UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF MECHANICAL ENGINEERING

Application of Reinforcement Learning Techniques in Streetfighter II Environment

by

SPILIOPOULOS CHARALAMPOS

Submitted in partial fulfillment of the requirements for the degree of Diploma in Mechanical Engineering at the University of Thessaly

Volos, 2022

UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF MECHANICAL ENGINEERING

Application of Reinforcement Learning Techniques in Streetfighter II Environment

by

SPILIOPOULOS CHARALAMPOS

Submitted in partial fulfillment of the requirements for the degree of Diploma in Mechanical Engineering at the University of Thessaly

Volos, 2022

Approved by the Committee on Final Examination:

Advisor	Dr. Konstantinos Ampountolas, Associate Professor, Department of Mechanical Engineering, University of Thessaly
Member	Dr. Pandelis Dimitrios, Professor, Department of Mechanical Engineering, Aristotle University of Thessaly
Member	Dr. Saharidis Georgios, Professor, Department of Mechanical Engineering, University of Thessaly

Date Approved: [30th of June, 2022]

Acknowledgments

I would like to especially thank my teacher and thesis supervisor Dr. Konstantino Ampountola, for giving me the chance to work on this project and for providing me with his valuable help and time. Moreover, I am thankful to the rest of the members of the examination committee, professors Dr. Pandeli Dimitrio and Dr. Saharidi Georgio for reading and evaluating my thesis.

Lastly, I am thankful to my parents, Vasili and Agapi for supporting me in every possible way and being patient with me, especially during the times of writing this thesis. I dedicate this work to my grandfather Charalampo, who would be more than happy to know i completed my thesis.

Application of Reinforcement Learning Techniques in Streetfighter II Environment

SPILIOPOULOS CHARALAMPOS

Department of Mechanical Engineering, University of Thessaly, 2022

Supervisor: Dr Konstantinos Ampountolas

Associate Professor of Automatic Control Systems in Mechanical Engineering

Summary

In an ever-increasing digital world, the application of Artificial Intelligence (AI) methods is becoming more and more common. Their contribution can accelerate the progress in many different fields, from logistics and security to engineering and medicine, providing solutions or accelerating their constitution thus making human life easier. One major form of AI is Reinforcement Learning (RL) whose way of learning a task can be summarized as a procedure of trial and error regardless of a problem's complexity. In this thesis, two state of the art actor-critic RL algorithms are explored, using the game environment of Streetfighter for their implementation. First of all, a retrospection of AI attempts and effectiveness in playing various games through the years is presented; along with RL's theoretical background and its basic methods. Next, the methods used in the experimental process are examined by analyzing their basic characteristics, followed by a description of the environment of Street fighter II and the parameters that were used for the implementation of these methods. Finally, the experimental results are presented, compared, and discussed, ultimately determining the best model for the present application.

ΕΦΑΡΜΟΓΗ ΜΕΘΟΔΩΝ ΕΝΙΣΧΥΤΙΚΗΣ ΜΑΘΗΣΗΣ ΣΤΟ ΠΕΡΙΒΑΛΛΟΝ ΤΟΥ STREETFIGHTER II

Σπηλιόπουλος Χαράλαμπος

Τμήμα Μηχανολόγων Μηχανικών, Πανεπιστήμιο Θεσσαλίας, 2022

Επιβλέπων Καθηγητής: Δρ. Κωνσταντίνος Αμπουντώλας,

Αναπληρωτής Καθηγητής Συστημάτων Αυτομάτου Ελέγχου

Περίληψη

Με τη διαρκή επέκταση της ψηφιοποίησης, η ανάπτυξη και η εφαρμογή μεθόδων Τεχνητής Νοημοσύνης γίνεται όλο και πιο συχνή και γι'αυτό καθίσταται επιτακτική η μελέτη τους και η αναλυτική προσέγγισή τους. Η συμβολή τους μπορεί να επιταχύνει την πρόοδο σε πολλούς διαφορετικούς τομείς, από τα logistics και την ασφάλεια μέχρι τη μηχανική και την ιατρική, δίνοντας λύσεις ή επιταχύνοντας τη λύση πολλών προβλημάτων, κάνοντας την ανθρώπινη ζωή ευκολότερη. Η Ενισχυτική Μάθηση αποτελεί μια ιδιαιτέρως σημαντική μορφή της τεχνίτης νοημοσύνης και συγκεκριμένα της Τεχνίτης Μάθησης και συνιστά ένα τρόπο εκμάθησης μιας εργασίας την οποία μπορεί να συνοψιστεί ως μια διαδικασία δοκιμής και σφάλματος, ανεξάρτητα από την πολυπλοκότητα της. Σε αυτή τη διατριβή, γίνεται πειραματική δόκιμη δυο σύγχρονων μεθόδων βαθιάς ενισχυτικής μάθησης actor-critic, χρησιμοποιώντας ως πεδίο αναφοράς, το περιβάλλον του παιχνιδιού Streetfighter II για την υλοποίησή τους. Πρώτα απ' όλα, παρουσιάζεται μια αναδρομή στις επιδόσεις εφαρμογών τεχνητής νοημοσύνης σε διάφορα παιχνίδια μέχρι και σήμερα, καθώς και το θεωρητικό υπόβαθρο της ενισχυτικής μάθησης και των βασικών της μεθόδων. Στη συνέχεια, εξετάζονται οι αλγόριθμοι που χρησιμοποιήθηκαν κατά την πειραματική διαδικασία αναλύοντας τα βασικά τους χαρακτηριστικά, ενώ ακολουθεί η περιγραφή του περιβάλλοντος του Street fighter II και των παραμέτρων που χρησιμοποιήθηκαν κατά την εκπαίδευση τους σε αυτό. Τέλος, παρουσιάζονται τα πειραματικά αποτελέσματα, τα όποια συγκρίνονται και σχολιάζονται, καθορίζοντας εν τέλη το καλύτερο μοντέλο στην παρούσα εφαρμογή.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

In this chapter, the basic ideas behind Reinforcement Learning are described, along with examples of its application in game platforms emphasizing its most important implementations.

1.1. Introduction to Reinforcement Learning

The science of Machine Learning is categorized in supervised learning, unsupervised learning and reinforcement learning [1]. The first category includes algorithms that given labeled datasets they derive mapping functions which are then able to then predict outputs from given inputs. In the second category, algorithms try to find patterns between sparse data without given sets of input and output data. In contrast to those methods, reinforcement learning requires little to no previous information and algorithms learn tasks thru trial and error. This process takes place in the interactions between the agent and the environment, which are the source of the data and consistently accumulate experiences which are highly correlated. Using this sequential set of information, an optimum solution may be found, even though this process creates one significant challenge for RL algorithms, known as exploration versus exploitation.

Within the agent-environment structure, the agent first evaluates the current environment state, denoted by , in discrete time steps t, and then decides upon an action, denoted by A_t. The action causes a reaction to the environment which then moves to a new state denoted by S_{t+1}. Additionally, the agent receives a reward denoted by the letter R_t for his action [2]. As seen in figure 1.1, the agent and the environment interact in a loop caused by an action that leads to the next state S_{t+1}.

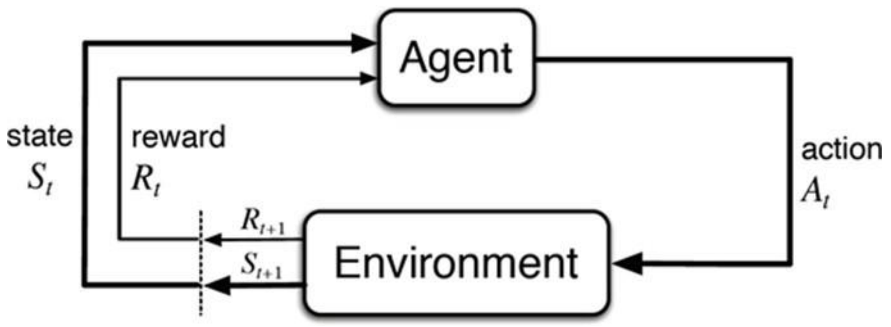


Figure 1.1: Depiction of the series of events in the agent-environment framework

As a consequence, the agent has to learn how to assess states and utilize these state evaluations to acquire the greatest outcome, considering the reward of a transition and all of the potential future rewards. By following the strategy that maximizes the future rewards, the best outcome is attained. To put it another way, the agent is trying to come up with a policy that maps every state in S, to an action in A, in such a way that when it is implemented, it provides a series of transitions that yields the maximum sum of rewards for each transition.

One major distinction in Reinforcement Learning methods is the possible existence of a model for the environment of a problem. In *model-based* learning, an agent creates a model that describes features of the environment he lives in, through attained experiences of interacting with it. In this kind of system, the direct effects of actions have less importance for the agent because of the predispositions created by the model. Thus, the values of the future states are taken into consideration rather than those of immediate actions (in contrast to model-free methods), and so are tied to the external structure of the environment and the internal model.
On the other hand, *in model-free* learning, the consequences of actions are found through experience. Specifically, an action will be carried out multiple times and a policy for optimal rewards will be adjusted, based on the outcomes. In other words, the case when the agent performs an action in order to find out what the result may be, corresponds to model-free learning. As a result, each of the two classes is useful for different applications.

A common analogy to RL is that of games, where the environment acts as the game itself and the agent is one of the players. The reward signal reflects how good the action was at the given state, for instance, the winning move in a game of chess would give the agent a positive reward, whereas a losing move would give the agent a negative reward (penalty). The aim of the agent is defined as maximizing the total rewards it receives from the environment. Thus, the reward signal can be understood as a stimulus to the agent guiding it toward an optimal strategy in the environment.

1.2. RL in Games

The idea of using computer systems and algorithms to train agents or just learn specific games is not new. For decades, scientists had thought of using computers to beat both computer and board games. Since the first tries, computer intelligence has come a long way and game platforms have a significant role in the advancement of the field. Some of them can be very complex with many limitations and parameters to consider and thus can showcase the capabilities of an algorithm. For that reason, it has become common place in the scientific community to use games as a testing platform for RL algorithms since they provide a perfect test-bed for measuring the progress of complicated AI systems.

The first time a computer program was successfully created to play against a human, was in 1979. In a friendly match, the BKG 9.8 backgammon computer program prevailed against the reigning World Backgammon Champion, Luigi Villa, by 7 points to have a historically important impact [3]. That was the first time that a human-created machine had ever been successful in dethroning a world champion in a recognized intellectual activity. There are a total of 10^{20} different positions that may be played in the game of backgammon, which is a game of both skill and chance (use of dice) and is comparable to the perplexity of checkers or bridge [4],[5]. Because it was heuristic software, it did not base its judgments on a comprehensive representation of the game of backgammon. Aside from a few pre-calculated tables, the whole software is based on heuristic functions that had been precisely optimized.

However, the incident that left a more significant mark in the history books happened from 1996 to 1997, when IBM challenged Garry Kasparov to a chess match against its chess-playing computer, Deep Blue [6]. Kasparov won the 1996 match 4-2 but in the next year's rematch, he lost 3 games while drawing 1 to the upgraded version of Deep Blue. It was a historic moment when a chess-playing computer, for the first time, prevailed against a reigning world champion under the standard conditions of a chess tournament. Since chess had been seen as the pinnacle of artificial intelligence evaluation for many decades, the achievement was commemorated as a watershed moment in the development of the field of artificial intelligence [7]. Deep Blue was artificial intelligence (AI), but in reality, it was more of a hybrid since it depended less on machine learning than modern systems do and more on a brute force approach. This meant that it used sheer processing capacity to test every option rather than sophisticated approaches to enhance efficiency [8]. The concept of big data was still in its infancy, and the technology at the time could not have supported enormous networks. The software was responsible for the more fundamental components of the chess calculations, while the hardware's accelerator chips were responsible for searching through a tree of potential outcomes to determine the optimal moves based upon an alpha-beta search algorithm.

In 2011, people all across the globe watched as IBM's Watson succeeded in a friendly match of Jeopardy against the show's two most renowned players, one of whom had won 74 straight programs in a row. Watson, a computer system that functions as a search engine and has remarkable natural language processing (NLP) and reasoning skills, has shown that computers are capable of not just excelling in mathematical strategy games but also games based on knowledge and communication. A Game State Evaluator, also known as a GSE, was trained using millions of rounds of simulated games between Watson and humans to assess the influence that a wager has on a player's likelihood of winning [36]. A feature description of the current game state is given as an input into the GSE, which then uses a neural network to achieve smooth nonlinear function approximation and generates an estimate of the chance that Watson would eventually win based on the game's present state. The players scores and different measurements of remaining game time were stored in a feature vector, which was used to train the algorithm [37].

In 2016 another milestone in the AI development took place when Google’s Deepmind managed to defeat the 18 times and current world champion (at the time) in GO, Lee

Sedolusing four to one [9]. Before then, it was expected that it would be at least 10 years before computers were able to beat professional Go players in the full-sized game. The earlier attempts of using artificial intelligence to defeat a human being in a game environment are not comparable to AlphaGo's implementation for which machine learning was utilized to figure out how to play and practice the game rather than pre-programmed probability calculations. It is claimed that the board game Go, which originated in China more than two thousand five hundred years ago, is the oldest board game that is still being played today [10]. The game of Go is deceptively difficult while having some rather straightforward rules. In comparison to chess, go has a larger board with a greater range of possible moves, lengthier games, and, on average, more potential moves to evaluate before making a decision. The number of atoms in the known universe is thought to be on the order of 10^{80}, while the number of permissible board places in Go has been computed to be roughly 2.1×10^{170} that is significantly higher. The program made use of three different policy networks, two of which were trained using supervised learning on experts' movements, and one of which was learned using policy gradients and self-play approaches (reinforcement learning) [11]. These three methods were merged into one in the Monte Carlo tree search, which used the value function obtained by the Rl algorithm to determine how its branches should be set up.

After their success, the company made two upgraded versions of the same algorithm, the AlphaGo Zero in 2017 and the AlphaZero in 2018. The first surprisingly outperformed its predecessor AlphaGo just 36hours later, and managed to win 100 to 0, only using one neural network and training only thru self-play which was the most important difference from AlphaGO [12], [14]. The latter was deployed across multiple computers and used a total of 48 tensor processing units (TPUs), but AlphaGo Zero only utilized a single system with 4 TPUs. AlphaZero is a generalized extension of its predecessor that is also capable of successfully playing chess and shogi [15]. Within the first twenty-four hours of its training, the algorithm acquired a top notch standard of competition in these games as shown by its victory against the world-champion programs Stockfishm, Elmo, and the three-day version of AlphaGo Zero[16]. Shogi is played on a larger board than chess and is a significantly harder game in terms of computational complexity. The Computer Shogi Association (CSA) world-champion Elmo, was the strongest shogi program and had only recently defeated human champions [17]. The next year, DeepMind published a new study describing MuZero [18], a new algorithm capable of generalizing AlphaZero's applications by playing both Atari and board games without knowledge of the game's rules or representations.

A year and a half later, the same firm revealed AlphaStar, a computer program taught to play Starcraft II, a video game that was a significant obstacle for artificial intelligence researchers for more than a decade. The AI that was trained achieved the level of GrandMaster after defeating the top player Grzegorz "MaNa" Komincz and his colleague Dario "TLO" Wünsch with a score of 5-0 [20]. It subsequently achieved a ranking that was higher than 99.8 percent of all current players. The game is an example of a multi-agent problem in which several participants fight against one another for influence and resources. It is an incomplete information game since in some cases the map can only be viewed up to a point by a local camera [19], [21]. The agent has to explore his surroundings to identify the opponent's state and integrate the essential knowledge. In addition, the action space is broad and varied, and the user utilizes a point-and-click interface to pick actions from a continuous space containing around 10^8 possible outcomes. Games often continue for many thousands of frames and actions, and the player is required to make choices early with implications that may not be evident until much later in the game, leading to a diverse set of problems. Human game replays were used to teach agents throughout the first stages of their training process. Afterwards, it continued with matches against actual league opponents. To build these models, a deep neural network was used, which was trained using just original data from the game. Subsequently, the policy parameters were taught by reinforcement learning, which made use of a policy gradient approach, the Asynchronous Advantage Actor-Critic (A3C).

The same year OpenAI, a promising non-profit organization in this field, created a machine learning system, OpenAI 5 which competed and won against a team of human gamers in Dota 2. Specifically, a system of 5 neural networks took on a team of five top 0,5% of professional gamers in 3 matches of which it won the 2 [24]. The algorithm used was Proximal Policy Optimization (PPO) and a self-play strategy which means that the algorithm learns to duel against itself, gradually getting better over a long period. Dota 2 is a very complicated game in which the agent has to take thousands of actions such as moving around by clicking, casting abilities, or buying items and at the same time taking into consideration the progress of each team. On top of that, like in Starcraft the payback for those actions only arrives much later in the game while it is also an imperfect information game. Its continuous action space has high dimensionality and it is much more complex for an algorithm to deal with in comparison to Atari's, which is a relatively small discrete one. To cope with all these difficulties, it trained by playing 180 years of game experience every day and using 128.000 CPU's and 256 of the biggest GPU's available [22],[23].

Another addition to the state of the art RL performances in games happened in 2021 when Sony AI in collaboration with Polyphony Digital (PDI), and Sony Interactive Entertainment (SIE) developed GT Sophy, a revolutionary superhuman racing agent that plays Grand Turismo and managed to win top human racers [25]. Grand Turismo is a pc game first released in 1989, which emphasizes in racing simulation and is designed to emulate the look and performance of a wide variety of cars, the majority of which are authorized reproductions of actual automobiles from the real world. Undoubtedly, GT Sophy raised the bar for game AI by overcoming the difficulty of a hyper-realistic simulator. This is accomplished by mastery of real-time control of vehicles with intricate dynamics, game strategies, and split-second decision-making, all while adhering to the rules of proper game etiquette. Gran Turismo Sport was used as the training ground for GT Sophy, which was taught using cutting-edge learning algorithms and training situations created by Sony AI and utilizing unique deep reinforcement learning methods. In particular, an innovative deep RL technique that trains neural networks asynchronously and goes by the name of QR-SAC was applied. QR-SAC is an expanded form of Soft Actor-Critic. This strategy educates both a policy (an actor) that chooses an action based on the agent's observations and a value function (a critic) that evaluates the potential future rewards of each action.

In addition, a French firm called NukkAI produced an artificial intelligence that defeated eight world champions in the card game bridge. The bridge is a game in which human superiority has successfully defied the advance of computers up until this point. A card game based on the concept of taking risks, bridge is played between two teams of two players each. The victory signified a new milestone for AI since players in bridge operate with partial knowledge and are required to respond to the behavior of numerous other players. This is a situation that is far closer to human decision-making when compared to games such as Go and chess [27]. These characteristics create challenges as well as opportunities for research into AI. As a result of just having access to some relevant information, the search area available for planning is exponentially bigger than the one for completely observable scenarios. Additionally, the multi-agent feature (two players versus two, both collaborative and antagonistic) makes the branching factor higher than in games with only two players. Instead of learning by endless repetition of a single game, it first learns the rules and then refines its skills through further practice. It combines systems that are rules-based with those that use deep learning [28] [29].

In conclusion, it has been proved that machine learning can be trained in different types of games that need completely different approaches successfully. More and more technology corporations nowadays, create algorithms that compete in different types of games searching for new challenges that will finally optimize and showcase their performance. Even though the current reinforcement learning algorithms are much simpler and less capable than human intellects, with enough training data and compute resources they can solve surprising complex problems. Some of these systems may not do much more than overfitting on a very dense sampling of the problem but it is evident that they have managed to solve very complex problems and actually win the top of the professional gamers in many different occasions. To create even more powerful AI, algorithms that could use the already acquired knowledge and adapt it to new environments achieving equal or better performance should be developed.

1.3. Thesis Organization

The rest of this thesis is compiled in five more chapters, specifically from chapters two to six

In Chapter 2: The background of reinforcement learning theory is described providing the necessary techniques and equations, on which more advanced RL methods, are based.

In Chapter 3: Deep Reinforcement Learning policy gradient and actor-critic methods are presented along with the theory behind them.

In Chapter 4: Elaboration on the basic parts of the game environment and the application of the algorithms.

In Chapter 5: The results of the RL methods that were implemented are presented and discussed.

In Chapter 6: The conclusion underlines the purposes of this thesis and suggestions for further research are proposed.

2. REINFORCEMENT LEARNING LITERATURE REVIEW

2.1. Fundamentals

The *agent* can be anything that understands and acts in an environment while trying to carry out his objectives and become better at the same time. He has an objective function whose expected *value* is trying to maximize. The sum of possible decisions he may take in the environment is called *action space* and can be either discrete or continuous. In the case of continuous spaces, discretization is used usually to make the problem simpler even though the accuracy is going to decrease.

As mentioned before, the *environment* is the word where the agent lives in and chooses actions to make. More specifically, anything that the agent cannot modify deliberately is deemed as part of its environment. The sum of the information the agent takes in at a specific time t is called *state space* and can be considered the same as the *observation space* if the environment is completely observable by the agent. In some games, not all of the information that there is in the environment can be accessed by the agent.

The environment can be categorized based on different traits. It can either be *deterministic* or *stochastic*. In the first, given the present state and action, the new state can certainly be predicted while in the second, it is not always possible to be foreseen.

There is also the *single-agent* and the *multi-agent* environment. Obviously, in the *single-agent* environment only one active agent is present and is able to interact with it while in the multi agent environment there can be more than one agent dealing with the environment simultaneously. Moreover, there are discrete and continuous environments which constitute a very important distinction, since that can be a factor in determining the right algorithm to use.

Two other distinctive types of environments that are of interest are *episodic* and *sequential*, which have different temporal properties. An episodic task is one that happens in episodes. An episode can be many things, such as games or decision points, but the thing that identifies them is that for each episode the reward accumulated can be summed before a new one begins. This allows for expressing basic parameters, such as the reward, in episodes instead of time steps and finally optimizing by finding a maximum score in most of these applications. In such a setting, the agent's activities are restricted to the current episode alone and are not dependent on any actions taken in earlier episodes.

Sequential environments do not have such an endpoint since the present actions are associated with the actions that happened in earlier stages of the environment and as such cannot rewards cannot be maximized in the same way.

Finally, there are the *fully observable* and the *partially observable* Environments which were mentioned in the previous part about the implementation of RL algorithms in games. The first category corresponds to the setting where an agent has full access to his environment and its overall status at a particular moment. However, when the agent cannot always observe the whole environmental conditions of the world where he lives, the environment is called partially observable.

The activities of the agent have an effect on the condition of the environment in the future, which in turn changes the options and possibilities that are available. The action space can also be divided in discrete and continuous and it hasn't had to match the environment's type space. For example, discrete action-space can exist in a continuous environment. Correct decision-making necessitates consideration of the long-term repercussions of actions and the *reward* function is a significant factor to consider. The *reward* is given to the agent after every action he makes in the environment and can be a number, negative or positive. Generally, it shows the significance of the state the agent is in a specific time t. The agent's purpose is to maximize the total return that he gets.

2.2. Markov Decision Process (MDP)

For the purpose of addressing the reinforcement learning issue, the agent-environment interaction is represented as a MDP. That means that the next state and reward can be forecasted using only the current state and action.

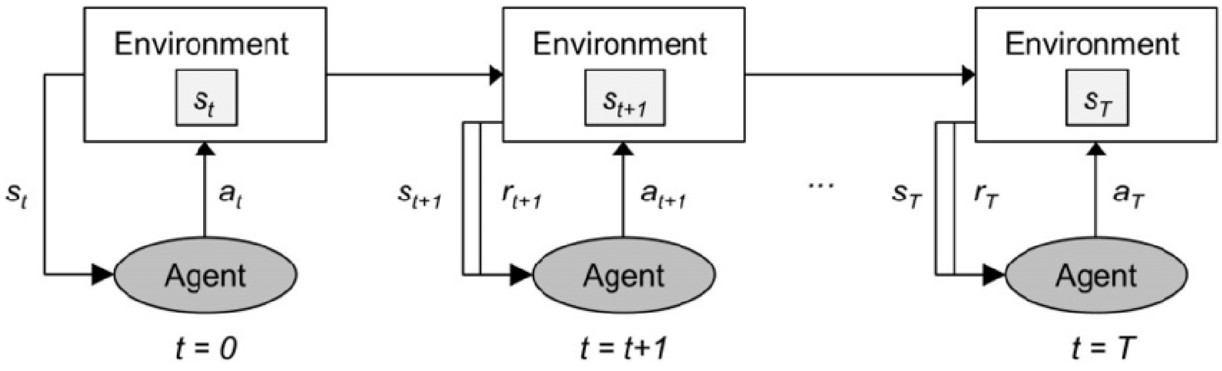


Figure 2.1: Interaction between agents and their environments in a MDP. Every step involves the agent choosing an action taken in response to environmental data and receiving a reward signal as a result of the action chosen [32].

Using the information given by the environment about the current state S_t, an action A_t is selected by the agent that will bring it with a certain probability to a new state S_{t+1}, and will earn him a reward R_{t+1} as a result of the chosen action, for every time step $t = 0, 1, \ldots,]$. The agent uses a policy that maps the possibilities of choosing each of the selectable actions from each which is symbolized by π and $\pi_t(\alpha|s)$ is the probability of $A_t = \alpha$ when $S_t = s$. If the process is finite, the agent will eventually reach the terminal state, and the series will come to an end [33]. The time sequence of events from the beginning state S_0 to the terminal state S_T is known as an episode in a finite MDP and must always occur in the following order:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2 \ldots, R_T, S_T, A_T. \tag{2.1}$$

Since the problem is defined as an MDP, each state and reward in an episode is solely influenced by the state and action pair that came before it. As a result, given a non-deterministic system, the probability of obtaining a specific reward and ending up in a specific state can be stated as

$$p(s', r \mid s, \alpha) = Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = \alpha\} \tag{2.2}$$

where s, s′, and α in (2.2) are states and actions that are contained inside the entire set of conceivable states *S,* and actions *A* respectively.

2.2.1 Rewards and expected returns

As mentioned previously, a reward is a signal that the agent receives at each time step as a gauge of how favorable it is to take a specific action. Although these rewards can be provided by the environment, they must be consciously selected in some applications.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.3}$$

The primary objective of reinforcement learning is to optimize the anticipated return, written as G_t, that is the accumulation of all the predicted rewards in certain episodes. As a result, partial rewards must be closely linked with the agent's ultimate objective. Additionally, the function (2.3) is modified using *γ,* which is the discount rate γ∈ [0, 1] that indicates the present significance of future rewards. Specifically, when it is closer to 1, future rewards gain more weight whereas immediate rewards are considered more when is closer to 0. It is a more generalized formulation that takes into account the decreasing importance of future rewards at the current time step.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

The statement above is known as the *expected discounted return* which is also appropriate for continuous tasks.

2.2.2 Value Functions

Learning a value function, which maps every possible state in the environment to its expected return when following a certain behavior, is a common strategy for solving the reinforcement learning problem. This behavior, known as policy in the literature and indicated by π, represents the agent's likelihood of picking a particular action when in a certain state, $\pi_t(\alpha/s)$. The *state-value* function can be written under policy π when using this notion.

$$v_\pi(s) = E_\pi[G_t|\, S_t = s] = E_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|\, S_t = s] \tag{2.5}$$

However, the *action-value* function (2.6), considers not only the value of existing in a specific state, but also the aftermath of taking a particular action firstly and then adhering to a specific policy π from that point forward:

$$q_\pi(s,a) = E_\pi[G_t|\, S_t = s, A_t = a] = E_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|\, S_t = s, A_t = a] \tag{2.6}$$

As previously stated, the agent's goal is to increase the expected return by doing a number of measures. For each state and action, if the real action-value function is known, the problem can be simply solved by picking the action that maximizes the expected return in each state. In that circumstance, the agent is said to be pursuing the optimal policy. As a result, any alternative policy's return must always be lower or equal to the optimal policy:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.7}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.8}$$

2.2.3 Bellman Equation

The Bellman equation describes a connection between the value of some state and state's that immediately follows it. It takes an average of all the possibilities and weights them according to the likelihood of them happening. It stipulates that the beginning state's value has to match the value of the estimated new one state adding the anticipated reward through the process. Thus, equation (2.5) can be demonstrated perpetually in the following states using come MDP properties as shown below:

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t | \, S_t = s] \\ &= E_\pi[R_{t+1} + \gamma G_{t+1} | \, S_t = s] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k \, R_{t+k+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)\big[r + \gamma E_\pi[G_{t+1} | S_{t+1} = s']\big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)[r + \gamma v_\pi(s')] \end{aligned} \qquad (2.9)$$

This equation is a series of equations, where each one corresponds to a state. Specifically, in N states, there are going to be N equations involving N unknowns. Taking into consideration that the policy is being selected so that the return is maximized the condition which is known as Bellman optimality equation for the state-value function must be:

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathrm{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | \, S_t = s] \\ &= \max_a E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \textstyle\sum_{s',r} \, p(s', r | s, a)[r + \gamma v_*(s')] \end{aligned} \qquad (2.10)$$

$$\begin{aligned} q_*(s, a) &= \mathrm{E}\left[R_{t+1} + \max_{a'} q_*(S_{t+1,} a') | \, S_t = s, A_t = a\right] \\ &= \textstyle\sum_{s',r} \, p(s', r | s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \qquad (2.11)$$

In case the state space is finite and the environment's behavior is determined, the solution to the preceding formulas can be obtained. As a result, the ideal policy can be discovered when acting greedily throughout every state from the start to the finish of the episode. It's worth noting that the greedy action in each state isn't always the one that maximizes the immediate reward in this case, but rather the end return, which is determined by the *Bellman optimality equation* in each state.

2.3. Dynamic Programming (DP)

A set of techniques known as DP can be used to compute optimal policies when a perfect model of the environment known as an MDP is given. However, traditional DP algorithms are rarely used in RL problems since they assume a perfect model and they need a lot of computations in order to be applicable, their contribution to the theoretical base of other algorithms is undeniable.

Assuming the environment can be characterized as an MDP, we assume that its state S, action A, and reward R are finite, and that its dynamics are given by a set of probabilities $\mathrm{p}(\mathrm{s}, \mathrm{r} \mid \mathrm{s}^{*}, \alpha)$. Using value functions for organizing the exploration to find effective policies is the core concept of DP and reinforcement learning in general. Thus, DP methods that can be defined like the Bellman equations are presented below [34], [35].

2.3.1. Policy Iteration

To evaluate learning, a technique called *policy evaluation* is utilized, that deals with the problem of calculating the value function for some arbitrary policy. There are two types of policy evaluation that are commonly used, which recursively help to add value to the other. One is the state evaluation v_π which is used to estimate the total value that a given state will bring following a certain policy, and is given by the Bellman equation (2.9) which is turned into an update rule, $v_k(s') \rightarrow v_{k+1}(\mathrm{s})$ and iteratively converges towards the solution, $v_\pi(s)$. In summary, $v_\pi(s)$ estimates the value of the current state, by foreseeing the next one and adding the reward that it would gain from the state transition. Additionally, there is a way of evaluating state-action pairs, q_π which is actually the inner component of the state evaluation and it takes into account only a single action [34].

$$q_\pi(s,a) = \sum_{s'} p(s' \mid s,a)\big(r(s,a,s') + \gamma v_\pi(s')\big) \tag{2.12}$$

According to the algorithm that is chosen in each case, either the state evaluation or the state-action evaluation is going to be used in the system, as they both give ways to different interpretations of the system.

On the other hand, there is the *policy Improvement* method that takes an old policy and makes a new and improved one by selecting greedy actions according to the value function of the original policy. In other words, this method determines if it would be more beneficial to change the already in use policy by moving to a new one.

In case $q_\pi(s, a)(2.12)$, the value of taking an action an in state s and then following the policy π, is greater than,$v_\pi(s)$ (2.9), then the better choice would be to once again take the action α whenever in state *s* and the new policy would be superior to the previous one. This special case is called the *policy improvement theorem*. The new greedy policy is given bellow:

$$\begin{aligned} \pi'(s) &= \arg\underset{a}{max}\, q_\pi(s, a) \\ &= \arg\underset{a}{max}\, \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \end{aligned} \tag{2.13}$$

, where $\arg\underset{a}{max}$ indicates the value of the action α that maximizes the following expression. Because the greedy policy is in agreement with the policy improvement thorium it cannot be worse than the original policy. The policy improvement theorem and the general ideal of the method can be also applicable for stochastic policies.

Combining the policy evaluation and improvement method, the *policy iteration* emerges. After improving a policy π, utilizing v_π to produce a better one π_1, another value function $v_{\pi 1}$ can be computed which can also be improved find an even better π_2. As a result, a series of policies with value functions can be generated that will be improved monotonically:

$$\pi_0 \xrightarrow{\mathrm{E}} v_\pi \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} v_{\pi 1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi^* \xrightarrow{\mathrm{E}} v_* \tag{2.14}$$

,'E' standing for *policy evaluation* and 'I' for *policy improvement*. As seen above, if the present policy is not optimal; every new policy will be an improvement of the previous one and this process continues until an optimal policy and optimal value function is found in a finite number of iterations.

```
θ: a small number, π: a deterministic policy, V ~ v*,
s.t  π~π*, V ~ v *
Function PolicyIteration is
/* Initialization
Initialize V(s) arbitrarily;
Randomly initialize policy π(s);
/* Policy-Evaluation
Δ ← 0
while Δ < θ do
    for each s ∈S do
        $v \leftarrow V(s)$:
        $V \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r+\gamma V(s')]$
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
    end
end
/* Policy Improvement
policy_stable ← True
for each s∈S do
    old_action ← $\pi(s)$
    $\pi(s) \leftarrow \arg max_a \sum_{s',r'} p(s',r|s,\alpha)[r+\gamma V(s')]$
    if old_action != $\pi(s)$ then policy-stable ← False
end
if policy_stable then
    return V ~ v*, and π~π*:
else
    go to Policy-Evaluation
end
```

Algorithm 1: Dynamic Programming - Policy Iteration

2.3.2 Value Iteration

This method avoids the application of policy evaluation during every iteration which is the case at the previous algorithm. The frequencies of policy evaluation step application can actually variate, while keeping the same convergence that is provided in the policy iteration method. Specifically, when it is applied only one time at each state, the method is called *value iteration*. It's based on the Bellman optimality equation (2.10) that is converted into an update rule. Its update is very similar to the policy evaluation update, with the only difference being that it is required that the maximum needs to be chosen for each action.

```
θ: a small number, π:  a deterministic policy,
$s.t\ \ \pi \sim \pi^*, V \sim v\ ^*$
/* Initialization
Initialize V(s) arbitrarily, except V (terminal)
V(terminal) ← 0
/* Loop Until Convergence
Δ ← 0
while Δ < θ do
    for each s ∈ S do
        $v \leftarrow$ V(s):
        V(s) ← $max_a \sum_{s',r'}\ p(s',r|s,a)[r+\gamma V(s')]$
        Δ ← $max(\Delta,|v - V(s)|)$
    end
end
/* Return Optimal Policy
return π  s.t. π(s) = ← arg $max_a \sum_{s',r'}\ p(s',r|s,\alpha)[r+\gamma V(s')]$
```

Algorithm 2: Dynamic Programming - Value Iteration

Theoretically, the algorithm ends when it converges to *v** but practically, it does so when there is not significant improvement between consecutive value functions.

2.3.3 General Policy Iteration

As previously seen in value iteration, the standard model of sequential interchange between policy evaluation and policy improvement is not always necessary to achieve converging to the optimal policy. The General Policy Iteration model represents that idea of alternating between the two policy methods without following the basic model used in policy iteration.

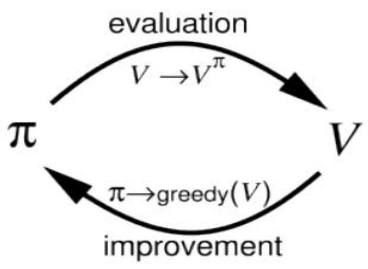


Figure 2.2: Depiction of the back and forth between the policy iteration and policy evaluation of the GPI's approach

General policy iteration can be used to describe most of the RL approaches that are used today, since they use value and policy functions that communicate in order to achieve optimality. Specifically, the latter gets better and better in accordance to the value function, while the first keeps approaching the policy's value function. Both become stable once a greedy policy in regard to its evaluation function is achieved.

2.4. Monte Carlo Methods

As mentioned before, dynamic programming algorithms require complete knowledge of the environment and the probability distributions of its transitions. The methods described in this section try to approximate the exact solution using simulations. These follow an empiric strategy based on experiences gathered by the agent when interacting with the environment.

The Monte Carlo algorithm (MC) for evaluating a given policy estimates the value function at every state by averaging the sample returns over various trajectories. In this method the value of all the states visited in the episode is updated only when the episode ends. Based on whether only the first visit or all the visits to the same state during an episode are used to calculate the average return, two algorithms with different theoretical properties are obtained. It can be proved that both methods converge to the optimal solution as the number of visits to all the states goes to infinity. The pseudo-code below corresponds to the first visit MC.

```
Initialize
    $\pi \leftarrow$ policy to evaluate
    $V \leftarrow$ initialization of value function
    Returns(s) $\leftarrow$ empty list, for all $s \in S$

Repeat:
    Generate an episode using $\pi$
    For each state S appearing in the episode:
        $G \leftarrow$ sum rewards from first instance of state to end of episode
        Returns(s) $\leftarrow$ append G
        $V(s) \leftarrow$ average (Returns(s))
```

Algorithm 3: First visit MC method for policy evaluation

In this and the following algorithms the boolean variable is used to indicate whether or not the episode has ended. The method can be easily modified to calculate q_π instead of v_π. However, these methods can only be used to evaluate a certain policy but not to find the optimal one. One way to do so would be to update the policy as new information about the value of every state and action comes in. This method is known in dynamic programming as generalized policy iteration (GPI).

2.4.1 On-policy Monte Carlo

On policy methods attempt to guarantee that all regions in the state-action space are sufficiently visited by initially deploying a more relaxed policy where the probability of selecting any of the actions in a given state is greater than 0.

One such policy is called ϵ-greedy and the parameter ϵ stands for the chance of selecting a random action as opposed to the greedy one. Hence, by combining this strategy with policy evaluation and policy improvement shown in dynamic programming an algorithm that can be shown to converge to the optimal solution is obtained.

```
Randomly Initialize: $\pi$
Initialize action value function: $Q$
Allocate memory: states, rewards, returns
repeat
  Sample initial state: $s$
  while not done do
      $\alpha_{max} \leftarrow \max_\alpha Q(s, \alpha)$
      $\alpha \leftarrow$ choose $\alpha_{max}$ or random $a$ with probability $\epsilon$ and 1- $\epsilon$
      $s', r$ done $\leftarrow$ take step
      states, actions, rewards $\leftarrow$ append $s, a, r$
      $s \leftarrow s'$
  end
  for each different s - a pair in states-actions do
      G $\leftarrow$ sum rewards from first instance of state-action to end of
      episode
      returns$(s, a) \leftarrow$ append G
      $Q(s, \alpha) \leftarrow$ average returns$(s, \alpha)$
  end
end
```

Algorithm 4: On-policy ϵ-greedy MC method for control

2.4.2 Off-policy Monte Carlo

Although in the previous section soft policies where introduced to allow some environment exploration instead of taking greedy action in every iteration, this had the consequence of making the method learn from a policy that was not entirely optimal. We will now present a different approach where two policies are used instead. The first one, known as behavior policy, is meant to explore and add new experiences, while the other, the target policy, uses the information provided by the first one to make improvements until converging to an optimal strategy.

A key concept in off-policy algorithms is importance sampling. This is used to correct for the fact that the data generated to update our target policy comes from a different distribution returns of the target policy.

$$V(s) = \frac{\sum_{t \in T(s)} \rho_{t:T(t)-1} G_t}{T(s) \vee} \tag{2.15}$$

, $\mathrm{T}(s)$ is the group of all the time steps in which state s has been visited. Equations 2.13 and 2.14 are the ordinary and weighted importance sampling estimators of V(S) respectively.

The $\rho_{t:T} - 1$ term is the importance sampling ratio which measures the relative probability that a trajectory occurs when following target and behavior policies:

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k \vee S_k) p(S_{k+1} \vee S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k \vee S_k) p(S_{k+1} \vee S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k \vee S_k)}{b(A_k \vee S_k)} \tag{2.16}$$

```
Initialize target policy π
Define behavioral soft policy b Initialize action value function: C
Set G = 0, W = 1
Allocate memory: states, rewards, returns
repeat
    Sample initial state: s
    while not done do
        action ← choose an action following $b$
        $s'$, $r$, done ← take step
        states, actions, rewards ← append $s, \alpha, r$
        $s \leftarrow Q(s, \alpha^*) s'$
    end
    for each ⟨ $s, \alpha, r$ ⟩ tuple from terminal to initial state do
        $G \leftarrow \gamma G + r$
        $C(s, \alpha) \leftarrow C(s, \alpha) + W$
        $Q(s, \alpha) \leftarrow Q(s, \alpha) + \frac{W}{C(s,\alpha)} [G - Q(s, \alpha)]$
        $\pi(s) \leftarrow \arg\max_{\alpha^* \in A} Q(s, \alpha^*)$
        $W \leftarrow W \frac{\pi(\alpha|s)}{b(\alpha|s)}$
        if W = 0 then
            break
        end
    end
end
```

Algorithm 5: Off-policy MC control method. Estimating optimal policy $\pi \approx \pi*$

Where W is a variable holding the importance sampling weights at every step of the episode and $C(S_t, A_t)$ is the cumulative sum of weights in each particular state-action pair.

The expression used for the action-value function update is equivalent to equation (2.15) and allows to reestimate the expected return incrementally as new information comes in.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)] \tag{2.17}$$

2.5. Temporal Difference methods

Even though, as discussed in the previous section, Monte Carlo techniques are proved to converge to the optimal solution they make inadequate use of the information and can perform poorly in practice. The fact that each episode has to be concluded in order to use the true sample return to make a change in our value function, can have a strong negative impact especially in situations where episodes are long or even infinite.

In contrast, temporal difference learning methods, TD, base the updates on previous estimations of the value function at subsequent states:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.18}$$

Where $R_{t+1} + \gamma V(S_{t+1})$, instead of G_t, is used as the approximation of the expected return at S_t. The TD learning alternative to algorithm 1 used for policy evaluation is outlined below

```
Define: policy π to evaluate
Initialize value function: V
repeat
    Sample initial state: s
    while not done do
        α ← choose an action following π
        s', r, done ← take step  ′
        V (s) ← V (s) + α [r + γV (s ) - V (s)]
        s ← s′
    end
end
```

Algorithm 6 : TD learning for policy evaluation

The technique applied in algorithm 4 is known as bootstrapping and it allows performing updates more frequently than the Monte Carlo methods while still converging to the optimal solution. Practically, the efficient use of data and memory in TD learning methods normally leads to faster learning. However, there is no mathematical proof supporting this empirical fact yet.

Although the methods presented in this and the following sections are based solely on the reward obtained after a single step, waiting a few more steps to do the update normally leads to faster convergence. This is the idea used by n-step TD methods which intend to combine the low bias high variance estimate of MC methods with the high bias low variance

1-step TD update. The value of S_t is updated after n steps using the intermediate rewards and the current estimate of the return at S_{t+n}.

2.5.1 On-policy Temporal Difference: Sarsa

The on-policy version of TD control algorithms is known as Sarsa. As in Monte carlo method, Generalized Policy Iteration is also applied to evaluate and improve the action choices but in this case the value function update is done online and not at the end of every episode.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.19}$$

At last, the objective is to get an approximation of the action-value function Q(s, a) at every state action pair in order to be able to make decisions on what actions to take, by comparing the expected returns in a particular state. Because this is an on policy method we need to deploy soft policies in order to allow some exploration around the state-action space. Algorithm 5 presents the Sarsa TD method for control with -greedy policy

```
Randomly initialize policy: π
Initialize action value function: Q
repeat
    Sample initial state: $s$
    $\alpha_{max} \leftarrow \max_{\alpha} Q(s, \alpha)$
    $\alpha \leftarrow$ choose $\alpha_{max}$ or random $\alpha$ with probability $\epsilon$ and 1- $\epsilon$
    while not done do
        $s', r,$ done $\leftarrow$ take step
        $\alpha_{max} \leftarrow \max_{\alpha'} Q(s', \alpha')$
        $\alpha' \leftarrow$ choose $\alpha'_{max}$ or random $a'$ with probability $\epsilon$ and 1- $\epsilon$
        $Q(s, \alpha) \leftarrow Q(s, \alpha) + \alpha[r + \gamma Q(s', \alpha') - Q(s, \alpha)]$
        $s \leftarrow s', \quad \alpha \leftarrow \alpha'$
    end
end
```

Algorithm 7: On-policy TD method for control, Sarsa

The algorithm above will certainly converge to the optimal policy if all the state- action pairs are visited an unlimited number of times and at the same time, the soft policy gradually becomes the greedy.

2.5.2 Off-policy Temporal Difference: Q-Learning

Q-learning appears as a simple but interesting off-policy variant of Sarsa. In this approach, the value function is modified after a single step by making use of the immediate reward and the current actual prediction of the greedy action in the new stage, which is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.20}$$

In this method the policy being followed is not the same as the one used in the greedy policy that updates the action value function. Even though it is labeled as an off-policy method, it differentiates from the off-policy standard mentioned in 2.4.2, because the target and behavioral policies are the same and the update is performed using the next state's greedy action. Therefore, the policy must be soft to ensure some exploration.

```
Randomly initialize policy: $\pi$
Initialize action value function: Q
repeat
    Sample initial state: $s$
    while not done do
        $\alpha_{max} \leftarrow \max_{\alpha} Q(s, \alpha)$
        $\alpha \leftarrow$ choose $\alpha_{max}$ or random $\alpha$ with probability $\epsilon$ and $1 - \epsilon$
        $s'$, r, done $\leftarrow$ take step
        $Q(s, \alpha) \leftarrow Q(s, \alpha) + \alpha [r + \gamma Q(s', \alpha') - Q(s, \alpha)]$
        $s \leftarrow s'$
    end
end
```

Algorithm 8: TD off-policy method, Q-learning

The method works very well in practice and has been proved to converge considering that all the state-action pairs are visited enough.

A straightforward strategy for achieving equilibrium between exploration and exploitation is the Epsilon-Greedy algorithm that involves making exploration and exploitation decisions in completely random order. The epsilon-greedy strategy, in which "epsilon" relates to the likelihood of selecting to explore, exploits for the most of the time while leaving a short window of opportunity to explore. When $\varepsilon=1$, it means exploration. A random number is created at each step to ascertain whether the agent will choose to engage in exploration or exploitation. If the random number is larger than e, the agent will decide to engage in exploitation as his next course of action. In such a scenario, the agent will select from the q table the action that has the greatest q value for the present condition. However, in the opposite scenario, the agent will select his next action randomly.

In general, exploration enables an agent to increase its existing knowledge about each choice he makes, which will possibly benefit him in the long-term. By increasing the precision of the anticipated action values, an agent will be able to make judgments in the future that are more in line with the available data. Exploitation, on the other hand, selects the course of the greedy-action that will result in the largest reward by taking into account the agent's most recent action-value assessments. In this way, greedy action-value estimations, might not lead to the highest reward but to a less ideal performance. During the exploration process, the agent obtains estimations of action values that have higher accuracy while during exploitation he has a chance of receiving greater rewards. The right balance between exploring and exploiting is a significant challenge of the field.

2.5.3 The λ-Return

An intermediary method between TD and Monte Carlo is to use the *n-step* return as the TD target. This samples experience from n interactions with the environment and bootstraps the remainder with the discounted estimate of the *n* state's value.

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^{n} \tilde{v}(s_{t+n}) \tag{2.21}$$

This bridges the TD and MC methods by reducing the prejudice that exists at TD. However, taking the average returns of n-steps can be a better middle ground between the two methods. That is exactly what the TD (λ) algorithm does, it summarizes the n-step updates where each is weighted proportionally to λ^{n-1} and normalized with $(1-\lambda)$ to make sure that the sum of all the weights is 1. This return is known as the λ-return (2.22).

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{2.22}$$

When λ which ranges in the space [0,1] equals 1, the resulting method is Monte Carlo while when λ = 0, it is 1-step TD. As the λ-return depends on late future rewards, a natural approximation is the truncated λ-return:

$$G_{t:t+k}^{\lambda} = (1-\lambda)\sum_{n=1}^{k-1} \lambda^{n-1} G_{t:t+n} + \lambda^{k-1} G_{t:t+k} \tag{2.23}$$

where the longest available k-step return $G_{t:t+k}$ is given a residual weight of λ^{k-1} . In order to use the truncated λ-return, k steps of experience have to be sampled before the TD update can be made. Most TD(λ) algorithms rely on the fact that (2.21) can be written as

$$G_{t:t+k}^{\lambda} = \hat{v}(s_t) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} \delta_i \tag{2.24}$$

With

$$\delta_t = R_{t+1} + \gamma \hat{v}(s_{t+1}) - \hat{v}(s_t) \tag{2.25}$$

The figure below depicts the TD(λ) algorithm and specifically how every n-step return is constructed by a number of states, actions and a reward and at the same time, how they are weighted.

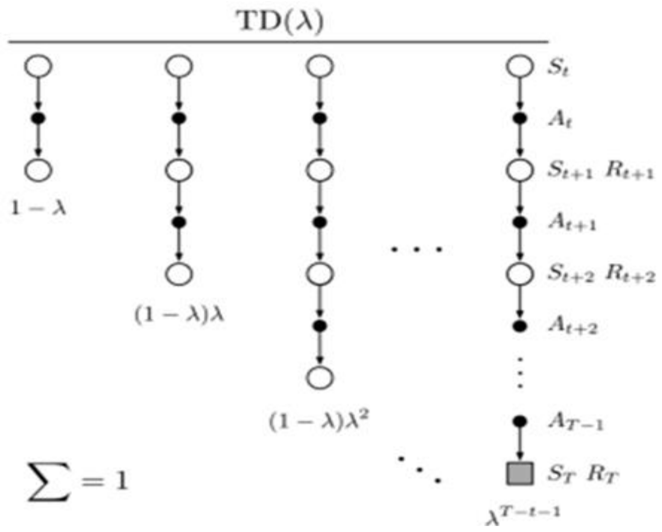


Figure 2.3: Depiction of the innerworkings of the TD (λ) method

3. DEEP REINFORCEMENT LEARNING AND ALGORITHMS

The fact that separates Deep RL methods from classic reinforcement learning is the utilization of deep artificial neural networks (ANN) that approximates the value functions and also estimates the policy parameters. ANNs are non-linear, differentiable functions which take a real-valued input $x \in R$ m and give a real-valued output $y \in R$ n :

$$f(x, w) = y, \tag{3.1}$$

, where w are the network weights. As f is differentiable it is possible to calculate the gradient according to w for some loss function and use gradient descend to optimize the network's predictions.

An ANN's basic infrastructure is a network that is made up of numerous layers, each consisting of several neurons. Prior to transmitting information to the next layer, a weighted sum of all the outputs is computed by every neuron, which is then passed via an activation function. Even more sophisticated structures exist which can be designed to gain mastery over a particular task. Deep Learning is a thriving research topic that is nowadays at the cutting edge of machine learning.

3.1 Stochastic Gradient Descent (SGD)

Before explaining the stochastic gradient descent method, it is important to describe gradient descent which is the optimization technique it was based on. Both are very popular methods in Machine Learning and constitute a base for plenty of algorithms.

A first-order iterative optimization process known as Gradient Descent is used to search for a differentiable function's local minimum value. It is an iterative process that begins at a random point on a function and works its way down the slope of that function in stages until it reaches the lowest point. This is done to discover the values that reduce the cost function as much as feasible. The update to the parameters is:

$$\theta = \theta - \eta * \nabla_{\theta} J(\theta) \tag{3.3}$$

The parameters of a model are used in order to parameterize the objective function $J(\theta)$. The value of θ is brought down to its smallest possible value, by performing an update on its parameters in the antipodes of the gradient of the objective function $\nabla_{\theta} J(\theta)$ [40]. The degree of progress that must be made to attain a (local) minimum is proportional to the learning rate η.

When used in big datasets, batch gradient descent might result in repetitive calculations because before the change of each parameter, it recalculates gradients for similar situations. SGD eliminates the need for this duplication by only requiring a single update at any given moment.

$$\theta = \theta - \eta * \nabla_{\theta} J\left(\theta, x^{(i)}, y^{(i)}\right) \tag{3.2}$$

Consequently, it can be considerably faster and in addition, it might be used to learn online (a type of learning where at each time step, the greatest predictor is updated based on data that become accessible sequentially and is used to update the best predictor for future data during every step). The regular updates with large variance that SGD uses, tend to cause significant shifts in the objective function. SGD's unpredictable nature allows it to explore new and maybe superior local minima but can also to destabilize existing ones because since SGD will continue to veer off course, convergence to the actual minimum might be hampered [40] [41]. All in all, it has been shown that SGD exhibits the same characteristic of convergence as batch gradient descent when the learning rate is progressively decreased. Accordingly, this results in an almost certain achievement of either global or local minimum for non-convex and convex functions correspondingly.

3.2 Policy Gradient

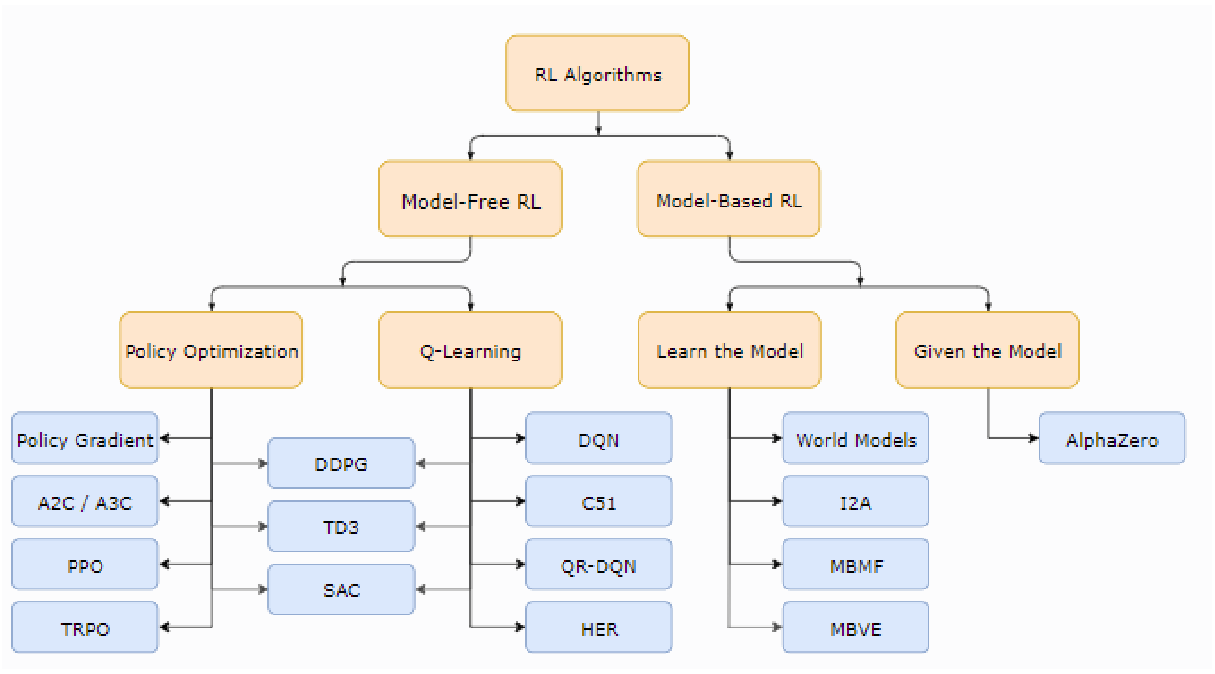


Figure 3.1: Categorization of the different RL algorithms

In the technique known as the value-function approach, all of the work put into approximating a function is put into forecasting a value function, and the action-selection policy is expressed explicitly as the "greedy" policy about the estimated values. In policy gradients, instead of estimating a value function and then utilizing it to calculate a deterministic policy, a stochastic policy is directly determined by employing an independent function approximator that has its own parameters. It is possible to model the policy using a neural network that receives state representations, outputs action selection probabilities, and policy parameters as its weights. After that, the policy parameters are adjusted proportionally to the gradient:

$$\Delta\theta \approx \alpha\, \partial\rho\, \partial\theta \qquad (3.4)$$

The parameter ρ measures the performance of the policy using the policy variables embedded in vector θ and α represents one positive-definite step during the policy implementation [38]. Given the fact that the stated goals are met, θ could typically be capable of guarantying that the performance measure will eventually converge to a local optimum policy. In contrast to the value-function method, in this technique minor adjustments in θ will not create serious changes to the policy and to the frequency of visiting a state. For any differentiable policy πθ the policy gradient is:

$$\nabla_\theta J(\theta) = E_\tau[\textstyle\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)] \tag{3.5}$$

In any regard, it is of great importance that the influence of policy shifts does not affect the distribution of states [39]. This makes the sampling process to get an approximation of the gradient quite easy. Additionally, $Q_\pi(s,a)$ is not often known and must be calculated. One strategy is to utilize the actual returns, R_τ as an estimate for each $Q_\pi(s_t,\alpha_t)$, as is seen in the following equation:

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) R(\tau)] \tag{3.6}$$

3.3 Actor-Critic Methods

Actor-Critic techniques are a kind of TD learning that is always on policy and combines the two distinct forms of Reinforcement Learning algorithms, policy-based and value-based, into a single framework. For this purpose, they have an individual memory structure that clearly represents their policy independent of the value function. As a result of its role in selecting acts and returning a probability distribution across the possible outcomes, the policy structure is referred to as "the actor." The critic is the name given to the estimated value function which is responsible for calculating the anticipated return for the agent. This is done by analyzing the agent's actions based on the current policy during a particular state [46]. In other words, the actor revises the policy distribution in the same general direction that the critic suggests (such as with policy gradients).

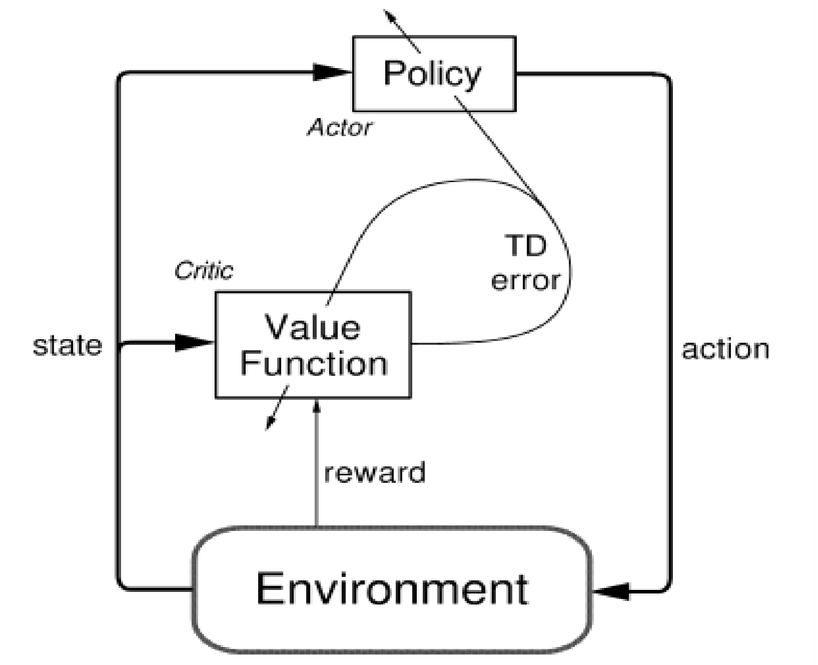


Figure 3.2: Depiction of the basic process in actor-critic methods with TD evaluation

Many actor-critic methods make use of the *Advantage function,* which approximates what the relative value of the selected action is at that stage. It is basically the subtraction of the baseline estimate from the discounted rewards, where the baseline estimate is the value function that tries to give an estimate of the final reward, at the end of the episode starting from the current one. The value keeps updating itself like a supervised problem. Simply put, the advantage estimate indicates if the action that the agent did had or worse results than expected. So if the advantage function is positive, which means the action that was taken was better than expected, the probability of choosing it again when being in the same state is increased.

$$A(s,a) = Q(s,a) - V(s) \tag{3.7}$$

There is more than one method that can be used to compute the advantage estimate which can be one of the bellow:

$$A_{\varphi}(s,a) = R(s,a) - V_{\varphi}(s) \tag{3.8}$$

$$A(s,a) = r + \gamma V(s') - V(s) \tag{3.9}$$

$$A_\varphi(s,a) = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_\varphi(s_{t+n+1}) - V_\varphi(s_t) \tag{3.10}$$

, with the first (3.8) being the Monte Carlo advantage estimate in which the actual return takes over the role of the Q-value of each action, the second (3.9) which is the TD advantage estimate and the third (3.10) which is the n-step advantage estimate [47].

3.3.1 Advantage Actor Critic (A2C)

The Advantage Actor Critic approach (A2C), is a synchronous, deterministic implementation that serves as an option for the asynchronous policy gradient version of A3C. Thru testing, OpenAI researchers concluded that it performs better and is less cost effective.

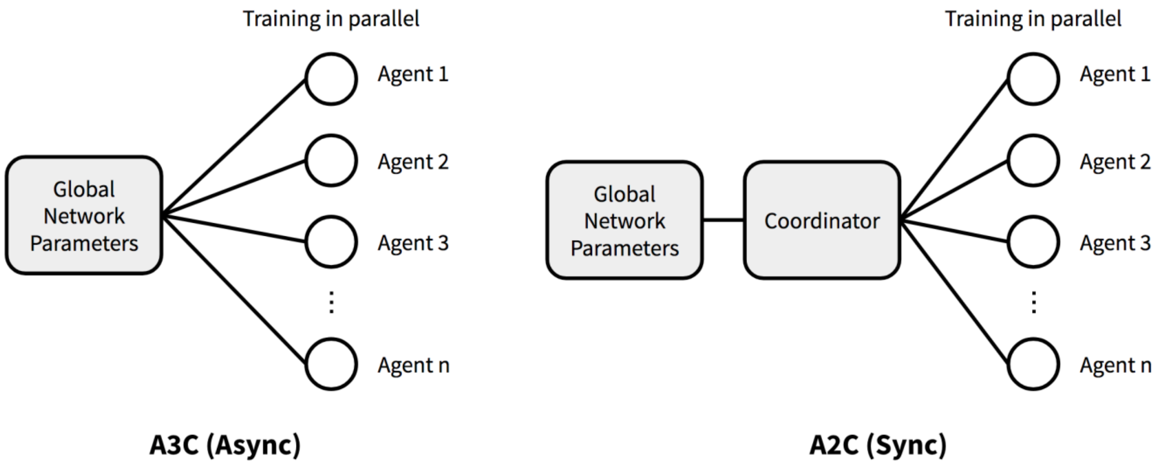


Figure 3.3: Depiction of the difference between the A3C asynchronous and the A2C synchronous algorithm

This method waits for each actor to complete their portion of the experience before it updates, and it takes an average across all of the actors [49]. The objective function that uses the n-step advantage is shown below:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\, A^{\pi_\theta}(s_t,a_t)\right] \tag{3.11}$$

$$\nabla_\theta J(\theta) = E_{s_t\sim\rho^\pi, a_t\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s_t,a_t)\left(\sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_\varphi(s_{t+n+1}) - V_\varphi(s_t)\right)\right] \tag{3.12}$$

```
Assume global shared parameter vectors $\theta$ and $\theta_v$
Initialize thread step counter $t$ ←1
Initialize episode counter $E$ ←1
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(\alpha_t | s_t, \theta)$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    for $i \in \{ t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma V_{\theta'_v}(s_t)$
        Accumulate gradients $d\theta \leftarrow d\theta + \nabla_{\theta'} log\, \pi_{\theta'}(s_t, a_t)\left(\sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V(s_{t+n+1}) - \gamma V_{\theta'_v}(s_t)\right)$
        Accumulate gradients $d\theta_v \leftarrow d\theta_v + ( R - V_{\theta'_v}(s_i))(dV_{\theta'_v}(s_i)/d\theta_v)$
    end for
    Update of $\theta, \theta_v$
    $E \leftarrow E + 1$
until $E > E_{max}$
```
Algorithm 9: Advantage Actor-Critic (A2C)

Within the domain of this technique, the term "expectation" (abbreviated as E) refers to the empirical average calculated from a finite batch of samples. The approach combines sampling and optimization in alternating steps.

$$L^{PG}(\theta) = \hat{E}_t\left[log\, \pi_\theta(a_t|s_t)\hat{A}_t\right] \tag{3.13}$$

.

Although it may seem like a good idea to execute many stages of optimization for the loss function (3.13) while using the same trajectories, doing so is not well justified, and practical evidence shows that it often results in big policy changes that can be very damaging to the performance of the algorithm [49].

3.3.2 Trust Region Policy Optimization

Although a simple policy gradient might work, it doesn't always give promising results. If a gradient descent keeps running on one batch of collective experience(on policy trait), the network's variables are then changed to such a great extent that they fall outside of the range in which these data were gathered; consequently, the advantage function will be utterly inaccurate, and the policy will be of no utility. This actor-critic method was derived in order to minimize the possibility of a false policy by using a vanilla policy gradient that takes advantage of Kullback–Leibler (KL) divergence as a constraint. The only other difference from a classic policy gradient method is that the log function is replaced with the policy ratio. These traits assure that there cannot be an extended difference between the old and the new policy [52] [53]. Thus the new algorithm is:

$$\underset{\theta}{maximize} = \hat{E}_t\left[\frac{\pi_\theta(a_t \vee s_t)}{\pi_{\theta_{\text{old}}}(a_t \vee s_t)}\hat{A}_t\right] \tag{3.14}$$

$$\text{subject to } \hat{E}_t\left[KL[\pi_{\theta_{\text{old}}}(\cdot \vee s_t), \pi_\theta(\cdot \vee s_t)]\right] \leq \delta. \tag{3.15}$$

The KL-divergence or relative entropy ensures that a new policy cannot be very different from the current one. This is a statistical distance that measures how one probability distribution is different from a second. A simple interpretation of the divergence of $\pi_{\theta_{\text{old}}}$ from π_θ is the expected excess surprise from using the second as a model when the actual distribution is $\pi_{\theta_{\text{old}}}$ [51].The loss function can be expressed as is in (3.17) where $r_t(\theta)$ is the probability ratio.

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

$$L^{CPI}(\theta) = \hat{E}_t\left[\left.\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\right)\hat{A}_t\right] = \hat{E}_t\left[r_t(\theta)\hat{A}_t\right] \tag{3.17}$$

3.3.3 Proximal Policy Optimization

The PPO algorithm is the evolution of the TRPO and is a more stable version that is easier to implement and at the same time has similar or even better performance. The existence of the constraint in TRPO, KL-divergence often disturbs the optimization of the algorithm and may result in unwanted behavior patterns during learning [55]. The PPO algorithm while being based on TRPO, manages to satisfy the same effect (old policy closer to the updated one) without using a separate constraint. The PPO objective function is:

$$L^{CLIP}(\theta) = \hat{E}_t\left[min\left(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{3.18}$$

Where ϵ is a hyperparameter that usually equals 0,2. The term, $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$, limits the probability ratio inside the space of $[1-\epsilon, 1+\epsilon]$ whenever it becomes much bigger or smaller in comparison to its previous rate. At the end, the goal is a lower constraint on the unclipped objective given the least of the clipped and unclipped objectives.

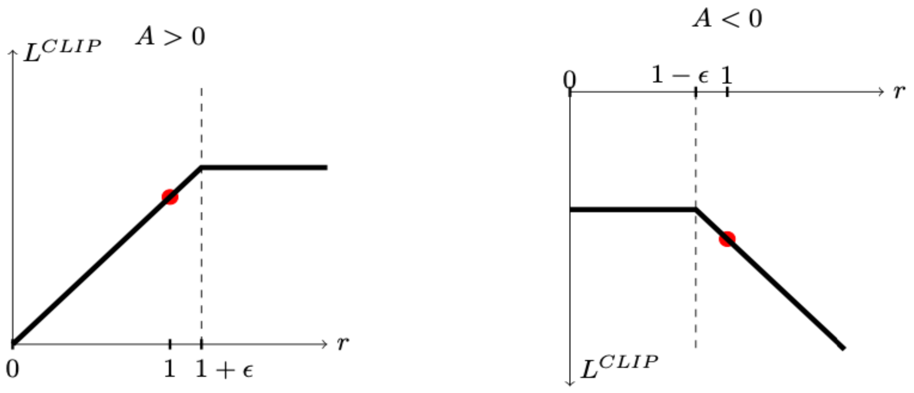


Figure 3.4: Clipping of the PPO loss function with limitation based on the advantage function.

For instance, in the left diagram where the actions yielded greater than expected return (A>0), if the return (r) gets too high at an instance, the advantage is flattened and as a result, the objective function gets limited to restrict the gradient update's influence. The most common implementation of PPO is via summing the already calculated loss function with two more as depicted below.

$$L_t^{PPO}(\theta) = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{3.19}$$

, the $c_1 L_t^{VF}(\theta)$ term is the value estimation network that estimates the average amount of discounted rewards that are expected to be received [54] [56]. A big part of the value network is shared with the policy network. The last term, $c_2 S[\pi_\theta](s_t)$ is the entropy which is in charge of ensuring that the agent does enough exploring during training as it is a measure of how unpredictable an outcome of this variable actually is. The higher the entropy, the higher the exploration and thus the random pick of actions. The c_1 and c_2 are hypermarameters of the two extra terms

```
Input: initialize policy parameters $\theta_0$, clipping threshold ε
for $k = 0, 1, 2, ..$ do
    Collect set of partial trajectories $D_K$ on policy $\pi_\kappa = \pi(\theta_\kappa)$
    Estimate advantages $\hat{A}_t^{\pi\kappa}$ using any advantage estimation algorithm
    Compute policy update
        $\theta_{\kappa+1} = \arg\max_\theta L_{\theta_\kappa}^{CLIP}(\theta)$
    by taking K steps of mini-batch SGD (via Adam), where
        $L_{\theta_\kappa}^{CLIP}(\theta) = \hat{E}_{\tau\sim\pi_\kappa}\left[\sum_{t=0}^{T}\left[min(r_t(\theta)\hat{A}_t^{\pi\kappa}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi\kappa})\right]\right]$
end for
```

Algorithm 10: PPO with Clipped Objective

Bellow the implementation of PPO based on actor critic methods is presented where T timesteps are collected by N actors in mini-batches of size M.

```
for iteration=1, 2, . . . do
    for actor=1, 2, . . . , N do
        Run policy πθold in environment for T timesteps
        Compute advantage estimates $A_1, . . . , A_T$
    end for
    Optimize surrogate L wrt θ, with K epochs and mini-batch size $M \leq NT$
    $\theta old \leftarrow \theta$
end for
```

Algorithm 11: Proximal Policy Optimization – Actor Critic edition

3.4 Convolutional Neural Networks (CNN)

The utilization of picture and audio signals as input makes the use of CNNs ideal. All the images used as an input, are represented by a 3 dimension tensor that includes the height, the width, and the color channels (RGB image).

3.4.1 Structure

CNN generally made up of three different layers, referred to as the *convolution*, the *pooling* and also the *fully connected layers*. The sophistication of the CNN rises with each layer, allowing it to recognize more of the picture characteristics as it progresses. The first one may be followed by any number of further convolutional layers or pooling layers, and then feature extraction can be carried out. The *fully connected layer* is the last of the structure and is responsible for mapping the characteristics extracted into the final output, for instance, classification [42].

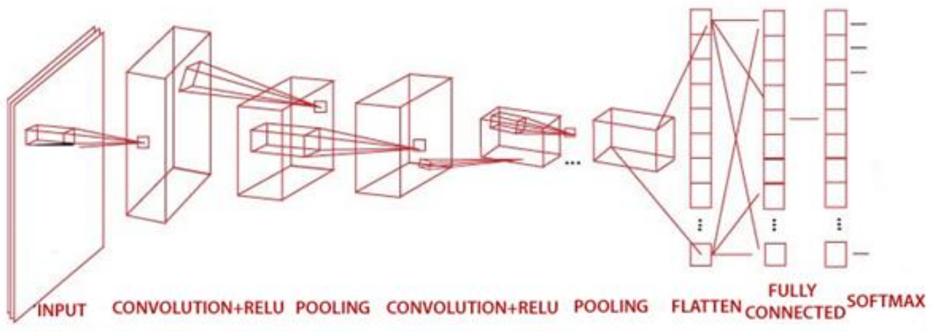


Figure 3.5: Depiction of Convolution Neural Network

The convolution layer is a specific kind of linear operation that is used for feature extraction. During this kind of process, a little range of integers known as kernels is employed from across input, which is another sequence of digits known as a tensor. This application's output, known as the feature map, is derived by performing a summary of the numbers at each position of the tensor in accordance with the kernel. This technique is carried out several times, with various kernels being applied each time. A random number of feature maps, each representing a particular part of the input tensors, are formed due through this process. Therefore, it is possible to consider the different kernels as the various feature extractors whose size and number are necessary hyperparameters. Following this, nonlinear activation is applied, usually utilizing ReLU algorithm, to the outputs of convolution after the linear process. This algorithm gives the neural network the ability to take in non-linear interactions and thus when applied to a given matrix, ReLU makes all negative values equal to zero while maintaining the other values at their original rate. Its formula is presented below:

$$g(z) = \max\{0, z\} \tag{3.20}$$

A common downsampling technique is provided by a *Pooling layer*, which decreases the in-plane dimensionality of the feature maps. This happens for two reasons. The first is to create a translation invariance on small changes and deformities which actually means that an object could be recognized even if its appearance on the image slightly changes. For instance, the ability to identify a face shown vertically, horizontally, with more or less light. The second is to restrict the number of variables that can be learned in the future. It is essential to take note that not a single one of the pooling layers has a variable that may be learned. However, during the pooling procedures, the filter-size along with stride and padding, are all examples of hyperparameters that are utilized. The max-pooling operation is by far the most common kind of pooling operation. This form of pooling, takes patches from the feature maps, returns the ones with the largest values, and throws away the remaining ones.

It is common practice to flatten the output features extracted from the final convolution or pooling layer and link them to one or more fully-connected layers (dense layers). Inside these layers, every input is linked to output by applying a learnable weight. Finally, an activation function is employed to the last of those layers. This function is not the same as the previous ones and its selection must be based on the specific requirements of each activity. Sigmoid, Softmax, and Identity are some of the most common options for the activation function of the last layer

3.4.2 Optimizers

The procedure of optimizing the parameters is carried out in order to reduce, as much as possible, the gap between the model's outputs and the ground truth labels. Algorithms and methodologies known as optimizers modify the neural network's attributes, such as its inputs and learning rate, in a way that lowers the network's overall function and minimizes losses [43]. A significant percentage of them, such as *RMSprop* and *Adam* which are two of the most popular for NN implementation, is based on gradient descent.

The RMSprop method is a gradient-based optimization strategy that is used in the training of neural networks. Geoff Hinton is the one who came up with the idea for this adaptive learning rate approach which has not been published yet. It is common for the gradients of very complicated functions, such as neural networks, to either, disappear entirely or explode as the data moves through the function. RMSprop is a stochastic learning algorithm designed specifically for use with mini-batch data.

The problem as mentioned above is addressed by the RMSprop algorithm, which normalizes the gradient by calculating a moving average of squared gradients. This normalization brings the step size (momentum) into balance by reducing it when the gradient is big enough and increasing it when it gets too little [43] [45]. The following constructs are used to generate the RMSprop update vector:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_{t^2} \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_{t+\epsilon}}} g_t \tag{3.21}$$

It uses an exponentially declining average of squared gradients to divide the learning rate rather than considering it as a hyper-parameter. Hinton recommends that γ should be set to 0.9 and that 0.001 would be an appropriate default number for the learning rate.

One equally or more significant optimization method for determining the adaptive learning rate of each variable is *Adaptive Moment Estimation (Adam)*. Not only does Adam keeps an exponentially decaying average of previous gradients like SGD with momentum m_t, but also an exponentially decaying average of previous gradients v_t like RMSprop:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{3.22}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{3.23}$$

m_t and v_t are estimates of the first two instances of each gradient correspondingly. In other words, an exponential moving average of g_t and g_t^2 is estimated and the decay rates of the moving averages are determined by the variables β_1 and β_2. It has been noticed that since m_t and v_t are both starting as vectors of zeros, they have a bias toward it, particularly at the first time steps and especially in situations when the decay rates are on the lower end [43] [44]. By producing approximations of the first and second bias-corrected moments they manage to lessen the impact of those biases. The Adam update rule is:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} m_t \tag{3.24}$$

The authors recommend the following settings for the default parameters: 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ε. Actual evidence is provided supporting that this optimizer can actually be effective and perform in the same range as other adaptive learning technique algorithms.

4. APPLICATION TO STREETFIGHTER II

4.1 Introduction to Streetfighter II

Figure 4.1: Streetfighter II Special Champion Edition for Sega-Genesis console

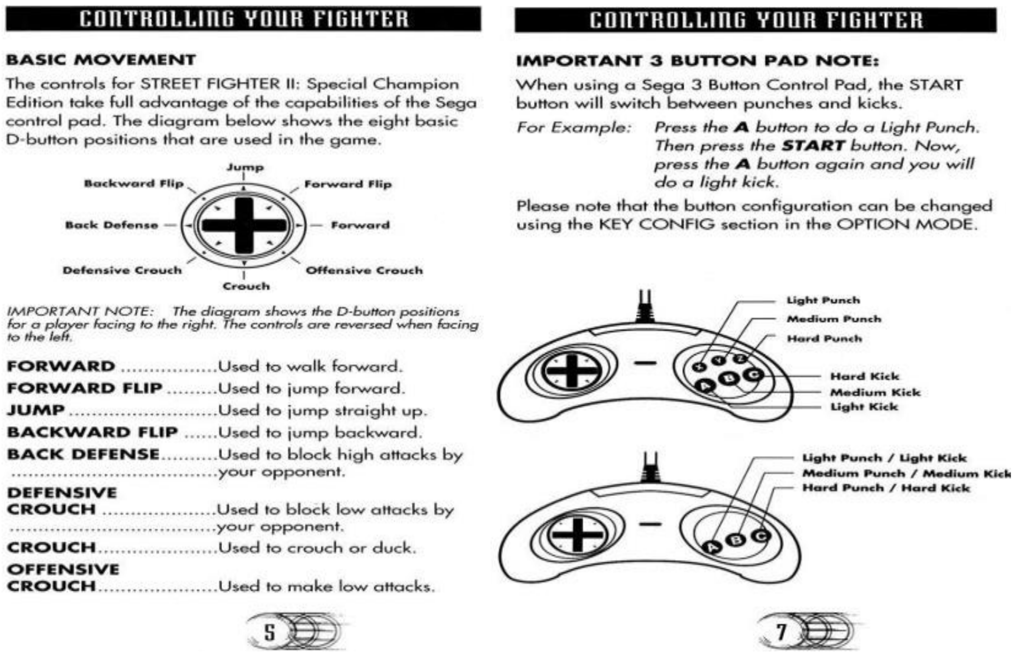
The Streetfighter II arcade fighting game was first made available in the year 1991 and is the sequel to the original game developed by Capcom [58]. It is mostly known for its well-known two player option, which demands players to compete against one another in a head-to-head setting, which kept the arcade video game industry alive which was in decline at that moment. With the release of Street Fighter II, the competitive dynamic in arcades evolved from just achieving high scores to beating other human contestants. It is regarded as one of the finest video games ever made, but it is also regarded as one of the most important fighting games that have ever been developed.

4.1.1 Game Mechanics

The player faces off against other players in one-on-one combat scenarios in a series of encounters where they compete to win two out of three games[60]. This video game offers a cast of eight different characters that the player may control. Each fighter has to face off against seven other key fighters in the single-player tournament, which is the format that will be employed for this dissertation. After that, the player has to face off against the Grand Masters, a set of four opponents which cannot be used by the player. Each game aims to reduce the opponent's vitality as quickly as possible before the timer expires. If both combatants have the same level of health remaining, this results in a "double KO," also known as a "draw game," and the contest continues until sudden death. A match may go for as many as four rounds. In the event that the conclusion of the last stage does not reveal a clear victor, the computer-controlled opponent will win by default in the single-player mode. In the same mode of the game, a bonus stage is unlocked after every third match that awards extra points. These bonus stages include a barrel-breaking stage, car-breaking stage, and drum-breaking stage. A global map is used to choose the next match site between each match.

4.1.2 Controls

The controls consist of a joystick with eight direction keys and either six or three attack buttons (which actually give access to six moves when the start button is pressed) [59]. By using the joystick that controls the movements, the fighter is able to move left and right, block, jump and crouch. There are three different punching power levels and three different kicking power levels, ranging from light to medium to heavy, on each button. Each type of move is useful under different situations in regards to the speed of the attack and the distance from the enemy.



CONTROLLING YOUR FIGHTER

BASIC MOVEMENT

The controls for STREET FIGHTER II: Special Champion Edition take full advantage of the capabilities of the Sega control pad. The diagram below shows the eight basic D-button positions that are used in the game.

Jump
Backward Flip
Forward Flip
Back Defense
Forward
Defensive Crouch
Offensive Crouch
Crouch

IMPORTANT NOTE: The diagram shows the D-button positions for a player facing to the right. The controls are reversed when facing to the left.

FORWARD	Used to walk forward.
FORWARD FLIP	Used to jump forward.
JUMP	Used to jump straight up.
BACKWARD FLIP	Used to jump backward.
BACK DEFENSE	Used to block high attacks by your opponent.
DEFENSIVE CROUCH	Used to block low attacks by your opponent.
CROUCH	Used to crouch or duck.
OFFENSIVE CROUCH	Used to make low attacks.

5

CONTROLLING YOUR FIGHTER

IMPORTANT 3 BUTTON PAD NOTE:

When using a Sega 3 Button Control Pad, the START button will switch between punches and kicks.

*For Example: Press the **A** button to do a Light Punch. Then press the **START** button. Now, press the **A** button again and you will do a light kick.*

Please note that the button configuration can be changed using the KEY CONFIG section in the OPTION MODE.

7

Figure 4.2: Controls for the Streetfighter II in Sega Genesis console

In every accessible position of the game, the fighter is able to conduct a range of basic moves, including grasping and throwing assaults. Combos have a higher impact on the player's overall performance and are accomplished by combining directional and button-based inputs. Special moves are performed by combining these two types of inputs and do more damage than the basic attack moves. Moreover, each fighter has his own “close attacks” and “special moves” that can be implemented by using a combo of moves under the right circumstances. The first category of attacks can only be accessed when the two fighters are right next to each other and consists of throws and holds while the second has no specific constraints.

4.2 Game and Environment Error! Bookmark not defined.

The programming language used for the experimenting of the algorithms used in this thesis for the StreetfighterII environment is Python and the code was implemented using Jupyter notebook. Additionally, Pytorch and Stable-baselines 3 were used for the application of the algorithms.

4.2.1 Gym Retro

OpenAI’s Gym is a Python toolkit that provides a natural cross platform and emulator compatibility for the purpose of performing RL agents that function inside the provided environments of different video games [64], [65]. Currently, there are thousands of games available to use with Gym Retro. Anyone can use the integration tool, which helps to locate memory addresses of game state variables such as score, to incorporate new ROMs into the library. The idea that Gym establishes an interface, which all of the agents and environments are required to comply with, is one of the system's many significant advantages. Consequently, the execution of an agent is not reliant on its surroundings and vice versa. Because the consistent interface will ensure that the dataset the agent obtains is virtually precisely the same in each context, it will be easier to compare and contrast the results of different environments. It is not necessary to make significant adjustments to an agent in order for it to function well in diverse settings. Because of this consistency, it is easier to compare one agent's efficiency under varying settings and the performance of many agents under the same conditions.

4.2.2 Game environment

Two different methods are described by the Gym interface. The first one is *reset*, which brings the environment to a new starting point and outputs the original observation. The use of this function is needed for the initiation of the next episode given the fact that the previous one has ended [65]. The second is *step,* which is a function that takes action as an input and returns the next observation (environment’s state) in addition to the reward that comes from carrying out the action.

The action space, the observation space, and the reward function of the environment have to be specified. In the case of Streetfighter II, the action space is Multi-binary(12), which means that the discrete actions are interpreted as a binary sequence of numbers that can be accessed thru 12 buttons (8 for movement, 3 for attacks, 1 for changing between punches and kicks). As a result, during each time step, there is a massive number of usable button combinations that correspond to 12! = 480000. However, using the ‘retro.Actions.Filtered’ command provided by Gym, only the applicable combinations are accessible preventing the training from unwanted behavior.

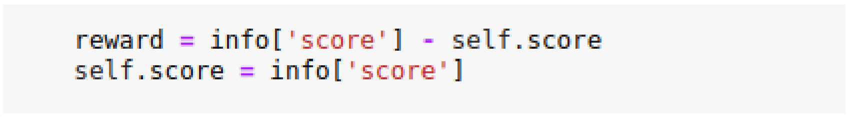
The observation space is Box [(200, 256, 3), uint8], which is interpreted as an 8-bit image with dimensions 200x256 that uses 3 RGB colors. To achieve faster training times, the preprocessing that has been implemented makes changes to the observation in three different ways. Firstly, the RGB images are subjected to greyscaling and then reframed from (200, 256, 3) to (84, 84, 1). Finally, instead of using as input the whole image, the frame_delta uses only the pixels of the image that have been subjected to changes between each time step.

4.3 Reward function

To find out the best way the agent can be trained, 3 different reward signals have been implemented. All three, use as input the information that is derived from the environment during each step. The information includes: 'enemy_matches_won', 'matches_won', 'enemy_health', 'health', 'score' and 'continue_timer'.

The parameter which includes the most data for each step is the score and is utilized for the two out of three rewards. For every fight, the player gets score points every time he inflicts damage to the opponent and every time he wins a fight. However, numerous factors influence the number of score points at every instance they are awarded. Specifically, more powerful attacks and better combos lead to even higher score points per move while if they are implemented during the end of the game. After each match that has been won, bonus points are given based on the time that is left on the game, the amount of the player's health that has been depleted along with the number of levels cleared. Every time a new fight begins, the player initially has zero points.

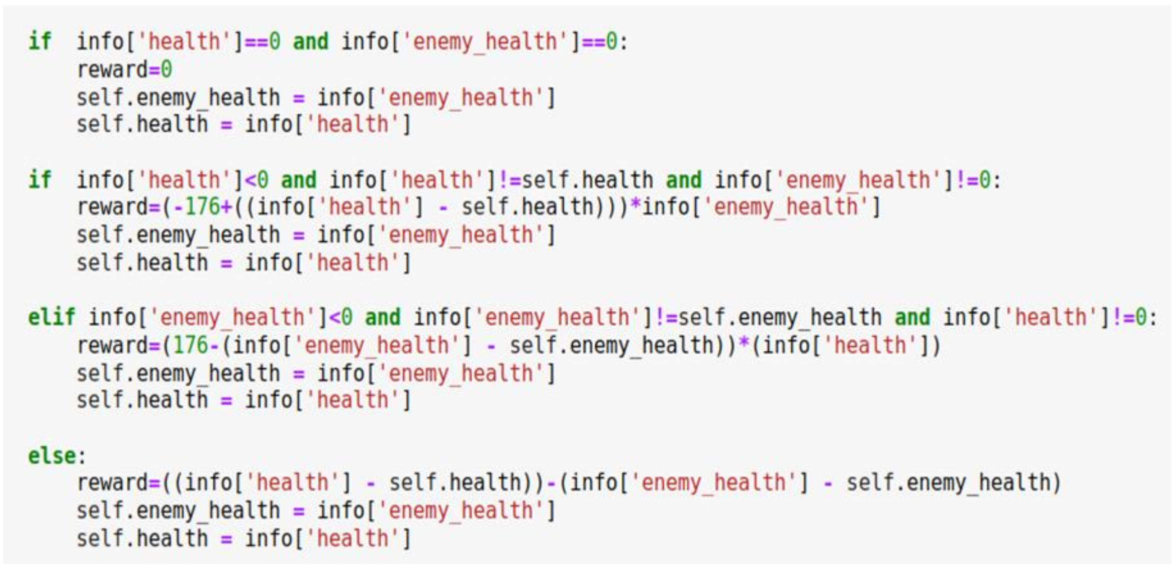
During the first tests, the simplest possible reward function was used for the training of each action. In this way, the agent only gets only positive rewards equal to the score of each time step and thus does not base his learning on his mistakes.



```
reward = info['score'] - self.score
self.score = info['score']
```

Figure 4.3: Reward Signal 1

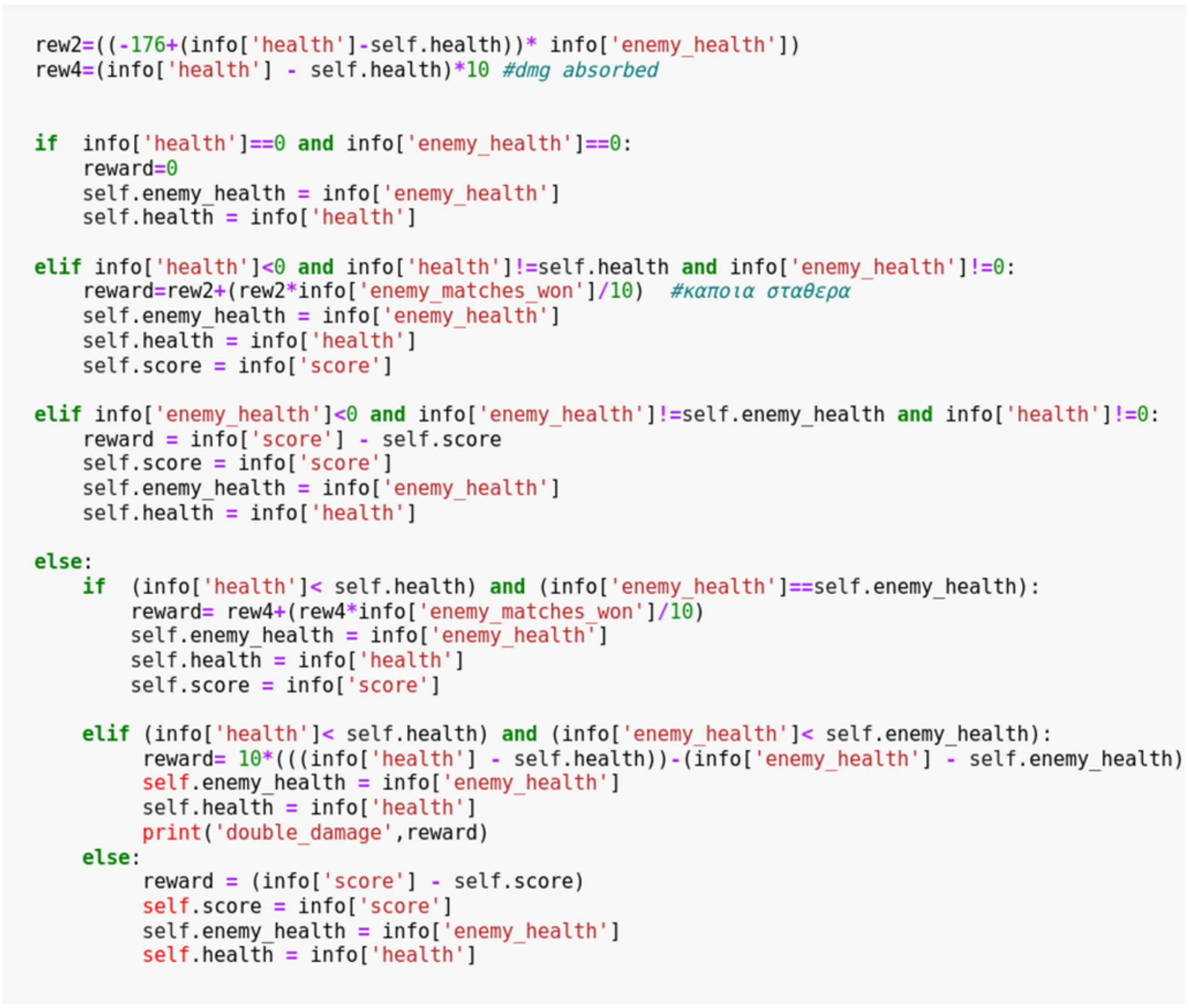
Apparently, this signal does not allow the agent to develop any defense mechanism since no negative rewards are given to the agent. Based on that idea a new reward signal was formed that emphasizes on the health of each player (‘enemy_health’, ‘health’) and does not take into account the ‘score’, which lacks information regarding the absorbing damage of the agent. In this way, positive rewards can be given for damage inflicted to the opponent and negative for the absorbed both providing equal high values. Additionally, a constant positive reward is given to the agent every time he wins a fight and a negative equal every time that he loses. Both of those rewards given at the end of each episode are significantly higher than those accumulated during the fight, in order to highlight the significance of actually winning the fight, a trait that also exists in the first reward function and is encoded in 'score'. This specific number is the maximum penalty or reward the agent can get during a fight



```
if  info['health']==0 and info['enemy_health']==0:
    reward=0
    self.enemy_health = info['enemy_health']
    self.health = info['health']

if  info['health']<0 and info['health']!=self.health and info['enemy_health']!=0:
    reward=(-176+((info['health'] - self.health)))*info['enemy_health']
    self.enemy_health = info['enemy_health']
    self.health = info['health']

elif info['enemy_health']<0 and info['enemy_health']!=self.enemy_health and info['health']!=0:
    reward=(176-(info['enemy_health'] - self.enemy_health))*(info['health'])
    self.enemy_health = info['enemy_health']
    self.health = info['health']

else:
    reward=((info['health'] - self.health))-(info['enemy_health'] - self.enemy_health)
    self.enemy_health = info['enemy_health']
    self.health = info['health']
```

Figure 4.4: Reward signal 2

Lastly, a third reward function was used that was based on both previous rewards. While the score function is used for positive rewards given during the fight and at the end of it, negative rewards equal to the absorbed damage multiplied by 10 every time the agent gets hit. In this way there is no lack of information regarding the positive values awarded and at the same time the penalties are also taken into account to transform a more complete reward. More than that, every time a fight between the agent and a new opponent concludes, the rewards given during the next fight are increased by 10%. In this way, the least amount of in game event information is lost while the agent is punished for losing life points and losing each game.



```
rew2=((-176+(info['health']-self.health))* info['enemy_health'])
rew4=(info['health'] - self.health)*10 #dmg absorbed


if  info['health']==0 and info['enemy_health']==0:
    reward=0
    self.enemy_health = info['enemy_health']
    self.health = info['health']

elif info['health']<0 and info['health']!=self.health and info['enemy_health']!=0:
    reward=rew2+(rew2*info['enemy_matches_won']/10)  #καποια σταθερα
    self.enemy_health = info['enemy_health']
    self.health = info['health']
    self.score = info['score']

elif info['enemy_health']<0 and info['enemy_health']!=self.enemy_health and info['health']!=0:
    reward = info['score'] - self.score
    self.score = info['score']
    self.enemy_health = info['enemy_health']
    self.health = info['health']

else:
    if  (info['health']< self.health) and (info['enemy_health']==self.enemy_health):
        reward= rew4+(rew4*info['enemy_matches_won']/10)
        self.enemy_health = info['enemy_health']
        self.health = info['health']
        self.score = info['score']

    elif (info['health']< self.health) and (info['enemy_health']< self.enemy_health):
         reward= 10*(((info['health'] - self.health))-(info['enemy_health'] - self.enemy_health)
         self.enemy_health = info['enemy_health']
         self.health = info['health']
         print('double_damage',reward)
    else:
         reward = (info['score'] - self.score)
         self.score = info['score']
         self.enemy_health = info['enemy_health']
         self.health = info['health']
```

Figure 4.5: Reward signal 3

4.4 Algorithms Implementation

Firstly the loss functions of each algorithm are presented according to the stable baselines default implementation [60], [61], [62].

$$L^{\pi}(\theta) = -\left[min\left(r_t(\theta)\hat{A}_t{}^{\pi\kappa}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\ \right)\right]$$
$$L^{v}(\theta) = (R_t - V(s_t))^2$$
$$L^{s}(\theta) = -\,log\,P_{\theta}(a_t|s_t)$$
$$L^{F}(\theta) = \hat{E}_t[L^{\pi}(\theta) + c_1L^{v}(\theta) - c_2L^{s}(\theta)]$$

Table 4.1: PPO loss functions shaped in the Stable Baselines Implementation

As observed bellow the only difference between the loss functions of each algorithm is the policy loss which in PPO uses a probability ratio $r_t(\theta)$ instead of the log function and also makes use of the clipping effect which was discussed in chapter 3

$$L^{\pi}(\theta) = -\left[\log\pi_{\theta}(a_t|s_t)\hat{A}_t\right]$$
$$L^{v}(\theta) = \left(R_t - V(s_t)\right)^2$$
$$L^{s}(\theta) = -\,log\,P_{\theta}(a_t|s_t)$$
$$L^{F}(\theta) = \hat{E}_t[L^{\pi}(\theta) + c_1L^{v}(\theta) - c_2L^{s}(\theta)]$$

Table 4.2: A2C loss functions shaped in the Stable Baselines Implementation

However, there are some other differences that take place during training of the two algorithms. Firstly, the PPO uses normalization of its advantage which even though it could enhance the algorithm’s execution, it could also cause significant information loss, regarding the significance of specific actions. Moreover, it does multiple gradient updates on mini-batches of the rollout data equal to 64. In contrast, only one gradient update is done in A2C on the whole batch of rollout data [26].

The algorithms PPO and A2C that were analyzed in chapter 3 were trained using the proposed hyperparameters by Stable Baselines [56], [66].

PPO Hyperparameters	Parameter Values
n_steps	2048
learning_rate	0.0003
gae_lambda	0.95
gamma	0.99
clip_range	0.2
ent_coef	0
vf_coef	0.5
max_grad_norm	0.5

A2C Hyperparameters	Parameter Values
n_steps	5
learning_rate	0.0007
gae_lambda	1
gamma	0.99
rms_prop_eps	0.00001
ent_coef	0
vf_coef	0.5
max_grad_norm	0.5

Table 4.3: Default Hyperparameters of PPO and A2C

Moreover, both of the algorithms were trained using the CNN policy that was chosen since it promises the best results when video and images are used as input. However, for the training of the A2C algorithm, 2 variants of that policy were implemented. The first one made use of Adam optimizer for the CNN policy while the used a custom variant of the RMSprop optimizer proposed by Stable Baselines[26], which is not included in Pytorch and is called RmspropTFLike. It has been reported to achieve better results when combined with the CNN policy, closer to the ones of PPO [66].

The CNN policy term stands for the class that controls all the networks used in the training process and not only the network used to predict actions [68], meaning that it is used for both policy and value prediction. The default shared CNN used in Stable Baselines is the one proposed in [67]. Its architecture consists of three convolutional layers. The first has input channels equal to 4, output channels to 32, kernel size to 8 and moving stride to 4. The next 2 layers following have output channels equal to 64 each, with strides 2 and 1 and kernel size 4 and 3 correspondingly. The hidden layers dimensionality is 512 [68], [69] .

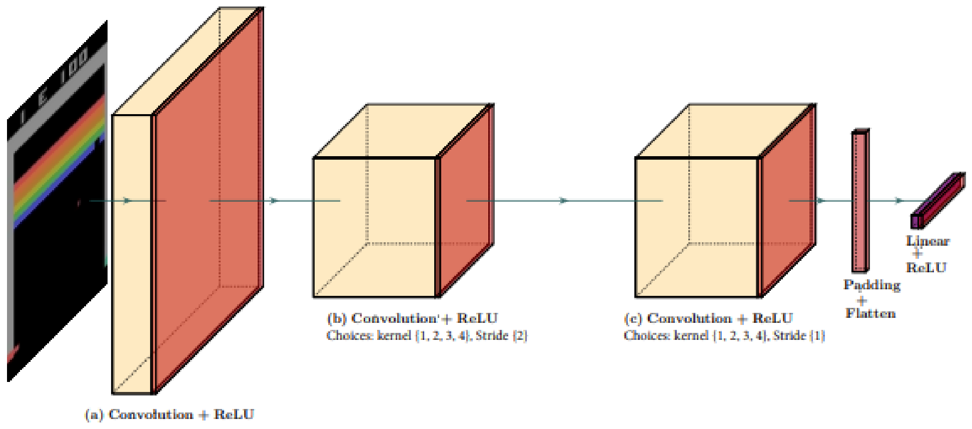


Figure 4.6: CNN used in A2C and PPO algorithms

4.5 Evaluation

Every algorithm was trained for 10 million steps and saved in batches of 10.000 time steps. The evaluation process of the agent's behavior includes quantitative and qualitative metrics. The first consists of diagrams taken from Tensorboard regarding the mean reward and episode length while also training diagrams like the entropy and value loss.

The mean reward of the models trained under different reward functions is not comparable and the mean episode length is not always indicative of progress since an agent can play more matches but beat less enemies. For instance, a model that has beaten three enemies may have played 9 games while another that has beaten four enemies could have only played 8 games if does not lose any game through the process. For that exact reason, after the different algorithms have been compared for every reward, the qualitative evaluation takes place, by inspecting renders of in-game progress. Specifically, the performance of the best algorithms is tested through 10 episodes and evaluated by calculating how many enemies are defeated during every episode.

5. RESULTS AND DISCUSSION

In this chapter the results of the training using PPO and A2C algorithms in the Streetfighter II environment are presented and compared using diagrams from Tensorboard. Each algorithm's training session needed around 10 hours to complete on a computer that consists of the following specifications.

GPU	GTX 1060 3GB VRAM
CPU	INTEL i5-4460 3.20GH
RAM	16GB DDR3
ROM	SSD 850 EVO

Table 5.1: Computer Spicifications

5.1. PPO RESULTS

Firstly the performance and training results of the PPO algorithm using the three different rewards and the default hyperparameters proposed by Stable Baselines are depicted through Tensorboard plots. Before analyzing the training diagrams of the PPO algorithm different models, it is important to understand the meaning of the metrics.

The value-loss parameter is the mean loss of the value function update. In other words, it shows if the capability the model has to successfully predict each state's value. During the training, there metric should be increasing for some time while the agent is still learning and thereafter keep decreasing until it stabilizes. Ideally, the line graph should show an upwards trend as the reward is increasing and by the time it stabilizes, should decrease and become constant near zero. The explained variance metric, estimates the percentage of the variance of the predictions made by each model. Practically, it is the difference between the expected value and the predicted value that indicates how efficiently the value network can predict the future reward. It should become as high as possible until it stabilizes. The entropy indicates the randomness of the model's decisions and for efficient training, it should slowly but constantly follow a downwards trend. However, proper tuning is required when it decreases fastly or remains the same. In the first case, the actions are chosen randomly from a subset of actions while in the second, the same actions are chosen repeatedly. The policy gradient loss is indicative of the changes that happen in policy and should oscillate until decreasing as the learning progresses.

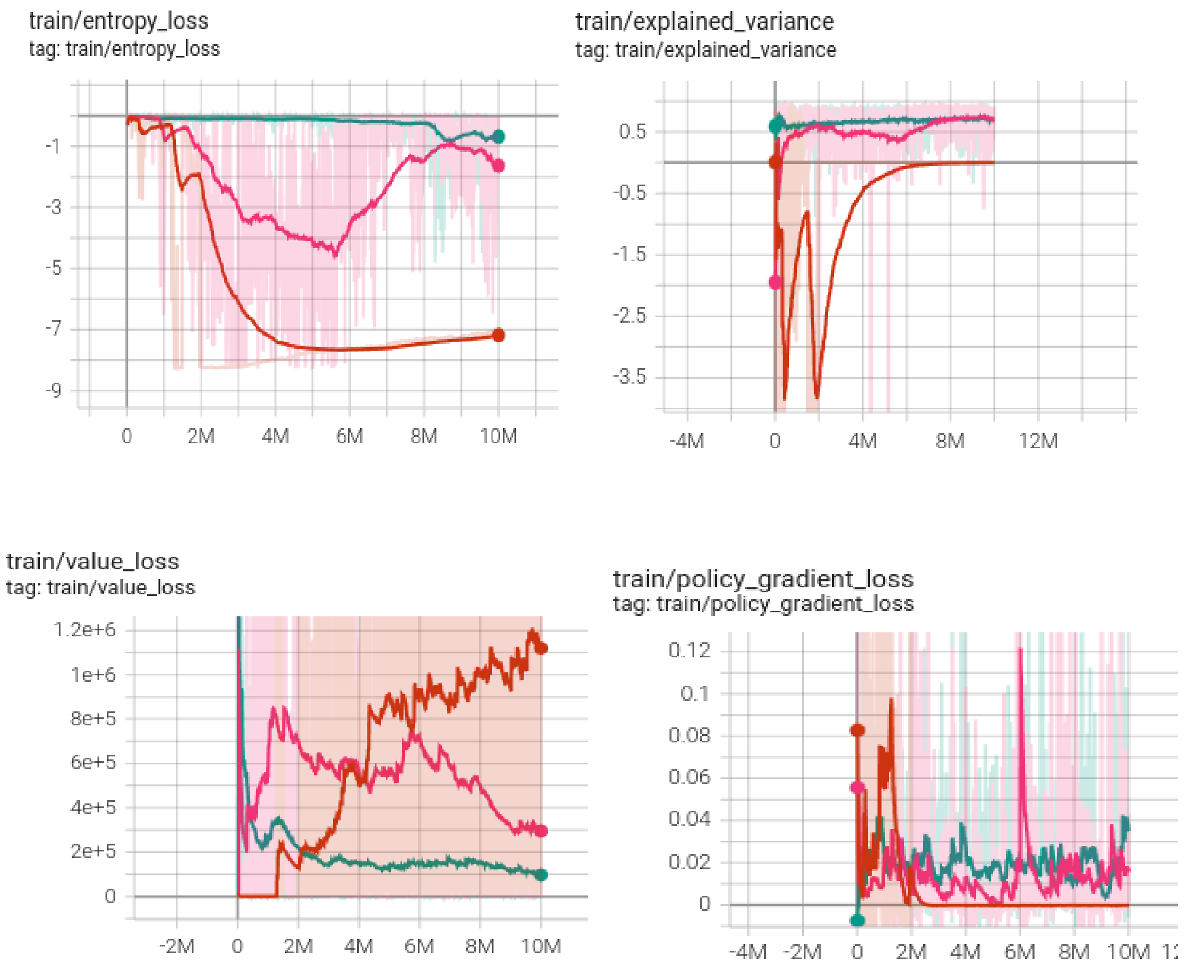


Table 5.2: Training diagrams of PPO including the entropy loss, the value loss, the explained variance and the policy loss. The training data of the algorithm trained with the first reward function is depicted by the red line, the second by the pink and the third by the green one.

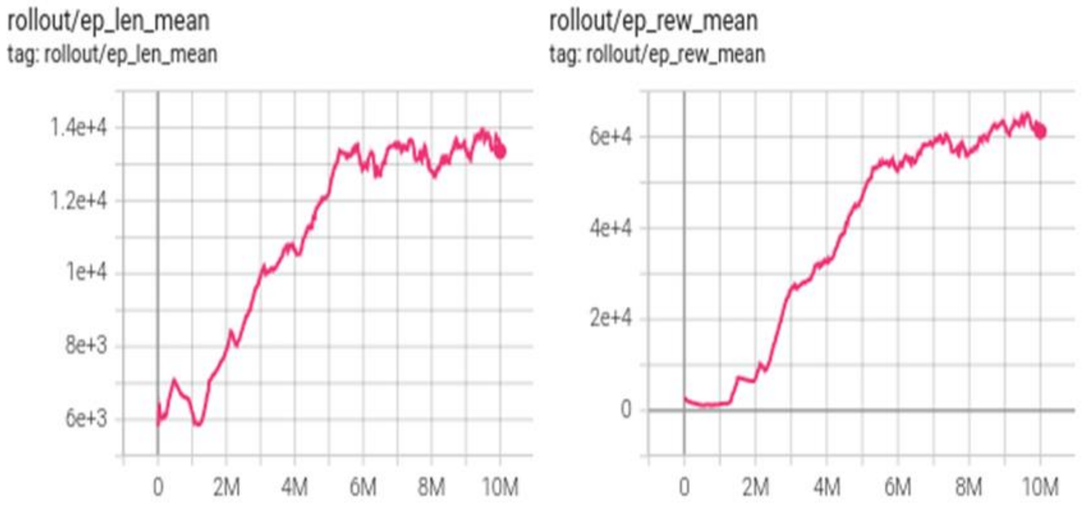


Table 5.3: Mean Reward and Mean Episode Length using PPO algorithm and reward 1

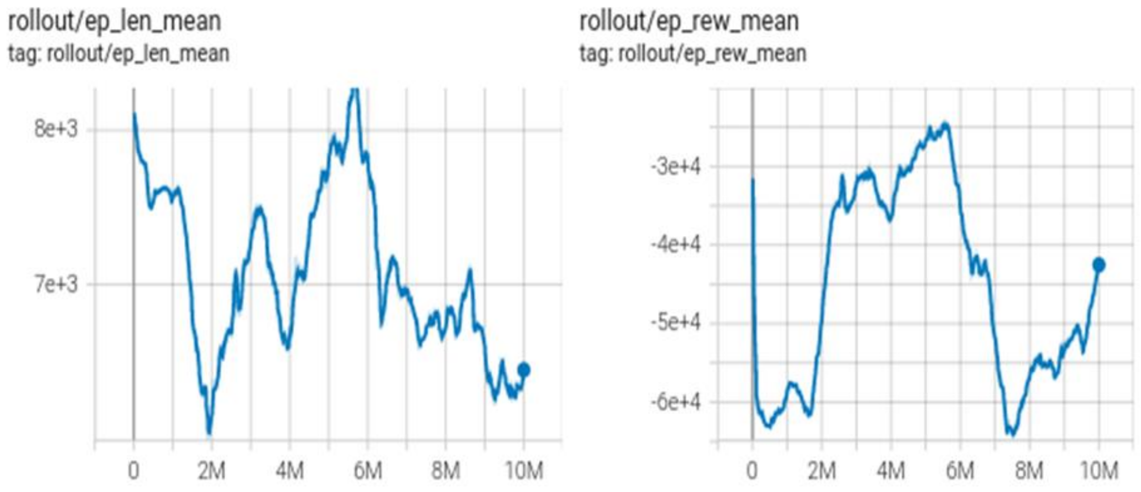
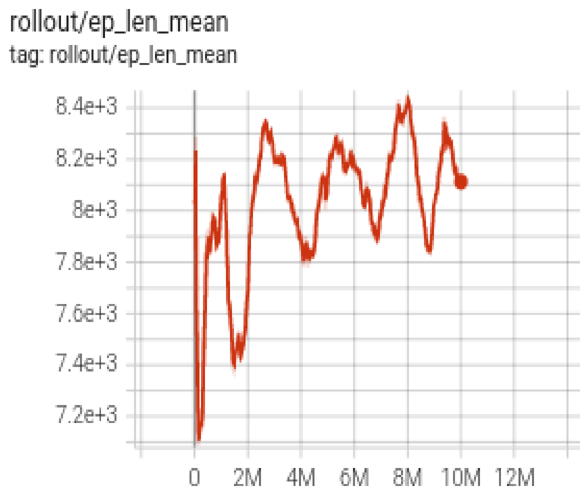


Table 5.4: Mean Reward and Mean Episode Length using PPO algorithm and reward 2



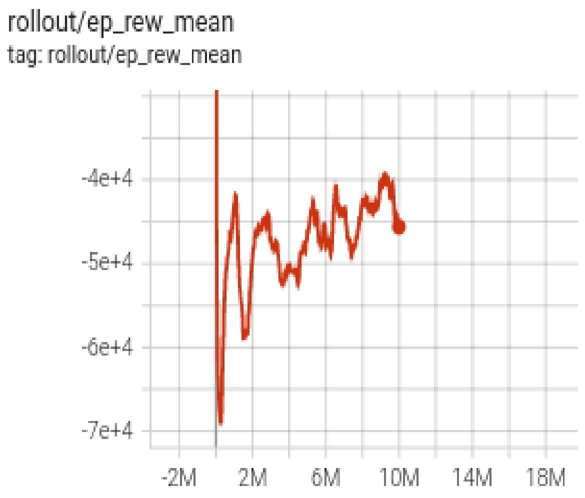


Table 5.5: Mean Reward and Mean Episode Length using PPO algorithm and reward 3

As discussed in paragraph 4.5, comparing directly the results of these algorithms is pointless since the rewards are different. The results of the model trained with reward 1 are controversial since they show good performance but inefficient training. The mean reward shows a constant rise and the mean episode length also follows the same trend while also being the highest of the models presented. However, the value loss instead of decreasing as the reward increases, it keeps surging. More than that, the entropy shows an early decline while the policy loss and the explained variance have plateaued near 0. It is thus comprehended that the model is unable to successfully predict the next rewards and at the same time it does not sufficiently explore different actions. The models trained with rewards 2 and 3 seem to play for significantly less time even though their training is smoother and more in line with the behavior that is preferred. However, unless the results are compared to the A2C ones and the in-game behavior inspected, we cannot be sure of the models' performance.

5.2. A2C Results

In this section, the same reward functions are tested using A2C algorithm and CNN policy that either uses Adam or Rms optimizer. In the first part, the algorithm is trained using the Adam optimizer for the actor-critic policy, while at the second; a custom RMS optimizer suggested by stable-baselines is used, called RMSpropTFLike. For continence the A2C model that used Adam optimizer for its policy network is referred to as ‘A2C.1’ while the one that used Rms is referred to as ‘A2C.2’.

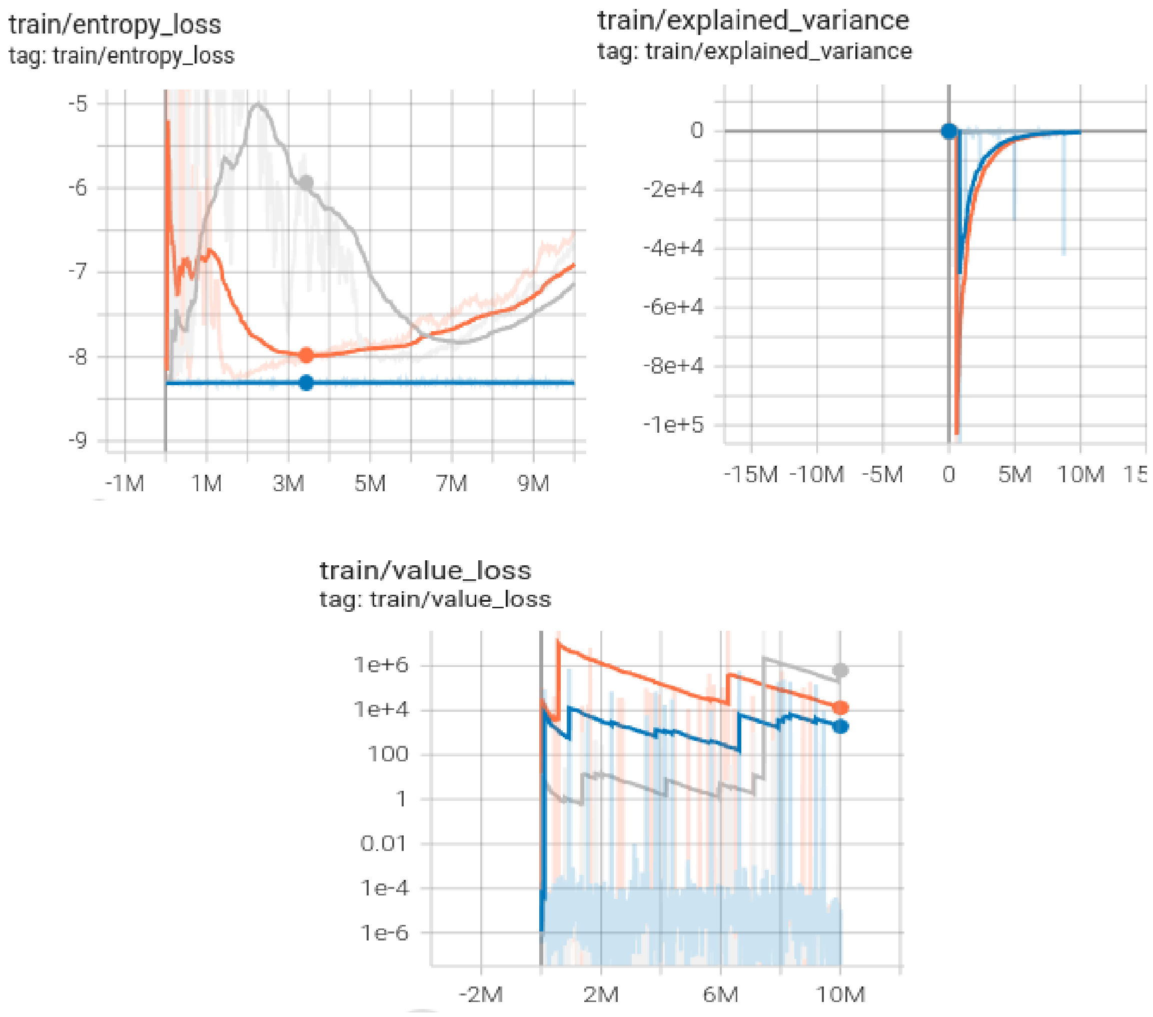


Table 5.6: Training diagrams of entropy loss, value loss and explained variance for the A2C.1 models. The training data for the reward 1 model is depicted by the blue line, for the reward 2 by the grey and for the reward 3 by the orange

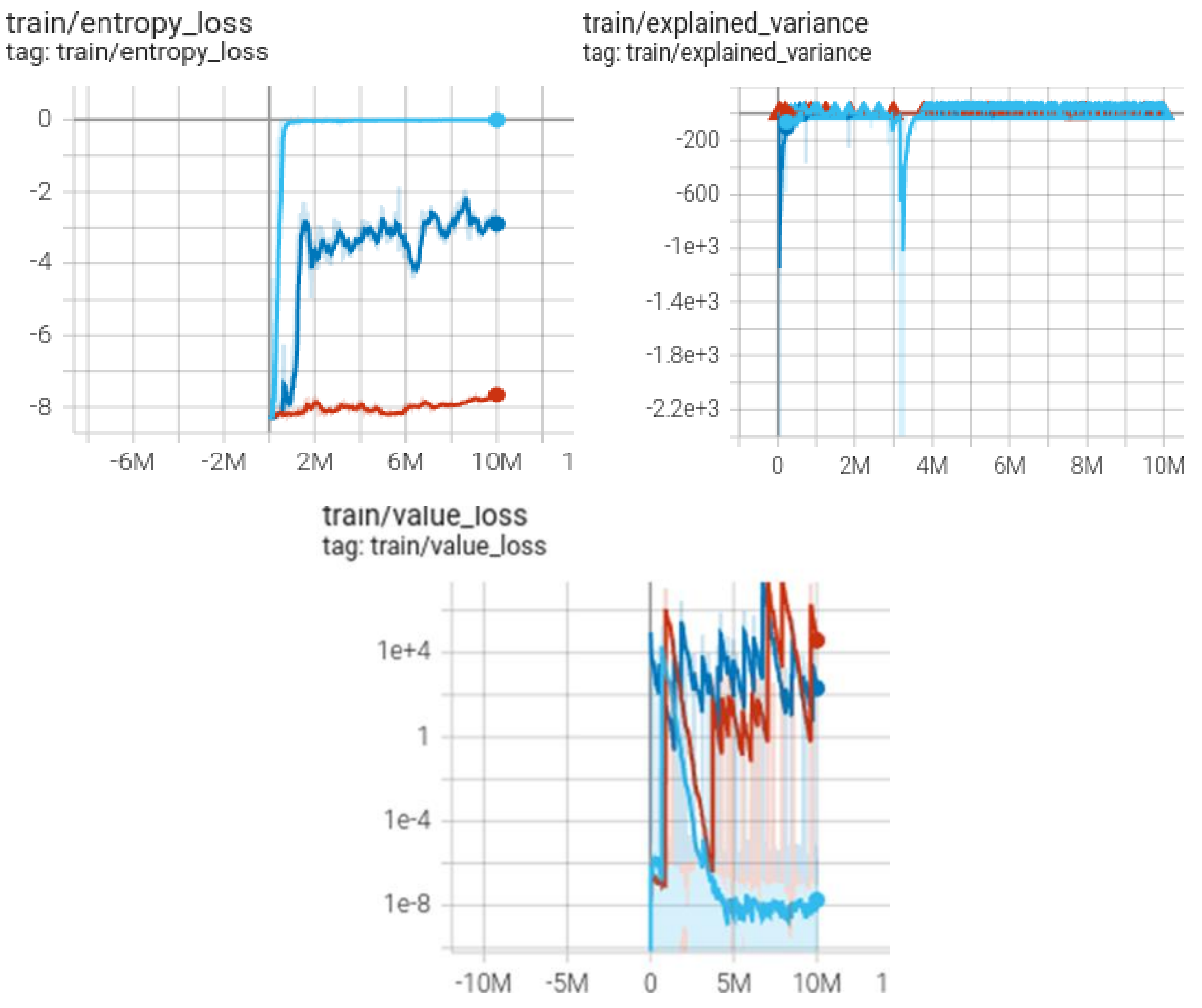


Table 5.7: Training diagrams of entropy loss, value loss, and explained variance for the A2C.2 models. The training data for the reward 1 model is depicted by the light blue line, for the reward 2 by the red and the reward 3 by the dark blue line.

The training graphs above illustrate a much worse behavior than the one provided by the PPO models. The A2C training shows a lack of exploration and ability to predict the future rewards, with models trained on reward 1 showing the worst performance.

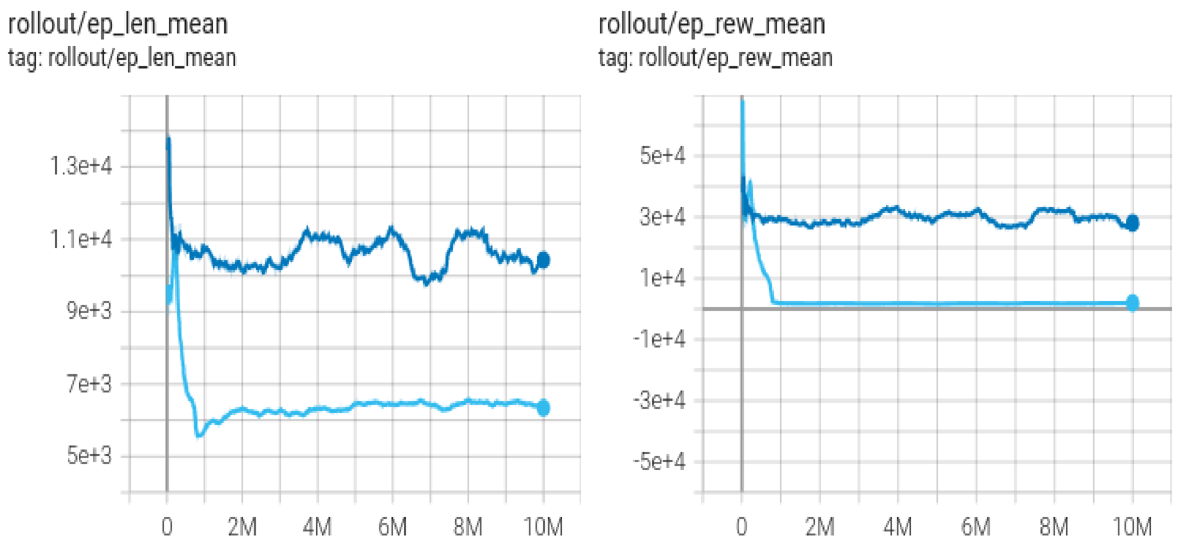


Table 5.8: The A2C.1 is depicted by the dark blue line while the A2C.2 by the light blue and both have provided underwhelming performance.

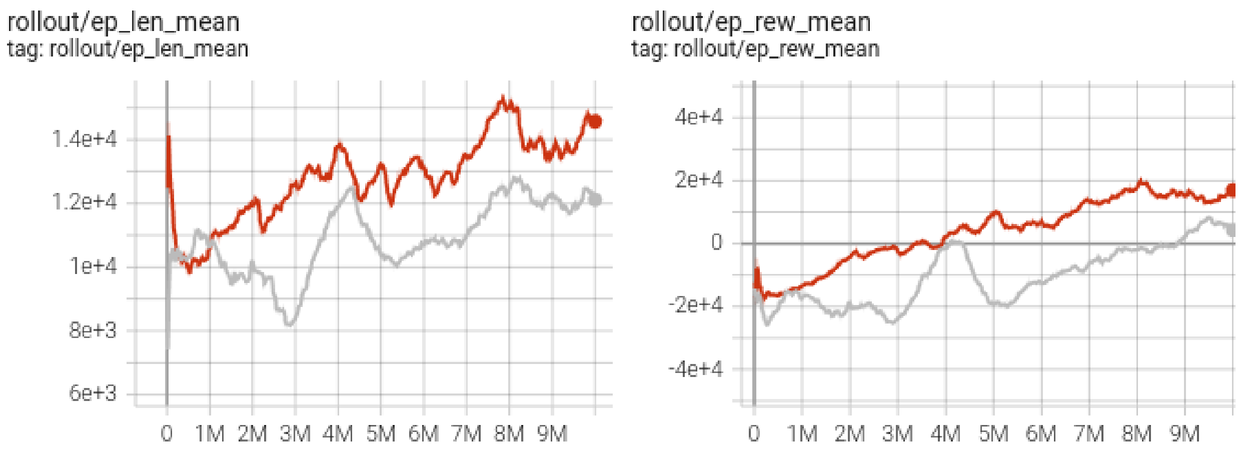


Table 5.9: Using reward function 2, A2C.2 which is depicted by the pink line, provides better statistics both in episode length and mean reward keeping an upwards trend with fewer fluctuations that when A2C.1.

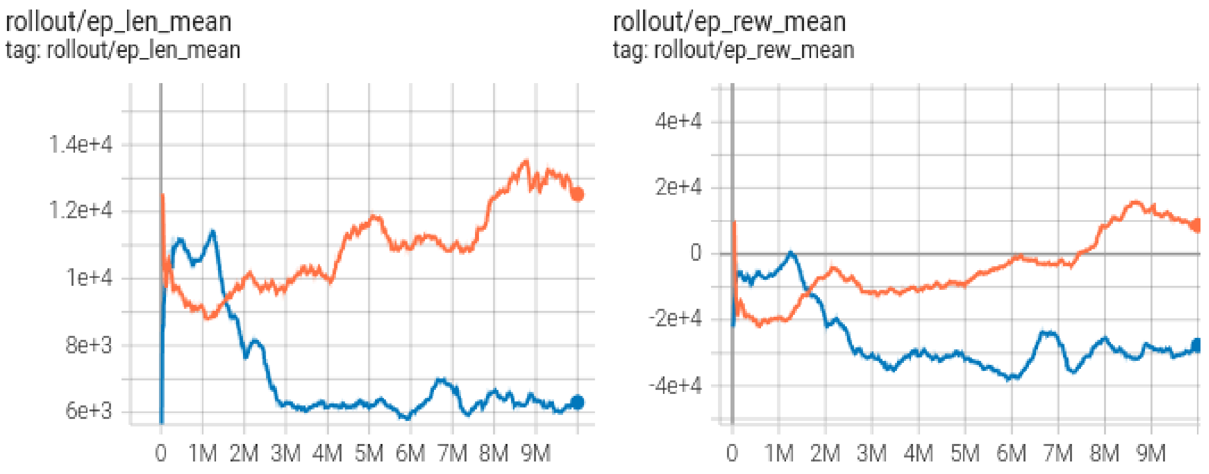


Table 5.10: Using reward function 3, A2C.1 depicted by the orange line provides better results having a steady increase in mean reward in contrast to A2C.2 depicted by the dark blue line, which shows a decline both in mean episode length and reward.

It is observed that the models using the Rms optimizer in the CNN policy network of the A2C, fail to show any progress when using reward 3 and especially 1. The only similarity between these rewards is the 'score' information which produces a big variety of results and thus, its complexity might be the reason it causes problems to the training. In contrast, the A2C models that were trained using Adam as the optimizer of the CNN policy, that in contrast to Rms, takes into account the average of the second moments of the gradient, produces much better results when the same reward functions are used. However, when reward 2 is used, which outputs less sparce results, the Rms trained model performs the best.

5.3. Comparison of the models

The first step to identify the best model is to compare the results of all the trained algorithms based on each reward function and point out the greatest for every category. Even though, extensive hyperparameter tuning of the models would change the behavior of these models and provide more successful training, for the purpose of this experimentation we want to identify the best model trained with the proposed parameters.

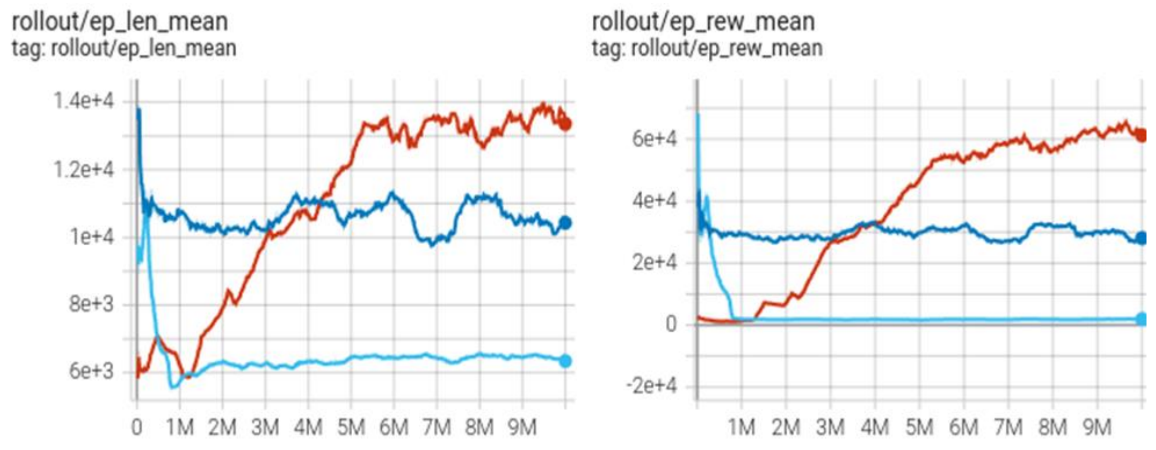


Table 5.11: Comparison of all the algorithms trained with Reward 1. The PPO model is depicted by the red line, A2C.2 by the dark blue and A2C.1 by the light blue line.

It can be clearly seen from the line graph above, that the PPO model performs the best while the A2C.2 performs the worst, plunging by the early stages of the training in both reward and episode length. Additionally, the A2C.1 shows little to no progress at all having stuck to a specific level of performance.

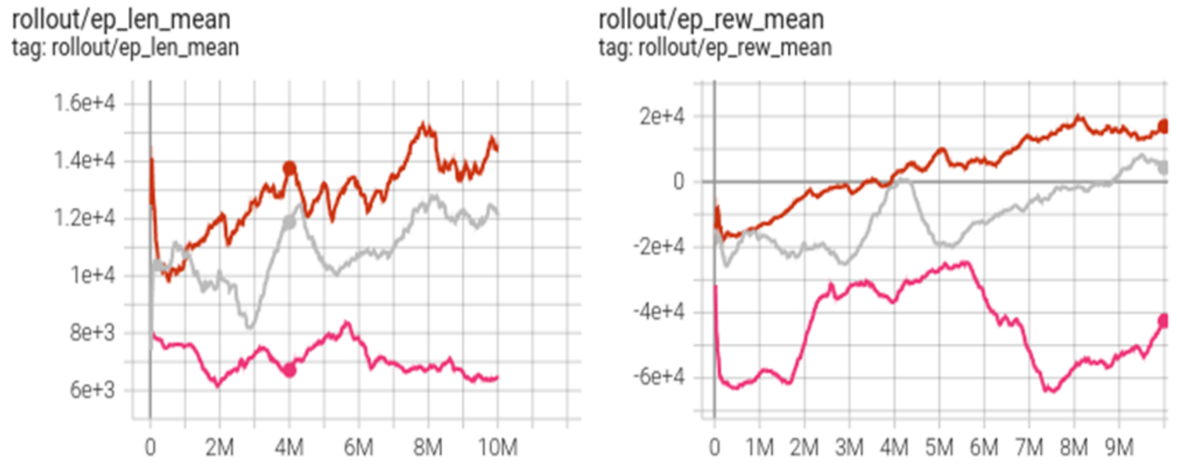


Table 5.12: Comparison of all the algorithms trained with Reward 2. The PPO model is depicted by the pink line, A2C.1 by the grey and A2C.1 by the red line.

In contrast to other comparisons, when trained with reward 2 the algorithms and especially the A2C ones, show a similar progress even though they yield different results through the process. The A2C.2 seems to perform the best yielding the highest stats and following a steady rise more in mean episode reward and less in mean episode length.

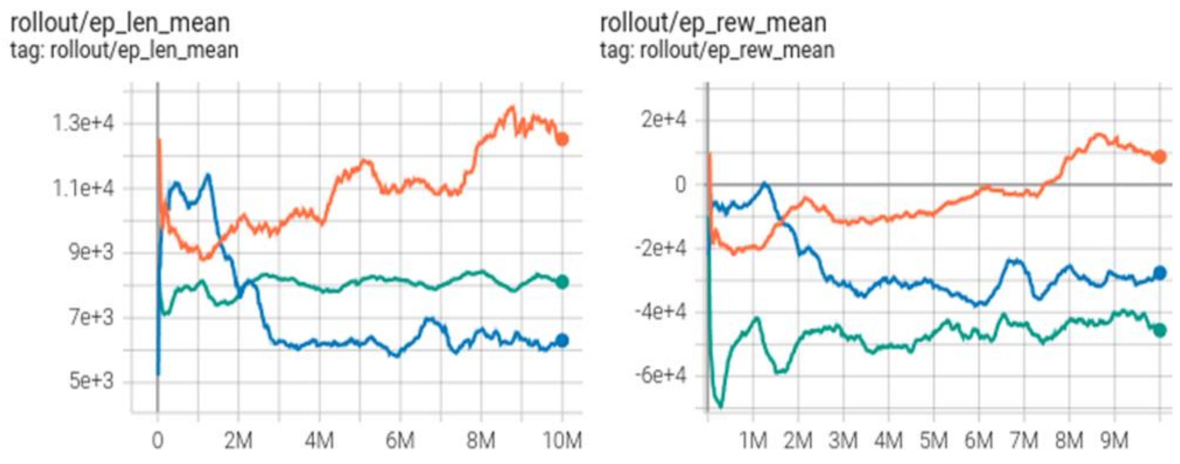


Table 5.13: Comparison between the PPO, A2C.1, and A2C.2 algorithms trained with reward 3. The PPO model is depicted by the orange line, the A2C.1 with the green and A2C.2 with the blue line.

Following the comparison of the models based on each reward, it is clearly seen that some algorithms achieved better performance than others while PPO algorithms had the steadier and most successful training sessions. In the case of using reward 1, the PPO model showed significantly better results than the A2C trained models with the A2C.2 completely failing. However, when the rewards 2 and 3 were used for the training, the results were more competitive and A2C algorithms performed the best, having A2C.2 prevail with reward 2 and A2C.1 with reward 3. As discussed in the paragraph 4.5, in order to identify the reward-algorithm pair that has the most success, their performance must be compared based on the number of enemies the agent is able to defeat. The results during the evaluation of 10 episodes are presented below:

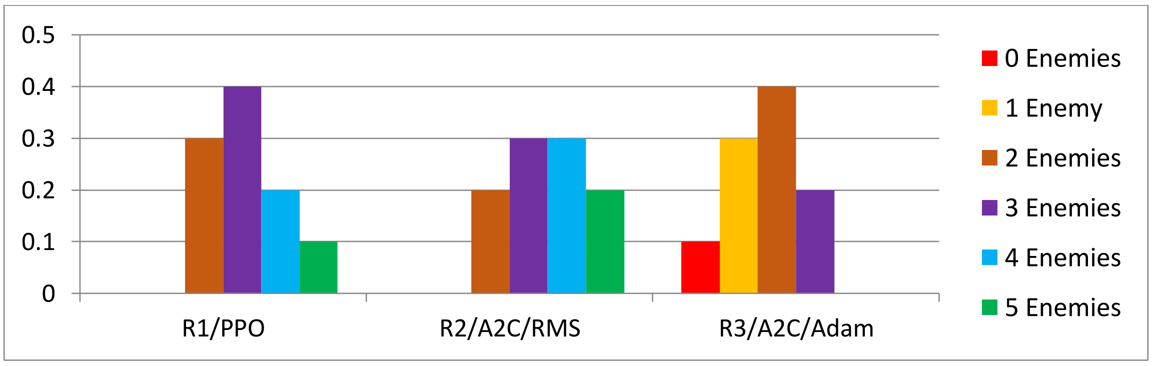


Table 5.14: Comparison between the number of enemies beaten during each episode of every algorithm's implementation

As can be clearly seen from the bar graph, the A2C algorithm with the Rms optimizer used for its policy network yields the best scores. The agent presented the highest percentages of beating 5 and 4 enemies corresponding to 20% and 30% of the trials respectively. The other 50% consists of 30% of beating 3 enemies and 20% of beating 2 enemies. None of the other 2 algorithms were able to beat more than 3 enemies 50% of the time. The second best performance is provided by the PPO trained model which beat 3 enemies 40% of the time and 2 enemies 30% of the time. In the remainder trials it beat two times 4 enemies and only once more than that. The best algorithm trained with the third reward, provided less significant results since the agent did not win more than 2 enemies for the majority of the testing.

By testing all of the three algorithms, it was observed that for every model, even though the mean reward and episode length would increase, the agent chose almost randomly specific actions included in the 'special moves' category, something that was expected when noticing the training results and the understanding the construction of the game. These moves inflict the most damage to the opponent and can also be used defensively, being equally or more effective that strictly defense moves in many cases. At the same time, there is not restriction in using them other than pressing a series of buttons with the correct timing which does not apply for a non-human player, and as a result the agent shows a big bias towards them. In more modern games of the same category as Streetfighter, special moves cannot be overused and each time they are executed, there is an energy bar that gets depleted. However, with proper hyperparameter tuning and more training this bias could possibly deteriorate.

All things considered, each algorithm performed best when paired with a specific reward, with the best being the A2C with its policy network optimized by the RMS optimizer.

6. CONCLUSION AND SUGGESTIONS FOR FURTHER STUDY

This thesis has focused on reporting the basic aspects of Reinforcement Learning, emphasizing on actor-critic methods and experimentation in the environment of the Streetfighter II game. After the presentation of the most impactful and successful implementations of Machine learning in digital and non-digital game environments, the primal concepts of the RL methods such as the Markov Decision Process. A variety of algorithms based on Dynamic Programming were then covered before introducing the characteristics of Deep Reinforcement Learning. Consequently, the content emphasized on policy gradient and actor-critic methods before introducing the proximal policy optimization (PPO) and the advantage actor-critic (A2C) algorithms that were used for the experimentation process. Last but not least, the inner workings of a CNN were discussed and some of the most favorable optimizers based on stochastic gradient descent were presented.

Following the theory presentation, the Streetfighter game and its adaption to the Gym Retro platform for the implementation of the algorithms were described. Specifically, that included the presentation of the mechanics and controls of the game along with an explanation of its action and observation space. Following, three reward signals that were developed based on different combinations of in-game information were outlined, and the thought process behind them along with their differences was explained. Lastly, the specifications of the implementation of the algorithm such as the loss functions and their hyperparameters were reported.

In the final chapter, the results of the training and testing of the algorithms were presented and discussed. It was observed, that none of the rewards shaped for this application was equally effective for the implementation of every algorithm and even the interchange of optimizers in the A2C implementations had a significant influence in the results. Indeed, it was observed that the Rms optimizer was possibly not as able to optimize the model with rewards 3 and 1, as it was with reward 1. The latter, achieved the overall best performance among all the algorithm trials, beating the most opponents in more than half of the evaluation process constituted of ten episodes.

However, many of the training sessions provided unstable data and showed inefficient learning even for models that achieved high rewards. Consequently, it would be interesting to test different parameters to the models and apply proper hyperparameter tuning in order to achieve the best performance possible. At the same time, the fact that the implementation of both algorithms included just one shared network between actor and critic, leaves the possibility of using 2 separate networks for each algorithm open for examination along with each network's architecture. Moreover, training the agent for every stage (enemy) of the game separately could make the learning process faster and more efficient, allowing the agent to progress even more in the game. Another worth exploring idea would be to apply a discrentizer and do the necessary changes to the code to allow the utilization of more algorithms that have proven to be effective, such as DQN. Last but not least, even more, reward function could be tested if there was access to more in-game information. A case in point would be to create a CNN that maps the movements of the players to output information regarding their distance at each time step. In this way, more complex rewards could be created that enforce specific strategies by exploiting the game mechanics and each enemy's characteristics.

REFERENCES

1 *Carbonell, J.G., Michalski, R.S., Mitchell, T.M. (1983) An Overview of Machine Learning In: Michalski, R.S., Carbonell, J.G., Mitchell, T.M. (eds) Machine Learning. Symbolic Computation. Springer, Berlin, Heidelberg*

2 *Richard S. Sutton, Andrew G. Barto (2014-2015) Reinforcement Learning: An Introduction*

3 *Hans J. Berliner, (1980) Backgammon Computer Program*

4 *R. Teal Witter***,** *(2021) Backgammon is Hard*

5 *Wikipedia contributors, (2022) Backgammon*

6 *Dennis DeCost, (1997) The Future of Chess-Playing Technologies and the Significance Kasparov Versus Deep Blue 7*

7 *Murray Campbell a,∗, A. Joseph Hoane Jr. b, Feng-hsiung Hsu c, (2002) Deep Blue*

8 *Wikipedia contributors, (2022) Deep Blue (chess computer)*

9 *David Silver1 *, Aja Huang1 *, Chris J. Maddison1 , Arthur Guez1 , Laurent Sifre1 , George van den Driessche1 , Julian Schrittwieser1 , Ioannis Antonoglou1 , Veda Panneershelvam1 , Marc Lanctot1 , Sander Dieleman1 , Dominik Grewe1 , John Nham2 , Nal Kalchbrenner1 , Ilya Sutskever2 , Timothy Lillicrap1 , Madeleine Leach1 , Koray Kavukcuoglu1 , Thore Graepel1 & Demis Hassabis1, (2016) Mastering the game of Go with deep neural networks and tree search*

10 *Wikipedia contributors, (2022) AlphaGo. In Wikipedia, The Free Encyclopedia*

11 *Deepmind, (2016) AlphaGo*

12 *David Silver1 *, Julian Schrittwieser1 *, Karen Simonyan1 *, Ioannis Antonoglou1 , Aja Huang1 , Arthur Guez1 , Thomas Hubert1 , Lucas Baker1 , Matthew Lai1 , Adrian Bolton1 , Yutian Chen1 , Timothy Lillicrap1 , Fan Hui1 , Laurent Sifre1 , George van den Driessche1 , Thore Graepel1 & Demis Hassabis1, (2017) Mastering the game of Go without human knowledge*

13 *Wikipedia contributors, (2022) AlphaGo Zero. In Wikipedia, The Free Encyclopedia*

14 *Deepmind, (2017) AlphaGo Zero: Starting from scratch*

15 *David Silver,1∗ Thomas Hubert,1∗ Julian Schrittwieser,1∗ Ioannis Antonoglou,1 Matthew Lai,1 Arthur Guez,1 Marc Lanctot,1 Laurent Sifre,1 Dharshan Kumaran,1 Thore Graepel,1 Timothy Lillicrap,1 Karen Simonyan,1 Demis Hassabis1, (2017) Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*

16 *Wikipedia contributors, (2022) AlphaZero. In Wikipedia, The Free Encyclopedia*

17 *Deepmind ,(2017) AlphaZero: Shedding new light on chess, shogi, and Go*

18 *Deepmind ,(2018) MuZero: Mastering Go, chess, shogi and Atari without rules*

19 *Kai Arulkumaran, Antoine Cully, Julian Togelius, (2019) AlphaStar: An Evolutionary Computation Perspective*

20 *Wikipedia contributors, (2022) AlphaStar. In Wikipedia, The Free Encyclopedia*

21 *Deepmind, (2019) AlphaStar: Mastering the real-time strategy game StarCraft II*

22 *Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung,
Przemysław "Psyho" Dębiak, Christy Dennison, David Farhi, Quirin Fischer,
Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson,
Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman,
Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang
Filip Wolski, Susan Zhang, (2019) Dota 2 with Large Scale Deep Reinforcement Learning*

23 *JOAKIM BERGDAHL, (2017) Asynchronous Advantage ActorCritic with Adam Optimization and a Layer Normalized Recurrent Network*

24 *Wikipedia contributors, (2022) OpenAI Five. In Wikipedia, The Free Encyclopedia*

25 *Peter R. Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J. Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, Leilani Gilpin, Piyush Khandelwal, Varun Kompella, HaoChih Lin, Patrick MacAlpine, Declan Oller, Takuma Seno, Craig Sherstan, Michael D. Thomure, Houmehr Aghabozorgi, Leon Barrett, Rory Douglas, Dion Whitehead, Peter Dürr, Peter Stone, Michael Spranger & Hiroaki Kitano, (2022) Outracing champion Gran Turismo drivers with deep reinforcement learning*

26 *Shengyi Huang, Anssi Kanervisto, Antonin Raffin, Weixun Wang, Santiago Ontañón, Rousslan Fernand Julien Dossa, (2022) A2C is a special case of PPO*

27 *Junkang LI,1 Solene THEPAUT, Veronique VENTOS, (2020) Reducing incompleteness in the game of Bridge using PLP*

28 *Veronique Ventos, Daniel Braun, Colin Deheeger, Jean Pierre Desmoulins, Jean Baptiste Fantun, Swann Legras, Alexis Rimbaud, Celine Rouveirol, Henry Soldano and Solene Thepaut, (2022) Construction and Elicitation of a Black Box Model in the Game of Bridge*

29 *Veronique Ventos, Yves Costel, Olivier Teytaud, Solène Thépaut Ventos, (2017) Boosting a Bridge Artificial Intelligence*

30 *PaulGarniera, Jonathan Viquerata, Jean Rabaultb, Aurélien Larchera, Alexander Kuhnlec, Elie Hachema, (2019) A review on deep reinforcement learning for fluid mechanics*

31 *Martijn van Otterlo, (2009) Markov Decision Processes: Concepts and Algorithms*

32 *Duarte, Fernando & Lau, Nuno & Pereira, Artur & Reis, Luís. (2020). A Survey of Planning and Learning in Games.*

33 *Martijn van Otterlo, (2009) Markov Decision Processes: Concepts and Algorithms*

34 *Edson Antônio Gonçalvesde Souzaa Marcelo SeidoNaganoa Gustavo Alencar Rolim, (2021), Dynamic Programming algorithms and their applications in machine scheduling: A review*

35 *Lucian Bus¸oniu, Robert Babuˇska, Bart De Schutter, and Damien Ernst, (2010) Reinforcement learning and dynamic programming using function approximators*

36 *Gerald Tesauro, David C. Gondek, Jonathan Lenchner, James Fan, John M. Prager, (2013) Analysis of Watson's Strategies for Playing Jeopardy!*

37 *David Ferrucci, Anthony Levas, Sugato Bagchi, David Gondek, Erik T.Mueller, (2012) Watson: Beyond Jeopardy!*

38 *Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour, (1999) Policy Gradient Methods for Reinforcement Learning with Function Approximation*

39 *OpenAI, (2018) Part 3: Intro to Policy Optimization*

40 *Sebastian Ruder, (2017) An overview of gradient descent optimization algorithms*

41 *Bottou, L. (2012). Stochastic Gradient Descent Tricks. Neural Networks: Tricks of the Trade*

42 *Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, Kaori Togashi, (2018) Convolutional neural networks: an overview and application in radiology*

43 *John C. Duchi, Elad Hazan, Yoram Singer (2011) Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*

44 *Diederik P. Kingma, Jimmy Lei Ba, (2015) ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*

45 *Tieleman, T. and Hinton, G. Lecture 6.5, (2012) RMSProp, COURSERA: Neural Networks for Machine Learning*

46 *Oguzhan Dogru, Kirubakaran Velswamy, Biao Huang (2021) Actor–Critic Reinforcement Learning and Application in Developing Computer-Vision-Based Interface Tracking*

47 *John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, Pieter Abbeel, (2018) HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION*

48 *Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu, (2016) Asynchronous Methods for Deep Reinforcement Learning*

49 *OpenAI Baselines authors, (2017) ACKTR & A2C*

50 *Wikipedia contributors, (2022) Kullback-Leibler divergence*

51 *James M. Joyce, (2014) Kullback-Leibler Divergence*

52 *OpenAi SpinningUp authors, (2016) Trust Region Policy Optimization*

53 *John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel, (2015) Trust Region Policy Optimization*

54 *John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, (2017) Proximal Policy Optimization Algorithms*

55 *Xingxing Liang, Yang Ma, Yanghe Feng, Zhong Liu, (2021) PTR-PPO: Proximal Policy Optimization with Prioritized Trajectory Replay*

56 *OpenAI SpinningUp authors, (2017) Proximal Policy Optimization*

57 *Nicholas Renotte, (2022) StreetFighterRL (https://github.com/nicknochnack/StreetFighterRL)*

58 *Sega Retro authors, (2019) Street Fighter II: Special Champion Edition (https://segaretro.org/Street_Fighter_II%27:_Special_Champion_Edition#)*

59 *Sega manual authors, Street Figher 2 Special Champion Edition Manual (https://manuals.sega.com/genesismini/pdf/STREET_FIGHTER_2.pdf)*

60 *Wikipedia Contributors, (2022) Street Fighter II: Champion Edition*

61 *Stable Baselines3 contributors, stable_baselines3.a2c.a2c (https://stable-baselines3.readthedocs.io/en/master/_modules/stable_baselines3/a2c/a2c.html#A2C)*

62 Stable Baselines3 contributors, stable_baselines3.ppo.ppo (https://stable-baselines3.readthedocs.io/en/master/_modules/stable_baselines3/ppo/ppo.html#PPO)

63 Stable Baselines3 contributors, stable_baselines3.common.policies (https://stable-baselines3.readthedocs.io/en/master/_modules/stable_baselines3/common/policies.html#ActorCriticCnnPolicy)

64 Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba, (2016) OpenAI

65 Gym Retro Docs authors, Gym Retro (https://retro.readthedocs.io/en/latest/#)

66 Stable Baselines3 contributors, A2C (https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html)

67 Stable Baselines3 contributors, Custom Policies (https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html)

68 N. Mazyavkina, S. Moustafa, I. Trofimov, E. Burnaev, (2021) Optimizing the Neural Architecture of Reinforcement Learning Agents

69 Stable Baselines contributors, Common Policies(https://stable-baselines.readthedocs.io/en/master/_modules/stable_baselines/common/policies.html)

Code

```
from gym import Env
from gym.spaces import Box, MultiBinary
import numpy as np
import cv2
import time
import numpy

class StreetFighter(Env):
    def __init__(self):
        super().__init__()
        self.observation_space = Box(low=0, high=255, shape=(84, 84, 1),
dtype=np.uint8)
        self.action_space = MultiBinary(12)
        self.game = retro.make(game='StreetFighterIISpecialChampionEdition-
Genesis', use_restricted_actions=retro.Actions.FILTERED)
        #self.score = 0

    def step(self, action):

        obs, reward, done, info = self.game.step(action)
        obs = self.preprocess(obs)

        frame_delta = obs


        ####--Shape reward--###

        #1st Variation

        #reward = info['score'] - self.score
        #self.score = info['score']


        #2nd Variation

        if  info['health']==0 and info['enemy_health']==0:
            reward=0
            self.enemy_health = info['enemy_health']
            self.health = info['health']

        if  info['health']<0 and info['health']!=self.health and in-
fo['enemy_health']!=0:
            reward=(-176+((info['health'] -
self.health)))*info['enemy_health']
            self.enemy_health = info['enemy_health']
            self.health = info['health']

        elif info['enemy_health']<0 and in-
fo['enemy_health']!=self.enemy_health and info['health']!=0:
```

```
            reward=(176-(info['enemy_health'] - 
self.enemy_health))*(info['health'])
            self.enemy_health = info['enemy_health']
            self.health = info['health']

        else:
            reward=((info['health'] - self.health))-(info['enemy_health'] - 
self.enemy_health)
            self.enemy_health = info['enemy_health']
            self.health = info['health']



         #3rd Variation

      #  rew2=((-176+(info['health']-self.health))* info['enemy_health'])
       # rew4=(info['health'] - self.health)*10 #dmg absorbed


        #if  info['health']==0 and info['enemy_health']==0:
         #   reward=0
          #  self.enemy_health = info['enemy_health']
            #self.health = info['health']
           #
        #elif info['health']<0 and info['health']!=self.health and in-
fo['enemy_health']!=0:
         #   reward=rew2+(rew2*info['enemy_matches_won']/20)#καποια σταθερα
          #  self.enemy_health = info['enemy_health']
           # self.health = info['health']
            #self.score = info['score']
            #
        #elif info['enemy_health']<0 and in-
fo['enemy_health']!=self.enemy_health and info['health']!=0:
         #   reward = info['score'] - self.score
          #  self.score = info['score']
           # self.enemy_health = info['enemy_health']
            #self.health = info['health']
            #
        #else:
         #   if  (info['health']< self.health) and (in-
fo['enemy_health']==self.enemy_health):
          #     self.enemy_health = info['enemy_health']
           #     self.health = info['health']
            #    self.score = info['score']

            #elif (info['health']< self.health) and (info['enemy_health']< 
self.enemy_health):
             #    reward= 10*(((info['health'] - self.health))-
(info['enemy_health'] - self.enemy_health))#den pairnei score se isopalia
              #   self.enemy_health = info['enemy_health']
               #  self.health = info['health']
                # print('double_damage',reward)
            #else:
             #    reward = (info['score'] - self.score)
              #   self.score = info['score']
               #  self.enemy_health = info['enemy_health']
                # self.health = info['health']

        return frame_delta, reward, done, info,

    def render(self, *args, **kwargs):
        self.game.render()

    def reset(self):
        self.previous_frame = np.zeros(self.game.observation_space.shape)

        # Frame delta
        obs = self.game.reset()
        obs = self.preprocess(obs)
        self.previous_frame = obs

        # Create initial variables
        self.score = 0
        self.enemy_health=0
        self.health=0

        return obs

    def preprocess(self, observation):
        gray = cv2.cvtColor(observation, cv2.COLOR_BGR2GRAY)
        resize = cv2.resize(gray, (84,84), interpolation=cv2.INTER_CUBIC)
        state = np.reshape(resize, (84,84,1))
        return state

    def close(self):
        self.game.close()

env = StreetFighter()

env.observation_space.shape

## Checking Rewards functionality
import time
obs = env.reset()
done = False
for game in range(5):
    while not done:
        if done:
            obs = env.reset()
        env.render()
        obs, reward, done, info = env.step(env.action_space.sample())
        if reward!=0:
           print(reward,info['health'],info['enemy_health'])
        time.sleep(0.01)

import torch
torch.cuda.empty_cache()

# Import A2C, PPO
from stable_baselines3 import A2C, PPO
# Import wrappers
from stable_baselines3.common.monitor import Monitor
from stable_baselines3.common.vec_env import DummyVecEnv, VecFrameStack
import os

LOG_DIR = './logs/'

from stable_baselines3.common.callbacks import BaseCallback
class TrainAndLoggingCallback(BaseCallback):

    def __init__(self, check_freq, save_path, verbose=1):
        super(TrainAndLoggingCallback, self).__init__(verbose)
        self.check_freq = check_freq
        self.save_path = save_path

    def _init_callback(self):
        if self.save_path is not None:
            os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self):
        if self.n_calls % self.check_freq == 0:
            model_path = os.path.join(self.save_path,
'best_model_{}'.format(self.n_calls))
            self.model.save(model_path)

        return True

CHECKPOINT_DIR = './train_a2c_rew2_rmsprop/'
#CHECKPOINT_DIR = './train_a2c_rew2/'
#CHECKPOINT_DIR = './train_ppo_rew2/'
callback = TrainAndLoggingCallback(check_freq=10000,
save_path=CHECKPOINT_DIR)

##Training
env.close()
env = StreetFighter()
env = Monitor(env, LOG_DIR)
env = DummyVecEnv([lambda: env])
env = VecFrameStack(env, 4, channels_order='last')

# A2C parameters
model_params = {'n_steps': 5, 'gamma': 0.99, 'gae_lambda':1, 'learn-
ing_rate': 7e-4, 'vf_coef': 0.5,'ent_coef': 0.0,'max_grad_norm':0.5,
'rms_prop_eps':1e-05 }
# PPO parameters
#model_params = {'n_steps': 2048, 'gamma': 0.99, 'learning_rate': 0.0003,
'clip_range': 0.2, 'gae_lambda': 0.95, 'ent_coef': 0.0, 'vf_coef': 0.5,
'max_grad_norm': 0.5}

model_params

import torch
from torch.optim import Optimizer
class RMSpropTF(Optimizer):
    """Implements RMSprop algorithm (TensorFlow style epsilon)
    NOTE: This is a direct cut-and-paste of PyTorch RMSprop with eps ap-
plied before sqrt
    and a few other modifications to closer match Tensorflow for matching
hyper-params.
    Noteworthy changes include:
    1. Epsilon applied inside square-root
    2. square_avg initialized to ones
    3. LR scaling of update accumulated in momentum buffer
    Proposed by G. Hinton in his
    `course
<http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf>`_
.
    The centered version first appears in `Generating Sequences
    With Recurrent Neural Networks
<https://arxiv.org/pdf/1308.0850v5.pdf>`_.
    Arguments:
        params (iterable): iterable of parameters to optimize or dicts de-
fining
```

```
            parameter groups
        lr (float, optional): learning rate (default: 1e-2)
        momentum (float, optional): momentum factor (default: 0)
        alpha (float, optional): smoothing (decay) constant (default: 0.9)
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-10)
        centered (bool, optional) : if ``True``, compute the centered
RMSProp,
            the gradient is normalized by an estimation of its variance
        weight_decay (float, optional): weight decay (L2 penalty) (default:
0)
        decoupled_decay (bool, optional): decoupled weight decay as per
https://arxiv.org/abs/1711.05101
        lr_in_momentum (bool, optional): learning rate scaling is included
in the momentum buffer
            update as per defaults in Tensorflow
    """

    def __init__(self, params, lr=1e-2, alpha=0.9, eps=1e-10,
weight_decay=0, momentum=0., centered=False,
                 decoupled_decay=False, lr_in_momentum=True):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= momentum:
            raise ValueError("Invalid momentum value: {}".format(momentum))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value:
{}".format(weight_decay))
        if not 0.0 <= alpha:
            raise ValueError("Invalid alpha value: {}".format(alpha))

        defaults = dict(
            lr=lr, momentum=momentum, alpha=alpha, eps=eps, cen-
tered=centered, weight_decay=weight_decay,
            decoupled_decay=decoupled_decay, lr_in_momentum=lr_in_momentum)
        super(RMSpropTF, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(RMSpropTF, self).__setstate__(state)
        for group in self.param_groups:
            group.setdefault('momentum', 0)
            group.setdefault('centered', False)

    @torch.no_grad()
    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the
model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad
                if grad.is_sparse:
                    raise RuntimeError('RMSprop does not support sparse 
gradients')
                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    state['square_avg'] = torch.ones_like(p)  # PyTorch in-
its to zero
                    if group['momentum'] > 0:
                        state['momentum_buffer'] = torch.zeros_like(p)
                    if group['centered']:
                        state['grad_avg'] = torch.zeros_like(p)

                square_avg = state['square_avg']
                one_minus_alpha = 1. - group['alpha']

                state['step'] += 1

                if group['weight_decay'] != 0:
                    if group['decoupled_decay']:
                        p.mul_(1. - group['lr'] * group['weight_decay'])
                    else:
                        grad = grad.add(p, alpha=group['weight_decay'])

                # Tensorflow order of ops for updating squared avg
                square_avg.add_(grad.pow(2) - square_avg, al-
pha=one_minus_alpha)
                # square_avg.mul_(alpha).addcmul_(grad, grad, value=1 - al-
pha)  # PyTorch original

                if group['centered']:
                    grad_avg = state['grad_avg']
                    grad_avg.add_(grad - grad_avg, alpha=one_minus_alpha)
                    avg = square_avg.addcmul(grad_avg, grad_avg, value=-
1).add(group['eps']).sqrt_()  # eps in sqrt
                    # grad_avg.mul_(alpha).add_(grad, alpha=1 - alpha)  # 
PyTorch original
                else:
                    avg = square_avg.add(group['eps']).sqrt_()  # eps moved 
in sqrt

                if group['momentum'] > 0:
                    buf = state['momentum_buffer']
                    # Tensorflow accumulates the LR scaling in the momentum 
buffer
                    if group['lr_in_momentum']:
                        buf.mul_(group['momentum']).addcdiv_(grad, avg, 
value=group['lr'])
                        p.add_(-buf)
                    else:
                        # PyTorch scales the param update by LR
                        buf.mul_(group['momentum']).addcdiv_(grad, avg)
                        p.add_(buf, alpha=-group['lr'])
                else:
                    p.addcdiv_(grad, avg, value=-group['lr'])

        return loss
```

```
model = A2C('CnnPolicy', env, tensorboard_log=LOG_DIR, verbose=1,
**model_params, policy_kwargs=dict(optimizer_class=RMSpropTF))
#For default Adam Optimizer: model = A2C('CnnPolicy', env, tensor-
board_log=LOG_DIR, verbose=1, **model_params)
#For PPO algorithm: model = PPO('CnnPolicy', env, tensorboard_log=LOG_DIR,
verbose=1, **model_params)

model.learn(total_timesteps=10000000, callback=callback)
env.close()

##Testing Loop
env = StreetFighter()
env = Monitor(env, LOG_DIR)
env = DummyVecEnv([lambda: env])
env = VecFrameStack(env, 4, channels_order='last')
final_reward=0
won_en=[]
for episode in range(10):
    obs = env.reset()
    done = False
    total_reward = 0
    enemies_won=0
    while not done:
        action, _ = model.predict(obs)
        obs, reward, done, info = env.step(action)
        env.render()
        #time.sleep(0.001)
        total_reward += reward
        if info['matches_won']==2:      #οτι νικησε δλδ εναν αντιπαλο
            enemies_won+=1
    won_en.append(enemies_won)

    print('Total Reward for episode {} is {} and total enemies won is
{}'.format(episode, total_reward, enemies_won))
    final_reward+=total_reward
print('Final Reward for 10 episodes is', final_reward, won_en)
```