Training a Customized Policy to Play the Atari Breakout Game

Arthur Costa Stevenson Mota[1] and Vinícius Freitas de Almeida[2]

***Abstract*— This article firstly presents a very brief introduction to Artificial Neural Networks and Reinforcement Learning concepts, and one class of methods based on them, that being Deep Q-Networks (DQN). Then it proposes a solution to train a DQN to play the Atari 2600 game Breakout.**

I. INTRODUCTION

Reinforcement Learning (RL) is a technique that has recently seen great success in performing various tasks, such as humanoid robot motions, self driving cars, and videogames. The latter has received great attention from the scientific community, and serves as a common entry-point for the field, with well documented libraries that provide implementations during several steps in the process of training an agent to perform a certain task. Based on this, this paper uses the Farama Foundation's Gymnasium library, formerly known as OpenAI Gym [1], as well as Stable-Baselines3 [2] as starting points to train a policy capable of playing the Atari 2600 game Breakout, by using the Deep Q-Learning Technique.

II. THEORY

A. Neural Networks

Firstly, a Neuron, presented in Figure 1, is, essentially a mathematical function, with several inputs (x_i), which each have their own weights (w_i) and a bias (b). These inputs, after being added up, are used as the input of an Activation Function, which provides the output of the Neuron. Since a single neuron is incapable of representing complex mathematical functions, it is natural to try and group them. Based on this, Neural Networks (NN), and Deep Neural Networks (DNN), presented in Figure 2, are groupings of interconnected neurons, divided into layers, which are themselves connected in some way. The first category refers to those that have a small number of hidden layers, which are the layers that do not correspond to the input or output of the NN, and, although better than a single Neuron, still have a limited representation capability.

The process of fitting these estimators to a certain mathematical function is called training, and is tipically done by minimizing a Loss function, which can have different forms.

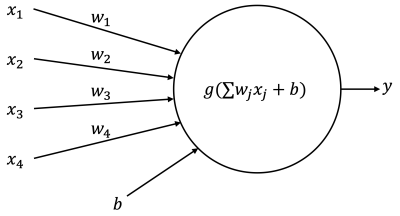

Fig. 1. A single Neuron with inputs x_i, weights w_i, bias b, and output y.

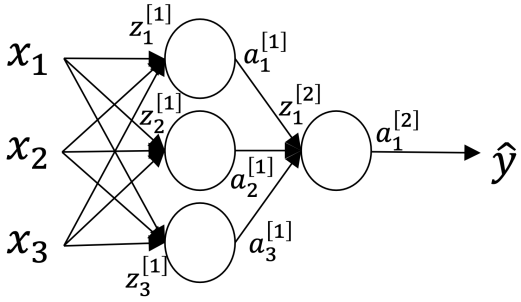

Fig. 2. Representation of a Neural Network with a single hidden layer.

B. Markov Processes

A Finite Markov Decision Process (MDP) is a classical formalization of sequential decision making, in which actions influence immediate and posterior situations. As seen in Figure 3, a learner and decision maker, called Agent, interacts with an Environment through Actions (A_t), and the latter responds to those presenting a new State (S_{t+1}) and a numerical value, called a Reward (R_{t+1}) [3].

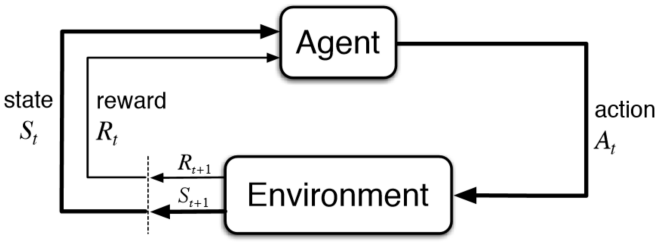

Fig. 3. Agent-environment interaction in a Markov Decision Process.

Some additional concepts related to Finite MDPs are the Value Function, presented in Equation (1), which corresponds to the expected return when starting from a state s and following a policy π thereafter, and the Action-Value Function, presented in Equation (2), corresponding to the expected return when starting from state s, taking action a, and thereafter following a policy π.

*This work was not supported by any organization

[1]Arthur Stevenson is a student in the Computer Engineering course at the Aeronautics Institute of Technology, São Paulo, Brasil. arthur.mota@ga.ita.br

[2]Vinícius Freitas de Almeida is a student in the Electronic Engineering course at the Aeronautics Institute of Technology, São Paulo, Brasil. viniciusvfa@ita.br

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{for all } s \in \mathcal{S}
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]
\end{aligned} \tag{2}
$$

From these definitions, derives an important equation, known as the Bellman Equation, which expresses the relationship between the value of a state and its successors. It is presented in Equation (3), and is widely used for training agents.

$$
v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma \cdot v_\pi(s')], \forall s' \in \mathcal{S} \tag{3}
$$

C. Reinforcement Learning

Reinforcement Learning is a popular method for training different types of artificial intelligences. It tries to solve MDPs, by training a policy (π), which aims to always perform the optimal actions, maximizing the reward received at the end. The RL agents have an Observation Space, which defines what they can perceive, and an Action Space, which determines what they can do.

A possible division of these methods is on-policy and off-policy. The former implies that the same policy μ is used when determining which action should be taken will eventually be optimized to be the optimal policy, and the latter, that the policy μ to determine the actions it takes, but another policy π, which is optimal, will be learned. The second class of methods is very useful, since it allows a learning agent to benefit from other's experiences.

Finally, these methods also have what are called *hyper-parameters*, which are variables that determine part of the behavior of the learning algorithm, such as how much one step of the policy optimization will differ from the other

D. ε-greedy policies

An ε-greedy policy is a common exploration strategy used in reinforcement learning. It balances the trade-off between exploration (trying new actions to discover better policies) and exploitation (choosing the currently known best action). The ε-greedy policy selects the best action with probability $1-\varepsilon$ (exploitation) and selects a random action uniformly at random from the action set with probability ε (exploration).

Mathematically, the ε-greedy policy for selecting action A_t at time step t in state S_t is defined as follows:

$$
A_t = \begin{cases} \underset{a \in A}{\operatorname{argmax}}\, Q(S_t, a) & \text{with probability } 1-\varepsilon \\ \text{random action from } A & \text{with probability } \varepsilon \end{cases} \tag{4}
$$

Here, $Q(S_t, a)$ represents the estimated value (Q-value) of taking action a in state S_t. The value of ε determines the exploration-exploitation trade-off. When ε is set to a high value, the agent is more exploratory, while setting it to a low value makes the agent more exploitative.

E. Q-Learning

Q-Learning is an off-policy RL method, which uses a behavioral policy μ to determine its actions, while it learns the optimal policy π. It works by updating an estimator of the Action-Value function according to Equation (5), which is derived from the Bellman Equation (3). The estimator employed is simply a cache for all the action-value pairs, which is referenced when determining when deciding on an action for a certain state.

$$
\begin{aligned}
Q_{k+1}(S_t, A_t) = &[Q_k(S_t, A_t)] \cdot (1-\alpha) \\
&+ \alpha \cdot (R_{t+1} + \gamma \max_{a' \in A} Q(S_{t+1}, a'))
\end{aligned} \tag{5}
$$

By using an ε-greedy policy during Q-learning, the agent can learn the optimal policy while still exploring different actions to improve its Q-value estimates for different state-action pairs. As the training progresses, it's common to reduce the value of ε over time to focus more on exploitation as the agent becomes more confident in its Q-value estimates.

F. Deep Q-Networks

Deep Q-Learning [4] is an extension of the Q-Learning method, which uses a DNN, represented as $Q(s, a; \theta)$, in place of the cache as the estimator of the Action-Value Function, thus allowing it to be applied in problems that have continuous Observation Spaces without the use of discretization, as well as problems with large Observation-Spaces, which would not fit into memory. The new structure is, therefore, called a Deep Q-Network (DQN), and the Loss function used to train this DNN is the one from Equation (6), and it changes every iteration.

$$
\begin{cases} \mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s, a; \theta_i))^2] \\ y_i = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\right] \end{cases} \tag{6}
$$

Typically, the learning process for these policies is done through stochastic gradient descent, which essentially calculates an estimate of the gradient of the loss function through experience, that is, by collecting the rewards for a number of episodes, and using them to create an estimate of $\nabla_{\theta_i}\mathcal{L}_i$.

III. METHODOLOGY

A. Gymnasium and Breakout

The game Breakout, released in 1972 for the Atari 2600, is a classic arcade game, in which the player must control a paddle, and use it to bounce a ball towards several bricks, which are destroyed upon contact with it. Thus, the objective of the game is to use the ball to eliminate as many bricks as possible, without dropping the ball.

This game is included in the Atari set of environments, provided in Farama Foundation's Gymnasium [1], and has several variations, mainly with regards to the observation space available for the agent. For this work, the variation `Breakout-ram-v5` was chosen, since it is the most up-to-date, and has the smallest observation space among the ones available, that being the 128 bytes of RAM on the virtual console, which is expected to lead to faster training times, due to a reduced amount of weights in the NN.

Therefore, the RL problem at hand has an Observation Space defined as 128 integer entries, varying from 0 to 255, due to the Atari 2600 having a 8-bit architecture, and an Action Space that has 4 different actions, which are **do nothing**, move the paddle to the **left**, to the **right**, and **launch the ball**.

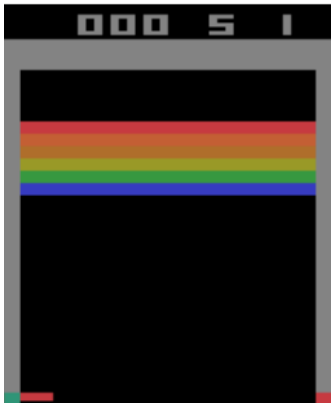

Fig. 4. Graphical interface of the Breakout game.

B. Stable-Baselines3

Stable-Baselines3 [2] presents itself as "a set of improved implementations of reinforcement learning algorithms in PyTorch". Therefore, it has a well-mantained implementation for most of the currently popular RL methods, which includes Deep Q-Networks, and is a suitable choice for a code implementation of this method. Naturally, the library allows the user to choose the hyperparameters for training the method, by setting them when the algorithm is instantiated, as well as to define the architecture of the DNN that will be trained, by using the parameter `policy_kwargs` that is passed to it.

The library also presents a wrapper class that can be used on Gymnasium environments, that turns them into parallel environments, allowing a configurable amount of them to run simultaneously, which speeds-up data collection for the learning process. Finally, it also provides a customizable logger, in this case, used to store the mean reward values for each batch that runs.

C. Network Architecture and Hyperparameters

Mostly based on experimentation and trial-and-error, the hyperparameters used are presented in Table I, where Train Frequency determines how many episodes must pass before updating the model, and Exploration Fraction represents the fraction of the entire training period during which the ε-greedy probability will be reduced from $\varepsilon_{\text{Start}}$ to ε_{End}. They are passed as arguments when initializing the DQN object with the implementation provided by Stable-Baselines3. Since the environment used provides the bits in memory as the agent's Observation Space, as opposed to pixels on a screen, it was considered that the layers of the DNN used by the DQN did not need to be Convolutional Layers, and thus, the structure presented in Table II was used.

Hyperparemeter	Value
Amount of Training Timesteps	10.000.000
Learning Rate	0.0005
Batch Size	16
Buffer Size	1.000.000 steps
γ	0.95
Exploration Fraction	0.1
$\varepsilon_{\text{Start}}$	0.1
ε_{End}	0.05
Train Frequency	1

TABLE I

HYPERPARAMETERS USED FOR TRAINING THE POLICY.

Type of Layer	Activation Function	Number of Neurons
Input Layer	Linear	128 (Observation Space)
Hidden Layer	ReLU	32
Hidden Layer	ReLU	64
Hidden Layer	ReLU	128
Hidden Layer	ReLU	32
Output Layer	Linear	4 (Action Space)

TABLE II

ARTIFICIAL NEURAL NETWORK ARCHITECTURE.

IV. RESULTS

Although the trained policy did improve when compared to the initial policy, as can be seen in the increase of the mean reward values for each episode, presented in Figure 5, which is also reflected in a longer episode time, as seen in Figure 6, the policy learned by the DQN Agent is not a very useful one. As presented in Figure 7, most of what it does is move the paddle to the corner of the screen, and keep it there. It does so due to the fact that the ball, when launched, moves towards one of the corners, and so, by placing the paddle there, the ball has a good chance of hitting the it, and bouncing towards the bricks, making the episode longer. This strategy, however, is limited only to the first bounce, since, when coming back from the bricks, the ball will certainly arrive at a different position from before, which leads to the agent losing a life. This cycle then repeats for every life available to the agent, at which point it loses, and the episode resets.

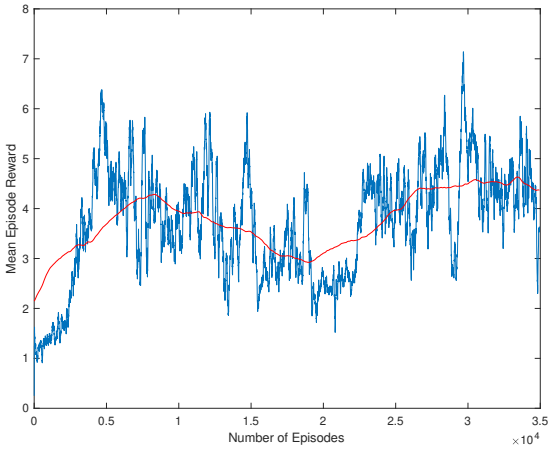

Fig. 5. Episode reward versus number of learning episodes, and moving average of the reward values, considering the last 2000 values.

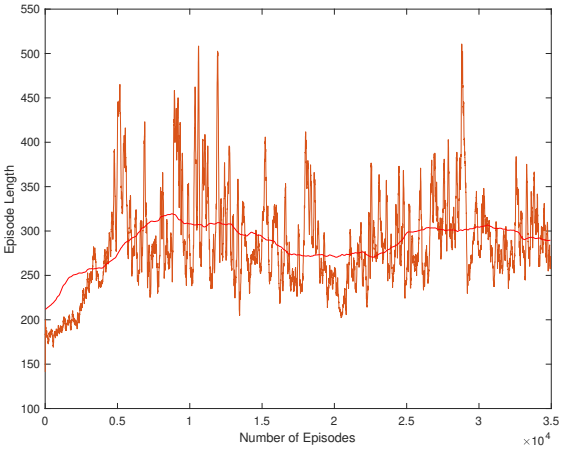

Fig. 6. Episode length versus number of learning episodes, and moving average of the length values, considering the last 2000 values.

V. CONCLUSION

In this work, we explored concepts and methods of Artificial Neural Networks and Reinforcement Learning. Our proposed solution involved training a custom policy to play the Breakout game using Stable-Baselines3, a library different from the implementations used in the course. Additionally, we utilized the Atari 2600 environment provided by Gymnasium, which differed from the environments typically used in our coursework. This combination of different libraries and implementations allowed us to address the problem effectively and achieve promising results in training the policy for game playing. As stated previously, although the policy did show signs of improvement,

REFERENCES

[1] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[2] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," 2019.

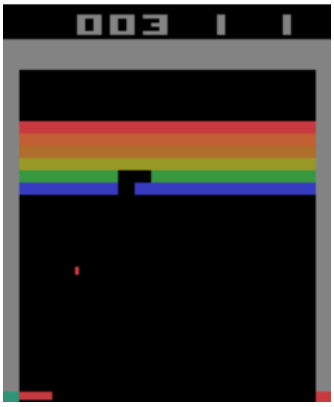

Fig. 7. Behavior of the policy during most of the episodes.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.