Università degli Studi di Padova

University of Padua, Department of Mathematics
Artificial Intelligence Project

A Comparative Study of Reinforcement Learning Strategies in the Snake Game

Michelle Elizabeth Sánchez Puma

ID 2141808

Academic Year 2024/2025

Contents

1 Reinforcement Learning

Reinforcement Learning is a machine learning paradigm along with supervised and unsupervised learning. This approach is characterized by relying on continuous interaction with the environment [2]. The fundamental premise of the reinforcement learning approach is the use of an agent as a human expert who takes actions in a given domain. This agent takes actions in the environment, causing a change in the current state of the environment. After these changes, the agent is rewarded according to the outcome; this reward can be a positive incentive for favorable actions and a negative incentive for unfavorable actions. Finally, the goal of the learning is to maximize the cumulative reward over time by discovering optimal policies [1].

The goal of the reinforcement Learning problem is modeled as a Markov Decision Process [5], which defines the approach using a tuple of:

$$(S, A, P, R, \gamma)$$

where:

- S is the set of states.
- A is the set of actions.
- $P(s' \mid s, a)$ is the probability of transitioning to state s' from state s given action a.
- $R(s, a, s')$ is the reward function.
- $\gamma \in [0, 1]$ is the discount factor.

The return G_t from time step t is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

The goal of the RL agent is to learn a policy $\pi : S \to A$ (or a probability distribution over actions) that maximizes the expected return.

A key challenge in reinforcement learning is the trade-off between exploration and exploitation, where agents must maximize rewards by taking actions they have previously tried and which are known to be positive. However, to discover these actions, agents must also experiment with different actions. This is a trade-off between exploration and exploitation, meaning that to achieve optimal performance, the agent must balance these two aspects [4].

1.1 Fundamental Elements

The principal elements of the reinforcement learning approach are [5]:

- **Agent:** is the decision-maker element that interacts with the environment by choosing and performing actions based on a learned policy.
- **Environment:** is the system on real word that responds to the agent's actions by returning the state and the reward.
- **State (S):** is the current configuration of the environment.
- **Actions (A):** are the set of all possible decisions that the agent can perform.
- **Reward (R):** is a numerical feedback that the environment sends to the agent.
- **Episode:** is a finite number of actions that ends when the agent achieves the final state.
- **Policy (π):** is the key goal of the agent and it defines the agent's behavior. It defines the probability of taking action a, given a state s, which is denoted as $\pi(a, s)$. Once the agent takes an action, the agent uses a value function to evaluate the action.
- **Value Functions:** these functions are a way to represent the reward as Q-value, it can be expressed as an estimation of the expected return from a given state (state-value function) or the pair state-action (action-value function) [1].

The agent's final goal is to select the action with the highest Q-value. Finding the optimal Q-value requires the selection of the optimal actions that result in maximum expected rewards using the optimal policy π. To do so, the Q-value must satisfy the Bellman optimality equation. This equation provides a recursive relationship between the value of a state (or state-action pair) and the values of its successor states, a requirement for optimality [2]. This can be expressed as:

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\big[R(s,a,s') + \gamma \max_{a' \in A} Q^*(s',a') \mid s,a\big] \\ &= \sum_{s' \in S} P(s' \mid s,a)\big[R(s,a,s') + \gamma \max_{a' \in A} Q^*(s',a')\big]. \end{aligned}$$

This Bellman Equation is the basis of reinforcement learning and it is used in the derivation of many RL algorithms, including Q-Learning and SARSA.

1.2 Classification

Reinforcement Learning methods can be categorized based on how the agent learns the environment and optimizes the policy.

1.2.a Model-Free Methods

In these methods, the agent learns only from the interaction with the environment, and it does not build an explicit model of it. In this category, there are value-based methods and policy-based methods [1].

Value-Based Methods

These methods focus on estimating the optimal action-value function using Bellman's equation. Regarding the policy, it is based on choosing actions that maximize this value function, which represents the cumulative reward [1]. The most common algorithms are: Q-learning and SARSA.

Q-Learning Algorithm: This is an off-policy algorithm that updates the value function using Bellman's equation in the following way:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big],$$

where α is the learning rate, r is the immediate reward, and s' is the next state.

This algorithm is simple and is able to learn the optimal policy regardless of the behavior of the current agent. Moreover, under sufficient exploration and using a decaying learning rate, Q-learning is guaranteed to get the convergence of the optimal policy, it is because of the use of the best future action. However, in large or continuous spaces, creating the Q-table becomes infeasible due to the large amount of data [4].

***SARSA (State-action-reward-state-action) Algorithm*:** SARSA is an on-policy algorithm that updates its value function based on the action taken by the current policy, using the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma Q(s',a') - Q(s,a)\big],$$

where a' is the action taken in the subsequent state s'.

This algorithm can result in safer policies because it uses only the actual action and not the expected action in the future as Q-learning does. However, the condition of being on-policy makes SARSA to have a slow convergence and totally dependent on the exploration strategy [4].

Policy-Based Methods

Policy-based methods learn the policy directly and can be policy gradient methods or actor-critic methods.

Policy Gradient Methods: Here, the policy parameters are optimized following the gradient of the expected return as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q^{\pi_\theta}(s,a)\right].$$

The principal algorithm is REINFORCE which computes gradients using the Monte Carlo policy that optimizes the policy adjusting the probability of taking actions to get the higher rewards. The principal advantage of this method is that is particularly useful for stochastic spaces; however, the gradient estimation can have high variance and can converge only to a local optimal [2].

Actor-Crit Methods: Actor-Critic methods combine value-based and policy-based approaches by simultaneously learning a policy (actor) and a value function (critic).

1.2.b Model-Based Methods

In these methods, the agent creates an explicit model of the environment and uses the model for planning the strategies because it simulates the different actions and their results before interacting with the environment [1].

1.2.c Deep Reinforcement Learning

This method integrates deep neural networks to handle high-dimensional inputs in complex environments, this is used for both: model-free and model-based techniques. The most used algorithm is the Deep Q Learning Network which uses a convolutional neural network instead of the Q-table to approximate the action-value function [1].

2 Snake Game Implementation

This project aims to develop a snake game from scratch using Python with the Pygame library and implement on it different reinforcement learning techniques in order to compare the results obtained. In this project, I will compare the following algorithms: Q-learning, SARSA, Policy-based algorithm (REINFORCE), and Deep Q-Learning.

The development of the project was done in two basic steps. First, the implementation of the snake game from scratch and setup the environment. Second, the creation of the models of each one of the four algorithms; and their implementation int the agent that calls any of the four algorithms. The relation between the steps and the code organization can be seen in the diagram of 2.1.

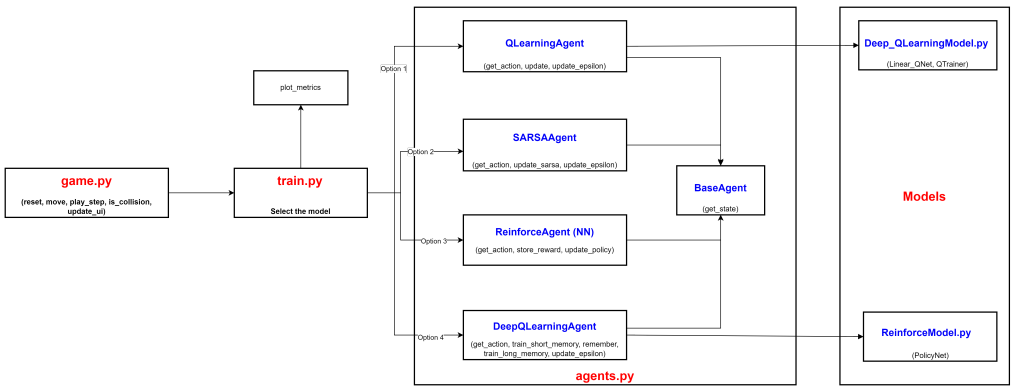

Figure 2.1: Code Organization

2.1 Step 1: Implementation of the game (Game.py)

The implementation of the game uses the library Pygame, which is a module designed for video games in Python. This code development was based on that created in [3].

To set up the environment and a game, I created a file called *game.py* which has a class called *SnakeGameAI*, in which it was defined the method *update_ui()*. This

method as shown in Figure 2.2 draws the snake using blocks of pixels, where the snake's body has a blue color and its head a green color.

```
def update_ui(self):
    """Update the user interface: draw snake, food, and score."""
    self.display.fill(BLACK)
    # Draw snake head in green
    head = self.snake[0]
    pygame.draw.rect(self.display, (0, 255, 0), pygame.Rect(head.x, head.y, BLOCK_SIZE, BLOCK_SIZE))
    # Draw the rest of the snake body in blue
    for pt in self.snake[1:]:
        pygame.draw.rect(self.display, BLUE1, pygame.Rect(pt.x, pt.y, BLOCK_SIZE, BLOCK_SIZE))
        pygame.draw.rect(self.display, BLUE2, pygame.Rect(pt.x + 4, pt.y + 4, 12, 12))
    # Draw food in red
    pygame.draw.rect(self.display, RED, pygame.Rect(self.food.x, self.food.y, BLOCK_SIZE, BLOCK_SIZE))
    # Display score
    text = font.render("Score: " + str(self.score), True, WHITE)
```

Figure 2.2: SnakeGame - update_ui() method.

Moreover, a function was also defined to initialize or reset the game, in which each time the game starts (even if it is the first game or if it was a game over state) the snake resets its size and a new food is placed. Food is always placed in a random way in the environment (see Figure 2.3).

```
def reset(self):
    """Reset the game state to start a new game."""
    self.direction = Direction.RIGHT
    self.head = Point(self.w / 2, self.h / 2)
    self.snake = [
        self.head,
        Point(self.head.x - BLOCK_SIZE, self.head.y),
        Point(self.head.x - (2 * BLOCK_SIZE), self.head.y)
    ]
    self.score = 0
    self.food = None
    self.place_food()
    self.frame_iteration = 0
```

Figure 2.3: SnakeGame - reset()

Regarding the states the snake can perfom, those are defined as follows:

- The direction of a danger for the snake. It means if there is a danger position straight, right, or left.
- The next direction of the snake: left, right, up, or down.
- The position of the food: left, right, up, or down.

In that way, the state of the snake is defined by 11 values, considering the 3 elements listed above.

The next function is one of the most important ones, the *play_step()* function determines the general rules of the game, making the snake move and giving the

agent a reward. **The reward system is easy**: if the snake eats food it will receive +10 points, if the game is over, the snake will have -10 points. In addition, to add a better approximation system, the agent receives a small reward of +0.1 points each time the distance from the food point to the snake head is reducing; to do so, the distance is calculated with a Manhattan formula, and the function created can be seen in the Figure 2.4.

```python
def play_step(self, action):
    self.frame_iteration += 1
    # Handle events (quit event)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    # Calculate previous Manhattan distance to food
    prev_distance = abs(self.head.x - self.food.x) + abs(self.head.y - self.food.y)
    # Move snake based on action
    self.move(action)
    # Calculate new Manhattan distance to food
    new_distance = abs(self.head.x - self.food.x) + abs(self.head.y - self.food.y)
    # Reward shaping: reward for moving closer to food
    reward_shaping = (prev_distance - new_distance) * 0.1

    self.snake.insert(0, self.head)
    game_over = False

    # Check for collisions or frame limit exceeded
    if self.is_collision() or self.frame_iteration > 100 * len(self.snake):
        game_over = True
        reward = -10
        return reward, game_over, self.score

    # Check if food is eaten
    if self.head == self.food:
        self.score += 1
        reward = 10 + reward_shaping  # bonus reward for eating food
        self.place_food()
    else:
        reward = reward_shaping
        self.snake.pop()

    self.update_ui()
    self.clock.tick(SPEED)
    return reward, game_over, self.score
```

Figure 2.4: Snake Game - play-step() with reward system

2.2 Step 2: Implementation of the agents (Agent.py)

In this project, I implement various reinforcement learning (RL) methods in a shared framework for the classic Snake game. The core functionality is in the *agents.py* file, which is structured to make comparative studies between different RL algorithms. The file begins with the *BaseAgent* class that provides a common method (get_state) to transform the game environment into an 11-dimensional binary state representation, explained in the above section.

From this, I developed several agent classes for each one of the algorithms:

2.2.a Q-Learning Algorithm

It applies a tabular Q-Learning algorithm. It possesses a *Q-table* and also updates the table based on the standard Bellman equation with an *epsilon-greedy policy* of action selection for balancing between exploitation and exploration. The major methods it applies are the following, also seen in the Figure 2.5.

- ***get_action(state):*** Has an epsilon-greedy policy that with a probability epsilon, a random action is chosen. Otherwise, the agent selects the action with the highest Q-value from the Q-table. The selected action is returned as a one-hot encoded vector.
- ***update(state, action, reward, next_state, done):*** Updates the Q-value for the pair (state, action) based on the Bellman equation. If the current state or the next state is not in the Q-table, they are initialized to zeros.
- ***update_epsilon():*** Slows down the exploration rate after every game so that the agent increasingly approximates switching from exploration to exploitation.

```python
class TableQAgent(BaseAgent):

    def get_action(self, state):
        """
        Choose an action using an epsilon-greedy strategy.
        """
        final_move = [0, 0, 0]
        if random.randint(0,200) < self.epsilon:
            action = random.randint(0, 2)
        else:
            state_key = tuple(state)
            if state_key not in self.q_table:
                self.q_table[state_key] = [0, 0, 0]
            action = np.argmax(self.q_table[state_key])
        final_move[action] = 1
        return final_move

    def update(self, state, action, reward, next_state, done):
        """
        Update the Q-table using the Q Learning update rule:
          Q(s, a) = Q(s, a) + alpha * [reward + gamma * max(Q(s', a')) - Q(s, a)]
        """
        state_key = tuple(state)
        next_state_key = tuple(next_state)
        if state_key not in self.q_table:
            self.q_table[state_key] = [0, 0, 0]
        if next_state_key not in self.q_table:
            self.q_table[next_state_key] = [0, 0, 0]

        action_index = np.argmax(np.array(action))
        current_q = self.q_table[state_key][action_index]
        max_next_q = max(self.q_table[next_state_key])
        target = reward + (0 if done else self.gamma * max_next_q)
        self.q_table[state_key][action_index] = current_q + self.alpha * (target - current_q)

    def update_epsilon(self):
        self.n_games += 1
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
```

Figure 2.5: Agents - Q Learning Algorithm

2.2.b SARSA Algorithm

Unlike the off-policy Q-Learning method, SARSA updates its Q-values using the actual action really taken in the next state, which can lead to more stable learning

in certain environments (see the code in Figure 2.6).

- ***get_action(state):*** Same as TableQAgent, this method employs an epsilon-greedy policy and returns a one-hot encoded action.
- ***update_sarsa(state, action, reward, next_state, next_action, done):*** Here, instead of using the maximum Q-value in the next state, it uses the Q-value for the particular action really taken in the next state (next_action). This "on-policy" approach leads to more stable, if occasionally more conservative, learning.
- ***update_epsilon():*** Like TableQAgent, it decays over time.

```python
class SarsaQAgent(BaseAgent):
    def get_action(self, state):
        """
        Choose an action using an epsilon-greedy strategy.
        """
        final_move = [0, 0, 0]
        if random.randint(0, 200) < self.epsilon:
            action = random.randint(0, 2)
        else:
            state_key = tuple(state)
            if state_key not in self.q_table:
                self.q_table[state_key] = [0, 0, 0]
            action = np.argmax(self.q_table[state_key])
        final_move[action] = 1
        return final_move

    def update_sarsa(self, state, action, reward, next_state, next_action, done):
        """
        Update the Q-table using the SARSA update rule:
          Q(s, a) = Q(s, a) + alpha * [reward + gamma * Q(s', a') - Q(s, a)]
        """
        state_key = tuple(state)
        next_state_key = tuple(next_state)
        if state_key not in self.q_table:
            self.q_table[state_key] = [0, 0, 0]
        if next_state_key not in self.q_table:
            self.q_table[next_state_key] = [0, 0, 0]

        action_index = np.argmax(np.array(action))
        next_action_index = np.argmax(np.array(next_action))
        current_q = self.q_table[state_key][action_index]
        # For terminal states, next Q is 0
        next_q = 0 if done else self.q_table[next_state_key][next_action_index]
        target = reward + self.gamma * next_q
        self.q_table[state_key][action_index] = current_q + self.alpha * (target - current_q)

    def update_epsilon(self):
        self.n_games += 1
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
```

Figure 2.6: Agents - SARSA Algorithm

2.2.c Reinforce Algorithm (Policy Based)

This agent employs the REINFORCE algorithm, which is a policy gradient method of learning a stochastic policy directly. The agent generates actions from the policy network probability distribution and saves the log probabilities and rewards encountered. An entropy bonus is incorporated into the loss function to enhance exploration. The code is shown in Figure 2.7.

- ***get_action(state):*** Passes the state through a policy network (*PolicyNet*) to output a probability distribution over actions (with a softmax layer). Then,

it saves the probability of the chosen action to use later at policy update time.

- ***store_reward(reward):*** Simply append the reward received into a list for the current episode.
- ***update_policy():*** Upon end of an episode, the agent:
  - Computes discounted returns on all time steps.
  - Scales the returns by normalizing in order to make training stable.
  - Computes loss as negative sum of (*log probability* × *adjusted return*) plus an entropy bonus to encourage exploration.
  - Takes a gradient descent step to adjust the policy network parameters.
  - Resets the cached log probabilities and rewards after adjusting for the next episode.

```python
class NNAgent(BaseAgent):
    def update_policy(self):
        """
        R = 0
        returns = []
        # Compute  See Real World Examples From GitHub  r)
        for r in reversed(self.rewards):
            R = r + self.gamma * R
            returns.insert(0, R)

        returns = torch.tensor(returns, dtype=torch.float)

        # Compute baseline as the moving average of past rewards
        baseline = np.mean(self.reward_baseline) if self.reward_baseline else 0
        returns = returns - baseline  # Reduce variance with baseline

        # Normalize returns
        if len(returns) > 1:
            returns = (returns - returns.mean()) / (returns.std() + 1e-9)

        policy_loss = 0
        entropy_loss = 0
        for log_prob, R in zip(self.log_probs, returns):
            policy_loss += -log_prob * R
            entropy_loss += -log_prob  # Approximation of entropy

        entropy_coef = 0.095  # Entropy coefficient to encourage exploration
        loss = policy_loss + entropy_coef * entropy_loss

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

Figure 2.7: Agents - Reinforce Algorithm

2.2.d Deep Q-Learning Algorithm

This agent employs Deep Q-Learning, employing a neural network (declared in *deep_qlearning_model.py*) as an approximator of Q-values. It applies the following methods (see Figure 2.8).

- ***get_action(state):*** Performs an epsilon-greedy action selection. Random action with probability or agent's neural network calculates Q-values and chooses the most valuable action.
- ***train_short_memory(state, action, reward, next_state, done):*** Trains the agent for one experience tuple by backpropagating the data through the *QTrainer*, which uses the Q-Learning rule to update with the neural network as an approximating function.
- ***train_long_memory():*** From time to time takes a batch of experience out of replay memory to make a more stable update to training.
- ***update_epsilon():*** Slows the rate at which the exploration rate decays after each episode.

Regarding the deep neural network used, it is as follows:

- ***Linear_QNet:*** A feedforward neural network consisting of an input layer of equal size to the state (11 in this project), a hidden layer (256 neurons), and an output layer with Q-values for each possible action (3 actions). The network uses ReLU as an activation function.
- ***QTrainer:*** This class is employed to train *Linear_QNet*. It computes the loss (utilizes mean squared error between target Q-values computed with the Bellman equation and predicted Q-values) and performs backpropagation using an optimizer such as Adam.

```python
class DeepQAgent(BaseAgent):
    def train_short_memory(self, state, action, reward, next_state, done):
        """
        Train on a single experience.
        """
        self.trainer.train_step(state, action, reward, next_state, done)

    def remember(self, state, action, reward, next_state, done):
        """
        Store an experience in the replay memory.
        """
        self.memory.append((state, action, reward, next_state, done))

    def train_long_memory(self):
        """
        Train on a batch of experiences sampled from the replay memory.
        """
        if len(self.memory) > self.batch_size:
            mini_sample = random.sample(self.memory, self.batch_size)
        else:
            mini_sample = self.memory
        states, actions, rewards, next_states, dones = zip(*mini_sample)
        self.trainer.train_step(states, actions, rewards, next_states, dones)

    def update_epsilon(self):
        self.n_games += 1
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
```

Figure 2.8: Agents - Deep Q-Learning Algorithm

3 Snake Game Results

For this project, tree metrics were used to evaluate the performance of each algorithm according to the game it performs. The metrics are as follows.

- **Max/Record Score per Episode:** Max/Record Score per episode measure shows the maximum score achieved for each episode and also keeps track of the maximum (record) achieved so far up to that point while training. In project analysis, this measurement is an indicator of the performance of the agent to optimally solve the task because a higher record score indicates greater efficiency in environmental exploration.
- **Episode Length (Number of Steps):** Episode Length, measured as the number of steps from initialization to termination, is employed as a measure of the survival and stability of the agent in the environment. More extended episodes indicate that the agent is successfully avoiding terminal states. However, more abbreviated episodes may indicate a lack of learning or suboptimal policy behavior.
- **Learning Curve (Mean Score)** The Learning Curve is the smoothed average score gained per episode over time. A rising learning curve with time implies that the agent is learning progressively and that the learning algorithm is moving towards an optimal policy.

Those metrics enable a detailed comparison of reinforcement learning algorithms. In this project, all methods were evaluated in a range of 250 games divided into periods of 50 repetitions each, the results of which are presented in the following subsections.

3.0.a Q-Learning Algorithm

The algorithm results show a trend towards improvement over time. As shown in Table 3.1, the first 50 games have a maximum score of 36 gained in game number 42, indicating that the algorithm learns quickly how to perform over the environment. The maximum score continued to grow with the number of games, and it can be noticed that between the 50 - 100 game period, it did not have a good improvement, indicating that the game was stuck on a plateau.

However, during the next periods, the score increased considerably, indicating that the game found an optimal policy, and finished with a maximum score of 66. The graphics results can also be seen in Figures 3.1, 3.2, 3.3, 3.4, 3.5, which corresponds to each of the periods of 50 games each. In those, the growth of the maximum score is represented, as well as the episode length over the periods of games. The graph of the learning curve is also noticeable, which shows a continuous improvement in agent learning.

Period of Games	Max Score	Game
0 - 50	36	42
50 - 100	37	71
100 - 150	56	121
150 - 200	59	164
200 - 250	66	243

Table 3.1: Q-Learning Algorithm - Results

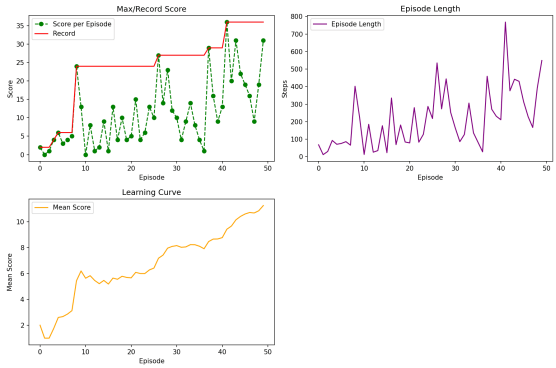

Figure 3.1: Q-Learning Algorithm - Results 0-50 games

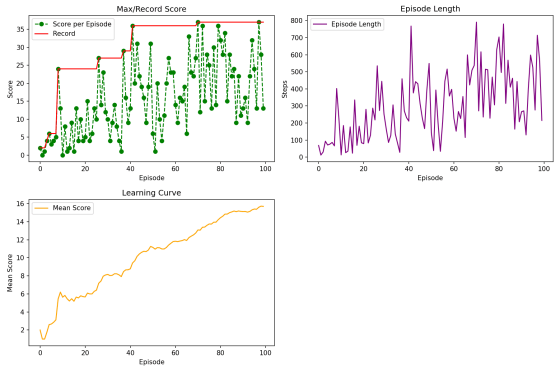

Figure 3.2: Q-Learning Algorithm - Results 50-100 games

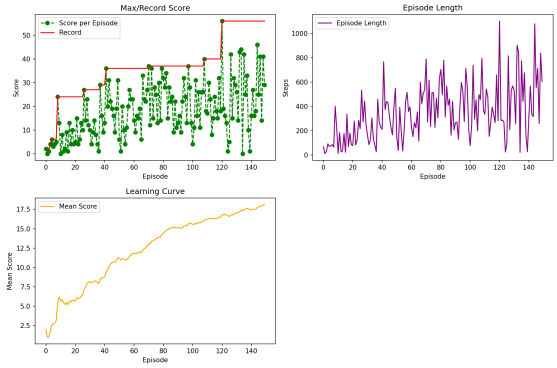

Figure 3.3: Q-Learning Algorithm - Results 100-150 games

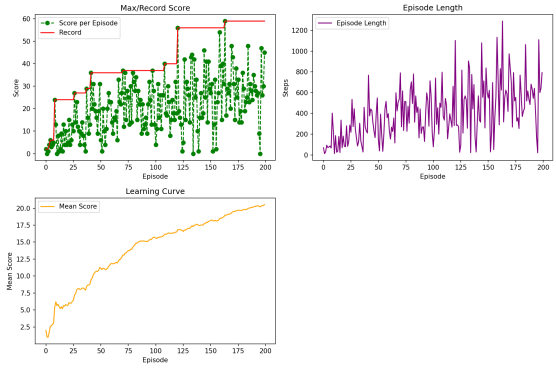

Figure 3.4: Q-Learning Algorithm - Results 150-200 games

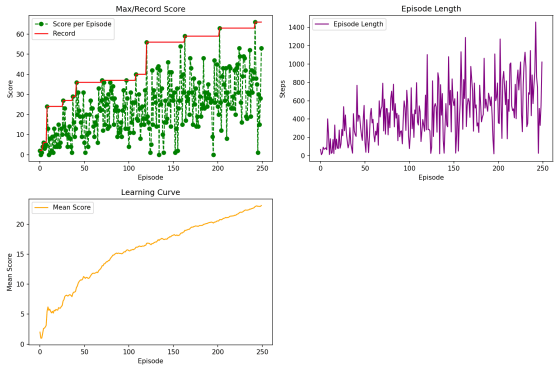

Figure 3.5: Q-Learning Algorithm - Results 200-250 games

3.0.b SARSA Algorithm

The algorithm results show a trend towards improvement over time. As shown in Table 3.2, the first 50 games have a maximum score of 25 gained in game number 14, indicating that the algorithm learns more slowly compared to the Q-Leaning algorithm, it is because SARSA uses a more conservative way to update the Q-value. The maximum score continued to grow with the number of games, and it can be noticed that between the 100 - 150 game period, it did not have any improvement, indicating that the game was stuck on a plateau. However, during the last period, the score increased considerably, indicating that the game found an optimal policy, and finished with a maximum score of 70. The graphics results can also be seen in Figures 3.6, 3.7, 3.8, 3.9, 3.10, which corresponds to each of the periods of 50 games each. In those, the growth of the maximum score is represented, as well as the episode length over the periods of games. The graph of the learning curve is also noticeable, which shows a continuous improvement in agent learning.

Period of Games	Max Score	Game
0 - 50	25	14
50 - 100	45	87
100 - 150	45	87
150 - 200	49	157
200 - 250	70	235

Table 3.2: SARSA Algorithm - Results

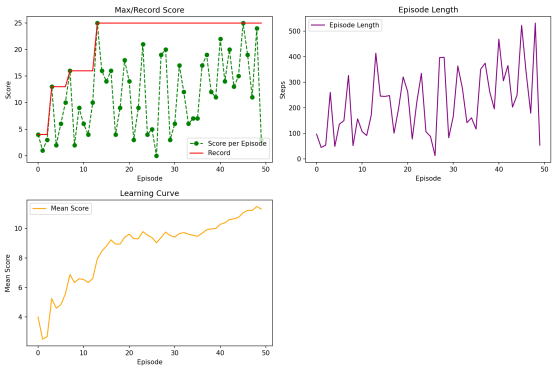

Figure 3.6: SARSA Algorithm - Results 0-50 games

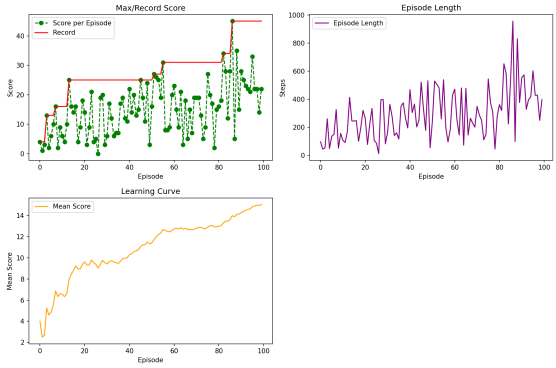

Figure 3.7: SARSA Algorithm - Results 50-100 games

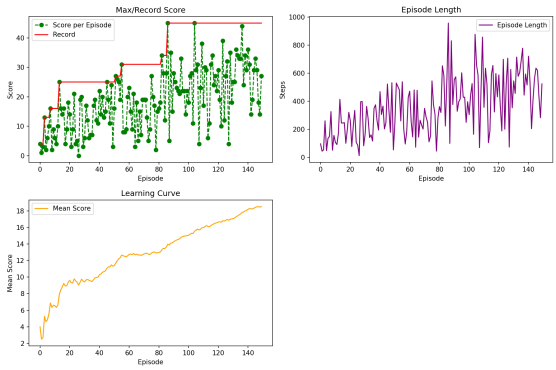

Figure 3.8: SARSA Algorithm - Results 100-150 games

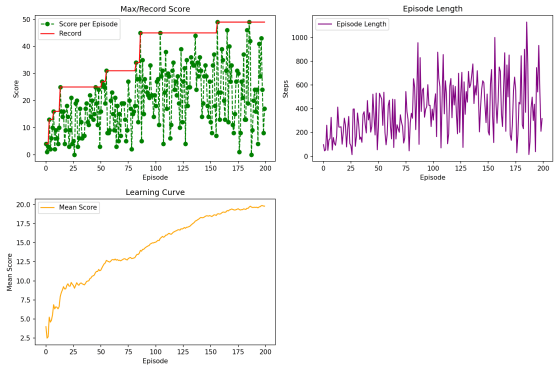

Figure 3.9: SARSA Algorithm - Results 150-200 games

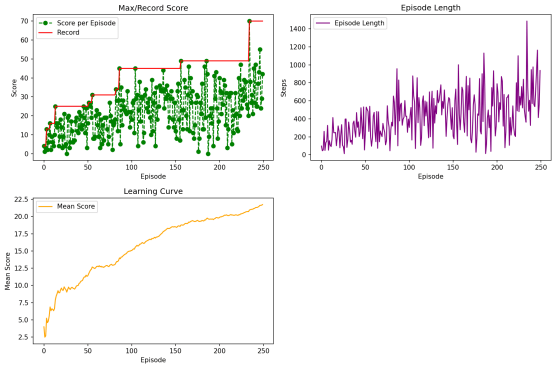

Figure 3.10: SARSA Algorithm - Results 200-250 games

3.0.c Reinforce Algorithm

The algorithm results show a gradual trend toward improvement over time. As shown in Table 3.3, the first 50 games have a maximum score of only 2 obtained in game number 33, indicating that this kind of algorithm only updates the policy after the end of each episode, and not at the end of each step as the Q-value algorithms do. Moreover, this algorithm learns a stochastic policy, which means that it needs more exploration, making that at the initial stages it takes longer to achieve a better performance. However, since the last period of 200 - 250 games, the algorithm improved, increasing the record to 27. The graphics results can also be seen in Figures 3.11, 3.12, 3.13, 3.14, 3.15, which corresponds to each of the periods of 50 games each. In those, the growth of the maximum score is represented, as well as the episode length over the periods of games. The graph of the learning curve is also noticeable, which shows a slow improvement in agent learning.

Period of Games	Max Score	Game
0 - 50	2	33
50 - 100	6	84
100 - 150	7	147
150 - 200	9	198
200 - 250	27	246

Table 3.3: Reinforce Algorithm - Results

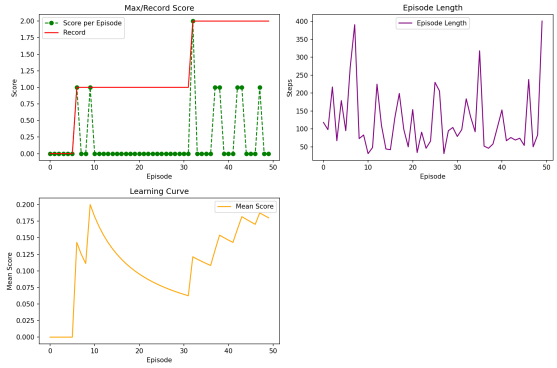

Figure 3.11: Reinforce Algorithm - Results 0-50 games

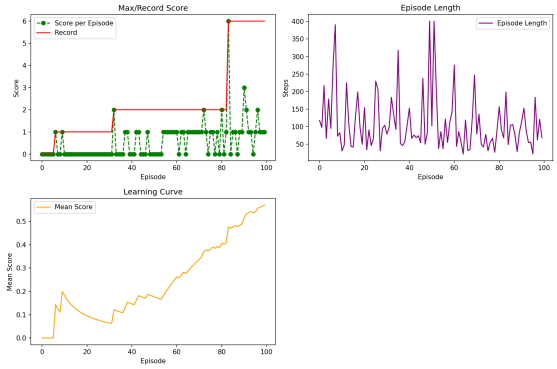

Figure 3.12: Reinforce Algorithm - Results 50-100 games

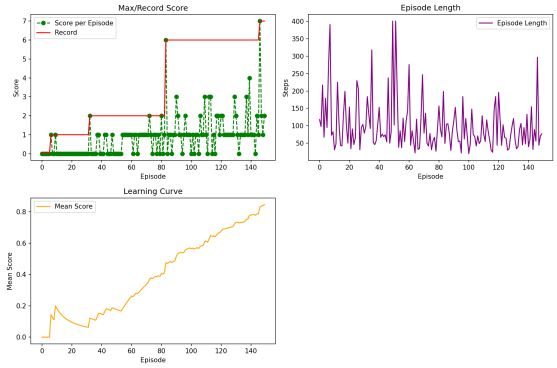

Figure 3.13: Reinforce Algorithm - Results 100-150 games

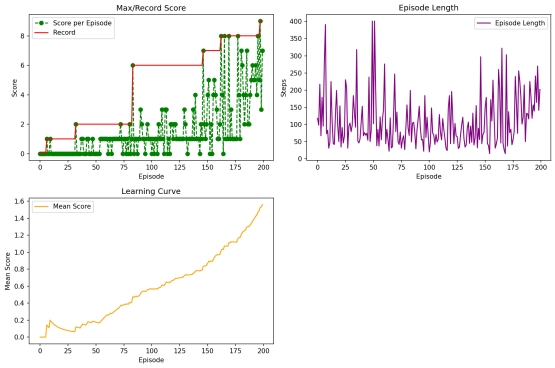

Figure 3.14: Reinforce Algorithm - Results 150-200 games

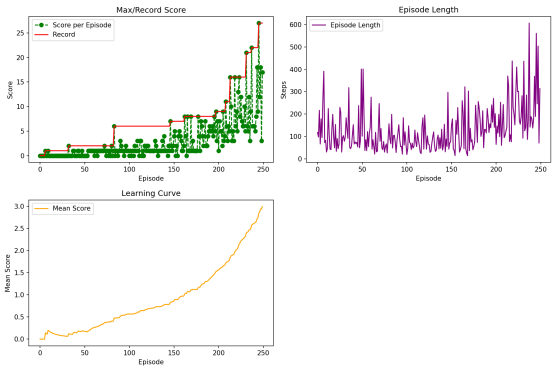

Figure 3.15: Reinforce Algorithm - Results 200-250 games

3.0.d Deep Q-Learning Algorithm

The algorithm results show a trend towards improvement over time. As shown in Table 3.4, the first 50 games have a maximum score of 38 gained in game number 50, indicating that the algorithm learns consistently over time, and this is the maximum value achieved compared to the other algorithms. The maximum score continued to grow with the number of games, and it can be noticed that between the 100-250 game period, there was no improvement, indicating that the game was stuck on a plateau. However, the maximum score is 71, which is again the best among the other algorithms, indicating that the game found an optimal policy before the others. The graphics results can also be seen in Figures 3.16, 3.17, 3.18, 3.19, 3.20, which corresponds to each of the periods of 50 games each. In those, the growth of the maximum score is represented, as well as the episode length over the periods of games. The graph of the learning curve is also noticeable, which shows a continuous improvement in agent learning.

Period of Games	Max Score	Game
0 - 50	38	50
50 - 100	59	77
100 - 150	71	101
150 - 200	71	101
200 - 250	71	101

Table 3.4: Deep Q-Learning Algorithm - Results

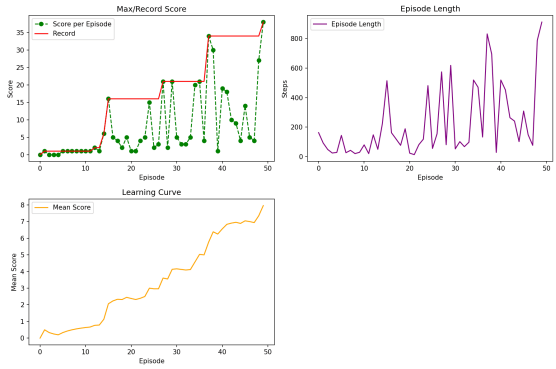

Figure 3.16: Deep Q-Learning Algorithm - Results 0-50 games

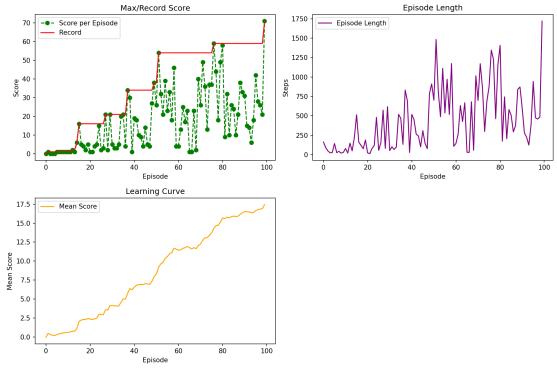

Figure 3.17: Deep Q-Learning Algorithm - Results 50-100 games

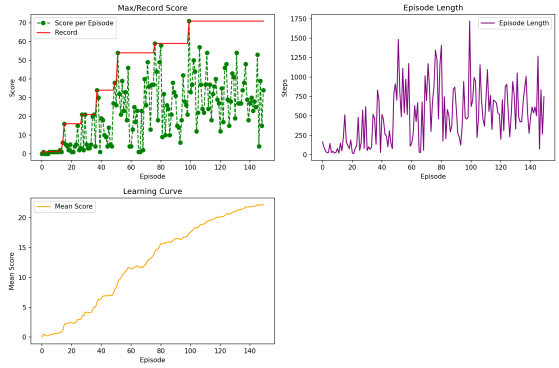

Figure 3.18: Deep Q-Learning Algorithm - Results 100-150 games

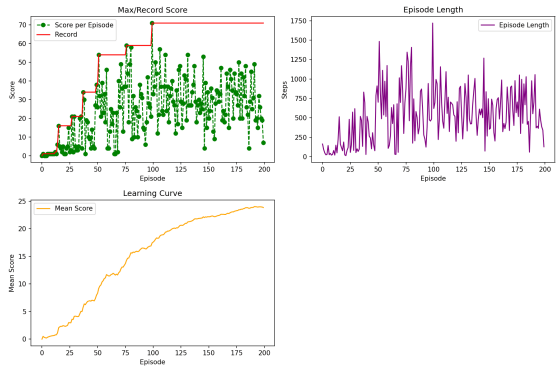

Figure 3.19: Deep Q-Learning Algorithm - Results 150-200 games

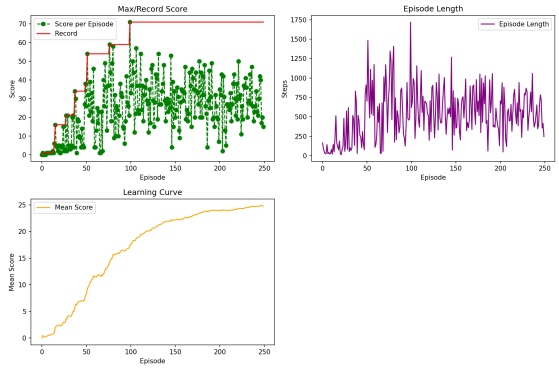

Figure 3.20: Deep Q-Learning Algorithm - Results 200-250 games

Finally, a video on how the complete code perform in the Deep Q-Learning can be accessed here: Video

Conclusions

In this project, I have implemented and compared four reinforcement learning algorithms: Tabular Q-Learning, SARSA, REINFORCE, and Deep Q-Learning in the Snake game environment. The evaluation was performed using three primary metrics: Max / record score per episode, episode length and learning curve (mean score). These metrics provided insight into each algorithm's performance in terms of maximum rewards, stability, and overall learning progression.

The tabular Q-learning algorithm showed rapid initial improvement, with a best score of 66 during the final period. Its performance is sometimes plateaued, due to limitations in the discretization of the state space. By comparison, the SARSA algorithm, which is an on-policy algorithm, had a more cautious learning style, improving at a slower rate but ultimately to a higher record score of 70. This indicates that SARSA's on-policy updates may result in more stable policies under certain circumstances, but at the expense of reduced convergence.

The REINFORCE algorithm, being a policy gradient algorithm, improved at a much slower rate. With a record score of only 27 after 250 games, its performance was compromised by excessive variance and large exploration needs. Finally, the Deep Q-Learning algorithm excelled over the rest by delivering the highest record score of 71. The ability to estimate the Q-function by a deep neural network allowed for better generalization and convergence speed, although learning curve plateaus are experienced from time to time.

Overall, the result indicates that although tabular algorithms such as Q-Learning and SARSA may perform well in a low-dimensional space, deep learning-based algorithms such as Deep Q-Learning offer superior performance on both final scores and convergence rate. The slower performance of REINFORCE helps to highlight the problem with high variance in policy gradient methods, showing the potential for implementing more advanced techniques, such as actor-critic architectures. Future work must develop these approaches and validate their scalability within more difficult environments.

Bibliography

[1] Fadi AlMahamid and Katarina Grolinger. "Reinforcement learning algorithms: An overview and classification". In: *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2021, pp. 1–7.

[2] Andrew G Barto. "Reinforcement learning". In: *Neural systems for control*. Elsevier, 1997, pp. 7–30.

[3] Patrick Loeber. *patrickloeber/snake-ai-pytorch*. original-date: 2020-12-10T15:30:00Z. Feb. 2025. URL: https://github.com/patrickloeber/snake-ai-pytorch (visited on 02/18/2025).

[4] Richard S Sutton. "Reinforcement learning: An introduction". In: *A Bradford Book* (2018).

[5] Marco A Wiering and Martijn Van Otterlo. "Reinforcement learning: A survey". In: *Adaptation, learning, and optimization* 12.3 (2012). Publisher: Springer, p. 729.