Assignment 03: Learning for Control

Andrea Del Prete[*] - andrea.delprete@unitn.it

December 2024

1 Description

As final project for this course, students can choose between three options:

- A) Learning a Value function with a Neural Network, and using it as terminal cost in an optimal control problem.
- B) Learning the Viability kernel of a system and using it as terminal constraint in an MPC framework to ensure recursive feasibility.
- C) Learning a Value function, then learning a policy by minimizing the corresponding Q function, and finally using this policy to warm start an optimal control solver.

Contrary to the other assignments, this time you have to write all the code (almost) from scratch.

2 Submission procedure

You are encouraged to work on the assignments in groups of 2 people. **If you have a good reason to work alone, then you can do it, but this has to be previously validated by one of the instructors**. Groups of more than 2 people are not allowed. The mark of this assignment contributes to 25% of your final mark for the class (i.e. 7.5 points out of 30). The project discussion then contributes to another 20% (i.e. 6 points).

When you are done with the assignment, please submit a single compressed file (e.g., zip). **The file name should contain the surnames of the group members**, and it must contain:

- A pdf file with a detailed description of the work, the **names and ID number** of the group members; you are encouraged to include plots and/or numerical values obtained through simulations. **This pdf does not need to be long. Four to six pages of text should be enough. You can then add other pages for plots and tables.**
- The complete orc folder containing all the python code that you have developed.

If you are working in a group (i.e., 2 people) only one of you has to submit.

Submitting the pdf file without the code is not allowed and would result in zero points. Your code should be consistent with your answers (i.e. it should be possible to produce the results that motivated your answers using the code that you submitted). If your code does not even run, then your mark will be zero, so make sure to submit a correct code.

[*]Learning and Optimization for Robot Control, Industrial Engineering Department, University of Trento.

3 Common Tools

3.1 Solving Optimal Control Problems (OCPs)

All the three projects described above (and detailed in the following) begin by solving a series of OCPs. We suggest students to use CasADi[1], which we have already used during our lab sessions on optimal control.

When using CasADi, the system dynamics can be explicitly written down in the code, or computed using a dynamic library compatible with CasADi. In our lab sessions, we used the library ADAM, which can read the robot description in URDF format (similar to an XML file), and provide CasADi functions for computing the robot dynamics. In the final project you'll have to work with the single and double pendulum. The dynamics of a single pendulum is trivial, therefore you could decide to write it down directly in the code.

The dynamics of a double pendulum is instead rather complex. Luckily, you can find a URDF file describing a double pendulum in *example-robot-data*, the Python package that we have already used for getting the UR5 robot description. This means that in our scripts you should simply replace this line:

```
robot = load("ur5")
```

with this line:

```
robot = load("double_pendulum")
```

3.2 Training Neural Networks

All projects require training neural networks. Students are free to use any library for training neural networks, such as *PyTorch*, *Tensor Flow/Keras* or *JAX/FLAX*. However, we suggest using *PyTorch* because it is easier to integrate with *CasADi* using the library *L4casadi*[2]. Consider that in project A and B the neural network needs to be embedded inside a CasADi optimization problem (either in the cost or in the constraints).

In the code folder you can find a file `neural_network.py` containing the `NeuralNetwork` class, which represents a simple 3-layer neural network with *PyTorch*. By default, this network uses `tanh` activation functions, which typically work better in optimization problems than `ReLU` activation functions. The method `create_casadi_function` of the `NeuralNetwork` class uses *L4Casadi* to create a CasADi function that replicates the computation of the neural network.

If you prefer not to use *L4Casadi*, an alternative is to re-implement the neural network function using CasADi data types and operators. As an example, this is how a student of mine implemented in casadi a network using ReLU activation functions:

```
def nn_decisionfunction(params, x):
    out = x
    it = 0

    for param in params:
        param = SX(param.tolist())

        if it % 2 == 0:
```

[1]https://web.casadi.org/

[2]https://pypi.org/project/l4casadi

```
            out = param @ out # linear layer
        else:
            out = param + out # add bias
            out = fmax(0., out) # apply ReLU function

        it += 1

    return out
```

3.3 Parallel Computation

To make your code run faster you can try to parallelize it, so that it can exploit multiple CPU cores. To do that, you can use the Python library `multiprocessing`.

4 Project A — Learning a Terminal Cost

The aim of this project is to learn an approximate Value function that can then be used as terminal cost in an MPC formulation.

First, we have to solve many OCP's with a fixed horizon N, starting from different initial states (either chosen randomly or on a grid). To keep things simple, I suggest you to use unconstrained problems, without inequality constraints and without terminal cost:

$$\begin{aligned} \underset{X,U}{\text{minimize}} \quad & \sum_{i=0}^{N-1} l(x_i, u_i) \\ \text{subject to} \quad & x_{i+1} = f(x_i, u_i) \quad i = 0 \dots N-1 \\ & x_0 = x_{init} \end{aligned} \tag{1}$$

For every solved OCP, we should store the initial state x_{init} and the corresponding optimal cost $J(x_{init}) = \sum_{i=0}^{N-1} l(x_i, u_i)$ in a buffer. Then, we should train a neural network to predict the optimal cost J given the initial state x_{init}.

Once such a network has been trained, we must use it as a terminal cost $J(x_{final})$ inside an OCP with the same formulation, but with a different horizon M (typically shorter):

$$\begin{aligned} \underset{X,U}{\text{minimize}} \quad & \sum_{i=0}^{M-1} l(x_i, u_i) + J(x_M) \\ \text{subject to} \quad & x_{i+1} = f(x_i, u_i) \quad i = 0 \dots M-1 \\ & x_0 = x_{init} \end{aligned} \tag{2}$$

We should be able to empirically show that, thanks to the introduction of the terminal cost, this problem is now approximately equivalent to a problem with horizon $N + M$, and therefore better than a problem with horizon M and without terminal cost. To show this, you should compare the following three MPC formulations:

1. horizon M without terminal cost;
2. horizon M with neural network function J as terminal cost;
3. horizon $N + M$ without terminal cost.

You should expect to see the last two formulations performing similarly, and better than the first one. The comparison should not be limited to a single test from a single initial state, but rather be statistically significant, including many tests from different initial states. As a comparison metric you may use the cost of the state-control trajectories obtained when controlling the robot with the chosen MPC controller (i.e., do not use the cost of the predicted trajectories).

Here are some tips to select meaningful values of N and M. For M, it should not be too large, otherwise the solution of the problem may not change significantly when you increase the horizon. Ideally, you want to set M as small as possible. You can use a trial-and-error approach. Pick a value of M (e.g., 10) and solve a few OCP's starting from random initial states with horizon M and a longer horizon (e.g., $2M$). Comparing the solutions obtained you should see a significant difference in the first part of the computed trajectories. If you do not, then it means that M is too large.

For choosing N, it should not bee too small, otherwise the computed terminal cost may have too little effect. Ideally, you want to set N as large as possible. However, the larger N, the longer it will take to solve the OCP's to generate the training data for the network. Therefore, you should choose N so that the resulting computation time is acceptable for you.

For this test, we suggest to start using a single pendulum, and then moving to a double pendulum. You may need to use different values of N and M when you change system and/or cost function.

5 Project B — Learning a Terminal Constraint Set

The aim of this project is to learn an approximate control-invariant set that can be used as terminal constraint in an MPC formulation.

More in detail, we want to learn the N-step backward reachable set of $\mathcal{S}$, where $\mathcal{S}$ is the set of equilibrium states:

$$\mathcal{S} = \{x \in \mathcal{X} : \exists\, u \in \mathcal{U}, x = f(x, u)\}$$

For a robot manipulator, this set typically contains all the zero-velocity states:

$$\mathcal{S} = \{(q, \dot{q}) : q_{min} \leq q \leq q_{max}, \dot{q} = 0\}$$

To understand if a state x_{init} belongs to the N-step backward reachable set of $\mathcal{S}$ we can solve the following OCP:

$$\begin{aligned} \underset{X,U}{\text{minimize}} \quad & 1 \\ \text{subject to} \quad & x_{i+1} = f(x_i, u_i) \quad i = 0 \dots N-1 \\ & x_{i+1} \in \mathcal{X}, u_i \in \mathcal{U} \quad i = 0 \dots N-1 \\ & x_0 = x_{init} \\ & x_N \in \mathcal{S} \end{aligned} \tag{3}$$

If this OCP has a solution then the state x_{init} belongs to the set, otherwise it does not. By sampling random values of x_{init} inside $\mathcal{X}$ and solving this OCP for each of them, we can generate a dataset of pairs (x, label), where label is either 0 or 1 to indicate whether the associated state belongs to the set or not.

After generating the data, we have to train a neural network to classify states as either inside, or outside the set. Then, we can use such this network for constraining the terminal state of an MPC formulation. Thanks to this extra constraint, we should be able to show that the MPC problem remains recursively feasible in situations where feasibility would be lost without using such a constraint.

For this test, we suggest to start using a single pendulum, and then moving to a double pendulum. To make the problem challenging in terms of constraint satisfaction, we suggest to use a cost function that pushes the system to reach a joint configuration that is close to the joint limits, while penalizing very little the joint velocities and torques.

6 Project C — Actor-Critic Learning

The first part of this project is similar to project A. First, we have to solve many OCP's starting from different initial states (either chosen randomly or on a grid). For every solved OCP, we should store the initial state x and the corresponding cost $V(x)$ in a buffer. Then, we should train a neural network (called "critic") to predict the cost V given the initial state x. After the critic has been trained, we should train another neural network, called "actor", that approximates the greedy policy with respect to the critic. To train this network we should minimize the action-value function corresponding to the critic, that is:

$$\pi(x) = \arg\min_u \, l(x,u) + V(f(x,u)), \tag{4}$$

where l is the running cost, V is the critic, and f is the system dynamics. To perform this minimization, one option is to do it with respect to the vector u, obtaining the (locally) optimal value u^*, which you can store in a dataset containing state-action pairs. Then, you can train the actor using supervised learning, to predict u^* given x. Otherwise, you can perform the minimization directly with respect to the weights w of the actor network, which is closer in spirit to what RL algorithms do:

$$w^* = \arg\min_w \, l(x, \pi(x;w)) + V(f(x, \pi(x;w))), \tag{5}$$

After training the actor, we can use it either to directly control the system, or to compute an initial guess (for both state and control) to provide to the OCP solver. In both cases we can measure the performance of the actor by evaluating the resulting cost of the OCP, and comparing it to the cost of the first batch of OCP's.

For this project, we suggest to start with a simple 1D single integrator, that has the following dynamics:

$$x_{t+1} = x_t + \Delta t \, u_t \tag{6}$$

To make the problem interesting (i.e. non-convex), you can use the following running cost:

$$l(x,u) = \frac{1}{2}u^2 + (x-1.9)(x-1.0)(x-0.6)(x+0.5)(x+1.2)(x+2.1) \tag{7}$$

Once things are working with this system, we can move to a more complex double integrator:

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + \begin{bmatrix} 0.5\Delta t^2 \\ \Delta t \end{bmatrix} u_t \tag{8}$$

7 Suggestions

When preparing the final report, make sure to account for the following tips.

1. Mathematically describe the optimal control problem formulation.
2. Describe the structure of the neural network (e.g., number of layers, number of neurons per layer, activation functions).

3. Report the problem constraints.
4. Include plots of some trajectories (position, velocity, torque) obtained controlling the system with the optimal policy.
5. Report the value of the hyper-parameters of the algorithm (e.g., learning rate, mini-batch size, number of data points).
6. Report the training time.
7. Include plots (color maps) of the Value and policy function whenever possible.
8. Include videos of the robots controlled with the policy found.
9. Include a discussion of potential improvements you did not have time to implement/test.