Tay-Runner, an AI model based in Reinforcement Learning to play Geometry Dash game

1st Saul Navajas Quijano
Escuela de Ingeniería de Fuenlabrada
URJC
Madrid, Spain
s.navajas.2020@alumnos.urjc.es

2nd Iván Porras Estébanez
Escuela de Ingeniería de Fuenlabrada
URJC
Madrid, Spain
i.porras.2020@alumnos.urjc.es

3rd Juan Miguel Valverde García
Escuela de Ingeniería de Fuenlabrada
URJC
Madrid, Spain
jm.valverde.2020@alumnos.urjc.es

4th Adrián Madinabeitia Portanova
Escuela de Ingeniería de Fuenlabrada
URJC
Madrid, Spain
a.madinabeitia.2020@alumnos.urjc.es

***Abstract*—The fields of Artificial Intelligence and Machine Learning has witnessed remarkable advancements in a considerable wide variety of domains in recent years. Specially, Reinforcement Learning has become a fundamental tool in multiple fields of application, such as Video Games Industry. Its usefulness spans from evaluating game difficulty, the feasibility of its levels or applying it for analytical, experimental and critical purposes. In this article, a Computer Science research is described, focusing on the mastery of a platformer video game – Geometry Dash – using sophisticated techniques, including Q-Learning, image analytics and high data collection.**

***Index Terms*—hyperparameters, learning policies, frames per second, training model.**

I. Introduction

In the current times, new generations grow and develop though technology and smart devices. For most of them, video games aid in relaxation, creativity and cognitive development. Over time, the integration of AI and its fields in them has expanded. However, the introduction of these fields into video games suggests multiple questions to be addressed: whether it is possible to create a model for any type of level to be tackled, how much training time is required for this, or if it is possible to create a generalized model capable of overcoming a wide variety of levels. Although many video game creators are receptive to the incorporation or utilization of AI in their creations, some of their fields such as Reinforcement Learning or Q-Learning can prove to be highly valuable in their development, experimentation and analysis. Reinforcement Learning can be described as a strong learning algorithm that learns the optimal policy through interaction with the environment without its model, using an agent that acquire the knowledge of the value function for an specific given policy (learning policy). Within this field, Q-Learning is defined as a specific algorithm that learns a Q-function (action value function) guiding the agent in its decision-making process over all the sucessive steps (states), starting from the actual state, in a specific environment. It was first introduced in 1989 [1] as a mathematical test to understand the learning and training processes of animals by rewarding them when it responds appropiately to commands or behaviours. In addition, they are also penalized every time they make a decision or action that is counterproductive or produces an undesirable outcome. The algorithm establishes an optimal policy, maximizing the expected value of the total reward over all steps, for any mathematical framework used to model decision-making in situations where the agent interacts with an environment over a sequence of discrete time steps. This framework is also known as a Markov Decision Process (MDP) [2].

In this brief, we will propose a model able to complete satisfactorily a level of the platformer video game Geometry Dash, based on a Q-Learning agent which is rewarded proportionally to the time it stays alive and, in turn, penalized every time it performs or takes a non-optimal or undesirable action.

II. Related Work

In Dueling Network Architectures for Deep Reinforcement Learning [3], the authors delve into the model of Adversarial Networks and highlight their advantages over the conventional single-layer linear structures. This paper serves as a foundational reference in understanding the benefits of neural networks, in capturing complex relationships.

A significant contribution is found in laying Atari with Deep Reinforcement Learnin [4], a foundational document that introduces the application of reinforcement learning to play Atari games. This work provides valuable insights into leveraging reinforcement learning techniques for gaming scenarios.

Additionally, an insightful study was identified on Medium, where the focus was on training an agent to play the Google Chrome dinosaur game. This medium article [5] contributes practical knowledge and showcases the adaptability of reinforcement learning in diverse gaming environments.

These references collectively contribute to the understanding and application of advanced AI techniques in gaming scenarios, providing a rich foundation for exploring similar methodologies in the context of Geometry Dash.

III. Agent Enviroment

The environment can be defined as the set of methods, functions, and techniques that allow the agent to interact with the program from which it wants to learn in order to perform a logical behavior that maximizes the performance measure. In our case, this program corresponds to the video game Geometry Dash. This environment is divided into several modules as can be seen in Figure 1.

A. Used tools

For the definition and implementation of this environment, several tools have been used that allow or facilitate the training and testing of the different models:

- Geometry Dash: The base platform that serves as a source of input and output data to the network
- Cheat Engine 7.5: A memory scanning and editing tool utilized to extract memory positions that contain data of interest
- GD Overlay: Game mod with different tools

B. Data collection

The collection of game data is essential as it will provide our model with enough information to be able to improve and adapt to different situations. Extracting data from a no open-source video game such as Geometry Dash can be complicated, the methodology applied for this case focuses on memory scanning through the Cheat Engine tool to be able to obtain the variables that are of interest to us, in this case: the number of frames, the percentage of level progress and the speed of the game. The Gd Overlay tool also has considerable importance in this part. It loads the values of these variables into static positions (static pointer + offset) in the process's virtual memory, which avoids scanning with each new run of the program.

To create a user-friendly and scalable environment, programming techniques based on object orientation have been used. Specifically, the data collection functions have been encapsulated in a "GdDataReader" class. This class is responsible for reading the variables that are properly mentioned and capturing the image of the game. This image will also be an essential element in the execution of the agent as described in the "Proposed Model" section. This class also contains some security components that will be explained in more detail in the subsection "Security mechanisms".

C. Output control

The agent will only have two possible actions, jump or stand. To interact with the game and be able to insert these actions, a class similar to the one mentioned in the previous sub-section has been implemented, which simulates the jump signal. An alternative approach involves the development of a module (.dll) that loads alongside the game process, enabling direct interaction and providing an additional avenue for addressing output control challenges. This approach offers flexibility and opens avenues for optimizing the agent's actions within the game environment.

D. Security mechanisms

In order to fortify the reliability and resilience of the agent's environments, it's necessary to implement a set of robust security mechanisms. These measures are designed to validate the integrity of the interaction with the game and ensure the smooth functioning of the Q-learning algorithm. The following security measures have been integrated:

- Process Verification: Regular checks are conducted to verify the presence and execution of essential processes, including the Geometry Dash game. This precautionary measure prevents the agent from attempting interactions with non-existent or terminated game processes, maintaining the stability of the learning environment.
- Screen Availability Check: Prior to capturing game images, the environment performs checks to confirm the availability of the game screen. This ensures that the agent operates in a state where it can accurately perceive the game state, avoiding errors caused by obscured or minimized game windows.
- Game State Validation: The environment incorporates checks to confirm that the game is not in a paused state or navigating through menus. These validations prevent the agent from making decisions based on incomplete or erroneous information, thereby upholding the reliability of the learning process.

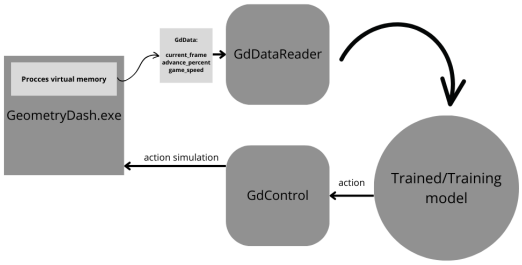

Fig. 1. Agent environment operating diagram

IV. Proposed model

A. Deep Q-Network (DQN)

In this study, we implemented the Deep Q-Network (DQN), a variant of Q-learning tailored for scenarios with a significant number of states. The need for this adaptation arises when managing an extensive state space, rendering the conventional Q-table impractical.

The fundamental concept of DQN involves employing a neural network to approximate the Q-values for each action, offering a pragmatic solution for handling complex problems

with numerous states. This neural network, often structured as a deep convolutional neural network (CNN), acts as a function approximator for the Q-function in Reinforcement Learning. By utilizing deep learning techniques, DQN extends the applicability of Q-learning to high-dimensional input spaces..

The neural network is trained to predict the Q-values for different actions based on the observed states, allowing the agent to make informed decisions in environments with a vast and continuous state space. The training process involves minimizing the discrepancy between the predicted Q-values and the actual rewards obtained during interaction with the environment. This adaptive learning approach enables DQN to generalize across similar states and actions.

1) Neural Network Architecture in DQN: In our specific case, the states are represented as images, prompting the utilization of a convolutional neural network (CNN) to process these visual inputs and derive the associated Q-values.

The original architecture comprises three convolutional layers designed to extract spatial relationships within the images. Initially, our intention was to employ a conventional neural network with three convolutional layers followed by an output layer. However, an alternative structure incorporating an adversarial neural network caught our attention.

The Dueling Network Architecture consists of convolutional networks, but unlike the traditional sequence of fully connected layers, it has two separate streams.

This novel architecture introduces a distinctive feature, separating the network into two pathways: one focusing on estimating the value function, representing the state's inherent value, and the other on assessing the advantage function, quantifying the relative importance of each action. By decoupling these two aspects of the Q-value calculation, the network gains enhanced flexibility and efficiency in learning.

This neural network follows the next expression:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

Breaking down the terms:

- $Q(s,a;\theta,\alpha,\beta)$: This is the action-value function, representing the estimated expected return when taking action a in state s, parameterized by the sets of parameters θ,α,β.
- $V(s;\theta,\beta)$: This is the state-value function, representing the estimated expected return when in state s, parameterized by θ,β.
- $A(s,a;\theta,\alpha)$: This is the advantage function, representing the advantage of taking action a in state s. It is the difference between the action-value and the state-value, parameterized by θ,α.

In the context of GANs, this equation can describe the relationship between the generator (which could be the value $V(s;\theta,\beta)$) and the discriminator (which could be the advantage $A(s,a;\theta,\alpha)$). In GANs, the generator aims to maximize this function, while the discriminator aims to minimize it.

The primary challenges encountered in DQN implementation include:

- Forgetting of Previous Experiences: Variability in weights and intricate correlations may lead to misalignment during agent training.
- Limited Memory: DQN tends to exhibit short-term memory characteristics. To address this, various strategies involve complementing convolutional networks with Long Short-Term Memory (LSTM) layers. However, this introduces considerable complexity. An alternative approach is the utilization of stacked frames to imbue the network with temporal memory.

DQN provides an effective means of handling extensive state spaces through the utilization of neural networks, particularly beneficial in scenarios where the traditional Q-table approach becomes impractical.

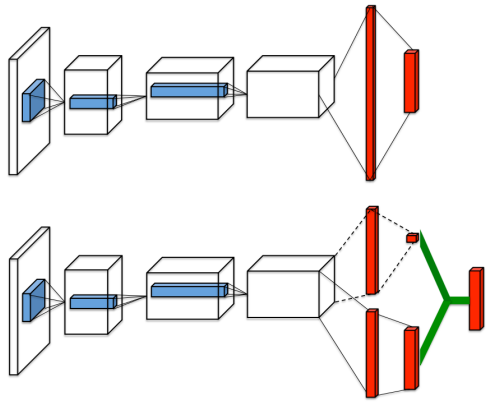

Fig. 2. A popular single stream Q-network(**top**) and the dueling Q-network (**bottom**)

B. Q-Learning

As mentioned before, the Q-Learning algorithm uses a system of rewards and penalties attached to each action and state. The resulting matrix, referred as Q-table, contains the reward given to each possible action in each state.

This matrix is generated during the training, in which the agent chooses an action, which may be random, or the most rewarded one. This is called exploration policy and exploitation policy. Finding a proper balance between them is crucial for the training.

This algorithm allows one to train an agent to do tasks without using examples of how it is done, which makes the data collection a lot harder. This method only needs the rewards to be as well defined as possible and the policy to be balanced to give the desired results.

The main elements of the Q-Learning Algorithm are the following:

1) Agent: The agent is the model to be trained in order to be able to complete some tasks. In this scenario, the agent would be responsible for taking the action of jump or not to avoid the obstacles.

2) Action: The actions are the decisions taken by the agent. Depending on the action taken, the next state will be different. In this case, the only input is jump, but, because the game is always moving forward, two action can be taken, to jump, or not to.

3) Environment: The environment is the set of rules and the possible actions to take, the agent will operate in the environment. In this case, the game itself, since it is what defines the rules and the actions to take.

4) State: The state is the set of circumstances in which the agent is at a given time. In each state, the agent will be able to take certain actions, depending on which, will gain more reward or less. In the case of this project, because the environment is the game itself, and the states are acquired from the game screen, we need to process the image with a neural net to get the states, due to the image being too complex for it being the raw state.

5) Rewards: The rewards are the main element used in the training. It defines whether an action is positive or not by assigning more reward or less to said action. For the sake of the training, it is very important to define as well as possible the rewards or penalties assigned to the actions to get the desired behavior. If the rewards are not well defined, the agent could maximize it without completing the task.

There are many ways to apply the rewards.

As for the rewards, the objective is the agent to finish each level so, of course, the reward will be how long the agent lasts, using the completion percentage of the level, hence the further the agent makes it, the greater the reward.

On the other hand, the penalties are not compulsory. In one attempt, we tried penalizing jumping, making the agent lose a bit of the reward for each jump. The objective was the agent to jump only when it is necessary, however, the result was not the expected. A different approach could be not penalizing any action, since just by avoiding obstacles, the agent will be able to finish the level.

The increase of the rewards in the Q-table is determined by the variable τ which updates the values with a soft update. [6]

V. Training

A. Hyper parameters

The following parameters were used and correctly adjusted during the phase of training of the agent:

1) Batch size: Sets the number of experiences sampled from the replay buffer during each update of the neural network.
2) γ: The discount factor determines the importance of future rewards. A smaller value prioritizes immediate rewards over future ones.
3) ϵ_{start}: Represents the initial value for the exploration probability (epsilon) in an epsilon-greedy strategy.
4) ϵ_{end}: Defines the minimum exploration probability that epsilon will reach over time.
5) ϵ_{decay}: Controls the rate at which the exploration probability decreases over time. A higher decay value makes the agent explore less quickly.
6) τ: Regulates the interpolation between the current policy and the target policy in the update of the neural network. It is a small value for a slow update, promoting stable learning.
7) Learning rate: Stands for Learning Rate, determining how much the model's parameters are adjusted during training based on the calculated gradient. A smaller learning rate provides more stable learning but may require more training iterations.

The first trial involved a batch size of 128, a γ value of 0.95, an initial ϵ probability of 0.9, and a final ϵ value of 0.05 with a decay of 500. The τ was set to 0.005, and the learning rate was 1e-4.

With this approach, the agent began to learn patterns and achieve good results. However, we noticed that there was a point where the exploration rate decreased significantly. Therefore, we tried increasing the batch size to 256 and the ϵ_{decay} value to 1000. With this configuration, considerably better results were achieved.

During this process, more parameters were modified, appreciating the following behaviors experimentally:

1) Batch size: If the batch size is set to a high number, the computational load will be high, and the neural network may not train in a sufficiently responsive time.
2) ϵ_{end}: If the final value of epsilon is too high, the agent will explore extensively, and it may struggle to converge to a solution. However, it is important to note that if it doesn't explore enough, it might get stuck in a local minimum of the solution.
3) ϵ_{decay}: This value needs to be adjusted based on the training time and complexity so that the agent maintains a proper balance between exploitation and exploration.

B. Policies

The Q-value (or action-value) represents the expected cumulative reward of taking a particular action in a given state and following the optimal policy thereafter. Its updated according to a specific criteria called policy. This learning policy is characterized by the exploration-exploitation trade-off. During the training, the agent needs to explore the environment to discover the optimal policy, but it also needs to exploit its current knowledge to make decisions that are likely to yield higher rewards.

Moreover, the choice of the policy is an extremely difficult decision. In the best-case scenario, the agent should start with a nearly-total exploration policy to, by chance, pass through

the first obstacles, increasing the reward of those actions in that state, which allows the agent, after many episodes, to transition to a more exploitation-based policy, choosing the most rewarded action in each state, hence passing through these same obstacles with ease. The following described learning policies where used in the training stage to compare the evolution of the agent knowledge:

1) Greedy Epsilon Policy: Is a simple strategy that involves selecting the action with the highest estimated Q-value most of the time (exploitation), but occasionally exploring random actions (exploration). It follows the following equation:

$$P(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \text{argmax}_{a'} Q(s, a') \text{ (exploitation)} \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise (exploration)} \end{cases}$$

Here, $P(a|s)$ represent the probability of selection action a in state s. ϵ is the exploration rate, the main item of this policy that needs to be adjust to a correct value. $|\mathcal{A}|$ is the action state (this is, the total number of possible actions). In addition, ϵ updates in each episode as followed:

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) e^{-(step/decay)}$$

Where ϵ_{min} and ϵ_{max} and the minimum and maximal values it can reach, step is the current iteration, and decay is the constant at which, the more iterations, the smaller ϵ will be, until $\epsilon = \epsilon_{min}$.

2) Softmax Policy: Softmax policy assigns probabilities to action based on their Q-values, allowing for a smooth transition between exploration and exploitation. Actions with higher Q-values are more likely to be selected, but all actions have non-zero probabilities. It follows the equation:

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}$$

Where $P(a|s)$ is the probability of selecting action a in state s and τ is the temperature parameter that controls the level of exploration (and also, the main item of this policy).

Softmax policy has a smoother transition of probabilities based on Q-values enables the agent to make continuous adjustments in its action. Also, the temperature parameter allows for dynamic control over the exploration-exploitation balance, which is crucial in navigating the diverse challenges of platformer levels.

C. Training environment

When training, a simplification of the game was initially developed to test that the network trained in an appropriate manner. This level was straightforward, allowing us to observe how it recognized patterns.

Subsequently, the agent was used in the Geometry Dash environment. First, a test level was created to assess whether it could also learn in this much more complex environment with many more geometric shapes.

D. Training results and analysis

The following results (Fig. 3) were obtained training the agent in an environment of length *t*=19s and applying softmax policy:

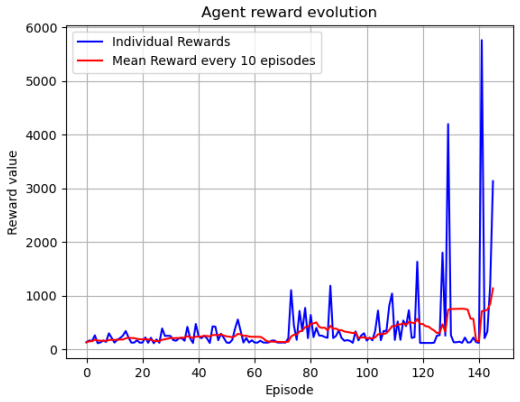

Fig. 3. Obtained reward value by the agent during the training stage

As it can be interpret by the graph, the agent experimented a constants progress in the completeness of the environment (game level) where it was trained. Initially, the agent focused on exploring different alternatives (actions) that would allow it to progress in the level. Starting from episode *E*=70, it can be observed how the agent significantly improves its reward results, with particularly high score peaks at *E*=125 and *E*=140, episode where the agent completes for first time the level. Similarly, a consistent evolution of the average reward value obtained by the agent over time can be observed, taking as the mean value the rewards obtained by the agent every 10 training episodes.

The obtained results suggest the possibility of training agents in environments with short-level platforms, with level times *t* less than 1 minute. Implementing more uniform policies in the exploration-exploitation balance using softmax, can reduce training time and, consequently, enhance model efficiency. However, implementing greedy-epsilon policies for short-level times may yield similar results to softmax policiy in short levels, specially when temperature regulation value approaches to zero.

VI. Conclusion

In conclusion, this paper can be divided into two parts which provide a new and innovative approach to the field of AIs based on Reinforcement Learning:

On the one hand, in contrast to the conventional practice of creating small simulations to imitate certain video games, our approach introduces a fresh perspective. We delve into a game, Geometry Dash, that presents an escalating level of difficulty. Due to this feature, it reaches points that challenge the capabilities of even the most skilled human players, offering a distinctive opportunity to explore the limits of AI capabilities in increasingly complex gaming scenarios.

On the other hand, unlike other similar projects where the states are defined by data extracted from the game, in this project the state is obtained from the game screen. To make that possible, it was needed a neural net to process the image and extract the information to define the states. Moreover, in order to the agent being able to understand the state and choose the action, the frames are needed to be processed as fast as possible. That is one of the reasons why this approach is less commonly used.

After many improvements, we managed to process the image fast enough to get the image at a rate of approximate 20 frames per second, granting the agent enough time to process and choose the action. The result of this is the agent being able to train and play at normal speed.

REFERENCES

[1] Christopher John Cornish Hellaby Watkins, "Learning from Delayed Rewards", King's College, May 1989.

[2] Martin L. Puterman, "Markov Decision Processes: Discrete Stochastic Dynamic Programming," John Wiley & Sons, Inc, April 1994

[3] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. arXiv preprint arXiv:1511.06581, 2016.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. arXiv preprint arXiv:2209.14940, 2013.

[5] Iustina Ivanova. How to play Google Chrome Dino game using reinforcement learning. Medium, 2021.

[6] Fadi AlMahamid. Playing Atari with Deep Reinforcement Learning. arXiv preprint arXiv:312.5602, 2022.

[7] LILLICRAP, Timothy P., et al. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[8] Vignesh Gopalakrishnan. Deep Q Learning (DQN) using Pytorch. Medium, 2023.

[9] Jonathan Hui. RL — DQN Deep Q-network. Medium, 2018.