Training of Reinforcement Learning Agents on Pac-Man with DQN and MLP

Treinamento de Agentes de Aprendizado por Reforço no Pac-Man com DQN e MLP

D. A. Sales; F. R. Santana; H. S. Siqueira; L. D. M. Cavalcanti; N. V. dos Santos

Departamento de computação, UFS, 49100-000, São Cristovão-SE, Brasil

This study investigates the use of a deep Q-network (DQN) approach combined with a multilayer perceptron (MLP) to train a reinforcement learning (RL) agent to play the Pac-Man game. Advanced techniques such as convolutional neural networks for feature extraction and training stabilization strategies including experience replay and target networks are applied. The experiments show that despite fluctuations in rewards, the agent continues to improve throughout the training process. Comparative analysis of 200, 500, and 1000 episodes shows that the agent's learning becomes more consistent, but hyperparameters still need to be adjusted to maintain stability. Monte Carlo Tree Search (MCTS) was initially considered but abandoned due to high computational cost.
Keywords: reinforcement learning, deep Q-network, Pac-Man.

Este estudo investiga o uso de uma abordagem de rede Q profunda (DQN) combinada com um perceptron multicamadas (MLP) para treinar um agente de aprendizagem por reforço (RL) para jogar o jogo Pac-Man. São aplicadas técnicas avançadas, como redes neurais convolucionais para extração de características e estratégias de estabilização de treinamento, incluindo repetição de experiência e redes alvo. As experiências mostram que, apesar das flutuações nas recompensas, o agente continua a melhorar ao longo do processo de treinamento. A análise comparativa de 200, 500 e 1000 episódios mostra que o aprendizado do agente se torna mais consistente, mas os hiperparâmetros ainda precisam ser ajustados para manter a estabilidade. O Monte Carlo Tree Search (MCTS) foi inicialmente considerado, mas abandonado devido ao alto custo computacional.
Palavras-chave: aprendizagem por reforço, rede Q profunda, Pac-Man.

1. INTRODUCTION

Atari games have been widely used as benchmarks for testing reinforcement learning (RL) algorithms, providing a challenging environment for the development and evaluation of intelligent agents. The complexity of these games combines high-dimensional visual insights with sequential decision-making, making them an ideal research area for advancements in deep learning applied to RL.

One of the pioneering works on using deep learning to play Atari games was presented by Mnih et al. (2013), who proposed the Deep Q Network (DQN), a deep neural network capable of learning policies directly from raw game pixels (Mnih, 2013). This approach demonstrated that agents trained with deep reinforcement learning could achieve performance equal to or better than that of human players in various Atari games (Mnih et al., 2015).

Complementary methods have been explored to improve training efficiency and the agent's generalization capability. For example, Guo et al. (2014) used offline Monte Carlo Tree Search (MCTS) to generate training data, allowing the neural network to learn effective policies without direct interaction with the environment (Guo et al., 2014). Later works, such as that of Silver et al. (2017), showed that advanced deep learning techniques combined with Monte Carlo tree search could achieve superhuman performance levels, as demonstrated in the game of Go (Silver et al., 2017).

In this work, we explore the construction and training of reinforcement learning-based agents to play Pac-Man using a hybrid approach that combines different learning strategies. The implementation leverages the Gym library, which provides the Atari simulation environment, as well as modern frameworks for building and training neural networks, such as PyTorch. The developed agent aims to learn efficient strategies to maximize scores in the game, exploring concepts such as trial-and-error learning, reward maximization, and deep neural network optimization.

The objective of this study is to analyze the performance of the agents, compare different RL methods, and understand the challenges and advancements in the field of reinforcement learning applied to games.

2. METHODOLOGY

The methodology adopted for the development of the Reinforcement Learning (RL) agent in the Pac-Man game was structured into three main stages: environment setup, agent implementation, and model training and evaluation. Each of these stages is described below.

2.1 Environment Setup

For the game simulation, we used OpenAI Gym, a widely used library for RL experiments that provides a standardized interface for Atari environments. The selected environment was MsPacman-v0, which presents strategic challenges such as maze navigation, point collection, and enemy evasion.

Additionally, the following libraries were used to support the agent's training:

- PyTorch: For implementing the agent’s neural network.
- NumPy and Matplotlib: For data manipulation and visualization of training statistics.
- Gym Atari: For loading the game environment in OpenAI Gym.

The simulation was conducted on a machine with an NVIDIA GPU to accelerate neural network training.

2.2 Agent Implementation

The agent was developed using an approach based on Deep Q-Network (DQN) (Mnih et al., 2015), which combines Deep Neural Networks with the Q-Learning algorithm to learn an optimal policy from interactions with the environment. The model was structured as follows:

- Input: Processed game frames converted to grayscale, with dimensionality reduction to speed up learning.
- Neural Network: A Convolutional Neural Network (CNN) with three convolutional layers for visual feature extraction, followed by fully connected layers for action estimation.
- Reward Function: Positive rewards were assigned for collecting points and fruits, while penalties were applied when the agent was captured by ghosts.
- Stabilization Techniques: Use of an experience replay buffer to store and sample past interactions, along with the Target Network technique to smooth weight updates.

2.2.1 Frame Preprocessing

In Atari games, raw screen images need to be processed before being used by the Deep RL agent. In this study, we implemented a frame preprocessing function that converts the original images into a more suitable format for model input. This function follows common approaches in the literature (Mnih et al., 2015), including Conversion to grayscale, Resizing to 84×84 pixels, Normalization to the range [-1,1], Conversion to tensor and Transfer to GPU. The following code snippet presents the implementation of the preprocessing function used in this study:

```
# Outras importações

import os
import gym
import torch
import torch.nn as nn
import cv2

# Códigos de inicialização


def preprocess_frame(frame):
    # Converte de RGB para escala de cinza e redimensiona
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
    frame_resized = cv2.resize(frame_gray, (84, 84))
    frame_normalized = (frame_resized / 255.0 - 0.5) * 2  # Normaliza para [-1,1]



    # Converte para tensor e move para GPU
    frame_tensor = torch.tensor(frame_normalized, dtype=torch.float32, device=device)
    return frame_tensor


# Restante do código
```

2.2.2 Neural Network Architecture

To enable the agent to make decisions in the Atari Pac-Man environment, we implemented a neural network based on Deep Q-Network (DQN) with structural enhancements. The architecture combines convolutional layers for image feature extraction with a deeper Multi-Layer Perceptron (MLP) for modeling the agent's decision-making process.

This approach follows the guidelines proposed by Mnih et al. (2015) ("Human-level control through deep reinforcement learning"), which demonstrated that convolutional networks can efficiently learn visual representations directly from raw game pixels. The following code presents the implementation of DQNWithEnhancedMLP, which incorporates these improvements:

```
//Importações
//Outras partes do código

DQNWithEnhancedMLP(nn.Module):
    def __init__(self, action_size):
        // parte convolucional
        super(DQNWithEnhancedMLP, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv2d(4, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten()
        )

       //utilizando uma MLP
        self.mlp = nn.Sequential(
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, 256),  # Camadas extras
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, action_size)
        )

    def forward(self, x):
        x = self.cnn(x)
        return self.mlp(x)

//Restante do código
```

The convolutional (CNN) part of the model is responsible for processing high-dimensional states (e.g., Atari environment images) and automatically extracting relevant features.

After extracting features with the CNN, the model uses an MLP (fully connected layers) to map these features into Q-values—values that indicate the quality of each action in a given state.

2.2.3 Why We Did Not Implement MCTS

To complement the DQN, and before implementing it with an MLP (Multi-Layer Perceptron), we considered using Monte Carlo Tree Search (MCTS) to select the best exploratory actions through simulations. We developed an initial implementation but quickly realized that this approach would not be viable.

In the context of the Atari Pac-Man game, we faced challenges that made MCTS less effective. This method relies on repeated simulations from the current state to estimate action values. In games like Go, where states are well-defined and simulations are fast, MCTS is highly efficient. However, in Pac-Man, where states are represented by images and the environment has complex dynamics, each simulation required a high computational cost.

This resulted in excessively long simulation times and significant memory usage, making the approach impractical given our computational limitations. Based on these challenges, we opted to proceed directly with the DQN implementation, which proved to be more suitable for handling the problem.

2.3 Training and Evaluation

The agent was trained using the DQN algorithm with Replay Memory over a series of episodes, adjusting hyperparameters such as exploration rate (ε-greedy), learning rate (α), and discount factor (γ). During training, the following metrics were monitored:

- Average score per episode: Measures the agent's efficiency over time.
- Exploration vs. exploitation rate: Evaluates the balance between trying new actions and exploiting known ones.
- Q-function convergence: Checks the stabilization of the learned policy.

At the end of each training session, the model was saved. In the next session, the most recent model was automatically loaded, updating the current weights and continuing training from where the previous session left off.

3. EXPERIMENTS

The training was carried out in multiple phases, adjusting parameters as needed. The key parameters included:

- **Number of episodes:** Defines the total number of episodes the agent will run, where each episode corresponds to a full game (from start to finish).
- **Gamma factor (γ):** The discount factor, determining how future rewards are valued compared to immediate rewards.
- **Learning rate (α):** The step size used by the optimizer to update the network's weights.
- **Epsilon (ε):** The initial value for epsilon in the epsilon-greedy strategy, which controls exploration.
- **Epsilon min:** The minimum epsilon value. As training progresses, epsilon decays (reducing exploration) until it reaches this minimum value.
- **Epsilon decay:** The factor used to gradually reduce epsilon after each episode.
- **Buffer size:** The maximum size of the replay buffer, where the agent stores its experiences (states, actions, rewards, next states, and episode termination flags).
- **Batch size:** Defines how many samples are randomly drawn from the replay buffer for each network update.

3.1 Parameters in Key Phases

Below is a table indicating the parameters used in each key training phase.

Table 1: Parameters by Phase

Parameters	Phase 1	Phase 2	Phase 3
nº of Episodes	200	500	1000
gamma	0.99	0.99	0.99
learning rate	0.0001	0.00005	0.00005
epsilon	1.0	1.0	1.0
epsilon min	0.1	0.2	0.2
epsilon decay	0.995	0.98	0.98
buffer size	100000	100000	100000
batch size	32	64	256

During training, the agent stored experiences in a buffer and performed batch learning using the MSELoss() function.

Although each phase in the table has a fixed number of episodes, training sessions with a lower (or higher) number of episodes within the ranges of each phase—while respecting the other parameters—were conducted to increase the agent's interaction. Additionally, the division into multiple training phases was also due to our limited access to GPUs in Google Colab accounts.

4. RESULTS AND DISCUSSION

4.1 Training Results in Phase 1

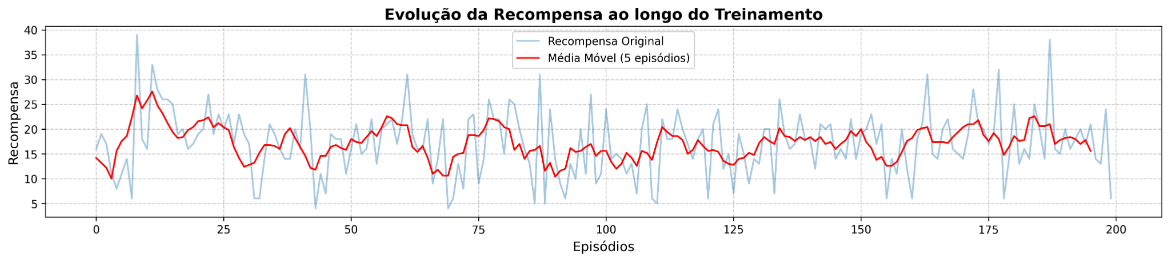

Figure 4.1: Reward Evolution Graph for Phase 1

In Figure 4.1, the reward evolution (blue line) represents the reward obtained in each episode, while the red line is the moving average over every 5 episodes. There is fluctuation in the episode-to-episode rewards, indicating that the agent has not yet fully stabilized its performance. The moving average (red line) suggests that, although some episodes achieve higher rewards, there is no clear upward trend. Instead, it oscillates around an intermediate level.

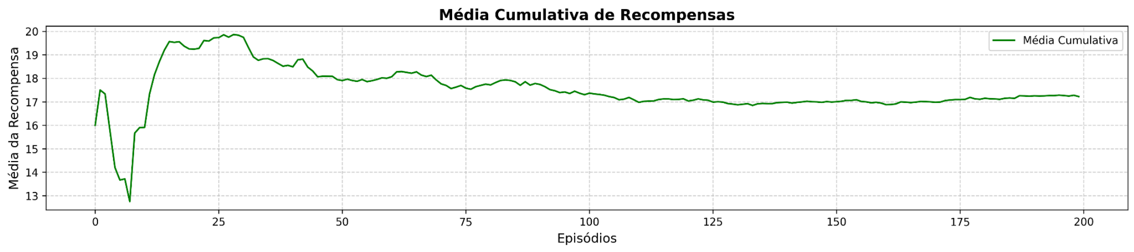

Figure 4.2: Cumulative Average Reward Graph for Phase 1

Initially, the cumulative average reward starts relatively high. As training progresses, it drops to a lower level and then stabilizes. This stabilization may indicate that the agent adopted a strategy that does not improve in the long term but also does not severely degrade performance.

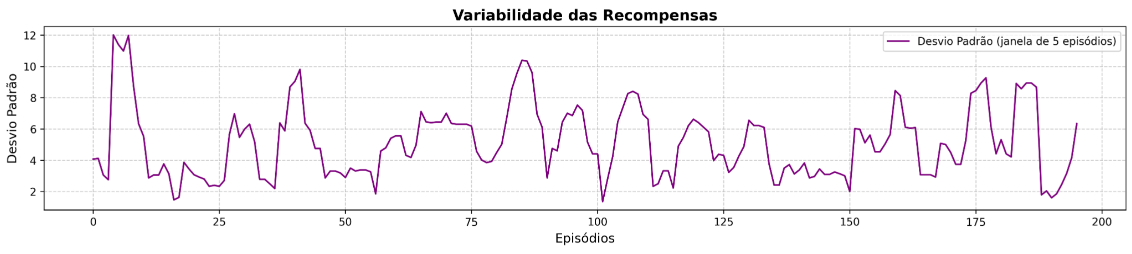

Figure 4.3: Reward Variability Graph for Phase 1

The purple line represents the standard deviation of rewards within a 5-episode window (i.e., how dispersed the rewards are within each window). Some points show peaks in variability, suggesting that the agent sometimes has very good episodes followed by poor ones. Lower variability, combined with higher rewards, would be the ideal scenario, as it would indicate consistency in performance.

4.2 Training Results in Phase 2

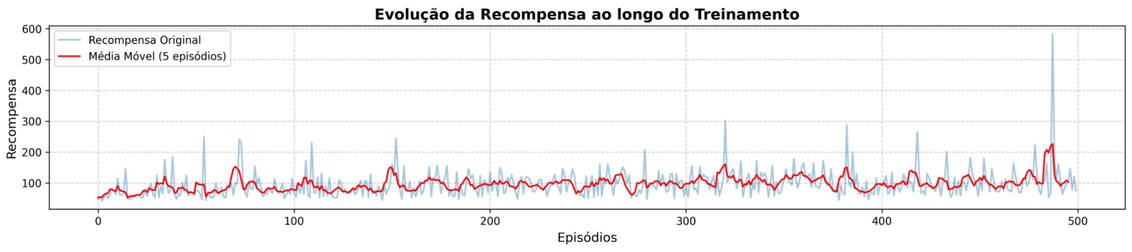

Figure 4.4: Reward Evolution Graph of Phase 2

Although there are peaks in some episodes, there is a trend of growth over time, albeit with fluctuations. This suggests that the agent occasionally obtains higher rewards but still goes through episodes with low or zero rewards.

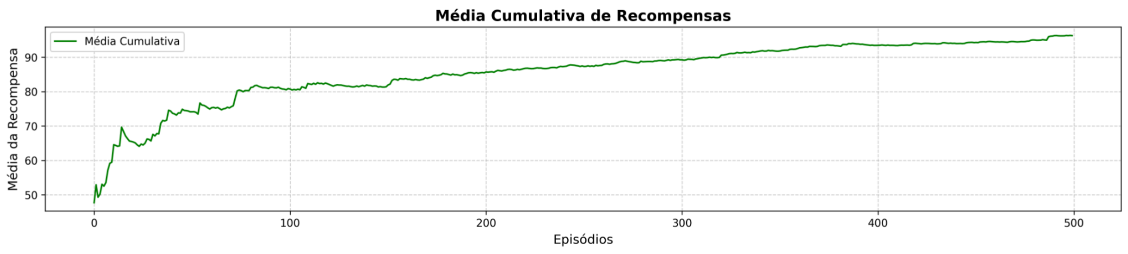

Figure 4.5: Cumulative Average Reward Graph of Phase 2

This graph rises relatively consistently, indicating an overall improvement in the agent's performance over the episodes. When the curve stabilizes or grows slowly, it means the agent is approaching a more consistent policy. In this case, we see an almost constant growth, which is a positive sign of learning.

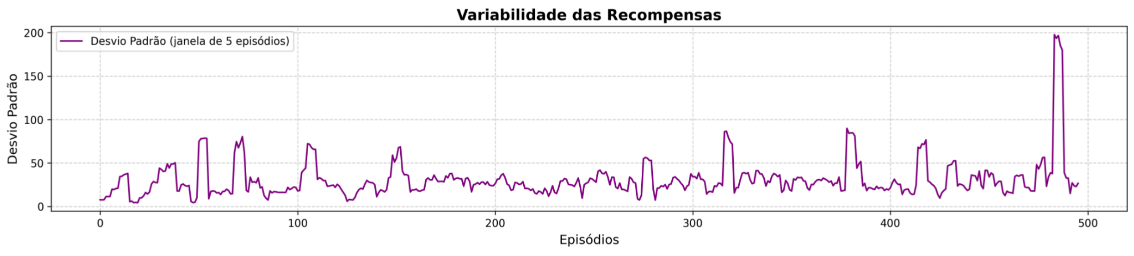

Figure 4.6: Reward Variability Graph of Phase 2

For most of the time, variability remains at a relatively low level, but with some peaks. These peaks indicate "outlier" episodes—where the agent performed significantly better (or worse) than the average. The trend of maintaining a generally low standard deviation (except for occasional peaks) suggests that the agent is gaining some stability in its playing strategy, though some unpredictability remains in certain situations.

4.3 Training Results in Phase 3

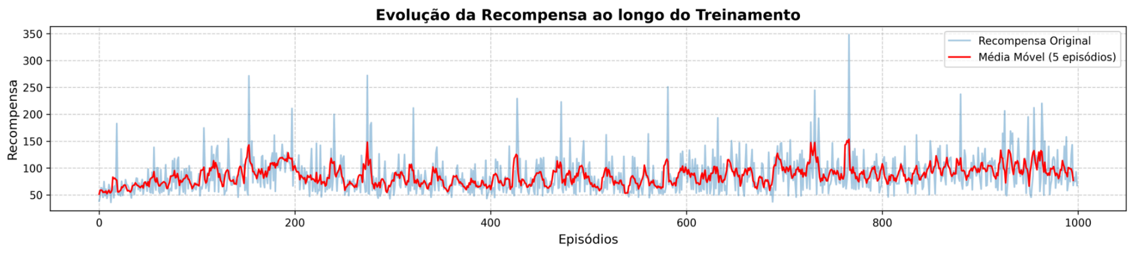

Figure 4.7: Reward Evolution Graph of Phase 3

There are episodes in which the agent obtains higher rewards, interspersed with episodes of low rewards. The red moving average does not show a very pronounced upward trend but seems to remain at a slightly increasing or nearly stable level.

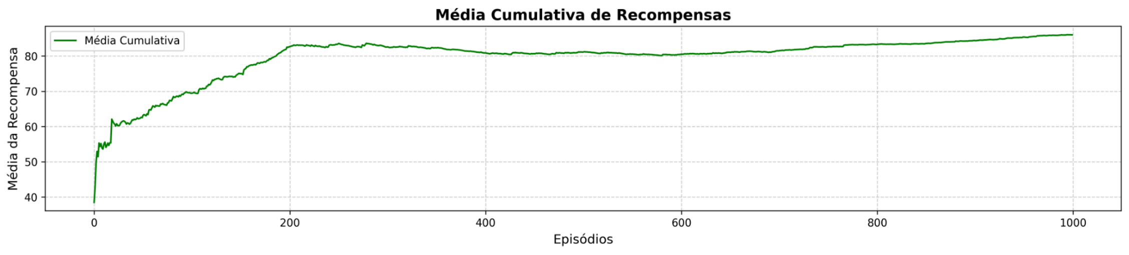

Figure 4.8: Cumulative Average Reward Graph of Phase 3

The curve rises relatively consistently, indicating overall improvement over the episodes. Between approximately 300 and 400 episodes, there is a faster gain, after which the growth slows down but remains positive. The cumulative average is increasing, showing that, in general, the agent has been improving over time.

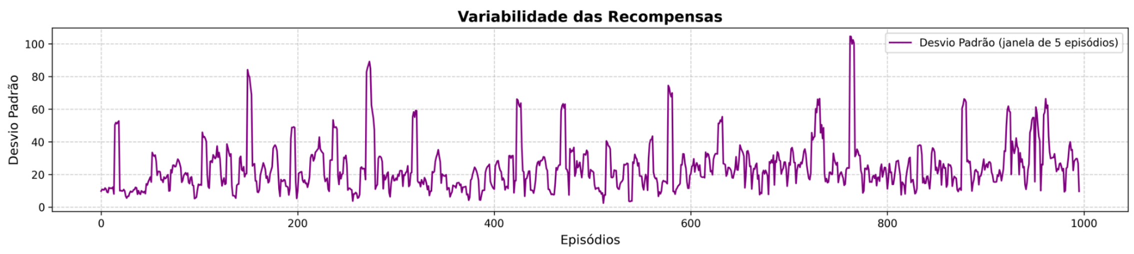

Figure 4.9: Reward Variability Graph of Phase 3

In this phase, there are several variability peaks, showing that performance occasionally fluctuates significantly between nearby episodes. Despite these peaks, there is no clear trend of sustained increase or decrease in variability, indicating that the agent is still alternating between good and bad episodes. This may occur if the agent is still exploring extensively, suggesting a need for adjustment in the exploration scheme (epsilon decay).

4.4 General Analysis

The three result series highlight the agent's gradual progress. From 200 to 500 episodes, there are clearer signs of improvement, and from 500 to 1000 episodes, the cumulative average increases even further, although with peaks and fluctuations. The agent is not 100% consistent but is continuously learning and obtaining higher average rewards over time. Overall, the agent is learning, but at a relatively slow pace, with good and bad episodes interspersed. This dynamic is common in reinforcement learning; with more training time and adjustments, the agent may stabilize at a higher reward level.

5. CONCLUSION

In this study, we explored the development and training of reinforcement learning (RL) agents to play Pac-Man using a deep Q-network (DQN) approach enhanced by a multilayer perceptron (MLP). We implemented advanced learning techniques, including convolutional neural networks to extract visual features and stabilization strategies such as experience replay and target networks.

Experiments show that, throughout the training phase, the agent gradually learns how to play, improves its performance, and accumulates higher rewards. However, we observed performance fluctuations over the episodes, suggesting that adjusting hyperparameters, such as the exploration rate and discount factor, could help improve learning stability and efficiency.

The cumulative average increases when comparing the rewards for each of the three stages (200, 500, and 1000 episodes), indicating that the agent is learning something useful in the environment. At all stages, there are reward fluctuations from one episode to another. This is a standard behavior for reinforcement learning algorithms operating in complex environments. However, as the number of episodes increases, agents tend to explore less (if ε decreases) and consolidate the learned policy. To achieve truly robust policies, Atari game policies, such as Pac-Man, typically require thousands or even millions of steps. Therefore, even 1000 episodes may not be sufficient to reach an optimal local policy. However, the evolution of the graphs shows that each phase adds learning.

The initial MCTS approach was considered unfeasible due to computational costs and long simulation times, confirming that DQN was suitable for this type of problem. A second key analysis was understanding that, although the agent has made significant progress, it can still be improved with more advanced techniques, such as Prioritized Experience Replay or dueling DQN, to enhance its decision-making and learning efficiency.

This study reinforces the importance of combining deep neural networks with reinforcement learning to create intelligent agents capable of learning in complex environments.

6. NEXT STEPS

The next steps include expanding and refining the agent's training to achieve more robust convergence and superior performance. First, it will be necessary to significantly increase the number of training episodes, considering that complex environments like Pac-Man on Atari often require thousands of episodes for policy consolidation. Additionally, hyperparameters related to exploration, such as the initial and minimum epsilon values and decay rate, should be adjusted to gradually reduce exploration and favor the use of the learned policy.

Other optimization points include refining the reward function by adjusting survival bonuses and termination penalties to encourage longer episodes and minimize premature terminations. The replay buffer capacity may also be revised to ensure the necessary diversity of experiences. Exploring variations in the network architecture, particularly in the MLP section, could further improve value function approximation. Finally, continuous monitoring of reward and variability graphs will help assess the impact of these adjustments and guide future modifications.

7. REFERENCES

1. Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, et al. OpenAI Gym [Internet]. arXiv preprint arXiv:1606.01540; 2016 [cited 24 Feb 2025]. Available from: https://arxiv.org/abs/1606.01540
2. Guo X, Singh S, Lee H, Lewis R, Wang X. Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. In: Advances in Neural Information Processing Systems (NIPS); 2014. [cited 24 Feb 2025].
3. Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, et al. Playing Atari with Deep Reinforcement Learning [Internet]. arXiv preprint arXiv:1312.5602; 2013 [cited 24 Feb 2025]. Available from: https://arxiv.org/abs/1312.5602
4. Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, et al. Human-level control through deep reinforcement learning. Nature. 2015 Feb;518(7540):529-33, doi:10.1038/nature14236 [cited 24 Feb 2025].
5. OpenAI. ChatGPT: modelo de linguagem baseado em IA [Internet]. OpenAI; 2025 [cited 24 Feb 2025]. Available from: https://chat.openai.com
6. Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, et al. Mastering the game of Go without human knowledge. Nature. 2017 Oct;550(7676):354-9, doi:10.1038/nature24270 [cited 24 Feb 2025].
7. Russell S, Norvig P. Artificial Intelligence: A Modern Approach [Internet]. Fourth Edition. Pearson; 2020 [cited 24 Feb 2025]. Available from: https://aima.cs.berkeley.edu