Assignments

You should now use the starter code for this project.

```
git clone https://github.com/WhoKnowsssss/RL-Training-MiniProject.git

mv RL-Training-MiniProject/ mjlab/
cd mjlab
```

If you are using Google Colab, please use the following link: https://colab.research.google.com/drive/1xQ31AbKAkpedh0xShIx8lM7zTfRr93_4?usp=sharing

1 Writing a Velocity Tracking Training Environment

This section describes, in detail, the conceptual purpose and required design choices to train a velocity-tracking locomotion policy for a quadrupedal robot.

Important disclaimer. Although this project is based on MJLab, you should not directly copy from the official implementation.

First, the version used in this assignment is intentionally simplified and modified; using the official code verbatim will produce **different results** than those expected from following the instructions. Submitting results that closely mirror the official implementation will result in **all credit** being deducted.

Second, the second part of the project requires you to design and implement your *own* RL training environment. Part 1 serves as a guided tutorial to help you understand how such an environment is constructed from scratch. Skipping this step will make the second part of the project **significantly more difficult** and may lead to a very frustrating experience.

(a) Setting Up the Robot Model

Before defining any control or reinforcement learning task, the first step is to load the robot correctly. This often involves the following essential steps:

- Load the robot's physical description and associated mesh assets.
- Identify all actuated joints and assign the correct physical parameters.
- Select PD gains and action scales for the actuators.

Load physical description With MJ-Lab, the quadrupedal robot Unitree Go1 is specified by an MJCF XML file that defines its geometry, joint structure, inertial properties, and collision shapes. The simulator must load this XML file together with its associated mesh assets so that the robot can be instantiated with physically meaningful dynamics.

Assign Actuator Parameters Once the base model is loaded, we must configure the actuator properties that determine how each joint physically behaves. The most important quantity is the joint *armature*, which represents the inertia seen at the output of the actuator. For electric motors with gearboxes, this is computed using the reflected-inertia formula

$$J = J_r g^2,$$

where J_r is the rotor inertia and g is the gear ratio. Using the correct reflected inertia ensures that joint accelerations behave realistically.

Other important physical parameters are the velocity and torque (effort) limits, which determine how fast and how strongly each joint can move. For the Go1, hip actuators use a velocity limit of 30.1 rad/s and an effort limit of 23.7 Nm, while knee actuators use 20.06 rad/s and 35.55 Nm. These values reflect the real hardware and keep the simulation within realistic bounds. Accurate limits are crucial because the RL policy learns behaviors that depend on the available torque and speed, and unrealistic values can result in behaviors that do not transfer to the actual robot.

Select PD Gains and Scales With the armature calculated, we add PD controllers to each actuator. Why do we use PD controllers here? Although PD control is often considered inadequate in classical robotics because it must follow a precise, time-indexed trajectory, it works extremely well for RL training [HLD+19]. This is because the RL policy **does not** need perfect tracking at every step. Instead, it learns to use the tracking errors and overshoots to generate the forces needed for movement and to remain robust under variations in the robot's kinematics. In practice, this makes PD control a simple and effective choice for learning locomotion behaviors with a low-frequency policy.

To determine these PD gains, we use a second-order system approximation with a chosen natural frequency and damping ratio:

$$k = J\omega^2, \qquad d = 2\zeta J\omega,$$

where J is the armature, ω the desired natural frequency (e.g. 10 Hz), and ζ the damping ratio (e.g. 2.0). This heuristic consistently produces more stable and better-behaved actuators than manual hand-tuning.

Finally, we compute an action-scaling factor that maps the normalized RL action space $[-1, 1]$ into a reasonable range of joint commands. A common and effective heuristic is

$$\text{action\_scale} = 0.25 \cdot \frac{\text{effort\_limit}}{k},$$

which ensures that the actuator can generate sufficient torques without saturating too easily, while still allowing meaningful exploration.

(b) Specifying the Command

Locomotion in this task is driven by a set of *velocity commands* that define the target motion the policy must track. These include forward, lateral, and yaw-velocity targets, and the structure of this command distribution is crucial for obtaining stable, diverse, and directionally consistent locomotion behaviors.

A key subtlety is that *angular-velocity commands alone* are insufficient for robust turning. A yaw-rate command (e.g., 0.3 rad/s) specifies only how fast the robot should rotate at that moment, without indicating the orientation it should ultimately face. Without an absolute directional goal, the robot receives only differential rotation instructions, making long-horizon heading regulation difficult and often leading to inaccurate turning.

To address this, a fraction of environments should instead issue explicit *heading commands*, providing an absolute target orientation sampled from $[-\pi, \pi]$. A simple proportional controller converts heading error into an angular-velocity target, allowing the policy to correct drift and maintain consistent orientation.

We provide a template command generator `mdp.UniformVelocityCommandCfg`. It should be configured with the following properties:

- Commands are resampled uniformly every **3.0 s to 8.0 s**, allowing the robot sufficient time to realize each target.
- Because true zero velocity is rarely sampled from continuous ranges, and balancing in place is important, **10%** of environments should explicitly enforce a standing (zero-velocity) command.
- To ensure meaningful turning behavior, **30%** of environments should use *heading-only* commands that provide explicit orientation targets.
- Linear velocity targets should lie within $[-1.0, 1.0]$ m/s for both forward (x) and lateral (y) directions.
- Yaw-velocity targets should lie within $[-1.0, 1.0]$ rad/s.
- Heading targets should be sampled uniformly from $[-\pi, \pi]$.
- The heading controller should use a stiffness of **1.0**, providing moderate corrective strength.

This command distribution exposes the robot to a wide variety of tasks—including standing, turning, multidirectional walking, and combinations thereof—leading to more robust velocity tracking policies.

(c) Writing the Main Objectives as Rewards

Remember that RL policies optimize the cumulative sum of rewards, so the task must be expressed through reward terms that directly promote the desired behavior. With the commands defined above, the primary reward components should incentivize the policy to track them accurately. To achieve this, we use Gaussian-shaped rewards, which provide smooth gradients and encourage precise tracking of both the commanded linear and angular velocities:

- **Linear velocity tracking**: The robot should match the commanded translational velocity. Use a Gaussian penalty with standard deviation $\sqrt{0.25}$ and weight **2.0**.
- **Angular (yaw) velocity tracking**: The robot should match the commanded yaw rate, using a Gaussian penalty with standard deviation $\sqrt{0.25}$ and the same weight **2.0**.

These two terms form the core of the task and should dominate the overall reward design.

(d) Writing Regularizations as Rewards

Regularization terms promote physically plausible, hardware-safe, and visually natural behavior. They prevent the policy from exploiting simulation artifacts and improve gait smoothness.

- **Uprightness**: Penalize deviations of the torso from upright using a Gaussian penalty with standard deviation $\sqrt{0.2}$ and weight **1.0**. This encourages stable posture.
- **Default joint posture**: Encourage joints to remain near a nominal reference posture using a mild penalty with weight **-0.1**. This prevents unnatural crouching or contorted stances.
- **Joint-limit penalty**: Penalize approaching joint limits with a stronger weight of **-1.0**, reinforcing safe and feasible joint motion.
- **Action-rate penalty**: Discourage abrupt changes in actions by applying a smoothness penalty with weight **-0.1**. This produces more natural control signals and improves sim-to-real transfer.

Together, these regularizers ensure stable, smooth, and hardware-friendly policies.

(e) Writing Termination Conditions

Early termination is an important technique in RL because it prevents the policy from spending large portions of an episode in failure states that provide no useful learning signal. Once the robot has clearly fallen, continuing the rollout only adds noise to the return and slows down training. For this reason, most RL environments, as well as this mini-project, include an early-termination condition to halt episodes as soon as the policy enters an unrecoverable state.

- **Fall-over termination**: If the robot's torso tilts beyond $60°$, the episode should end immediately. This avoids prolonged simulation where the robot has fallen over.

(f) Writing Observations

A locomotion controller require observations that capture both the robot's state and the task objectives. These observations should provide the policy with sufficient proprioceptive information to maintain balance and coordinate its limbs, as well as task-related information needed to track the commanded motion. In addition, injecting appropriate sensor noise into the policy observations is crucial for improving robustness, as it helps the learned controller remain stable and effective when deployed on real-world hardware.

- **Base linear velocity**: Estimated from the IMU; noise in $[-0.5, 0.5]$ encourages robustness to velocity-estimation errors.
- **Base angular velocity**: Also from the IMU; noise in $[-0.2, 0.2]$ models realistic gyroscope noise.
- **Projected gravity**: Encodes torso orientation relative to the world; noise in $[-0.05, 0.05]$ simulates accelerometer noise.
- **Relative joint positions**: Provides posture information; small noise in $[-0.01, 0.01]$ prevents overfitting to exact joint readings.
- **Relative joint velocities**: Captures dynamic state; broader noise in $[-1.5, 1.5]$ reflects higher noise in encoder velocity reading.
- **Previous action**: Helps smooth the control sequence by giving the policy temporal context.
- **Command information**: The generated twist command must be observed so the policy knows what velocity to track.

The **critic** receives the same set of observations but without corruption. Recall that in actor–critic RL, the critic's role is to approximate the value function, providing a low-variance learning signal to guide the policy (the actor). Thus, providing the critic with clean, uncorrupted observations, and sometimes additional ground-truth information unavailable to the policy, is known as *asymmetric critic training*. This technique improves value estimation, accelerates learning, and does not introduce bias, since the actor is still trained only on its own noisy observation space.

(g) Domain Randomization

Domain randomization (DR) [TFR+17] is a technique in which key physical or sensory parameters of the simulation are intentionally varied during training. The goal is to prevent the policy from overfitting to a single, overly idealized simulator configuration. By exposing the policy to a wide range of environmental conditions, DR encourages the controller to rely on strategies that remain robust to real world parameters that differ from the simulator. This approach has been widely adopted in robotics for improving sim-to-real transfer.

To improve robustness in this project, DR should be applied along two key dimensions:

- **Foot-friction randomization**: At environment startup, randomize the friction coefficient of all foot geometries within the interval $[0.3, 1.2]$. Foot–ground friction is one of the most difficult aspects of locomotion to model accurately, and even small discrepancies between simulated and real friction can dramatically change gait stability, slip behavior, and turning dynamics. Randomizing friction ensures that the policy does not rely on a single, precise contact condition and instead learns strategies that remain stable across a range of real-world surfaces (e.g., concrete, carpet, wood).
- **Random perturbations**: Apply random velocity pushes at intervals sampled uniformly between **1.0 s and 3.0 s**. Each perturbation should add to the base velocity in the x and y directions within $[-0.5, 0.5]$ m/s. These disturbances simulate unexpected external forces—such as bumps, slips, or minor collisions—that are common in real-world deployment but difficult to model precisely. Exposing the policy to such perturbations during training encourages it to learn fast, stable recovery strategies, improving robustness and reducing the likelihood of falling when similar disturbances occur on hardware.

These DR strategies significantly enhance the robustness of the learned controller and are standard practice in sim-to-real locomotion.

Deliverables

After completing these steps, you are now ready to begin training. Refer to Section 1 for the training and play commands. Training stops at 1000 iterations by default, but if your implementation is correct, the reward should already be high by around 400 iterations. On Google Colab with a T4 GPU, reaching 400 iterations typically takes about 12 minutes, and a full 1000-iteration run takes roughly 30 minutes. If you do not obtain a reasonable policy within the first 15 minutes, stop and debug rather than letting the training run to completion.

Your submission for this section should include the following components:

1. **Screencasts or videos of the Go1 walking under sampled commands.** Demonstrate your trained controller responding to different commanded velocities, including forward walking, lateral walking, turning, and standing. The video should clearly show stable and consistent locomotion behavior.

2. **Training curves showing overall learning progress.** Include plots of the following quantities logged during training:

   - `Train/mean_episode_length` — indicates stability and how frequently early terminations occur.
   - `Train/mean_reward` — measures overall task performance and learning progress.

   These curves should run long enough to show clear convergence trends.

3. **Training curves for velocity-tracking performance.** Provide plots of both the linear and angular velocity tracking rewards and the corresponding error metrics throughout training. These curves should show that your controller is improving in its ability to follow both translational and rotational commands over time.

4. **Test-time command-tracking curve.** After training, evaluate your final policy on a sequence of commanded velocities and plot the *actual* measured linear and angular velocities alongside the *commanded* ones. This curve should demonstrate how well the trained controller follows time-varying commands at test time. Show a sequence of the following commands (125 environment steps each):

   - Forward walking: $(v_x, v_y, \omega_z) = (0 \to 0.6,\ 0,\ 0)$.
   - Lateral walking: $(v_x, v_y, \omega_z) = (0,\ 0.4,\ 0)$.
   - Turning command: $(v_x, v_y, \omega_z) = (0,\ 0,\ 0.4)$.
   - Mixed command: $(v_x, v_y, \omega_z) = (0.5,\ 0,\ 0.3)$.

All required plots should be clearly labeled and generated from your own training logs. Videos should demonstrate the behavior of your trained policy using the commands described in the assignment.

2 Improving your Velocity Tracking Controller

The basic velocity-tracking controller introduced in Part 1 produces reasonable locomotion, but it often lacks the refinement and robustness required for high-quality deployment. In this section, you will augment your controller with additional design components that improve gait quality, stability, and learning efficiency.

(a) Writing Gait-Shaping Reward Terms

Even with correct tracking objectives and moderate regularization, the resulting locomotion may exhibit dragging, shuffling, or excessive foot slip. Gait-shaping rewards mitigate this by encouraging cleaner and more natural foot trajectories.

- **Foot clearance**: When the commanded speed exceeds $0.05\,\mathrm{m/s}$, the swing foot should achieve a clearance of approximately $\mathbf{0.1\,m}$. Use a penalty weight of **-2.0** discourages dragging.

- **Swing height consistency**: When the foot is not in ground contact (as indicated by the contact sensor), its height should remain near the target swing height of **0.1 m**. This term should use a lighter weight of **-0.25**.
- **Foot slip**: To encourage stable stance, slipping of a foot while in contact should be penalized with weight **-0.1**.

Hint: Use `site_names`, which is the list of feet sites for height and clearance computation.

These gait terms help produce cleaner steps, improved ground clearance, and reduced slipping, all of which are crucial for hardware deployment.

(b) Adding Privileged Information to Your Critic

In actor–critic reinforcement learning, the critic is not constrained to use the same observations as the policy. Because the critic is used only during training and not at test time, it may access additional *privileged information* that is not available to the actor. This extra information allows the critic to produce more accurate value estimates, which in turn stabilizes training and provides lower-variance gradients to the policy. A better critic leads to faster learning, more reliable convergence, and improved robustness of the final controller.

For legged locomotion, foot-related quantities are especially informative for evaluating the quality of a state. Characteristics such as contact timing, ground reaction forces, and swing heights directly influence stability and energy usage. While the actor should not rely on these privileged signals—since they are not always available or reliable in real-world sensing—the critic can benefit greatly from them.

In MJLab, you can strengthen your critic by adding the following privileged observation terms:

- **Foot height**: Provides the height of each foot relative to the base. Useful for judging gait symmetry, step timing, and whether the foot is appropriately lifted during swing.
- **Foot air time**: Measures how long each foot has been off the ground. Helps the critic infer gait phase and detect irregular timing patterns.
- **Foot contact state/force**: Indicates whether each foot is currently touching the ground. This simplifies reasoning about stance vs. swing phases and helps the critic evaluate stability.

These terms should be added *only* to the critic observation group. The actor should rely solely on proprioception and commands to avoid dependence on ground-truth quantities unavailable on hardware.

Deliverables

Your submission for this section should include the following analysis components.

1. **Effect of gait-shaping rewards on foot motion.** Plot the front-left foot position trajectory for two models: one trained *with* gait-shaping rewards and one trained *without* them. Discuss what you observe. For example, examine differences in swing height, clearance consistency, dragging, or irregularities in step timing.
2. **Effect of privileged critic observations.** Plot two sets of metrics, `slip_velocity` and the linear velocity-tracking error, for models trained *with* and *without* the additional critic-only observations. Explain the differences you observe and describe how privileged critic information influences training, tracking accuracy, and gait quality.

Each deliverable should include both the plot and a concise explanation of the behavior difference revealed by the comparison.

3 Training a Backflip Policy for Go2

In this section, you will design and train a highly dynamic control policy that performs a backflip with the Unitree Go2. Unlike velocity tracking, this task focuses on executing a precise, time-varying maneuver that involves flight, rotation, and a stable landing.

(a) Robot Setup

You are provided with an MJCF XML file for the Go2 robot. Your first step is to set up a Go2 robot configuration analogous to the Go1 setup from the previous section:

- Load the Go2 MJCF XML and associated assets.
- Define actuator constants (reflected inertia, PD gains, velocity and effort limits) as you did for Go1. You may assume that Go2 uses the same hip motors as Go1. However, its knee motor has a gear ratio of 12, a velocity limit of 15.70 rad/s, and an effort limit of 45.43 Nm.
- Compute action scaling for Go2.

This configuration will be reused across the backflip and goal-reaching tasks.

(b) Designing a Backflip MDP

You will write a new MDP tailored for the backflip task. Unlike the velocity-tracking environment, this MDP should execute a *single, coordinated maneuver* from an initial stance to a completed backflip and landing.

Hint: Begin by designing a command generator that can output simple reference quantities, such as a desired base height or a desired orientation, that roughly describe the progression of a backflip. These references do not need to be precise trajectories; even coarse, hand-crafted shapes are sufficient to guide learning. Other components should follow naturally from this setup: include observations that allow the robot to understand its motion and its current progress; design a small number of straightforward reward terms that encourage following the intended flip behavior; and implement basic termination conditions for clear success or failure.

(Optional) Phase variable. For such a time-critical motion, it is often convenient to introduce a *phase* variable $\phi \in [0, 1]$ that parameterizes progress through the backflip:

- $\phi = 0$: initial standing pose.
- $\phi \approx 0.25$: takeoff and beginning of rotation.
- $\phi \approx 0.5$: midair, upside-down orientation.
- $\phi \approx 0.75$: approaching landing.
- $\phi = 1$: final landing and recovery pose.

Deliverables

1. **Training curves.** Plot `Train/mean_reward` and `Train/mean_episode_length` over the course of training. Describe your development process: what design choices or changes improved learning, what attempts did not work as expected, and how these observations guided you toward your final working solution.
2. **Demonstration video.** Provide a screencast or video showing the Go2 successfully performing backflips. The video should clearly show the full maneuver: preparation, takeoff, rotation, and landing.

Rubrics

Part 1: Go1 Velocity Tracking Environment (40 points)

Implementation (20 pts)

- Robot model setup (4 pts): correct reflected inertia, PD gains, limits, and action scaling.

- Command generator (4 pts): correct velocity/heading sampling, standing mode, heading-only mode.
- Rewards + termination (6 pts): correct linear/angular tracking terms, regularizers, fall-over termination.
- Observations + critic (4 pts): correct noisy actor obs, clean/privileged critic obs.
- Domain randomization (2 pts): friction randomization + velocity perturbations.

Deliverables (20 pts)

- Locomotion video (8 pts): stable walking under diverse commands.
- Training curves for mean reward + episode length (4 pts).
- Training curves for velocity tracking rewards and metrics (4 pts).
- Test-time command tracking plots (4 pts).

Part 2: Controller Improvements (20 points)

Implementation (10 pts)

- Gait shaping (5 pts): correct clearance, swing-height, and slip rewards.
- Privileged critic (5 pts): correct inclusion of foot-height/air-time/contact in critic only.

Deliverables (10 pts)

- Gait shaping plots (5 pts): foot trajectory comparison + short explanation.
- Privileged critic plots (5 pts): slip + tracking comparison + short explanation.

Part 3: Go2 Backflip (40 points)

Implementation (20 pts)

- Go2 robot setup (5 pts): correct actuators and scaling.
- Backflip MDP (15 pts): reasonable command design, observations, rewards, and terminations.

Deliverables (20 pts)

- Training curves (10 pts): reward + episode length, with brief notes on what worked and what did not.
- Demo video (10 pts): clear backflip execution (takeoff, rotation, landing).

Your grade will be evaluated according to these criteria.

Academic Integrity & Originality Deductions

- Using the official MJLab environment instead of your own implementation: **−100 points**
- Using logs or videos from pretrained MJLab/IsaacLab controllers: **−100 points**

Total: 100 points

References

[HLD+19] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, 2019.

[TFR+17] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.