AISE 2024 Final Project Report

Filipovic Grcic Leo

January 2025

1 Problem 1 - Training the FNO to Solve the 1D Wave Equation

In this problem, we want to train a model to approximate the solutions of the 1D wave equation. I attempt this using **Neural Operators**, a generalization of Deep Neural Networks that learns mappings between function spaces. In particular, I use a **Fourier Neural Operator (FNO)**.

The FNO architecture is loosely based on the structure outlined in [1]. The input consists of two channels, representing the values of $u(x)$ at time $t = 0$, and the 1D x-grid on which the function is evaluated. The input is lifted to higher dimensionality by a linear layer `linear_p` (*lifting layer*). After this, three layers `fourier_layer` are applied. Each `fourier_layer` concurrently performs a linear transform (`Conv1d`) and a **Spectral convolution** `SpectralConv1d`, sums them and passes through a `Tanh` activation function. Finally, there is a linear layer `linear_q` (*projection layer*) which reduces the channels to a 1-dimensional output, representing the solution u at time $t = 1.0$.

The key element of the model is the Spectral convolution `SpectralConv1D`. In the implementation, it makes use of the *Fast Fourier Transform* `torch.fft.rfft()` to compute the Fourier modes of the input. It then applies a linear transform on the modes, filtering out the higher modes. Finally, it returns to physical space with the inverse FFT `torch.fft.irfft()` .

1.1 Task 1: One-to-One Training

In Task 1, the aim is to train the FNO on the `train_sol.npy` dataset. In the training, 64 trajectories are used, selecting only the first ($t = 0.0$) and last ($t = 1.0$) time snapshots. The remaining trajectories are used for validation, and testing is carried on the `test_sol.npy` dataset, focusing only on predictions at $t = 1.0$. We obtain an **average relative L2 error** of **10.7887%**.

1.2 Task 2: Testing on Different Resolutions

I proceed by testing the trained model on the datasets `test_sol_res_s.npy` for $s \in 32, 64, 96, 128$. Somewhat surprisingly, the model performs marginally

Resolution	Avg. Rel. L2 Error
32	11.5606%
64	10.0994%
96	9.9856%
128	10.7887%

better on the 96 resolution dataset - although this is likely a product of chance. Otherwise, as expected, we can see the performance worsening for resolutions that differ from the training resolution of 64. In fact, when moving to the lower resolution of 32, the error worsens by roughly 1.5 percentage points.

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

Finally, we test the model on the *out-of-Distribution* `test_sol_OOD.npy` dataset. Here, the computed **average relative L2 error** is of **13.9163%**, significantly higher than the one in Task 1, indicating that the model does not generalize particularly well to unfamiliar trajectories.

1.4 Task 4: All2All Training

For Task 4, the FNO model is modified by making it time-dependent. This is done by adding **Time-conditional batch normalization** `FILM` after each layer. For training, 64 trajectories are used from `train_sol.npy`, and all provided time snapshots ($t = 0.0, 0.25, 0.50, 0.75, 1.0$). By using all admissible input-output pairs, I create **640 training samples** for the **All2All** training. Focusing only on predictions for $t = 1.0$, we get an **average relative L2 error** of **37.8036%**. Compared to our original FNO model, the error is significantly higher. The model is able to reproduce the evolution of the trajectory only in broad strokes.

1.5 Bonus Task

In the Bonus Task, the **average relative L2 error** is computed for each possibile time difference ($0.25, 0.50, 0.75, 1.0$: Meanwhile, testing only on the OOD

Time difference	Avg. Rel. L2 Error
0.25	33.4545%
0.50	42.4036%
0.75	45.8621%
1.00	37.8036%

dataset, we get an avg. rel. L2 Error of **46.5839%**. Notice that the model performs better on the 0.25 time step, presumably because it is the most frequent case in training, and because it may capture better smaller time step. The error grows for $t = 0.50$ and $t = 0.75$, and becomes smaller again for $t = 1.0$.

2 Problem 2 - PDE-Find: Reconstructing PDEs from data

In the second problem, I implement a sub-sampling, **STRidge**-based version of the **PDE-FIND** tool for data-driven discovery of the governing PDE from observed data, as described in [2]. In particular, I attempt to discover time-dependent PDEs using sparse regression methods to identify the key terms driving the dynamics. The data consists of solutions from three distinct PDE systems, progressively more difficult to analyze.

2.1 Implementation

The problem assumes that the unknown time-dependent PDE is expressed as:

$$u_t = D(u, u_x, u_{xx}, u_y, u_{yy}, u_{xy}, \ldots),$$

where u is a scalar field. For coupled systems, the method generalizes to include multiple components (e.g., u and v).

In the implementation, there are three key steps:

1. Subsample points across spatial and temporal dimensions, in order to avoid computationally prohibitive quantities of data.
2. Construct a library $\Theta(u)$ of candidate terms, including linear, nonlinear, and partial derivative combinations.
3. Use **ridge regression** with **hard thresholding** to solve the linear system: $u_t = \Theta(u)\xi$, while also enforcing sparsity of the result ξ.

The first step is achieved by uniformly sub-sampling across spatial and temporal dimensions. This enables a trade-off between accuracy and computational efficiency. The second step is achieved using the method `construct_theta()`, which builds the matrix Θ, filling its columns with the values of all admissible candidate terms. For the third step, the method `sequential_ridge()` is called. The method is based on **STRidge** algorithm outlined by the authors in [2]. It iteratively performs ridge regression, eliminating terms with coefficients below a specified threshold `tol`. The pseudocode implementation is shown below:

Listing 1: Pseudocode implementation of STRidge

```
def sequential_ridge(Theta, ut, lambda_val, tol, max_iter):

    # Normalize columns of Theta
    Theta_n = Theta * normalizer

    # First Ridge Regression
    ridge = Ridge(alpha=lambda_val, fit_intercept=False)
    ridge.fit(Theta_n, ut)
    xi = ridge.coef_

    # Iterate sequential Ridge Regressions
```

```
    for _ in range(max_iter):
        big_ids = np.where(abs(xi) >= tol)[0] # eliminate small
            elements
        Theta_sparse = Theta_n[:, big_ids]
        ridge.fit(Theta_sparse, ut)
        xi_sparse = ridge.coef_
        xi[big_ids] = xi_sparse

    xi = xi * normalizer.reshape(-1, 1)
    return xi
```

2.2 Dataset 1

Library: With polynomial degree up to 2, derivatives up to the second order u_{xx} and combinations.
Result: $D = [0.0998]u_{xx} + [-0.9992]uu_x$
PDE Guess: Burger's equation with an additional diffusive term:

$$u_t = 0.1u_{xx} - uu_x$$

2.3 Dataset 2

Library: Polynomial degree up to 2, derivatives up to third order u_{xxx} and combinations.
Result: $D = [-1.0620]u_{xxx} + [-6.1038]uu_x$
PDE Guess: Korteweg–de Vries (KdV) Equation, with two solitons traveling at different speeds:

$$u_t = -u_{xxx} - 6uu_x$$

2.4 Dataset 3

Library: Variables u, v, polynomial degree up to 2, derivatives up to the second order *w.r.t.* x and y, and combinations.

Result: (*here rounded to two decimal points for clarity*)

$$u_t = [0.10]u_{xx} + [0.10]u_{yy} + [1.00]v^3 + [0.99]u + [-0.99]uv^2 + [1.00]u^2v + [-0.99]u^3$$
$$v_t = [0.10]v_{xx} + [0.10]v_{yy} + [0.99]v + [-0.99]v^3 + [-1.00]uv^2 + [-0.99]u^2v + [-1.00]u^3$$

PDE Guess: repeating with different values of the regularization parameter `lambda_val` yielded different results. The cleanest result was obtained with `lambda_val = 10e-5, tol = 1.0, max_it = 10`. The governing PDE is in this case a Coupled Reaction-diffusion System:

$$u_t = 0.10\Delta u + f(u, v)$$
$$v_t = 0.10\Delta v + g(u, v)$$

2.5 Discussion and Future Work

The implemented algorithm is able to identify the governing PDEs for all three datasets, although it often requires significant manual tuning of the parameters `lambda_val, tol, max_iter` to achieve clear results - which I also had to validate by comparing with visual plottings of the datasets. For example, in the second PDE, the method usually returned a pure transport equation, even though it was clearly a case of KdV. Furthermore, it was hard to determine a priori how large a library to employ. A significant issue with the third dataset was its large size, which ultimately led me to apply uniform data subsampling to all three problems. Finally, the current implementation requires the manual construction of derivative terms, which are then passed to `construct_theta()` - but with more than two variables and higher-order derivatives, this becomes quite tedious.

Future Extensions:

- Incorporate alternative / neural network-based derivative estimation for noisy or irregularly spaced data;
- Investigate methods for automatic tuning of parameters such as `lambda_val, tol, max_iter`, and sample size;
- Automate construction of standalone derivative terms for all variables and orders;
- Investigate alternative regularization techniques to improve sparsity and stability

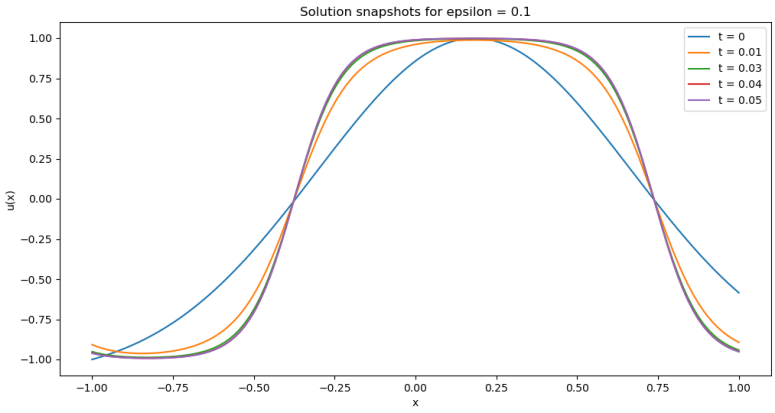
3 Problem 3 - Foundation Models for Phase-Field Dynamics

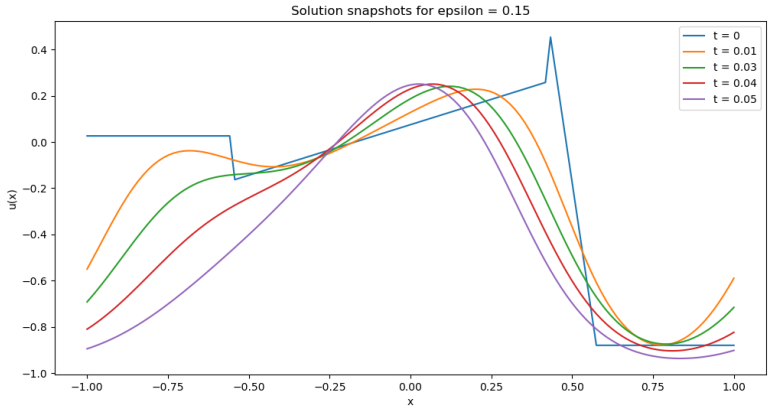
In the third problem, I attempt to implement a neural foundation model capable of solving the Allen-Cahn equation across different parameter regimes and initial conditions. My solution is based on the Fourier Neural Operator, with time embedding and parametrized batch normalization.

3.1 Data Generation

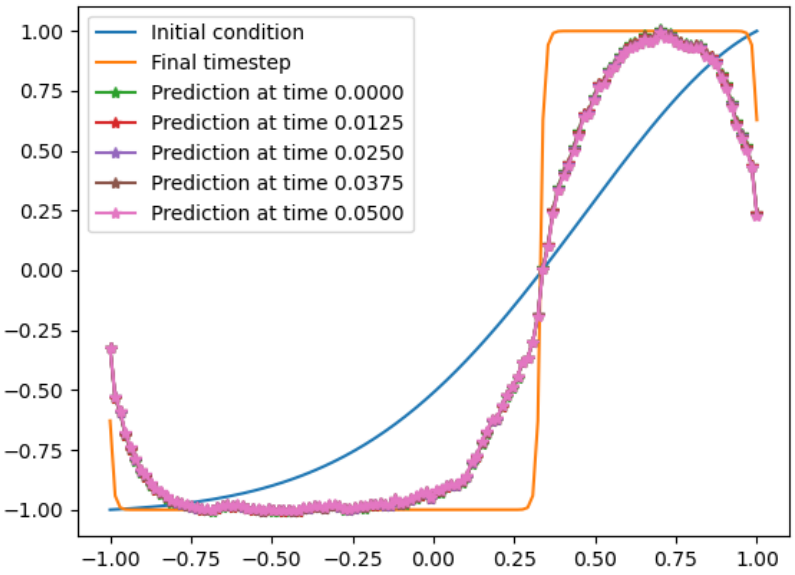
In order to train the model, appropriate training datasets need to be generated. As a first step, I implement the three functions `generate_fourier_ic`, `generate_gmm_ic` and `generate_piecewise_ic`, which allow for the generation of the Initial Conditions based on the random Fourier series, Gaussian mixture model, and piecewise linear functions respectively. Then, I implement the function `allen_cahn_rhs`, which returns the right-hand-side of the Allen-Cahn equation. This function is then used by `scipy.integrate.solve_ivp` to compute the trajectory of the solution, given an initial condition and a time span. In order to generate diverse training data, I choose $\varepsilon = [0.15, 0.10, 0.01]$ and

$t_{max} = 0.05$. Then I create 150 samples for each ε value, using all three types of initial conditions. Here below are shown two examples of the solution evolution with different initial conditions, ε values and on a time span of 0.05.

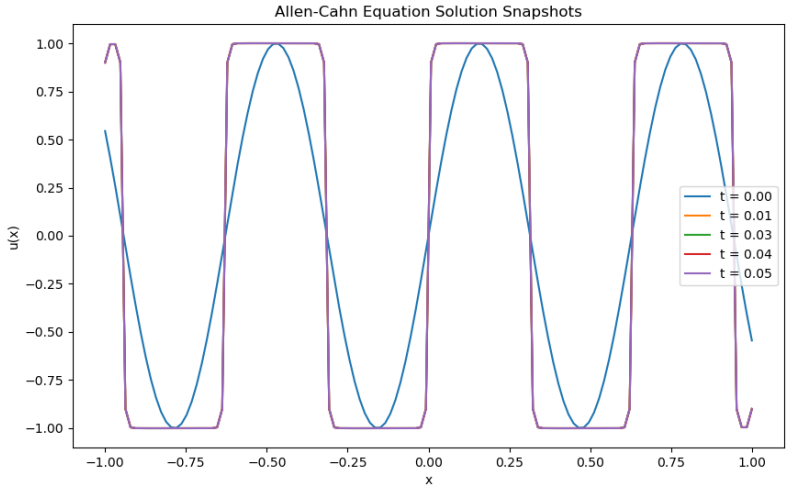


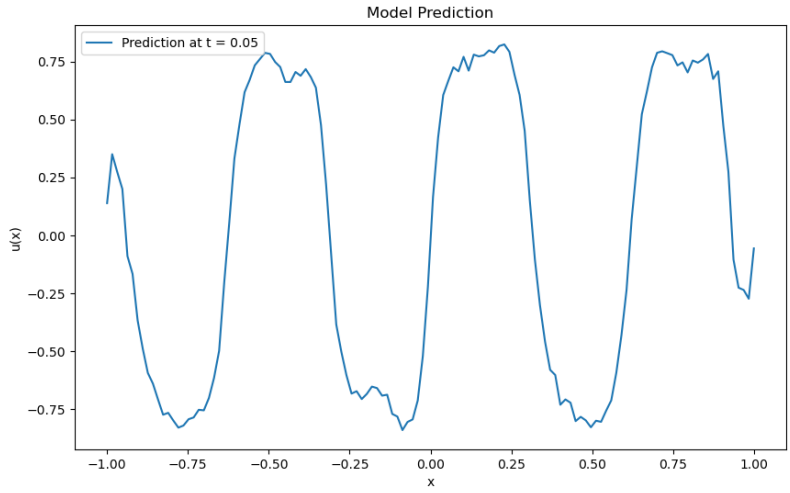


In general, for smaller ε, the solutions reach a steady state with sharper interfaces faster. For the time-dependent neural solver, I repurpose the Fourier Neural Operator from Problem 1, by expanding its architecture and adding ε-parametrized batch normalizations. The architecture exhibits three consecutive Fourier layers, each containing two consecutive `FILM` class batch normalizations. I also implement **All2All** training, in order to exploit fully the generated datasets of solutions. This increases the total number of samples to 4500, which I split for training/validation with a 80/20 rule. Unfortunately, after training, the model exhibits a high **Relative L2 Test Norm**: 24.639%. In fact, visualizing an example prediction, we notice that the model doesn't learn the time-dependence well, but rather learns an "average" of later-time solutions:



This problem is even more evident when the model is provided with an **OOD** initial condition with a high frequency sine wave and particularly small $\varepsilon = 0.01$:





Therefore, I conclude that the implemented architecture could not appropriately learn the time and ε dependence of the solutions.

3.2 Bonus Task - Stability Analysis

We begin by subtracting the weak forms for u and $\tilde{u}$:

$$(\partial_t w, v) + (\nabla w, \nabla v) = -\varepsilon^{-2}(f(u) - f(\tilde{u}), v), \quad \forall v \in H^1(\Omega)$$

We then let $v = w$ be the test function:

$$(\partial_t w, w) + (\nabla w, \nabla w) = \frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)} + \|\nabla w\|^2_{L^2(\Omega)} = -\varepsilon^{-2}(f(u) - f(\tilde{u}), w) \quad (1)$$

If we assume that f is smooth enough, we can bound the rhs. Assume that:

$$|f(u) - f(\tilde{u})| \leq c_f |u - \tilde{u}| = c_f |w|$$

c_f is a constant that depends on bounds for $f'(u)$. Then:

$$|(f(u) - f(\tilde{u}), w)| \leq c_f \|w\|^2_{L^2(\Omega)}$$

and:

$$\frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)} + \|\nabla w\|^2_{L^2(\Omega)} \leq \varepsilon^{-2} c_f ||w||^2_{L^2(\Omega)} \quad (2)$$

We can drop the term $\|\nabla w\|^2_{L^2(\Omega)}$ and apply Grönwall's inequality to bound $\|w\|^2_{L^2(\Omega)}$:

$$\frac{d}{dt}\|w\|^2_{L^2(\Omega)} \leq 2\varepsilon^{-2} c_f \|w\|^2_{L^2(\Omega)}$$

$$\implies \|w(t)\|^2_{L^2(\Omega)} \leq \|w(0)\|^2_{L^2(\Omega)} \exp(2\varepsilon^{-2} c_f t)$$

$$\implies \sup_{t\in[0,T]} \|w(t)\|^2_{L^2(\Omega)} \leq \|w(0)\|^2_{L^2(\Omega)} \exp(2\varepsilon^{-2}c_f T) \tag{3}$$

We can integrate both sides of inequality (2):

$$\int_0^T \frac{1}{2}\frac{d}{dt}\|w\|^2_{L^2(\Omega)} + \|\nabla w\|^2_{L^2(\Omega)} dt \leq \int_0^T \varepsilon^{-2} c_f ||w||^2_{L^2(\Omega)} dt \leq \frac{1}{2}\|w(0)\|^2_{L^2(\Omega)} (\exp(2\varepsilon^{-2}c_f T) - 1)$$

where the last inequality is obtained by plugging the Gronwall bound on $\|w(t)\|^2$ in the integral. On the lhs we have:

$$\frac{1}{2}\|w(T)\|^2_{L^2(\Omega)} - \frac{1}{2}\|w(0)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w\|^2_{L^2(\Omega)} dt$$

Removing the positive term $\frac{1}{2}\|w(T)\|^2_{L^2(\Omega)}$, moving $\frac{1}{2}\|w(0)\|^2_{L^2(\Omega)}$ to the rhs and combining with our result from (3), we have:

$$\sup_{t\in[0,T]} \|w(t)\|^2_{L^2(\Omega)} + \int_0^T \|\nabla w(t)\|^2_{L^2(\Omega)}\, dt \leq 2\|w_0\|^2_{L^2(\Omega)} \exp\left((1 + 2\varepsilon^{-2}c_f)T\right)$$

thus concluding our proof.

References

[1] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier Neural Operator for Parametric Partial Differential Equations, May 2021. arXiv:2010.08895.

[2] Samuel H Rudy, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Data-driven discovery of partial differential equations. *Science advances*, 3(4):e1602614, 2017.