REINFORCEMENT LEARNING FOR SNAKE

Q-learning approach to the problem

Chengheng Li Chen
Computer Science and Pure Math
UCLA, University of Barcelona

Taro Iyadomi
Data Theory
UCLA

Lizabeth Annabel Tukiman
Mathematics of Computation
UCLA

Abstract

Reinforcement Learning (RL) has been widely utilized over the years to train agents for complex games such as Go and StarCraft. Inspired by these achievements, particularly AlphaZero, we aim to develop an RL model for the popular game of Snake. This study focuses on applying Q-Learning (QL) and Deep Q-Learning (DQL) to evaluate their performance in this context. Simplifying the state definition by using the relative positions of the snake's head with respect to the apple proves to be a decent approximation for vanilla Q-Learning. However, this state simplification poses challenges for the advanced DQL version, leading to overfitting and suboptimal performance. This paper discusses the methodologies employed, the challenges faced, and the comparative performance outcomes of both techniques.

Code and additional resources: Github Link

1. Introduction

The classic snake game was created in 1976 (Wikipedia, 2023) and consists of controlling a long, thin creature simulating a snake, which moves around a bordered playing area attempting to consume apples that re-spawn in random positions. The snake grows longer each time it eats an apple, and as it grows, it becomes increasingly difficult to avoid colliding with the walls or the snake's own body, which would end the game [1].

This game first appeared in arcade machines, and by 1998 it was released on the iconic Nokia phones (Wikipedia, 2024b), where the player uses the arrow buttons to move the snake on a tiny greenish screen (see Figure 1). In those years, the game rose to popularity because the catalog of the famous Nokia phones was limited to fewer than five games. Moreover, the game's simplicity contributed to its success, as it did not require advanced technology.

Over the years, this simple game has become a challenge for AI researchers. Researchers try to create autonomous agents capable of attaining the maximum score, which has proven to be a difficult task due to the random apple positions on the grid and other stochastic factors. As a result, creating a high-performing Snake AI demands advanced algorithms and problem-solving methods.

Figure 1: Snake in a Nokia phone (Reddit)

Therefore, the aim of this project is to explore the performance of two well-known reinforcement learning algorithms, Q-learning and its advanced version, deep Q-learning, within the environment of the snake game. We aim to compare the performance of these two algorithms on different reward functions and check whether deep Q-learning is better than Q-learning in the domain of this game.

2. Preliminaries

Humans have been learning through a system of punishment and reward since childhood, which is corroborated by several studies in neuroscience and cognitive science (Edita Navratilova and Frank Porreca, 2014). For example, failing an exam often results in a type of punishment, while achieving a good grade typically leads to some kind of reward. This method is not exclusive to humans; it can also be applied to many animals, such as lions and elephants. Unfortunately, we can find that in several cases, animals are frequently forced to perform tasks causing them pain and when they complete the task, they are rewarded with food. Repeating this process several times teaches animals that performing the task well will help them avoid suffering and earn a reward. This proves that the punishment and reward system is highly effective.

Similarly, we can apply this system to train our AI agent to learn how to behave in an es-

[1] The game can be played here https://g.co/kgs/Hw7Psby

tablished environment by punishing it if it performs an incorrect action and rewarding it if it gets closer to the goal. This model has been proven very useful in several fields where we can give direct feedback to the agent concerning the action it performed, such as in video games or aligning large language models (LLMs) using the reinforcement learning from human feedback (RLHF) technique.

2.1. Reinforcement learning

Reinforcement learning is a machine learning technique inspired by the system of punishment and reward. It involves allowing an AI agent to explore the environment freely and providing feedback on its actions.

Formally, we can define the model as an agent that can explore an environment represented by a Markov Decision Process (MDP). An MDP can be defined as a 5-tuple $(\mathcal{S}, s_0, \mathcal{A}, R, P)$ where:

- $\mathcal{S}$ is the set of all valid states.
- $s_0 \in \mathcal{S}$ is the initial state of the agent.
- $\mathcal{A}$ is the set of all valid actions.
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ represents the reward that the agent receives at state s by performing action a and ending at state s', denoted by $R(s, a, s')$.
- $P : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ is the transition function that provides the set of all possible next states (s') for a given state s and action a. Then we have that $P(s' \mid s, a)$ is the probability of transitioning into state s' if the agent starts at state s and take action a.

Our agent interacts with an environment to achieve a specific goal. The agent starts at an initial state s_0 and selects an action a. The environment responds by transitioning to a new state s' with a probability $P(s' \mid s, a)$. The agent then receives a reward $R(s, a, s')$ based on the action taken and the resulting state (see Figure 2). This process continues iteratively, with the agent selecting actions, transitioning between states, and receiving rewards, until it either reaches a goal state or a predefined depth limit, at which point the training episode ends and a new one begins.

The objective of the agent is to learn a policy $(\pi : \mathcal{S} \to \mathcal{A})$ that maximizes the cumulative reward over time. This involves exploring or exploiting (see subsection 2.4) different actions and learning from the outcomes to make better decisions in the future [2].

[2] More information here: OpenAI Spinning Up

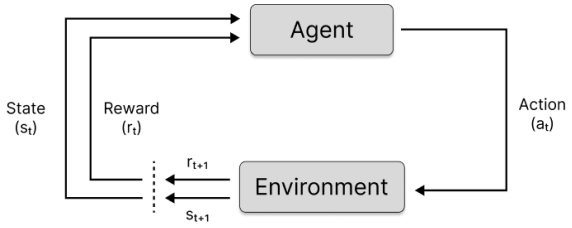

Figure 2: An iteration of reinforcement learning

2.2. Q-Learning

Q-learning is a model-free reinforcement learning algorithm that aims to learn the optimal action-selection policy for an agent interacting with an environment.

Given an MDP, the Q-value for a state-action pair (s, a) is updated using the Temporal Difference (TD) error, controlled by the learning rate α. The TD error is the difference between the target value and the current Q-value. The target value typically includes the reward received after taking action a in state s and the maximum Q-value of the next state s' (see Figure 3).

$$Q(s,a) \leftarrow Q(s,a) + \underbrace{\alpha[\underbrace{r + \gamma \max_{a'} Q(s',a')}_{\text{Target value}} - Q(s,a)]}_{\text{Temporal difference (TD)}}$$

$Q(s, a)$ is the **Q-value**, representing the expected cumulative reward of taking action a in state s; $\alpha \in [0, 1]$ is the **learning rate**, that determines how much new information overrides old information; $r = R(s, a, s')$ is the **immediate reward** received after taking action a in state s; $\gamma \in [0, 1]$ is the **discount factor**, that quantifies the importance of future rewards; $\max_{a'} Q(s', a')$ is the maximum Q-value over all possible actions a' in the next state s'.

Figure 3: Q-learning update rule

The core of the Q-learning algorithm is the aforementioned update rule based on the Bellman equation, which iteratively refines the estimated value of taking a specific action in a given state based on the received rewards and the estimated future rewards, converging to the optimal action-value function (Watkins, Christopher J. C. H., 1992).

Then the policy of our agent given a determined state $s \in \mathcal{S}$ will be:

$$\pi(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s, a)$$

In Q-learning, Q-values are stored in a table of size $|\mathcal{S}| \times |\mathcal{A}|$ (see Figure 4). In other words, the size of the table is proportional to the number of states, which makes this algorithm unsuitable for

tackling complex problems with a vast number of states. For example, in chess, with an estimated number of 10^{120} different positions (Wikipedia, 2024a), creating a Q-table for all states is infeasible due to current memory limitations.

2.3. Deep Q-Learning

Deep Q-learning, an advanced version of Q-learning, uses neural networks instead of a Q-table to approximate the Q-values for state-action pairs. These neural networks approximate unvisited states with already visited states, reducing the number of states we need to explicitly visit to achieve an optimal policy. Hence, it is a better approach for more complex problems.

The neural network architecture consists of an input layer, one or more hidden layers, and an output layer. The input layer receives the state information as an input. This input then propagates through the hidden layers, with each hidden layer containing neurons that apply non-linear transformations to the data. Finally, the output layer is sized to match the action space, with each neuron representing the Q-value of a specific action (see Figure 4).

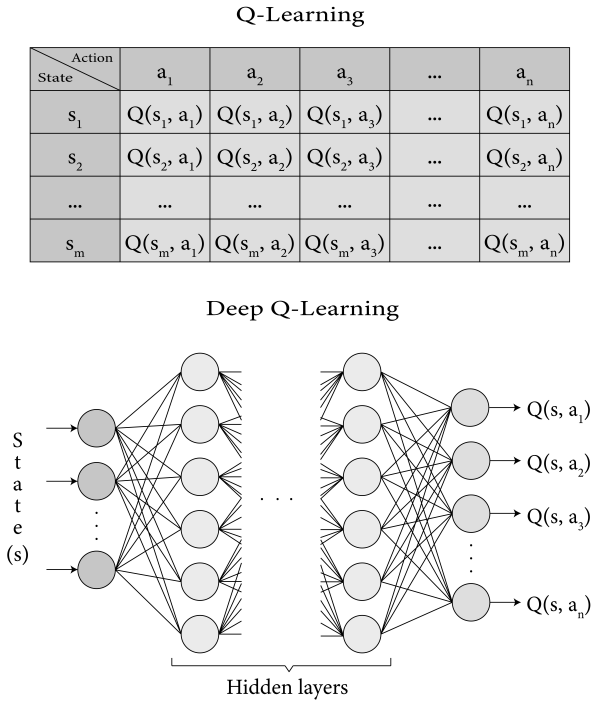

Figure 4: Q-Learning vs Deep Q-learning

Then, the loss function minimizes the difference between the predicted Q-values and the target Q-values, which are computed using the Bellman equation:

$$Q(s,a) \leftarrow r + \gamma \max_{a'} Q(s',a')$$

Using a neural network to compute Q-values allows us to efficiently handle high-dimensional state spaces and large action spaces. This approach maintains the simplicity of accessing the policy, as it remains consistent with selecting the action with the highest Q-value. However, instead of selecting the maximum value from a row in a table, we select it from the output of a neural network.

2.4. Exploration vs Exploitation

The balance between exploration and exploitation is a critical aspect of making good decisions in reinforcement learning. Exploration is essentially trying new actions to see what rewards they have attached to them, while exploitation is the process of picking actions that the agent already knows will have high rewards from prior experience.

One of the most popular methods to deal with this trade-off is the ε-greedy approach. With a probability ε, the agent will take any random action and with the remaining probability $1-\varepsilon$, the agent will take an action associated with a higher reward based on estimate. ε becomes smaller over time as the agent learns more about the environment, moving from exploration to exploitation.

This adaptive behavior allows the agent to initially explore a wide range of actions in an attempt to learn more about the environment until it learns enough, at which point it slowly starts to trade off exploration with exploitation to maximize the cumulative rewards.

3. Problem formalization

For the snake game, we can see that the agent moves in a 2D world that can be defined as a grid of size $W \times H$, where W is the grid's width and H is the grid's height. Moreover, our agent can only move forward, right, or left. Then we can define the MDP as follows.

3.1. State Space ($\mathcal{S}$)

Our agent will be able to move in a 2D world (see Figure 5), and in each cell of the grid, the agent could be facing a determined direction. Therefore, an intuitive definition of a state is given by a 3-tuple of w, the position of the snake on the x-axis, h, the position of the snake on the y-axis, and d, the direction in which the snake is facing. Additionally, we will add the final state "Game Over", which is the state to transition to when the snake collides and dies. Formally, our set of states will be:

$$\mathcal{S}_1 = \{(w,h,d) \mid 1 \le w \le W \textbf{ and } 1 \le h \le H \textbf{ and } d \in \{\text{Up}, \text{Down}, \text{Right}, \text{Left}\}\} \cup \{\text{Game Over}\}$$

However, this definition of states does not work in a practical world since it yields a huge number of states of the order $W \times H \times 4$, making it quite difficult to visit all the states and train the agent for every situation. Moreover, these states do not track the position of the apple. Hence, if we were to add the position of the apple to the tuple, the number of states grows significantly as the grid gets bigger.

A better practice is to define a state as the relative position of the snake's head with respect to dangers on the grid and the apple. To do so, we will keep track of 6 parameters as Booleans:

- d_f: if there's danger directly at the front.
- d_r: if there's danger directly on the right.
- d_l: if there's danger directly on the left.
- f_f: if the food is to the front.
- f_r: if the food is to the right.
- f_l: if the food is to the left.

Here, "danger" refers to a wall or part of the snake's body.

Formally, we have that

$$\mathcal{S}_2 = \{(d_f, d_r, d_l, f_f, f_r, f_l) \mid d_f, d_r, d_l, f_f, f_r, f_l \in \{0, 1\}\} \cup \{\text{Game Over}\}$$

For example, in Figure 5, the state is $(0, 0, 0, 1, 1, 0)$. Therefore, we can observe that the set of states is independent of the grid size, making this definition more scalable.

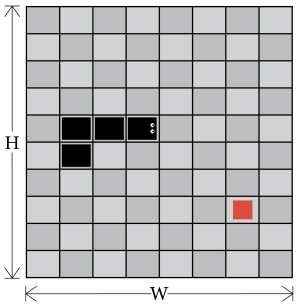

Figure 5: Snake 2D World

Symmetry of 4 directions: We claim that our definition of states in $\mathcal{S}_2$ is independent of the direction of the snake.

Proof. Consider the state representation $(d_f, d_r, d_l, f_f, f_r, f_l)$. We need to show that this representation is invariant under the four possible orientations of the snake's head: up, down, left, and right.

Let the snake's head initially face up. The parameters are defined as follows:

- d_f: danger in the cell directly above.
- d_r: danger in the cell directly to the right.
- d_l: danger in the cell directly to the left.
- f_f: food in a cell above.
- f_r: food in a cell to the right.
- f_l: food in a cell to the left.

Now, consider the snake's head facing right (a 90-degree clockwise rotation). The new parameters are:

- d_f: danger in the cell directly to the right.
- d_r: danger in the cell directly below.
- d_l: danger in the cell directly above.
- f_f: food in a cell to the right.
- f_r: food in a cell below.
- f_l: food in a cell above.

Applying 90-degree rotations reiteratively like in Figure 6, the relative positions of danger and food with respect to the snake's head are consistently defined in terms of front, right and left. Therefore, the state $(d_f, d_r, d_l, f_f, f_r, f_l)$ remains invariant under 90 degrees rotations, proving that our definition of states in $\mathcal{S}_2$ is directionally invariant. □

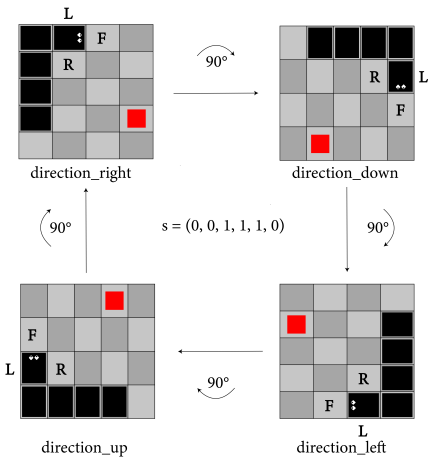

Figure 6: Invariant states via rotation

3.2. Initial state (s_0)

The initial state of the agent will vary each game since the generation of the apple on the grid is random. However, from the definition of

states $\mathcal{S}_2$, we can see that the agent's initial position does not bias the training process, since the agent learns to make decisions based on the relative conditions it encounters, which are consistent regardless of its initial position on the grid.

3.3. Action Space ($\mathcal{A}$)

Since the snake is a very simple game, the only actions that the agent can take in from a determined state will be going to the right, going to the left, or continuing forward (see Figure 7). Therefore, our action space has the following elements:

$$\mathcal{A} = \{\text{right},\ \text{left},\ \text{forward}\}$$

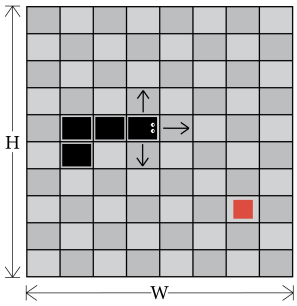

Figure 7: Actions that our agent can take

3.4. Reward function (R)

The reward function is the main challenge of this problem, where we have to find an optimally aligned reward. We will start with a naive version and then implement more complex reward functions. Hence, we will dedicate an entire section to discussing it. See section 5.

3.5. Transition function (P)

The transition function in the Snake game can be easily determined when the snake is moving towards an apple, as the game is deterministic in these moments. Given a specific state s and action a, you can predict the next state s' exactly.

However, when the snake eats the apple, the apple respawns in a random location on the grid. At this point, the next state becomes stochastic. The new state will be equally probable across all possible states that maintain the danger cells d_f, d_r, and d_l.

4. Problem implementation

4.1. Environment setup

For the environment setup, we got the Snake Game code from an open-source GitHub repository[3]. This repository included the core parts of the snake game, including its playability.

[3] https://github.com/patrickloeber/python-fun/blob/master/snake-pygame/snake_game.py

We added several functions to implement the reinforcement learning algorithms, such as the function to get the current state of the game and other types of getters.

4.2. Q-learning

The Q-learning agent implementation can be found in *QLearning.py*. To implement this agent, we used dictionaries to represent the Q-tables, allowing us to access the Q-value for each state-action pair as follows: `Q_table[<state>][<action>]`.

4.3. Deep Q-learning

The Deep Q-learning agent implementation can be found in *DeepQLearning.py*.

One of the main challenges of using neural networks for Q-learning is the instability caused by shifting Q-values. Q-values are updated using the Bellman equation, which relies on the current network's predictions. If we use these predictions to update the network itself, it creates a feedback loop where the network chases a moving target. This can lead to divergence or oscillations in the Q-values.

Deep Q-Networks (DQN) introduce a target network (`target_net`) to address this issue. The target network is a duplicate of the main Q-network (`policy_net`), but its weights are updated less frequently, typically every few thousand steps. This provides a more stable target for updating the Q-values, reducing oscillations and helping the Q-values to converge more reliably.

In addition to the target network, we employ a soft update mechanism to further enhance stability. Soft updates incrementally adjust the target network's weight towards the main network's weights using a parameter τ, instead of updating the target network's weights all at once. The update rule is given by:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

where θ is the weights of the main Q-network and θ' is the weights of the target network. This approach ensures that the target values change smoothly, reducing abrupt changes and further stabilizing the training process.

One more problem in reinforcement learning is the high correlation of experiences, mostly in sequential tasks. Training a neural network on such correlated data can result in inefficient learning and poor generalization. Experience replay is a mechanism that saves the agent's experiences: state, action, reward, and next state. It samples the mini-batches of experiences during training

from this buffer randomly for updating the Q-network. Random selection breaks the link between successive experiences, resulting in more consistent and efficient learning.

This allows the agent to reuse experiences many times for greater data efficiency and helps the network learn from experiences that seldom occur but are important. Because experiences are sampled at random, the network is less dependent on previous experiences, which helps avoid overfitting the network to recent experiences and makes sure that the network works well for different states and actions.

5. Reward function

5.1. Naive reward function

The naive reward function involves rewarding 1 point to the agent when it eats an apple and penalizing 1 point from the agent when it ends the game either by colliding to a wall or to itself.

Game over	Eat apple	Other
-1	+1	0

Table 1: Naive reward table

5.2. Advanced naive reward function

Similar to the naive reward function, the advanced naive reward function involves rewarding the agent when it eats an apple and penalizing it when it ends the game. The difference is in the magnitude of the reward and penalty given the direction of the agent.

	Game over	Eat apple	Other
F	-2	+2	0
L/R	-1	+1	0

F: Forward — L: Left — R: Right

Table 2: Advanced naive reward table

5.3. Manhattan distance to the apple

The Manhattan distance reward function retains the reward and punishment system for eating an apple and losing the game from the naive reward function and adds a Manhattan distance computation between the head of the snake and the apple before and after each turn. If the agent doesn't reach an apple or end the game, it's rewarded if the Manhattan distance to the apple is less than its previous state and punished otherwise.

Game over	Eat apple	Closer	Other
-100	+30	+1	-5

Table 3: Manhattan distance reward table

6. Training

Agents were trained in a 10×10 grid environment, and their performance was subsequently evaluated in larger grid environments, as state representations are independent of grid size (see Figure 10). Firstly, the optimal number of training episodes (games) required for effective agent training was determined. Subsequently, multiple agents were trained using the established episode count across various reward functions.

6.1. Episode Count Determination

The primary objective of this phase was to ascertain the appropriate number of episodes necessary for the training process. The advanced naive reward function was employed and hyperparameters were kept constant throughout the experiments. A range of episode counts was selected, from which several reasonable values were chosen. For each selected value, 10 different agents were trained to assess performance. For Q-learning agents, episode counts varied between 10 and 50,000 (see Figure 8), while for deep Q-learning agents, episode counts ranged from 5 to 1,000 (see Figure 9).

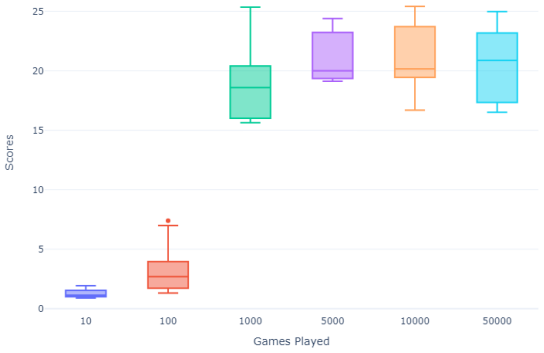

Figure 8: Scores vs. Episode Count (Q-learning)

Upon detailed analysis of the figures, it was determined that an episode count of 5000 is appropriate for Q-learning, as performance improvements were observed to plateau beyond this threshold. For deep Q-learning, an episode count of 75 was selected based on similar observations.

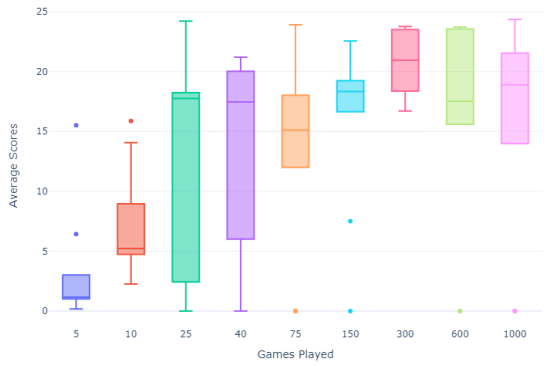

Figure 9: Scores vs. Episode Count (Deep Q-learning)

This selection also took into account the trade-off between the training time required and the performance of the agent.

6.2. Hyperparameter tuning

For the deep Q-learning agents, a random search was conducted to determine the optimal values for each hyperparameter, utilizing the advanced naive reward function. This approach was not applied to Q-learning, as the default hyperparameters were found to be quite stable and yielded satisfactory results. During this process, 100 agents were trained with parameters selected randomly from either a uniform or log-uniform distribution, with incremental steps of a determined size, as detailed in Table 4.

Param.	Type	Start	End	Step
α	LogU	10^{-5}	10^{-1}	N/A
γ	Unif	0.8	0.99	0.01
ε start	Unif	0.5	1.0	0.1
ε end	Unif	0.01	0.1	0.01
ε decay	Unif	0.990	0.999	0.001
τ	Unif	0.001	0.01	0.001

LogU: Log-Uniform — Unif: Uniform

Table 4: Distributions from which each hyperparameter was selected

After completing the process, we kept the hyperparameters that yielded the best results, which are detailed in Table 5.

6.3. Training Top-Performing Agents

After gathering all parameters information, 100 agents were trained for each model-reward combination using the selected hyperparameters and episodes. For each agent trained with a specific reward function and model, 1,000 games were played, and the average score across all games was computed. The agent with the highest average score for each reward function was selected. This process resulted in six best agents: three from Q-learning and three from deep Q-learning, with agents in each group trained using different reward functions.

Parameter	Optimal values
α	0.000125
γ	0.96
ε start	0.5
ε end	0.0999
ε decay	0.991
τ	0.006

Table 5: Optimal values of each hyperparameter

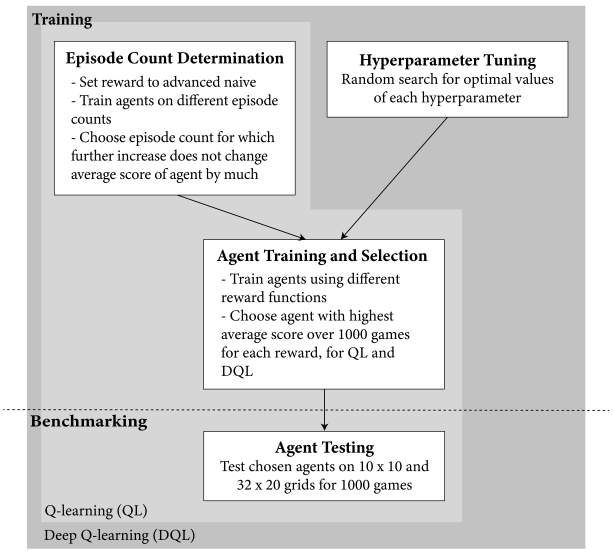

Figure 10: Summary of training and benchmarking process

7. Benchmarking

To evaluate the performance and generalization ability of the trained agents, experiments were conducted on grids of different sizes. This approach helps to understand which aspects of the environment the agents learned during the training process. The agents were tested on the training grid of size 10×10 and on a more commonly used grid of size 32×20.

7.1. Grid 10 x 10

Using the top-performing agents from Section 6.3, we benchmarked each of them on a 10×10

grid for 1000 games. The results of each reward function for Q-learning and deep Q-learning are shown in Table 6 and Table 7, respectively.

Reward	Avg score	Max score
Naive	30.65	50
Advanced naive	31.27	51
Manhattan dist.	28.59	45

Table 6: Scores across reward functions for Q-learning on a 10x10 grid

Reward	Avg score	Max score
Naive	24.06	50
Advanced naive	23.83	53
Manhattan dist.	**34.73**	**57**

Table 7: Scores across reward functions for deep Q-learning on a 10x10 grid

Notably, the deep Q-learning agent trained using the Manhattan distance reward function exhibited superior performance, achieving both the highest average score and the highest individual score across 1,000 games. However, Q-learning agents utilizing other reward functions consistently outperformed their deep Q-learning counterparts on average. This suggests that neural networks can extract information from the Manhattan distance, enhancing the deep Q-learning approach.

7.2. Grid 32 x 20

We repeated the experiments of the previous section on an expanded grid (32x20) to test their generalizability. Unlike our previous results, all three Q-learning agents, including the one using the Manhattan distance reward, outperformed their deep Q-learning counterparts in terms of average scores as seen in Table 8 and Table 9. Although the best performing agent on the training grid, the deep Q-learning agent using the Manhattan distance reward, had a higher top score than its Q-learning equivalent, its average score was the lowest out of all six agents tested. This suggests that deep Q-learning fails to generalize, which we will discuss in the conclusion (see section 8).

8. Conclusion

One of the main advantages of our model's state definition is its grid size independence, allowing us to tune and train the agent in small grid sizes and use it in larger grid environments. However, our state representation is very simple, limiting the agent's capacity to learn. For instance,

Reward	Avg score	Max score
Naive	82.41	134
Advanced naive	**86.56**	**136**
Manhattan dist.	72.08	122

Table 8: Scores across reward functions for Q-learning on a 32x20 grid

Reward	Avg score	Max score
Naive	61.04	121
Advanced naive	60.15	114
Manhattan dist.	54.02	131

Table 9: Scores across reward functions for deep Q-learning on a 32x20 grid

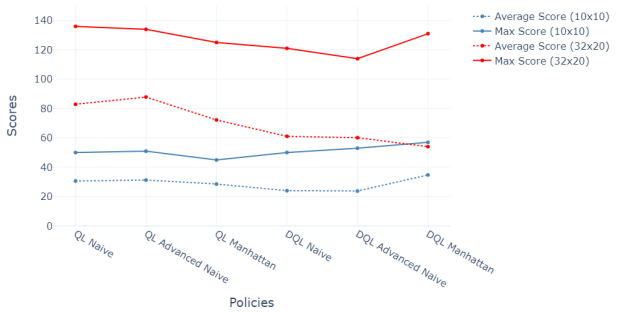

Figure 11: Scores Comparison Across Reward Functions and Grid Sizes

the Manhattan distance does not improve performance since the agent could be getting closer or farther from the apple while staying in the same state.

After benchmarking, we noticed that in small state spaces, DQL often performs poorly or has more difficulty generalizing compared to traditional QL due to several factors:

- **Model Complexity**: The neural network in DQL introduces significant complexity, leading to overfitting and the capturing of noise instead of underlying patterns.
- **Sample Efficiency**: QL updates Q-values directly for each state-action pair, making it more sample-efficient, whereas DQL requires more samples to train the neural network effectively.
- **Stability and Convergence**: DQL requires careful hyperparameter tuning for stability and convergence, while QL's tabular method is more straightforward and stable.

- **Computational Overhead**: Training the neural network in DQL introduces unnecessary computational overhead, slowing down the learning process in small state spaces.

Consequently, the increased complexity, potential of overfitting, and computational overhead of DQL lead to its poorer performance compared to the simpler and more direct QL approach in small state spaces.

However, we can see that in the 10×10 training grid, the best policy is given by the DQL agent using the Manhattan distance. Our hypothesis is that the Manhattan distance reward function performs well for DQL in the 10×10 training grid because the neural networks can extract implicit information such as the size or shape of the grid from the reward. This may explain why the policy does not generalize well to other grid sizes.

9. Future work

After this project, the most natural question is how we can improve the model to score more points. One natural approach would be defining more complex states that could grow with respect to the grid size, such as $\mathcal{S}_1$, and trying to use DQL to see if the performance increases. Introducing the concept of the absolute position of the snake makes the definition of the distance to the apple more meaningful.

Moreover, observations of the agent playing the game suggest that guiding the agent directly towards the reward is not necessarily the best option. When the snake length is long, the agent may take a longer path to avoid collisions. Therefore, we may introduce a more complex type of reward function.

One suggestion is the Euclidean distance reward function. It is similar to the Manhattan distance reward function but uses a Euclidean distance computation instead of a Manhattan one. It incorporates the difference between the distances in the current and next states in a different manner. If the agent does not reach an apple or end the game, it is rewarded or penalized by the logarithm of the ratio of the sum of the length of the snake (L_t) and the distance between the snake's head and the apple in the current state (D_t) to that in the next state (D_{t+1}), which is positive when $D_t > D_{t+1}$ and negative otherwise. By considering the length of the snake, this function allows for more tolerance when the snake is longer (Wei, Z., Wang, D., Zhang, M., Tan, A.-H., Miao, C., Zhou, Y., 2018).

Game over	Eat apple	Other
-1	+1	$\log_{L_t}\left(\frac{L_t+D_t}{L_t+D_{t+1}}\right)$

Table 10: Euclidean distance reward table

Another common approach to improve the model would be using other reinforcement learning models such as Proximal Policy Optimization (PPO), which deals with constraints better and can potentially yield higher performance improvements.

Contributions

The project was a collaborative effort, with the entire team participating in reviewing the project and providing feedback on the completed work. However, the primary contributions of each team member can be highlighted as follows:

- **Chengheng Li Chen**: Provided theoretical knowledge of reinforcement learning, formalized the problem, and implemented the models in Python.
- **Taro Iyadomi**: Conducted benchmarking and training execution, and developed visualizations tools.
- **Lizabeth Annabel Tukiman**: Researched and defined the reward functions, and designed most graphical content.

Appendix

Training Process Details

The source code that implements the training process for the models, including episode count determination, hyperparameter tuning, and agent training, can be found in `src/training.ipynb` in the GitHub repository. The best-performing agents have been saved in `src/policies`, which can be loaded and benchmarked using `src/benchmark.ipynb`. Remember to set up the environment by following the instructions in `Readme.md` to install all the dependencies.

Hardware Specifications

The models were trained using the following hardware:

- **Processor**: AMD Ryzen 7 3700X, 8-Core
- **RAM Memory**: 16 GB DDR4 RAM
- **GPU Card**: NVIDIA GeForce RTX 3060

Challenge Our Best Agent

Anyone can play against our best agent on the 32×20 grid. To do so, clone the repository from `https://github.com/ChenghengLi/RL_Snake` and execute `play_snake.py`. Follow the instructions in `Readme.md` to set up the environment.

References

Edita Navratilova and Frank Porreca. 2014. Reward and motivation in pain and pain relief.

Watkins, Christopher J. C. H. 1992. Q-learning. Mach Learn 8, 279–292 (1992).

Wei, Z., Wang, D., Zhang, M., Tan, A.-H., Miao, C., Zhou, Y. 2018. Autonomous agents in snake game via deep reinforcement learning.

Wikipedia. 2023. Snake (video game). [Online; accessed 17-May-2024].

Wikipedia. 2024a. Shannon number. [Online; accessed 30-May-2024].

Wikipedia. 2024b. Snake (1998 video game). [Online; accessed 17-May-2024].