QuadMove

Project Report

Gargi Gupta
Advait Desai

September 2025

Acknowledgements

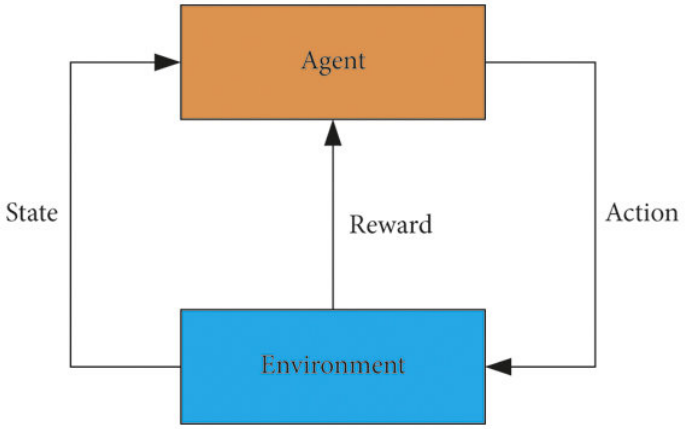
We would like to express our sincere gratitude to our mentors, Ansh Semwal and Prajwal Avhad, whose invaluable advice and assistance were crucial to the successful development of our reinforcement learning-based quadruped, QuadMove. We also thank the Society of Robotics and Automation, VJTI, whose resources and guidance enabled this project.

1 Overview

In our project QuadMove, we used a Proximal Policy Optimization (PPO)-based gait policy to train Go2, one of Unitree's quadruped, to walk in simulation. In order to accomplish this goal, we studied a variety of algorithms and their intricate implementations, laying the groundwork for Reinforcement Learning (RL) from the ground up. This report describes our experience, approach, and the main takeaways from creating a quadrupedal robot's locomotion policy from the ground up.

2 Foundations of Reinforcement Learning

2.1 Theoretical Backbone



We learned the basics of Reinforcement Learning from David Silver's lectures.

Markov Reward Process
Defined by the tuple

$$(S, P, R, \gamma)$$

where:

- S: set of states,
- P(s'—s): transition probabilities,
- R(s): expected reward in state s,
- γ: discount factor.

The value function gives the expected return starting from state s:

$$V(s) = \mathbb{E}[G_t \mid s_t = s]$$

Markov Decision Process extends the MRP by introducing action (A) in the tuple

$$(S, A, P, R, \gamma)$$

where:

- P(s'—s, a): transition probability given state s and action
- R(s, a): expected reward in state s.

Bellman Expectation Equation
It splits the return into two parts:

- Immediate reward you get right away.
- Discounted value of the next state, averaged over all possible actions and transitions.

This recursive form makes it possible to compute state values without explicitly simulating the entire future.

$$V^{\pi}(s) = \mathbb{E}_{\pi}\big[R_{t+1} + \gamma V^{\pi}(S_{t+1}) \mid S_t = s\big]$$

Bellman Optimality Equation
Instead of averaging the actions dictated by a policy π, we now choose the best action at each step. So, the expectation over π is replaced with a maximum over all actions a.

$$V^{*}(s) = \max_{a} \mathbb{E}\big[R_{t+1} + \gamma V^{*}(S_{t+1}) \mid S_t = s, A_t = a\big]$$

Exploration v/s Exploitation
While the Bellman Optimality Equation defines what the optimal policy looks like, in practice an agent must still learn this policy through trial and error. This gives rise to the **exploration–exploitation dilemma**:

- **Exploitation** means choosing the action that currently seems best, based on past experience.
- **Exploration** means trying out actions that may not look optimal now but could reveal better long-term strategies.

A common approach to balance the two is the ϵ-greedy strategy, where the agent chooses a random action with probability ϵ (exploration) and the best-known action with probability $1 - \epsilon$ (exploitation). Over time, reducing ϵ allows the agent to focus more on exploitation once it has gathered enough knowledge of the environment.

2.2 Classical Methods

Monte Carlo
These methods estimate value functions by averaging returns over complete episodes. Because rewards are only observed at the end, the agent learns from entire trajectories rather than single steps. Monte Carlo is an on-policy method, as it evaluates and improves the same stochastic policy used to generate experience.

Q Learning
Q-learning is a value-based method that updates the action-value function $Q(s, a)$ at each time step, so learning does not require an entire episode to finish. The update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big]$$

Here, α is the learning rate, and the term inside the brackets is the temporal-difference error. Unlike Monte Carlo, Q-learning is an **off-policy** method, since it learns the value of the greedy target policy while potentially following a different behavior policy to gather data.

2.3 Implementations

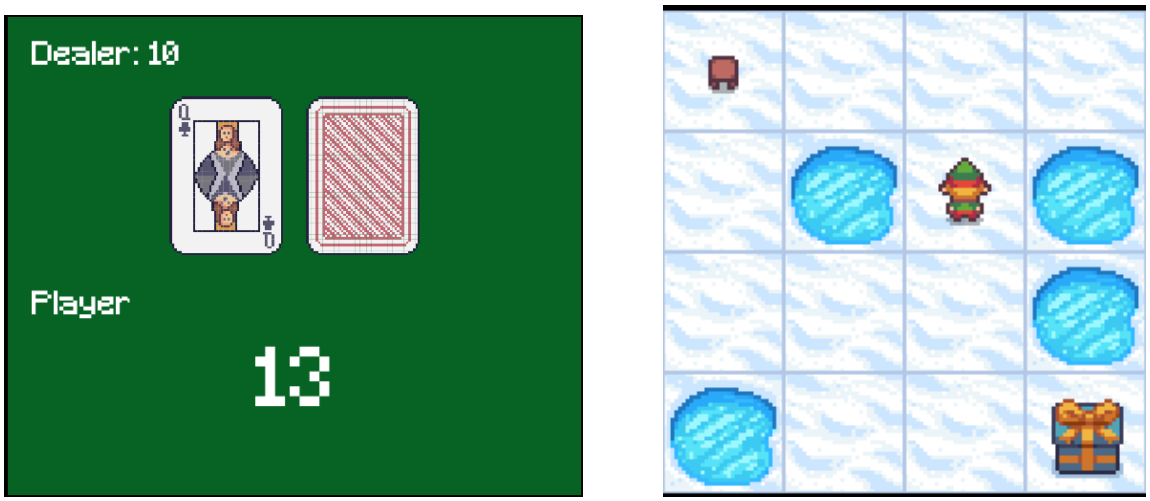


Figure 1: Blackjack (left) and Frozen Lake (right).

Blackjack

Blackjack is a simple card game setup where the agent learns when to hit or stick to maximize rewards. Since the result is only known at the end of a round, it fits nicely with Monte Carlo methods, which learn from complete episodes.

Frozen Lake

Frozen Lake is a grid-based navigation task where the agent has to reach the goal without falling into holes. The twist is that the environment is slippery, so actions do not always lead to the expected outcome. This randomness makes it a neat testbed for Monte Carlo, as the agent needs many episodes to average out the uncertainty.

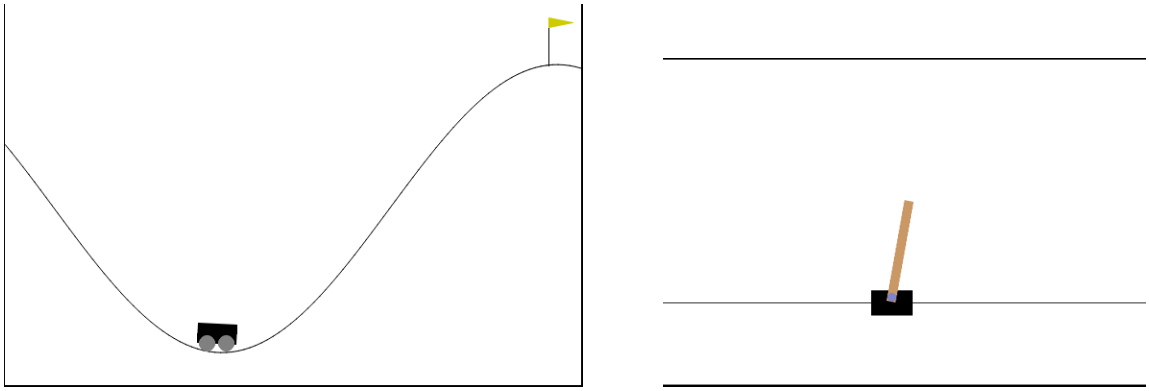


Figure 2: Mountain Car (left) and Cart Pole (right).

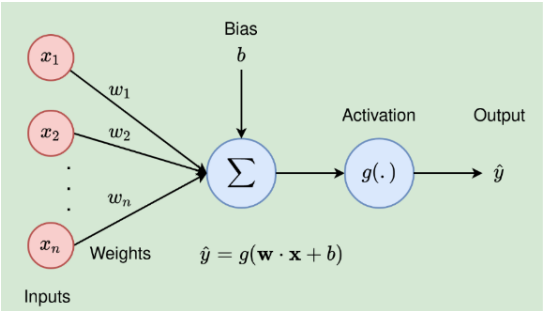
Mountain Car

The state of the car (position and velocity) is continuous, so we discretize it into bins to build a finite Q-table. Q-learning then updates the action values step by step using the TD error, instead of waiting until the episode ends. This incremental learning helps the agent figure out the non-obvious strategy -backing up first to build momentum before accelerating forward -even though rewards are sparse.

Cart Pole

Cart position/velocity and pole angle/angular velocity are also continuous, so we divide them into discrete ranges to form table indices. With Q-learning's per-step updates, the agent quickly adjusts its action values after each small correction. The combination of discretization and incremental TD updates allows the agent to learn a stable policy to keep the pole balanced.

3 Neural Networks



3.1 Components

A neural network is made up of simple processing units called perceptrons. Each unit takes inputs, multiplies them by weights, adds a bias, and passes the result through an activation function.

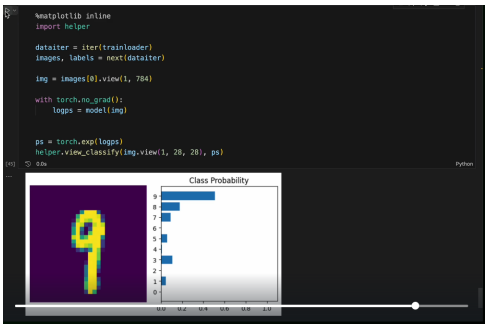
$$y = g\left(\mathbf{w} \cdot \mathbf{x} + b\right)$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector, b is the bias term, and $g(\cdot)$ is the activation function. When perceptrons are arranged in layers, the input layer passes data into one or more hidden layers, and finally to the output layer. The hidden layers allow the network to learn intermediate patterns that improve prediction. The working process involves two main steps:

- **Forward propagation:** inputs are passed through the network to produce an output.
- **Backward propagation:** the error is calculated and used to adjust weights and biases through gradient descent.

This process is repeated during training until the network achieves the desired performance.

3.2 Implementation



We worked with the MNIST dataset, which is a large collection of handwritten digits from 0 to 9. Each image is 28×28 pixels in grayscale. Before training, the images were normalized and split into training and testing sets. The idea was to teach a simple neural network to recognize which digit is in each image. In this way, we could check how well the model learns patterns in handwritten numbers.

4 Deep Reinforcement Learning Algorithms

4.1 Deep Q-Network

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

DQN extends Q-learning by using a neural network to approximate the action-value function, making it suitable for environments with large or continuous state spaces. It introduces several improvements for stability and efficiency:

- **Function approximation**: A deep neural network is used instead of a Q-table to estimate $Q(s, a)$.
- **Experience replay**: Transitions (s, a, r, s') are stored in a replay buffer and sampled randomly to break correlations between consecutive experiences.
- **Target network**: A separate target network is maintained to compute stable target values, which is periodically updated from the main network.
- **Training**: The network parameters are updated by minimizing the squared error between the predicted Q-values and the target values using stochastic gradient descent.

A limitation of DQN is that it tends to overestimate action values, since the same network is used both to select and evaluate actions when computing target values. This overestimation can lead to unstable or suboptimal policies.

4.2 Double Deep Q Network

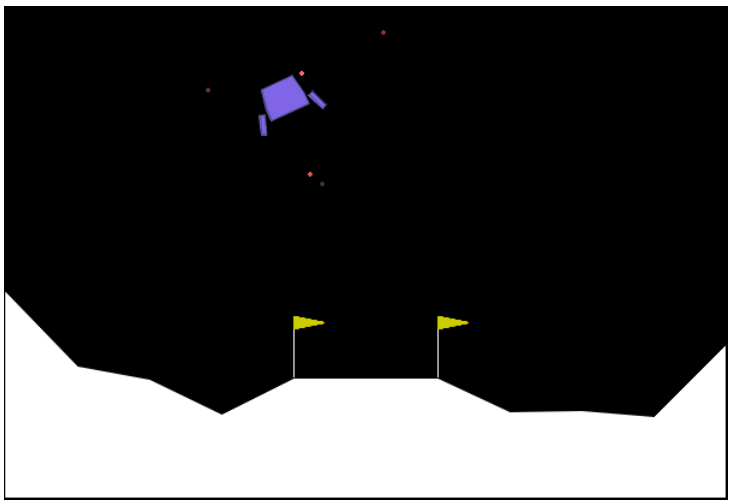
Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

- The **online network** decides which action to take.
- The **target network** judges the value of that chosen action.

By separating the "chooser" and the "evaluator," DDQN keeps the Q-values more accurate and the learning process more stable.

4.3 Implementation



The Lunar Lander environment presents a challenging control problem, as the agent must manage continuous thrusts and precise positioning under the influence of gravity. In this setting, DQN is capable of learning a policy but is prone to instability due to overestimation of action values. DDQN mitigates this issue by separating action selection and evaluation during updates, thereby reducing overestimation bias. Empirically, this results in more stable learning and improved landing performance compared to the standard DQN.

5 Policy Gradient Methods

5.1 Vanilla Policy Gradients and Surrogate Objectives

Vanilla policy gradient, or REINFORCE, adjusts the policy parameters so that rewarding actions become more likely. The objective is

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_t \log \pi_\theta(a_t|s_t) R_t\right]$$

with gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) R_t\right].$$

This gradient is estimated from sampled trajectories and backpropagated through the policy network to update its weights. While simple, vanilla policy gradients suffer from high variance and slow convergence.

To improve stability, surrogate objectives are used. Instead of directly maximizing returns, the update is based on the probability ratio between the new and old policies:

$$L(\theta) = \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t\right]$$

Here, π_θ is the updated policy, $\pi_{\theta_{\text{old}}}$ is the previous policy, and A_t is the advantage. Maximizing this surrogate objective improves performance while keeping the new policy close to the old one, forming the basis of TRPO and PPO.

5.2 Trust Region Policy Optimization

- Pseudocode:

  for iteration=$1, 2, \ldots$ **do**

  Run policy for T timesteps or N trajectories

  Estimate advantage function at all timesteps

$$\begin{aligned} &\underset{\theta}{\text{maximize}} \sum_{n=1}^{N} \frac{\pi_\theta(a_n \mid s_n)}{\pi_{\theta_{\text{old}}}(a_n \mid s_n)} \hat{A}_n \\ &\text{subject to} \quad \overline{\text{KL}}_{\pi_{\theta_{\text{old}}}}(\pi_\theta) \le \delta \end{aligned}$$

  end for

Surrogate objectives do not control update sizes, hence large policy updates can cause performance to collapse. TRPO avoids this by limiting how far the new policy can move from the old one.

The method maximizes the surrogate objective but adds a trust region constraint based on the Kullback-Leibler (KL) divergence.

$$\max_\theta \; \mathbb{E}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t\right] \quad \text{subject to} \quad \mathbb{E}_t\Big[D_{\text{KL}}(\pi_{\theta_{\text{old}}} \,||\, \pi_\theta)\Big] \le \delta$$

Here, D_{KL} measures the difference between old and new policies, and δ is a small constant that limits the step size.

This keeps policy updates stable and prevents sudden drops in performance. The update step is found using conjugate gradient methods.

Downsides: TRPO needs second-order optimization (involving Hessian-vector products), which is harder and more expensive than first-order methods. It can also be slow and harder to scale to big neural networks.

5.3 Proximal Policy Optimization

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

PPO simplifies TRPO by avoiding second-order optimization while still controlling the size of policy updates. Instead of enforcing a hard KL constraint, PPO modifies the surrogate objective using a clipping term. This prevents the policy from moving too far in a single update.

The clipped objective is:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)A_t,\ \text{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)A_t\right)\right]$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}.$$

The clipping ensures that the probability ratio $r_t(\theta)$ does not move too far from 1, which keeps updates stable while still encouraging improvement.
PPO is easier to implement, uses only first-order optimization, and scales well with large neural networks. However, it does not always guarantee monotonic improvement.

5.4 Implementation

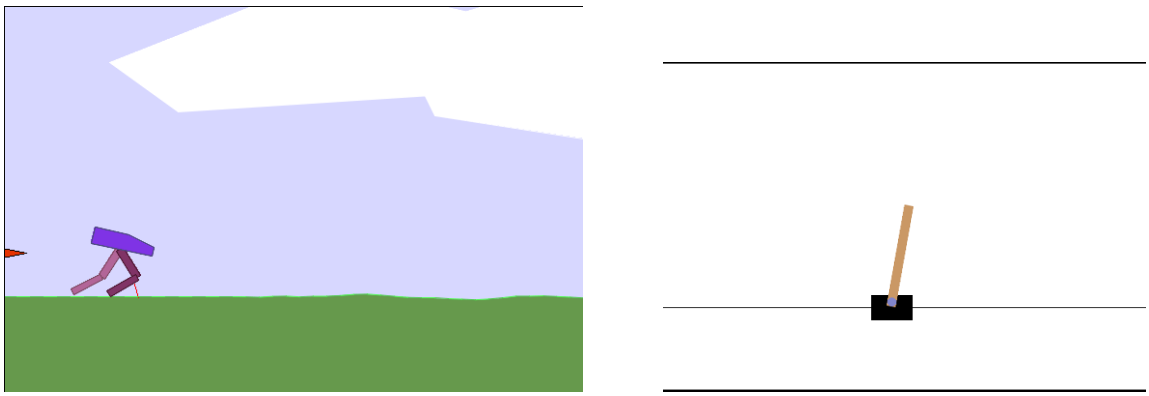


Figure 3: Bipedal Walker (left) and Cart Pole (right).

The **Bipedal Walker-v3** environment involves the agent learning to coordinate continuous joint torques to walk forward without falling. The state includes joint angles, velocities, and contact sensors, while the action space is continuous, representing the torques applied to the four leg motors. The clipped surrogate objective of PPO stabilizes updates in high-dimensional continuous action spaces, making it effective for locomotion problems. Unlike TRPO, it avoids expensive second-order methods, allowing faster training while still preventing large destructive policy shifts.

6 Simulating in MuJoCo

6.1 XMLs / URDFs

Robots in MuJoCo are described through **XML files**, which act as blueprints for the simulator. These files specify the robot's physical structure: body links, joints, collision geometry, actuators, and sensor definitions. Sometimes these XMLs are directly converted from a **URDF** (Unified Robot Description Format), which is a standard used in robotics.

For the Go2 quadruped, the XML defines each leg segment, the hinge joints at the hips and knees, and the torque limits for the motors. This ensures the simulated robot behaves within realistic physical constraints.

The **action space** is continuous, with values in the range $[-1, 1]$. The policy outputs actions in this normalized range, which are then rescaled to the torque limits defined in the XML. For example, an action of $+1$ means maximum positive torque on a joint, while -1 corresponds to maximum torque in the opposite direction. This keeps the agent's learning process clean, while still allowing it to control the robot effectively.

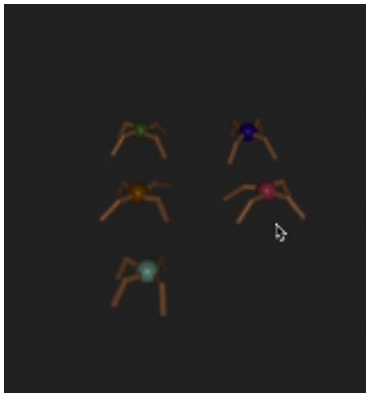
The **observation space** acts as the robot's senses. It includes joint angles and velocities, the body's orientation (roll, pitch, yaw), and its linear and angular velocities. At every simulation step, the action is applied, MuJoCo advances the physics, and the updated state is returned to the policy. This cycle of acting, simulating, and observing is what trains the quadruped on our policy.

6.2 Our First Tryst with MuJoCo

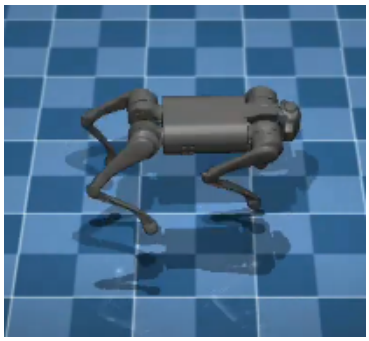
Before working on the Go2 quadruped, we wanted to get comfortable with MuJoCo. These first experiments were more about curiosity than results — just testing the simulator and observing how different robots moved.

We began with the classic **Ant** environment that comes with MuJoCo, mainly to see how multi-legged agents are modeled and controlled. After that, we tried the **Go1**, since it is structurally close to the Go2 and gave us a feel for joint configurations and contact dynamics in quadrupeds. Out of curiosity, we also loaded the **fruit fly** model from the MuJoCo Menagerie. This was more of a side experiment, but it showed us how flexible the simulator can be — from quadrupeds all the way down to tiny biologically inspired agents.

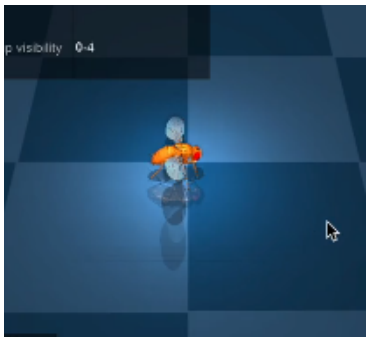
While these early runs did not directly contribute to training the Go2, they gave us valuable hands-on experience with MuJoCo's XML system, the action and observation spaces, and the quirks of different morphologies.



(a) Ant environment



(b) Go1 robot



(c) Flying agent

Figure 4: Different simulated environments and agents.

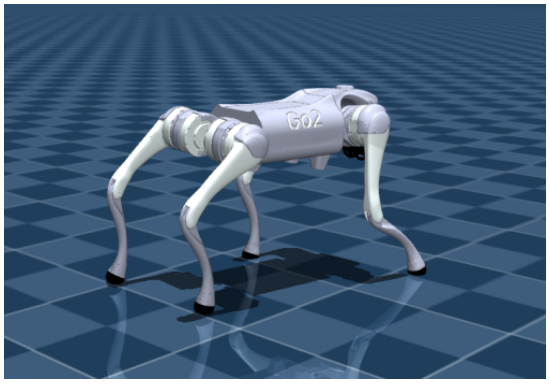
7 PPO on Go2

7.1 Objective

The Go2 quadruped was simulated in MuJoCo with the goal of learning a stable, dog-like walking gait. Most of the locomotion resources on the Web have used GPU-based simulators such as Isaac Gym, but we chose MuJoCo for its accurate contact dynamics and efficiency on CPU. Our objective was to achieve smooth, balanced walking gait.

7.2 Physics of Walking

Walking is basically controlled falling once the body is balanced. The robot leans forward, and the legs catch and push it into the next step. To stay stable, the center of mass has to stay above the legs on the ground, while roll and pitch are kept in check. Taking longer steps gives more speed but also makes it easier to lose balance.
For a quadruped like Go2, at least two legs remain in contact with the ground while others swing forward, allowing steady forward motion.



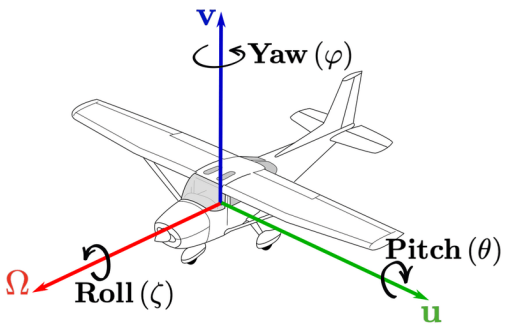


Figure 5: Go2 simulation (left) and orientation visualization (right).

7.3 Reward Function Design

The reward function was designed keeping all the above facts in mind.

- **Forward velocity:** Rewards speed near a target velocity.
- **Height maintenance:** Penalizes large deviations from desired body height.
- **Posture control:** Strong penalty on roll and pitch to prevent falling or nose-diving.
- **Joint penalty:** Discourages extreme joint angles while allowing flexibility.
- **Smoothness:** Encourages gradual changes between actions.
- **Control cost:** Small penalty on large torque commands.
- **Lateral stability:** Reduces side-to-side drift.
- **Step length:** Rewards longer and more efficient steps.
- **Leg spread:** Prevents the rear hips from collapsing inward.
- **Alive bonus:** Constant reward for staying upright.

The final reward was a weighted sum of these terms, tuned to emphasize forward velocity and posture control. This helped the agent learn a gait that was both stable and dynamic.

If we pushed the reward too much toward speed, the robot ran, but kept tripping. If we focused too much on posture, it walked too carefully, leading to almost no forward motion. We had to find a middle ground that gave both balance and decent pace.

8 Challenges & Solutions

Training Time
At first, our training loop was not very efficient. Running just 1000 iterations could take nearly 3 hours, which slowed progress and made it difficult to experiment with different reward functions or hyperparameter settings. This was a major bottleneck because reinforcement learning is heavily based on iteration and trial-and-error. To overcome this, we shifted to using Stable Baselines3 (SB3), which came with a highly optimized PPO implementation. With SB3, we were able to train up to 1.5 million iterations in under 20 minutes. This speedup allowed us to test many reward structures and hyperparameter combinations in a reasonable timeframe.

Once we identified what worked best, we translated those learnings back into our self-written PPO code. We tuned the PPO parameters more carefully and fixed performance issues. While choosing the scene.xml, we had to choose between friction and its absence. Without friction the agent could learn faster, but the gait was unrealistic and unstable. Adding friction made training harder but produced more natural and stable walking, so we ultimately went with it.

Reward Function Tuning
As mentioned earlier, balancing the different reward terms was a delicate task, so we ended up experimenting with a wide range of variations. In addition to the terms we finally settled on, we also tried:

- Rewarding whenever two or more feet stayed on the ground,
- Giving extra reward if diagonal legs were the ones in contact,
- Penalizing feet crossing over each other,
- Penalizing unnecessary swinging of the legs.

None of these experiments gave the kind of gait we were aiming for, but they helped us understand reward engineering much better. They also taught us how sensitive the Go2's action space can be to even small changes. In the end, we found that sticking to a relatively simple, "vanilla" reward structure gave the most stable results.

9 Conclusion

This project started with the basics of reinforcement learning and slowly built up through smaller environments like Blackjack, Lunar Lander, and Bipedal Walker. All this helped us get a better grip on how policies are trained and how different action spaces are handled. The main goal was getting the Go2 quadruped to walk in MuJoCo using PPO. Here the real challenge turned out to be the reward function. Forward velocity, posture, stability, control costs - all had to be balanced carefully, and even small tweaks could completely change the gait.

References

References

[1] Gymnasium. *Loading the Quadruped Model Tutorial*. https://gymnasium.farama.org/tutorials/gymnasium_basics/load_quadruped_model/

[2] YouTube. *Reinforcement Learning Video Tutorial Playlist*. https://youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ

[3] Barto, A. G., & Sutton, R. S. *Reinforcement Learning: An Introduction*. https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf

[4] Udacity. *Deep Learning with PyTorch*. https://www.udacity.com/course/deep-learning-pytorch--ud188

[5] YouTube. *Reinforcement Learning Deep Dive Playlist*. https://www.youtube.com/playlist?list=PLwRJQ4m4UJjNymuBM9RdmB3Z9N5-0IlY0

[6] Hugging Face. *Deep Reinforcement Learning Course — Unit 4*. https://huggingface.co/learn/deep-rl-course/en/unit4/introduction

[7] Sarrocco, F. (2023). *Making Quadrupeds Learn to Walk*. https://federicosarrocco.com/blog/Making-Quadrupeds-Learning-To-Walk

[8] Aractingi, M., Léziart, P.-A., Flayols, T., Perez, J., Silander, T., Souères, P., et al. (2023). Controlling the Solo12 quadruped robot with deep reinforcement learning. *Scientific Reports*, 13, 11945. https://www.nature.com/articles/s41598-023-38259-7

[9] Cheng, Y., Liu, H., Pan, G., Ye, L., Liu, H., & Liang, B. (2024). Quadruped robot traversing 3D complex environments with limited perception. *arXiv preprint*. https://arxiv.org/html/2404.18225v1

[10] Unitree Robotics. *unitree_rl_gym*. GitHub repository. https://github.com/unitreerobotics/unitree_rl_gym

[11] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint*. https://arxiv.org/abs/1707.06347