Optimization for Learning - FRTN50

Extra Credit Assignment: Differential Neural Networks

Written by: Mattias Fält

Latest update: October 18, 2019

Introduction

Background

In this hand-in you will try to estimate differential equation using a techniques from this course with the purpose of identifying the dynamics of a quad-rotor. The high level model is a continuous time differential equation

$$\dot{x} = f(x).$$

There are many techniques from the field of system identification that could be used to estimate such a model. In this hand-in we will be using a new approach known as Neural Differential Equations. This approach was presented in 2018 and won the prestigious best paper award at the machine learning conference NeurIPS [1] . The idea is to directly identify the continuous dynamics f using a neural network, instead of, as was previously the norm, a discrete mapping from the initial point $x(0)$ to the states at some fix times $(x(t_1), x(t_2), ...)$. However, learning the function f is not trivial. Even if f could be represented by a neural network, the derivatives $\dot{x}$ are generally not available, so the problem does not reduce to a standard function-approximation problem. Instead, the current guess has to be simulated, and gradient back-propagation (or similar) has to be done trough the differential equation solver. Although the theory on how to do those computations is well know, it is not trivial in practice. But now, because of the speed and versatility of the Julia language, and some very well written libraries in Julia, in particular the differential equations library `DifferentialEquations.jl` and the learning library `Flux.jl`, it turns out to be quite manageable. To simplify some of the details, the package `DiffEqFlux.jl` [2] was released in early 2019 for specifically this task.

Problem formulation

When doing machine learning or system identification, it is important to incorporate as much knowledge and insight as possible into the model before trying to learn it. For example, in image recognition this is done by using convolutional-layers to signify that adjacent pixels are related. In this task, we will not try to learn the full model f from scratch, this would probably be almost impossible,

and require a large amount of data. Instead, we will start with a know mathematical model, learn only the parts that are unknown. The description of the model can be written as

$$\dot{x} = f(x, u, p) + m(x)$$

where f is the dynamics of the model of the quad-rotor that we know, u is the control signal, p is a set of unknown parameters, and m is an additional unknown parts of the model. The goal is then to identify the parameters p, and the unknown function m. This makes it possible to train with a much smaller data-set, and allows us to learn about the real quantities p. A more detailed description of the quad-rotor model can be found in the *Model* section.

Closed loop

Since a quad-rotor is a highly unstable system, it is not possible to send random inputs $u(t)$ and simply observe the output $x(t)$. And even if the inputs $u(t)$ were chosen wisely when generating the data, a non-perfect model (as is the case when training) would result in simulations where the sates quickly go towards infinity. The data is instead gathered "in closed loop" where the control signal is generated by a controller $u(t) = c(x, r(t))$, where r is some known reference trajectory. However, this introduces theoretical problems with identifying exactly how f depends on the input u. To alleviate this, a small noise $r_n(t)$ is added to the control signal to "excite" the system properly, without making it unstable. Moreover, the full state vector $x(t)$ is not measurable. In the case of the quad-rotor, we assume that we can only measure the position in the room, and not the velocities, angles and angular velocities. If we let $y = (x_{y_1}(t), x_{y_2}(t), ...)$ be the states we can measure, the final model can be described as

$$\begin{aligned} \dot{x} &= f(x, u, p) + m(x), \quad x(0) = x_0 \\ y &= (x_{y_1}, x_{y_2}, ...) \\ u &= c(x, r) + r_n \end{aligned}$$

where the signals $y(t), r(t), r_n(t)$ are known, as well as the functions f and c and the initial state x_0. Finally, the control signal is computed at a fixed time points $(t_1, t_2, \ldots, t_k)$ with $t_i - t_{i-1} = \Delta t$, and kept constant until the next time point, known as a zero-order-hold output.

Problem data

Input vectors $r[k]$ and $r_n[k]$ have been randomly generated and applied to the system as zero-order-hold inputs $r(t)$ and $r_n(t)$ at a set time points $(t_0, t_2, \ldots, t_k)$ from $t_0 = 0$, to $t_k = 10$s, and the output $y(t)$ was measured and stored as the vectors $y[k]$ when the system was initiated at x_0.

Tasks

Four different data sets with increasing difficulty has been generated from different models. For each task the data `u0s, rds, rdns, yreals` is available and

corresponds to the initial states $x(0)$ the (discrete) references $r[k]$, the (discrete) input noise $r_n[k]$ and the recorded outputs $(x[k], y[k], z[k])$. Note that the parameters I_x, I_y, I_z change between the different tasks, so you can not reuse the result from one task to the next. As initial guess, you can use $[I_x, I_y, I_z] = [6, 7, 11] \cdot 10^{-6}$. Note: although noise is added to tasks 3 and 4, we assume that the controller still had access to the true positions in the feedback for all tasks.

Task 1

The reference system f was simulated with $m(x) = 0$. The task is to identify the inertia parameters I_x, I_y, I_z. The problem data for this task is available through the function `u0s, rds, rdns, yreal = data1()`. The data reference noise is set to zero in this first task, i.e. $r_n[k] = 0$.

Task 2

The reference system f was simulated with a drag model $m(x)$ added to the velocities u, v, w. The task is to identify the inertia parameters I_x, I_y, I_z and the drag model. Data is available with `data2()`.

Task 3

The reference system f was simulated with a drag model $m(x)$ added to the velocities u, v, w. The model $m(x)$ is slightly different form in Task 2. Only noisy measurements are available now, i.e $y[k] = (x_{y_1}[k] + n_1[k], x_{y_2} + n_2[k], ...)$, where $n_i[k]$ is zero mean gaussian noise with standard deviation 3cm. The task is to identify the inertia parameters I_x, I_y, I_z and the drag model. Data is available with `data3()`.

Task 4

The reference system f was simulated with a drag model added to the velocities u, v, w. Additional constant wind was present in each of the x, y, z directions (in the global frame) during simulation which also affects the velocities u, v, w. The drag model is the same as in Task 3, but you need to understand of how the wind affects it if you chose to re-use the model you identified in Task 3. Only noisy measurements are available now, i.e $y[k] = (x_{y_1}[k] + n_1[k], x_{y_2} + n_2[k], ...)$, where $n_i[k]$ is zero mean gaussian noise with standard deviation 3cm. The task is to identify the inertia parameters I_x, I_y, I_z and the wind + drag model. If you use your understanding of the model, it should be possible to estimate a constant wind vector $[w_x, w_y, w_z]$. Data is available with `data4()`.

Report

There are no specific requirements on what you have to try or how you identify your model. There are many different possible things to try such as: different optimizers, hyper-parameters, network structure and size, pre-processing of data, validation sets, regularizations and so on. You do not have to try everything, or solve all of the tasks, the important thing is that you discuss and

reason about the choices you made, and how good you think your predictions are.

In addition to reporting the identified parameters I_x, I_y, I_z for each test, and possibly some explanation of the models $m(x)$, you should submit predictions for each of the tasks you solve. The function `u0s, rds, rdns = data_test1()` (and `data_test2()` and so on) provides you with inputs for which you can simulate the system, but contains no reference output. The output (x, y, z) of these simulations with your model should be saved using `save_trajectories(outputs, taskid)`, and submitted to us together with your report. This makes it possible for us to test how well your model generalizes to unseen data. You should also submit the code `model.jl` you used to train your models.

Code

The four files `model-ex.jl`, `model.jl`, `data.jl` and `data.jld2` contain everything you need to get going. `model-ex.jl` contains a (relatively) simple example of a model and the surrounding code to train and evaluate it. You can use this as a reference for how the code and training on the quad-rotor can be done.

`model.jl` contains the quad-rotor dynamics, the controller and a set of useful functions to get started with the task. `data.jl` constrains the functions for loading the training data from `data.jld` as well as for saving the data you need to submit. The data is loaded as `u0s, rds, rdns, yreals = data2()`, where the first index correspond to each simulation. I.e. `u0s[1]` contains the initial state for the first simulation, `rds1[1][2]` contains the reference vector at the second time step of the first simulation, and so on.

Model

This section contains the quad-rotor dynamics. These are not necessary to understand, but as with any real problem, it is always useful with a better understanding of the model. This is particularly true for the last task. The dynamics can be described by the differential equation

$$
\begin{aligned}
\dot{\phi} &= p + r\cos(\phi)\tan(\theta) + q\sin(\phi)\tan(\theta) \\
\dot{\theta} &= q\cos(\phi) - r\sin(\phi) \\
\dot{\Psi} &= r\cos(\phi)/\cos(\theta) + q\sin(\phi)/\cos(\theta) \\
\dot{p} &= rq(I_y - I_z)/I_x + \tau_x/I_x \\
\dot{q} &= pr(I_z - I_x)/I_y + \tau_y/I_y \\
\dot{r} &= pq(I_x - I_y)/I_z + \tau_z/I_z \\
\dot{u} &= rv - qw - g\sin(\theta) + f_{wx}/m \\
\dot{v} &= pw - ru + g\sin(\phi)\cos(\theta) + f_{wy}/m \\
\dot{w} &= qu - pv + g\cos(\theta)\cos(\phi) - f_t/m + f_{wz}/m \\
\dot{x} &= R_1(\phi, \theta, \Psi)[u, v, w] \\
\dot{y} &= R_2(\phi, \theta, \Psi)[u, v, w] \\
\dot{z} &= R_3(\phi, \theta, \Psi)[u, v, w]
\end{aligned}
$$

where ϕ, θ, Ψ are the pitch, roll and yaw Tait–Bryan angles describing the rotation of the quad-rotor (radians), p, q, r are their respective angular-velocities, u, v, w are the velocities of the quad-rotor in its own coordinate system (m/s), and x, y, z are the positions in the global coordinate frame (position in the room) in meters. τ_x, τ_y, τ_z are torques generated by the motors, I_x, I_y, I_z are the inertia around the axes of the quad-rotor, and $R = \begin{bmatrix} R_1(\phi, \theta, \Psi) \\ R_2(\phi, \theta, \Psi) \\ R_3(\phi, \theta, \Psi) \end{bmatrix}$ is the rotation matrix from the quad-rotors coordinate system to the global, given by

$$R = \begin{bmatrix} c_\theta c_\Psi & c_\Psi s_\phi s_\theta - c_\phi s_\Psi & s_\phi s_\Psi + c_\phi c_\Psi s_\theta \\ c_\theta s_\Psi & c_\phi c_\Psi + s_\phi s_\theta s_\Psi & c_\phi s_\Psi s_\theta - c_\Psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\phi c_\theta \end{bmatrix}.$$

where $s_\alpha := \sin(\alpha)$ and $c_\alpha := \cos(\alpha)$. The reverse transformation (from global to local frame) is given by $R^{-1} = R^T$. The 4 control signals $\Omega_1, \Omega_2, \Omega_3, \Omega_4$ (one for each motor) are responsible for the generated torques τ_x, τ_y, τ_z and the force f_t. f_{wx}, f_{wy}, f_{wz} are possible additional forces (in the quad-rotors local coordinate system).

References

[1] Tian Qi Chen et al. "Neural ordinary differential equations". In: *Advances in neural information processing systems*. 2018, pp. 6571–6583. URL: https://papers.nips.cc/paper/7892-neural-ordinary-differential-equations.pdf.

[2] Christopher Rackauckas et al. "DiffEqFlux.jl - A Julia Library for Neural Differential Equations". In: *CoRR* abs/1902.02376 (2019). arXiv: 1902.02376. URL: http://arxiv.org/abs/1902.02376.