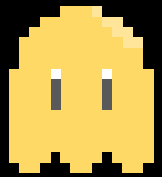
Reinforcement Learning agent: Snake Game



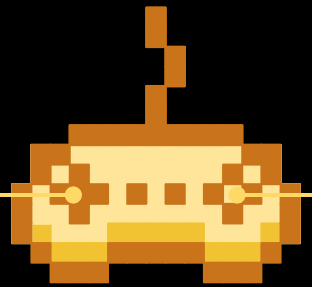
Team LLMH

Nova FCT - 2024-2025

FACULDADE DE CIÊNCIAS E TECNOLOGIA

ABOUT THE TEAM

- GALLO Lorenzo 72719
- WERCK Hugo 72692
- LEKBOURI Lina 72697
- LICHTNER Marc 72690



- Noticeable results:
  - Task 1:
  - Task 2:
  - Task 3:

Development

Task 1

Basic DQN Implementation without Experience Replay and Target network

Task 2

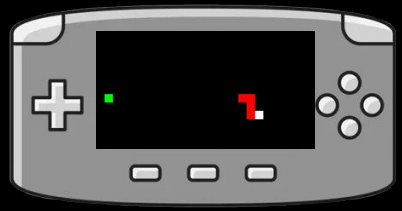
Enhanced DQN with Experience Replay and Target Network

Task 3

Exploration Strategies

Task 1

01



Basic DQN Implementation without Experience Replay and Target network

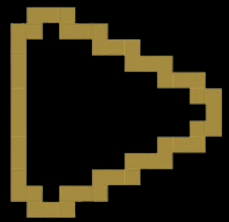
Task 1 development

→ Neural network architecture and 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.



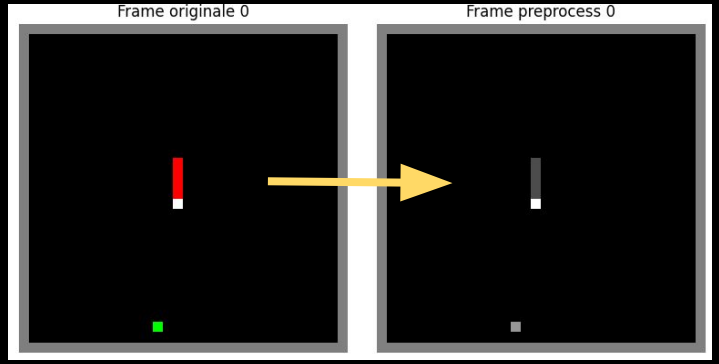
Preprocessing step

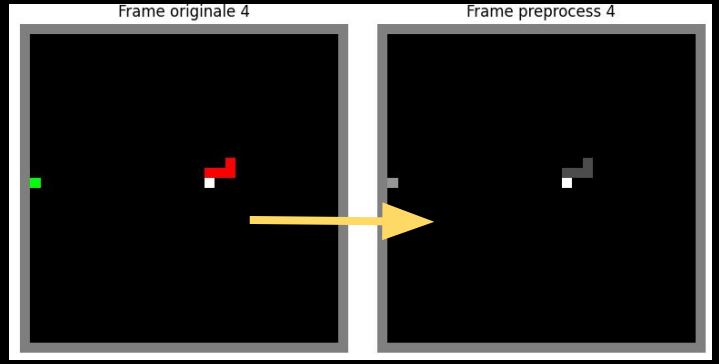
→ Convert RGB to **grayscale**

→ Convert the numpy array to a **Pytorch tensor**

→ Add **a batch and a channel** to fit the dimension

→ Crop area ? We usually can use this tool, but in our case the game board is already clean.

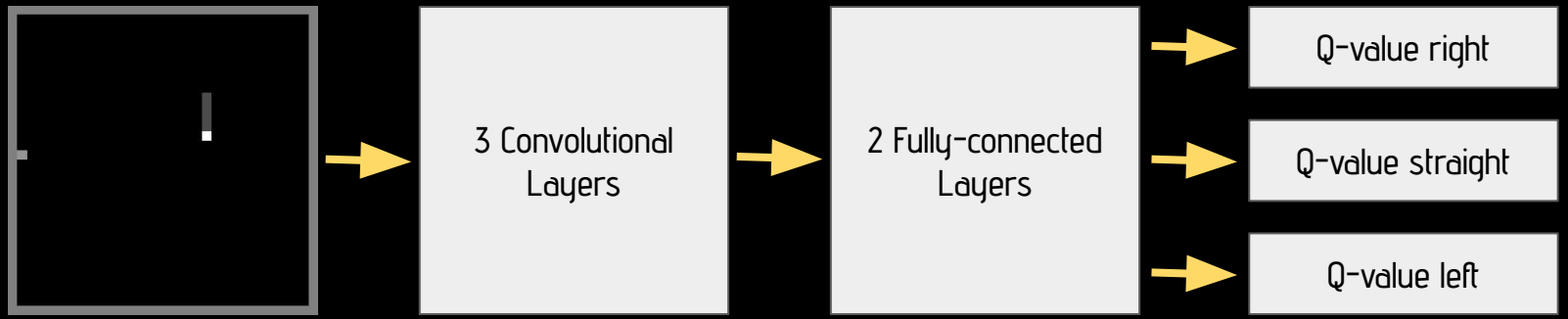




Model architecture

⟶ we use architecture of the DQN, this one is often used for this type of problem:

- 3 convolutional layers
- 2 fully-connected layers



Reminder, the DQN is the CNN model that will take as an input a preprocessed frame of the grid of the game and will provide as an output the Q-values for each possible actions.

Effectiveness of the model for the task

→ **Why a CNN?** Our snake game is divided in different frames of the game, that are images. Indeed our model need to learn from the images, hence the fact that we use a CNN.

→ **Is the architecture efficient?** It is complicated to determine in the first task if the model is efficient. Indeed, we don't have anything to compare to the result and since the model never store when it find a good action (task 2) we can't see clear efficiency in our results.
So, the choice was to use **the simplest model** we can create.

Q-learning Loss function

→ Implementation: it computes the difference between the predicted Q-values and the target Q-values. Taking the policy and the batch as an input, and returning the loss.

→ **Formula**: $$\text{Loss} = \left(r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)\right)^2$$

→ Why **MSE Loss**? Because Q-learning is a regression problem: we want the predicted Q-value to match the target Q-value as closely as possible.

Example on random values

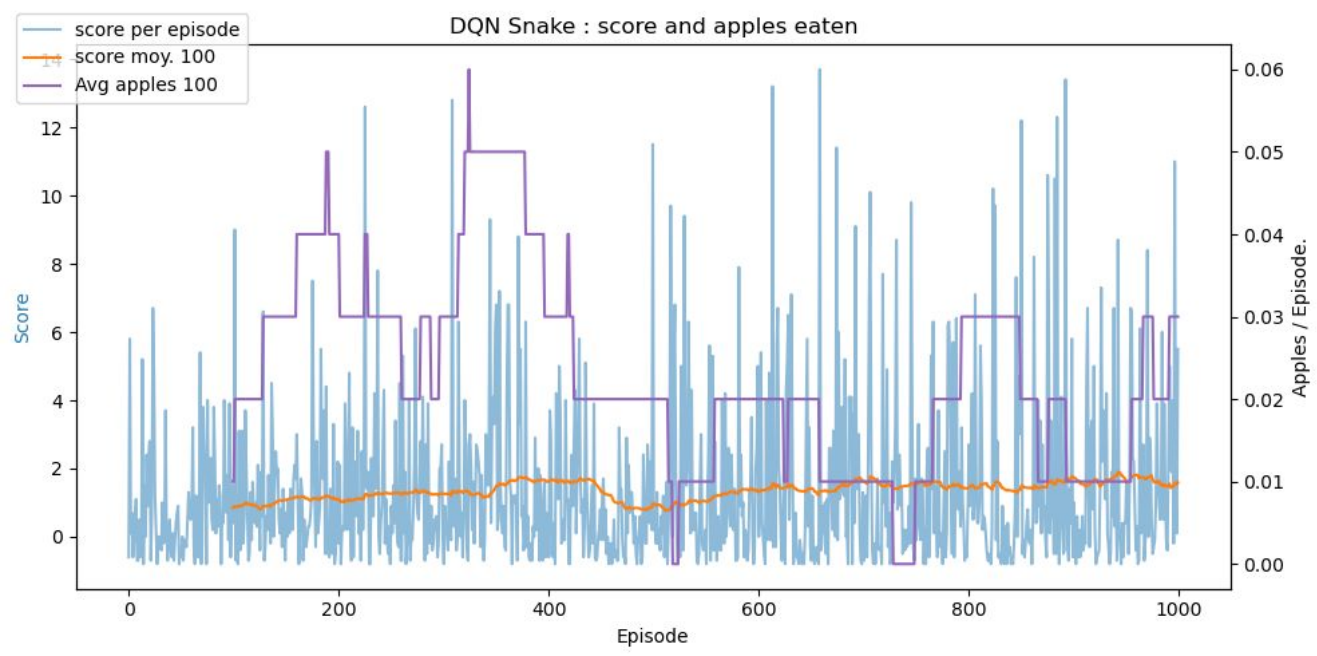
```
# Small test to verify that everything is fine and doesn't produce errors during execution
s      = torch.rand(1, 1, 32, 32, device=device)
s_next = torch.rand(1, 1, 32, 32, device=device)
a      = torch.tensor([[1]], device=device)
r      = torch.tensor([[0.5]], device=device)
d      = torch.tensor([[0]], device=device)

batch  = Transition(s, a, s_next, r, d)
loss   = Q_learning_loss(policy_net, batch)
print(loss.item())
```

✓ 0.0s

```
0.25003451108932495
```

Training loop

→ Results



Training loop

→ Interpretation of results:

- **High variability** in episode scores, indicating that the agent sometimes succeeds but lacks consistent performance.
- **Slow and limited learning progress**: The agent is learning but very slowly and its learning saturates early.
- **Very few apples eaten** on average: the agent rarely learn how to eat apples.

Training loop

→ Why are they as low?

- **No experience replay**: the agent train on correlated transition, causing instability.
- **No target network:** the same network is used to compute both Q(s,a) and the target, leading to overestimation problem.

Performance metrics

→ How do we measure the performances of the model?

- Score per episode (blue bars):
  - Total reward per episode
  - Capture immediate agent behavior
- Average score over the last 100 episodes (orange line):
  - Shows overall learning trend
- Average apples eaten per episode (purple line):
  - Measures task-specific success

Heuristic policy algorithm

→ **Objective or the heuristic**: improve agent meaning by providing a better than random decision.

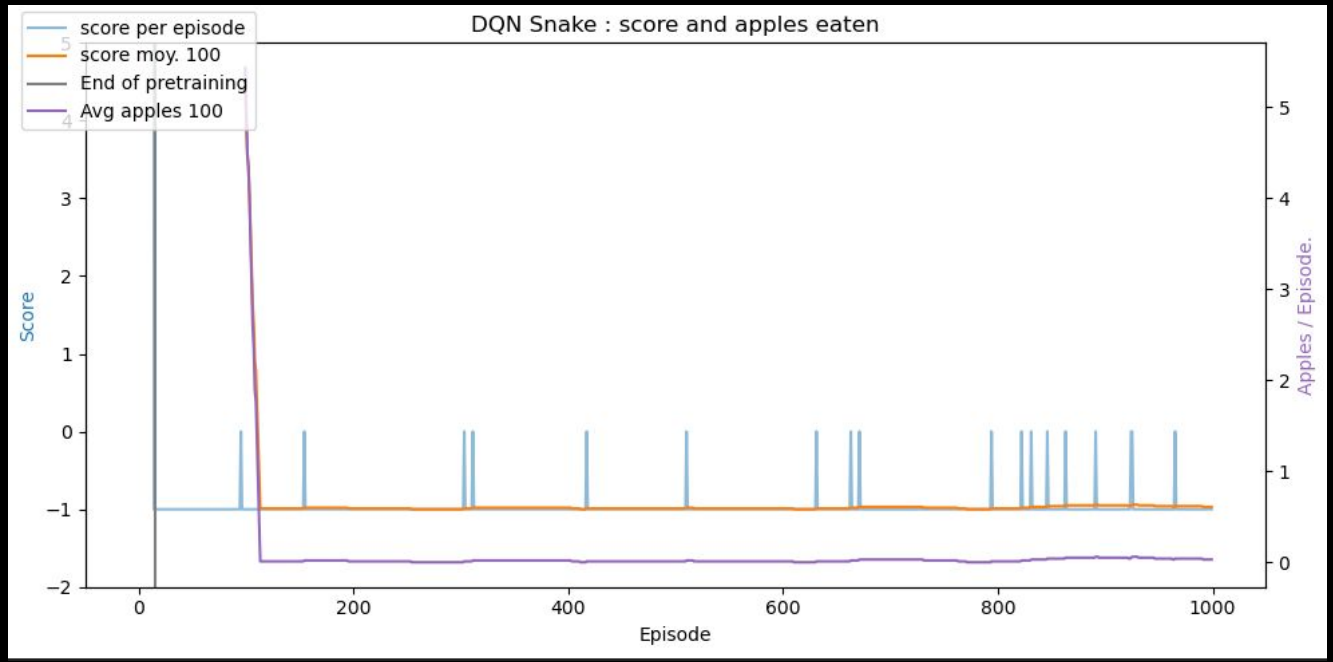
→ Decision process of the heuristic:

- Compute the preferred direction based on **distance to apple**
- For each possible action, we stimulate next position and check for **border or tail collision** and measure new **Manhattan distance**
- Select the **safest move**.

→ Why this heuristic? **Simple and easily interpretable**.

Training Loop with an Heuristic Policy

→ Results



Training Loop with an Heuristic Policy

→ Comparison with random play:

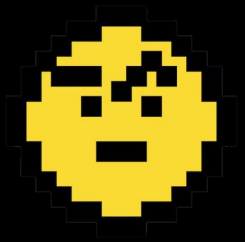
- **Learning progress**: With heuristic, the agent shows gradual improvement, without it stagnates.
- **Score evolution**: The heuristic policy leads to high and variable score.
- **Average score**: As the score is higher with heuristic, its average rises as well.
- **Apples eaten**: Heuristic results in a slow increase in apple collected whereas without the agent almost never eats apples.
- **Exploration quality**: The heuristic guide the agent towards better actions, without it the actions are random and inefficient.
- **Reward signal**: Heuristic policy helps the agent finds rewards, without it the agent rarely experience positive feedbacks.

Training Loop with an Heuristic Policy

→ Interpretation of the results with heuristic policy:

- Faster learning start: useful initial behavior provided
- Improved score over time: the average score increase steadily
- Successful reward acquisition
- Better state exploration
- Stronger Q-value estimates: allowing the network to converge more effectively.

Ablation study



⟶ **Model:** Simpler networks may train faster, while deeper ones could better capture complex patterns but risk instability.

⟶ **Number of training episodes:** Too few episodes prevent learning; too many can lead to saturation or overfitting.

⟶ **Loss function variance:** Testing MSE vs. Huber loss shows how learning reacts to prediction errors. Huber is more robust to outliers, potentially improving stability

⟶ **Heuristic policy variance:** Smarter heuristics can accelerate learning, while poor ones may mislead the agent.

⟶ We can also change the discount factor or the reward shaping.

Training time and efficiency

→ GPU/CPU possibility

```
# Select the corresponding device
device = torch.device(
    "cuda" if torch.cuda.is_available() else
    "mps" if torch.backends.mps.is_available() else
    "cpu"
)
```

→ Training time

For 1000 episodes	CPU	Kaggle's GPU: P100
Without heuristic	2min03s	1min21s
With heuristic policy	2min22s	1min41s

Want better result?

Swipe for Task 2 improvements!

Development

Task 1

Basic DQN Implementation without Experience Replay and Target network

Task 2

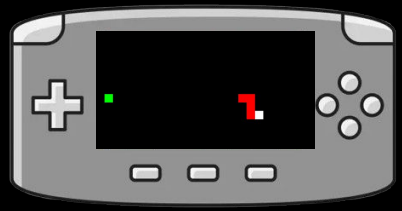
Enhanced DQN with Experience Replay and Target Network

Task 3

Exploration Strategies

Task 2

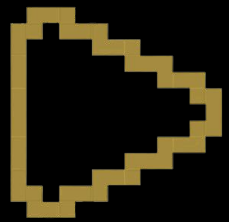
02



Enhanced DQN with Experience Replay and Target Network

Task 2 development and comparative analysis

→ Implementation details of Experience Replay.
→ Target Network update.
→ Comparative analysis.
→ Visualizations of learning curves.

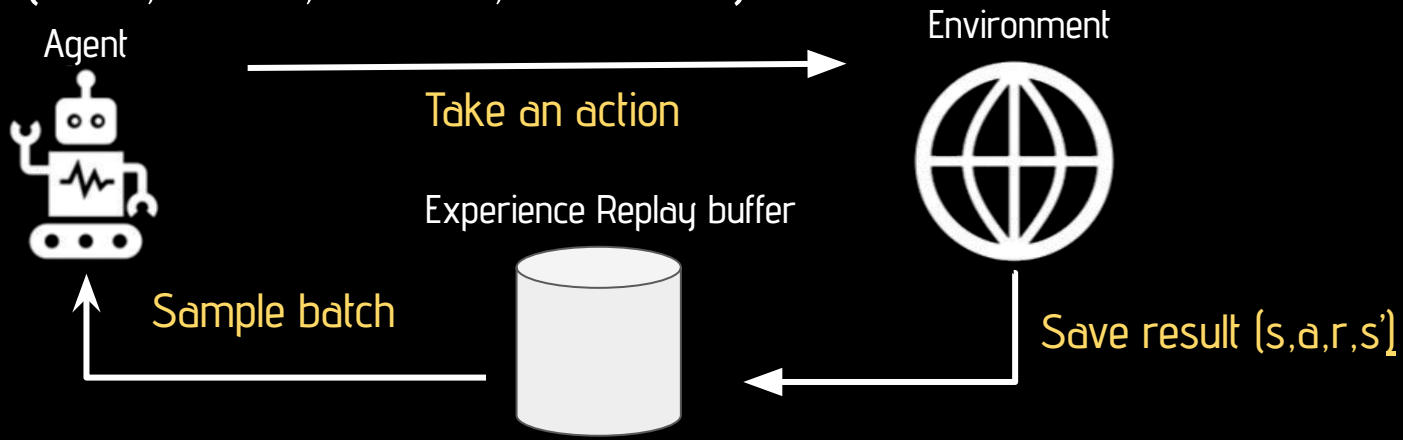


Experience Replay implementation

→What is **Experience Replay**?

It is a technique that improves learning stability and sample efficiency. Instead of learning directly from the most recent experience, the agent stores past experiences in a **replay buffer**.

It stores (state, action, reward, next state).



Experience Replay implementation

→ Implementation

We use a **Replay Buffer class** with fixed capacity. Each time we train the model, we store the tuple **(state, action, reward, next state)** in the replay buffer and when we train we **sample a batch from it**, with the corresponding method.

```python
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = 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)
```

Experience Replay implementation

→ Buffer size experiments:

We tried the experiment with different size of buffer, we implement the list that will contain all the buffer size we want to try:

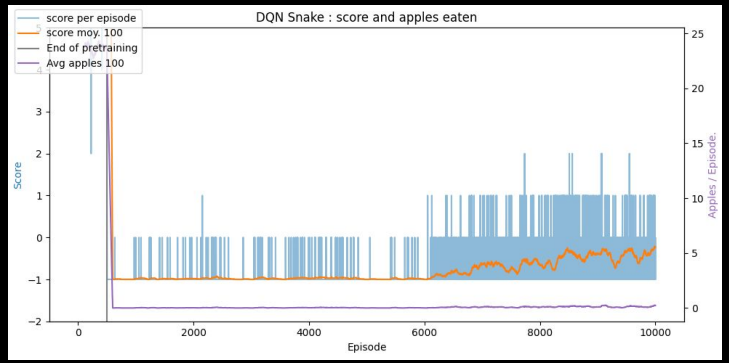
```
BUFFER_CAPACITIES = [10,10000]    # Array of differents values for the buffer capacity
```

This will be use in our Main training loop, giving result for each buffer size:

```
for capacity in BUFFER_CAPACITIES:
    memory = ReplayBuffer(capacity)
    scores, losses, apples = training_loop(memory, capacity)
    plot_training_results(scores, losses, apples)
```

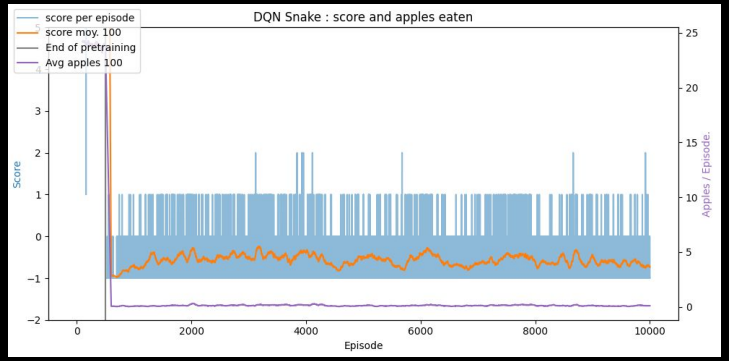
Experience Replay implementation

⟶ Impacts on performance: with 10 VS 10,000



⟶ The agent does **not show consistent improvement** over time.
⟶ The moving average score and apples consumed are stagnant, indicating **limited learning**.
⟶ A replay buffer of size 10 is **too small** to provide meaningful experience diversity. This results in **poor convergence and unstable training.**

⟶ The score and the moving average are **slightly higher and more stable.**
⟶ There is still performance variability, but with **less extreme fluctuations**.
⟶ A large buffer offers a richer variety of experiences.
⟶ While the overall performance is still modest, the training is noticeably **more stable and slightly improved**.

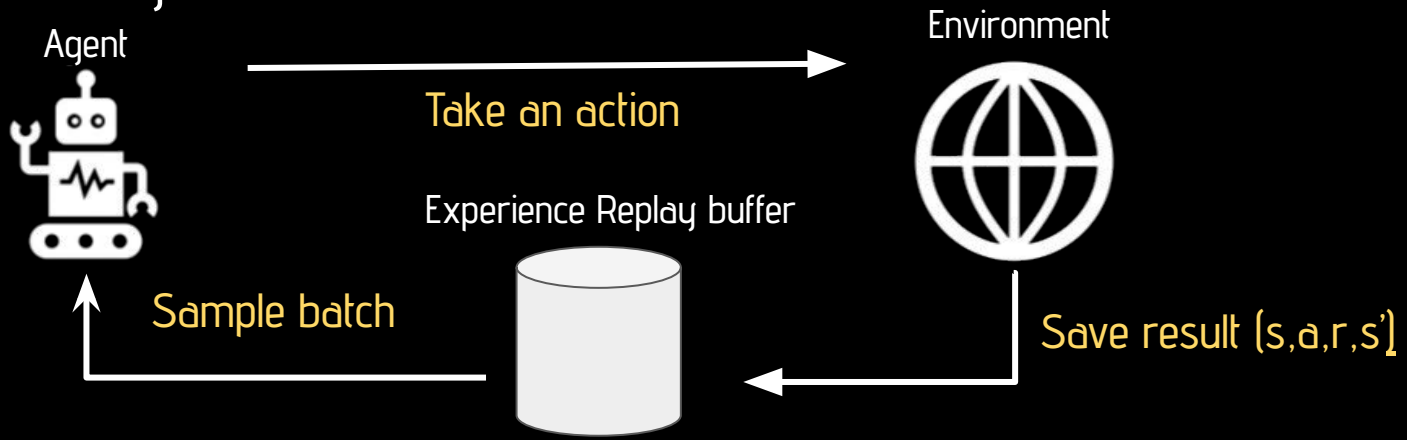


Target Network update

→What is a **Target Network**?

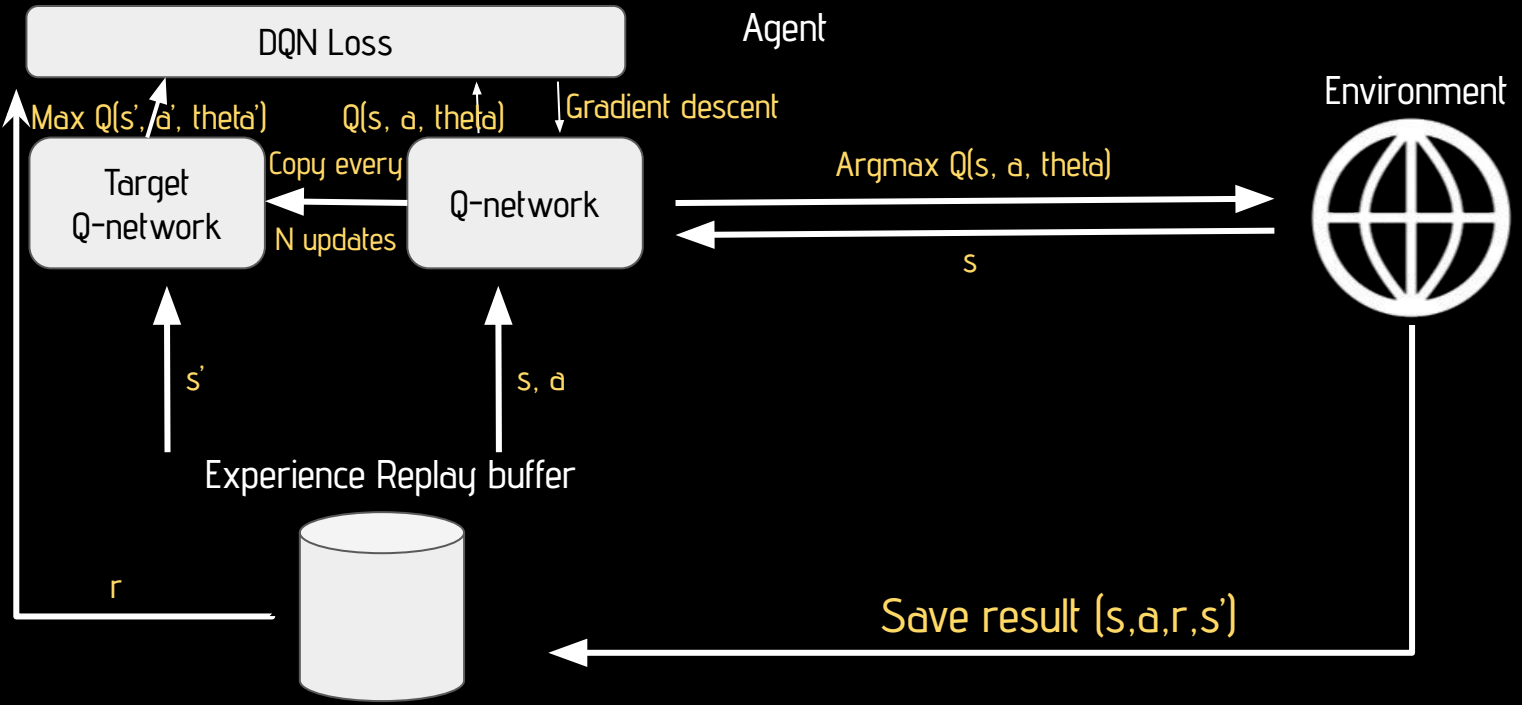
It is a **stabilisation technique**, that is a copy of the main Q-network, but its parameters are **updated less frequently**. Indeed, it helps **reduce instability** caused by correlated Q-value updates.

Zoom in the agent:



Target Network update

→What is a **Target Network**?



Target Network update

→ Implementation

When initializing our model, we initialize a **target network** that have the exact **same weights** of the policy network:

```
# Create the target network as a deep copy of the policy network (Step: initialization)
policy_net = DQN()
policy_net = policy_net.to(device)
optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)

target_net = DQN().to(device)
target_net.load_state_dict(policy_net.state_dict()) # Copy the weights from the policy network
```

This target network is updated **periodically** following **different strategies** (hard or soft) by copying the policy network during training. This network is used during training to compute the **training loss** to not be affected by **noisy updates** on the main network relative to this **formula**:

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

Target Network update

→ Strategies tested

Hard strategy:
We update the target network every N = 100 steps **brutally**:

$$\theta_{\text{target}} \leftarrow \theta_{\text{main}}$$

It is the most simple and commonly used strategie, but it may cause instability if weights change too quickly and that is something we really don't want.

Soft strategy:
With this strategy, we update the target network very slowly every N = 100 steps brutally (it can also be updated every step) following this formula:

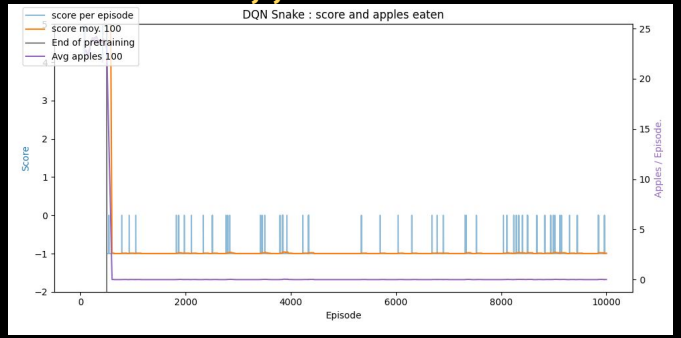
$$\theta_{\text{target}} \leftarrow \tau\theta_{\text{main}} + (1-\tau)\theta_{\text{target}}$$

With τ = 0.01.
This strategy is indeed more more **stable** and offers **smoother** transition.

Target Network update

⟶ Effects on training stability

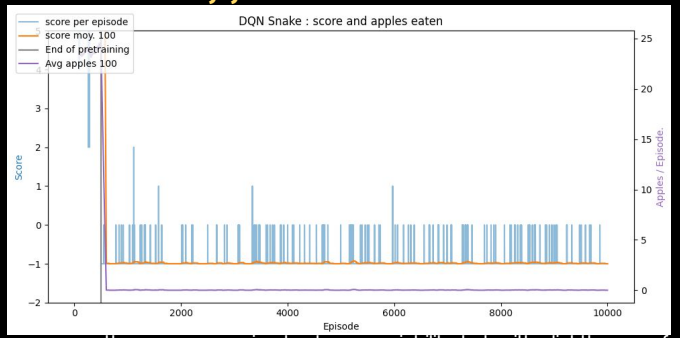
Hard strategy:



- high variability of the score with occasional spikes.
- relatively **low and stable moving average**, indicating that the learning is **not progressing** well and number of apples eaten per episode is also low

⟶ the **hard** update strategy is very **unstable** and it prevents the network from converging to an optimal policy.

Soft strategy:



- the score per episode shows variability but with slightly more frequent spikes.
- We see a little improvement in the moving average but it remains very low and the number of apples eaten per episode shows some improvement but is still not optimal.

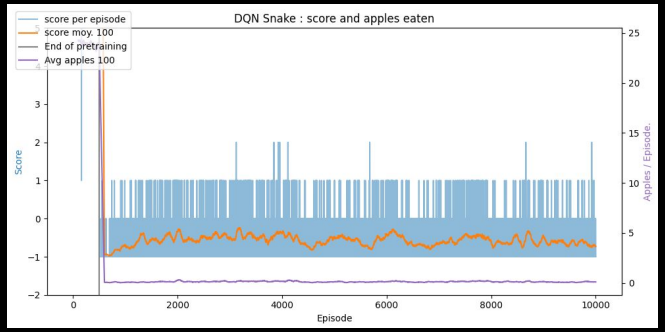
⟶ The **soft** update strategy provides **more stability in learning compared to the hard update**, as the gradual updates help in smoothing the learning process, though it is still **not very optimal.**

Conclusion: As this result show, to have the best result possible, we should use the **soft strategy** even if our graph show better results without the target network.

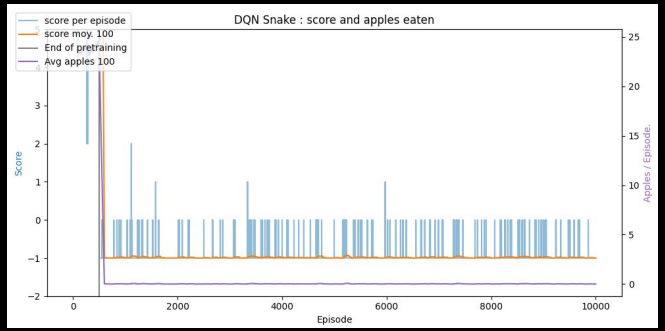
Comparative analysis

→ Results in performance metrics

Without target network

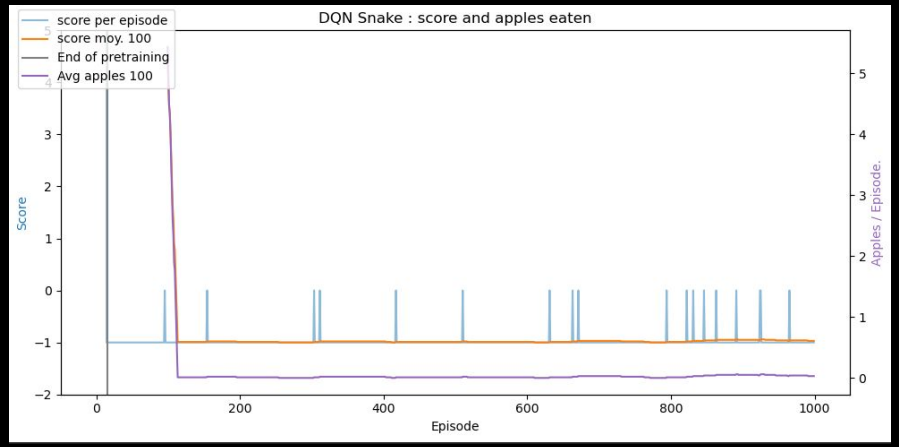


With target network



→ Comparison over the basic DQN

(without replay buffer and target network, with heuristic and 1000 episodes)



Comparative analysis

→ Performance metrics interpretation and comparison:

- **Without a target network**, the score per episode shows less variability and more improvement over time. The moving average is also higher and the number of apples eaten per episode is significantly higher, showing that the agent is learning to play the game more effectively with much learning progress.

→ Without a **target network**, the learning process is **more straightforward**, the network adapts faster and it eliminates the potential instability introduced by the updates to the target network, converging to a more stable learning.

- **With a target network,** as we explain before, it shows a lot of **instabilities** and **not much improvement** in our performance.

- **With the basic DQN**, the scores per episode show high variability and are **very low**, like the moving average and the number of apple eaten, depicting **poor performanc**e in learning.

→ Basic DQN simplicity may not be able to detect the complexity of the game and learn, which can lead to **over optimistic** value estimates and **obstruct learning**. Indeed, it performs the worst among the strategies, with low scores and minimal improvement, highlighting the **importance of experience replay in learning.**

Comparative analysis

→ Training time

For 10000 episodes	CPU	Kaggle's GPU: P100
Replay buffer of 10000	(as i was too long even with GPU, we didn't try with CPU to not damage our CPU)	~3h30
Replay buffer + Target network	(as i was too long even with GPU, we didn't try with CPU to not damage our CPU)	6 min for hard strategy 12min30 for soft strategy

→ Comparison over the basic DQN

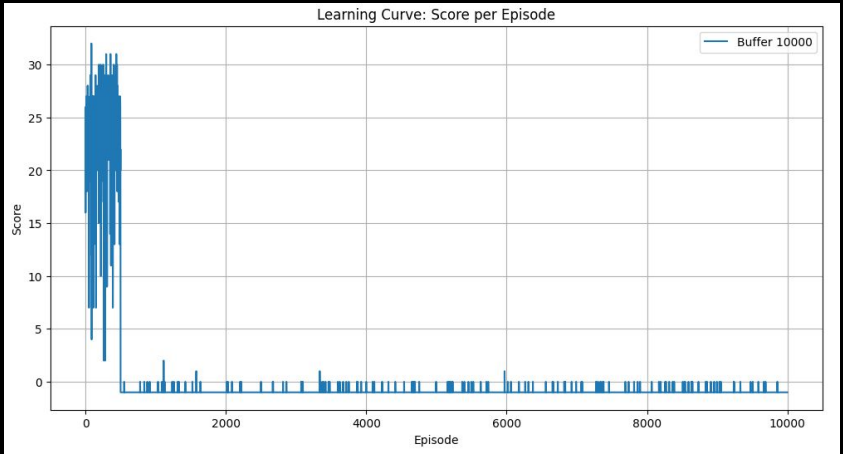
For 1000 episodes	CPU	Kaggle's GPU: P100
Without heuristic	2min03s	1min21s
With heuristic policy	2min22s	1min41s

Visualizations of learning curves

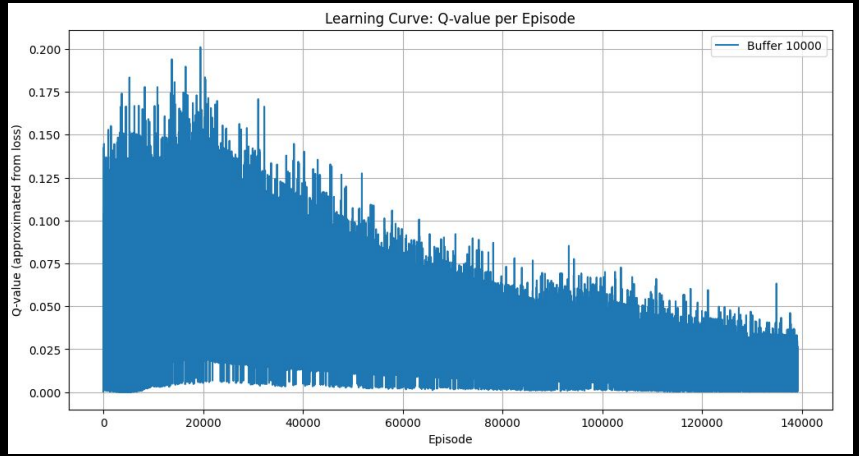
→ Results and interpretation:

Replay buffer with capacity of 10,000, 10,000 episode with a target network update using soft strategy

This graph represents the score achieved by the DQN agent per episode over 10,000 episodes.
As it depicts there is a high variability in scores, it does not show consistent upward trend. After the initial peak, the scores drop dramatically and remain low with occasional small spikes.
→ IT depicts that the agent learned a somewhat effective policy but failed to sustain or improve it, so **a failed in the learning process**.



This curve shows the Q-values approximated from loss over the episodes, reflecting the agent's learning of state-action values.
As depicted, there is **noticeable variability** in Q-values throughout the episodes, **with a lot of spikes**.
The decreasing trend in Q-values indicates that the **agent's confidence in its learned policy is decreasing**. The variability and lack of stabilization in Q-values suggest that the agent is **struggling to converge to an effective policy.** The soft update strategy for the target network may **not be effective** for the agent in learning a stable and optimal policy.



Visualizations of learning curves

→ Stabilizing effects of these enhancements

Replay buffer: helps in reducing the correlation between consecutive experiences and provides a more diverse set of experiences for learning. We saw a lot of improvements with it.

Target Network (hard or soft strategy): The target network did not show significant benefits in this context. Indeed, the learning curves indicate instability and a lack of consistent improvement, suggesting that the target network may not be necessary for achieving optimal performance.

Overall performances: The enhancements, such as the target network with soft updates, did not lead to the expected improvements in learning stability or performance but the DQN model with a replay buffer and without a target network appears to be more effective for our work.

In conclusion, with these enhancements, we show the importance of carefully selecting and tuning parameters to ensure their positive contribution to the learning process. Finally, in our work, **it is better to use a DQN model with a replay buffer BUT without target network and its updating strategies that may cause instability in learning.**

Improvements

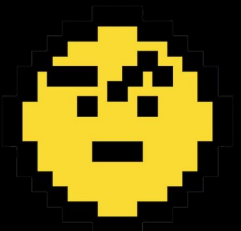
⟶ Ideas to **improve** our result:

As we saw in our graph previously, the results are **not really satisfactory**, indeed it seems like our model is **not learning a lot** even after integrating even the replay buffer or the target network. To try to improve that, we try to:

- Increase the **number of episodes**: it leads to a small improvement in the loss (1K to 10K)
- Change the **image** we give to the model: no conclusive results
- Change de **Epsilon value**: a little bit conclusive
- Change the **buffer capacity**: again it shows a small improvement.

After trying to integrate performance improvement, apart from the number of episode and the Epsilon value, nothing is very conclusive...

Ablation study



→ **Replay buffer size:** Varying replay buffer sizes helps assess the trade-off between learning stability and computational complexity. A larger replay buffer shows better results.

→ **Target network with different update strategies:** Comparing no target network, hard update, and soft update strategies reveals their impact on learning stability and performance. In our task, not using any target network shows up to be the best choice.

→ **Discount factor:** Different discount factors influence the agent's focus on immediate versus long-term rewards, affecting learning stability and convergence.

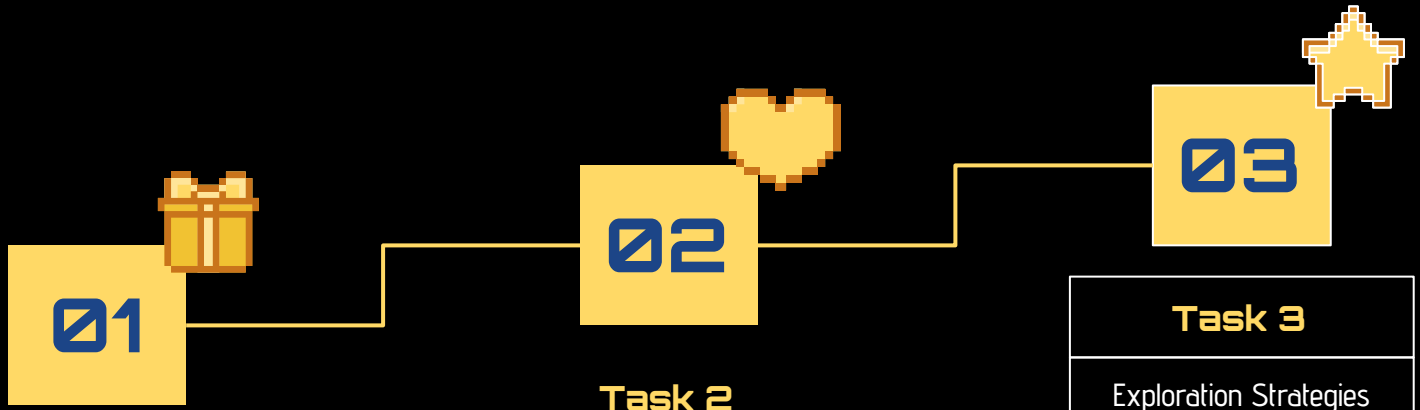
→ **Reward shaping:** To help our model learn, we can change the reward model, we already did that, rewarding the model any time it takes a step without dying, helping him to continue the game and try to improve its performances.

→ We systematically evaluated the impact of different components and enhancements on the DQN model's performance. The findings suggest that though certain enhancements like replay buffers can really improve learning stability and performances, others, like target networks with soft update strategies, may not always benefit the learning process.

Not convinced yet?

Swipe for Task 3 improvements!

Development



Task 1

Basic DQN Implementation without Experience Replay and Target network

Task 2

Enhanced DQN with Experience Replay and Target Network

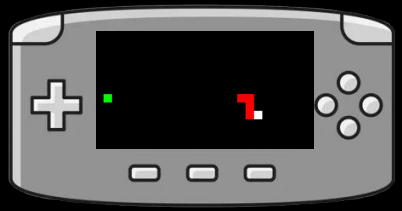
Task 3

Exploration Strategies

Task 3

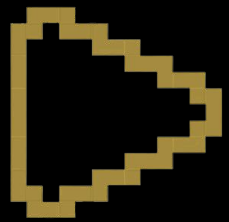
03

Exploration Strategies



Task 3 development and final version

→ Implementation details for both exploration strategies.
→ Epsilon-Greedy vs. Boltzmann exploration.
→ Visualizations
→ Which strategy for the snake game?



Exploration strategies implementation

→What is an Exploration strategy?

In reinforcement learning, an agent must balance two competing goals:

- **Exploitation**: Chooses the action that it currently believes is best, based on past experience
- **Exploration**: Try out actions that may seem not optimal right now, in order to discover better strategies for the future.

An "**exploration strategy**" is the rule the agent uses to decide when to explore, and how to explore. Without enough exploration, the agent can get "stuck" repeatedly choosing a mediocre action, never discovering a higher-reward path. Without enough exploitation, the agent spends all its time wandering around randomly and never converges to a good policy.

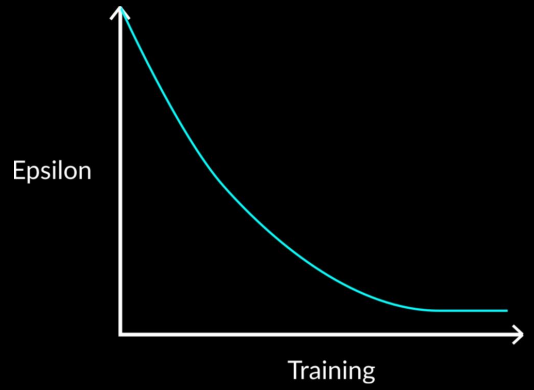
Epsilon greedy strategy

The **core idea** is that we have two probabilities:

- probability ε (exploration), pick a uniformly random action,
- probability 1 − ε (exploitation), pick the action that maximizes the current Q-value.

ε starts near 1.0 ⟶ Almost every step is random.

As training proceeds ε decays ⟶ Fewer random moves, more greedy picks based on learned Q-values.



Epsilon greedy implementation

Hyperparameters:

- Initial **ε = 0.05** (i.e., 5 % random actions at start).
- Exponential decay schedule every timestep t:

$$\varepsilon_t = \varepsilon_{\min} + (\varepsilon_{\max} - \varepsilon_{\min}) \cdot e^{-t/\tau_\varepsilon}$$

→ ε smoothly goes from 0.05 → ~0.0001 over first ≈ 50 000 timesteps, then stays near zero.

Why low initial ε (0.05)?

The Snake environment is relatively high-dimensional (30×30 grid). A small nonzero ε still forces occasional random exploration without being completely chaotic.

We experimented with 0.1 and 0.01; 0.05 yielded the best **trade-off** in early apple-discovery vs. training stability.

Boltzmann strategy

For Boltzmann strategy, instead of "flipping a coin", the Q-values are converted into a probability distribution using a temperature T:

$$P(a \mid s) = \frac{\exp\big(Q(s,a)/T\big)}{\sum_{a'} \exp\big(Q(s,a')/T\big)}.$$

Starts with **T≈5.0** → policy is almost uniform, so the agent explores widely.

As Training Proceeds T decays → higher-Q actions gain exponentially more probability → strong exploitation.

Why Boltzmann? Even during exploration, "better" actions get higher probability (informed exploration) and smoother transitions from exploration → exploitation as T lowers.

Boltzmann implementation

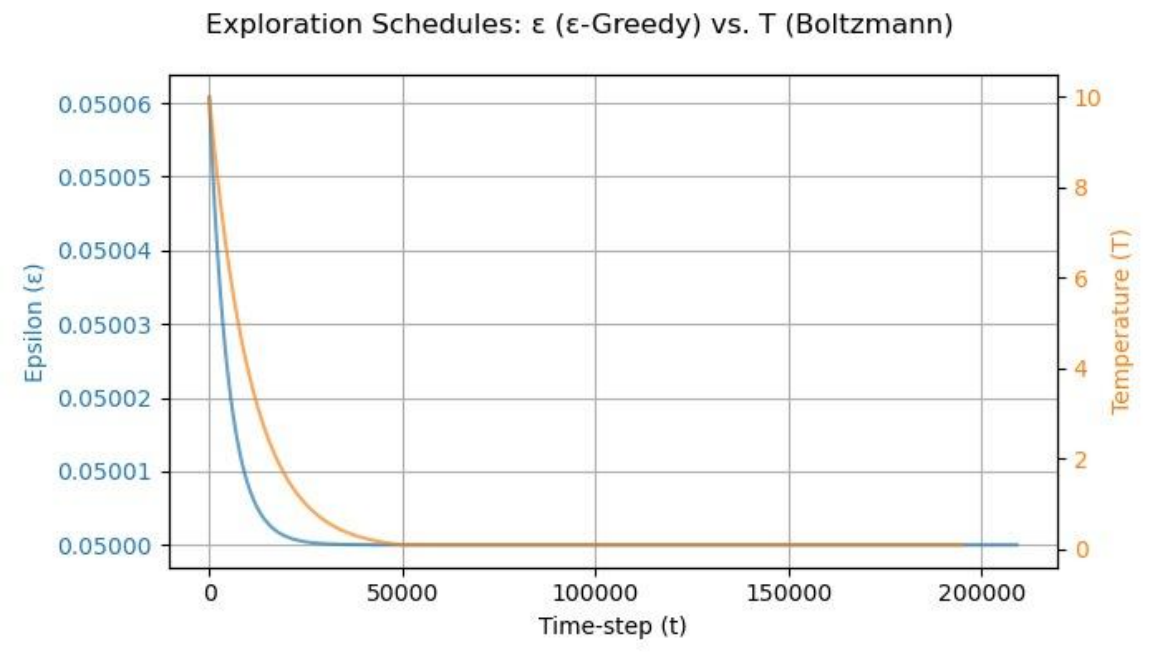
Hyperparameters:

- Initial **T = 10.0** (agent begins by exploring almost uniformly).
- Decay schedule: Exponential decay every timestep t:

$$T_t = T_{\min} + (T_{\max} - T_{\min}) \cdot e^{-t/\tau_T}$$

- After ~**50 000 timesteps**, T is so low that the policy effectively becomes greedy.

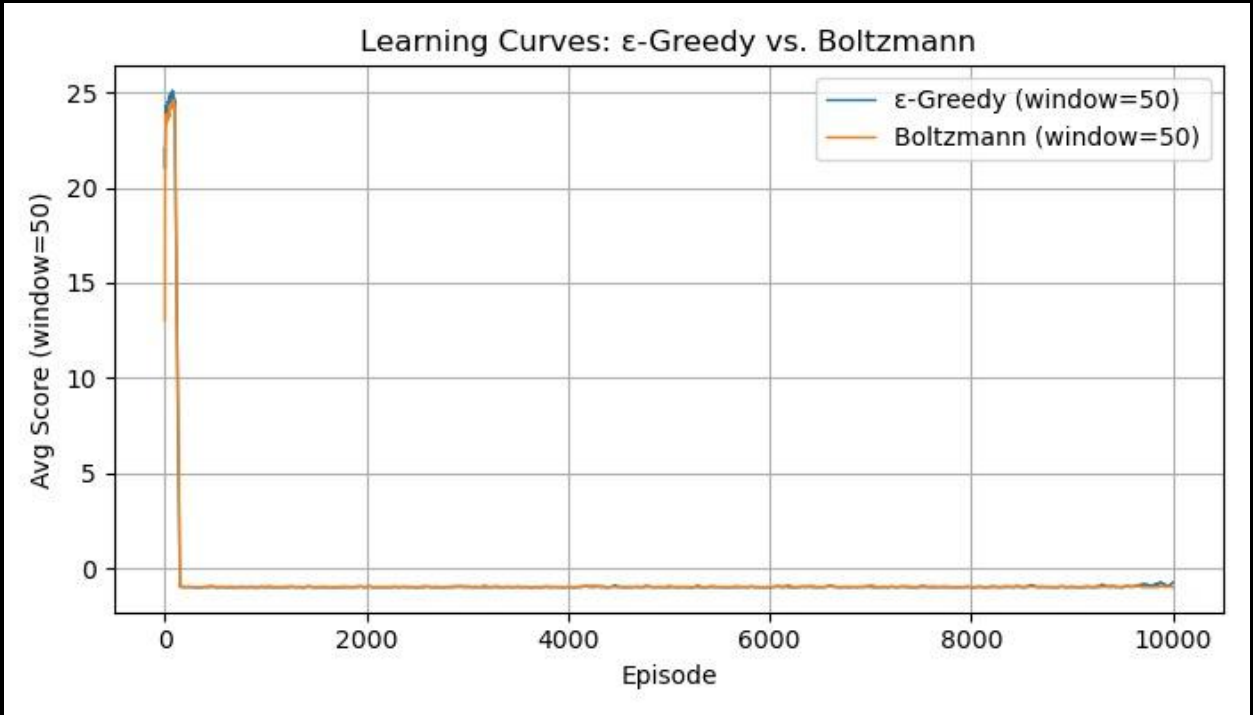
Decay Schedules for ε and T



Decay Schedules for ε and T

1. Both ε and T follow exponential decay so that by ~50 000 steps the agent is **nearly fully greedy**.
2. We set decay rates (τ_{ε}, τ_{T}) so that:
   - ε drops from 0.05 → 0.0001 over ~50 000 steps.
   - T drops from 10.0 → 0.01 over ~50 000 steps.
3. Plot of ε_t and T_t vs. timestep (0–200 000).
   - Notice the steep drop in T from 10 → ~1 over first 10 000 steps, whereas ε only drops from 0.05 → ~0.025.
   - After 50 000 steps, both are essentially zero—policy is purely greedy.

Learning Curves: ε-Greedy vs. Boltzmann



Learning Curves: ε-Greedy vs. Boltzmann

We trained both agents for 10 000 episodes each, tracking average score (per episode).

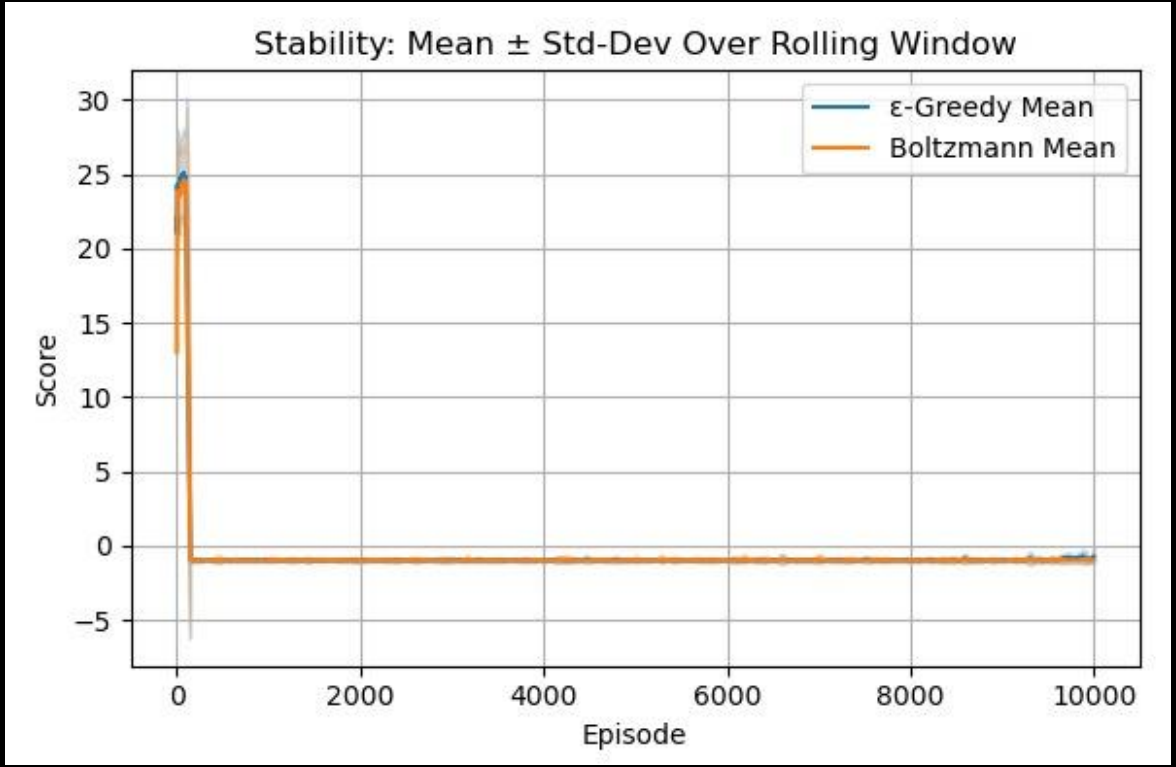
To smooth out noise, we plot a 50-episode rolling average of score.

Observation: Toward early episodes (0–200), both strategies quickly spike in average score (~25), then collapse to near zero and remain there.

⇒ This indicates that both strategies initially find a path to eat apples, but ultimately **fail to converge** to a stable high-reward policy.

Slight early advantage to ε-Greedy (blue) vs. Boltzmann (orange), but beyond episode 200, learning plateaus near **zero for both.**

Stability Analysis: Mean ± Std-Dev Over Rolling Window



Stability Analysis: Mean ± Std-Dev Over Rolling Window

Early episodes (0–200):

- ε-Greedy shows slightly lower variance (tighter band) than Boltzmann.
- Boltzmann's "informed" exploration causes more fluctuations at first.

After ~200 episodes:

- Both curves converge to mean ≈ 0, std ≈ 0.1.
- Indicates neither strategy discovers a reliably high-reward policy.

Conclusion: **ε-Greedy is marginally more stable** in early learning; afterward, **both are equally "flat"** (no learning gains).

State space coverage

Greedy Coverage:

- Random actions occur only ε = 0.05 → 0.0001 → very sparse exploration of unseen grid cells.
- After ε decays, agent repeatedly follows its learned Q-policy, often looping in the same corridor or corner.
- **Result**: Large portions of the 30×30 board (like opposite quadrants) remain virtually unvisited.

Boltzmann Coverage:

- Early (T≈10) → nearly uniform probability → visits most regions, including apple-rich areas.
- Mid (T≈2–5) → "mildly informed" exploration favors some high-Q states but still tries novel moves.
- Late (T→0.01) → collapses to greedy, causing coverage to shrink to narrow loops similar to ε-Greedy.
- **Overall**: Better initial spread but still fails to sustain broad coverage once T decays.

Implication:

- Neither strategy maintains high coverage long enough to reliably discover robust apple-eating trajectories.
- Post-annealing, the snake's head frequently cycles through a small subset of states, leaving most of the board unexplored.

Training efficiency

ε-Greedy:

- First peak (~25 score) reached around episode 70.
- Score increases steadily with relatively low variance in early training.
- Decayed ε (0.05 → 0.0001) lets Q-network gather diverse experiences before exploitation dominates.

Boltzmann (Softmax):

- First peak (~25 score) reached sooner (around episode 50) thanks to high-T "informed" exploration.
- Higher early variance—sometimes over-commits to noisy Q-estimates.
- Rapid T decay causes a steeper collapse in performance once exploitation takes over.

Key takeaway:

- Boltzmann learns slightly faster at the very start, but ε-Greedy is more stable overall.
- Neither strategy sustains high scores beyond ~episode 200, but ε-Greedy's smoother rise is preferable for consistent updates.

Which strategy for the snake game?

Why ε-Greedy?

- **Simplicity**: Only one ε schedule to tune; fewer hyperparameters.
- **Stability**: Lower early variance—more consistent Q-updates in initial episodes.
- **Comparable Performance**: Both methods plateau near zero long-term, but ε-Greedy reaches its first peak more steadily.
- **Easier to Extend**: If more exploration is needed, simply adjust ε_{max} or decay rate rather than redesigning temperature schedule.

→ **Conclusion:**

- **ε-Greedy is preferred for our Snake DQN setup.** Its simpler, more stable behavior outweighs Boltzmann's marginal early advantage, and neither strategy solves Snake without further enhancements.

Why the model is not learning?

Grayscale-only input:

- Apples, walls, and snake all appear similar in intensity, so the CNN can't reliably distinguish key elements.
- Attempts to re-weight or enhance pixel values didn't meaningfully improve feature separation.

Insufficient training/pre-training:

- 10 000 episodes is likely too few for a 30×30 Snake grid with sparse rewards.
- Without a CNN pre-training or many more steps (like ≥1 000 000 frames), the network never sees enough varied apple/collision examples to learn robust features.

Possible silent implementation bugs:

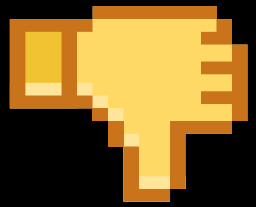
- A variable we use in the computation of the training that doesn't behave the way we think leading to miscalculations and biased learning
- Replay-buffer or target-network update schedule might be flawed, leading to noisy Q-value targets without any Python errors thrown.

Resulting low signal-to-noise:

- CNN struggles to identify apples/walls, and noisy or incorrect actions corrupt Q-updates.
- With sparse + terminal rewards, the model never converges to a stable apple-eating policy.

Lessons learned and insights gained

→ We understand more **Q-learning** by implementing this example and explaining each method to improve the quality of the learning helps us understand more how each one works, is implemented. We have now a clear and precise definition of Q-learning.
→ We saw how the model behave on a **random playing strategy** and how an **heuristic**, even the simplest one can change the result.
→ We understand the construction of **replay buffer** and the integration of the **target network** and its different strategies.
→ As we try to improve our different result, we **fine-tuned all of our hyperparameters** to find better results, most of the time it wasn't conclusive....
→ **ε-Greedy** offers stable learning but limited coverage, while **Boltzmann** provides faster **exploration** at the risk of premature **exploitation**.



What went wrong

→ We fail to improve our result by fine-tuning our hyperparameters, even if we change a lot of them, we only saw very small changes that was not very conclusive.

→ Adding replay buffer and target network do not give as improvement as we thought.

→ We "turned off" exploration way too fast, so the snake just got stuck in the same bad loops instead of actually checking out the rest of the board.

What went great

→ We manage to understand and have now a clear **definition of Q-learning** and its different optimization strategies.
→ The implementation of the **heuristic** shows a lot of improvements in front of the random play strategy, the opposite would have been surprising.
→ Following the **plan given by the assignment was very clear**, we manage to do everything step by step, and understand each of our result, even when we fail to improve them and results in having not as good, as we thought, model.
→ **Both ε-Greedy and Boltzmann** helped the agent discover an apple-eating trajectory early in training.