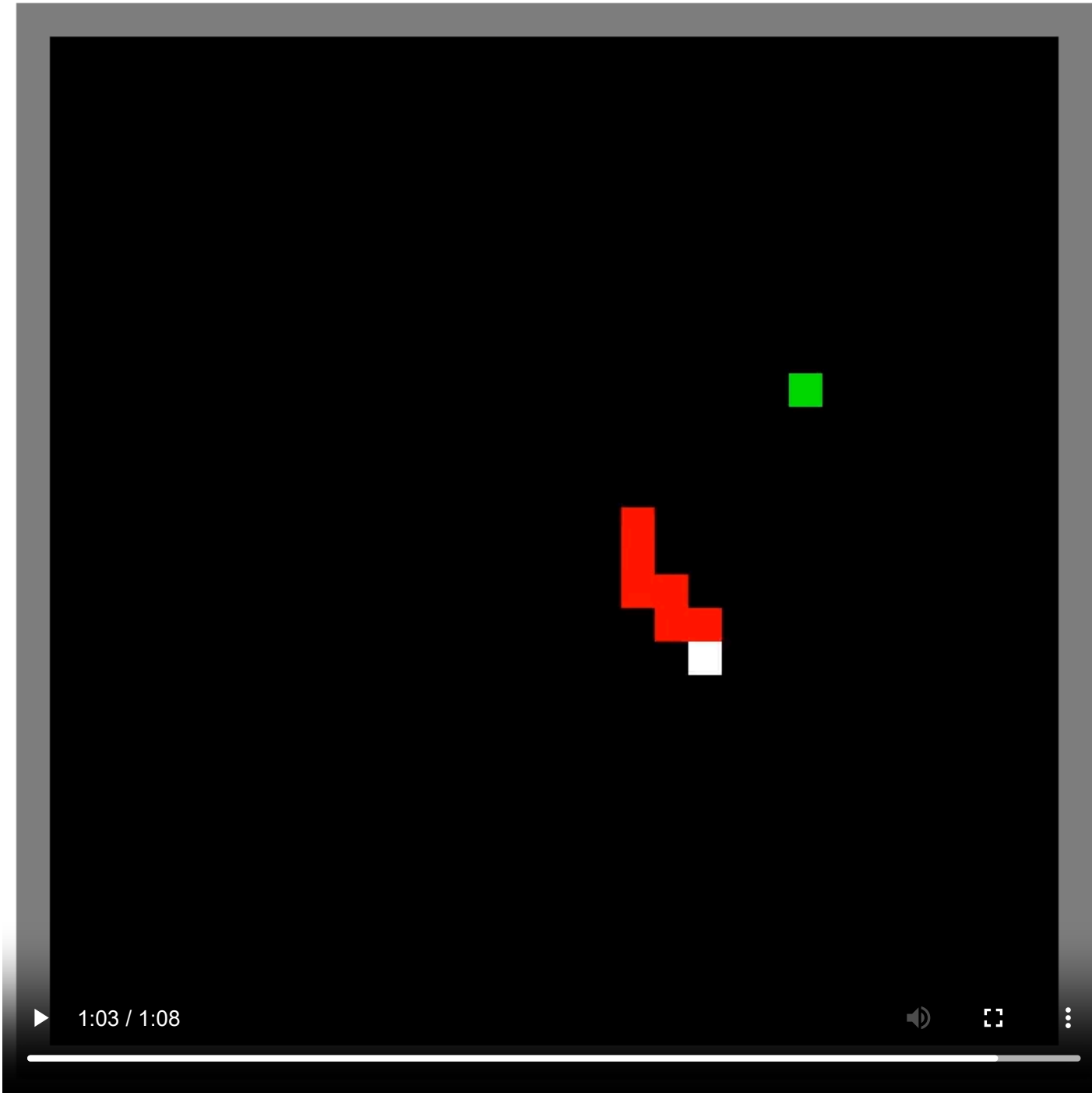
Assignment #2

Dates	segunda-feira → 6 de junho de 2025
Type	📌 Assignment
Topics	Vazio
After class	Vazio
Before class	Vazio
Files and media	Environment.zip
Resources	Vazio
Status	In Progress
Summary	Vazio

Reinforcement Learning agent to play Snake

For this assignment, you need to train a deep neural network to play the snake game by observing images that represent the board state. As a suggestion, you can aim for a 30x30 board with a 1-pixel border. In this case, the board image will be a 32x32 RGB image. The video below shows ten games played by an agent trained for this game, with a maximum of 1000 steps per game.



Objective

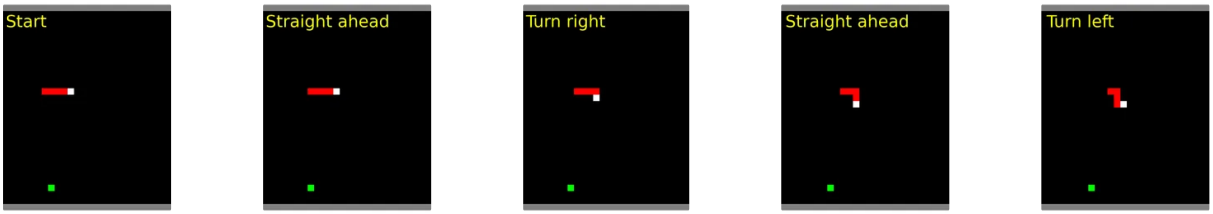
The main goal of this assignment is to provide you with hands-on experience in reinforcement learning. Your focus should be on experimenting with different scenarios, testing various approaches, and understanding what aids the agent's learning and how deep reinforcement learning works. Although the scenario shown above (a 30x30 board with one food, no grass, and a one-tile border) is a suitable starting point, you are encouraged to try different scenarios as well.

The game and scenarios

The `SnakeGame` class has two methods you can use to run a game: `reset()` to start a new game and `step(action)` to make a move. The `action` parameter provided in the `step(action)` method should be one of the following options:

- **-1** Turn the snake to the left.
- **0** Keep moving in the same direction.
- **1** Turn the snake to the right.

Note that the effect of an action depends on the direction in which the snake is already moving. To see how the game works and how you can save images from it, refer to the file `snake_demo.py`. The images below show examples of these actions:



When creating a game object you can also choose different scenarios using the following parameters:

```
def __init__(self, width, # width of free region of the board height, # he
ight of the free region of the board food_amount=1, # number of food items
present in the board border = 0, # gray border added outside the board gra
ss_growth = 0, # how much the grass grows per step in the board max_grass
= 0 # maximum amount of grass per location
```

When the snake head enters a cell, it receives a reward equal to the amount of grass present in that cell. This can be useful for creating a vegetarian snake. To encourage exploration, you can set the amount of grass to 0 after it is eaten, using a small amount such as 0.05 of the maximum grass.

To grow the grass slowly, you can use the `grass_growth` parameter (e.g. 0.001).

By adjusting the border and size of the free region of the board, you can create simpler scenarios. For instance:

```
game = SnakeGame(width=14, height = 14, border = 1)
```

The game will have a board with 14x14 free locations surrounded by a gray border of 1 cell, resulting in a 16x16 image. This size, being a power of two, is practical for convolution and pooling.

Since training the agent can take some time (minutes or hours), you may opt for smaller boards or variants of the game. For example, here is an agent trained to play the game with 0.05 of maximum grass.

The `step(action)` method returns the following four values:

- The state of the board after the action is taken, represented by a matrix with the image of the board.
- The reward for the last action and state transition.
- A flag with a value of True if the state reached is a terminal state (the snake died), or False otherwise.
- A dictionary with the total score so far.

The `reset()` method returns the same values, but the board state is the initial state and the reward is zero.

Using better examples with an heuristic policy

Playing the game randomly may not result in many good examples for the agent to learn. To improve this, you can use a set of experiences taken from games played by other agents. For instance, gathering many games played by humans could enrich the training data for your agent.

However, it may not be practical to gather so many games played by humans. Instead, you can write your own code to play the game following some heuristic. For example, the video below shows some games being played by a heuristic that tries to go straight for the food. Although this is often suicide, it's better than playing randomly.

If you want to enhance your pool of examples with some heuristic, you can use the `get_state()` method of the snake game object to cheat and get the coordinates of the snake, its direction, and the coordinates of the food items. The values returned by `get_state()` are:

- The current total score.
- A list with the coordinates of the food sources (the list has only one item for the default of only one piece of food at a time).
- The coordinates of the head of the snake.
- A list with the coordinates for the tail segments of the snake.
- The current direction of movement, an integer from 0 through 3 indicating, in order, N, E, S, W.

Note that this information should not be used by your agent. **The agent must learn solely from the images depicting the board state**. However, you can use this information to generate examples to start the training.

Deep Q-Network

Deep Q-Network (DQN) receives a state s as input and outputs the Q-value Q(s, a) for every action a. This allows the network to act as a regular table, where you can look up the Q-value using the state as the input key, and observe the produced values as outputs for every action. Typically, the DQN is trained using experience replay and a target network.

Exploration strategies

The tradeoff between exploration and exploitation has been extensively studied in the literature. Many strategies exist. We have discussed ϵ-**Greedy** where the chosen action a is the maximizer of que Q-value function with probability $1 - \epsilon$ or a random action taken uniformly from the set $\mathcal{A}$ of possible actions:

$$a(s) = \begin{cases} \arg\max_{a \in \mathcal{A}} Q_\pi(a, s) & \text{with probability } 1 - \epsilon \\ \text{randomUniform}(\mathcal{A}) & \text{with probability } \epsilon \end{cases}$$

The probability of exploration ϵ is, in general, decreased according to a decay pattern we saw in class. But there are other possibilities. For example, you could investigate other relevant distributions, namely depending on the current state s, or the states already seen so far. Or you could try a constant ϵ throughout training. You should implement, test and analyze two exploration strategies.

Boltzmann Exploration assigns a probability to each possible action based on its estimated value, promoting a stochastic selection mechanism that favors higher-value actions while still allowing for exploration of lesser-known options. The probability $P(a_i)$ of selecting action a_i is calculated using the softmax function

$$P(a_i) = \frac{\exp(Q(a_i)/T)}{\sum_j \exp(Q(a_j)/T)},$$

where $Q(a_i)$ is the estimated value (expected return) of action a_i, and T is a temperature parameter that controls the level of exploration. The temperature parameter T plays an important role in determining the exploration-exploitation balance. If T is high a more uniform probability distribution across actions, encouraging exploration. All actions have nearly equal chances of being selected, regardless of their Q-values.

Experience Replay

Maintaining a replay buffer allows us to reuse collected data multiple times. Additionally, sampling batches randomly from the buffer breaks the correlation between consecutive data, which can make training more stable. Implementing the replay buffer enables experience replaying.

In RL, the distinction between on-policy (like SARSA) and off-policy (like Q-learning) methods affects replay buffer design. On-policy methods learn from experiences generated by the current policy, so a smaller buffer (1,000-3,000) with recent experiences works best. Off-policy methods like the DQN can learn from any past experiences regardless of which policy generated them, allowing for larger buffers (5,000-10,000) that provide more diverse training data and improve stability. Nevertheless, if the experience was generated by a very different policy, it might not provide useful learning signals or could even hinder performance. Maintaining a balance between buffer size and recency of experiences is important for effective learning.

Target network

During the update process, we attempt to push the predicted Q values towards the target Q values, which are the immediate reward plus the bootstrapped Q values. Since a single update affects the entire network, it causes the non-stationarity of the target Q values. To make the target Q values stationary, we can use another network to provide the target Q values. We can periodically update this network using the weights of the original network. This is known as the target network.

Tasks

1. Basic DQN Implementation without Experience Replay and Target network

Implement a basic Deep Q-Network (DQN) without using Experience Replay or Target Network. This implementation should focus on the core DQN architecture and Q-learning loss function.

1.1 Set up the neural network architecture

Design and implement a neural network architecture suitable for processing the game's state images and outputting Q-values for each possible action. Consider using convolutional layers to process the image input effectively. Note that a smaller network will be more stable to train.

1.2 Define and implement the Q-learning loss function

This is a critical component of the DQN implementation. Create a loss function that computes the difference between the predicted Q-values and the target Q-values calculated using the Bellman equation:
Loss = $(r + \gamma * \max(Q(s', a')) - Q(s, a))^2$
where r is the immediate reward, γ is the discount factor, Q(s', a') represents the Q-values for the next state, and Q(s, a) represents the current predicted Q-value. This loss function drives the learning process by pushing the predicted Q-values toward their target values.

1.3 Implement the training loop and evaluation

Create a training loop that allows the agent to interact with the environment, collect experiences, and update the network using the defined loss function. Test the implementation by running experiments to establish a Q-Learning baseline performance, documenting the agent's learning progress and any challenges encountered.

1.4 Implement an heuristic policy

Implement a heuristic policy that helps the agent learn more effectively by providing better examples than random play. The heuristic should make reasonable decisions about snake movement based on food location, avoiding collisions with walls and itself. You can implement this by using the game's state information (through the get_state() method) to create a policy that moves the snake toward food while avoiding obstacles, as explained above, in ¶ **Using better examples with an heuristic policy**. Document the heuristic baseline performance without RL.

- **Resource Constraint:** Training must not exceed 4 hours.
- **Evaluation:** Performance (score and Q-value) and efficiency (score / training time, Q-value / training time) baseline measurements.

Deliverables:

- Code implementation.
- A brief presentation detailing:
  - The neural network architecture and its effectiveness for the task.
  - Implementation details of the Q-learning loss function.
  - Results of training experiments and baseline performance metrics.
  - Heuristic policy algorithm and its performance compared to random play.

2. Enhanced DQN with Experience Replay and Target Network

- Improve the DQN of Task 1 with:
  - Experience Replay Buffer (size ≤ 10,000).
  - Target Network (updated around every 100 steps).

2.1 Experience Replay Buffer Implementation

The experience replay buffer stores transitions (state, action, reward, next_state, done) to break the correlation between consecutive samples and improve training stability. Here are key considerations for implementing an effective replay buffer:

- **Buffer Structure:** Use a data structure that allows efficient storage and sampling of experiences. A circular buffer or deque with a maximum size works well.
- **Transition Storage:** Store each transition as a tuple of (state, action, reward, next_state, done) where:
  - state: The current state (image of the board)
  - action: The action taken
  - reward: The reward received
  - next_state: The resulting state after taking the action
  - done: Boolean indicating if the episode terminated
- **Memory Management:** When the buffer reaches capacity, replace old experiences with new ones (FIFO approach). Check the deque data structure in Python.
- **On-Policy and Off-Policy training and the size of the Replay Buffer:** test different replay buffer capacities, for example, {1, 100, 10000}.
- **Batch Sampling:** Implement random sampling to pull a batch of transitions for training. This randomization helps break temporal correlations.
- **Prioritized Experience Replay (Optional):** Consider implementing prioritized sampling where transitions with higher TD errors are sampled more frequently.

Temporal Difference (TD) errors represent the difference between the predicted value of a state-action pair and the actual observed value during reinforcement learning. Mathematically, the TD error is defined as:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

- r is the immediate reward received
- γ is the discount factor
- Q(s', a') is the estimated value of the next state-action pair
- Q(s, a) is the estimated value of the current state-action pair

In Prioritized Experience Replay, transitions with larger TD errors are considered more "surprising" or informative, as they represent experiences where the agent's predictions were furthest from reality. By sampling these high-error transitions more frequently during training, the agent can focus learning on the most informative experiences, potentially accelerating the learning process and improving overall performance.

The sampling probability for each transition can be determined using the formula:

$$P(i) = \frac{|\delta_i|^\alpha}{\sum_j |\delta_j|^\alpha}$$

Where α is a hyperparameter that determines how much prioritization is used (α = 0 corresponds to uniform sampling).

Here is a base implementation of the Replay Buffer:

```
class ReplayBuffer: def __init__(self, capacity=10000): self.buffer =
collections.deque(maxlen=capacity) def add(self, state, action, reward,
next_state, done): # Add experience to buffer self.buffer.append((state,
action, reward, next_state, done)) def sample(self, batch_size): # Random
sampling of experiences experiences = random.sample(self.buffer,
min(batch_size, len(self.buffer))) # Separate the tuple into batches
states, actions, rewards, next_states, dones = zip(*experiences) # Convert
to appropriate tensor format return (np.array(states), np.array(actions),
np.array(rewards), np.array(next_states), np.array(dones)) def
__len__(self): return len(self.buffer)
```

2.2 Integration with DQN Training

After implementing the replay buffer, integrate it into your training loop with these steps:

- **Initial Population:** Collect initial experiences (using your epsilon-greedy exploration) to populate the buffer before starting training. Also try the heuristic policy to warm-start the Replay Buffer.
- **Experience Collection:** For each step in the environment:
  - Select an action using your exploration strategy
  - Execute the action and observe the reward, next state, and done flag
  - Store this transition in the replay buffer
- **Training:** After each step (or periodically):
  - Sample a random batch from the replay buffer
  - Compute target Q-values using the target network
  - Update the primary network using the calculated loss
- **Efficiency Tips:**
  - Use batching
  - Normalize state inputs to stabilize training
  - Warm start using the heuristic policy

2.3 Target Network Implementation

The target network is a copy of the main Q-network that provides stable Q-value targets during training:

- **Initialization:** Create an identical copy of your main DQN network.
- **Update Strategy:** Update the target network every 100 steps as specified, using either:
  - Hard update: Copy all weights directly
  - Soft update: Gradually update using a small τ value (e.g., τ = 0.01), where target_weights = τ*online_weights + (1-τ)*target_weights
- **Target Q-Value Calculation:** Use the target network to compute the max Q-value for the next state when calculating the TD target.
- **Resource Constraint:** Training must not exceed 4 hours.
- **Evaluation:** Performance (score and Q-value) and efficiency (score / training time, Q-value / training time) improvements.

Deliverables:

- Code implementation.
- A comparative analysis presentation, improving the previous presentation, plus highlighting:
  - Implementation details of Experience Replay, including buffer size experiments and their impact on performance.
  - Target Network update strategies tested and their effects on training stability.
  - Comparative analysis showing improvements in both performance metrics and training efficiency over the basic DQN.
  - Visualizations of learning curves demonstrating the stabilizing effects of these enhancements.

3. Exploration Strategies

Implement and compare two different exploration strategies for your DQN agent:

- **Epsilon-Greedy:** Implement the standard ε-greedy strategy with a decaying epsilon parameter. Experiment with different initial epsilon values and decay rates to find an optimal balance between exploration and exploitation. Document how different decay schedules affect learning performance.
- **Boltzmann Exploration:** Implement temperature-based Boltzmann exploration using the softmax function as described earlier in the assignment. Test different temperature schedules and analyze how they affect exploration patterns compared to ε-greedy.

3.1 Implementing Epsilon-Greedy

For the ε-greedy strategy, implement the following:

- **Initialization:** Set initial epsilon value (e.g., 1.0 for full exploration at the beginning)
- **Decay Schedule:** Implement a decay function that reduces epsilon over time. Consider linear, exponential, or step-based decay schedules.
- **Action Selection:** With probability epsilon, select a random action; otherwise, select the action with the highest Q-value.
- **Analysis:** Track and visualize how epsilon changes throughout training and how it affects the agent's performance.

3.2 Implementing Boltzmann Exploration

For Boltzmann exploration, implement:

- **Temperature Parameter:** Set an initial temperature value and implement a schedule for decreasing it over time.
- **Softmax Calculation:** Compute action probabilities using the softmax function based on Q-values and temperature.
- **Action Sampling:** Sample actions according to the calculated probability distribution.
- **Analysis:** Track how temperature affects the action distribution and compare the exploration patterns with ε-greedy.

3.3 Comparative Analysis

Perform a thorough comparison of both exploration strategies:

- **Learning Efficiency:** Compare how quickly each strategy learns effective policies.
- **Final Performance:** Evaluate which strategy achieves higher final scores and Q-values after equivalent training time.
- **Exploration Behavior:** Analyze how each strategy explores the state space differently.
- **Stability:** Assess which strategy produces more stable learning curves.
- **Hyperparameter Sensitivity:** Determine how sensitive each strategy is to its hyperparameters.

Deliverables:

- Code implementation.
- A comprehensive final presentation, improving the previous presentation, plus discussing:
  - Implementation details for both exploration strategies, including parameter settings and decay/temperature schedules.
  - Comparative analysis of Epsilon-Greedy vs. Boltzmann exploration with respect to learning efficiency, exploration behavior, and final performance.
  - Visualizations showing how different exploration strategies affect learning curves and state space coverage.
  - Recommendations for which exploration strategy works best for the Snake game environment and why.

Grading Rubric

Task	Criterion	Weight (%)	Level 1
Task 1: Basic DQN	Neural Network Architecture	5%	Incomplete or ina architecture; lacks design.
	Q-Learning Loss Function	10%	Incorrect or incon implementation; la understanding of principles.
	Training Loop and Evaluation	5%	Incomplete or inc loop; lacks evaluat
	Heuristic Policy	10%	No heuristic imple poorly designed; l
	Presentation Quality	3%	Disorganized or u presentation; fails information effect
	Peer Review	2%	Does not participa review or provides feedback.
Task 2: Enhanced DQN with Experience Replay and Target Network	Experience Replay	10%	Incorrect or incon implementation; la understanding of