Advanced RL for Robotics Applications

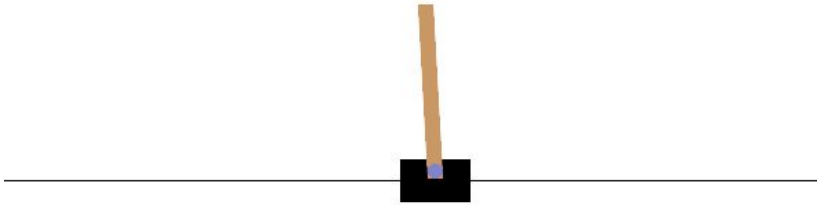
by VantTec & RASTEC

Reinforcement Learning

Reinforcement learning is a form of **machine learning** that is trained on **experiences** and their attributed **rewards**.

It involves an **environment**, which rewards certain behaviors at certain states, and an **agent** that **interacts** with the environment by performing observations and actions, then interpreting the rewards to improve on further actions.

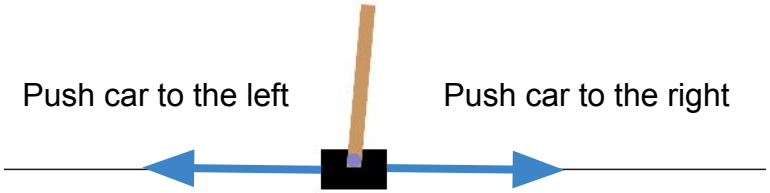
Environment



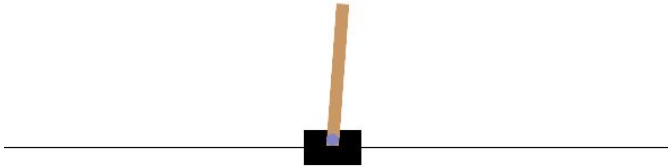
Cart Pole

Robot dog on a floor

Actions



Rewards



Still up: **+1**

Fallen over or reached edge: **Terminated**

Observation space

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Q-Learning

Common and effective RL algorithm

Goal: Maximize total discounted return

Logic:

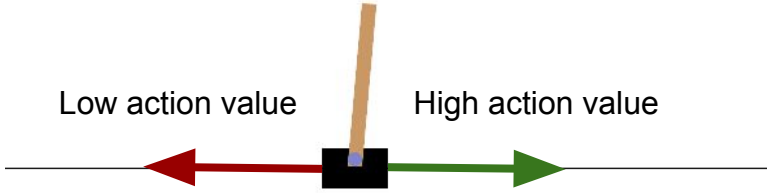
- In each state, choose the action with the best attributed action value (Q).
- If the best action is always chosen, then the best trajectory is taken.
- First action values need to be estimated for each state, then those estimated values are followed by the policy.

Action Values

The value of each action depends on the state, so they are expressed as a function of both the state (s) and the action (a).

$$ActionValue = Q(s, a)$$

For example, if the state describes that the pole is leaning to the right, we expect the action of moving to the right to be more valuable than moving to the left.



Action Values

Action values are estimated using the **temporal difference** method.

- We want to value both the immediate and long term rewards for each action, so we need to use both the reward and the total return.
- We want to distinguish each action's contribution to the total return.

$$\text{Action value} = r_t + \gamma \cdot Q(s_{t+1}, a_{t+1})$$

To balance how much we care about immediate rewards or future returns, we use a **discount factor** (gamma).

Q-Learning

For Q-Learning, we estimate the future return using the best possible action in the next state. So, after performing each action, its resulting experienced value can be calculated:

$$\text{Experienced action value} = r_t + \gamma \cdot Q(s_{t+1}, a_{best})$$

Learning rate

Because of uncertainty, we cannot guarantee that the action will always have the same value, instead of setting the estimate to the experienced value, we move it closer by a set **learning rate** (alpha).

$$\text{New value} = \text{Previous estimate} + \alpha \cdot (\text{Experienced value - Previous estimate})$$

- Higher learning can cause instability.
- Low learning rate can be inefficient.

Exploration vs exploitation.

As stated before, when selecting an action, Q-Learning selects the action with the highest action value. However, because there is much uncertainty in the known action values, it is important to **explore** different actions, as there might be a better, unknown trajectory. So, for every action, there is a set probability (epsilon) to pick a random action instead.

- As training continues, epsilon is reduced to make sure the algorithm **exploits** the good known trajectories.

Summary

1. The agent begins picking actions at random, assuming equal value to every action.
2. The model estimates an action's value through both its reward and the **discounted** value of the best known next action.
3. The corresponding action's value gets updated to reflect this estimated value according to a **learning rate**.
4. The agent now sometimes takes the best possible action and sometimes explores, according to the exploration factor **epsilon**.
5. Over time, the difference between an action's known value and its estimated value decreases, as does epsilon, resulting in convergence.

Deep Q Learning

For continuous observations, we can no longer store an action value for each possible action at each possible state, since there are infinite states. Instead, we create a **neural network** that maps the observation (the information that describes the state) to a set of the action values of that state.

- Neural networks can have varying numbers of **layers** and **neurons**.
- More layers and neurons can help fit more complex relationships but result in lower efficiency and the possibility of over-fitting.

Hyperparameters

For each reinforcement learning application, hyperparameters will need to be tuned. For deep Q-Learning, the following are most relevant:

1. Discount factor
2. Learning rate
3. Initial and final epsilon (exploration)
4. Epsilon decay rate
5. Layers and neurons of NN

DQN Limitations

1. Often overestimate action values.
2. Highly sensitive to hyperparameters.
3. Can be relatively unstable.
4. Exploration is primitive.
5. **Cannot handle continuous action spaces.**

Workaround

The action space can be discretized, so, for example, for an action space of force between -3 and 3, it can be split into 5 actions, which map to the action space.

0	1	2	3	4
-3	-1.5	0	1.5	3

PPO

Proximal Policy Optimization

Model-Based vs Model-Free

There are many parts involved in the design of algorithms, and many characteristics have to be considered before deciding which algorithm best fits our actual needs.

Here we can check some distinctions about Model-Based and Model-Free algorithms.

Model Based: This one requires a known model of the environment, because it carries precious information that can be used to find the desired policies, however, in most of the cases, the model is almost impossible to obtain, as it is quiet easy to obtain the full model of a tic-tac-toe game but not that easy a model for the waves of the sea.

Model-Free: This one doesn't need a known model of the environment, the algorithms on this one run trajectories within a given policy to gain experience and improve the agent by learning from the unknown model.

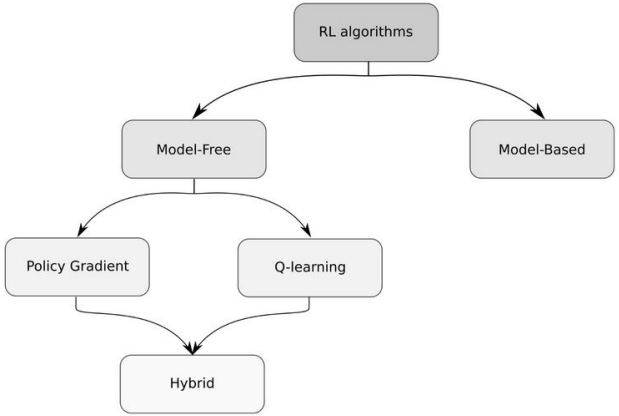


Figure 3.2. Categorization of RL algorithms

Isaac Lab Framework

Grasping Manipulator Training

Isaac Lab Framework

What is Isaac Lab?

Isaac Lab is a unified and modular framework for robot learning that aims to simplify common workflows in robotics research. It is built on Nvidia IsaacSim to leverage the latest simulation capabilities for photo-realistic scenes, and fast and efficient simulation.

Core advantages

- **Modularity**: Easily customize and add new environments, robots, and sensors.
- **Agility**: Adapt to the changing needs of the community.
- **Openness**: Remain open-sourced to allow the community to contribute and extend the framework.
- **Batteries-included**: Include a number of environments, sensors, and tasks that are ready to use.

IsaacLab vs IsaacSim

It is important not to get confuse between the core differences between IsaacLab and IsaacSim, so here we present some of the most important differences.

	IsaacSim	IsaacLab
Main purpose	General purpose 3D robotics simulations and visualizations	Research platform for robot trainings using RL
Level of Abstraction	Low- direct control of the scenes,physics, materials and sensors	High- modular RL frameworks with their respective configuration classes
Primary Focus	Prototyping, testing, and validating robotics systems in photorealistic environments	Large-scale trainings and evaluation of autonomous robots using learning based methods.
Target Users	Engineers and Researchers	Engineers and Researchers
Dependencies	Standalone simulator	Built on top of IsaacSim for physics, rendering and sensors
Main Output	Visualizations, sensor data, data sets	Trained policies performance

ROS2 + Isaac Sim

One of the most important advantages of using Isaac Lab and Isaac Sim for training and deploying policies for real robots, is that they can also be integrated with ROS2 framework, as we can see with this teleoperated turtlebot that is integrated with a basic Lidar that will allow us to use Rviz to visualize the point cloud.

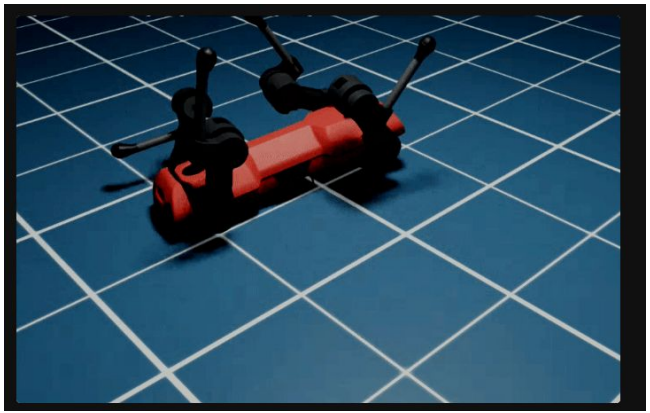
Insertar video (falta tomarlo en la ws)

RL Templates for Isaac Lab

We can actually choose these two different have of templates, which can have different advantages and disadvantages.

Model-Based

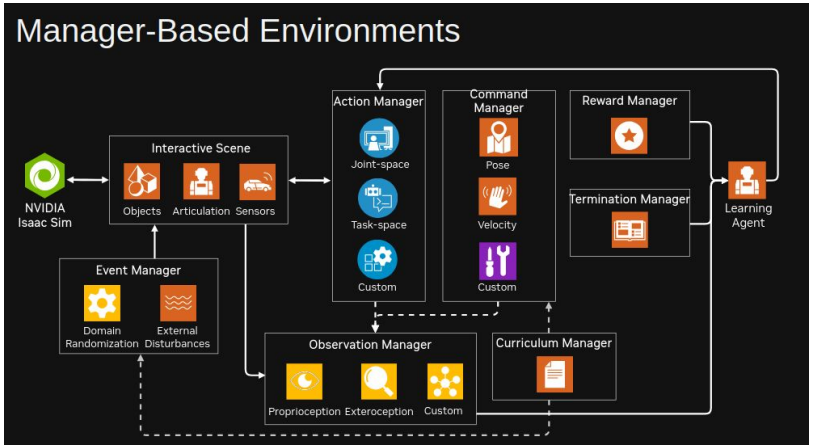
The environment is decomposed into individual components (or managers) that handle different aspects of the environment (such as computing observations, applying actions, and applying randomization). The user defines configuration classes for each component and the environment is responsible for coordinating the managers and calling their functions.

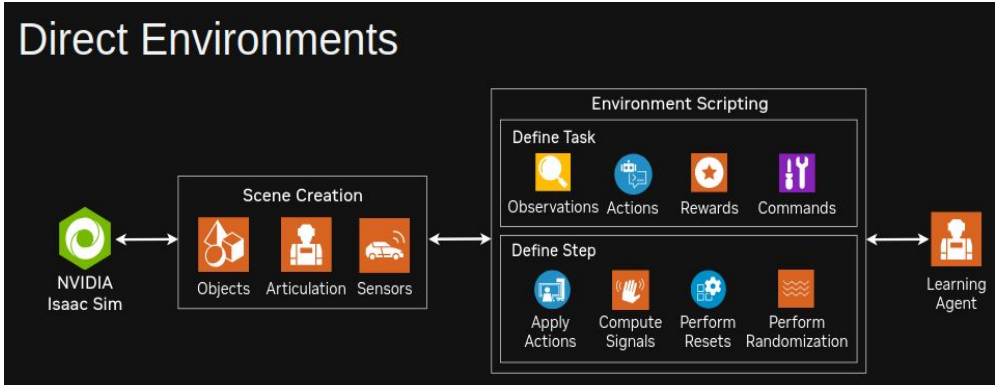


Direct

The user defines a single class that implements the entire environment directly without the need for separate managers. This class is responsible for computing observations, applying actions, and computing rewards.

But Which one should we use?





The manager-based workflow is more modular and allows different components of the environment to be swapped out easily. This is useful when prototyping the environment and experimenting with different configurations. On the other hand, the direct workflow is more efficient and allows for more fine-grained control over the environment logic. This is useful when optimizing the environment for performance or when implementing complex logic that is difficult to decompose into separate components.

So for this presentation we will be using Manager-based templates.

Configclasses

Before getting started with the code we need to define what our 7 main config classes consist about.

- **Action**
  This class intends to define the action space for the robot within the MDP.
- **Command**
  This class purpose is to define the range and frequency of target poses that the robot's end-effector must reach during the training.
- **Observations**
  This configuration defines the observations of the environment that will be send to the agent.
- **Terminations**
  This class specifies when a training episode should end. For this case we will use a timeout.
- **Events**
  This manager introduces domain randomizations, helping the policy generalize by exposing it to varied starting conditions.
- **Curriculum**
  This class defines curriculum learning terms for MDP, which gradually increase task difficulty over the course of the training.
- **Reward**
  Defining the reward function is a key challenge in training a robot for doing an specific task.

```python
@configclass
class RewardsCfg:
    """Reward terms for the MDP."""

    # task terms
    end_effector_position_tracking = RewTerm(
        func=mdp.position_command_error,
        weight=-0.2,
        params={"asset_cfg": SceneEntityCfg("robot", body_names=["ee_link"]), "command_name": "ee_pose"},
    )
    end_effector_position_tracking_fine_grained = RewTerm(
        func=mdp.position_command_error_tanh,
        weight=0.1,
        params={"asset_cfg": SceneEntityCfg("robot", body_names=["ee_link"]), "std": 0.1, "command_name": "ee_pose"},
    )
    end_effector_orientation_tracking = RewTerm(
        func=mdp.orientation_command_error,
        weight=-0.1,
        params={"asset_cfg": SceneEntityCfg("robot", body_names=["ee_link"]), "command_name": "ee_pose"},
    )

    # action penalty
    action_rate = RewTerm(func=mdp.action_rate_l2, weight=-0.0001)
    joint_vel = RewTerm(
        func=mdp.joint_vel_l2,
        weight=-0.0001,
        params={"asset_cfg": SceneEntityCfg("robot")},
    )
```

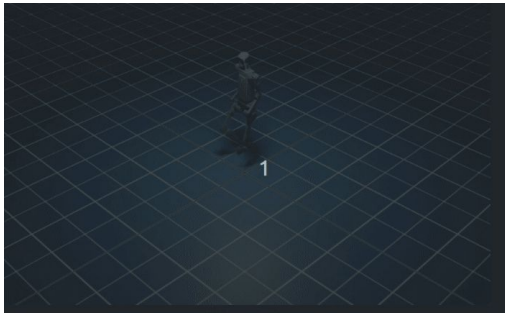
Code Time!!

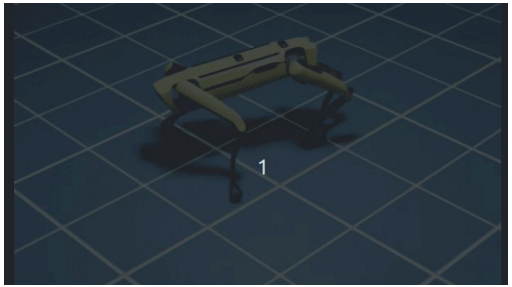
https://docs.google.com/document/d/1lRFEfLU0GrG55HAbYbGTz0HkSJ9aP7UvhC6crNbtZOg/edit?usp=sharing

Debugging Tips

When we are dealing with policy deployments from IsaacLab -> IsaacSim, we can have some troubles by doing this, so here are some tips in order to do some sanity checks.

- **Verify your policy:** Before trying to deploy it into real world/IsaacSim, verify it's correct behaviour by using **play.py** script.
- **Verify robot joints order:** If the policy works on IsaacLab you should verify the joint order from the policy corresponds to the order of your robot.
- **Verify robot joint initial positions:** Make sure the initial joint position of your robot in IsaacSim corresponds to the policy joint initial positions.
- **Verify the simulation environment:** If the robot matches exactly and still not working, you can verify the simulation parameters such as.
    - Time stepping
    - Physx
- **Observation and Action Tensor:** You need to make sure that the action and observation tensor structure, data passed and scale factor are correctly setted





Sim2Real Deployment

Quadruped Locomotion

Why learn in Simulation?

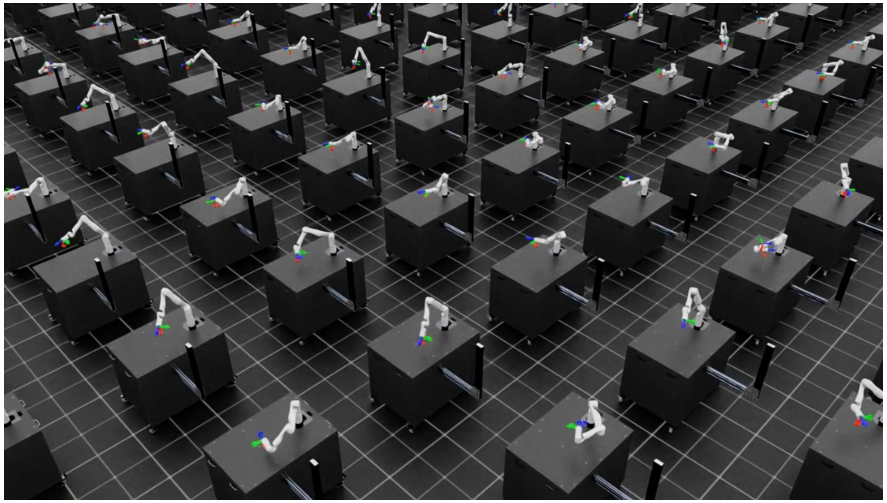
What we want:

Why learn in Simulation?

What we DON'T want:

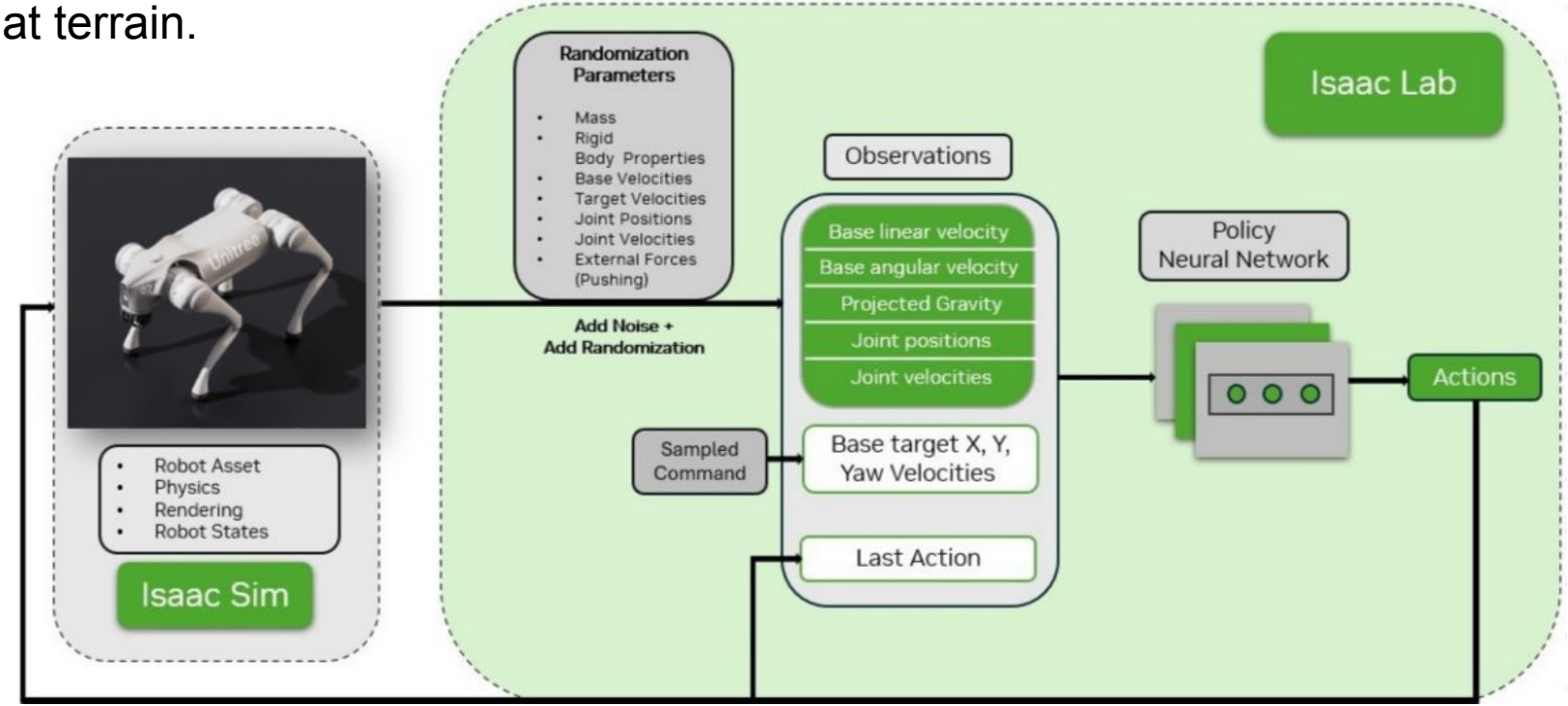
Why learn in Simulation?

- Safety & Risk Mitigation
- Massive Data Generation & Speed
- Cost-Effectiveness
- Rapid Iteration & Experimentation



The Goal

Train the GO2 robot to track target x, y, and yaw base velocities while walking on flat terrain.



Observation Space

Numerical vector representing robot’s current state and goal.

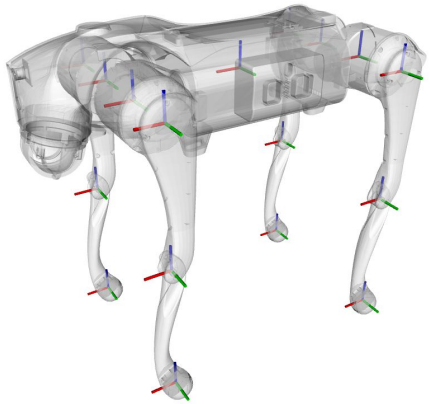
Input to the Neural Network

Proprioception (Positions, Velocities, IMU)

Randomized Target Values

Last Action - Smooth motion

Source: IMU, Encoders



Action Space

The numerical vector output by the policy neural network, representing the commands sent to the robot's actuators

The robot’s control levers

12 numbers that correspond to the target angular position for the robot’s 12 leg joints.

Relative offsets added to the default standing pose

Motor torques are used to try and reach the desired poses

Observation Space

```python
@configclass
class PolicyCfg(ObsGroup):
    """Observations for policy group."""

    # observation terms (order preserved)
    base_ang_vel = ObsTerm(func=mdp.base_ang_vel, scale=0.2, clip=(-100, 100), noise=Unoise(n_min=-0.2, n_max=0.2))
    projected_gravity = ObsTerm(func=mdp.projected_gravity, clip=(-100, 100), noise=Unoise(n_min=-0.05, n_max=0.05))
    velocity_commands = ObsTerm(
        func=mdp.generated_commands, clip=(-100, 100), params={"command_name": "base_velocity"}
    )
    joint_pos_rel = ObsTerm(func=mdp.joint_pos_rel, clip=(-100, 100), noise=Unoise(n_min=-0.01, n_max=0.01))
    joint_vel_rel = ObsTerm(
        func=mdp.joint_vel_rel, scale=0.05, clip=(-100, 100), noise=Unoise(n_min=-1.5, n_max=1.5)
    )
    last_action = ObsTerm(func=mdp.last_action, clip=(-100, 100))

    def __post_init__(self):
        # self.history_length = 5 # Example: Uncomment for history
        self.enable_corruption = True # Enable noise defined in terms
        self.concatenate_terms = True # Flatten observations into a single vector
```

```python
@configclass
class CriticCfg(ObsGroup):
    """Observations for critic group (privileged info)."""

    base_lin_vel = ObsTerm(func=mdp.base_lin_vel, clip=(-100, 100))
    base_ang_vel = ObsTerm(func=mdp.base_ang_vel, scale=0.2, clip=(-100, 100))
    projected_gravity = ObsTerm(func=mdp.projected_gravity, clip=(-100, 100))
    velocity_commands = ObsTerm(
        func=mdp.generated_commands, clip=(-100, 100), params={"command_name": "base_velocity"}
    )
    joint_pos_rel = ObsTerm(func=mdp.joint_pos_rel, clip=(-100, 100))
    joint_vel_rel = ObsTerm(func=mdp.joint_vel_rel, scale=0.05, clip=(-100, 100))
    joint_effort = ObsTerm(func=mdp.joint_effort, scale=0.01, clip=(-100, 100))
    last_action = ObsTerm(func=mdp.last_action, clip=(-100, 100))
```

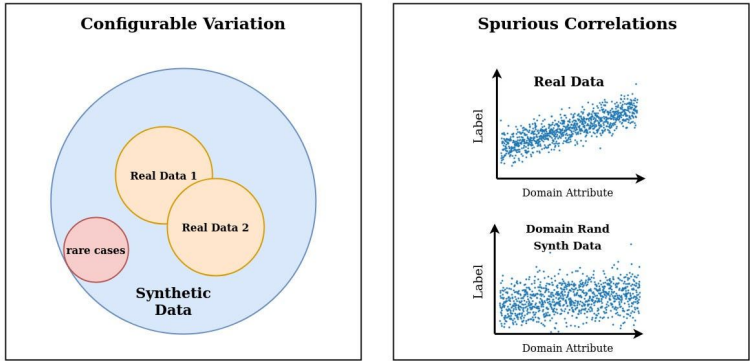
Action Space

```python
# -- Action Configuration --


@configclass
class ActionsCfg:
    """Action specifications for the MDP."""

    JointPositionAction = mdp.JointPositionActionCfg(
        asset_name="robot",
        joint_names=[".*"], # Apply to all joints
        scale=0.25, # Scale the output of the policy
        use_default_offset=True, # Action is added to the default joint angles
        clip={".*": (-100.0, 100.0)} # Clip action values (safety)
    )
```

Domain Randomization

Simulation physics ≠ Real-world physics (The "Sim-to-Real Gap"). Policies trained only on perfect simulation physics often fail on the real robot.

The solution: Domain Randomization, force the RL policy to learn a robust strategy that works across a range of conditions, making it less sensitive to simulation inaccuracies and more likely to succeed in reality.



Startup Randomization (cd /)

Randomization applied once when each individual parallel simulation environment is initialized at the beginning of the training process.

Each defined environment (num_envs) will get its own randomized friction and mass.

```python
@configclass
class EventCfg:
    """Configuration for events."""

    # startup
    physics_material = EventTerm(
        func=mdp.randomize_rigid_body_material,
        mode="startup",
        params={
            "asset_cfg": SceneEntityCfg("robot", body_names=".*"),
            "static_friction_range": (0.3, 1.2),
            "dynamic_friction_range": (0.3, 1.2),
            "restitution_range": (0.0, 0.15),
            "num_buckets": 64,
        },
    )

    add_base_mass = EventTerm(
        func=mdp.randomize_rigid_body_mass,
        mode="startup",
        params={
            "asset_cfg": SceneEntityCfg("robot", body_names="base"),
            "mass_distribution_params": (-1.0, 3.0),
            "operation": "add",
        },
    )
```

Reset Randomization (cd /)

Randomization applied every time an individual environment instance resets, which usually happens at the beginning of each new episode (after the robot falls or time limit is reached)

Every time an episode ends and a new one begins, the robot will be reset to a new random starting position, orientation, and joint state within the specified ranges.

```python
# reset
base_external_force_torque = EventTerm(
    func=mdp.apply_external_force_torque,
    mode="reset",
    params={
        "asset_cfg": SceneEntityCfg("robot", body_names="base"),
        "force_range": (0.0, 0.0), # Example: set non-zero for randomization
        "torque_range": (-0.0, 0.0), # Example: set non-zero for randomization
    },
)

reset_base = EventTerm(
    func=mdp.reset_root_state_uniform,
    mode="reset",
    params={
        "pose_range": {"x": (-0.5, 0.5), "y": (-0.5, 0.5), "yaw": (-3.14, 3.14)},
        "velocity_range": {
            "x": (0.0, 0.0),
            "y": (0.0, 0.0),
            "z": (0.0, 0.0),
            "roll": (0.0, 0.0),
            "pitch": (0.0, 0.0),
            "yaw": (0.0, 0.0),
        },
    },
)

reset_robot_joints = EventTerm(
    func=mdp.reset_joints_by_scale,
    mode="reset",
    params={
        "position_range": (1.0, 1.0), # Resets to default joint positions
        "velocity_range": (-1.0, 1.0), # Small random initial velocity
    },
)
```

Interval Randomization (cd /)

Randomization applied periodically during an ongoing episode within each environment instance.

The interval_range_s=(5.0, 10.0) means that for each of the 4096 robots, a random push will occur sometime between 5 and 10 seconds after the last push occurred in that specific environment.

```
# interval
push_robot = EventTerm(
    func=mdp.push_by_setting_velocity,
    mode="interval",
    interval_range_s=(5.0, 10.0),
    params={"velocity_range": {"x": (-0.5, 0.5), "y": (-0.5, 0.5)}},
)
```

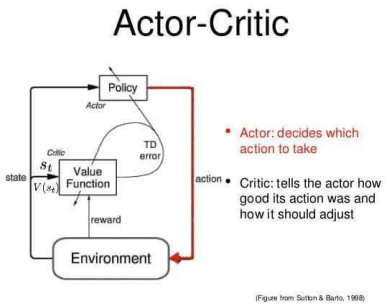
Network architecture and RL algorithm details

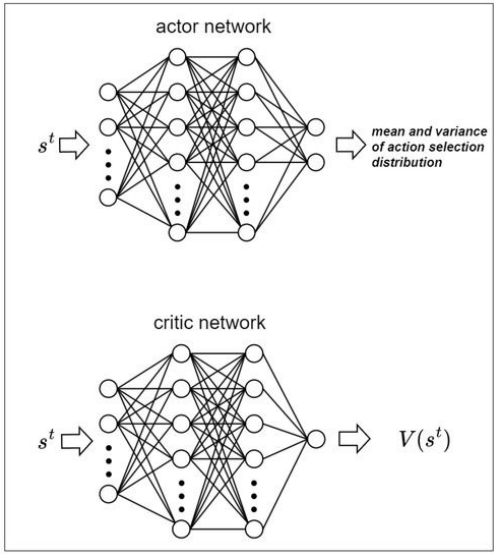
The goal is to train a Neural Network to map observations to actions.

An Actor-Critic architecture:

- Actor (Policy): Decides the action. Takes Observations -> Outputs Actions (12 joint positions). This is the part we deploy.

- Critic (Value): Estimates future rewards. Takes Observations -> Outputs a single Value (how good is this state?). Used only during training to guide the Actor.





MLP with 3 hidden layers [521, 256, 128]

Network architecture and RL algorithm details

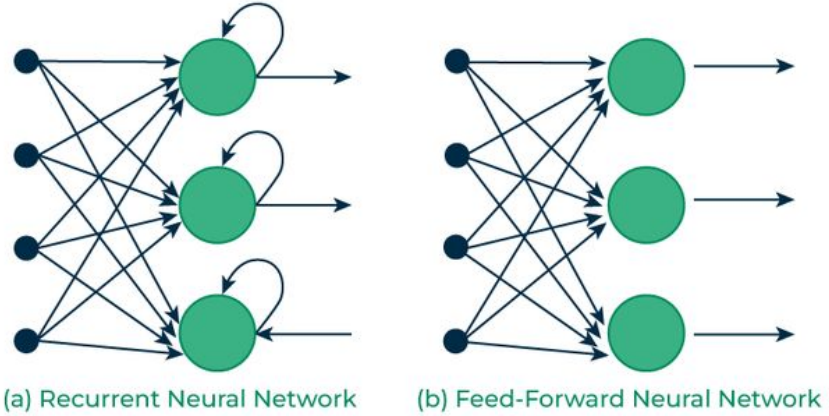
But what if our AI needs memory?? RNNs

Recurrent Neural Networks process sequential data, remembering information from previous timesteps.

These take the current observation and its own hidden state (memory) from the previous step as input.

Output the action and an updated hidden state for the next step.

Useful if key state information (like body velocity) isn't directly observed or is very noisy, requiring estimation over time.

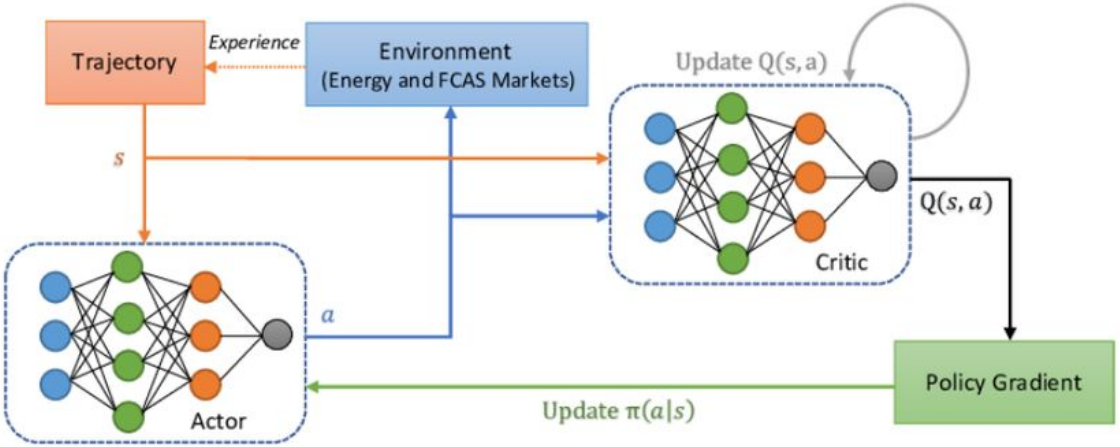


Network architecture and RL algorithm details

State of the art algorithms such as PPO aim to make the biggest possible improvement to the policy at each step without changing it so drastically that performance collapses.

Its actor-critic nature limits how much the new policy can differ from the old policy when making updates

PPO balances well sample efficiency and stability. It also works well with continuous action spaces.



RSL-RL - GPU-Optimized Training

RSL-RL or Robotics Systems Lab - Reinforcement learning is a PyTorch-based reinforcement learning library, developed by NVIDIA, providing optimized implementations of algorithms like PPO. It's the specific library used in unitree_rl_lab.

It was designed from the ground up to leverage GPU acceleration. This dramatically speeds up the training process, essentially for handling thousands of parallel environments.

It handles data collection, policy updates, logging and saving model checkpoints.

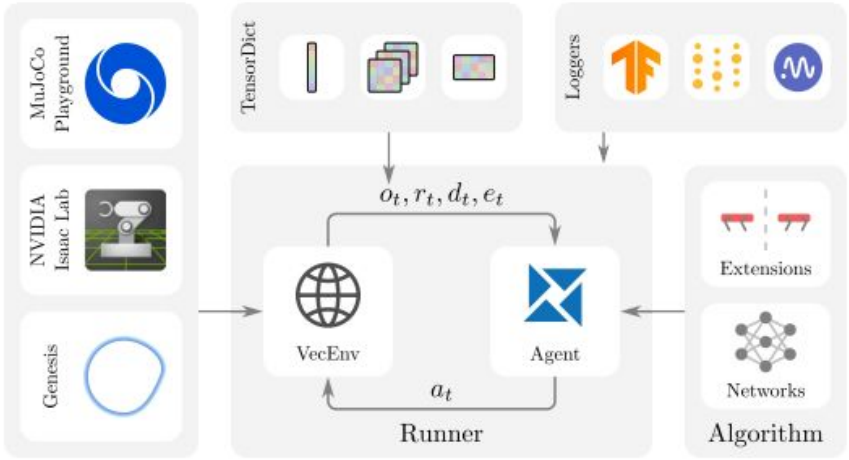


Figure 1: **Overview of the framework.** RSL-RL consists of three main components: `Runners`, `Algorithms`, and `Networks`, which can be easily modified independently. The framework comes with support for common logging choices and useful extensions for robotics.

RSL-RL PPO Configuration

This configuration file tells the RLS-RL trainer exactly how to set up and run the PPO algorithm.

Runner Settings: Controls the overall training loop (how much data per update, total training time, checkpoint frequency).

Policy Network: Defines the "brain's" structure – an MLP Actor-Critic with 3 hidden layers ([512, 256, 128] neurons each) using the ELU activation function. init_noise_std helps initial exploration.

PPO Algorithm:
clip_param=0.2: Key PPO parameter limiting policy changes per update (ensures stability).

learning_rate=1.0e-3: Initial step size for network updates (often adjusted automatically by schedule="adaptive").

gamma=0.99: How much the agent values future rewards vs. immediate ones (close to 1 means far-sighted).

num_learning_epochs=5, num_mini_batches=4: Control how the collected data is used for updates in each iteration.

```python
@configclass
class BasePPORunnerCfg(RslRlOnPolicyRunnerCfg):
    # --- Runner Settings ---
    num_steps_per_env = 24       # How much data to collect per env before update
    max_iterations = 50000       # Total number of training updates
    save_interval = 100          # Save checkpoint every 100 iterations
    experiment_name = ""         # Log directory name (set automatically)
    empirical_normalization = False # Use fixed observation normalization

    # --- Policy Network Architecture ---
    policy = RslRlPpoActorCriticCfg(
        init_noise_std=1.0,          # Initial action noise (for exploration)
        actor_hidden_dims=[512, 256, 128], # Actor MLP layers/neurons
        critic_hidden_dims=[512, 256, 128],# Critic MLP layers/neurons
        activation="elu",            # Neuron activation function (e.g., ELU)
    )

    # --- PPO Algorithm Hyperparameters ---
    algorithm = RslRlPpoAlgorithmCfg(
        value_loss_coef=1.0,         # Weight for Critic's loss
        use_clipped_value_loss=True,# PPO value clipping improvement
        clip_param=0.2,              # PPO policy clipping range (limits updates)
        entropy_coef=0.01,           # Encourages exploration slightly
        num_learning_epochs=5,       # How many times to iterate over collected data
        num_mini_batches=4,          # Split data into batches for updates
        learning_rate=1.0e-3,        # How big are the update steps (Adam optimizer)
        schedule="adaptive",         # Adjust learning rate based on KL divergence
        gamma=0.99,                  # Discount factor (future rewards importance)
        lam=0.95,                    # GAE lambda (advantage estimation)
        desired_kl=0.01,             # Target for adaptive learning rate schedule
        max_grad_norm=1.0,           # Clip gradients to prevent explosions
    )
```

Sim2Sim - Pre Real Deployment

The Problem: Even good simulators aren't perfect replicas of reality OR each other. A policy might accidentally learn to exploit specific quirks or inaccuracies of its training simulator (overfitting to the sim).

Sim-to-Sim Validation: Running the policy (trained in Sim A) inside a different high-fidelity simulator (Sim B, like MuJoCo) before going to the real robot.

Intermediate Safety Check: Identifies problems without risking the physical robot. Cheaper and safer than discovering failures on hardware.

Tests Robustness: Does the policy work even when the underlying physics calculations (especially contacts, friction) are slightly different?

Increases Confidence: Success in multiple simulators gives stronger evidence that the policy has learned a generalizable skill, increasing confidence for real-world deployment.

Sim2Sim - Isaac Sim vs MuJoCo

	MuJoCo
Engine: Omniverse Platform + PhysX Physics.	Engine: MuJoCo Physics.
Key Strength: Massive GPU Parallelization. Ideal for the huge data demands of RL training. Runs thousands of environments simultaneously.	Key Strength: Known for fast, stable, and accurate contact physics. Widely respected and used in robotics control research (often CPU-focused).
Focus: High visual fidelity, large-scale simulation, optimized for generating training data fast.	Focus: Precise dynamic simulation, especially interactions and contacts, excellent for validating control policies.

Sim2Sim - How2Validate

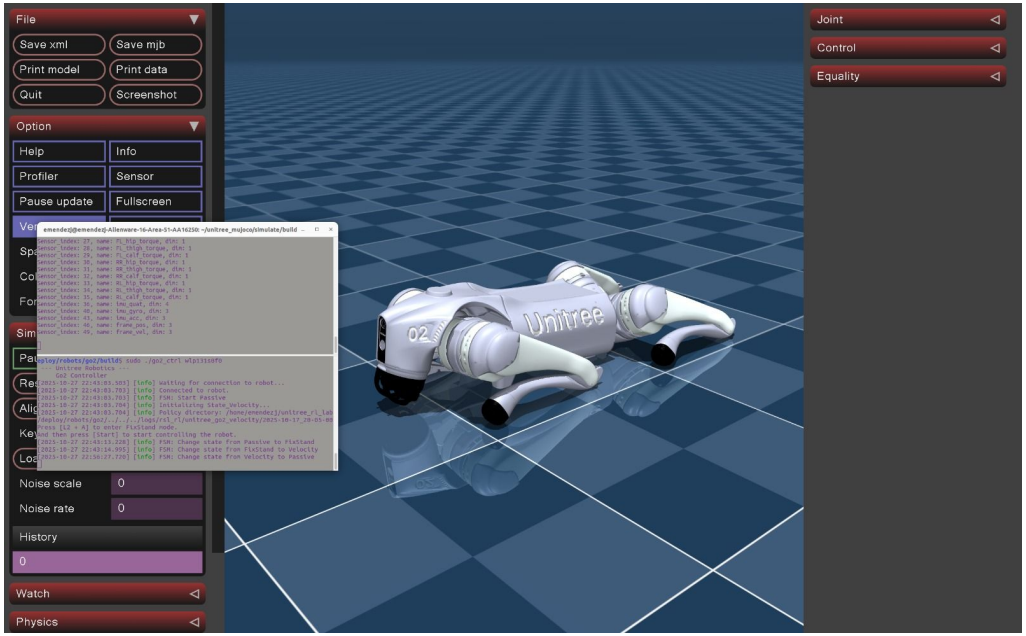
The Execution:

Run MuJoCo Sim (Terminal 1): sudo ./unitree_mujoco -> This is the "virtual body," waiting via DDS.

Run C++ Controller (Terminal 2): sudo ./go2_ctrl <your_interface> -> This is the "brain," loading the ONNX policy and connecting via DDS.

Activate Policy: Use the gamepad (L2+Up, R1+X) -> go2_ctrl sends commands via DDS -> MuJoCo robot moves.

Key Insight: The C++ controller (go2_ctrl) is the same one used for the real robot! It communicates via the standard Unitree DDS interface, unaware if it's talking to MuJoCo or the actual Go2.

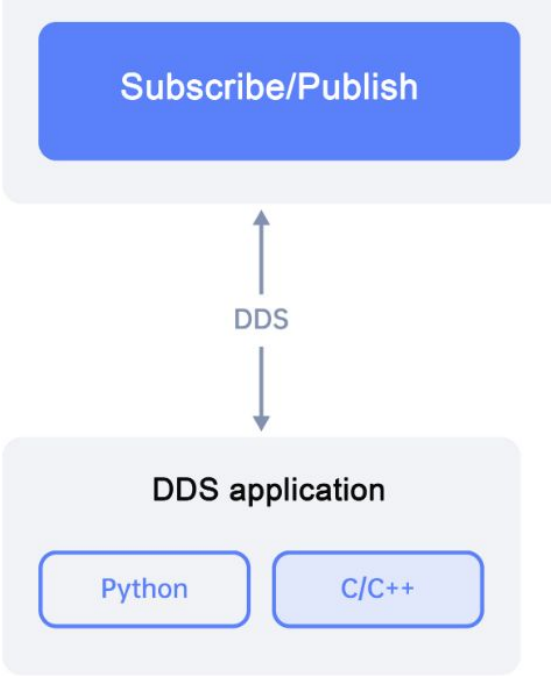


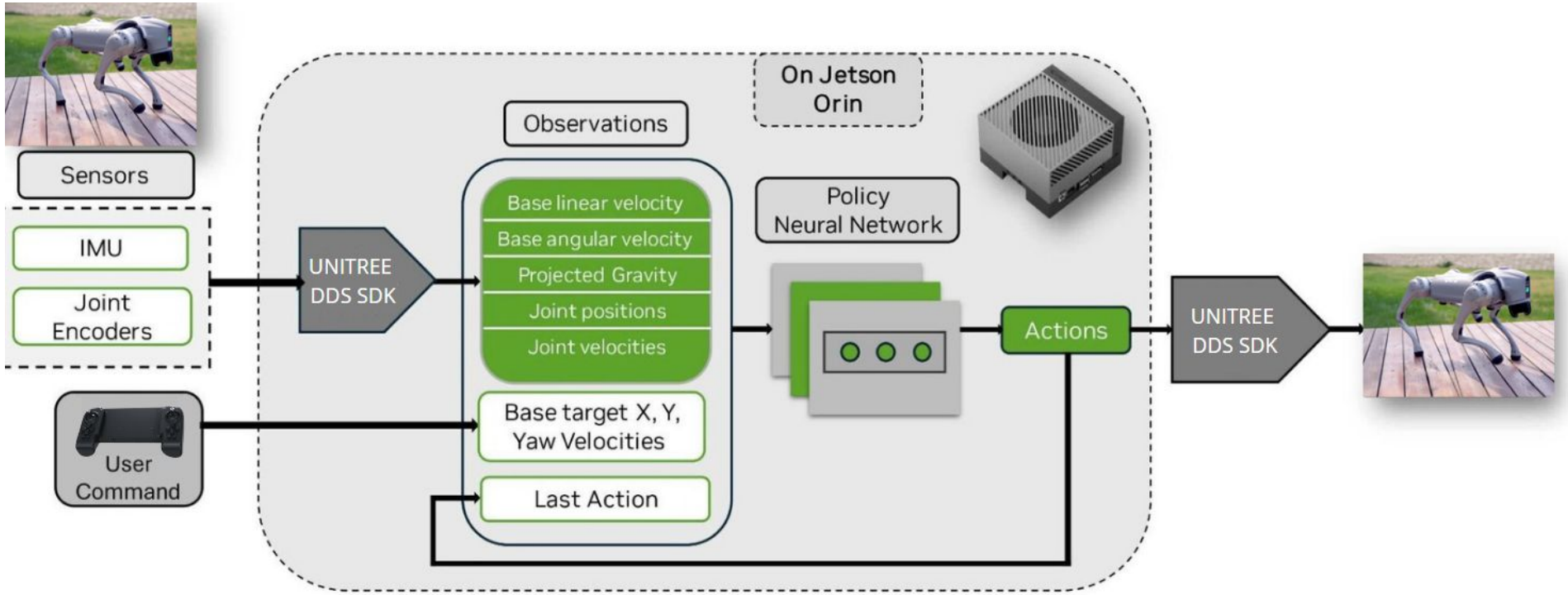
DDS SDK

DDS = Data Distribution Service: A standardized middleware protocol for real-time, reliable data sharing between applications.

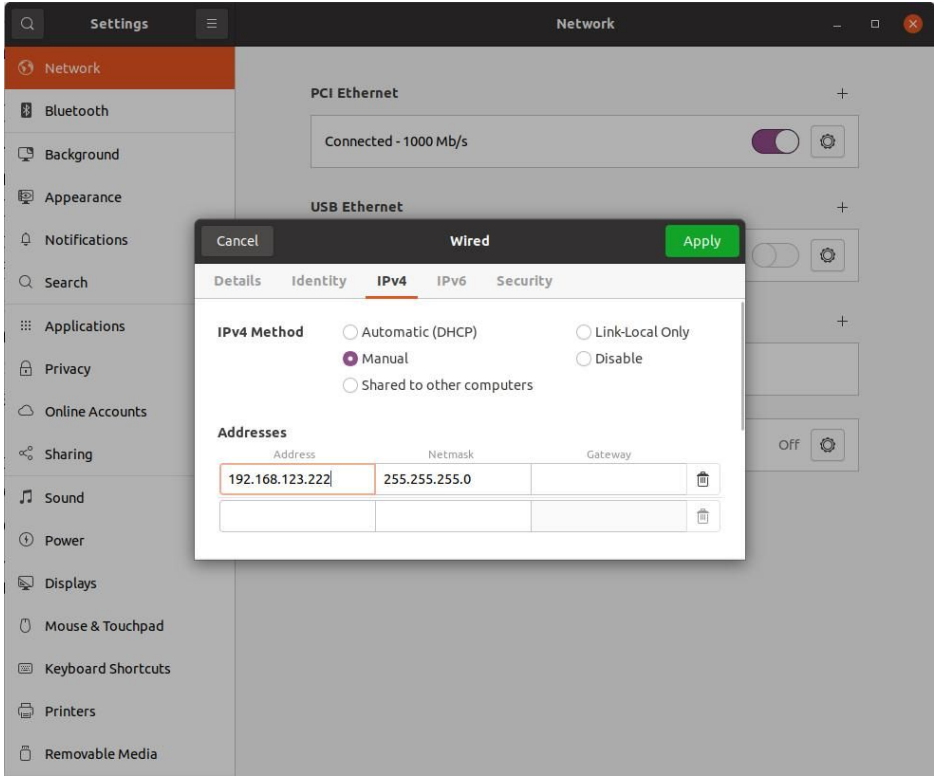
Unitree SDK: Provides C++ libraries (libddsc.so, libddscxx.so) that wrap the DDS communication details.

Our go2_ctrl subscribes to rt/lowstate (sensor data) and publish to rt/lowcmd (motor commands)




Sensors
IMU
Joint Encoders
User Command
UNITREE DDS SDK
On Jetson Orin
Observations
Base linear velocity
Base angular velocity
Projected Gravity
Joint positions
Joint velocities
Base target X, Y, Yaw Velocities
Last Action
Policy Neural Network
Actions
UNITREE DDS SDK

Sim2Real - Network ID




```
liang@xiaoliangstd:~$ ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:e3:51:19:87  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eno1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.4.135  netmask 255.255.254.0  broadcast 192.168.5.255
        inet6 fe80::ee85:8ac4:f415:12f1  prefixlen 64  scopeid 0x20<link>
        ether 04:42:1a:96:33:79  txqueuelen 1000  (Ethernet)
        RX packets 1896133  bytes 1518864737 (1.5 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1164643  bytes 199679779 (199.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 16  memory 0xa1200000-a1220000

enxf8e43b808e06: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.123.222  netmask 255.255.255.0  broadcast 192.168.123.255
        inet6 fe80::2326:9805:e699:dd24  prefixlen 64  scopeid 0x20<link>
        ether f8:e4:3b:80:8e:06  txqueuelen 1000  (Ethernet)
        RX packets 8153  bytes 8827164 (8.8 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8384  bytes 2906296 (2.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Sim2Real - Deployment

First we run an unitree example to deactivate the main process (commercial controller)

```
~/unitree_sdk2/build/bin$ sudo ./go2_stand_example enx0c37969f434c
```

Then we run our policy driven controller. Network identification is crucial to enable the rt/lowstate publisher

```
:~/unitree_rl_lab/deploy/robots/go2/build$ sudo ./go2_ctrl --network enx0c37969f434c
```

Bibliography

https://isaac-sim.github.io/IsaacLab/main/index.html

Lonza, A. (2019). Reinforcement Learning Algorithms with Python. Packt Publishing Limited. 2019.