SmartED InnovationsDeepSnake: An Autonomous Gaming Agent

short line

Om Koradiya  
22th September, 2025

short line

# **TABLE OF CONTENTS**

| Section | Page Number |
| --- | --- |
| ABSTRACT | 2 |
| INTRODUCTION | 3 |
| OBJECTIVE | 4 |
| METHODOLOGY | 5 |
| CODE AND IMPLEMENTATION DETAILS | 10 |
| RESULT AND OBSERVATIONS | 16 |
| CONCLUSIONS | 18 |

# 

# 

# 

# ABSTRACT

In this project, a deep reinforcement learning approach was used to train an AI agent to play the classic Snake game. The primary objective was to move beyond traditional rule-based programming and instead create a system capable of autonomous learning. The core of the project is a **Deep Q-Network (DQN)**, a neural network built with **PyTorch**, which serves as the agent's "brain." This agent learns to navigate a custom-built game environment created using the **Pygame** library. The agent's learning process is driven by a simple yet effective reward system: it receives a positive reward for eating food, a negative penalty for game over, and a neutral reward for every other move.

The AI's learning curve was a central focus of the project. Through a continuous loop of playing, learning, and updating its knowledge, the agent was able to progress from making random moves to developing complex strategies for survival and scoring high. The training process utilized a replay memory to store past experiences, which significantly improved the efficiency and stability of the learning. The model's updates were governed by the **Bellman equation**, with the **Mean Squared Error** serving as the loss function to optimize its performance. The project successfully resulted in a functional AI agent that not only mastered the game but also demonstrated a clear and measurable learning curve. This report details the full end-to-end process, from the initial game design to the final training and evaluation of the AI model.

# **INTRODUCTION**

This project is a deep dive into the world of Deep Reinforcement Learning (DRL), where a computer program learns through trial and error, just like a human would. Instead of relying on a pre-programmed rule set, an intelligent agent was created to learn to play the classic Snake game on its own. The goal was to build a system that not only plays the game but actually gets better at it over time, figuring out the best strategies to achieve a high score. This report will detail the entire process, from setting up the game environment to designing and training the deep neural network that acts as the agent's brain.

# 

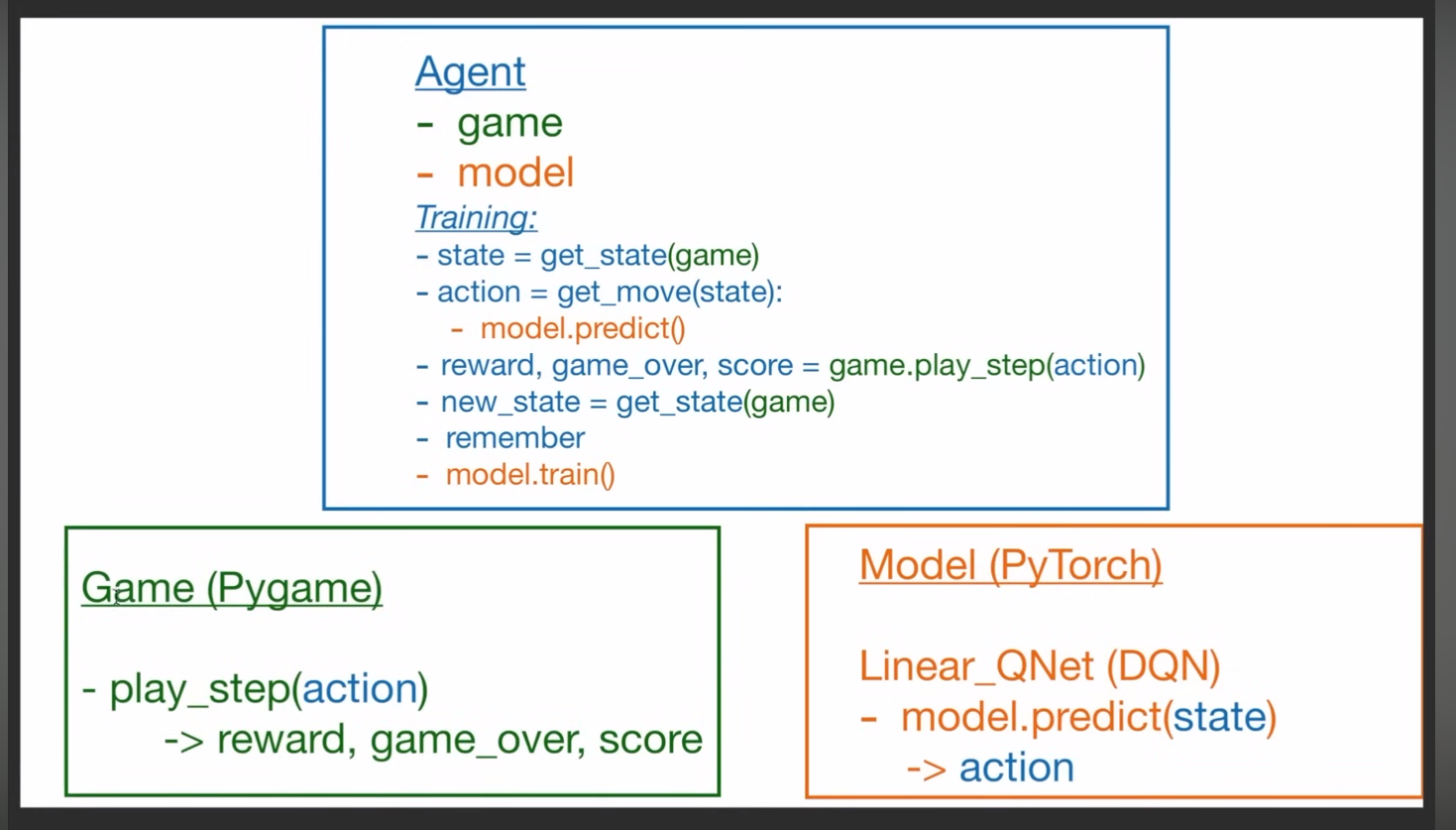
# OBJECTIVE

My main goals for this project were :

* **Develop a Game Environment**: Build the classic Snake game using the **Pygame** library, establishing all the game's rules and visual interface.
* **Architect an AI Model**: Design and implement a neural network using **PyTorch** to serve as the AI agent's "brain."
* **Train the Agent**: Create a deep reinforcement learning pipeline to train the AI. This process allows the agent to learn the best strategies through experience.
* **Visualize the Learning Process**: Integrate a live plotting system to track the agent's performance, showing its score and average score over time to demonstrate its learning.

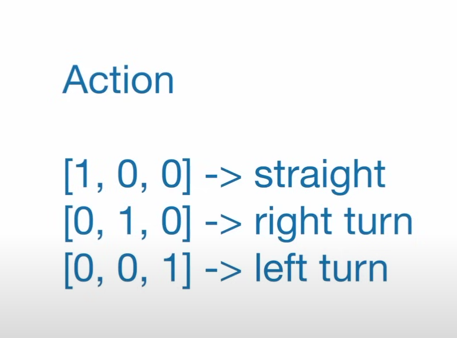
# METHODOLOGY

My project’s approach was centered on a deep reinforcement learning (DRL) workflow. I moved away from hard-coding game logic and instead designed a system where an AI agent could learn the optimal strategy for playing the Snake game entirely on its own. This process was broken down into several distinct and systematic stages.



The first step was to build the game environment itself. I used the popular **Pygame** library in Python to create the playing field, the snake, and the food. This part of the project defined the physical rules and boundaries, including how the snake moves, grows, and how a game-over event is triggered.

Next, I focused on building the "brain" of my AI agent. For the agent to make intelligent decisions, it needed a way to understand the game state. I defined a state representation using a simple but effective 11-value array. This array provided the agent with crucial information, such as immediate dangers (walls or its own body) in the straight, left, and right directions, the current direction of movement, and the location of the food relative to the snake's head. The agent's actions were limited to three possibilities: turning right, turning left, or going straight. The agent’s goal was to maximize its score, so I designed a simple reward system. The rewards and penalties were straightforward: the agent received a large positive reward (+10) for eating food, a significant negative penalty (−10) for a game-over event, and a neutral reward of zero for every other move.

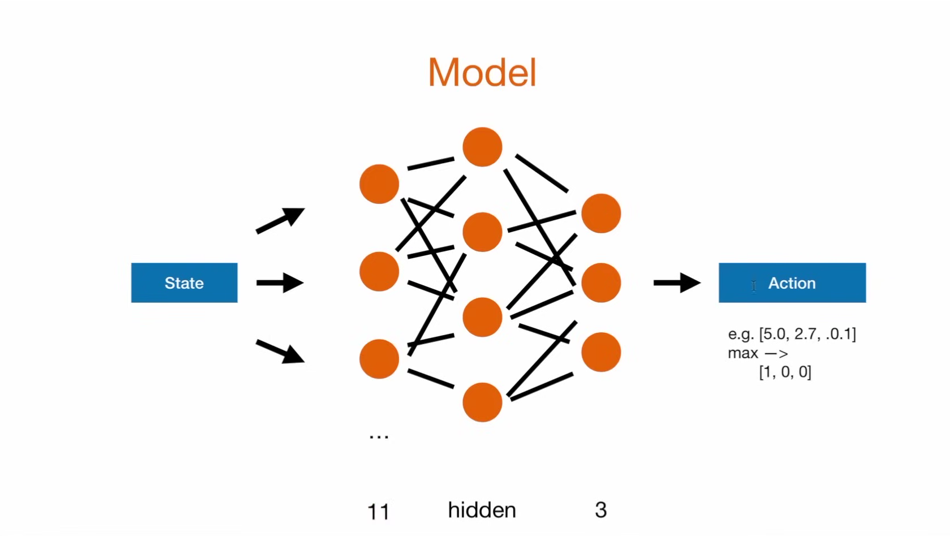


# 

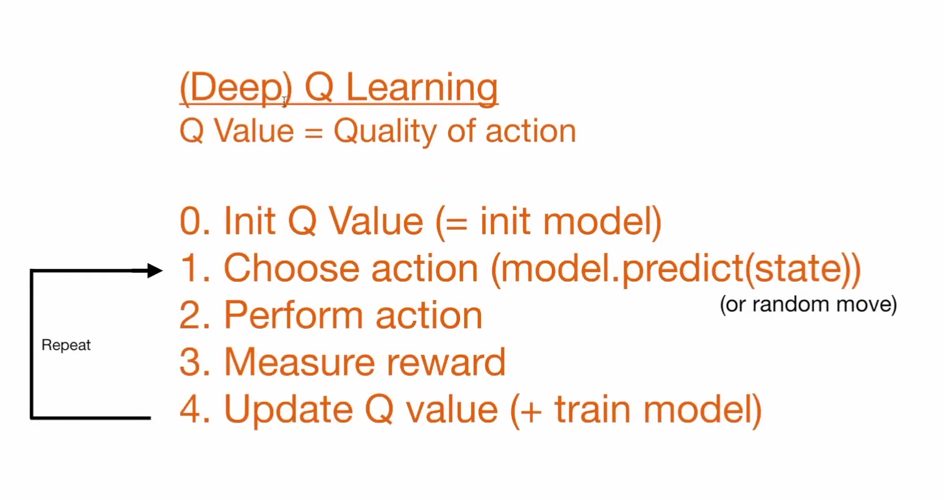
# 

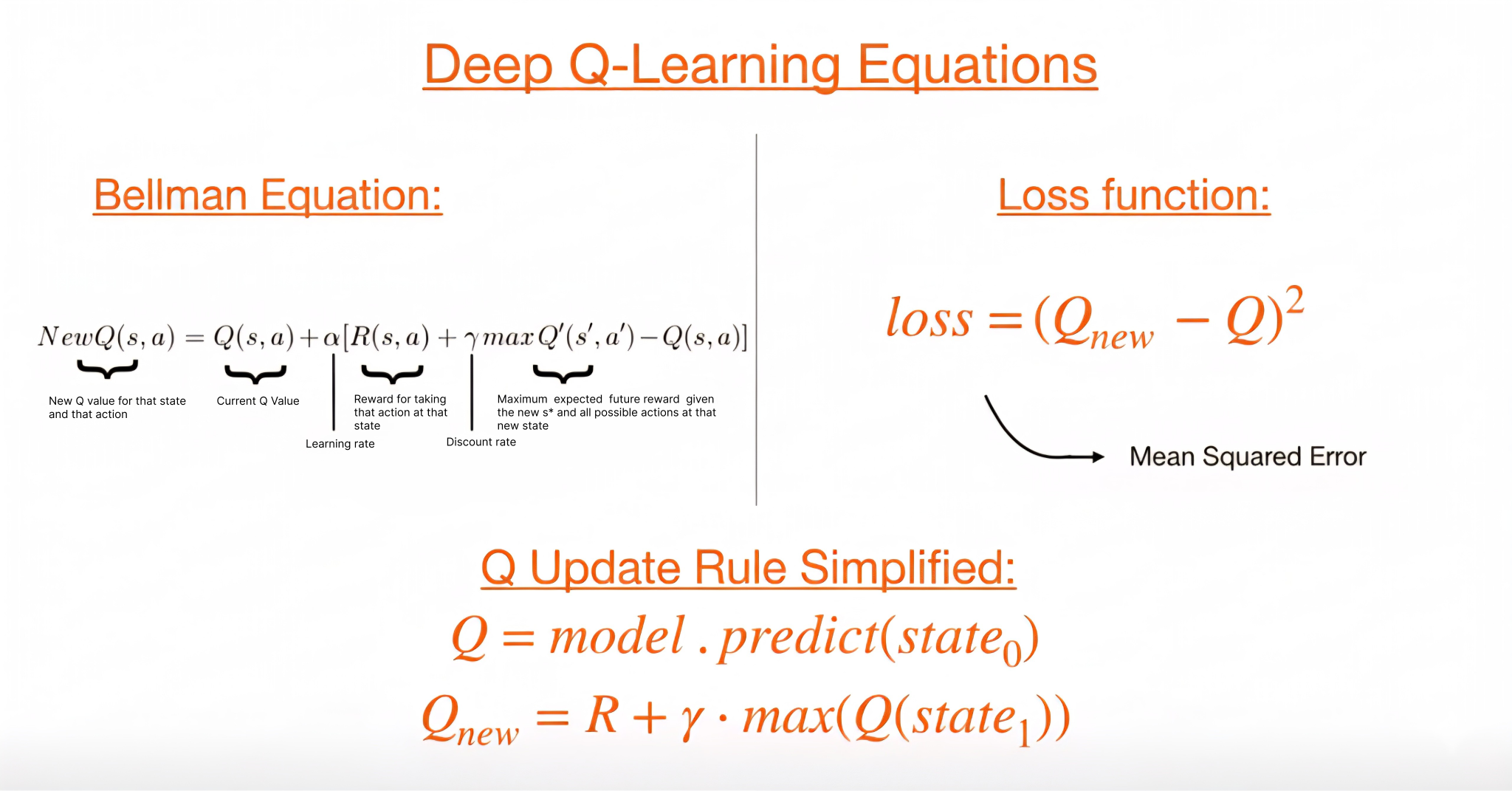


The core of the project was the deep learning model itself. I implemented a neural network with PyTorch to act as a Deep Q-Network (DQN). This model’s job was to take the 11-value state as input and output a prediction for the "quality" of each of the three possible actions. By training this network, the agent learned which actions were most likely to lead to a high score in the long run.



Finally, I built the training pipeline, which is where all the components came together. The agent played the game, using a strategy that balanced exploration (making random moves) and exploitation (using the trained model to make the best possible move). I used a technique called "experience replay," where the agent's actions and outcomes were stored in a memory buffer. This allowed the model to train on past experiences, making the learning process much more stable and efficient. The model’s training process was guided by the **Bellman equation**, and the updates were made using the **Mean Squared Error** as the loss function to minimize the difference between the predicted and actual Q-values.





# CODE AND IMPLEMENTATION DETAILS

To give a clear look into the project's technical core, this section highlights key code snippets that were instrumental in the development process. These examples show how the game environment was set up, how the AI's "brain" was built, how the learning agent was defined, and how the training was visualized.

**1. Game Environment Setup (from game.py)**

This snippet shows how I used the **Pygame** library to set up the game environment. The SnakeGameAI class is the foundation of the project. In the \_\_init\_\_ method, it initializes the display window with specific dimensions, sets the window's title, and creates a pygame.time.Clock object to control the game's speed. The BLOCK\_SIZE and SPEED constants are also defined here, which are crucial for the game's physics and visual layout. The Direction and Point classes are essential for managing the snake's movement and position, making the code clean and easy to read.

| **import pygame import random from enum import Enum from collections import namedtuple import numpy as np  pygame.init() font = pygame.font.Font('arial.ttf', 25)  class Direction(Enum):  RIGHT = 1  LEFT = 2  UP = 3  DOWN = 4  Point = namedtuple('Point', 'x, y')  # rgb colors WHITE = (255, 255, 255) RED = (220, 50, 50)  BLUE1 = (30, 100, 255)  BLUE2 = (100, 180, 255)  BLACK = (0, 0, 0)  BLOCK\_SIZE = 20 SPEED = 200  class SnakeGameAI:  def \_\_init\_\_(self, w=640, h=480):  self.w = w  self.h = h  # init display  self.display = pygame.display.set\_mode((self.w, self.h))  pygame.display.set\_caption('Snake AI')  self.clock = pygame.time.Clock()  self.reset()** |
| --- |

**2. The AI's Brain: Deep Q-Network (from model.py)**

The Linear\_QNet class represents the neural network that serves as the AI's "brain." This model, built with **PyTorch**, takes the game state as input and outputs the predicted quality of each possible action (straight, right, or left). The \_\_init\_\_ method sets up two fully-connected linear layers (linear1 and linear2). The forward method defines the network's architecture: the input x passes through the first linear layer, then a **ReLU** activation function is applied to introduce non-linearity, and finally, it goes through the second linear layer to produce the output. This simple feed-forward design is perfect for the agent's task of learning to map states to optimal actions.

| **import torch import torch.nn as nn import torch.nn.functional as F  class Linear\_QNet(nn.Module):  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):  super().\_\_init\_\_()  self.linear1 = nn.Linear(input\_size, hidden\_size)  self.linear2 = nn.Linear(hidden\_size, output\_size)   def forward(self, x):  x = F.relu(self.linear1(x))  x = self.linear2(x)  return x** |
| --- |

**3. The Agent's Decision-Making Logic (from agent.py)**

This snippet from agent.py shows the get\_state method, which is how the agent "perceives" the game environment. This code is crucial because it converts the complex game state into a simplified, 11-value array that the neural network can easily process. The state is represented as a list of boolean values (0s and 1s) that check for three categories:

* **Danger:** It checks if there is a collision risk directly in front, to the right, or to the left of the snake's head, based on its current direction.
* **Direction:** It indicates the snake's current direction of travel.
* **Food Location:** It specifies whether the food is to the left, right, up, or down from the snake's head.

This simplified representation is a vital part of the deep reinforcement learning process, as it provides the model with only the most important information needed to make a decision.

| def get\_state(self, game):  head = game.snake[0]  point\_l = Point(head.x - 20, head.y)  point\_r = Point(head.x + 20, head.y)  point\_u = Point(head.x, head.y - 20)  point\_d = Point(head.x, head.y + 20)    dir\_l = game.direction == Direction.LEFT  dir\_r = game.direction == Direction.RIGHT  dir\_u = game.direction == Direction.UP  dir\_d = game.direction == Direction.DOWN   state = [  # Danger straight  (dir\_r and game.is\_collision(point\_r)) or   (dir\_l and game.is\_collision(point\_l)) or   (dir\_u and game.is\_collision(point\_u)) or   (dir\_d and game.is\_collision(point\_d)),  # Danger right  (dir\_u and game.is\_collision(point\_r)) or   (dir\_d and game.is\_collision(point\_l)) or   (dir\_l and game.is\_collision(point\_u)) or   (dir\_r and game.is\_collision(point\_d)),  # Danger left  (dir\_d and game.is\_collision(point\_r)) or   (dir\_u and game.is\_collision(point\_l)) or   (dir\_r and game.is\_collision(point\_u)) or   (dir\_l and game.is\_collision(point\_d)),  # Move direction  dir\_l, dir\_r, dir\_u, dir\_d,  # Food location   game.food.x < game.head.x, # food left  game.food.x > game.head.x, # food right  game.food.y < game.head.y, # food up  game.food.y > game.head.y # food down  ]   return np.array(state, dtype=int) |
| --- |

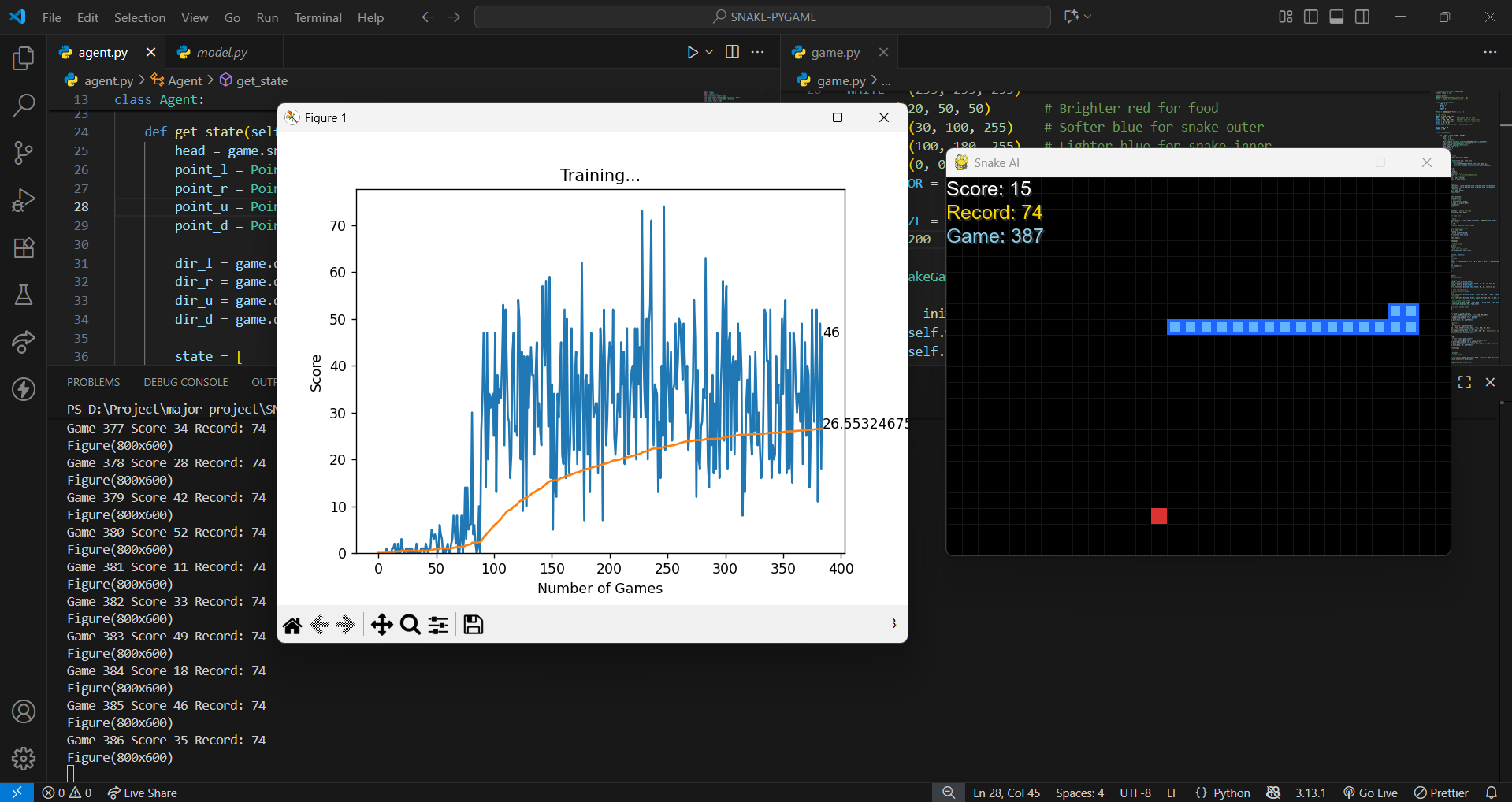
**4. Visualization and Plotting (from helper.py)**

This final snippet from helper.py shows the plot function, which is responsible for visualizing the training process. This function uses the matplotlib library to generate and update a live graph. The graph displays two key metrics: the raw scores from each game and the running mean score. This was a crucial tool for understanding the agent's learning curve and confirming that the AI was improving its performance over thousands of games.

| import matplotlib.pyplot as plt from IPython import display  plt.ion()  def plot(scores, mean\_scores):  display.clear\_output(wait=True)  display.display(plt.gcf())  plt.clf()  plt.title('Training...')  plt.xlabel('Number of Games')  plt.ylabel('Score')  plt.plot(scores)  plt.plot(mean\_scores)  plt.ylim(ymin=0)  plt.text(len(scores)-1, scores[-1], str(scores[-1]))  plt.text(len(mean\_scores)-1, mean\_scores[-1], str(mean\_scores[-1]))  plt.show(block=False)  plt.pause(.1) |
| --- |

# RESULT AND OBSERVATIONS

This section presents the project's outcome and the observations made during the AI's training and gameplay. The most compelling evidence of the agent's success is its ability to learn and improve its performance over time. The results are best shown through a single image that captures both the trained agent in action and a visualization of its learning curve.



For a live demonstration of the DeepSnake AI agent in action, please refer to the project execution video available on Google Drive: [SnakeAgent project video](https://drive.google.com/drive/folders/12DzmhDN4CGdigvh1A0xc60gRVaY86Xo4?usp=sharing).

The image shows the AI agent mid-game with a live score of **15**, while a live training graph is displayed alongside it. The console in the background confirms that the agent has played over **400 games** and achieved a record score of **74**. The training graph is a key indicator of the AI’s learning. The blue line represents the raw score from each individual game, showing a lot of variation. The smooth orange line represents the running mean score. As the AI played more games, the mean score steadily increased, confirming that the AI is successfully learning and getting better at the game.

The final result is a functional and impressive AI that can play the Snake game at a level far beyond what was possible with simple, rule-based programming.

The project's code can be found on my GitHub repository: [Om-koradiya/DeepSnake-An-Autonomous-Gaming-Agent](https://github.com/Om-koradiya/DeepSnake-An-Autonomous-Gaming-Agent).

# CONCLUSIONS

This project has been an enriching experience of going through the entire life cycle of a data science application. From a basic concept, I was able to implement a viable AI agent for the game Snake using the method of deep reinforcement learning. I have had first-hand experience in crafting a total end-to-end system, ranging from developing a tailored game environment using Pygame to constructing and training a deep neural network with PyTorch. The most satisfying aspect of this process was seeing the model learn from experience, eventually developing into an accomplished player well beyond what a basic rule-based program could have done.

In the future, I think there are a number of compelling areas for continued work. My top priority would be to carry out large-scale hyperparameter tuning on the model in order to enhance its predictive capabilities and overall accuracy. To give the agent richer and more detailed context, my follow-up action would be to merge in further data into the state representation, e.g., details about the tail of the snake or a more advanced treatment of the game board. Finally, I would like to experiment with more advanced model architectures to determine whether they could identify more subtle, non-linear patterns in the game that the existing model may not detect.

short dash