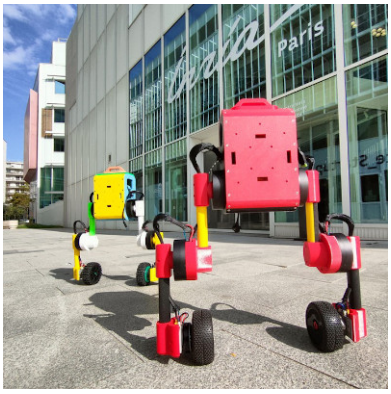
Robotics MVA 2025–26 Final exam topics

Reinforcement learning for wheeled-bipedal push recovery



Context: Upkie [1] is a wheeled biped robot developed in the Willow team at Inria. It can balance itself upright and traverse various terrains. One important sub-task of balancing is *push recovery*, the ability to withstand external disturbances like pushes. In this project, we will train balancing policies and measure their robustness as the maximum push that the robot can sustain without falling. We will define this quantity as the *maximum sagittal force over one second*, which we will write MSFOS for short.

Goal: evaluate the balancing performance of a linear feedback controller, then train a more robust behavior by reinforcement learning of a neural-network policy.

Pre-flight checks: before selecting this topic, pay attention to the following two points : (1) RL is compute-intensive, so you should make sure the provided software runs fast enough on your computer, and (2) that software *only works on Linux and macOS*. Don't select this topic if you only have Windows.

Project plan

Let us start with making quantitative measurements of the push-recovery capability of a given policy :

- Clone the upkie repository.
    - After cloning the repository, checkout v8.1.1 of the robot software (this is the version that corresponds to the training environment you will use thereafter): `git checkout v8.1.1`
- Try out a first balancing policy using only pure linear feedback of the base pitch angle, following the readme or adapting the PD-balancing example.
    - You can start the Bullet simulator using `./start_simulation.sh` as documented in the upkie v8.1.1 readme.
- Adapting one of the examples, apply a *sagittal* (forward-backward) force to the robot for 1 second. With your policy from Question 2, what is the resulting MSFOS (in N) that you can apply before the robot falls ?
- Update your policy to linear feedback of the full observation vector, and tune the different feedback gains to get the best performance you can.

Question 1: what is the best MSFOS you can achieve with a linear-feedback policy?

From there on, let us switch to training a neural-network policy by proximal policy optimization (PPO) [3].

- Clone the PPO balancer and run the pre-trained policy following readme instructions. How well does it perform in MSFOS ?
- Run the training script to train a new policy, adjusting the number of processes to your computer. What is a proper number of processes for your computer ?
- Run your trained policy and check its MSFOS.

Question 2: what are the MSFOS of the pre-trained and your re-trained policy ? On average, in how many steps does your training converge?

Improve your policy using one of the techniques seen in class that is not implemented in the PPO balancer, for instance:

- Curriculum learning,
- Reward shaping, or
- Teacher-student distillation.

Question 3: what is the MSFOS of your final policy? On average, in how many steps did you train it?

Make sure you keep the parameters file `params.zip` of your best policy to show it in simulation (or on the real robot! If it behaves reasonably well...) during the final evaluation.

Extension to all robot joints

Let us now consider a more general balancing policy that will use not only the wheels, like we did so far, but also hip and knee joints. With these additional leg joints, an Upkie can for example bend its legs in reaction to a push. For this extension, you have two options:

1. Keep using the Bullet simulator and switch from the *UpkieGroundVelocity* to the *UpkieServos* environment. This simulator runs on CPU and will work on Linux and macOS.
2. If you are feeling playful: try mjlab for massively-parallel training on GPU. ⚠ This alternative may only work on Linux with an NVIDIA GPU. This extension spends more time building up the training environment, but once set up it will train faster.

E1: UpkieServos in Bullet

With the *UpkieServos* Gymnasium environment, we can send position or velocity commands to each joint of the robot. The goal of this extension is to train a balancing policy for this more capable environment.

First, check out the new observations and actions from this environment. Note that the pitch angle is not returned in observations any more. Rather, it is returned in the `info['spine_observation']` dictionary returned by `reset` and `step`, and documented in this page.

- Implement a wrapper around the *UpkieServos* environment with vector observations and actions. You can read pitch and ground position from the observation dictionary, as done in UpkieGroundVelocity (which is, as a matter of fact, also a wrapper).
- Your observation vectors should include joint position, joint velocities, the pitch angle of the floating base and its derivative. You can also experiment with including ground position/velocity in observations.
- Your action space should include 2 × hip positions, 2 × knee positions and 1 × ground velocity.
- Update `train.py` and `run.py` to create an UpkieServos environment and add your custom wrapper around it, instead of a UpkieGroundVelocity environment.
- Train a new balancing policy for this extended environment.

Question 4: what is the MSFOS of your trained policy? On average, in how many steps did you train it, and how much (wall) time did it take?

Action shaping is a way to make the policy less end-to-end by adding some engineering to the environment. For instance, since the two limbs of the robot have roughly the same length, it is usually a good idea to set knee ankle = ±2 * hip angle so that the action becomes [crouching, wheel velocity] for each leg.

- Add some action shaping to your environment.

Question 5: same as Question 4 after action shaping. Did it improve MSFOS? What is the average number of training steps until convergence?

E2: UpkieServos in mjlab

⚠ This alternative uses bleeding-edge software that will only run on a Linux system with an NVIDIA GPU. Make sure you fall back to the Bullet version of the extension at the first sign of a bug.

The goal of this extension is to train a balancing policy for a more capable environment where all robot joints receive position commands.

- Clone mjlab and check that the demo works on your system: `uv run demo` (if not, switch to the Bullet extension)

Next, set up the training environment by creating a new mjlab task :

- Download the following robot model: rl-push-recovery-mjlab-template.zip
- Follow the Creating a New Task tutorial, using that model for step 1
- You can use a joint position action for all joints.
- For observations, the orientation of the robot's floating base is stored in `env.sim.data.qpos[:, 0:7]`, 3 translation coordinates followed by 4 quaternion coordinates.
    - Compute the pitch angle, i.e., the angle between the z-axis of the floating base frame and the z-axis of the world frame.

    - You can also experiment with including ground position/velocity (computed from wheel position/velocity) in observations.
- Train a balancing policy for your custom task.

Question 4: what is the MSFOS of your trained policy? On average, in how many steps did you train it, and how much (wall) time did it take?

Action shaping is a way to make the policy less end-to-end by adding some engineering to the environment. For instance, since the two limbs of the robot have roughly the same length, it is usually a good idea to set knee ankle = ±2 * hip angle so that the action becomes [crouching, wheel velocity] for each leg.

- Add some action shaping to your environment.

Question 5: same as Question 4 after action shaping. Did it improve MSFOS? What is the average number of training steps until convergence?

References:

- [1] Upkie wheeled biped robots (2022–2025)
- [2] Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations.
- [3] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.