RL Training Mini-Project

For Legged Robots

November 18, 2025

Background and Getting Started

Reinforcement Learning Basics

Reinforcement learning formulates control as a Markov Decision Process (MDP), described by the tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \gamma),$$

where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P(s' \mid s, a)$ is the transition model, $r(s, a)$ is the reward, and $\gamma \in (0, 1)$ is the discount factor.

At each time step t, the policy $\pi_\theta(a \mid s)$ outputs an action, and the goal of RL is to maximize the expected discounted return:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right].$$

To optimize the policy with lower variance, modern RL methods use an *actor–critic* structure. The actor is the policy, and the critic is a neural network that estimates how good a state is under the current policy through the value function:

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=T}^{\infty} \gamma^t r(s_t, a_t) \,\middle|\, s_T = s\right].$$

This learned estimate provides a low-variance training signal for the actor.

Using the critic, we define the advantage:

$$A^\pi(s, a) = r(s, a) + \gamma V^\pi(s') - V^\pi(s),$$

which measures how much better an action is compared to what the critic expected. The policy is then updated using an advantage-weighted gradient:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, A^\pi(s_t, a_t)].$$

A industry-standard RL training framework for legged robots is provided by **MJLab** and **RSL-RL**, a lightweight and modular research codebase built on top of MuJoCo. Inherited from IsaacLab, MJLab offers a clean abstraction for defining robots, actuators, rewards, observations, and domain randomization strategies, while RSL-RL provides efficient PPO-based training loops optimized for large-scale parallel simulation. It is designed specifically for locomotion research and emphasizes clarity and configurability: students edit configuration files rather than rewriting algorithms, allowing them to focus on understanding robot modeling and task design. Throughout this homework, you will implement each component of the Go1 velocity-tracking environment directly within the MJLab framework.

Getting Started

Here, we provide an example training task, standing up from a random pose, as a reference. You can directly pull the project, start training, and run the policy to see a quadrupedal robot balance itself from random initial configurations. You may use this repository as an example to guide your own project implementation.

Local Installation

First, clone the MJLab repository and change into its directory:

```
git clone https://github.com/WhoKnowsssss/RL-Training-MiniProject-Example.git
mv RL-Training-MiniProject-Example/ mjlab/
cd mjlab
```

We provide a ready-made training script for the Unitree Go1 velocity tracking task. After completing Section 2, you can launch training with:

```
MUJOCO_GL=egl uv run train Mjlab-Velocity-Flat-Unitree-Go1 --env.scene.num-envs 4096
```

Here, `MUJOCO_GL=egl` enables headless rendering suitable for cluster or server environments, `uv run train` invokes the training entry point, `Mjlab-Velocity-Flat-Unitree-Go1` selects the Go1 flat-ground velocity task, and `--env.scene.num-envs 4096` sets the number of parallel environments. A larger number of environments decreases the variance of the collected experience but also increases the time required per iteration. Empirically, 4096 is a good starting point. Training logs in Tensorboard and checkpoints will be written under `logs/rsl_rl/go1_velocity/` by default. You can visualize the training curves by `tensorboard --logdir=logs/rsl_rl/go1_velocity/`

Once training finishes and you have a checkpoint, you can visualize the learned policy using the `play` command:

```
uv run play Mjlab-Velocity-Flat-Unitree-Go1 --checkpoint_file={your path}
```

Replace {`your path`} with the actual checkpoint filename you want to load (for example, `model_xxxx.pt`). This will start an interactive viewer where you can observe the Go1 executing the learned velocity tracking behavior using the configuration you implemented in the earlier sections. Remember to click the `Track camera` option when recording videos.

Using Google CoLab

For students who do not have GPUs or who prefer not to install MJLab locally, a ready-to-use Google Colab notebook is provided:

https://colab.research.google.com/drive/1p3izqzddTcmUL8MaEPtRlhgup2sCi003?usp=sharing

This notebook contains a fully configured environment with all necessary dependencies pre-installed, allowing you to run training, visualization, and debugging entirely from the browser. Remember to connect to a *GPU runtime*.

Warning: Colab frequently disconnects runtimes, and any files stored on the instance will be deleted when the session resets. Please make sure to keep a **local copy** of all your work and sync it to Colab as needed. Do **NOT** rely on Colab's temporary storage for saving checkpoints, logs, or important code.

Basics for Coding in Isaac/MJLab

MJLab organizes each reinforcement learning environment as a collection of modular components that together define the MDP. To complete this project, you will frequently read and modify these components, so it is important to understand where each part of the environment lives and how they connect.

Where the MDP is defined. The core implementation of the MDP is located in:

- `src/mjlab/envs/mdp/`
- `src/mjlab/tasks/velocity/mdp/`

Each file in these directories corresponds to one aspect of the environment:

- **Reward terms:** `src/mjlab/tasks/velocity/mdp/rewards.py`
- **Observation terms:** `src/mjlab/tasks/velocity/mdp/observations.py`

- **Termination criteria:** `src/mjlab/tasks/velocity/mdp/terminations.py`
- **Command generator:** `src/mjlab/tasks/velocity/mdp/velocity_command.py`
- **Curriculum learning:** `src/mjlab/tasks/velocity/mdp/curriculums.py`

These are the files you will modify when implementing new reward terms, observation terms, termination conditions, or command generation logic.

Where terms are assembled into an environment. The complete environment configuration for the velocity tracking task is defined in:

`src/mjlab/tasks/velocity/velocity_env_cfg.py`.

This file builds the full RL environment by assembling rewards, observations, terminations, and events into a single `ManagerBasedRlEnvCfg`. The configuration classes used here, such as `RewardTermCfg`, `ObservationTermCfg`, `EventTermCfg`, and `TerminationTermCfg`, are defined in:

`src/mjlab/managers/manager_term_config.py`.

All term configurations follow the same basic structure:

```
XXXXTermCfg(
    func = mdp.xxx,         # function to call
    params = {
        "yyy": value,       # arguments passed to mdp.xxx(...)
        "zzz": value,
    },
    ...additional options...
)
```

Here, `mdp.xxx` must be a function you have defined in one of the MDP files, and it must follow the signature:

`def xxx(env:  ManagerBasedRlEnv, yyy, zzz):`

The environment is always passed in automatically, followed by any parameters you specify in the configuration.

Writing your own term. As an example of how an MDP term is written, consider the linear velocity tracking reward:

```
def track_linear_velocity(
    env: ManagerBasedRlEnv,
    asset_cfg: SceneEntityCfg,
    std: float,
    command_name: str,
) -> torch.Tensor:
    # Retrieve robot object
    asset: Entity = env.scene[asset_cfg.name]

    # Retrieve command term
    command = env.command_manager.get_command(command_name)

    # Compute tracking error
    actual = asset.data.root_link_lin_vel_b
    xy_error = torch.sum(torch.square(command[:, :2] - actual[:, :2]), dim=1)
    z_error = torch.square(actual[:, 2])
    lin_vel_error = xy_error + z_error

    return torch.exp(-lin_vel_error / std**2)
```

This function retrieves the robot's base linear velocity, compares it to the command, computes the tracking error, and returns a Gaussian-shaped reward. Once such a function is implemented, the environment will call it automatically every simulation step using its corresponding `RewardTermCfg` entry in `velocity_env_cfg.py`.

Understanding this workflow, i.e., defining a term in the MDP directory, then registering it in the environment configuration, is essential for completing both parts of the project.