QuickNav-RL: Navigation using Reinforcement Learning in JAX

Hugo Adamove[1], Jakub Pekár[2]
[1]492982
[2]492788

1 Introduction

Reinforcement-learning (RL) agents learn by stepping through simulated worlds, so called "environments", yet most of these academic environments are written in `NumPy` [1] and run on the CPU [2]. At the same time the agent network(s) and optimizer already sit on the GPU, so every step involves costly copies between the devices and synchronization. Multiple authors identify this "CPU-environment + GPU-agent" split as today's dominant bottleneck and show that moving the environment to the accelerator can yield 10–1000 times speed-ups [2].

`JAX` [3] allows us to eliminate the CPU-GPU split. Its jit, vmap and pmap primitives compile pure-function environments into fused GPU kernels, and let training and environment simulation share the same device. Earlier projects such as `Brax` [4], `Gymnax` [5], `Pgx` [6] and `JaxMARL` [7] demonstrate the approach across continuous control, Atari-like 2-D games and multi-agent benchmarks, reporting 10× to 1000× wall-clock gains compared to `NumPy` environments [6].

Building on these results, we set three goals for our project:

1. Implement **QuickNav-RL** - a room navigation environment written entirely in `JAX`.
2. Re-implement the identical environment in `NumPy` and measure throughput (steps/s) to compare against the `JAX` version and confirm the performance improvement.
3. Evaluate multiple RL algorithms (PPO [8], SAC [9], TD3 [10]) and agent network architectures (MLP [11], LSTM [12], GTrXL [13]) on our `JAX` QuickNav environment.

2 Related Work

Popular RL environment libraries like `OpenAI Gym` [14], `Gymnasium` [15], and `PettingZoo` [16] use Python and `NumPy` but lack efficient batching and hardware acceleration. High-performance alternatives include `NVIDIA Isaac Lab` [17], which relies on `CUDA` for parallelism, and `Brax`, a `JAX`-based physics engine using JIT compilation and vectorization. Isaac Gym demands `CUDA`-specific engineering, while `Brax` integrates more easily with TPUs and GPUs.

Within the domain of 2D point-to-point navigation, a notable example is `MiniGrid` [18], a widely used library offering configurable grid-world environments where agents navigate mazes to reach a goal, primarily designed for CPU-based research with discrete state and action spaces.

Another relevant project is the `DRL-robot-navigation` repository on GitHub [19], which we heavily drew inspiration from. This project implements training of mobile robots in simulated 2D environments using various DRL algorithms (like TD3, SAC, PPO, DDPG) with `ROS` [20] and `Gazebo` [21], or a simpler Python-based simulator (`IR-SIM` [22]). It demonstrates practical robot navigation to random goal points while avoiding obstacles, often using LiDAR-like sensor inputs.

3 Methods

Environment Design

The environment consists of a two-dimensional square room with configurable dimensions (in meters), enclosed by walls that act as static obstacles. The spatial resolution of the environment grid defines the granularity of obstacle placement and determines the smallest unit of wall or free space.

The simulated robot is equipped with differential drive (two wheels) and a 2D lidar sensor. Several physical and sensory parameters are configurable, including the robot's body size, wheelbase, maximum linear speed, the number of lidar beams, their angular spread (field of view), and the maximum sensing distance. Each episode begins by spawning the robot at a random free-space location within the room, with a randomly placed target the agent is required to reach. To make the task more challenging, multiple potential target locations are initially sampled, and the one farthest from the robot is selected.

Procedural Room Generation

To introduce variation and improve generalization, each episode features a procedurally generated room layout. We implemented a simple randomized wall-carving algorithm based on a stochastic grid walk [23].

The generation process begins with a grid filled entirely with wall tiles. A carving agent, initialized at the center of the grid, performs a random walk, converting wall tiles into free space as it moves. At each step, the agent chooses one of four directions (up, down, left, right) uniformly at random. Additionally,

with a small probability (e.g., 5%), the agent may return to the origin. The process terminates once a desired percentage of free space is reached or after a maximum number of steps. A visualization of this process is shown in Figure 1.

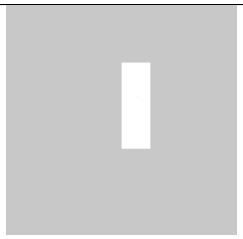
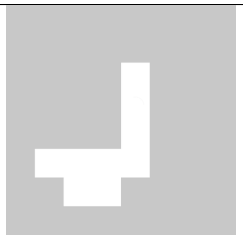
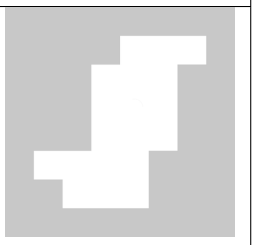
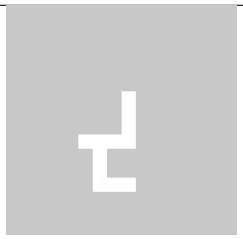
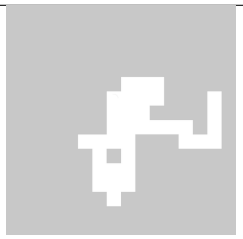
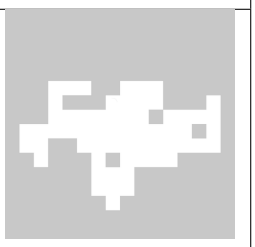
Grid Size	**Ratio of tiles carved-out**		
	0.05	0.15	0.3
8×8			
16×16			

Figure 1: Iterative generation of room environments for various grid sizes of the room. The algorithm iteratively carves out tiles using random walk algorithm.

Lidar Sensor Simulation

A key component of the agent's observation space is its simulated 2D lidar sensor, which provides depth and object-type information in the robot's immediate vicinity. The lidar emits a number of rays beams uniformly spaced within a specified field of view. Both the number of ray beams and maximum range of the lidar are configurable through environment settings.

Each ray is cast from the robot's current position and direction, computing intersections with all nearby obstacles and the target goal as shown in Figure 2. For each ray, the environment records the distance to the nearest intersection (up to the sensor's maximum range—along) with the type of object intersected (obstacle, goal, or empty space).

Agent Observations

The agent receives structured observations at each timestep, which serve as inputs to its reinforcement learning policy. These observations are designed to capture essential spatial and sensory information to support navigation under partial observability.

In our setup, each observation consists of three main components:

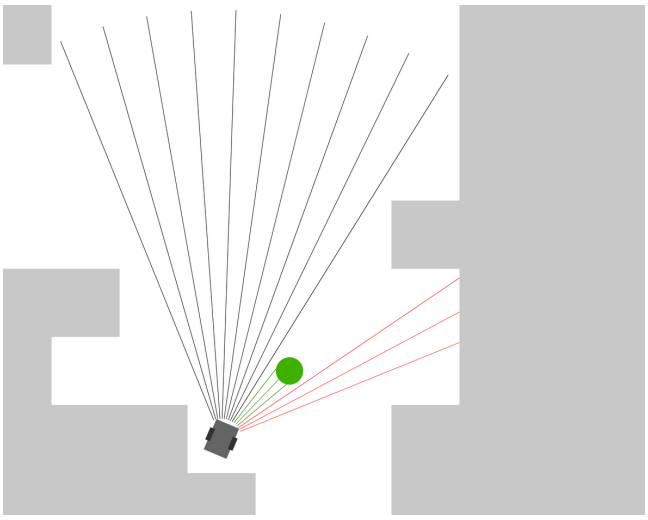


Figure 2: A visualization of the lidar sensor's beams, showing 16 beams with a 10-meter maximum range. Grey rays indicate that the lidar did not detect any obstacles, red rays represent intersections with obstacles, and green rays indicate that the lidar has detected the goal.

- **Ego-Position Information**: The agent's current pose is provided as a tuple of its (x, y) coordinates and orientation angle θ within the environment frame. This allows the policy to track self-localization over time.
- **Lidar Readings**: A fixed-length vector representing the distances measured by the simulated lidar beams, capturing the layout of nearby obstacles and indicating the presence of the goal if detected along any ray. Each beam returns the normalized distance to the closest intersecting object within its range, along with an optional type label (e.g., obstacle or goal).
- **Agent's Memory**: An internal memory vector is maintained and updated at each timestep. This allows the agent to accumulate information over time, enabling it to reason beyond the current observation—crucial for navigation in environments with occlusions and ambiguous layouts.

The combination of positional, sensory, and temporal information enables the agent to plan and adapt its behavior as it explores and approaches the goal.

Agent Actions

The agent controls its movement through a simple differential drive mechanism. The action space consists of two continuous values, each representing the normalized velocity (in the range $[-1, 1]$) of the left and right wheels, respectively. These values are scaled by a predefined maximum wheel speed.

This action representation allows the agent to perform a variety of maneuvers: moving forward or backward, turning in place, or executing smooth curves. The simplicity of this low-level control model keeps the action space minimal while preserving sufficient expressiveness for learning complex navigation strategies.

Rewards & Penalization

At each timestep, the agent receives a scalar reward signal that reflects its progress toward the navigation goal, while also incorporating various forms of penalization to discourage undesirable behavior.

The reward function is defined as a weighted sum of individual components, with the following key elements:

- **Wall Collision Penalty** (λ_w): The agent incurs a negative reward if the action would cause its body to intersects with any wall segment, encouraging obstacle avoidance.
- **Step Penalty** (λ_s): A small constant penalty is applied at every timestep to incentivize efficient solutions and discourage unnecessary wandering or stalling.
- **Cycle Penalty** (λ_c): To prevent the agent from revisiting the same locations, it is penalized whenever it returns to a previously visited position. This promotes exploration.
- **Progress Reward** (λ_p): If the agent's distance to the goal decreases compared to the previous timestep, a positive reward proportional to the progress is granted. Conversely, if the agent moves farther from the goal, it receives a negative reward of the same magnitude.
- **Goal Reward** (λ_g): A large positive reward is assigned when the agent successfully reaches the target goal location, serving as the primary training signal for completing the navigation task.

Formally, the total reward r_t at timestep t can be expressed as:

$$r_t = -\lambda_s - \mathbb{I}_{wall\_collision} \cdot \lambda_w - \min(\sum_{0}^{t} \mathbb{I}_{cycling_t}, 10) \cdot \lambda_c + \text{sign}(d_{t-1} - d_t) \cdot \lambda_p + \mathbb{I}_{goal\_reached} \cdot \lambda_g$$

where d_t denotes the distance to the goal at time t, and the indicator functions ($\mathbb{I}$) evaluate to 1 when the corresponding event occurs. The specific setup of λ parameters is denoted in the section 4.

RL Algorithms

To train our reinforcement learning agents, we employed a range of modern policy optimization algorithms, each representing a distinct design philosophy in terms of exploration, stability, and sample efficiency. We summarize the selected three widely used optimization strategies below:

Proximal Policy Optimization (PPO) PPO is an on-policy policy gradient method that stabilizes learning by limiting the magnitude of policy updates. It achieves this through a clipped surrogate objective that prevents large deviations from the current policy, thereby reducing the risk of performance collapse.

PPO strikes a balance between ease of implementation, training stability, and performance. It is particularly popular in benchmarks due to its robustness across a range of tasks.

- **Type:** On-policy, policy gradient
- **Key Idea:** Clipped objective function to prevent drastic policy updates
- **Advantages:** Simpler and more stable than TRPO; general-purpose applicability

Soft Actor-Critic (SAC) SAC is an off-policy actor-critic algorithm based on the maximum entropy reinforcement learning framework. It seeks to simultaneously maximize the expected return and policy entropy, thereby promoting stochastic policies that explore more effectively.

Due to its off-policy nature and entropy regularization, SAC is highly sample-efficient and well-suited to continuous action spaces, making it an excellent fit for robotics and navigation tasks.

- **Type:** Off-policy, stochastic actor-critic
- **Key Idea:** Encourages exploration via entropy maximization
- **Advantages:** High sample efficiency; stable training in continuous control settings

Twin Delayed Deep Deterministic Policy Gradient (TD3) TD3 builds upon the DDPG algorithm by addressing its known overestimation bias and training instability. It introduces three major innovations: using two Q-networks to take the minimum value estimate (twin critics), delaying policy updates relative to value function updates, and applying noise to the target policy for smoothing.

TD3 operates with deterministic policies, making it suitable for environments with relatively low stochasticity and where fine control precision is important.

- **Type:** Off-policy, deterministic actor-critic
- **Key Idea:** Addresses overestimation bias through twin critics and delayed updates
- **Advantages:** More stable than DDPG; effective in low-noise, continuous control domains

Each of these optimization strategies brings distinct strengths, and their comparative evaluation in our environment highlights trade-offs between exploration, convergence speed, and robustness in navigation under partial observability.

Implementation Details: The implementations of these algorithms are sourced from the `rejax` project [24], which provides efficient and flexible reinforcement learning algorithms in `JAX`. The environment conforms to the `gymnax` [5] API, ensuring compatibility with `rejax` and other `JAX`-based reinforcement learning libraries.

Models

To evaluate the impact of memory mechanisms on navigation performance in partially observable environments, we implemented and compared three distinct agent architectures: a simple Multi-Layer Perceptron (MLP), a Long Short-Term Memory (LSTM) network, and a Gated Transformer-XL (GTrXL)-based model.

MLP

The MLP model serves as a strong baseline for memory-free or minimal-memory navigation. Despite its simplicity, such architectures are often sufficient for reactive behaviors and short-term decision-making in structured environments.

To provide the MLP with limited temporal reasoning, we extended the architecture with a lightweight external memory. The model consists of a standard feedforward backbone followed by two output heads:

- **Action Head:** Predicts actions.
- **Memory Head:** Produces an updated memory vector to be passed into the next timestep.

At each timestep, the model receives as input a concatenation of the current observation vector and the memory vector from the previous step. This allows it to maintain a compressed representation of its past states, although in a less sophisticated manner than recurrent or transformer-based models.

LSTM

The LSTM model introduces an explicit mechanism for temporal memory, enabling the agent to reason over extended sequences of past observations. LSTMs are a well-established choice for tasks involving partial observability, where decisions must be informed by a history of inputs rather than the current state alone. Our LSTM-based architecture consists of a multi-layer LSTM module with a fixed hidden state dimensionality. Specifically, each LSTM layer maintains a

hidden state vector h_t and a cell state vector c_t, both of which constitute the agent's memory and are propagated forward through time. The LSTM layers process sequences of input vectors formed by the concatenation of the agent's current observation and the previous memory state. Following the recurrent layers, the output hidden state of the final LSTM layer is passed through a fully connected (dense) layer to produce the action predictions.

GTrXL

GTrXL is a transformer architecture tailored for reinforcement learning. It builds on Transformer-XL by introducing gating mechanisms in place of residual connections to improve training stability and long-term memory handling—key issues in RL due to non-stationary targets and partial observability. The model retains segment-level recurrence and attention over memory, while gates regulate information flow, enabling better convergence in tasks like MiniGrid and Atari. The actions are obtained using simple linear layer attached to the outputs of the transformer model.

4 Experiments & Setups

Project Setup

The project is designed to be easily reproducible and straightforward to install, leveraging modern Python dependency management tools. Specifically, it uses the `uv` package manager, which enables fast and deterministic environment setup through the use of `pyproject.toml` and `uv.lock` files.

To set up the project environment, users first need to install `uv`. Below are the platform-specific installation instructions:

macOS:

On macOS, you can install `uv` using Homebrew:

```
brew install uv
```

Linux:

```
curl -sSL https://install.uv.sh | bash
```

Windows (via `winget`):

```
winget install uv
```

Setting Up the Python Environment

Once `uv` is installed, create a complete Python virtual environment under `".venv"` directory with all necessary dependencies by executing the following command:

```
uv sync
```

Enabling CUDA for GPU Support

If your device has a GPU with CUDA support, you can enable it during the environment setup to make training significantly faster by running:

```
uv sync --extra cuda
```

Jupyter Notebooks

The repository contains multiple examples in the form of Jupyter notebook files under `"./examples"` directory. When selecting a kernel for the notebook execution, ensure that you select the `.venv` kernel. This will ensure that all required dependencies from the virtual environment are present for seamless execution of the notebooks.

Implementation Details

Environment

For all experiments, we standardized the environment configuration to ensure fair comparison across models. Each room was set to a size of $8m \times 8m$, with a grid resolution of 16 cells in each dimension. This results in a grid cell size of $0.5m \times 0.5m$, which defines the minimum possible width of any corridor or gap between obstacles.

We employed a procedural room generation algorithm with a carved percentage of 80%, producing rooms of moderate complexity. Representative examples of generated layouts can be seen in Figure 1.

The simulated robot agent was configured with a radius of 15 cm (i.e., total width of 30 cm), ensuring that it can navigate all valid passages without collision. The robot's maximum linear speed was limited to 1m/s.

Rewards & Penalties

Unless otherwise specified, all experiments used the following default reward and penalization setup: $\lambda_c = 0$, $\lambda_p = 1$, $\lambda_s = 0.1$, $\lambda_w = 10$ and $\lambda_g = 50$.

Models

- **MLP:** The architecture of the MLP backbone—including the number of layers and neurons per layer—was treated as a hyperparameter during optimization. Each layer was followed by a ReLU activation function and a dropout layer with a rate of 0.5 to encourage regularization. The memory vector had a fixed dimensionality of 32 and was initialized with zeros at the start of each episode.

- **LSTM:** The dimensionality of the LSTM network, including the number of layers and the sizes of the hidden state h_t and cell state c_t, was optimized during hyperparameter search. The output from the final LSTM layer was passed through a fully connected layer to produce the action distribution. Memory is represented implicitly via the LSTM's internal recurrent states, which were initialized to zero vectors at the beginning of each episode.
- **GTrXL:** For the GTrXL model, we fixed the memory size to 32 and the number of attention heads to 2. During hyperparameter optimization, we searched for the optimal embedding size, attention head dimensionality, and the number of transformer layers. The memory was also initialized with zeros.

5 Results

JAX vs NumPy Performance Comparison

We first investigate the performance benefits of using a vectorized `JAX`-based environment compared to a traditional `NumPy`-based implementation. The primary goal of this experiment is to evaluate raw simulation throughput independently of model learning.

To isolate the computational performance of the environments, we replaced the policy model with a random action generator. We measured the total time required to execute 100,000 environment steps. To assess how environment complexity impacts performance, we varied the grid size across three configurations: 4×4, 8×8, and 16×16.

Figure 3 presents the wall-clock times for both implementations. As expected, the `JAX`-based environment demonstrates a significant speed advantage.

Hypothesis: With increased environment size, the LiDAR collision detection code becomes the main bottleneck and benefits the most from vectorization. This is because, as the grid size grows, the number of potential collision checks increases, and the efficiency gains from vectorizing these operations become more pronounced.

RL Algorithms Comparison & Tuning

Following the environment performance evaluation, we assessed the effectiveness of three prominent reinforcement learning algorithms: PPO, SAC, and TD3. Our objective was to evaluate their relative performance under a controlled setting using a shared network architecture. We employed a simple 2-layer MLP architecture with 128 neurons per layer and no memory mechanism. To compensate for the absence of recurrent or external memory, the agent was given direct access to the goal coordinates as part of its observation space.

To ensure a fair comparison, we conducted a hyperparameter optimization for each algorithm using the Optuna framework. Each algorithm underwent

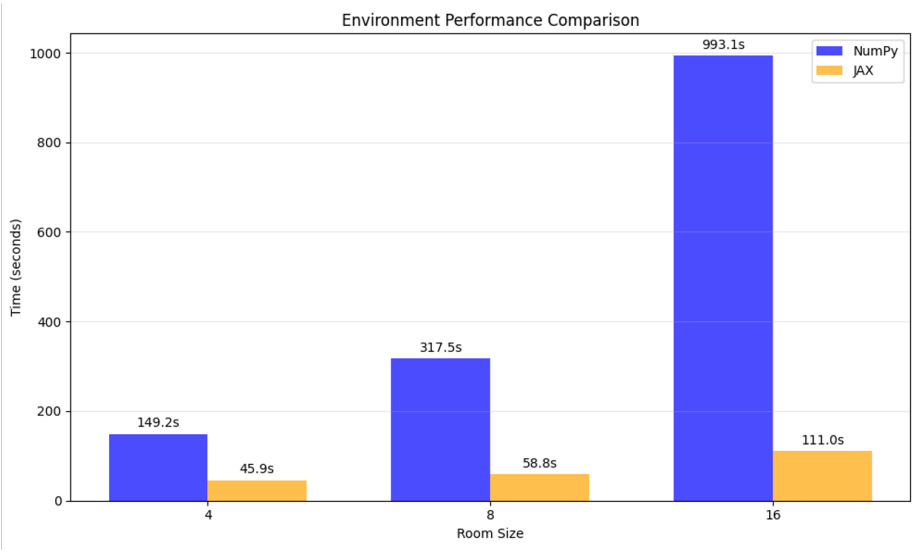


Figure 3: Wall-clock time (in seconds) of 100,000 steps for `JAX` vs. `NumPy` environments across different room sizes. `JAX` shows consistently faster performance, especially in larger environments.

200 optimization trials. The search objective was to maximize the mean reward over 10 training episodes, serving as a proxy for early learning performance in partially observable navigation scenarios.

The final optimized hyperparameter configurations for each algorithm as well as the search spaces are summarized in section 6.

To assess the practical effectiveness of the tuned algorithms, we further evaluated the mean reward and standard deviation achieved on a held-out test set using the best hyperparameter settings found in the Optuna search. These results are presented in Table 1. They reflect the stability and generalizability of each algorithm under partially observable conditions with a memoryless MLP model. The training curves of these algorithms are presented in section 6.

Table 1: Test set performance for PPO, SAC, and TD3 using the best hyperparameters found via Optuna. Each value represents the mean reward ± standard deviation over 10 evaluation episodes.

Algorithm	**Mean Reward ± Std. Dev.** (higher is better)
PPO	$\mathbf{96.87 \pm 8.50}$
SAC	73.52 ± 51.59
TD3	85.99 ± 36.22

The results suggest that PPO achieved the most stable and highest-performing

behavior in this simplified setting, followed by TD3 and SAC. We note that the absence of memory limits long-term reasoning, so these findings mainly reflect the ability of each algorithm to exploit short-term and spatial cues under a consistent policy gradient framework.

Model Comparison & Tuning

Building on the results from the simplified setup, we proceeded to evaluate the capabilities of advanced models equipped with memory mechanisms. In this phase, we removed the goal position from the observation space to simulate a more realistic partially observable environment. Agents were required to navigate using only LiDAR readings, their current position, and internal memory representations.

Based on earlier experiments, we selected PPO as the sole optimization algorithm for this stage, as it consistently demonstrated superior performance in our benchmarks. Given the significant impact of the reward function on learning behavior, **we extended the tuning process** beyond model architecture (e.g., number of layers, layer size) **to include critical reward and penalty parameters**: step penalty (λ_s), progress reward (λ_p), and collision penalty (λ_c).

We used the PPO hyperparameters obtained from the previous hyperparameter search. Since the reward function was under active tuning, we selected the **mean number of steps** required to reach the goal as the main optimization objective, as it is independent of reward scales. Hyperparameter optimization was implemented using Optuna across 100 trials per model. The final reward and penalty configurations as well as the search space are summarized in Appendix 6.

Table 2 reports the mean number of steps required to reach the goal on the test set for each tuned model. The LSTM model achieved the best performance, demonstrating the effectiveness of recurrent memory in partially observable settings. While the GTrXL model is theoretically expected to outperform recurrent networks due to its attention-based architecture and longer memory horizon, practical constraints on available hardware limited the model size we were able to train, likely contributing to the suboptimal performance observed in our experiments.

Table 2: Test set performance: mean number of steps to finish an episode ($\pm$ standard deviation) over 10 evaluation episodes for each algorithm using the best hyperparameters found via Optuna.

Algorithm	**Mean Steps $\pm$ Std. Dev.** (lower is better)
MLP	214.61 ± 94.53
LSTM	$\mathbf{126.60 \pm 75.71}$
GTrXL	133.90 ± 88.94

6 Conclusion

In this work, we developed a lightweight yet flexible `JAX`-based vectorized environment for robotic navigation using LiDAR in procedurally generated room layouts. The environment design allows for fully customizable parameters including room size, grid resolution, obstacle density, and sensor configuration.

We demonstrated that the `JAX` implementation offers significant performance benefits over a traditional `NumPy`-based approach, particularly as environment complexity increases. This makes it well-suited for scalable training and experimentation in RL research.

To evaluate control strategies, we benchmarked three prominent RL algorithms—PPO, SAC, and TD3—under a simplified setup using a memoryless MLP model. PPO consistently outperformed the others in terms of stability and average reward, highlighting its robustness for navigation tasks even in partially observable environments.

Furthermore, we developed and evaluated three distinct agent architectures implemented in Flax: an MLP with lightweight external memory, a multi-layer LSTM, and a Gated Transformer-XL (GTrXL). These models were assessed on the full navigation task, which required reasoning over sparse LiDAR observations and leveraging internal memory mechanisms. Among the evaluated models, the LSTM achieved the best overall performance. While the GTrXL architecture holds promise due to its ability to model long-term dependencies through attention mechanisms, our experiments were constrained by limited hardware resources, which prevented the training of larger, more expressive transformer models. This limitation likely contributed to the subpar performance observed for the GTrXL in our setting.

The full codebase is publicly available at: https://github.com/hadamove/quicknav-rl

References

[1] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[2] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning, 2018. URL https://arxiv.org/abs/1810.05762.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

[4] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021. URL http://github.com/google/brax.

[5] Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022. URL http://github.com/RobertTLange/gymnax.

[6] Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning, 2024. URL https://arxiv.org/abs/2303.17503.

[7] Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garar Ingvarsson, Timon Willi, Ravi Hammond, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu, and Jakob Nicolaus Foerster. Jaxmarl: Multi-agent rl environments and algorithms in jax. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[8] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

[9] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. URL https://arxiv.org/abs/1801.01290.

[10] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018. URL https://arxiv.org/abs/1802.09477.

[11] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. doi: 10.1038/323533a0.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[13] Emilio Parisotto, Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant M. Jayakumar, et al. Stabilizing transformers for reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, pages 7487–7498, 2020.

[14] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. URL https://arxiv.org/abs/1606.01540.

[15] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032.

[16] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34: 15032–15043, 2021.

[17] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023. doi: 10.1109/LRA.2023.3270034.

[18] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. In *Advances in Neural Information Processing Systems 36, New Orleans, LA, USA*, December 2023.

[19] Reinis Cimurs, Il Hong Suh, and Jin Han Lee. Goal-driven autonomous exploration through deep reinforcement learning. *IEEE Robotics and Automation Letters*, 7(2):730–737, 2022. doi: 10.1109/LRA.2021.3133591.

[20] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[21] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004. doi: 10.1109/IROS.2004.1389727.

[22] Ruihua Han et al. IR–SIM: An open-source lightweight simulator for robot navigation, control, and learning, 2024. URL https://github.com/hanruihua/ir-sim.

[23] Tianhan Gao, Jin Zhang, and Qingwei Mi. Procedural generation of game levels and maps: A review. In *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 050–055, 2022. doi: 10.1109/ICAIIC54071.2022.9722624.

[24] Jarek Liesen, Chris Lu, and Robert Lange. rejax, 2024. URL https://github.com/keraJLi/rejax.

Appendix

RL Algorithms Hyperparameter Tuning

Hyperparameter Search Spaces

In this section, we provide the hyperparameter search spaces for each of the three RL algorithms (PPO, SAC and TD3), as detailed in section 5.

PPO:

- **Learning Rate:** (1e-5, 1e-3), log
- **Number of Minibatches:** [16, 32, 64, 128, 256]
- **Number of Steps:** [64, 128, 256, 512]
- **GAE Lambda:** (0.9, 1.0)
- **Entropy Coefficient:** (0.0, 0.1)
- **Clip Epsilon:** (0.1, 0.3)
- **Gamma:** (0.9, 0.999)

SAC:

- **Learning Rate:** (1e-5, 1e-3), log
- **Batch Size:** [64, 128, 256, 512]
- **Buffer Size:** [50,000, 100,000, 200,000, 500,000]
- **Polyak:** (0.9, 0.999)
- **Target Entropy Ratio:** (0.5, 1.0)
- **Target Update Frequency:** [1, 2, 3, 4]
- **Gamma:** (0.9, 0.999)

TD3

- **Learning Rate:** (1e-5, 1e-3), log
- **Exploration Noise:** (0.1, 0.5)
- **Buffer Size:** [50,000, 100,000, 200,000, 500,000]
- **Batch Size:** [64, 128, 256, 512]
- **Polyak:** (0.9, 0.999)
- **Target Noise:** (0.1, 0.4)
- **Policy Delay:** [1, 2, 3, 4]

Optimal Hyperparameters

In this section, we provide the optimal hyperparameters obtained through hyperparameter tuning for each of the three RL algorithms , as detailed in section 5.

PPO:

- **Learning Rate:** 0.0004
- **Number of Minibatches:** 256
- **Number of Steps:** 512
- **GAE Lambda:** 0.93
- **Entropy Coefficient:** 0.0095
- **Clip Epsilon:** 0.146
- **Gamma:** 0.96

SAC:

- **Learning Rate:** 0.0009
- **Batch Size:** 512
- **Buffer Size:** 500,000
- **Polyak:** 0.90
- **Target Entropy Ratio:** 0.82
- **Target Update Frequency:** 3
- **Gamma:** 0.96

TD3:

- **Learning Rate:** 0.0005
- **Exploration Noise:** 0.44
- **Buffer Size:** 50,000
- **Batch Size:** 256
- **Polyak:** 0.97
- **Target Noise:** 0.16
- **Policy Delay:** 1

Training Curves

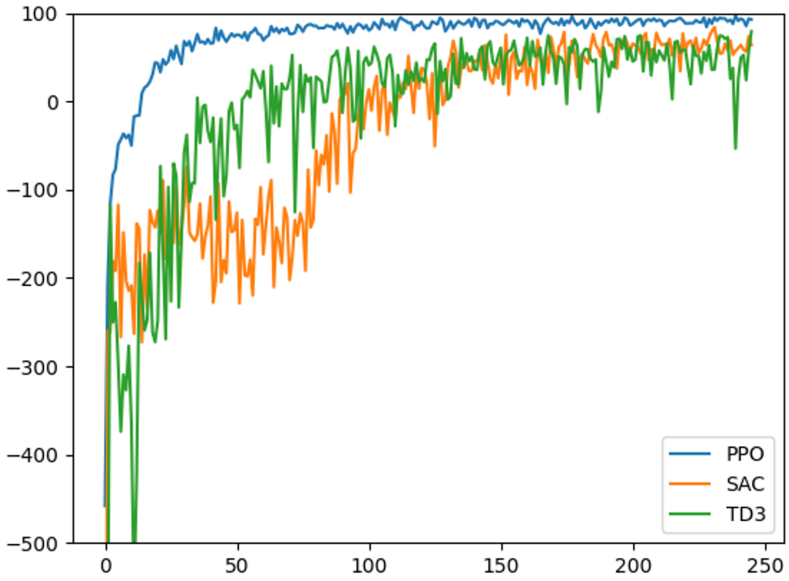


Figure 4: The rewards collected throughout the training of the RL algorithms.

Generated Trajectories

Figure 7 and Figure 8 display the trajectories generated by each algorithm after hyperparameter adjustment, evaluated over 9 test episodes. Note that the trajectories are truncated after 100 environment steps, meaning the agent may not always reach the goal within this limit.

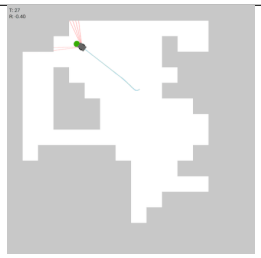
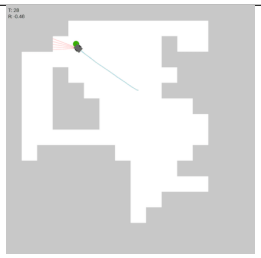
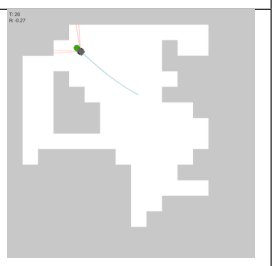
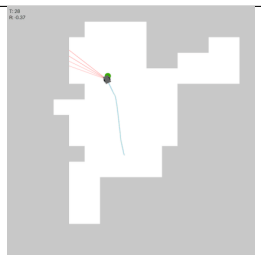
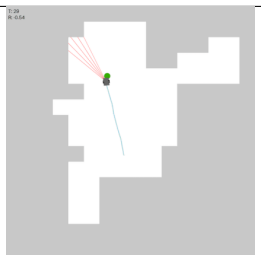
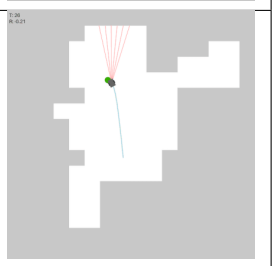
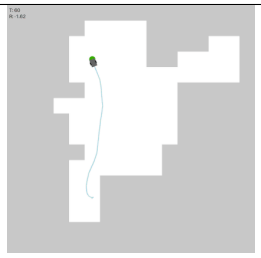
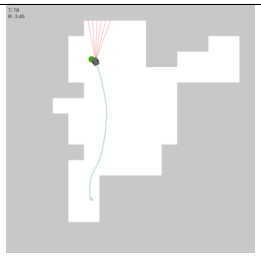
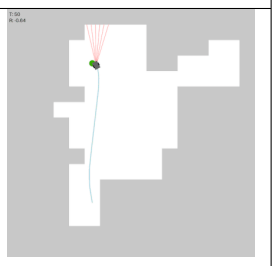
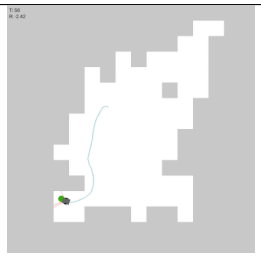
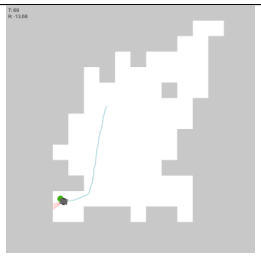
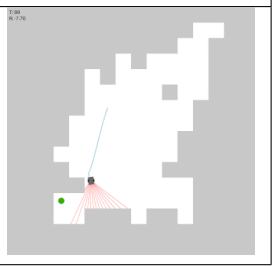
Episode	PPO	SAC	TD3
1			
2			
3			
4			

Figure 5: Comparison of trajectories generated by tuned PPO, SAC, and TD3 algorithms with MLP across evaluation episodes 1 to 4.

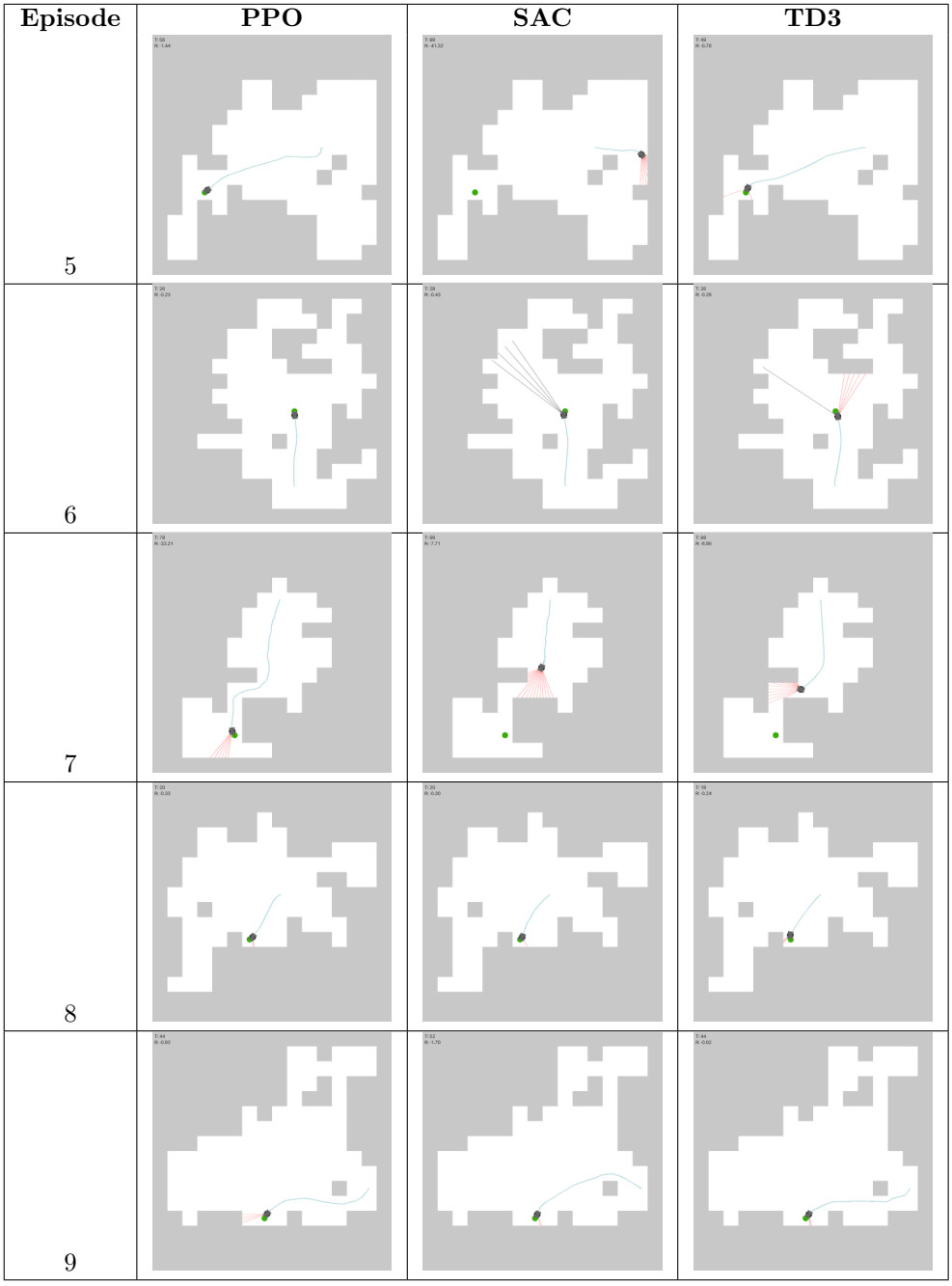


Figure 6: Comparison of trajectories generated by tuned PPO, SAC, and TD3 algorithms with MLP across evaluation episodes 5 to 9.

Model Architectures Hyperparameter Tuning

Hyperparameter Search Spaces

In this section, we provide the hyperparameter search spaces for each of the three models (MLP, LSTM and GTrXL), as detailed in section 5.

MLP:

- **# Hidden layers:** (1, 3)
- **Hidden layer size:** [64, 128, 256]
- **Step Penalty:** (0.01, 0.5)
- **Progress Reward:** (0.1, 1.0)
- **Cycling Penalty:** (0.001, 0.5)

LSTM:

- **# Layers:** (2, 10), int
- **Layer Dimensionality:** (5, 16)
- **Step Penalty:** (0.01, 0.5)
- **Progress Reward:** (0.1, 1.0)
- **Cycling Penalty:** (0.001, 0.5)

GTrXL:

- **Embedding Dimensionality:** [8, 16]
- **Head Dimensionality:** (1, 8)
- **# Layers:** (2, 5)
- **Step Penalty:** (0.01, 0.5)
- **Progress Reward:** (0.1, 1.0)
- **Cycling Penalty:** (0.001, 0.5)

Optimal Hyperparameters

In this section, we provide the optimal hyperparameters obtained through hyperparameter tuning for each of the three model architectures (MLP, LSTM, GTrXL), as detailed in section 5 .

MLP:

- **# Hidden layers:** 1
- **Hidden layer size:** 256
- **Step Penalty:** 0.191
- **Progress Reward:** 0.611
- **Cycling Penalty:** 0.352

LSTM:

- **# Layers:** 2
- **Layer Dimensionality:** 15
- **Step Penalty:** 0.370
- **Progress Reward:** 0.443
- **Cycling Penalty:** 0.203

GTrXL:

- **Embedding Dimensionality:** 8
- **Head Dimensionality:** 4
- **# Layers:** 2
- **Step Penalty:** 0.232
- **Progress Reward:** 0.291
- **Cycling Penalty:** 0.276

Generated Trajectories

Figure 7 and Figure 8 display the trajectories generated by each model after hyperparameter adjustment, evaluated over 9 test episodes. Note that the trajectories are truncated after 200 environment steps, meaning the agent may not always reach the goal within this limit.

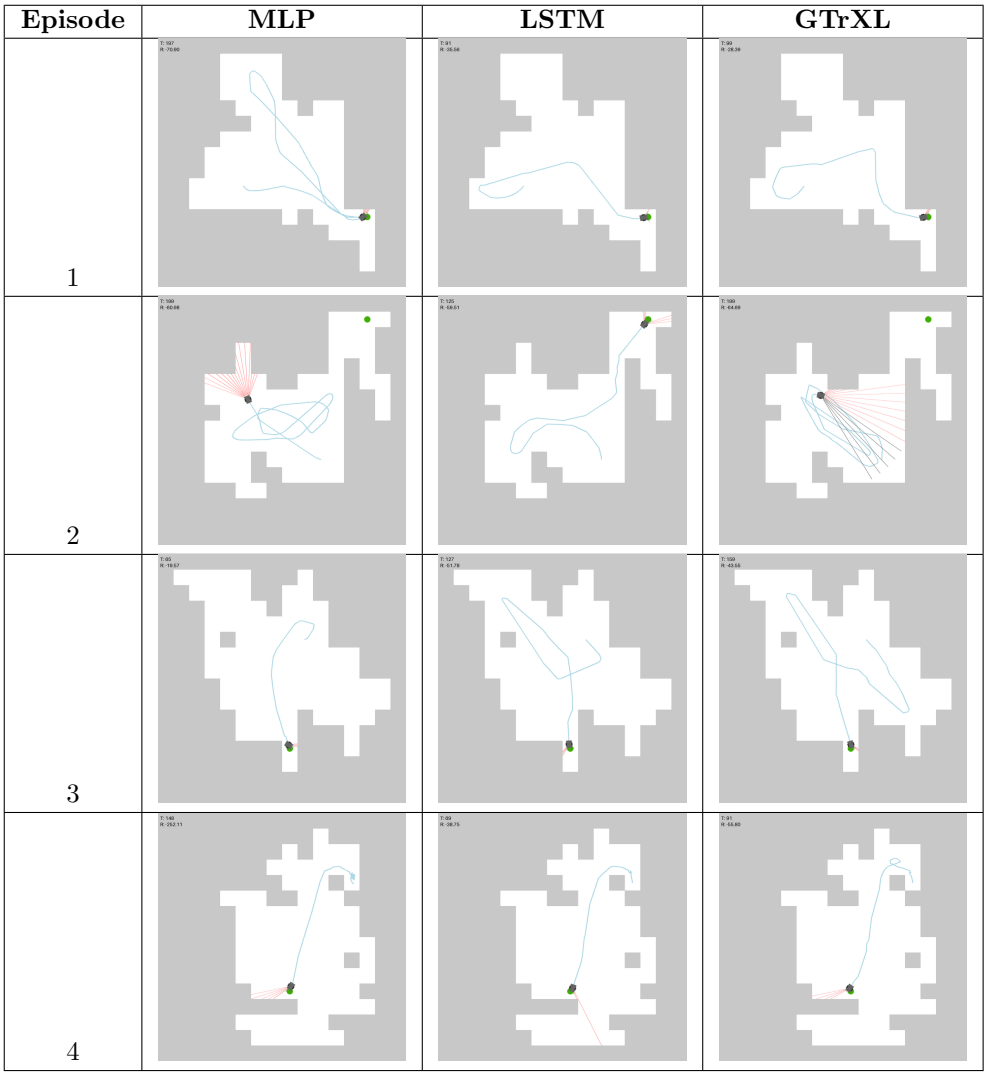


Figure 7: Comparison of trajectories generated by tuned MLP, LSTM, and GTrXL models (trained by PPO) across evaluation episodes 1 to 4.

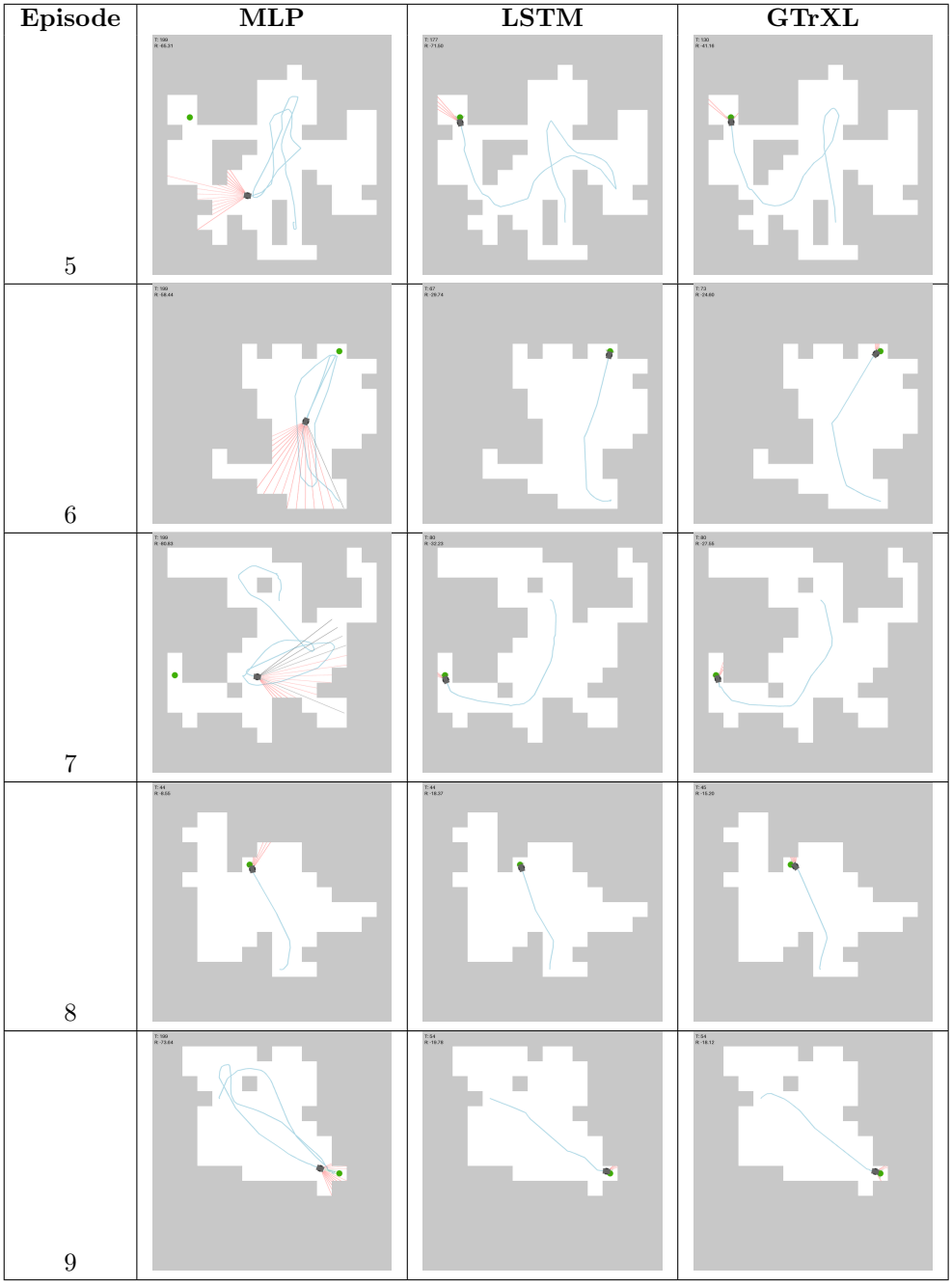


Figure 8: Comparison of trajectories generated by tuned MLP, LSTM, and GTrXL models (trained by PPO) across evaluation episodes 5 to 9.