On the Development of an Autonomous Agent for a 3D First-Person Shooter Game Using Deep Reinforcement Learning

Paulo B. S. Serafim* Yuri L. B. Nogueira Creto A. Vidal Joaquim B. Cavalcante-Neto

Federal University of Ceará (UFC), Department of Computing, Brazil

Figure 1: The three ViZDoom's scenarios played by the agent. **Left**: Basic. **Center**: Defend the Line. **Right**: Medikits and Poisons.

ABSTRACT

First-Person Shooter games have always been very popular. One of the challenges in the development of First-Person Shooter games is the use of game agents controlled by Artificial Intelligence because they can learn how to handle very distinct situations presented to them. In this work, we construct an autonomous agent to play different scenarios in a 3D First-Person Shooter game using a Deep Neural Network model. The agent receives as input only the pixels of the screen and should learn how to interact with the environments by itself. To achieve this goal, the agent is trained using a Deep Reinforcement Learning model through an adaptation of the Q-Learning technique for Deep Networks. We evaluate our agent in three distinct scenarios: a basic environment against one static enemy, a more complex environment against multiple different enemies and a custom medikit gathering scenario. We show that the agent achieves good results and learns complex behaviors in all tested environments. The results show that the presented model is suitable for creating 3D First-Person Shooter autonomous agents capable of playing different scenarios.

Keywords: 3D first-person shooter, autonomous agent, reinforcement learning, deep neural networks.

1 INTRODUCTION

First-Person Shooter games have a big popular appeal. This subgenre of action games had a great impact on audience since the first games of this type, like *Wolfenstein3D* (id Software, 1992) and *Doom* (id Software, 1993), and keep impacting the players until today with popular series, like *Call of Duty* (Activision), *Battlefield* (Electronic Arts) and *Counter-Strike* (Valve). The development of the game industry brought the need of better characters controlled by Artificial Intelligence in these games. Although scripted players are becoming more easy to make, they still need to have all behaviors explicitly created by the programmers.

To solve this problem, the creation of an autonomous agent that learns how to behave in different environments is a desired objective. However, creating an autonomous agent to play a complex game like First-Person Shooters is very hard. Such agent must be capable of taking a wide range of actions, like exploring your surroundings, aiming and shooting enemies, picking up items and surviving as long as possible. Ideally, the agent should be capable of learning complex behaviors with the minimum number of information given to him directly, i.e., scripted.

One desired characteristic is that the input received by the agent is the current screen view. Using only the pixels of the screen guarantees input fairness, which means that the input is the same for every player, even if he is a human player. However, the construction of an autonomous virtual player that receives as input only raw pixels is a very difficult task. Deep Learning [11] is a technique which can handle this type of problem. Since its development, [7] it was extended to solve different tasks. Recently, it is also being used to model autonomous game agents that receive as input only the pixels of the screen [15]. Thus, the creation of an autonomous game character that learns complex behaviors by himself begins to become feasible.

The goal is to create an agent who learns adequate behaviors by receiving a reward for every action taken in a given moment. Reinforcement Learning [22] is the appropriate learning paradigm to solve this type of problem. The learning process moves towards a solution that maximizes the numerical reward by choosing one of the possible actions in that state. In other words, the agent must learn the best action to take in every state. Combining Deep Learning and Reinforcement Learning, we have a strong technique to handle such problem.

In this work, we develop a Deep Reinforcement Learning autonomous agent to play three different scenarios in a 3D First-Person Shooter game. We use the API developed by Kempka et al. [8], ViZDoom, a research platform based on the game Doom (Figure 1) intended for research in Machine Visual Learning and Deep Reinforcement Learning. This tool gives a direct access to the

*e-mail: paulobruno@alu.ufc.br

game engine and allows the construction of a custom game agent that sends commands to the engine and receives information about the current state of the game.

In a previous work [20], we used a Deep Neural Network architecture intended for solving a Supervised Learning problem. This network was validated in a classification problem and was applied in an autonomous agent for playing a 3D First-Person Shooter Game. In that work, we discussed the possibility of creating a general network capable of solving problems from different learning paradigms. In the present work, we focus on the development of an agent to play the game. A network architecture was created, it is more suitable for game playing and behaves better on the proposed environments.

This paper is organized as follows. In Section 2, we summarize some works which use a Deep Neural Network model to play digital games. Section 3 brings a description of Q-Learning, the main algorithm used in Reinforcement Learning problems. We describe the scenarios played by the agent in Section 4. The description of the performed experiments is shown in Section 5. In Section 6, we show obtained results and discuss them. Lastly, some possible future works are suggested in Section 7.

2 Related Work

The first use of a Deep Learning model in Reinforcement Learning was made by Mnih et al. [15]. They trained a Deep Neural Network using a variant of the Q-Learning algorithm. Using as input only raw pixels from the screen, this approach has been used successfully to produce agents capable of playing digital games. In this pioneering work, a Deep Q-Learning model learned to play seven Atari 2600 games using as input a brute high-dimensional input, only the raw pixel data. The authors also used an adaptation of Q-Learning called Experience Replay [13], which improved the efficiency of data use, increased the learning speed and led to a better choice of parameters to avoid local minima.

Mnih et al. [16] increased the size of their previous work. The authors created a variant of the technique, called Actor-Critic model. They tested this version in 49 games of Atari 2600, reaching human-level performance in 29 of them.

These works inspired many others and games from Atari 2600 became vastly used as testbeds. New techniques for Deep Learning based models arose. Hasselt et al. [6] created Double Q-Learning, a variant of Q-Learning in which two simultaneous value functions were learned. This approach resulted in two different sets of weights, which improved the performance over the traditional algorithm. They tested it in 57 Atari 2600 games, eight more than in Mnih et al. [16].

Nair et al. [17] made another related advance. The authors proposed a modification of the technique presented in the work of Mnih et al. [15][16], by introducing a distributed system architecture. In comparison with previous approaches, the parallelization proposed improved the performance of the agent. The model was tested in the same 49 Atari 2600 games and in 41 of them the technique outperformed the non-distributed models.

A new approach was used by Wang et al. [24], dividing a network to represent two separate estimators: one to be concerned with the results yet to come and the other to be concerned with immediate actions. The first estimator was used for the state-value function and the second was used for the state-dependent action advantage function.

Parisotto et al. [19] proposed a new method of training a single Deep Network of actions over a set of related tasks. The method consists of expert teachers orienting the agent to make actions. Thus, the agent learned his decision making through the received orientation. This model was called *Actor Mimic*. The authors showed that this technique played several Atari 2600 games simultaneously, at the same level as an expert.

Mnih et al. [14] proposed a new model using asynchronous gradient descent for controller optimization. Four variations of Reinforcement Learning techniques were presented and the authors showed that their method outperformed the state-of-the-art techniques. Furthermore, the computation effort was greatly reduced. The model was trained in a single multi-core processor in half the time of conventional training.

Lample and Chaplot [10] used as testbed ViZDoom [8]. The authors used an architecture of a Deep Recurrent Network connected to a Long-Short Term Memory Neural Network. In training, the agent explored information of game features, such as presence or absence of an enemy and/or items, to simultaneously learning them while minimizing the objective function. This variation improved the performance of the agent and greatly reduced the training speed.

Another Deep Reinforcement Learning application in ViZDoom was used by Kulkarni et al. [9]. The authors used a model called Successor Representations, which decomposes the value function into two components: a reward predictor and a successor map. They trained an agent to solve two problems: finding the exit of a maze in MazeBase and reaching a goal room in ViZDoom.

Dosovitskiy and Koltun [3] combine a high-dimensional sensory stream and a lower-dimensional measurement stream to create a model that provides a rich supervisory signal. The agent was trained to play four scenarios of ViZDoom using Supervised Learning techniques. The agent trained won the Full Deathmatch track of the 2016 Visual Doom AI Competition.

Bhatti et al. [2] augmented the raw input image of the Deep Neural Network by adding information of localization of the player and details of objects and structural elements encountered. They use ViZDoom to evaluate the efficacy of this approach. They show that the proposed augmented framework consistently learns effective policies.

A Neuroevolution approach was used by Alvernaz and Togelius [1]. The authors trained an autoencoder network to create a low-dimensional representation of the environment observation and then used it as an input of a CMA-ES to train the controllers. They tested the model on the health gathering scenario of ViZDoom, where the player loses health over time and needs to pick up health packs to survive longer.

In a previous work [20], we presented a network architecture to solve a Supervised Learning problem, the classification of a handwritten dataset, and a Reinforcement Learning problem, a 3D First-Person Shooter game. We used a Deep Neural Network model to solve both problems. In both cases, the input was only the pixels of an image. ViZDoom's environment was also used as the testing environment for Deep Reinforcement Learning. We showed that a single network architecture was suitable for the classification task and was capable of playing the game.

3 Background

In this section, we present a short summary of the Deep Q-Network model used to train an autonomous agent in a Reinforcement Learning problem [10][15].

3.1 Deep Q-Network

Reinforcement Learning tasks are sequential decision problems in which the objective is to find the best policy in order to maximize the sum of received rewards. A policy is a set of actions that should be taken by the agent for every possible state. The agent analyzes the current state s, and decides what action, a, to take according to the policy, π. The goal of the agent is then to find the best policy in which the expected sum of discounted rewards, R_t, is maximum

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i \quad (1)$$

where T is the last iteration, t is the current iteration and γ is the discount factor, which varies in the interval [0,1]. The discount factor trades-off the importance of future rewards, so that immediate rewards are more important than later ones.

The value of an action a in a state s under a given policy π, the Q-function, is defined as the expected return

$$Q^{\pi}(s,a) = \mathbb{E}[R_t|s_t = s, a_t = a] \quad (2)$$

The optimal value of $Q^{\pi}(s,a)$, Q^*, is defined as the highest expected return by following any strategy

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad (3)$$

An optimal policy is then derived from the optimal values by choosing actions with the highest value in each state. Instead of trying to learn all action values, it is easier to learn a parameterized value function Q_θ, in which Q^{π} is close to the optimal Q-function Q^*. Therefore, the algorithm tries to find a θ such that $Q_\theta(s,a) \approx Q^*(s,a)$.

The optimal Q-function verifies the Bellman optimality equation

$$Q^*(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \quad (4)$$

where s' is the reached state and a' represents the following set of actions. Therefore, if $Q_\theta(s,a) \approx Q^*(s,a)$, it is natural to think that Q_θ should be close to verify the Bellman optimality equation, which will lead to the loss function

$$L_t(\theta_t) = \mathbb{E}[(y_t - Q_{\theta_t}(s,a))^2|s,a,r,s'] \quad (5)$$

where s is the current state, a is the action taken, r is the reward received, s' is the state reached and

$$y_t = r + \gamma \max_{a'} Q^*(s',a') \quad (6)$$

Thus,

$$L_t(\theta_t) = \mathbb{E}[(r + \gamma \max_{a'} Q_{\theta_{t-1}}(s',a') - Q_{\theta_t}(s,a))^2|s,a,r,s'] \quad (7)$$

The value of y is fixed for the previous iteration, which leads to the following gradient by differentiating the loss with respect to the weights

$$\nabla_{\theta_t} L_t(\theta_t) = \mathbb{E}[(y_t - Q_{\theta_t}(s,a)) \nabla_{\theta_t} Q_{\theta_t}(s,a)|s,a,r,s'] \quad (8)$$

Instead of using an accurate estimate of the above gradient, we compute it using the approximation

$$\nabla_{\theta_t} L_t(\theta_t) \approx (y_t - Q_{\theta_t}(s,a)) \nabla_{\theta_t} Q_{\theta_t}(s,a) \quad (9)$$

The Q-learning updates, using the loss function estimations of (5), are stable and perform well in practice [16].

4 Environments

The agent was trained using a Deep Neural Network model with Q-Learning [25] as the learning algorithm. Using as input only visual information from screen buffer, he was trained to play the three different scenarios below.

4.1 Basic

This is the simpler scenario present in ViZDoom. Its purpose is to evaluate if the model is capable of training and learning in a 3D First-Person shooter environment.

In this scenario (Figure 1left), the agent faces only one enemy (Figure 2a). They are positioned in opposite sides of a rectangular room. The agent always starts in the center of his side and the monster starts in a random position along its side. The player can make one of three actions: move left, move right or shoot. The monster is static, does not move and do not make any action and one shot is enough to kill it.

(a) Static enemy. (b) Worm enemy. (c) Snake enemy.

Figure 2: Enemies. **(a)**: enemy present in Basic scenario. **(b)** and **(c)**: enemies present in Defend the Line scenario.

4.2 Defend The Line

A much more complex scenario, here, the goal is to verify if the agent can learn to kill different monsters with very little information. In this scenario, very common actions in First-Person Shooter games are possible, like aiming at an enemy and killing different types of enemies.

The player faces multiple enemies at the same time in a rectangular map (Figure 1center). There are two different types of enemies: a red worm who moves towards the player (Figure 2b) and a brown snake who cannot move but is capable of shooting fire balls (Figure 2c).

At the start of an episode, the agent and the team of enemies are spawned in opposing sides. The player is spawned in the center of his side. Six monsters, three of each type, are spawned on the opposing side. Initially, the monsters die with only one shot. However, a killed enemy will respawn after some time and will become stronger, needing more shoots to be killed.

Every time the player is touched by the red worm or hit by a fire ball, he will lose some health. The score of this environment is the number of enemies killed. Therefore, the goal of the player is to maximize the number of enemies killed.

4.3 Medikits and Poisons

We constructed this custom scenario to observe the learning and behavior of the character in an environment that can be explored by the agent. Picking up certain items and avoiding others is very common in First-Person Shooter games. No enemy is present and the goal of the agent is to explore the scenario picking up medikits and avoiding poisons.

A medikit (Figure 3a) is a gray box with a white cross on a green background. This item increases some health of the player. A poison (Figure 3b) is represented by a blue and gray capsule with a small red cross on its top. This item decreases some health of the player.

The environment is a big rectangular room with a red lava floor, grayish walls and a dark sky. The lava floor hurts the player periodically, decreasing his health.

Initially, there some medikits and poisons spread uniformly over the map. From time to time, one medikit drops from the sky in a random position. Similarly, after some time one poison drops from the sky in a random position. These positions are not fixed and may even be behind the player.

The agent has three possible actions: turn left, turn right and move forward. These actions grant a freedom of movement to the player, allowing him a free exploration of the environment.

(a) Medikit. (b) Poison.

Figure 3: Items found on Medikits and Poisons environment.

5 EXPERIMENT

5.1 Hyperparameters

All the following parameters were chosen arbitrarily. The Q-learning discount factor, γ, was set to 0.99 and the learning rate was equal to 0.0001. The network weights were all initialized with random values using Xavier's Initialization [4] and all bias were initialized with a value of 0.1.

The RMSProp algorithm was used to train the network, using mini-batches of size 64. The RMSProp is an adaptation of Mini-Batch Gradient Descent which divides the gradient by a running average of its recent magnitude [23].

We used Experience Replay [13] to reduce correlation between consecutive frames. A replay memory keeps track of the latest ten thousand frames and a randomly chosen sample is passed to the network at every update. To reduce overfitting, we used the Dropout technique [21] during training with a probability of 0.7.

An ε-greedy policy [25], with linear decay from 1.0 to 0.1, was used to balance the trade-off between exploration and exploitation. That is, the agent has a probability of ε of choosing a random action to make rather than the current best action.

If the agent performs one action per frame, the difference between the previous states and the current one is so subtle that learning will become hard. To solve this problem we used a frame repeat value of eight frames. This means that an action is chosen, repeated through eight frames and only then another action is chosen.

5.2 Network Architecture

Figure 4 summarizes the network architecture presented below. A matrix of floating point numbers, representing a 64 by 48 grayscale image is the input of the network. The inputs are then passed to a convolutional layer. A convolutional layer, developed by LeCun et al. [12], has the characteristic of detecting specific features according to the spatial position of the input. That is, it has the ability to filter some aspects of the input.

This first convolutional layer has a kernel width and height of size four, and a stride with width and height of size two. This convolution setting halves the size of the inputs. We use ReLU [18][5] as the activation function. In this layer, 64 features are computed.

Then, the outputs are passed to another convolutional layer. This layer also has a kernel size of 4 by 4 and a stride of 2 by 2, halving the size of its inputs. ReLU is also used as the activation function. This layer computes 128 features.

After that, the 128 images of size 16 by 12 are flattened and are then fully connected to a layer of 512 neurons. This layer has the objective of putting together all the previous features found. A dropout is applied before the output layer and the results are passed into the readout layer, which has three neurons, one for each possible action.

5.2.1 Input and Output

The raw colored 640 by 480 pixels image of the screen is reduced into a 64 by 48 pixels gray scale image, which is the input of the network (Figure 5). Thus, the input is a layer of 64 by 48 neurons with values varying from 0.0 to 1.0.

Every action has its own q-value, therefore every action implies in one output neuron. The output layer has three neurons because all scenarios have three possible actions, namely:

Basic: move left, move right and shoot;

Defend the Line: turn left, turn right and shoot;

Medikits and Poisons: turn left, turn right and move forward.

5.3 Training and Testing Regime

In all environments, the agent was trained for 50 lifespans of 10000 learning steps each. Every learning step is an update of the network using Q-Learning with mini-batches of size 64, thus is in this moment that the learning actually happened. After every lifespan, the agent was tested for 50 episodes.

5.4 Evaluation Metrics

Every scenario has a different score metric:

5.4.1 Basic

The agent loses 1 point for every step he remains alive, i.e. gains a -1 reward, and he loses 5 points for shooting. A score of 101 is rewarded if he kills the enemy. Therefore, the best possible score is 95. An episode finishes when the monster is killed or after 300 steps have passed.

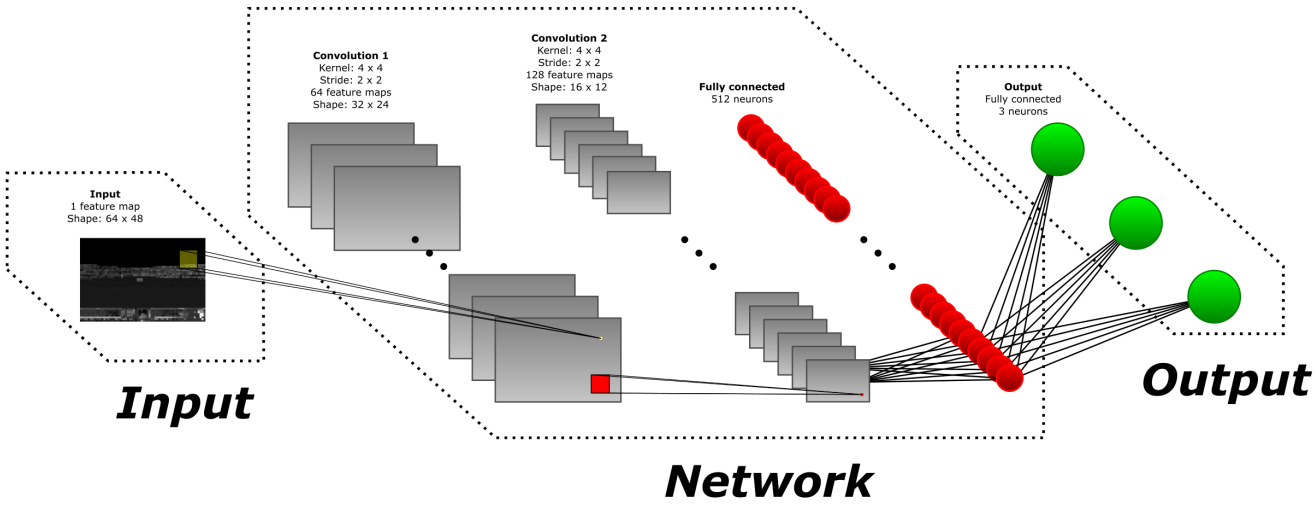
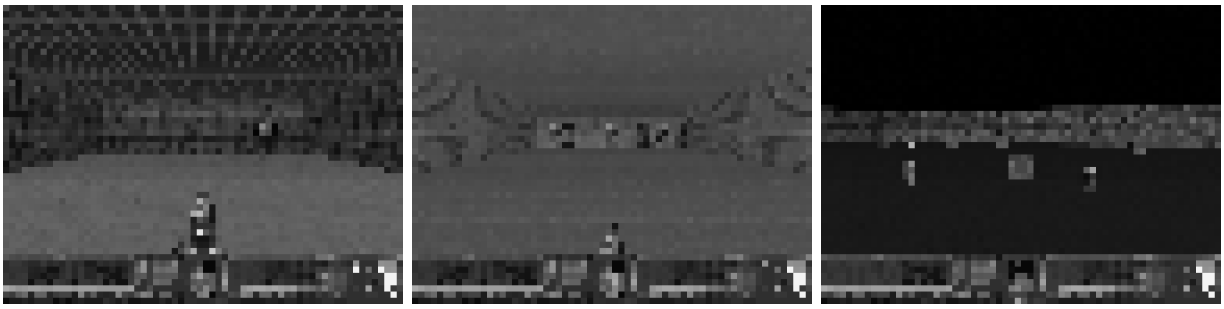


Figure 4: A representation of the Network Architecture. **Input**: a matrix of floating point numbers representing a grayscale image. **Network**: the input image passes through two convolutional layers and the output is flattened and fed into a fully connected output layer. **Output**: the three neurons give the q-value of each action.



(a) Basic. (b) Defend the Line. (c) Medikits and Poisons.

Figure 5: Vision of the agent in every environment. These frames are the same of Figure 1.

5.4.2 Defend The Line

The only reward awarded to the player is one point after killing an enemy. He is hurt if touched by the worm enemy (Figure 2b) or hit by a fire ball. However, the agent does not receive this information as a numerical feedback, only on screen. An episode ends when the player is killed.

5.4.3 Medikits and Poisons

For every step the agent stays alive, he gains 1 point. Picking up a medikit grants him a health increase, while picking up a poison decrease his health. If he picks up a medikit he receives 10 points and loses 10 points for picking up a poison. If the player gathers three poisons in sequence, he dies. The lava floor of the environment decreases the health of the agent periodically, but this information is not given directly to him, only on screen. The player loses 100 points for dying. An episode ends when the player dies or after 3000 steps have passed.

6 Results and Discussion

To see examples of the results described below, please refer to the accompanying video.

6.1 Basic

An optimal behavior for the player is moving until being in front of the enemy and quickly kill it with a single shot. If the monster is spawned right in front of the agent, an instant shot will grant a score of 95. If the enemy is spawned as far left or right as possible, moving towards the monster and killing it in the first shot will grant a score of about 63. Thus, a mean expected reward for an optimal behavior is about 78.

We can see in Figure 6 that the player achieves the optimal expected mean score since the first lifespan. Because of the simplicity of the environment, one lifespan of 10000 network updates is enough to teach the agent how to behave correctly.

While in testing the actions taken are always the ones that have the best value, the actions taken in training on the first lifespans are mostly random, which leads to very bad scores. Over time the randomness in training is reduced, which increases the mean score. The growth of the curve in training shows us that the learning is also happening during training.

6.2 Defend The Line

This scenario is much more complex than the previous one. Because the agent receives feedback only when he kills an enemy, he should learn how to aim and shoot different targets by himself. In this environment, there is no optimal behavior. However, we expect that the agent turns towards an enemy and shoot it quickly.

Figure 7 shows that the first lifespan of training is enough to teach the player a behavior that grants him a score of about 14. By

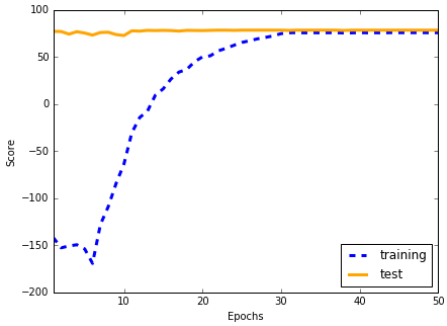


Figure 6: Mean score of Basic scenario. The higher the score, the faster the enemy was killed. The dashed line shows the evolution during training and the continuous line shows the during testing.

the growth of the training curve, we observe that the agent is clearly learning. Not only that but also his performance is improving over time. On final lifespans, the training results stabilized. At this moment, the agent could kill about 24 monsters, seven more than in our previous work [20].

This score shows that the agent is capable of learning how to behave in a complex environment without having direct information. The player was capable of aiming and shooting moving targets. He learned not only to kill monsters but the importance of killing them to stay alive as long as possible.

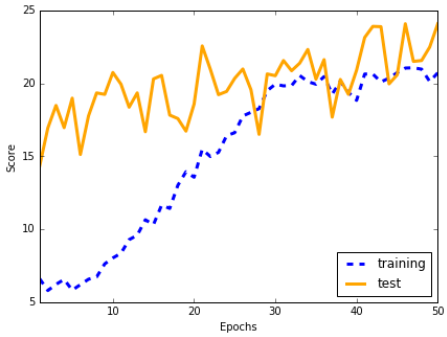


Figure 7: Mean score of Defend the Line scenario. The score represents the number of enemies killed. The dashed line shows the evolution during training and the continuous line shows the evolution in testing.

6.3 Medikits and Poisons

Another complex scenario, now the player should learn to identify medikits and poisons, perceive that picking up a medikit is good and picking up a poison is bad. Furthermore, the agent should explore the environment to look for medikits. An expected behavior is seeing the player gathering only medikits and avoiding picking up poisons.

We see from Figure 8a that after the first lifespan of the training the agent has a very bad mean score of 518.47. If the agent acts lifeless, taking very small steps and not picking anything, he finishes the episode with a score of 508. Thus, we see that the average behavior after the first lifespan is not gathering any medikit or poison.

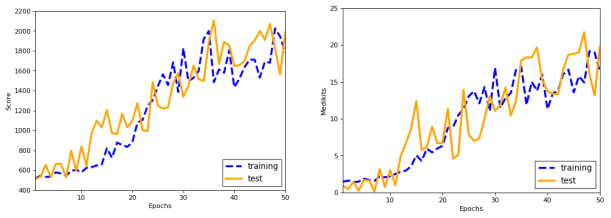
However, the training mean score grows over time and the performance of the agent grows together. Thus, we can see that the agent is becoming better in exploring the scenario and picking up medikits. After the final lifespan, the performance of the agent is much better and he is able of living for about 1900 timesteps.

At this time, the agent is capable of looking around for a medikit and moving towards it when he finds it. This behavior can be seen on the screenshots from Figure 9. The player cannot see any medikit and he starts to turn right looking for one. When he finds it, he starts to move forwards to gather it.

Some undesired behaviors were also encountered. For example, when a poison is too close of a medikit, the agent cannot see it and pick up both medikit and poison. Another problem is concerned with the vision of the player: if the medikits are too far, the agent cannot see them and keep searching. Even with these situations, the agent is capable of living as long as possible.

6.3.1 Number of medikits picked up

Another useful way to observe the success of the learned behavior is to evaluate the number of medikits picked up per lifespan (Figure 8b). We observe that the player learns to pick up more medikits over time. This shows us that he becomes better at exploring the environment and gathering more medikits, which is in accordance with the expected behavior and allows him to live longer.



(a) Score per epoch.

(b) Medikits picked up per epoch.

Figure 8: Results of Medikits and Poisons scenario. The dashed line shows the evolution during training and the continuous line shows the evolution in testing. **(a)** mean score of the agent per epoch. The greater the score, the longer the agent remained alive. **(b)** mean number of fruits eaten per epoch. The more fruits are eaten, the longer the agent can survive.

7 Future Works

7.1 Changing Neural Network Settings

Increasing the size of the network can lead to better results because more features can be learned and passed throughout the layers. Furthermore, a bigger network may have more information to work with. However, increasing size also increases complexity, which will decrease the learning speed. A bigger network will be much harder to train and can have difficulties to achieve good behaviors.

Making adaptations on the network, as seen in [10] and [2], is also a good option that can improve the performance of the agent. The scenarios may be easily adapted to give additional information. However, giving more information to the player must be done carefully. It is interesting that the agent figures out some characteristics and learns how to handle some situations on his own.

7.2 Using Other Agents

In this work, we created an agent based on Deep Reinforcement Learning. Other agents can be tested in the same environments. These agents do not need to use necessarily Neural Networks. On the contrary, the comparison between different learning algorithms may raise interesting questions. Although these different approaches may need some adaptation, we think that inputs and outputs should be the same as those presented in this work. That is, the agent should make one action out of the three possible in each scenario and the input should be a representation of the screen. These points will ensure fairness in comparisons with the agent presented here.

7.3 Testing Different Environments

Three scenarios were discussed in this paper, but the ViZDoom environment offers even more. Moreover, custom scenarios, like Medikits and Poisons presented here, can be easily made with different characteristics and purposes. The more scenarios are tested, the more accurately it is to evaluate the versatility of the player.

However, it is necessary to evaluate carefully the behavior of the agent. Some environments will be very easy, which will lead the agent to achieve good scores and others can be very hard, which will lead to very bad scores. This raises the question that discovering what is the best setting for every scenario is not easy. A change that leads to an improvement of behavior in one scenario may imply in a decrease of score in others. Creating an agent which could deal with any scenario is a very challenging task.

7.4 Playing Other Games

A further step which makes use of the general input characteristic of the agent presented here is to test him in other games. The agent receives as input only an image and then makes one possible action. This input-output dynamic is very general and allows us to treat the internal processes of the network as a black box. Thus, a great variety of games can be used as the environment for the agent. Some games, like the ones presented in section 2, are common testbeds for Deep Reinforcement Learning.

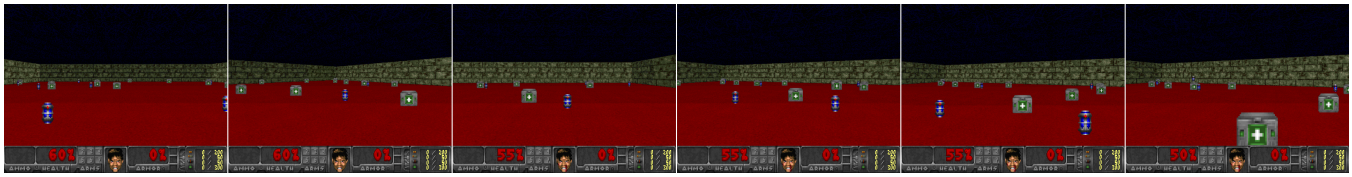


Figure 9: **From left to right, 1st frame**: the agent cannot see any medikit. **2nd frame**: he turns right looking for a medikit. **3rd frame**: he finds one. **4th frame**: he aligns himself with the medikit. **5th and 6th frames**: he moves forward to pick it up.

8 Conclusion

We showed that a Deep Reinforcement Learning game agent was capable of learning how to play well different environments in a 3D First-Person Shooter game. An optimal behavior was easily learned in the simpler scenario, namely Basic. The next two environments were much more complex. In Defend the Line the agent learned to turn towards an enemy and kill it. Moreover, he learned how to survive long enough to kill several of them. The agent also showed that he is able to succeed in a scenario designed for exploration. He learned to search items that would increase his life and avoid items that could kill him.

All examples show that the Deep Reinforcement Learning agent proposed in this work was capable of learning complex behaviors in all environments receiving as input an image. The agent is prepared to be trained in several different environments without further configuration. This gives us a starting point in constructing an autonomous 3D First-Person Shooter game character which can be able to handle very distinct situations using only an image as input.

Acknowledgments

The authors would like to thank CNPq, the National Council for Scientific and Technological Development, and the Graduate program (MS and PhD) in Computer Science at the Federal University of Ceará (MDCC/UFC) for their financial support.

References

[1] S. Alvernaz and J. Togelius. Autoencoder-augmented neuroevolution for visual doom playing. *CoRR*, abs/1707.03902, 2017.

[2] S. Bhatti, A. Desmaison, O. Miksik, N. Nardelli, N. Siddharth, and P. H. S. Torr. Playing doom with slam-augmented deep reinforcement learning. *CoRR*, abs/1612.00380, 2016.

[3] A. Dosovitskiy and V. Koltun. Learning to act by predicting the future. *CoRR*, abs/1611.01779, 2016.

[4] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[5] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[6] H. V. Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016.

[7] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[8] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016.

[9] T. D. Kulkarni, A. Saeedi, S. Gautam, and S. J. Gershman. Deep successor reinforcement learning. *CoRR*, abs/1606.02396, 2016.

[10] G. Lample and D. S. Chaplot. Playing FPS games with deep reinforcement learning. *CoRR*, abs/1609.05521, 2016.

[11] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Dec 1989.

[13] L.-J. Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1993. UMI Order No. GAX93-22750.

[14] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv*, 48:1–28, 2016.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. a. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[17] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv:1507.04296*, page 14, 2015.

[18] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In J. Fürnkranz and T. Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010.

[19] E. Parisotto, L. J. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR*, abs/1511.06342, 2015.

[20] P. B. S. Serafim, Y. L. B. Nogueira, C. A. Vidal, and J. B. Cavalcante-Neto. Towards playing a 3d first-person shooter game using a classification deep neural network architecture. In *Proceedings of the 19th Symposium on Virtual and Augmented Reality (SVR 2017)*, in press.

[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[22] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[23] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[24] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling network architectures for deep reinforcement learning. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[25] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.