1. Introduction

Reinforcement learning provides a useful framework for studying how agents learn complex behaviors through interaction, and the Snake game offers a clear environment where both short-term decisions and long-term planning are required for success. Because the game becomes increasingly difficult as the snake grows, it serves as an effective testbed for evaluating how well an agent can adapt its strategy over time. Understanding this process is valuable for applications in game AI, autonomous navigation, and other sequential decision-making tasks.

We chose this project to apply concepts from class, such as Markov Decision Processes, Q-Learning, and Deep Q-Networks to a real system where we could observe learning behavior directly. Our goal was to design an agent capable of improving its gameplay performance through trial and error and to analyze how modeling choices, such as reward structure and state representation, influence learning outcomes.

The main challenges we encountered involved giving the agent enough information to avoid self-trapping, stabilizing learning with function approximation, and dealing with the randomness of food placement. We addressed these by modeling Snake as an MDP, implementing a Double DQN, and refining our feature set from 9 to 21 dimensions to improve the agent's awareness of its environment.

Overall, our results show clear improvement across training attempts. The enhanced state representation significantly increased both maximum and average scores and reduced common failure modes like corner trapping. While long-term planning remained difficult for the agent, the progress demonstrates how reinforcement learning techniques can be effectively applied to strategic gameplay environments like Snake.

2. Machine Learning Task

The goal of our project is to train an agent to play Snake using **reinforcement learning**, where the model learns through interaction rather than labeled examples. Formally, we define the task as a Markov Decision Process (MDP) with the following components:

- **Inputs (State Representation):**
  A feature vector describing the current game state, including the snake's position and direction, relative food location, and indicators of danger in multiple directions. We use two state configurations in our experiments: a 9-dimensional feature set and an expanded 21-dimensional feature set.
- **Outputs (Action Space):**
  One of three discrete actions: **turn left**, **turn right**, or **move forward**.

- **Learning Setting:**
  A **Deep Reinforcement Learning** framework using a Double Deep Q-Network (Double DQN). The agent learns a value function $Q(s,a)Q(s,a)$Q(s,a) that estimates long-term reward for each action in a given state.

External Resources

- **Programming Language:** Python 3.12
- **Libraries:**
  - PyTorch 2.x for neural networks and optimization
  - NumPy for numerical processing
  - Matplotlib for plotting results
  - Pygame for environment visualization and game mechanics
  - tqdm for progress tracking
- **Hardware:**
  Training was performed on a standard CPU environment; no GPU acceleration was required for our model sizes.
- **Repository:**
  Full implementation available at: *github.com/seth-aguilar/cpts-437-rl-snake-game*

Data and Optimization Target

The agent does not rely on a static dataset. Instead, data is generated interactively as the agent plays episodes of Snake, collecting sequences of *(state,action,reward,next state)(state, action, reward, next\ state)*(state,action,reward,next state) transitions. The optimization target is to maximize the **expected cumulative reward**, balancing immediate gains (collecting food) with survival-oriented strategies (avoiding collisions).

Research Questions

Throughout the project, we focus on the following questions:

1. How does state representation (9 vs. 21 features) impact learning stability and final performance?
2. How effectively can a Double DQN agent learn long-term strategies in a constrained and increasingly crowded environment?
3. What performance patterns emerge across training—such as convergence speed, score distribution, and behavioral consistency?

4. Which failure modes persist, and what aspects of the problem make them difficult for DQN-based agents?

Key Challenges

Several difficulties make Snake a nontrivial reinforcement learning task:

- **Sparse and delayed rewards:** Food appears infrequently relative to the number of steps.
- **Dynamic difficulty:** The playable area shrinks as the snake grows, increasing planning complexity.
- **Partial observability and state aliasing:** Limited features can cause the agent to misjudge dangerous configurations.
- **High variance in training episodes:** Random food placement creates large performance fluctuations.
- **Long-horizon credit assignment:** Early mistakes can cause self-trapping many steps later, making it difficult for the agent to learn safe long-term strategies.

3. Technical Approach

Overview

Our approach combines several established techniques in deep reinforcement learning with domain-specific adaptations for the Snake game. The core algorithm is **Double Deep Q-Network (Double DQN)**, which addresses the overestimation bias in standard DQN by decoupling action selection from action evaluation. We enhance this with **curriculum learning** to progressively increase task difficulty, **reward shaping** to provide denser feedback signals, and an **engineered state representation** that encodes critical spatial information for navigation decisions.

State Representation

The state representation is critical for enabling the agent to make informed decisions. Our 21-dimensional state vector includes:

Immediate Danger Detection (3 features): - danger_straight: Binary indicator if moving forward leads to immediate death (wall or body collision) - danger_left: Binary indicator for left turn danger - danger_right: Binary indicator for right turn danger

Look-Ahead Danger Detection (6 features): - Danger indicators for 2 cells ahead in each relative direction - Danger indicators for 3 cells ahead in each relative direction

Diagonal Danger Detection (4 features): - Danger in the four diagonal positions adjacent to the head (front-left, front-right, back-left, back-right)

Wall Proximity (3 features): - Normalized distance to the nearest wall in each relative direction (straight, left, right) - Values range from 0 (far from wall) to 1 (adjacent to wall)

Direction Encoding (4 features): - One-hot encoding of current movement direction (UP, DOWN, LEFT, RIGHT)

Food Information (5 features): - food_dx: Normalized x-distance to food (−1 to 1) - food_dy: Normalized y-distance to food (−1 to 1) - food_ahead: Binary indicator if food is ahead - food_left: Binary indicator if food is to the left - food_right: Binary indicator if food is to the right

Neural Network Architecture

We employ a **Multi-Layer Perceptron (MLP)** for Q-value approximation:

Input Layer: 21 neurons (state dimension)
Hidden Layer 1: 256 neurons + ReLU activation
Hidden Layer 2: 256 neurons + ReLU activation
Output Layer: 3 neurons (Q-values for each action)

The network maps states to Q-values through the following forward computation:

$$Q(s,a;\theta)=W_3 \cdot \mathrm{ReLU}(W_2 \cdot \mathrm{ReLU}(W_1 \cdot s+b_1)+b_2)+b_3$$

where $\theta=\{W_1,b_1,W_2,b_2,W_3,b_3\}$

represents the learnable parameters.

Double DQN Algorithm

Standard DQN uses the same network to both select and evaluate actions, leading to systematic overestimation of Q-values. Double DQN addresses this by using:

- **Online network**

Q(s,a;): Selects the best action

- **Target network**

Q(s,a;−): Evaluates the selected action

The target for updating the online network becomes:

$y = r + \gamma \cdot Q(s', \text{argmax}_a Q(s', a; \theta); \theta^-)$

This decoupling reduces the positive bias in value estimation, leading to more stable learning and better policies.

Curriculum Learning Strategy

Training proceeds through four stages of increasing difficulty:

Stage	Grid Size	Episode	Promotion Threshold	Max Steps without Food
1 - Tiny	8x8	150	Avg Score ≥ 2.5	100
2 – Small	12x12	200	Avg Score ≥ 4.0	150
3 - Medium	16x16	300	Avg Score ≥ 6.0	200
4 - Full	20x20	Remaining	N/A	200

The agent must achieve the promotion threshold (50-episode rolling average) to advance to the next stage. If the threshold is not met within the allocated episodes, training continues on the same stage. This approach allows the agent to:

1. Learn fundamental movement and danger avoidance on the tiny grid
2. Develop food-seeking strategies on the small grid
3. Handle longer snake bodies on the medium grid
4. Master the full game complexity on the final grid

Reward Shaping

To address sparse rewards, we implement distance-based reward shaping:

$r_{shaped} = r_{base} + 0.1 \cdot \mathbb{1}[d_{new} < d_{old}] - 0.1 \cdot \mathbb{1}[d_{new} > d_{old}]$

where *d* represents the Manhattan distance to food. This provides immediate feedback on whether the agent is making progress toward the goal, significantly improving sample efficiency in early training.

Training Protocol

The complete training loop is summarized:

Algorithm: Curriculum Double DQN Training

```
Initialize:
 - Online network Q with random weights θ
 - Target network Q̂ with weights θ⁻ = θ
 - Replay buffer D with capacity 50,000
 - Epsilon ε = 1.0

For each curriculum stage:
 Create environment with stage-specific grid size

 While not promoted and episodes remaining:
  state = env.reset()

  For each step until done:
   # Epsilon-greedy action selection
   If random() < ε:
    action = random action
   Else:
    action = argmax_a Q(state, a; θ)

   # Execute action
   next_state, reward, done, info = env.step(action)

   # Store transition
   D.push(state, action, reward, next_state, done)

   # Sample and learn
   If len(D) >= batch_size:
    batch = D.sample(64)

    # Compute Double DQN targets
    next_actions = argmax_a Q(next_states, a; θ)
```

```
    targets = rewards + γ * Q̂(next_states, next_actions; θ⁻)

    # Update online network
    loss = MSE(Q(states, actions; θ), targets)
    θ = θ - α * ∇loss

  # Update target network periodically
  If steps % 100 == 0:
    θ⁻ = θ

  state = next_state

# Decay exploration
ε = max(0.01, ε * 0.997)

# Check promotion
If avg_score(last 50) >= threshold:
  Advance to next stage
```

Addressing Key Challenges

Sparse Rewards: Our reward shaping mechanism provides step-by-step feedback based on distance to food, giving the agent gradient information even when food is far away.

Self-Trapping: The enhanced state representation includes look-ahead danger detection (2-3 cells ahead) and diagonal danger indicators, allowing the agent to anticipate traps before committing to dangerous paths.

State Space Complexity: The hand-engineered features provide a compact, informative representation that captures the essential decision-relevant information without requiring the network to learn to extract these features from raw grid observations.

Exploration-Exploitation: We use epsilon-greedy exploration with gradual decay from 1.0 to 0.01 over training. The curriculum structure also naturally encourages exploration by starting with simpler environments where random actions are less likely to be immediately fatal.

4. Evaluation Methodology

Since Snake doesn't rely on an external dataset, all evaluation is based on the transitions the agent generates while interacting with our environment. The “data” consists of state action reward next state pairs produced during training, and the only preprocessing involved is the normalization built into our feature vector. Because everything is created inside our own environment, there are no licensing issues and no filtering or cleaning steps.

Reinforcement learning does not use a train/validation/test split in the usual sense, so we evaluate the agent online by tracking performance over recent episodes. Rolling averages (e.g., last 50 episodes) let us measure learning progress without leaking evaluation data into training, since the agent never trains directly on these summary metrics. After training, we also run evaluation only episodes on the full grid to check whether the learned policy generalizes.

Our main metrics are average score, maximum score, and survival length. These reflect the actual objectives of Snake collecting food consistently, avoiding death, and maintaining stable behavior despite randomness in food placement. We also examine score distributions to understand how consistent the policy is, since large variance often indicates brittle strategies.

For baselines, we compare against a random action agent and our earlier 9 feature DQN model. The random agent shows how the environment behaves with no learning, while the earlier model provides a stronger baseline for evaluating the gains from Double DQN and the expanded feature set.

Hyperparameters were tuned through small controlled trials, adjusting one parameter at a time and watching for improvements in stability or score trends. All model versions were trained under the same curriculum and replay settings to keep comparisons fair. Because randomness affects both exploration and environment dynamics, we fix seeds for Python, NumPy, and PyTorch during major runs, and the spread in our learning curves reflects the remaining uncertainty episode-to-episode rather than across multiple separate trainings.

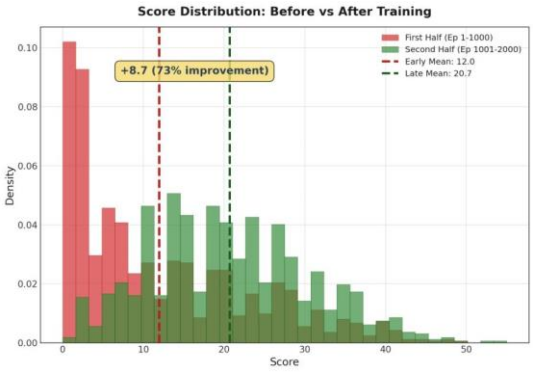
5. Results and Discussion

Our Double DQN agent showed clear improvement across both training attempts over 2,000 episodes. In the first attempt, we used 9 input features to describe the game state. The agent reached a maximum score of 54 and averaged 21.38 points by the end of training, which was a 73% improvement from the beginning (early average of 12.0 to late average of 20.7), as shown in the learning curve (Figure 1). However, we noticed the main

problem was that the agent kept trapping itself—it didn't have enough information to avoid getting stuck in corners as the snake grew longer. To fix this, we increased the input features from 9 to 21 dimensions, giving the agent better awareness of dangers in multiple directions around it. This change led to much better results in the second attempt: the maximum score jumped to 65 (+20%), the final average score reached 31.23 (+46%) (Figure 2), and the score distribution changed dramatically. Instead of most scores being close to zero, they spread out with an average around 26, which was a 410% improvement from the early training average of 5.1 (Figure 3 for first attempt; Figure 4 for second attempt).

(Figure 1)

(Figure 2)



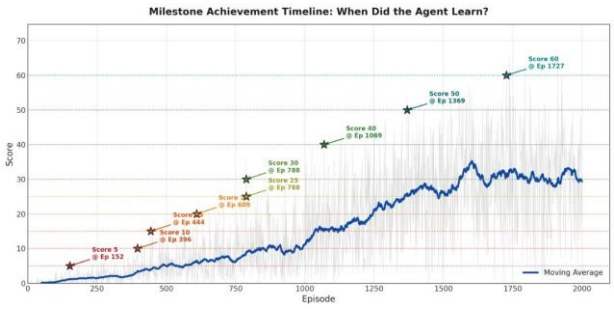
(Figure 3)

(Figure 4)

The progress timeline shows how the agent gradually hit different score milestones. It reached scores of 5, 10, 20, 30, 40, 50, and 60 at episodes 152, 396, 444, 788, 1069, 1369, and 1727 (Figure 5 for first attempt; Figure 6 for second attempt). While the improved state features greatly reduced self-trapping, the agent still leveled off around a score of 30. This suggests it struggles with long-term planning when the snake takes up a lot of space

on the board. The high variation in episode scores (shown by the shaded area in the learning curves) means the agent's performance was inconsistent, probably because food appears in random locations and the game gets harder as the board fills up. To improve further, we should add smarter path-finding so the agent can always reach the food, use curriculum learning to slowly make the game harder during training, and test memory-based models (like LSTM-DQN) that remember recent moves to help the agent plan better in crowded situations.

(Figure 5)



(Figure 6)

Contributions

Abdur Islam: 25%

Spencer Conn: 25%

Seth Aguilar: 25%

Quinn McCarthy: 25%