Deep Reinforcement Learning to train agents in a multiplayer First Person Shooter: some preliminary results

Daniele Piergigli
Dept. of Computer Science
University of Milan
Italy
daniele.piergigli@studenti.unimi.it

Laura Anna Ripamonti
Dept. of Computer Science
University of Milan
Italy
laura.ripamonti@unimi.it
https://orcid.org/0000-0001-8167-7870

Dario Maggiorini
Dept. of Computer Science
University of Milan
Italy
dario.magiorini@unimi.it
https://orcid.org/0000-0002-7460-2966

Davide Gadia
Dept. of Computer Science
University of Milan
Italy
davide.gadia@unimi.it
http://orcid.org/0000-0003-4491-9150

***Abstract*—Training agents to play in contemporary multiplayer actions game is a challenging task, especially when agents are expected to cooperate in a hostile environment while performing several different actions at the same time. Nonetheless, this topic is assuming a growing importance due to the rampaging diffusion of this game genre and its related e-sports. Agents playing in a multiplayer survival first person shooter game should mimic a human player, hence they should learn how to: survive in unexplored environment, improve their combat skills, deal with unexpected events, coordinate with allies and reach a good ranking among the players community. Our aim has been to design, develop and test a preliminary solution that exploits Proximal Policy Optimization algorithms to train agents without the need of a human expert, with the final goal of creating teams composed only by artificial players.**

Keywords — machine learning, neural network, deep reinforcement learning, e-sports, video games, shooter games, artificial intelligence, first person shooter game, artificial player.

I. Introduction

Artificial Intelligence (AI) is exploited by the game industry to support gameplay, to enhance the immersivity of games and to aid game and level designers in their everyday work. Although its progresses are impressive, it struggles in keeping up the pace with the players-driven innovation regarding, among the others, how video game can be fruited. The explosion of the so-called e-sports is a clear evidence of this phenomenon. New games became huge successes in a brief amount of time, and tons of fans crowd e-sports tournaments to see their favourite champions playing. Also, the streaming of e-sports and their related events is undergoing a rampaging diffusion. The most successful games feature multiplayer cooperative interaction patterns (such as League of Legends and Fortnite), thus requiring teams of expert players that train together. It is quite easy to foresee that, in the future, team composed by human players and artificial ones could develop, provided that the AI managing the agents is good enough to produce reliable teammates. We can even conjecture that teams composed uniquely by artificial players would, sooner or later, sprang out. In such a hypothesis, training such teams by hand would become an impossible task (especially if simulations are very realistic – e.g. soccer championships).

In this context, the goal of our work has been *to design, prototype and test a preliminary solution aimed at automatically training members of a team composed entirely by agents to play a survival shooter game in an unexplored environment*. We have picked shooters game as a model for our research basing on the current most diffused e-sports. Our prototype aims at mimicking the training necessary to become good at playing a multiplayer survival First Person Shooter (FPS) game. Hence, each agent will train – alone or in couples – in a hostile (littered with enemies) 3D environment procedurally generated at runtime. Also, random events will affect the performances of each agent, to force the algorithm to adapt (thus simulating what would happen in a real-life situation, where, e.g., a player gets hurt during a match). Last but not least, to simulate the competitiveness that characterizes this type of games, a leaderboard system will rank the agents' performances. We have exploited Machine Learning (ML) techniques and Neural Networks (NNs) to train agents. The procedural generation of the environment has been configured in such a way to guarantee the creation of maps whose structure forces some kind of strategical thinking onto the agents.

The remaining of this work is organized as follows: §II briefly examines the state-of-the art in AI applied to video games, the following §III describes our methodological approach, while §IV digs into how the prototypal game has been designed and implemented. The subsequent §V tackles the problem of how the learning system has been implemented. Sections VI and VII gets into the details of some testing we have done, and, finally, §VIII derives some conclusions and suggestions for future work.

II. Artificial Intelligence and video games

AI in video games has some peculiarities (see, e.g., [1,2]), that distinguish it from the classical AI, especially because, in many cases, it must deal with real-time applications and not necessarily needs to optimize results. It can be exploited for many purposes, that can be collected in three main macro-categories: assisting gameplay, enhancing the player immersion in the game world (also simulating the psychology of the agents representing Non-Playing Characters – NPCs) and supporting the work of game and level designers. Among the most diffused AI techniques, we can count those used to procedurally generate contents (see e.g. [3,4,5,6,7]) and those

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

aimed at supporting the decision-making system of the artificial agents (see e.g. [8,9]). Machine Learning (ML) and Neural Networks (NN) have been applied to games since a long time [10], but their use has recently known a renewed interest and addresses a wide variety of topics (see e.g., [11,12,13,14]). Nonetheless, using these techniques to train agents in complex environments, with several possible concurrent actions – such as moving while aiming and firing – is quite a challenging result to achieve, as underlined by Harmer et al. [15], the authors of one among the most recent and interesting approaches to the problem. To tackle this dilemma, they adopt a Deep Reinforcement Learning architecture to train an agent to perform multiple actions while playing a First-Person Shooter (FPS) game. In particular, they exploit Imitation Learning [16,17] and Temporal Difference Reinforcement Learning. On one hand this produces effective results, but on the other it suffers of several limitations that makes it difficult to adapt it straight-away at the problem we are tackling. Actually, it requires the intervention of a human expert that plays the prototypal game they have developed in order to supply data that contribute in the training of the agent. Moreover, its main aim is to train, in a predefined fixed environment, one single agent that combats with one enemy, thus reducing a lot the complexity of the task.

III. Methodological approach

Our aim has been to teach to several artificial agents how to play appropriately in a multiplayer FPS survival game, thus simulating a human player. This implies that, beside learning to cooperate, each agent should also learn to explore the environment, to improve its aiming skill and to decrease its reaction time. To tackle this problem, we have split our study into several subsequent steps. First of all, in order to obtain a sound simulation, we have examined the characteristic shared among shooter games, so to derive their core characteristics to build a model of the game. This phase included the analysis of the game world, the player enemies' behavior and the set of skills that an artificial player would have needed to resemble a real human one, including how to simulate the incomplete perception of the game world. Our second step has been to examine the current state-of-the-art Machine Learning (ML) and Neural Network (NN) techniques to select the most appropriate solution and build a solid theoretical model for the problem. Since we had to deal with real-time decision-making with implication on the survival of the agent, we have opted for designing a learning system based on the Proximal Policy Optimization algorithm (PPO) [18], which is rooted into Deep Reinforcement Learning techniques. Then, since – for the sake of generality – we wanted to adapt our learning system to the use with games developed with state-of-the-art commercial game engines, we have dealt with the problem of creating a communication channel between the artificial intelligence managing the agent while in-game and the external learning system. This meant passing to the learning system the observations and the experiences (observations plus actions and rewards) collected by the agent while playing and, on the other way around, obtaining from the learning systems suggestions about the decisions that needed to be taken in-game. Basing on the prior phases' outcomes, we have developed a prototypal survival-shooter game, developed in C#, exploiting the Unity 3D game engine. Unity 3D offers a dedicated software development kit to implement reinforcement learning algorithms. The "brain" of the agent has been implemented in Phyton, exploiting TensorFlow, and communicates with the game via API. Once our prototype has been fully operational, we have set up several experiments to test whether the agents were able to learn and develop strategies solely based on the environment in which they were embedded.

IV. Designing and building the prototypal game

A. Main features of Shooter Games

Shooter games are a sub-genre of *action* games [19], whose preeminent characteristic is to test the players' "twitch" (i.e. their reaction time and hand-eye coordination) and their ability to develop strategies in real time under condition of stress. As a matter of facts, it is the actual ability of the player that determines the "victory", not the mere characteristics and "power" of the character she uses to play, as often happens in other game genres. They are widely diffused and, according to recent market analysis, they represent the best-selling type of game. Part of this success derives from the widespread diffusion of online, multiplayer, cooperative shooters (e.g., Overwatch). Shooter games came in a wide variety of sub-genres, among which, anyway, a set of well-defined common features are shared. The emphasis is always put on the players' character actions, which generally is equipped with some sort of long-ranged weapon (e.g. rifles, bow and arrows, etc.). The goal of the game is to kill (artificial) enemies, while staying alive till the end of the level/game or of a predefined time interval. When dealing with *survival shooter games*, a "victory conditions" rarely exists: the goal of the player is to survive the longest possible time in a hostile environment, before succumbing to the overwhelming enemy's forces. Most shooter games are played in a 3D environment, where the main difference, in term of impact on the player experience, derives from the positioning of the camera through which the player observes her surroundings. In FPSs, such as Doom by id Software, the player has a first-person view, while in Third Person Shooters (TPSs) the camera is positioned behind and over the player's character, which can be observed in its entirety by the player. In many shooter games, both the environment and part of the contents (including enemies) can be procedurally generated, to increase the replayability and unpredictability of the game. Typically, the environment offers the possibility to find places to ambush enemies, and to observe their behavior unnoticed.

B. Procedural Generation of the Environment

Basing on what happens in commercial shooter games, we have prototyped the environment as a set of 3D maps, constituted by obstacles distributed pseudo-randomly on grids composed by *MxN* cells (see Fig.1). The number of maps,

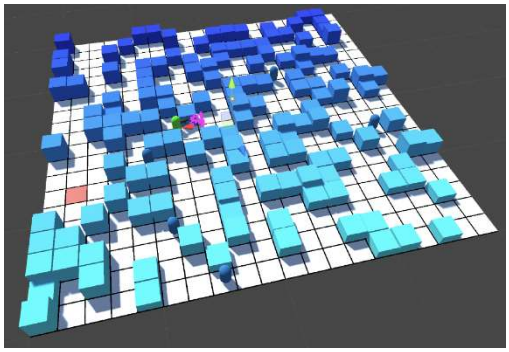

Figure 1 – Example of a map of our prototype

cells and the numerosity and height of obstacles can be varied depending on the type of training and tests to which the agent is undergoing. Obstacles are unsurmountable cells in the grid, and, by varying their number, it is possible to reduce the visibility the agent has of its surroundings and the number of different actions among which it can choose. Moreover, the different heights of the obstacles allow the agents to examine freely its surroundings (e.g. to locate NPCs), but not to aim and fire at enemies through the obstacles. Thus, simulating environments supporting stealth strategies and testing the strategical thinking of the agents, which, for example, could decide to hide and shoot the enemy just around the corner, instead of attacking it in the open. For each map, the algorithm generates a grid composed by *NxM* blocks measuring 2x2mt, basing on the values for *N* and *M* chosen by the user. In the following step, the generation algorithm places on the grid σ blocks that constitute the obstacles. The parameter σ is calculated as $\sigma=\rho.M.N$, where ρ belongs to the interval [0;0.9] and it is defined by the user before starting the generation. Each time it places a block, the algorithm verifies whether its positioning creates an isolated area or not. Those areas must be avoided in order to exclude the possibility, for an enemy or an agent, to be spawn in unreachable spots. The height – in meters – of each obstacle is randomly chosen in the interval [1;2]. Once the generation has been completed, unsurmountable blocks are planted around the borders of the map, thus avoiding the agent to wander outside the playing area and be accidentally killed (this would negatively affect the training). Finally, for each map, the values chosen for its dimensions and for the "density" and positioning of obstacles are saved as a "seed" that can be used to replicate the same experiment.

C. *The artificial player and its enemies*

Our first assumption has been that the agent should be able to learn how to survive alone and to improve by itself its aiming skill and its strategical and tactical thinking. As a matter of facts, this is what happens in multiplayer games when the player's teammates get killed or when she is self-training. Only when appropriately trained to survive alone, the agent should learn how to cooperate with teammates. Our artificial player is represented by a 2mt high green capsule that moves 5mt/s on the map, and its movement is bonded to the terrain surface (i.e. no jumping or flying over obstacles is allowed). At the beginning of each match, it spawns on a random free cell in the centre of the map. It can cross every cell that does not contain an obstacle by moving along the X and Z axis, also, it can rotate anti/clockwise on its own axis. As we already pointed out, it cannot jump over or onto obstacles: it is only allowed to peer over them, when possible. It is equipped with a semi-automatic long-range weapon able to deal 2 damage points to enemies. The weapon cannot fire uninterruptedly, thus limiting the intrinsic ability of the agent to fire at each frame, to best suit what a human player would do. The agent's perception of its environment was initially based on what it can "see" through its camera, since the knowledge it has must be a subset of the current game state. Anyway, the processing of the scenes "seen" by the agent became quite a burden on the whole system, hence, after a first phase of experimentation, we have substituted it with a more performant perception system based on ray-casting. The enemies of the agent are very simple agents, whose behaviour is managed by a state machine with only 3 states: *stand still*, *chase*, *attack* (Fig.2). Enemies are represented by blue capsules 2mt high, which move at 4.5mt/s. They, too, can cross only empty spaces of the grid, and the frequency of their spawning and the amount of health points they have can be set in the setup phase of the experiment. There is no maximum number of enemies to generate: they simply continue to spawn (on random free cells) until the agent impersonating the player is defeated, as it happens in actual games. They have no long-range weapons, they only deal 2 points of contact damage when they are near enough to the agent.

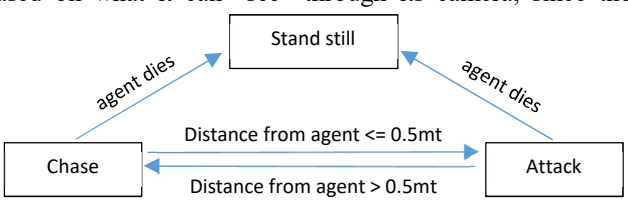

Figure 2 - Enemies' state machine

V. TEACHING SURVIVAL TO THE ARTIFICIAL PLAYER

A. *The overall architecture: Learning Environment (LE) and Learning System (LS)*

The basic idea is that the AI simulating the human player should take its decision basing on what it has learned from its experience while playing. This implies the necessity to devise an architecture that allows the flow of information and data between the prototypal game and the LS. This structure should be able to supply perceptual data about the game state to the LS and, the other way around, to translate what the LS outputs into in-game actions (such as: move back, turn to your left, fire, etc.). The architecture we have devised is represented in Fig.3. The LE includes: the perception of the game scene (i.e. the agent interacting with enemies and environmental obstacles), the AIs which manage the agents (A_i), one or more Brains (each Brain is connected to one or more agents, but each agent, in turn, can be connected to only one Brain) and the Academy (equipped with an External Communicator - EC to interact via socket with the LS).

The Brain manages rewards and observations about the agent's surroundings, it oversees the policy for each agent and determines which action should be performed next, both during the training and the inference phase. In the first case, decisions are taken externally: Brains collect observations and rewards and send them to the LS via EC and get the next action to perform for each agent. During the inference phase, instead, decisions are taken internally, basing on the policies developed by the LS during the training and then passed on to

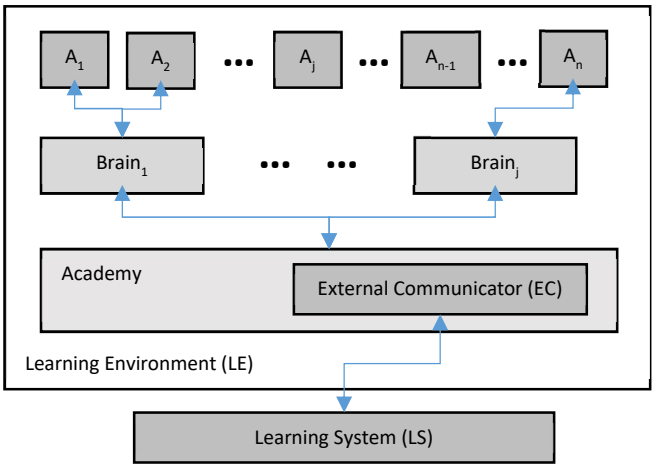

Figure 3 – The overall structure of our project

the Brains. To note that, if necessary, it is also possible to map different actions on different Brains and to train them separately, to exploit them jointly during the inference phase. Each Brain is connected to the Academy, which synchronizes Brains and agents and manages the observations and the decision processes. Moreover, the Academy is a layer where the values of several global parameters can be set and modified by the user to tune the whole application (e.g., the rendering quality). Obviously, the more agents are connected to each Brain and the more Brains are connected to the Academy, the best the learning, at the expense of the CPU performances.

B. Design and implementation of the Learning System

The LS is basically constituted by a Neural Network (NN) which exploits Deep Reinforcement Learning (DRL) techniques. We have adopted the Proximal Policy Optimization (PPO) algorithm [18], since the most suited for our peculiar scope and more performant than, for example, the Trust Region Policy Optimization (TRPO) algorithm [20]. Actually, this technique not only allows to use a States and an Actions Space continuous, but it also exploits a NN to approximate the ideal function which associates the observations of an agents to the actions it can perform (thus cutting down the complexity of the calculus necessary to revise the model policy). This feature makes PPO a particularly suitable solution when dealing with real-time problems. This is testified, e.g., by the fact that few months ago, OpenAI beat some of the best Valve Corporation's Dota2 human players of the world with OpenAI Five (https://openai.com/five/), a team of 5 agents based on PPO.

The reinforcement learning approach that we developed is based on an *Observations Space* (OS) constituted by only a subset of the information useful for describing entirely the current game state, hence it is slightly different from the States Space as it is usually defined. This, as already pointed out, allows us to simulate the human visual perception. As already stated, our starting approach was based on what the agent saw through the camera. Hence, it was not necessary to define an OS for the learning algorithm: at each frame, the scene seen by the agent was transformed into a 2D texture and then in a features map that was processed by a NN. When this approach has been substituted with ray-casting, it has been necessary to define the number of observations (θ) the agent should take into consideration during the training, where:

$$\theta = k + (\alpha * \gamma) \qquad (1)$$

$k = 3$ is a constant empirically defined basing on the velocity of the agent, $\alpha = 21$ is the number of rays casted by the agent (one each 7° in the interval [15°,165°], to simulate human vision) and γ depends on the number of observations generated by each ray. It is possible to *label* elements present on the scene, hence observations are generated each time a labelled object is encountered (thus the agent is able to distinguish among: enemies, obstacles and the map borders). Moreover, since the number of observation vectors produced by the rays casted is directly proportional to the learning performance and to the time necessary for processing the information, we have employed 4 different observation vectors: one keeps track of the agent velocity, and the remining 3 record the observation of 7 out of 21 rays each. The *Actions Space* includes all the 5 actions the agent can perform (move forward, move backward, turn right, turn left, fire). Finally, the *Reward* is a scalar that evaluates the performance of the agent during specific moments of the training. In particular, the agent gets a negative reward when it stands still (camping) or each time it gets hit or dies; it gets a positive reward when it moves (explores the map) or fires/kills an enemy. Also, to simulate a leaderboard, the higher the agent's position in the ranking of the "players", the higher the reward it gets (this last reward is used only when the agent competes with other agents).

During the learning phase, the brain stores experience, that is sent via socket to the LS each time is has reached a predefined dimension or the agent has been killed and the Academy has reset the game. Before being processed, the experience is randomized, in order to avoid that the LS learns mainly from the situation that was affecting the agent immediately before its death. The experience is memorized in a tuple <*state, action, reward, nextstate*> and stored in a buffer (whose dimension can be set during the setup of the experiment), before being transformed into a tensor and sent to the NN to be processed. The experience can be composed by continuous or discrete observations or by pictures taken by the agent's "camera". In particular, when we were using images, the Brain sent an array of 2D pictures taken by the camera. The images were processed by a two-layered convolutional network with an exponential Linear Unit (eLU) activation function [21], which pre-processed them and reduced their dimensions in order to be easily handled. Also, no polling layer had been applied, since it would have made the NN unaware of information that are relevant in a game (e.g. the position of the ball in a soccer match is essential to calculate the possible reward), but irrelevant in image processing. The result was then reduced to a unidimensional tensor (continuous or discrete) of observations. The discrete and continuous observations, instead, are processed by a feedforward NN composed by *n* layers completely connected (Fig.4). The activation function we have used is Swish [22], since it performs better than others with PPO. While continuous observations are directly fed to the NN, the discrete ones pass through a *one-hot-encoding* [23] for each element of the tensor, categorizing it according to the action type to which it refers. The resulting processed tensor, in both cases, defines the *policy* $\pi(s)$ and the *value function* $V(s)$ for the actor-critics system (in the continuous case, $\pi(s)$ and $V(s)$ are managed in two different ways, hence we process the observations two times). $\pi(s)$ and $V(s)$ measure respectively the validity of a certain state and the consequent probability of performing certain actions. Anyway, to improve its decision-

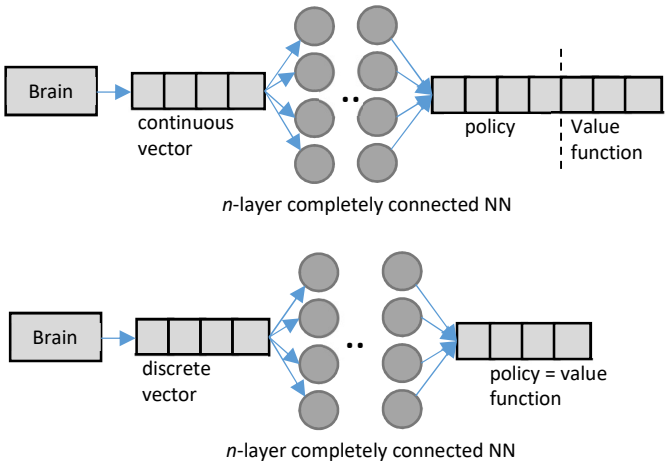

Figure 4 - Processing of continuous and discrete observations

making process, the agent should also be able to remember the decisions it took in the past. To provide this functionality, we have used two Recurrent Neural Networks (RNNs) in their Long Short-Term Memory variant (LSTM) [23]. Thus, we are both introducing the temporal dimension and mitigating the effect of the reduction and disappearance of the gradient by controlling the overfitting, hence allowing the agent to learn on several instances and providing it a mean to connect effects to causes on a certain time horizon. In particular, each of the vectors produced by the previous layer is taken distinctly as an input by a *completely connected feedforward neural network*; consequently, the agent can use the resulting estimate of the *V(s)* to adjust *π(s)* in what it is, at least in our opinion, a smarter way than the more traditional gradient approaches. The RNN-LMST is applied to the feedforward network in one single hidden layer with a linear activation function (Fig.5). Moreover, to improve overall performances, instead of back-feeding the whole set of gradients detected during the training, only the most relevant half of them is considered, consistently with the approach described in [24]. Basing on the input, a tensor is generated for each possible action. This tensor is then processed in two different ways according to its nature. In the discrete case, the tensor is processed through a *softmax* function, that produces a probability distribution on the possible effect of different actions (thus allowing to the agent to ignore the least interesting choices). In the continuous case, instead, an opportune gaussian in produced. In both cases the results are then processed by the PPO algorithm. Anyway, to function properly, the PPO algorithm needs also an entropy value. In the continuous case, the entropy depends on the extension of the gaussian curve, while in the discrete case the entropy is applied to each single action.

Once we have evaluated *π(s)*, *V(s)* and the entropy, the PPO algorithm calculates the Surrogate Loss Function as follows:

$$L_t^{CLIP+VF+S} = \hat{E}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \quad (2)$$

Where: L_t^{CLIP} is the target function of PPO (see e.g. [18]), c_1 and c_2 are constants varying in the interval [0;1], S is the entropy, L_t^{VF} is the square loss function, θ is the set of observations that have determined the current policy, and s_t is the state of the agent at time t. For each step of the network, provided that the hyperparameters (see §VI) are well balanced, the agent's performance should increase, since the entropy decreases and the coherence of *π(s)* and *V(s)* values increases. Hence, for each step, the network updates the experience (observations, actions and corresponding rewards) table, which is then used by the brain to take new decisions (Fig.6). To note that this architecture allows also a multiagent training, since more than one agent can be connected to the same Brain in the same moment (thus also accelerating the network convergence). Actually, no operation happens in parallel: each agent passes its information to the network, which generates a policy, whose effect is to modify the experience table by updating its rewards. As a consequence, the Brain applies the same policy to every agent connected to it, choosing, for each of them, the most suitable action given their current state. A nice side effect of our approach is that the overall training time shrinks, thank to the fact that dead times (occurring, for example when one of the agents waits for the environment to reset) are cut down. Nevertheless, this implies also a degradation in the CPU performance, due to the burden deriving form processing more than one agent at time.

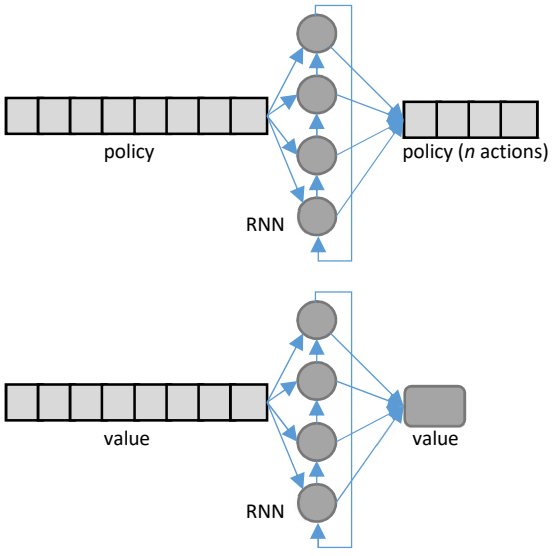

Figure 5 – Use of RNN-LMST to keep track of past decisions

VI. Testing the skills of the agents

Once the architecture described in §V.A has been deployed, we started to test and fine tune it, via a set of experiments. We have tested the behaviour and learning of our agents under the following conditions:

- *1. Basic*: agents trained alone in a randomly generated environment, where no events or leaderboard are present. The results of this test constituted the benchmark for the following phases;
- *2. With random events*: in this case, an event system has been added to the basic training conditions, to simulate human fallacies. The basic idea was to apply some bonuses/maluses to the rewards randomly and verify whether or not the system altered accordingly the agents performances;
- *3. One-man on the same map*: the agent was trained, under the basic conditions, always on the same map (generated from the same seed). The scope has been to evaluate to what extent the randomness of the environment affected the network convergence;
- *4. Competitive*: agents trained alone in an environment where events were deactivated, but a leaderboard was active. This simulated the competitiveness of human players;

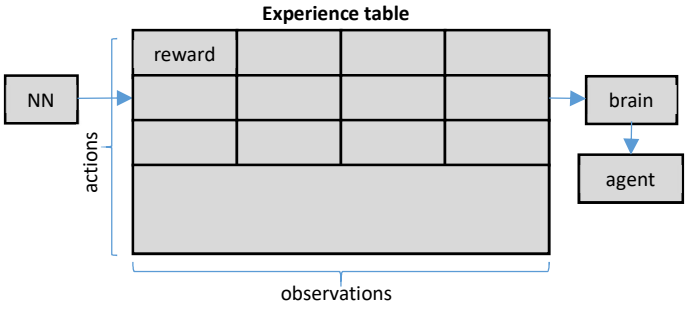

Figure 6 – Update of the experience table

- *5. Cooperative*: couples of agents were spawn in randomly chosen maps, where they trained together (no leaderboard, nor event systems were turned on).

For each type of test, several different sets of experiments have been run, varying some features of the LS. These variations are summarized by Tab.1: *LM* is the length of the experience sequences processed by the RNN (if any) during the training, and *DM* is the dimension of the arrays used to memorize the hidden layer of the RNN (if any). The variant D, which uses visual observations, has been abandoned immediately after the basic test, since it performed very poorly. In the same vein, the observation space is maninly *continuous*, since the preliminary results we have collected with discrete observations demonstrated an unsatisfactory behaviour, due to the complexity of the environment.

Table 1 – Experimental configurations of the LSs

			RNN	
Exp.	**Action Space**	**Visual Observations**	**LM**	**DM**
A	discrete	no	-	-
B	continuous	no	-	-
C	continuous	No	64	256
D	continuous	Yes	64	256

The general setting of the *environment* has been the same for all the tests we have ran, thus producing comparable results. Specifically, the map dimensions were 20x20mt and obstacles occupied 40% of the cells. Except for the experiment "one-man on the same map", in which we have used always the same seed, in all the other cases, maps were generated starting from all different seeds. The agents contemporary running were always 16. Using 16 agents – all connected to the same Brain – moving into maps generated from different seeds, offered us the possibility to train them to develop a more general-purpose policy. Enemies were spawn at the rate of 1 each 0.5sec. and they had 2 life points, while the agent's life was set to 10 points. The values we have set for the *rewards* were: camping 0.2, agent got hit 0.2, agent fires 0.05, death of the agent 0.5, enemy killed 1. The reward associated with the presence in the leaderboard was set either to 0.5 or to 0, depending on whether the leaderboard was active or not. Moreover, since we wanted to simulate a human player, we took into consideration the possibility that her performances are affected by unforeseeable external accidents or events (e.g. illness or a particularly euphoric moment). To simulate this, we randomly applied transient modifiers (bonuses or maluses, that we omit in this manuscript for the sake of brevity) to the values of the rewards. This approach permits to introduce a bit of variation without affecting the structure of the LS. Finally, the number of *observations* was set to 108, except for the cooperative test, where they were 129. This number has been calculated with (1), were $\gamma = 5$, since the one-hot-encoding uses 3 labels (obstacles, map borders and enemies). Visual observations have been used only during one basic test, and then have been substituted with ray-casting for the motivation already stated. The network has been trained by 500,000 decisional steps of the Brain: this value has been fixed empirically (it is the value for which no more relevant improvement of the cumulated reward were registered), both for the discrete and the continuous case.

The *hyperparameters* of the feedforward NN has been set as follows:

- $\phi = 0.9$ is the discount factor for future rewards;
- $\lambda = 0.93$ is used for the estimation of the advantage (how much the agent is basing on its esteem of the current value when it is updating the value of the reward);
- *BuD* value is 5,120 when *AS* is discrete, 20,480 in the continuous case. It represents the buffer dimension (number of experiences – observation, action and corresponding reward – that must be collected before any update to the policy);
- *BaD* value is set to 512 and 2,048 in the discrete and continuous case respectively and it is the batch dimensions (i.e. the quantity of experience used for each iteration in the algorithm applied to reduce the gradient);
- $E = 3$ is the epochs number (how many times the experience passes through the buffer during the gradient reduction);
- $LR = e^{-4}$ is the learning rate (how much the gradient is reduced at every interaction);
- $TO = 256$ is the time horizon (number of steps performed by the agent before collecting experience);
- $\beta = e^{-3}$ measures the entropy;
- $\varepsilon = 0.2$ is the acceptable divergence threshold between the new and old policy during the updates to decrease the gradient;
- $LY = 2$ is the number of hidden layers in the NN,
- $HU = 128$ is the number of units in each layer of the NN;
- η=3 is the "fire rate" of the agent.

VII. Outcomes of the Experiments

In the following Tab.2 we report the cumulated values of the rewards and we summarize qualitatively the result we have obtained so far. The values of the first column correspond to the configuration of the LS summarized in Tab.1. To evaluate the efficacy of the different LS solutions, we have tracked the trends of the following variables: *R* – total cumulated reward (it should grow in time, its trend is qualitatively measured), *S* - entropy (it should decrease in time), *LR* - learning rate (it should drop in time), *PL* - policy loss (should stay under a predefined threshold), *V* - value (measures the future reward the agent thinks to receive in the future), and *VL* - value loss (should drop when the cumulated reward stabilizes, since the action chosen by the policy should ideally be the same predicted by *V*). Except for the final value of the cumulated reward, actual values of the functions are omitted for the sake of brevity, since it would have been necessary to add a relevant number of graphs, that would have gone beyond the scope of a conference paper.

Table 2 – Qualitative description of the experiments outcome

Exp	R	R trend	S	LR	PL	V	VL
1. Basic (on casual maps)							
A	-269.68	Initial growth, then drops	*tf*	*cd*	*c*	*mR*	*os*
B	259.58	Grows	*cd*	*cd*	*c*	*mR*	*os*
C	438.66	Grows	*cd*	*cd*	*os*	*mR*	*os*
D	-953.43	<0, no growth	*tf*	*cd*	*c*	*mR*	*fd*
2. Random events							
A	-118.72	Initial growth, then stabilizes	*tf*	*cd*	*c*	*mR*	*os*
B	548.17	Grows	*cd*	*cd*	*c*	*mR*	*itd*
C	-61.38	Grows	*cd*	*cd*	*os*	*mR*	*itd*
3. One-man on the same map							
A	-462.47	Initial growth, then drops	*tf*	*cd*	*c*	*mR*	*os*
B	115.90	Grows	*cd*	*cd*	*c*	*mR*	*os*
C	321.54	Grows	*cd*	*cd*	*os*	*mR*	*os*
4. Competitive							
A	393.43	Steep initial growth	*tf*	*cd*	*c*	*mR*	*itd*
B	403.65	Grows	*cd*	*cd*	*c*	*mR*	*itd*
C	250.17	Grows	*cd*	*cd*	*os*	*mR*	*itd*
5. Cooperative							
A	-959.03	Initial growth, then drops	*tf*	*cd*	*c*	*mR*	*<0*
B	-424.72	No growth	*cd*	*cd*	*c*	*mR*	*os*
C	-359.27	No growth	*cd*	*cd*	*os*	*mR*	*os*

tf = decrease too quick
cd = smooth and constant decrease
c = decrease rate quite constant (very small oscillations)
os = relevant oscillations around a certain value
mR = the trend of V mirrors that of R (correctly)
itd = the trend of VL initially evidences an increases and then drops (correctly)
fd = fast drop

The *basic* experiment (#1 in Tab.2) produced the results that we have used as a benchmark for the remaining ones. The main problem of the first experiment (A) in the group derived from the fact that *S* dropped too quickly, thus not leaving enough time to the agent to learn how to behave before starting to take decisions all by itself. The following two cases (B, C) produced better results. As already stated, the case with visual observation (D) performed very poorly, since, differently from what happens in situations as those described, e.g., in [24], the agent is unable to determine a schema in the enemies' appearance, therefore negatively affecting its learning. During the subsequent inference phase, we have observed that the agents behaved correctly from the perspective of the movement (also avoiding standing still for too long, thus preventing to attract enemies) in all the cases, except with visual observations. The combat strategy, instead, has not been completely understood in all the experiments. In particular, in experiment A, agents fired quite randomly, while in the two subsequent experiments they fired more correctly, but they generally get confused when hit in the shoulder or when they got too near to enemies. In the totality of cases, the agents seemed to oscillate when moving. This effect derived from the fact that the rays casted cannot accurately simulate what would have been perceived by a camera. Last but not least, since no reward has been set for "staying alive", the agents showed "suicide" behaviours (they attacked more than one enemy at the same time in the tentative of collecting a higher reward).

Experiment 2, that considered the presence of random *events* affecting the agents' performances, registered a strong effect on *R*, which improved in the first two cases (A, B) and decreased a lot in the case of continuous actions space with RNN. The first two cases seem to suggest that the random variation in the reward has not been memorized by the network, which, consequently, has been able to adapt to varying circumstance. On the contrary, the presence of unforeseeable events heavily conditioned the RNN. To note that in the discrete case the entropy fell too quickly, thus negatively affecting the agent's learning process. During the inference phase, we noticed that the agents' behaviour slightly improved in the cases with no RNN (they moved more and quicker and had a better aim), especially in the continuous case, while it drastically worsened in the remaining one. In this latter situation, the agents seemed to move and fire randomly, likely because they did not understand the scope of the bonuses/maluses.

In the *one-man on the same map* experiment (#3), we obtained quite unexpected results: *R* trend was similar to that of the basic experiment, but all the values were smaller. This effect derived from the fact that, by observing always the same map, the LS processed always the same information, thus shrinking the action space that the AI could explore. In particular, when the RNN was present, instead of quickening the learning process due to the redundancy of information, *R* struggled to increase, and it resulted even worse than in the basic case. In the inference phase, the agents substantially behaved in the same way that in the basic case.

In the *competitive* case (#4), despite the high values of *R*, the RNN seemed to get confused by the fact that the reward deriving from the agent's rank in the leaderboard was independent from the observations about the environment. Actually, the network memorized to have reached a certain reward with a specific sequence of actions, but – obviously – their repetition not necessarily produced the same result. This effect seemed evident especially in the continuous case with RNN, where the policy loss struggled to stabilize. Anyway, the discrete case performances were greatly improved, with a learning curve that approximated that of the continuous case. During the inference phase, the agents seemed to get more aggressive and aimed at getting quickly a high reward. The agents moved rapidly, to collect more information on the environment and to detect enemies. Although the understanding of the movement strategy seemed to be improved, the attack tactic was still lacking: the agents showed imprecise aiming and semi-random fire. Moreover, the opportunity to ambush enemies seemed to be completely ignored.

Probably the worst performance that we obtained is in the *cooperative* case (#5). Actually, in neither experiment the agents managed to get a $R>0$, with the worst result in the discrete case. This is coherent with the *S* trend, which, in the discrete case, dropped to 0 very soon and with the trend of the *VL*, that in all the cases showed anomalies, particularly evident

in the discrete case. Tu sum up, the network seemed unable to generate an effective model to pass on to the Brain. The situation has been sadly confirmed in the inference phase: not only the agents performed poorly, but they even aimed and fired to the ally, although it was labelled differently from the enemies (and correctly detected during the ray-casting).

As shown in Tab.2, the agents perform quite well in at least more than half of the cases, since they manage to collect high values of *R*, even in very complex situations. Nevertheless, if we take into considerations the trend of the remaining indicators, we notice that – overall – the LS performances evidence problems, especially in the competitive and cooperative cases.

VIII. Conclusions and Future Developments

In the present work we have started to tackle the problem of training agents without any human intervention to play in a multiplayer survival shooter game. Our goal has been to move some steps in the direction of producing self-training teams of agents able to compete with humans in unexplored environments. We have designed, developed and tested a preliminary solution that couples a learning system to a prototypal game with randomly generated maps. In particular we have adopted a deep reinforcement learning approach based on PPO, an algorithm that seems to perform well when dealing with real-time situations. We also have exploited the characteristic of RNNs to provide the agent with a "memory", thus allowing it to remember which actions have produced the highest rewards. We have tested different possible configurations of the LS, and the results we have obtained so far (see §VII), although not yet completely satisfactory, seems promising. In the majority of cases, the best results in terms of cumulated reward have been obtained with a continuous observation space, but without the use of the RNN, that got confused in the less "standard" situations (e.g. unforeseeable events affecting the agent performance). Nevertheless, the results we have obtained so far are still improvable. Probably a finer tuning of the hyperparameters we have used could improve the agents' overall performances. Nonetheless, we think that the LS we have devised would benefit from a finer-grained and an ad-hoc solution, able to better pair with the complexity of a game in which the agent is called to perform multiple actions at the same time (e.g. moving, aiming and firing at enemies in an unexplored map), while developing a survival strategy and coordinating with teammates [25]. This is precisely the next aim of our research.

IX. Acknowledgments

We wish to warmly thank Prof. Nicolò Cesa-Bianchi for his help and encouragements during the development of this project.

References

[1] I. Millington, and J. Funge. Artificial Intelligence for Games, Second Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009. ISBN 0123747317, 9780123747310.

[2] G. N. Yannakakis. Game AI revisited. In Proceedings of the 9th conference on Computing Frontiers, pages 285-292. ACM, 2012.

[3] D. Karavolos, A. Liapis, and G. N. Yannakakis. Using a Surrogate Model of Gameplay for Automated Level Design. 2018 IEEE Conference on Computational Intelligence and Games (CIG), 2018.

[4] C. Mazza, L. A. Ripamonti, D. Maggiorini, and D. Gadia. Fun pledge 2.0: A funny platformers levels generator (rhythm based). In Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter, CHItaly '17, pages 22:1-22:9, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5237-6. doi: 10.1145/3125571.3125592.

[5] L. A. Ripamonti, M. Mannalà, D. Gadia, and D. Maggiorini. Procedural content generation for platformers: designing and testing FUN pledge. Multimedia Tools Appl., 76(4):5001-5050, 2017. doi: 10.1007/s11042-016-3636-3.

[6] A. Guarneri, D. Maggiorini, L. A. Ripamonti, and M. Trubian. GOLEM: generator of life embedded into mmos. In Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems: Advances in Artificial Life, ECAL 2013, Sicily, Italy, September 2-6, 2013, pages 585{592, 2013. doi: 10.7551/978-0-262-31709-2-ch084.

[7] D. Norton, L. A. Ripamonti, M. Ornaghi, D. Gadia, and D. Maggiorini. Monsters of Darwin: A strategic game based on artificial intelligence and genetic algorithms. In Proceedings of the 1st Workshop on Games-Human Interaction (GHITALY 2017) co-located with CHItaly 2017, the 12th Edition of the biannual Conference of the Italian ACM SIGCHI Chapter, Cagliari, Italy, September 18, 2017.

[8] C. Guerrero-Romero, S. M. Lucas, and D. Perez-Liebana. Using a Team of General AI Algorithms to Assist Game Design and Testing. 2018 IEEE Conference on Computational Intelligence and Games (CIG).

[9] L. A. Ripamonti, S. Gratani, D. Maggiorini, D. Gadia, and A. Bujari. Believable group behaviours for NPCs in FPS games. In: Computers and Communications (ISCC), 2017 IEEE Symposium on. p. 12-17, IEEE, ISBN: 9781538616291, grc, 2017, doi: 10.1109/ISCC.2017.8024497.

[10] A. L. Samuel. Some studies in machine learning using the game of checkers. IBM. Journal of research and development, 3(3):210-229, 1959.

[11] M. Świechowski, T. Tajmajer, and A. Janusz. Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms. 2018 IEEE Conference on Computational Intelligence and Games (CIG).

[12] Z. Yang, and S. Ontañón. Learning Map-Independent Evaluation Functions for Real-Time Strategy Games. 2018 IEEE Conference on Computational Intelligence and Games (CIG).

[13] S. F. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen, B. Kozakowski, R. Meurling, and L. Cao. Human-Like Playtesting with Deep Learning. 2018 IEEE Conference on Computational Intelligence and Games (CIG).

[14] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587):484-489, 2016.

[15] J. Harmer, L. Gisslén, J. del Val, H. Holst, J. Bergdahl, T. Olsson, K. Sjöö, and M. Nordin. Imitation Learning with Concurrent Actions in 3D Games. 2018 IEEE Conference on Computational Intelligence and Games (CIG).

[16] K. Subramanian, G. Tech, C. L. I. Jr, G. Tech, and A. L.Thomaz. Exploration from Demonstration for Interactive Reinforcement Learning. Aamas, 2016. ISSN 15582914.

[17] G. Andersen, P. Vrancx, and H. Bou-Ammar. Learning Highlevel Representations from Demonstrations. CoRR, 2018.

[18] J. Schulman, F. Wolski, P. Dhariwal, Al. Radford, and O. Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[19] M. McGuire, and O.C. Jenkins. Creating Games: Mechanics, Content, and Technology. Ak Peters Series. Taylor & Francis, 2009. ISBN 9781568813059.

[20] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In International Conference on Machine Learning, pages 1889-1897, 2015.

[21] D. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). ICLR 2016: International Conference on Learning Representations 2016.

[22] P. Ramachandran, B. Zoph, and Q. V. Le. Swish: a self-gated activation function. arXiv preprint arXiv:1710.05941, 2017.

[23] S. Hochreiter, and J. Schmidhuber. Long short-term memory. Neural computation, 9(8):1735-1780, 1997.

[24] G. Lample, and D. S. Chaplot. Playing fps games with deep reinforcement learning. In AAAI, pages 2140{2146, 2017.

[25] V. M. Petrovic. Artificial Intelligence and Virtual Worlds – Toward Human-Level AI Agents. IEEE Access, Vol.6, pp.39976-39988, 2018.