Learning fault-tolerant navigation with self-reconfiguring modular robots

Gilles Abdel Ahad[1], Nancy Awad[1], Julien Bourgeois[1], Justin Werfel[2], and Benoit Piranda[1]

[1] University of Marie and Louis Pasteur , FEMTO-ST Institute, CNRS, France
firstname.lastname@umlp.fr
[2] Harvard University, USA
jkwerfel@seas.harvard.edu

Abstract. Navigating dynamic environments is a fundamental challenge in distributed robotic systems, particularly when faults occur within the system itself, resulting in a changing connectivity graph. Classical graph search algorithms such as A* provide optimal paths as long as the graph is static. However, faults are a part of real life applications and cannot be ignored; classical approaches scale poorly in such scenarios because updating the graph topology requires extensive inter-robot communication, recombination of local maps, and replanning. This paper proposes a reinforcement learning (RL)-based approach that enables agents to learn navigation policies without requiring global knowledge of the graph. Each agent observes only its immediate neighborhood, making locally reasonable decisions about navigating toward a target location that collectively achieve near-optimal global performance. Through training on randomly chosen faults, our model learns robust traversal behaviors that adapt online to topology changes, reducing communication overhead compared to a more basic A*-based approach in a faulty environment.

Keywords: Modular Robots, Self-reconfiguration, 3D Catoms, Scaffold, Programmable Matter, Reinforcement Learning

1 Introduction

Modular Self-Reconfigurable (MSR) systems are composed of multiple autonomous modules capable of changing their physical configuration to adapt to different tasks and environments. They offer remarkable qualities such as scalability, versatility, and robustness, making them highly suitable for applications ranging from exploration and construction to adaptive structures. For these promises to be fulfilled in real-life applications, fault tolerance is paramount in the development of any solution based on MSR [1]. Several MSR architectures have been developed over the years, differing in geometry, actuation principles, and connection mechanisms. Among these, lattice-based systems, where modules occupy discrete positions in a 2D or 3D grid, stand out for their structured coordination and predictable movement models [2]. Our work focuses on fault-tolerance for a specific type of lattice-based MSR: the *3D Catom* [3]. The *3D Catom* is a millimeter-scale, quasi-spherical module that connects to its neighbors through electrostatic attraction, allowing the collective to form and reconfigure complex three-dimensional structures. However, this electrostatic mechanism introduces potential points of failure. A *3D Catom* 's electrostatic face may fail to activate due to factors such as charge dissipation, misalignment, dust accumulation, humidity, or local power issues, leading to broken connections or movement errors during reconfiguration. Given the millimeter-scale size of *3D Catoms*, constructing even simple structures requires a large number of modules; consequently, even if a *3D Catom* is theoretically expected to exhibit a very low failure probability, the aggregate likelihood of multiple faults arising within a collective becomes significant, making robust fault-tolerance mechanisms indispensable.

Currently, algorithmic solutions developed for *3D Catoms* are implemented in a simulation environment, VisibleSim [4,5], where conditions are idealized and faults rarely considered. Yet, as we move toward real-world implementations, the need for fault-tolerance becomes critical, as environmental variability and hardware imperfections can disrupt reconfiguration sequences. Existing distributed algorithms rely on predefined movement plans that assume all modules function correctly; consequently, a single failure can propagate through the system, causing congestion, deadlocks, or incomplete configurations [6]. It is computationally infeasible to enumerate in advance all possible fault situations and determine appropriate actions for each. Therefore it is crucial for us to find a way to adapt online to faults. This challenge serves as the central motivation of our work: developing dynamic fault-tolerance mechanisms for *3D Catoms* that provide reliable and autonomous self-reconfiguration under uncertain and imperfect operating conditions. In this paper we introduce a machine learning based solution that uses reinforcement learning to adapt online to the faults simulated in the environment.

2 Context

To form a desired structure, *3D Catoms* arrange themselves in repeating configurations called scaffolds. These scaffolds are the skeleton of the structure, whose goal is enabling smooth and rapid internal movements that accelerate the overall construction process. Because of the repeating substructure of a scaffold, as long as we can provide a method to traverse this repeating structure, traversing the entire topology of any desired structure becomes systematically achievable.

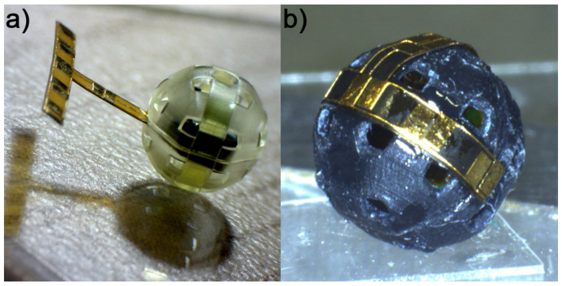

Fig. 1. Picture of two *3D Catom* prototypes. (a) has its electrostatic connectors unwrapped and (b) has the connectors wrapped made for up to 6 different connections (not the final version)

We focus our work on the porous structure introduced in [7]. This structure's main advantage is its ability to store meta-modules within one another. Meta-modules refer to groups of modules arranged to form a porous configuration, as illustrated in figure 1. Because coordinating a large number of individual modules is highly complex, researchers have adopted meta-module strategies in which collections of modules are treated as unified units to simplify both planning and execution. We can break the reconfiguration process into two pieces: determining a sequence of meta-modules to travel through, and determining a sequence of moves to get to the next meta-module in the sequence. High-level planners like RePoSt [8] and ASAPs [9] use Max-flow algorithms to coordinate meta-module motions. Such approaches are highly effective in enabling meta-module relocation through porous structures while preserving system connectivity, and incorporate measures to avoid collisions and deadlocks through a traffic-light-like system.

These planners produce precomputed motion sequences for individual modules, which are executed deterministically during reconfiguration. In this paper, we consider the output of these planners to be a sequence of metamodules for each module to pass through. If we assume that the frequency of faults is low enough that *some* path exists between any two sequential metamodules in a sequence, then the challenge that remains is to give individual modules the ability to navigate within each metamodule to reach the next, in the presence of faults.

With full information about the state of its local metamodule, a module could plan its path from its current metamodule to the next, for instance using A* [10]. However, obtaining a full graph of the metamodule can be very challenging and costly on time since to gather the graph the moving module would have to send messages to all other modules in the section and wait for their responses.

Our work being primarily focused on *3D Catoms*, we need to take into consideration the constraints that govern their movements. Since VisibleSim, a simulator made for MSR, is a controlled environment, developing solutions within it allows us to take advantage of its guarantees: it prevents any movements that would be impossible for real *3D Catoms*.

Hence we introduce a flexible, reactive layer that communicates directly with VisibleSim, for individual module locomotion. It uses reinforcement learning algorithm that would learn the repeating pattern in a scaffold allowing it to make reliable decisions to move towards its final goal.

In this paper, we present a novel approach that replaces deterministic single-module motion planning while preserving the high-level planning efficiency of meta-module approaches.

3 Proposed Solution

Machine learning appears to be a promising approach for handling dynamic environments, as it should be capable of guiding modules away from faults and towards their objectives [11] [12]. However, training a machine learning model to adapt effectively to faulty conditions typically requires large datasets that capture the full range of possible situations and constraints, making such an approach difficult to implement. To maintain compatibility with VisibleSim and avoid the extensive data-collection process, we chose to implement a reinforcement learning algorithm capable of running millions of simulations directly within VisibleSim. This allows the model to learn from simulated experience while automatically respecting the constraints enforced by the simulator.

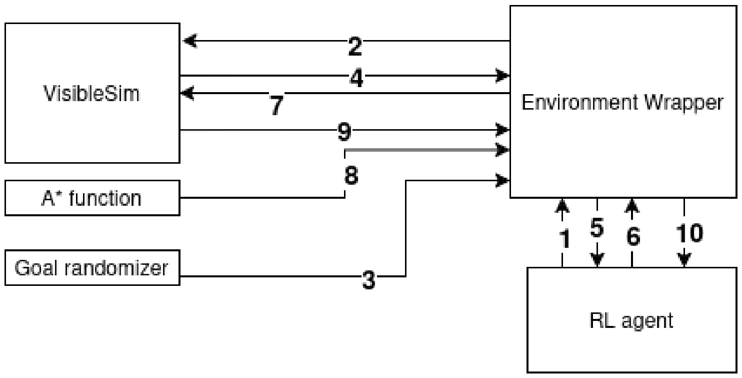

Fig. 2. Overview of the communication pipeline between the RL agent, the environment wrapper, and VisibleSim. Numbered arrows indicate the ordered sequence of messages.

Legend of Figure 2

- 1 Reset: RL agent initiates environment reset
- 2 Reset to initial state: the Environment wrapper resets VisibleSim
- 3 Randomize new goal: Goal randomizer sends new goal to environment
- 4 Sends new set of motions: VisibleSim sends motions to environment

– 5 Sends new observation: Environment sends observation to RL agent
– 6 Initiates Step: RL agent initiates environment step
– 7 Executes the action: Environment executes RL agent's action in VisibleSim
– 8 Sends A* distance: A* function sends distance calculation to environment
– 9 Sends new set of motions: VisibleSim returns updated motions
– 10 Send new observation: Environment sends new observation

3.1 Environment Wrapper

To link VisibleSim to the rl-agents each controlling a single module, we define an environment wrapper that follows the standard Gymnasium [13], implementing the reset() and step() methods to manage the simulation lifecycle. During reset(), the environment restores the simulator's XML configuration to its initial state, clears any previous motion data, and launches the simulator to generate a new set of valid coordinates. A new navigation goal is then sampled at random, and the environment constructs the initial observation composed of the current possible motions.

The step() function executes one iteration of the agent's interaction with the simulator. It receives the agent's chosen action (corresponding to a target coordinate), validates it against the available coordinate list, and updates the XML configuration accordingly. The simulator is then rerun to reflect the new state, and the environment computes a reward based on the agent's progress toward the goal. Each step returns the updated observation, the computed reward, and termination flags (terminated, truncated) that signal whether the episode has ended—either because the goal was reached or a maximum number of steps was exceeded.

The observation comprises all information available to the agent at each iteration. It includes the set of possible motions provided by *VisibleSim* as well as the final goal. Because each *3D Catom* can have up to a fixed maximum number of possible motions, but this maximum is not always reached, we apply a masking mechanism to invalidate the placeholder $(0, 0, 0)$ coordinates used to fill unused entries in the discrete observation vector. At the end of the observation, we append the goal and similarly mark it as invalid in the mask, ensuring that the agent can identify the target location without treating it as an available motion option.

3.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [14] was selected as the reinforcement learning framework for this application due to its stability, adaptability, and robustness in non-stationary, dynamically constrained environments. In the environment wrapper, the agent faces a spatial decision-making problem where it must select coordinates from a variable-sized action space determined by the current graph topology and goal configuration. Traditional value-based methods such as Deep Q-Networks (DQNs) [15] are ill-suited to this setting because they assume fixed, discrete action sets. In contrast, PPO, being a policy-gradient method, optimizes a stochastic policy directly and can naturally accommodate changing action spaces through action masking or contextual encoding. Its clipped surrogate objective further mitigates destabilizing policy updates, enabling steady learning even when the environment or reward landscape shifts significantly between episodes. Moreover, PPO's on-policy formulation allows rapid adaptation to new goals and transitions, while its ability to leverage structured observations (e.g., spatial coordinates) supports strong generalization across diverse graph configurations. Together, these characteristics make PPO a theoretically sound and practically effective choice for this problem domain, offering a balanced combination of learning stability, sample efficiency, and adaptability in an evolving, coordinate-based environment.

In summary, the environment wrapper translates the VisibleSim output (i.e., the set of possible module motions) into a structured input for the RL agent. This includes a discrete observation augmented with the goal configuration, a mask indicating which actions are valid given the current state, the reward computed for the previous action, a termination flag (true when the goal is reached), and a truncation flag (true when the episode exceeds a predefined step limit).

4 Experiments

4.1 Neural Network Architecture

The agent was implemented using the PPO algorithm from the Stable Baselines3 framework, specifically employing the [16] variant to support dynamic action masking during training. The policy network used the "MultiInputPolicy" configuration, which processes multiple input modalities (e.g., spatial coordinates, graph-based features, and environmental states). By default, Stable Baselines3 [16] initializes PPO with a two-layer feedforward neural network consisting of 64 hidden units per layer and Tanh activation functions for both the policy (actor) and value (critic) networks. This relatively compact architecture is well-suited for environments with low-dimensional state spaces or limited variability. It also allows us to save space in the memory of each module.

4.2 Reward Function

Figuring out the reward function of the model is the most important step because it determines how we influence the model to learn the proper features of the structure. Since the goal was to train it on a repeating structure of *3D Catoms* we already had the graph of possible motions, and the fastest way to traverse that graph.

Distance-Based Shaping At each timestep t, the reward is defined as:

$$r_t = \Delta d_t - c_1 - \delta(s_t \in \mathcal{V}) \cdot c_2 + \delta(s_t = s_{\text{goal}}) \cdot R_{\text{goal}}$$

Where:

- s_t is the current state (location) at timestep t.
- s_{goal} is the state of the goal.
- $\Delta d_t = (d(s_{t-1}, s_{\text{goal}}) - d(s_t, s_{\text{goal}}))$ which represents progress toward the goal measured as the reduction in A* distance between consecutive states, where $d(s_{t-1}, s_{\text{goal}})$ is the A* distance from the previous state to the goal and $d(s_t, s_{\text{goal}})$ is the A* distance from the current state to the goal.
- c_1 is a step penalty added to promote shorter paths.
- $\delta()$ is defined as 1 if its argument is true and 0 if its argument is false.
- $\mathcal{V}$ set of previously visited states
- c_2 is a penalty for revisiting $\mathcal{V}$ to ensure loops are minimized.
- R_{goal} is a terminal reward only given if goal is reached.

The constants c_1, c_2, and R_{goal} were determined empirically through repeated experiments to reliably reproduce the reported results. The constants used to replicate our experiments are:

- $c_1 = 0.1$
- $c_2 = 0.5$
- $R_{\text{goal}} = 150$

4.3 Training

For the training environment, we used a repeating structure composed of three meta-modules and simulated the movement of a single module navigating within this pattern. Because the structure is periodic, a module only needs to learn how to move within one instance of the repeating pattern to generalize its navigation to the entire structure as it scales. We defined 11 target goals for the module to reach, corresponding to positions that mimic traversing the repeated structure and moving into place to construct a new meta-module. In our experiments, the module starts at the bottom of the repeating structure mimicking building a structure upwards and goes to one of the 11 goals that are the next entrance to new repeating structures or positions to add new meta-modules to the structure. The next step was to train the default model using the reward function. After 700,000 iterations, the model successfully reached all goals with 100 % accuracy, requiring an average of 7.09 motions per goal, whereas A* required an average of 6.27 motions per goal. As seen in Table 1 the RL-agent seems to come really close to the A* algorithm's performance, with the biggest difference between the paths being of 3 extra steps and 5 out of 11 goals having the same path length as A*. As seen in Figure 3 the A* path is slightly faster and more direct than the one chosen by the RL-agent but the difference is minimal, in the case of the trajectory in figure 3 the RL-agent requires one extra move. These results demonstrate that RL agents can effectively traverse a static graph and achieve path lengths comparable to those produced by the A* algorithm.

4.4 KL Divergence fault training

Having shown in the previous section that RL agents are suitable for discrete motion and can achieve performance comparable to A* in MSR systems, our next

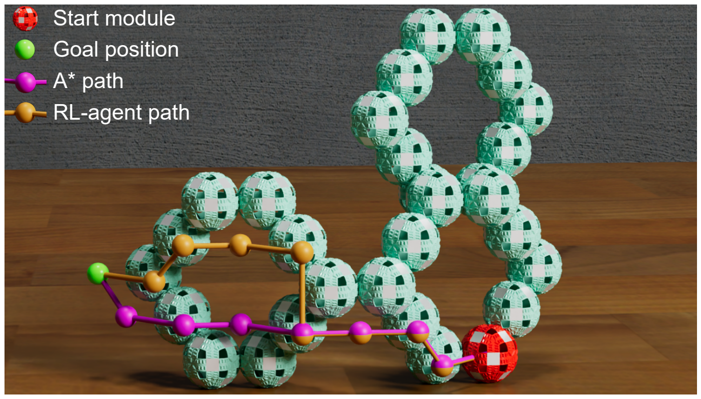

Fig. 3. Comparison of A* and RL-agent paths.

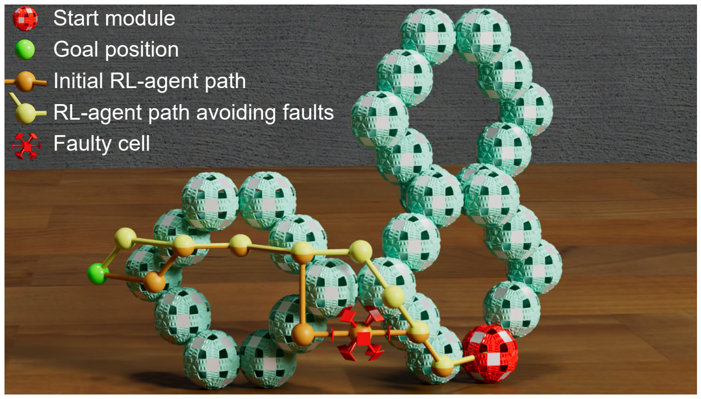

Fig. 4. Comparison of RL paths without and with a faulty cell.

objective is to demonstrate that RL agents can also operate effectively in faulty environments, a setting where A* fails, as it relies on a static graph. To this end, we simulated faults within our repeating structure, by removing one connector randomly from the RL agent chosen path. In that dynamic environment it was able to reach the goal 39 % of the time after testing for 1000 different episodes where 1 connector is removed at random from the path. This is an improvement over discrete options for module movements, but still not optimal. To improve the agent at this task, training the model on an environment with random faults on the path to the goal is the logical next step.

When fine-tuning a PPO agent, its policy parameters θ are being updated to improve expected return under a (possibly new) environment. However, if the updates push the policy too far from the original policy, the agent may forget behaviors that were good before. This would mean that the agent becomes worse at handling graphs with no faults, in some cases being unable to reach the final goal. To prevent that, we decide to penalize deviation from the original (pre-finetuned) policy using a KL-divergence term. Allowing us to improve performance in faulty graphs without hindering our previous results on fault free graphs.

After 400,000 iterations of finetuning with KL-divergence we manage to achieve a 90% success rate after testing for 1,000 different episodes where 1 connector is removed at random from the path. The average path length in the case of a single faulty connector in the agent's path is 7.44, slightly higher than for the average path length in case of no faults for the same agent at 7.09. In Figure 4 we can see an example comparing the finetuned RL-agent with a fault in its path shown in red and the RL-agent's usual path with no faults. The no fault path shown in orange uses the faulty connector that the finetuned RL-agent manages to skip using the yellow path, this change of path ultimately adds no extra movements. We also importantly keep the 100% success rate in cases where there are no faults. As highlighted in Table 1 the paths chosen by the finetuned algorithm are very similar to the paths chosen by the RL-agent, the average path length for both the RL-agent and the finetuned-agent are the exact same at 7.09.

Goal position	**A* Path Length**	**RL-agent Path Length**	**Finetuned-agent Path Length**
(17, 5, 6)	8	9	9
(15, 5, 8)	9	11	10
(14, 5, 8)	8	9	8
(13, 6, 10)	6	6	6
(11, 6, 12)	6	9	6
(10, 6, 12)	7	7	8
(8, 4, 10)	5	6	6
(8, 5, 6)	3	3	3
(11, 5, 4)	2	2	2
(14, 5, 4)	7	7	8
(15, 5, 4)	8	9	10
Avg Path Length	**6,27**	**7,09**	**7,09**

Table 1. Comparison of path lengths between the deterministic A* planner, the reinforcement learning agent, and the finetuned reinforcement learning agent for each goal configuration on a no fault environment. The colors help compare the agents to the A* algorithm with green being same path length yellow being one extra move, orange being two extra moves, and finally red being three extra moves.

In the video (https://youtu.be/uhveT2gw65I), we first show an example of how the A* algorithm navigates to the goal. If there is an unreachable cell in its path (due to a broken connector, for example), the A* path is not usable.

However, as shown in the third part of the video, the RL agent is able to swiftly move away from using online navigation.

5 Conclusion

Modular Self-Reconfigurable (MSR) systems promise adaptability, scalability, and robustness, yet achieving these properties in practice requires explicit mechanisms for fault tolerance. Existing approaches to MSR motion planning typically rely on static, predefined graphs and therefore cannot accommodate structural changes or failures that arise in real deployments. To address this gap, we proposed a reinforcement learning–based motion planning strategy capable of adapting online to local graph variations, a key capability for bridging the simulation-to-reality divide where faults are unavoidable.

Our approach integrates a PPO agent trained on a static graph with KL-regularized finetuning, enabling online adjustment of the navigation policy as the graph evolves. Experiments show that the agent successfully adapts its behavior to dynamically missing edges while preserving stable overall performance. This demonstrates that dynamic, learning-based motion planning can compete with classical approaches and provide resilience in scenarios where deterministic planners alone would require a lot of extra communication.

While our study focused on the motion of a single *3D Catom*, this does not hinder coordination at the system level; multi-module interactions can be governed through a traffic-light-like system that makes sure to avoid collisions. However, the current reward design, centered on immediate changes in A* distance, encourages short-horizon reasoning and may not fully capture global navigation structure. Enhancing this component could provide the agent with a more informative signal and potentially yield more globally consistent behaviors. In addition to that the starting position of the module always being at the bottom of the repeating structure is correct for building structures up but for reconfigurations, it would be ideal if the starting positions would be one of the 11 goals.

Future work may explore richer reward formulations, integrate more expressive policy architectures, and extend the framework to multi-agent learning scenarios or heterogeneous fault models. Beyond algorithmic refinements, investigating hybrid control strategies, where classical planning governs nominal operations and the RL policy is activated only under structural anomalies, could offer a practical path toward deploying resilient motion planning in real MSR robots.

References

1. Kenta Kikuzumi, Ryo Ariizumi, and Fumitoshi Matsuno. Automatic robot design for fault-tolerant robots. *Scientific Reports*, 15(1):29642, August 2025.
2. John W. Romanishin, Kyle Gilpin, and Daniela Rus. M-blocks: Momentum-driven, magnetic modular robots. In *Proceedings of the 2013 IEEE/RSJ International*

Conference on Intelligent Robots and Systems (IROS), pages 4288–4295. IEEE, 2013.

3. Benoit Piranda and Julien Bourgeois. Geometrical study of a quasi-spherical module for building programmable matter. In *Distributed Autonomous Robotic Systems: The 13th International Symposium*, pages 387–400. Springer, 2018.
4. Benoit Piranda. Visiblesim: Your simulator for programmable matter. In *Dagstuhl Seminar*, 2016.
5. Pierre Thalamy, Benoît Piranda, André Naz, and Julien Bourgeois. Visiblesim: A behavioral simulation framework for lattice modular robots. *Robotics and Autonomous Systems*, 147:103913, 2022.
6. Gilles Abdel Ahad, Nancy Awad, Benoit Piranda, Jad Bassil, Justin Werfel, and Julien Bourgeois. Quantitative comparison of self-reconfiguration algorithms for 3d catoms. In Alexandra Nilles, Kirstin H. Petersen, Tin Lun Lam, Amanda Prorok, Michael Rubenstein, and Michael Otte, editors, *Distributed Autonomous Robotic Systems*, pages 85–99, Cham, 2026. Springer Nature Switzerland.
7. Jad Bassil, Benoît Piranda, Abdallah Makhoul, and Julien Bourgeois. A new porous structure for modular robots. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, pages 1539–1541, 2022.
8. Jad Bassil, Benoît Piranda, Abdallah Makhoul, and Julien Bourgeois. Repost: Distributed self-reconfiguration algorithm for modular robots based on porous structure. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12651–12658. IEEE, 2022.
9. Jad Bassil, Benoît Piranda, Abdallah Makhoul, and Julien Bourgeois. Asaps: Asynchronous hybrid self-reconfiguration algorithm for porous modular robotic structures. 2023.
10. Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
11. Paulina Varshavskaya, Leslie Pack Kaelbling, and Daniela Rus. Automated design of adaptive controllers for modular robots using reinforcement learning. *The International Journal of Robotics Research*, 2007. Preprint.
12. Paulina Varshavskaya, Leslie Pack Kaelbling, and Daniela Rus. *Efficient Distributed Reinforcement Learning through Agreement*, pages 367–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
13. Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.
14. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
15. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.
16. Antonin Raffin et al. Stable-baselines3 contrib: Experimental reinforcement learning algorithms, 2021.