

Reinforcement Learning for Motion Control Using Unity and ML Agents

Raul Colón Rodríguez¹; Gael Yahir Vázquez Matos²; Erimar F Díaz Sierra³; Luis Miguel Traverso Aviles⁴

^{1,4} Universidad Ana G. Méndez, Recinto de Gurabo, Puerto Rico, USA, rcolon212@email.uagm.edu,
gvazquez203@email.uagm.edu, ediaz341@email.uagm.edu, traverso11@uagm.edu

Abstract— *The growing adoption of reinforcement learning in robotic motion control creates a need for trained engineers working at the intersection of robotics and machine learning. Even though this area is one that requires knowledge of advanced mathematical concepts such as policy optimization and deep neural network algorithms, its effective implementation depends on a system-level framework. Undergraduate students can study this abstraction inside a simulation and implement it using predefined coding libraries and tools. Two case studies are presented: a model of a racing car that learns to navigate three different racetracks and another model of a quadruped robot that learns to walk towards a target. The students develop the framework using a combination of preexisting elements such as physical environments and policy optimizers and encoding abstract concepts such as actions, observables, rewards, and training episode structures. Results show that undergraduate students can create models that exhibit emergent learning behavior by training policies.*

Keywords—reinforcement learning, robotic motion control, Unity ML-agents, project-based learning, system-level design.

I. INTRODUCTION

Reinforcement learning (RL) is a machine learning (ML) subfield that establishes computational methodologies for sequential decision making where an agent learns to perform specific actions [1]. It is a cornerstone of modern artificial intelligence (AI), particularly in robotics, controls and embodied intelligence [2], [3]. RL has also enabled real-world robotic motion control [3], [4]. The breadth and diversity of its applications has made complex challenges approachable and accessible [5]. Like many other tools inside ML, RL has been heavily implemented and its influence reached outside the graduate student level community [2]. There is inherently a traditional barrier for undergraduate (UG) students to adopt its use due to its heavy reliance on probability theory, optimization, Markov decision processes and gradient-based methods. Its advanced mathematics exceeds the preparation of many UG [6]. There is a pedagogical gap between how conceptually intuitive this field is and the fact that its instruction is typically algorithmic first [7]. There is limited earlier engagement, experimentation and conceptual fluency, which are also key for effective implementation.

In the methodology of RL an agent learns to execute specific actions through constant interaction with an environment with the objective to accumulate the maximum action rewards possible. By means of using sensors, the agent receives observables that approximate the state of his world which includes itself and the environment. Concepts such as state, action, reward, and policy are formalized by decision-theoretic frameworks that are rooted in Markov Decision Processes (MDPs)[1]. Besides RL involving an advanced

mathematics optimization layer, there is a system-level framework design layer. This system-level framework design layer is something that is accessible to undergraduate students. The historical abstraction of this framework emerges from dynamic programming and optimal control [8]. Also, interaction-based learning transitions from an insufficient fixed model-based solution to adaptive experience learning [9]. Subsequently techniques such as using function approximation [10] and deep neural networks [11] add to previous abstractions and strengthen the system-level framework as fundamental in RL design.

There is a separation of conceptual design and optimization mechanics. Optimization mechanics that involve selection and manipulation of policy gradients, value estimations, and optimization strategies can be handled by existing frameworks for which established code templates can be exploited [6]. Tools such as Pytorch, and Tensor flow already have preexisting libraries, commands that process and connect the input data in a seamless workflow. The system design layer, which includes the definition of the observables, action parameters, reward functions, and episode structure can be coded in conjunction with a virtual environment traditionally used for gaming, which UG students can manipulate, thus focusing RL on an applied engineering framework[7].

Simulation environment has an important role in developing scalable robotic DRL research by providing a system-design interface. From the most common platforms, Unity ML-Agents has shown to be an accessible and versatile framework that combines physics parameters, modifiable environments, and deep learning implementations [12]. ML-Agents is a Unity package that allows researchers that are creating a deep learning model to define and specify observation spaces, continuous action spaces, reward functions, and training structures while providing visual, graphical, and physical reference. This categorizes ML-Agents as a suitable tool to study motion, where the interaction between rigid bodies, joints, and physical objects is of great importance for behavioral learning.

This paper serves as a contribution by showing a case study of how students implemented agents in simulations of RL in robotics using Unity. For 2 cases (car in racetrack and quadruped robot) the students ran the entire framework using preexisting optimization pipelines and iterating/refining well-aligned observations, actions, reward functions and episode structure. This constitutes an authentic system-level RL practice and aligns with perspectives that view RL agents as components embedded in larger decision systems [13].

The educational significance of this study lies in students understanding system-level thinking and reasoning about

observations, actions, rewards and feedback loops. The behavior of agents emerges from the policy throughout its history as a response to the environment. The students also engage in a design activity that helps them grasp conceptual fluency before deep algorithmic theory. This early exposure may lower entry barriers and prepare them for later mathematical depth. This experience mirrors how RL is used in practice [1].

While previous work has already focused on improving algorithmic performance for continuous control [15P], this work is not a comparison of algorithms, and it does not propose a new optimization method. The performance of the agents is evaluated qualitatively by the emergence of behavior, and no benchmark superiority is studied. It is also limited to the study of a simulated environment rather than actual transfer to physical robotic systems. Instead, this paper demonstrates that using an accessible simulation environment to support motion control, the students can implement authentic system-level RL by designing the previously mentioned framework. The conceptual system design is separated from the optimization mechanics to enable accessible learning to undergraduate students that do not have a deep mathematical background.

II. METHODOLOGY

To develop a strong methodology to train a quadruped and race car agents through Unity software tools and packages, such as ML-Agents, previous fine-tuned projects were referenced and studied first. Some of these include the walker and crawler agents, which are part of the ML-Agents tutorial kit available through GitHub. A structured training process was developed using their scripts, virtual environments, and behavior parameters as examples of efficient reinforcement learning. The following methodology was performed by undergraduate students aiming to prove that the state of robotic control using RL is accessible to most student engineers without prior expertise of deep neural network, reinforcement learning, or modern robotics. The methodology is divided into 2 main sections. The 1st one is of general aspect and will encompass the setup common for all models. The 2nd section will establish considerations for the race car models and the quadruped robot import model.

A. General methodology for all models

Environment Setup - The first step toward training consists of the creation of a virtual environment setup where the agent interacts with physics, terrain, and task-specific conditions required for learning. To create a virtual environment, ML-Agents was first installed as a package in both the Unity project and the folder that contained all the Python-related packages, such as PyTorch and NumPy. The Python packages are necessary to provide communication between the Unity project and the command prompt, where the training commands are executed.

After installing ML-Agents, a 3D model of the agents was imported into the project. A Unified Robot Description Format (URDF) importer was installed and used to place the agent of a

quadruped robot in the environment. The other agents did not need this importer since their formats were already compatible with their respective Unity projects. Additionally, a game object was created through Unity tools to simulate the floor or ground on which the agent would interact.

Colliders, which are game object components that provide information to the agent regarding whether it is in contact with another object, were applied to all 3D models and interacting game objects. Depending on the object, components such as rigid bodies, articulation bodies, and configurable joints were placed under their hierarchy to provide essential physical information, such as mass, center of mass, moment of inertia, degrees of freedom, stiffness, and other properties that affect motion. By defining these parameters, the virtual environment was created thus inputting the agents and objects that would interact during training and providing the physical boundary conditions that would allow the training to extrapolate data from.

Policy Configuration - The next step involved defining the neural network architecture and reinforcement learning hyperparameters needed so the agents can learn. Some of these hyperparameters include trainer type, batch size, learning rate, and others. Most of the fine-tuning regarding the hyperparameters was focused on the learning rate, where different rates were tested to see if there was any change in the agent's training.

The trainer type was kept as a proximal policy optimization (PPO) type, where the neural network approximates an idealized function that maps the agent's observables to the best action. Hyperparameters are called from a single file used by the agent's scripts throughout the training procedure. In simplest terms, this step focused on structuring the behavioral parameters that would dominate the structure in which the agent model is trained.

The next step was identifying which observables were available to provide the agent with a sense of state recognition throughout the training iterations. It consisted of selecting and structuring the sensor inputs that the agent receives each step. These inputs included, but were not limited to, joint angles and positions, body orientation, velocities, angular velocities, height, and ground contact. These observables approximate the state of the agent within the environment and are inputs to the policies which generate actions needed to achieve the desired outcome. This step can be summarized as the section where the state-defining parameters were identified and provided to the agent.

The next step consisted of defining the action space. This space consists of what variables the agent could change to achieve the desired outcome. These changes are referred to as actions and simulate torque values sent to a motor. For most of the agents trained, joint drivers were applied to vary the angular position of each limb. Other options available were the force applied to the limb and the velocity of the limb. This step takes place during the script development for the agent.

The following step involved the creation of reward signals that would encourage the agent to achieve stable standing, efficient walking, speed control, and directional accuracy. Like the previous step, the assignment of a reward function takes place during script development. The quality of the reward function affects both the accuracy and the time required for the training to achieve the desired output.

These functions were reworked, redefined, and reapplied in various scenarios to test their effect on the agent's training. Simpler reward functions achieved simpler tasks in moderate time, while more complex rewards tended to achieve more difficult tasks, but with increased training steps. These functions considered parameters such as velocities, position, height, direction, orientation, stillness, uprightness, and other prioritized observables. Depending on the function, whether parabolic, exponential, additive, subtractive, multiplicative, or divisive, the agent would focus more effort on fine-tuning specific variables than others. In summary, this step directed the agent toward the desired outcome by providing well-structured functions that rewarded the agent when its actions brought it closer to that goal.

The training step consisted of running multiple reinforcement learning cycles using ML-Agents to optimize the policy and improve motion performance. When satisfactory models were achieved, additional training was done over them to optimize the model by applying new behaviors that the agent could learn. These sessions took about 1 to 2 days, depending on the agent and the desired outcome.

During training, the agent's action evolution was observed and analysed to determine which reward function was more effective and efficient. This step can be summarized as the procedure where different training structures were tested and used to train the agent to obtain a satisfactory model.

After the training sessions were completed, the resulting models were tested in inference mode to observe their output accuracy. This step involved assessing the motion quality, stability, and responsiveness of the trained policy. Using inference mode, the models were compared and evaluated side by side. The most satisfactory model would then proceed to be the base for the next models. This step can be summarized as the analysis of resulting models after training.

By using this methodology, different functioning models were developed for the quadruped robot, and car on simple racetracks, allowing each to successfully learn their respective movement tasks and exhibit motion behaviors that reflected the structure of their reward functions, observations, and action spaces. This process highlights how variations in environment setup, reward design, and training configurations influence the emergence of distinct and successful motion strategies. With the objective of implementing these virtual models to real-world mechanisms, this procedure facilitates the development of models that simulate physical attributes and could be translated to real mechanical actions by transferring the policy into a microcontroller capable of executing said actions.

B. Particular models: the race car and the quadruped robot

The initial straight line runner environment was designed to introduce the agent (simple car) to the most basic concept of autonomous driving: moving forward in a desired direction. The environment consisted of a small, rectangular area with a fixed goal direction pointing straight ahead. Fig. 1 shows the model and environment for the straight-line runner.

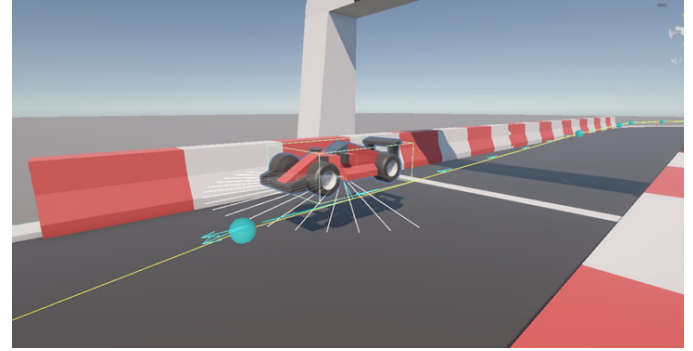


Fig. 1. Straight line simple car runner agent and environment.

The earliest reward function was intentionally minimal, providing positive reward for forward motion along the goal direction and a small penalty when the agent's orientation deviated from that direction. The base reward function was defined as

$$r = r_{\text{alive}} + k_p \langle \vec{v}, \hat{d} \rangle + k_s (-||\vec{v}|| - v_{\text{target}}|) - k_c |l| - k_h |\theta| + k_u s - k_e (a^2 + \omega^2), \quad (1)$$

where all the constant's k are the weights of the different rewards. Subtracted terms are penalties that diminish the rewards function. The dot product of the velocity vectors and the forwards distance maximized alignment of the race and the difference between the actual velocity and target velocity ensures a setpoint. Angle theta represents the heading error, s is the upright stability, and a and omega term are throttle and steering actions which are penalized to minimize jerk. Additional terms were added in the refining process such as immediate penalties for off-track detection increasing linearly with time. Ultimately episode ended when the agent spent enough time off track or after collisions with wall.

Transitioning to an oval track introduced a curved centerline and the requirement to complete laps. Forward progress was redefined as motion along the tangent of the closest point on the centerline. The reward formulation added penalties for lateral deviation and heading error:

$$r += k_p \max(0, \langle \vec{v}, \hat{t} \rangle) \Delta t - k_c \min(l_n, c_{\text{max}}) - k_h \theta_n \quad (2)$$

where unit vector t is the path tangent direction, l_n is a normalized lateral offset, and the angle theta is the normalized heading error. Stability and smoothness were improved by incorporating yaw rate and action penalties inherited from the straight-line runner. To improve robustness, the centerline penalty was clamped to prevent excessive negative rewards, and termination rules were relaxed to allow short recoveries. Wall collisions were handled non-terminally, with an initial penalty followed by a continuous penalty during contact.

The multi-turn circuit shown in Fig. 2 reused the reward structure developed for the oval track. The same shaping terms for progress, centerline alignment, heading error, stability, smoothness, and collision handling were retained. However, the increased curvature and complexity of the track made explicit track legality enforcement more critical. Semantic surface classification was used to distinguish between track and off-track regions. Continuous off-track penalties, immediate entry penalties, and time-based termination conditions were applied, while positive reward coefficients such as forward progress and alive bonus were slightly reduced to prevent cancellation of penalties.

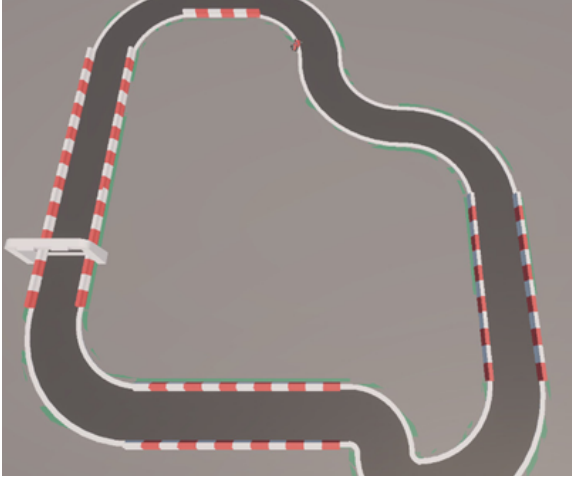


Fig. 2. Multi turn racetrack environment.

The quadruped robot had a more complex methodology. More degrees of freedom had to be incorporated into the RL framework design process. The downloaded URDF model of the quadruped robot shown in Fig. 3 consisted of eight actuated joints: four shoulder joints and four elbow joints. Each joint was controlled using a target position (desired angle) and assigned stiffness and damping parameters to approximate realistic servo motor behavior.

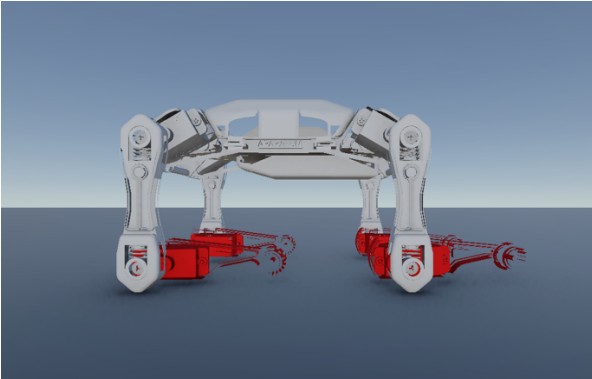


Fig. 3. Imported URDF model of quadruped robot.

Custom C# scripts were developed and attached to both the torso and legs of the robot providing torso and leg-specific state information to as observables. Once the physical model and joint limits were correctly configured, development efforts were primarily focused on refining the training script. Multiple

observation sets were tested to approximate the sensing capabilities of the real robot as closely as possible.

To facilitate successful training in simulation, additional virtual sensors were introduced to promote faster and more stable learning. Initially, observations were limited to simulated inertial measurement unit (IMU) data, including torso orientation along the x-, y-, and z-axes. Absolute position information was intentionally excluded, as it would not be available on a physical robot without external sensing systems. The torso's height relative to the ground was included to identify standing states, along with upright alignment metrics and relative linear velocity.

Since stable learning could not be achieved using this limited observation set alone, additional observations were introduced, including foot-ground contact states and joint position values. These additions significantly improved the agent's ability to infer its physical state and resulted in more stable learning behavior.

Within the training script, the initial reward structure was designed to encourage stable standing and locomotion behavior. Rewards emphasized maintaining an upright torso orientation and sufficient torso height while penalizing excessive velocity. In later training stages, the reward function was modified to promote walking behavior. In this case, forward linear velocity was rewarded, while constraints on posture and stability were retained. The locomotion reward equation is as follows:

$$r_t = w_s \left(1 - \text{clip}_{[0,1]} \left(\frac{||v_t| - v_{\text{ref}}|}{v_{\text{ref}}} \right) \right)^2 \text{sign}(v_t) + w_{\text{sm}} \text{clip}_{[0,1]} \left(\frac{1}{1 + \frac{|v_t - v_{t-1}|}{\Delta t}} \right) + w_u \text{clip}_{[0,1]} \left(\frac{u_t + 1}{2} \right) + w_h \left(1 - \text{clip}_{[0,1]} \left(\frac{|h_t - h_{\text{ref}}|}{h_{\text{ref}}} \right) \right)^2 + w_c c_t + a - 10|h_t - h_{\text{ref}}| - 5(1 - |u_t|) + \mathbf{1}_{\{v_t < 0\}} w_b v_t \quad (3)$$

In this case, the equation consists of 9 terms, 6 rewards and 3 penalties. Rewards are given for forward speed matching, motion smoothness, upright orientation, height regulation, ground contact consistency, and an alive bonus. Penalties are given to discourage height deviation, body tilt, and backward motion.

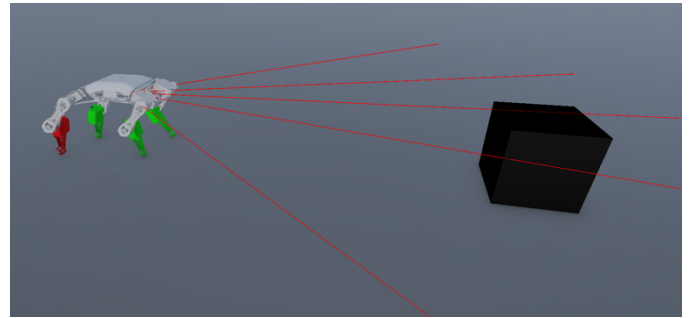


Fig. 4 Ray tracing vectors used by the model to detect a target object.

A 2nd model shown in Fig. 4 used ray tracing vectors to support the additional task objectives of reaching a cube target by walking towards the correct direction. Ray tracing can be used to mimic the behavior of an ultrasonic sensor, allowing the agent to estimate the position of the target. This information was then incorporated into the agent's decision-making process

through additional reward functions derived from the ray cast data. The following equation shows the reward function structure that enabled the quadruped to reach to a target that changed on each episode.

$$r_t = r_t^{\text{loc}} + r_t^{\text{ray}} \quad (4)$$

Equation “(4)” includes the locomotion term already described in equation “(3)” plus a ray cast term that is defined as follows.

$$r_t^{\text{ray}} = 3 R_{\text{prox}} + 2 R_{\text{center}} \quad (5)$$

Equation “(5)” incorporates a proximity term and a centering term. Numbers 3 and 2 are refined weights indicating that more importance is slightly given to getting closer to the cube than to centering. These terms are calculated taking into consideration 5 rays, for which a hit status and distance is determine using the ray tracing tools of Unity.

III. RESULTS

The results presented in this section focus on observed behavior and learning dynamics that emerges during training. For the straight-line runner, the minimal reward formulation successfully encouraged forward motion but led to unstable behaviors such as jerky steering and occasional spins. Introducing yaw rate and action penalties significantly improved stability, producing smoother trajectories while preserving forward progress. Semantic off-track penalties and strict wall collision handling ensured that illegal behavior was consistently disadvantageous. The agent learned to remain within the valid region and maintain smooth control. Fig. 5 shows the rewards values VS the steps during training.

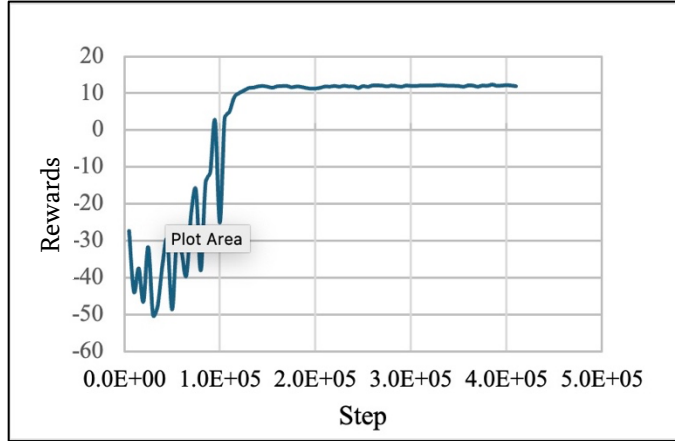


Fig. 5 Reward values for the straight-line race car.

For the oval runner, the baseline oval reward allowed continuous motion around the track but exhibited instability in tight corners. Stability and smoothness penalties reduced oversteering and improved consistency. Clamping the centerline penalty and relaxing termination conditions enabled better recovery from disturbances. Non-terminal wall penalties proved effective, allowing the agent to correct its behavior rather than immediately failing. The lap completion bonus aligned local shaping with the global task objective, resulting in sustained, coherent lap completion. Fig. 6 shows the rewards

values VS the steps during training for the race car in the oval track.

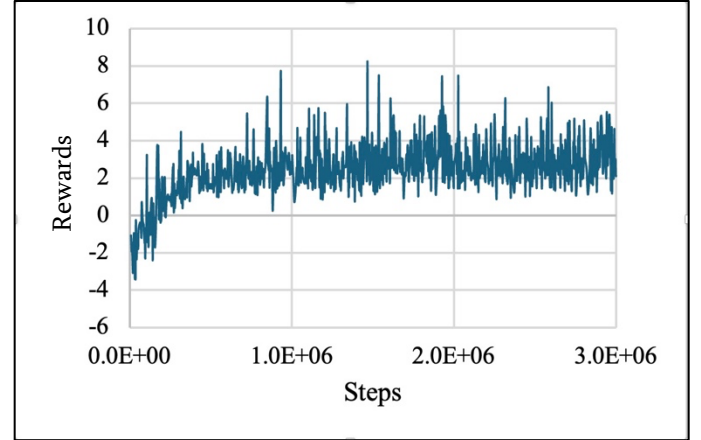


Fig. 6 Reward values for the race car in oval track.

Applying the oval-trained reward function directly to a multi-turn circuit revealed that centerline-based rewards alone were insufficient to enforce track legality. The introduction of semantic off-track detection resolved this issue. Continuous penalties, entry penalties, and terminal conditions ensured that off-track behavior was always net negative. The final reward formulation balanced guidance and constraint, promoting smooth, fast, and stable driving while strongly discouraging illegal behavior. This enabled robust learning on complex multi-turn circuits rather than only simple oval tracks. Fig. 7 shows the rewards values VS the steps during training and Fig. 8 shows images of the car learned performance on curves.

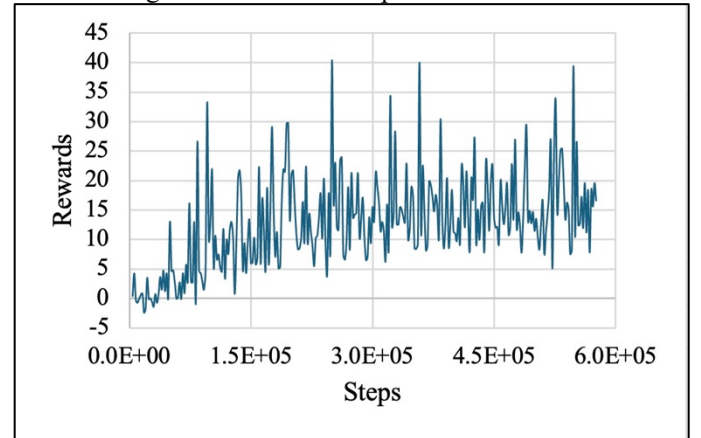


Fig. 7 Reward values for the race car in multi-turn ciurcuit.

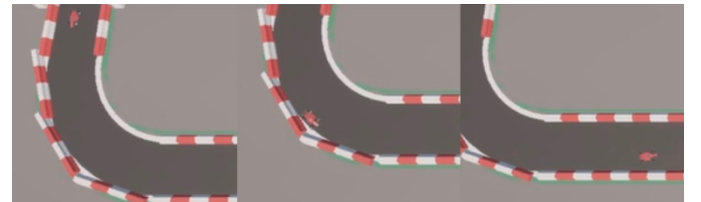


Fig. 8 Racing car turning on corners on the multi-turn circuit.

For the the quadruped agent a simple successful model was able to walk. Multiple versions of the training script were tested to narrow down the most effective training logic. The observations

used during training were limited to those that realistically represent the agent's physical capabilities and operating conditions. Variables such as forward speed, height, and uprightness were included, as these parameters could later be related to real sensors on a physical agent. Additional variables such as the agent's foot contact states were included, allowing the agent to detect which feet, and how many of them, were in contact with the ground. Although these observations are not currently obtainable from the real-world agent, they proved to be quite helpful during training to obtain the preferred motion. From the latest successful training attempt, as seen in Fig. 9, at the beginning of the chart, there is a rise before a significant drop in the reward value. The rise in the beginning is due to the agents' starting position. At the beginning of the episode the quadruped is in a kneeling position and acquires rewards for lifting itself up even when not walking. Once the agent gets up and walk, it loses balance and falls which results in penalties which causes the sudden drop in rewards. After many steps the agent begins to acquire balance and gains a subsequent steady rise in reward values in which the walking behavior emerges.

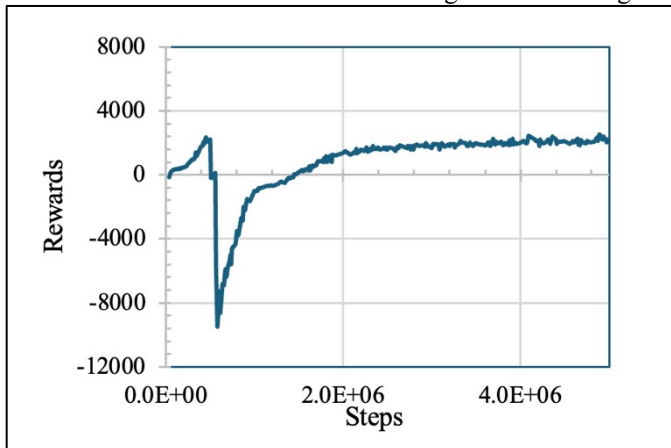


Fig. 9 Reward values for walking quadruped.

Additional attempts were made to train the agent to follow a designated object, switching the goal of simply achieving stable walking. To do this ray casting was implemented to perceive the object while learning to walk. Fig. 10 shows rewards graph for quadruped using ray casts for an environment with target and one without the target. During the walking learning phase both obtain similar rewards. A sudden jump in the reward graph for the model with the cube target shows when the quadruped learns the behavior of chasing the target. Fig. 11 shows the quadruped walking towards the cube target and reaching it. This success was consistent for every episode after the behavior was learned.

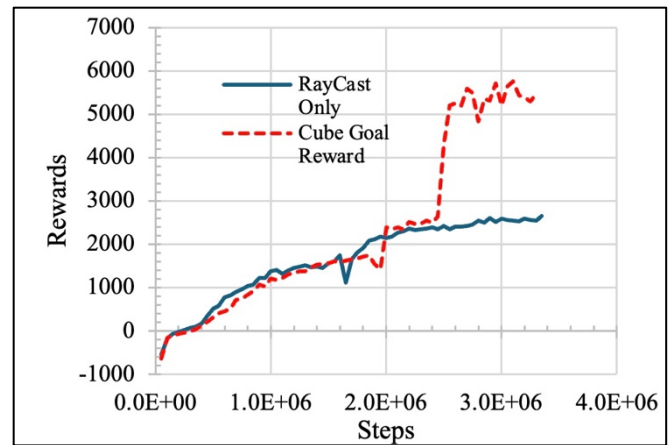


Fig. 10 Reward values for walking quadruped with ray cast with and without cube target.

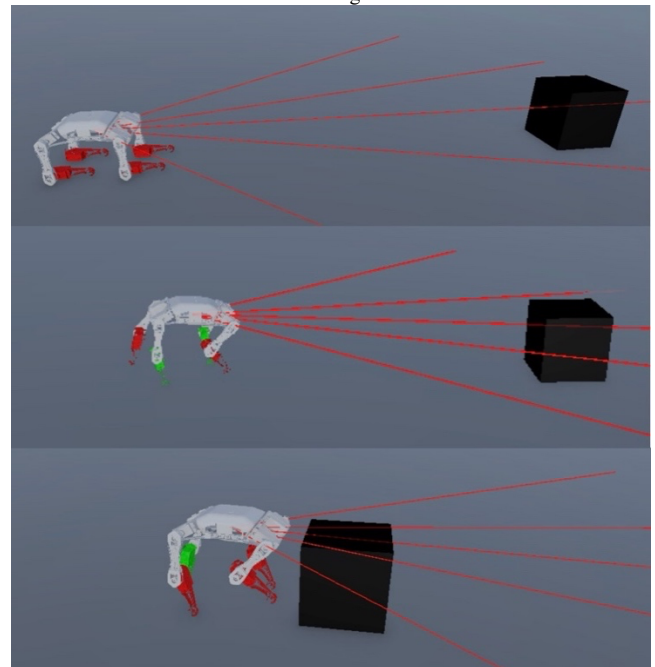


Fig. 11 Quadruped chasing and arriving to the target using ray cast.

IV. CONCLUSION

This work demonstrated that undergraduate students can implement complete reinforcement learning pipelines for robotic motion control by focusing on system-level design elements rather than algorithmic derivations. Across multiple simulated tasks, agents exhibited stable and task-aligned emergent behaviors, supported by qualitative observations and reward evolution during training. These results suggest that introducing reinforcement learning through system-level abstraction enables early conceptual fluency and provides students with experience that closely mirrors practical RL workflows used in robotics. While the study is limited to simulation-based environments, future work may explore policy transfer to physical robotic platforms, expanded task diversity, and longitudinal assessment of learning outcomes.

REFERENCES

- [1] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA, USA: The MIT Press, 2018.
- [2] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," IEEE Signal Process. Mag., vol. 34, no. 6, pp. 26–38, 2017.
- [3] C. Tang, B. Abbatematteo, J. Hu, R. Chandra, R. Martín-Martín and P. Stone, "Deep reinforcement learning for robotics: A survey of real-world successes," *Annu. Rev. Control, Robotics, and Auton. Syst.*, vol. 8, pp. 153–188, May 2025, doi:10.1146/annurev-control-030323-022510.
- [4] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," J. Mach. Learn. Res., vol. 17, no. 39, pp. 1–40, 2016.
- [5] M. Ghasemi, A. H. Moosavi and D. Ebrahimi, "A comprehensive survey of reinforcement learning: From algorithms to practical challenges," arXiv:2411.18892, Nov. 2024.
- [6] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Goyal and T. Hester, "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis," *Machine Learning*, vol. 110, no. 13, pp. 2419–2468, 2021. doi:10.1007/s10994-021-05961-4
- [7] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor and S. Levine, "How to train your robot with deep reinforcement learning: Lessons we have learned," *Int. J. Robot. Res.*, vol. 40, no. 4–5, pp. 698–721, Apr. 2021.
- [8] R. E. Bellman, Dynamic Programming. Princeton, NJ, USA: Princeton University Press, 1957.
- [9] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," IEEE Trans. Syst., Man, Cybern., vol. SMC-13, no. 5, pp. 834–846, Sep./Oct. 1983.
- [10] J. N. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," IEEE Trans. Autom. Control, vol. 42, no. 5, pp. 674–690, May 1997.
- [11] Y. Li, "Deep reinforcement learning: An overview," arXiv:1701.07274, 2017.
- [12] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar and D. Lange, "Unity: A general platform for intelligent agents," arXiv:1809.02627, Sep. 2018. S.
- [13] Koenig and R. G. Simmons, "Complexity Analysis of Real-Time Reinforcement Learning," in Proc. 11th Nat. Conf. Artif. Intell. (AAAI), Washington, DC, USA, 1993, pp. 99–105.
- [14] A. L. Kolodner, C. Hmelo, H. Narayanan, D. Guzdial, J. L. Kolodner, and L. Edelson, "Problem-based learning meets case-based reasoning in the middle-school science classroom: Putting learning by design into practice," Educ. Technol. Res. Dev., vol. 51, no. 1, pp. 1–19, 2003.
- [15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," arXiv:1509.02971, Sep. 2015.