Project 3: Neural differential equations, model discovery, and transfer learning

ETH zürich

AI in the Sciences and Engineering
Due date: June 26th, 2024

Training and testing data can be found on the Moodle page: `https://moodle-app2.let.ethz.ch/course/view.php?id=22389`

IMPORTANT INFORMATION

To get ECTS credits for the course you need to submit and obtain a passing grade on **each** of the **three** projects, which are to be completed **individually**. Submitting these projects is the only form of assessment for the course - there are no group projects this year.

This document describes the **third** project. The due date for this project is **26 June 2024**.

This project consists of three tasks on different topics. For each task, you will be asked to train a learning model and provide predictions on a testing set. **The final submission consists of the predictions files, the code and a report of maximum 2000 characters per task** where you should succinctly describe the procedure followed in each task. The submissions have to be collected in a zip folder named as *yourfirstname_yoursecondname_yourleginumber.zip*.

Bonus system

- Each project will have its own **bonus, optional** question – you can get full marks on the project (6/6) without touching it.
- Each such question will be scored from 0 to 100 – due to this question being open-ended, we will decide on what a 'reasonable attempt' is based on the submissions of **all** students.
- Getting at least a 'reasonable attempt' on **all** bonus questions across **all** projects will earn you a final grade bonus of 0.5. Note the final grade is still capped at 6.

Task 1

In this task, you will train a neural network to control an inverted pendulum. The inverted pendulum is described in fig. 1.

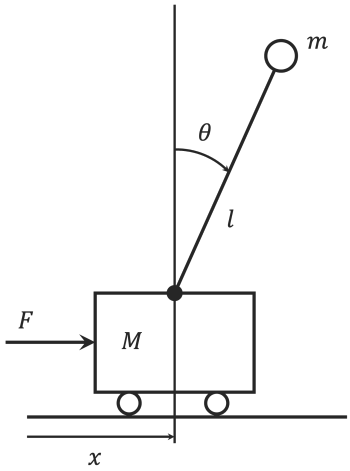

Figure 1: Inverted pendulum

Consider a pendulum of length l and mass m attached to a cart of mass M. Let $x(t)$ be the horizontal position of the cart, and $\theta(t)$ be the angle of the pendulum from the vertical. The equations of motion are derived from Newton's second law and are given by:

$$(M+m)\ddot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta = F \tag{1}$$

$$l\ddot{\theta} - g\sin\theta + \ddot{x}\cos\theta = 0 \tag{2}$$

where $F(t)$ is the external time-varying force applied to the cart, g is the acceleration due to gravity, $\ddot{x}$ is the acceleration of the cart, and $\ddot{\theta}$ is the angular acceleration of the pendulum.

These can be re-written as follows:

$$\ddot{x} = \frac{F - mg\cos\theta\sin\theta + ml\dot{\theta}^2\sin\theta}{M + m - m\cos^2\theta} \tag{3}$$

$$\ddot{\theta} = \frac{g\sin\theta - \ddot{x}\cos\theta}{l} \tag{4}$$

Task 1.1 - Solving the coupled ODE system

Your first task is to write a numerical solver to solve this coupled ODE. That is, given the initial state of the system $\mathbf{s}_0 = (x(t_0), \dot{x}(t_0), \theta(t_0), \dot{\theta}(t_0))$, and the values of M, m, l, and of the external force at each timestep, $\{F(t_0), F(t_1), ..., F(t_N)\}$, solve for $\{x(t_0), x(t_1), ..., x(t_N)\}$ and $\{\theta(t_0), \theta(t_1), ..., \theta(t_N)\}$.

Hint: you can use a standard ODE solver to solve these equations, by solving for the time evolution of the state variables $\mathbf{s} = (x, \dot{x}, \theta, \dot{\theta})$ and using eq. (3) and eq. (4) to update the state at each timestep. You can either define your own solver, or use an ODE solver package, for example `diffrax` in JAX. We recommend using a high-order solver (e.g. RK4 or similar) to avoid numerical error. Also make sure the solver function is written in an autodifferentiation library, so you can use it in the next task below.

Deliverables: produce a plot of $x(t)$ and $\theta(t)$ from $t_0 = 0$ to $t_N = 5$ for the following initial conditions and force function: $M = 1.0, m = 0.1, l = 1.0, g = 9.81, x(t_0) = 0, \dot{x}(t_0) = 0, \theta(t_0) = \pi/4, \dot{\theta}(t_0) = 0, F(t) = 10\sin(t)$.

Task 1.2 - Learning to balance the pendulum

Your second task is to learn to balance the pendulum. That is, given the initial state of the system, find a forcing function $F(t)$ such that $\theta(t) \approx 0$ for $t \geq 0.75t_N$. You should do this by representing the forcing function as a neural network, i.e. $F(t) = NN(t, \phi)$, where ϕ are the network's learnable parameters. You should assume the pendulum starts with the same initial conditions above (aside from the forcing function), and that $t_N = 5$.

Hint: to solve this problem, you need to carry out three steps: 1) incorporate and call the neural network into your ODE solver, 2) define a suitable loss function that matches the ODE solver's output to the desired pendulum's behaviour and 3) train the network using gradient descent (e.g. Adam) on this loss function. A small fully-connected network is a suitable architecture.

Deliverables: produce a plot of $x(t)$, $\theta(t)$, and your learned $F(t)$ from $t_0 = 0$ to $t_N = 5$.

Task 2

The objective of this task is to use a regression method *PDE-FIND* [1] to discover the governing time-dependent partial differential equation (PDE) of an unknown system by using measurements of the solution to the PDE.

The *PDE-FIND* method is able to select, from a large library, the correct linear, nonlinear, and spatial derivative terms, resulting in the identification of PDEs from data. Only those terms that are most informative about the dynamics are selected as part of the discovered PDE. Let us assume that the unknown time-dependent PDE is given in the form of

$$u_t = \mathcal{D}(u, u_x, u_{xx}, u_y, u_{yy}, u_{xy}, ..., x, y, ..., t), \tag{5}$$

where subscripts denote partial differentiation, and we assume the solution is a scalar field, i.e. $u(x, y, ..., t) : \mathbb{R}^d \rightarrow \mathbb{R}^1$. An example of the operator $\mathcal{D}$ is Burgers' equation, given by $\mathcal{D} = -uu_x + \mu u_{xx}$, where μ is a scalar viscosity coefficient.

Suppose we have n observations of the solution to the PDE at many known coordinates in the domain. PDE-FIND begins by first constructing a column vector, $\mathbf{u} \in \mathbb{R}^{\mathbf{n}}$, containing all of the solution values. Next, similar column vectors are constructed which each compute the value of a

possible (linear or non-linear) term in the PDE at each observational point. These column vectors are collected together to form a matrix $\Theta(\mathbf{u}) \in \mathbb{R}^{n \times D}$ of candidate terms in the PDE, where D is the total number of candidate terms, for example

$$\Theta(\mathbf{u}) = \begin{bmatrix} 1 & \mathbf{u} & \mathbf{u}^2 & \mathbf{u}_x & \mathbf{u}\mathbf{u}_x & \dots \end{bmatrix}. \tag{6}$$

Partial derivatives (such as $\mathbf{u}_x$) at each observational point can be estimated in a number of ways. If the observations are on a regular grid, a simple approach is to use finite differences. When the data is noisy, or irregularly spaced, polynomial interpolation can be used. Another approach is to use a neural network to fit the observational data, i.e. train $NN(x, y, ..., t; \theta) \approx u(x, y, t)$ and to estimate derivatives at query points using autodifferentiation.

Given the library of terms, we then assume that the PDE at each point can be written as

$$\mathbf{u}_t = \Theta(\mathbf{u})\xi, \tag{7}$$

where $\xi \in \mathbb{R}^D$ is a column vector of coefficients and each non-zero entry in ξ corresponds to a term in the PDE. It is assumed that the operator $\mathcal{D}$ may be expressed as a sum of a small number of terms (e.g. < 10 terms), which is certainly the case for the PDEs considered here and is widely used in practice. We therefore aim for a **sparse** vector ξ. Note the matrix $\Theta(\mathbf{u})$ must contain all the operators in the unknown PDE, so that the unknown PDE can always be written as a weighted sum of *a few* terms included in it. We require the sparsest vector ξ that satisfies 7 with a small residual.

Solving for ξ simply means solving a (large) linear system. To ensure we learn a sparse ξ, PDE-FIND uses **ridge regression** with hard thresholding (see the reference [1] for the exact method).

Your task: In this folder are 3 files containing observations of the solutions of 3 different PDEs. Your task is to predict the governing PDE for each file.

The files are roughly in order of increasing difficulty. Files 1 and 2 contain measurements of a 1+1D PDE (i.e. $u(x, t)$). File 3 contains measurements of a 2+1D PDE, where the solution is a vector field with two components, i.e. $u(x, y, t)$ and $v(x, y, t)$. In this case the PDE is a set of two coupled equations of the form

$$u_t = \mathcal{D}_1(u, u_x, v, v_x, u_{xy}, uv, ...), \tag{8}$$
$$v_t = \mathcal{D}_2(u, u_x, v, v_x, u_{xy}, uv, ...). \tag{9}$$

Thus, for file 3, you must work out how to generalise 7 so that it can represent this coupled PDE (hint: $\mathbf{u}_t$ and ξ should be replaced with matrices instead of column vectors).

Hint: You may assume for all files that the PDE only includes linear and non-linear combinations of the solution components and/or its (mixed) partial derivatives (and not the domain coordinates), and that only up to (and including) third order (mixed) partial derivatives are used. Carry out the following steps: first, write code which estimates mixed partial derivatives of the solution at each observational point. You can either use an interpolation-based, neural network-based, or finite difference-based approach. Then, decide on an appropriate library of possible PDE terms to include, and build the matrix Θ. Finally, either use an existing sparse linear system solver, or write your own solver to solve 7.

Deliverables: In your project report, state your guess of the PDE for each file. Describe how your algorithm works, the size of the library D you use for each file, what convergence issues you encounter, and the possible future extensions you would consider to improve the convergence and/or generality of your method.

Bonus Task

The objective of this task is to gain practical experience the full pipeline of ML for solving PDE – this will include a trip into data generation, training, and fine-tuning. More specifically, you will gain expericence with:

- Implementing stochastic samplers of discretized functions from three different classes of functions – Gaussian processes (**GP**), piecewise-linear functions (**PL**) and Chebyshev polynomials (**CP**).
- Using a finite difference solver for the Poisson equation for various forcing terms.
- Performing zero-shot learning and transfer learning / fine-tuning to explore the generalization properties of your models.

Bonus 1.1 - Dataset Creation

Implement the following procedure for generating a dataset that will later be used for training and evaluation – please mind that we are only working in $L^2([-1,1])$. This will require you to implement randomized samplers to draw discretizations of functions from the 3 given function spaces.

Algorithm 1 Generate Dataset from Stochastic Samples and Solve Poisson Equation

Initialize dataset $\mathcal{D} \leftarrow \{\}$
for class $\in \{\text{GP}, \text{PL}, \text{CP}\}$ **do**
    for $i = 1$ to 100 **do**
        Sample $f_{\text{class},i}(x)$ from function class class over the domain $[-1, 1]$
        Solve $-\Delta u_{\text{class},i}(x) = f_{\text{class},i}(x)$ with boundary conditions $u(-1) = u(1) = 0$
        Store pair $(f_{\text{class},i}(x), u_{\text{class},i}(x))$ in dataset $\mathcal{D}$
    end for
end for
Output: Dataset $\mathcal{D}$

Notes:

- You are allowed to use any finite-element library / software you prefer for building your dataset of source terms / solutions.
- You can tinker with the number of samples, it doesn't necessarily need to be 100.

Bonus 1.2 - Training and fine-tuning

Now that you have your dataset, you can train one model per class – implement the following procedure:

Algorithm 2 Train one Neural Operator per Class

Initialize dataset $\mathcal{D}_{\text{GP}}, \mathcal{D}_{\text{PL}}, \mathcal{D}_{\text{CP}}$ from previous algorithm
Initialize results storage $\mathcal{R} \leftarrow \{\}$
for train_class $\in \{\text{GP}, \text{PL}, \text{CH}\}$ **do**
    Initialize neural operator $N_{\text{train\_class}}$
    Train $N_{\text{train\_class}}$ on $\mathcal{D}_{\text{train\_class}}$
        - Use minibatches of the form $\{(f_{\text{train\_class},i}, u_{\text{train\_class},i})\}$
        - Minimize the L2 error: $L = \frac{1}{\text{batch\_size}} \sum_i \|N_{\text{train\_class}}(f_{\text{train\_class},i}) - u_{\text{train\_class},i}\|_2^2$
end for
Output: Trained operators $\{N_{\text{GP}}, N_{\text{PL}}, N_{\text{CP}}\}$

We can now do zero-shot evaluations (zero-shot evaluation involves assessing the performance of the model on different distributions than the ones it was trained on) and some fine-tuning – implement the following procedure:

Algorithm 3 Fine-tune Operators on Other Classes

Initialize dataset $\mathcal{D}_{\text{GP}}, \mathcal{D}_{\text{PL}}, \mathcal{D}_{\text{CH}}$ from previous algorithm
Initialize results storage $\mathcal{R} \leftarrow \{\}$
Trained operators $\{N_{\text{GP}}, N_{\text{PL}}, N_{\text{CH}}\}$ from Algorithm 2
for train_class $\in \{\text{GP}, \text{PL}, \text{CH}\}$ **do**
    for eval_class $\in \{\text{GP}, \text{PL}, \text{CH}\} \setminus \{\text{train\_class}\}$ **do**
        Evaluate zero-shot performance of $N_{\text{train\_class}}$ on $\mathcal{D}_{\text{eval\_class}}$
        Sample 20 examples $\mathcal{D}_{\text{eval\_class,finetune}} \subset \mathcal{D}_{\text{eval\_class}}$
        Fine-tune $N_{\text{train\_class}}$ on $\mathcal{D}_{\text{eval\_class,finetune}}$
            - Use minibatches of the form $\{(f_{\text{eval\_class},i}, u_{\text{eval\_class},i})\}$
            - Minimize the L2 error: $L = \frac{1}{\text{batch\_size}} \sum_i \|N_{\text{train\_class}}(f_{\text{eval\_class},i}) - u_{\text{eval\_class},i}\|_2^2$
        Evaluate fine-tuned performance on $\mathcal{D}_{\text{eval\_class}}$
        Compute relative L_2 distance and store in $\mathcal{R}$
    end for
end for
Output: Fine-tuned results $\mathcal{R}$

Bonus 1.3 - Write-up

Document your results and qualitatively discuss the results you encountered in zero-shot evaluations and evaluations performed after fine-tuning your model. Also answer:

- Does training on any one specific class offer (on average) superior zero-shot performance on the others? Explain why you think this happens.
- Are PINNs a suitable choice for solving this task? Explain your reasoning.

Bonus - Function Spaces details

Gaussian Process with an RBF Kernel

A Gaussian process (GP) is a collection of random variables, any finite number of which have a joint Gaussian distribution. A GP is fully specified by its mean function $m(x)$ and covariance function $k(x, x')$.

- Mean function: $m(x) = \mathbb{E}[f(x)]$.
- Covariance function (RBF kernel): $k(x, x') = \sigma_f^2 \exp\left(-\frac{\|x-x'\|^2}{2l^2}\right)$.

For drawing discretized function samples from a GP with fixed mean / kernel function, you can for example follow the methodology presented here.

Piecewise Linear Functions

Piecewise linear functions are defined by a set of linear segments connecting a series of points. Formally, a piecewise linear function $f(x)$ on the interval $[a, b]$ is defined by:

$$f(x) = \begin{cases} \alpha_1 x + \beta_1 & \text{for } x \in [x_0, x_1], \\ \alpha_2 x + \beta_2 & \text{for } x \in [x_1, x_2], \\ \vdots & \vdots \\ \alpha_k x + \beta_k & \text{for } x \in [x_{k-1}, x_k], \end{cases}$$

where $\{x_0, x_1, \ldots, x_k\}$ are the points at which the segments join. For randomization purposes, you can choose random break points and slopes / offsets on the lines on each segment.

Chebyshev Polynomials

Chebyshev polynomials are a sequence of orthogonal polynomials that arise in various mathematical contexts. The n-th Chebyshev polynomial $T_n(x)$ is defined recursively as follows:

- $T_0(x) = 1$
- $T_1(x) = x$
- $T_{n+1}(x) = 2x \cdot T_n(x) - T_{n-1}(x)$

A 'random' Chebyshev polynomial function $f(x)$ on the interval $[-1, 1]$ can be defined as a sum of the first N Chebyshev polynomials, each weighted by randomly drawn coefficients:

$$f(x) = \sum_{i=0}^{N-1} c_i \cdot T_i(x)$$

where c_i are randomly drawn coefficients.

Note: You are allowed to use whatever existing library / software for sampling these functions and solving the associated Poisson equations.

References

[1] S. H. Rudy, S. L. Brunton, J. L. Proctor, and J. N. Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(4):e1602614, 2017.