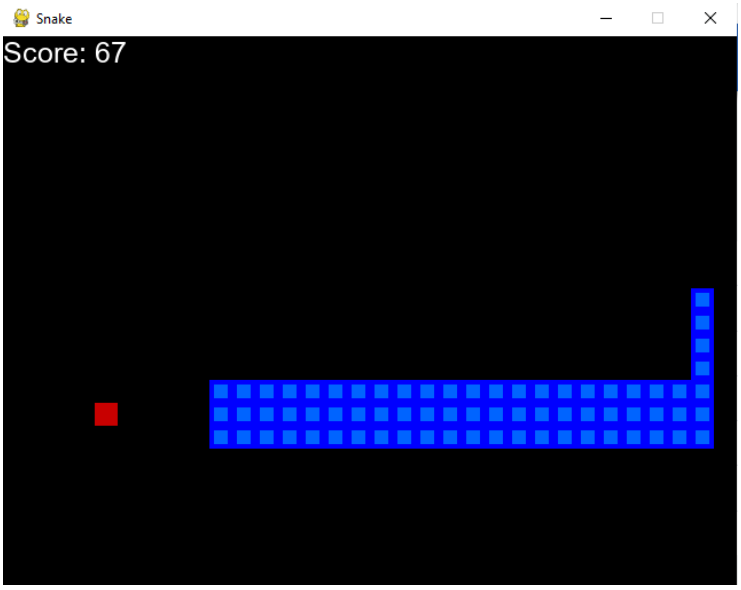
Autonomus and Adaptive System : Snake Game

Sumit Kumar Mishra sumitkumar.mishra@studio.unibo.it
0001055762

Jan 2024



Contents

1 Introduction

1.1 What is Reinforcement Learning?

Reinforcement learning is a type of machine learning paradigm where an agent learns to make decisions by interacting with its environment. The agent receives feedback in the form of rewards or punishments for the actions it takes, and its objective is to maximize the cumulative reward over time.

Here's a brief overview of the key components of reinforcement learning:

1. **Agent**: The entity that makes decisions and takes actions within an environment.
2. **Environment**: The external system with which the agent interacts. The environment responds to the agent's actions and provides feedback.
3. **State**: A representation of the current situation or configuration of the environment.
4. **Action**: The decision or move made by the agent in a particular state.
5. **Reward**: A numerical signal that indicates the immediate benefit or cost of the agent's action.
6. **Policy**: A strategy or mapping from states to actions, indicating the agent's decision-making strategy.

The agent's goal is to learn an optimal policy that allows it to take actions in different states to maximize the cumulative reward over time. Reinforcement learning has been successfully applied in various domains, including robotics, gaming, finance, and more.

1.2 What is DeepQ learning?

Deep Q-Learning (DQN) is a type of reinforcement learning algorithm that combines Q-learning with deep neural networks. Q-learning is a traditional reinforcement learning technique used for making decisions in an environment with the goal of maximizing cumulative rewards over time. By incorporating deep neural networks, DQN can handle more complex and high-dimensional state spaces, making it suitable for tasks such as image-based decision-making in computer vision applications and playing video games.

2 Snake Reinforcement Learning With PyTorch and Pygame

The Snake Reinforcement Learning project involves implementing a reinforcement learning algorithm to train an agent to play the classic Snake game. This report details the use of PyTorch for deep learning and Pygame for game development, creating an interactive environment for the reinforcement learning agent.

2.1 Technological Components:

1. **PyTorch:** PyTorch is a popular deep learning framework that provides tools for building and training neural networks. In this project, PyTorch is used to implement the deep Q-learning algorithm for training the Snake-playing agent.

2. **Pygame:** Pygame is a cross-platform set of Python modules designed for video game development. It provides functionality for creating game windows, handling user input, and managing game states. Pygame is utilized to develop the Snake game environment and interface.

2.2 Project Components:

Snake Game Environment: The Snake game environment is created using Pygame, providing a grid-based playing area where the snake navigates to consume food and grow longer.

Deep Q-Learning Algorithm: The reinforcement learning agent is trained using a deep Q-learning algorithm implemented in PyTorch. The neural network model approximates the Q-function, helping the agent make decisions based on the current state of the game.

Experience Replay: To enhance learning stability, an experience replay mechanism is implemented, allowing the agent to learn from past experiences stored in a replay buffer.

Target Network: A target network is employed to stabilize the learning process by providing more consistent Q-value targets during training.

2.3 Process:

2.3.1 Implementation of the Game with Pygame:

The first part involves creating the game using the Pygame library. This likely includes setting up a game window, creating a grid-based environment for the Snake to navigate, defining the Snake's initial state, placing food, and handling user input for controlling the Snake's movements.

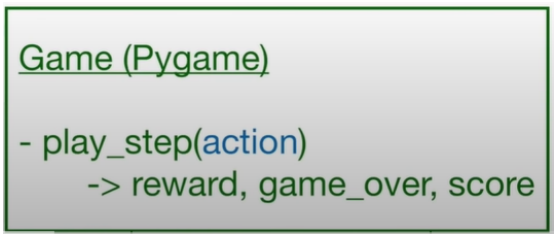
2.3.2 Implementation of the Agent:

After the game is set up, the next step is to implement the reinforcement learning agent. The agent is the entity responsible for making decisions within the game environment. It interacts with the game by selecting actions, receiving feedback in the form of rewards, and learning to improve its decision-making through reinforcement learning techniques.

2.3.3 Implementation of the Model with PyTorch:

Once the agent is defined, the third part involves implementing the underlying model using the PyTorch deep learning framework. This model is likely a neural network that approximates the Q-function, which helps the agent make decisions based on the current state of the game. The neural network is trained using reinforcement learning algorithms, such as deep Q-learning.

3 Game Loop and Interaction:

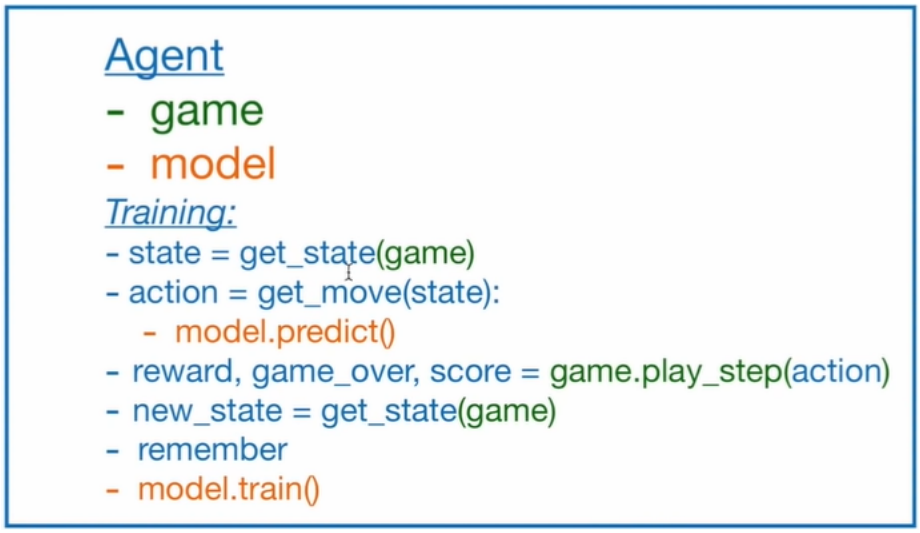


The final part involves integrating the game, agent, and model. A game loop is implemented, representing the main loop that runs the game. Within each iteration of the loop (a game step or playstep), the following sequence of actions takes place:

- The agent selects an action based on the current state of the game.
- The selected action is applied to the game, and the Snake is moved accordingly.
- The game returns information such as the current rewards obtained, whether the game is over or not (i.e., whether the Snake collided with itself or the game boundaries), and the current score.

4 Role of the agent:

The role of the agent in the reinforcement learning setup and describes the process of the training loop. Let's break down the key points:



- **Agent's Role:** The agent acts as the central component that connects and orchestrates interactions between the game environment and the neural network model. It needs to have knowledge about both the game and the model.
- **Agent's Knowledge:** The agent needs to be aware of the game details, such as the current state of the game, the actions it can take, and the rules governing the game dynamics. Additionally, it must be aware of the neural network model, which is responsible for predicting actions based on the current state.
- **Storage in the Agent:** Both the game and the model are stored within the agent. This implies that the agent encapsulates the necessary information and functionality to interface with both the game environment and the neural network model.
- **Training Loop:** The training loop is a crucial part of reinforcement learning. It involves iteratively interacting with the game, collecting experiences, and updating the model based on these experiences. The loop follows these steps:

  a. State Calculation: Based on the current state of the game, the agent calculates a state representation. This could involve extracting relevant features or observations from the game state

  b. Action Prediction: Using the neural network model, the agent predicts the next action to take in the current state. This is typically done by calling the predict method of the model

c. Next Gameplay Step: The predicted action is applied to the game environment, and a "next play step" is performed. This involves updating the game state based on the action taken.

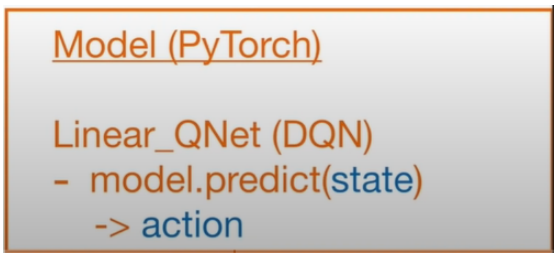
d. Reward and Feedback: The agent receives feedback from the game, including the reward obtained, information about whether the game is over, and the current score

e. New State Calculation: With the obtained feedback, the agent calculates a new state representation based on the updated game state.

f. Memory and Storage: The agent stores the information from this interaction, including the old state, the new state, the game over status, and the score. This information is stored in memory for later use in the training process.

This training loop repeats for multiple iterations, allowing the agent to learn and improve its decision-making capabilities over time. The stored experiences in the agent's memory are used for training the neural network model through methods like experience replay in order to enhance learning stability and efficiency.

5 Model:



5.1 Model Architecture:

The model used for this reinforcement learning task is called "linear qnet." It is described as a feedforward neural network with a few linear layers. A feedforward neural network is a type of artificial neural network where information flows in one direction, from input to output. The term "linear" suggests that the network comprises linear layers, typically followed by activation functions.

5.2 Input Information:

The model requires specific information as input to perform training. In this case, it needs both the new state and the old state as inputs. These states represent different snapshots or configurations of the game environment.

5.3 Training Process:

The training process involves updating the parameters (weights and biases) of the neural network based on the experiences collected during the training loop. These experiences include information about the old and new states, rewards obtained, and other relevant feedback from the game environment.

5.4 Calling model.predict:

During the training process, the model is called using the predict method. This method likely takes the current state as input and produces predictions for the next action to take. The predicted action is then used in the gameplay to influence the agent's decisions.

5.5 Training the Model:

The model is trained using a supervised learning approach. The training involves comparing the model's predicted actions with the actual actions taken during the gameplay. The goal is to minimize the difference between predicted and actual actions, updating the model's parameters to improve its performance.

5.6 Iterative Training:

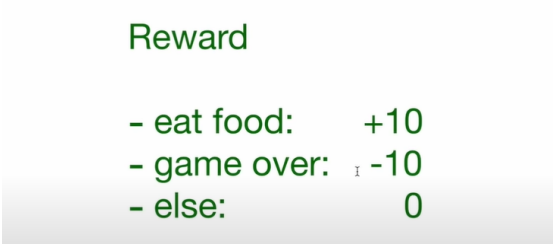
The training of the model is an iterative process that occurs over multiple training loops. Each loop involves playing the game, collecting experiences, updating the model, and refining the decision-making capabilities of the agent.
The "linear qnet" model is a feedforward neural network with linear layers used in a reinforcement learning setting. It is trained by providing information about the old and new states of the game environment, and the training process involves iteratively adjusting the model's parameters to improve its ability to predict actions and enhance the agent's overall performance in the game.

6 Variables

Delve into specific aspects of these variables, such as the nature of the actions, the intricacies of the states, or the significance of the rewards.

6.1 Rewards

The reward system is a crucial component in reinforcement learning. It provides a numerical signal to the agent (the Snake in this case) indicating the desirability of its actions.

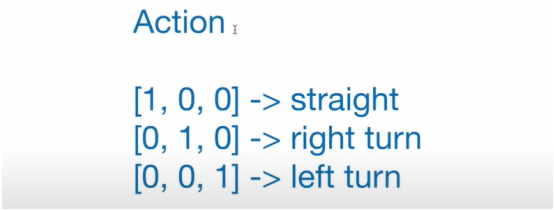


1. **Positive Rewards:** When the Snake successfully eats a food item, it receives a positive reward of +10. This encourages the agent to associate the action of consuming food with a favorable outcome, aiming to maximize its cumulative rewards over time.
2. **Negative Rewards:** In contrast, when the game ends (Game Over), such as when the Snake collides with itself or hits a boundary, a negative reward of -10 is assigned. This penalty discourages undesirable actions that lead to the premature termination of the game.
3. **Neutral Rewards:** For all other situations or states not covered by the positive or negative scenarios, the reward remains at 0. This includes the intermediary steps and movements of the Snake that do not directly result in food consumption or game termination.

The reward system is designed to reinforce desired behaviors (eating food) and discourage undesirable behaviors (collisions leading to game over) in the Snake game. It plays a pivotal role in shaping the learning process of the reinforcement learning agent.

6.2 Action

The choice of actions is carefully designed to avoid problematic scenarios:



- Traditionally, four actions are considered: left, right, up, and down.
- To prevent 180-degree turns, a more refined approach is adopted:
  - Use three distinct numbers to represent actions.
  - The choice of action depends on the snake's current direction.

  - For example:
    * 1 0 0: Stay in the current direction (go straight).
    * 0 1 0: Perform a right turn relative to the current direction.
    * 0 0 1: Perform a left turn relative to the current direction.

- This modified action representation eliminates the possibility of 180-degree turns.
- It also simplifies the task for the model, as it only needs to predict one of three possible actions.

6.3 State Calculation and Model Design

In the reinforcement learning setup for the Snake game, calculating the state is crucial as it informs the snake about relevant aspects of the game environment. The state is composed of 11 values, representing different pieces of information:

- **Danger Information:**
  - Danger straight
  - Danger ahead
  - Danger right
  - Danger left
- **Current Direction:** Left, right, up, or down
- **Food Information:**
  - Food left
  - Food right
  - Food up
  - Food down

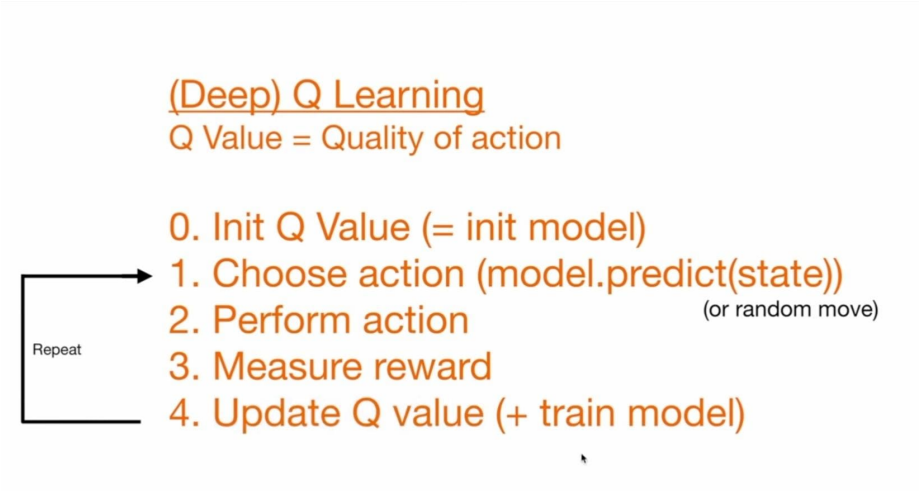
Each of these values is a boolean (0 or 1), providing essential information to the snake about the game environment.

Model Design

The model is designed as a feedforward neural network with an input layer, a hidden layer, and an output layer. The input layer has 11 neurons corresponding to the 11 values in the state. The choice of the hidden layer size is flexible. For the output layer, three neurons are needed since the model predicts one of three possible actions. These output values don't have to be probabilities; they can be raw numbers.

For example, if the output values are $1, 0, 0$, it corresponds to the action "straight," meaning the snake should keep its current direction.

Q-Learning and the Bellman Equation



To train the model, the reinforcement learning approach involves Q-learning, where Q stands for the quality of an action. The objective is to improve the Q values, representing the quality of the snake's actions.

Q-Value Initialization

The Q value is initialized by setting up the model with random parameters. Actions are chosen by calling `model.predict(state)`, sometimes incorporating random moves, especially in the early stages of training. The trade-off between exploration and exploitation becomes apparent later in the training loop.

Training Loop

The iterative training loop involves the following steps:

1. Choose an action using the model's predictions.
2. Perform the chosen action in the game, measuring the reward.
3. Update the Q value based on the obtained reward and the maximum expected future reward.
4. Train the model with the updated Q value.
5. Repeat the above steps.

The Bellman Equation

The Bellman equation provides a mathematical formula for updating the Q value:

$$Q_{\text{new}} = Q_{\text{current}} + \text{learning rate} \times \Big(\text{reward} + \text{gamma} \times \max_{\text{actions at new state}} Q_{\text{new state}}\Big)$$

Here, the key components are:

- **Learning rate:** A factor determining the extent to which new information overrides old information.
- **Gamma:** A discount factor that balances immediate and future rewards.

The loss function for training is then defined as the mean squared error:

$$\text{loss} = (Q_{\text{new}} - Q_{\text{current}})^2$$

This equation guides the optimization process during training.

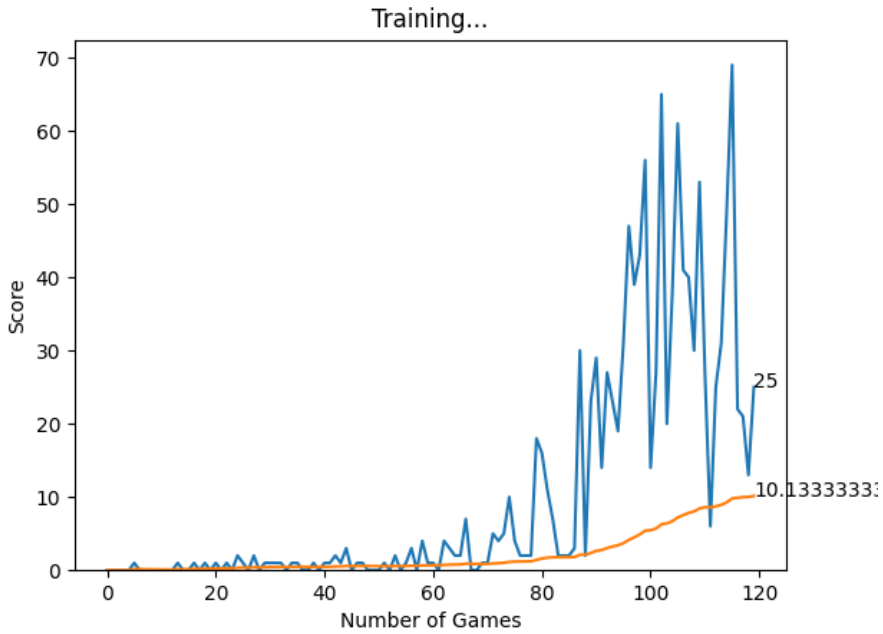
7 Conclusion

In conclusion, this project aimed to apply reinforcement learning techniques to train a Snake-playing agent. The key components of the project include defining a reward system, designing actions to guide the Snake's movements, calculating the state to inform the agent about the game environment, and implementing a Q-learning approach for model training.

The reward system served as a crucial feedback mechanism, reinforcing positive behaviors such as consuming food while penalizing undesirable actions leading to the game's premature termination. Actions were carefully designed to prevent 180-degree turns, ensuring more controlled and effective movement.

The state calculation involved gathering 11 pieces of information, including danger indicators, current direction, and food positions. This information was used to feed a feedforward neural network model, allowing the agent to make decisions based on the game state.

The Q-learning training loop iteratively improved the agent's decision-making abilities. Exploration and exploitation were balanced, and the Bellman equation guided the update of Q values, providing a mathematical foundation for the reinforcement learning process.



As the iterations progressed, we observed a consistent improvement in both the individual scores and the mean score. After conducting approximately 120 games, the agent attained its highest score of 68, showcasing a notable advancement in its performance. Furthermore, the mean score surpassed 10, indicating a steady enhancement in the overall gameplay proficiency over the course of the training iterations.

The project highlights the significance of combining game development with deep reinforcement learning, providing a hands-on exploration of concepts such as Q-learning, neural network training, and the trade-off between exploration and exploitation.

Future work could involve refining the model architecture, exploring hyperparameter tuning, or extending the project to tackle more complex games. Overall, this project serves as a foundation for understanding and implementing reinforcement learning in gaming scenarios.