ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Grado en Ingeniería Informática

Desarrollo de jugadores automáticos mediante aprendizaje profundo por refuerzo para videojuegos

Development of automated players by deep reinforcement learning for videogames

Realizado por

Antonio David Ponce Martínez

Tutorizado y cotutorizado por

Ezequiel López Rubio y

Jesús Benito Picazo

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2019

Fdo. El/la Secretario/a del Tribunal

Resumen

En este proyecto se presenta un modelo de aprendizaje profundo capaz de aprender a realizar varias tareas usando el juego de 1993 DOOM como entorno. El agente es entrenado con los píxeles en crudo de la pantalla de juego y usa una variante de aprendizaje profundo del algoritmo Q-learning. Varias técnicas de optimización fueron aplicadas para maximizar el rendimiento y los resultados.

Palabras clave: aprendizaje profundo, Q-learning, inteligencia artificial, DOOM, Q-learning profundo.

Abstract

This project presents a deep learning model able to learn how to perform several tasks using the 1993 game DOOM as environment. The agent is trained using raw pixels from the game screen and uses a deep learning variant of the Q-learning algorithm. Several optimizations techniques were applied in order to maximize performance and results.

Key words: deep learning, Q-learning, artificial intelligence, DOOM, deep Q-learning.

Contents

1. Introduction **1**

1.1. Motivation 1

1.2. General goal 2

1.3. State of the art 2

1.4. Report structure 3

2. Background **5**

2.1. Reinforcement learning 5

2.2. Q-learning 6

2.3. Neural Networks 7

2.3.1. Neuron 7

2.3.2. Learning Process 8

2.4. Convolutional Neural Networks 9

2.4.1. Input data 9

2.4.2. Filter 10

2.5. Deep Q-Learning 10

2.5.1. Experience Replay 11

2.5.2. Learning Process 13

2.6. DOOM (1993 video game) 14

3. Implementation **15**

3.1. Technologies 15

3.1.1. Keras . . . 15
3.1.2. VizDoom . . . 15
3.1.3. scikit-image . . . 16
3.2. Hardware specification . . . 16
3.3. Input processing . . . 16
3.4. Agent architecture . . . 17
3.5. Populating the Replay Buffer . . . 19
3.6. DQN parameters . . . 19
3.7. Main structure . . . 21

4. Results **23**
4.1. Basic . . . 23
4.2. Defend the center . . . 25
4.3. Improvements . . . 27
4.3.1. Double DQNs . . . 28
4.3.2. Dueling DQN . . . 28
4.3.3. Prioritized Experience Replay . . . 29
4.3.4. Higher-level input . . . 30

5. Conclusions **33**
5.1. Technical problems encountered . . . 33

5. Conclusiones **35**
5.1. Problemas técnicos encontrados . . . 35

A. UML diagrams **41**
A.1. Class diagram . . . 41
A.2. Sequence diagram . . . 42

Chapter 1

Introduction

1.1. Motivation

Achieving complex behaviours using high-dimensional sensory raw data as input is one of the main challenges of reinforcement learning (RL). Most RL algorithms uses a combination of high-level or hand-crafted features, which makes the system heavily rely on the quality of these features representation.

Deep learning techniques have, on the other hand, shown that it is possible to extract high-level features from raw data, highly proved by the recent advances on computer vision and speech recognition, so it seems natural to try these feature extraction techniques on similar problems with sensory data.

However, most successful deep learning algorithms depend on the previous labeling of input data, which is not always possible. RL techniques overcome this issue by learning using a scalar reward signal. Nevertheless, this signal is often sparse, noisy and delayed in contrast with supervised learning approaches, in addition to the need of independence between the data samples.

For this project, an agent has been developed using a Deep Q-Network to successfully maximize the rewards of some tasks using raw video data, overcoming previous mentioned challenges.

1.2. General goal

These project goals are the following:

- Create an agent that is able to perform several tasks using the 1993 game DOOM as environment.
- Study the results obtained and possible optimizations.
- Mention the challenges encountered on this task and their solutions.

1.3. State of the art

Advances in hardware, the boom of Big Data and the development of new frameworks have put machine learning in the spotlight. New applications are being discovered rapidly and techniques such as Reinforcement Learning (RL) begin to gain importance. These algorithms based on RL are able to solve the difficult problem of correlating immediate actions with delayed returns. They can be expected to perform well in these environments where it can be difficult to understand which actions leads to which outcome.

Back in 1995, one of the best-known success of reinforcement learning was achieved: TD-gammon. It consisted in an agent capable to play backgammon, achiving a super-human level. This was achieved entirely by a reinforcement learning algorithm similar to Q-learning which approximated its value function using a single-hidden layer Neural Network [1].

In January of 2013, the company DeepMind published a paper called "Playing Atari with Deep Reinforcement Learning" [2]. This publication presented an agent able to learn to play Atari games using as input just the screen pixels. For this, they developed what they called a Deep Q-Network, which is a variant of the classic Q-learning algorithm.

Later in 2015, this same company developed AlphaGo, an agent capable of playing the Chinese strategy board game Go. It was the first program to

defeat a Go world champion [3] and it was trained with thousand of games from amateur and professional players to learn how to play the game. Recently, they improved this algorithm into what they call AlphaGo Zero. This new agent was not trained using human data, but simply by playing games against itself [4].

In January 2019, DeepMind revealed AlphaStar. This AI system was able to win a professional player at the game StarCraft II using deep supervised learning, reinforcement learning and evolutionary computation. This agent was trained directly from raw game data and was able to show extremely complex behaviours [5].

1.4. Report structure

This chapter consisted on a brief introduction to this report. Chapter 2 will explain the main techniques used for the development of the agent. Chapter 3 will discuss the technologies and the actual implementation of the agent. Chapter 4 will show and discuss the results that it obtained and some possible improvements that could be made. Finally, Chapter 5 will discuss some conclusions about the work done. This report will be closed with the bibliography and Appendix A, where one could find some UML diagrams from this project.

Chapter 2

Background

2.1. Reinforcement learning

Reinforcement learning is an area of machine learning where the agent learns to take actions in an environment to maximize a given reward. In contrast with supervised learning, the desired output for a given input does not need to be provided. The environment is usually modeled following the Markov assumption, which implies that the probability of future states depends only upon the present state and not on the sequence of events that preceded it (i.e. the process is memoryless).

Considering ξ to be an environment formed by an emulator, a sequence of actions and rewards and also considering $A = \{1...K\}$ to be the set of game legal actions that the agent can perform. When an action is said to be performed, it means that it is emulated, modifying the internal state and game score of ξ.

When the agent decides to perform an action, the internal state is not observed by it though. It observes just an image $x_t \in \Re^d$ from the emulator, in this case a matrix of raw pixels from the game screen. In addition, a reward r_t is given to the agent, which may depend on the prior sequence of actions and observations made by the agent. The agent must interact with its environment

by selecting actions in a way that maximizes future rewards.

2.2. Q-learning

Let be S and A two finite sets of states and actions. Then a function Q can be defined such as:

$$Q : S \times A \rightarrow \Re$$

Before learning begins, Q is initialized arbitrarily. After each step t, the agent which is in the state s_t takes an action a_t and receives a reward r_t given by $Q_t(s_t, a_t)$ advancing to the next state s_{t+1}.

The function Q is updated iteratively using the following policy:

- The agent observes its current state s_t.
- It selects an action to perform a_t.
- The agent receives a reward r_t and enters the next state s_{t+1}.
- Adjust Q_t values using a learning factor α_t and discount factor γ_t according to Bellman's equation:

$$Q_{t+1}(s,a) = \begin{cases} (1-\alpha_t)Q_t(s,a) + \alpha_t[r_t + \gamma \max_a Q(s_{t+1},a)] & \text{if } s = s_t \text{ and } a = a_t \\ Q_t(s,a) & \text{otherwise} \end{cases}$$

The discount factor γ determines the importance of future rewards. It is multiplied by the greatest Q-value that can be achieved at s_{t+1}. A γ factor of 0 means that the agent will only consider the current reward. As γ approach 1, actions will be made looking for a long-term high reward, at cost of propagating error and instabilities [6].

Broadly speaking, what the Q-learning algorithm is doing is a look-up table where you can see what is the reward for any given action and state. Then, the

agent can simply perform a greedy approach and always perform the action with the highest reward for its current state.

2.3. Neural Networks

Neural Networks are computing system made of several processing units called neurons working in parallel. Neural Networks are used to perform several tasks, such as pattern recognition, classification, natural language processing, computer vision, etc.

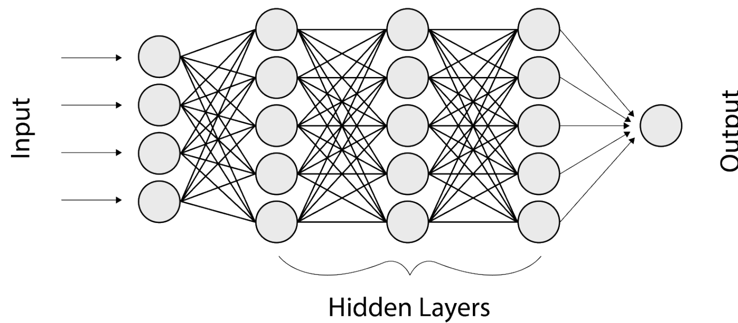

Figure 2.1: Example of Neural Network [7].

If a Neural Network consists in several layers of neurons, structured in layers forming a mesh where the output of some neurons is the input to others, it is called it a Deep Neural Network (DNN). Neurons on this networks perform a nonlinear function on their inputs, feeding forward their outputs until they reach the output layer [8].

2.3.1. Neuron

The Neural Networks are made up of neurons connected to each other. Each neuron has a weight value ω associated with each connection that states the importance of this connection by multiplying it by the input x value of the neuron. Positive weights activate the neuron while negative ones inhibit

it. Each neuron also has what its known as its bias value b, which can be seen as the activation threshold of the neuron.

In addition, each neuron has an activation function φ that is normally non-linear which takes as input the synaptic potential h of the neuron, which is the sum of the original input x multiplied by ω minus its bias b. This gives as result the final output of the neuron y [9].

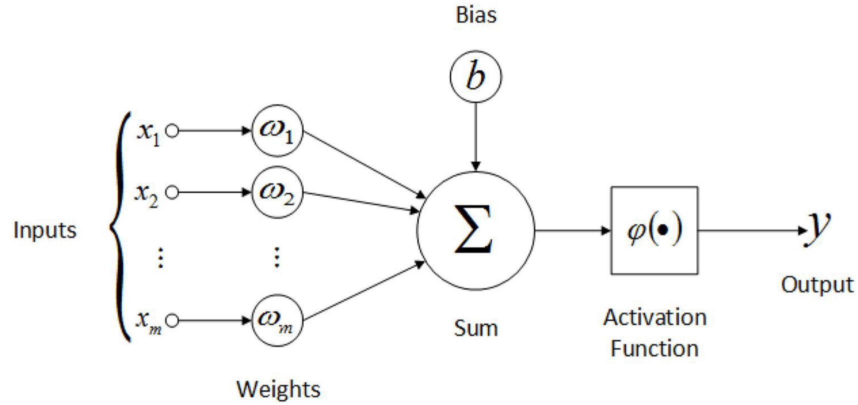

Figure 2.2: Artificial neuron [9].

2.3.2. Learning Process

The learning process of a neural network consists in the progressive modification of weight values w and biases b to minimize the mean error of the network. This operation is a iterative process of forward and backward data propagation.

The first phase, forward propagation, consists in the passing of the input data x across the network to predict its final output y. When this is done, a loss function is used to estimate the loss (or error) on the prediction compared to the expected value (in the case of supervised or reinforcement learning). The network goal is to minimize that error, which is done by gradually adjusting each neuron weights and biases.

The second phase, back propagation, consists in the propagation of the

error from the output layer of the network back to the input layer, adjusting each neuron weight to minimize the overall network loss. For this, a technique called gradient descent is used:

$$\Delta w_j(k) = -\eta \frac{\partial E}{\partial w_j(k)} = \eta \cdot [z^k - y(k)] \cdot g'(h) \cdot x_j^k$$

The above expression indicates how each weight w_j is going to be updated at iteration k, where η is the learning rate, z^k is the expected output, $y(k)$ the actual output of the neural network, $g'(\cdot)$ is the derivative of the activation function, h is the synaptic potential of the neuron and x_j^k is the neuron input[10].

2.4. Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a Deep Learning technique which can take as input a multidimensional matrix (e.g., an RGB image) instead of a flat vector. This gives the network the property of capture the spatial and temporal dependencies of elements in an image, such as rotations, scale and movement (if a series of images is provided as input).

2.4.1. Input data

As previously mentioned, the input data is a multidimensional matrix of numbers. This allows to the network to recieve data in an RGB format, a sequence of images, RGB-D images, etc. The role of CNN is to transform the input image into a form easier to process for the network without losing any critical features [11].

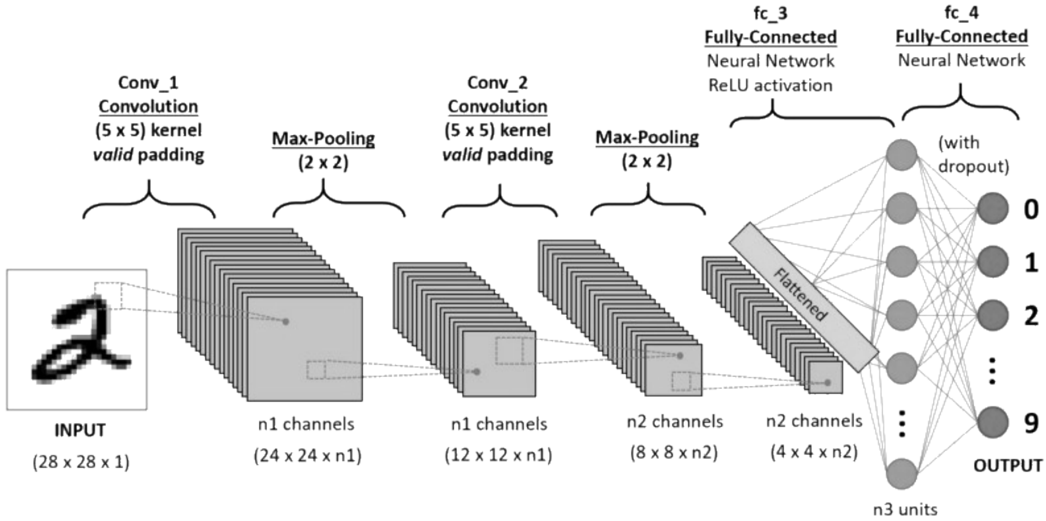

Figure 2.3: Structure of a Convolutional Neural Network [11].

2.4.2. Filter

The main operation involved in CNN are convolutions, which consist in a series of matrix multiplications between a kernel or filter and the incoming image. The filter has the same depth as the input image and moves along it making a matrix multiplication for each pixel in the image. All results are then summed with the bias to generate a new one-depth image.

This process expects to extract the high-level features, such as edges and shapes, from the input image and since it generates a new image, its output can be forwarded to the next convolutional layer to try extracting even more complex features.

2.5. Deep Q-Learning

As previously explained, Q-Learning consist in creating a look-up table for each pair state-action. Although this technique performs well, it is not scalable. Luckily, when the state of possibilities S is either continuous or unfeasible to be stored in a table, it can be successfully modeled by a neural

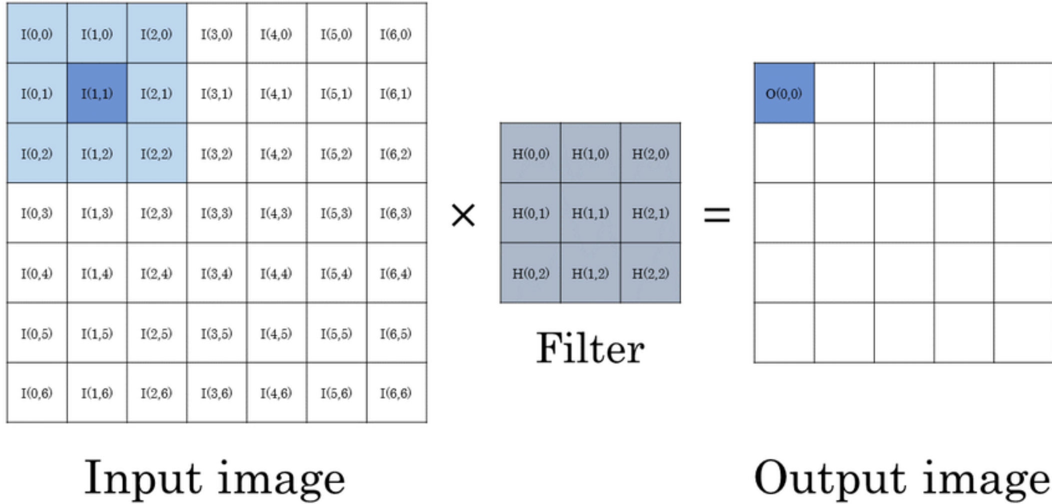

Figure 2.4: Example of convolution. Here, the filter has been applied to the highlighted region of the image. It will move along each row from left to right, generating a new single-depth image as output [12].

network (DQN). This network will take as input a state s_t and return a list of rewards $[r_{t1}...r_{tn}]$, which is the reward for each action a of the set of actions $\{a_1...a_n\}$.

Nevertheless, the fact of using a neural network creates some problems to be considered. If the network takes as training inputs its current state while playing, due to the fact that every action affects the next state, the sequence of inputs generated will have a strong correlation. In addition, states that have been seen at the beginning of the training can be progressively forgotten. This issues will be discussed below.

2.5.1. Experience Replay

At any time step t of the simulation, the agent receives a tuple (s_t, a_t, r_t, s_{t+1}) that is called experience. The agent will learn from this experience and go on to the next state s_{t+1}. This experiences have the problem of being sequential samples from the interactions of the agent with its environment, which causes it to overwrite old experiences with new ones.

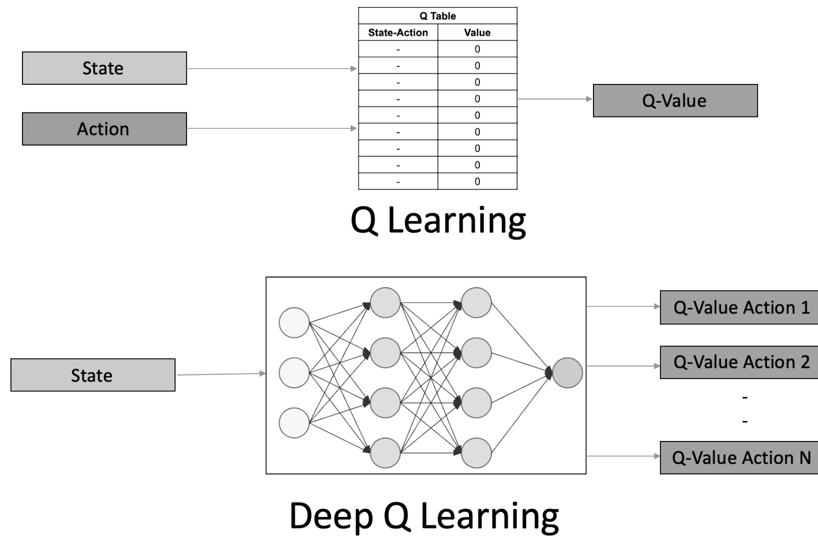

Figure 2.5: Q-Learning vs. Deep Q-Learning [13].

This problem can easily be solved by the use of a replay buffer. This buffer will store old experience tuples while the agent keeps interacting with its environment. Each game step the agent will be trained using a small batch of experiences from the replay buffer. This allows the network to keep training on old experiences, that way preventing it from forgetting previously seen tuples.

Another problem to consider is the strong correlation between experiences: each experience was generated by the previous one, so if the agent is trained using data from the memory buffer sequentially, it can be influenced by their correlation. In order to prevent this, training batches should be extracted from the memory buffer at random, breaking their correlation. This will prevent the network from oscillating or diverging [14].

This technique allows the agent not to train directly from its environment, which lets the agent to try different things in order to explore the state space, saving those experiences in the replay buffer for later usage. This way the agent avoids being just in one region of the state space, thus having this way a better set of experiences.

2.5.2. Learning Process

The network should try to minimize its error for each output predicted. As previously mentioned, Q-Learning did this using Bellman's equation. The DQN will take advantage of the stochastic gradient descent technique and will update its weight using the following equation:

$$\Delta w_t = \alpha[(r_t + \gamma max_a \hat{Q}(s_{t+1}, a_t, w_t)) - \hat{Q}(s_t, a_t, w_t)]\nabla_w \hat{Q}(s_t, a_t, w_t)$$

As it can be seen, in order to calculate the network loss, the difference between the target Q-value and the estimated Q-value is used, but the real target is unknown. The Bellman's equation shows that the target value is the reward of taking the action a_t at the state s_t plus the highest Q-value of the next state s_{t+1} discounted by some factor.

However, the network is using its parameters for estimating both the target and the Q-value. This produces a big correlation between the target and the network weights updated, making both values shift at every training iteration. This leads to a big oscillation in training and slow convergence.

To prevent this, a technique called fixed Q-targets is used. The agent will be trained using another network with fixed weights w^- that will be used to estimate the targets value to train the agent network. After a number τ of training steps, the DQN network weights w will be copied to the target network.

$$\Delta w_t = \alpha[(r_t + \gamma max_a \hat{Q}(s_{t+1}, a, w^-)) - \hat{Q}(s_t, a_t, w_t)]\nabla_{w_t} \hat{Q}(s_t, a_t, w_t)$$

And every τ steps, the DQN weights are copied into the target network: $w^- \leftarrow w$

2.6. DOOM (1993 video game)

DOOM (1993) is the first release of the Doom series. It is a first-person shooter (FPS) developed by id Software for MS-DOS and it is considered one of the most influential games in its genre and the whole video game history, since it popularized the FPS, pioneered technologies of immersive 3D graphics, used the business model of online distribution and promoted networked multiplayer gaming.

In DOOM, the player is an unnamed space marine who must fight his way through hordes of monster and demons from Hell. For this purpose, the player must manage supplies of ammunition, health and armor that can be picked up. Monsters have very simple behaviours: either they move towards the player to bite, claw or hit him or just throw fireballs and rockets at him. Levels are often labyrinthine and contain several traps such as crushing ceilings, fire and toxic waste pits.

Figure 2.6: Doom gameplay screenshot.

In 1997 the game's source code was released, leading to multiple fan adaptations. The game was ported to countless other platforms and its fan base grew even more. Recently, some libraries has been created for machine learning purposes, such as VizDoom, due to the lightweight, low resolution and fast performance of the game.

Chapter 3

Implementation

3.1. Technologies

3.1.1. Keras

Keras is a high-level neural networks API written in Python. It provides a friendly interface that helps fast development and prototyping[15].

Keras is able to run over TensorFlow, CNTK or Theano. For this project, TensorFlow was finally selected due to its high popularity, maintenance and recently standardizing its high-level APIs to match those on Keras[16]. The fact that TensorFlow support GPU hardware acceleration also was an important argument considered.

3.1.2. VizDoom

VizDoom is a library that allows to get the screen buffer information of a copy of the 1993 game Doom. It is primarily intended for research in machine visual learning since it allows to easily gather the visual information of the game (screen buffer). In this project, VizDoom is used as the environment of the agent. Its API provides ways to perform actions as the player in the game, access the RGB channels of the screen, gather information of each step

of simulation, such as the score for a given action and more [17].

It is based on ZDoom, which is currently the most popular modern source-port of DOOM engine for running on modern operating systems under the GPL licence [18]. This means that VizDoom is compatible with a range of tools and resources that can be used to create custom scenarios. In this case, the agent will use scenarios that were made so that the agent had to fulfill a task and have a restricted set of movement for those. This two scenarios will be described in Chapter 4.

3.1.3. scikit-image

In order to preprocess the images extracted from the screen buffer scikit-image was used, which is a collection of image processing algorithms. In this case, it was used to reduce the resolution from VizDoom frames and increase the contrast of those images.

3.2. Hardware specification

As previously mentioned, both Keras and TensorFlow take advantage from GPU hardware acceleration. For this reason, this agent was trained using a GTX 1060 6GB, Intel i7-8750H, 16GB DDR4, 256GB NVMe SSD laptop.

3.3. Input processing

Each simulation step, a new frame is added to the screen buffer. This data consist in a RGB image of $160x120$ pixels. Nevertheless, this information alone is not enough: a single frame image does not provide any information about motion. One way of solving this issue is using a LSTM neural network [19]. These networks have a recursive structure, allowing information to persist from one iteration to another. However, a simpler approach can be used: instead

of using a single image as input, several frames will be stacked together. This allows to the DQN to infer a sense of where and how fast objects are moving.

Nonetheless, this comes with some drawbacks: just as the input dimensionality increases, so too the DQN increases in complexity. In order to prevent this, the size of the input data must be reduced as much as possible. First, all the game frames can be converted to gray-scale. This reduces by three times the input size. In addition, the agent field of view contains some areas with little to no information or just noise, so those sections can be cropped out and increase the overall image contrast in order to reduce noisy patterns such as walls, ceilings, floors, etc. Finally, to further reduce the input size, the image is scaled down to $42x42$ pixels, just enough to still be able to discern the target enemies.

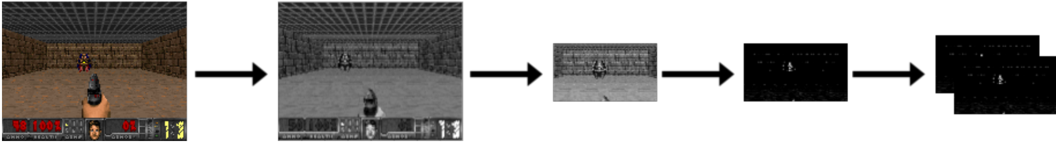

Figure 3.1: Input processing pipeline.

This processing is vital to the agent training algorithm due to memory issues. If two stacked frames were considered as the input, each experience would consist in 2x42x42 unsigned integers. Considering that each unsigned integer is 1 byte long, if the memory buffer contains 1.000.000 experiences, regardless processing each frame, it would use almost 3.3GB of RAM memory and some problems require as much as 10.000.000 experiences [2].

3.4. Agent architecture

The agent consists in a Deep Q-Network of five layers. It takes as input a 2x42x42 vector representing an experience and outputs a 1xN vector of Q-values, where N is the number of possible actions. The agent DQN architecture

can be divided on two main parts.

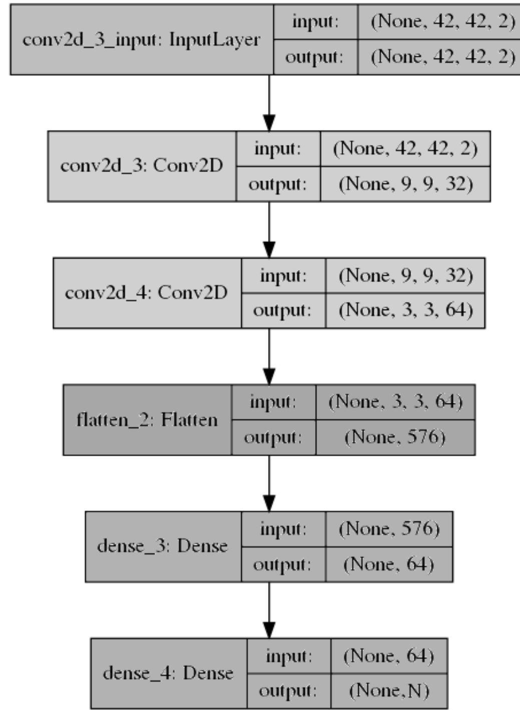

Figure 3.2: Model graph.

The first part consist of two convolutional layers in charge of processing the input vector. Since it is a 2x42x42 vector, it can be interpreted as an image with two color channels and that is exactly what these convolutional layers do. These layers try to extract high level information about the image to be later processed by the second part of the network. The first convolutional layer has a kernel of 8x8 and uses 32 filters, reducing the input dimensionality to 9x9x32 and the second one uses a kernel size of 4x4 and 64 filters. The final output is 3x3x64, which hopefully contains all the high-level features.

The second part of the DQN is in charge of predicting which will be the Q-value associated with each action. To do this, the output of the convolutional layers must be converted to a single dimension vector by a flatten layer. Then, this vector will go through two densely-connected layers. The first one consist of 64 neurons and the second one of N output neurons, where N is the number

of possible actions the agent can perform.

3.5. Populating the Replay Buffer

As previously explained, the agent will not be trained using the current environment, but instead it will use a small batch of experiences from the replay buffer. However, when the agent starts learning, the replay buffer does not contain many experiences and those, although randomly selected, will probably have a great correlation between them, which may cause the agent to learn slower.

To solve this problem, the replay buffer is populated with some experiences gathered by performing random actions. This experience gathering process is quite fast since the DQN is not trained during this phase.

3.6. DQN parameters

Our DQN approach has multiple parameters to be considered, not only related to the model architecture but to the Deep Q-Learning algorithm, network training and replay buffer as well.

In order to explore most of the state space, the agent must try actions that can lead to a state that it has never been. The simplest way to do this is using a parameter ϵ. When the agent needs to know which action to perform, it will ask the DQN to provide the best action, but it will also have a probability ϵ of not taking that action and instead doing a random one. Epsilon should start high at the beginning and slowly decrease towards zero. This will lead to an initial exploration of the state space, but also allows the DQN to converge. This parameter ϵ is updated every game step t according to the following function:

$$\epsilon_t = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})e^{-t\epsilon_d}$$

Where ϵ_{max} is the initial value of epsilon (1.00), ϵ_{min} is the minimum value that epsilon can get (0.01) and ϵ_d is a decay value. This function follows

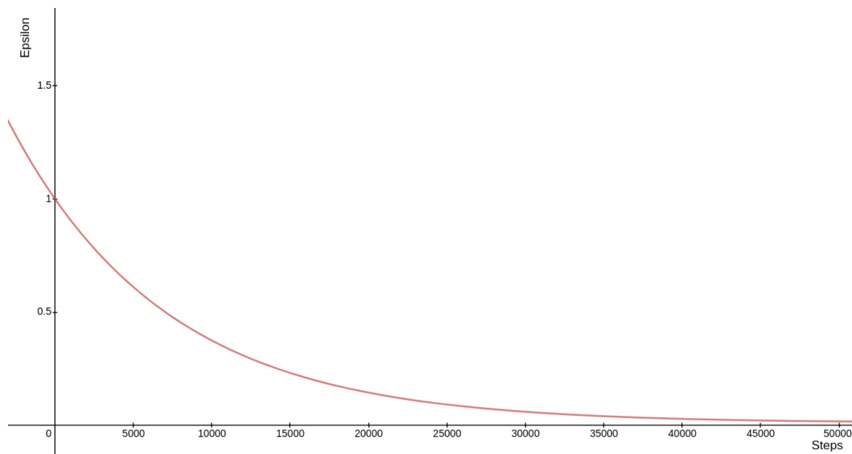

Figure 3.3: Update function of ϵ.

Another parameter to consider is the discount factor γ. This value is especially important, because in most of cases, the actual action of the agent will not reward it with a positive value, but it will lead it to the actual goal. For this reason, a value of 0.95 is used.

The DQN learning parameters are related to its training. The first one is the batch size. The batch size determines the number of samples that will be propagated through the network. The bigger this value, the better will be the estimated gradient, making the network to fluctuate less. Nevertheless, a smaller batch size means faster epochs and less required memory as well. This is especially important due to the size of the dataset [20]. In this case, a batch size of 32 samples has been used, which seems a good trade-off between smaller fluctuations and performance.

Regarding the replay buffer, it has a max length of 100.000 experiences, which may seem a lot but it is not as previously mentioned. It is populated with 5000 experiences before starting the actual DQN training to minimize correlation. A higher number of base experiences does not seem to increase performance since randomly selecting an action does not lead to any complex

behaviours. In addition it causes the agent to follow a kind of Gaussian series of actions where it tends to stay in the middle of the screen due to the same chances of going to the left or right.

3.7. Main structure

The main section of this program consists in the training of the agent. This is done using an endless loop where the agent keeps exploring the state space, interacting with the environment and storing experiences in the replay buffer, while each frame the DQN is trained as well using those stored experiences.

Every time the agent performs an action, its environment emulates it properly and gives back an new state and reward. This data is what will be stored in the replay buffer.

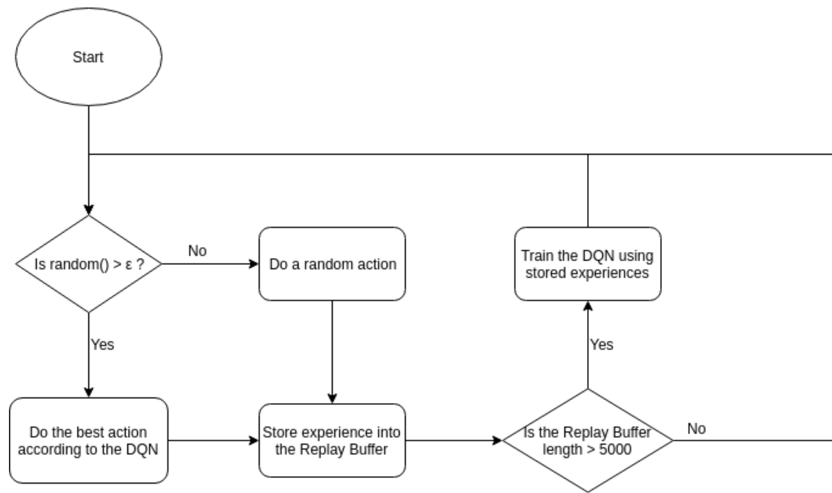

Figure 3.4: The agent keeps training endlessly, using ϵ to explore the states space.

Furthermore, as previously stated, the replay buffer is being populated with 5000 experiences before starting any learning process. Those experiences are just gathered performing random actions, which in this diagram is represented by the fact of ϵ being 1 at the beginning of the program execution. This

value will only change when the network actually start training and it will not happen until the replay buffer has at least 5000 experiences stored.

In order to stop the program without losing any behaviours learned by the agent, the DQN weights are saved every 10 episodes into a file using the Keras API. If those weights wanted to be used, it would just be necessary to compile the DQN and load them using Keras again.

Chapter 4

Results

As previously stated, the main goal of the agent is to perform a series of tasks using VizDoom as environment. These tasks were divided in two different maps that will be called respectively ”Basic” and ”Defend the center”. These maps are provided as part of the VizDoom library [21].

4.1. Basic

This scenario consist in a rectangular room with gray ceiling and brick walls. The agent is spawned near one wall and a monster which is spawned randomly somewhere along the opposite wall. The agent can only perform three actions: go left, right and shoot. The episode finishes either when the agent kills the monster or the time is out after 100 steps.The monster takes just one hit to be killed and does not attack the agent.

The main goal of this scenario is to show if the DQN can develop a complex behaviour such as determining where the monster is, align itself to aim and hit it. This might seem a trivial task to do, but remember that the agent is just using raw pixels from the screen to perform all of this. To encourage this, several rewards are given to the DQN.

The agent will receive a reward of +101 points for killing the monster. This

Figure 4.1: Frame from the agent training in the ”Basic” scenario.

is the main incentive to guide the agent. However, other factors will be taken into account as well. The agent will receive -5 points every time it shoots and misses the monster. This way, the agent will try to shoot just when it is sure to hit the monster. In addition, each step in the simulation without hitting its target, the agent will receive -1 points to encourage it to be as fast as possible performing its task.

Results where taken after nearly two hours of training, gathering more than 100.000 experiences. To visualize the improvements made by the agent along its training, four graphs where made. The first one correspond to a totally random behaviour. The next three ones are the results after 1000, 2000 and 3000 episodes each.

The high scores achieved in some episodes by the totally random behaviour and the early stages of the agent are caused when the target appears in front of the agent and it randomly hits it. Nevertheless, it can be seen how the agent become more consistent after more episodes of training. After 3000 epochs, the mean score of the agent did not seem to improve, which probably mean there was not any margin for further improvement.

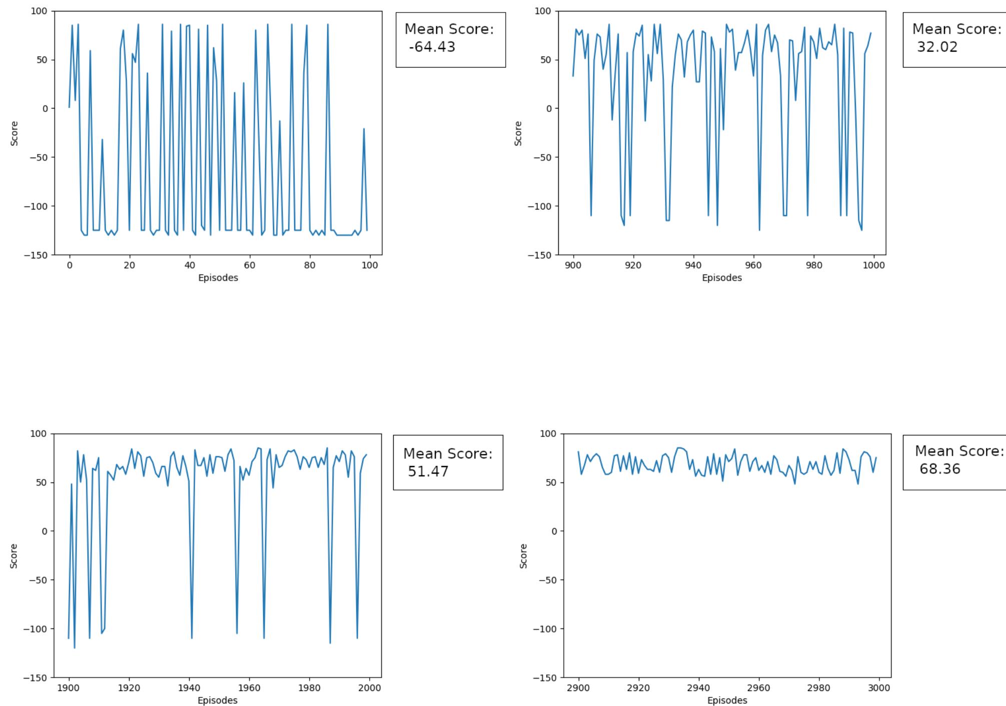

Figure 4.2: Results from the "Basic" scenario. Each graph also shows its mean score.

4.2. Defend the center

This second task consists in a more complex version of the previous one. It consists in a large circular room. The agent is spawned in the exact center and five melee-only monsters are spawned along the walls. Every time the agent kills one monster, it spawns again after some time in another random location near the walls. The episode finishes when either when the agent is killed or the time is out after 500 steps. Ammunition is limited.

The goal of this task is to kill as many monsters as possible, which means that the agent should not let be killed by allowing the enemies to come too close and should not waste ammunition unnecessarily.

The agent will receive +1 points every time it kills an enemy by hitting it with one shot. It will get -1 points if it dies in that episode too. This means

Figure 4.3: Frame from the agent training in the "Defend the center" scenario.

that in order to increase its score, it needs to last long enough to the killed enemies to reappear and have enough ammunition to kill them as well.

New challenges appear as well, since the agent does not have access to all the information about the environment: enemies can be on its back. This means that, although no enemies may be in sight, the agent must actively look for them (without spastically wasting ammunition). Distance to the enemies must be taken into account as well. Due to the fact that the agent can only move left or right a discrete amount, sometimes it must wait until the target comes close enough to correctly aim and shoot it.

Episodes took much longer to complete compared with the previous task, due to the fact that, opposite to what happened in the "basic" scenario, as the agent improved it meant that it would take longer to be killed and thus to finish. After 250 episodes, almost 150.000 experiences where already stored and 2 hours had passed.

As previously mentioned, this results took more time to gather than the previous ones due to the greater length of each episode, which mean that the

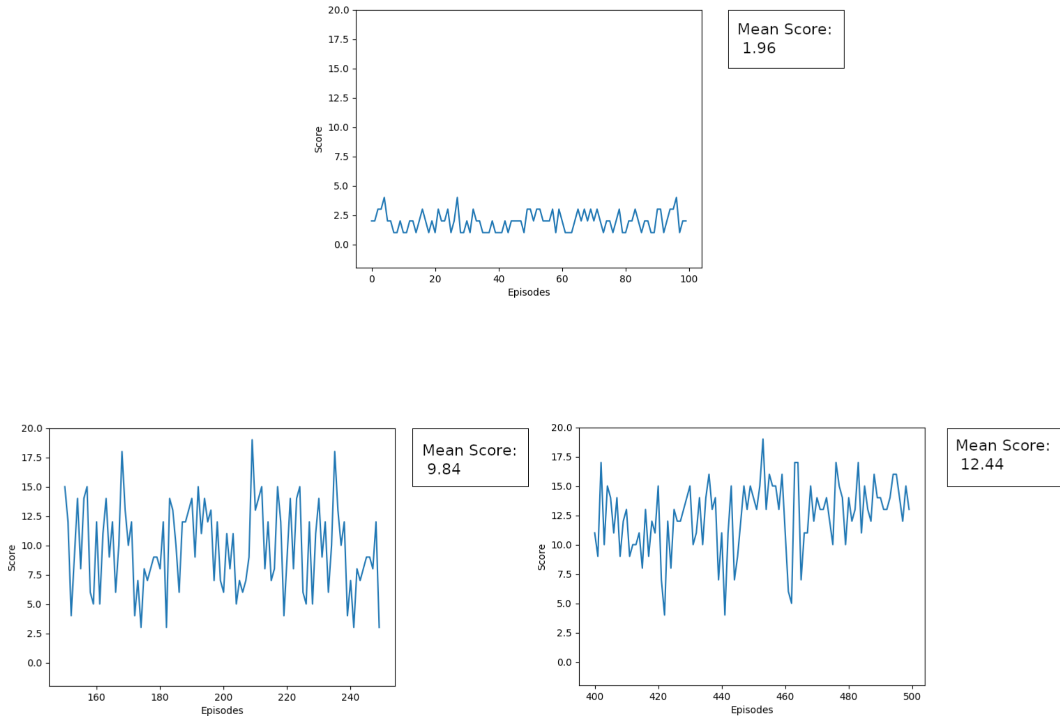

Figure 4.4: Results from the "Defend the center" scenario. Each graph also shows its mean score.

network also trained more for each episode. Similar to the previous results, the first graph shows the score of a totally random approach.

The main problem that the agent had to learn was not to waste ammunition, since it was able to find and kill the enemies without any troubles. Nevertheless it usually tend to shoot sometimes randomly. This may be caused by the low resolution of the input data, that makes difficult for the DQN to difference a far enemy from the background wall. However, it was not viable to increase the resolution without drastically impact the overall training performance.

4.3. Improvements

Further improvements can be applied to this project. Although these were not too difficult to implement, they would have meant a significant increase

in processing power requirements, so that is why they were not considered for this project. Below are highlighted some of them.

4.3.1. Double DQNs

Introduced by Hado V. Hasselt in 2010 [22], this approach attempts solving the problem of estimating which is the action with the highest Q-value for the state s_{t+1} in Bellman's equation.

This issue is caused by the fact that the agent do not have enough information about which is the best action to take and this can lead to a higher Q-value than the optimal best solution and a difficult learning process.

The solution to this is using two different networks: one that calculates the best action to take for the state s_{t+1} (the one with a higher Q-value) and another one to calculate the actual value of taking that action at the state s_{t+1}.

$$Q(s_t, a_t) = r(s, a) + \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a, w), w^-)$$

This helps reducing the overestimation of Q-values and therefore makes training faster and more stable.

4.3.2. Dueling DQN

According to the Bellman's equation, the DQN tries to calculate a Q-value corresponding to how good is an action a_t given the current state s_t and how advantageous is to be at the next state s_{t+1}. Thus, Bellman's equation can be rewritten as follows:

$$Q(s_t, a_t) = A(s_t, a_t) + V(s_t)$$

Where $A(s_t, a_t)$ is the advantage of taking that action at that state and

$V(s_t)$ is the value of being at that state. To estimate this functions, two new streams will be added to the model architecture and will be combined using through an aggregation layer. This is what is called a Dueling DQN (DDQN).

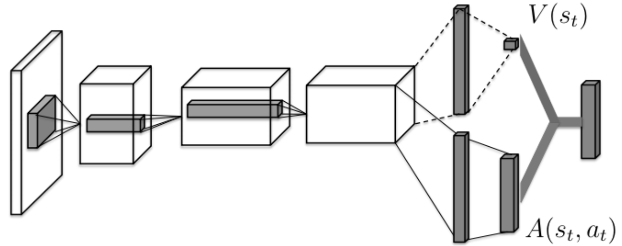

Figure 4.5: Model architecture of a DDQN [23].

This will allow the DQN to learn which states are valuables or not without having to learn the effects of taking an action at that state. This is very useful for states where actions do not affect the environment in a relevant way [23].

4.3.3. Prioritized Experience Replay

When experiences are stored, some of them may be more important than others although being less frequent. This is an issue considering that sampling consist of randomly select a batch out of the replay buffer uniformly, so these important experiences will hardly ever be selected.

Prioritized Experience Replay (PER) [24] consists in giving each experience a probability of being selected p_t, changing this way the sampling distribution. This value will depend on the magnitude error of the estimated value since it means that the network still has to learn that experience and will be saved in the replay buffer as well.

$$p_t = |\delta_t| + e$$

Where $|\delta_t|$ is the magnitude of the error and e is a constant value to assure that every experience has at least a probability higher than 0 to be chosen.

An stochastic prioritization should be used to sample the replay buffer:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

Where a is an hyperparameter that introduces some randomness in the experience selection. If $a = 0$ it would be a pure uniform distribution and if $a = 1$ it would only select the experiences with the highest priorities.

Nevertheless, this produces a new problem: the fact of sampling not being purely random introduces a bias toward high-priority experiences, increasing the risk of overfitting. To correct this, importance sampling weights can be used. This weights will be updated every time that the experiences are sampled.

$$(\frac{1}{N} \cdot \frac{1}{P(i)})^b$$

Where N is the size of the replay buffer and b is a parameter that controls how much these importance weights affects learning. It starts close to 0 at the beginning and annealed up to 1 over the duration of training.

4.3.4. Higher-level input

The advancements on hardware are making possible the use of deep learning techniques to solve complex AI problems. However, these approaches still consume lots of memory and processing power. Although this work shows that using barely raw unprocessed data, it still needs a large dataset and training steps to achieve a satisfactory result. This problem could be reduced by feeding the network with higher-level data.

Currently, new techniques are being studied by companies such as DeepMind to decompose a 3D scene into components or units using just raw pixels as input [25]. This process is being used now as part of a reinforcement learning

approach to play the game StarCraft II to identify the background and units in the game [26]. This should improve data efficiency and transfer performance on any deep neural network.

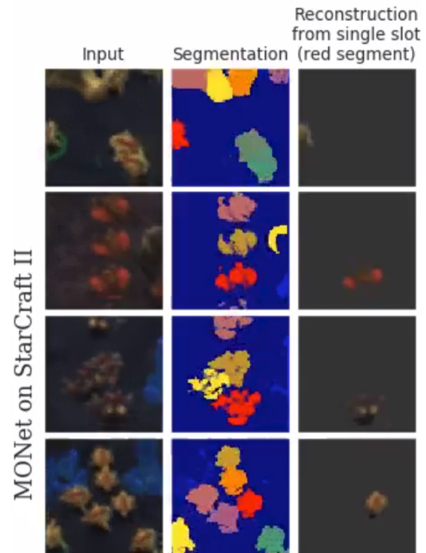

Figure 4.6: Segmentation of units from StarCraft II using a Multi-Object Network (MONet) [26].

Chapter 5

Conclusions

Through the development of this project, it was explained the implementation of an agent capable of successfully perform several tasks on the Doom game using a Deep Q-Network.

Results have shown that this approach was feasible, performing better in both scenarios as episodes went on. Nevertheless, as previously stated, there are several ways to theoretically improve these results in exchange of computational power.

Therefore, although being Deep Q-Learning a feasible technique to work with this environments and sensory data, it may consume an excessive amount of resources if it is not optimized through higher-features extraction or similar techniques.

Future work will tell, as more and more complex agents are being created using new Deep Reinforcement Learning techniques, such as AlphaStar [5], and future advances are still before us.

5.1. Technical problems encountered

Several problems were encountered through the development of this project. They are listed below:

- Firstly, drivers issues were found when installing the Keras library. In order to take advantage of the GPU, Keras must be able to use the CUDA library and had the correct driver versions. After installing these drivers Keras was finally able to use the GPU for hardware acceleration.

- Furthermore, memory was always an issue. The fact that the agent needed such a large amount of experiences meant that it would consume way more RAM than available. This finally was solved by reducing each experience from the original planned size of 84x84x4 to a more feasible one of 42x42x2.

  However, this still was not enough since the replay buffer could store too many experiences. Originally, after some research, an usage of 1.000.000 experiences was planned, but finally it had to be cut down to 150.000. This experiences contained a lot useless information from the walls, ceilings and floors. After some attempts of training, the agent was not able to learn how to perform the task. However, after increasing the contrast of each experience, it succeeded in both scenarios.

Capítulo 5

Conclusiones

A lo largo del desarrollo de este proyecto, se ha explicado la implementación de un agente capaz de realizar de manera exitosa varias tareas en el juego Doom utilizando una Deep Q-Network.

Los resultados han mostrado que esta propuesta es factible, mejorando en ambos escenarios conforme pasaban más episodios. Sin embargo, como se ha mencionado anteriormente, teóricamente hay varias formas de mejorar estos resultados a cambio de un mayor coste computacional.

Por lo tanto, aunque Deep Q-Learning sea una técnica factible para trabajar con este tipo de entornos y datos sensoriales, puede consumir una cantidad excesiva de recursos si no se optimiza mediante la extracción de características de alto nivel o técnicas similares.

Los futuros trabajos e investigaciones dirán, ya que agentes más y más complejos están siendo creados usando nuevas técnicas de Aprendizaje por Refuerzo Profundo, como AlphaStar [5], y futuros avances están aún por llegar.

5.1. Problemas técnicos encontrados

Varios problemas surgieron a lo largo del desarrollo de este proyecto. Éstos están listados a continuación:

- En primer lugar, se encontraron problemas relacionados con lo drivers al instalar la librería de Keras. Para que pudiera aprovechar el uso de la GPU, Keras debía poder usar la librería de CUDA y tener las versiones correctas de los drivers. Tras instalarlos Keras finalmente era capaz de usar la GPU para la acceleración por hardware.

- Además, la memoria siempre fue un problema. El hecho de que el agente necesitara una cantidad tan masiva de experiencias significaba que consumiría mucha más RAM de la disponible. Al final esto fue resuelto reduciendo cada experiencia desde su tamaño originariamente pensado de 84x84x4 a un tamaño más factible de 42x42x2.

  No obstante, esto seguía sin ser suficiente ya que el Replay Buffer podía almacenar demasiadas experiencias. En un comienzo, tras un poco de investigación, se planeó el uso de 1.000.000 de experiencias, pero al final tuvo que reducirse a 150.000.

  Estas experiencias contenían mucha información inútil para el agente sobre las paredes, techos y suelos. Tras algunos intentos de entrenamiento, el agente no era capaz de aprender cómo realizar la tarea. Sin embargo, después de incrementar el contraste de cada experiencia, obtuvo éxito en ambos escenarios.

Bibliography

[1] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[5] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, and et al. Alphastar: Mastering the real-time strategy game starcraft ii. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/.

[6] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier, 1995.

[7] Sean Noah, Joel Frohlich, and Alexandria Weaver. Machine yearning: The rise of thoughtful machines - artifical neural networks, Mar 2019.

[8] Carmen Alcaraz Garófano. Digital images analysis by deep learning techniques for melanoma diagnosis, Sep 2018.

[9] Jayesh Bapu Ahire. The artificial neural networks handbook: Part 4, Nov 2018.

[10] UMA Department of computer science and programming languages. Artificial neural network supervised learning., Sep 2018.

[11] Sumit Saha. A comprehensive guide to convolutional neural networks-the eli5 way, Dec 2018.

[12] Chaim Baskin, Natan Liss, Evgenii Zheltonozhskii, Alex M Bronstein, and Avi Mendelson. Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 162–169. IEEE, 2018.

[13] Ankit Choudhary. Introduction to deep q-learning for reinforcement learning (in python), May 2019.

[14] freeCodeCamp.org. An introduction to deep q-learning: let's play doom, Mar 2018.

[15] Keras. Keras: The python deep learning library.

[16] TensorFlow. Standardizing on keras: Guidance on high-level apis in tensorflow 2.0, Dec 2018.

[17] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. Vizdoom competitions: Playing doom from pixels. *IEEE Transactions on Games*, 2018.

[18] Daniel Gimmer, Rachael Alexanderson, and Xaser Acheron. About zdoom.

[19] Arthur Juliani. Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (a3c), Dec 2016.

[20] itdxer. What is batch size in neural network? Cross Validated. URL:https://stats.stackexchange.com/q/153535 (version: 2019-04-05).

[21] Mwydmuch. mwydmuch/vizdoom, Nov 2018. https://github.com/mwydmuch/ViZDoom/tree/master/scenarios.

[22] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[23] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[24] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[25] Christopher P Burgess, Loic Matthey, Nicholas Watters, Rishabh Kabra, Irina Higgins, Matt Botvinick, and Alexander Lerchner. Monet: Unsupervised scene decomposition and representation. *arXiv preprint arXiv:1901.11390*, 2019.

[26] DeepMindAI. Deepmind. Twitter, March 2019. https://twitter.com/DeepMindAI/status/1109135270035755011.

Appendix A

UML diagrams

A.1. Class diagram

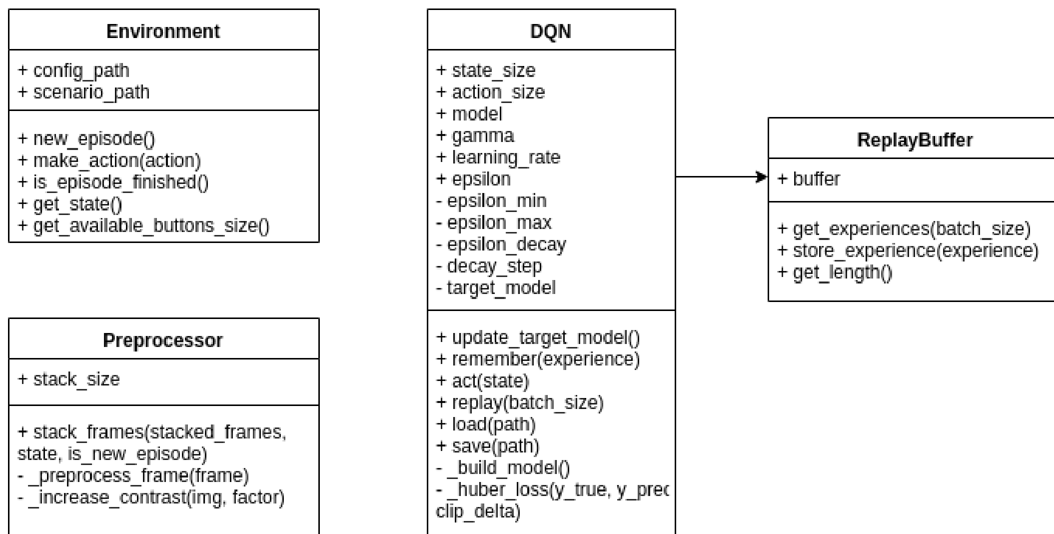

Figure A.1: Class diagram of the project.

A class Environment was created as an interface for the actual DoomGame class from VizDoom, reducing this way the complexity of the entity.

The main goal of the class Preprocessor is to isolate all the functions related to preprocessing each game frame: crop each frame, increase its contrast, downscale it and add it to the frames stack of each experience.

The DQN class contains the actual model of the neural network. It also has all the training parameters related to the Deep Q-Learning algorithm, methods to store experiences in the buffer and to train itself using a batch from the replay buffer. There are also a couple of methods to save and load the weights of the DQN.

Finally, the replay buffer is implemented as a third class, which contains a queue instance from python that is used as buffer and two methods to both store and retrieve experiences from it.

Although this last class might seem pointless, it is important to consider that other implementations Experience Replay might be more complex, such as Prioritized Experience Replay. These approaches may select experiences in a non-trivial way, a behaviour that should be conveniently isolated in the ReplayBuffer class.

A.2. Sequence diagram

In Figure A.2, a sequence diagram of one iteration of the main loop can be seen.

The agent retrieves the current state of the environment, stack this state with the previous one and uses the Deep Q-Network predictions to retrieve the next action. Then, it tells the environment to emulate that action, retrieving a new state and reward for it. This experience is then stored in the replay buffer. Next, the DQN is trained using a random batch from the replay buffer.

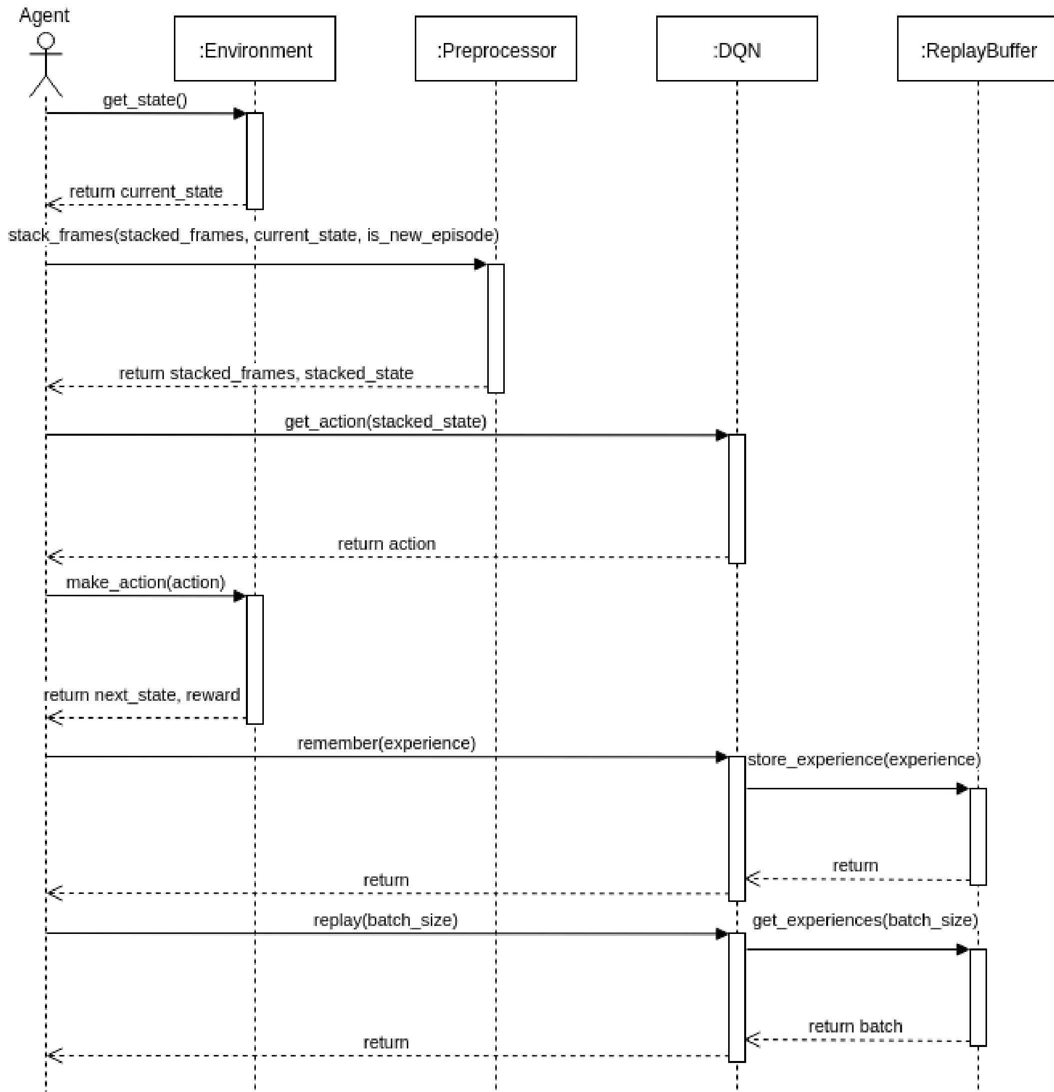

Figure A.2: Sequence diagram of one iteration of the main loop. All of this happens every game step.