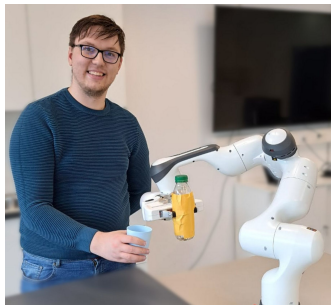
Technical Tutorial: Docker, ROS, Gazebo, Mujoco

tuwien.at/etit/ict/asl

Robot Learning 384.195
01.12.2023

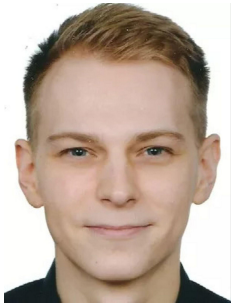
Prof Dongheui Lee

Daniel Sliwowski

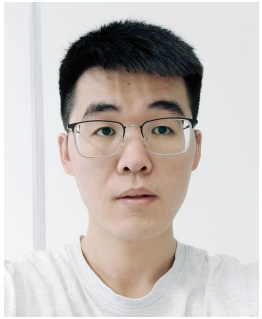


Esteve Valls Mascaro

Johannes Heidersberger



Yashuai Yan



Schedule

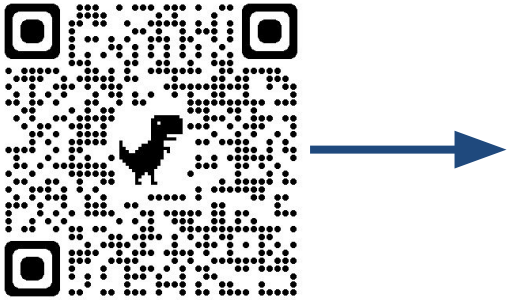
Nr	Date	Contents	Classroom
1	13.10	Introduction	EI 8 Pötzl HS
2	20.10	Learning from Demonstrations	EI 8 Pötzl HS
3	27.10	Deep Learning 1 (Pytorch, Perceptron, MLP, CNN)	EI 8 Pötzl HS
4	3.11	Deep Learning 2 (Attention, Transformer)	EI 8 Pötzl HS
	10.11	Cancelled (due to a business trip)	Seminarraum 384
5	17.11	Reinforcement Learning 1 (Basics, Actor-Critic)	EI 8 Pötzl HS
6	24.11	Reinforcement Learning 2 (PPO, DDPG)	EI 8 Pötzl HS
7	1.12	Projects, Technical tutorials (ROS, Docker, Colab, Simulator, OpenAI Gym)	EI 8 Pötzl HS
	8.12	Holiday (No lecture date)	
8	15.12	Student Projects (with TA consultation)	CD0208
	22.12 - 5.1	Holiday (No lecture period, Xmas)	
9	12.1	Student Projects (with TA consultation)	CD0208
10	19.1	Student Projects (with TA consultation)	CD0208
11	26.1	Project Report Submission	
	2.2	Final Project Presentation	EI 8 Pötzl HS

We are here

Project Lab

Project Lab

DEEP LEARNING PROJECT



Deep Learning for Robot Action Classification

Course: Robot Learning E384.195

Project Overview

In this project, Master's students will dive into the fascinating world of deep learning and apply their knowledge to a real-world problem: classifying robot actions. The primary objective of this project is to develop a deep learning model that can accurately classify robot actions based on video footage of the robot and the corresponding proprioceptive data. Moreover, students will also learn how to teach and reproduce skills based on human demonstrations. For that, one will use Franka Emika Research robots provided by the Autonomous Systems Lab (ASL) for data collection and reproduction.

Index

1. Competition
2. Docker
   a. What is docker?
   b. How to use it for the project?
3. ROS
   a. What is ROS?
   b. ROS Nodes
   c. ROS Topics
   d. ROS Launch
   e. ROS Bag
   f. catkin Build System
   g. Creating catkin Packages
   h. Gazebo
   i. ROS tutorials

Competition

Splits

Inside the splits directory one can find variosu relevant files:

- **splits.json**. This file is used to separate the data from the *data* folder provided in train, val and test. It is important to use the provided splits as the evaluation of your model will be done accordingly.
- **train/val/test_students**.json. These files are used to generate the predictions that will be uploaded to the evaluation platform. The file contains, per each set, a dictionary with the relative path of the file as keys. Each value of the dictionary is composed by :
  - "id". Refers to the ID of the robot execution
  - "action". Refers to the action of the ID. As the test set is not provided, we filled in with -1.
  - "hl_actions". Refers to the HL action of the ID. As the test set is not provided, we filled in with -1.

The students need to train a model according to the splits generated, and predict the *action* and *hl_actions* per execution (ID) in the test set. Then, they need to generate a *test_pred.json* file by filling the *action* and *hl_actions* with their predictions. Finally, one need to use the code provided **dump_predictions** on that *test_pred.json* file, which will generate two .csv files which will be uploaded to the platform to evaluate the performance of the model.

RL: Deep Learning for Robot Action Classification

Classification of the robot actions based on video footage of the robot and the corresponding proprioceptive data

Overview Data Discussion Leaderboard Rules Team Submissions Host

Overview

In this project, Master's students will dive into the fascinating world of deep learning and apply their knowledge to a real-world problem: classifying robot actions. The primary objective of this project is to develop a deep learning model that can accurately classify robot actions based on video footage of the robot and the corresponding proprioceptive data.

Start
6 days ago

Close
2 months to go

Competition Host
Esteve Valls

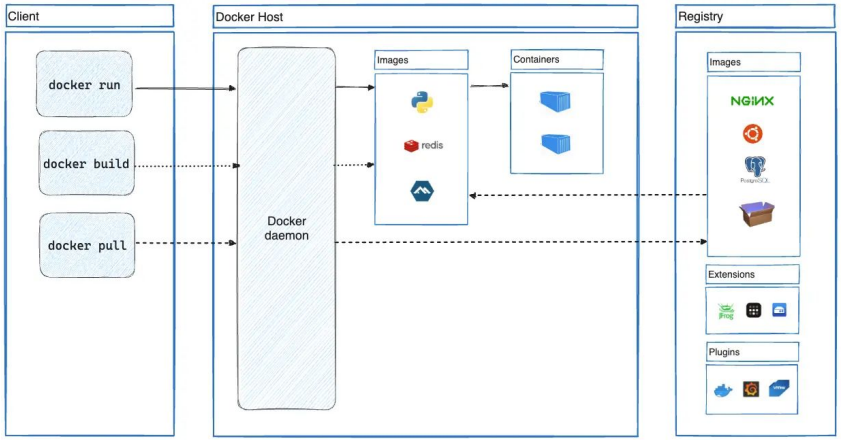
Prizes & Awards
Kudos
Does not award Points or Medals

Participation
0 Competitors
0 Teams
0 Entries

Tags

What is docker?

- Docker provides the ability to package and run an application in a loosely isolated environment called a container.
- Containers are lightweight and contain everything needed to run the application, so you don't need to rely on what's installed on the host.
- An image is a read-only template with instructions for creating a Docker container.



Installing docker

Linux

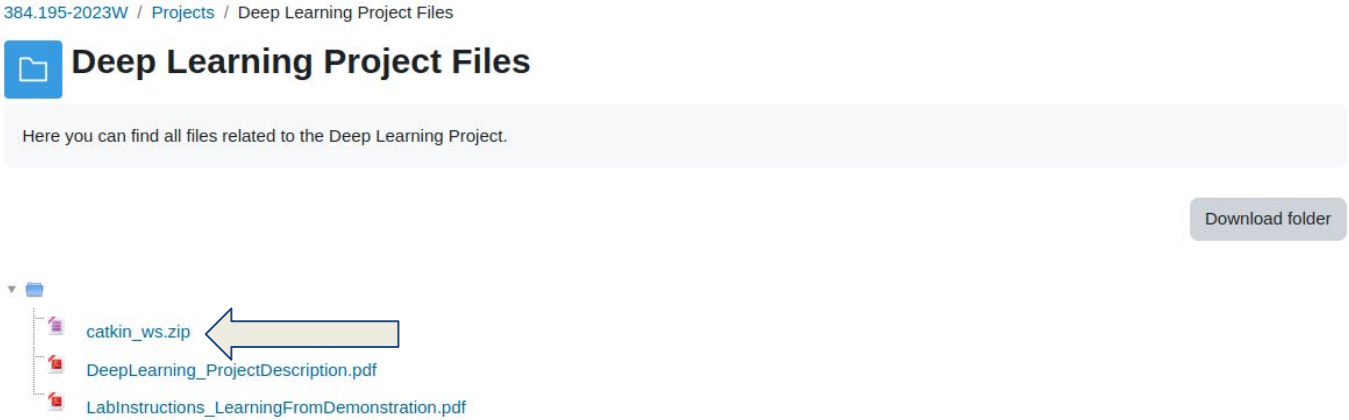
- More straight forward
- Does not require any additional software to run GUIs in the containers
- Nvidia GPUs can be used inside docker with NVIDIA Container Toolkit
- Installation instructions can be found here
- Post installations steps to make our lives easier: here

Windows

- A bit more complicated as it requires to enable WSL
- Additional software needs to be installed to run GUIs in the container
- Nvidia GPUs can be used inside docker with WSL 2 GPU Support
- Installation instructions can be found here
- Tutorial for getting the X11 running so GUIs can be used inside the container: here (You can skip creating an boulding the docker file and use our image)

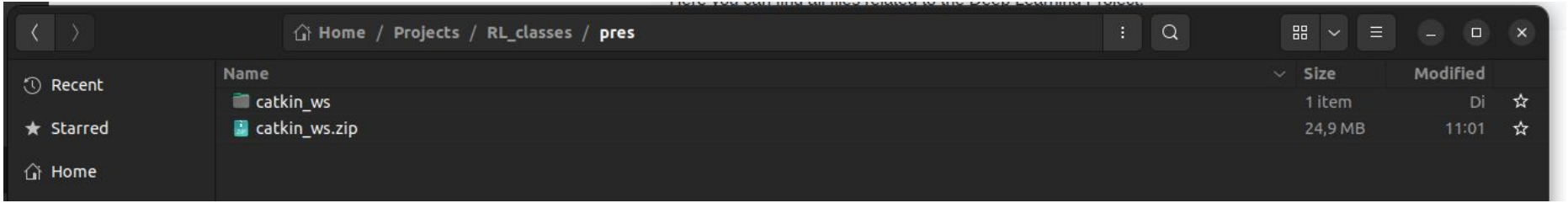
Installing our image

1. Download the compressed catkin workspace from TUWEL



Installing our image

2. Extract the catkin workspace



3. Open an terminal and go to the extracted workspace, and grant X11 permissions

```
> cd /home/dsliwowski/Projects/RL_classes/pres/catkin_ws
> ls
  src
> xhost +local:*
  non-network local connections being added to access control list
```

Installing our image

4. Start the docker container (it should pull it automatically)

```
> docker run -it -e DISPLAY=$DISPLAY -v /tmp/.X11-unix/:/tmp/.X11-unix/ -v $(pwd):/home/catkin_ws
asltuwien/franka_base
```

Testing the image

1. Inside the container go to /home/catkin_ws

```
> cd /home/catkin_ws
```

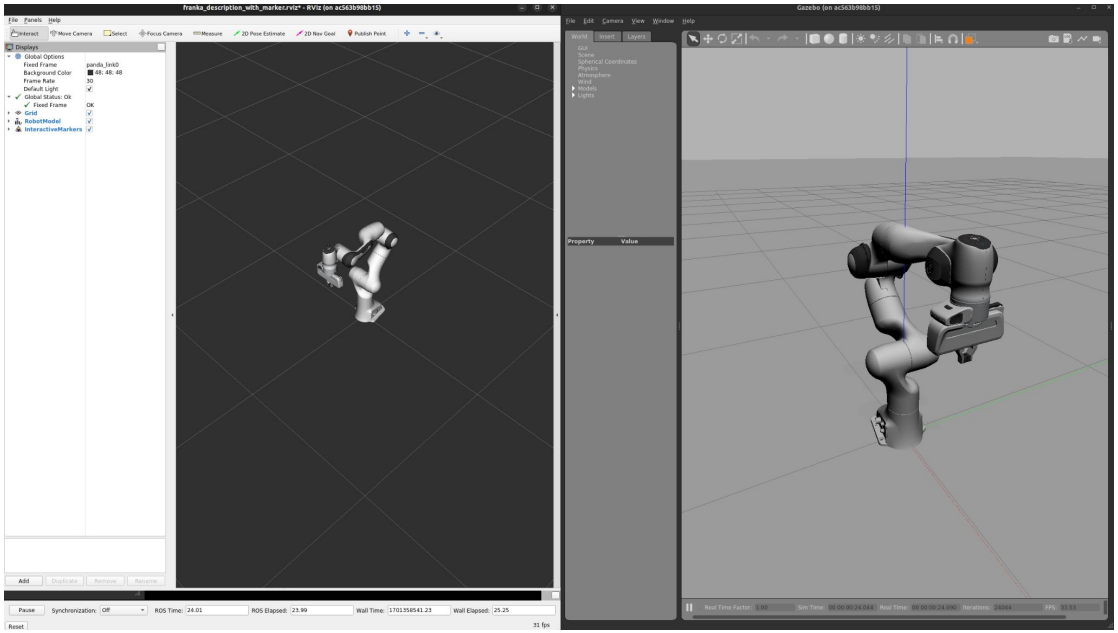
2. Build and source the workspace

```
> catkin_make -DCMAKE_BUILD_TYPE=Release -DFranka_DIR:PATH=/libfranka/build
> source devel/setup.bash
```

Testing the image

3. Launch the simulator

```
> roslaunch franka_gazebo panda.launch # By default the cartesian impedance controller will be launched
```



Connecting new terminal

1. Get container ID

```
> docker ps
CONTAINER ID IMAGE                  COMMAND                  CREATED       STATUS       PORTS  NAMES
ac563b98bb15 asltuwien/franka_base "/ros_entrypoint.sh …" 6 minutes ago Up 6 minutes        laughing_lumiere
```

2. Connect the terminal

```
> docker exec -it ac563b98bb15 /bin/bash
```

3. Source the base ros packages in the new terminal

```
> source ros_entrypoint.sh
```

WARNING

When starting the container we have mounted the catkin directory on your local drive into the catkin directory in the container (the parameter -v $(pwd):/home/catkin_ws) this means that whatever eddits you do in the catkin directory on the host machine, they will also be applied in the container (and vice versa). Moreover, as the edited files are stored on the host machine, they will be persistent, and not removed when you close the container. Be sure that any files you create/modify in the container are **inside the catkin_ws** directory, otherwise they will be removed upon closing the container.

What is ROS?

ROS (Robot Operating System) is a robotics middleware suite

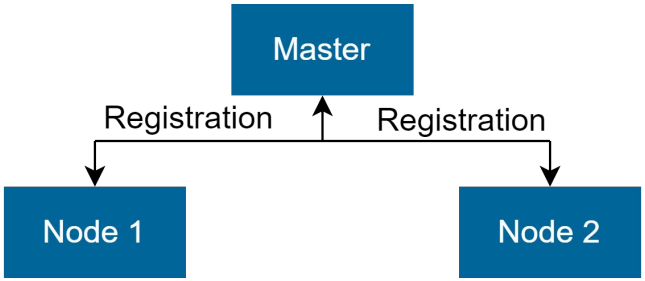
- Peer-to-peer:
  ROS system consists of connected nodes exchanging messages
- Distributed:
  Programs can run on multiple computers and can communicate over the network
- Multi-lingual:
  Programs can be written in any language for which a client library has been written, e.g. C++, Python, Java, MATLAB,...
- Lightweight:
  Stand-alone packages are wrapped in a thin ROS layer
- Free and open-source:
  Many open-source packages available

ROS Nodes

ROS master manages communication between nodes

ROS nodes:

- Single-purpose executable program
- Individually compiled, executed and managed
- Organized in packages



Start a master with

```
> roscore
```

Run a node with

```
> rosrun package_name node_name
```

List active nodes with

```
> rosnode list
```

Show information about a node with

```
> rosnode info node_name
```

Additional information:
http://wiki.ros.org/Nodes

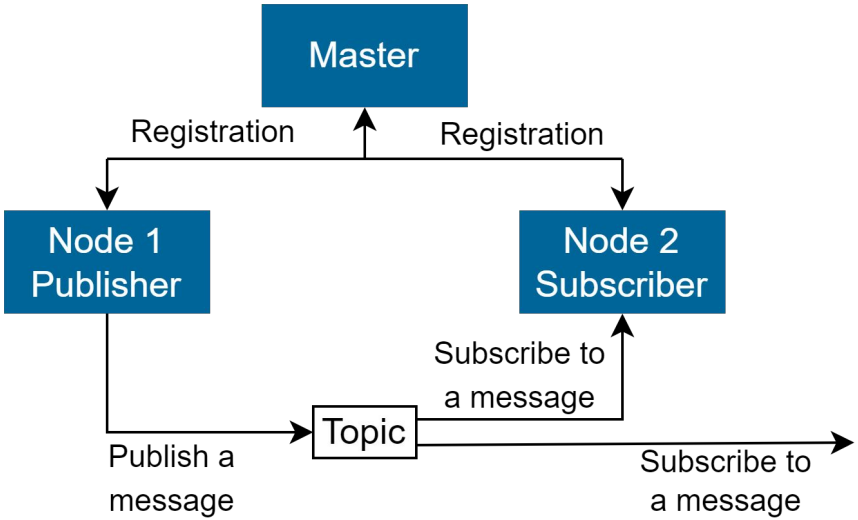
ROS Topics

Topics are named buses with a defined type over which messages are streamed

Nodes can publish and subscribe to topics

Publisher/subscriber architecture:
1-to-many broadcasting

Messages are data structures with defined types used for inter-node communication, e.g. boolean, float, PoseStamped



List active topics with

```
> rostopic list
```

Show information about a topic with

```
> rostopic info topic_name
```

Subscribe and print messages from a topic with

```
> rostopic echo topic_name
```

Additional information:
http://wiki.ros.org/Topics
http://wiki.ros.org/Messages

ROS Launch

Launch is a tool for launching multiple nodes and setting parameters

If no master is currently running, a roscore is started

Start a launch file

```
> roslaunch package_name file_name.launch
```

Simplified example of a launch file

```xml
<?xml version="1.0"?>
<launch>
    <!-- Arguments -->
    <arg name="arg1_name" default="default_value"/>
    <arg name="arg2_name" default="default_value"/>

    <!-- Start a node -->
    <node name="node_name" pkg="your_package" type="your_node" output="screen">
        <!-- Pass parameter to the node if needed -->
        <param name="node_param" value="$(arg arg1_name)"/>
    </node>

    <!-- Start another launch file -->
    <include file="$(find your_package)/launch/launch_file.launch">
        <!-- Pass argument to the included launch file if needed -->
        <arg name="included_file_arg" value="$(arg arg2_name)"/>
    </include>
</launch>
```

Additional information:
http://wiki.ros.org/roslaunch

ROS Bags

Bags are the primary tool of data logging in ROS

Bags subscribe to one or more topics and store the serialized message data in a file

Record a bag file of the specified topics with

```
> rosbag record topic_name1 topic_name2
```

Show summary of a bag file with

```
> rosbag info bag_name.bag
```

Playback (i.e. publish) a bag file with

```
> rosbag play bag_name.bag
```

Additional information:
http://wiki.ros.org/rosbag

catkin Build System

catkin is the ROS build system to generate executables, libraries, and interfaces

The catkin workspace contains the following directories:

- src (work here):
  Contains source code. This is where code is created, cloned, edited for the packages to build.
- build (don't touch):
  Contains cache information and other intermediate files from the CMake build process.
- devel (don't touch):
  Contains build targets.

Additional information:
http://wiki.ros.org/catkin

Build a package with

```
> catkin_make [package_name]
```

After building update your environment with

```
> source devel/setup.bash
```

Creating catkin Packages

When creating a new catkin package the structure of the catkin workspace should be maintained

Created files:

- package.xml:
  Defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages
- CMakeLists.txt:
  Defines how to build the software packages, including dependencies, source files, and targets for the catkin build system

Additional information:
http://wiki.ros.org/ROS/Tutorials/CreatingPackage

Create a package with

```
> catkin_create_pkg package_name depend_pkg
```

After creating the package built it with

```
> catkin_make [package_name]
```

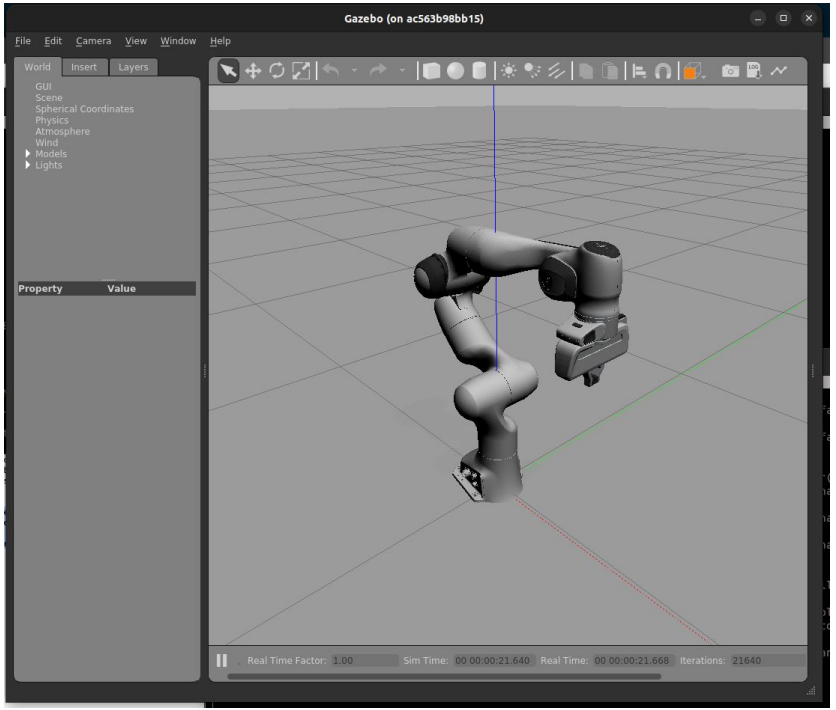
Typical structure of catkin workspace

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

Gazebo

It is beneficial to check a safety of your algorithms in a simulator before deploying them on a real robot.

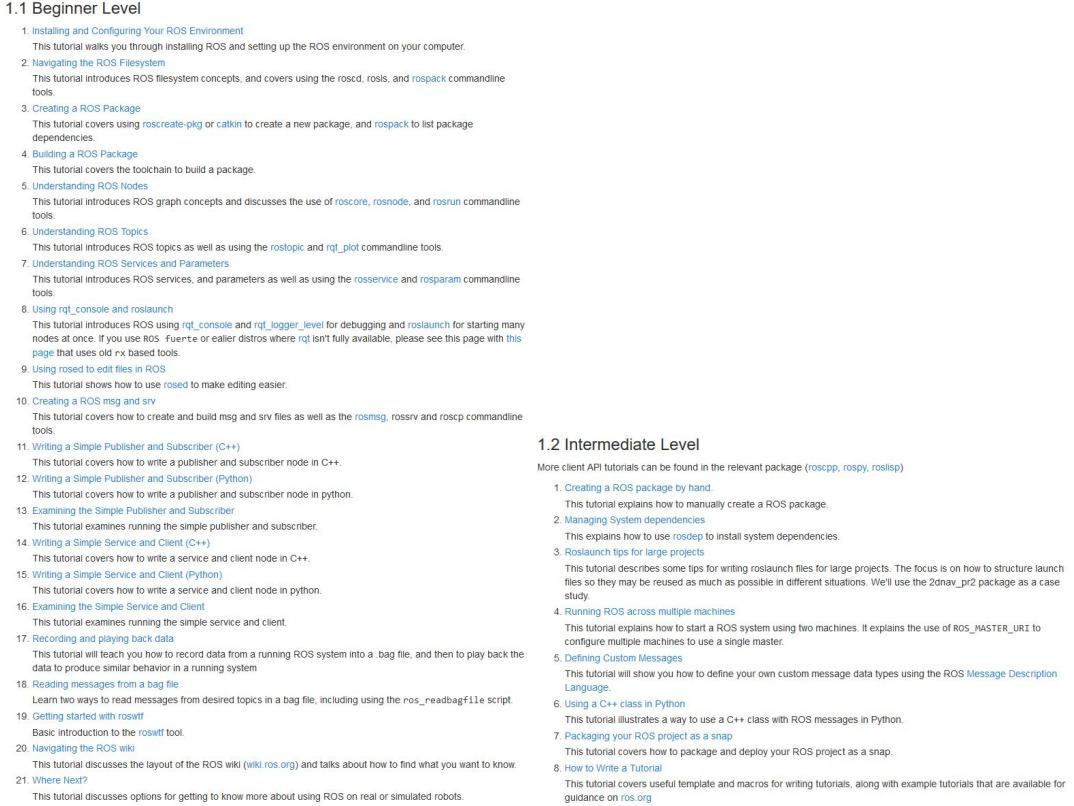
Gazebo is a commonly used open-source simulator for robots and Franka Emika already provides ready-to-use simulation environments files.



Additional information:
https://gazebosim.org/home

ROS tutorials

- ROS Cheat Sheet: https://www.clearpathrobotics.com/ros-robot-op erating-system-cheat-sheet/
- ROS Best Practices: https://github.com/leggedrobotics/ros_best_practices/wiki
- ROS Package Template: https://github.com/leggedrobotics/ros_best_practices/tree/master/ros_package_template
- ROS Wiki: http://wiki.ros.org/
- ROS Installation: http://wiki.ros.org/ROS/Installation
- ROS Tutorials: http://wiki.ros.org/ROS/Tutorials
- Franka Control Interface: https://frankaemika.github.io/docs/index.html
- Gazebo Tutorials: https://classic.gazebosim.org/tutorials



1.1 Beginner Level

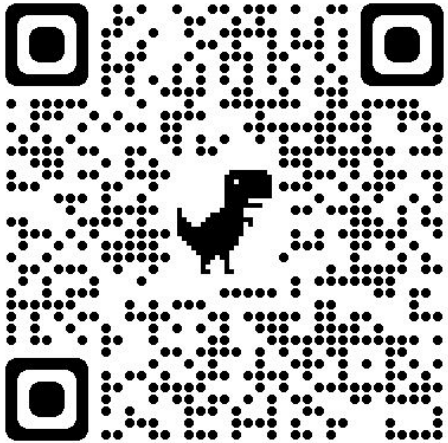
1. Installing and Configuring Your ROS Environment
   This tutorial walks you through installing ROS and setting up the ROS environment on your computer.
2. Navigating the ROS Filesystem
   This tutorial introduces ROS filesystem concepts, and covers using the roscd, rosls, and rospack commandline tools.
3. Creating a ROS Package
   This tutorial covers using roscreate-pkg or catkin to create a new package, and rospack to list package dependencies.
4. Building a ROS Package
   This tutorial covers the toolchain to build a package.
5. Understanding ROS Nodes
   This tutorial introduces ROS graph concepts and discusses the use of roscore, rosnode, and rosrun commandline tools.
6. Understanding ROS Topics
   This tutorial introduces ROS topics as well as using the rostopic and rqt_plot commandline tools.
7. Understanding ROS Services and Parameters
   This tutorial introduces ROS services, and parameters as well as using the rosservice and rosparam commandline tools.
8. Using rqt_console and roslaunch
   This tutorial introduces ROS using rqt_console and rqt_logger_level for debugging and roslaunch for starting many nodes at once. If you use `ROS_fuerte` or eailier distros where rqt isn't fully available, please see this page with this page that uses old `rx` based tools.
9. Using rosed to edit files in ROS
   This tutorial shows how to use rosed to make editing easier.
10. Creating a ROS msg and srv
    This tutorial covers how to create and build msg and srv files as well as the rosmsg, rossrv and roscp commandline tools.
11. Writing a Simple Publisher and Subscriber (C++)
    This tutorial covers how to write a publisher and subscriber node in C++.
12. Writing a Simple Publisher and Subscriber (Python)
    This tutorial covers how to write a publisher and subscriber node in python.
13. Examining the Simple Publisher and Subscriber
    This tutorial examines running the simple publisher and subscriber.
14. Writing a Simple Service and Client (C++)
    This tutorial covers how to write a service and client node in C++.
15. Writing a Simple Service and Client (Python)
    This tutorial covers how to write a service and client node in python.
16. Examining the Simple Service and Client
    This tutorial examines running the simple service and client.
17. Recording and playing back data
    This tutorial will teach you how to record data from a running ROS system into a .bag file, and then to play back the data to produce similar behavior in a running system
18. Reading messages from a bag file
    Learn two ways to read messages from desired topics in a bag file, including using the `ros_readbagfile` script.
19. Getting started with roswtf
    Basic introduction to the roswtf tool.
20. Navigating the ROS wiki
    This tutorial discusses the layout of the ROS wiki (wiki.ros.org) and talks about how to find what you want to know.
21. Where Next?
    This tutorial discusses options for getting to know more about using ROS on real or simulated robots.

1.2 Intermediate Level

More client API tutorials can be found in the relevant package (roscpp, rospy, roslisp)

1. Creating a ROS package by hand.
   This tutorial explains how to manually create a ROS package.
2. Managing System dependencies
   This explains how to use rosdep to install system dependencies.
3. Roslaunch tips for large projects
   This tutorial describes some tips for writing roslaunch files for large projects. The focus is on how to structure launch files so they may be reused as much as possible in different situations. We'll use the 2dnav_pr2 package as a case study.
4. Running ROS across multiple machines
   This tutorial explains how to start a ROS system using two machines. It explains the use of `ROS_MASTER_URI` to configure multiple machines to use a single master.
5. Defining Custom Messages
   This tutorial will show you how to define your own custom message data types using the ROS Message Description Language.
6. Using a C++ class in Python
   This tutorial illustrates a way to use a C++ class with ROS messages in Python.
7. Packaging your ROS project as a snap
   This tutorial covers how to package and deploy your ROS project as a snap.
8. How to Write a Tutorial
   This tutorial covers useful template and macros for writing tutorials, along with example tutorials that are available for guidance on ros.org

REINFORCEMENT LEARNING



Deep reinforcement learning for locomotion

Course: Robot Learning E384.195

Project Overview

This project aims to employ reinforcement learning (RL) techniques to develop control algorithms for a quadruped robot, enabling it to achieve efficient and adaptable locomotion patterns **(e.g. walking, running, jumping and etc.).** The project will involve implementing RL methods to learn the control policy, showcasing the potential of RL in advancing quadruped robot locomotion.

Index

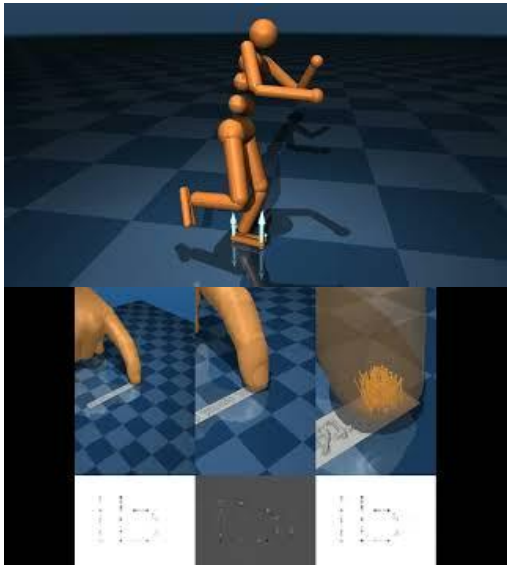
1. Mujoco – what it is?
2. OpenAI Gym – how to use it for the project?
3. Project framework

What is Mujoco?

MuJoCo stands for **Mu**lti-**Jo**int dynamics with **Co**ntact.

What you can do with MuJoCo

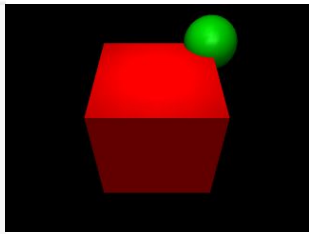
Sensors in MuJoCo



What is Mujoco?

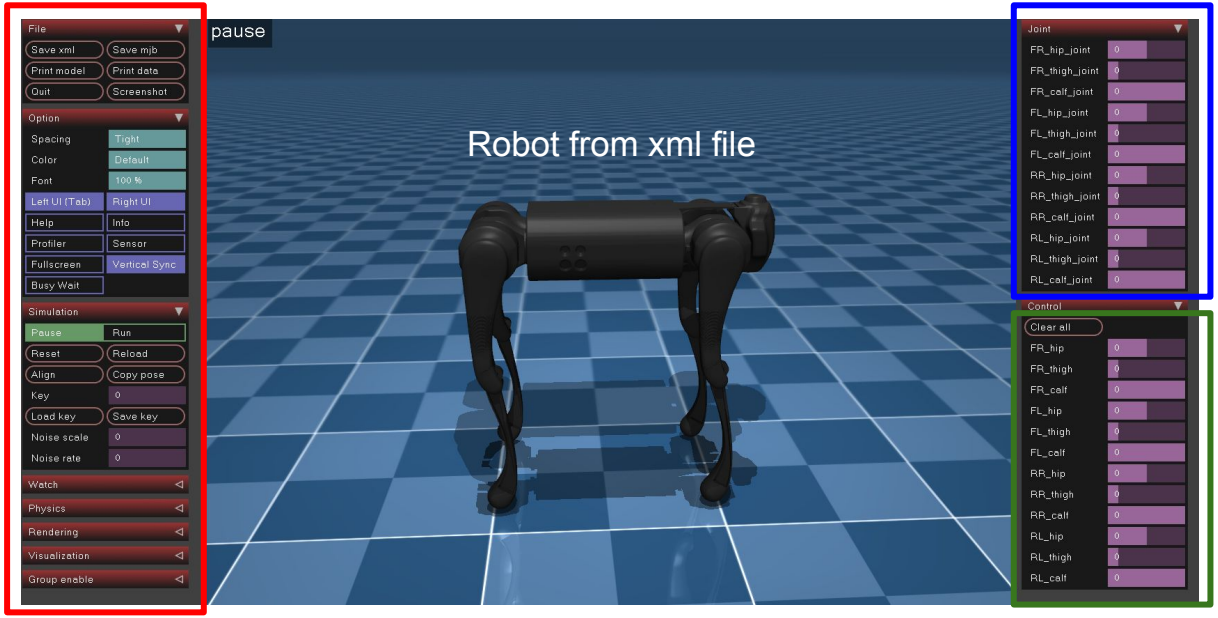
In MuJoCo, models are defined in XML file

```
xml = """
<mujoco>
  <worldbody>
    <light name="top" pos="0 0 1"/>
    <geom name="red_box" type="box" size=".2 .2 .2" rgba="1 0 0 1"/>
    <geom name="green_sphere" pos=".2 .2 .2" size=".1" rgba="0 1 0 1"/>
  </worldbody>
</mujoco>
"""
model = mujoco.MjModel.from_xml_string(xml)
data = mujoco.MjData(model)
renderer = mujoco.Renderer(model)

mujoco.mj_forward(model, data)
renderer.update_scene(data)

media.show_image(renderer.render())
```



What is Mujoco?

Control Panel



Joint Values

Joint Control

Install Mujoco

1. Download mujoco-2.3.7 binary from this website:

   https://github.com/google-deepmind/mujoco/releases

2. Extract it and save to folder: ~/.mujoco/XXX

3. Run the command in terminal or save it in .bashrc file:

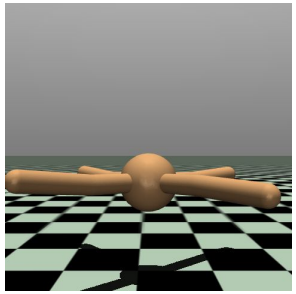
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/username/.mujoco/XXX/bin:/usr/lib/nvidia

4. You should be able to open MuJoCo by run: ~/.mujoco/XXX/bin/simulate

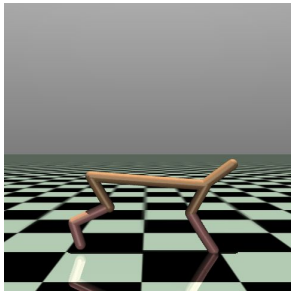
5. You can drag/drop XML file to it and you will see the robot in simulator.

What is OpenAI Gym?

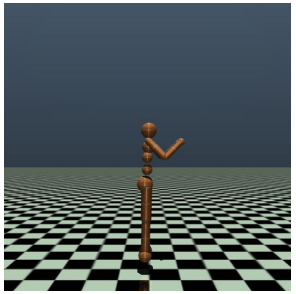
An API standard for reinforcement learning with a diverse collection of environments.



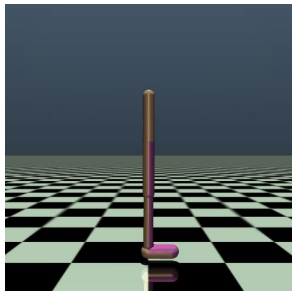
Ant



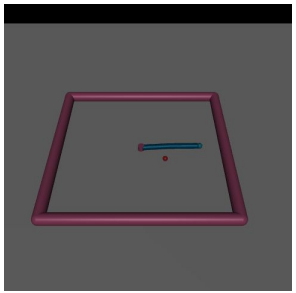
Half Cheetah



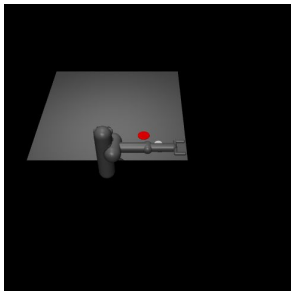
Humanoid



Walker2d



Reacher

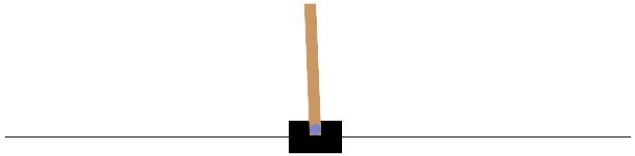


Pusher

How to use OpenAI Gym?

```
import gym
env = gym.make('CartPole-v0')
```

- Get the environment of CartPole from Gym.
- “env” provides states and reward.



How to use OpenAI Gym?

```
state = env.reset()

for t in range(100):
    env.render()
    print(state)

    action = env.action_space.sample()
    state, reward, done, info = env.step(action)

    if done:
        print('Finished')
        break

env.close()
```

A window pops up rendering CartPole.

A random action.

"done=1" means finished (win or lose the game)

Install Environment

Recommend to use conda environment for this project.

1. Install miniconda from this website:

   https://docs.conda.io/projects/miniconda/en/latest/index.html

2. Create and activate conda environment:

   ```
   conda create -n robotlearning python==3.9.0
   conda activate robotlearning
   ```

3. Install dependencies with given requirements.txt:

   ```
   pip install -r requirements.txt
   ```

Project Framework: Overview

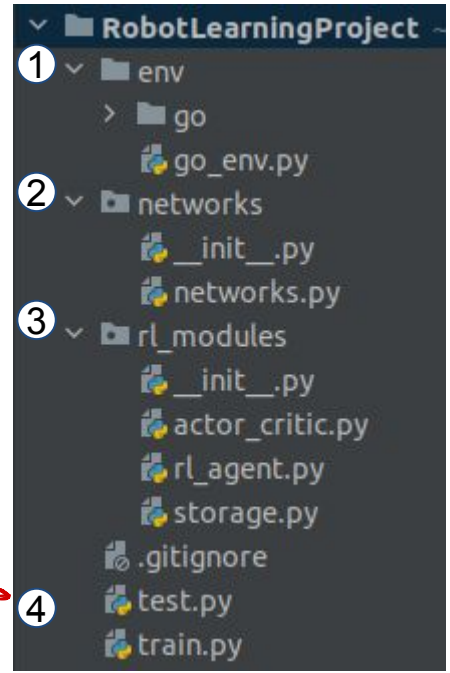
For the project, you will get a framework which implements a simple RL algorithm.
(here A2C is implemented)

① An environment with Unitree-Go robot

② Simple neural network are defined here, for example MLP.

③ An RL pipeline are written here.

④ Train and test your agent.



"test.py" probably needs to be adapted if code is changed →

Project Framework: Environment

Environment is defined in “*go_env.py*”

Two rewards are defined

```python
def _reward_lin_vel(self, before_pos, after_pos):
    target_vel = np.array([0.5, 0, 0])
    lin_vel = (after_pos - before_pos) / self.dt
    return np.exp(-10*np.linalg.norm(target_vel - lin_vel))

def _reward_healthy(self):
    return (self.is_healthy - 1) * 5
```

↖ unhealthy when on the floor
or when walking wrong direction?

Adapt Rewards !

↳ Orientation of the robot !

small deviation (random?)

Simulate one step

current joint angles + dq

```python
def step(self, delta_q):
    action = delta_q + self.data.qpos[-12:]
    action = np.clip(action, a_min=self.lower_limits, a_max=self.upper_limits)

    before_pos = self.data.qpos[:3].copy()
    self.do_simulation(action, self.frame_skip)
    after_pos = self.data.qpos[:3].copy()

    lin_v_track_reward = self._reward_lin_vel(before_pos, after_pos)
    healthy_reward = self._reward_healthy()
    total_rewards = 5.0*lin_v_track_reward + 1.0*healthy_reward

    terminate = self.terminated
    observation = self._get_obs()
    info = {...}

    if self.render_mode == "human":
        self.render()
    return observation, total_rewards, terminate, info
```

delta_q: predict action changes

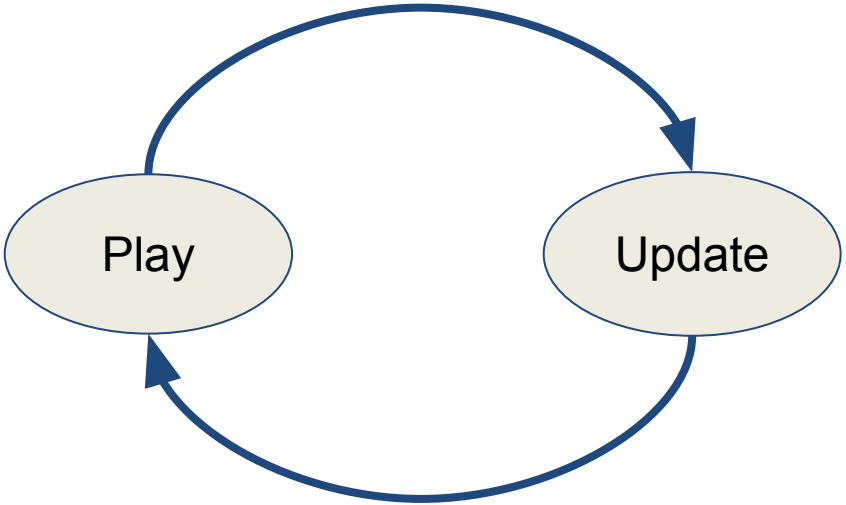
Simulation happens here

Project Framework: RL-pipeline

calls step !

Steps of training RL agents:

1. Play games to collect data.
2. Update policy with collected data




```python
def play(self, is_training=True, early_termination=True):
    obs, _ = self.env.reset()
    infos = []
    for _ in range(self.storage.max_timesteps):
        obs_tensor = torch.from_numpy(obs).to(self.device).float().unsqueeze(dim=0)
        with torch.no_grad():
            if is_training:
                action = self.act(obs_tensor)
            else:
                action = self.inference(obs_tensor)
        obs_next, reward, terminate, info = self.env.step(action*self.action_scale)
        infos.append(info)
        if is_training:
            self.store_data(obs, reward, terminate)
        if terminate and early_termination:
            obs, _ = self.env.reset()
        else:
            obs = obs_next
    if is_training:
        self.compute_returns(torch.from_numpy(obs_next).to(self.device).float())
    return infos
```

first reset env

rollout an episode

sample an action from policy

perform one step action

collect data to storage

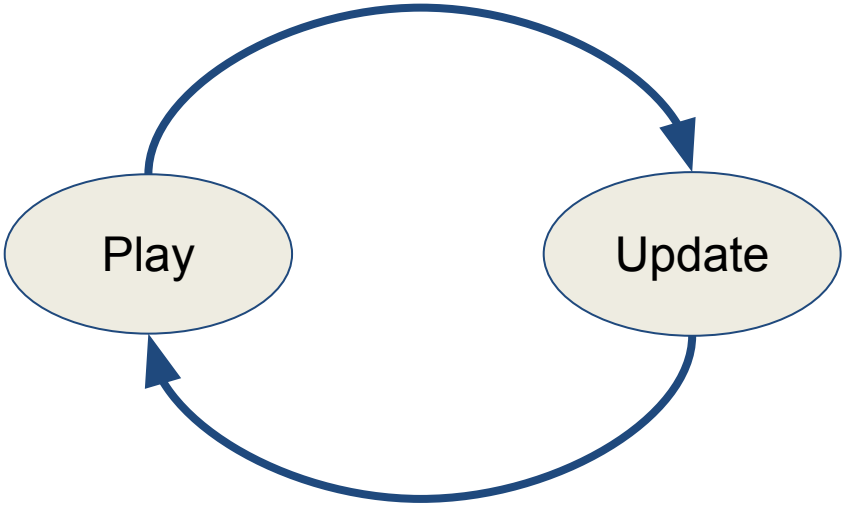
reset env if player in bad situation

Prepare data for update, e.g., advantage estimate

Project Framework: RL-pipeline

Steps of training RL agents:

1. Play games to collect data.
2. Update policy with collected data

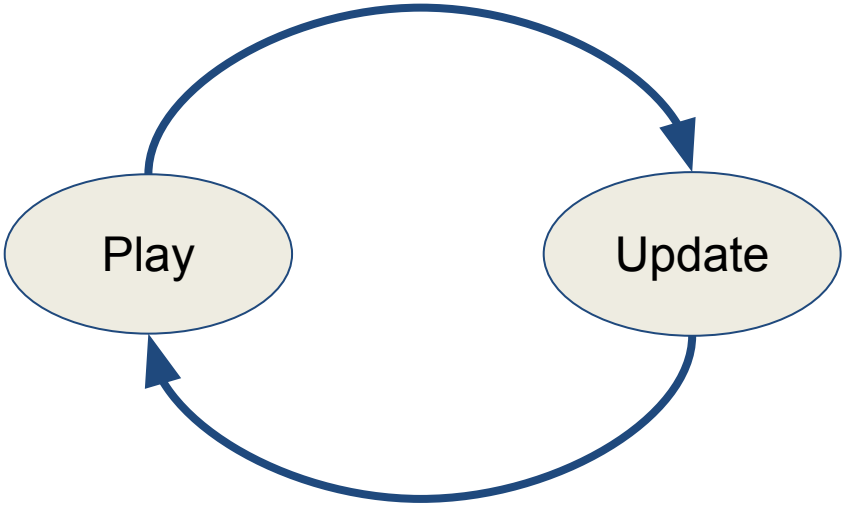



```
def update(self):
    mean_value_loss = 0
    mean_actor_loss = 0
    generator = self.storage.mini_batch_generator(self.num_batches, self.num_epochs, d
                                                  get data from storage

    for obs_batch, actions_batch, target_values_batch, advantages_batch in generator:
        self.actor_critic.act(obs_batch)          evaluate policy
        actions_log_prob_batch = self.actor_critic.get_actions_log_prob(actions_batch)

        actor_loss = (-advantages_batch * actions_log_prob_batch).mean()
        critic_loss = advantages_batch.pow(2).mean()
        loss = actor_loss + self.value_loss_coef * critic_loss
                                                  compute losses

        # Gradient step
        self.optimizer.zero_grad()
        loss.backward()                           update the parameters
        self.optimizer.step()
```

Project Framework: RL-pipeline

Steps of training RL agents:

1. Play games to collect data.
2. Update policy with collected data




```python
def learn(self, save_dir, num_learning_iterations=1000, num_steps_per_val=50):
    for it in range(num_learning_iterations):
        # play games to collect data
        infos = self.play(is_training=True)
        # improve policy with collected data
        mean_value_loss, mean_actor_loss = self.update()

        if it % num_steps_per_val == 0:
            infos = self.play(is_training=False)
            self.save_model(os.path.join(save_dir, f'{it}.pt'))
```

play & update

Project Framework: RL-pipeline

In the framework, the simple advantage estimation are given.

$$\mathcal{A}(s_t, a_t) = r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$$

```python
def compute_returns(self, last_values):
    for step in reversed(range(self.max_timesteps)):
        if step == self.max_timesteps - 1:
            next_values = last_values
        else:
            next_values = self.values[step + 1]
        next_is_not_terminate = 1.0 - self.dones[step]
        delta = self.rewards[step] + next_is_not_terminate * self.gamma * next_values - self.values[step]
        self.advantages[step] = delta
        self.returns[step] = self.advantages[step] + self.values[step]
```

Try to implement GAE, which provide a better advantage estimate.

Project Framework: Train & Test

```python
def train():
    device ='cuda:0'
    log_name = datetime.now().strftime("%Y-%m-%d/%H-%M-%S")
    # create environment
    go_env = GOEnv(render_mode="human")
    # create actor critic
    actor_critic = ActorCritic(state_dim=go_env.obs_dim, action_dim=go_env.action_dim).to(device)
    # create storage to save data
    storage = Storage(obs_dim=go_env.obs_dim, action_dim=go_env.action_dim, max_timesteps=1000)
    rl_agent = RLAgent(env=go_env, actor_critic=actor_critic, storage=storage, device=device)

    save_dir = f'checkpoints/{log_name}'
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    rl_agent.learn(save_dir, num_learning_iterations=1000, num_steps_per_val=100)
```

Start to learn !!!

Project Framework: Train & Test

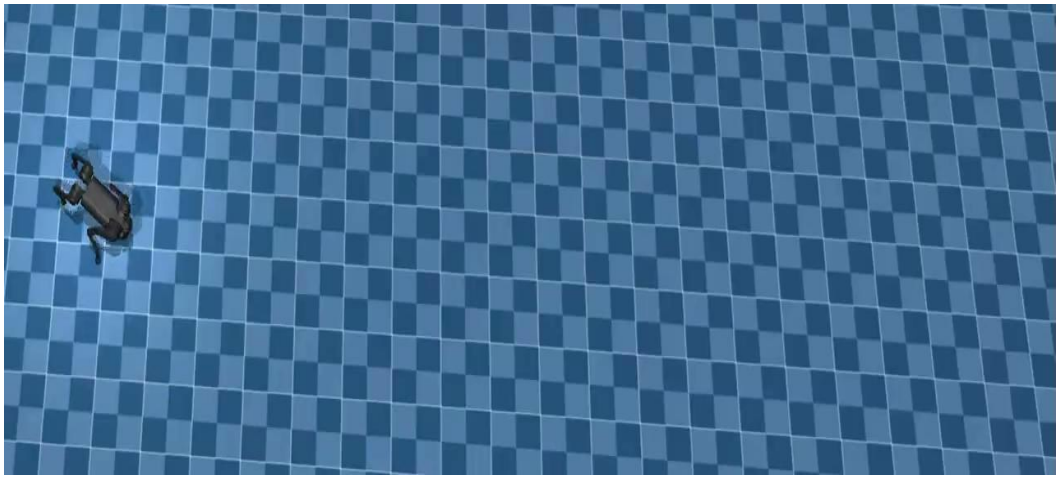
```python
def test():
    device ='cuda:0'
    # create environment
    go_env = GOEnv(render_mode="human")
    # create actor critic
    actor_critic = ActorCritic(state_dim=go_env.obs_dim, action_dim=go_env.action_dim).to(device)
    # create storage to save data
    storage = Storage(obs_dim=go_env.obs_dim, action_dim=go_env.action_dim, max_timesteps=1000)
    rl_agent = RLAgent(env=go_env, actor_critic=actor_critic, storage=storage, device=device)

    rl_agent.load_model('checkpoints/2023-11-29/16-35-42/best.pt')
    rl_agent.play(is_training=False)
```

load model and play

Project Framework: Train & Test

By simply run: **python train.py**
You will see:

Acceptable result might looks like this.



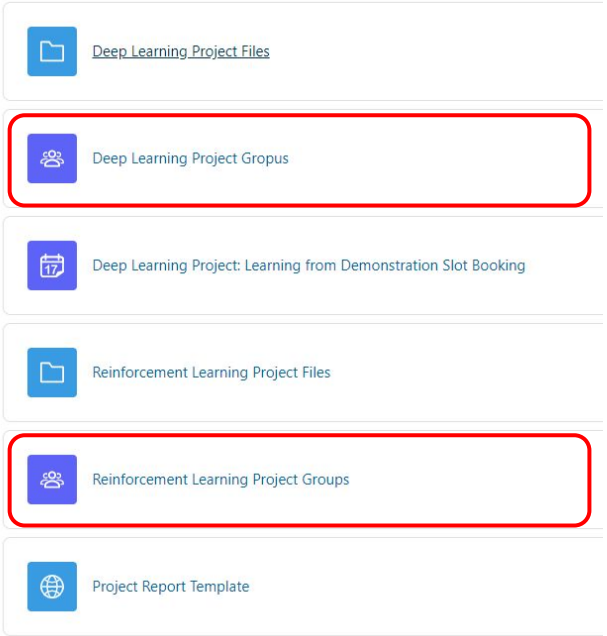
Project Framework: Conclusion

The current framework covers:

1. An Gym environment with Unitree-Go robot.
2. An RL pipeline for training RL agents (A2C).
3. Training and Test Interface.

The current framework will not work directly to make the robot walking forward. You are encouraged to improve it, for example, exploring reward functions, and applying more advanced techniques: PPO, GAE, and ect.

Team up! Decide your own groups

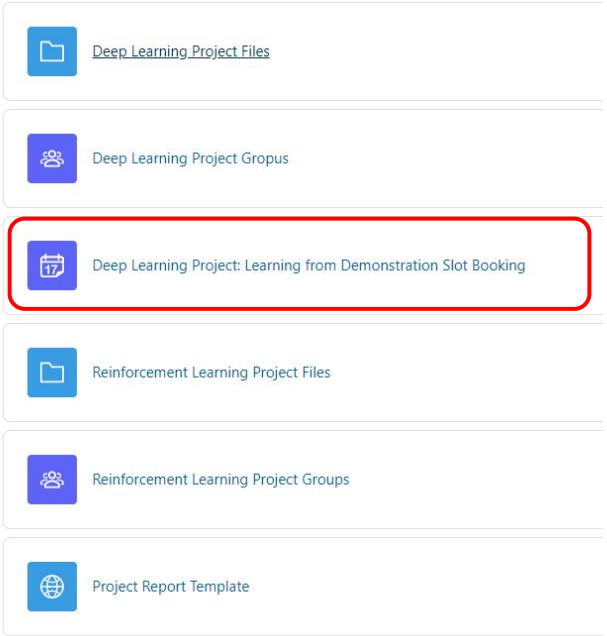


Remember.

Groups of max 4 people

If someone does not have a group, check free spots, contact your classmates and join teams.

Learning From Demonstration Lab Day



Remember.

All groups that selected the Deep Learning Project must sign-up for a date for the Learning From Demonstration Lab