🧠 1. Model Logic (Deep Q-Network)

This is the AI brain that learns to play the game.

🔷 **Model Class** — `DQN`

```
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(11, 64),
            nn.ReLU(),
            nn.Linear(64, 4)   # Outputs Q-values for 4 actions
        )

    def forward(self, x): # makes predictions and moves on
        return self.fc(x)
```

```
def __init__(self):
```

This is the constructor method for the class. It's called automatically when an object of `DQN` is created.
This calls the constructor of the parent class `nn.Module`. (super(DQN, self).__init__())

First fully connected (dense) layer with:

```
nn.Linear(11, 64),
```

- `11` input features (i.e., state size = 11).
- `64` output features (hidden layer size).

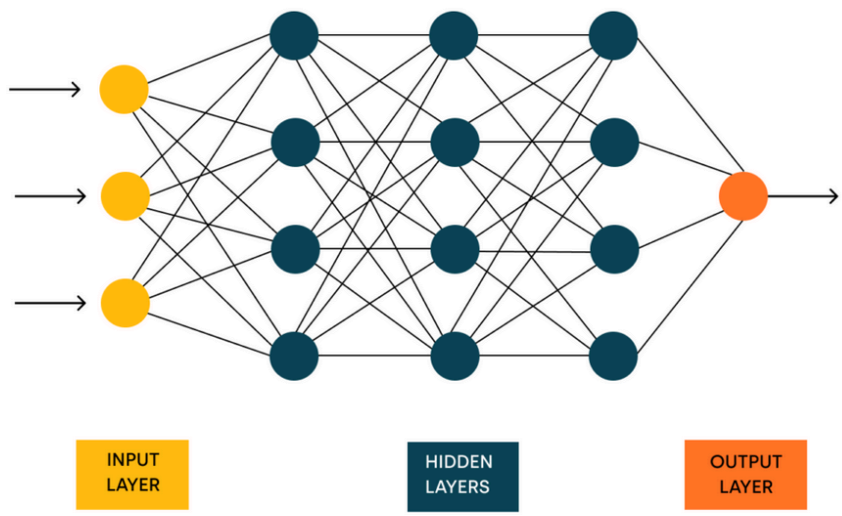
Activation function: **ReLU (Rectified Linear Unit)** adds non-linearity to the model, helping it learn complex patterns. [nn.ReLU()]

Second fully connected layer:

- Takes input from the hidden layer (64 features).
- Outputs 4 values → These represent the **Q-values for 4 possible actions** the agent can take.

nn.Linear(64, 4) # Outputs Q-values for 4 actions

Example of the node :



🔷 State Vector Calculation

```
def state_to_tensor(state):
    ...
    return torch.FloatTensor([
        food left, food right, food up, food down,
        direction flags,
        self collision,
        wall collision X, wall collision Y
    ])
```

4 binary values: Indicate the **relative position of food** compared to the snake's head.
Direction flags = `moving_left`, `moving_right`, `moving_up`, `moving_down`

`Self-Collison`

`A value indicating whether the` **`next step`** `would cause the snake to collide with its own body.`

- `1` if yes, `0` if safe.

Example Vector:

```
torch.FloatTensor([
    1, 0, 0, 0,          # food is to the left
    0, 1, 0, 0,          # moving right
    0,                   # no self collision
    0, 1                 # safe on X, wall ahead on Y
])
```

🔷 **Model Training (Q-Learning Update)**

```
for s, a, r, s_, d in batch:
    q_vals = model(state_to_tensor(s))
    target = q_vals.clone().detach()
    target[a] = r if d else r + 0.9 *
torch.max(model(state_to_tensor(s_))).item()

    loss = criterion(q_vals, target)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

✅ This is the **core of reinforcement learning**, where the model learns using:

- `reward`
- `discounted future reward` if not done
- `loss between predicted Q and target Q`

element in the batch is a tuple:

- `s`: current state
- `a`: action taken
- `r`: reward received
- `s_`: next state (after taking action `a`)
- `d`: done flag → `True` if the episode ended after this transition.

```
    target[a] = r if d else r + 0.9 *
torch.max(model(state_to_tensor(s_))).item()
```

It is a Bellman equation   DQN Main FORMULA

loss = criterion(q_vals, target)
Calculates the **loss** between:

- Predicted Q-values (`q_vals`
- and Target Q-values (`target`)

Remaining Lines : Backpropagate the loss and update the network

🎮 2. **Main Logic** (Game Loop, RL Interaction)

```
class SnakeEnv:
    def reset()
    def spawn_food()
    def step(action)
    def get_state()
```

✅ This simulates the environment. It:

- Handles movement
- Handles food and collisions
- Returns reward and new state

🔷 **Main Training Loop** — `train_one_episode()`
This starts one full game:

```
def train_one_episode():
    ...
    state = env.reset()
    ...
    def step_loop():
        ...
```

```
        next_state, reward, done = env.step(action)
        ...
        if done:
            scores.append(env.score)
            train_one_episode()  # Restart
        else:
            root.after(SPEED, step_loop)
```

state = env.reset()

Resets the game/environment to the initial state.
`state` is the first observation of the episode.
`env` is the environment (**snake_env**)

next_state, reward, done = env.step(action)

`next_state`: the new state after taking the action.
`reward`: reward for this step.
`done`: whether the game/episode has ended.

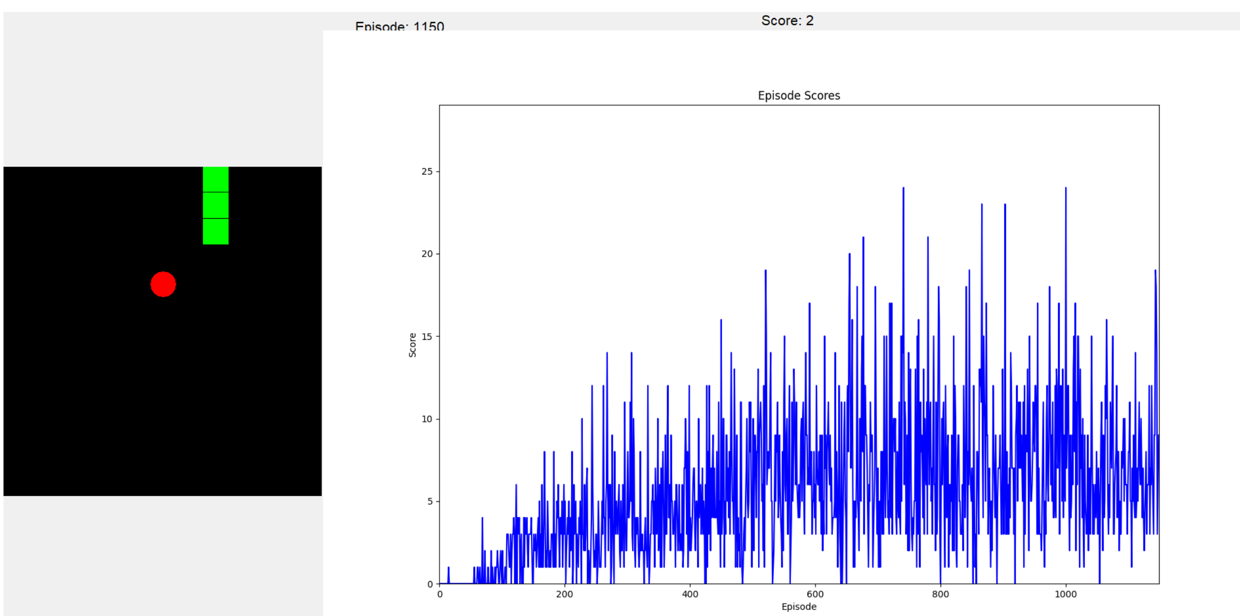
If done:
Saves the final score of the episode to a `scores` list for tracking performance over time.
Recursively calls `train_one_episode()` to **start a new episode** after game over.(continous)

Else:
If the game is not over, schedules the next call to `step_loop`
This keeps the step loop running at a constant speed using Tkinter’s timer function.

```
[Game State]
     ↓
┌─────────────────────────┐
│  state_to_tensor(state)│
└─────────────────────────┘
     ↓
  [Model (DQN)]
     ↓
Predict Q-values → Select Action
     ↓
  env.step(action)
     ↓
  New state, reward
     ↓
Store in memory & train model
     ↓
```

Here we have used DQN model = policy + target combined (simplified easy)
Policy network: Learns every batch
Target network: Slowly follows the policy, providing stable Q-value targets

By the end of 1150th Episode



Code :

```
from tkinter import *
import random
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt

# Constants
GAME_WIDTH = 500
GAME_HEIGHT = 500
SPACE_SIZE = 40
SPEED = 50
SNAKE_COLOR = "#00FF00"
FOOD_COLOR = "#FF0000"
BACKGROUND_COLOR = "#000000"


# ===============🧠 MODEL LOGIC===================
# DQN model
class DQN(nn.Module):
    def __init__(self):
        super(DQN, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(11, 64),
            nn.ReLU(),
            nn.Linear(64, 4)
        )

    def forward(self, x):
        return self.fc(x)

def state_to_tensor(state):
    head = state["snake_head"]
    food = state["food"]
    direction = state["direction"]
    body = state["snake_body"]
```

```
    return torch.FloatTensor([
        int(food[0] < head[0]),
        int(food[0] > head[0]),
        int(food[1] < head[1]),
        int(food[1] > head[1]),
        int(direction == "left"),
        int(direction == "right"),
        int(direction == "up"),
        int(direction == "down"),
        int(head in body),
        int(head[0] < 0 or head[0] >= GAME_WIDTH),
        int(head[1] < 0 or head[1] >= GAME_HEIGHT)
    ])
# ==========================================================================


#===================🎮 ENVIRONMENT LOGIC ==============================
class SnakeEnv:
    def __init__(self):
        self.reset()

    def reset(self):
        self.snake = [[SPACE_SIZE * 5, SPACE_SIZE * 5]]
        self.direction = "right"
        self.spawn_food()
        self.score = 0
        self.done = False
        return self.get_state()

    def spawn_food(self):
        while True:
            x = random.randint(0, (GAME_WIDTH // SPACE_SIZE) - 1) *
SPACE_SIZE
            y = random.randint(0, (GAME_HEIGHT // SPACE_SIZE) - 1) *
SPACE_SIZE
            if [x, y] not in self.snake:
                self.food = [x, y]
                break
```

```python
    def step(self, action):
        if action == 0 and self.direction != "down":
            self.direction = "up"
        elif action == 1 and self.direction != "up":
            self.direction = "down"
        elif action == 2 and self.direction != "right":
            self.direction = "left"
        elif action == 3 and self.direction != "left":
            self.direction = "right"

        x, y = self.snake[0]
        if self.direction == "up":
            y -= SPACE_SIZE
        elif self.direction == "down":
            y += SPACE_SIZE
        elif self.direction == "left":
            x -= SPACE_SIZE
        elif self.direction == "right":
            x += SPACE_SIZE

        new_head = [x, y]
        self.snake.insert(0, new_head)

        if x < 0 or x >= GAME_WIDTH or y < 0 or y >= GAME_HEIGHT or
new_head in self.snake[1:]:
            self.done = True
            return self.get_state(), -10, self.done

        if new_head == self.food:
            self.score += 1
            self.spawn_food()
            return self.get_state(), 10, self.done
        else:
            self.snake.pop()
            return self.get_state(), 0, self.done

    def get_state(self):
        return {
            "snake_head": self.snake[0],
            "snake_body": self.snake[1:],
```

```python
            "food": self.food,
            "direction": self.direction
        }

# ======================================================================

# RL Setup
env = SnakeEnv()
model = DQN()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.MSELoss()
memory = deque(maxlen=1000)
epsilon = 1.0
scores = []

# GUI Setup
root = Tk()
root.title("Snake AI with Live Score")

frame = Frame(root)
frame.pack(side=LEFT)

canvas = Canvas(frame, bg=BACKGROUND_COLOR, height=GAME_HEIGHT,
width=GAME_WIDTH)
canvas.pack()

score_label = Label(root, text="Score: 0", font=("Arial", 16))
score_label.pack()

episode_label = Label(root, text="Episode: 0", font=("Arial", 16))
episode_label.place(x=GAME_WIDTH + 50, y=10)



# Matplotlib Plot in Tkinter
fig, ax = plt.subplots(figsize=(5, 4))
score_plot, = ax.plot([], [], 'b-')
ax.set_title("Episode Scores")
ax.set_xlabel("Episode")
ax.set_ylabel("Score")
```

```python
ax.set_ylim(0, 30)

plot_canvas = FigureCanvasTkAgg(fig, master=root)
plot_canvas.get_tk_widget().pack(side=RIGHT, fill=BOTH, expand=1)
plot_canvas.draw()



# ================🚀 MAIN TRAINING LOOP======================
episode = 0
def train_one_episode():
    global epsilon, episode
    episode_label.config(text=f"Episode: {episode}")

    state = env.reset()
    total_reward = 0
    canvas.delete("all")
    episode += 1
    episode_label.config(text=f"Episode: {episode}")  # 👈 add this line


    def step_loop():
        global epsilon
        nonlocal state, total_reward

        if random.random() < epsilon:
            action = random.randint(0, 3)
        else:
            q_vals = model(state_to_tensor(state))
            action = torch.argmax(q_vals).item()

        next_state, reward, done = env.step(action)

        memory.append((state, action, reward, next_state, done))
        state = next_state
        total_reward += reward

        canvas.delete("snake")
        canvas.delete("food")
```

```
        for x, y in env.snake:
            canvas.create_rectangle(x, y, x + SPACE_SIZE, y + SPACE_SIZE,
fill=SNAKE_COLOR, tag="snake")

        fx, fy = env.food
        canvas.create_oval(fx, fy, fx + SPACE_SIZE, fy + SPACE_SIZE,
fill=FOOD_COLOR, tag="food")

        score_label.config(text=f"Score: {env.score}")

        if done:
            scores.append(env.score)
            update_plot()
            if len(memory) >= 64:
                batch = random.sample(memory, 64)
                for s, a, r, s_, d in batch:
                    q_vals = model(state_to_tensor(s))
                    target = q_vals.clone().detach()
                    target[a] = r if d else r + 0.9 *
torch.max(model(state_to_tensor(s_))).item()
                    loss = criterion(q_vals, target)
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()
            epsilon = max(0.1, epsilon * 0.995)
            root.after(500, train_one_episode)  # Start next episode
        else:
            root.after(SPEED, step_loop)

    step_loop()

# ================================================================

def update_plot():
    score_plot.set_data(range(len(scores)), scores)
    ax.set_xlim(0, max(10, len(scores)))
    ax.set_ylim(0, max(10, max(scores) + 5))
    plot_canvas.draw()

# Start training visually
```

```
train_one_episode()
root.mainloop()
```

🧠 DQN Learning Steps

1. Initialize the environment and Q-network with random weights.
2. Observe the initial state from the game environment.
3. Choose an action using ε-greedy policy (explore or exploit using Q-values).
4. Execute the action, observe reward and next state from the environment.
5. Store the experience (state, action, reward, next_state, done) in replay memory.
6. Sample a random mini-batch from the replay memory for training.
7. **Compute target Q-values using the Bellman equation:** `r + γ * max(Q(s', a'))`.
8. Update the network by minimizing the loss between predicted and target Q-values.
9. Decay epsilon to reduce exploration over time.
10. Repeat the process until the episode ends, then start a new episode.