SLITHERING SNAKE: Final Project Report (CS 596 - Prof. Zhang)

Jake de los Reyes, Han Luu, Itzel Orozco, Matthew Smith, Jenny Tran, Genesis Anne Villar

Department of Computer Science, San Diego State University, San Diego, CA, USA

***Abstract*—This paper presents the implementation, analysis, and refinement of a Deep Q-Network (DQN) reinforcement learning approach applied to the classic Snake game. We developed an agent that learns optimal movement strategies through direct interaction with the game environment rather than using pre-collected datasets. Our methodology employs an 11-dimensional binary state representation, a single-layer neural network with ReLU activation, and a carefully designed reward system to encourage efficient apple collection while avoiding collisions. Through systematic experimentation with four different configurations of reward structures and network architectures, we achieved an 11.6% increase in mean score over our baseline model, with the balanced reward configuration proving most effective (mean score of 28.6 with 602.88 frames average survival). Our comparative analysis between model and human performance revealed distinctive failure modes, with human players typically failing due to reaction delays, while our agent demonstrated strategic self-trapping behaviors. The complete implementation, training scripts, and experimental results are available at https://github.com/msmith6127/SnakeGameAI**

***Index Terms*—deep Q-network (DQN), reinforcement learning, machine learning, Python**

I. INTRODUCTION AND RESEARCH PROBLEM

THIS project explores the application of reinforcement learning to autonomous game-playing agents, a field that serves as an important testbed for AI techniques before they're applied to more complex real-world problems. Our focus aims to optimize the classic Snake game using machine learning approaches. The game takes place within a grid environment where every space in the grid is either empty space, the snake itself, or an apple. The snake starts three tiles long and every time that it eats an apple (by running into it), it grows one tile longer and the apple is replaced by another in a different random location. The game ends if the snake runs into a wall or itself. The goal of the game is to have the snake eat as many apples as possible in order to grow it as large as possible without running into anything. Movement is restricted to either turning left (pressing the "a" key), turning right (pressing the "d" key), or continuing straight (pressing nothing).

The complexity and variability caused by having such a large pool of possible game states make traditional strategies such as rule-based (conditional statements) or hard coding impractical, however, the simple movement, goal, and losing conditions of Snake make it the ideal game to be modeled using machine learning. In particular, the subfield of reinforcement learning is a great fit as it involves agents (in this case, the snake) interacting with environments (the grid the game takes place in), while receiving rewards (apples) or penalties (losing the game by running into walls or itself). This all collectively allows Snake to be analyzed as a turn-based, sequential decision-making problem. As the snake grows longer, the available safe moves decrease, requiring increasingly sophisticated planning and pattern recognition. This exact paradigm is the reason for our team's decision in choosing snake, as it encapsulates the right environment to study sequential decision-making under increasing complexity.

The primary research problem for our project is whether a relatively simple neural network architecture can successfully achieve expert-level proficiency in the Snake game through reinforcement learning. This question explores the fundamental balance between model complexity and task difficulty in reinforcement learning systems. Our team aims to create a model that will maximize reward (score) through evaluation of many simulations of the game, despite the challenges the model will have to deal with dynamic state transitions and increasing complexity as the game progresses.

II. RELATED WORK (LITERATURE REVIEW)

Reinforcement learning (RL) for game-playing agents have been extensively studied, with significant advancements in recent years. This section reviews key works relevant to our implementation of a Snake game AI agent, organized by thematic areas. The following sources to develop this project: University of Toronto[1], Medium[2], GeeksForGeeks[3], and ScienceDirect Vol. 503[4] helped us to get a better understanding of Deep Q-Learning and its practical applications. We used this knowledge in our implementation of our model.

A. Deep Q-Learning Fundamentals

Deep Q-Learning forms the foundation of our approach. Highlighted through the work done by Mnih et al.[1], it was demonstrated how deep neural networks could be used to approximate Q-functions effectively in Atari games, introducing key concepts like experience replay and target networks that we adopted in our implementation, especially with stabilizing training in high-dimensional state spaces. This was further expounded on by Amin of Medium [2], where their research delved into some key concepts with deep Q-learning,

such as Q-function approximation, experience replay, target network, and the Bellman Equation in deep Q-network (DQN). Particularly valuable was Amin's example of DQN implementation with the CartPole, a classic reinforcement learning game where an agent attempts to learn how to balance a pole on a moving cart. This resource details real-world applications in video game AI, robotics, and finance, illustrating the versatility of deep Q approaches and the fundamental challenges in Q-learning, particularly the instability in training neural networks with reinforcement signals. Amin[2] also provided examples of some real-world applications of deep Q-learning (such as video game AI, robotics, and finance) to help to explain the vast applicability of this type of machine learning.

The GeeksForGeeks team [3] complemented these resources with a deeper technical analysis of how deep learning enhances Q-learning to overcome challenges in high-dimensional state spaces and continuous input data. Their detailed explanation of exploration rate decay informed our implementation of the epsilon-greedy strategy, where we adopted a linear decay approach to balance exploration and exploitation. The article also includes information on the architecture of deep-Q-networks as well as some further applications of DQN, similarly to Amin's[2] article for Medium which helped to reinforce our understanding of these topics and possibilities.

B. State Representation and Architecture Design

While many implementations, including Mnih et al. [1], utilize raw pixel data as input, our approach diverges by employing a carefully engineered compact state representation. This design choice aligns with research showing that for certain environments, engineered state features can outperform raw pixel inputs while requiring significantly less computational resources. The core difference between our team's project and Mnih et al.[1] working on different games (various Atari games), whereas we were focusing on modeling exclusively the game Snake. Our 11-dimensional binary state vector approach builds upon observations from Mnih et. al's[1] work, regarding the importance of state representation, though we've adapted their approach for the specific constraints of the Snake game environment. This design choice reflects our hypothesis that a compact, domain-specific feature representation would be more efficient than raw pixel inputs for this particular application.

The selection of ReLU as our activation function was inspired by Dubey et al.[4], who conducted a comprehensive survey of activation functions in deep learning. Their research demonstrated that ReLU offers several advantages over traditional sigmoid and tanh functions, including reduced computational complexity and improved gradient flow during training-critical factors for our relatively simple neural network architecture. As they note, "The ReLU function solves the problem of computational complexity of the Logistic Sigmoid and Tanh functions" [4], which aligned perfectly with our goal of creating an efficient model for the Snake game environment.

C. Research Gap and Our Approach

Through our literature review, we identified a research opportunity regarding the minimal necessary complexity for effective Snake gameplay. We've noticed that some implementations employ quite convolutional neural networks or recurrent architectures, our team questioned whether a simpler feed-forward network could achieve comparable performance. The insights gained from this literature review directly informed our design decisions, particularly our choice to implement a relatively simple DQN architecture with careful state engineering and reward shaping, rather than a more complex neural network architecture. This decision reflects our interest in exploring the balance between model complexity and task performance in reinforcement learning systems.

In summary, Amin's[2] Medium article introduced us to some key concepts and applications while providing us some example code on how to implement these concepts, the GeeksForGeeks article provided us with a much more in depth understanding of the topics mentioned in the Medium article, and the University of Toronto article gave us some examples of how to go about creating our own machine learning solution to simple video games.

III. METHODOLOGY AND TECHNICAL DETAILS

A. Dataset Description

This section presents our approach to developing a reinforcement learning agent for the Snake game, detailing our dataset generation strategy, state representation, neural network architecture, and training methodology. The primary dataset for this project is not a static pre-collected dataset but rather dynamically generated data from the environment during the Snake game play itself. Each game run produces a series of states, actions, rewards, and next state transitions that are stored and used for training the reinforcement learning model.

We created a synthetic dataset during training in a tuple:
`(state_old,action_index,reward,state_new,done)`

- state_old: the current/next game state. Each state is an 11-dimensional binary vector representing:
  - Danger in the forward, right, left direction
  - Current snake direction: left, right, up, down
  - Relative position of food to head in four directions: left, right, up, down
- action_index: the action(s) taken in one of three directions - straight, left, right
- reward: reward(s) received:
  - Positive (+10) for eating food,
  - Negative (-10) for dying or aimlessly moving without making progress(eating food) to promote efficiency/improvement rather than just mere surviving
- state_new: environment's response to the agent's action
- done: indicates whether the episode was terminated

This approach created a self-improving feedback loop, allowing the agent to learn from its own experiences rather than from pre-determined examples. The environment itself was implemented using Pygame. providing a 20x20 grid where the snake navigates to consume food while avoiding collisions.

B. Data Pre-processing

A critical design decision in our implementation was the development of a compact, binary state representation. Rather than using raw pixel data as input, we engineer an 11-dimensional binary vector encoding.

Feature engineering: The state vector is manually engineered for each game frame to be compact and informative by converting raw game grid values into meaningful data represented by 11 binary values:

- 1.) Danger detection (3 binary values): uses data from the snake's direction and position or wall position, via a look-ahead mechanism, to check if moving in a direction will result in a collision with itself/the wall for each left, right, straight direction
- 2.) Movement direction (4 binary values): current snake direction (up, down, left, right)
- 3.) Relative food location (4 binary values): calculated by comparing food and snake head coordinates in the four directions

Normalization: Since all values in the state vector are represented in binary (0 or 1), no normalization was needed.

Augmentation process: No traditional data augmentation was applied, as reinforcement learning does not use any static datasets in the traditional sense. However, gameplay variability combined with balancing of exploration and exploitation over many episodes did ensure diversification of state-action combinations.

This preprocessing pipeline was executed in real-time during gameplay, with each frame being transformed into our 11-dimensional state vector before being passed to the neural network for action selection or training.

C. Neural Network Design

The core of our model is a Deep Q-Network (DQN) implemented in PyTorch. The architecture is a simple feed-forward neural network with 3 layers:

- Input layer: 11 neurons (based on the state vector)
- Hidden layer: 128 neurons, ReLU as activation function
- Output layer: 3 neurons, snake move in direction: left, right, straight

The following activation functions were chosen based on several considerations:

- ReLU activation: ReLU is used for hidden layers due to its simplicity, efficiency in handling non-linearities and ability to reduce the diminishing gradient problem and computationally faster than alternatives like sigmoid or tanh [4].
- Hidden layer size (128 neurons): This size strikes a balance between model expressiveness and computational efficiency, enough to learn relevant patterns without causing overfitting or slow training.
- Single hidden layer: This is sufficient for the task, making the model less complex and faster to train while still effectively learning the Q-values.

These parameters were selected based on empirical testing and aligned with common practices in DQN implementations.

D. Training and Implementation

The core of our training strategy is based on Deep Q-Learning, an off-policy reinforcement learning algorithm that uses a neural network to approximate the optimal action-value function. The training loop is implemented in `train.py`, where the agent repeatedly interacts with the game environment to learn an optimal policy for maximizing score.

Each training iteration consists of the following 5 steps:

1) Current state extraction:

```
state_old = agent.get_state(game)
```

The current state of the game is captured using the get_state() method, represented as an 11-dimensional binary vector.

2) Action selection, derived from current state:

```
final_move = agent.get_action(state_old)
```

The agent chooses an action using an epsilon-greedy strategy via `get_action(state_old)` where the exploration rate epsilon linearly decays over time. Specifically:

$$\epsilon = \max(0, \epsilon_0 - n)$$

Where:

- $\epsilon_0 = 80$ is the initial epsilon value
- n is the number of games played so far

After each training episode of n, the agent selects a random action with probability $\frac{\epsilon(n)}{200}$, otherwise chooses the action with the highest predicted Q-value

$$a = \begin{cases} \text{random action}, & \text{if rand}(0,1) < \frac{\varepsilon(n)}{200} \\ \arg\max_a Q(s,a), & \text{otherwise} \end{cases}$$

Where:

- a is the action chosen
- $Q(s,a)$ is the predicted reward of taking action a in state s
- $rand(0,1)$ is a random number drawn uniformly between 0 and 1
- $\epsilon(n)$ is linearly decayed presented above

This strategy balances exploration (random actions) and exploitation (choosing the best-known action). Epsilon acts as a dynamic threshold that decays over time. This ensures the agent explores more early on and gradually shifts into exploitation as it gains experience. The resulting action is represented as a one-hot vector for the three possible moves:

- `[1, 0, 0]` = straight
- `[0, 1, 0]` = right turn
- `[0, 0, 1]` = left turn

3) Environment feedback and resulting next state:
`reward, done, score = game.play_step(final_move)` and `state_new = agent.get_state(game)`

The selected action is passed to the game engine through play_set(final_move), which returns a reward:

- Positive: +10 for eating food
- Negative: -10 for dying, -10 for aimlessly moving without making progress(eating food)

Afterwards, the new game state after the predicted action performed is extracted using the get_state() function from the agent.

4) Short-term training:
`agent.train_short_memory(state_old, final_move, reward, state_new, done)`

The agent performs a single-step update using train_short_memory(). This step trains the neural network immediately on the most recent transition tuple `(state_old, action_index, reward, state_new, done)`. This fast feedback helps the model learn critical transitions, especially in early training. In this method, it calls on `train_step()` from `model.py`, which implements the core Q-Learning update, based on the equation:

$$Q_{\text{target}} = \begin{cases} r, & \text{if done} \\ r + \gamma \max_{a'} Q(s', a'), & \text{otherwise} \end{cases}$$

Where:

- r is the reward
- γ is the discount factor, $\gamma = 0.9$
- $max_{a'}Q(s', a')$ is the predicted maximum Q-value for the next state s' across all possible action a', if the training run is not done.

When the run is done, the future rewards don't contribute, so the Q-target is simply the reward r. This equation defines how the model updates its action-value function based on both immediate reward and the predicted value of future states (if not done). This function above is also reused late during long-term training.

The predicted Q-values are adjusted using the equation above, and the network weights are updated to minimize the loss between the current prediction and the target Q-values using MSE loss and back propagation. We use the Adam optimizer with a learning rate LR = 0.001 to update the model parameters. Adam is chosen for its adaptive learning rate capabilities and efficiency on sparse or noisy gradients, making it well-suited for our reinforcement learning model.

5) Experience replay:
`agent.remember(state_old, final_move, reward, state_new, done)`

The same tuple is stored in the replay memory buffer via `remember()`. This buffer is later sampled randomly in `train_long_memory()` to train models also via `train_step()` on a batch of past experiences, which helps reduce variance and stabilizes the Q-network updates.

This loop is the core of the reinforcement learning process, implementing the agent's interaction with the environment. This training loop is repeated indefinitely across game runs. After each game (when done == True), the long term memory is used to train our model with a batch of previously stored experiences. This mechanism enables the model to generalize better across different situations and improves stability. It encapsulates the agent-environment loop where the neural network gradually improves its policy based on feedback.

Software and libraries utilizations:
Programming language: Python
Notable libraries:

- Torch (2.6.0): for building and training the neural network, chosen for its dynamic computation graphs
- Pygame (2.6.1): for developing the game environment, allowing direct control over the game logic and rendering
- Numpy (2.2.4): for numerical operations, handling state representations and vector operations
- Matplotlib (3.10.1): for live visualization of scores

IV. EXPERIMENTAL RESULTS AND ANALYSIS

This section presents the evaluation of our DQN-based Snake agent across multiple experimental configurations, analyzing performance trends and behavioral patterns.

A. Evaluation Metrics

To evaluate our Deep Q-Learning agent for the Snake game, we utilized several key metrics:

- Score: track both the average score per game and the highest score achieved during training
- Survival Time: measured the average number of steps/frames the agent survived per game
- Learning Progression: analyzed the reward trend over time and the loss curve to determine how effectively the agent was learning from experience
- Exploration-Exploitation Balance: monitored how the epsilon value (starting at 80 and decreasing with the number of games) affected our agent's decision-making process over time

For each configuration, we trained the agent for 500 games and recorded all metrics at regular intervals.

B. Results

We systematically tested four agent configurations to evaluate the impact of different choices:

1.) Baseline Configuration: A standard DQN implementation with simple reward signals (+10 for food collection, -10 for any collisions)
2.) Distance-Based Rewards (CHANGE 1): added rewards based on the snake's proximity to food to encourage directed movement
3.) Balanced Rewards (CHANGE 2 + CHANGE3): implemented a balanced reward system incorporating:
    - Distance-based food rewards
    - Wall avoidance penalties
    - Self-collision avoidance
    - Time efficiency penalty to discourage aimless wandering
4.) Enhanced Architecture (CHANGE4): extended the previous configuration with:
    - Increased neural network size (256 in hidden layer)
    - Modified weight initialization strategy
    - Flood-fill pathfinding algorithm to avoid entrapment

TABLE I. Performance Comparison Across Agent Configurations

Configuration	**Avg Score**	**Record**	**Avg. Survival**	**Loss**	**Explore / Exploit**
Baseline	25.62	76	612.57	0.4435	0.5%/99 .5%
Distance Rewards	25.83	71	577.37	0.4473	0.5%/99 .5%
Balanced Rewards	28.6	83	602.88	0.5037	0.6%/99 .4%
Enhanced Architecture	26.22	64	543.93	0.6159	0.6%/99 .4%

FIGURE I. After 500 games (our baseline):

- Game 500 - Score: 12 - Record: 76 - Survival Time: 333 - Loss: 0.5313 - Explore/Exploit: 0.6%/99.4%

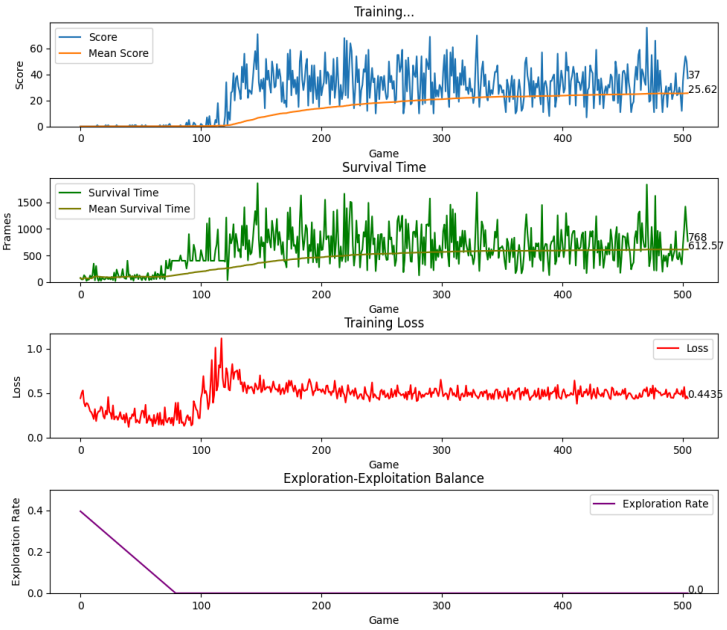


FIGURE II. After 500 games with distance rewards (Added change1):

- Game 500 - Score: 49 - Record: 71 - Survival Time: 1136 - Loss: 0.4473 - Explore/Exploit: 0.5%/99 .5%

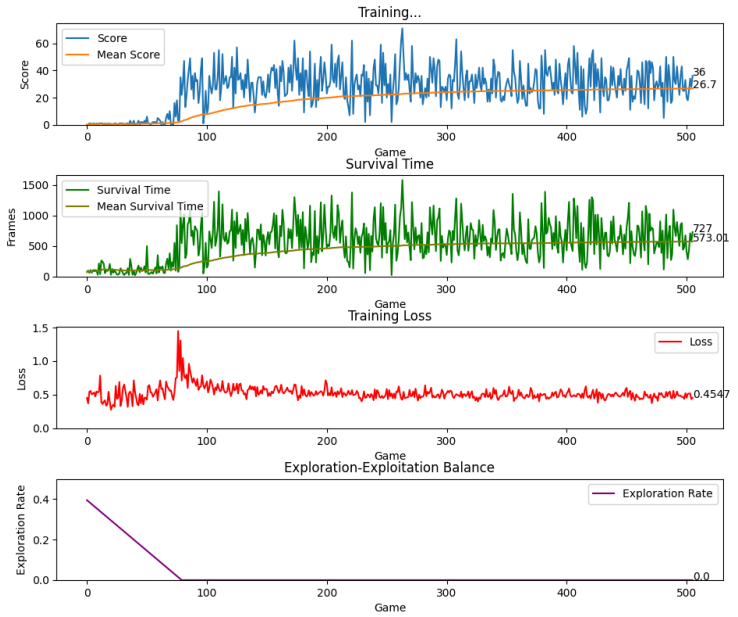


FIGURE III. After 500 games with more balanced rewards: distance, wall avoidance, self-collision + time efficiency penalty (Added change3):

- Game 500 - Score: 24 - Record: 83 - Survival Time: 516 - Loss: 0.5037 - Explore/Exploit: 0.6%/99.4%

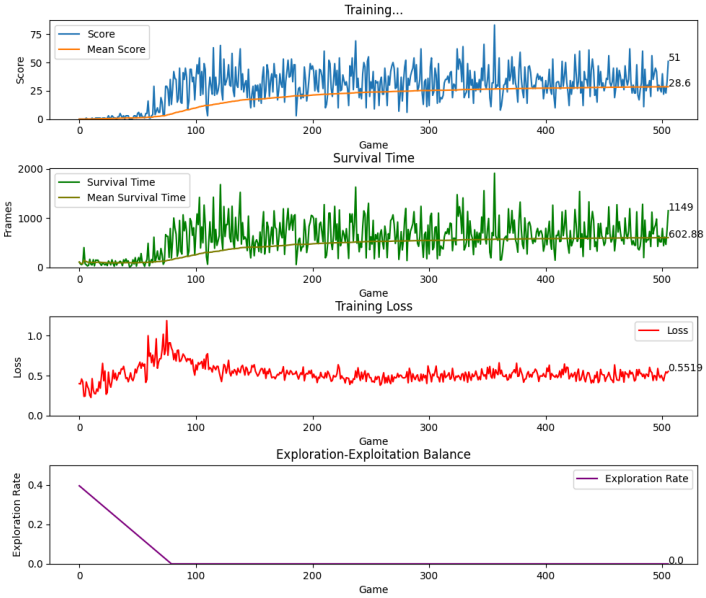
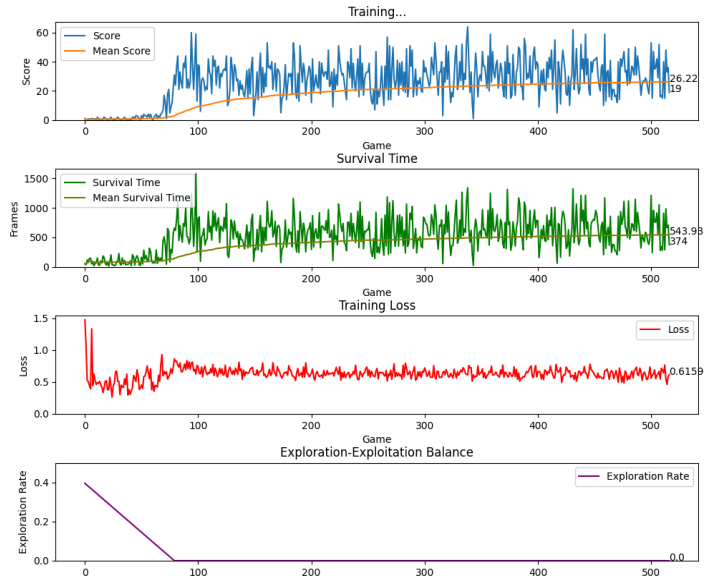


FIGURE IV. After 500 games with distance rewards + wall avoidance + self collision + increased neural network size and weight initialization + flood path algorithm (Added change4):

- Game 500 - Score: 29 - Record: 64 - Survival Time: 525 - Loss: 0.7357 - Explore/Exploit: 0.6%/99.4%
- Increased neural network and implemented behavior to avoid self-collision but made snake more cautious



C. *Analysis*

While studying our initial snake model, we noticed some unusual patterns in its learning behavior at the beginning of training. These patterns included infinitely circling around itself (like a dog chasing its own tail) and staying within close proximity of its own body while ignoring the objective. Both of these patterns were phased out as the agent began to learn.

The original training strategy that we employed had the agent learning basic survival without any strategic growth. This strategy had a mean score of 25.62 and a mean survival time of 612.57. This strategy had some decent stability, however, it also caused the performance to plateau very quickly.

After adding change 1 (adding distance-based rewards), we saw a minimal increase in mean score (+0.21) and a drop in mean survivability (-35). The minimal difference here appears to be much more likely to be the result of random chance rather than an actual substantial difference between the training strategies. We have interpreted this to mean that these distance-based rewards may lead to some inconsistency in training, potentially guiding the agent toward food but at the expense of risky decisions.

Change 2, which implemented wall avoidance and self-collision avoidance as well as the aforementioned distance-based rewards, saw a slightly more significant increase in survival (+1.07), but a large drop in survivability (-39). We have interpreted this to mean that the model has become more assertive, which in some cases leads to more risky behavior (like change 1), which results in a decrease in survivability.

Our third change, which balanced rewards and implemented a time-efficiency penalty, had the highest mean score (28.6) and the second highest mean survivability (602.88). As per our expectations, finding the “sweet spot” that balanced caution and risk was performed the most successfully.

Our last change involved a larger neural network with flood fill and initialization tuning. It performed decently, however, significantly less effectively than the previous iteration. This may be because of some overhead/complexity.

While comparing the model’s performance to a typical human’s (the members of our group), we noticed that the model’s failure modes were very different from a human’s failure modes. Most notably, the most common human failure mode was mechanical error. In other words, our human players often reacted too late and accidentally ran into walls or their own snake body. The model, by comparison, never had this failure mode as it can play the game at its own pace, actively deciding on what it would do for every single state/tick of the game. That being said, the model also often created failure modes that rarely came up within human games. The most common model-specific failure mode is situations where the snake wraps around itself for seemingly no reason and then has no choice but to run into itself. This situation very rarely occurs in human games as the human player typically recognizes this as a poor decision, whereas the model does not have this level of comprehension and inadvertently dooms itself. Despite these very different failure modes, we

experienced very similar mean scores. That being said, the model typically had a much longer lifespan as it often did not quickly focus on the objective of the game (eating the apples).

V. CONCLUSION AND FUTURE WORK

In this research, we employed Deep Q-Learning to develop a reinforcement learning agent capable of playing the classic Snake game. With verifiable performance gains over time, such as longer survival times and higher average scores, our model demonstrated its capacity to learn from the game environment. The agent was able to adapt to the game's increasing complexity by training directly from experience on a dynamically generated dataset. Our model's development was mainly based on core concepts such as exploration-exploitation balancing, state-action-reward modeling, and experience replay.

Despite its excellent outcomes, the agent has several limitations and areas for improvement. Occasionally, the agent's performance plateaued, indicating that there may be problems with the hyperparameter configuration or the current neural network design. Furthermore, even if it is concise and simple to comprehend, the state representation may be insufficient to accurately reflect complex spatial arrangements as the snake grows.

Future work could focus on:

- Using more advanced RL approaches, such as Double DQN or Dueling DQN, can improve stability and learning efficiency.
- Convolutional layers that work directly on the game grid improve the state representation.
- Applying curricular learning by gradually raising the difficulty of the setting (i.e. more obstacles)
- Using multi-agent systems or competitive modes to assess adaptability and strategic behavior.
- Experimenting with transfer learning or training the agent on different grid-based games to investigate generalization.
- The behavior of the agent could be tested and assessed in real-time against human players to compare performance.
- Consider using recurrent neural networks, which allows the agent to remember its past runs and incorporate new strategies that will help it make smarter decisions.

VI. CONTRIBUTIONS OF EACH MEMBER

Jake de los Reyes primarily worked on the final report for this project. His contributions include the abstract, index terms, Section I: Introduction and Research Problem, Section II: Related Work (Literature Review), and Section IV-C: Analysis.

Han Luu designed and implemented the neural network architecture and preprocessing pipeline that formed the core of our learning system. Her work included developing the 11-dimensional state representation that efficiently encoded game states, implementing the Deep Q-Network in Pytorch with appropriate activation functions and layer configurations, and integrating experience replay and target network mechanisms to stabilize the training. She also established the agent-environment interface that connected the neural network to the game, implementing the epsilon-greedy action selection strategy that balanced exploration and exploitation during learning.

Itzel Orozco was a part of documentation by providing clarity on logic and functionality of the reinforcement learning agent, adding comments to help explain further on the code. Generally, provided team support through giving and receiving feedback as the project progressed smoothly.

Matthew Smith developed the core Snake game environment using Python and Pygame, implementing the fundamental mechanics including snake movement, random food generation, and collision detection systems. He established the game parameters (20x20) grid, scoring mechanism, and state transition logic that was critical for the reinforcement learning process. Additionally, he designed and implemented the comprehensive data logging framework that captured essential gameplay metrics at each step, including snake position, food location, movement direction, and performance statistics–pricing the foundation of our agent's training and evaluation.

Jenny Tran's contribution was mostly towards this report in Section III: Methodology and Technical Details, analyzing and providing insight for the logic behind the neural network as well as the formatting of this paper was done by her. She also contributed to documentation by providing additional details and math logic for several methods in the agent and model.

Genesis Anne Villar led the training and evaluation of the reinforcement learning agent, conducting multiple experimental configurations to optimize performance. She implemented adjustments to the performance monitoring framework that added survival time, loss curves, and exploration-exploitation balance. Her contributions to data visualization provided critical insights that guided the refinement of the neural network and reward parameters. Additionally, she conducted a comprehensive peer review of both the codebase and final paper to ensure quality and consistency.

ACKNOWLEDGMENT

The authors of this research report would like to express their gratitude to Professor Xin Zhang for her guidance and valuable feedback throughout the development of this project. Her insightful comments on reinforcement learning approaches significantly contributed to the direction and implementation of our work.

REFERENCES

[1] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.
[2] S. Amin, "Deep Q-Learning (DQN)," *Medium*, 2021. [Online]. Available: https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae (accessed: May 5, 2025).
[3] "Deep Q-Learning in Reinforcement Learning," *GeeksforGeeks*.[Online]. Available: https://www.geeksforgeeks.org/deep-q-learning/ (accessed: May 5, 2025).
[4] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," Neurocomputing, vol. 503, pp. 92–108, Sep. 2022, doi: https://doi.org/10.1016/j.neucom.2022.06.11