Autonomous Escape Learning: Training an RL Agent for Efficient Navigation in Confined Environment

Siddharth Reddy Maramreddy
IMT2022031
IIIT - Bangalore
Bengaluru, India
Siddharth.Maramreddy@iiitb.ac.in

Sai Venkata Sohith Gutta
IMT2022042
IIIT - Bangalore
Bengaluru, India
Sohith.Gutta@iiitb.ac.in

I. Problem Statement

In many real-world environments, autonomous agents are required to navigate through confined spaces that are filled with obstacles, with the goal of reaching a specific target or exit. The task is further complicated when the exit is small or located in a difficult-to-reach part of the space. Navigating these environments efficiently is a significant challenge, especially when the agent must avoid obstacles that can cause delays, damage, or failures in the mission.

In this scenario, the problem is to train an agent to navigate a closed, obstacle-filled room with a small exit. The environment is designed with numerous obstacles scattered throughout the space, creating a maze-like structure that limits available paths and increases the difficulty of finding a viable escape route. The challenge lies in the agent's ability to:

- **Navigate the environment effectively** while dealing with the constraints of a small exit and the unpredictability of obstacle placement.
- **Minimize the number of collisions** with obstacles, as each collision represents a failure in the agent's decision-making process and adds unnecessary complexity to the task. Each hit may also result in performance penalties, which further complicate the agent's ability to escape successfully.
- **Optimize the escape process** by balancing the speed of movement towards the exit with the need to avoid obstacles, creating an efficient path that minimizes unnecessary detours or risky maneuvers.

Given the complexity of the environment and the high penalty associated with obstacles, the problem is to develop a system capable of autonomously learning the most efficient way to escape the room, while considering both speed and safety. The primary goal is to minimize the number of obstacle hits, which directly correlates with better agent performance and a safer escape route.

This problem mimics real-world scenarios where autonomous systems must navigate through physically constrained environments, such as drones operating in confined disaster zones or robots working in cluttered or hazardous areas. The difficulty in solving this problem lies in the agent's need to both react to and anticipate the obstacles within a dynamic environment, ultimately achieving the task with minimal errors.

II. Environment Design

The environment for this reinforcement learning task was developed using Unity and the ML-Agents Toolkit. It simulates a confined 3D space filled with dynamic obstacles and a small, fixed exit. The main objective is for the agent to navigate from a randomly initialized position to the exit while avoiding collisions with moving obstacles. The setup is designed to closely mimic challenging real-world navigation scenarios requiring both spatial awareness and dynamic planning.

A. Room Layout and Exit

The environment is a closed rectangular room bounded by solid walls on all sides. A small exit is carved into one of the walls, which serves as the goal for the agent. The exit is implemented using a Unity trigger collider, allowing the environment to detect when the agent has successfully reached the goal. The exit does not change position between episodes, ensuring consistency in goal localization.
Additionally, there is an outer (south) wall surrounding the environment, designed to prevent the agent from falling out of the room. This wall is made transparent to enhance the visual experience, allowing the agent to move freely while still ensuring the boundary is respected.

B. Obstacles

The room contains several types of physically interactive and moving obstacles designed to challenge the agent's ability to plan and react(as shown in Fig. 2). These obstacles are:

- **Moving Walls:** Linear-moving barriers that traverse fixed paths, blocking or opening routes periodically.
- **Vertical Spinners:** Rotating arms aligned vertically that can push the agent off course or into other obstacles.

- **Horizontal Spinners:** Flat, rotating objects mounted horizontally, rotating around their vertical axis, requiring precise timing to pass.
- **Pendulum:** A swinging obstacle simulating a pendulum, introducing dynamic motion and requiring the agent to wait or time its movement.

Each of these obstacles interacts physically with the agent via Unity's physics engine. Colliding with obstacles may displace the agent or interrupt its progress. These interactions make timing and movement strategy essential for successful navigation.

C. Lighting and Visibility

To ensure the agent can effectively perceive its surroundings, lighting has been added to the environment. Two lights(as shown in Fig. 3) are placed at the top corners of the room to provide adequate visibility. These lights not only illuminate the space but also enhance the contrast between obstacles and the environment, aiding the agent's decision-making process during navigation.

D. Agent Characteristics

The agent(as shown in Fig. 2) is represented by a cube and is spawned at a random location within the room at the start of each episode. This randomness ensures generalization of the learned policy and prevents overfitting to a single start position.

The agent's movement capabilities are:

- **Translational Movement:** Move forward, backward, left, and right along the XZ-plane.
- **Jumping:** A single jump from the ground; double jumping is not allowed. The jump is subject to Unity's physics engine.

E. Sensing and Perception

The agent is equipped with a `Ray Perception Sensor 3D` (as shown in Fig. 1) from the ML-Agents Toolkit. This sensor emits rays in multiple directions from the agent's body to detect nearby objects such as obstacles, walls, and the exit. It enables the agent to gather spatial information and build a representation of the surrounding environment, even with partial observability.

F. Reward Mechanism

The reward structure is designed to guide the agent toward safe and efficient escape behavior:

- **Positive Reward:** Granted upon successfully reaching the exit.
- **Negative Reward:** Applied when the agent collides with an obstacle.
- **Time Penalty:** A small penalty per timestep to encourage faster completion.

This reward scheme balances exploration with efficiency, discouraging reckless behavior and promoting strategic planning.

G. Physics and Interaction

All interactions in the environment are governed by Unity's built-in physics engine. Obstacles have rigidbody components and colliders, allowing realistic dynamic responses. The agent also has a rigidbody, making it subject to forces such as gravity and collisions.

H. Summary

This environment combines random initial conditions, dynamic physical obstacles, and limited perception to create a challenging task for an autonomous agent. It is designed to test and improve the agent's ability to perceive its surroundings, reason about movement strategies, and adaptively navigate toward the goal while avoiding penalties.

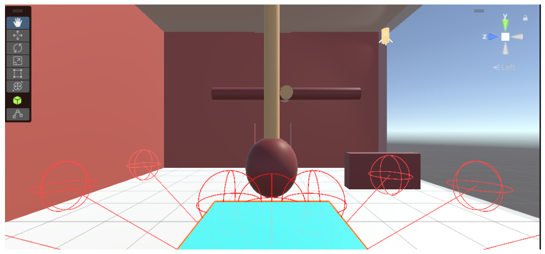


Fig. 1. Third-person perspective from the agent, showcasing the use of the Ray Perception Sensor 3D to detect nearby obstacles and environment features.

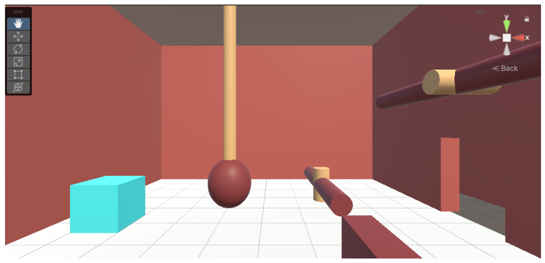


Fig. 2. Internal view of the room environment highlighting dynamic obstacles such as spinners and moving walls, which the agent(blue cube) must navigate through.

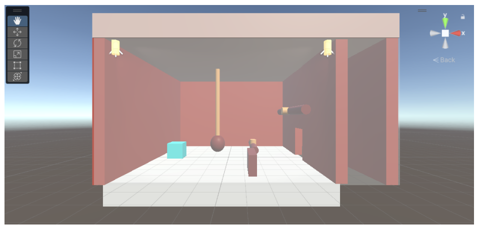


Fig. 3. Full environment overview showing the spatial arrangement of the agent, obstacles, and the exit point carved into the far wall.

III. MDP Formulation

The problem of training an agent to escape a room filled with dynamic obstacles can be modeled as a Markov Decision Process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where:

- $\mathcal{S}$ is the state space,
- $\mathcal{A}$ is the action space,
- $\mathcal{P}$ is the state transition probability function,
- $\mathcal{R}$ is the reward function, and
- γ is the discount factor.

A. State Space $\mathcal{S}$

The agent's observation vector consists of:

- The agent's local position (x, y, z),
- The agent's current velocity vector (v_x, v_y, v_z),
- A binary flag indicating whether the agent is grounded,
- A 55-dimensional vector from the Ray Perception Sensor 3D, encoding spatial information about surrounding obstacles and objects.

This results in a total of $3 + 3 + 1 + 55 = 62$ observation dimensions:

$$\mathcal{S} \subseteq \mathbb{R}^{61} \times \{0, 1\}$$

B. Action Space $\mathcal{A}$

The environment supports both continuous and discrete action spaces, used in different experiment settings.

1) Continuous Action Space: The agent uses a continuous action space consisting of three values:

- a_1: movement along the X-axis (left/right),
- a_2: movement along the Z-axis (forward/backward),
- a_3: jump command (value > 0.5 triggers a jump if grounded).

Thus, the continuous action space is:

$$\mathcal{A}_{\text{cont}} = [-1, 1]^2 \times [0, 1]$$

2) Discrete Action Space: In the updated version, a discrete action space is employed, which consists of two branches:

- **Movement action** (a_1):

0:	No Move
1:	Left
2:	Right
3:	Forward
4:	Backward

- **Jump action** (a_2):

0:	No Jump
1:	Jump (if grounded)

Thus, the discrete action space is:

$$\mathcal{A}_{\text{disc}} = \{0, 1, 2, 3, 4\} \times \{0, 1\}$$

with a total of $5 \times 2 = 10$ possible discrete action combinations.

C. Transition Dynamics $\mathcal{P}$

The transition function $\mathcal{P}(s'|s, a)$ is governed by Unity's physics engine. Agent dynamics such as jumping, collisions, and interactions with moving obstacles are fully differentiable and governed by rigidbody physics. State transitions are therefore deterministic under fixed initial seeds but generally treated as stochastic due to randomized spawn locations and dynamic environment components.

D. Reward Function $\mathcal{R}$

The reward function is designed to encourage the agent to reach the exit quickly while avoiding collisions with walls and obstacles:

- $+1.0$ reward for reaching the exit (success),
- -0.05 penalty for colliding with obstacles,
- -0.1 penalty for colliding with walls,
- -0.01 penalty for jumping,
- -0.001 per time-step penalty to encourage faster solutions.

Formally:

$$\mathcal{R}(s, a) = \begin{cases} +1.0, & \text{if agent reaches the exit,} \\ -0.05, & \text{if agent hits an obstacle,} \\ -0.1, & \text{if agent hits a wall,} \\ -0.01, & \text{if agent jumps,} \\ -0.001, & \text{otherwise (per time step).} \end{cases}$$

Reward Shaping Analysis: Alternate versions of the reward function were also tested to study their effect on exploration behavior. These included (i) equal penalties for wall and obstacle collisions, and (ii) higher penalties for obstacle collisions than for wall collisions. However, both strategies resulted in the agent becoming overly conservative, frequently hesitating or refusing to approach the exit. This behavior indicates that excessive penalization discouraged necessary exploration and risk-taking required to find a viable path through the environment. The final reward structure balances encouragement for task completion with light penalties to promote safety without stalling learning.

E. Discount Factor γ

A discount factor $\gamma \in (0, 1]$ is used to balance immediate versus long-term rewards. For this environment, γ is typically set to 0.99 to ensure the agent considers long-term strategies while emphasizing timely completion.

IV. Algorithms

A. Proximal Policy Optimization (PPO)

PPO is a policy-gradient method that strikes a balance between performance and stability. It adjusts the agent's policy in small, constrained steps to avoid destabilising updates.

We used the config file `ppo_config.yaml`. These are some of the important hyperparameters:

- *Batch size*: 1024
- *Buffer size*: 10240
- *Learning rate*: 3e-4 (with linear decay)
- *Clipping (epsilon)*: 0.2 — controls how much the policy can change per update
- *Entropy regularisation (beta)*: 0.005 — encourages exploration, decays over time
- *Network*: 2-layer MLP with 128 hidden units
- *Gamma (discount)*: 0.99 — emphasises future rewards
- *Max steps*: 500,000 training steps

In the MLAgents library, there is an inbuilt implementation of PPO. So, we used the built-in implementation instead of writing our own implementation.

B. Soft Actor Critic (SAC)

SAC is an off-policy algorithm that maximises both expected return and entropy, encouraging the agent to explore diverse behaviours. It's particularly effective for environments with complex or continuous action spaces.

We used two configurations with SAC. These are the important hyperparameters for each one of them.

`sac_config1.yaml`:

- *Learning rate*: 3e-4
- *Batch size*: 128
- *Replay buffer*: 500,000 samples
- *Entropy coefficient*: 0.2
- *Network*: 2 layers × 256 hidden units
- *Gamma*: 0.99
- *Max steps*: 200,000

`sac_config2.yaml`:

- Learning rate: 2e-4
- Batch size: 256
- Replay buffer: 1,000,000 samples
- Entropy coefficient: 0.1
- Network: 3 layers × 256 hidden units
- Gamma: 0.995
- Max steps: 500,000

Similar to PPO, the MLAgents library has an inbuilt implementation of PPO. So, we used the built-in implementation instead of writing our own implementation.

C. Deep Q-Network (DQN)

DQN is a value-based reinforcement learning algorithm that learns to estimate the Q-value (expected future return) for each possible action in a given state, and selects actions based on these estimates.

Since DQN only works with a discrete action space, we converted our continuous action space to a discrete action space for this algorithm. You can look at the discrete action space in the 'MDP Formulation' section.

Since the MLAgents library doesn't have an in-built implementation for DQN, we have written our own implementation of DQN.

Neural Network Architecture

- *Raycast encoder*: 1d convolutional layers extract spatial features.
- *Vector encoder*: MLP for positional data.
- *Fusion layer*: Concatenates encoded features and outputs Q-values for all actions.

Training Loop

- Uses epsilon-greedy exploration with decaying ϵ.
- Transitions are stored in a replay buffer.
- Periodically samples mini-batches to compute TD error and update the Q-network using MSE loss.
- A target network is updated periodically to stabilise training.

Hyperparameters

- *Episodes*: 500
- *Time Scale*: 20x for faster training
- *Target Network Sync*: Every 10 episodes
- *Replay Buffer Size*: 50,000
- *Batch Size*: 64
- *Discount Factor (γ)*: 0.99
- *Learning Rate*: 0.001

V. Experiments

We first tried to train the agent in an environment without obstacles using the PPO algorithm. We used the default PPO configuration provided by the MLAgents library.

Next, we tried to train the agent in environments with one or two obstacles. We used the same default PPO configuration provided by the MLAgents library.

Next, we tried to train the agent with all the obstacles present in the environment. Here, we faced a major problem. There was a glitch in Unity3d, which is causing our agent (a cube) to fall through the ground randomly during training. To fix this, we restricted the agent's rotation about the x and z directions and only allowed rotation about the y direction. We also performed many experiments after removing the vertical rotator obstacle from the environment, which we believed was causing the problem. We also used a different material for the ground. The experiments are as follows

1) PPO algorithm using `ppo_config.yaml` configurations and the discrete action space.
2) PPO algorithm using the default configuration and the discrete action space.
3) SAC algorithm using `sac_config1.yaml` and `sac_config2.yaml` configurations and the discrete action space.
4) PPO algorithm using `ppo_config.yaml` configuration and the continuous action space.
5) PPO algorithm using the default configuration and the continuous action space.
6) SAC algorithm using `sac_config1.yaml` and `sac_config2.yaml` configurations and the continuous action space.

Finally, we also added the fourth obstacle (the vertical spinner) and performed some more experiments. We also added 5 stacked vectors in some of the experiments. This means that the state space also contains the state vectors from the previous four time steps. This is so that the agent remembers information from previous time steps to decide the next action. The experiments are as follows

1) PPO algorithm using `ppo_config.yaml` configuration and the continuous action space.
2) SAC algorithm using `sac_config1.yaml` and `sac_config2.yaml` configurations and the continuous action space.

3) PPO algorithm using `ppo_config.yaml` configuration and the continuous action space and 5 stacked vectors.
4) SAC algorithm using `sac_config1.yaml` and `sac_config2.yaml` configurations and the continuous action space and 5 stacked vectors

We also ran our DQN algorithm with the discrete action space in the environment without the vertical spinner.

VI. Results

Fig. 4 shows the mean reward over the number of steps for different runs. Fig. 5 shows the mean reward over the number of steps for different runs in the logarithmic scale. The colour to run mapping is shown in Fig. 6

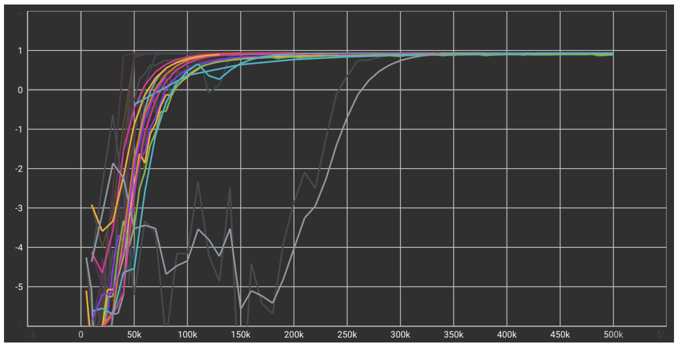


Fig. 4. The graph showing mean reward vs steps for different runs

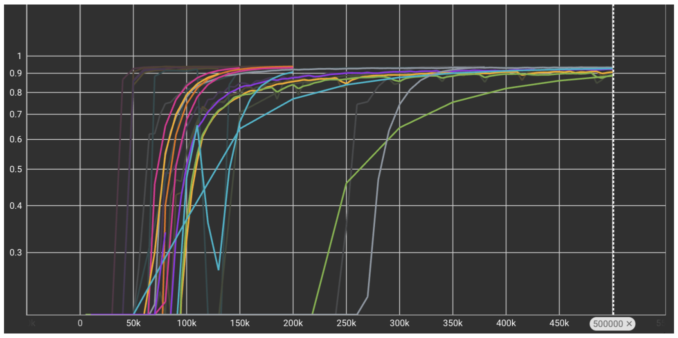


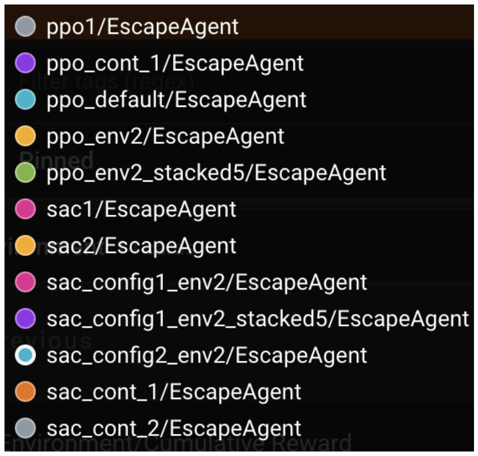
Fig. 5. The graph showing mean reward vs steps for different runs in logarithmic scale

`ppo1` is the run of PPO in the environment without the vertical spinner using `ppo_config.yaml` configuration and the discrete action space.

`ppo_default` is the run of PPO in the environment without the vertical spinner using `ppo_config.yaml` configuration and the discrete action space.

`sac1` and `sac2` correspond to the runs of SAC in the environment without the vertical spinner using `sac_config1.yaml` and `sac_config2.yaml` configurations and the discrete action space.

`ppo_cont1` is the run of PPO in the environment without the vertical spinner using `ppo_config.yaml` configuration and the continuous action space.



● ppo1/EscapeAgent
● ppo_cont_1/EscapeAgent
● ppo_default/EscapeAgent
● ppo_env2/EscapeAgent
● ppo_env2_stacked5/EscapeAgent
● sac1/EscapeAgent
● sac2/EscapeAgent
● sac_config1_env2/EscapeAgent
● sac_config1_env2_stacked5/EscapeAgent
● sac_config2_env2/EscapeAgent
● sac_cont_1/EscapeAgent
● sac_cont_2/EscapeAgent

Fig. 6. The legends for Fig. 4

`sac_cont_1` and `sac_cont_2` correspond to the runs of SAC in the environment without the vertical spinner using `sac_config1.yaml` and `sac_config2.yaml` configurations and the continuous action space.

`ppo_env2` is the run of PPO in the environment with all the obstacles using `ppo_config.yaml` configuration and the continuous action space.

`sac_config1_env2` and `sac_config2_env2` correspond to the runs of SAC in the environment with all the obstacles using `sac_config1.yaml` and `sac_config2.yaml` configurations and the continuous action space.

`ppo_env2_stacked5` is the run of PPO in the environment with all the obstacles using `ppo_config.yaml` configuration, the continuous action space and five stacked vectors in the state space.

`ppo_config1_env2_stacked5` is the run of SAC in the environment with all the obstacles using `sac_config1.yaml` configuration, the continuous action space and five stacked vectors in the state space.

VII. Analysis

From the reward curves Fig. 4, Fig. 5 and experimental logs, several insights can be drawn regarding the performance of different algorithms, action spaces, and input representations.

A. *Effectiveness of PPO vs SAC*

Proximal Policy Optimization (PPO) demonstrated faster initial learning and greater stability across training runs, particularly in the discrete action space. Soft Actor-Critic (SAC), while generally slower to converge, achieved higher peak rewards in continuous action environments with fewer obstacles. This is consistent with SAC's entropy-regularized objective, which encourages broader exploration.

In more complex environments with full obstacle sets, PPO's on-policy nature made it more sensitive to environmental noise and abrupt dynamics. In contrast, SAC's off-policy formulation allowed for more stable long-term learning and better recovery from rare but high-reward transitions.

B. Impact of Action Space

Discrete action spaces yielded more stable and interpretable training behavior for both PPO and SAC. Continuous control enabled smoother and more precise movement—particularly effective in timing-sensitive scenarios like jumping over pendulums—but came at the cost of increased training time and policy complexity.

C. Role of Observation Stacking

Observation stacking significantly improved agent performance, especially in dynamic environments. Agents trained with 5 stacked observations could better anticipate periodic motions, leading to smoother navigation and higher task success rates. This effect was most noticeable in `ppo_env2_stacked5` and `sac_config1_env2_stacked5`, which outperformed their non-stacked counterparts.

D. DQN Performance

The custom Deep Q-Network (DQN) struggled to perform even in the environment which did not have the vertical spinner. Its mean reward was oscillating between negative and positive and it did not converge. As a value-based method, DQN is inherently limited in handling continuous actions or high-dimensional control.

E. Training Stability and Hyperparameter Sensitivity

PPO was generally robust to hyperparameter variation, while SAC—especially with the deeper configuration in `sac_config2.yaml`—was more sensitive to entropy coefficients and replay buffer sizes. Deeper networks (3 layers $\times$ 256 units) offered better representation power but increased training variance.