Breakout AI Game: Q Learning and Approximate Function

Rupal Jain
MS Computer Science, The University of Arizona
jainrupal@arizona.edu

***Abstract*—This paper explores the application of Q-Learning and Approximate Q-Learning to the classic Breakout game using a custom Pygame environment. I developed an AI agent that learns to optimize gameplay strategies through reinforcement learning. The study begins with Q-Learning to establish basic gameplay mechanics, then advances to Approximate Q-Learning to handle the same state space more efficiently. By tuning parameters such as the learning rate, discount factor, and exploration rate, the agent demonstrates significant improvements in gameplay. This approach underscores the adaptability of reinforcement learning algorithms to complex decision-making environments and highlights the potential for broader applications in similar tasks.**

1. Introduction

Reinforcement learning (RL) is a pivotal area of machine learning where agents learn to make decisions by interacting with an environment. Unlike supervised learning, RL agents are not trained with the correct answers but must discover them through trial and error, maximizing cumulative rewards. Among the various RL algorithms, Q-Learning stands out for its ability to handle problems with stochastic transitions and rewards without needing adaptations to the underlying model. This project employs Q-Learning and its variant, Approximate Q-Learning, to autonomously navigate the strategic decisions of playing the Breakout game.

1.1. Motivation:

The primary purpose of this research is to demonstrate the capability of Q-Learning algorithms to effectively train an AI agent that can master the Breakout game—a classic arcade game where success can be clearly measured by the score. The project explores how these algorithms can be implemented in a custom environment built with the Pygame library, focusing on their ability to learn and optimize gameplay strategies over time. The agent learns to determine the best actions based on its current state, which includes the positions of the paddle and ball, and the ball's velocity.

The choice of Breakout for this study is driven by several factors:

- **Simplicity and Control:** Breakout's straightforward mechanics allow the agent to focus on learning basic RL principles without the overhead of more complex game dynamics.
- **Quantifiable Success:** The clear objective of maximizing the score by breaking bricks provides a direct measure of the agent's performance and learning progression.
- **Challenge in Simplicity:** Despite its simplicity, the game presents challenges in optimizing strategies, particularly in handling random ball movements and efficiently clearing bricks.

1.2. Objective

The project aims to achieve the following objectives:

- **Algorithm Implementation:** To develop a robust framework for Q-Learning that can be applied to a discrete, simplified game environment. This includes setting up the game's state and action spaces, defining reward structures, and implementing the learning algorithm to interact with these components.
- **Performance Enhancement:** To optimize the Q-Learning algorithm's parameters, such as the learning rate (alpha) and the discount factor (gamma), to improve the AI agent's efficiency in learning and decision-making. This involves experimenting with different parameter settings to find the most effective combinations for rapid and sustainable learning.
- **Introduction of Approximate Q-Learning:** To extend the capabilities of traditional Q-Learning by incorporating Approximate Q-Learning, which uses feature extraction and generalization techniques to manage larger state spaces.
- **Comparative Analysis:** To conduct a systematic comparison between the standard Q-Learning and Approximate Q-Learning in terms of learning speed, long-term rewards, and computational efficiency. This will help in understanding the trade-offs and benefits of each approach under varying game dynamics and complexity levels.

1.3. Scope

This paper details the construction of the Breakout game environment, the application of Q-Learning and Approximate Q-Learning algorithms, and the evaluation of the AI agent's performance across different training scenarios. Additionally, the study discusses the implications of algorithmic adjustments such as learning rate variations, discount factor settings, and the impact of different exploration

strategies on the agent's ability to learn and adapt to the game dynamics.

1.4. Game Environment Description

The custom-built Breakout game environment is designed to facilitate the training and evaluation of reinforcement learning algorithms, specifically Q-Learning and Approximate Q-Learning. This environment is developed using the Pygame library and conforms to the OpenAI Gym interface standards, providing a standardized approach for defining agents' actions, states, and rewards. Below are the key components of the environment:

1.4.1. Game Dynamics and Setup. The environment simulates the classic Breakout game, where the primary objective is to break bricks arranged in a grid at the top of the game window using a ball that bounces off a paddle controlled by the player (or AI agent). The game area is a 640x480 pixel window with a black background, providing a clear contrast for the game elements.

1.4.2. Paddle. The paddle is positioned at the bottom of the game window and can move horizontally. The agent controls the paddle with three possible actions:

- Stay in place
- Move left
- Move right

The paddle's movement is restricted to ensure it remains within the game's horizontal boundaries.

1.4.3. Ball. The ball starts from the middle of the window just above the paddle and moves in a set direction at game start. Its movement includes both vertical and horizontal components, with its initial velocity randomly determined to add variability to the game. Collisions with the game window's walls, the paddle, or bricks affect the ball's trajectory. Specifically, contact with the paddle and left, right and top walls reverses the ball's vertical direction, while hitting bricks removes the bricks and the ball keeps on going.

1.4.4. Bricks. Bricks are arranged in multiple rows at the top of the screen. Each brick is removed when hit by the ball, scoring points for the agent. The layout and number of bricks can be adjusted to increase the game's difficulty.

1.4.5. Observations and States. The agent's observation space provides a comprehensive view of the game state at any given moment, which is crucial for making informed decisions. The observation is derived from the game environment and includes:

- **Paddle Position:** The discretized x-position of the paddle within the game window, normalized to a range or set of discrete segments. This helps the agent determine where the paddle is and how it might need to move to intercept the ball.
- **Ball Position:** The x and y coordinates of the ball are also discretized. The x-coordinate helps in understanding the ball's lateral movement, while the y-coordinate indicates its vertical position, crucial for predicting ball trajectories.
- **Ball Velocity:** The velocity of the ball, including both horizontal (left or right) and vertical (up or down) directions. This is vital for predicting future positions of the ball and planning the paddle's movement accordingly.

1.4.6. Rewards and Game Termination. The agent's performance is directly influenced by a structured reward system, which motivates specific behaviors conducive to success in the game.

The reward system is designed to reinforce beneficial actions while penalizing detrimental ones, as follows:

- **Hitting a Brick:** Each brick hit by the ball grants a reward of +1. This incentivizes the agent to aim at the bricks to maximize score.
- **Hitting the Paddle:** When the ball contacts the paddle, the agent receives a reward of +1. This encourages the agent to keep the ball in play by positioning the paddle under the ball.
- **Survival Reward:** Every tick that the game continues without losing the ball, the agent earns a small survival reward of +0.5. This encourages prolonged play.
- **Losing the Ball:** If the ball falls below the paddle, indicating a missed catch, the agent incurs a significant penalty of -50. This large negative reward helps in emphasizing the importance of ball retention.
- **Clearing All Bricks:** Clearing all bricks on the screen results in a bonus of +10. This large reward serves as an incentive for the agent to clear the level efficiently.

The game terminates under two primary conditions, which delineate the boundaries of an episode for the RL agent:

- **Win Condition:** All bricks are destroyed, which not only gives a large bonus but also signals a successful completion of the game level.
- **Loss Condition:** The ball falls below the paddle, ending the current game session with a substantial penalty and necessitating a game reset.

This environment setup not only challenges the agent to learn effective strategies for maximizing the score but also provides a controlled platform for studying the behavior and performance of different learning algorithms in a dynamic and visually interpretable context.

2. Methodology

This section outlines the reinforcement learning (RL) methodologies employed in the project to train an AI agent to play the Breakout game. The methods discussed include Q-Learning, a foundational RL technique, and its extension, Approximate Q-Learning, which is adapted to handle larger state spaces.

2.1. Reinforcement Learning Fundamentals

Reinforcement Learning involves training agents to make a sequence of decisions by interacting with a specific environment. In RL, an agent learns to perform actions in various states based on the feedback received in the form of rewards. The ultimate goal is to maximize cumulative rewards over time. This project utilizes RL principles to enable autonomous learning in the Breakout game, focusing on long-term reward maximization.

2.2. Q-Learning

Q-Learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It operates by estimating the value of action-state combinations using the Q-function, which is updated using the Bellman equation.

2.2.1. Algorithm Description. The agent updates its knowledge base, represented as a Q-table, with the formula:

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

where α is the learning rate, γ is the discount factor, $R(s,a)$ is the reward received after taking action a in state s, and s' represents the new state after the action.

2.2.2. Exploration vs. Exploitation. To balance exploration (discovering new actions) and exploitation (using known information), an ϵ-greedy strategy is employed. Here, the agent chooses a random action with probability ϵ and the best-known action with probability $1-\epsilon$. Over time, ϵ is decayed to shift the agent's strategy from exploration to exploitation.

2.3. Approximate Q-Learning

As the complexity of the environment increases, the state space becomes too large for a basic Q-table to handle efficiently. Approximate Q-Learning addresses this challenge by approximating the Q-function with a function approximator, such as a linear combination of features extracted from the state.

2.3.1. Feature Extraction. In Approximate Q-Learning, the state representation is transformed into a feature vector, which reduces dimensionality and captures essential information. Each feature corresponds to a characteristic of the state that is relevant to the decision-making process.

2.3.2. Learning with Function Approximation. The weights associated with these features are learned using a similar update rule as in basic Q-Learning, adjusted for function approximation:

$$Q(s,a,\mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s,a)$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[r + \gamma \max_{a'} Q(s',a',\mathbf{w}) - Q(s,a,\mathbf{w})]\mathbf{x}(s,a)$$

where $\mathbf{w}$ represents the weight vector, and $\mathbf{x}(s,a)$ is the feature vector for state s and action a.

2.4. Parameter Tuning

Critical to the success of learning algorithms is the tuning of parameters, including the learning rate (α), discount factor (γ), and exploration rate (ϵ). The optimal values for these parameters are typically determined through a combination of empirical testing and theoretical considerations.

This methodology provides a robust framework for training an AI agent not only to play but also to excel in the Breakout game by continuously learning and adapting its strategies based on the game dynamics.

3. Code Description

Game Environment Description:
The Breakout environment includes a paddle, a ball, and a set of bricks. The paddle can move left or right to hit the ball and break bricks. AI Implementation:

- **State Space:** Includes the paddle's position, ball's position and velocity.
- **Action Space:** Consists of three actions—stay, move left, and move right.
- **Reward Structure:** Rewards are given for hitting the ball with the paddle, breaking bricks, and penalties are applied for losing the ball.

3.1. Breakout.py

This file defines the Breakout game environment using the gym framework, suitable for reinforcement learning experiments. It includes mechanics for initializing the game, managing game states, rendering visuals, and processing game actions.

3.1.1. Libraries Used.

- **random:** Used for generating random numbers.
- **OpenAI gym:** The main framework for creating and managing the game environment.
- **numpy:** Supports numerical operations, particularly for managing game states and calculations.
- **pygame:** Used for rendering the game visually. time: (Although imported, it's not used in the provided script).

3.1.2. Environment Initialization. The `Breakout` class initializes the game environment, including the dimensions, color settings, and game dynamics:

- **Action Space:** Defines the possible actions the agent can take. In this game, there are three discrete actions: staying in place, moving left, and moving right. This is defined using `spaces.Discrete(3)`.
- **Observation Space:** The state space is represented by the game's window dimensions, though for agent observations, it is discretized based on the position and velocity of the ball and the position of the paddle.

- **Game Components:** The environment initializes game components such as bricks, the ball, and the paddle. Bricks are statically placed at the top, while the ball and paddle are dynamically controlled during the game.

3.1.3. Game Dynamics. The dynamics of the game are defined through several key methods:

- **Reset:** The `reset` method reinitializes the game to start a new episode, setting all game components to their initial state and returning the initial observation.
- **Step:** The `step` method advances the game by one timestep using the action chosen by the agent. It updates the positions of the ball and paddle, checks for collisions, and returns the new state, reward, and whether the episode has ended.
- **Render:** The `render` method updates the game's visual representation on the screen or returns it as an RGB array, depending on the mode specified.

3.1.4. Collision Handling. Collision detection is a critical component of the game's logic:

- **Ball and Paddle:** Collision between the ball and paddle reverses the ball's vertical movement and adjusts its position to prevent overlapping.
- **Ball and Bricks:** Collisions with bricks deactivate the respective brick and increment the score (reward) for the agent.
- **Ball and Walls:** Collisions with the game's boundaries cause the ball to bounce back, reflecting its movement based on the surface it hits.

The game environment provides a comprehensive platform for training reinforcement learning agents, with clear rules and feedback mechanisms that allow for the effective application of learning algorithms.

3.2. Agent.py

This file implements a Q-learning agent for the Breakout game, developed with Python's numpy library and integrated into a custom game environment built using the Pygame framework. The agent learns optimal actions by interacting with the game environment to maximize the cumulative rewards.

3.2.1. Libraries and Modules.

- **numpy:** Used for numerical operations on arrays, crucial for handling the state and Q-table updates.
- **breakout:** Custom module for the Breakout game environment.
- **csv:** Module to handle reading from and writing to CSV files, used here for logging training results.
- **plotting:** Custom plotting module to handle the visualization of training results and learning curves.

3.2.2. Key Components and Attributes.

- **Environment:** Instance of the Breakout game, providing the necessary methods to interact with the game.
- **Q-table:** Stores the action-value pairs used to determine the optimal action for each observed state.
- **Learning Parameters:** Includes epsilon (exploration rate), alpha (learning rate), and gamma (discount factor), all essential for the learning algorithm.
- **Logging:** Data such as rewards, steps per episode, epsilon values, and alpha values are recorded for performance analysis.

3.2.3. Methods.

- **`train`:** Orchestrates the learning process over a specified number of episodes, adjusting the Q-table based on received rewards and updating the exploration rate and learning rate.
- **`test`:** Evaluates the performance of the trained agent by running a specified number of episodes and observing the agent's behavior without further learning adjustments.
- **`plot_metrics` and `plot_learning_curve`:** These methods from the plotting module visualize the learning progress and results.

3.2.4. Training Process. During training, the agent performs actions in the environment based on either exploration (random actions) or exploitation (actions based on the Q-table). The outcomes of these actions (rewards and new state observations) are used to update the Q-table according to the Q-learning formula. The exploration rate gradually decreases, allowing the agent to explore various actions initially and gradually rely more on learned experiences as training progresses.

3.2.5. Data Logging and Visualization. Training data such as rewards per episode, steps per episode, and the values of parameters like epsilon and alpha are logged into CSV files. This logged data is later used for analysis and visualization, providing insights into the agent's learning progression and the efficacy of the Q-learning algorithm implemented.

This comprehensive approach to implementing and analyzing a Q-learning agent helps understand the dynamics of reinforcement learning in a controlled game environment, highlighting challenges like exploration vs. exploitation trade-off and parameter tuning.

3.3. ApproximateQLearning.py

This file implements the Approximate Q-Learning algorithm for the Breakout game environment. Approximate Q-Learning is an extension of the basic Q-learning algorithm

that uses a function approximator to estimate the Q-values, which is particularly useful in environments with large or continuous state spaces where a traditional Q-table would be impractical.

3.3.1. Libraries and Dependencies.

- **numpy:** Used for numerical operations, particularly for handling arrays and matrix multiplication needed for feature computation and Q-value calculation.
- **sys:** Utilized for accessing system-specific parameters and functions, including command-line arguments.
- **csv:** Facilitates reading from and writing to CSV files, which store training metrics such as rewards and steps.
- **plotting:** A custom module to handle the visualization of training and testing metrics.
- **random:** Generates random numbers for the epsilon-greedy policy.

3.3.2. Key Components.

- **ApproximateQAgent Class:** Defines the agent that learns and makes decisions based on the approximate Q-learning algorithm.
  - Initializes with parameters such as learning rate, discount factor, and epsilon for exploration.
  - Uses a feature extractor to reduce the state space into a manageable format that the learning algorithm can efficiently process.
- **FeatureExtractor Class:** Extracts features from the state for use in the learning algorithm, reducing the dimensionality of the problem.
- **train_agent Function:** Manages the training process of the agent over a specified number of episodes.
- **test_agent Function:** Evaluates the performance of the trained agent on a set number of test episodes, providing insights into the generalization capabilities of the model.

3.3.3. Training Process. The training process involves interacting with the Breakout environment, where the agent selects actions based on an epsilon-greedy policy, observes rewards, and updates the feature weights according to the approximate Q-learning update rule. The weights are updated to minimize the temporal difference error, a measure of the difference between estimated and actual rewards.

3.3.4. Performance Evaluation. The performance of the agent is periodically evaluated by plotting metrics such as rewards per episode and steps per episode. These plots are crucial for understanding the learning progression and tuning the learning parameters.

3.3.5. Functionality and Flexibility. The module is designed with flexibility, allowing parameters such as learning rate, discount factor, and epsilon to be dynamically adjusted based on the agent's performance and convergence criteria. This adaptability is crucial for fine-tuning the agent's learning process to various environments or game settings within Breakout.

The Approximate Q-Learning implementation is a cornerstone for experimenting with more complex reinforcement learning strategies in environments where state spaces are prohibitively large for traditional methods, showcasing the adaptability and scalability of reinforcement learning algorithms.

3.4. Plotting.py

This module is designed to handle all the plotting needs of the project, specifically for visualizing the performance metrics such as alpha (learning rate), epsilon (exploration rate), steps per episode, and rewards.

3.4.1. Module Functionality.

- **Environment Variables Configuration:** Sets up environment variables to ensure that plots are correctly scaled and displayed, especially on systems with high DPI settings.
- **plot_metrics:** Generates plots for alpha, epsilon, and steps from the training process. This function takes in filenames for data sources and a base name for the output files, producing detailed graphs that help in understanding the training dynamics.
- **plot_learning_curve:** Visualizes the learning curve of the agent by plotting average rewards per episode, providing a clear view of the agent's performance improvement over time.

3.4.2. Key Technologies Used.

- **matplotlib:** Used for creating static, interactive, and animated visualizations in Python.
- **pandas:** Facilitates data manipulation and analysis, particularly useful for handling CSV files that store training metrics.

This module significantly aids in the evaluation of the agent's learning process, providing essential insights that are critical for refining learning parameters and strategies.

3.5. Challenges

- **State Space Complexity:** Managing a large state space with high dimensionality posed significant challenges in terms of both memory requirements and computational efficiency.
- **Algorithm Tuning:** Fine-tuning the learning rate, discount factor, and decay rates of epsilon to optimize the learning performance proved to be intricate. These parameters significantly influence the balance between exploration and exploitation, impacting the agent's ability to learn effectively.

- **Consistency and Local Maxima:** Ensuring consistent agent performance across different training sessions and avoiding local maxima in the learning landscape were ongoing challenges. These issues highlight the stochastic nature of training deep reinforcement learning models.

4. Results

In my experimental setup, I conducted a series of tests with variations in learning rates, epsilon values, and discount factors to observe their effects on the performance of Q-learning and Approximate Q-learning agents in the Breakout game environment.

4.1. Experimental Setup

I executed eight separate runs with distinct parameter combinations to compare the effectiveness of different configurations:

1) Standard Q-Learning:
   - Learning rate: 0.1 and 0.2
   - Epsilon: 0.8 and 0.9
   - Discount factor: 0.99
2) Approximate Q-Learning:
   - Learning rate: 0.1 and 0.2
   - Epsilon: 0.8 and 0.9
   - Discount factor: 0.99

Each agent was trained over 20,000 episodes, providing a comprehensive view of the learning progress over an extended period.

4.2. Observations and Analysis

The learning curves and steps per episode metrics provided insights into how quickly and effectively each agent configuration adapted to the game environment.

4.2.1. Learning Curves. The learning curves demonstrated variability in the average reward per episode, with some configurations showing a higher baseline or quicker improvement:

- Agents with a higher epsilon value tended to explore more at the beginning but achieved higher rewards as the training progressed.
- Approximate Q-learning generally reached higher rewards more consistently compared to standard Q-learning, suggesting that the feature-based approach might be capturing useful information more effectively.

4.2.2. Steps per Episode. Steps per episode also varied significantly across different settings, illustrating the impact of parameter tuning on the agent's ability to sustain longer playtimes or achieve goals faster:

- Higher learning rates often resulted in more abrupt changes in agent behavior, reflecting in the variability of steps per episode.

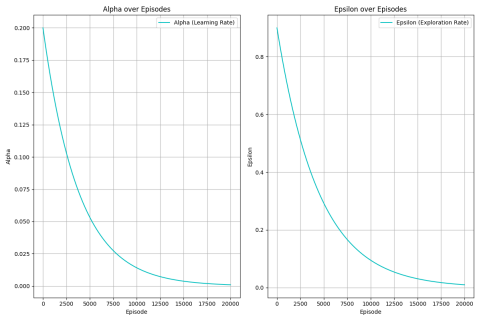

Figure 1. Learning rate and Epsilon per Episode

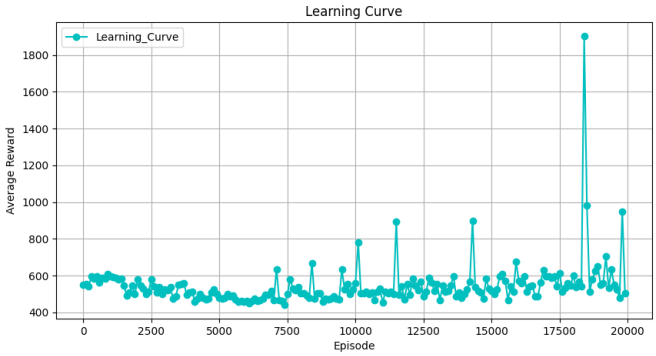

Figure 2. Q-Learning: Average Reward per 100 episode [Epsilon: 0.8, Alpha: 0.1]

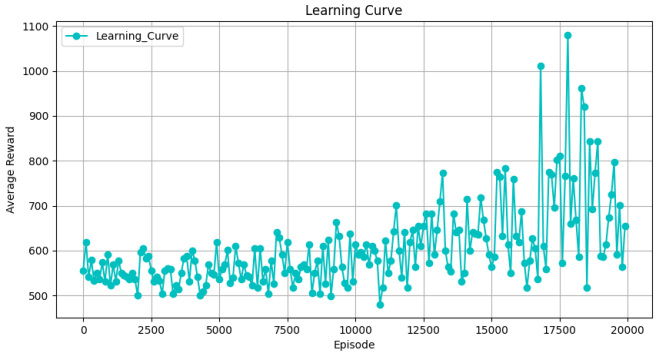

Figure 3. Approximate Q-Learning: Average Reward per 100 episode [Epsilon: 0.8, Alpha: 0.1]

- Consistent with the learning curves, Approximate Q-learning showed a trend towards higher efficiency in terms of steps needed to achieve similar or better rewards.

Note: Additional detailed graphs and data visualizations for each experimental setup are available in the project's GitHub repository, linked in the README.

5. Discussion

The results affirm the hypothesis that parameter tuning has a profound impact on the learning and operational efficiency of reinforcement learning agents. Both Q-Learning and Approximate Q-Learning demonstrated capacity for

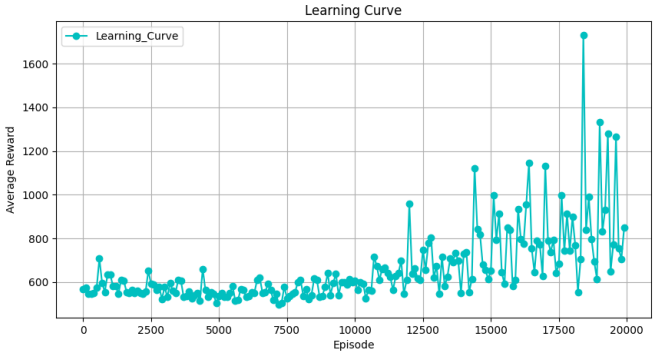

Figure 4. Q-Learning: Average Reward per 100 episode [Epsilon: 0.9, Alpha: 0.2]

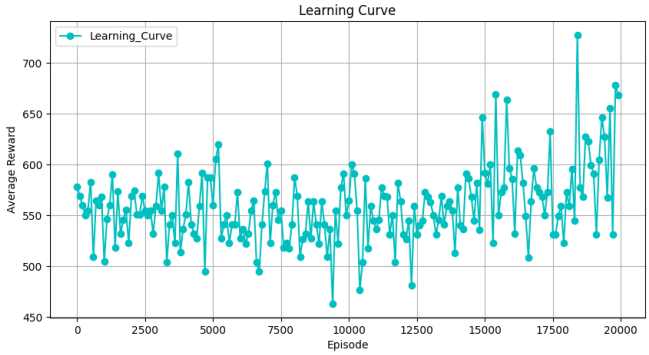

Figure 5. Approximate Q-Learning: Average Reward per 100 episode [Epsilon: 0.9, Alpha: 0.2]

significant improvement in game performance, with Approximate Q-Learning generally outperforming in most configurations due to its more nuanced state representation.

6. Conclusion

This study underscores the adaptability and potential of reinforcement learning techniques in game environments like Breakout. While both Q-Learning and Approximate Q-Learning have shown promising results, the latter offers a compelling approach for dealing with complex state spaces in dynamic settings. Future investigations could explore hybrid models or alternative reinforcement learning algorithms to further enhance learning efficiency and decision-making prowess.

7. Resources

For additional resources, detailed code explanations, and data used in this project, please refer to the following GitHub repository:

- GitHub: https://github.com/Rupaljain27/Breakout_QLearning

This repository includes all the necessary files and instructions required to replicate the findings presented in this report.

References

[1] University of Washington, "Approximate Reinforcement Learning," CSE473: Introduction to Artificial Intelligence, Autumn 2016. [Online]. Available: https://courses.cs.washington.edu/courses/cse473/16au/slides-16au/18-approx-rl2.pdf.

[2] "Implementing an Iterable Q-Table in Python." Medium, IECSE Hashtag. Available: https://medium.com/iecse-hashtag/rl-part-5-implementing-an-iterable-q-table-in-python-a9b515c2c1a.

[3] OpenAI. "Gym Documentation.". Available: https://gym.openai.com/docs/.

[4] A. Chen, T. Dewan, M. Trivedi, et al. "The Use of Reinforcement Learning in Gaming The Breakout Game Case Study," TechRxiv, Apr. 2020. DOI: 10.36227/techrxiv.12061728.v1. Available: https://www.techrxiv.org/users/662208/articles/675589-the-use-of-reinforcement-learning-in-gaming-the-breakout-game-case-study.

[5] "Title of the document." Stanford University. Available: https://cs.stanford.edu/~rpryzant/data/rl/paper.pdf.

[6] "Introduction to Q-Learning." DataCamp. Available: https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial.

[7] Swarthmore College, "Approximate Q-Learning," CS63: Artificial Intelligence, Spring 2016. [Online]. Available: https://www.cs.swarthmore.edu/~bryce/cs63/s16/slides/3-25_approximate_Q-learning.pdf.

[8] Gibberblot, "Function Approximation in Reinforcement Learning," RL Notes. [Online]. Available: https://gibberblot.github.io/rl-notes/single-agent/function-approximation.html.