Artificial Intelligence for Classic Snake Game

Dawid Kosiński

1. Introduction

In this project, we focus on developing a reinforcement learning-based artificial intelligence (AI) for the classic game Snake. The project explores the integration of machine learning techniques, specifically using Deep Q-Learning (DQL), to train an AI agent capable of playing Snake autonomously. The objective is to create an AI that not only learns from its environment but also improves its performance over time through continuous learning. This project highlights the process of integrating reinforcement learning with game development and delves into the challenges and solutions involved in this task.

2. Analysis of the task

2.1 Possible approaches to solve the problem

Various approaches can be adopted to tackle the problem of training an AI agent to play Snake. The primary methods include:

- Rule-Based Approach: One possible approach to solving the problem is by implementing a rule-based system where the snake follows a predefined set of rules to avoid obstacles and chase the food. This approach is simple to implement and requires minimal computational resources.

However, it lacks flexibility, as the snake will only behave according to the hardcoded rules and will not adapt to new scenarios or learn from its experiences.

- Heuristic-Based Approach: A more sophisticated method could involve using heuristics such as the A* algorithm or other pathfinding algorithms to navigate the snake toward the food while avoiding collisions. This method provides better performance compared to the rule-based approach, especially in more complex game scenarios. However, it still lacks the ability to learn and improve over time.

- Machine Learning Approach (Reinforcement Learning): The selected methodology for this project is based on reinforcement learning (RL), specifically using a deep Q-network (DQN) to train the AI agent. This approach allows the AI to learn from its experiences by interacting with the game environment and receiving rewards for successful actions. The DQN is capable of handling high-dimensional input spaces and can generalize its learning to new situations, making it a more powerful and adaptive solution compared to rule-based and heuristic methods.

  Pros:

  - Adaptability: The AI can learn and adapt to new situations.
  - Performance Improvement: Continuous learning can lead to improved performance over time.
  - Generalization: The model can generalize its learning to handle unseen game states.

  Cons:

  - Computationally Intensive: Training the DQN requires significant computational resources.
  - Complexity: Implementing a deep Q-network is more complex compared to simpler rule-based approaches.
  - Training Time: The training process can be time-consuming, especially when dealing with large state spaces.

2.2 Presentation of possible datasets

In the context of this project, the "dataset" refers to the state-action-reward tuples that are generated during the gameplay. Unlike supervised learning, reinforcement learning does not rely on a predefined dataset. Instead, it collects data as the agent interacts with the environment. The key components of the data are:

- State: The current configuration of the game, including the snake's position, direction, and the location of the food.
- Action: The decision made by the AI (move left, right or go straight).
- Reward: The feedback provided to the AI, which can be positive (when the snake eats the food) or negative (when the snake hits a wall or itself).

The dataset is dynamically generated and updated throughout the training process. This real-time data collection is integral to the reinforcement learning process, enabling the AI to refine its strategy and improve its performance.

2.3 Analysis of available tools and libraries

Several tools and libraries were considered for implementing the AI-powered Snake game:

- Pygame: Pygame is a popular Python library for creating video games. It provides functionalities for handling graphics, sound, and user inputs, making it an ideal choice for developing the Snake game. Pygame was selected due to its ease of use and robust support for 2D game development.

- PyTorch: PyTorch is an open-source machine learning library that is widely used for deep learning applications. It provides a flexible platform for building and training neural networks. PyTorch was chosen for implementing the deep Q-network due to its powerful tools for automatic differentiation and GPU acceleration.

- NumPy: NumPy is a fundamental package for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy was used in this project for handling the numerical operations required in the Q-learning process.

- Matplotlib: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It was utilized in this project to plot the training progress, such as the score evolution over episodes and the average score.

The combination of Pygame, PyTorch, NumPy, and Matplotlib was selected to implement the Snake game, train the AI model, and visualize the results. This selection was driven by the need for a flexible and powerful environment to develop, train, and test the AI, as well as to present the outcomes effectively.

3. Internal and external specification of the software solution

3.1 Classes and their implementation

The software solution is composed of several key classes, each handling different aspects of the AI training and game simulation process.

- SnakeGameAI Class

This class is responsible for managing the game environment. It initializes the game state, handles player actions, updates the game state based on the agent's moves, and checks for collisions. It also implements the reward system that guides the AI's learning process.

- DeepQNetwork Class

  This class defines the neural network architecture used by the AI agent. The network receives the game state as input and outputs Q-values for possible actions. The architecture is relatively simple, with an input layer, a hidden layer, and an output layer, utilizing the ReLU activation function.

- NeuralNetworkTrainer Class

  The NeuralNetworkTrainer class manages the training process. It implements the loss function, optimization step, and updates the model weights based on the agent's experiences. The class uses the Adam optimizer, which is well-suited for reinforcement learning tasks due to its adaptive learning rate.

- Agent Class

  This class represents the AI agent that interacts with the game environment. The agent stores experiences in memory, selects actions based on an epsilon-greedy policy, and trains the neural network using both short-term and long-term memory. It also manages the exploration-exploitation trade-off through epsilon decay.

3.2 External specification

The software solution is designed to be executed in a Python environment with the necessary libraries installed. The key external interfaces include:

- AI Training and Execution: The AI-driven version of the game (Snake_AI.py) runs autonomously, with the AI making decisions based on the learned policy. The train_snake_ai.py script is used to train the AI model, which can be executed from the command line.
- Visualization: The training progress can be visualized using the plot generated by plot_utils.py, providing insights into the AI's learning curve.

4. Experiments

The experiments conducted in this project aimed to evaluate the performance and learning capabilities of an AI agent trained to play the Snake game using reinforcement learning techniques, specifically deep Q-learning. The primary goal was to observe how various parameters and conditions affected the AI's ability to learn and perform effectively in the game environment.

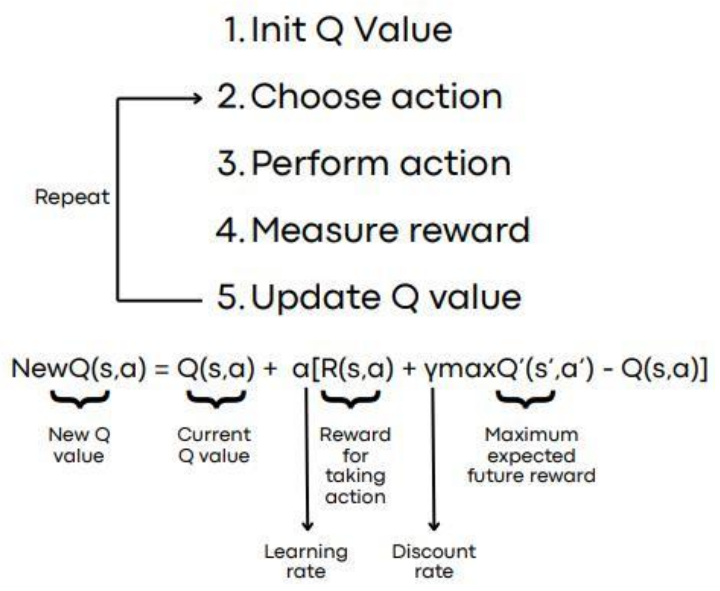
Parameters and Conditions Subject to Change:

- Learning Rate: The learning rate is a crucial hyperparameter in the training of neural networks. It determines the size of the steps the optimizer takes when updating the network's weights. A too high learning rate might cause the network to converge quickly to a suboptimal solution, while a too low learning rate can result in slow convergence. Experiments were conducted by varying the learning rate to identify the optimal value that balances convergence speed and performance.

- Discount Factor (Gamma): The discount factor determines the importance of future rewards in the Q-learning process. A higher discount factor makes the AI focus more on long-term rewards, while a lower factor emphasizes short-term gains. Different values of gamma were tested to observe how the AI's strategy and overall performance were influenced.
- Exploration Rate (Epsilon): The exploration-exploitation trade-off is managed by the epsilon parameter in the epsilon-greedy policy. Higher epsilon values encourage exploration of new actions, while lower values favor exploiting known successful strategies. The experiments involved adjusting epsilon decay rates to find the best approach for balancing exploration and exploitation.
- Reward Function Design: The design of the reward function is pivotal in reinforcement learning as it directly influences the agent's behavior. Various reward schemes were tested, including simple rewards for eating food and penalties for collisions, as well as more sophisticated schemes that also rewarded the AI for moving towards the food or exploring new areas of the board.

Deep Q Learning

In DQN, a neural network is used to approximate the Q-values, which estimate the value of taking a certain action in a given state.



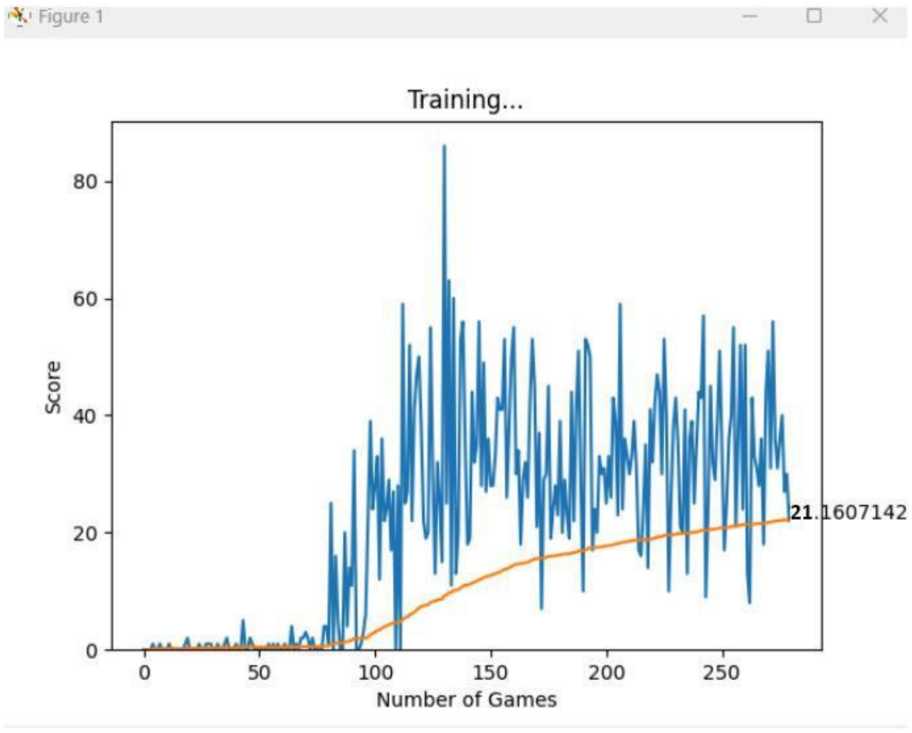
Parameters of our best result:

- MAX_MEMORY = 100_000: The size of replay memory.
- BATCH_SIZE = 1000: The size of the training batch.
- LR = 0.001: The learning rate.
- gamma = 0.9: The discount rate for future rewards.

First results of 290 games:

Average Score: 21.16,

Highest Score Achieved: 86.

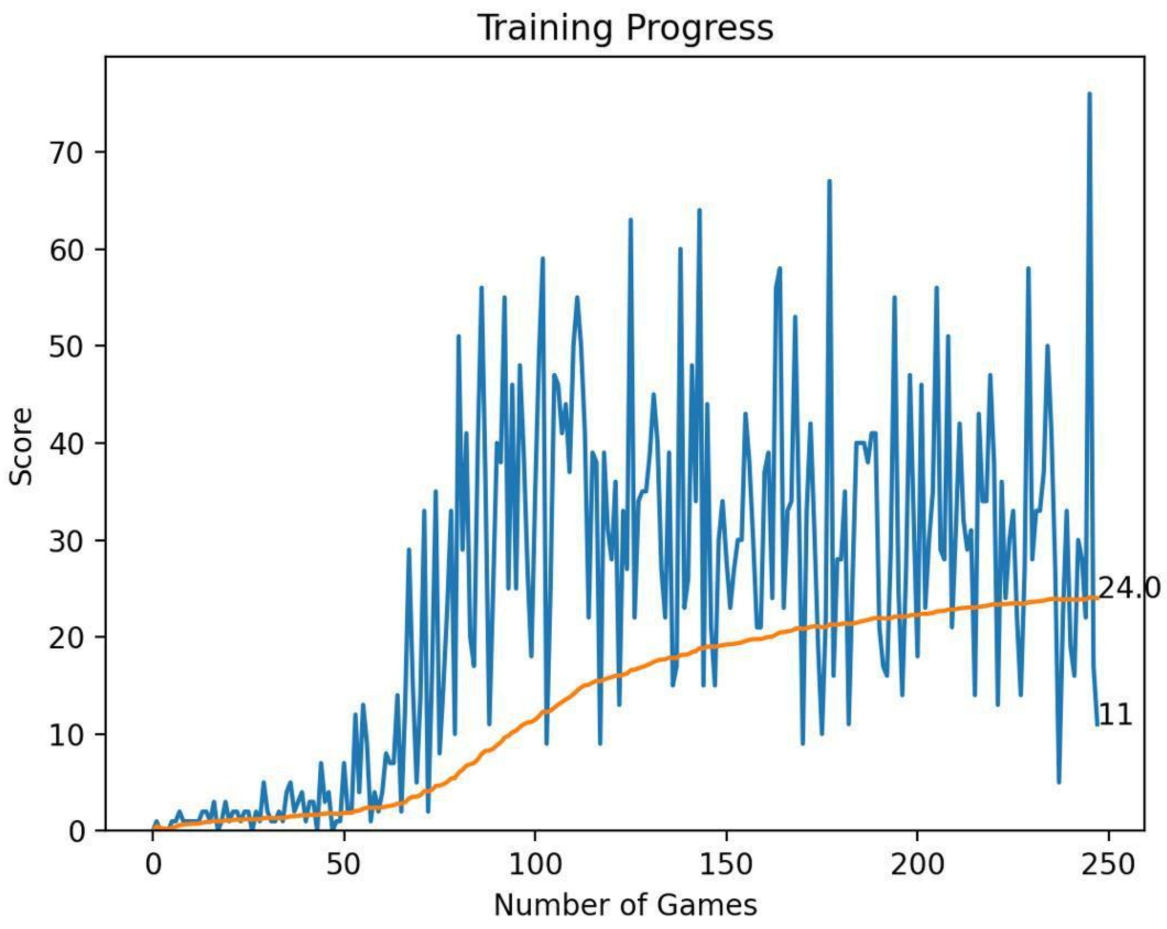


Results when snake gets rewards when he moves towards the food and negative points when goes away from the food:

Results after 248 games:

Average Score: 24,

Highest Score Achieved: 76.



Observations: The snake avoids walls very well, but when the tail is too long it sometimes enters a loop created by its own body from which there is no escape.

5. Summary

This project successfully implemented and trained an AI agent using deep Q-learning to play the Snake game. Through a series of experiments, various parameters such as the learning rate, discount factor, exploration rate, network architecture, reward function, and batch size were optimized to enhance the AI's performance. The AI was able to learn effective strategies for navigating the game environment, avoiding obstacles, and maximizing its score.

The project demonstrated the effectiveness of reinforcement learning, specifically deep Q-learning, in training an AI to autonomously play a complex game like Snake. Key conclusions include:

- The importance of hyperparameter tuning in achieving optimal AI performance.
- The need for a well-designed reward function to guide the AI's learning process.
- The benefits of using experience replay and larger batch sizes for stabilizing the training process and improving the AI's ability to generalize.

Possible future work:

- More Complex Reward Functions: Developing more sophisticated reward functions that take into account additional factors, such as the distance to the food or the length of the snake, could lead to more nuanced AI behavior and better performance.
- Multi-Agent Scenarios: Extending the AI to operate in multi-agent environments, where multiple snakes compete for food, could provide valuable insights into competitive strategies and cooperative behaviors.
- Transfer Learning: Investigating the potential of transfer learning, where the AI is trained on a simpler version of the game and then fine-tuned on more complex scenarios, could lead to faster and more efficient learning.
- Adaptive Learning Rates: Implementing adaptive learning rate strategies, such as using a learning rate scheduler, could further enhance the training process by adjusting the learning rate dynamically based on the training progress.

6. References

https://docs.python.org/3/

https://matplotlib.org/stable/users/index.html

https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

https://www.youtube.com/watch?v=wc-FxNENg9U

https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc