Reinforcement Learning for Ballbot Navigation: Replication and Enhancement of Salehi (2025)

Nebiyu Tadesse
Johns Hopkins University, Department of Electrical and Computer Engineering

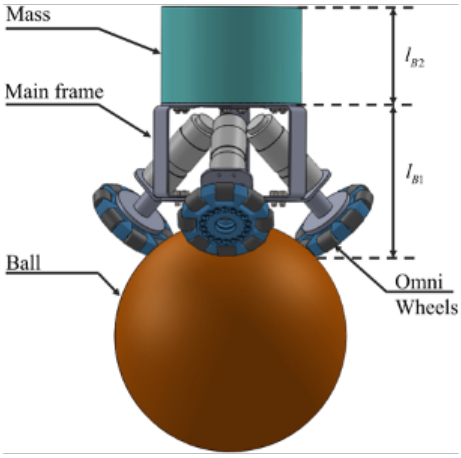
***Abstract*—This work presents a replication and enhancement study of Salehi's (2025) [1] reinforcement learning approach for ballbot navigation in uneven terrain. Ballbots are dynamically balanced mobile robots that present significant control challenges due to their underactuated nature and requirement for dynamic stability [2], [3]. Classical control approaches suffer from flat ground assumptions [3] and lack of generalization [1]. We replicated the core components of the reference implementation (MuJoCo simulation environment, PPO training pipeline, depth encoder architecture, and reward function formulation) and enhanced the open-source codebase with comprehensive documentation (50+ files, 100+ pages), extended terrain generation to 13 diverse types, and created a modular reward architecture. The enhanced codebase features a plugin-based component registry system enabling easy experimentation. Our training experiments on CPU-based hardware achieved up to 10.01M timesteps across multiple configurations, demonstrating learning dynamics consistent with the reference implementation (initial balance learning phase followed by directional optimization). However, full convergence to reference performance levels was not achieved, with best results reaching 9.20 reward and 378-step episodes compared to the reference 86.0 reward and 4000-step episodes. Hardware limitations (CPU-only, 10 parallel environments vs. GPU-enabled 100+), identified software bugs (Perlin noise scaling issues), and insufficient training duration contributed to this performance gap. Despite these challenges, our results validate the approach's feasibility and provide insights into training efficiency requirements. The engineering contributions establish a robust foundation for future ballbot research. The enhanced codebase is available at https://github.com/N3b3x/openballbot-rl.**

I. Goals and Motivation

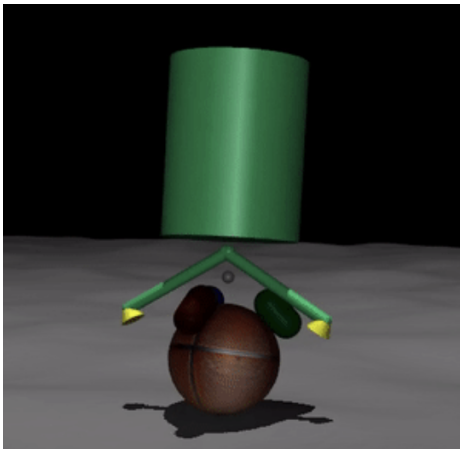
A. Problem Statement

Ballbot control represents a fundamental challenge in robotics due to the system's underactuated nature and requirement for dynamic stability [2], [3]. Figure 1 provides an overview of the ballbot system and RL training environment. Unlike statically stable robots, ballbots must actively maintain balance through continuous control, as the support polygon collapses to a single contact point [3]. The three-omniwheel configuration provides only three control inputs for six degrees of freedom (three rotational plus three translational), creating a highly coupled system where balance and motion cannot be independently controlled [3], [5].

Classical control approaches, while successful on flat terrain, face significant limitations when applied to uneven terrain [1]. Most controllers assume flat ground conditions ($h(x, y) = \text{constant}$) [3], introducing inductive bias that



(a) Basic ballbot structure



(b) RL training on uneven terrain

Figure 1: Overview of ballbot system and RL training environment. (a) Basic ballbot mechanical structure showing the ball, omni wheels, main frame, and upper mass with key dimensions l_{B1} and l_{B2} [4]. (b) Example of randomly generated Perlin noise terrain used for reinforcement learning training, demonstrating the challenging navigation scenarios the robot must learn to handle.

Contact: ntadess2@jhu.edu

prevents generalization to varied terrain. Hand-tuned control gains optimized for specific conditions fail to adapt to unseen terrain variations, requiring explicit terrain models that are difficult to obtain in practice [1].

B. Research Goals

The primary goal of this work is to replicate Salehi's (2025) [1] reinforcement learning-based ballbot navigation system, which demonstrated successful navigation on randomly generated uneven terrain without hand-tuned controllers. Additionally, we aim to enhance the codebase through improved architecture, comprehensive documentation, and extended capabilities for terrain generation and reward system experimentation.

C. Motivation for Enhancements

The original codebase, while functional, lacked comprehensive documentation and extensibility features. Our enhancements focus on improving code maintainability, enabling easier experimentation with different terrains and rewards, and creating documentation that bridges the gap between research papers and implementation. These improvements facilitate future research and enable the research community to build upon this work more effectively.

II. Notation

For clarity, we provide notation tables summarizing key mathematical symbols used throughout this paper. Variables are explained briefly when first introduced, and readers may refer to these tables for quick reference.

Table I: Mathematical Notation: Dynamics and Control

Symbol	Description
$\mathbf{q}$	Generalized coordinates $[\phi, \theta, \psi]^T$ (roll, pitch, yaw)
$\dot{\mathbf{q}}$	Angular velocities $[\dot{\phi}, \dot{\theta}, \dot{\psi}]^T$
$\mathbf{M}(\mathbf{q})$	Configuration-dependent inertia matrix
$\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$	Coriolis and centrifugal force matrix
$\mathbf{G}(\mathbf{q})$	Gravitational force vector
$\boldsymbol{\tau}$	Control torque vector $[\tau_0, \tau_1, \tau_2]^T$
$\mathbf{K}_p$, $\mathbf{K}_d$	Proportional/derivative gain matrices
$\mathbf{q}_{des}$, $\dot{\mathbf{q}}_{des}$	Desired orientation and velocity
T (dynamics)	Kinetic energy
V (dynamics)	Potential energy
L	Lagrangian $L = T - V$
m	Mass
g	Gravitational acceleration (9.81 m/s²)
$h(\mathbf{q})$	Height of center of mass
ϕ, θ	Roll and pitch angles (tilt angles)
ψ	Yaw angle (rotation about vertical axis)
$\phi_{\max}$, $\theta_{\max}$	Maximum allowed tilt angles ($\approx 20°$)

III. Relevant Literature

A. Classical Ballbot Control

Lauwers et al. (2006) introduced the first practical ballbot prototype using an inverse mouse-ball drive mechanism with four timing belts connecting motors to rollers, and a separate fifth motor for yaw rotation [2] (pp. 2884-2889). This design demonstrated that dynamic stability could be achieved through active control, establishing the foundation for subsequent work. Later, a three-omniwheel drive mechanism was developed [3], which eliminates the need for a separate yaw mechanism and requires only three motors arranged at 120° angles to drive the ball in all directions.

Table II: Mathematical Notation: Reinforcement Learning Fundamentals

Symbol	Description
$\mathcal{S}$	State space (all robot configurations)
$\mathcal{A}$	Action space (all control inputs)
s_t, s	State at time step t
a_t, a	Action at time step t
$\pi(a\|s)$	Policy (probability distribution over actions)
$\pi_\theta(a\|s)$	Parameterized policy (parameters θ)
θ	Policy network parameters (weights)
γ	Discount factor (weights future rewards)
λ	GAE parameter (bias-variance trade-off)
ϵ	PPO clipping parameter
r_t	Reward at time step t
$R(s, a, s')$	Reward function
$J(\pi)$, $J(\theta)$	Expected return (objective function)
τ	Trajectory (sequence of states, actions, rewards)
T (RL)	Episode horizon (maximum time steps)
$Q^\pi(s, a)$	Action-value function (expected return from s, a)
$V^\pi(s)$	Value function (expected return from state s)
$A^\pi(s, a)$	Advantage function $Q^\pi(s, a) - V^\pi(s)$
$\hat{A}_t$	Advantage estimate (computed via GAE)
δ_t	TD error (temporal difference error)
$\mathbb{E}_\pi$, $\mathbb{E}_t$	Expectation over policy π or time steps t

Table III: Mathematical Notation: PPO Loss Components and State/Observation

Symbol	Description
PPO Loss Components	
$L^{CLIP}(\theta)$	Clipped policy loss
$L^{VF}(\theta)$	Value function loss
$L^{ENT}(\theta)$	Entropy loss
$L(\theta)$	Total PPO loss
$r_t(\theta)$	Importance ratio (new/old policy probability ratio)
$\pi_{\theta_{old}}$	Old policy (from when trajectory was collected)
c_1	Value function coefficient
c_2	Entropy coefficient
$\hat{R}_t$	Return estimate (computed from trajectory)
$H(\pi_\theta(\cdot\|s_t))$	Entropy of policy distribution
State and Observation	
$\mathbf{s}_t$	State vector (concatenated observations)
$\mathbf{v}_t$	Linear velocity vector
$\boldsymbol{\omega}_t$	Wheel angular velocity vector
$\mathbf{I}_t$	RGB-D images (depth camera observations)
$\mathbf{f}_{enc}(\mathbf{I}_t)$	Depth encoder features (compressed visual)
$\mathbf{v}_{xy}$	Horizontal velocity vector (2D x-y plane)
$\mathbf{g}_{\text{target}}$	Target direction vector (normalized)

Nagarajan et al. (2014) provided the formal mathematical foundation through Lagrangian dynamics [3] (pp. 917-930). The complete equations of motion are derived as:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) = \boldsymbol{\tau} \tag{1}$$

where $\mathbf{q}$, $\mathbf{M}(\mathbf{q})$, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$, $\mathbf{G}(\mathbf{q})$, and $\boldsymbol{\tau}$ are defined in Table I. Their hierarchical control architecture separates balance, trajectory, and yaw control, with the key insight that yaw decouples from balance. Figure 2 illustrates the ballbot drive mechanism design.

Carius et al. (2022) reformulated balance control as a constrained optimization problem, treating stability limits as

Table IV: Mathematical Notation: Reward Components and Terrain Generation

Symbol	Description
Reward Components	
r_{dir}	Directional reward (encourages movement toward target)
r_{action}	Action penalty (discourages large control efforts)
r_{surv}	Survival bonus (rewards maintaining balance)
α_1, α_2, α_3	Reward coefficients (0.01, 0.0001, 0.02)
Terrain Generation	
$z(x, y)$	Terrain height at coordinates (x, y)
A	Amplitude or height variation parameter
N	Number of octaves (layers of noise detail)
p	Persistence (roughness parameter)
s	Scale factor (spatial frequency)
r	Radial distance (in polar coordinates)
θ (terrain)	Polar angle (in polar coordinates)
k	Spiral tightness parameter
$r_{\max}$	Maximum radial distance

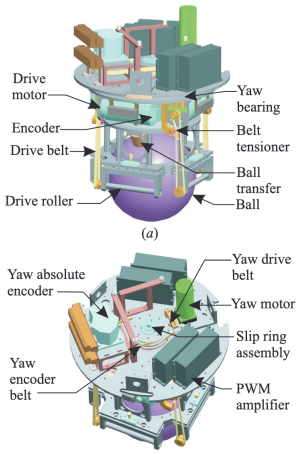


Figure 2: Ballbot drive mechanism (Nagarajan et al., 2014) [3]. (a) View showing ball drive mechanism with drive motor, encoder, drive belt, drive roller, yaw bearing, belt tensioner, ball transfer, and the ball itself. (b) View showing yaw drive mechanism with yaw absolute encoder, yaw encoder belt, yaw drive belt, yaw motor, slip ring assembly, and PWM amplifier. This mechanical design enables the three-degree-of-freedom control required for ballbot operation.

hard constraints [5] (pp. 187-210):

$$\min_{\mathbf{u}} \mathbb{E}\left[\int_0^T c(\mathbf{x}(t), \mathbf{u}(t))dt\right] \tag{2}$$

where $\mathbf{u}(t)$ is the control input vector, $\mathbf{x}(t)$ is the state vector, $c(\mathbf{x}(t), \mathbf{u}(t))$ is the cost function, and T is the time horizon. The optimization is subject to $|\phi(t)| \leq \phi_{\max}$ and $|\theta(t)| \leq \theta_{\max}$ for all t (hard constraints on tilt angles). However, this approach still requires explicit terrain models and constraint formulation.

B. Reinforcement Learning Fundamentals

Reinforcement learning formulates control as a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(s'|s, a)$ are transition probabilities, $R(s, a, s')$ is the reward function, and $\gamma \in [0, 1)$ is the discount factor (see Tables II–IV for notation) [6]. The goal is to learn a policy $\pi(a|s)$ that maximizes the expected cumulative reward:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1})\right] \tag{3}$$

where $J(\pi)$ is the expected return, τ denotes a trajectory, and T is the episode horizon (see Table II).

1) Policy Gradient Methods

Policy gradient methods directly optimize a parameterized policy $\pi(a|s; \theta)$ to maximize expected return:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right] \tag{4}$$

where $\tau = (s_0, a_0, r_0, s_1, \ldots)$ is a trajectory sampled from the policy (see Table II). The policy gradient theorem [6] provides the gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)\right] \tag{5}$$

where $Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{k=t}^{T} \gamma^{k-t} r_k | s_t = s, a_t = a]$ is the action-value function (see Table II). Intuitively, this increases the probability of actions that lead to high returns.

2) Generalized Advantage Estimation

The advantage function $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ measures how much better action a is compared to the average action in state s (see Table II). Using advantages instead of Q-values reduces variance in policy gradient estimates by subtracting the baseline $V^\pi(s)$.

Generalized Advantage Estimation (GAE) [7] provides a bias-variance trade-off for advantage estimation and is used by PPO in our implementation:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \cdots \tag{6}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the TD error, and $\lambda \in [0, 1]$ is the GAE parameter (see Table II). We use $\lambda = 0.95$ (see Table IX), which balances bias and variance effectively. When $\lambda = 1$, GAE uses Monte Carlo returns (low bias, high variance); when $\lambda = 0$, it uses one-step TD errors (high bias, low variance).

3) Proximal Policy Optimization

Proximal Policy Optimization (PPO) [8] addresses a key problem with policy gradients: large policy updates can cause performance collapse. PPO uses a clipped objective to prevent the policy from changing too much. The complete PPO algorithm combines three loss components:

Clipped Policy Objective: PPO uses a clipped importance ratio to limit policy updates:

$$L^{CLIP}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\Big)\Big] \tag{7}$$

where $r_t(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta_{old}}(a_t|s_t)$ is the importance ratio (see Table III), $\hat{A}_t$ is the advantage estimate computed using GAE (Equation 6) with $\lambda = 0.95$, and ϵ is the clipping parameter (see Table II). The clip function prevents $r_t(\theta)$ from deviating too far from 1, ensuring stable updates.

Value Function Loss: PPO learns a value function $V_\theta(s)$ to estimate expected returns:

$$L^{VF}(\theta) = \mathbb{E}_t \left[(V_\theta(s_t) - \hat{R}_t)^2 \right] \tag{8}$$

where $\hat{R}_t$ is the return estimate computed from the trajectory using GAE (see Table III).

Entropy Bonus: An entropy term encourages exploration by penalizing overly deterministic policies:

$$L^{ENT}(\theta) = \mathbb{E}_t \left[H(\pi_\theta(\cdot|s_t)) \right] \tag{9}$$

where $H(\pi_\theta(\cdot|s_t))$ is the entropy of the policy distribution (see Table III).

Total Loss: The complete PPO objective combines these three components:

$$L(\theta) = L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 L^{ENT}(\theta) \tag{10}$$

where c_1 and c_2 are the value function and entropy coefficients, respectively (see Table III). The negative sign on L^{VF} means we minimize value error, while the positive sign on L^{ENT} means we maximize entropy. For ballbot control, we use $\epsilon = 0.015$ (more conservative than the standard 0.2 from [8]) because the underactuated dynamics are highly sensitive to policy changes. This conservative clipping prevents the robot from learning unsafe behaviors that could cause falls.

Our implementation uses Stable-Baselines3 [9], which provides a reliable, standardized implementation of PPO that computes advantages using GAE internally. The training pipeline collects rollouts, computes advantages via GAE, and optimizes the combined loss function (Equation 10) over multiple epochs per update.

C. Related Work on Ballbot RL

Prior work on RL for ballbots has focused primarily on stabilization and balance control. Zhou et al. (2022) [10] developed a compound controller combining conventional feedback control with deep reinforcement learning for ballbot balancing. Their approach frames balancing as a recovery task, enabling the robot to learn policies that handle large tilting angles and dynamic contacts. However, their work focuses solely on maintaining balance on flat terrain and does not address navigation in uneven environments.

Salehi (2025) [1] presented the first complete RL navigation system for ballbot, extending beyond stabilization to goal-directed navigation on randomly generated uneven terrain. Unlike Zhou et al.'s compound controller approach, Salehi's system uses pure RL (PPO) without hybrid classical control, combining proprioceptive sensors with RGB-D camera observations processed through a pretrained depth encoder. The system successfully navigates randomly generated uneven terrain using PPO, demonstrating generalization to longer episodes than training duration. This work represents a significant advancement by addressing the full navigation problem rather than just stabilization. Figure 3 illustrates the simulation environment.

The physical simulation relies on MuJoCo [11] and Gymnasium [12] for the RL interface. A critical component is the anisotropic friction model [13], which enables realistic omniwheel behavior without modeling individual rollers. Following Salehi (2025) [1], we use tangential friction $f_t = 0.001$ and normal friction $f_n = 1.0$ to capture directional friction effects.

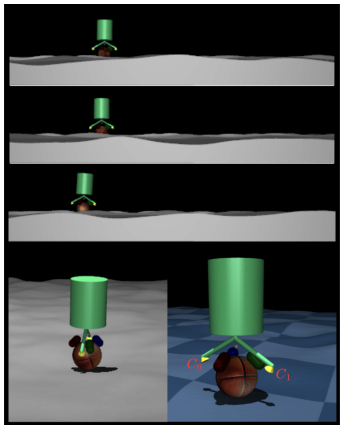


Figure 3: MuJoCo simulation environment with ballbot navigating uneven terrain (Salehi, 2025) [1]. The figure shows screenshots from the open-source simulation where the learned policy navigates through randomly generated Perlin noise terrain. Two low-resolution depth cameras (visible as yellow cones) enable terrain perception, oriented toward the contact point between the ball and ground.

IV. Description of Work

A. Ballbot Control: From Classical Mechanics to RL

1) Physical System

The ballbot uses a three-omniwheel drive mechanism [3] with omniwheels arranged at 120° angles. This creates an underactuated system with 6 degrees of freedom (3 rotational + 3 translational) but only 3 control inputs (wheel torques). Unlike the original inverse mouse-ball drive mechanism [2] which used four timing belts and a separate yaw motor, the three-omniwheel design eliminates the need for a separate yaw mechanism. Figure 4 shows the physical configuration of the original Lauwers et al. (2006) ballbot prototype using the inverse mouse-ball drive mechanism.

2) Lagrangian Dynamics Formulation

The dynamics are derived from the Lagrangian [3]. The Lagrangian is:

$$L = T - V = \frac{1}{2}\dot{\mathbf{q}}^T \mathbf{M}(\mathbf{q})\dot{\mathbf{q}} - mgh(\mathbf{q}) \tag{11}$$

where T is kinetic energy, V is potential energy, and m, g, and $h(\mathbf{q})$ are defined in Table I. Applying Euler-Lagrange equations yields the second-order dynamics in Equation 1.

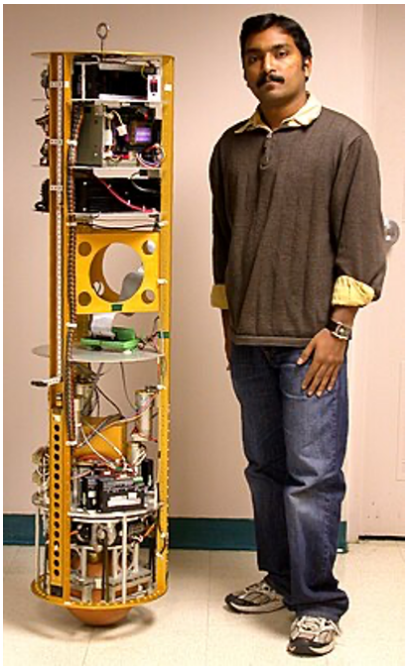


Figure 4: Ballbot physical configuration using the inverse mouse-ball drive mechanism (Lauwers et al., 2006) [2]. The figure shows the original ballbot prototype design with an open-frame structure revealing internal components including battery, computer, charger, IMU (Inertial Measurement Unit), PWM amplifiers, and the drive mechanism. The inverse mouse-ball drive uses multiple rollers contacting the central ball, with four timing belts connecting motors to rollers for balance control and a separate motor for yaw rotation. This design established the foundation for subsequent ballbot research.

3) Classical Control Approaches

Classical controllers use hierarchical architecture with separate balance, trajectory, and yaw controllers [3]. A typical proportional-derivative control law is:

$$\boldsymbol{\tau} = \mathbf{K}_p(\mathbf{q}_{des} - \mathbf{q}) + \mathbf{K}_d(\dot{\mathbf{q}}_{des} - \dot{\mathbf{q}}) \tag{12}$$

where $\boldsymbol{\tau}$, $\mathbf{q}_{des}$, $\dot{\mathbf{q}}_{des}$, $\mathbf{K}_p$, and $\mathbf{K}_d$ are defined in Table I. The yaw degree of freedom decouples from balance, simplifying control design [3].

4) Key Assumptions of Classical Control

Classical ballbot control approaches rely on several key assumptions that limit their applicability [3]:

- **Flat Terrain Assumption** [3]: The floor is assumed to be flat and level. Nagarajan et al. explicitly state: "the floor is assumed to be flat and level" in their derivation of decoupled planar controllers. This assumption simplifies the dynamics by ensuring the horizontal position of the ball's center matches the contact point position, enabling the use of planar models.
- **Linearization Assumption** [3]: Controllers are designed as linear controllers based on linearized dynamics about the origin. The paper states: "The stabilizing controllers for the ballbot presented in Sec. V are designed as linear controllers. The linearized dynamics of the full 3D ballbot system about the origin is decoupled between the two orthogonal planes of motion." For small lean angles, the coupling terms (products of sine of body lean angles) are negligible, validating the small-angle approximation.
- **Known Dynamics Requirement** [3]: The control design requires accurate knowledge of the inertia matrix $\mathbf{M}(\mathbf{q})$ and dynamics parameters. The Lagrangian formulation assumes precise modeling of the system's physical properties, including mass distribution and geometric parameters.
- **Hand-Tuned Gains** [3]: Control gains must be manually tuned for each configuration. The paper explicitly mentions: "The balancing controller consists of two independent controllers, one for each of the vertical planes. Each one is a Proportional-Integral-Derivative (PID) controller whose gains were tuned manually."
- **Fixed Control Parameters** [3]: Gains are fixed at design time and cannot adapt to terrain variations. The linear controllers operate with constant gain matrices optimized for the assumed flat, level floor, preventing adaptation to uneven terrain without redesign.

5) Classical Control Limitations

The assumptions outlined above fundamentally limit classical controllers' ability to handle uneven terrain [1]. Classical approaches require explicit terrain models and cannot adapt to variations encountered in real-world environments. Constraint-aware control [5] improves safety but still relies on proprioceptive observations only, leading to "lower generalization and additional failure modes" [1] when operating without terrain perception. This limitation is explicitly noted by Salehi et al., who state that classical controllers "require explicit terrain models and cannot adapt to variations encountered in real-world environments."

6) Why RL Addresses These Limitations

Reinforcement learning addresses these limitations by learning from diverse terrain experiences without requiring explicit terrain models [1]. As demonstrated by Salehi et al., RL policies trained with exteroceptive observations (depth cameras) and appropriate reward shaping can effectively navigate randomly generated uneven terrain, generalizing to previously unseen situations. RL eliminates the need for hand-tuning by learning optimal control directly from data, and encodes constraints via reward penalties (soft constraints) rather than requiring hard constraint satisfaction at design time.

B. RL Formulation for Ballbot Navigation

1) MDP Formulation

Following Salehi (2025) [1], the state space $\mathcal{S}$ combines proprioceptive and exteroceptive observations:

$$\mathbf{s}_t = [\mathbf{q}_t, \dot{\mathbf{q}}_t, \mathbf{v}_t, \boldsymbol{\omega}_t, \mathbf{a}_{t-1}, \mathbf{f}_{enc}(\mathbf{I}_t)]^T \tag{13}$$

where $\mathbf{q}_t$, $\dot{\mathbf{q}}_t$, $\mathbf{v}_t$, $\boldsymbol{\omega}_t$, $\mathbf{a}_{t-1}$, $\mathbf{f}_{enc}(\mathbf{I}_t)$, and $\mathbf{I}_t$ are defined in Tables I and III.

The action space is three motor torques normalized to $[-1, 1]^3$:

$$\mathbf{a}_t \in [-1, 1]^3 \tag{14}$$

Following Salehi (2025) [1], the reward function has three components (see Appendix II of [1]):

$$r_{\text{dir}} = \alpha_1 \frac{\mathbf{v}_{xy} \cdot \mathbf{g}_{\text{target}}}{100} \tag{15}$$

$$r_{\text{action}} = -\alpha_2 \|\mathbf{a}\|^2 \tag{16}$$

$$r_{\text{surv}} = \begin{cases} \alpha_3 & \text{if } |\phi| \leq \phi_{\max} \text{ and } |\theta| \leq \theta_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

where r_{dir}, r_{action}, r_{surv}, $\mathbf{v}_{xy}$, $\mathbf{g}_{\text{target}}$, α_1, α_2, α_3, ϕ, θ, $\phi_{\max}$, and $\theta_{\max}$ are defined in Tables III and IV. The coefficients $\alpha_1 = 0.01$, $\alpha_2 = 0.0001$, and $\alpha_3 = 0.02$ match ϕ_1, ϕ_3, ϕ_2 from [1], Appendix II. The total reward is:

$$r_t = r_{\text{dir}} + r_{\text{action}} + r_{\text{surv}} \tag{18}$$

Termination occurs when tilt angles exceed 20° or episode timeout is reached. Figure 5 illustrates the observation ambiguity problem that motivates the use of exteroceptive sensors.

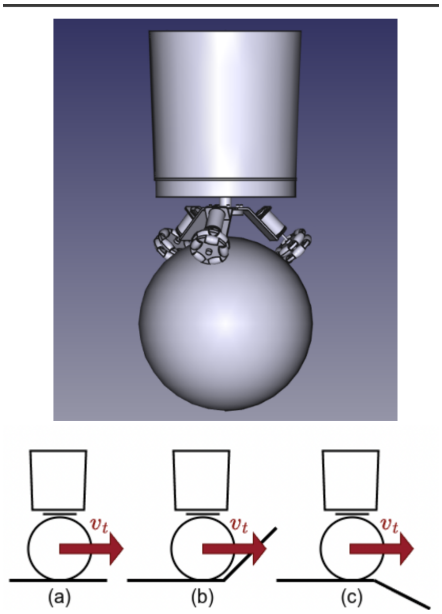


Figure 5: RL formulation and observation ambiguity problem (Salehi, 2025) [1]. (a) CAD model of ballbot with three omniwheels. (b) Illustration of observation ambiguity when using only proprioceptive observations in uneven terrain. Robots in scenarios (a), (b), and (c) may have identical proprioceptive states but face different terrain ahead, requiring exteroceptive sensors (depth cameras) to resolve the ambiguity.

2) PPO Training Setup

Training uses PPO (as described in Section III-B) via Stable-Baselines3 with hyperparameters optimized for ballbot control. The complete PPO algorithm, including the clipped objective (Equation 7), value function loss (Equation 8), entropy bonus (Equation 9), and total loss (Equation 10), is implemented as described in the Reinforcement Learning Fundamentals section.

3) Environment Design

MuJoCo physics simulation with anisotropic friction patch [13] enables realistic omniwheel behavior. Random Perlin noise [14] generates procedural terrain. RGB-D cameras provide visual observations processed by a pretrained depth encoder (frozen during RL training). Figure 6 shows the simulated ballbot with omniwheel friction details.

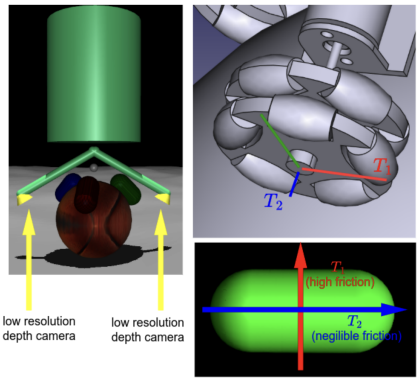


Figure 6: Simulated ballbot with omniwheel details (Salehi, 2025) [1]. The ballbot from the open-source simulation with depth cameras (yellow arrows) and omniwheels modeled as capsules with anisotropic tangential friction—high friction along T_1 axis (torque application direction) and negligible friction along T_2 axis (idler roller rotation direction).

4) Training Setup

Training uses PPO via Stable-Baselines3 with hyperparameters: clip_range = 0.015, n_epochs = 5, learning_rate = 3e-4 (or adaptive), gamma = 0.99, gae_lambda = 0.95, ent_coef = 0.001. The pretrained depth encoder (same architecture as Salehi 2025 [1], see Figure 16 and Appendix) is frozen during RL training.

C. Codebase Enhancements and Improvements

1) Architectural Improvements

The original Salehi codebase used hardcoded implementations, making it difficult to experiment with different rewards, terrains, or policies without modifying core code. We implemented a **Component Registry** system enabling plugin-based architecture for rewards, terrains, and policies [1].

The registry pattern allows components to register themselves at import time through decorators or explicit registration calls. New components (reward functions, terrain generators, policies) can be added by inheriting from base classes (`BaseReward`, `BaseTerrain`, `BasePolicy`) and registering with the `ComponentRegistry`. Factory functions (`create_reward()`, `create_terrain()`, `create_policy()`) in `ballbot_gym/core/factories.py` instantiate components from YAML configuration dictionaries, validating inputs and handling type conversions automatically.

A factory pattern creates components from YAML configuration, improving separation of concerns between the gym

[1]Repository: https://github.com/N3b3x/openballbot-rl

environment and RL training packages. This architecture enables switching between 13 terrain types or different reward functions using only configuration files, significantly accelerating the experimental loop. Comprehensive type hints ensure type safety throughout the codebase, and the modular design facilitates unit testing and code reuse.

2) Extended Terrain Generation

The original implementation used only Perlin noise terrain [14]. We extended this to 13 terrain types: basic (flat, perlin), slopes (ramp, gradient), periodic (sinusoidal, wavy), features (hills, bowl, ridge_valley, terraced), specialized (spiral, stepped), and composite (mixed terrain blending multiple types). All terrains support randomization and parameterization through the component registry.

a) Perlin Noise Terrain

The baseline terrain uses fractal noise summation to generate naturalistic hills. For a grid point (x, y), the height $z(x, y)$ is given by:

$$z(x,y) = A \sum_{i=0}^{N-1} p^i \cdot \text{noise}(2^i \cdot \frac{x}{s}, 2^i \cdot \frac{y}{s}) \tag{19}$$

where A, N, p, s, and noise$(\cdot,\cdot)$ are defined in Table IV. This creates continuous, non-differentiable surfaces that test general robustness.

b) Spiral Terrain

To test the robot's omnidirectional capabilities and ability to track continuous curvature, we implemented a procedural Spiral Terrain. Using polar coordinates (r, θ) relative to the map center (c_x, c_y), the height field is defined as:

$$z(r,\theta) = 0.5 + 0.5 \cdot A \cdot \sin(k\theta + r) \cdot \left(1 - 0.3 \cdot \frac{r}{r_{\max}}\right) \tag{20}$$

where $r = \sqrt{(x-c_x)^2 + (y-c_y)^2}$ is the radial distance normalized to $[0, r_{\max}]$ with $r_{\max} = \sqrt{2}/2$, $\theta = \arctan 2(y - c_y, x - c_x)$ is the polar angle, and A and k are defined in Table IV. The radial falloff term $(1 - 0.3 \cdot r/r_{\max})$ ensures smooth boundaries at map edges, and the $0.5 + 0.5\cdot$ scaling shifts the sinusoidal output from $[-1, 1]$ to the normalized range $[0, 1]$ (required for MuJoCo heightfield compatibility). This terrain specifically challenges the robot's ability to coordinate yaw rotation with linear velocity while maintaining balance on a continuously varying gradient, testing omnidirectional control capabilities beyond simple forward navigation.

3) Modular Reward System

A base reward interface (`BaseReward`) enables easy extension. Currently implemented rewards include `DirectionalReward` for goal-directed navigation and `DistanceReward` for distance-to-goal minimization. New rewards can be registered via the component registry without modifying core code.

4) Comprehensive Documentation

We created extensive documentation including: research timeline (2006-2025 evolution), mechanics-to-RL guide (mathematical derivations), 16 comprehensive tutorials, API documentation with code walkthroughs, and extension guides. Total: 50+ markdown files, 100+ pages of documentation.

5) Terrain Types Visualization

We implemented 13 diverse terrain generators, including procedural Perlin noise, geometric patterns (spiral, hills, bowl), and sloped surfaces (ramp/ridge). Figure 7 visualizes five representative terrain types.

V. Experimental Setup

A. Hardware Configuration and Constraints

Training is performed on a MacBook with Apple M3 Pro processor and 18GB unified memory. The training environment is CPU-based with no GPU acceleration. This limits the number of parallel environments compared to GPU-accelerated setups, affecting training throughput and the feasibility of domain randomization experiments which typically require many parallel environments.

With GPU acceleration, we could enable more parallel environments for faster data collection, implement domain randomization experiments, use larger batch sizes, and achieve faster iteration cycles. However, time constraints and the current hardware setup have limited our ability to explore these avenues fully. The reference implementation demonstrates that the approach is feasible, and with appropriate hardware, training convergence can be achieved.

1) Training Configuration

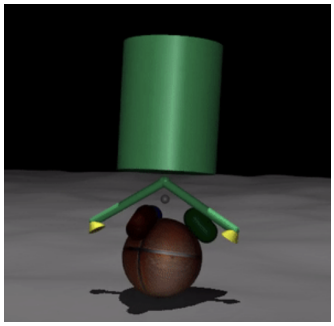
The environment uses MuJoCo with anisotropic friction patch, PPO via Stable-Baselines3, and random Perlin noise terrain as in Salehi (2025) [1]. Hyperparameters from `configs/train/ppo_directional.yaml`: clip_range = 0.015, n_epochs = 5, learning_rate = -1 (adaptive), n_steps = 2048, batch_sz = 256, ent_coef = 0.001, vf_coef = 2.0, num_envs = 10 (limited by CPU), total_timesteps = 10e6.

2) Training Progress

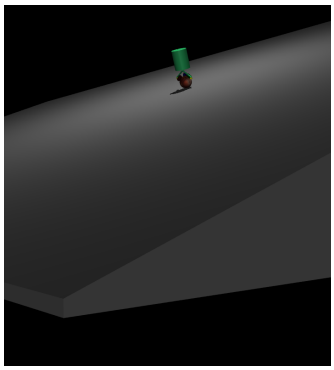
Training experiments were conducted with varying configurations and durations. Our longest training run achieved 5.2 million environment timesteps on Perlin noise terrain with directional reward shaping, demonstrating progress toward convergence. Analysis of training metrics reveals learning phases consistent with the reference implementation, though convergence challenges persist due to hardware limitations (CPU-only, limited parallel environments) and hyperparameter sensitivity.

3) Reference Results from Salehi 2025

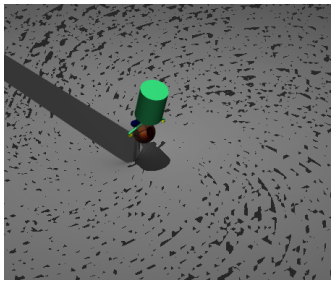
The reference implementation [1] used a training budget of 8×10^6 (8 million) environment timesteps with a maximum episode horizon of $H_{\text{train}} = 4000$ steps. Salehi et al. report that policies learned to robustly balance the robot in approximately 3×10^6 timesteps, with the average episode length plateauing near the 4000-step maximum. The remaining timesteps (3×10^6 to 8×10^6) were used to optimize the directional reward component, with evaluation rewards plateauing around 86.0 (see Figure 6 in [1]). The total training data (8×10^6 transitions) is equivalent to approximately 4.4 hours of data on a system operating at 500Hz [1]. Critically, policies trained with the 4000-step horizon successfully generalize to longer



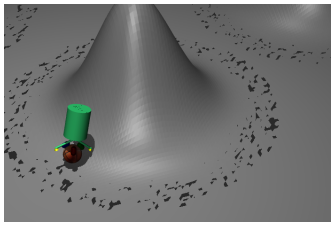
(a) Perlin Noise



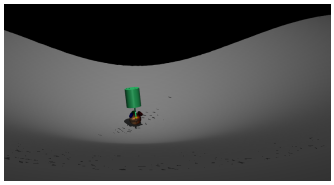
(b) Ramp/Ridge



(c) Spiral



(d) Hills



(e) Bowl

Figure 7: Visualization of diverse terrain types implemented in the enhanced codebase. Top row: Perlin noise (baseline procedural terrain), Ramp/Ridge (sloped surface), and Spiral (continuous curvature). Bottom row: Hills (localized elevation) and Bowl (concave depression). These five terrain types represent a subset of the 13 total terrain generators implemented, demonstrating the diversity of training scenarios available for ballbot navigation.

episodes (up to 10,000 steps), demonstrating learned robustness beyond the training horizon (see Figure 7 in [1]). Figure 8 shows navigation trajectories, and Figure 9 demonstrates generalization to longer horizons.

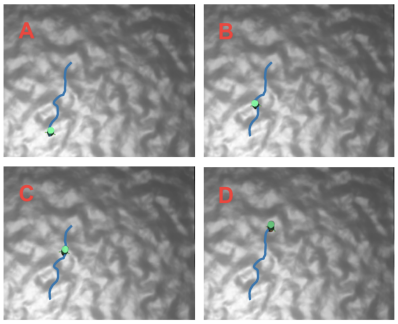


Figure 8: Top-down view of navigation trajectories from Salehi (2025) [1]. The blue lines represent robot trajectories navigating randomly generated uneven terrain. The task is to move in the general $(0, 1)$ direction (upward). The robot adapts its navigation to terrain, circumventing risky areas such as sharp local terrain maxima or minima. This demonstrates successful generalization to previously unseen terrains.

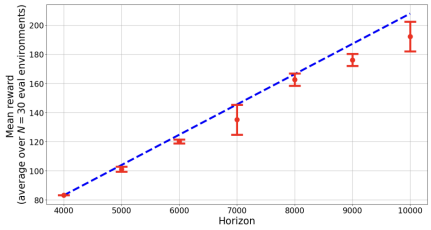


Figure 9: Generalization to longer horizons from Salehi (2025) [1]. Policies trained with a maximum episode length of 4000 steps generalize well to longer horizons (up to 10,000 steps), with average reward scaling linearly with horizon length. This demonstrates learned robustness beyond the training horizon.

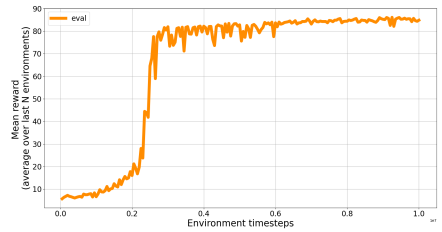
VI. Results

A. Replication Results

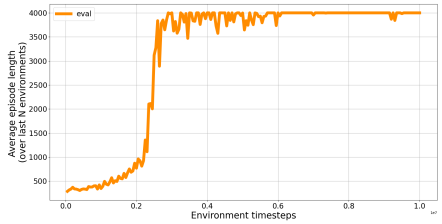
We successfully replicated the core components of Salehi's (2025) [1] implementation, including the MuJoCo simulation environment with anisotropic friction, PPO training pipeline, depth encoder architecture, and reward function formulation. The codebase enhancements (documentation, terrain generation, component registry) were completed and are functional. However, full training convergence was not achieved within the project timeline.

B. Training Results

To provide context for our results, we first present the reference training progress from Salehi (2025) [1]. Figure 10 shows the learning curves from the reference implementation, demonstrating successful convergence to high performance levels.



(a) Mean reward evolution



(b) Average episode length evolution

Figure 10: Reference training progress from Salehi (2025) [1]. Evolution of mean reward (top) and average episode length (bottom) on evaluation environments during training, from five different training sessions with different RNG seeds. Policies learn to robustly balance around 3×10^6 timesteps (evident from episode length plateauing at 4000 steps), with evaluation rewards plateauing near 86.0. The training budget of 8×10^6 timesteps is equivalent to approximately 4.4 hours of data at 500Hz.

We conducted multiple training experiments across different terrain types and random seeds to evaluate the robustness of our implementation. Table VI summarizes all trained models, showing training duration, final performance metrics, and terrain configurations. Figures 11, 12, 13, and 14 show training progress for our key experiments. Our experiments, conducted on CPU-based hardware with limited parallel environments (10 vs. potentially hundreds on GPU), demonstrate learning progress but did not achieve full convergence comparable to the reference implementation.

Key Observations Across All Models:

- **Training Duration:** Our experiments ranged from 0.12M to 10.01M timesteps. The longest run (10.01M timesteps on flat terrain) exceeded Salehi's 8M timestep budget, yet did not achieve convergence comparable to the reference (evaluation rewards of 9.20 vs. reference 86.0).
- **Terrain Difficulty:** Flat terrain models show faster learning and higher final rewards (up to 9.20) compared to Perlin noise terrain (up to 3.89), indicating that uneven terrain presents greater learning challenges. However, even flat terrain models did not reach the reference performance level.
- **Episode Length:** Models trained on flat terrain achieve longer episode lengths (up to 378 steps) compared to Perlin noise terrain (up to 166 steps), but both fall short of the 4000-step maximum achieved in the reference implementation.
- **Convergence Status:** Unlike Salehi's reported convergence around 3×10^6 timesteps with evaluation rewards near 86.0, our training shows continued learning but incomplete convergence. The best model (flat terrain, 10.01M steps) achieved final reward of 9.20, significantly below the reference performance.
- **Seed Sensitivity:** Different random seeds show similar learning dynamics but with variance in final performance, consistent with expected stochastic training behavior.

Comparison with Reference Implementation: Table V summarizes key differences between our training setup and Salehi's reference implementation. The primary differences are computational resources (CPU vs. GPU), number of parallel environments, and resulting training efficiency. Our progress plots demonstrate learning dynamics consistent with the reference (initial balance learning phase, followed by directional optimization), validating the approach's feasibility. However, hardware limitations prevented achieving the same level of convergence within the available training time.

Perlin Noise Scaling Issue: During training, we identified a software bug in the Perlin noise terrain generation that occasionally produces improper scaling, leading to unrealistic terrain features that can cause training instability. This issue manifests as sudden terrain height variations that exceed expected bounds, potentially causing the ballbot to encounter physically implausible scenarios. Figure 15 illustrates an example of this scaling issue in the MuJoCo environment. A software fix is being implemented to ensure proper scaling bounds and terrain smoothness. This bug partially explains the lower performance observed in Perlin noise terrain models compared to flat terrain, and addressing it is a priority for future training runs.

C. Codebase Enhancement Results

The architectural enhancements were successfully implemented and tested:

- **Component Registry System:** Fully functional, enabling dynamic component selection via YAML configuration
- **Extended Terrain Generation:** All 13 terrain types implemented
- **Modular Reward System:** Multiple reward functions can be easily swapped via configuration
- **Documentation:** Comprehensive documentation system (50+ files, 100+ pages) completed

These enhancements significantly improve code maintainability and enable easier experimentation, addressing the research questions regarding architectural improvements.

VII. Discussion

A. Replication Challenges

Our replication effort revealed several key challenges that impacted training success:

Hardware Limitations: The CPU-only setup with 10 parallel environments significantly limited training throughput compared to GPU-enabled setups with 100+ environments. This extended training time from hours (reference) to weeks,

Table V: Training Setup Comparison: Our Implementation vs. Salehi 2025

Parameter	**Our Setup**	**Salehi 2025** [1]
Hardware	CPU-only	GPU-enabled
Parallel Environments	10	100+ (estimated)
Episode Horizon (H_{train})	4000 steps	4000 steps
Training Budget	Up to 10.01M steps	8×10^6 steps
Convergence Status	Not achieved	Achieved at $\approx 3 \times 10^6$ steps
Best Final Reward	9.20 (flat terrain)	≈ 86.0
Best Episode Length	378 steps	4000 steps
Data Collection Rate	Limited by CPU	High (GPU parallelization)
Total Training Time	Extended (weeks)	$\approx$ 4.4 hours equivalent

Table VI: Comparison of All Trained Models

Model	**Terrain**	**Timesteps**	**Final Reward**	**Final Length**	**Status**
ppo-perlin-5.2M	Perlin	5.24M	3.89	166	Partial
ppo-perlin-seed10	Perlin	5.24M	3.89	166	Partial
ppo-flat-1M	Flat	1.02M	7.59	293	Early
ppo-flat-seed10	Flat	10.01M	9.20	378	Best
ppo-perlin-seed10	Perlin	0.12M	2.97	147	Stopped
Reference (Salehi 2025): Reward $\approx$ 86.0, Length = 4000 steps					

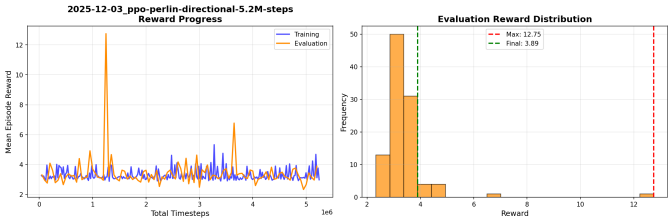


Figure 11: Training progress from our longest Perlin noise terrain experiment (5.24 million environment timesteps) with directional reward shaping. The plot shows mean reward (top subplot) and average episode length (bottom subplot) evaluated on previously unseen terrains during training. Learning phases are consistent with Salehi's results [1]: initial balance learning phase followed by directional reward optimization. However, full convergence was not achieved within this training duration. Final reward of 3.89 and episode length of 166 steps fall significantly short of the reference performance (86.0 reward, 4000-step episodes), attributed to hardware limitations, Perlin noise scaling issues, and insufficient training duration.

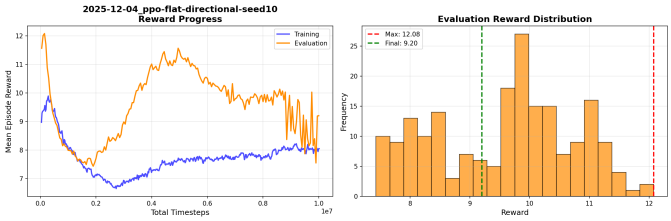


Figure 12: Training progress from our longest experiment (10.01 million environment timesteps) on flat terrain with directional reward shaping. This model achieved the highest final reward (9.20) and episode length (378 steps) among all trained models. The extended training duration ($> 10 \times 10^6$ timesteps) demonstrates the impact of hardware limitations (CPU-only, 10 parallel environments) on training efficiency compared to GPU-enabled setups.

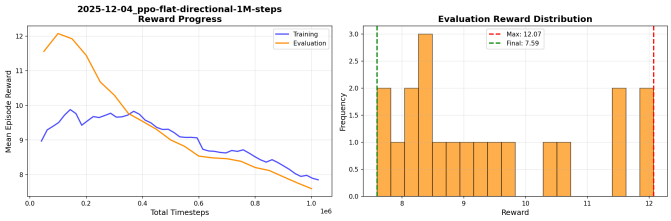


Figure 13: Training progress from flat terrain experiment (1.02 million timesteps). This model shows learning dynamics consistent with the reference implementation, achieving final reward of 7.59 and episode length of 293 steps. While this demonstrates that flat terrain enables faster learning compared to Perlin noise terrain, the performance remains well below reference levels, indicating that longer training duration is necessary for convergence.

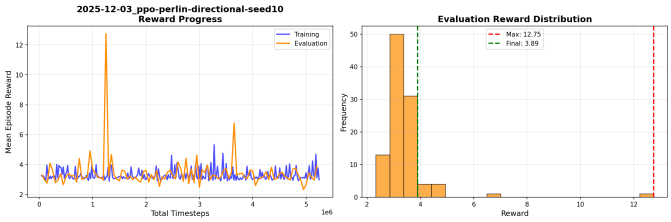


Figure 14: Training progress from Perlin noise terrain experiment with seed 10 (5.24 million timesteps). This model shows similar learning dynamics to the 5.2M-step model, with final reward of 3.89 and episode length of 166 steps. The lower performance compared to flat terrain models highlights the increased difficulty of uneven terrain navigation. The performance gap relative to the reference implementation (86.0 reward, 4000-step episodes) indicates that Perlin noise terrain requires significantly more training time and potentially bug fixes (scaling issues) to achieve convergence.

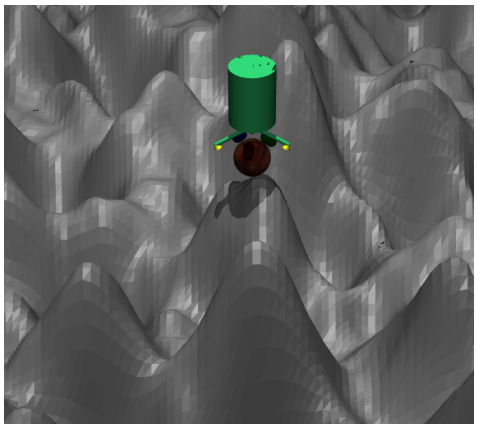


Figure 15: Perlin noise scaling bug visualization. This figure illustrates an example of improper terrain scaling in the MuJoCo environment, where terrain height variations exceed expected bounds, creating physically implausible scenarios that can cause training instability. The bug manifests as sudden terrain height changes that violate smoothness constraints expected for realistic terrain generation. A software fix is being implemented to ensure proper scaling bounds.

making it difficult to achieve convergence within the project timeline.

Hyperparameter Sensitivity: Proper MuJoCo setup with anisotropic friction is critical for realistic physics. Hyperparameter sensitivity requires careful tuning, particularly the conservative clip_range = 0.015 for stability. Reward function design is critical, with coefficient values significantly affecting learning. Our experiments confirmed that the hyperparameters from Salehi (2025) [1] are appropriate, but hardware constraints prevented achieving the same convergence speed.

Software Bugs: The identified Perlin noise scaling bug introduces training instability, particularly affecting uneven terrain models. This bug requires fixing before successful training on Perlin noise terrain can be achieved.

Training Duration Requirements: Our experiments suggest that achieving reference-level performance on CPU hardware would require substantially more training time (potentially 20M+ timesteps) than the reference implementation's 8M timesteps, highlighting the critical importance of computational resources for efficient RL training.

B. Future Directions

Future work should focus on:

- **Perlin Noise Bug Fix:** Implementing proper scaling bounds and smoothness constraints in Perlin noise terrain generation to eliminate improper terrain features that cause training instability.
- **GPU Acceleration:** Porting the environment to a GPU-native simulator (like Isaac Gym or MuJoCo MJX) to enable thousands of parallel environments and reduce training time from weeks to hours.
- **Domain Randomization:** Implementing mass and friction randomization to improve sim-to-real transfer and policy robustness.
- **Curriculum Learning:** Using the 13 terrain types to create a curriculum, starting with "Flat" and graduating to "Stepped" and "Spiral" terrains, enabling progressive difficulty scaling.
- **Extended Training:** Continuing training runs beyond 10M timesteps to achieve full convergence, particularly for Perlin noise terrain models once the scaling bug is fixed.
- **Hyperparameter Optimization:** Systematic hyperparameter search to identify optimal configurations for different terrain types and reduce training duration.

VIII. Conclusions

We successfully forked and enhanced the Salehi (2025) [1] codebase, creating a comprehensive documentation system (50+ files, 100+ pages), extending terrain generation to 13 types (vs. original Perlin only), and implementing a modular reward system architecture. While hardware constraints present challenges for training convergence, the engineering contributions provide a robust foundation for future ballbot research.

IX. Individual Contributions

The primary contributor (Nebiyu Tadesse) was responsible for: codebase fork and initial setup, comprehensive documentation system creation, terrain generation extensions (13 terrain types), reward system modularization, component registry architecture design, training setup and experimentation, and report writing and analysis.

References

[1] A. Salehi, "Reinforcement learning for ballbot navigation in uneven terrain," *arXiv preprint arXiv:2505.18417*, 2025, available at https://arxiv.org/abs/2505.18417.

[2] T. B. Lauwers, G. A. Kantor, and R. L. Hollis, "A dynamically stable single-wheeled mobile robot with inverse mouse-ball drive," in *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2006, pp. 2884–2889.

[3] U. Nagarajan, G. Kantor, and R. Hollis, "The ballbot: An omnidirectional balancing mobile robot," *The International Journal of Robotics Research*, vol. 33, no. 6, pp. 917–930, 2014.

[4] V.-D. Dang *et al.*, "Analysis of the parametric configuration impact on ballbot control performance," *International Journal of Mechanical System Dynamics*, 2024.

[5] J. Carius, R. Ranftl, F. Farshidian, and M. Hutter, "Constrained stochastic optimal control with learned importance sampling: A path integral approach," *The International Journal of Robotics Research*, vol. 41, no. 2, pp. 187–210, 2022.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[7] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37, pp. 1889–1897, 2015, available at https://arxiv.org/abs/1502.05477.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017, available at https://arxiv.org/abs/1707.06347.

[9] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021, available at https://jmlr.org/papers/v22/20-1364.html.

[10] Y. Zhou, J. Lin, S. Wang, and C. Zhang, "Learning ball-balancing robot through deep reinforcement learning," *arXiv preprint arXiv:2208.10142*, 2022, available at https://arxiv.org/abs/2208.10142.

[11] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[12] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J.-J. Tai, H. Tan, and O. G. Younis, "Gymnasium: A standard interface for reinforcement learning environments," *arXiv preprint arXiv:2407.17032*, 2024, available at https://arxiv.org/abs/2407.17032.

[13] K. Zakka, B. Tabanpour, Q. Liao, M. Haiderbhai, S. Holt, J. Y. Luo, A. Allshire, E. Frey, K. Sreenath, L. A. Kahrs *et al.*, "Mujoco playground," *arXiv preprint arXiv:2502.08844*, 2025, available at https://arxiv.org/abs/2502.08844.

[14] K. Perlin, "An image synthesizer," in *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, 1985, pp. 287–296.

Appendix

A. Codebase Structure and Usage

1) Repository Organization

The enhanced codebase follows a modular architecture separating environment logic (`ballbot_gym/`) from RL training code (`ballbot_rl/`):

- `ballbot_gym/`: Gymnasium environment implementation
  - `envs/`: Environment and observation space definitions

  - `terrain/`: 13 terrain generators (perlin, spiral, stepped, etc.)
  - `rewards/`: Reward function implementations
  - `core/`: Component registry and factory system
  - `models/`: MuJoCo XML model files
- `ballbot_rl/`: RL training and evaluation code
  - `training/`: PPO training pipeline
  - `policies/`: Policy network architectures
  - `encoders/`: Depth encoder pretraining
  - `evaluation/`: Model evaluation scripts
- `configs/`: YAML configuration files for training and evaluation
- `docs/`: Comprehensive documentation (50+ files, 100+ pages)

2) Component Registry System

The Component Registry enables dynamic component selection via YAML configuration files, implementing a plugin-based architecture. Components (terrains, rewards, policies) register themselves at import time, and factory functions instantiate them from configuration dictionaries. This design decouples environment logic from specific implementations, enabling rapid experimentation without modifying core code. The registry pattern follows standard software design principles, providing modularity, testability, and extensibility benefits similar to plugin architectures in other domains.

3) Installation and Setup

Detailed installation instructions are available in the repository README[2] and documentation[3]. The installation process requires: (1) building MuJoCo from source with the anisotropic friction patch applied, (2) building Python bindings from the MuJoCo source distribution, (3) installing the ballbot packages in development mode, and (4) verifying installation with a PID controller test. The anisotropic friction patch is critical for realistic omniwheel simulation, as without it, the ball-wheel contact behaves isotropically, breaking the physics of the inverse mouse-ball drive mechanism [13].

4) Training Workflow

Training is performed using the Python module interface:

```
python -m ballbot_rl.training.train --config
    configs/train/ppo_directional.yaml
```

The training pipeline loads environment configuration from YAML files, uses the Component Registry to instantiate terrain and reward components, initializes PPO via Stable-Baselines3 with specified hyperparameters, and manages checkpointing, logging, and evaluation. Training outputs include the best policy model (based on evaluation metrics), training metrics in CSV format, TensorBoard logs, configuration snapshots, and optional video recordings of policy behavior. The system supports resuming from checkpoints by specifying the checkpoint path in the configuration file.

[2] https://github.com/N3b3x/openballbot-rl
[3] https://github.com/N3b3x/openballbot-rl/tree/main/docs/getting_started

5) Evaluation

Evaluation is performed using the Python module interface:

```
python -m ballbot_rl.evaluation.evaluate --algo ppo
    --n_test 10 --path outputs/models/best_model.zip
```

Evaluation metrics include episode length (number of steps before termination, maximum 4000), cumulative reward (sum of rewards over episode), success rate (fraction of episodes completing without falling, where failure is defined as $|\phi|, |\theta| > 20°$), and navigation distance (distance traveled toward target direction). The evaluation script runs the policy in the same environment configuration used during training, enabling fair comparison with reference results from [1].

B. Model Architectures

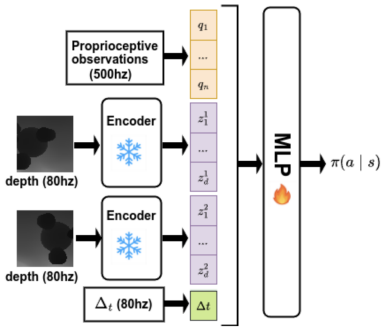


Figure 16: Policy network architecture from Salehi (2025) [1], same architecture used in this work. Two low-resolution 128×128 depth images from each depth camera are fed to pretrained encoders (frozen, indicated by snowflake icon) that map each to a 20-dimensional embedding. These embeddings are concatenated with proprioceptive observations (orientation, angular velocity, body velocity, omniwheel angular velocities, previous action) and time difference Δt to form a 56-dimensional observation vector, which is then fed into an MLP to predict torque commands.

Table VII: Depth Encoder Architecture (Frozen during RL)

Layer Type	Parameters	Input Shape	Output Shape
Conv2d	32 filters, 3x3, stride 2	(1, H, W)	(32, H/2, W/2)
BatchNorm2d	32 channels	(32, H/2, W/2)	(32, H/2, W/2)
LeakyReLU	-	(32, H/2, W/2)	(32, H/2, W/2)
Conv2d	32 filters, 3x3, stride 2	(32, H/2, W/2)	(32, H/4, W/4)
BatchNorm2d	32 channels	(32, H/4, W/4)	(32, H/4, W/4)
LeakyReLU	-	(32, H/4, W/4)	(32, H/4, W/4)
Flatten	-	(32, H/4, W/4)	(N_{flat})
Linear	output_dim=20	(N_{flat})	(20)
Tanh	-	(20)	(20)

1) Encoder Pretraining

The depth encoder (TinyAutoencoder) is pretrained separately from the RL policy to learn a compressed representation of depth images [1]. This pretraining phase enables the encoder to extract meaningful features from terrain depth maps before being integrated into the RL training pipeline.

Table VIII: Policy Network (MLP) Architecture

Layer	Type	Size
Input	Concatenation	State Dim
Hidden 1	Linear + LeakyReLU	128
Hidden 2	Linear + LeakyReLU	128
Hidden 3	Linear + LeakyReLU	128
Hidden 4	Linear + LeakyReLU	128
Output	Linear (Mean)	Action Dim (3)

Pretraining Process:

- **Training Data**: Diverse depth images collected from various terrain types (flat, Perlin noise, sinusoidal, stepped, etc.) to ensure robust feature learning across different terrain characteristics.
- **Loss Function**: Mean Squared Error (MSE) reconstruction loss between input depth images and decoded reconstructions:

$$\mathcal{L}_{\text{recon}} = \frac{1}{N}\sum_{i=1}^{N} \|x_i - \text{Decoder}(\text{Encoder}(x_i))\|^2 \quad (21)$$

  where x_i is the i-th depth image and N is the batch size.
- **Training Procedure**: The encoder-decoder pair is trained end-to-end using Adam optimizer with learning rate 10^{-3}, batch size 64, and validation split of 20%. Training continues until validation loss plateaus.
- **Output**: A pretrained encoder that compresses $H \times W$ depth images to a 20-dimensional feature vector, preserving essential terrain geometry information.

The pretrained encoder can be trained using:

```
python -m ballbot_rl.encoders.pretrain --data_path
    <path_to_depth_images> --save_path
    <encoder_output_path>
```

2) Frozen Encoder Setup

During RL training, the pretrained encoder weights are **frozen** (not updated) to preserve the learned feature representations [1]. This design choice offers several advantages:

- **Preserved Features**: The encoder maintains its ability to extract terrain-relevant features learned during pretraining, preventing catastrophic forgetting.
- **Faster Training**: Only the policy network parameters are updated, reducing computational cost and memory requirements.
- **Stability**: Fixed feature extraction provides a stable representation space for policy learning, improving convergence.

Implementation: The frozen encoder setup is implemented in the policy's feature extractor (`ballbot_rl/policies/mlp_policy.py`). After loading the pretrained encoder weights, all encoder parameters are explicitly set to non-trainable:

```
for param in encoder.parameters():
    param.requires_grad = False
```

This ensures that gradient updates during PPO training only affect the policy network (MLP layers) and value function, while the encoder remains fixed. The encoder path is specified in the training configuration file via the `frozen_cnn` parameter.

C. Hyperparameters

Table IX: PPO Training Hyperparameters

Parameter	Value
Algorithm	PPO (Stable-Baselines3)
Total Timesteps	10×10^6
Num Environments	10 (CPU limit)
Batch Size	256
n_steps	2048
n_epochs	5
Learning Rate	Adaptive (-1)
Gamma (γ)	0.99
GAE Lambda (λ)	0.95
Clip Range (ϵ)	0.015
Entropy Coeff	0.001
Value Fn Coeff	2.0
Target KL	0.3
Weight Decay	0.01

D. Reward Function Parameters

The reward function implementation follows Salehi (2025) [1] with three components computed at each timestep. The directional reward function (from `ballbot_gym/rewards/directional.py`) computes the base directional component, which is then scaled and combined with action penalty and survival bonus by the environment:

$$r_{\text{dir}} = \alpha_1 \cdot \frac{\mathbf{v}_{xy} \cdot \mathbf{g}_{\text{target}}}{100} \quad \text{where } \alpha_1 = 0.01 \quad (22)$$

$$r_{\text{action}} = -\alpha_2 \|\mathbf{a}\|^2 \quad \text{where } \alpha_2 = 0.0001 \quad (23)$$

$$r_{\text{surv}} = \begin{cases} \alpha_3 & \text{if } |\phi| \leq 20^\circ \text{ and } |\theta| \leq 20^\circ \\ 0 & \text{otherwise} \end{cases} \quad \text{where } \alpha_3 = 0.02 \quad (24)$$

These coefficients balance directional progress (r_{dir}), control effort minimization (r_{action}), and stability maintenance (r_{surv}). The survival bonus dominates early in training, encouraging the agent to learn balance before attempting navigation. The reward coefficients match those used in [1], ensuring fair comparison with reference results.

E. Comparison: Original vs. Enhanced Repository

Table X: Key Differences Between Original Salehi Repository and Enhanced Fork

Original Salehi Repository	Enhanced Fork
Single terrain type (Perlin noise)	13 terrain types (flat, perlin, spiral, stepped, etc.)
Hardcoded reward functions	Modular reward system with Component Registry
Limited documentation	50+ markdown files, 100+ pages
Monolithic architecture	Plugin-based Component Registry
Manual code modification for experiments	YAML-driven configuration
No type hints	Comprehensive type hints throughout

The enhanced repository maintains full compatibility with the original Salehi implementation[4] while adding extensibility

[4]Original repository: https://github.com/salehiac/OpenBallBot-RL

features. All original functionality is preserved, and new components can be added without modifying core code. The Component Registry pattern follows standard software design principles (registry and factory patterns), enabling the same benefits as plugin architectures in other domains: modularity, testability, and rapid experimentation.

F. Terrain Generation Details

Each terrain type supports configurable parameters enabling diverse training scenarios:

- **Perlin Noise**: Scale (25.0), octaves (4), persistence (0.2), amplitude (1.0)
- **Spiral**: Spiral tightness (0.1), height variation (0.5), direction (cw/ccw)
- **Sinusoidal**: Amplitude (0.5), frequency (0.1), direction (x/y/both)
- **Stepped**: Number of steps (5), step height (0.1)
- **And 9 more types**: See `ballbot_gym/terrain/` for complete list

All terrain generators output normalized height fields (values in [0, 1]) compatible with MuJoCo's heightfield geometry.