Playing Breakout with Deep Reinforcement Learning

E. S. Ramos ; C. C. Aquino ; R. P. Leite ; L. S. Lima ; J. F. F. Neto

Department of Computing/Laboratory D02, Federal University of Sergipe, 49100-000, São Cristóvão-Sergipe, Brazil

{ edcarlosufs , riquelmewin , cainacastro29 , falcao ,waynewyn } @ academico.ufs.br

The development of intelligent agents capable of playing video games has been a widely researched topic in the field of Artificial Intelligence, driven by techniques such as Deep Learning and Reinforcement Learning. This study proposes the implementation of an intelligent agent to play Breakout using the Deep Q-Network (DQN) architecture. DQN employs deep neural networks to estimate action values and optimize the decision-making process based on visual representations of the environment.

Breakout, one of the most well-known benchmarks in Reinforcement Learning, presents challenges such as sequential action control, visual state interpretation, and the balance between exploration and exploitation. Our approach utilizes the OpenAI Gym environment for training and evaluating the agent. The expected results aim to demonstrate the efficiency of DQN in learning optimized strategies, contributing to advancements in the development of autonomous agents in gaming environments.

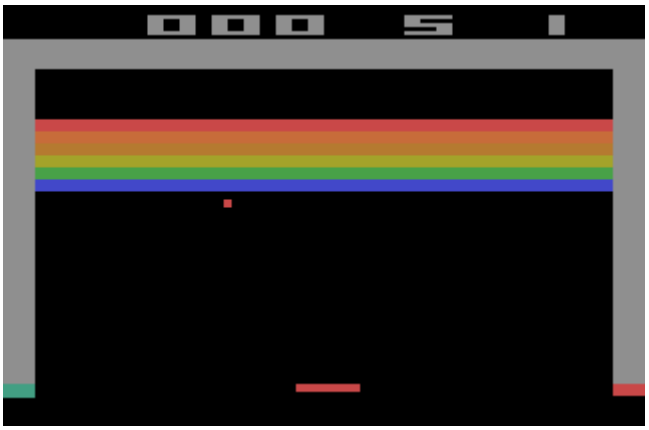
1. Introduction

The development of intelligent agents capable of playing video games has been a widely explored research field in Artificial Intelligence (AI), especially following advancements in Deep Learning and Reinforcement Learning techniques. Classic games like Breakout provide an ideal environment for testing and improving these techniques, as they present challenges that require planning, adaptation, and the learning of optimized strategies over time.

The use of Deep Neural Networks (DNNs) in combination with Reinforcement Learning led to the development of Deep Q-Network (DQN), a technique introduced by Mnih et al. (2015), which enabled trained agents to surpass human performance in several Atari 2600 games. DQN employs a neural network to estimate the action-value function (Q-learning), allowing the agent to learn effective decision-making strategies directly from visual representations of the environment. Since then, several improvements have been proposed, such as Double DQN (van Hasselt et al., 2016), Prioritized Experience Replay (Schaul et al., 2016), and Dueling DQN (Wang et al., 2016), all aiming to enhance the stability and efficiency of the learning process.

Deep Q-Network (DQN) is a deep reinforcement learning algorithm that combines Q-Learning with deep neural networks to learn optimal strategies in complex environments. DQN revolutionized the AI field by enabling agents to outperform human performance in various Atari games. Reinforcement learning consists of an agent that interacts with an environment, receives rewards, and adjusts its actions to maximize future gains. DQN addresses challenges such as the high dimensionality of states and actions using Replay Buffer (storage and reuse of experiences) and Target Network (an auxiliary network for stability). Additionally, extensions such as Double Q-Learning, Prioritized Experience Replay, and Dueling Network Architecture improve learning efficiency and stability.

The game Breakout, available in OpenAI's Atari Gym environment set, is one of the most popular benchmarks for reinforcement learning experiments. Originally released by Atari in 1976, the game challenges the player to control a paddle at the bottom of the screen to bounce a ball and break bricks at the top. The objective is to break all the bricks without letting the ball fall.



In OpenAI Gym, the *Breakout-v0* environment (and its variants, such as *BreakoutNoFrameskip-v4*) provides a standardized interface for RL experiments. The agent receives screen images (game frames) as input and can perform four discrete actions: stay still, move left, move right, and launch the ball. The game returns a positive reward when a brick is destroyed and ends when all lives are lost.

The main challenges of *Breakout* for RL agents include:

- **Sequential action control**: The agent needs to plan its movements in advance to correctly bounce the ball.
- **Pixel-based observation**: Since the input is the game image, the model must learn to interpret visual states and make decisions based on them.
- **Exploration and exploitation**: The agent must find effective strategies (such as creating tunnels on the sides) while exploring new approaches.

In this work, we propose the development of an intelligent agent that plays *Breakout* using the DQN architecture. Our approach consists of training a deep neural network model to learn effective strategies through interaction with the environment.

2. METHODOLOGY

2.1 Environment Details

The agent's training was conducted in the Breakout-v4 environment, provided by the Arcade Learning Environment (ALE). ALE offers a standardized platform for evaluating reinforcement learning agents in Atari 2600 games.

2.1.1 Environment Parameters

- **Environment version**: Breakout-v4
- **Frame Skip**: 3

The frame skip parameter determines how frequently the agent executes actions in the environment. Instead of processing each individual frame, the agent selects an action and maintains it for three consecutive frames. This technique reduces computational complexity and improves learning stability by preventing the agent from making decisions at every frame, which could lead to unnecessary variations and noisy behaviors.

2.1.2 State Format

The environment state is represented by a game-rendered image at each frame. Each state is a three-dimensional matrix (H, W, C), where:

- **H** represents the image height (originally 210 pixels).
- **W** represents the image width (originally 160 pixels).
- **C** corresponds to the number of image channels (originally 3 RGB channels).

2.2 Image Preprocessing

To ensure that the environment images are suitable as input for the neural network, a preprocessing step was performed, consisting of the following stages:

- **Grayscale Conversion**: The original image, captured in RGB format, was converted to grayscale. This conversion reduces input dimensionality by removing unnecessary chromatic information, improving computational efficiency without compromising the agent's ability to interpret the environment dynamics.

- **Resizing**: The images were resized to a fixed format of 84 × 84 pixels. This standardization allows the convolutional network to receive inputs of consistent size, facilitating the extraction of relevant visual features.

- **Tensor Conversion and Normalization**: After resizing, the images were converted into a PyTorch tensor and normalized. Normalization was applied to stabilize the training process, ensuring that pixel values remain within a suitable range for gradient propagation in the neural network.

2.3 Neural Network Architecture

The deep convolutional network used consists of three convolutional layers followed by two fully connected (FC) layers.

Convolutional Layers:

- **Layer 1**: 32 filters of size 8×8 with a stride of 4.
- **Layer 2**: 64 filters of size 4×4 with a stride of 2.
- **Layer 3**: 64 filters of size 3×3 with a stride of 1.
- The output of the convolutional layers has dimensions **64 × 7 × 7**.

Fully Connected Layers:

- **Flatten**: Transforms the output of the last convolutional layer (64 × 7 × 7) into a one-dimensional vector of **3,136 neurons** (64 × 7 × 7 = 3,136), preparing the data for the fully connected layers.
- **FC1**:
    - Input: **3,136 neurons**
    - Output: **512 neurons**
    - Activation function: **ReLU**
- **FC2**:
    - Input: **512 neurons**
    - Output: **4 neurons**, where each neuron represents the estimated Q-value function **Q(s,a)** for one of the four possible actions of the agent.

This architecture enables the model to learn an efficient representation of the environment and make action decisions based on the Q-function estimation.

CNN pseudocode:

```
CNN Class:
    Input: action_size (Number of possible actions in the environment)

    Define Layers:
        - Convolution 1: (input_channel1, output_channel1, conv1 kernel size, conv1 stride value)
        - Convolution 2: (output_channel1, output_channel2, conv2 kernel size, conv2 stride value)
        - Convolution 3: (output_channel2, output_channel3, conv3 kernel size, conv3 stride value)
        - Fully Connected Layer 1: (FC1 input unit, FC1 output unit)
        - Fully Connected Layer 2: (FC1 output unit, ACTION_SIZE)

    forward(x) method:
        Apply ReLU activation function to the output of Convolution 1
        Apply ReLU activation function to the output of Convolution 2
        Apply ReLU activation function to the output of Convolution 3

        Flatten output to a vector (Flatten)

        Apply ReLU activation function to the output of Fully Connected Layer 1

        Return output of Fully Connected Layer 2 (Q-values of actions)
```

2.4 Hardware Environment Specifications

For the experiments, the Google Colab Pro subscription was used, ensuring access to enhanced computational resources. The execution environment was configured to utilize an NVIDIA T4 GPU with 15 GB of VRAM, enabling more efficient training of the deep neural network. Additionally, the system had 51 GB of RAM, allowing better handling of large datasets and experience buffers. The available storage in the environment was 235.7 GB of disk space, with 15 GB dedicated to temporary execution storage.

2.5 Reward System:

2.5.1 Base Environment Reward

The agent receives **positive rewards** for achieving good results in the game environment. The original reward is **multiplied by a constant**. After a fixed number of steps, the reward is multiplied again, encouraging **longer survival** in the game.

2.5.2 Life Loss Penalty

The code checks whether the number of lives has changed. If the agent **loses a life**, it **receives a penalty**. This encourages the model to avoid actions that lead to life loss.

2.6 Hyperparameters

Table 1:

	Training Parameters
Batch_Size	16
Gamma	0.99
Epsilon	1.0
Epsilon_min	0.05
Epsilon_decay	0.001
Learning_rate	0.00025
Buffer_size	80000
Target_update	2000

Table 2:

Reward	Reward Parameters
	Gross reward from the environment
Penalty for losing life	-1
Reward multiplier	2
Steps to double reward	500

AgentDQN pseudocode:

```
Class trainingAgentDQN:
    For each episode in NUM_EPISODES:
        Initialize frame_stack (buffer to store the last 4 frames)
        Restart the environment and process the first frame
        Get the initial number of player lives
        Fill the frame_stack with the first frame 4 times
        Create the state by stacking the 4 frames
        Initialize total_reward, action_noop_count and score

        For each step in the episode (STEPS_PER_EPISODE):
            Select an action based on the current policy and stacked state
            Execute the action in the environment and get the next state, reward and game status
            Adjust the reward:
                - Double the base reward
                - If more than 400 steps have passed, double the reward again
                - If the reward is positive, increase the score
                - If the action is 'no-op' (action 0), increase action_noop_count
                - If the number of lives decreases, apply a penalty for loss of life

            Update the stacked state with the next processed frame
            Store the transition in the experience buffer
            Update the current state
            Training the DQN:
                If the replay buffer has enough samples:
                - Sample a mini-batch of transitions
                - Calculate the Q-values of the current policy for the chosen actions
                - Calculate the expected Q-values using the target network
                - Calculate the loss (SmoothL1Loss)
                - Update the neural network weights
            Update the target network periodically
            If the game is over, display the number of steps and exit the loop
        Reduce the epsilon value to control the exploration
```

3. EXPERIMENTS

To evaluate the performance of the agent trained using the Deep Q-Network (DQN) approach, we conducted experiments in the Breakout-v4 environment from Gymnasium. The agent was trained over 5,000 episodes, using the previously defined hyperparameters. During training, the exploration-exploitation policy was adjusted through an **ε**-greedy decay, allowing the agent to balance the exploration of new actions with the exploitation of the best-learned strategies.

The loss function was minimized using the Adam optimizer, with a learning rate adjusted to favor training stability. Additionally, we implemented a Replay Buffer to store and sample past experiences, reducing the correlation between consecutive updates and improving model

convergence. To mitigate instability in the value function estimation, we used a target network, which was updated periodically.

After training, we conducted tests with the trained agent over 112 episodes to evaluate its performance in an execution scenario without additional exploration. The graph presented in Figure 1 illustrates the evolution of the agent's score over these episodes. We observed an upward trend in average score, indicating learning progress. However, the score still exhibits significant variability, suggesting possible instabilities in the training process, which may be related to factors such as hyperparameter selection or the need for a longer training period.

For a more detailed quantitative analysis, Table 3 presents the highest score recorded during the experiments, representing the maximum number of blocks destroyed in a single game.

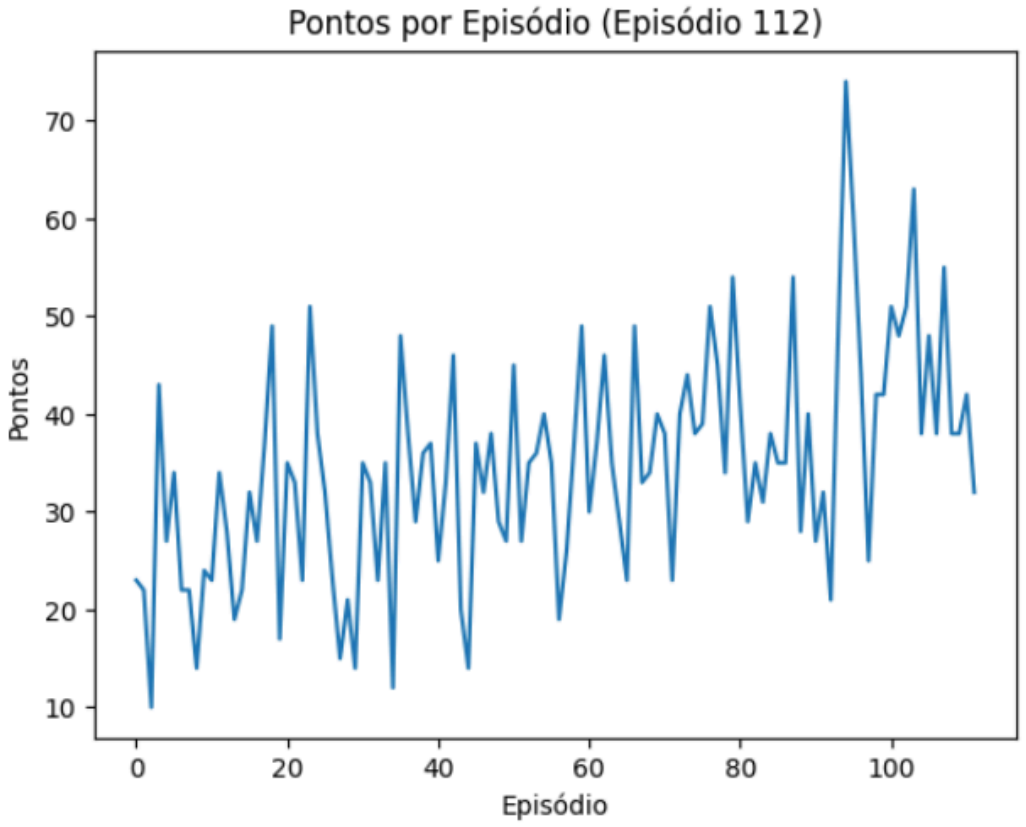


Table 3

	Types of Maximum Scores
Record	**Breakout-v4**
Destroyed blocks	41
By points (from the game)	132

4. Results and discussion

The agent trained with DQN achieved a maximum of 41 blocks destroyed, a reasonable performance but still below the optimal level expected for this environment. This limitation may

be related to the agent's difficulty in learning advanced strategies, such as refined ball control to optimize the destruction of upper blocks.

The results indicate that, while DQN was able to learn basic gameplay patterns, it has limitations in acquiring more sophisticated strategies. This may be attributed to several factors:

- **Limited Exploration**: The DQN algorithm relies on an ε-greedy strategy, which may not be sufficient to ensure efficient action-space exploration. More advanced methods, such as Prioritized Experience Replay (PER), could help mitigate this issue.
- **Learning Instabilities**: The variation in score suggests that the agent may be experiencing instabilities in the value function, which could be minimized using approaches like Double DQN or Dueling DQN.
- **Reward Function Efficiency**: The agent may be optimizing its reward without developing long-term strategies, such as keeping the ball trapped in the upper part of the screen to destroy multiple blocks without interference. Reformulating the reward function, including incentives for reaching upper regions, could enhance this behavior.

5. CONCLUSION

During the testing phase, an instability in the agent's performance was observed, associated with the emergence of undesirable addictive behaviors in the trained model. Such behaviors can compromise the agent's generalization ability and efficiency in different situations within the environment.

To mitigate these issues, the following improvements are proposed:

- Application of regularization techniques in the neural network: Introducing methods such as dropout, weight decay, or batch normalization can help reduce overfitting and improve learning stability.
- Increasing the number of training episodes: Training the agent for more episodes will allow better convergence of network parameters, reducing fluctuations and ensuring more robust behavior.
- Modifying the exploration policy: Adjusting the exploration strategy can prevent the agent from getting stuck in addictive behaviors that limit its performance.

6. BIBLIOGRAPHIC REFERENCES

1. BELLEMARE, M. G.; NADDAF, Y.; VENESS, J.; BOWLING, M. **The Arcade Learning Environment: An Evaluation Platform for General Agents.** *Journal of Artificial Intelligence Research*, v. 47, p. 253-279, June 2013.

2. MNIH, Volodymyr. **Playing Atari with Deep Reinforcement Learning.** *arXiv preprint*, arXiv:1312.5602, 2013. Available at: https://tinyurl.com/atari-rl. Accessed on: Feb. 14, 2025.

3. MNIH, Volodymyr; et al. **Human-Level Control Through Deep Reinforcement Learning.** *Nature*, v. 518, n. 7540, p. 529-533, 2015. Available at: https://tinyurl.com/human-level-nature. Accessed on: Feb. 14, 2025.

4. SILVER, David; et al. **Mastering the Game of Go Without Human Knowledge.** *Nature*, v. 550, n. 7676, p. 354-359, 2017. Available at: https://tinyurl.com/mastering-go. Accessed on: Feb. 14, 2025.

5. GYMNASIUM. **Basic Usage.** Available at: https://gymnasium.farama.org/introduction/basic_usage/. Accessed on: Feb. 14, 2025.

6. GOOGLE. **Google Colaboratory.** Available at: https://colab.google/. Accessed on: Feb. 20, 2025.