapter, we will expose an overall about Reinforcement Learning, its goal, methods to find an optimal solution (policy) for a problem, advantages, and disadvantages to use some method, and finally, how we can take advantage to use of two methods as a hybrid method, particularly, Proximal Policy Optimization (PPO).

Reinforcement Learning (RL) is a subarea of Machine Learning that aims at teaching an artificial agent to make decisions when facing a certain situation through trial and error [40]. When learning, the agent receives an stimuli in the form of a reward so that it can evaluate if the chosen action was good for the decision making process or not. The main purpose is to find an optimal mapping of a situation to an action trying to maximize the accumulated reward.

An RL task can be formally described as a Markov Decision Process (MDP) [45, 5], which is defined as a 5-tuple  $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$  composed of the following components:

- A finite set of states  $\mathcal{S}$ .
- A set of actions  $\mathcal{A}$ .
- A reward function  $\mathcal{R} : \mathcal{S} \Rightarrow \mathbb{R}$ .
- A transition function  $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \Rightarrow [0, 1]$ .

- A discount factor  $\gamma \in [0, 1]$ .

At each time step  $t$  of the decision making process, the environment is represented by a state  $s_t \in \mathcal{S}$ , and the agent may choose an action  $a_t \in \mathcal{A}$  that is available in the state  $s_t$ . The transition function  $\mathcal{T}(s_t, a_t, s'_{t+1})$  defines the probability of reaching the state  $s'$  in the instant  $t + 1$ , given that the agent performs the action  $a$  in the instant  $t$  when it was in the state  $s$ . After going from the state  $s_t$  to the state  $s'_{t+1}$  by performing  $a_t$ , the agent receives an immediate numerical reward  $R(s'_{t+1})$  and the discount factor indicates how future rewards matter compared to the current one. Then, by interacting with the environment, the agent can accumulate rewards with some penalization in the future (Equation 3.1).

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots = \sum_{t=0}^{\infty} \gamma^t R_t \quad (3.1)$$

In RL, a policy  $\pi(s) \rightarrow a$  represents the behavior that the agent should follow, by providing the action  $a$  that he should take when it is in a specific state  $s$ . To induce a policy, an RL algorithm may consider a *value function* (equation 3.2), which is the total reward that the agent can expect to accumulate in the future when it starts in the state  $s$  and follows the policy  $\pi$ .

$$V^\pi(s) = \mathbb{E}^\pi \left[ \sum_{h=0}^{\infty} \gamma^h R_h(s_h, a_h) \mid s_t = s \right] \quad (3.2)$$

The purpose of an RL method is to find a *policy*  $\pi^*$  (equation 3.3) that maps from states to actions to maximize the accumulated reward  $R$  (obtained from the environment in every rollout of policy) of all states. In other words,  $\pi^*$  will be the one that maximizes the value function.

$$\pi^*(s) = \arg \max_{\pi} (V^\pi(s)) \quad (3.3)$$

## 3.1 An Overview of Reinforcement Learning Methods

In this section, we describe the two main methods for inducing a policy within an RL task.

### 3.1.1 Methods based on Value Function

These methods aim at maximizing the value function  $V^*(s) = \max_{\pi} V^{\pi}(s)$ , and, although the agent usually does not know in which state it will end, it is possible to influence this by choosing an appropriate action. For that, it is used a quality function  $Q^{\pi}(s, a)$  (equation 3.4), which indicates the total reward expected for the agent when it starts in the state  $s$  and takes the action  $a$ .

$$Q^{\pi}(s_t, a_t) = \mathbb{E}^{\pi} \left[ \sum_{h=0}^{\infty} \gamma^h R_h(s_h, a_h) | s_t, a_t \right] \quad (3.4)$$

Consequently, an optimal value for the  $Q^{\pi}(s_t, a_t)$  function will tell us how good it was for the decision making to choose an action  $a$  when it was in the state  $s$  in the instant  $t$ . In this way, it is possible to say that the maximum total reward expected when starting at  $s$  is equal to the maximum value of  $Q^{\pi}(s_t, a_t)$  when we choose the action  $a$  over all the others. Therefore, an optimal policy  $\pi^*(s_t)$  can be obtained using  $Q^*(s_t, a_t)$ , as exhibited in equation 3.5 since it determines which action will maximize the given reward.

$$\pi^*(s) = \arg \max_a Q^{\pi}(s_t, a_t) \quad (3.5)$$

Q-learning [44] is an algorithm that applies these principles, where at each iteration it aims to approximate to an optimal  $Q^*(s_t, a_t)$  function value, under the idea that the maximum future reward is the reward received by the current state  $s_t$  plus the maximum future reward received in the next state  $s_{t+1}$  (equation 3.6).

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+s} + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)) \quad (3.6)$$

For continuous spaces, the q-value may be included in a table updated at each iteration to find the optimal policy. However, for the problems situated into continuous spaces, we should use a function approximation method to try to find a lower dimensional representation of the optimal value function. Another possibility is to discretize the continuous action space, after which the optimization over the action space becomes a matter of enumeration. This approach undermines the ability to use continuous actions.

In this work, we choose not to follow a purely value-based method, as it only indirectly induces the policy through the value function. Instead, we investigate a technique that directly influences the policy, but the process is combined with elements from value-based



methods, yielding a hybrid approach.

### 3.1.2 Methods based on Policy Search

The policy gradient methods target at directly modeling and optimizing the policy. The policy is usually modeled with a parameterized function with respect to  $\theta$ ,  $\pi_\theta(a|s)$ . Then, finding the can be seen as an optimization problem where the objective is to find the best parameters  $\theta$  that maximizes a score function,  $J(\theta)$ :

$$J(\theta) = \mathbb{E}^{\pi_\theta}[\sum \gamma r] \quad (3.7)$$

A standard approach in Machine Learning to act in maximization (or minimization) problems is employing an ascending (or descending) gradient method, as represented in equation 3.8.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta) \quad (3.8)$$

Here comes the challenge, how do we find the gradient of the objective above which contains the expectation. Integrals are always prohibitive in a computational setting. We need to find a way around them. The first step is to reformulate the gradient starting with the expansion of expectation (with a slight abuse of notation).

$$\nabla \mathbb{E}^\pi[r(\tau)] = \nabla \int \pi(\tau) r(\tau) d\tau \quad (3.9)$$

$$= \int \nabla \pi(\tau) r(\tau) d\tau \quad (3.10)$$

$$= \int \pi(\tau) \nabla \log \pi(\tau) r(\tau) d\tau \quad (3.11)$$

$$\nabla \mathbb{E}^\pi[r(\tau)] = \mathbb{E}[r(\tau) \nabla \log \pi(\tau)] \quad (3.12)$$

Doing 3.12, it is possible to update the policy at each iteration. However, the only guarantee we have is to reach a local maximum. A significant advantage of this kind of method over the value-based one is that they allow the policy to generate actions in the entire continuous action space, and by using gradient descent methods they have a strong property of convergence. In contrast with this, there is a significant variance of the estimated gradient and the missing of knowledge about past estimations.

## 3.2 Reinforcement Learning with Neural Networks

The goal of RL is creating an agent that may learn by interacting with the environment and based on the experience it will optimize its manner to act. However, the environment may be stochastic, and the agent may only partially observe the current state, the observations may be high-dimensional, and so on. The conventional RL approaches are limited to domains with low-dimensional state spaces or handcrafted features, as, for example, when we use the traditional Q-learning we must fill large tables, which would require a vast amount of observed transitions. For these reasons, combining Neural Network (NN) [21] methods with RL aim at embracing both the representation power of the universal function approximation of neural networks and the generalization ability from RL because in practice state spaces might become very large or even infinite so that a table based representation is not possible.

Thus, Neural RL (NRL) is a particular type of RL, with neural networks approximating the value function  $\hat{v}(s; \theta)$ , the policy  $\pi(a|s; \theta)$ , the transition model, or even the reward function, where  $\theta$  are the parameters of the NN. NRL has had good results in complicated tasks without high prior knowledge because it can learn different levels of abstractions from data. Amongst recent works using NRL, we can find the attempt of create an algorithm to teach the agent to play seven popular ATARI games [26] by using game frames and relying on deep neural networks, demonstrating that is possible to train agents based on raw, high-dimensional observations and a reward signal. When the neural network underlying the RL method is a deep network, we have the area of Deep Reinforcement Learning (DRL). DRL has been applied to real-world problems such as robotic and self-driving cars entirely in a virtual world that can adapt to the real environment [30]. Nevertheless, the use of DRL in video games has many approaches that have evolved along the time, to try to solve problems in more complex environments. For example in the literature, we can find [36], where the agent has to defeat the real Go players. Or more recently, the OpenAI<sup>1</sup> created an AI which beats the professional players of Dota 2 and it trains using a version of Proximal Policy Optimization. This is the same method employed in this dissertation and we bring more details about it in the next section.

---

<sup>1</sup><https://openai.com/the-international/>

### 3.3 Proximal Policy Optimization

The framework designed in this dissertation includes the hybrid value/policy-based method Proximal Policy Optimization (PPO). Most of the hybrid techniques that take advantage of the characteristics of both the methods based on optimizing the Value Function and the Policy Search in a framework are within the area of Actor-Critic Methods [16]. In this architecture, the Actor implements the policy  $\pi(s, a, \theta)$ , controlling how the agent behaves which in turn is monitored by the Critic using the Value function  $\hat{q}(s, a, w)$ . An abstract view of this framework is exhibited in the Figure 3.1. As we have two models (the Actor and the Critic) both of them must be trained, which means that we have two sets of weights ( $\theta$ , for the Actor and  $w$  for the Critic) that must be optimized separately (equations 3.13 and 3.14).

$$\nabla \theta = \alpha \nabla_{\theta} (\log \pi_{\theta}(s, a)) \hat{q}_w(s, a) \quad (3.13)$$

$$\nabla w = \beta (R(s, a) + \gamma \hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t)) \nabla_w \hat{q}_w(s_t, a_t) \quad (3.14)$$

The value-based methods have high variability and, to reduce this, we can use the advantage function instead of the value function. The advantage function indicates the improvement compared to the average the action taken at a particular state. In other words, this function calculates the extra reward we get if we take some action. The A2C critic method relies on the gradient estimator coupled with an advantage function  $\hat{A}_t$  (equation 3.15).

$$\hat{A}_t(s, a) = Q(s, a) - V(s) \quad (3.15)$$

Proximal Policy Optimization is a policy search-based method primarily inspired by the Advantage Actor-Critic (A2C) framework [25]. The central idea of the algorithm is avoiding to have a large policy update and for that, it defines a ratio (equation 3.16) between the probability of an action under the current policy divided by the probability of the action under the previous policy.

$$r_t(\theta) = \frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \quad (3.16)$$

Then, we can determine if an action is more (or less) probable in the current policy than the older one. Consequently, it is possible to define a new objective function

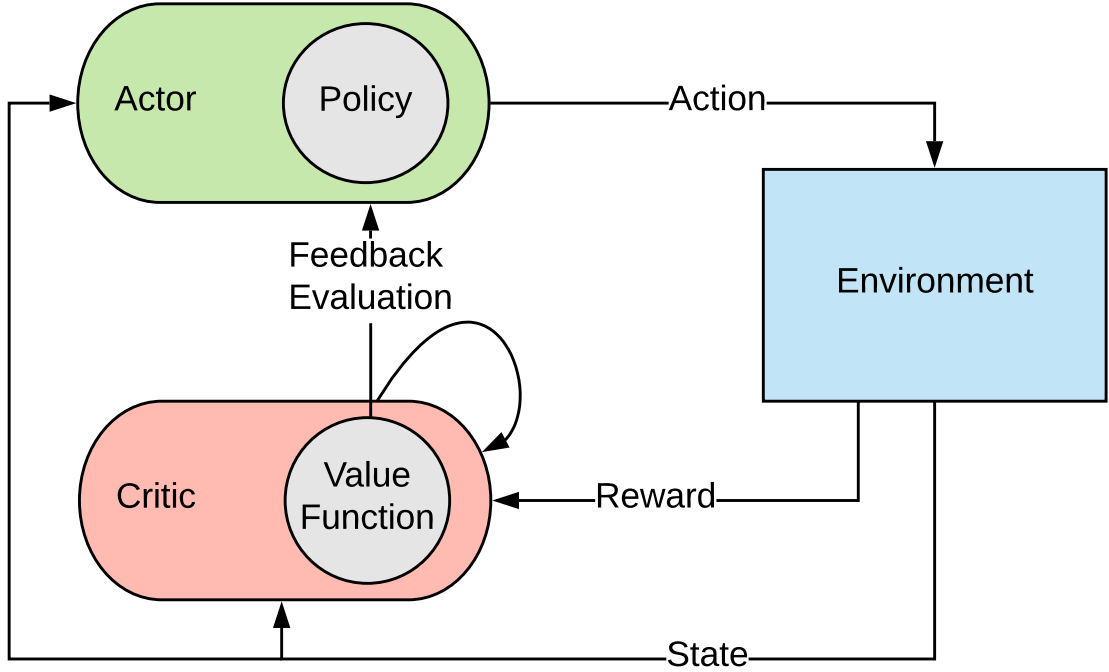


Figure 3.1: Actor-Critic RL Method

(equation 3.17), and however, if the action taken is much more probable in the current policy, this would lead to a significant policy gradient step and, hence, an excessive policy update.

$$J^{CPI}(\theta) = \hat{\mathbb{E}}_T[r_t(\theta)\hat{A}(s, a)] \quad (3.17)$$

Therefore, it is necessary to limit this objective function by penalizing changes that lead to a ratio that will go far away from 1. That means that the policy will not have a very large update because the new policy cannot be too different from the older one. The PPO clips the probability ratio directly in the objective function with its clipped surrogate objective function (equation 3.18).

$$J^{CLIP}(\theta) = \mathbb{E}[\min(r(\theta)\hat{A}_t(s, a), \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t(s, a))] \quad (3.18)$$

The function  $\text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)$  limits the ratio within  $[1 - \epsilon, 1 + \epsilon]$  with  $\epsilon$  as a hyperparameter. The objective function of PPO takes the minimum between the original value and the clipped value and, therefore, this loses the interest in increasing the policy update to extremes, for better rewards. The clipped surrogate objective restricts the range that the new policy can vary from the old one, causing the removal of the incentive

for the probability ratio to move outside of the interval. As the clip has the effect of a gradient, if the ratio is more than  $1 + \epsilon$  or less than  $1 - \epsilon$  the gradient will be equal to 0. Thus, both of these clipping regions prevent the algorithm from getting too greedy and trying to update too much at once and updating outside of the region where this sample offers a good approximation.

By using the PPO algorithm on the network architecture with shared parameters for the policy and the value functions (the actor and the critic respectively), apart from the clipped reward, the objective function is increased with an error term on the value estimation  $c_1(V_\theta(s) - V_{target})^2$  and an entropy term  $c_2 H(s, \pi_\theta(.))$  to encourage enough exploration with  $c_1$  and  $c_2$  as hyperparameters. Thus, the final Clipped Surrogate Objective Loss is:

$$J^{CLIP'}(\theta) = \mathbb{E}[J^{CLIP}(\theta) - c_1(V_\theta(s) - V_{target})^2 + c_2 H(s, \pi_\theta(.))] \quad (3.19)$$

The algorithm 1 reproduces the PPO algorithm extracted from [32], and this is the algorithm we used in our proposal for balancing the agent during a game. PPO introduced another innovation which is training  $N$  (parallel) actors collect during  $T$  timesteps of data by running  $K$  epochs of gradient descent over samples of mini-batches. Thus, we synchronously update the global network, *i.e.*, the algorithm waits for each actor to finish their segment of experience before updating the global parameters. Then, we restart a new segment of experience with all the parallel actors possessing the same new parameters.

---

**Algorithm 1:** PPO, Actor-Critic Style as presented in [32]

---

```

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

```

---

We choose PPO because we model the environment as a continuous space, since 3D games require continuous state spaces defined by means of continuous variables such as position, velocity, torque, etc. It has been previously discussed that the policy search-based methods are more appropriate for this kind of environment and PPO goes a step further by including a critic based on a value function. Besides that, PPO has succeeded in

a number of other game-based tasks (ATARI games) such as Alien<sup>2</sup>, Atlantis<sup>3</sup>, Kangaroo<sup>4</sup> and other 44 more [32].

---

<sup>2</sup>[https://www.retrogames.cz/play\\_251-Atari2600.php](https://www.retrogames.cz/play_251-Atari2600.php)

<sup>3</sup>[https://www.retrogames.cz/play\\_082-Atari2600.php](https://www.retrogames.cz/play_082-Atari2600.php)

<sup>4</sup>[https://www.retrogames.cz/play\\_195-Atari2600.php](https://www.retrogames.cz/play_195-Atari2600.php)

## Chapter 4

# A Game Balancing Framework Based on ML-Agents Toolkit

In this chapter, we will describe the plugin functionalities, how every part of them works jointly, how the toolkit organizes its components to create a learning environment in the scene, and finally, it will be explained how concepts as observation, action, and reward are integrated into the plugin to the training. Then, our framework will be introduced, describing step by step the process for training agents to game balancing and how the result of each step is used as an input to the next step, giving a model to balance the same game as a final result.

Unity<sup>1</sup> is a cross-platform (it has compilation support of different types of game platforms like PC, Web, Mobile Devices and so on) game engine and has also become a suitable tool to create interactive simulations and virtual environments. The Unity ML-Agents Toolkit [15] is a free-source component that allows the development of virtual environments by using the Unity Editor coped with Machine Learning algorithms. Developers can use this plugin in many situations like automatic testing of games or to command the NPC behavior in a game. To those, it includes a Python API for training intelligent agents, using state-of-the-art Machine Learning algorithms and the Deep Learning framework TensorFlow [1]. One of the provided implementations in the ML-Agents Toolkit is the Proximal Policy Optimization Algorithm for Reinforcement Learning, which we used in this dissertation.

To create the virtual agents, the developers need to understand how the structure and elements of the toolkit work. For training, it contains a set of components necessary to get an organized workspace and the items based on RL concepts. Before presenting the

---

<sup>1</sup><https://unity3d.com/>



framework we have developed as part of this dissertation, we detail the plugin parts and how it works.

## 4.1 The Core Functionalities

The toolkit contains a set of core functionalities to create an agent so that it can interact with an environment. Based on the Unity ML-Agents Toolkit Documentation<sup>2</sup> and [15], we describe them as following:

1. **The ML-Agents SDK:** Every functionality to create new environments with the help of the Unity Editor and C# Script belongs to this component, which is composed of two high-level elements:
  - **Learning Environment:** It is not more than a Unity Scene with the SDK imported. Here we can find all the necessary elements to prepare the agent for training, as shown in figure 4.1, it will be better explained in Section 4.1.1.
  - **The External Communicator:** It is part of the Learning Environment, and allows the Unity components to communicate with the Python API.
2. **The Python API:** It allows the developers to interact with the environment created by the ML-Agents SDK, and it is composed of all the training algorithms that induce the policy of an agent. As this is not a sub-component of the Unity, this API and Unity communicate with each other using a gRPC communication protocol.

### 4.1.1 The Learning Environment Components

As the Learning Environment holds the necessary elements to develop a game, the plugin should be organized according to the three following components, to get a workspace easy to understand during the training process :

1. **The Agent:** This component is used to indicate that an entity within the scene is an RL Agent, with the ability of aggregating observations from the environment, performing actions, and receiving a reward or a punishment. The virtual physical agent is linked to it by a Unity GameObject. The agent is associated with a C#

---

<sup>2</sup><https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Readme.md>

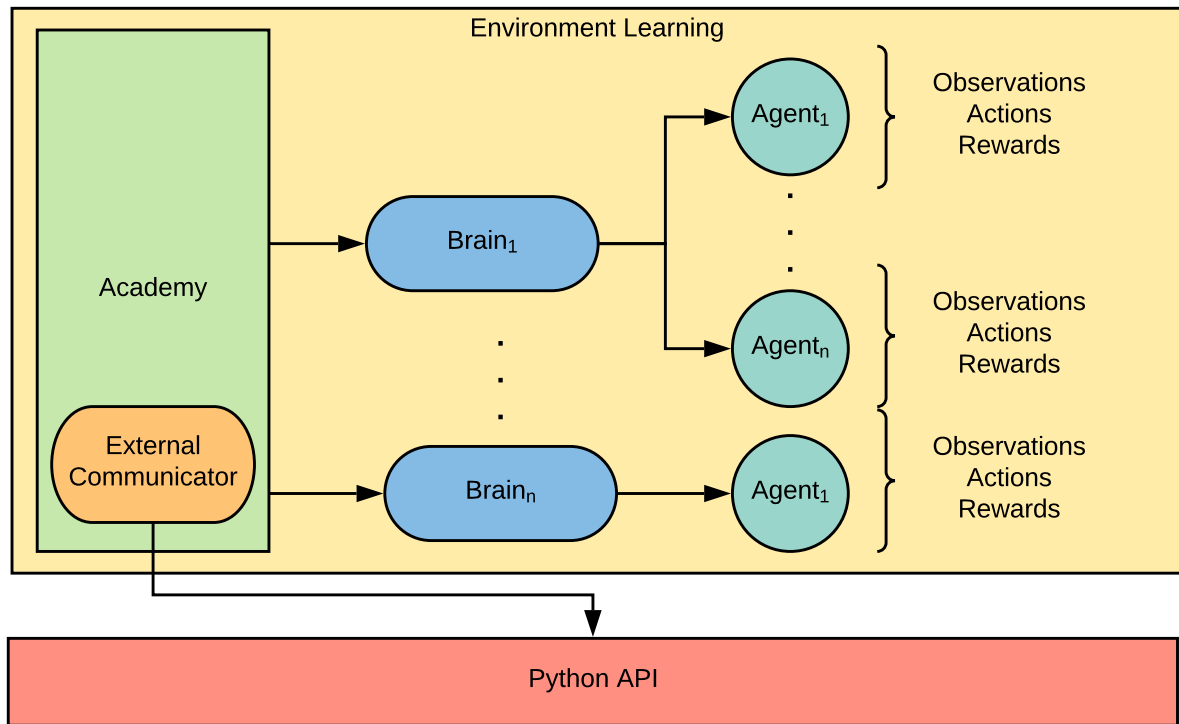


Figure 4.1: Overview of the of the ML-Agents Toolkit Structure

Script, extending from an Agent Class. This class holds every method based on RL to train the learner as we explain later in Section 4.2.

2. **The Brain:** This component is responsible for receiving the observations gathered from the environment by the agent and the own rewards. It uses that information to decide which action the agent should perform as a result of the policy learned so far. Depending on the situation, we can use a brain as a Player (to execute the actions that the player performs through inputs), as a Heuristic (containing a script that tells the agent what to do), as an Internal component (to use a trained model), and/or as an External component (to interact with the Python API for training). Every Agent can have precisely one Brain, but different agents can use the same brain at the same time as shown in Figure 4.1. The Brain is an abstract class that holds the logic for an agent making a decision.
3. **The Academy:** This component is responsible for coordinating the environment simulation and allows changing some parameters involved in the training, such as the speed to run the game loop in the process or the maximum number of steps. It contains the external communicator that connects the Python API with the learning environment.

## 4.2 Building Blocks of a RL Task in ML-Agents

With the learning environment established, it is possible to start creating different RL tasks. This job is possible with the definition of actions, observations, and rewards, defined inside of the Agent component. They are methods outlined in this component script, and the developers can overwrite them according to the target that they intend to reach or the environment of study. As an example, consider the representation of actions that are defined depending on the movements of the agent in the game. Following, we describe these RL building blocks, responsible for determining a task:

1. **The Observation:** It is the numerical information provided by the Agent component about the environment in every step. It is passed to the Brain component so that it decides which action the agent should execute based on the information.

Inside of the overwrite *CollectObservations()* method, which belongs to the Agent class, it is possible to collect, one by one, all the required information to be sent as the observation vector (by calling the *AddVectorObs()* function). After passing the numerical inspection as a parameter and finishing the calling of the *CollectObservations()* method, all of this information is sent to the Brain component like a vector, jointly. The observation can be either numerical (floating-point number vector) or visual (images taken by the cameras linked to the agent, representing what the agent can see).

2. **The Action:** Represented by the *AgentAction()* method, it is possible to overwrite it and indicate what actions the agent can do based on the vector inputs that the player or the Internal/External Brain are going to produce. Within this function, it is possible to evaluate the result of executing an action by the learner, so we can send some reward to the Brain to notify it about the successful/weak performance that will lead it to good learning.

The set of inputs can be represented as a vector of discrete or continuous values, depending on the type of action the agent can do. For example, if the agent performs simple steps, like moving in some directions (up, down, right and left), we only need to save continuous values to represent it. For example, 1 is equal to up, or 2 is equal to down and so on. On the other hand, if we need to execute a more complex action, we can use continuous values. For example, when we want to represent the acceleration, it is necessary to know for how long time a key was pressed to express the relative acceleration the agent performed.

3. **The Reward:** This method is used to send a numerical signal, as a parameter, indicating if the agent is doing well or not. The numerical value can be positive or negative (reward or punishment respectively) according to the meaning that the developers want to make understandable to the agent as a result of his actions. The value is sending to the Brain component by using the *AddReward()* function, and since we do not need to redefine it, we can only call it to indicate the direction of the training. By using this function, we send the numerical value that resulted from our function reward, explained in Section 5.1.2.

Besides, it is essential to mention the *Done()* method, which helps to indicate to the Brain component that it needs to reset the process cause to the job is done or that it is necessary to start again in another point to reach the target.

## 4.3 Training Process

By using an external Python training process, the toolkit carries out the training, and this external process communicates with the Academy component to create a block of agent experiences as a training set, which, in turn, is used to optimize the policy with neural networks. Together with the result of the training, the toolkit returns a Tensorflow data graph with mathematical operations and weights selected during the training (all this as a model file), which we can use as an Internal Brain component within the agent to map observations to actions.

In order, to train the agent, a set of examples that use different RL algorithms implemented in the toolkit are available for using or extending, and to deploy new environments from them. Concerning the RL algorithms currently provided in the toolkit, there is the Proximal Policy Optimization (PPO) [32], which is implemented in TensorFlow and communicates with the running Unity application over a socket to run as a separate Python process. Also, to start the training process with the plugin, it is necessary to call the *mlagents-learn* command and define every hyperparameter that we consider appropriate into the *trainer\_config.yaml* file, like the *batch\_size*, learning rate, number of epochs and so on. The toolkit documentation shows some of the typical range of them.

Moreover, this plugin gives us the possibility of analyzing the results of the training by using the Tensorboard, based on the configuration to the training, for example, the accumulated reward in training and the loss entropy.

## 4.4 The Framework for Game Balancing

In this work, we propose a framework for training artificial game agents addressing game balancing as illustrated in Figure 4.2. Each step of the framework yields a unique element that is used in the following step, as follows:

1. **Game Development:** This step consists of a regular game creation process by using a game engine (in our case, Unity). Here, we define all the actions of the agent and their inputs to be executed inside of the *ActionAgent()* function, extended from the Agent class, when we import the SDK in the scene. We have a scene with all the available components to be called, for our training process.s
2. **ML-Agents SDK:** This part is one of the most critical elements of our framework because here we define the necessary components to train the agent. Thus, as a result of the previous step, we have a game scene, but still without actions performing in it, so, only after the SDK is imported that the scene becomes an environment learning. The main task during this step is to assign a new script to the agent, extended from the Agent class of the SDK. After that, it is necessary to indicate which values will be used as observations, which actions the agent will perform and the rewards that it receives (as described in Section 4.2). From that, we try to achieve the following goals:
  - (a) **Learning how to play:** We assume that the agent still does not to know how to act in that particular game environment when the training begins. That means that the learner agent should acquire the knowledge of how to be part of the game by trying every possible action, like an average player until it reaches the desired skills.
  - (b) **Learning to keep up with the player to achieve game balancing:** The agent does not only need to learn to play, but it also needs to learn how to be changeable and well balanced for the player.

The **State of the Balancing** informs that the agent has achieved the same or a little better ability level compared to the player. It will aim to be more skilled or not, depending on the goal and the level of difficulty that the developer wishes to imprint in the game. In other words, the developer can aim to get a balanced game (the opponent and the player have almost equal performance concerning their skill - equation 4.1- or having an unbalanced situation - equation 4.2. For example, we

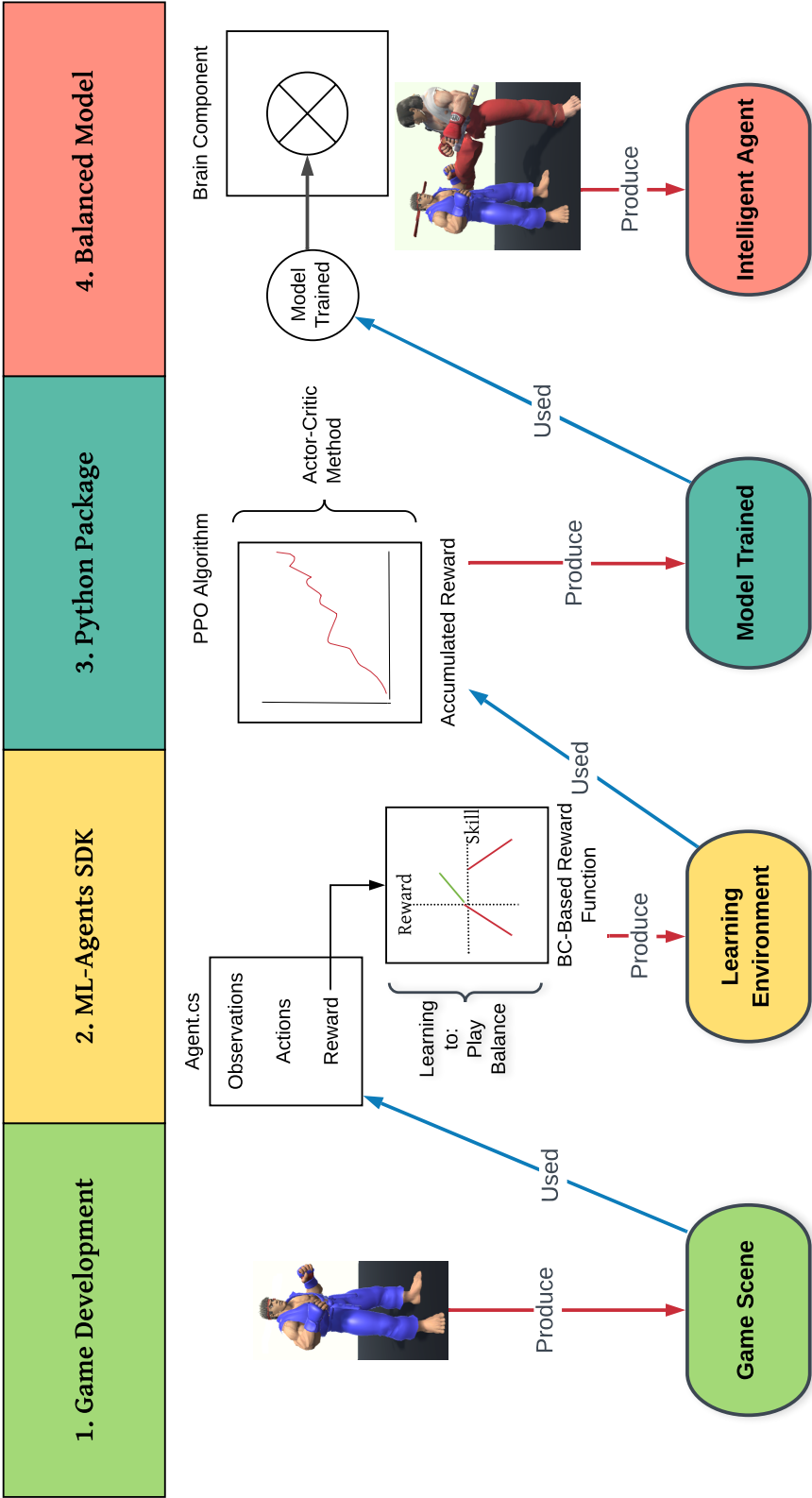


Figure 4.2: Framework for Game Balancing

can measure the skill of how the agent and the player can produce damage to one another measured by the health of the avatar. Thus, if they have relatively the same health score, the game is considered balanced and unbalanced otherwise.

$$Skill_{Agent} \cong Skill_{Player} \quad (4.1)$$

$$Skill_{Agent} + Difficulty \cong Skill_{Player} \quad (4.2)$$

To take care of both of the aforementioned situations, we use a reward function based on a flexible balancing constant. This contribution will be explained in Section 5.1.

3. **Python Package:** Here, we use the prepared environment learning for training the agent by using the algorithms implemented in Tensorflow and our framework, composed of a Reinforcement Learning task. We use the PPO algorithm implemented in the Unity ML-agents toolkit to prepare the agent to follow our objectives. Then, the result of this step is a model file with mathematical operations and optimal weights that is ready to be used as an Internal Brain by the agent, helping it to take balanced decisions learned in training.
4. **Balanced Model:** This final step consists of using the resulting balanced model to be incorporated inside the Brain component agent, to try to balance the game. It is essential to indicate that the model only can be used by agents that have the same configurations as the same actions, observations and so on. Another possibility consists of using this model as a starting point to train other agents, starting from the equal weights but not being necessary to begin the training process from the beginning.

In this chapter, we presented details about the ML-Agents Toolkit, its main functionalities, and composition. In addition, it was introduced our framework developed to target balanced games, which is based by using the plugin. In the next chapter, we present the main contribution of this dissertation that works jointly with the framework as its core.

## Chapter 5

# Balancing Games with Neural Reinforcement Learning and a Balancing Constant

The previous chapter described the components of the Unity toolkit to create intelligent agents and how we apply them in our work to develop a framework for videogame balancing. Every step of our framework produces an element that will be used in the next step. Thus, the first step (Game Development) outputs a game scene that the second step (ML-Agent SDK) will transform into a Learning Environment where we incorporate our main contribution for achieving the balancing: a reward function based on a balancing constant. Then, the third step (Python API) uses the environment to run the training process by using the PPO algorithm and, finally, the fourth step (Model Trained) incorporate the resultant model to give balancing knowledge to the agent.

Thus, in this chapter, we present the core of the balancing game training process, our reward function, how we understand what a state of balancing is, and how we managed to teach this for the agent. Finally, we explain the metric to determine if we genuinely achieved the desired balance.

### 5.1 Reward Function

The reward function is a key component of a RL task because it is the tool used to stimulate or penalize the learner, according to the result of its actions in the environment. With this function, we can suggest possible directions to the agent so that it can know how far (or near) he is from the desired state of the balancing and in case it is already within it, how it should behave to maintain close to it. In this sense, this is the component we choose to tackle in a proper way for aiming at the balancing. We design it around a



balancing constant, that it is used to point out the inferior and superior limits of the skill to the learner. Next, we give more details about it.

### 5.1.1 Balancing Constant (BC)

To achieve game balancing is essential to notify the agent when it is in a state of balancing. In this work, we focus on skill-based balancing (equations 4.1 and 4.2) and assume that we achieve game balancing when the difference of the skills between the agent and the player is in a certain range of difficulty tolerance. Consequently, we define a balancing constant (BC) that serves as the upper limit of the balancing interval (equation 5.2) and determines the tolerance range of difference between the capacity that the agent should achieve concerning the user, to assume that game as balanced.

$$\Delta Skill = Skill_{Agent} - Skill_{Player} \quad (5.1)$$

Thus, the BC tells the maximum difference of a skill (equation 5.1) that the agent and the player can have during the game. Then, the agent reached the state of balancing when it has a certain level of behavior that is equal to or higher than the player ( $\Delta Skill \geq 0$ ) but requiring that the agent's level does not exceed the BC ( $\Delta Skill \leq BC$ ). Considering this, we can see the BC as a value that defines the maximum challenge that a game can have concerning any player in particular.

$$0 \leq \Delta Skill \leq BC \quad (5.2)$$

Next, we devise the reward function coped with the balancing constant.

### 5.1.2 BC-Based Reward Function (BCR)

The reward function we propose is composed of:

- **An independent variable  $\Delta Skill$** , it can vary throughout the iterations because the player and the opponent can have a better and/or worse performance. Hence, the environment should yield different reward values depending on the balancing situation.
- **The balancing constant  $BC$** , this value is the maximum difficulty barrier that

the agent can represent to the player. It remains constant throughout the training process.

- **The maximum possible difference of abilities in the game  $Skill_{max}$ .** This constant value is used to normalize the reward value through the range of  $[-1; 1]$  and represents the maximum difference that both can have in the case that one of them is more capable than the other during the whole game, producing a significant disadvantage.

Regarding the balancing state, two situations may be considered: balanced and unbalanced. The unbalanced state may be due to two reasons:

1. The lack of ability of the agent to at least deal with the ability of the player, which means that the learner agent has a bad performance - inferior to the user's capacity. This may lead to a boring situation.
2. Excessive skill acquired by the agent to beat the player, i.e., the agent became smarter than the player, being much more difficult to be confronted. This may lead to a stressful situation.

For these reasons, the proposed reward function stimulates the agent in two ways for learning a balanced behavior: as soon as the agent is in either one of the two unbalanced states, he receives a punishment (negative reward). If in the next step the agent is closer to be balanced, it will get a punishment, so that it receives an input that it needs to continue improving. Nevertheless, it gets a positive reward when it is maintaining the game in a balanced state. For so, the reward value (equation 5.3) contemplates one of the following cases:

- **Subjugated Punishment (SP), defined as  $\Delta Skill < 0$ :** In this case, the negative reward value is directly proportional to the inability of the agent to confront and/or represent an obstacle for the player, i.e., i.e., the less skilled he is, he will receive more punishment. As one of the objectives in RL is to maximize the reward, the function intends to teach the learner to try turning up more closely to the balanced state in the following step (it will receive a punishment smaller than the last step) or, otherwise, it will be more punished.
- **Conservation Reward (CR), defined as ,  $0 \leq \Delta Skill \leq BC$ :** In this case, the positive reward value represents compensation for maintaining the degree of

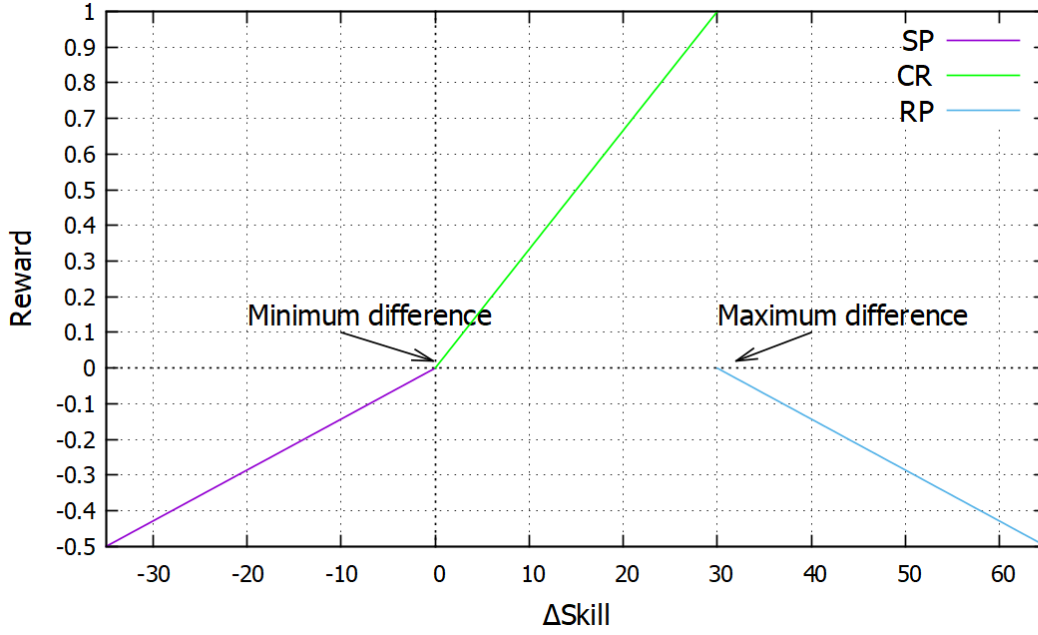


Figure 5.1: The BC-Based Reward Function with  $BC = 30$  and  $Skill_{max} = 100$ .

competition within the desired range of the BC. However, when the agent achieves the goal state it can stay in the same situation (neither more or less advantage concerning the player) or it could try to be even better than the last step until reaching the limit of equilibrium. But, the previous mentioned depends on the goal or difficulty that the game developer intends.

- **Rebellious Punishment (RP)**, defined as  $BC < \Delta Skill$ : This case is an inverse approach to the Subjugated Punishment situation because the punishment is obtained by overcoming the harassment limit (difficulty tolerance limit) that the agent can have concerning its opponent, so it does not generate a stress situation to the player. Then, the composition of the reward tries to inform it to lower its level until it is equal or less than the BC.

$$BCR(\Delta Skill) = \begin{cases} \frac{\Delta Skill}{Skill_{max}}, & \text{if } \Delta Skill < 0 \\ \frac{\Delta Skill}{BC}, & \text{if } 0 \leq \Delta Skill \leq BC \\ -\frac{\Delta Skill - BC}{Skill_{max} - BC}, & \text{if } BC < \Delta Skill \end{cases} \quad (5.3)$$

It is possible to observe in the Figure 5.1 that  $BCR(\Delta Skill)$  is a piece-wise function that reflects our idea of the three possible significant cases that the agent may achieve during training. Thus, if the agent is very distant from the range of balancing, as a consequence, it receives a more significant punishment; on the other hand, if it finds that

the game is balanced, it is stimulated to stay in the range limit of balancing.

## 5.2 BC-Based Balancing Metric (BCM)

The BCM depicted in equation 5.4 is a metric that allows the evaluation of the percentage of times that an agent is in one of the relevant states to achieve balancing: Unbalanced by Subjugate, Balanced, or Unbalanced by Rebellious.

$$BCM = \frac{T_B}{T_{US} + T_B + T_{UR}} \quad (5.4)$$

Our hypothesis is that as a base case of balancing, the player can perform different actions that may cause the agent to receive a punishment for not keeping the game within the BC interval, so the agent must perform an action to control this situation. In other words, an unbalanced state will make the agent to execute an action to move the game to a balanced state, as shown in figure 5.2.

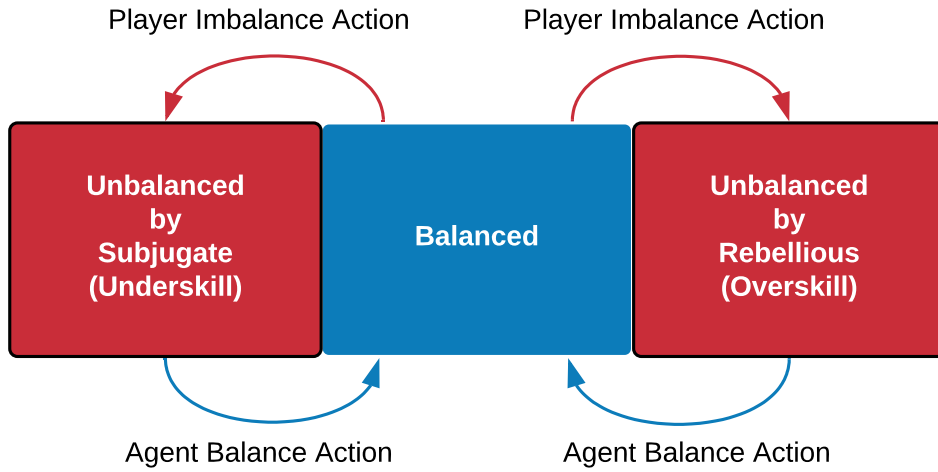


Figure 5.2: Minimal Agent Balancing Interaction.

Therefore, in the worst case, when the BC value is proximal to zero (the agent and the player have almost similar level), the agent will try to keep the balance after the player acts. That means, the game must be balanced at least half of the time; then, the value of our measure should be greater than or equal to 50% because per each action of the player to unbalance the game, the agent will perform a new action for trying to balance the game.

In this chapter, we presented our reward function, the main idea of our contribution,

---

and how it was used as stimulation to direct to the agent to achieve the balance state and main on it during the game. In addition, we explained a metric to eval how many times the agent will perform actions to get the balancing. In the next chapter, we will present details about the experimentation process, our case of study and the respective fitting of our framework for it.

# Chapter 6

## A case Study of the NRL Balancing Framework in a Fighting Game

In this chapter, we present our test scenario and show the results we have obtained with the game balancing method we proposed in this dissertation.

### 6.1 Test Game Scenario

We evaluate our game balancing method in an environment similar to the commercial game Capcom Street Fighter, inspired by [3]. The game consists of the confrontation of two entities, where one is called the *learner agent* and the other is called the *player* or the *enemy*, where both of them have the same goal: to defeat the opponent through physical attacks. As usual in these type of games, the entities have some life health points which decrease according to the effect of the enemy attack. Thus, the game ends when one of them makes the other to be lifeless or when the time of 100 seconds has passed. Naturally, in the first case, the surviving entity is the winner of the game, while, in the second case, the winner is the player that has more life points at the end of the game. We have developed the game in a 3D environment using the Unity engine so that we could show that our approach is applicable in a more challenging environment than a flat-based one where there is not much room to explore the field.

#### 6.1.1 Player Simulation

To simulate the behavior of players with distinctive nature, we developed two types of enemies to confront our learner agent, as follows.

- **Defensive player:** This programmed enemy emulates the behavior of a player whose goal is still to win but not at the expense of losing too many life health points, even if this means that his adversary will not suffer much damage as well. Thus, he adopts a defensive posture to avoid taking any damage as much as possible while still maintaining some advantage over its adversary. This simulation is a little more complicated respect to the previous one because the agent will attempt to learn to defend himself and to attack someone, which protects itself to have not more damage than its opponent so the agent will have to observe attack and defensive actions of the simulation and he will try to learn of them.
- **Aggressive player:** The behavior of this other type of emulated player differs from the previous one because this one tries to win with the maximum possible difference of life health score compared to its enemy. Thus, it tries to attack all the times to generate the most significant damage to our learner agent. With this kind of simulation, we pretend to teach our agent to defend himself of with players that constantly attack and, at the same time, to show him how to attack them to get an advantage (and stay inside the state of balancing). This work is easy because the learner only will have to observe the attack actions of the player to avoid damage, but maybe he could have problems trying to attack someone that always attack.

### 6.1.2 Skill Study

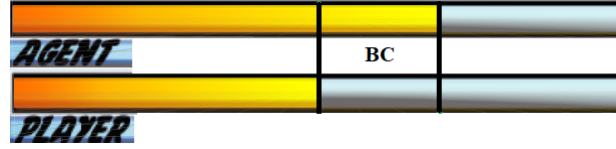
Naturally, as we work with a fighting game, we consider adequate to measure how balancing the game is by computing the difference in the life health score of the learner agent and the player (Equation 6.1). As a result, we consider the state of balancing of the game when the agent maintains a difference of life score within the BC interval established at Equation 6.2).

$$\Delta H = H_{Agent} - H_{Player} \quad (6.1)$$

$$0 \leq \Delta H \leq BC \quad (6.2)$$

In this way, the learner agent will we seek to maintain this health difference  $\Delta H$  between 0 (meaning that the learner agent ability is similar to the player, as they are both with very close health scores) and the balancing constant ( $BC$ ) (the maximum

capacity that we would like that the learner agent reaches to excel against the player). For example, if we want to balance this game concerning a  $BC = 30$ , in time  $t_i$  the difference of health score will be minor or equal to 30 (with an advantage of the agent). And if in the next time  $t_{i+1}$  the player hurts our agent, he will perform some action to get some prevalence again (a difference minor or equal to 30) like the Figure 6.1 shows.

(a) Balancing State at the time  $t_i$ (b) Balancing State at the time  $t_{i+1}$ Figure 6.1: Persistence of the BC throughout the game with  $BC = 30$ 

### 6.1.3 Observation Parameters (State) and Actions

The reinforcement learning task requires that we describe the decision problem that the agent faces in terms of states and actions so that the agent can learn the policy function. Thus, we must abstract the environment into the state representation (without process frames), and the brain maps this observation into action according to the policy. After executing an action, the agent receives a reward from the environment, which will guide him towards reaching the goal.

Figure 6.2 illustrates these elements within the learning loop of our study game. We send directly numeric information(continuous and discrete) to represent the observation to avoid the trouble of extracting the features directly from the frames. Thus, as we tackle a fighting game, besides the information coming directly from the physical environment itself, we include the observation that the learner agent has from the current state  $S_t$  of the game using the following set of information:

- Information related to the physical space: distance between the learner agent and the player.
- Information related to the balancing constant: the value of the difference between the health of the agent and the health of the player and the type of reward the agent



receives (subjugated or rebellious punishment, or conservative reward).

- Information related to the actions: the last action of the agent and the player, as well as a binary value that tells us if the player still performs the same action (he has the same behavior), this could help the agent to avoid some players actions that they are acting to achieve something.

All of these values are normalized between -1 and 1, as a recommendation of the Unity plugin for better performance.

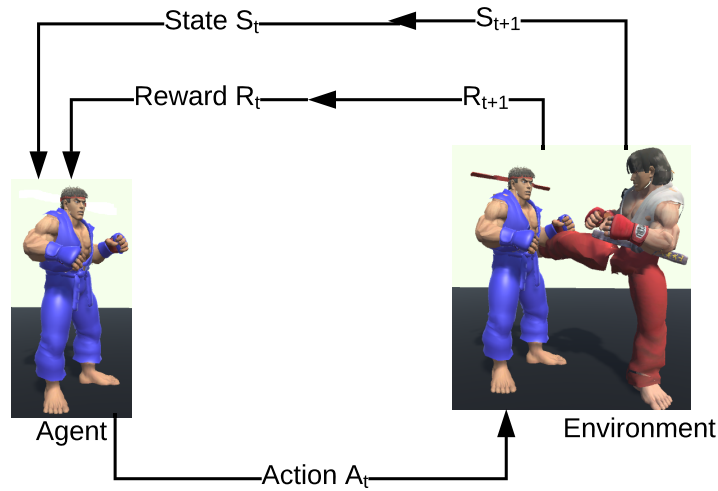


Figure 6.2: RL process focused on games.

Concerning the actions, both agents can Walk forward, Walk Backward, Duck, Kick, and Punch.

## 6.2 Experimental Results

Next, we proceed to present the results we have obtained from the game scenario we have just described, starting with the experimental setting before going to the results themselves.

### 6.2.1 Experimental setting

We experimented with three different values of BC (20,30,40), each one of them considered by the learner agent as the limit of its ability in a separate training process. As we described earlier, we created two different types of emulated players to face our agent and to allow him to train with them. In the end, our ultimate goal was to see the behavior

of our learner agent when facing these different types of enemies. Thus, we expected to have the following trained agents with balanced policies:

- **Agents presenting the same type of behaviour as their opponents:** In this case, we expect two different types of agents, namely an aggressive and a defensive one, which fights and behave within the boundary of the BC when confronted with their same kind of agent emulated. Thus, we expect the agent gets more aggressive when it is trained against the defensive emulated enemy since it is towards the victory. However, it should not be so offensive to the point that the balancing is impaired. Likewise, we expect to train a defensive agent when it is against the aggressive emulated enemy, as a reaction from the aggressive behavior. In the same way, it should not be so defensive that the player easily wins, as it stills stands for a balanced game.
- **An agent that does not assume an unique behaviour of their opponents:** The third type of agent (Mixed Agent) is trained with the two different types of emulated players. With that, we would like to demonstrate that given the experience of facing different types of players, the agent will be able to adapt to them when it becomes necessary.

As we described in Section 4.4, we used the PPO algorithm implemented in the Unity ML-agent plugin, and it has a default setting like Table 6.1 shows, this configuration is used as a baseline for new trainings. Based on preliminary validation tests, we change the values of four parameters (Table 6.2) to improve the training like following describe:

- The first one is the  $\beta$  parameter, responsible for defining how the agent explores its actions. The greater is this value, the more at random the agent chooses its actions and, therefore, the more it will explore the action space during its training. This variable is directly related to making the policy "more random," and we set this variable as  $(1e - 5)$  so that our agent experiences new possibilities and applies the learned model to reach the goal. During the preliminary experiments, smaller values of this  $\beta$  parameter made the agent to learn only a certain number of actions that, while helping it reach part of its goals, were very repetitive.
- The learning rate parameter was also modified with the same intent of allowing better exploration during the training phase. We set this variable to have the value of  $(5e - 4)$  so that the agent maintains an equilibrium between neither learning too fast nor delaying too much when exploring new actions.

Setting	Description	Value
batch_size	The number of experiences in each iteration of gradient descent.	1024
beta	The strength of entropy regularization.	$5.0e - 3$
buffer_size	The number of experiences to collect before updating the policy model.	1024
epsilon	Influences how rapidly the policy can evolve during training.	0.2
gamma	The reward discount rate for the Generalized Advantage Estimator (GAE).	0.99
hidden_units	The number of units in the hidden layers of the neural network.	128
lambd	The regularization parameter.	0.95
learning_rate	The initial learning rate for gradient descent.	$3.0e - 4$
max_steps	The maximum number of simulation steps to run during a training session.	$5.0e4$
memory_size	The size of the memory an agent must keep. Used for training with a recurrent neural network.	256
normalize	Whether to automatically normalize observations.	false
num_epoch	The number of passes to make through the experience buffer when performing gradient descent optimization.	3
num_layers	The number of hidden layers in the neural network.	2
time_horizon	How many steps of experience to collect per-agent before adding it to the experience buffer.	64
sequence_length	Defines how long the sequences of experiences must be while training. Only used for training with a recurrent neural network.	64
summary_freq	How often, in steps, to save training statistics. This determines the number of data points shown by TensorBoard.	1000

Table 6.1: ML-Agents Default Settings

Setting	Value
beta	$1.0e - 5$
learning_rate	$5.0e - 4$
max_steps	$1.0e6$
num_layers	4

Table 6.2: Our Training Settings

- The maximum number of training steps was experimented with  $500k$  and  $1M$ .

All the other parameters were kept with the default values of the plugin.



Figure 6.3: Initial stage of the training process with 9 scenes in parallel.

We created nine independent copies of the scene to accelerate the training (every copy of the agent is like a new dataset of the training) by making them working together to find a better policy; they synchronously update the global network as we exposed in the last part of 3.3. Thus, all of these parallel scenes had agents fighting against the same type of emulated agent, except for the mixed agent training that had four passive and five aggressive emulations. The figure 6.3 shows how each of them is independent since some learn to balance (green platforms) while some of them do not achieve the best policy at the same time as the others (red platforms).

## 6.2.2 Results

Based on the experiments carried out, we start by indicating that from 500k steps<sup>1</sup> until 1M (about an hour and a half), the agent tends to continuously improve its policy, generating an increasing greater reward as shown in Figure 6.5. We can observe from the Figure that the mixed agent has an average performance between them because it has to learn to defend himself, to attack, and to look for the enemy to obtain the advantage and to stay balanced. This way, he has more work to do, but this has given him more experience to try new actions and not maintaining the same policy, as was the first case. Comparing

<sup>1</sup>Step: Each iteration where we can make an observation, to perform an action or receive a reward.

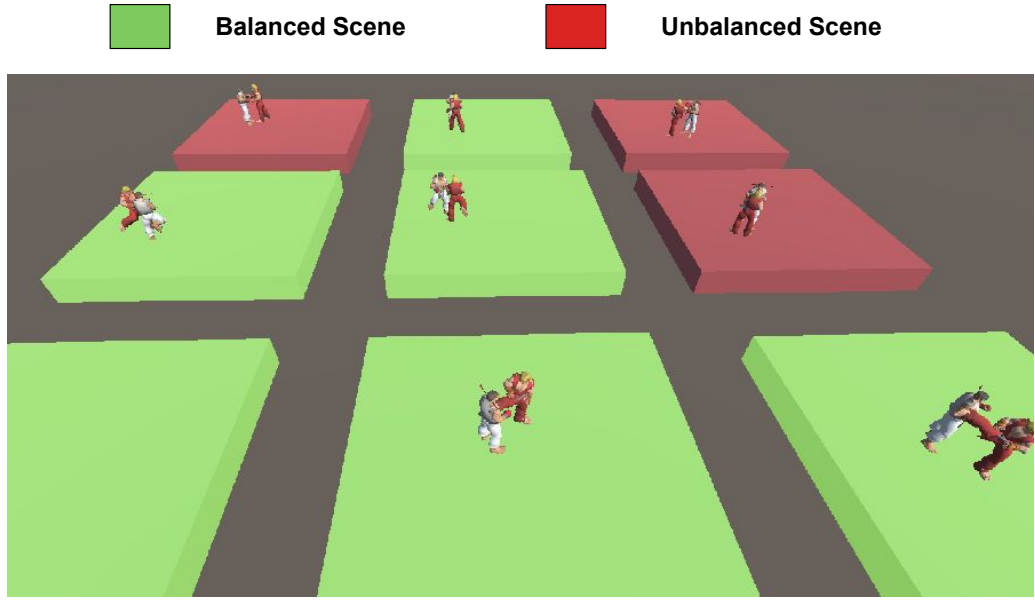


Figure 6.4: Intermediate stage of the training process with 9 scenes at the same time.

the agents that face single-behaviour enemies, the agent that fights against the aggressive emulated player learned more slowly, resulting in smaller accumulated reward, because, in addition to learning attack and/or defense policies to maintain the advantage, he had to learn how to look for his enemy to attack it within the scene.

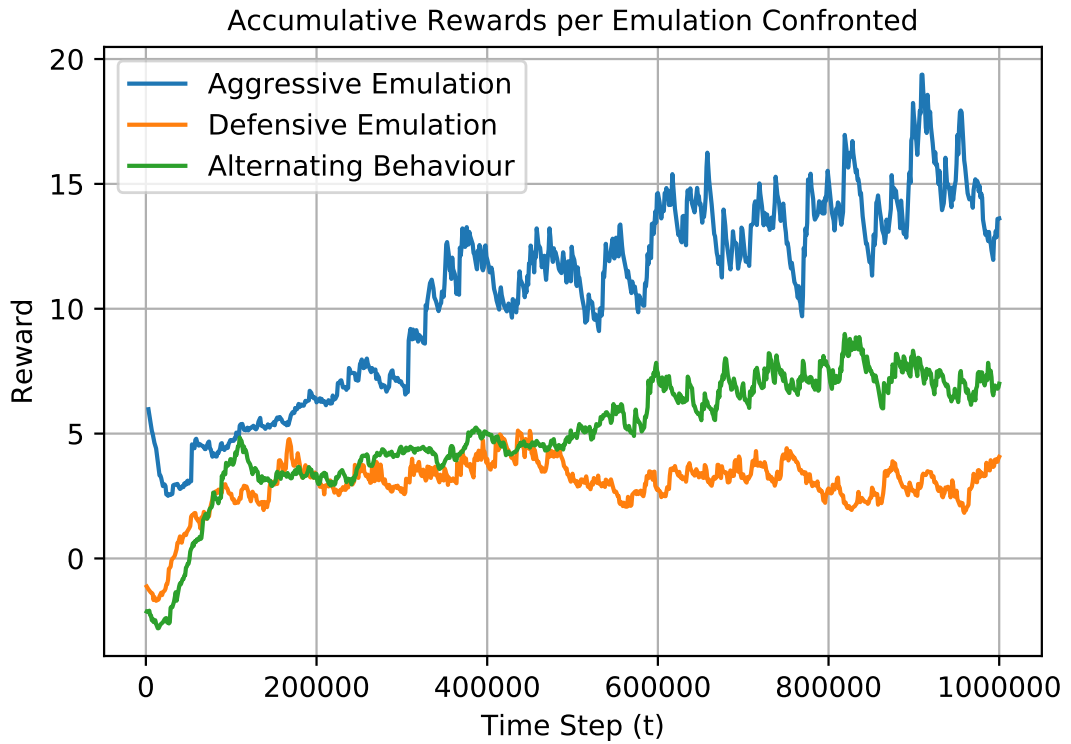


Figure 6.5: Accumulated Reward throughout the training for  $BC = 30$ .

Hence, in the case of the agents that learn from only one type of player, we observed that the agent who faces aggressive emulation did not take long to learn the (defensive) policy and then maintain it until finished the number of iterations. Because the emulation is so offensive, the agent determined that it is better to defend itself and keep a minimum difference than trying to reach a  $\Delta Skill = BC$  or higher (be more aggressive). Unlike the other, he tries to be more aggressive, and his training goes more slowly since the agent must also learn how to attack (to reach the balancing state) and defend himself (to stay within the balancing state). Figure 6.6 shows how the agent that learns with the mixed emulation of two types of players has an average reward gain between the other two cases. This situation is likely because he must learn how to behave when facing two different strategies of its opponents. Consequently, he has more things to learn. Regarding the constant balancing values, the results show that when the BC is higher, the agent can maintain his skill difference concerning the player into the interval of balancing. He is also able to remain balanced even when he suffers some damage by the opponent because there is still some life difference between them.

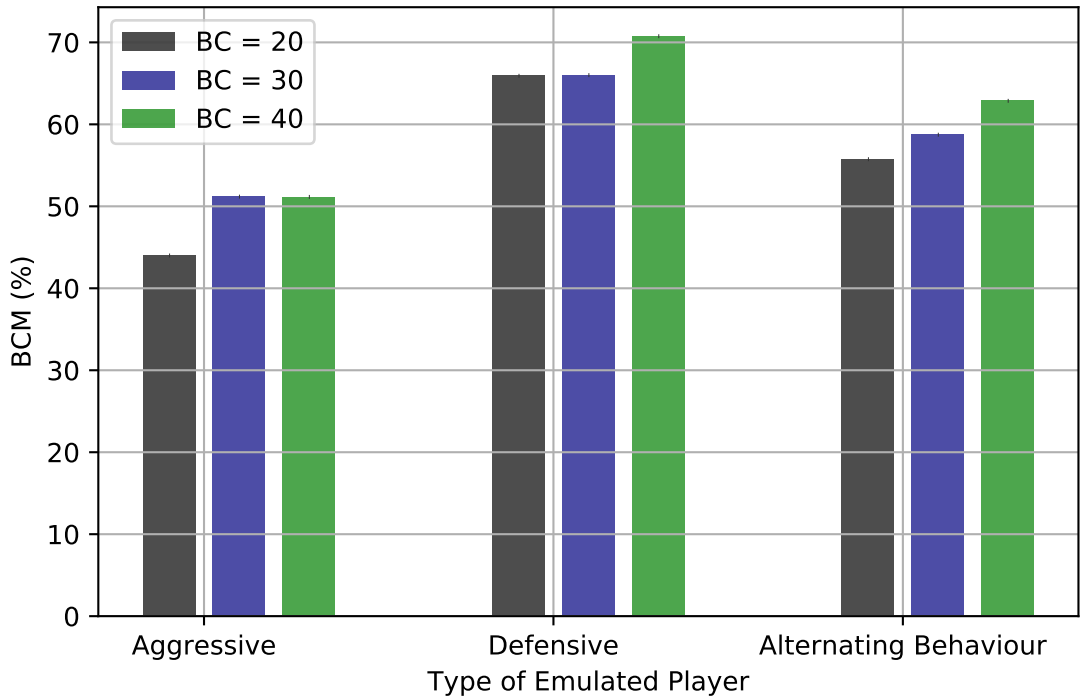


Figure 6.6: Evaluation on BC-Metric

Conversely, it is quite hard to preserve the equilibrium when the BC is small because any wrong action by the agent could take it to any of the unbalanced states. For example, we can observe in Figure 6.6 when the agent has a  $BC = 20$ , and he needs to fight against an aggressive player, the former spends a little less than 50% of the time in our target

balanced state. However, one may wonder if this is a little contradictory, since in those cases we allow for the learner agent to have a greater distance in life points than its adversary. The agent has the possibility to stay more time inside the balancing state because, if he makes a mistake, possibly he can still have an advantage over the player but, at the same time, the game could seem to unbalance because the difference of skill is farther than zero (the agent is more difficult to face by the player).

Our BC-Metric showed three interesting points: First, the agent remains at least 50% of time in the state of balancing in all cases, except when  $BC = 20$  as we described before. Second, we see some evidence to confirm the hypothesis about a small value of BC, showing that in this case, it is difficult to stay balanced with a narrow range. Third, the results show that by offering new experiences through the mixed agent, it can achieve better decisions because he learns a new type of behavior that one style alone cannot provide. In this case, the results exceed 50% of BC-Metric, even with a  $BC = 20$ , as exhibited in Figure 6.6.

Type of Emulation	BC	Win	Lose	$\overline{\Delta Skill}$
<b>Aggressive</b>	20	0.69	0.31	12.19
	30	0.69	0.31	22.62
	40	0.63	0.37	35.15
<b>Defensive</b>	20	0.72	0.28	10.14
	30	0.68	0.32	23.91
	40	0.66	0.34	38.94
<b>Alternating Behaviour</b>	20	0.67	0.33	-2.58
	30	0.69	0.31	-0.42
	40	0.70	0.30	1.02

Table 6.3: Results during 100 game sessions

Finally, as part of our experiments, we also evaluate the number of times that our agent wins (or loses) during 100 game sessions, and at the same time, we calculated the mean of skill difference ( $\overline{\Delta Skill}$ ), the mean difference of health score in our case, to analyze if the learner really is balancing the game according to the BC. Thus, the experiments demonstrated us the agents are winning at most 60% of sessions, and when it faces singly an aggressive (or defensive) simulation, the agent maintains  $\Delta Skill$  around the value BC, but if it faces alternating both styles of the simulation the mean is around 0 because it is trying to deal with two different styles. Consequently, we can determine the agent learned to balance the game, but it is limited to not exceed the BC because we are leading to our learner to win with at most by a difference of BC, by using our reward function during

---

the training.



# Chapter 7

## Conclusions and future work

Game customization is an essential subject for increasing the player's satisfaction in game sessions. There are different ways to achieve this customization, such as modifying the game space and scenario, game mechanism, storytelling, characters, among other elements. Another strategy for customizing consists of balancing the game challenge, making it becomes easier or harder. Among different ways for doing so, we detach: (1) manual difficulty configuration, such as selecting easy, medium or hard modes; (2) approaches with heuristics and/or probabilistic functions to evaluate or predict situations and finally, (3) Machine Learning approaches, using classifiers or basic Reinforcement Learning strategies to train agents. All these methods may present limitations, such as to find an adequate mode of difficulty (1), the absence of sufficient functions to model specific situation of a particular game (2), and imprecise models to represent a specific player (3).

In this work, we contributed with a more general neural reinforcement learning strategy, which can avoid the usage of players configuration, predefined models. Our solution also avoids the necessity of having specific players representation and building dedicated fitting functions. We created a framework based on the Unity ML-Agent Toolkit, that takes advantage of the advances of DRL. In our work, we do not need the player model or classifier, since we were focused on training the agent based on the game state observation through the usage of a proposed reward function. Our proposed function uses a balancing constant to limit and measure the behavior of the agent. By training the agent with such a function, we were able to not only restrict the agent to behave like the player (when the BC is zero), but also to give him the possibility of stimulating himself to learn the games' actions so that it is still a good adversary for the player. With the experimental results, we could observe that the agent stays as long as possible in a balanced state (at least 50% of the game). Basically, the agent achieved to adapt its behavior based on

the observations and the BC value used to the training. Regarding this last element, we understood that when the value is proximal to zero (similar skill level), the agent will have to perform better actions to try to keep the game balancing, since if it has comparable levels, any movement of some of them will unbalance the game, generating advantage for someone.

Through our results, we observed that the agent has a good performance when it faces an enemy based on a specific player. Nevertheless, when it trained with different types of players, it was able to learn more actions and, consequently, became more skilled and with better performance than an agent that was trained with only a difficult emulation. It is possible to conclude that with enough different experiences an agent may continuously adapt itself to balance well a game.

## 7.1 Limitations

Along the work, we observed some limitation since the theory until part of the application to run our experiments. Concerning the theory baseline, the different authors do not have an agreement about using the same form of terminology about the description since Markov Decision Process until the state-of-the-art of algorithms with RL. Thus, it difficulty the process of learning, understanding and describing to reproduce them in new papers or this dissertation. Concerning software available to implement our work, we chose Unity ML-Agent Toolkit because it contains the PPO and with it is possible to create many learning environments with less effort, but during the process, we found some bugs in the plugin and new modification on it. Consequently, we needed to modify the implementation of our test and so on.

## 7.2 Future Works

As future work, first, we would like to verify if our framework can balance other types of video games and analyze the impact of using different kinds of reward functions that also aim to balance a game. In this way, it is essential to consider that the resulting model of the training process could be used as a starting point to the training process of another agent in a different game, so this could help us to not train our balancing agent from zero (without any knowledge) and possibly achieve a good result faster by using Transfer Learning [29]. We also plan to allow for our agent to behave like an online learner agent,

so that when facing real players it can continuously learn from the user style of playing.

As a next step, we consider it is essential to conduct a series of experiments with real players so that we could observe how the balancing agent performs in a real environment. Thus, we would like to verify if the player is satisfied with the game, which is one of the main goals of the balancing games area. To that, it is essential to define some satisfaction metrics, in a way that we can evaluate the results not only questionnaires but also evaluation functions that do not entirely depend on the way the player would answer our survey.

Finally, we would like to study the balancing challenge with multi-agents that cooperates to achieve their goals employing communication and cooperation methods [14, 43].

To conclude, we would like to point out that this work has been published at [27]. We hope that the paper and this dissertation have contributed a little more to advance the integration of AI and Games to achieve balanced games, focusing on user entertainment.

# References

- [1] ABADI, M., OTHERS. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), p. 265–283.
- [2] AGUIAR, M., BATTAIOLA, A. L. Gameplay: uma definição consensual a luz da literatura. *XV Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames)* (2016), 531–538.
- [3] ANDRADE, G., RAMALHO, G., GOMES, A. S., CORRUBLE, V. Dynamic game balancing: An evaluation of user satisfaction. *AIIDE 6* (2006), 3–8.
- [4] ANDRADE, G., RAMALHO, G., SANTANA, H., CORRUBLE, V. Extending reinforcement learning to provide dynamic game balancing. In *Proc. of the Workshop on Reasoning, Representation, and Learning in Computer Games, 19th Int. Joint Conference on Artificial Intelligence (IJCAI)* (2005), p. 7–12.
- [5] ARULKUMARAN, K., DEISENROTH, M. P., BRUNDAGE, M., BHARATH, A. A. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38.
- [6] BAKKES, S., TAN, C. T., PISAN, Y. Personalised gaming: a motivation and overview of literature. In *Proc. of The 8th Australasian Conference on Interactive Entertainment: Playing the System* (2012), ACM, p. 4.
- [7] BAKKES, S., WHITESON, S., LI, G., VIŞNIUC, G. V., CHARITOS, E., HEIJNE, N., SWELLENGREBEL, A. Challenge balancing for personalised game spaces. In *2014 IEEE Games Media Entertainment* (Oct 2014), p. 1–8.
- [8] BAKKES, S. C., SPRONCK, P. H., VAN DEN HERIK, H. J. Opponent modelling for case-based adaptive game ai. *Entertainment Computing* 1, 1 (2009), 27–37.
- [9] CHARLES, D., MCNEILL, M., MCALISTER, M., BLACK, M., MOORE, A., STRINGER, K., KÜCKLICH, J., KERR, A. Player-centred game design: Player modelling and adaptive digital games.
- [10] CSIKSZENTMIHALYI, M., CSIKSZENTMIHALYI, I. S. *Optimal experience: Psychological studies of flow in consciousness*. Cambridge university press, 1992.
- [11] GHAVAMZADEH, M., KAPPEN, H. J., AZAR, M. G., MUNOS, R. Speedy q-learning. In *Advances in neural information processing systems* (2011), p. 2411–2419.
- [12] HORNIK, K., STINCHCOMBE, M., WHITE, H. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

- [13] HUNICKE, R., CHAPMAN, V. Ai for dynamic difficulty adjustment in games.
- [14] JIANG, J., LU, Z. Learning attentional communication for multi-agent cooperation. In *Advances in Neural Information Processing Systems* (2018), p. 7265–7275.
- [15] JULIANI, A., BERGES, V.-P., VCKAY, E., GAO, Y., HENRY, H., MATTAR, M., LANGE, D. Unity: A General Platform for Intelligent Agents. *ArXiv e-prints* (setembro de 2018).
- [16] KONDA, V. R., TSITSIKLIS, J. N. Actor-critic algorithms. In *Advances in neural information processing systems* (2000), p. 1008–1014.
- [17] LAMPLE, G., CHAPLOT, D. S. Playing fps games with deep reinforcement learning. In *AAAI* (2017), p. 2140–2146.
- [18] LAURSEN, R., NIELSEN, D. Å., CAPRANI, O. Investigating small scale combat situations in real time strategy computer games.
- [19] LOPES, R., EISEMANN, E., BIDARRA, R. Authoring adaptive game world generation. *IEEE Transactions on Games* 10, 1 (2018), 42–55.
- [20] MICHALSKI, R. S., CARBONELL, J. G., MITCHELL, T. M. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.
- [21] MINDEL, M., LYONS, K., WIGGLESWORTH, J., Eds. *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering, CASCON 2017, Markham, Ontario, Canada, November 6-8, 2017* (2017), IBM / ACM.
- [22] MISSURA, O., GÄRTNER, T. Player modeling for intelligent difficulty adjustment. In *Int. Conference on Discovery Science* (2009), Springer, p. 197–211.
- [23] MITCHELL, A., MCGEE, K. Designing storytelling games that encourage narrative play. In *Joint International Conference on Interactive Digital Storytelling* (2009), Springer, p. 98–108.
- [24] MITCHELL, T. *Machine learning*. McGraw-Hill Boston, MA:, 1997.
- [25] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILICRAP, T., HARLEY, T., SILVER, D., KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In *Int. Conference on Machine Learning* (2016), p. 1928–1937.
- [26] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLU, I., WIERSTRA, D., RIEDMILLER, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
- [27] NOBLEGA, A., PAES, A., CLUA, E. Towards adaptive deep reinforcement game balancing. In *ICAART* (2019), vol. 2, p. 693–700.
- [28] OLESEN, J. K., YANNAKAKIS, G. N., HALLAM, J. Real-time challenge balance in an rts game using rtneat. *2008 IEEE Symposium On Computational Intelligence and Games* (2008), 87–94.
- [29] PAN, S. J., YANG, Q. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.

- [30] PAN, X., YOU, Y., WANG, Z., LU, C. Virtual to real reinforcement learning for autonomous driving. *arXiv preprint arXiv:1704.03952* (2017).
- [31] PEERDEMAN, P. Sound and music in games. *And Haugehåtteit, O* (2006).
- [32] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., KLIMOV, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
- [33] SHAKER, N., YANNAKAKIS, G. N., TOGELIUS, J. Towards automatic personalized content generation for platform games. In *AIIDE* (2010).
- [34] SILVA, M. P., DO NASCIMENTO SILVA, V., CHAIMOWICZ, L. Dynamic difficulty adjustment through an adaptive ai. In *Computer Games and Digital Entertainment (SBGames), 2015 14th Brazilian Symposium on* (2015), IEEE, p. 173–182.
- [35] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., OTHERS. Mastering the game of go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.
- [36] SILVER, D., SCHRITTWIESER, J., SIMONYAN, K., ANTONOGLOU, I., HUANG, A., GUEZ, A., HUBERT, T., BAKER, L., LAI, M., BOLTON, A., OTHERS. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.
- [37] SOUTHEY, F., HOLTE, R. Semi-automated gameplay analysis.
- [38] SPRONCK, P., PONSEN, M., SPRINKHUIZEN-KUYPER, I., POSTMA, E. Adaptive game ai with dynamic scripting. *Machine Learning* 63, 3 (2006), 217–248.
- [39] SPRONCK, P., SPRINKHUIZEN-KUYPER, I., POSTMA, E. Difficulty scaling of game ai. In *Proc. of the 5th Int. Conference on Intelligent Games and Simulation (GAME-ON 2004)* (2004), p. 33–37.
- [40] SUTTON, R. S., BARTO, A. G., BACH, F., OTHERS. *Reinforcement learning: An introduction*. MIT press, 1998.
- [41] TENG, C.-I. Customization, immersion satisfaction, and online gamer loyalty. *Comput. Hum. Behav.* 26, 6 (novembro de 2010), 1547–1554.
- [42] VÉRON, M., MARIN, O., MONNET, S. Matchmaking in multi-player on-line games: studying user traces to improve the user experience. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop* (2014), ACM, p. 7.
- [43] WANG, B., OSAWA, H., SATOH, K. How implicit communication emerges during conversation game. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems* (2018), International Foundation for Autonomous Agents and Multiagent Systems, p. 2118–2120.
- [44] WATKINS, C. J., DAYAN, P. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.
- [45] WHITE III, C. C., WHITE, D. J. Markov decision processes. *European Journal of Operational Research* 39, 1 (1989), 1–16.
- [46] YANNAKAKIS, G. N., PAIVA, A. Emotion in games. *Handbook on affective computing* (2014), 459–471.