Jesse the (ro)bot

Robot vacuum cleaner simulation
powered by Reinforcement Learning trained agent

Complex Systems: Models and Simulation
Alberto Sormani
August 2025

Contents

Chapter 1

Introduction

1.1 Purpose

This project represents the final assignment for the course *Complex Systems: Models and Simulation*, taught by Professors Giuseppe Vizzari and Daniela Briola.

The objective of the work is not to deliver a finished industrial product, but rather to explore, study, and apply the theoretical concepts introduced during the course. My personal goal was to deepen my understanding of complex systems, reinforcement learning, and simulation techniques, while also enjoying the process of experimenting and learning by doing.

1.2 Project

The project revolves around the design of an artificial agent capable of controlling a robotic vacuum cleaner in simulated environments. The ultimate aim is to identify and learn the most efficient cleaning strategies under realistic constraints.

The robot begins without prior knowledge of its environment: it must progressively build a map of the space in real time, while simultaneously performing its task of cleaning. During each episode, the robot must:

- clean the floor surface as effectively as possible,
- avoid collisions with walls and obstacles,
- manage its limited battery autonomy,
- and return to the charging base before running out of energy.

In this sense, the robot embodies the challenges of autonomous navigation: perception, mapping, planning, and decision-making must all work together in order to achieve success.

1.3 Practical limitations

It is important to clarify that, although the conceptual design and simulation framework have been completed, the project could not be carried out to its full experimental extent. The primary reason is the limited computational resources at my disposal: training reinforcement learning agents of this type generally requires extensive parallel computation on GPUs or access to high-performance computing clusters.

In my case, experiments were restricted to the CPU of my personal laptop, which is not sufficient to handle the scale of simulation steps and the complexity of training algorithms such as PPO. As a result, the project should be considered as a theoretical and methodological exploration rather than a finalized, fully-trained system. Despite these constraints, the implementation provides a working foundation, and the lessons learned regarding modeling, simulation, and reinforcement learning remain highly valuable.

1.4 Structure of this work

The remainder of this work is organized as follows:

- Chapter 2 introduces the main **topics** of the project, including complex systems, SLAM, neural networks, and reinforcement learning (Q-learning and PPO).
- Chapter 3 describes the **environment**, both the continuous simulation and its discrete abstraction, with assumptions, mapping, and base placement.
- Chapter 4 presents the **agent**, first the Q-learning baseline and then the PPO agent in its single-MLP and multi-encoder versions, followed by a comparison.
- Chapter 5 explains the **training process** of the PPO multi-encoder agent, covering the loop, reward shaping, curriculum learning, rollouts, and outcomes.
- Chapter 6 concludes with the **results and future work**, reflecting on achievements, limitations due to computational resources, and possible extensions.

This structure is meant to guide the reader step by step, from theoretical foundations to the implementation details and finally to the results and perspectives for future development.

1.5 About the title and the cover image

The title of this project, *"Jesse the (ro)bot"*, and its cover illustration are not accidental choices. The robot depicted on the cover is a vacuum cleaner disguised as a Wild West bandit. This playful idea originates from the connection with the well-known *multi-armed bandit problem* in reinforcement learning. Although the bandit problem itself is not directly addressed in this project, the word "bandit" naturally evokes the imagery of outlaws.

For this reason, I decided to name the robot **Jesse**, as a reference to the infamous outlaw Jesse James. The combination of a robotic vacuum cleaner and a bandit character creates a lighthearted metaphor: the robot is both a cleaning agent and a mischievous "outlaw" exploring its environment.

The "(ro)bot" part of the title reflects another important aspect. While the project is inspired by real domestic robots, what has been developed here is not a physical robot but a **simulation**. Thus, it is more accurate to describe it as a "bot." The title therefore plays with this duality: Jesse is both a robot in spirit and a bot in implementation.

Note to the professors

Since I was unable to find any similar project available online, I decided to make my work publicly accessible. For this reason, I chose to write the report in English and to adopt a clear, accessible style that may be useful not only within the scope of this course but also as a reference for anyone interested in robotics, reinforcement learning, or simulation. My hope is that this document can serve both as a course project and as a small contribution to the broader learning community.

1.6 The GitHub repository

All the code developed for this project is publicly available in a GitHub repository. The repository is organized to separate the core simulation from configuration and utility scripts. The main folder `rlrc` contains the agents, the robot and its sensors, the environment, the encoders, and all training and evaluation scripts. Constants and predefined maps are stored in a dedicated subfolder, while the entry point `play.py` allows quick execution of the simulation.

The goal of publishing the code is twofold: to ensure transparency and reproducibility of the results presented in this report, and to provide a resource for other students or practitioners interested in reinforcement learning applied to robotics.

The repository can be accessed at:

https://github.com/Albi99/Jesse-the-Robot-Vacuum-Cleaner

Chapter 2

Topics

Before diving into the details of this project, it is worth stepping back and looking at the bigger picture: which key ideas, concepts, and technologies come into play here? This chapter is a sort of "toolbox unpacking." We will briefly walk through the main topics involved, explaining not only what they are but also why they matter in our context. Think of it as a friendly guided tour through the theory we are going to put into practice.

2.1 Complex Systems: Models and Simulations

2.1.1 Complex Systems

Complex systems are everywhere: from ant colonies to social networks, from the brain to financial markets. They are made up of many interacting parts (agents, individuals, components—you name it), and the magic lies in how simple local interactions can give rise to unexpected global patterns.

In this project, the robot plays the role of the agent, while the environment provides the rules, constraints, and feedback that shape the agent's behavior. The environment is not an agent itself—it cannot observe or act—but it evolves in response to the robot's actions, and this continuous interaction gives rise to dynamics that are richer than either element in isolation. In this sense, the robot–environment loop can be viewed as a small-scale complex system, where emergent behavior arises from the interplay between decision-making and contextual feedback.

2.1.2 Models

A model is essentially a simplified representation of reality. Instead of dealing with all the messy details of the real world, we strip things down to their core mechanics. Models can be mathematical, computational, or even conceptual. For this project, the model is the simulated environment in which the robot operates, along with the assumptions we make about sensors, obstacles, and movements. Without a good model, simulation and learning would be either inaccurate or simply impossible.

2.1.3 Simulations

Simulations are what bring models to life. They let us experiment with "what if" scenarios without the cost (or danger) of real-world testing. Imagine crashing a thousand virtual robots into walls before risking a single real one—that's simulation at work. In this project, simulation plays a central role: we can test strategies, tweak algorithms, and push the agent to its limits, all within a safe digital playground.

2.2 Simultaneous Location And Mapping

2.2.1 What is SLAM?

SLAM stands for *Simultaneous Localization And Mapping*. The idea is deceptively simple: while moving around in an unknown environment, a robot must build a map of its surroundings and, at the same time, keep track of its own position within that map. It is a chicken-and-egg problem: you cannot map without knowing where you are, and you cannot know where you are without a map. SLAM algorithms solve this

paradox by continuously updating both the map and the estimated position of the robot as new sensor data comes in.

2.2.2 LiDAR

One of the coolest technologies in robotics is LiDAR (Light Detection And Ranging). Picture it as the robot's way of "seeing" the world: it shoots laser beams, measures how long they take to bounce back, and from that builds a map of its surroundings. Within the SLAM framework, LiDAR provides precise distance measurements that allow the robot to detect walls and obstacles. Combined over time, these measurements help to construct and refine a global representation of the environment.

2.3 (Deep) Neural Network

2.3.1 Neural what?

Neural networks are inspired by the way our brain works, but with far fewer neurons and no coffee addiction. They consist of layers of connected nodes (neurons) that transform inputs into outputs step by step. The idea is that, by stacking many of these layers, the network can learn incredibly complex functions—essentially, it learns to map situations to decisions.

2.3.2 Environments

In reinforcement learning, an environment is simply the "world" the agent interacts with. It defines what the agent perceives (observations), what it can do (actions), and how it gets feedback (rewards). For this project, the environment is a simulated 2D world full of walls, empty spaces, and a little robot eager to clean. It is important to stress: the environment is not an agent. It does not make decisions; instead, it reacts deterministically (like in this project) or stochastically to the agent's actions and updates its state accordingly.

2.3.3 Agents

The agent is the protagonist of the story. It's the decision-maker, the learner, the explorer. The agent receives observations from the environment, takes actions, and gets rewards (or punishments) in return. Over time, it learns a policy—that is, a strategy for deciding what to do in different situations.

2.3.4 (Deep) Convolutional Neural Networks

Sometimes the input to an agent is not just numbers but entire maps or images. That's where Convolutional Neural Networks (CNNs) come in. They are particularly good at processing spatial data, spotting patterns, and extracting relevant features. In robotics, CNNs are often used to interpret sensor data, like LiDAR scans or occupancy grids. The concept is: if the agent's brain is a neural network, CNNs are the parts specialized in visual/spatial reasoning.

2.4 Reinforcement Learning

2.4.1 What is it?

> Welcome to the world of reinforcement learning (RL), a paradigm of machine learning that's inspired by how humans and animals learn through interaction with their environment. Imagine teaching a dog a new trick: you don't explicitly tell it what to do, but you reward good behavior and discourage bad behavior. Over time, the dog learns to perform the trick to maximize the treats it receives. This is essentially how RL works, but instead of dogs and treats, we have algorithms and reward signals. [7]

2.4.2 How it work?

Reinforcement learning is a branch of machine learning that's all about learning by doing. It focuses on how intelligent agents should take actions in an environment to maximize cumulative rewards. In a RL setting, an agent interacts with its environment by observing the current state, taking an action, and receiving a reward signal based on the outcomes of those actions. The goal of the agent is to learn a strategy (called a policy), which maps states to actions that maximize the expected cumulative reward over time. Based on the reward feedback attributed by the underlying reward function of this environment, the agent learns how to perform the task of interest through trial and error. It's kind of like learning to play a game: you try different moves, see what works and what doesn't, and gradually improve your skills until you can beat the competition. [7]

2.4.3 Reward shaping

Rewards are the fuel of RL. But designing them is tricky: if they're too sparse, the agent never learns; if they're misleading, the agent learns the wrong thing. Reward shaping is the art of crafting signals that guide the agent toward the desired behavior without accidentally teaching it bad habits (like spinning in circles forever because it found a tiny reward there).

2.4.4 Exploration vs Exploitation

The exploration vs. exploitation trade-off resides on a spectrum: on the one end, if we exploit, we obtain short-term rewards by reaching good deployment performance given what we have known so far; on the other end, if we explore, we obtain long-term rewards because we are building more reliable models by collecting knowledges in uncertain problem spaces. RL algorithms can navigate between these two ends to get a dynamic strategy according to the problem and performance. It is adaptive, so it can continually learn to accommodate for the changes in the data. It is about optimizing for the expected future rewards, so it can innately predict or plan ahead for future events. Since it is mostly driven by its reward feedback, it is also generalizable to new tasks or uncertainty patterns by modifying its reward structure. [7]

Think about this:

- If we exploit entirely, we make long-term sacrifices by potentially missing unknown optimal actions, because we are only choosing the best action given the current information.
- If we explore entirely, we make short-term sacrifices by missing out on known rewards, because we are always gathering more information rather than capitalizing on what we've learned.

[7]

2.4.5 Curriculum Learning

Humans don't learn calculus before arithmetic, and robots shouldn't tackle the hardest environments first. Curriculum learning is about starting simple and gradually increasing difficulty. For this project, this means letting the agent train in small, easy maps before facing larger, more complex ones. The step-by-step progression makes learning faster and more stable.

2.4.6 Q-learning

Q-learning is one of the most fundamental algorithms in reinforcement learning. It is an off-policy, value-based method that estimates the optimal action-value function by applying the Bellman equation iteratively. In this project, Q-learning was implemented with a neural network (a DQN-style architecture) that approximates $Q(s, a)$. The update rules used are:

Bellman update:

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

Target values:

$$y_i = \begin{cases} r_i & \text{if } d_i = 1 \\ r_i + \gamma \max_{a'} Q_\theta(s'_i, a') & \text{if } d_i = 0 \end{cases}$$

Loss function:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \big(Q_\theta(s_i, a_i) - y_i\big)^2$$

Here d_i is the termination flag, r_i the reward at step i, and γ the discount factor. This formulation allows the neural network to approximate the Q-values and improve action selection over time.

Key references: [3] [4] [5] [6] [7] [8]

2.4.7 PPO

Proximal Policy Optimization (PPO) is a modern reinforcement learning algorithm based on policy gradients. In this project, PPO was adopted because of its stability and efficiency when training agents in complex environments. The method combines policy optimization with constraints that prevent updates from deviating too much from the previous policy. The core formulas used are:

Advantage estimation (GAE-λ):

$$\delta_t = r_t + \gamma(1 - d_t)V(s_{t+1}) - V(s_t)$$

$$\hat{A}_t = \delta_t + \gamma\lambda(1 - d_t)\hat{A}_{t+1}$$

Returns:

$$R_t = \hat{A}_t + V(s_t)$$

Probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

Clipped surrogate objective:

$$L^{\text{CLIP}}(\theta) = E\Big[\min\big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t\big)\Big]$$

Value loss:

$$L^{\text{VF}} = \big(V_\theta(s_t) - R_t\big)^2$$

Entropy bonus:

$$L^{\text{ENT}} = E[H(\pi_\theta(\cdot \mid s_t))]$$

Final objective:

$$L(\theta) = -L^{\text{CLIP}}(\theta) + c_v L^{\text{VF}} - c_s L^{\text{ENT}}$$

Key references: [1] [2] [6] [7] [8]

Chapter 3

Environment

3.1 Purpose

The purpose of this project is to simulate the behavior of a robotic vacuum cleaner navigating within an indoor environment. To achieve this, two complementary environments must be considered:

- A **continuous "real-world" simulation**, which represents the physical world where the robot moves. In this project, the term "real" is placed in quotation marks because the environment is itself a digital approximation of reality.
- A **discrete internal representation**, constructed by the robot as it senses the world through its virtual sensors. This abstract view enables the agent (the neural network controlling the robot) to reason and act efficiently.

The actual robot interacts with the continuous simulated environment, while the learning agent operates on the discrete abstraction built by the robot. This duality—between the underlying world and the robot's internal map—is central to how autonomous agents perceive and act.

3.2 Agent–environment interaction

3.2.1 Continuous vs discrete perspectives

It is important to highlight the dual nature of the simulation:

- In the *continuous polygonal environment*, the robot's physical motion, collisions, and sensor measurements are simulated.
- In the *discrete occupancy grid*, the agent's neural network makes decisions and plans actions.

The robot acts as the interface between these two layers, translating continuous dynamics into discrete information and discrete decisions into continuous movements.

3.2.2 Relevance to reinforcement learning

This separation mirrors how reinforcement learning is applied in robotics: the agent does not directly perceive the world in its raw, continuous complexity, but rather through an abstraction designed for tractability. The richness of the environment ensures that learning remains non-trivial, while the discrete representation makes the problem computationally feasible.

3.3 Real-world simulation

3.3.1 Assumptions

In order to simplify the modeling and focus on essential dynamics, we assume the following conditions:

- The robot operates in indoor, enclosed environments such as apartments, warehouses, or shops. The accessible area is limited to a few hundred square meters at most.

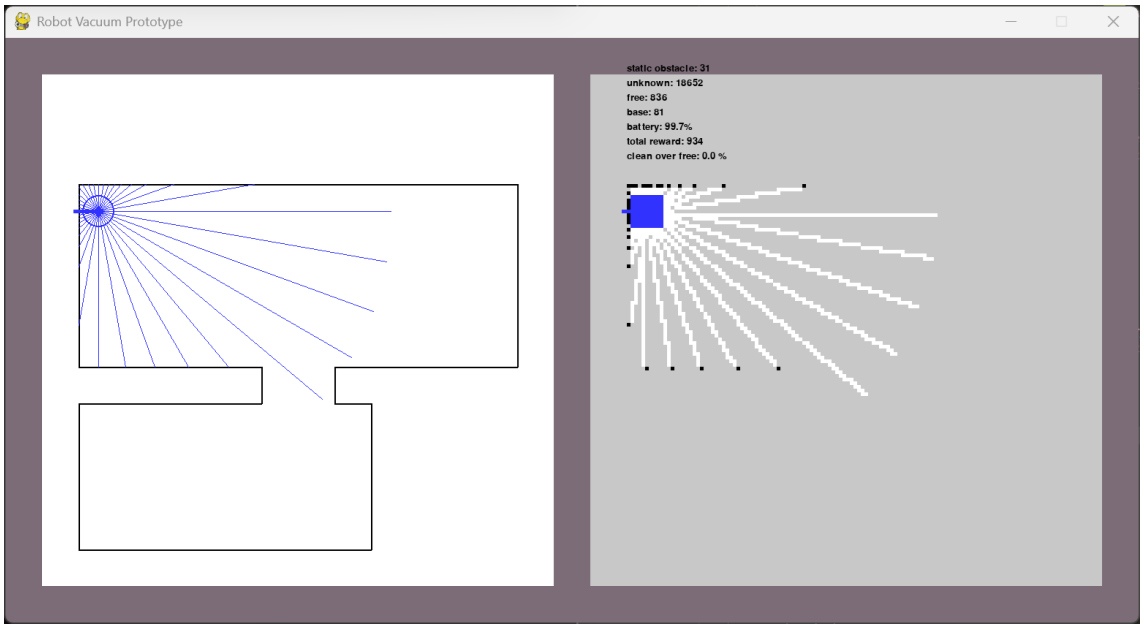

Figure 3.1: Snapshot of the environment and LiDAR beams after initialization (step 0).

- The environment is entirely two-dimensional: the robot does not require three-dimensional perception for this task.
- At this stage, the robot is deployed only in spaces free of people or animals, avoiding the complexities of dynamic obstacles.

These assumptions allow us to create a tractable yet realistic setting that captures the main challenges of navigation and cleaning without overcomplicating the simulation.

3.3.2 Environment structure

The "real-world" is simulated as a continuous two-dimensional space bounded by a rectangular region. Each environment corresponds to a simplified floor plan of an indoor space. The only structural elements are the internal sides of walls, which are represented as segments forming one polygon.

This geometric representation is particularly convenient: it allows efficient collision detection, LiDAR ray tracing, and straightforward rendering. Examples are shown on left-hand side in Figures 3.1 and 3.2, where the environment is displayed alongside the robot's exploration.

3.4 Discrete abstraction of the environment

3.4.1 Assumptions of the discrete environment

To make the discrete abstraction both realistic and tractable for reinforcement learning, a set of assumptions is introduced regarding the robot and its interaction with the environment:

- The robot is modeled as a common domestic vacuum cleaner with a circular body of diameter **45 cm**.
- A **LiDAR sensor** is mounted at the exact center of the robot's top surface. It is vertically aligned (perpendicular to the floor) and rotates a full $360°$, providing distance measurements in all directions.
- The robot is equipped with **collision sensors** distributed along its entire perimeter. Since the body is circular in the 2D representation, this guarantees uniform coverage.
- The robot can **rotate by any angle** around its central axis, changing its orientation continuously.
- The robot can **move only forward** relative to its current orientation, and the distance traveled can be any positive real value.

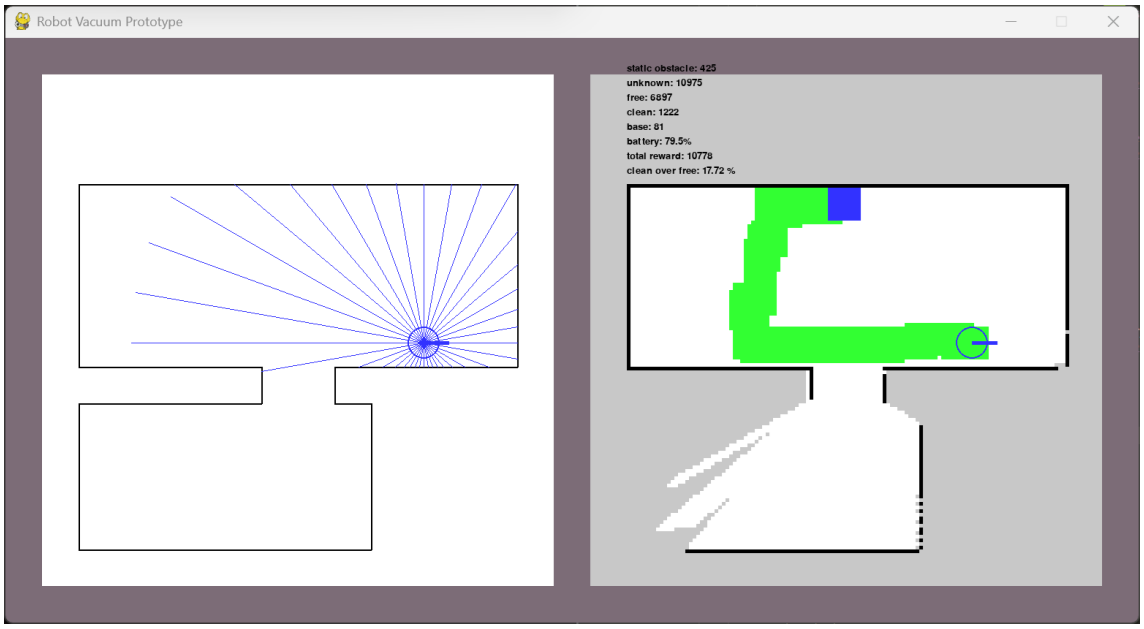

Figure 3.2: Snapshot of the environment and LiDAR beams after exploration (step n).

- Collisions with walls are possible and strictly enforced: the robot **cannot cross walls** under any circumstance.
- The floor may contain **light dirt** such as crumbs, dust, or soil particles.
- Cleaning occurs automatically whenever the robot's body covers a given area. Formally, the robot cleans all dirt located inside the square bounding box that circumscribes its circular body.
- The robot has **limited autonomy**, constrained by a finite battery. Energy is consumed as the robot moves.
- Each episode begins with the robot placed on its **charging base**.
- The charging base is positioned randomly along a wall segment, always adjacent and perpendicular to the wall.
- The robot's initial orientation is set to be **opposite to the wall** on which the charging base is located, so that it starts facing into the environment.
- An episode terminates when the robot either **returns to base** successfully or **runs out of battery**.
- During the episode, the robot **builds, stores, and updates** an internal representation of the surrounding environment using its LiDAR sensor.

These assumptions provide a balance between realism and simplicity: the robot is constrained in ways that mirror commercial vacuum cleaners, while the representation remains manageable for simulation and learning. These assumptions allow the agent to reason on a structured, finite state space without losing essential geometric information from the continuous environment.

3.4.2 Environment structure

The robot's abstract representation of the surrounding world is a discrete two-dimensional environment bounded by a rectangular region. The environment is discretized into a grid of square cells, each with side length $5\,\mathrm{cm}$. Every cell is assigned exactly one of the following labels:

- *Obstacle*
- *Unknown*
- *Free*

- *Clean*
- *Base*

Initially, all cells are labeled *Unknown* (except the cells that represent the charging base), and they may be updated at every robot step. Time is discrete and indexed by robot steps. At each step, the robot selects an orientation angle from $\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\}$ (corresponding to *right*, *down*, *left*, *up*) and moves forward by exactly one cell, cleaning the surface beneath its body. After completing the movement (unless a collision occurs), the robot uses its LiDAR to update the abstract representation of the environment. See the right-hand side of Figures 3.1 and 3.2.

3.4.3 Mapping process

Each LiDAR beam provides a measurement of distance until the first obstacle. This information is discretized into grid cells: the cell at the obstacle's location is marked as occupied, while the cells along the ray are marked as free. Over time, this process fills the occupancy grid, gradually transforming unknown regions into known free or occupied areas.

The robot's body also updates the grid: the cells under its footprint are marked as cleaned. This dual update (from sensing and cleaning) ensures that the internal map captures both geometry and task progress.

An additional mechanism was introduced to improve the abstraction of obstacles. Due to discretization into $5\,\mathrm{cm}$ cells, walls may sometimes appear further away than they actually are, leaving a row of *free* cells that can cause unexpected collisions. To mitigate this, we apply a **one-sided padding** of obstacles: for each wall segment, we expand its rasterization by exactly one additional cell only on the *lower-index side* along each axis (i.e., toward the origin). In practice, this yields an effective two-cell thickness *only in that direction*, while the opposite side remains unchanged. This asymmetric dilation slightly reduces navigable space but reliably prevents collisions caused by under-approximation in the grid.

3.4.4 Base station placement

The base station, where the robot recharges, is placed at the edge of the environment but always inside the polygonal structure. Its location is automatically chosen along a wall segment so that it remains accessible while still being an integral part of the robot's abstract map.

Returning to base is not trivial: the agent must learn to navigate efficiently back to this point once cleaning is complete or the battery runs low.

Chapter 4

Agent

4.1 Introduction

The *agent* is the decision-making component of the project: it observes the environment through the robot's sensors, selects actions, and updates its behavior from feedback in the form of rewards and penalties. Two reinforcement-learning approaches were implemented and compared: a classical **Q-learning** agent and a **PPO (Proximal Policy Optimization)** agent. They control the same simulated robotic vacuum cleaner but differ in how they represent knowledge and learn from experience. This chapter outlines their state and action interfaces, learning objectives, and high-level behavior.

4.2 State representation

At each step, the environment is encoded into a fixed-length vector, that blends local and global information, this is all the agent perceives, divided into 4 macro groups:

- **Scalar values:**
  - robot (x, y) discrete positions on the grid;
  - base (x, y) discrete positions on the grid;
  - base (x, y) discrete positions of the last point of collision,
    if the last action led to a collision,
    otherwise $(0, 0)$;
  - collision flag $\in \{0, 1\}$ indicating if the last action led to a collision
  - orientation, as $\sin\theta$ and $\cos\theta$;
  - battery level $\in [0, 1]$;
  - the global label counts of the occupancy grid:
    - the number of cells labeled as "obstacle";
    - the number of cells labeled as "unknown";
    - the number of cells labeled as "free";
    - the number of cells labeled as "clean".
- **LiDAR distances:**
  a vector of 36 distances $\in [0, max\_LiDAR\_distance]$, one for each laser beam.
  See the blue radial lines in figure 4.1, on the left.
- **A local subgrid patch centered on the robot:** A 33-cell square submatrix of the occupation grid, centered on the robot's center.
  Represented by the area under the semi-transparent blue box in the figure 4.1, on the right.
- **Side-views** 4 vectors, one for each side of the robot's minimum bounding box, each containing:
  9 vectors, one for each cell on the side of the robot's bounding box, each containing
  4 copulas, composing as follows:

- (number of cells labeled ”obstacle”, number of cells of distance between the side of the robot’s bounding box and the first cell labeled ”obstacle”)
- (number of cells labeled ”unknown”, number of cells of distance between the side of the robot’s bounding box and the first cell labeled ”unknown”)
- (number of cells labeled ”free”, number of cells of distance between the side of the robot’s bounding box and the first cell labeled ”free”)
- (number of cells labeled ”clean”, number of cells of distance between the side of the robot’s bounding box and the first cell labeled ”clean”)

They provide a simplified, but long-range, representation of the information in the areas under the four semi-transparent blue rectangles projected by the robot in figure 4.1, on the right.

This composite encoding gives the agent both a coarse global context and fine-grained local cues. Simplified graphic representation on figure 4.1.

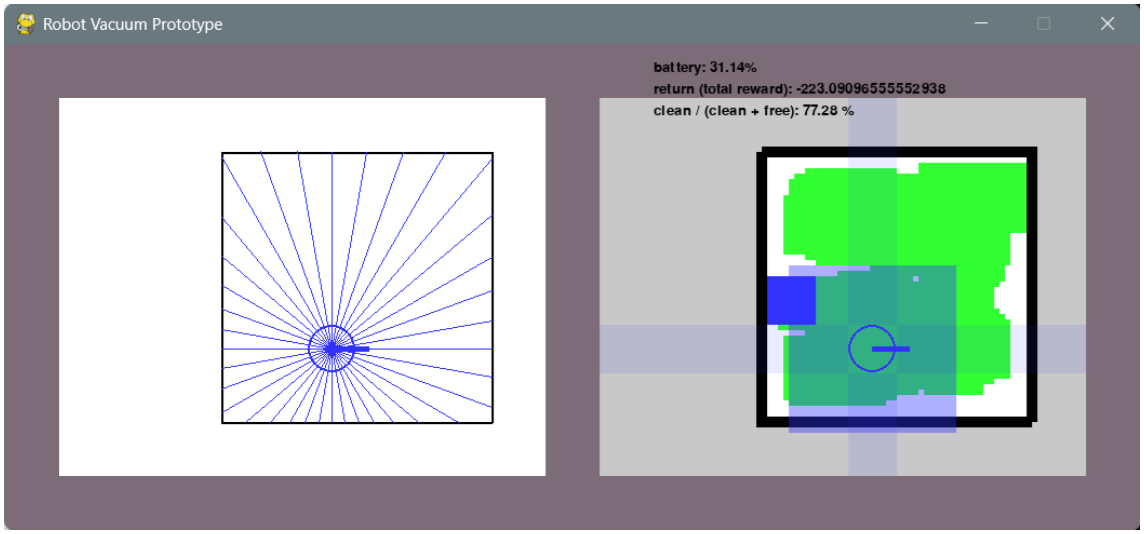

Figure 4.1: Visual rappresentation of LiDAR rays, local subgrid path and side-views.

4.3 Action model

Definition. By *action model* we mean the formalization of the set of commands available to the agent and their semantics in the simulated environment: how an action chosen by the policy is interpreted by the environment and which state variables are updated as a result (position, orientation, battery, collision signals, etc.).

Action space. Both agents (Q-learning and PPO) share a discrete action space of four grid-aligned cardinal moves:

$$\mathcal{A} = \{\textsc{Right},\ \textsc{Down},\ \textsc{Left},\ \textsc{Up}\}.$$

”At a high-level” we can see the robot advance along the occupancy grid, in one of the four cardinal directions, one cell at a time. This mechanism is implemented by changing the robot’s orientation according to the action selected by the policy, and by making the robot move forward in the ”real world” by a distance equal to the side of the cells.

In terms of elementary displacements, each action induces a translation vector $\Delta a \in \{(1,0),(0,1),(-1,0),(0,-1)\}$ on the occupancy grid.

Transition semantics. Given the state s_t (which includes all the information introduced in §4.2) and an action $a_t \in \mathcal{A}$, the environment dynamics apply the following high-level rule:

$$s_{t+1} = T(s_t, a_t), \quad \text{where} \quad \begin{cases} \wedge \begin{cases} \text{the robot does not move} \\ \text{the collision flag is set} \\ \text{the collision coordinates are stored} \end{cases} & \text{if the target cell is an } \textit{obstacle} \\ \wedge \begin{cases} \text{the position is updated to the adjacent cell consistent with } a_t \\ \text{the collision flag is reset.} \end{cases} & \text{otherwise} \end{cases}$$

Orientation, battery level and map labels (e.g. *free*/*clean*) are updated according to the environment rules; these operational details do not alter the action space but only the resulting state components.

Action parametrization in RL agents. The action model is the same for both algorithms; what changes is the way in which the agent decides which action to perform.

- **Q-learning.** The agent maintains an estimate of the action-value function $Q_\phi(s, a)$, which measures the expected return of taking action a in state s. In practice, the Linear-QNet outputs four real values, one for each discrete action $a \in \mathcal{A}$. The action is then chosen by

  $$a_t = \arg\max_{a \in \mathcal{A}} Q_\phi(s_t, a),$$

  with occasional random exploration according to the ε-greedy rule. This means the agent mostly selects the action with the highest predicted value, but sometimes explores alternatives.

- **PPO (Proximal Policy Optimization).** Here the agent directly learns a *policy* $\pi_\theta(a|s)$, i.e. a probability distribution over actions. The neural network produces four logits $z_\theta(s) \in R^4$, one per action, which are converted into probabilities through the softmax function:

  $$\pi_\theta(a|s) = \frac{e^{z_\theta(s,a)}}{\sum_{a' \in \mathcal{A}} e^{z_\theta(s,a')}}$$

  The action a_t is then sampled from this distribution, so that higher-probability actions are chosen more often, while still allowing occasional less likely moves. Additionally, PPO estimates a state-value function $V_\theta(s)$, used as a baseline to stabilize policy gradient updates.

Independence from state encoding. The *Single-MLP* variant flattens the entire state into a single vector; the *Multi-encoder* variant uses dedicated encoders for the local patch (2D CNN), LiDAR (1D CNN), side-views and scalars, fused into an MLP trunk. In both cases, the action interface remains the same four-move set $\mathcal{A}$: only the state representation feeding the policy changes, which in turn affects the quality of action selection.

Algorithm 1 ApplyAction(s_t, a_t) – abstract scheme

$(x, y) \leftarrow$ position in s_t
$\Delta \leftarrow$ vector associated with a_t ▷ $(1,0), (0,1), (-1,0), (0,-1)$
if cell$(x + \Delta_x, y + \Delta_y)$ is OBSTACLE **then**
    set COLLISION_FLAG$\leftarrow 1$;
    set LAST_COLLISION_POS$\leftarrow (x + \Delta_x, y + \Delta_y)$;
else
    set COLLISION_FLAG$\leftarrow 0$;
    set LAST_COLLISION_POS$\leftarrow (0, 0)$;
    set $(x', y') \leftarrow (x + \Delta_x, y + \Delta_y)$;
end if
Update orientation/battery/labels according to environment rules
return s_{t+1} with position (x', y') and updated signals

4.4 Q-learning agent

The Q-learning agent is a value-based learner that approximates the action–value function $Q(s, a)$ with a feed-forward neural network (*Linear-QNet*). The network outputs four Q-values, one for each discrete action (*right*, *down*, *left*, *up*).

Action space and learning

The action set is discrete with four moves aligned to the grid. The agent follows an ε-greedy policy: with probability ε it explores, otherwise it exploits the current Q-estimates. Learning proceeds by sampling transitions from a replay buffer and minimizing the mean-squared error between predicted and target Q-values computed with the Bellman update. Gradient clipping is used to stabilize training. Over time, the agent learns to improve coverage, reduce unnecessary collisions, and return to the base before the battery is depleted.

4.5 PPO agent

The PPO agent is a policy-gradient learner that directly optimizes a stochastic policy $\pi_\theta(a \mid s)$ together with a value function $V_\theta(s)$. Two complementary configurations were implemented.

4.5.1 Single-MLP configuration

In the first setup, the entire state is flattened into a single, normalized feature vector (positions, orientation, battery, collision flags, LiDAR, label counts, local patch, and side-views). A multi-layer perceptron processes this vector and produces:

- **policy logits** over the four discrete actions (defining a categorical distribution);
- a **state-value estimate** $V(s)$ used as a baseline.

Training is on-policy: rollouts are collected, advantages are computed with GAE-λ, and the clipped PPO surrogate objective is optimized with minibatches, combining policy loss, value loss, and an entropy bonus for exploration.

4.5.2 Multi-encoder configuration (4 encoders + trunk MLP)

In the second setup, the state is decomposed into four modalities, each handled by a dedicated encoder before fusion:

- a **patch CNN** that processes a 31×31 local occupancy-grid window (one-hot over labels);
- a **LiDAR encoder** (either an MLP or a lightweight 1D CNN) for the 36 beam distances;
- a **side-views MLP** that summarizes labels counts and first-hit distances around the robot;
- a **scalars MLP** that aggregates battery, positions, orientation, collision flags, and global grid counts.

The resulting feature vectors are concatenated and passed through a shared trunk MLP, which branches into a policy head (action logits) and a value head ($V(s)$). This modular design lets each modality be processed with an appropriate inductive bias (e.g., spatial locality for the patch via convolutions), improving representation quality without changing the action space or the PPO training recipe.

4.6 Comparison and discussion

Working with both agents gave me a clear picture of their differences.

Q-learning learns by updating its own estimates of future rewards and reusing past experiences from a replay buffer. This makes it relatively sample-efficient, but also fragile: if the estimates are off, the whole learning process can drift in the wrong direction.

PPO, on the other hand, relies on fresh data collected on-policy and applies small, clipped updates to the policy. This usually makes training more stable and smoother, but it also means that new trajectories must constantly be generated, which is more expensive in terms of computation.

Since I only had access to a CPU, both methods hit their limits quickly. Q-learning made better use of each experience by replaying it, but still needed large batches to stabilize. PPO was more reliable, especially with the multi-encoder architecture, but the cost of running rollouts and multiple optimization epochs became heavy without a GPU.

Another key difference is how they handle the state. Flattening everything into a single vector (used in Q-learning and the PPO MLP version) is simple, but it mixes very different types of information. The PPO multi-encoder model, instead, keeps spatial and numerical inputs separated: 2D CNN for the occupancy patch, dedicated 1D CNN for LiDAR, side views, and scalars. This structured design made it easier for the agent to follow walls, anticipate obstacles, and find its way back to the base.

Exploration also changes between the two. Q-learning uses the classic ε-greedy trick: sometimes act randomly, otherwise trust the network. PPO adds an entropy bonus, which keeps the policy more random early on and then gradually sharper over time. Combined with reward shaping, both approaches managed to avoid trivial behaviors like spinning in circles.

Chapter 5

Training process

5.1 Overview

This chapter describes the learning process of the **PPO multi-encoder agent**, which represents the latest and most performant configuration developed during the project. Earlier experiments with Q-learning and PPO-MLP informed the design, but only the encoder-based agent is considered here, as it proved to be the most structured and effective. The following sections detail how the training loop is organized, how rewards are shaped, how curriculum learning and data augmentation are applied, and how PPO rollouts are managed.

5.2 Training loop

Each training episode follows the standard reinforcement learning cycle:

1. **Environment reset:** at the beginning of an episode, the robot is placed in a randomly selected training map (with random rotation and base placement, see Curriculum Learning below).
2. **State encoding:** the robot's perception is processed by four encoders (patch 2D CNN, LiDAR 1D CNN encoder, side-views, scalars).
3. **Action selection:** the policy network outputs logits for the four discrete actions; an action is sampled stochastically.
4. **Environment step:** the action is executed, the environment updates position, battery, and occupancy grid, and a shaped reward is returned.
5. **Buffering:** the transition $(s, a, r, \log \pi(a|s), V(s))$ is added to the PPO buffer.
6. **Update:** when the buffer reaches the rollout size (n steps) or the episode terminates, the agent performs a PPO update using GAE-λ and the clipped surrogate objective.

5.3 Reward shaping

The reward function focuses primarily, but not exclusively, on the tasks of cleaning cells labeled "free" and avoiding collisions with cells labeled "obstacle." However, given the complexity of the model (in this case, we are referring to the discrete model, created internally in real time by the robot), the need arose to enrich the reward function with additional information derived from reward shaping. Below, the reward function and reward shaping are introduced together, just as they are perceived by the agent, defining them first verbally and then formally.

5.3.1 Verbal introduction

These rules together balance short-term exploration and cleaning with the long-term objective of finishing the task and returning safely.

- **Exploration bonus:** proportional to the reduction of unknown cells in the occupancy grid. This drives the robot to map unseen areas instead of circling locally.

- **Cleaning bonus:** awarded for each newly cleaned cell. This ensures that forward motion is directly tied to useful progress.

- **Edge cleaning bonus:** extra reward if the robot cleans a cell adjacent to a wall. This encourages coverage near obstacles.

- **Penalty for inactivity:** if no new cells are cleaned for several steps, a penalty is applied. This prevents wasting energy without making progress.

- **Penalty for stalling:** if the robot remains on the same cell for more than one step, an increasing penalty is applied. This discourages the agent from "stalling" in place without progressing through the environment.

- **Battery-aware penalties:** when more than 80% of the environment is already clean, or the battery drops below 20%, a small penalty is applied at every step to discourage aimless wandering and encourage a return to base.

- **Nudge to base:** once cleaning is nearly complete or battery is low, the agent receives a positive incremental reward if it moves closer to the base. If battery $\leq$10%, the shaping reward is doubled to stress urgency.

- **Successful return:** if the robot covers at least half of its footprint over the base when cleaning $\geq$80% of free cells or battery $< 20\%$, the episode ends with a strong positive reward proportional to the cleaned fraction.

- **Failure penalty:** if the battery is depleted before returning, a large negative reward is assigned.

- **Collision penalty:** each collision with a wall results in a small negative reward.

5.3.2 Formal definition

Below, a simplified definition of the reward function and reward shaping, together.

$$
Reward(s_{i-1}, a_i, s_i) = \wedge \begin{cases}
\textit{bonus when it returns to base} \\
+10.0 \cdot clean\_over\_free \quad \text{if} \quad \wedge \begin{cases} percent\_on\_base > 0.50\% \\ \vee \begin{cases} clean\_over\_free >= 0.99\% \\ battery < 0.20\% \end{cases} \end{cases} \\
\\
\textit{bonus nudge to base} \\
\vee \begin{cases} +2.0 \cdot \Delta_{base} \quad \text{if} \quad battery <= 10\% \\ +1.0 \cdot \Delta_{base} \quad \text{if} \quad battery > 90\% \end{cases} \quad \text{if} \quad \wedge \begin{cases} clean\_over\_free >= 0.99\% \\ battery <= 0.20\% \end{cases} \\
\\
\textit{bonus for following straight trajectories} \\
+0.25 \cdot streak \\
\\
\textit{bonus exploration} \\
+(-\Delta_{unknow}/450) \\
\\
\vee \begin{cases} \wedge \begin{cases} \textit{bonus cleaning} \\ +\Delta_{clean}/90 \\ \\ \textit{bonus cleaning edge} \\ +edge\_count/255 \end{cases} \quad \text{if} \quad \Delta_{clean} > 0 \\
\\
\textit{malus not cleaning} \\ -0.25 \quad \text{otherwise} \end{cases} \\
\\
\textit{malus for inefficient cleaning strategy} \\
-0.25 \quad \text{if} \quad clean\_over\_free > 80\% \\
-0.25 \quad \text{if} \quad battery < 20\% \\
\\
\textit{malus anti-stuck} \\
-0.25 \cdot same\_cell\_steps\_malus\_coeff \\
\\
\textit{malus when a collision occurs} \\
-1.0 \quad \text{if a collision occurs} \\
\\
\textit{malus when it run out of battery} \\
-10.0 \quad \text{if} \quad battery = 0
\end{cases}
$$

Where:

- a_i is the current action;
- s_i is the current state;
- s_{i-1} is the previous state;
- $battery \in [0, 1]$ is the batter level;

- $percent\_on\_base \in [0, 1]$ represents how much the surface of the minimum bounding box in which the robot is embedded overlaps with the square of the base;
- $\Delta_{base} \in [-\inf, +\inf]$ represents how much the robot has moved closer to (or further away from) the base compared to the previous state;
- $\Delta_{unknown} \in N$ is the difference in the number of cells labeled "unknown" compared to the previous state;
- $\Delta_{clean} \in N$ is the difference in the number of cells labeled "clean" compared to the previous state;
- $clean\_over\_free \in [0, 1] = \frac{\#\text{ "clean" cells}}{\#\text{ "clean" cells}+\#\text{ "free" cells}}$ represents the percentage of clean surface, compared to the currently known and obstacle-free surface;
- $edge\_count \in N$ is the sum of the sides adjacent to an obstacle of the cleaned cells in the current state, so it represents how much the freshly cleaned surface borders with the obstacles;
- $streak \in N_{>1}$ is the last number of consecutive actions performed in the same direction and same direction receiving a positive reward for each turn, so it measures how many steps the robot has taken in a straight line without receiving a penalty (i.e. if the total reward for each turn is positive);
- $same\_cell\_steps\_malus\_coeff \in N_{>1}$ is the last number of consecutive actions remaining in the same position with respect to the previous state.

5.4 Curriculum learning and data augmentation

Training maps are organized into difficulty levels. The agent begins on the simplest layouts; once its average success rate (cleaning $\geq$80%, battery remaining $\geq$20%, and zero collisions) exceeds a threshold on test maps, the next level is unlocked.

Within each difficulty level, **map sampling is randomized every episode**:

- one of the available maps is chosen at random;
- the map is rotated by a random multiple of $90°$;
- the base station is placed at a random valid position along a wall.

This simple augmentation strategy multiplies the variety of training scenarios, forcing the agent to learn generalizable behaviors instead of memorizing layouts.

The figure 5.1 shows the draft maps for the first four difficulty levels. The ones crossed out in red were eliminated because they contained too narrow points where the base would get stuck. The maps highlighted in blue are reserved for testing, while the remaining ones are those used in the training phase. The fifth and final difficulty level was intended to contain the actual floor plans of three two-room apartments.

5.5 Rollouts and updates

The PPO agent collects trajectories in **rollouts** of up to n steps. During a rollout, every interaction is stored in the buffer: state encodings from all four encoders, actions taken, log-probabilities, rewards, value predictions, and done flags. When the buffer is full (or when an episode terminates), the agent computes advantages using GAE-λ, normalizes them, and optimizes the policy.

The PPO objective balances three terms:

- the clipped surrogate policy loss,
- the value loss (mean squared error between predicted and target returns),
- and an entropy bonus to maintain exploration.

Multiple epochs and minibatches are used to refine the parameters before the buffer is cleared and the process restarts. This cycle repeats until the agent reaches the highest curriculum level or training is manually stopped.

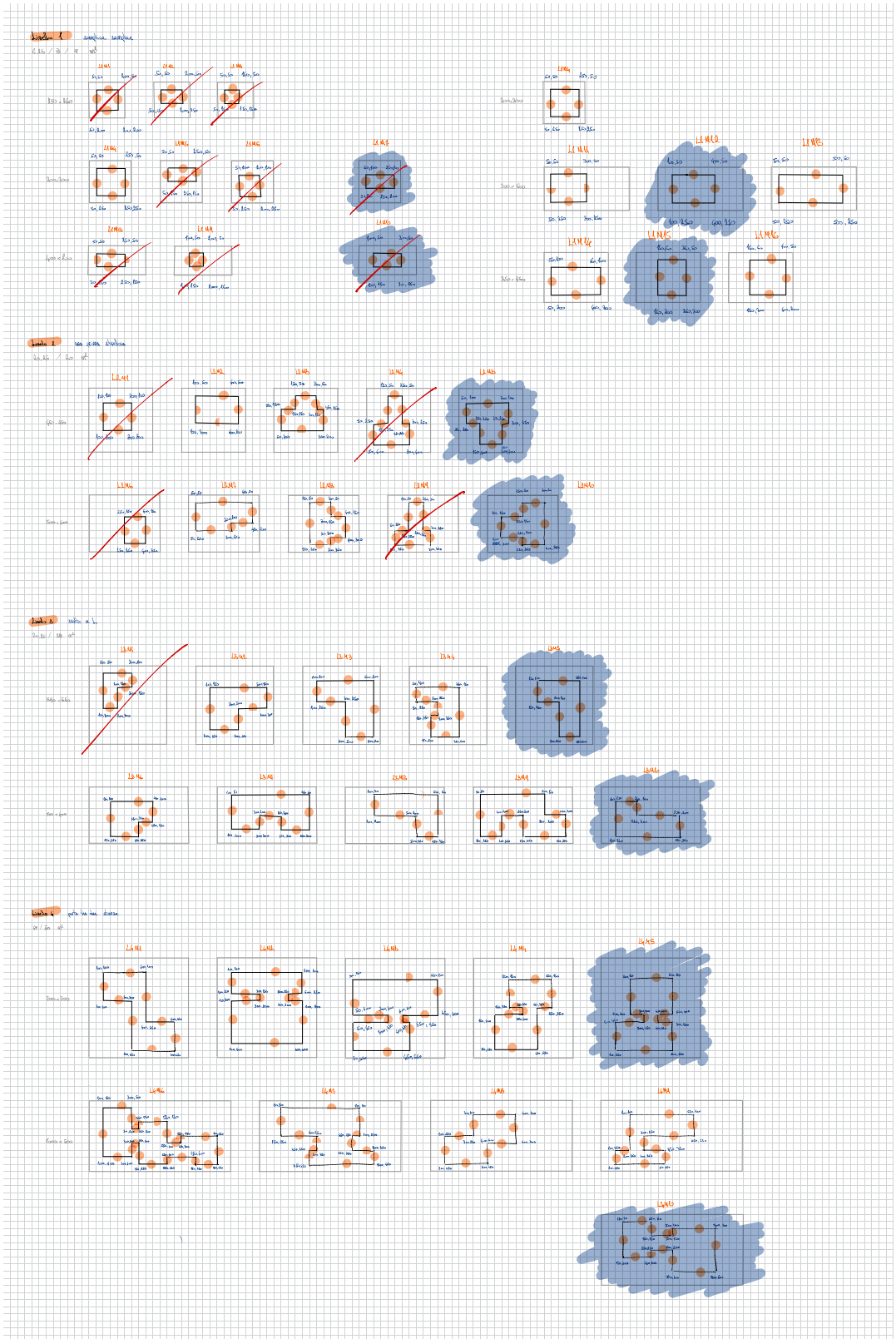

Figure 5.1: Maps draft (the first four level of difficulty)

5.6 Monitoring and evaluation

During training, several metrics are tracked:

- **Return (score):** cumulative reward obtained in an episode.
- **Mean return:** moving average of episode returns.
- **Collisions:** number of wall impacts per episode.
- **Battery usage:** percentage of battery remaining at the end of an episode.
- **Cleaned area:** percentage of free cells that have been cleaned.

Results are plotted live, figure 5.2, and saved at checkpoints whenever a new level of the curriculum is unlocked.

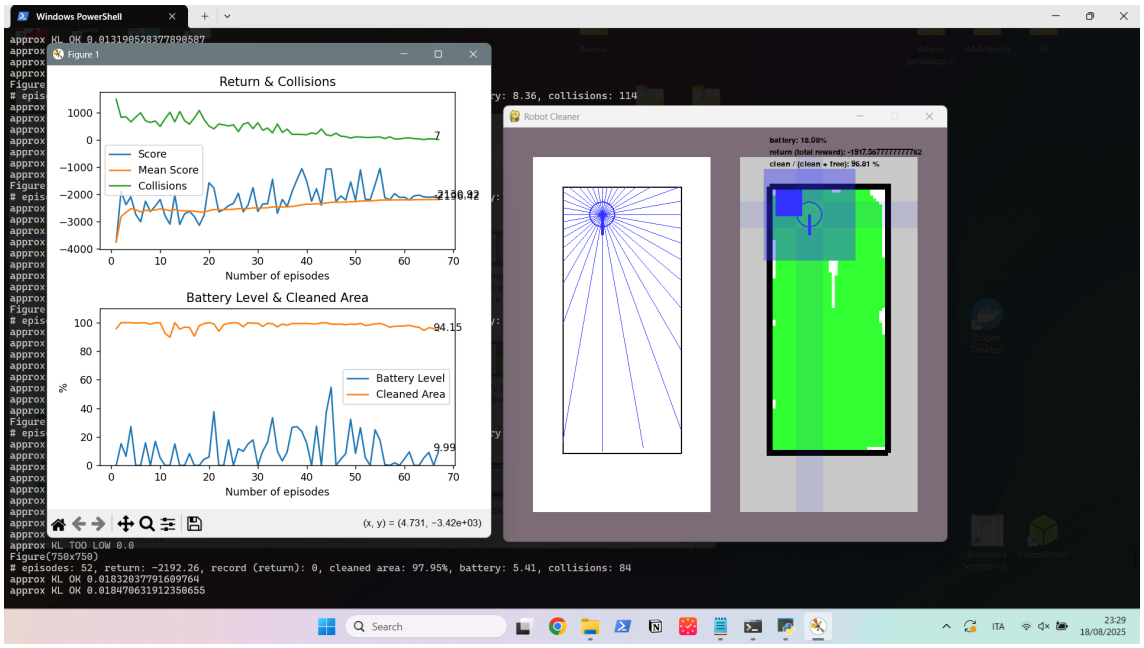

Figure 5.2: Example of live plotting

5.7 Training outcome

Despite being limited to CPU-only resources, the encoder-based PPO agent developed structured cleaning strategies, meaningful exploration, and base-return behavior under energy constraints. The combination of curriculum learning, rich reward shaping, and structured state encoding proved essential to achieve these results.

Chapter 6

Results and Future Work

6.1 Results

Looking back at this journey, the project succeeded in building a full simulation framework and training pipeline for a robotic vacuum cleaner governed by reinforcement learning. The PPO multi-encoder agent demonstrated meaningful behaviors: systematic exploration of new areas, consistent cleaning, and attempts to return to base under energy constraints. Curriculum learning, reward shaping, and modular state encoding all proved to be essential ingredients.

At the same time, the project could not be brought to full completion due to computational limitations. Training required long wall-clock times: with the CPU of my personal laptop, each meaningful adjustment to the architecture, reward function, or hyperparameters demanded at least **12 hours** of training before its effects could even be evaluated. Without access to GPU acceleration or HPC resources, it was not possible to unlock the higher levels of curriculum maps or converge to stable long-term policies.

Nevertheless, reaching this point has been remarkable. It has been both challenging and fascinating to learn reinforcement learning in practice, to design environments, to integrate encoders, and to observe how an agent can gradually develop purposeful behaviors from scratch. The journey itself—understanding the theories, translating them into code, and watching them come alive—has been the most valuable result.

Below I report some training outcomes to illustrate the continuous progression observed during the project.

6.1.1 Results of Q-learning

The initial experiments with the Q-learning agent quickly revealed its limitations in this setting. The learned policies tended to collapse into a single repeated action, resulting in poor exploration and a rapid decline in performance. Figures 6.1 and 6.2 illustrate these outcomes: rewards decreased steadily, and the agent eventually converged to trivial, ineffective behavior.

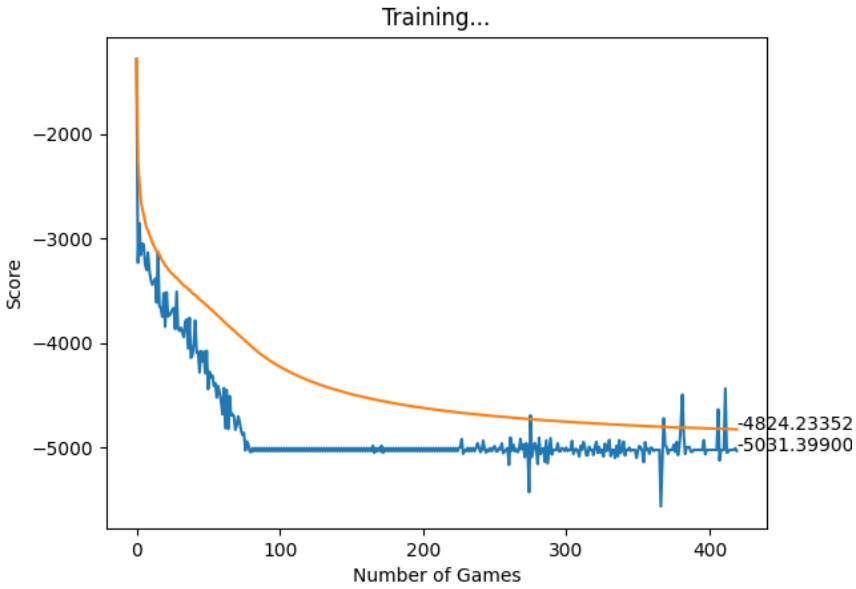

Figure 6.1: Training curve (returns) of the Q-learning agent. The trend shows rapid deterioration due to policy collapse.

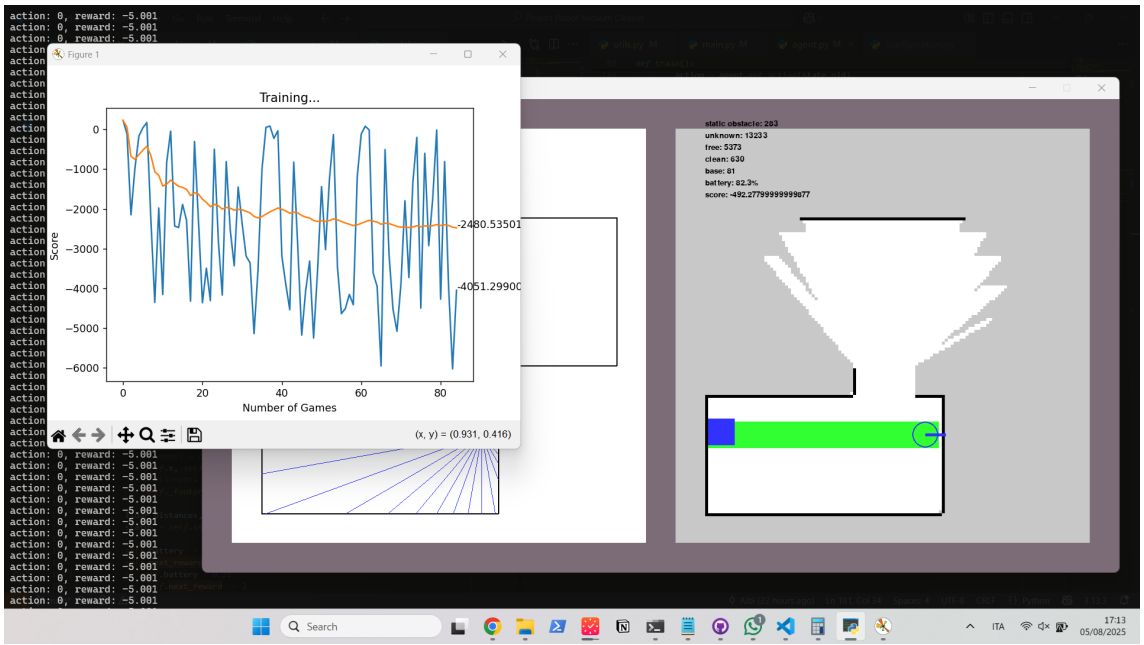

Figure 6.2: Example of a Q-learning policy collapsing into a single repeated action, preventing meaningful exploration.

6.1.2 Results of PPO (multi-encoder configuration)

The PPO multi-encoder agent initially suffered from issues related to suboptimal reward shaping and return-to-base mechanics. As shown in Figure 6.3, early training runs were unstable, and the agent failed to generalize cleaning and base-return behavior consistently.

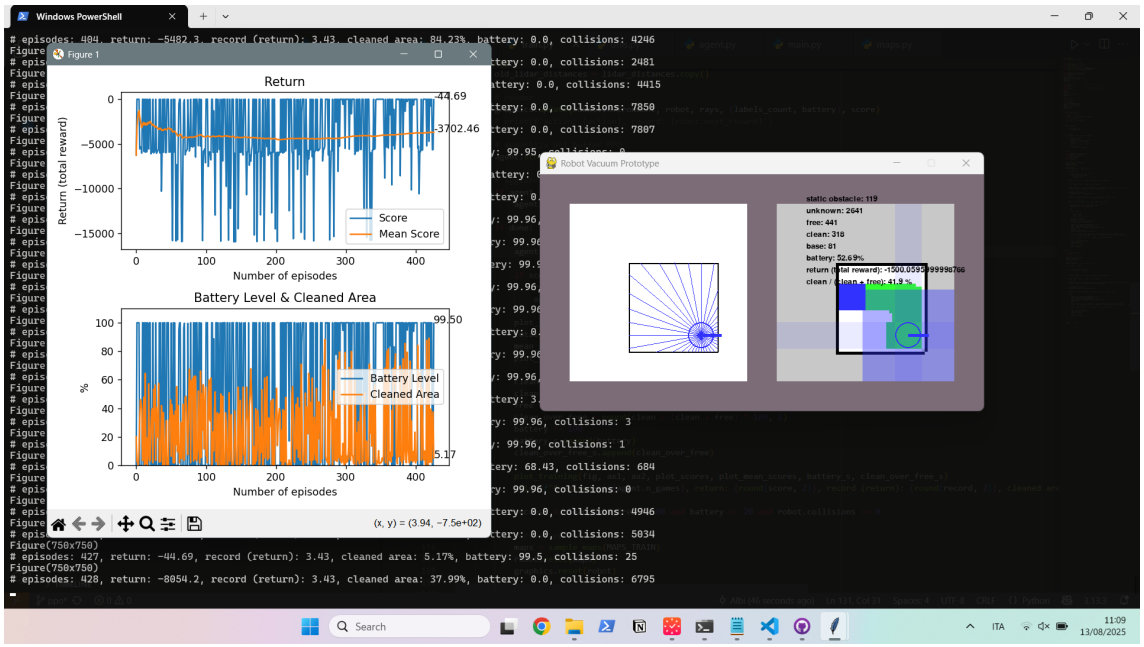

Figure 6.3: Early training of the PPO multi-encoder agent showing unstable returns due to incomplete reward shaping.

After revising the reward function and adjusting hyperparameters, training became significantly more stable. Figures 6.4, 6.5, and 6.6 show later stages of training. The agent achieved consistent cleaning performance around 90% of free cells, a dramatic reduction in collisions (from more than 40 per episode to fewer than 5), and demonstrated the ability to autonomously return to its base once cleaning was completed or the battery was low. Battery management still leaves room for improvement, but the emergence of such complex behaviors represents a substantial success given the computational constraints.

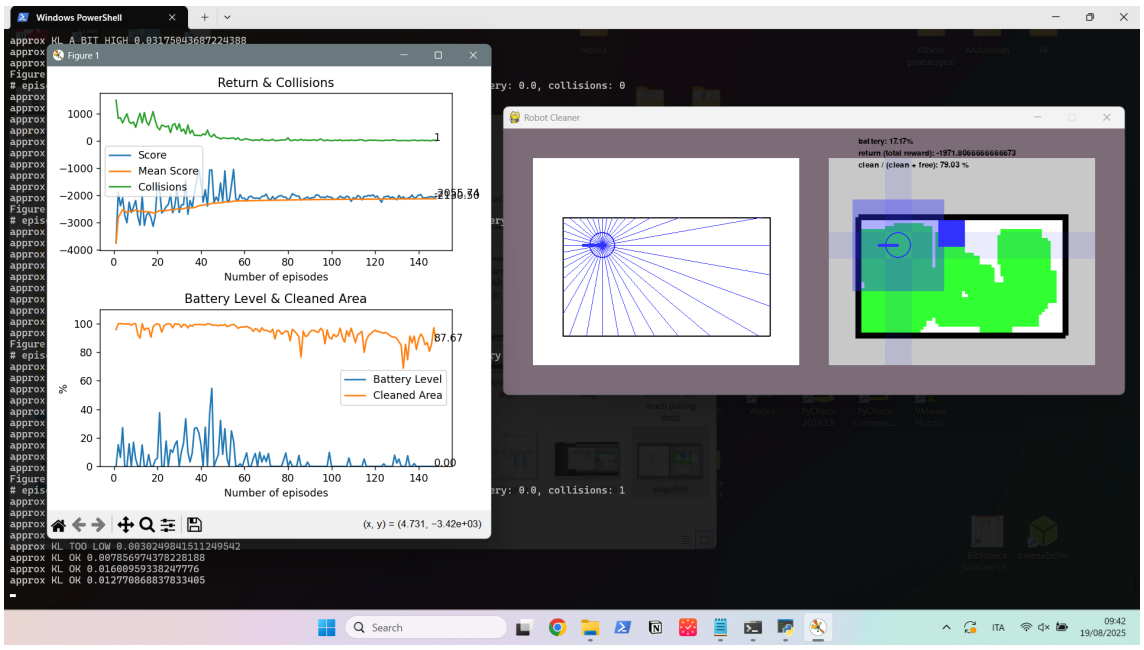

Figure 6.4: Training curves of the PPO multi-encoder agent after curriculum learning: stable returns and improved collision avoidance.

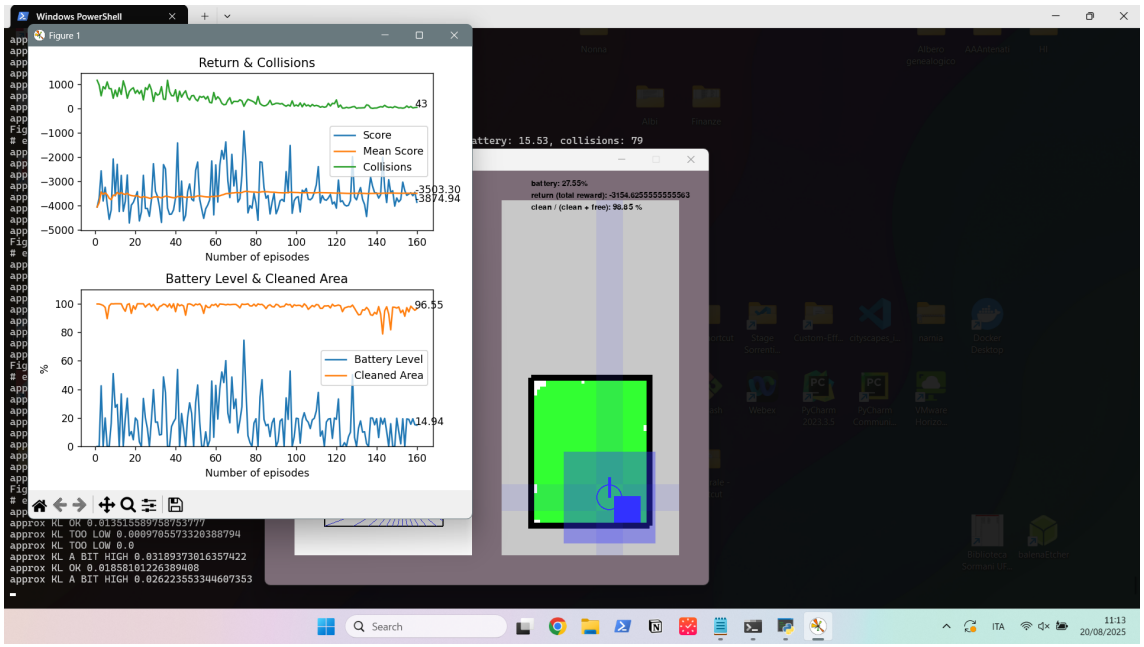

Figure 6.5: Progressive improvement in cleaning efficiency, reaching around 90% of free cells.

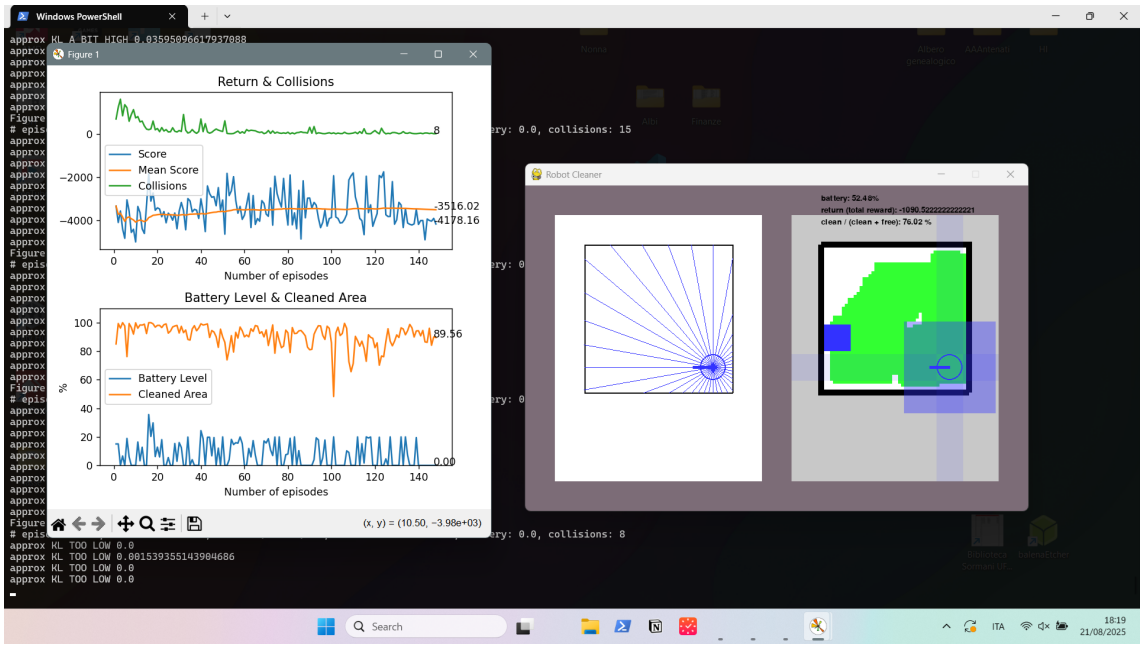

Figure 6.6: Decline in collisions per episode as training progresses, confirming more reliable navigation.

Although incomplete, these results already suffice to demonstrate the emergence of targeted strategies, which was both surprising and gratifying, given the project's limitations.

6.2 Future work

This project opens many possible directions for improvement and extension:

- **Reward shaping:** refining the reward function remains the hardest and most critical challenge. Better balancing exploration, cleaning, and safe return could further stabilize training.
- **Curriculum progression:** continue training on more difficult maps, provided adequate computational resources are available, to test scalability to realistic apartment layouts.

- **Neural architectures:** experimenting with the design of the five networks (four encoders + trunk MLP), for example by deepening the CNN or adding attention layers, could improve feature extraction.
- **Noise in dynamics:** introduce stochasticity in the motion model, simulating wheel slip or odometry errors, to make policies more robust.
- **Noisy sensors:** inject noise in LiDAR distances or collision detection to approximate real-world sensor uncertainty.
- **Realistic sensors:** implement more realistic collision sensors
- **Static obstacles:** add random furniture or walls inside the environment, beyond the main perimeter, to increase diversity.
- **Dynamic obstacles:** simulate moving entities (e.g., kitties) to challenge reactivity and robustness.
- **Battery model:** extend energy consumption to include costs for each rotation or acceleration, making motion planning more realistic.

In summary, the project lays a strong foundation but leaves ample space for development. With greater compute power, richer environments, and more refined reward design, Jesse the (ro)bot could evolve from a conceptual study into a competitive autonomous cleaning system.

Bibliography

[1] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," in *ICML*, 2015, pp. 1889–1897.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[3] Code Bullet. "A.i. learns to play snake using deep q learning [video]." Canale Code Bullet, Accessed: Aug. 22, 2025. [Online]. Available: https://www.youtube.com/watch?v=-NJ9frfAWRo

[4] NeuralNine. "Python + pytorch + pygame reinforcement learning – train an ai to play snake [video]." Canale NeuralNine, Accessed: Aug. 22, 2025. [Online]. Available: https://www.youtube.com/watch?v=L8ypSXwyBds

[5] DeepFindr AI. "Q learning simply explained — sarsa and q-learning explanation [video]." Canale DeepFindr AI, Accessed: Aug. 22, 2025. [Online]. Available: https://www.youtube.com/watch?v=MI8ByADMh20

[6] OpenAI, *Gpt-4o mini*, version 2024-07-18, Used in the project until the introduction of GPT-5 (2025-08-07)., Jul. 18, 2024. Accessed: Aug. 22, 2025. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[7] B. Lin, *Reinforcement Learning Methods in Speech and Language Technology*. Springer, 2025, ISBN: 9783031537196.

[8] OpenAI, *Gpt-5*, version 2025-08-07, Current model as of 2025-08-22., Aug. 7, 2025. Accessed: Aug. 22, 2025. [Online]. Available: https://openai.com/index/introducing-gpt-5/