Reinforcement Learning Agents for in-Game Tactics

Group 13 Project Report

Prakhar Dubey, Zarin Tasnim Biash

30 May 2025

Abstract

This report presents the application of Deep Q-Learning (DQL) to train an autonomous agent to play a competitive 2D painting game, where the objective is to maximize canvas coverage in a time-constrained environment. We reengineered the original JavaScript-based game to allow real-time communication with a Python-based reinforcement learning agent. Initially, this was achieved through a WebSocket-controlled interface embedded in the browser, later evolved into a fully headless Python simulation to accelerate training. The agent learns by observing simplified game states and choosing actions using a neural network trained with experience replay. To guide learning, we designed a reward system that encourages both painting new areas and exploring less-visited parts of the canvas. Our setup allows for quick training and better performance over time. This report explains our system design, training methods, results, and ideas for future improvements in multi-agent learning.

Contents

Chapter 1

Introduction

PaintBattle is a competitive 2D multiplayer browser game where players navigate a shared canvas and attempt to paint as much area as possible using their assigned colors. The game is time-constrained, and players can overwrite each other's painted areas. It features real-time movement, drawing mechanics, and collectible power-ups that temporarily enhance abilities. Originally designed for human keyboard input, the game includes multiple players but no built-in artificial intelligence. The strategic objective is to maximize canvas control before the timer runs out.

To help readers visualize the gameplay, here's a short clip in single player mode demonstrating how the original PaintBattle game works: **Gameplay Clip – PaintBattle**

Learning to make optimal decisions in real-time dynamic environments is a core challenge in reinforcement learning. In this project, we address this challenge through a custom 2D painting game, where an agent must cover as much area as possible within a fixed time. The original game was designed for human players and controlled via keyboard input. Our task was to convert this interactive game into a platform suitable for Deep Q-Learning. To achieve this, we restructured the game's control mechanism, allowing real-time communication with an external Python agent through WebSockets. The game now streams partial state information — including position, direction, drawing status, and canvas coverage — to the learning agent, which returns discrete movement commands (LEFT, RIGHT, FORWARD). Our contributions include:

- Modifying the browser-based game to support WebSocket-based AI control
- Developing a pipeline for exchanging game states and actions between the game and the RL agent
- Creating a headless Python simulation to enable high-speed training
- Designing a DQL framework that incorporates a custom reward function to encourage strategic exploration
- Visualizing the best trained model built using the headless Python simulator

The following sections elaborate on the system design, learning mechanisms, and performance evaluation.

Disclaimer: This project was carried out collaboratively by Groups 12 and 13, who worked together to develop the core infrastructure of the Paint Battle game environment, including modifications to the original JavaScript game, WebSocket-based communication, and the headless Python simulator. As a result, several foundational components of this report — particularly Sections 1 (Introduction), 2 (Background), and **some parts** of section 3 (System Design and Modifications) — are common across both the Reinforcement Learning and Genetic Algorithm reports. Each group independently explored and implemented their respective learning algorithms (DQL and GA), agent designs, and evaluation strategies. The shared sections reflect joint development efforts, while the methodology and results sections showcase each group's unique research direction.

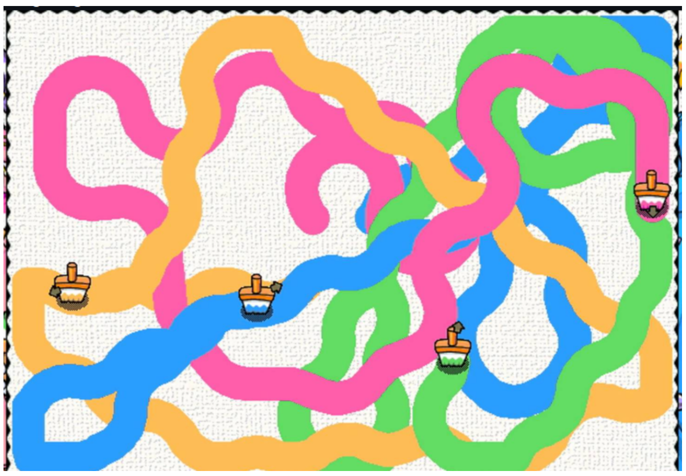

Figure 1.1: Paint Battle

Chapter 2

Background

The development of intelligent agents for games has seen significant advancements with the integration of deep reinforcement learning (DRL) techniques. DRL combines reinforcement learning (RL) with deep neural networks to enable agents to learn optimal policies directly from raw inputs. This approach has demonstrated reasonable success across various domains, including video games, robotic control, and autonomous systems [3].

A primary challenge in Q-learning methods is the overestimation of action values, which can lead to suboptimal policy convergence. van Hasselt et al. addressed this issue with the introduction of Double Q-Learning, which decouples action selection from action evaluation to reduce bias and improve performance in complex environments [2]. In our implementation, we focused on applying base Q-learning and advanced methods will be a part of future work.

The original *PaintBattle* game features up to four players, where only Player 1 is controlled via keyboard input. The remaining players follow random movements with no underlying intelligence or algorithmic control. The objective is to navigate a shared canvas and paint as much area as possible in the player's designated colour. Players compete to maintain control over painted regions while collecting random pickups—temporary bonuses that appear briefly on the map and expire if not collected in time. Any painted area can be overwritten by another player; the most recently traversed colour takes precedence.

The gameplay includes mechanics such as collision detection, timers, and various power-up effects (e.g., speed boosts or drawing penalties), all rendered in real time using JavaScript on an HTML5 canvas. The canvas dimensions are fixed at `600`×`600` pixels. The game lacks any form of artificial intelligence or learning mechanism [1], and all interaction is through manual keyboard input. No interface exists to connect external agents or automation tools, making it unsuitable for AI experimentation in its original form.

Player movement is controlled by adjusting an angular direction (referred to as "degree" in the codebase). At each time step, the player can choose to move forward, or adjust their heading to the left or right. Internally, movement is computed by updating the direction θ (in degrees), and calculating the new position (x, y) as follows:

$$\theta \leftarrow \theta + \Delta\theta \mod 360 \tag{2.1}$$

$$x \leftarrow x + \cos\left(\frac{\theta\pi}{180}\right) \cdot v \tag{2.2}$$

$$y \leftarrow y + \sin\left(\frac{\theta\pi}{180}\right) \cdot v \tag{2.3}$$

where $\Delta\theta$ is the user-controlled change in angle, and v is the current speed. This angular motion model introduces a continuous navigation space, which poses challenges for discrete-action agents.

To enable experimentation with learning-based agents, modifications were required to allow external control and real-time state exchange, which is addressed in the next section.

Chapter 3

Body

3.1 System Design and Modifications

To enable agent-based control and training in the *PaintBattle* game, several key modifications were made to the original codebase. These adaptations were essential for allowing an external program to perceive game state and interact with the environment through actions. The major components of the system design are described below.

3.1.1 WebSocket Integration for Agent Control

Purpose: Our first major challenge was bridging communication between the Python-based agent and the browser-based game. We addressed this by implementing a WebSocket interface. It is created to allow the Python DQL agent to communicate directly with the game and control the player's movement.

Before:

```
window.onload = function () {
    PB.main.init();
};
```

After:

```
const ws = new WebSocket("ws://localhost:9080/rl-agent");
ws.onmessage = function (event) {
    const data = JSON.parse(event.data);
    if (data.action === "RESET") {
        PB.main.init();
    } else {
        PB.main.playerAction(data.action);
    }
};
```

Explanation: Originally, the game started automatically for human input via keyboard. We replaced this with a WebSocket client that listens for commands from the agent and triggers actions like RESET or movement (left, right, forward):

- `LEFT` – decreases the player's heading angle.
- `RIGHT` – increases the player's heading angle.
- `FORWARD` – moves the player in the current heading direction.

3.1.2 Sending Game State to the Agent

Purpose: To provide the agent with relevant game information for decision-making.

Before: Game logic executed internally without sharing state.

After:

```
const stateUpdate = {
    event: "STATE_UPDATE",
    player: {
        x: player.position.x, // x-coordinate of player
        y: player.position.y, // y-coordinate of player
        degree: player.angle, // Heading angle (0-360)
        canDraw: player.canDraw, // Boolean flag to indicate if player can paint
    },
    coverage: paintedPercentage // Percentage of canvas painted
};
PB.socket.sendToAgent(stateUpdate);
```

Explanation: This addition ensures the agent receives a real-time snapshot of the player's position, direction, ability to draw, and total canvas coverage.

3.1.3 Introducing rl-agent.js

Purpose: A script to manage two-way communication between the agent and the game allowing the game to send its current state to the Python agent and receive the agent's chosen action in return.

Snippet:

```
const socket = new WebSocket("ws://localhost:9080/rl-agent");
socket.onmessage = function(event) {
    const action = JSON.parse(event.data).action;
    PB.main.playerAction(action);
;
```

Explanation: We introduced this module to isolate and streamline message handling logic, reducing clutter in main.js and centralizing WebSocket control. It acts as a bridge between the game's rendering logic and the external WebSocket connection to the Python agent. While earlier modifications shifted some control into main.js, rl-agent.js organizes key functions like:

- Receiving messages from the server
- Extracting and formatting game state
- Dispatching user commands (e.g., RESET, actions)

3.1.4 Game Over Messaging

Before: No communication on episode end.

After:

```
PB.socket.sendToAgent({ event: "GAME_OVER", coverage: coverage });
```

Explanation: When a game round finishes, this line sends the final coverage score to the agent, enabling reward calculation and episode logging.

3.1.5 Player State Enhancements

Before: Player's angle or draw status were not transmitted.

After:

```
player.angle = ...;
player.canDraw = ...;
```

Explanation: Angle (heading direction) and draw status (canDraw) were added to state tracking to give the agent deeper context.

3.1.6 Python DQL Agent (dql.py)

Purpose: Implements the core Deep Q-Learning algorithm.

Snippet:

```
self.fc1 = nn.Linear(9, 64)
self.fc2 = nn.Linear(64, 64)
self.fc3 = nn.Linear(64, 3)
```

Explanation: The agent uses a 3-layer neural network to estimate Q-values for each of the three possible actions. Input includes position, angle, paint status, coverage, and density measures.

3.1.7 Reward Function

Before: No explicit feedback to guide the agent. **After**:

```
reward = r + (1 + overall_d) * (2*(1 - local_d) + (1 - medium_d) + 0.5*(1 - far_d)) * (
    coverage / 100)
```

Explanation: Combines canvas coverage and spatial density to encourage the agent to paint new, diverse areas while avoiding retracing paths.

3.1.8 WebSocket Server (websocket.py)

Purpose: Middleware to relay messages between browser game and DQL agent.

Explanation: This Python script creates two endpoints: one for the game and one for the agent. It forwards state and action data between them in real time.

These modifications collectively transformed the original human-controlled browser game into a fully automated reinforcement learning environment, allowing an agent to learn and improve its strategy over time. This allowed us to quickly prototype RL training, although it was limited by browser performance, which is why we decided to build a headless simulator for faster training.

3.2 Headless DQL Agent and Simulator Enhancements

Next, we wanted to enable fast and efficient training of a DQL agent without the overhead of a visual game environment, and to refine the reward logic using improved density metrics, stride scaling, and memory-efficient state updates. Ergo, to accelerate the training process, a separate headless simulator was implemented. This simulator replicates the core mechanics of the original game—including player movement and painting logic while omitting collisions, pickups and user interface components. It maintains compatibility with the original game's state and action interfaces, enabling faster training by eliminating visual overhead. For simplicity, the simulation environment includes only a single player. This setup allows the agent to focus on maximising canvas coverage without interference from other players. While this does not capture the full dynamics of the multi-player game, it provides a predictable environment that helps shape an effective base behaviour. Its effectiveness in real multi-agent scenarios remains to be evaluated through further testing. The major components of the system architecture are described below.

3.2.1 Introduction of headless_battle_painter.py

This file contains a streamlined simulation of the original game. Major changes include:

Before: No simulation environment was present and all the interactions occurred in a browser.

After: A fully Python-based environment has been built which is capable of processing game logic, movement, coverage, and canvas update.

Snippet:

```
self.canvas = np.zeros((self.grid_height, self.grid_width), dtype=bool)
self.density_maps = {
    "local": np.zeros_like(self.canvas, dtype=float),
    "medium": np.zeros_like(self.canvas, dtype=float),
    "far": np.zeros_like(self.canvas, dtype=float)
}
```

Explanation: The canvas is a boolean matrix updated when the player draws. Separate density maps track how often each grid cell is visited ultimately helping assess exploration behaviour.

3.2.2 Canvas Coverage Grid Logic

Purpose: Originally, coverage in the browser-based game was determined at the pixel level, which worked well for human players but was computationally expensive for reinforcement learning. Once the game was ported to a headless Python simulator, pixel-level accuracy became impractical. To simplify and accelerate learning, we redefined the canvas as a grid, enabling faster and more efficient reward computation. This change made it easier to track coverage progress and allowed the agent to understand its environment in more discrete and meaningful units. It can now simulate the real-world painting behavior in a grid-based environment by marking grid cells as painted when overlapped by the agent's brush, even partially.

Before: The original browser-based game utilized pixel-level rendering for coverage, which was not computationally efficient for reinforcement learning. There was no dedicated coverage grid logic for learning.

After: In the headless Python simulation, the canvas is implemented as a binary 2D array:

```
self.canvas = np.zeros((self.grid_height, self.grid_width), dtype=np.bool_)
```

During each update, if the player's brush overlaps with a grid cell, it is marked as painted:

```
if dx*dx + dy*dy <= self.player_radius * self.player_radius:
    self.canvas[y, x] = True
```

Explanation: The canvas is represented as a binary grid, where each cell indicates whether a region is unpainted (0) or painted (1). Initially, all cells are set to 0. As the agent moves, it uses a brush with a radius of 25 pixels. Any cell that falls within this brush radius—even if only partially—is marked as painted (1). The grid resolution is set to 30, meaning each brush stroke typically affects multiple cells.

In simpler terms, the brush is larger than each grid cell, so even a partial overlap is enough to mark a cell as painted. This setup effectively mimics the behavior of real-world painting and allows for more precise tracking of coverage, which is essential for calculating rewards during training.

3.2.3 Reward Function with Multi-Level Density Analysis

Initial Attempt (Discarded): Initially, the agent was rewarded based solely on how much new area it painted:

Before:

```
reward = current_coverage - previous_coverage
```

This worked at first, but we quickly noticed a problem: the agent would loop over the same painted area just to gain small rewards repeatedly. It was not exploring new zones. To fix this, we needed a smarter reward mechanism that would guide the agent toward unexplored areas and discourage redundant movement.

Alternate Attempt (Discarded): Our first alternative was to randomly pick grid cells that weren't painted and reward the agent for reaching them. But this didn't match how the original game worked—there was no directional logic or continuity. The brush seemed to teleport randomly, breaking the realism of the painting motion. So, we discarded this approach and designed a new system grounded in spatial density.

Final Approach: Coverage + Density-Based Reward: We split the reward into two parts:

- **Coverage Reward** — how much new area is painted.
- **Density Reward** — whether the area is already familiar or unexplored.

After:

```
coverage_reward = (current_coverage - previous_coverage) * 100
...
density_reward = progress_factor * (local_reward + medium_reward + far_reward)
...
total_reward = coverage_reward + density_reward
```

How are densities calculated?

We define three spatial zones around the agent using circular radii:

Type	Radius	Purpose
Local Density	Closest region to the player	Penalize staying in one spot
Medium Density	Surrounding mid-range	Encourage moderate exploration
Far Density	Distant region	Gently incentivize deeper territory access

Table 3.1: Comparison of three density regoins

Each time the agent draws, we scan these areas and count how many grid cells have been painted. This happens in the update_density_maps() function, which loops over all nearby cells:

```python
for dy in range(-self.radius_far, self.radius_far + 1):
    for dx in range(-self.radius_far, self.radius_far + 1):
        xi = (x + dx) // self.stride_x
        yi = (y + dy) // self.stride_y
        d = dx*dx + dy*dy

        if d <= self.radius_local**2:
            self.density_maps["local"][yi, xi] += 1
        elif d <= self.radius_medium**2:
            self.density_maps["medium"][yi, xi] += 1
        elif d <= self.radius_far**2:
            self.density_maps["far"][yi, xi] += 1
```

Over time, the counts in these grids increase. When calculating rewards, we normalize the counts to [0, 1] and then invert them using (1 - density) — this way, lower density means higher reward.

Reward Formulas:

```
# We set a strong reward for painting in low-density areas
local_reward = 2.0 * (1.0 - local_density) * coverage_reward

# Medium reward for heading toward low-density areas nearby
medium_reward = 1.0 * (1.0 - medium_density) * coverage_reward

# Slight reward for strategic positioning toward emptier regions
far_reward = 0.5 * (1.0 - far_density) * coverage_reward
```

Then we apply a progress factor (based on overall canvas coverage) to make unexplored areas even more valuable later in the game:

```
# Factor in overall progress - as game progresses, finding unpainted areas becomes more
  valuable
progress_factor = 1.0 + overall_density  # Increases as more of the canvas is painted

# Combine density rewards with progress weighting
density_reward = progress_factor * (local_reward + medium_reward + far_reward)
```

Why does this work?

- If the agent keeps revisiting the same area, local_density becomes close to 1, making (1 - local_density) close to 0.
- That leads to local_reward $\rightarrow$ 0, effectively discouraging redundant movement.
- Conversely, moving to a fresh zone with low density boosts reward.
- Over many episodes, the agent learns that higher long-term rewards come from exploring instead of repeating.

This combination of coverage and density creates a balanced reward function that mimics human-like painting behavior and encourages intelligent territory control.

Detailed explanations of local, medium, and far rewards are given below:

- **Local Reward**: Measures how frequently the agent has visited immediate surrounding cells. It discourages hovering over the same positions.
- **Medium Reward**: Evaluates the visitation within a broader radius. It encourages the agent to explore more widely within the local region.
- **Far Reward**: Assesses areas that are distant from the current agent position. This metric gently guides the agent towards under-explored regions of the canvas.

By combining these, the reward function helps the agent develop strategies that maximize coverage while intelligently avoiding redundancy, resulting in more efficient painting behaviour.

Furthermore, **Local, Medium, and Far Densities**: These values reflect how much the agent has already painted in nearby zones. If a cell (or area) has been painted repeatedly, its density is close to 1. The term (1 - local_density) becomes very small if the local area is already painted. That means the agent receives less reward for going there again. Over many episodes, the agent learns through trial and error that actions leading to repeated paths yield lower cumulative rewards. The DQN minimizes loss by updating Q-values in favor of higher-reward, low-density areas. In this way, the agent avoids repeating already-painted areas.

3.2.4 Agent State Representation

Snippet:

```
[x, y, degree, can_draw, coverage, local_density, medium_density, far_density,
    overall_density]
```

Explanation: All the values are normalized, and the inclusion of multiple densities gives the agent better spatial awareness of its environment.

3.2.5 Frames per Second (FPS) and Speed Changes

Before:

- FPS: 60 (60 states per second sent to agent)
- Player Speed: 2 pixels per second
- Player Turn Speed: 3 pixels per second
- Time per Episode: 60s

After:

- FPS: 10 (10 states per second sent to agent)
- Player Speed: 20 pixels per second
- Player Turn Speed: 30 pixels per second
- Time per Episode: 30s

Explanation: For headless training, we send fewer states (frames) to the agent as we don't need a high number of actions per second. We also increase the speed of the player and reduce the total time for an episode to enable faster training.

3.2.6 DQL Model and Learning Loop

Neural Network Architecture: The DQN agent uses a feedforward neural network to estimate Q-values for three discrete actions. The input vector consists of 9 normalized features representing game state. The network architecture is given below as a snippet.
Snippet:

```python
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

#state_size = 9
#action_size = 3
```

3.2.7 Inference using Visualization

Before:

- Model trained on the javascript based game was used
- Model performance was inferred during training loops itself

After:

- Best model trained on the headless simulator was used
- Performance check done on the game real time without the model training during inference

Chapter 4

Results

Model Choice - We trained our agent for 5000 episodes and saved the model that performed the best, that is, gave the highest canvas coverage, over a moving average of 50 episodes. This avoids picking a model that performs well but is an outlier and unstable for evaluation. We finally picked the model that gives us the best average performance at around 3500 episodes.

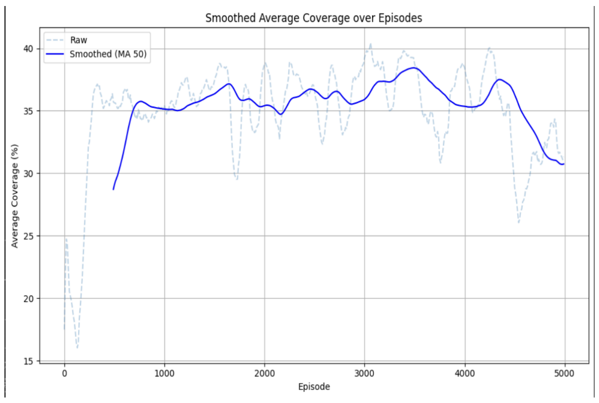

Figure 4.1: The coverage curve over episodes (with a moving average of 50)

Model Performance - At inference, the agent trained on the best model covers a minimum of 41% of the canvas over 5 different runs in a 30s timespan. While this performance is not optimal yet, we can see from Figure 4.1 that the model has clearly learned to cover more of the canvas over it's training period. Here is a link to one of the inference time runs by the agent with 41% coverage - **Inference Clip (30s)**

Analysis - As seen from the clip, the agent learns to not keep repainting the same area over and over again. It also learns to get unstuck from positions around the edges of the canvas. There is another interesting mechanical move where the agent wiggles while moving, to cover more of the canvas while going in a particular direction.

Chapter 5

Related Work

Our work builds on foundational reinforcement learning methods, particularly Mnih et al.'s Deep Q-Network (DQN) [2], and adapts it for a dynamic 2D environment. The original DQN demonstrated that deep neural networks, when combined with Q-learning and experience replay, can learn effective policies directly from raw pixel input, achieving human-level control in Atari 2600 games. However, its reliance on value overestimation in action selection posed generalization challenges in complex environments.

To mitigate this, we incorporated insights from the Double DQN algorithm proposed by van Hasselt et al. [2], which reduces overestimation bias by decoupling action selection and evaluation. This adaptation is particularly relevant for non-deterministic, adversarial settings like our multiplayer 2D game, where accurate action-value estimation is critical.

Beyond value-based methods, recent surveys such as Terven's [3] provide a comprehensive view of the evolution of deep reinforcement learning, highlighting the increasing use of state abstraction and hybrid actor–critic architectures. Inspired by these trends, we implement a type of state abstraction that reduces redundant spatial information. Our training pipeline also uses headless simulation to allow for efficient agent evaluation without graphics rendering overhead.

Research in model-free multi-agent reinforcement learning has been carried out to learn strategies in general-sum games beyond traditional zero-sum formulations. Casgrain et al. [4] propose Nash-DQN, a data-efficient Deep Q-learning framework for learning Nash equilibriums in general-sum stochastic games. By using a locally linear-quadratic approximation of the game dynamics, Nash-DQN enables analytically controllable optimal actions. It also maintains the flexibility of deep neural networks for function approximation. Unlike classical Q-learning approaches that struggle with scalability in multi-agent settings, this method circumvents the need to explore all state-action pairs explicitly. Its success in learning competitive strategies in electronic trading environments suggests strong potential for similar structured yet dynamic domains like *Battle Painters*, where agents must adapt to both cooperative and adversarial behaviors.

Chapter 6

Discussion

Our reinforcement learning pipeline demonstrates several notable strengths:

- **Modular architecture:** The transition from the original JavaScript game to a headless Python environment significantly accelerated training by eliminating browser-related bottlenecks and enabling direct integration with machine learning tools.
- **Reward shaping with density awareness:** Incorporating a density-based penalty discouraged the agent from revisiting already painted areas, leading to more efficient coverage strategies.
- **Compact state representation:** A normalized, low-dimensional input space helped ensure training stability and allowed the agent to generalize better across different states.

However, several challenges and limitations were encountered:

- **WebSocket communication handling:** Reliable agent-game communication required robust WebSocket management, which initially posed synchronization issues.
- **Slow in-game training:** Early-stage training within the browser was inefficient and resource-heavy, prompting the shift to headless simulation.
- **Reward modeling:** Designing a meaningful reward function based on grid values (e.g., partial coverage, density zones) required careful tuning to avoid unintended behaviors.
- **In-game inference complexity:** Real-time decision-making during live gameplay introduced latency and made debugging harder.
- **Lack of temporal modeling:** The use of a simple feedforward neural network limited the agent's ability to learn from past actions or sequences. Temporal models like LSTM or RNN could capture richer patterns.
- **Single-agent limitation:** The current setup supports only solo training, whereas multiplayer scenarios could introduce more complex and realistic dynamics.
- **Implicit environment modeling:** The agent indirectly inferred the canvas state from reward signals instead of having an explicit internal map of the painted area.

Possible Enhancements: Looking ahead, several improvements can strengthen the pipeline:

- Implementing hyperparameter tuning to optimize learning performance.
- Exploring alternative reward formulations, such as location-specific incentives rather than density-based penalties.
- Supporting multiplayer training and inference, allowing agents to learn cooperative or adversarial strategies.
- Introducing in-game pickups or power-ups to diversify gameplay mechanics and enrich agent learning objectives.

– Replacing the basic neural network with more advanced architectures (e.g., convolutional networks or attention-based models) to better capture spatial and temporal patterns.

Chapter 7

Comparison with Genetic Algorithm

Aspect	**Deep Q-Learning (DQL)**	**Genetic Algorithm (GA)**
Learning Type	Model-free RL	Evolutionary optimization
Exploration Strategy	ϵ-greedy, noise	Mutation and crossover
Data Usage	Experience replay buffer	Population of candidate solutions
Update Mechanism	Gradient descent (via backprop)	Fitness-based selection
Convergence Speed	Faster with stable policy	Slower and stochastic
Hyperparameters	ϵ, learning rate, discount γ	Mutation rate, crossover rate
Suitability	Dynamic environments	Complex fitness landscapes
Parallelism	Difficult	Naturally parallel

Table 7.1: Comparison of Deep Q-Learning and Genetic Algorithm approaches

Chapter 8

Conclusions and Future Work

This project successfully showcases the potential of Deep Q-Learning in autonomous gameplay through the development of an intelligent agent for a customized 2D painting game. Starting from a browser-based environment designed for human interaction, we transformed the game into a reinforcement learning-compatible system. By integrating WebSocket communication, we enabled real-time coordination between the game engine and a Python-based agent. To accelerate training and eliminate visual dependencies, we further decoupled the user interface and implemented a high-performance, headless simulation environment.

One of the most impactful contributions of this work lies in the design of a dynamic and strategic reward function. Instead of relying solely on canvas coverage, we introduced a density-aware mechanism that encourages the agent to explore new regions and avoid redundant paths. This approach, combined with Deep Q-Learning techniques such as experience replay and target network stabilization, allowed the agent to learn robust policies that generalize well across episodes.

Looking forward, there are several promising directions to extend this work. Introducing multiple agents would enable competitive or cooperative multi-agent reinforcement learning scenarios, enriching the strategic complexity of the environment. The observation space can also be expanded to incorporate additional contextual information, such as opponent positions and power-up locations, enabling more nuanced decision-making. Moreover, testing recurrent neural network architectures could help the agent model temporal dependencies and plan across longer horizons.

This work represents a strong foundation for bridging reinforcement learning with real-time strategy games and highlights the importance of thoughtful simulation design, reward shaping, and training architecture in developing intelligent agents.

Chapter 9

Contributions

The work presented in this report was conducted jointly, with both authors actively collaborating on all aspects of the project, including ideation, development, and experimentation. All decisions and results were discussed and executed together.

Group 13:

- **Prakhar Dubey**: Played an equal role in all stages of the project and later focused on visualizing the headless simulator after we got our best trained model.
- **Zarin Tasnim Biash**: Played an equal role in all stages of the project and later focused on writing and compiling the report after we got our best trained model.

Bibliography

[1] A1rPun. *PaintBattle – a multiplayer web game made with node.js and phaser.* Available at: `https://github.com/A1rPun/PaintBattle`.

[2] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., *et al.*, *Human-level control through deep reinforcement learning*, Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[3] Juan Terven. *Deep Reinforcement Learning: A Chronological Overview and Methods.* AI, vol. 6, no. 3, article 46, 2025. Available at: `https://www.mdpi.com/2673-2688/6/3/46`. DOI: 10.3390/ai6030046.

[4] Philippe Casgrain, Brian Ning, and Sebastian Jaimungal. *Deep Q-Learning for Nash Equilibria: Nash-DQN.* Applied Mathematical Finance, vol. 29, pp. 62–78, 2022. DOI: 10.1080/1350486X.2022.2136727.