**Prompt**

*If you need to use ML to train and enhance the logic of this game so it will maximize the score, how will you do it*

**Reasoned about ML to enhance game play for 6 seconds (WTF!!)**

**Integrating ML to Maximize Score**

OK, let's imagine using machine learning to turbocharge game logic! I'm thinking about reinforcement learning—tools like Q-learning or DQNs. These methods could help the AI figure out the best strategy to boost the score, balancing snake length and trap challenges.

**Mapping the strategy**

I’m focusing on reinforcement learning. The game model includes the board as a state, actions, and rewards like eating apples or hitting traps. We’ll also define a clear reward function.

**Setting the course**

I'm piecing together a reward system with bonuses for apples and penalties for crashes. Training involves deep reinforcement learning techniques to fine-tune the agent and enhance its strategy.

**Piecing together the strategy**

OK, let's piece together using reinforcement learning to boost game logic. Training a deep Q-network (DQN) with a small state space seems ideal.

Using machine learning (ML) to train an agent that maximizes the score in this Snake game means turning the game into a reinforcement learning (RL) problem. In an RL framework, the agent (the snake) learns by interacting with an environment (the game) and receiving rewards or penalties for its actions. Here’s an outline of how you could approach this:

**1. Define the Environment**

Before training an RL agent, you must formalize the game as an environment. This includes:

* **State Representation:**  
  Determine how the game state will be represented for the ML model. For example:
  + **Grid-based Representation:** A 20×20 matrix where each cell encodes information (e.g., snake body parts, apple, traps, walls).
  + **Feature-based Representation:** A vector containing key features such as the snake’s head position, relative position of the apple, distances to the nearest obstacles, current snake length, number of traps, etc.
* **Actions:**  
  The agent’s possible actions are the directional moves (UP, DOWN, LEFT, RIGHT).
* **Reward Function:**  
  Design a reward signal that encourages the desired behavior. For example:
  + **Positive Reward:** For eating an apple (with higher rewards as the snake gets longer or as traps increase).
  + **Negative Reward:** For colliding with itself, hitting a trap, or hitting a wall.
  + **Step Penalty:** A small penalty per move to encourage efficiency (optional).
* **Episode Termination:**  
  The game (or episode) ends when the snake collides with itself, a wall, or possibly when some maximum number of moves is reached.

A common approach is to implement your environment in the style of an OpenAI Gym environment. This makes it easier to interface with many RL algorithms available in libraries like Stable Baselines3, RLlib, or custom implementations using TensorFlow or PyTorch.

**2. Choose an RL Algorithm**

Depending on the complexity of your state representation and how “continuous” your problem is, you might choose among several algorithms:

* **Deep Q-Network (DQN):**  
  If you represent the state as a fixed-size array or image, you can use DQN. The network takes the state as input and outputs Q-values for each action. The agent selects the action with the highest Q-value, and the network is trained using the Bellman equation.
* **Policy Gradient Methods:**  
  Methods like REINFORCE, Actor-Critic, or Proximal Policy Optimization (PPO) learn a policy directly. These methods can be more stable in environments with continuous action spaces or when the reward signal is sparse or noisy.
* **Other Advanced Methods:**  
  Depending on your requirements and computational resources, methods like Advantage Actor-Critic (A2C) or even model-based RL could be considered.

For a game like Snake with a discrete state and action space, DQN or PPO are both popular choices.

**3. Training the Agent**

**a. Implement the Environment:**

* Create a class (e.g., SnakeEnv) that encapsulates the game logic. This class should have methods like reset(), step(action), and render().
* The step() function should return:
  + The new state,
  + The reward for the taken action,
  + A boolean flag indicating whether the episode is done,
  + Additional info (if needed).

**b. Create the Neural Network Model:**

* If using DQN, design a neural network that takes the state (either the raw grid or a feature vector) and outputs Q-values for the four actions.
* For image-like inputs (the 20×20 grid), you can use convolutional layers; for a feature vector, fully connected layers might be sufficient.

**c. Training Loop:**

* **Experience Replay:** Store transitions (state, action, reward, next state, done) in a replay buffer.
* **Target Network:** For DQN, maintain a target network to stabilize training.
* **Epsilon-Greedy Exploration:** Start with a higher exploration rate (epsilon) and gradually reduce it so that the agent explores the state space before exploiting learned behavior.
* **Loss Calculation:** Use the temporal-difference (TD) error for DQN or a policy gradient loss for policy-based methods.
* **Backpropagation:** Use an optimizer (e.g., Adam) to update the network weights.

You might set up your training loop as follows (pseudo-code for a DQN approach):

env = SnakeEnv()

dqn = DQNModel() # Your neural network

replay\_buffer = ReplayBuffer(capacity=10000)

epsilon = 1.0 # Exploration factor

for episode in range(num\_episodes):

state = env.reset()

done = False

while not done:

# Epsilon-greedy action selection

if random.random() < epsilon:

action = random.choice([UP, DOWN, LEFT, RIGHT])

else:

q\_values = dqn.predict(state)

action = argmax(q\_values)

next\_state, reward, done, info = env.step(action)

replay\_buffer.add(state, action, reward, next\_state, done)

# Sample mini-batch and perform gradient descent on the DQN loss

if replay\_buffer.size() > batch\_size:

batch = replay\_buffer.sample(batch\_size)

loss = dqn.train(batch)

state = next\_state

# Update target network and decrease epsilon

if episode % target\_update\_frequency == 0:

dqn.update\_target\_network()

epsilon = max(epsilon\_min, epsilon \* epsilon\_decay)

For policy gradient methods, the structure would be similar, but you’d accumulate rewards and update the policy network directly according to the gradient of the expected return

**4. Evaluating and Integrating the Trained Agent**

Once the agent is trained:

* **Evaluation:**  
  Run the agent in your game environment (without exploration noise) to assess its performance. Measure metrics such as the average score over many episodes.
* **Integration:**  
  Replace the current rule-based AI logic in your game with the trained ML agent. Every game step, feed the current state into your model and use the agent’s predicted action.
* **Fine-Tuning:**  
  You may need to adjust the reward function or training parameters if the agent finds unintended “shortcuts” or if the game dynamics (such as traps appearing or apple scoring) change.

**5. Additional Considerations**

* **Simulation Time:**  
  Training an RL agent may require many episodes (potentially millions of steps) to learn effective strategies, especially if the reward signal is sparse or if the game dynamics are complex.
* **Compute Resources:**  
  Deep RL can be computationally intensive. Using GPUs or cloud resources can speed up training.
* **Safety Constraints:**  
  If you want to prevent risky behavior during training (e.g., the snake running into traps deliberately), you can shape the reward function to heavily penalize such actions.
* **Exploration vs. Exploitation:**  
  Balancing exploration and exploitation is crucial. You may also experiment with more advanced exploration techniques (e.g., curiosity-driven exploration).

**Conclusion**

By turning the Snake game into an RL environment and training an agent with algorithms such as DQN or PPO, you can enhance the game’s logic so that the snake learns to maximize its score—even as the difficulty increases with a longer snake and more traps. The process involves carefully designing the state representation, action space, and reward function, then training a neural network model via reinforcement learning. Once trained, the agent can replace the heuristic-based AI logic with one that has learned to optimize the cumulative reward (score) over many game episodes.

o3-mini-high

**Prompt**

*Ok implement this plan*

**Reasoned following the RL plan for a SnakeEnv (couple of minutes):**

**Crafting RL training for a snake game**

I'm working on defining the snake game as an OpenAI Gym environment. Then, I'll use RL training, likely with DQN or PPO.

**Setting up snake environment**

I’m defining the snake game as an OpenAI Gym environment. The observation space will likely be a matrix or an array, encoding the grid with values for empty, snake body, fruit, and traps.

**Crafting the snake game**

I’m developing the snake game environment, including action and observation spaces. The observation space might be a matrix, a flattened array, or a feature vector for objects.

**Mapping the grid**

Thinking through the grid representation, snake movements, and trap appearances. Rewards for apple eating, self or wall hits, and step penalties are also considered.

**Mapping player inputs**

I'm noting that RL training should avoid overcomplicating with time-based dynamics, like traps, to keep inputs numeric and manageable.

**Setting up the environment**

I'm crafting a Snake game environment, defining grid dimensions, starting snake length and direction, apple placement, and traps every 'n' steps for RL training with DQN.

**Strategizing rewards and penalties**

I’m thinking through rewards for apples, penalties for game over, and step penalties. The state representation might involve a matrix or channels for snake, apple, and trap.

**Creating grid states**

I’m opting for a 2D integer array to represent the snake game state: 0 for empty, 1 for snake, 2 for apple, and 3 for trap. This simplifies the process.

**Deciding on game mechanics**

I’m thinking about the snake’s response to eating apples or hitting traps, including growth, collision penalties, and game-over scenarios. This makes me think the choices are solid.

**Clarifying game ending rules**

I’m ensuring clarity on when the game ends—whether the snake hits a trap, wall, or its own body, and the corresponding feedback sounds and actions.

**Evaluating collision responses**

I’m examining how different collision scenarios affect the snake: traps reduce its length with a crash sound, while walls or self-collisions end the game.

**Adjusting collision rules**

I'm refining collision rules: game over when hitting walls or self; length reduction and crash sound on trap collision, without sound in the RL training environment.

**Clarifying collision effects**

I'm aligning game mechanics with RL training implications: game over for self or wall collisions, length halved for trap collisions, and crash sounds ignored.

**Clarifying collision rules**

Trap collisions may not end the game, but they should incur a penalty. This contrasts with wall or self-collisions, which do end the game.

**Revisiting collision rules**

I’m mapping out the collision rules: length halved, crash sound for traps, game over for self-hit. This feedback loop ensures the snake’s behavior aligns with expected outcomes.

**Refining collision responses**

OK, let me see. Colliding with the snake's body or wall ends the episode. A trap collision reduces the snake's length without ending the game.

**Clarifying trap collision rules**

OK, I’m thinking through trap collisions: they halve the snake's length and generate a crash sound, but don't end the game. Self or wall collisions, though, do end it.

**Rewarding the snake**

I’m piecing together a reward system: positive for apples, negative for traps and collisions, and a subtle penalty for each step to keep the game engaging.

**Mapping out movement**

I'm thinking through the snake's direction mapping and ensuring reversing into itself is discouraged. The game will update the snake's position, handle collisions, and manage rewards accordingly.

**Piecing together observations**

I’m adding traps to the grid, adjusting the snake's size upon collision, and setting up direction vectors and initial conditions for consistent state representation.

**Retracing steps**

I’m restarting the render code to integrate the Snake game with Pygame, keeping the RL loop separate. Adding text-based rendering for versatility, similar to the last approach.

**Crafting custom environment**

I’m setting up a unique Snake game environment for RL training, using Gym for spaces, then training the DQN model with stable-baselines3. Progress is steady, focusing on increasingly complex tasks.

**Crafting the foundation**

I’m shaping the SnakeEnv class, setting up action and observation spaces. Implementing reset and \_get\_random\_free\_position ensures the environment's grid is initialized and updated correctly.

**Navigating the grid**

I’m detailing the step function: updating the snake's direction, managing collisions with boundaries and its body, growing or reducing the snake's length when eating an apple or hitting a trap, and refreshing the apple's position.

**Enhancing the environment**

I’m finalizing grid rendering with Pygame for visual feedback. Considering randomizing apple colors for variety, though it might complicate RL training observables.

**Mastering environment rendering**

I’m setting up Pygame to depict cells and possibly scores, ensuring visuals are clear and score details are accurate. This setup will ease adding training code with stable-baselines3.

**Crafting and testing the model**

I’m working on a DQN model for SnakeEnv, incorporating detailed steps for training and evaluation, but ditching the sound files for RL.

**Charting the course**

I’m planning to define the environment and training script in a single file. It will clearly state the imports and use numpy for necessary functions.

**Mapping the environment**

I’m setting up the SnakeEnv environment with gym, numpy, and pygame imports. Constants for grid dimensions and cell size are also defined.

**Mapping the environment**

I’m defining the SnakeEnv class, detailing grid dimensions, apple placements, snake movements, trap intervals, and sectors like action-to-direction and tracing collisions with built-in methods.

Below is an end‐to‐end example that turns the Snake game into a Gym‑compatible environment and then uses a deep Q‑network (DQN) from Stable Baselines3 to train an agent to maximize the score. In this example:

* The environment is defined as a subclass of gym.Env with a discrete 20×20 grid.
* The snake (a list of grid positions) moves based on one of four actions (UP, DOWN, LEFT, RIGHT).
* An apple is placed on a free cell; when eaten the snake grows and the agent receives a reward that increases with the snake’s current length and the number of traps.
* Traps are added every few steps; when the snake’s head lands on a trap its length is cut in half and a penalty is applied.
* Collisions with walls or with the snake’s own body are terminal (ending the episode with a heavy penalty).
* A simple rendering method (using Pygame) lets you visually inspect the agent’s play (if you choose to render during training or evaluation).

Note:  
• To run this code you’ll need to install the required packages:

*pip install gym stable-baselines3 pygame numpy*

• Training an RL agent may take a while (and you may need to adjust hyperparameters, the reward function, or the network architecture for better performance).

Below is the complete code. You can save it (for example, as snake\_rl.py) and run it.

import gym

import numpy as np

import random

from gym import spaces

import pygame

# --- Global Constants ---

GRID\_WIDTH = 20

GRID\_HEIGHT = 20

CELL\_SIZE = 20

# Colors (RGB)

COLOR\_BG = (0, 0, 0) # Black background

COLOR\_GRID = (40, 40, 40) # Dark gray grid lines

COLOR\_SNAKE = (0, 255, 0) # Green snake

COLOR\_TRAP = (128, 0, 128) # Purple traps

# Candidate apple colors (avoid green, purple, black, and grid color)

APPLE\_COLORS = [

(255, 0, 0), # Red

(255, 165, 0), # Orange

(255, 255, 0), # Yellow

(0, 0, 255), # Blue

(0, 255, 255), # Cyan

(255, 0, 255) # Magenta

]

# Map actions to directions.

# 0: UP, 1: DOWN, 2: LEFT, 3: RIGHT

ACTION\_TO\_DIRECTION = {

0: (0, -1),

1: (0, 1),

2: (-1, 0),

3: (1, 0)

}

# Utility: Get the opposite action (to prevent immediate reversal)

OPPOSITE\_ACTION = {0:1, 1:0, 2:3, 3:2}

# --- The Snake Environment ---

class SnakeEnv(gym.Env):

"""

Gym environment for Snake.

Observation:

A 2D numpy array (shape: GRID\_HEIGHT x GRID\_WIDTH) of integers:

0 = empty

1 = snake (any segment)

2 = apple

3 = trap

Action:

Discrete(4) – 0: UP, 1: DOWN, 2: LEFT, 3: RIGHT.

Reward:

• + (10 + len(snake) + number\_of\_traps) when eating an apple.

• -10 when hitting a trap (and the snake’s length is cut to half).

• -0.1 per normal move.

• -100 if the snake collides with the wall or itself (episode termination).

"""

metadata = {'render.modes': ['human', 'rgb\_array']}

def \_\_init\_\_(self):

super(SnakeEnv, self).\_\_init\_\_()

self.grid\_width = GRID\_WIDTH

self.grid\_height = GRID\_HEIGHT

# Define action and observation spaces.

self.action\_space = spaces.Discrete(4) # 4 possible directions.

# Observation: grid with values in {0,1,2,3}

self.observation\_space = spaces.Box(low=0, high=3,

shape=(self.grid\_height, self.grid\_width),

dtype=np.int8)

# How many steps between adding a new trap.

self.trap\_interval = 10

self.steps\_since\_last\_trap = 0

# Pygame rendering attributes.

self.window = None

self.clock = None

self.\_init\_pygame()

self.reset()

def \_init\_pygame(self):

"""Initialize Pygame for rendering."""

pygame.init()

self.window = pygame.display.set\_mode((self.grid\_width \* CELL\_SIZE,

self.grid\_height \* CELL\_SIZE))

pygame.display.set\_caption("Snake RL Environment")

self.clock = pygame.time.Clock()

def \_get\_random\_free\_position(self, occupied):

"""Return a random (x,y) not in the occupied set."""

while True:

pos = (random.randint(0, self.grid\_width - 1),

random.randint(0, self.grid\_height - 1))

if pos not in occupied:

return pos

def \_get\_observation(self):

"""Return the current grid state as a numpy array."""

obs = np.zeros((self.grid\_height, self.grid\_width), dtype=np.int8)

# Mark snake segments as 1.

for (x, y) in self.snake:

obs[y, x] = 1

# Mark the apple as 2.

ax, ay = self.apple

obs[ay, ax] = 2

# Mark traps as 3.

for (x, y) in self.traps:

obs[y, x] = 3

return obs

def reset(self):

"""Reset the environment state and return the initial observation."""

# Initialize snake at center with 3 segments.

mid\_x, mid\_y = self.grid\_width // 2, self.grid\_height // 2

self.snake = [

(mid\_x, mid\_y),

(mid\_x - 1, mid\_y),

(mid\_x - 2, mid\_y)

]

# Start moving to the right.

self.current\_direction = ACTION\_TO\_DIRECTION[3]

# Place the first apple.

occupied = set(self.snake)

self.apple = self.\_get\_random\_free\_position(occupied)

self.apple\_color = random.choice(APPLE\_COLORS)

self.traps = []

self.steps\_since\_last\_trap = 0

self.score = 0

self.done = False

return self.\_get\_observation()

def step(self, action):

"""

Execute one time step within the environment.

"""

# Prevent reversal if snake has length > 1.

if len(self.snake) > 1:

current\_action = self.\_direction\_to\_action(self.current\_direction)

if action == OPPOSITE\_ACTION[current\_action]:

action = current\_action

# Update current direction.

self.current\_direction = ACTION\_TO\_DIRECTION[action]

head\_x, head\_y = self.snake[0]

dx, dy = self.current\_direction

new\_head = (head\_x + dx, head\_y + dy)

# Check wall collision.

if not (0 <= new\_head[0] < self.grid\_width and 0 <= new\_head[1] < self.grid\_height):

self.done = True

reward = -100

return self.\_get\_observation(), reward, self.done, {}

# Check collision with self.

if new\_head in self.snake:

self.done = True

reward = -100

return self.\_get\_observation(), reward, self.done, {}

# Default reward per move.

reward = -0.1

# Apple eaten?

if new\_head == self.apple:

self.snake.insert(0, new\_head) # Grow snake.

apple\_reward = 10 + len(self.snake) + len(self.traps)

reward = apple\_reward

self.score += apple\_reward

# Place a new apple.

occupied = set(self.snake) | set(self.traps)

self.apple = self.\_get\_random\_free\_position(occupied)

self.apple\_color = random.choice(APPLE\_COLORS)

# Hit a trap?

elif new\_head in self.traps:

self.snake.insert(0, new\_head)

# Cut snake length to half (minimum length 1).

new\_length = max(1, len(self.snake) // 2)

self.snake = self.snake[:new\_length]

reward = -10

else:

# Normal move: advance the snake.

self.snake.insert(0, new\_head)

self.snake.pop()

# Update trap counter and add a trap if interval reached.

self.steps\_since\_last\_trap += 1

if self.steps\_since\_last\_trap >= self.trap\_interval:

occupied = set(self.snake) | set(self.traps) | {self.apple}

new\_trap = self.\_get\_random\_free\_position(occupied)

self.traps.append(new\_trap)

self.steps\_since\_last\_trap = 0

return self.\_get\_observation(), reward, self.done, {}

def \_direction\_to\_action(self, direction):

"""Convert a (dx, dy) tuple into an action number."""

for action, d in ACTION\_TO\_DIRECTION.items():

if d == direction:

return action

return 3 # Default to RIGHT.

def render(self, mode='human'):

"""Render the current state using Pygame."""

for event in pygame.event.get():

if event.type == pygame.QUIT:

self.close()

self.window.fill(COLOR\_BG)

# Optionally draw grid lines.

for x in range(self.grid\_width):

for y in range(self.grid\_height):

rect = pygame.Rect(x \* CELL\_SIZE, y \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

pygame.draw.rect(self.window, COLOR\_GRID, rect, 1)

# Draw traps.

for (x, y) in self.traps:

rect = pygame.Rect(x \* CELL\_SIZE, y \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

pygame.draw.rect(self.window, COLOR\_TRAP, rect)

# Draw the apple.

ax, ay = self.apple

rect = pygame.Rect(ax \* CELL\_SIZE, ay \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

pygame.draw.rect(self.window, self.apple\_color, rect)

# Draw the snake.

for (x, y) in self.snake:

rect = pygame.Rect(x \* CELL\_SIZE, y \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)

pygame.draw.rect(self.window, COLOR\_SNAKE, rect)

pygame.display.flip()

self.clock.tick(10) # Limit to 10 FPS.

def close(self):

pygame.quit()

# --- Training the RL Agent ---

#

# We use Stable Baselines3’s DQN to train an agent on our custom SnakeEnv.

#

# The training loop is handled by the library’s .learn() method.

#

if \_\_name\_\_ == "\_\_main\_\_":

# Create the environment.

env = SnakeEnv()

# (Optional) Check that the environment follows the Gym API.

# from stable\_baselines3.common.env\_checker import check\_env

# check\_env(env, warn=True)

# Import DQN from stable\_baselines3.

from stable\_baselines3 import DQN

# Create the DQN model using a multilayer perceptron (MLP) policy.

model = DQN("MlpPolicy", env, verbose=1)

# Train the model for a specified number of timesteps.

total\_timesteps = 100000 # Adjust as needed.

model.learn(total\_timesteps=total\_timesteps)

# Save the trained model.

model.save("dqn\_snake\_model")

print("Training complete. Now running an evaluation...")

# --- Evaluation Loop ---

obs = env.reset()

done = False

while not done:

# Predict an action using the trained model.

action, \_ = model.predict(obs, deterministic=True)

obs, reward, done, info = env.step(action)

env.render()

env.close()

**How This Works**

1. **Environment (SnakeEnv):**
   * The environment uses a grid (20×20) where each cell is encoded with a value:  
     • 0 = empty  
     • 1 = snake  
     • 2 = apple  
     • 3 = trap
   * The snake is initialized in the center and moves in one of four directions. Apples are placed in free cells with a random color chosen from a list. Every 10 steps a new trap is added.
   * The reward function gives a bonus for eating an apple (scaled by snake length and number of traps), a small step penalty, and a heavy penalty (–100) if the snake collides with a wall or itself (which ends the episode). Hitting a trap gives a –10 penalty and cuts the snake’s length by half.
2. **Rendering:**
   * A basic Pygame window draws the grid, traps (purple), the apple (in its random color), and the snake (green).
3. **Training with DQN:**
   * The environment is used with Stable Baselines3’s DQN algorithm (with an MLP policy). The model is trained for a specified number of timesteps and then saved.
   * An evaluation loop then runs the trained agent in the environment, rendering its behavior.

This implementation provides a complete framework for applying reinforcement learning to improve the snake’s gameplay. You can experiment with reward parameters, network architectures, or even switch to another RL algorithm (such as PPO) depending on your requirements. Happy training!

A screenshot of a computer

Description automatically generated

* *There appears to be a type casting error which I hand modified for action being an ndarray and thus cannot be a number and used for indexing.*

*See EPA fix at line 145 in step() function*

**Prompt**

*This appeared to train the model with print outs of episodes and results of training but after training is complete, I did not see anything for the evaluation. No output, not even a grid*

**Reasoned**

**Confirming Pygame execution**

OK, let me see. The user is confirming Pygame executed its job by training with episodes and results. However, the post-training evaluation loop didn't display any output or grid. The code aims to showcase Pygame's use for evaluating the model's final state, so it's noteworthy that the expected display wasn't observed.

**Spotting potential issues**

Hmm, let me see. The code shows that the Pygame window might be blocked or closed too quickly. It could also be hidden behind other windows or hindered by environment restrictions.

**Ensuring window visibility**

To tackle potential issues with the Pygame window, I’m incorporating user input handling and debugging outputs. This should maintain visibility and responsiveness, addressing current shortcomings effectively.

**Streamlining the evaluation**

I'm working through options to ensure the Pygame window stays visible, like adding a delay or awaiting a key press after evaluating. A pygame.time.wait(100) might also be useful.

**Enhancing evaluation visibility**

I’m suggesting a more dynamic loop to keep the Pygame window active, including a frame delay and a cue to press 'Q' to quit. This aims to enhance visibility and control.

Below is an updated version of the evaluation portion of the code. In some cases the evaluation loop may run very quickly or end as soon as the agent dies, causing the window to close before you have a chance to see anything. The following modifications will help:

1. **Frame Delay:**  
   We add an explicit delay (using pygame.time.wait) after each frame so that you can actually see the grid updating.
2. **Post-Evaluation Wait Loop:**  
   Once the evaluation episode finishes (or even if it runs continuously), a loop at the end will keep the Pygame window open until you manually close it.

Below is the revised evaluation section. (You can insert this after the training code in your script.)

# --- Evaluation Loop ---

obs = env.reset()

done = False

print("Starting evaluation. Close the window to exit.")

while not done:

# Predict an action using the trained model.

action, \_ = model.predict(obs, deterministic=True)

obs, reward, done, info = env.step(action)

env.render()

# Wait for a short time (in milliseconds) to slow down the display.

pygame.time.wait(100)

# Once the episode is done, keep the window open until the user closes it.

print("Evaluation complete. Press the window close button to exit.")

running = True

while running:

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

env.close()

**Explanation**

* **Frame Delay:**  
  The pygame.time.wait(100) call waits for 100 milliseconds after each frame. This slows down the evaluation so you can visually follow the agent's moves.
* **Post-Evaluation Wait Loop:**  
  After the episode ends (i.e. when done becomes True), the script enters another loop that simply checks for the QUIT event. This keeps the window open until you close it manually.

With these changes, you should see the grid (rendered via Pygame) during evaluation. If your agent dies quickly, you’ll still see the final state on the screen until you close the window.

Make sure you run the script in an environment where a GUI window can be created (i.e. not in a headless server unless you set up a virtual display).

**Prompt**

*After the training is complete and the evaluation was run, I finally saw the snake game rendered but only for a couple of seconds. How do I get the final model preserved and reusable by an actual snake game python script?*

**Reasoned for a few seconds to explain showcasing model deployment**

**Saving and integrating**

I’m saving the final model after training using model.save(). In another script, I load it with DQN.load() and use the agent's predict() for actions.

**Showcasing model deployment**

I’m detailing how to preserve the final model with Stable Baselines3's save method, load it, and integrate it into a snake game script using env.render() for predictions and rendering.

To preserve and reuse your final trained model in a separate snake game script, you need to:

1. **Save the Trained Model:**  
   When training is complete, use Stable Baselines3’s built‑in model.save() method to save the model (typically as a .zip file). For example, after training, you might have:

model.save("dqn\_snake\_model")

This command creates a file named dqn\_snake\_model.zip (and possibly some auxiliary files) that contains all the network parameters and training configuration.

1. **Load the Model in Another Script:**  
   In your separate snake game script, import the same RL algorithm class and load the saved model. For example:

from stable\_baselines3 import DQN

from **your\_snake\_env\_module** import SnakeEnv # Ensure your ***SnakeEnv*** is accessible

# Load the trained model.

model = DQN.load("dqn\_snake\_model")

# Create the environment.

env = SnakeEnv()

# Optionally, run the agent.

obs = env.reset()

done = False

while not done:

action, \_ = model.predict(obs, deterministic=True)

obs, reward, done, info = env.step(action)

env.render()

# Slow down if needed

pygame.time.wait(100)

env.close()

Replace **your\_snake\_env\_module** with the actual module name if you have separated your environment code into its own Python file.

1. **Integrate the Agent into Your Snake Game:**  
   Now you can incorporate this logic into an actual snake game script. This script could either run continuously or wait for user input (if you’re mixing human control with AI decisions). In a typical usage scenario, the AI agent uses its policy to decide the next move based on the current observation from the environment.
2. **Keep the Environment Consistent:**  
   Make sure that the environment (observation and action spaces) used during training is identical to the one in your final game script. If you have made changes (for example, different rendering or additional features), the model might not work as expected.

**Summary**

* **During Training:**  
  The model trains without rendering (or with minimal rendering) for performance reasons.
* **After Training:**  
  Save the model using model.save("dqn\_snake\_model").
* **In a Separate Script:**  
  Load the model with DQN.load("dqn\_snake\_model"), initialize your SnakeEnv, and then run an evaluation loop that uses the model’s predict method to control the snake. This lets you see the AI playing the game in real time.

By following these steps, you’ll have a final trained model preserved on disk and a separate, reusable Python script that uses it to run your snake game with the AI agent.

o3-mini-high