The Design and Implementation of the Greedy Snake Fights AI

Ting Lu 2020533132 luting1@shanghaitech.edu.cn
Xinhe Chen 2021533093 chenxh1@shanghaitech.edu.cn
Yixuan Li 2021533140 liyx3@shanghaitech.edu.cn
Lijinxuan Xia 2021533126 xialjx@shanghaitech.edu.cn
Wenting Liu 2021533058 liuwt@shanghaitech.edu.cn

Abstract

The Greedy Snake Fights is an online multiplayer combat game. In this project, we use random, minimax, feature-based Q-learning and neural network methods to implement artificial intelligence agents respectively, and connect them to games for simulation experiments. Through simulation experiments, we compare the relative performance of these AI agents. Our code is available at code sources.

Contents

1. Introduction

1.1. Motivation

The game is a good representation of a multiplayer battle game, where the roles of different players are exactly the same. We can make good use of this feature to connect different AI agents to the game and compete on the same stage. This can be a good intuitive display of the performance of different AI agents.

1.2. The basic framework of the game

The original game that inspired us, *the Greedy Snake Fights*, is shown in Figure 1.

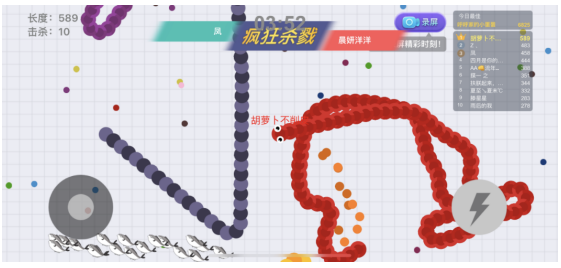

Figure 1: Multiplayer online game:*the Greedy Snake Fights*

The player plays the game by controlling the direction of a snake. Players need to get food while avoiding bumping into other snakes' bodies and borders.

We have made some changes from that base. To simplify the task, we pixelated the game. All tasks will be performed on the grid. Each snake can only move in 3 directions instead of 360 degrees in any direction. The simplified game scene is shown in Figure 2, in which the yellow dots represent food.

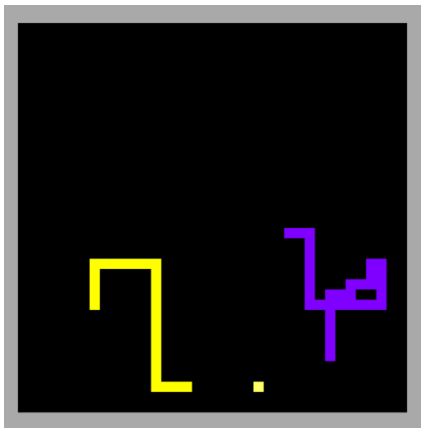

Figure 2: A simplified game where two snakes are fighting for food

2. Methods and preliminary experiments

2.1. *Basic agent*

2.1.1. *Random agent*

We designed the random agent using the method of random return action, which can be used as an obstacle in our testing and training.

2.1.2. *Directional agent*

We set up a simple rater to keep the agent close to the food and avoid going too far away from the border and bumping into other snakes' bodies. The snake appears to be able to compete nimbly for food while avoiding obstacles as much as possible. The snake does not actively attack other snakes. We use it as the most common test opponent.

2.1.3. *Directional combined with random agent*

The agent introduces randomness based on 2.1.2 and is specially used for minimax testing.

Algorithm 1 Directional with Random Agent

$dir \leftarrow$ directionalAgent(*state parameters*)
$\epsilon \leftarrow 0.1$
$actions \leftarrow [(1,0),(-1,0),(0,1),(0,-1)]$
if rand() $< \epsilon$ **then**
    $valid\_dirs \leftarrow$ non-opposite directions to dir
    $dir \leftarrow$ random choice from $valid\_dirs$
else
    $dir \leftarrow$ directionalAgent(*state parameters*)
end if
return dir

2.2. *Minimax agent*

Minimax algorithm is a widely-used game tree search algorithm for solving turn-based two-player games where both players act alternately. The goal of the algorithm is to find an optimal action for the current player that maximizes his own score, assuming the opponent also takes optimal countermeasures.

2.2.1. *Basic idea*

Minimax algorithm constructs a game tree recursively, where the root node is a *MAX* layer, representing the turn of the current player. For the remaining layers, It alternates *MIN* and *MAX* layer, simulating the actions of the player and opponent. Each edge in the tree represents a legal action in the current state of a player, and each node represents a new state reached after an action is taken from the previous state. When recursing to a leaf node, we use an evaluation function to evaluate the current game state. The evaluation function takes into account the distance to food, the distance to the nearest snake head and snake body, and the distance to the boundary. A higher score indicates more favourable to the current player.

After obtaining the evaluation scores of the leaf nodes, we start backtracking in reverse. At *MAX* layer, we select the action with the largest score among the child node and at *MIN* layer, we select the action with the smallest score. We alternate until we get to the root node.We finally choose the action with the highest score in the root node as the optimal strategy for the current player.

Algorithm 2 Minimax Algorithm For Greedy Snake

Data: Game state *state*, Current depth *depth*, Current snake index *currentSnake*
Initialize depth limit=*depthLimit*, Current depth $depth = 0$
procedure MINIMAX(*state*, *depth*, *currentSnake*)
    if *depth* = *depthLimit* or *state* is a terminal state **then**
        return EVALUATE(*state*)
    end if
    if *currentSnake* = minimaxSnake **then**
        $maxValue \leftarrow -\infty$
        for *childState* **do**
            $value \leftarrow$ Minimax(*childState*, *depth*, *index of first opponent snake*)
            $maxValue \leftarrow \max(maxValue, value)$
        end for
        return *maxValue*
    else
        $minValue \leftarrow \infty$
        for *childState* **do**
            if *currentSnake* = the last opponent snake **then**
                $value \leftarrow$ Minimax(*childState*, $depth + 1$,*index of minimaxSnake*)
            else
                $value \leftarrow$ Minimax(*childState*, *depth*, *index of first opponent snake*)
            end if
            $minValue \leftarrow \min(minValue, value)$
        end for
        return *minValue*
    end if
end procedure

2.2.2. *Experiments and findings*

In order to test the performance of our minimax Agent, We test it in the following three environments, each with search depth 1, 2, 3:

- Compete against two *randomAgent* Snake.
- Compete against one *directionalAgent* Snake.
- Compete against one *directional_random_Agent* Snake.

After 30 rounds of competitions in each environment with three depth respectively, we record the winning rate and average score (The score is the number of food and opponent snakes eaten by the player snake) of our minimax Agent , as shown in Table1 below:

	depth=1	depth=2	depth=3
Random	100% 2.7	96.7% 3.9	100% 2.6
Directional	100% 6.9	100% 5.3	100% 3.6
Directional_Random	96.7% 6.6	93.3% 6.8	96.7% 3.2

Table 1: Winning rate and average score results of minimax Agent in three test environments

From Table1 we can learn that:

- **Minimax has extraordinary performance**
  One outstanding advantage of minimax algorithm is that it can predict actions and states of the opponent in the next few steps, therefore improves its effect of obstacle avoidance. It prefers to avoid walls and opponents in order to get higher scores, which is the winning target of the greedySnake game. So it performs well.
- **Opponent's strategy affects minimax' performance**
  For each search depth, the winning rate of *minimaxAgent* when competing with *directionalAgent* is always higher than when competing with *directional_random_Agent*. That's because minimax algorithm assumes that all the opponents takes optimal strategy and act optimally. For *directional_random_Agent*, it is less likely to act optimally because it has randomness ϵ. Therefore, we can learn that the opponent's strategy effects on minimax, the closer it is to optimal, the better performance minimax will show.
- **Search depth affects minimax' performance**
  In our experiment, the runtime differs obviously among differnt depths. Larger depth always takes much more time and the running speed is much slower. That's because as search depth increasing, the search space grows exponentially, leading to a very high time complexity.

2.2.3. *Optimization*

In order to reduce time complexity of minimax algorithm, we apply $\alpha - \beta$ pruning method to it. It prunes away branches that doesn't influence the final decision and returns the same action as minimax does. After applying $\alpha - \beta$ pruning method, the runtime of minimax algorithm gets significantly reduced in our experiments.

2.3. *Feature-base Q-learning agent*

2.3.1. *feature design*

To enable snakes to automatically find food, avoid borders, and avoid other snakes, we select three eigenvalues, which are set as follows:

$$f_1 = \frac{1}{manhattan\ distance(\text{food, snake head}) + 1} \tag{1}$$

$$\text{distob} = \min(x, y, \text{width} - x, \text{height} - y)$$

$$f_2 = \begin{cases} -5 & \text{distob} = 0 \\ -\frac{1}{\text{mindistance}+1} & \text{otherwise} \end{cases} \tag{2}$$

$$\text{distos} = \min_{\text{body} \in \text{other snake body}} (\text{distance(body, snake head)})$$

$$f_3 = \begin{cases} -5 & \text{distos} = 0 \\ -\frac{1}{\text{mindistance}+1} & \text{otherwise} \end{cases} \tag{3}$$

It is worth noting that we have tried to add a fourth feature, namely the minimum distance between the other snake's head and the second half of the snake to reflect the ability to kill other snakes. However, this method does not converge, which is attributed to the fact that the simple linear combination cannot adequately represent the over-complex task.
In addition, our eigenvalues are generally taken between -5 and 1 to avoid convergence difficulties when updating.

2.3.2. *Training process*

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a) \tag{4}$$

$$w_i \leftarrow w_i + \text{learning rate} \cdot \text{difference} \cdot f_i(s, a) \tag{5}$$

Since we don't need to end the game quickly, we take gamma=1. We use 40×40 grid, add a random agent and a directional agent as the training environment. The learning rate is 0.01. The weight changes in the training process are as follows, in which our model converges steadily after 1000 rounds.

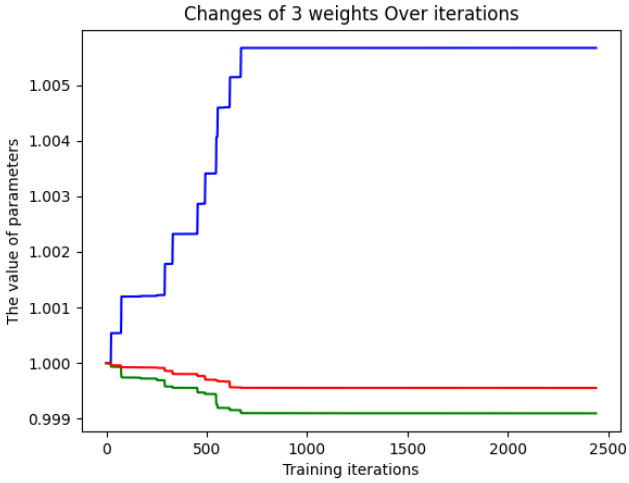

Figure 3: Changes in w1,w2,w3 during training

2.3.3. *Experiments and findings*

In order to test the performance of our Q-learning Agent, We let it compete with Random Agent, Directional Agent, Directional Random Agent, and record their winning rate and final scores respectively. The result is shown in Table 2 below:

	Q-learningAgent
RandomAgents	96.7% 2.1
DirectionalAgent	100% 8.9
Directional_Random_Agent	96.7% 5.2

Table 2: winning rate and average score reult of Q-learning Agent in three test environments

Unexpected findings To our surprise, unexpected strategies emerged in the training process of Q-learning Agent. Sometimes it can gather itself together to defence against other snakes' attack. Here's an example video, the link is as follows: Q-learning self-protect video link.

2.4. *Neural Network agent*

During the gradual training and learning evolution, we gradually evolve the intelligences using neural networks from the initial clumsy state to entities with more intelligent behaviors by continuously improving the state, network, learning rate and rewards (from REWARD rewards for eating food and protecting themselves to eating other snakes).

2.4.1. *Induced Reward Design*

In the initial phase, we tried to use real scores as rewards, which were passed into a multilayer perceptual machine for training, but the results were not satisfactory. We speculate that this may occur for two main reasons. First, due to the low probability of the snake eating food in the initial phase, this led to instability in training. Second, due to the relatively small true reward received for eating the food, this also affected the training effect.

Given this sparse reward situation and the fact that the true score is not very good, we considered using the Q-learning feature function as a reward passed into the MLP for training. The results are much improved compared to using the true score as a reward, which suggests that the Q-learning feature function is more adapted to our training needs and provides an effective reward signal for the snake to learn better behaviors in the environment.

2.4.2. *Improved state representation*

In addition, we improved the state. At first, we only encoded food and its own location.

$$\begin{aligned}\text{state} =&[\text{food\_x}, \text{food\_relative\_y}]\\ &+ [\ \text{pos\_x}\ ,\ \text{pos\_y}\ ]\end{aligned}$$

but later we added the location and environment of other snakes. We ended up using

$$\begin{aligned}\text{state} =&[\text{normalized\_relative\_x},\text{normalized\_relative\_y}]\\ &+ \text{normalized\_obstacles}\\ &+ \text{min\_body}\\ &+ \text{min\_kill}\end{aligned}$$

where normalized_relative_x and normalized_relative_y represent the normalized distances from the agent's coordinates to the food, normalized_obstacles denotes the normalized distance boundaries in each direction, normalized_min_body indicates the minimum distance from the snake's body in four directions (up, down, left, and right) after normalization, and normalized_min_kill signifies the minimum distance from the head of other snakes to the back half of the snake's body after normalization. The sophisticated and detailed state coding enhances our understanding of the environment. This improvement provides the intelligences with more accurate and comprehensive information, enabling them to respond more flexibly to complex decision-making scenarios.

These variables play a crucial role in the neural network, aiding in the effective avoidance of the agent moving towards the boundaries, enemy bodies, and its own tail. Instead, the agent strategically moves towards the direction of the food. Additionally, the normalization of these variables ensures that the model remains unbiased and prevents excessive bias in the loss function.

2.4.3. *Training process*

We use a 40x40 grid and add a random agent and a directed agent as a training environment. The learning rate is multiplied by 0.8 for each epoch mode 10, which reduces the learning rate quickly. But this method has a disadvantage - the performance is good when the test environment is similar to the training situation, e.g. when going from two agents to one agent, the efficiency drops instead.

2.4.4. *Experiments and findings*

In order to test the performance of our MLP Agent, We test it in the following three environments, and record their winning rate and average scores respectively. The result is shown in Table 3 below:

- Compete against one *randomAgent* Snake and one *directionalAgent* Snake.
- Compete against two *randomAgent* Snake.
- Compete against two *directionalAgent* Snake.

Unexpected findings Although our MLP model is not stable enough and doesn't perform as well as minimax and q-learning Agent, we still find some surprising strategy it learned in the training process. The MLP snake will sometimes choose to attack between getting food and attacking other snakes. Here's an example video, the link is as follows: MLP attack other snakes video link.

	MLP Agent
Random&Directional	67.7% 3.1
Random&Random	73.6% 0.4
Directional&Directional	52.3% 2.8

Table 3: Winning rate and average score results of MLP Agent in three test environments

2.5. *DQN agent*

2.5.1. *Overview of Deep Q-Learning (DQN)*

Deep Q-Learning (DQN) is a significant advancement in machine learning, specifically designed to address the challenges in complex and high-dimensional environments. Traditional Q-learning relies on a Q-table to associate rewards with every possible state-action pair, but becomes unmanageable in environments with a large number of possible states and actions.

DQN resolves this issue by replacing the Q-table with a neural network, referred to as the Q-network. This network is trained to approximate the Q-values, thereby eliminating the need to maintain a comprehensive table of all state-action pairs. As a result, DQN can efficiently manage tasks in complex, high-dimensional settings by learning effective strategies where classic Q-learning is not feasible.

2.5.2. *Implementation in the game*

Here, we outline the two main strategies employed by DQN to stabilize training:

- **Experience Replay:** The agent's experiences, defined as tuples of (s_t, a_t, r_t, s_{t+1}), are stored in a replay buffer. During training, mini-batches of these experiences are sampled randomly to update the Q-network.
- **Target Network:** Instead of using a single network for value prediction and target value estimation, DQN utilizes two networks with identical architectures. The *target network*'s weights are periodically updated with the weights of the *prediction network*, which actively learns from new experiences.

The loss function used to train the Q-network is the Mean Squared Error (MSE) between the predicted Q-values and the target Q-values, computed as:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \quad (6)$$

where θ are the parameters of the prediction network, θ^- are the parameters of the target network, and D is the replay buffer. Here, $\mathbb{E}_{(s,a,r,s')\sim D}$ represents the expected value computed over samples drawn from the replay buffer D.

In our implementation of the DQN algorithm for the snake game, the Convolutional Neural Network (CNN), serving as the function approximator, takes the state of the game as input and outputs the estimated Q-values for each possible action. The training process is iterative, with the agent continuously refining its policy based on the feedback from the environment. We utilize experience replay to store and sample from a diverse set of past experiences. The target network provides a stable reference for Q-value updates. We provide a pseudocode summary that includes the key steps of the DQN algorithm as applied to our snake game:

Algorithm 3 Deep Q-Learning for Snake Game

Initialize replay buffer $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1 to M **do**
  Initialize state s_1
  for t = 1 to T **do**
    Choose action a_t with probability ϵ randomly, else $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch from $\mathcal{D}$: (s_j, a_j, r_j, s_{j+1})
    Set $y_j = r_j$ for terminal s_{j+1}, else $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
    Perform gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. θ
    if t mod C == 0 **then**
      Update $\hat{Q}$ with weights $\theta^- = \theta$
    end if
  end for
  Decay ϵ
end for

In our customized implementation of the DQN algorithm for the snake game, we have adapted both the state representation and action definitions to align with the specific requirements of the game. Our trials have shown promising improvements in the performance of the player-controlled snake's movements. However, there remains significant potential to further enhance the DQN agent's performance in this complex environment.

We believe the challenges in optimizing our DQN agent may arise from factors such as limited time for training, the inherent instability associated with neural network training, and the complexities in the game's dynamics, such as interactions with other snakes. By dedicating more time to fine-tune the network and employing advanced strategies, we hope to enhance the performance of our DQN agent in the future.

3. Experiments

3.1. *Mimimax v.s. Q-learning*

We carry out several competitions between minimax and Q-learning to compare their performance. They both perform really well and keep alive even at a long body length.

Here's an example video, the link is as follows: Q-learning V.S. minimax.

*3.2. **Minimax v.s. MLP & Q-learning v.s. MLP***

We also compare the performance of Minimax and Q-learning with MLP respectively:Minimax v.s. MLP, Q-learning v.s. MLP.

We find that MLP's performance is much worse than Minimax and Q-learning because MLP just try to fit a specific training environment, so the generalization ability is poor. Once the opponent is replaced, the effect decreases rapidly because the MLP has not really learned the key factors of the problem.

4. Summary and conclusions

Minimax

Pros: It's a recursive algorithm, so it can predict the state of the opponent in the next few steps, thereby improving the effect of obstacle avoidance.

Cons: As search depth grows deeper, the search space grows exponentially, leading to a very high time complexity. We can improve it by alpha-beta pruning. Another disadvantage is that the demands on the opponent are high. We assume that all opponent snakes are using the optimal strategy, however, when they use other strategies, the prediction results we get become inaccurate.

Q-learning

Pros: Training speed is fast, generalization ability is good.

Cons: Limited by linear combinations, the ability to undertake overly complex tasks is modest.

MLP

Pros: Fitting can take on complex tasks such as killing other snakes.

Cons: Unstable. Performs well when the test environment is similar to the training situation, e.g. when going from two agents to one agent, the efficiency drops instead. Moreover, the generalization ability is poor. Once the opponent is replaced, the effect decreases rapidly because the MLP has not really learned the key factors of the problem.

5. External Libraries

- `pygame`: Creates graphic elements that capture input in response to the keyboard.
- `random`: A Python library that aids in generating random numbers, useful in various aspects of game logic and GUI development.
- `numpy`: A fundamental package for scientific computing in Python, essential for handling large, multi-dimensional arrays and matrices efficiently.
- `torch`: A machine learning library providing a wide range of tools and libraries for deep learning applications, crucial for implementing AI algorithms.
- `collections`: A Python module that offers specialized container datatypes, such as deques, which are optimized for quick append and pop operations from both ends.