Neural Networks for Optimal Control - Exercise session Solutions

Clara Galimberti and Leonardo Massai

Introduction

This is the exercise set for the course "Neural Networks for Optimal Control." The starter code for the exercises can be found at: http://github.com/DecodEPFL/NNs-for-OC.
Before beginning the exercises, please ensure that you have all the necessary tools installed on your computer. We will use Python, and the following packages are required:

- `torch`
- `numpy`
- `matplotlib`
- `jax`
- `pip`
- `tqdm`

Detailed installation instructions can be found in Appendix C.
Once everything is installed, run the file `test_installation.py`. If it completes without errors, it means that the installation was successful.

1 Control of the water level in a tank

1.1 Model description

We consider a water tank, as shown in Figure 14. The system model is given by:

$$A\big(x(t)\big)\dot{x}(t) = u_{tot}(t) - a\sqrt{x(t)}, \tag{1}$$

where $a > 0$ and $A(x)$ is defined as $A(x) = b + x$ with $b > 0$. The state $x(t)$ represents the water level in the tank, and the input $u_{tot}(t)$ corresponds to the inflow of water into the tank. For this exercise, we will use the parameter values listed in Table 2.

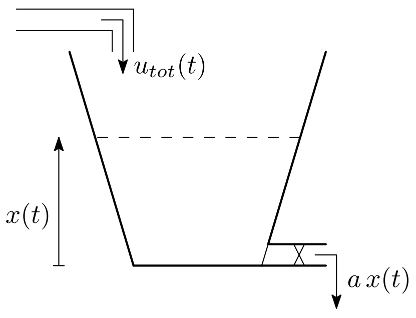

Figure 1: Water tank scheme

a	0.1
b	0.5
$\bar{x}$	0.2
h	0.1
N	200

Table 1: Parameters for the water tank system

1.2 Objective

Our goal is to stabilize the system around a fixed water level $\bar{x} > 0$. Additionally, we aim to achieve this as quickly as possible while minimizing energy consumption. Furthermore, we must comply with the following physical constraints:

R1 the input $u_{tot}(t)$ should be *smooth*;

R2 the water inflow cannot be negative ($u_{tot}(t) \geq 0$ for all t);

R3 the water level must remain within the range $0 \leq x(t) \leq x_{max}$, for all t.

To achieve this, we will follow the two-step procedure introduced in the course. First, we will design a basic pre-stabilizing controller. Then, we will implement a performance-boosting controller to tackle the requirements R1-R3.

1.3 Exercises

1.3.1 Pre-stabilizing controller

Given a desired equilibrium $\bar{x}$, a base controller that asymptotically stabilizes the tank system is given by

$$u_{tot}(t) = a\sqrt{\bar{x}} + u(t). \tag{2}$$

The dynamics of the pre-stabilized system can be written as

$$\dot{x} = \frac{1}{x+b}(-a\sqrt{x} + a\sqrt{\bar{x}} + u). \tag{3}$$

1. Validate the asymptotic stability of the system by plotting the phase portrait x vs. $\dot{x}$. To do so, run the script `run_validation_prestab.py`, which can be found in the `experiments/tank/` folder.

Solution: The plot in Figure 2 confirms the asymptotic stability of the scalar system since:

- $\dot{x} = 0$ when $x = \bar{x}$,
- $\dot{x} > 0$ when $x < \bar{x}$, and
- $\dot{x} < 0$ when $x > \bar{x}$.

1.3.2 System discretization

To simulate the system, we use the forward Euler integration scheme with a discretization step size h. The discretized dynamical system is then given by:

$$x_{k+1} = x_k + \frac{h}{x_k + b}(-a\sqrt{x_k} + a\sqrt{\bar{x}} + u_k). \tag{4}$$

We assume that the system is affected by additive process noise, meaning that (4) is subject to disturbances w_k, leading to:

$$x_{k+1} = x_k + \frac{h}{x_k + b}(-a\sqrt{x_k} + a\sqrt{\bar{x}} + u_k) + w_k. \tag{5}$$

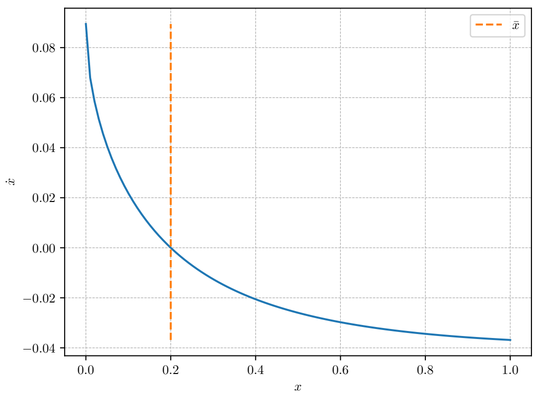

Figure 2: Graphical validation of asymptotic stability of the CT system

This noise can be interpreted as a disturbance introduced by the discretization process or as an error due to model mismatch.

2. Simulate the pre-stabilized system starting from different initial conditions by running the script `run_OL_trajectories.py`. Verify that the system stabilizes around $\bar{x}$ when considering $u_k = w_k = 0$.

   Remark 1. *Note that the system is defined in the class* `TankSystem`*. Details of this class can be found in Appendix A. To simulate the system, a controller model is created. In this case, we use the* `ZeroController` *class, which outputs zero at all times. Then, the controller is then applied in the* `TankSystem.rollout` *method.*

   Solution: The plot in Figure 3 shows five system trajectories starting from different initial conditions and simulated for N steps. It can be observed that all trajectories converge towards $\bar{x}$.

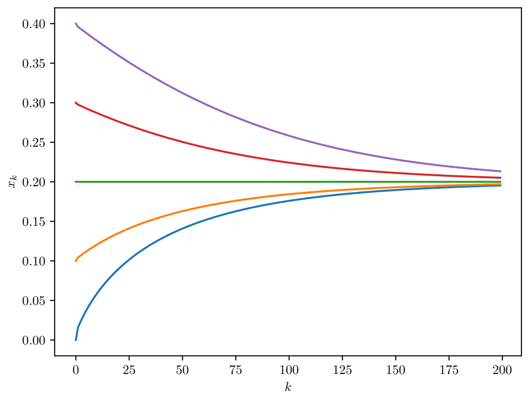

Figure 3: Five trajectories of the pre-stabilized system

1.3.3 Performance-boosting controller

The system is now stable around $\bar{x}$. However, to improve the performance of the controller, we will design performance-boosting controllers that aim to minimize different cost functions. As described in

Section 3.2, our goal is to stabilize the system as quickly as possible while controlling the supplied energy and complying with the physical requirements R1-R3.

3. Run the script `run_tank_LQ.py` which designs a PB controller for improving the performance of the tank system with respect to the following quadratic cost:

$$L_1 = \sum_{k=0}^{N} \frac{1}{2}(q(x-\bar{x})^2 + ru^2), \tag{6}$$

   when starting from $x_0 = 0.1$, while considering disturbances $w_k \sim \mathcal{N}(0, 0.005)$.

   The objective of this point is to familiarize oneself with all the elements needed to train a PB controller. In particular:

   - The training procedure (a `for` loop that does a gradient descend step at each iteration) can be found in the `run_tank_LQ.py` file.
   - The file `controllers/PB_controller.py` contains a PB controller implementation in the class `PerfBoostController`. Note that one can choose between RENs and SSMs for a neural network implementation of an $\mathcal{L}_2$ operator. Details of this class can be found in Appendix B.
   - The cost function (6) to be minimized is implemented in the class `TankLoss` which can be found in the file `experiments/tank/loss_functions.py`.

   Test the training for different hyperparameter weights.

   Solution: Figure 4 shows a closed-loop trajectory after training when running the script `run_tank_LQ.py`

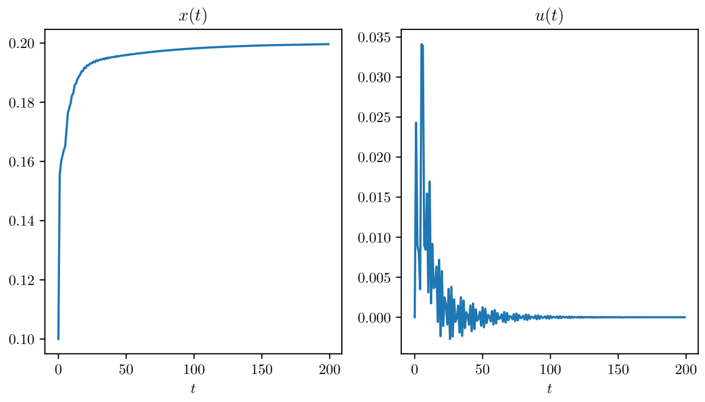

Figure 4: State and input trajectories of the Tank system after training a PB controller with (6) as cost function.

4. The function `TankLoss.loss_smooth_input`, defined in `experiments/tank/loss_functions.py`, seeks to promote R1 by penalizing the absolute difference between two consecutive inputs. Design and test a new controller by incorporating this cost term into the loss function of the optimization problem.

   Solution: One solution can be obtained by running the script `tank_sol_smooth.py`. Figure 5 shows a closed-loop trajectory after training.

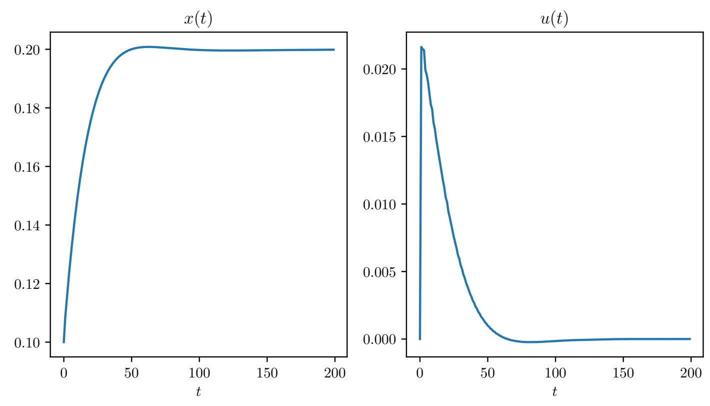

Figure 5: State and input trajectories of the Tank system after training a PB controller while promoting R1.

5. To promote R2, design a barrier-like cost function that penalizes negative input flows, and use it to design a new PB controller.

   Hint 1: You can design a new method inside the `TankLoss` class (in `loss_functions.py`) and add it to the `TankLoss.forward` method.

   Hint 2: Recall that a negative input flow implies $u_t < -a\sqrt{\bar{x}}$.

   Hint 3: This cost function is relevant (meaningful?) when training a controller starting at $x_0 > \bar{x}$. Train the controller for an initial condition that satisfies this inequality.

   Solution: One solution can be obtained by running the script `tank_sol_u_pos.py`. In this case, the controller is trained for an initial condition $x_0 = 0.5$. Figure 6 shows a closed-loop trajectory after training.

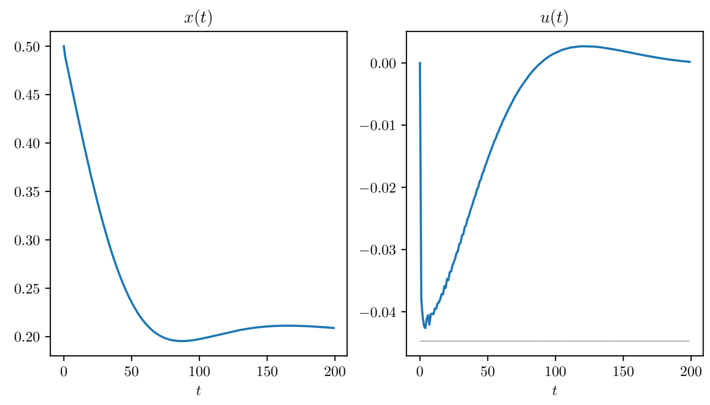

Figure 6: State and input trajectories of the Tank system after training a PB controller while promoting R1-R2. Grey: $-a\sqrt{\bar{x}}$.

6. Requirement R3 states that if the initial condition of the system is within the interval $[0, x_{max}]$, the state must remain in this interval. Design and include an invariant-like cost function in the training procedure to promote requirement R3. Test the resulting controller.

   Solution: One solution can be obtained by running the script `tank_sol_barrier.py`. In this case, the controller is trained for an initial condition $x_0 = 0.2$ and $\bar{x} = 0.01$. Figure 7 shows a closed-loop trajectory after training.

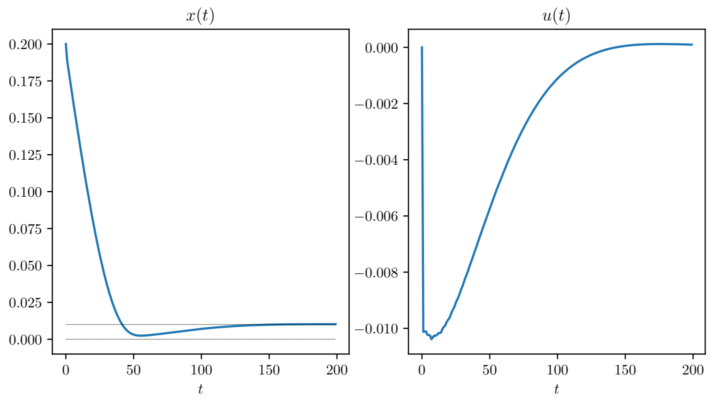

Figure 7: State and input trajectories of the Tank system after training a PB controller while promoting R1-R3. Grey: zero and $\bar{x}$.

7. **Bonus Exercise (if time allows):** Experiment with different hyperparameters and explore alternative models for the $\mathcal{L}_2$ operator: change dimensions and activation funcitons of the State-Space Model (SSM), use Recurrent Equilibrium Networks (REN) instead of SSMs, etc. Observe how these changes affect the performance of the neural network controller.

2 Control of a vehicle on the horizontal plane

We now consider a point-mass vehicle. The vehicle is described by its position $p_t \in \mathbb{R}^2$ and velocity $q_t \in \mathbb{R}^2$, subject to nonlinear drag forces (e.g., air or water resistance). The discrete-time model is given by

$$\begin{bmatrix} p_{t+1} \\ q_{t+1} \end{bmatrix} = \begin{bmatrix} p_t \\ q_t \end{bmatrix} + h \begin{bmatrix} q_t \\ -\frac{1}{m}C(q_t) + \frac{1}{m}F_t \end{bmatrix} + w_t \,, \tag{7}$$

where $m > 0$ is the mass of the robot, $F_t \in \mathbb{R}^2$ denotes the control input force, $h > 0$ is the sampling time, w_t represents an unknown disturbance affecting the system, and $C : \mathbb{R}^2 \to \mathbb{R}^2$ is a *drag function* given by

$$C(s) = b_1 s - b_2 \tanh(s) \,, \tag{8}$$

for some $0 < b_2 < b_1$.

2.1 Stability

The robot must reach the origin with zero velocity in a stable manner. This elementary goal can be achieved using a base proportional controller:

$$F'_t = -K'(p_t) \,, \tag{9}$$

where $K' = \operatorname{diag}(k_1, k_2)$ and $k_1, k_2 > 0$.

The pre-stabilized system is then given by:

$$x_{t+1} = x_t + h \left(\begin{bmatrix} 0 & 1 \\ -\frac{K'}{m} & -\frac{b_1}{m} \end{bmatrix} x_t + \begin{bmatrix} 0 \\ \frac{b_2}{m} \tanh(q_t) \end{bmatrix} + \begin{bmatrix} 0 \\ u_t \end{bmatrix} \right) + w_t \tag{10}$$

where $x_t = \begin{bmatrix} p_t & q_t \end{bmatrix}^\top$, and u_t is the remaining input to the system, i.e. $F_t = F'_t + u_t$.

1. Simulate the vehicle dynamics, prior to setting the pre-stabilizing controller, i.e., with $k_1 = k_2 = 0$, and no disturbances. Use the following input signals:

   (a) A step signal for 20 steps, then zero.

   (b) A sinusoidal signal for 20 steps, then zero.

   The file `run_robot.py` can be used as a template.

   Hint 1: Use the `rollout` method with a "controller" that outputs the desired u_t, as per `InputController` which can be found in `controllers/input_signal.py`.

   Hint 2: Use the functions `plot_trajectories` and `plot_traj_vs_time`, which can be found in the folder `experiments/robot/plot_functions.py`, to plot the obtained state and input trajectories.

   Solution: One solution is implemented in `robot_sol_OL.py`. Figures 8 and 9 show the time evolution of the state and input trajectories of the vehicle for each input signal.

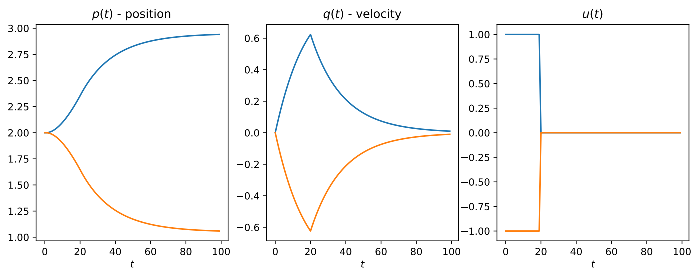

Figure 8: State and input trajectories for a step input.

2. Add a base controller and simulate the vehicle starting from an initial condition different from the equilibrium point, with disturbances sampled from $\mathcal{N}(0, 0.01)$.

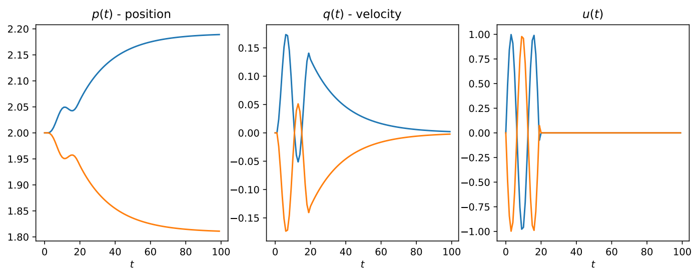

Figure 9: State and input trajectories for a sinusoidal input.

Solution: One solution is implemented in `robot_sol_base.py` using $k_1 = k_2 = 1$. Figure 10 shows the time evolution of the state trajectories of the robot.

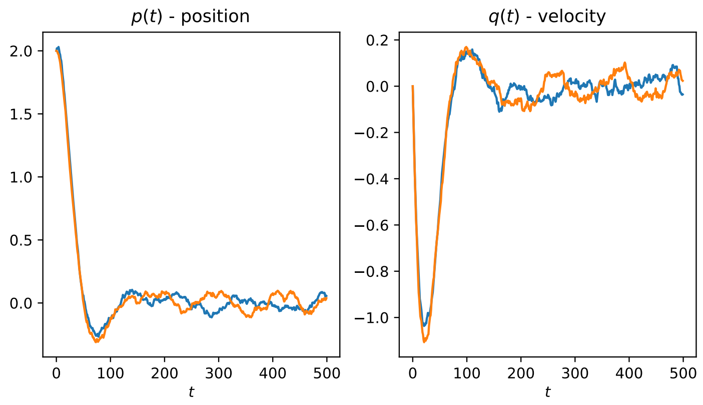

Figure 10: State and input trajectories of system with base-controller

Note: Thanks to the use of the pre-stabilizing controller (9), one can show that $\boldsymbol{\mathcal{F}}(\mathbf{u}, \mathbf{w}) \in \mathcal{L}_2$.

2.2 Objective

The goal of the performance-boosting policy is to enforce additional desired behaviors, on top of stability. In this exercise, we focus on avoiding collisions with an obstacle. We consider that the vehicle, modeled as a point mass, is a circular element with radius r_r, and the obstacle has a radius of r_{obs}. Furthermore, assume that the obstacle is placed at $p_{obs} \in \mathbb{R}^2$.

We select a loss function $L(x_{T:0}, u_{T:0})$ as the sum of stage costs $l(x_t, u_t)$, that is,

$$l(x_t, u_t) = l_{traj}(x_t, u_t) + l_{obs}(x_t)\,, \tag{11}$$

where $l_{traj}(x_t, u_t) = x_t^\top Q x_t + u_t^\top R u_t$ with $Q \succeq 0$ and $R \succ 0$. The function l_{traj} penalizes the distance of the robot to its target(s) and the amount of used energy, and $l_{obs}(x_t)$ penalizes collisions with obstacles. The latter is a barrier-like function:

$$l_{obs}(x_t) = \begin{cases} \alpha_{obs}(d_t + \epsilon)^{-2} & \text{if } d_t \leq D_{\text{safe}}, \\ 0 & \text{otherwise}, \end{cases} \tag{12}$$

and where $\epsilon > 0$, $d_t = |p_t - p_{obs}|_2$, and $D_{safe} = 1.2(r_r + r_{obs})$.

3. Implement and train a PB controller for the pre-stabilized system of point 2.

   Hint 1: use the already defined `PerfBoostingController` class.

   Hint 2: the class `RobotsLoss` provides some useful cost functions that can be used during the training.

Solution: One solution is implemented in `robot_sol_pb.py` using $k_1 = k_2 = 1$. The initial position of the robot is $\begin{bmatrix} 2 & 2 \end{bmatrix}$, and it must go to the origin while avoiding a grey obstacle. Figure 11 shows the trajectory on the xy-plane of the robot after training.

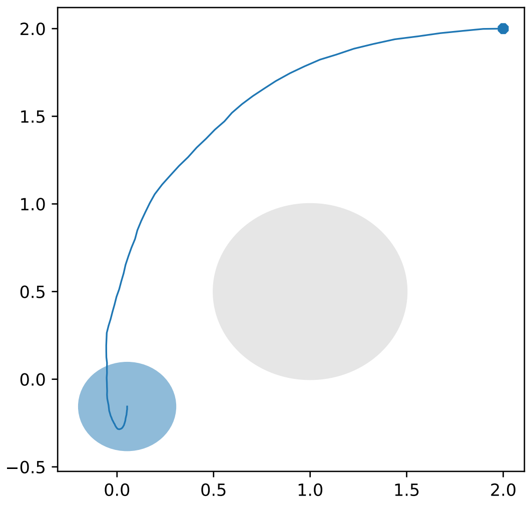

Figure 11: One trajectory obtained after training a PB controller.

2.3 Temporal Logic formulations

In the previous exercises, we have always implemented cost functions which can be written as a summation in time. However, we can also implement more general cost functions which do not rely on time averaging.

In the folder `experiments/robot_TL`, we implemented a training procedure that uses a different cost function. This is in the form of a Truncated Linear Temporal Logic (TLTL) formulation [1, 2]. Check the algorithm and explain why it is a possible candidate solution.

Solution: The algorithm addresses a TLTL cost function. This function promotes:

- always go to the goal (l_1);
- always avoid big inputs (l_2);
- always avoid the obstacle (l_3);
- if goal is visited, don't visit it again (i.e., visit it only at the end!) (l_4).

These items are promoted by different max/min terms (l_1, l_2,l_3,l_4) that then are evaluated together. Since our objective is to minimize the cost function, all these terms are then condensed in a cost function given by

$$L_{TL} = \max(l_1, l_2, l_3, l_4)\,, \tag{13}$$

where

$$l_1 = \max_{t\in[0,T]} d_t^{\text{goal}}\,, \tag{14}$$

$$l_2 = \max_{t\in[0,T]} u_t\,, \tag{15}$$

$$l_3 = \min_{t\in[0,T]} \big(\max(r_{obs} - d_t^{obs}\,, 0\big) \tag{16}$$

$$l_4 = \min_{t\in[0,T]} \max\Big(-(\epsilon - d_t^{\text{goal}})\,, \min_{\tilde{t}\in[t+1,T]} -(\epsilon - d_t^{\text{goal}})\Big)\,. \tag{17}$$

Here, d_t^{goal} and d_t^{obs} are the distances to the goal and the obstacle at step t, respectively. The constant $r > 0$ represents the minimal distance for avoiding the obstacle and $\epsilon > 0$ takes a small value.

Figures after training: Figure 12 and 13 show an example of trajectory after training

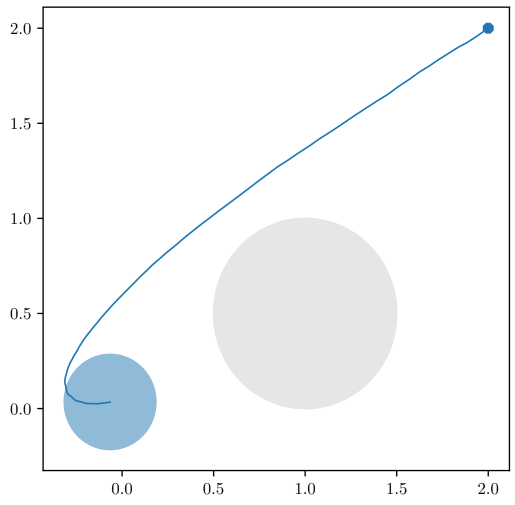

Figure 12: One trajectory in the xy-plane obtained after training a PB controller with TLTL cost.

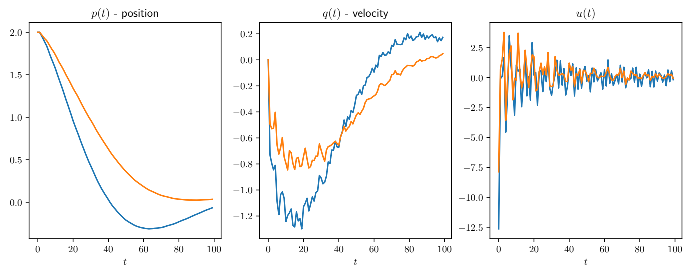

Figure 13: Trajectories of state and input obtained after training a PB controller with TLTL cost.

3 System identification of an input-output water dynamical water tank model

3.1 Model description

We consider the same water tank of the previous exercise, as shown in Figure 14, where the input given to the system is the inflow of water needed to achieve a certain water level. More in detail, we consider the dynamics:

$$\dot{x} = \frac{a}{x+b}(\sqrt{u} - \sqrt{x}). \tag{18}$$

The input given to the system is such that, for a constant $u > 0$, the system converges to u itself. Upon discretization via forward Euler and adding an additive process noise term, we get

$$x_{k+1} = x_k + \frac{ha}{x_k + b}(\sqrt{u_k} - \sqrt{x_k}) + w_k. \tag{19}$$

This noise can be interpreted as a disturbance introduced by the discretization process or as an error due to model mismatch.

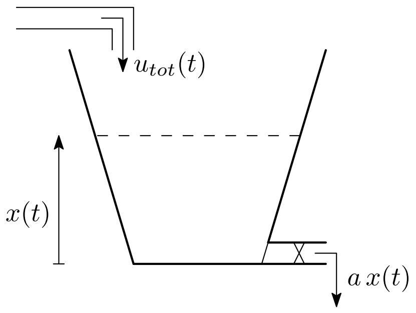

Figure 14: Water tank scheme

The system model is given by:

a	0.6
b	0.5
$\bar{x}$	0.2
h	0.1
N	200

Table 2: Parameters for the water tank system

3.2 Objective

Our goal is to identify the tank system from input/output pairs $(\tilde{u}, \tilde{x})$, where the output corresponds directly to the system state, i.e., the water level in each tank. A key requirement is that the identified model not only fits the data but also preserves the stability properties of the true system. This ensures that the learned model remains a stable dynamical system, maintaining physically meaningful behavior over time.

3.3 Exercises

3.3.1 Simulate the model and generate input/output pairs

We need to generate input/output pairs to create the dataset that we will use to learn the model. The inputs should be sufficiently exciting in order to properly excite the system and allow for an effective identification. In the script `tank_dataset_sysid.py` we find the discretized model of the tank, as well as functions that generate two different types of input: piecewise constant and sinusoidal ones. By running this script, we generate a default number of 600 input/output trajectories, each of length $T = 200$. 400 of these will be used for training the model, while the other 200 for validation. The noise in this case is a white Gaussian process $w_k \sim \mathcal{N}(0, 0.02)$. The plot will be shown by running `run_tank_sysid.py`

1. Play around with the parameters of the model and the inputs, and generate the data set that we will use for training.

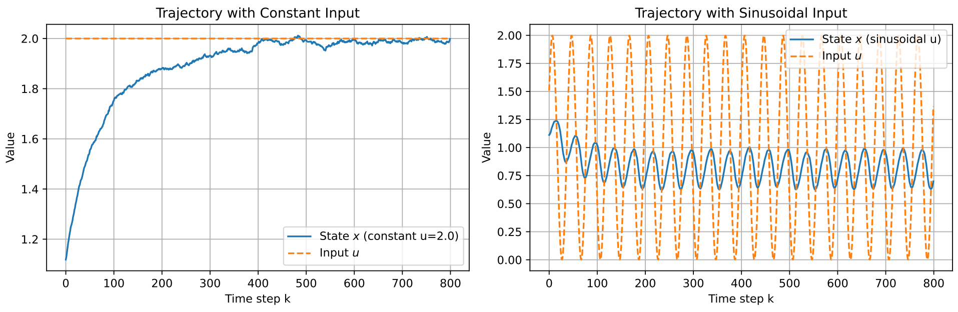

Figure 15: Output evolution for a constant and sinusoidal input.

3.3.2 System identification

We now want to learn the tank model by fitting an $\mathcal{L}_2$ stable parametrized operator $\mathcal{M}_\theta$. More in detail, given N training trajectories and a time horizon of T, we want to solve

$$\begin{cases} \min\limits_{\theta} & \dfrac{1}{NT}\sum\limits_{s=1}^{N}\sum\limits_{k=1}^{T}\|\tilde{y}_k^s - y_k^s\|_2^2 \\ \text{s.t.} & y_k^s = \mathcal{M}_\theta(\tilde{u}_{k:0}^s), \quad \forall s = 1, \dots, N \end{cases} \tag{20}$$

4. Find a solution for this problem (numerically) by using the generated dataset of trajectories $(\tilde{u}, \tilde{x})$ and using different families of operators $\mathcal{M}_\theta$ such as RENs, SSMs, RNNs, etc... The python code that implements these models can be found in `experiments/tanks_sysid/Models_sysid.py` while the training procedure is in `experiments/tanks_sysid/run_tank_sysid.py`.
5. Compare the different results in terms of accuracy in training and validation and produce plots showing how the trained model compares against validation data.
6. Try to change the hyperparameters of each operator (number of layers for SSMs etc...), start from small models and then increase the complexity.

Solution:

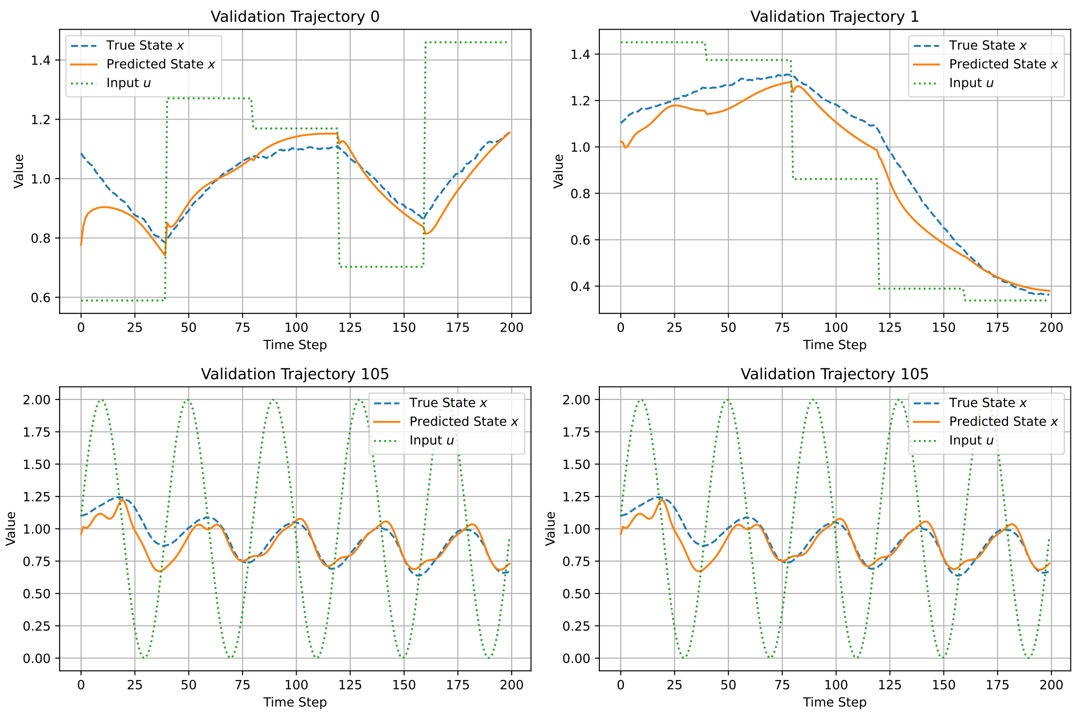

Figure 16: Validation performance for 1-layer SSM (540 parameters) after 1500 epochs

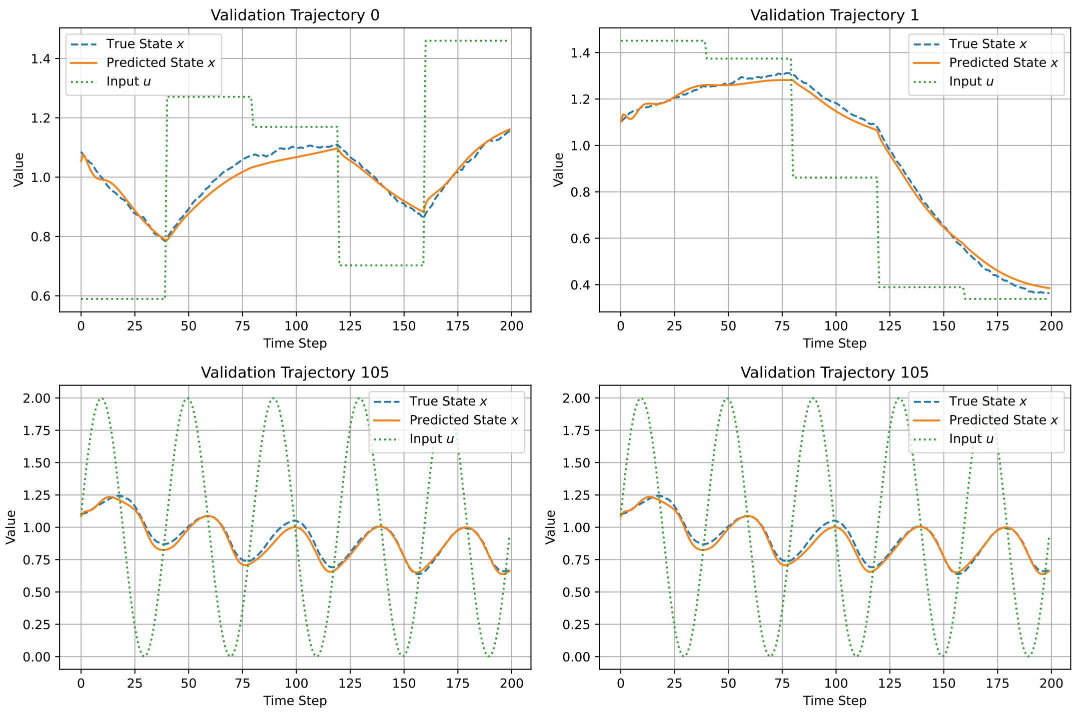

Figure 17: Validation performance for 3-layer SSM (1620 parameters) after 1500 epochs

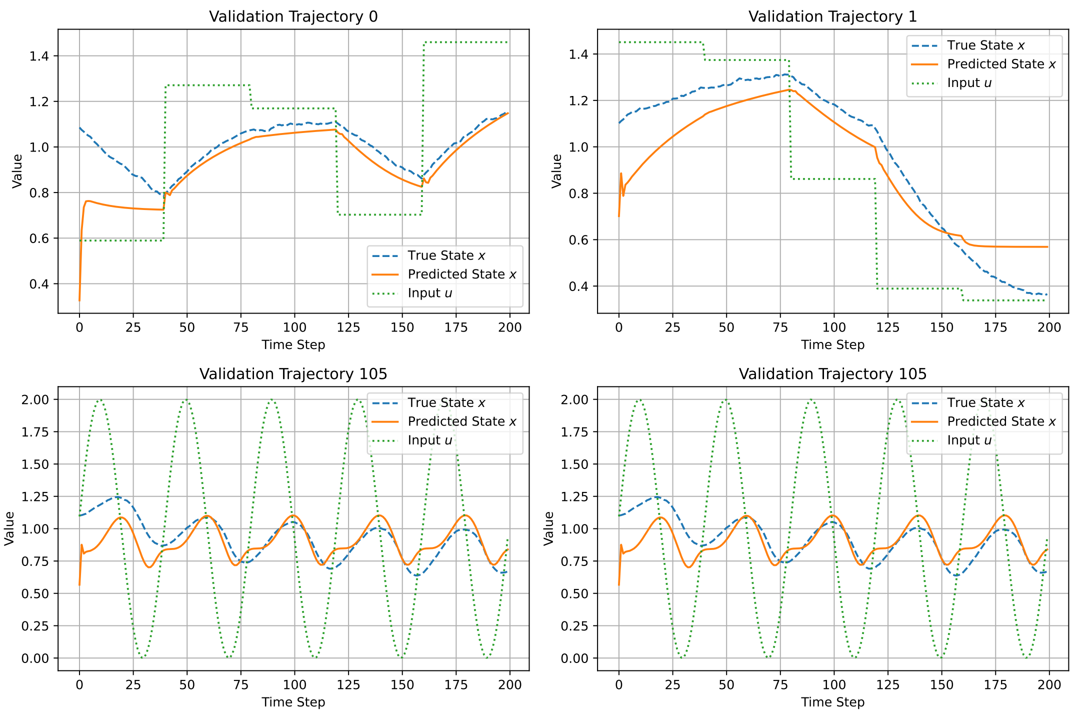

Figure 18: Validation performance for contractive REN (1486 parameters) after 1500 epochs

A Code for tank system

The tank system is implemented in the `TankSystem` class, located in the file `plants/tanks/tank_sys.py`. This class defines the system dynamics and provides several key methods:

- `TankSystem.__init__`: This method initializes the tank system, setting its parameters (e.g., a, b, ...) and defining the nominal equilibrium state.
- `TankSystem.dynamics`: This method computes the right-hand side of (18), which defines the system's continuous-time dynamics.
- `TankSystem.forward`: This method computes the next state of the system given its current state, input, and process noise. It integrates the dynamics (given by the `TankSystem.dynamics` method) using the forward Euler integration scheme. Note that this method is invoked automatically when calling an instance of `TankSystem` (i.e., `my_tank(x, u, w)`) because `TankSystem` inherits from `nn.Module`, which overrides the `__call__` method to internally execute `forward`.
- `TankSystem.rollout`: This method simulates the closed-loop system over a given time horizon. It requires a controller and a sequence of disturbances as inputs. Internally, it calls `TankSystem.forward` to update the system state at each time step. It uses the `TankSystem.forward` method and outputs the state and input trajectories.

Since the tank system is only defined for $x \geq 0$, the `TankSystem.forward` method includes a saturation mechanism: if the computed next state is negative, it is clipped to zero. Similarly, as the system input must be nonnegative ($u_{tot} \geq 0$), the input is filtered before being applied to the system.

B Code for performance-boosting controller

The performance-boosting controller is implemented in the `PerfBoostController` class, located in the file `controllers/PB_controller.py`. Below, we describe its main components:

- `PerfBoostController.__init__`: This method initializes the controller, setting its internal parameters and defining the neural network model used for the $\mathcal{L}_2$-stable operator. It supports two architectures: RENs (Recurrent Equilibrium Networks) and Deep SSMs (State-Space Models). As the controller has a copy of the system, this function requires as parameter `noiseless_forward` the dynamics of the dynamical system to be controlled.
- `PerfBoostController.forward`: This method implements one forward step of the controller. Given an input measurement (e.g., the state of the plant), it computes the next control action. The process follows these steps:

  1. Compute the noise-free input using the `noiseless_forward` function.
  2. Reconstruct the disturbance affecting the system.
  3. Pass the disturbance through the $\mathcal{L}_2$-stable operator (REN or SSM) to compute the control output.
  4. Update the internal states and increment the time step.

  Since `PerfBoostController` inherits from `nn.Module`, calling an instance of the controller (e.g., `my_controller(input_t)`) automatically invokes the `forward` method.
- `PerfBoostController.reset`: This method resets the internal states of the controller, setting the time to zero and reinitializing the stored input and output values. Additionally, it resets the internal states of the chosen $\mathcal{L}_2$-stable model.

Since the controller operates in a closed-loop setting, its input corresponds to the measured state of the plant, while its output determines the control action applied to the plant.

C Code for robot system

The `RobotsSystem` class implements a dynamical system representing a robot that can move in the horizontal plane. The vehicle has nonlinear dynamics, where the nonlinearity arises from speed-dependent friction. Below, we describe its key components:

- `RobotsSystem.__init__`: This method initializes the robot system, by setting its parameters. Particularly, parameter $k \geq 0$ is the gain of the pre-stabilizing controller. Note that setting $k = 0$ removes the pre-stabilizing controller.
- `RobotsSystem.noiseless_forward`: Computes the next state of the system without process noise. It applies the state transition model.
- `RobotsSystem.forward`: Implements the system dynamics, incorporating process noise. Given the current state x, input u, and noise w, it returns the next system state.
- `RobotsSystem.rollout`: Simulates the system evolution over time under a given controller. The simulation can be run in training mode (with gradient tracking) or evaluation mode (disabling gradient tracking for efficiency).

Installing tools for Python development

In this section, we are concerned with the Python installation, the required libraries, and the configuration of our development environment using PyCharm.

Note: If you are already familiar with development environments, feel free to use your preferred one (e.g., Visual Studio Code (VS Code)). *Note:* If you use VS Code, you need to add the path manually. One way to do this is by adding a `base_folder.pth` file in `venv/Lib/site-packages` containing the complete path, e.g., `C:/ .../NNs-for-OC`.

1. **Installing Python (min 3.10)**
   If Python is not already installed on your computer, you can go to the official Python website and download the installer by following the steps indicated in their website.

2. **Verify the installation**
   Open a terminal and type:

   ```
   python --version
   ```

   This command should display the installed Python version (minimum Python 3.10), confirming a successful installation.

3. **Installing PyCharm as an Integrated Development Environment (IDE)**
   For effective coding, we recommend installing PyCharm. This IDE provides useful features such as syntax highlighting, debugging, and version control. Please refer to its respective website for installation instructions. Note the following instructions are tested for the Professional version.

   (a) Go to the official JetBrains website: https://www.jetbrains.com/pycharm/download/.
   (b) Choose the appropriate version:
       - The **Professional** version (paid) offers advanced features.
       - The **Community** version (free) is suitable for basic Python development.
   (c) Download the installer suitable for your operating system (Windows, macOS, or Linux).
   (d) Run the installer and follow the on-screen instructions.
   (e) Once installed, launch PyCharm.

4. **Clone the repository in a new PyCharm project**
   We will clone the repository that contains all the material for the exercises of this course and create a new project.

   (a) Open PyCharm.
   (b) Click **Git** → **Clone**.
   (c) Select **Repository URL**.
   (d) Choose the directory where you want to clone the repository.
   (e) In the **URL** field, paste the URL of the repository `http://github.com/DecodEPFL/NNs-for-OC`.
   (f) Click **Clone** and wait for the process to complete.
   (g) Once cloned, the project will open automatically in PyCharm.

5. **Setting a virtual environment [1] and installing the required packages**

   - **Automatic setting of the virtual environment.**
     If the IDE asks to create a virtual environment select the desired location for the virtual environment (e.g. same folder of the project) and click ok.

[1] A **Python environment** is an isolated workspace that contains a specific Python version and a set of packages. It allows you to manage dependencies for a project without interfering with other projects. In PyCharm, you do not need to activate the environment manually every time you open the IDE—PyCharm automatically activates the correct environment for your project. Moreover, once you install a package in the environment, it becomes immediately available without the need for reactivation.

- **Manual setting of the virtual environment.**
  Open the PyCharm terminal.
  (a) Look at the left panel of the PyCharm window.
  (b) Click on **View** → **Tool Windows** → **Terminal**.
  (c) A terminal window will appear at the bottom of the PyCharm interface.
  (d) You can now execute commands just as you would in a regular command-line interface.

  Run the following line:

  ```
  python setup.py
  ```

  Note: If `python` *does not work, try with* `python3`.
  This creates a new virtual environment (`venv`) in our root folder of the project as well as installs the required packages: `numpy`, `matplotlib`, and `torch`...

6. Once the installation is finished, we need to indicate PyCharm where our `venv` is:
   (a) Click **PyCharm** → **Settings** (or **File** → **Settings** in Windows)
   (b) Go to **Project: NNs_for_OC** → `Python Interpreter`.
   (c) Click `Add Interpreter`→ **Add Local Interpreter..** . (Note, some PyCharm versions have a gear icon to add the interpreter)
   (d) Choose `Virtualenv Environment` on the left.
   (e) Select `Existing environment` and click the three dots `...` to browse for your `virtualenv`. Choose the Python executable from the `bin` (macOS) or `Scripts` (Windows) folder.
      - Windows: `NNs_for_OC\venv\Scripts\python.exe`
      - macOS \Linux: `NNs_for_OC\venv\bin\python`

      Note: if you have more than one python executable try choosing the latest one. If it does not work, try with the previous ones.
   (f) Click `OK` or `Apply` to set it as your project's interpreter.
   (g) Check the bottom right corner, near a small padlock icon, to see if the project's interpreter is set to the correct virtual environment.

7. Test the installation.
   Open and run (click the play button on the top right side of the IDE) the file:

   ```
   test_installation.py
   ```

   If the play button is not available, navigate to **View** → `Tool Windows` → `Run` and then click the run button on the left.

References

[1] X. Li, C.-I. Vasile, and C. Belta, "Reinforcement learning with temporal logic rewards," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 3834–3839.

[2] K. Leung, N. Aréchiga, and M. Pavone, "Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods," *The International Journal of Robotics Research*, vol. 42, no. 6, pp. 356–370, 2023.