Group 88 Progress Report: "SNAKE"

Weichen Hou, Zihe Shi
{houw10,shiz40}@mcmaster.ca

1 Introduction

This project aims to develop a reinforcement learning (RL)-based control system for two autonomous agents (snakes) operating within a 2D grid-based game environment. The objective is to enable these agents to learn coordinated behaviors that allow them to navigate efficiently, reach designated targets, and avoid collisions with walls, obstacles, and each other. The environment is implemented using Python and Pygame, providing real-time visualization and dynamic interactions between the agents and their surroundings. The reinforcement learning framework is designed to optimize the decision-making of agents' through iterative training, where each agent observes its environment, selects an action, and receives feedback in the form of rewards or penalties.

A key focus of the project is the design of an effective reward structure that promotes cooperative behavior between agents. The reward function must balance multiple objectives — encouraging agents to approach the target while penalizing collisions or redundant movements. This requires careful tuning to prevent one agent from dominating the task or causing deadlock situations. To facilitate learning, the project uses state representations that include spatial awareness, directional orientation, and relative positioning of obstacles and targets.

In the current stage, the implementation focuses on a single autonomous snake to establish the foundational reinforcement learning environment, ensure stable training, and verify the consistency of the reward. The single-agent setup serves as a prototype for later expansion into a multi-agent cooperative system, where inter-agent communication and coordination strategies will be introduced. Once the environment and reward mechanisms are fully validated, the next phase will extend the framework to support multi-agent reinforcement learning (MARL), allowing both snakes to learn collaborative and competitive strategies simultaneously.

2 Related Work

Many prior efforts have explored applying reinforcement learning (RL) and deep-learning methods to the classic game Snake and related grid-world problems. Early work investigated using convolutional neural networks and Q-learning to learn Snake state-action mapping with reduced state spaces and reward shaping. Later, deep Q-networks (DQN) were applied more directly to Snake with full image-based inputs, demonstrating feasibility but also overfitting and limited generalization.

In the multi-agent domain, frameworks such as the Battlesnake Challenge allow simultaneous snakes competing or cooperating with human-in-the-loop training, presenting additional complexity when compared with our single-agent setting. No previous research or project addresses exactly our proposed task of multi-agent cooperative Snake. The most closely related is the deep Q-learning single-snake work Sebastianelli et al. (2021) which focuses purely on one agent navigating in isolation. We build on these foundations and extend them with cooperative dynamics, collision avoidance between agents.

3 Dataset

In this project, the data used for training and evaluation are not pre-collected static samples but are instead dynamically generated from the Snake game environment. Each episode of the game produces a sequence of state–action–reward–next state tuples that form the experience dataset used by the reinforcement learning (RL) agent. This dataset evolves continually as the agent interacts with the environment, learns from exploration, and updates its Q-values.

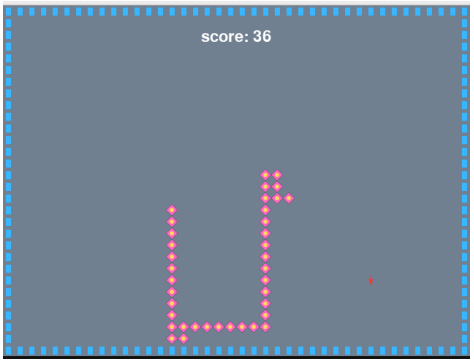


Figure 1: SNAKE demo

3.1 Environment Representation

This project uses the classic game 'snake' as the training environment, the game has a grid based map with 30 rows and 40 columns. The snake starts with 4 units length snake body and is able to move in the map matrix every sampling time to the left, right or straight direction based on itself. At every sampling interval, the snake can choose to move left, right, or continue straight based on its current direction. Every snake body unit will inherit the previous snake body's position and the first snake body will inherit the snake head's position. A piece of food will be generated on the map, When the snake's head reaches the same position as the food, the food is consumed and the snake's length increases by one unit. If the snake head collides with the its own body or is out the bound, the episode terminates and game is considered over.

Before training, all state matrices are flattened and normalized to values in the range $[0, 1]$. Each experience batch is randomly sampled from the buffer to break temporal correlations between consecutive states. The target Q-values are computed using the Bellman equation, and the dataset is iteratively updated after each gradient step, forming a self-reinforcing learning process.

4 Features

The state vector encodes the current environment information perceived by the Snake agent at each time step. It consists of three main categories of binary features: danger indicators, direction indicators, and food (point) indicators. The first twelve variables capture potential collision risks in different directions. The terms `danger_lb_r`, `danger_lb_l`, `danger_lb_u`, and `danger_lb_d` represent immediate one-step dangers (such as walls or the snake's body) in the right, left, upward, and downward directions, respectively. Similarly, `danger_lw_*` variables denote long-range dangers detected several cells ahead, providing early awareness of obstacles, while `danger_b_*` variables specifically identify whether the snake's own body is adjacent in a given direction. The next four variables, `dir_l`, `dir_r`, `dir_u`, and `dir_d`, form a one-hot encoding of the snake's current movement direction. Finally, the last four variables, `point_l`, `point_r`, `point_u`, and `point_d`, indicate the relative position of the food with respect to the snake's head, where each variable equals one if the food lies in that direction. Together, these twenty binary features provide a compact yet informative state representation that allows the reinforcement learning model to reason about obstacles, orientation, and food location when making movement decisions.

5 Implementation

5.1 Environment Setup

Each grid cell is assigned an integer index with the origin at the top-left corner. The movement direction is encoded by $\{0, 1, 2, 3\}$ for right, left, up, and down, respectively. Given the current direction `0,1,2,3`, the turns are updated as

$$\texttt{next\_idx} = \texttt{(idx + 1) \% 4} \quad (1)$$

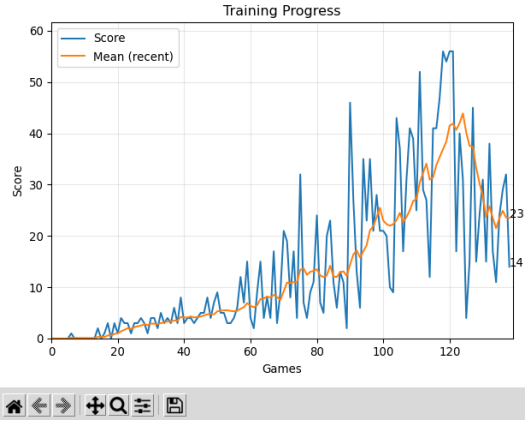
$$\texttt{next\_idx} = \texttt{(idx - 1) \% 4} \quad (2)$$

representing clockwise and counter-clockwise turns

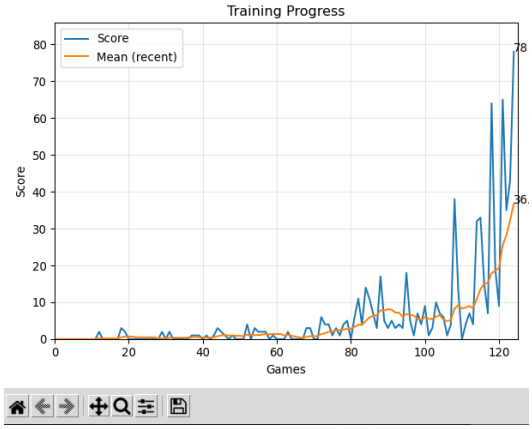
5.2 Reward and Penalty Analysis.

As illustrated in Figure 2, different reward-penalty settings lead to distinct learning behaviors in the Deep Q-Network (DQN) training process. Under the configuration of (+10, -1), the average score increases steadily and converges smoothly after approximately 100 episodes, indicating a balanced trade-off between exploration and exploitation. When the penalty is excessively large, as in (+10, -100), the learning progress stagnates for most of the training phase. This occurs because the strong negative reward discourages exploration, resulting in conservative but inefficient behavior. Conversely, assigning a higher positive reward, such as (+30, -1), accelerates the agent's improvement and yields higher peak scores; however, the increased variance suggests that the policy becomes

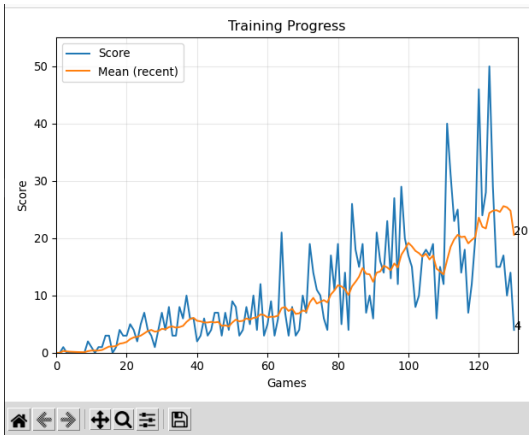
more aggressive and less stable. Overall, the (+10, -1) setting demonstrates the most stable and effective learning performance among the tested configurations.



(a) Reward (+10) and Penalty (-1)



(b) Reward (+10) and Penalty (-100)



(c) Reward (+30) and Penalty (-1)

Figure 2: Comparison of reward and penalty settings used in this project.

5.3 Network Architecture and Optimization.

The Deep Q-Network (DQN) agent is implemented using a fully connected neural network with one hidden layer. The input layer receives a 20-dimensional state vector representing environmental features such as wall proximity, body collision risks, movement direction, and food position. The hidden layer consists of 128 neurons with ReLU activation functions, enabling nonlinear feature extraction and efficient representation learning from the binary state inputs. The output layer produces four Q-values corresponding to the possible movement actions (*up*, *down*, *left*, *right*). The network parameters are optimized using the Adam optimizer with a learning rate of 0.001, and the loss function is defined as the mean squared error (MSE) between the predicted Q-values and target Q-values computed from the Bellman equation. A separate target network is periodically updated to stabilize learning and mitigate oscillations during temporal difference updates. For baseline comparison, the agent's performance is evaluated against a random-action policy, which selects each move uniformly without considering environmental feedback. This baseline provides a reference point to measure the improvement achieved by reinforcement learning through reward-driven training.

6 Results and Evaluation

Unlike traditional supervised learning approaches, as there is no fixed dataset to split into training, validation, or testing subsets. To evaluate the performance of the model, the key evaluation include:

- Average score per episode: Measures the average cumulative reward obtained by the agent, reflecting its learning progress over time.
- Maximum score achieved: Indicates the best performance reached during training or testing.
- Standard deviation of episode scores: Evaluates the consistency and stability of the agent's behavior across multiple episodes.

In addition, the score trend across episodes is also analyzed which suggesting that the agent may have over-optimized for short-term rewards or adopted the best survival strategies. This trend provides useful feedback for fine-tuning the reward design and improving policy stability.

7 Feedback and Plans

7.1 Feedbacks from TA

According to the received feedback, the overall quality and progress of the project is well presented. However, it was suggested that the report could be further improved by adding network structure or training framework diagrams, as well as graphs or tables that visualize experimental results and feature comparisons. Including more details about training parameters (e.g., learning rate, neurons) would also make the presentation more informative and professional.

7.2 Extension to Multi-Agent Scenarios

For multi-agent experiments, the dataset is expanded to include additional states representing the positions and actions of the other snake. Each transition includes joint or independent rewards for a cooperative setting. This allows the model to learn both individual survival strategies and team-based coordination patterns. Under this setting, an additional constraint is introduced to assign the reward to the snake that is closer to the target, thereby minimizing the time required to obtain the reward.

8 Equations

8.1 Q-Learning Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right] \tag{3}$$

8.2 Bellman Optimality Equation

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q^*(s', a') \,\middle|\, s, a \right] \tag{4}$$

In the DQN process, this two logic appear conceptually in the target calculation step.

8.3 Deep Q-Network (DQN) Loss Function

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \big)^2 \Big] \tag{5}$$

In this project, the loss function measures how close the Q-network's predictions are to the target values given by the Bellman equation. The target value is calculated as $y_t = r + \gamma \max_{a'} Q(s', a'; \theta^-)$, where θ^- are the weights of the target network. The Q-network with parameters θ predicts $Q(s, a; \theta)$ for the current state and action. The loss is the squared difference between the target and the prediction: $L(\theta) = (y_t - Q(s, a; \theta))^2$.

In the code, this is implemented using PyTorch's mean squared error loss (`torch.nn.MSELoss`), and the optimizer updates the network weights to make future predictions closer to the target values. This helps the agent learn better actions over time.

Team Contributions

For this project, Weichen Hou was primarily responsible for developing the mathematical model and implementing the reinforcement learning algorithm. Zihe Shi focused on setting up the training environment, adjusting reward and penalty parameters, and validating the mathematical consistency of the model. Both members collaborated on performance evaluation and final report writing.

References

Firdiansyah Ramadhan and Suyanto Suyanto. 2020. Royale heroes: A unique rts game using deep reinforcement learning-based autonomous movement. In *2020 3rd International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, pages 494–498.

Alessandro Sebastianelli, Massimo Tipaldi, Silvia Liberata Ullo, and Luigi Glielmo. 2021. A deep q-learning based approach applied to the snake game. In *2021 29th Mediterranean Conference on Control and Automation (MED)*, pages 348–353.

Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. 2019. A survey of deep reinforcement learning in video games. *Preprint*, arXiv:1912.10944.

Gabriel Synnaeve and Pierre Bessière. 2011. A bayesian model for rts units control applied to starcraft. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 190–196.

Zhepei Wei, Di Wang, Ming Zhang, Ah-Hwee Tan, Chunyan Miao, and You Zhou. 2018. Autonomous agents in snake game via deep reinforcement learning. In *2018 IEEE International Conference on Agents (ICA)*, pages 20–25.