Project 1: Training the FNO to Solve the 1D Wave Equation

Wu, You
Department of Mathematics
youwuyou@ethz.ch

Problem Setup

In this project, we consider the one-dimensional wave equation:

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = f(x,t) \tag{1}$$

We seek for the scalar field $u : \tilde{\Omega} \mapsto \mathbb{R}$ defined on the spatio-temporal domain $\tilde{\Omega} \coloneqq \Omega \times (0, T]$, with the final observation time at $T = 1$. The spatial domain spans over $\Omega = [0, 1]$ with domain boundary $\partial\Omega = \{0, 1\}$. In particular, we consider the special case when the equation is homogenous ($f \equiv 0$) and the superposition principle holds. By incorporating suitable initial and boundary conditions, we can formulate the following initial boundary value problem (IBVP):

$$\begin{cases} \frac{\partial^2 u}{\partial t^2} &= c^2 \frac{\partial^2 u}{\partial x^2} & (x,t) \in \Omega \times (0,T] \\ u(x,0) &= u_0(x) & x \in \Omega \\ \frac{\partial}{\partial t} u(x,0) &= 0 & x \in \Omega \\ u(x,t) &= 0 & (x,t) \in \partial\Omega \times (0,T] \end{cases} \tag{2}$$

Since the equation is second-order in time, we supplemented **two initial conditions** for the resulting evolution problem to be well-posed. Additionally, the initial data u_0 are sampled from some unknown distribution.

Theory: Neural Operator

We can approach the problem in a generic setting, in which we consider the solution operator $\mathcal{G}$ to the system of PDEs (2). It is an infinite-dimensional mapping between some function spaces defined on bounded domain Ω and maps from the initial condition u_0 at the final observation time $u(t = 1)$:

$$\mathcal{G} : u_0 \mapsto u(\cdot, t = 1) \tag{3}$$

We aim to approximate $\mathcal{G}$ by constructing a parametric map $\mathcal{G}_\theta$ with parameters from the finite-dimensional space $\theta \in \mathbb{R}^p$ such that $\mathcal{G}_\theta \approx \mathcal{G}$. The approximation can be done by using a **neural operator** architecture and leverage the power of **operator learning** [1]. We write the generic form of such multi-layer operator learning architecture as follows:

$$\mathcal{G}_\theta \coloneqq \mathcal{G}_\theta^{(L)} \circ \mathcal{G}_\theta^{(L-1)} \circ ... \mathcal{G}_\theta^{(1)} \tag{4}$$

The architecture is a composition of linear and nonlinear operators $\mathcal{G}_\theta^{(l)}$ between function spaces. Each hidden layer $\mathcal{G}_\theta^{(l)}$ consists of a local linear map $W^{(l)}$, a non-local kernel-integral operator $K^{(l)}$, a bias function $b^{(l)}$, and a nonlinear activation function σ as in the following form:

$$\mathcal{G}_\theta^{(l)} \coloneqq \sigma\big(W^{(l)} + K^{(l)} + b^{(l)}\big), \text{where } l = 1, ..., L \tag{5}$$

Fourier Neural Operator (FNO)

The FNO architecture is of the generic form as in equation (4) but with an additional add-on that it also includes a lifting layer P and a projection layer Q as follows:

$$\mathcal{G}_\theta := \underbrace{Q}_{\text{Projecting}} \circ \mathcal{G}_\theta^{(L)} \circ \mathcal{G}_\theta^{(L-1)} \circ ... \mathcal{G}_\theta^{(1)} \circ \underbrace{P}_{\text{Lifting}} \tag{6}$$

The additional operators P and Q are used to lift the parameters into the large latent space and project them back to the physical space in the last forward pass. The key difference that distinguishes the FNO from other neural operators is the fact that the kernel-integral operator $K^{(l)}$ in each generic hidden-layer (5) is constrained to a translation-invariant convolution operation in the Fourier space:

$$K^{(l)}(u) := \mathcal{F}^{-1}\big(R^{(l)} \circ \mathcal{F}\big)(u) \tag{7}$$

where $\mathcal{F}, \mathcal{F}^{-1}$ are the Fourier and inverse Fourier transforms respectively. In the discrete implementation, we learn the weights directly in the Fourier domain through the matrix $R^{(l)}$, and we use F, F^{-1} for discrete Fourier transform (DFT) and its inverse.

Learnable Parameters

The complete set of learnable parameters in the FNO architecture can be represented as:

$$\theta = \big[W^{(l)}, R^{(l)}, P, Q\big] \tag{8}$$

where $W^{(l)}$ represents the weights in the local linear map for residual connections, $R^{(l)}$ appears in the Fourier layer (7) as the complex Fourier multiplication matrix at each layer l. P, Q

Operator Learning

The mapping $\mathcal{G}_\theta$ that is parameterized by the set of learnable FNO parameters (8) can be learned by approximating the solution operator $\mathcal{G}$ from observations:

$$\Big\{u_0^{(i)}, \mathcal{G}\Big(u_0^{(i)}\Big)\Big\}_{i=1}^{N} \tag{9}$$

where $u_0^{(i)} \sim \mu, \forall 1 \leq i \leq N$ is an i.i.d. sequence drawn from some probability measure μ on the function space. In such discretized setting, we aim to minimize the following L2-norm:

$$J(\theta) := \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} w_j \left\| \underbrace{\mathcal{G}\Big(u_0^{(i)}\Big)(y_j)}_{\text{Ground Truth}} - \underbrace{\mathcal{G}_\theta\Big(u_0^{(i)}\Big)(y_j)}_{\text{Approximation}} \right\| \tag{10}$$

For each discretization point y_j and quadrature weight w_j, we compute the L2-norm difference between the ground truth solution $\mathcal{G}\Big(u_0^{(i)}\Big)(y_j)$ and our neural operator approximation $\mathcal{G}_\theta\Big(u_0^{(i)}\Big)(y_j)$. Here, j indexes over M spatial domain points and i over N function space samples, with the loss averaged over all samples.

Task 1: One-to-One Training

Description

- ✓ Use **64** trajectories from the training dataset.
- ✓ Select the first (t = 0.0) and last (t = 1.0) time snapshots for these trajectories.
- ✓ Train an FNO model to learn the mapping: $\mathcal{G} : u_0 \to u(t = 1.0)$
- ✓ Use the remaining trajectories for validation.
- ✓ Test the trained model on the `test_sol.npy` dataset, focusing **only** on predictions at t = 1.0 (the map $u_0 \to u(t = 1.0)$).
- ✓ Report the **average relative L2 error**

Model Architecture

Our FNO implementation follows the architecture proposed by Li et al. [2], which we already detailed in the preliminary section. In our implementation, the `SpectralConv1d` class corresponds to the Fourier integral operator, whereas `FNO1d`[1] corresponds to the entire multi-layer FNO architecture as in (6). We used `nn.Linear` for the implementation of both lifting and projection layers P, Q.

$$\mathcal{G}_\theta := \underbrace{Q}_{\text{Projecting}} \circ \mathcal{G}_\theta^{(L)} \circ \mathcal{G}_\theta^{(L-1)} \circ ... \mathcal{G}_\theta^{(1)} \circ \underbrace{P}_{\text{Lifting}} \tag{11}$$

Then for each Fourier layer $\mathcal{G}_\theta^{(l)}$, the activation function σ is selected to be `GELU`. We have experimented around various nonlinear variants that are provided by the PyTorch library such as the classical ReLU, Tanh, GELU. Among those, GELU has shown the best results on our implementation, which is a smoothened version of ReLU. Bias and residual connections are also included in each Fourier layer.

$$\begin{aligned} \mathcal{G}_\theta^{(l)}(u) &:= \sigma\big(W^{(l)} + K^{(l)} + b^{(l)}\big)(u), \text{where } l = 1, ..., L \\ &= \sigma\left(W^{(l)}(u) + \underbrace{\mathcal{F}^{-1}\big(R^{(l)} \circ \mathcal{F}\big)(u)}_{\text{Fourier Integral Operator}} + b^{(l)} \right), \text{where } l = 1, ..., L \end{aligned} \tag{12}$$

For Fourier integral operator as we highlighted in equation (12), we used the real FFT `torch.fft.rfft` and `torch.fft.irfft` for inverse FFT back into the physical space in our Fourier layer for more cost-efficient computation. This is permissible because the provided datasets consist of real-valued data. The model-specific parameters can be seen in the table 1 as shown below:

Number of modes	Fourier Layer Depth	Fourier Layer Width	Activation Function
30	2	16	GELU

Table 1: Model Architecture Parameter.

Training Strategy

We used the provided `train_sol.npy` dataset for training, which consists of 128 trajectories at all 5 time snapshots. Each trajectory at a time snapshot is reported by 64 equally-spaced nodal values that represent the solution of the system of PDEs (2) at a given time t. However, for **One-to-One training**, we will **only make use of the time snapshots at $t = 0$ and $t = 1.0$** to train our model for predicting $u_0 \mapsto u(\cdot, t = 1)$. We split the 128 trajectories into 64 trajectories for training, and the remainder 64 trajectories is kept to perform cross-validation.

[1] Model architecture implementation locates under `project_1/fno.py`

[2] locates under `project_1/dataset.py`

For convenience, we provide a `OneToOne`[2] class that provides data in form as we discussed above directly. It inherits from the abstract `Dataset` class from the `torch.utils.data` module.

In order to align with the nature of the wave equation (1) that is second-order in time, we provide both initial conditions explicitly by passing the initial velocity $v_0 := u_t(x, 0) = 0$ to our model in a separate input channel.[3] In Table 2, we listed out the relevant parameters used in the training process, we opted for an AdamW optimizer as it performed the best throughout testings. `StepLR` scheduler is used with multiplicative factor of learning rate decay `gamma = 0.1`.

Training Samples	Validation Samples	Early stopping	Maximal Number of Epochs
64	64	After 80 epochs	800
Batch Size	**Step size**	**Learning Rate**	**Optimizer**
5	300	$1 \cdot 10^{-3}$	AdamW

Table 2: Training configuration for FNO.

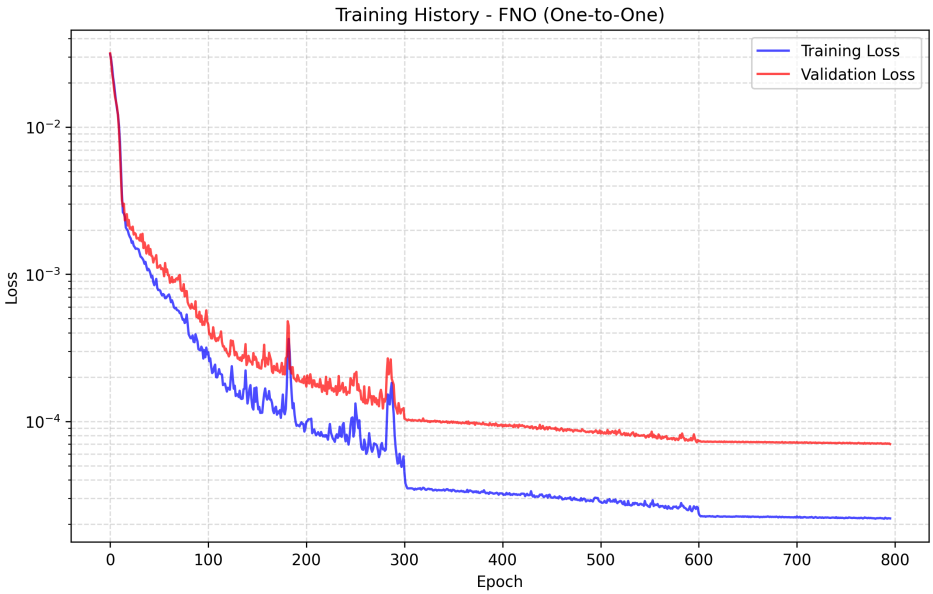

Figure 1: Training history of our FNO model when providing both u_0, v_0 by passing it to a separate channel.

Results and Observation

In Figure 1, we included the training history of the FNO implementation that we will use extensively throughout next task 1-3. For task 1, we report the **average relative L2 error**, which is defined over 128 trajectories for solutions of the evolution problem at $t = 1$:

$$\text{err} = \frac{1}{128} \sum_{n=1}^{128} \frac{\left\| u_{\text{pred}}^{(n)}(t = 1.0) - u_{\text{true}}^{(n)}(t = 1.0) \right\|_2}{\left\| u_{\text{true}}^{(n)}(t = 1.0) \right\|_2} \tag{13}$$

On `test_sol.npy` dataset, predictions at $t = 1.0$ with a spatial resolution have the following average relative L2 errors. Additionally, we also report the results from not explicitly passing $v_0 = 0$. These results will also be used in the comparison with error from OOD data from task 3:

FNO (u_0)	FNO (u_0, v_0)
5.05%	5.21%

Table 3: Average relative L2 error.

[3]For task 1-3, whether we explicitly provide v_0 does not make a big difference. However, for All2All training we find it to be necessary. Thus, we decide to stay consistent and use models with explicit v_0 throughout the report.

Task 2: Resolution Study

Description

- ✓ 1. Test the trained model from Task 1 on the datasets `test_sol_res_{s}.npy` for s ∈ {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?

Theory: Neural Operators for Zero-shot Super-resolution

An operator that is capable of doing **zero-shot super-resolution** can evaluate inputs on a higher resolution in a zero-shot manner, despite the fact that it was trained on a lower resolution. Specifically in the case for the FNO, recall that we have learned parameters directly in the Fourier space. When we want to resolve functions in the physical space, we can simply project on the basis $e^{2\pi i\langle x,k\rangle}$. This is well-defined everywhere on $\mathbb{R}^d$ [2].

Results and Observation

Now we examine the **zero-shot super-resolution** property of neural operators. The Fourier neural operator $\mathcal{G}_\theta$ that we have trained on $128 \times 5 \times 64$ resolution via One-to-One training[4] is evaluated on testing datasets of $128 \times 2 \times s$ resolution, with varying spatial resolution indicated by $s \in \{32, 64, 96, 128\}$. Since we again have 128 trajectories per resolution as testing data, we can use the same error formula (13) as before.

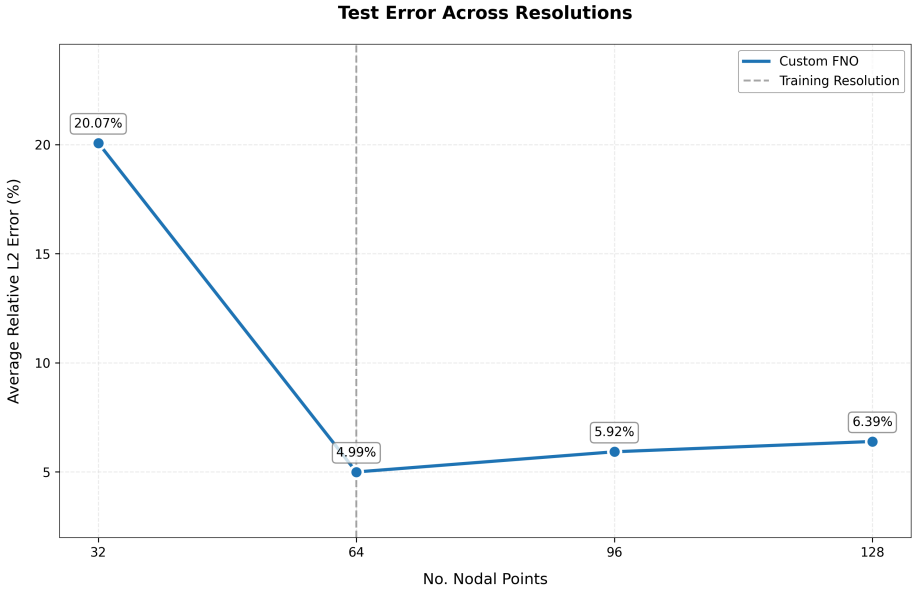

Figure 2: Average relative L2 error for the custom FNO implementation for 128 training trajectories. We marked the training spatial resolution with 64 nodal points with the grey dashed line.

Smallest Errors at Trained Resolution ($s = 64$)

In Figure 2, the lowest errors on test data can be found at the training resolution $s = 64$, which is indicated by the grey dashed line on the plot. Our implementation shows an average relative L2 error of 4.99%.

[4]Model parameters as wl as the training parameters can be seen under Table 1 & Table 2 in task 1.

Success in Higher Resolutions ($s > 64$)

More importantly, for higher resolutions which is at $s \in \{96, 128\}$, we achieve accuracy of 5.92% and 6.39% respectively for our implementation. This demonstrates our implementation can perform zero-shot super-resolution, which is an important property of the FNO.

Failure in Lower Resolution ($s < 64$)

A notable observation is the failure of the FNO in resolving lower resolution. In Figure 2, We can see both implementations exhibit highest error in coarser grid that consists of 32 nodal points. This is an expected behavior as we discussed in class, reflecting the fact that the *FNO is not a Representation Equivalent Neural Operator (ReNO)*. The continuous-discrete equivalence (CDE) property is not preserved. This caveat is brought by the choice of nonlinear activation functions. In Bartolucci et al. [3], it was shown that nonlinear activation function drives us out of bandlimited function space, introducing aliasing error. For lower resolutions, the high-frequency pollution introduced by nonlinear activation functions strongly affect coarse grid resolution and we fall into the zone where there is no representation equivalence. Our custom implementation gives an 20.07% error.

Conclusion

To conclude task 2, we reported the average relative L2 error in Figure 2. We verified our implementation can do zero-shot super-resolution by showing it exhibits similar error magnitudes in $s \in \{96, 128\}$, compared to the training resolution where $s = 64$. However, we observed the failure of FNO in resolving lower resolutions, reflecting the fact that FNO is not a ReNO and the quality of approximated solutions is resolution-dependent.

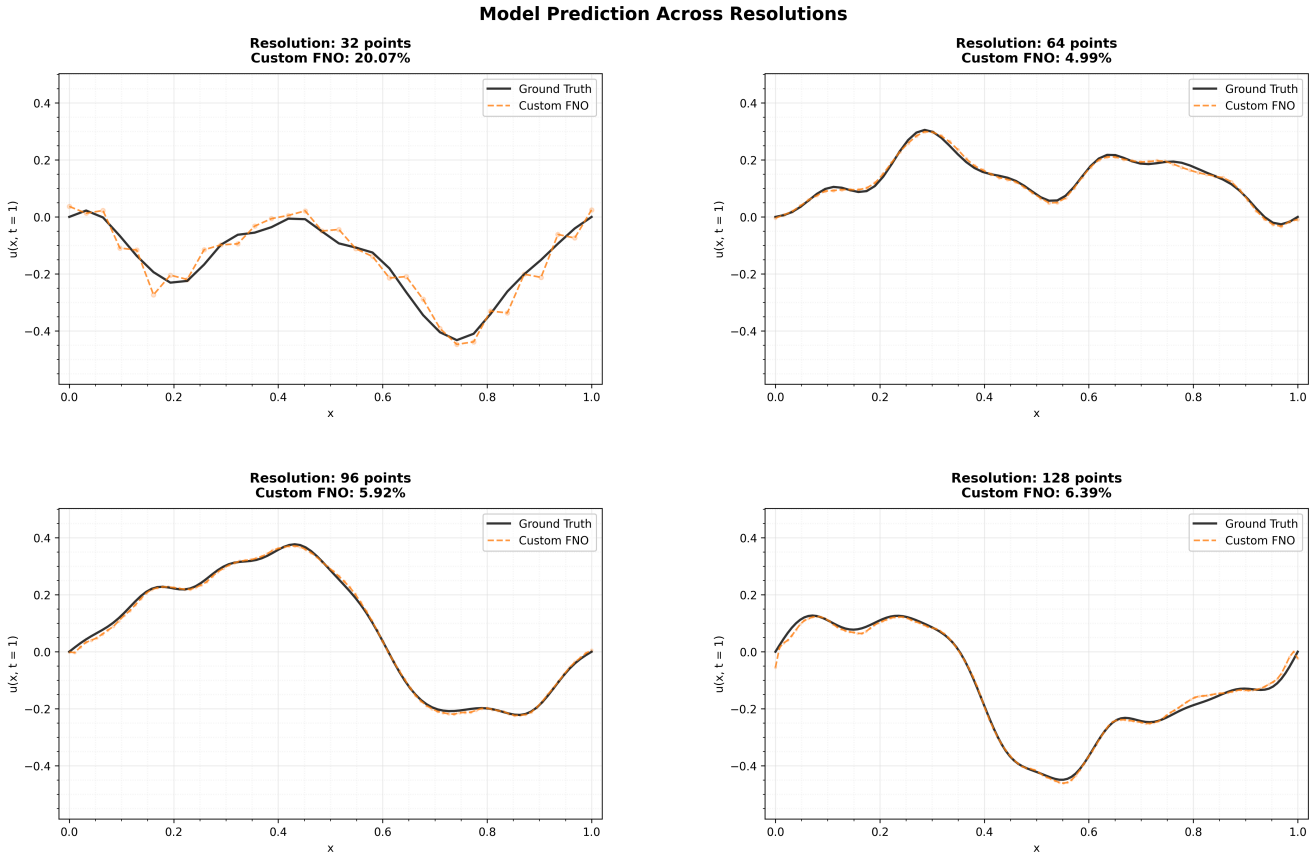

Figure 3: For completeness, we reported the model prediction for first trajectory of each `test_sol_res_{s}.npy` data set.

Task 3: Out-of-Distribution Analysis

Description

- ✓ 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?

In **task 3**, we test our final solution on OOD dataset that is of resolution $128 \times 2 \times 64$. Since no other spatial resolutions are provided, we will only compare our results on OOD dataset against the ones obtained from **task 1** throughout this task.

Theory: Out-of-Distribution Generalization

Recall our implementation approach to resolve the 1D wave equation with its associated evolution problem: we considered the solution operator $\mathcal{G}$ that maps between infinite-dimensional function spaces, and approximated it with the trained Fourier Neural Operator (FNO), which we denoted as $\mathcal{G}_{\#\mu} := \mathcal{G}_\theta$ in this task.

The aim is to learn $\mathcal{G}_{\#\mu}$ by approximating the solution operator $\mathcal{G}$ from observations:

$$\left\{u_0^{(i)}, \mathcal{G}\left(u_0^{(i)}\right)\right\}_{i=1}^{N} \tag{14}$$

where $u_0^{(i)} \sim \mu, \forall 1 \leq i \leq N$ is an i.i.d. sequence drawn from some probability measure μ on the function space.

In-Distribution Testing

So far, we have examined our operator $\mathcal{G}_{\#\mu}$, that has been trained on random testing samples drawn from μ on both task 1 and 2 on datasets `test_sol.npy` and `test_sol_res_{s}.npy` for $s \in \{32, 64, 96, 128\}$. These are in-distribution datasets, meaning that the initial data distribution of the test data was chosen as $u_0 \sim \mu$, which has been seen by $\mathcal{G}_{\#\mu}$ during pretraining.

Our model has demonstrated in both tasks its ability to both resolution on the same spatial resolution as it was trained on, as wl as to do super-resolution.

Out-of-Distribution (OOD) Testing

One question that may arise with using in-distribution dataset for testing is whether $\mathcal{G}_{\#\mu}$ has actually learned the underlying physics, instead of learning the particular data distribution μ that was trained on.

To examine this, we utilize the OOD dataset `test_sol_OOD.npy`. Here we consider another input measure v, supported on the same function space, and choose the OOD initial data $\nu \sim v$ accordingly, such that $\nu \approx \mu$ but $\nu \neq \mu$.

Results and Observation

To better compare our model performance for both testing methods, we report the error distribution in the form of histogram in this section.

In Figure 4, we compare the relative L2 test errors for both the in-distribution testing dataset, and the out-of-distribution (OOD) testing dataset. Both testing dataset consists of 64 nodal values as its spatial resolution. For comparison, we again utilized both FNO implementation.

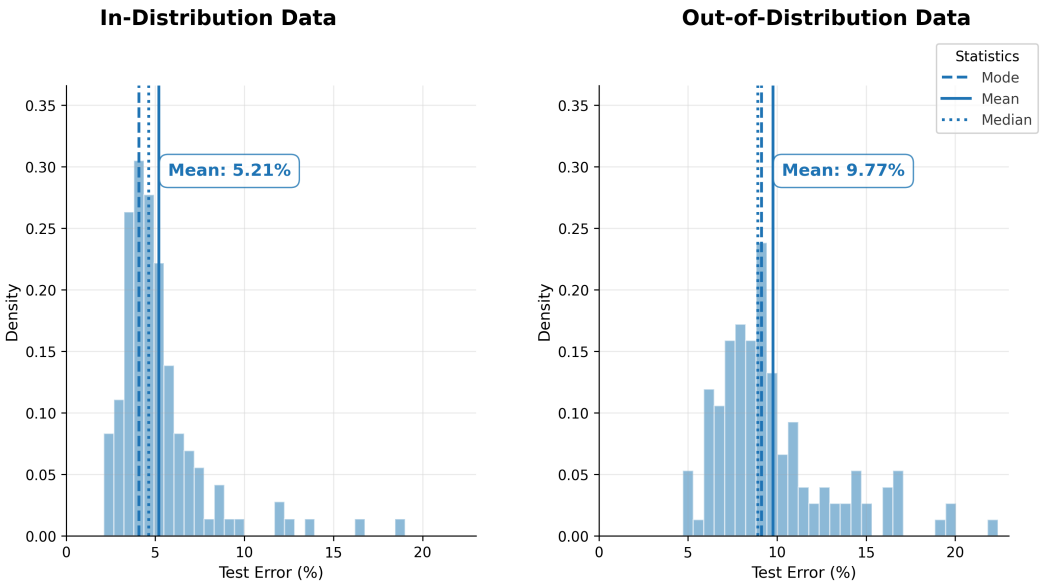

Figure 4: Histogram distributions of relative L2 test errors over 128 testing trajectories for our FNO implementation. Results show performance on both in-distribution **(task 1)** and out-of-distribution **(task 3)** test data, with vertical lines indicating the mode (dashed), mean (solid), and median (dotted) of each distribution.

As reported in the error distribution plot above, the average L2 test errors for in-distribution data are lower. Our implementation achieved 5.21% of average relative L2 error across 128 trajectories.

For OOD data, we observed our models performed worse regardless of the implementation details. Both models exhibit similar errors, our implementation has an average L2 error of 9.77% in resolving the test data.

Conclusion

Compared to task 1, the average relative L2 error we obtained in OOD dataset is higher. This is matching our expectation because the nature of the OOD dataset makes it inherently harder for our model to predict, because the initial data $v \sim \nu$ used in the OOD dataset has not been seen by our model in pretraining process.

Task 4: All2All Training

Description

- ✓ 1. Use 64 trajectories from the training dataset.
- ✓ 2. Use all provided time snapshots for these trajectories to train a time-dependent FNO model.
- ✓ 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?

Training Strategy: Vanilla vs. All2All

In our implementation of the `All2All` class for loading training dataset, we provide two possible training strategies in relation to the utilization of temporal data. The first one is the **one-to-all training (vanilla)** where the aim is to approximate the solution operator of the system (2) for some final observation time t_{out}, where the system evolves from an initial distribution $u_0 = u(\cdot, t = 0)$. In this strategy, the starting time is fixed to $t = 0$ but the end time can vary.

An alternative that can better make use of the training data is the **All2All strategy** as proposed in Herde et al. [4]. In this strategy, both the starting time as well as the end time can be chosen freely. As long as it preserves the ordering of time $t_{\text{in}} \leq t_{\text{out}}$. It boils down to train for the following mapping with some fixed timestep size Δt, starting from t_{in} such that $t_{\text{out}} = t_{\text{in}} + \Delta t$. The system evolves from some intermediate distribution $u_{t_{\text{in}}}$:

$$\mathcal{G}_\theta : u_{t_{\text{in}}} \mapsto u(\cdot, t_{\text{in}} + \Delta t) \tag{15}$$

Approximating Velocities with Finite Difference

⚠️ Through trial and error, we found out that it is very essential in **All2All** training strategy to ensure that the velocity at the input time $v_{t_{\text{in}}}$ is explicitly provided to some input channel of the model. When using **One-to-One** or **one-to-all** training, the model can implicitly learn that $v_0 = 0$ if we consistently provide input data from the same starting point $t = 0$. However, when performing **All2All** training our input distribution $u(\cdot, t)$ may not necessarily be at any intermediate time point that the dataset spans over, and we need to explicitly provide the data such that our FNO can learn the underlying function of $u_t(\cdot, t)$. Therefore, for consistency and to better align with the nature of the underlying system, we approximate the velocities $v_i := u_t(\cdot, t_i)$ for all $i \in \{0, 1, 2, 3, 4\}$ for 5 time snapshots with finite difference. The approximated velocities v_i are passed to an input channel of our FNO, together with u_i, Δt and an array representing the 1D discretization of the domain Ω.

Total Number of Samples

The biggest difference between two strategies lie in the way how the training data is utilized. Since we are not constrain to learn our solution operator starting from the initial distribution u_0, we can learn $\mathcal{G}_\theta$ starting from any intermediate distributions $u(\cdot, t)$, as long as t is one of the time points where snapshots of the training data are available. As a result, All2All allows us to utilize quadratic $O(k^2)$ samples per trajectory whereas the vanilla training only makes use of $O(k)$ training samples per trajectory. In the table below, we report the number of samples used for both strategies with the provided training dataset `training_sol.npy`. **For All2All approach, the total number of samples used for training are 1920, where 960 samples are dedicated for training and the other 960 for validation purpose.** We also took the ordering of time points into account and ensure $t_{\text{in}} \leq t_{\text{out}}$.

Metric	One-to-All Training	All2All Training
Training Trajectories	64	64
Validation Trajectories	64	64
Sample Pattern	(t_0, t_i) sequential for $i \in [k]$	$\frac{k\times(k+1)}{2}$ combinations
Samples per Trajectory	5	$\frac{5\times6}{2} = 15$
Total Training Samples	$64 \times 5 = 320$	$64 \times 15 = 960$
Total Validation Samples	$64 \times 5 = 320$	$64 \times 15 = 960$
Total Samples	$320 + 320 = 640$	$960 + 960 = 1920$

Table 4: Number of samples used for one-to-all (vanilla) and All2All strategies for $k = 5$ timesteps

Model Architecture

In order to account for time inputs as an extra dimension, we need to modify our `FNO1d` class, which corresponds to the multi-layer architecture in equation (6). We added `FILM` layers in our model. If `time_dependent = True` is specified, the temporal information Δt will be passed through such time-conditional batch normalization layer. The `SpectralConv1d` class for the Fourier integral operator is kept unchanged.

For training, we used same parameters as in table 1 & 2 for the model in task 1, with the only difference that we increased the width of our model from 16 to 32. The `time_dependent = True` parameter is also set for this training. We tested our trained models with two different training strategies on the `test_sol.npy` dataset, focusing **only** on predictions at $t = 1.0$. We trained two time-dependent FNO models with identical hyperparameters, but only differs the way how we utilize the input data. Following, we will report results from time-dependent FNO trained from both **one-to-all (vanilla)** and **All2All** strategies.

Results and Observation

In table 5, we report the average relative L2 error (13) as before over 128 trajectories, tested on the `test_sol.npy` dataset for $t = 1.0$ only. For conciseness, we do not include the training history here, but it can be found under the `project_1/checkpoints` folder for each model used.

One-to-One (Task 1)	One-to-All: Direct Inference	All2All: Direct Inference
5.21%	7.74%	8.20%

Table 5: Average relative L2 error from our FNO implementation with different training strategies, evaluated for the in-distribution dataset at $t = 1.0$. The result from One-to-One training is taken from task 1.

As we can tell from table 5, models based on our time-dependent FNO implementation can resolve the solution of the PDE system at $t = 1.0$ at a rather high accuracy. **Time-dependent FNO results (7.74% and 8.20%) are indeed slightly worse than the time-independent model (5.21%) from task 1.** But this may due to the fact that we have a more complex loss function to minimize and our training procedure could be further improved. Furthermore, we do not observe a huge difference in results obtained from one-to-all (vanilla) or All2All strategy yet. Both time-dependent FNO models exhibit errors of similar magnitude at $t = 1.0$. **In all, we conclude that our time-dependent training is successful.** This success is due to the fact that we approximate the velocity $u_t(\cdot, t)$ from $u(\cdot, t)$ via finite difference, and pass both data as initial conditions of the IBVP (2) when training our model. Thinking about the fact that we have a wave equation which is second-order in time, providing two initial conditions appears to be very natural. This way we can ensure our problem is well-posed.

So what would happen if only $u(\cdot, t)$ is provided, without approximating $u_t(\cdot, t)$ and providing the second initial condition to a separate input channel? We attempted this earlier and observed that, for the FNO that was trained on one-to-all time pairs, it had worse performance ($\sim 10\%$ of error), but it was still demonstrating correct behavior. However, for All2All, it was devastating and led to errors of very high magnitude ($\sim 30 - 40\%$).

Bonus Task

Description

- [x] 1. Use the model from Task 4 to make predictions at multiple time steps.
- [x] 2. Compute the average relative L2 error for each time step.
- [x] 3. What do you observe about the model's performance over time?
- [x] 4. Use the model from Task 4 to make predictions on the OOD dataset at t = 1.0.
- [x] 5. What do you observe about the model's performance?

Results and Observation

In this section, we consider only the model trained using All2All strategy. In Figure 5, we report results at various time points using trained models from task 4. We report the **average relative L2 error** across 128 trajectories with the formula (13).

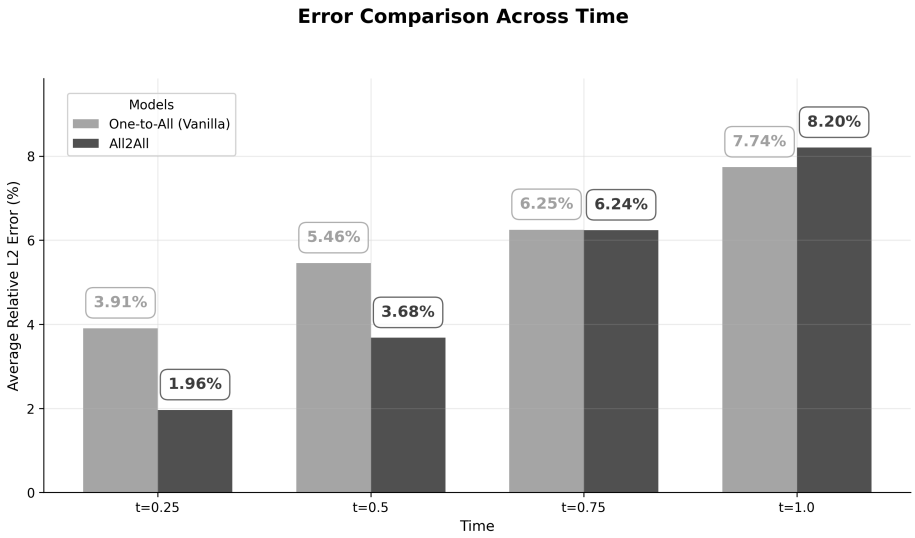

Figure 5: Comparison of average relative L2 errors obtained via time-dependent FNO trained by one-to-all and All2All strategies for 128 trajectories across time, where $t \in \{0.25, 0.5, 0.75, 1.0\}$. The direct inference is used.

In Figure 5, we report the errors between the ground truth and our predictions using direct inference from two models. Overall, models trained via different data loading strategies show similar performance over time. However, when smaller timestep sizes are used in direct inference, the All2All model shows better performance with 1.96% and 3.68% of error for results at $t = 0.25$ and $t = 0.5$, about halved comparing to 3.91% and 5.46% of error obtained via the one-to-all model.

For $t = 1.0$, the one-to-all model outperforms slightly, we have seen these results from task 4 already. The reason why the one-to-all model outperforms is probably due to the fact that it was targeted to perform mapping from $t = 0.0$ to the final observation time $T = 1.0$, and thus does a better job than the more versatile All2All model here.

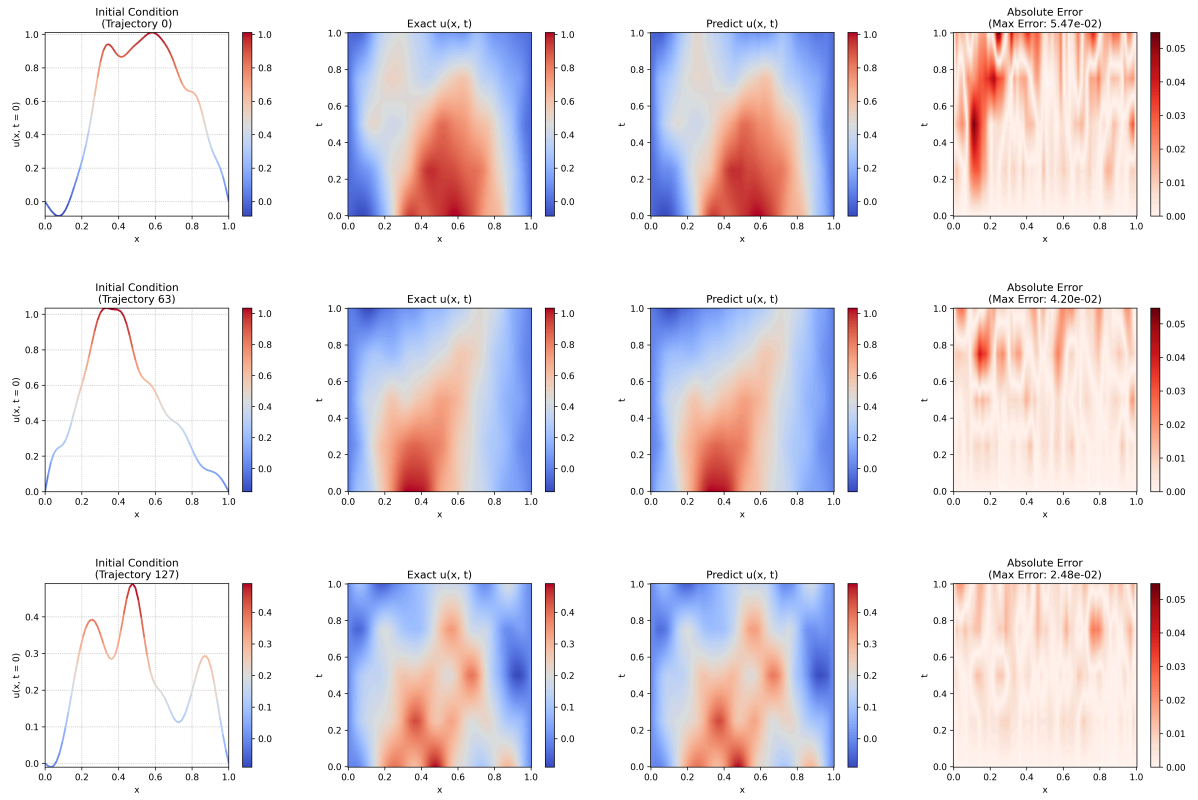

Figure 6: The trajectories with index 0, 63 and 127 in the `test_sol.npy` dataset are reported. The heatmap is plotted with linear interpolation across temporal resolutions. The predictions for intermediate time $t \in \{0.25, 0.5, 0.75, 1.0\}$ are obtained via direct inference using the all2all trained model.

For completeness, we report the space-time solution predicted by the All2All model via direct inference for the system of PDEs (2) from $t = 0$ to the final observation time $T = 1.0$. As shown in Figure 6, **our FNO is capable of resolving the nuances of the ground truth across time**, with largest error of only 1×10^{-2} in magnitude. We can also observe that the errors are larger at later time t. This is mainly due to the fact that the results above are obtained via direct inference. When the Δt used in direct inference is large, it may be hard for the model to directly predict the results because the solution deviates more from the initial state.

OOD Testing

Next, we tested our model on the OOD dataset. As shown in the table below, both time-dependent models perform worse on the OOD dataset, showing $\mathbf{13.02\%}$ and $\mathbf{14.16\%}$ of error, almost doubled comparing to results 7.74% and 8.20% from the same model respectively from task 4 on the in-distribution dataset. This shows a consistent behavior as we observed before on the One-to-One trained model due to the limit of the FNO architecture itself to correctly capture unseen data.

Nevertheless, the results are not too bad because time-dependent FNOs only show slightly higher errors on the OOD data, compared to the 9.77% of error, which we obtained in task 3 using the plain FNO model, independent of time.

Dataset Type	One-to-One	One-to-All: Direct Inference	All2All: Direct Inference
In-Distribution	5.21% (Task 1)	7.74% (Task 4)	8.20% (Task 4)
OOD	9.77% (Task 3)	**13.02%**	**14.16%**

Table 6: Average relative L2 error for custom FNO implementations with different training strategies, evaluated for the in-distribution and out-of-distribution (OOD) datasets at $t = 1.0$. Both consist of 128 trajectories.

Appendix

Even though All2All strategy is computationally more expensive, but the fact that we can increase the number of samples from $O(k)$ to $O(k^2)$ samples per trajectory makes it attractive to try. We also expect a better performance from the model trained via All2All data loading strategy than the one-to-all (vanilla) strategy accordingly.

However, in Table 5 of Task 4, it was shown that one-to-all and All2All exhibit errors of similar magnitude. Moreover, one-to-all strategy even had a lower error. One may ask, is All2All strategy really beneficial?

Theory: Inference Strategy

To better compare our time-dependent FNO models trained on different data loading strategies, we want to analyze the temporal stability of both models when performing **autoregressive rollouts**. Before we dive directly into the question, we would like to recap existing inference strategies and explain what the concept of autoregressive rollout means.

In Figure 7, we demonstrate two inference strategies for evaluating our time-dependent model at a desired time t. The red arrow indicates the **direct inference** strategy. By evaluating $\mathcal{G}_\theta(t = 1, u_0)$ directly using the initial distribution $u_0 = u(\cdot, t = 0)$, we can directly obtain the solution with just one model inference. This approach has the advantage that we has low inference cost and there are no accumulative errors. However, it could lead to unstable and false results if there are large changes in the physics of the model problem across time.

On the other hand, the **autoregressive inference** as illustrated with blue arrows requires multiple inferences, but it may provide better predictions for the solution at the final time despite the fact that it also accumulates errors everytime when it performs an inference. For our specific example, after $k = 4$ successive applications of the trained FNO, we can approximate the solution operator at time $t = 1.0$ by:

$$\mathcal{G}(t = 1, u_0) \approx \mathcal{G}_\theta(t_4 - t_3, \mathcal{G}_\theta(t_3 - t_2, \mathcal{G}_\theta(t_2 - t_1, \mathcal{G}_\theta(t = 0, u_0)))) \tag{16}$$

where we defined $t_i := i\Delta t$ with $i \in \{0, 1, 2, 3, 4\}$, $\Delta t = 0.25$.

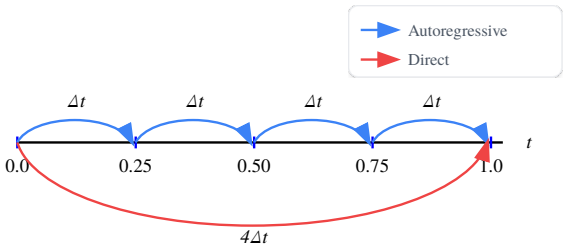

Figure 7: Comparison of different inference strategies

Comparison: Vanilla vs. All2All

We consider again the `test_sol.npy` dataset that consists of 5 time snapshots. In contrast to the direct inference in which we take $\Delta t = 1.0$ to reach the final time $t = 1.0$ directly, we will now perform autoregressive inference, in which we take several time steps. Each timestep taken is of the size that is some multiple of the base unit $\Delta t = 0.25$.

Timestep Sequence (No. of Δt)	One-to-All (Vanilla)	All2All
1+1+1+1	128.45%	78.45%
1+1+2	116.08%	61.26%
1+2+1	109.51%	62.60%
2+1+1	105.47%	67.04%
2+2	105.26%	54.56%
1+3	75.56%	65.89%
3+1	63.19%	57.04%
4 (Direct)	**7.74%**	**8.20%**

Table 7: The table reports the average relative L2 error at $t = 1.0$ between ground truth and our model predictions. We consider all possible combinations from autoregressive rollouts with $\Delta t = 0.25$, and report errors from two models trained with different data loading strategy, All2All and one-to-all (vanilla) here. The timestep sequence indicates the way how timesteps are taken to reach the final time $t = 1.0$.

Results

In the table above, we reported results obtained from autoregressive evaluation considering all possible timestep combinations. First thing one can confirm is the fact that inference strategies are also of high importance and can significantly affect the quality of results we obtain. Overall, both time-dependent models, regardless of the data loading strategy, exhibit significant weaknesses on performing autoregressive evaluations. This may indicate FNO's weaknesses in capturing the true underlying physics of the model across time.

However, there is still something positive. We can confirm from the autoregressive results that the model trained on All2All strategy performs way better than the model trained on one-to-all strategy. The All2All strategy can indeed help us to maximize usage of temporal data.

Some Personal Thoughts

In fact, we could potentially further extend the All2All strategy in which we also allow timepairs $(t_{\text{in}}, t_{\text{out}})$ to have reversible ordering, meaning that we loosen the constraint and allow for also $t_{\text{in}} > t_{\text{out}}$. This is inspired by the fact that reversibility of the time-stepping method in traditional numerical simulations is a highly desirable property. If we allow our model to approximate solutions by going back-ward in time, we not only have more training data available, but can also potentially make our solution operator structure-preserving.

References

[1] N. Kovachki *et al.*, "Neural Operator: Learning Maps Between Function Spaces," *Journal of Machine Learning Research*, vol. 23, pp. 1–97, 2021, doi: 10.5555/3648699.3648788.

[2] Z. Li *et al.*, "Fourier Neural Operator for Parametric Partial Differential Equations," *ICLR 2021 - 9th International Conference on Learning Representations*, 2020, [Online]. Available: https://arxiv.org/abs/2010.08895v3

[3] F. Bartolucci, E. de Bézenac, B. Raonić, R. Molinaro, S. Mishra, and R. Alaifari, "Representation Equivalent Neural Operators: a Framework for Alias-free Operator Learning," *Advances in Neural Information Processing Systems*, vol. 36, 2023, [Online]. Available: https://arxiv.org/abs/2305.19913v2

[4] M. Herde *et al.*, "Poseidon: Efficient Foundation Models for PDEs," 2024, [Online]. Available: https://arxiv.org/abs/2405.19101v2