Final Project

ETH zürich

AI in the Sciences and Engineering
Due date: January 21st, 2025

IMPORTANT INFORMATION

This document describes the final project. The due date for student submissions is set for **21 January 2025**.

This project consists of three main tasks, each worth 40 points, for a total of 120 possible points. **The final submission consists of the predictions files, the code and a report of maximum 2 pages per task (applies to bonus as well)** 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*.

Grading System

- A perfect grade of 6.0 requires accumulating at least 100 points out of the 120 possible points.
- The passing grade of 4.0 is achieved at 60 points.
- As a guideline, for each task you can earn up to 20 points by:
  - Demonstrating clear understanding of the task in your report.
  - Providing a complete, working codebase that addresses all aspects of the task.

  The remaining 20 points per task will be awarded based on the quality of your implementation and results.
- This is not a rigid scheme - we aim to evaluate your understanding and effort rather than enforce strict performance metrics.

Bonus System

- The project includes **two bonus questions**.
- To earn the final grade bonus of 0.5, you must make reasonable attempts at **both** bonus questions.
- Attempting only one bonus question will not earn any bonus points, regardless of the quality of the attempt.
- What constitutes a "reasonable attempt" will be determined based on the submissions of all students.
- Note that the final grade, even with bonus, is capped at 6.0.

Training the FNO to Solve the 1D Wave Equation

In this exercise, you will train a model to approximate the solution of the 1D wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad t \in (0, 1], \quad x \in [0, 1], \quad c = 0.5$$

with boundary conditions

$$u(0, t) = u(1, t) = 0$$

and initial conditions

$$u(x, 0) = u_0(x), \quad u_t(x, 0) = 0$$

where u_t is the partial derivative with respect to time. Note that the initial conditions are sampled from an unknown distribution. Knowing the exact expression for u_0 is not necessary to complete the tasks below.

You should use a **Fourier Neural Operator (FNO)** for all the tasks.

Dataset Details

You are provided with the following datasets in this **folder**:

1. **Training Dataset:** `train_sol.npy`
   - Shape: $(128, 5, 64)$
   - Description:
     - 128: Number of trajectories.
     - 5: Time snapshots of the solution. For a given trajectory u, the time snapshots are:

       $u[0]$: Initial condition u_0 at $t = 0.0$,
       $u[1]$: Solution at $t = 0.25$,
       $u[2]$: Solution at $t = 0.50$,
       $u[3]$: Solution at $t = 0.75$,
       $u[4]$: Solution at $t = 1.0$.

     - 64: Spatial resolution of the data.
     - Please see Figure 1 for visualization.
2. **Testing Datasets:**
   - `test_sol.npy`: Similar shape as the training dataset $(128, 5, 64)$. Contains 128 trajectories with all 5 time snapshots.
   - `test_sol_res_{s}.npy`: Testing datasets at varying spatial resolutions $s \in \{32, 64, 96, 128\}$. Shape: $(128, 2, s)$, where:

     $u[0]$: Initial condition u_0 at $t = 0.0$,
     $u[1]$: Solution at $t = 1.0$.

     Note: Intermediate time snapshots are not included.

- `test_sol_OOD.npy`: Out-of-distribution (OOD) testing dataset. Shape: $(128, 2, 64)$, where:

$u[0]$: Initial condition u_0 at $t = 0.0$,
$u[1]$: Solution at $t = 1.0$.

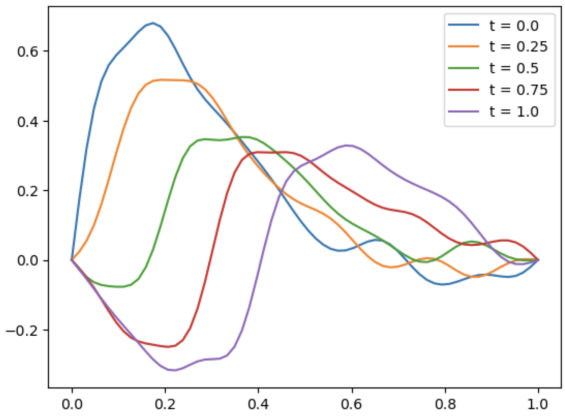

Figure 1: Training Dataset - One trajectory

Tasks

Task 1: One-to-One Training

1. Use **64** trajectories from the training dataset.
2. Select the first ($t = 0.0$) and last ($t = 1.0$) time snapshots for these trajectories.
3. Train an FNO model to learn the mapping:

$$G : u_0 \to u(t = 1.0)$$

4. Use the remaining trajectories for validation.
5. Test the trained model on the `test_sol.npy` dataset, focusing **only** on predictions at $t = 1.0$ (the map $u_0 \to u(t = 1.0)$).
6. Report the **average relative L2 error**:

$$\text{err} = \frac{1}{128} \sum_{n=1}^{128} \frac{\|u_{\text{pred}}^{(n)}(t = 1.0) - u_{\text{true}}^{(n)}(t = 1.0)\|_2}{\|u_{\text{true}}^{(n)}(t = 1.0)\|_2}$$

Task 2: Testing on Different Resolutions

1. Test the trained model from Task 1 on the datasets `test_sol_res_{s}.npy` for $s \in \{32, 64, 96, 128\}$.
2. Compute and report the **average relative L2 error** for each dataset.
3. What do you observe about the model's performance across different resolutions?

Task 3: Testing on Out-of-Distribution (OOD) Dataset

1. Test the trained model from Task 1 on the OOD dataset `test_sol_OOD.npy`.
2. Compute and report the **average relative L2 error**.
3. Compare the error to the one obtained in Task 1. What do you observe? Is the error higher or lower?

Task 4: All2All Training

1. Use 64 trajectories from the training dataset.
2. Use **all provided time snapshot**s ($t = 0.0, 0.25, 0.50, 0.75, 1.0$) for these trajectories to train a time-dependent FNO model. Note that this is similar to the task that we had in time-dependent CNO tutorial. *Hint: Use time-conditional batch normalization and include time as one of the input channels.*
3. What is the total number of samples used for training in the **All2All** approach?
4. Test the trained model on the `test_sol.npy` dataset, focusing **only** on predictions at $t = 1.0$.
5. Report the **average relative L2 error**.
6. Compare the error to the one obtained in Task 1. What do you observe?

Bonus Task

1. Use the model from Task 4 to make predictions at multiple time steps: $t = 0.25, t = 0.50, t = 0.75, t = 1.0$.
2. Compute the **average relative L2 error** for each time step.
3. What do you observe about the model's performance over time?
4. Use the model from Task 4 to make predictions on the OOD dataset at $t = 1.0$.
5. What do you observe about the model's performance?

PDE-Find: Reconstructing PDEs from data

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

$$u_t = \mathcal{D}(u, u_x, u_{xx}, u_y, u_{yy}, u_{xy}, ..., x, y, ..., t), \tag{1}$$

where subscripts denote partial differentiation, and we assume the solution is a scalar field, i.e. $u(x, y, ..., t) : \mathbb{R}^d \to \mathbb{R}^1$. 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

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

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

$$\mathbf{u}_t = \Theta(\mathbf{u})\xi, \tag{3}$$

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 3 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 [**pdefind**] 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

$$u_t = \mathcal{D}_1(u, u_x, v, v_x, u_{xy}, uv, ...), \tag{4}$$
$$v_t = \mathcal{D}_2(u, u_x, v, v_x, u_{xy}, uv, ...). \tag{5}$$

Thus, for file 3, you must work out how to generalise 3 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 3.

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.

Foundation Models for Phase-Field Dynamics

The Allen-Cahn equation is a fundamental model in materials science that describes phase separation dynamics in binary systems:

$$\frac{\partial u}{\partial t} = \Delta_x u - \frac{1}{\epsilon^2}(u^3 - u), \quad t \in [0, 1], \quad x \in [-1, 1] \tag{6}$$

with periodic boundary conditions. The parameter ϵ controls the width of transition layers between phases, with smaller values leading to sharper interfaces. This equation exhibits rich dynamics, from smooth diffusion-dominated behavior at large ϵ to rapid phase separation with sharp interfaces at small ϵ.

In this task, you will develop a neural foundation model capable of solving the Allen-Cahn equation across different parameter regimes and initial conditions. Template code is provided for generating training data with various initial conditions drawn from three distributions: random Fourier series, Gaussian mixtures, and piecewise linear functions. The code also includes the numerical solver using `scipy.integrate.solve_ivp`.

Your task: Explore how neural architectures can learn and generalize across different dynamical regimes. Starting from the code templates from this **folder** (which you are **NOT** forced to use, it's just there to give you a starting point):

- Complete the data generation code to create training datasets with different ϵ values (e.g. 0.1, 0.05, 0.02) and initial condition types. The data should capture the full range of behaviors from smooth evolution to sharp interface dynamics (you can start from `allen-cahn-template.py`).
- Develop a time-dependent neural solver that can handle the entire trajectory of the solution. Consider how to effectively embed both the time dependency and the ϵ parameter in your architecture.
- Investigate the model's ability to generalize across:
  - Different ϵ values, including interpolation and extrapolation
  - Various initial condition types
  - Higher frequency components and sharper transitions than seen in training

The provided template includes functions for generating different types of initial conditions and solving the PDE. You will need to implement the core components, including the Allen-Cahn right-hand side and the initial condition generators.

Data generation specifics: Each training trajectory consists of 5 temporal snapshots that capture as much as possible of the dynamics before reaching a steady state, on a spatial grid of 128 points. Therefore both the time-scale t and ϵ are to be considered as hyper-parameters you need to tune to ensure capturing the full-range of the dynamics.

Note: Think carefully about your choices of ϵ and the expected behavior of solutions as $\epsilon \to 0$ – how do the dynamics change, in terms of the competing diffusive versus nonlinear effects?

Your final submission should include:

- The completed data generation code
- Your neural PDE solver implementation
- A report (maximum 2 pages) that includes:
  - Visualization of the generated data across different regimes
  - Analysis of your model's performance, including error metrics and convergence behavior
  - Discussion of generalization properties, supported by plots comparing predictions with true solutions
  - Investigation of how the model handles challenging cases, such as very small ϵ values or high-frequency initial conditions

Deliverables: For evaluation, focus on the relative L2 error between predicted and true solutions, but also consider qualitative physical aspects such as interface width.

Your report (max 2 pages long) should demonstrate understanding of both the mathematical properties of the Allen-Cahn equation and the capabilities and limitations of your neural approach.

Bonus Task: Stability Analysis

Consider the following stability theorem for the Allen-Cahn equation:

Theorem (Stability)**.** *Let $u \in H^1([0,T]; H^{-1}(\Omega)) \cap L^\infty([0,T]; H^1(\Omega))$ be a weak solution of the Allen-Cahn equation with $|u| \leq 1$ almost everywhere in $[0,T] \times \Omega$. Let $\tilde{u} \in H^1([0,T]; H^{-1}(\Omega)) \cap L^2([0,T]; H^1(\Omega))$ satisfy $|\tilde{u}| \leq 1$ almost everywhere in $[0,T] \times \Omega$, and $\tilde{u}(0) = \tilde{u}_0$, $f \in H^1([0,T]; W^{1\infty}_{loc}(\Omega))$ and solve*

$$(\partial_t \tilde{u}, v) + (\nabla \tilde{u}, \nabla v) = -\varepsilon^{-2}(f(\tilde{u}), v)$$

for almost every $t \in [0,T]$, all $v \in H^1(\Omega)$. Then we have

$$\sup_{t\in[0,T]} \|u - \tilde{u}\|^2_{L^2(\Omega)} + \int_0^T \|\nabla(u - \tilde{u})\|^2 dt \leq 2\|u_0 - \tilde{u}_0\|^2 \exp((1 + 2c_f\varepsilon^{-2})T).$$

Provide a complete proof of this stability result. Your proof should:

1. Explain the choice of test function for deriving the energy estimate
2. Show how to handle the nonlinear term $f(u)$
3. Use appropriate Gronwall-type arguments
4. Carefully track the dependence on ε

Hint: Consider the difference $w = u - \tilde{u}$ as a test function and analyze how the nonlinear term contributes to the energy estimate through appropriate bounds on f'.

References

[1] Samuel H Rudy et al. "Data-driven discovery of partial differential equations". In: *Science advances* 3.4 (2017), e1602614.