COMP0245 Final Project: Learning to Control a Robot Arm

November 04, 2024 (Last Update: 20241104–1856)

1 Introduction

Many robotics applications, such as the one shown in Figure 1, involve manipulating and grasping objects. These tasks often require robot arms to undergo a complex sequence of motions which require careful modelling to understand the dynamics and behaviour of the robot arms.

Figure 1: A Kuka robot picking up a box. From "The Inevitability of Warehouse Robots", 2020.

This project has three main themes:

1. An initial task in which you will attempt to solve a 1D simplified problem using PyTorch.
2. A second task in which you will attempt to learn to repeat robot trajectories.
3. A final task in which you will apply this to control the simulated Panda Robot.

This project will require some basic coding with PyTorch. The first task will provide the background you require when using this library.

2 Prerequisites

Before starting the tasks, pleasure ensure that you have updated the RoboEnv repository and you have installed the latest version of `simulation_and_control`. (If you do not, when running the code for this project you will encounter errors about a missing folder called `env_scripts`).

You will need to ensure that two sets of software are installed: Scikit and PyTorch. Both of these need to be installed in your `roboenv2` environment.

Scikit

The necessary parts of scikit can be installed using:

```
conda install scikit-learn scikit-optimize
```

PyTorch

To install PyTorch, use `pip`. You can either install a CPU version or a GPU version. Note that the CPU version is sufficient for this project. Although GPUs can significantly speed up the code, GPU installations are notoriously difficult to get working because of the need to have compatible hardware, operating system version, driver version, and PyTorch and support library versions. Therefore, you should only consider installing a GPU if you are familiar with such issues and are feeling brave.

- **CPU (all platforms):**

  ```
  pip3 install torch torchvision torchaudio \
      --index-url https://download.pytorch.org/whl/cpu
  ```

- **GPU (Linux / Windows): With Nvidia GPU and the driver (cuda 12.4) and Nvidia toolkit correctly installed.**

  ```
  pip3 install torch torchvision torchaudio
  ```

  For different versions of cuda driver refer to this page https://pytorch.org/get-started/locally/. Always proceed with the `pip` installation otherwise it will break roboenv2

- **GPU (Mac): On a Mac PyTorch supports Metal acceleration**

  Follow the instructions at `https://developer.apple.com/metal/pytorch/`.

  However, `pip` reports a large number of conflicts with `roboenv2` package versions, so this is not recommended at this time.

3 Tasks

Task 1: Correcting an Erroneous Model of Motor Behaviour (40%)

In this first task, we will look at how machine learning can be used to help fit a nominal, erroneous model to real-world data. In particular, we look at how to improve the models used to describe the behaviour of each individual motor which is used to actuate each joint

of the robot arm. The motor behaviour is described in Appendix A. The nominal model that's been proposed does not include a damping term which applies to the actual motor.

For this task, you will be working with the file `task1.py`. The file contains dataset generation together with the class `ShallowCorrectorMLP` which implements a shallow neural network utilizing the ReLU activation function. Your will explore the properties of this network with various training settings. You'll also implement a simple deep neural network to evaluate its performance.

1. **Task 1.1**

   By looking at training loss and prediction accuracy, analyse the impact of changing the number of hidden nodes in `ShallowCorrectorMLP` between 32 and 128 in steps of 32 units. What overall trends do you see? How do these results compare if no attempt at correcting the model is used at all?

2. **Task 1.2**

   Create a second network called `DeepCorrectorMLP`, which contains a second fully-connected hidden layer. The number of units per layer should be parameterizable by the input variable `num_hidden_nodes`. Compare the performance with the shallow network for the hidden units varying from 32 to 128 in steps of 32.

   To add another hidden layer, insert additional `nn.Linear` and `nn.ReLU` layers in the constructor. Since the layer is fully connected, the input and output layers of `nn.Linear` will be `num_hidden_units`.

3. **Task 1.3**

   The learning rate influences the size of each step taken by the optimizer during the training. Test the effect of changing the learning rate for both `ShallowCorrectorMLP` and `DeepCorrectorMLP` for learning rates of 1.0, 1e-1 (the default value), 1e-2, 1e-3 and 1e-4. How do these values affect the training loss and generalization performance (over the entire domain, and not just the training data)?

   The learning rate is specified by the value `lr` passed to the optimizer constructor.

4. **Task 1.4**

   Batch size affects how many training examples are used to estimate the gradient. In this task, use 32 hidden units. Evaluate the effect of using batch sizes of 64, 128, 256 and 1000 (all the training data) on both `ShallowCorrectorMLP` and `DeepCorrectorMLP`. Evaluate the impact of each batch size on training times, training loss and test loss. Analyze how the batch size affects the stability and efficiency of the training process.

   The batch size is controlled by the `batch_size` parameter in the constructor of the `DataLoader` for the `train_loader`.

Hints

- This task requires you to carry out a number of comparisons of fairly simple configuration changes. See to what degree you can automate this. For example, you might want to consider parameterising the code to support these changes, and then have the code automatically generate subdirectories in which to store various intermediates and results files. The names of these subdirectories could reflect the different parameter settings used. In LaTeX, you can specify directory names as part of the image file path.

Task 2: Training to Control the Motion of the Robot (40%)

Now that the behaviour of each individual motor has been corrected, the next stage is to develop a system that plans the movement of the robot arm from its start to its goal location. This consists of two parts. First a smooth trajectory for the end effector is determined using the cubic spline interpolation method described in Appendix B. Second, the joint angles and joint velocities must be computed using the method in Appendix C. These two parts can be used in a global optimization problem. Although this will produce extremely efficient trajectories, it can be computationally costly to run the optimizer every single time. Machine learning offers a way to cache optimal solutions, and to generalize quickly (with some sacrifice of optimality) to novel tasks (in this case target end effector positions).

The goal of this task is to learn models that predict the joint positions over a time window of 5 seconds to drive the robot to a desired Cartesian goal position by training regressors on the available data. You will use different machine learning methods to achieve this and compare their performance. You will use `task21.py`, `task22.py`, and the dataset `data.pkl` to complete this task. The file `data.pkl` is a dataset which computes a number of optimal trajectories for the robot arm.

We assume that the motors for each individual model have been correctly modelled. Therefore, it is not necessary to use your results from Task 1 to complete this task.

Activities

1. **Task 2.1**

   Train the Multilayer Perceptron (MLP) model in `task21.py` for each joint to predict its position and velocity given the time input. What are the the training losses and test losses for this MLP?

   By choosing 10 goal positions randomly, evaluate the generalization capability of the MLP. Your evaluation could include quantitative measures (such as Mean Squared Error (MSE) from the target location) and qualitative measures (such as a discussion of the types of behaviours observed in the trajectories?)

2. **Task 2.2**

   Train the Random Forest regression models in `task22.py` for each joint to perform the same task of predicting joint positions to drive the robot to a desired Cartesian goal.

   Experiment with the hyperparameters to explore their impact on the tree. Initially set `n_estimators` to 100 and `max_depth` to `None` to allow the model to grow as deep as necessary. What are the results? How deep is the learned tree? Is there any evidence of overfitting? Also experiment with the effect of different depths of the random forests (2–10). What do you notice about the computed trajectories with the different depths?

3. **Task 2.3**

   Compare and contrast the results of the two methods (MLP and Random Forest). Analyze which method performs better in terms of accuracy, generalization to the new Cartesian goals, and computational requirements. Discuss the advantages and disadvantages of each method in the context of modeling joint trajectories for robotic tasks.

Additional Information

- `data.pkl` **is a Python pickle file that contains the dataset which is used to train and test the models. The dataset includes the following components:**
  - `time`**: An array that records the timestamps at which the joint data was captured. The data has been collected with a frequency of 2 ms over 5 seconds.**
  - `q_mes_all`**: A matrix containing measured joint positions. Each column corresponds to one of the joints in the robotic arm (seven joints total), and each row represents a snapshot of these joint positions at the corresponding time.**
  - `goal_positions`**: An array that includes the target or goal positions in Cartesian coordinates** (x, y, z) **that the robotic arm aims to achieve. Each row corresponds to a goal position associated with the time steps recorded in the** `time` **array.**
- `training_flag` **is used to control whether the training phase of the machine learning models should be executed. Setting this flag to** `True` **initiates the loading of data, training of the model across specified epochs, and the eventual saving of the trained model parameters.**
- `test_cartesian_accuracy_flag` **determines whether to execute the testing phase that specifically assesses the model's accuracy in reaching new, unseen goal positions. This part of the script uses the trained models to predict joint positions based on new goal inputs and evaluates how well these predictions match the expected outcomes in terms of Cartesian coordinates. The usage of this flag is critical when:**
  - **You need to validate the model's performance on new data that was not part of the training dataset.**
  - **You are interested in specifically understanding the model's effectiveness in practical scenarios, such as accurately positioning a robotic arm in a simulation or real-world environment.**
  - **You want to conduct targeted tests without re-running the entire training process, thus saving time and computational resources.**

  Note this will fail if you haven't trained all the joint values.

Task 3 (20%)

Task 2 trained regressors to generate a sequence of target joint angles over time. However, no attempt was made to use this sequence to control the motion of a robot arm. In this task, you will use the trained model from `task2.py` **to command a simulated 7 DoF Franka Panda Robot. The code for the simulator is provided in** `task3.py`**. To drive the robot, you will need to pass to a feedback linearization controller the sequence of desired joint positions** (q_d) **and velocities** ($\dot{q}_d$)**. For the random forest, you should compare the performance of trees with depth 2 and depth 10.**

Activities

1. **Task 3.1**

   Understand how different machine learning models affect the smoothness of the generated joint trajectories and how this, in turn, impacts the robot's motion. Run

the simulation twice: once using the MLP (Neural Network) models by setting `neural_network_or_random_forest` to "neural_network", and once using the Random Forest models by setting it to `"random_forest"`. Observe the robot's behavior during both simulations, paying close attention to the smoothness of movements and any abrupt changes or hesitations. Extract the predicted joint positions (`q_mes`) over time for both models and plot these trajectories for each joint to visualize the differences. Analyze how the smoothness of the trajectories affects the robot's motion and explain any differences observed between the two models.

2. **Task 3.2**

   Investigate how discontinuities in the joint trajectories impact the performance of the feedback linearization controller and the robot's ability to follow the desired path. Calculate the tracking error by computing the difference between the desired joint positions (`q_des`) and the measured joint positions (`q_mes`) over time, and plot the tracking error for both models. Analyze the control inputs (`tau_cmd`) generated during the simulations, observing any spikes or large variations in control torques when using the Random Forest model. Discuss how trajectory discontinuities affect the tracking error and the control torques.

3. **Task 3.3**

   Explore methods to mitigate the issues caused by discontinuous trajectories from the Random Forest model. Implement a smoothing filter, such as an exponential moving average filter, on the predicted joint positions from the Random Forest model. Re-run the simulation using the smoothed trajectories and observe any changes in the robot's motion. Plot and compare the smoothed trajectories against the original Random Forest predictions and the MLP predictions. Analyze how smoothing the trajectories affects the robot's motion and control performance, discussing the advantages and potential drawbacks of applying smoothing techniques to the Random Forest outputs.

Hints

- **Task 3.1 Hint: Since the code is provided, focus on analyzing the predicted joint trajectories generated by the MLP models. Use the existing code to generate the desired joint positions q_d and velocities $\dot{q}_d$ over time. Plot the predicted joint positions and velocities for each joint to visualize the trajectories. Pay attention to the smoothness and continuity of these trajectories. Additionally, compare these trajectories with those generated by the Random Forest models by changing the `neural_network_or_random_forest` variable in the code. Plot both sets of trajectories on the same graphs to highlight the differences between the models.**

- **Task 3.2 Hint: Run the simulation using the provided code, which integrates the generated q_d and $\dot{q}_d$ trajectories into the feedback linearization controller. Observe the robot's motion during the simulation. Specifically, note how smoothly the robot moves and how accurately it follows the desired path. You may want to record the simulation or take snapshots at key moments to assist in your analysis. Consider also observing any differences in the robot's behavior when using trajectories from the MLP models versus the Random Forest models.**

- **Task 3.3 Hint: After running the simulation, collect data on the actual joint positions q and velocities $\dot{q}$ over time. Plot the actual joint positions and velocities alongside**

the desired ones to visualize the tracking performance. Compute and plot the tracking errors (the differences between the desired and actual values) for each joint over time. Analyze these plots to assess how well the robot tracks the desired trajectories. Additionally, evaluate the robot's ability to reach the goal position by comparing the final end-effector position to the target. Discuss any deviations observed and consider potential reasons for these errors, such as model inaccuracies or the impact of trajectory discontinuities on the controller's performance.

A Motor Dynamics

A motor is commanded to achieve, at time t, a desired target angle $\theta_T(t)$ with angular velocity $\dot{\theta}_T(t)$. The actual motor position and velocity are $\theta(t)$ and $\dot{\theta}(t)$ respectively.

The motor's angle and velocity are governed by the motor's torque. The torque is controlled by a PD controller,

$$\tau(t) = K_p\left(\theta_T(t) - \theta(t)\right) + K_d\left(\dot{\theta}_T(t) - \dot{\theta}(t)\right). \tag{1}$$

Given this, the ideal angular acceleration of the motor is

$$\ddot{\theta}^*(t) = \frac{\tau(t)}{m} \tag{2}$$

However, the actual motor has unmodelled damping. Therefore, the actual acceleration of the motor is

$$\ddot{\theta}(t) = \frac{\tau(t) - b\dot{\theta}(t)}{m} \tag{3}$$

This can be writen as

$$\ddot{\theta}(t) = \ddot{\theta}^*(t) + \Delta\ddot{\theta}(t), \tag{4}$$

where $\Delta\ddot{\theta}(t)$ is a correction term which is to be estimated from the data.

B Cubic Interpolation of End Effector Location

B.1 Introduction

In this appendix we explain the mathematics behind a cubic spline interpolation used to transition a robot's end-effector from a starting Cartesian position $\mathbf{p}_0$ to a goal position $\mathbf{p}_f$ over a specified duration T. The spline assumes zero initial and final velocities, ensuring a smooth start and stop.

B.2 Problem Statement

Given:

- **Starting position: $\mathbf{p}_0 = [x_0, y_0, z_0]^T$**
- **Goal position: $\mathbf{p}_f = [x_f, y_f, z_f]^T$**
- **Duration: T (e.g., $T = 5$ seconds)**

Find a time-dependent position function $\mathbf{p}(t)$ and velocity function $\dot{\mathbf{p}}(t)$ for $t \in [0, T]$ such that:

$$\mathbf{p}(0) = \mathbf{p}_0 \quad \textbf{(initial position)} \tag{5}$$
$$\mathbf{p}(T) = \mathbf{p}_f \quad \textbf{(final position)} \tag{6}$$
$$\dot{\mathbf{p}}(0) = \mathbf{0} \quad \textbf{(initial velocity)} \tag{7}$$
$$\dot{\mathbf{p}}(T) = \mathbf{0} \quad \textbf{(final velocity)} \tag{8}$$

B.3 Cubic Polynomial for Each Axis

We construct a cubic polynomial for each Cartesian axis (i.e., x, y, and z):

$$p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$
$$\dot{p}(t) = a_1 + 2a_2 t + 3a_3 t^2$$

Our goal is to find the coefficients a_0, a_1, a_2, and a_3 for each axis that satisfy the boundary conditions in equations (5)–(8),

$$p(0) = a_0 = p_0$$
$$\dot{p}(0) = a_1 = 0$$
$$p(T) = a_0 + a_1 T + a_2 T^2 + a_3 T^3 = p_f$$
$$\dot{p}(T) = a_1 + 2a_2 T + 3a_3 T^2 = 0$$

The coefficients which match these conditions are

$$a_0 = p_0$$
$$a_1 = 0$$
$$a_2 = 3\frac{(p_f - p_0)}{T^2}$$
$$a_3 = -2\frac{(p_f - p_0)}{T^3}$$

C Joint Positions and Velocities

C.1 Introduction

This appendix provides a mathematical explanation of the method implemented in the provided code for computing the desired joint positions and velocities given the desired Cartesian position and orientation of a robot's end-effector. This appendix is rather mathematical and is mainly provided for completeness. You do not need to understand the details of it to complete the project.

The method involves:

- **Computing the error between the desired and current Cartesian positions and orientations.**
- **Applying proportional gains to these errors to compute desired Cartesian velocities.**
- **Using the robot's Jacobian to map desired Cartesian velocities to desired joint velocities via pseudoinverse.**

- **Applying constraints such as dead zones and joint velocity saturation.**
- **Computing desired joint positions using kinematic integration.**

C.2 Problem Statement

Given:

- **Current joint positions $\mathbf{q} \in \mathbb{R}^n$.**
- **Current joint velocities $\dot{\mathbf{q}} \in \mathbb{R}^n$.**
- **Desired Cartesian position $\mathbf{p}_{\text{des}} \in \mathbb{R}^3$.**
- **Desired Cartesian velocity $\dot{\mathbf{p}}_{\text{des}} \in \mathbb{R}^3$.**
- **Desired orientation represented as a quaternion $\mathbf{q}_{\text{des}}$.**
- **Desired angular velocity $\boldsymbol{\omega}_{\text{des}} \in \mathbb{R}^3$ (if provided).**
- **Time step Δt.**
- **Proportional gains k_p^{pos} and k_p^{ori}.**
- **Joint velocity saturation limits $\dot{\mathbf{q}}_{\text{max}}$.**

Compute:

- **Desired joint velocities $\dot{\mathbf{q}}_{\text{des}}$.**
- **Desired joint positions $\mathbf{q}_{\text{des}}$.**

C.3 Mathematical Formulation

The relationship between joint velocities and end-effector velocities is given by the Jacobian $\mathbf{J}(\mathbf{q})$:

$$\begin{bmatrix} \dot{\mathbf{p}} \\ \boldsymbol{\omega} \end{bmatrix} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \tag{9}$$

where:

- **$\dot{\mathbf{p}} \in \mathbb{R}^3$ is the linear velocity of the end-effector.**
- **$\boldsymbol{\omega} \in \mathbb{R}^3$ is the angular velocity of the end-effector.**
- **$\mathbf{J}(\mathbf{q}) \in \mathbb{R}^{6\times n}$ is the geometric Jacobian.**

C.4 Error Computation

C.4.1 Position Error

The position error is computed as:

$$\mathbf{e}_{\text{pos}} = \mathbf{p}_{\text{des}} - \mathbf{p}(\mathbf{q}) \tag{10}$$

where $\mathbf{p}(\mathbf{q})$ is the current end-effector position computed from forward kinematics.

C.4.2 Orientation Error

The orientation error can be represented using quaternions.

Let:

- $\mathbf{q}_{\text{cur}}$ **be the current orientation quaternion derived from** $\mathbf{R}(\mathbf{q})$**.**
- $\mathbf{q}_{\text{des}}$ **be the desired orientation quaternion.**

Compute the error quaternion:

$$\mathbf{q}_{\text{err}} = \mathbf{q}_{\text{cur}}^{-1} \otimes \mathbf{q}_{\text{des}} \tag{11}$$

Extract the vector part (imaginary components) of the quaternion to obtain the orientation error in $\mathbb{R}^3$**:**

$$\mathbf{e}_{\text{ori}} = \mathbf{q}_{\text{err}}.\mathbf{vec} = \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} \tag{12}$$

Optionally, rotate the error into the base frame:

$$\mathbf{e}_{\text{ori}}^{\text{base}} = \mathbf{R}(\mathbf{q})\mathbf{e}_{\text{ori}} \tag{13}$$

C.4.3 Desired Cartesian Velocities

Apply proportional control in Cartesian space:

$$\begin{bmatrix} \dot{\mathbf{p}}_{\text{cmd}} \\ \boldsymbol{\omega}_{\text{cmd}} \end{bmatrix} = \mathbf{K}_p \begin{bmatrix} \mathbf{e}_{\text{pos}} \\ \mathbf{e}_{\text{ori}} \end{bmatrix} + \begin{bmatrix} \dot{\mathbf{p}}_{\text{des}} \\ \boldsymbol{\omega}_{\text{des}} \end{bmatrix} \tag{14}$$

where $\mathbf{K}_p$ **is a diagonal gain matrix:**

$$\mathbf{K}_p = \begin{bmatrix} k_p^{\text{pos}} \mathbf{I}_{3\times3} & \mathbf{0} \\ \mathbf{0} & k_p^{\text{ori}} \mathbf{I}_{3\times3} \end{bmatrix} \tag{15}$$

C.4.4 Desired Joint Velocities

Compute the desired joint velocities using the pseudoinverse of the Jacobian:

$$\dot{\mathbf{q}}_{\text{des}} = \mathbf{J}^{\dagger} \left(\mathbf{K}_p \begin{bmatrix} \mathbf{e}_{\text{pos}} \\ \mathbf{e}_{\text{ori}} \end{bmatrix} + \begin{bmatrix} \dot{\mathbf{p}}_{\text{des}} \\ \boldsymbol{\omega}_{\text{des}} \end{bmatrix} \right) \tag{16}$$

where $\mathbf{J}^{\dagger}$ **is the Moore-Penrose pseudoinverse of the Jacobian.**

C.4.5 Desired Joint Positions

Compute the desired joint positions using kinematic integration:

$$\mathbf{q}_{\text{des}} = \mathbf{q} + \dot{\mathbf{q}}_{\text{des}}^{\text{sat}} \Delta t \tag{17}$$