DEEPFDM: A SCIENTIFIC COMPUTING METHOD FOR NEURAL PARTIAL DIFFERENTIAL EQUATION (PDE) OPERATORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Solving Partial Differential Equations (PDE) has long been a critical challenge in many scientific and engineering domains. Recently, neural networks have shown great promise in solving PDEs by learning solution operators from data, offering a flexible and adaptive alternative to traditional numerical solvers. Despite these advancements, there is still a need for systematic benchmarking of neural operator methods against conventional approaches and for the development of datasets representing diverse distributions for robust evaluation.

In this paper, we introduce DeepFDM, a benchmark method for learning PDE solution operators based on numerical PDE solvers. DeepFDM leverages the structure of the PDE, in order to achieve better accuracy and generalization compared to neural solvers. It is designed as a solver for a specific class of PDEs and not as a replacement for neural solvers. Moreover, because DeepFDM learns the coefficients of the PDEs, it offers inherent interpretability. We also introduce a principled method for generating training and test data for PDE solutions, allowing for a quantifiable measure of distribution shifts. This method provides a structured approach to evaluate the out-of-distribution (OOD) performance of neural PDE operators.

Our work sets a foundation for future comparisons of neural operator methods with traditional scientific computing approaches, providing a rigorous framework for performance benchmarking, at the level of the data and at the level of the neural solver.

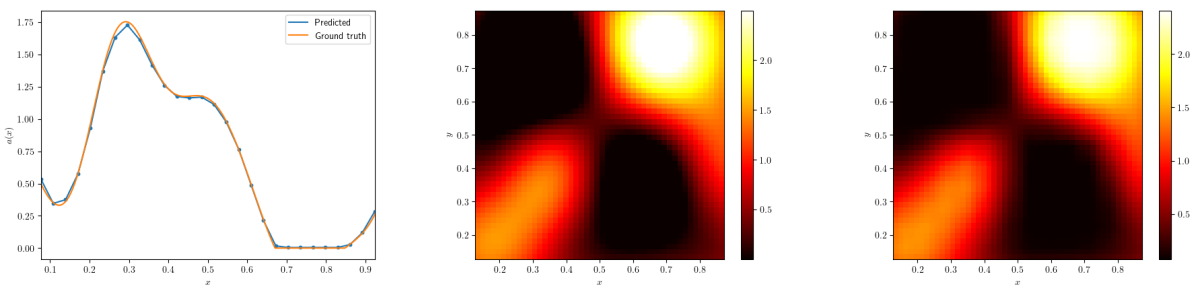
1 INTRODUCTION

Solving Partial Differential Equations (PDE) has long been a critical challenge in many scientific and engineering domains. Recently, neural network methods have been applied to solve PDEs, with impressive results (Karniadakis et al., 2021; Lu et al., 2021; Li et al., 2020a;b). These methods learn PDE solution operators from PDE solution data sets. They bypass the strict requirements of numerical solvers by treating the problem of solving PDEs as a problem of learning from data. Neural network methods offer a highly flexible and adaptive approach to solving PDEs, compared to traditional numerical PDE solvers, as they can work with a wide variety of equations and input data.

Although recent work has focused mainly on performance gains, with the large number of recent contributions, there is a need for (i) benchmarking neural operator methods against standard scientific computing approaches and (ii) generating test and training datasets from different distributions.

Contribution In this work, we implement a benchmark scientific computing approach to PDE operator learning, DeepFDM, based on numerical PDE solvers and numerical inverse problems. In contrast to traditional inverse problem methods, DeepFDM is implemented as a feedforward convolutional neural network and works with a family of PDEs. This allows it to be used on benchark PDEs and trained using standard neural network methods. Because DeepFDM is designed to work with a specific family of PDEs, it is expected to have better accuracy and generalization than neural PDE solvers. As such, it will be useful for benchmarking the accuracy and generalization of the more flexible neural PDE operator methods. See Table 1. We propose it as a benchmark method for neural solvers, since it corresponds to the scientific computing ground-truth solution of a given PDE, with a given dataset.

DeepFDM works by learning the coefficients of the PDE from the data. DeepFDM is not a replacement for existing neural operator methods, as it leverages the more structurally constrained PDE solver framework. In particular, it will lose accuracy on PDEs outside the class for which it was designed. However, when applied to PDEs in the class, it is both theoretically justified and empirically validated, outperforming current neural PDE solvers in both test accuracy and out-of-distribution (OOD) accuracy. Furthermore, because this method is based on learning the coefficients of PDEs, it is inherently interpretable; see Figure 1.



(a) 1D diffusion process (b) 2D ground truth coefficients (c) 2D learned coefficients

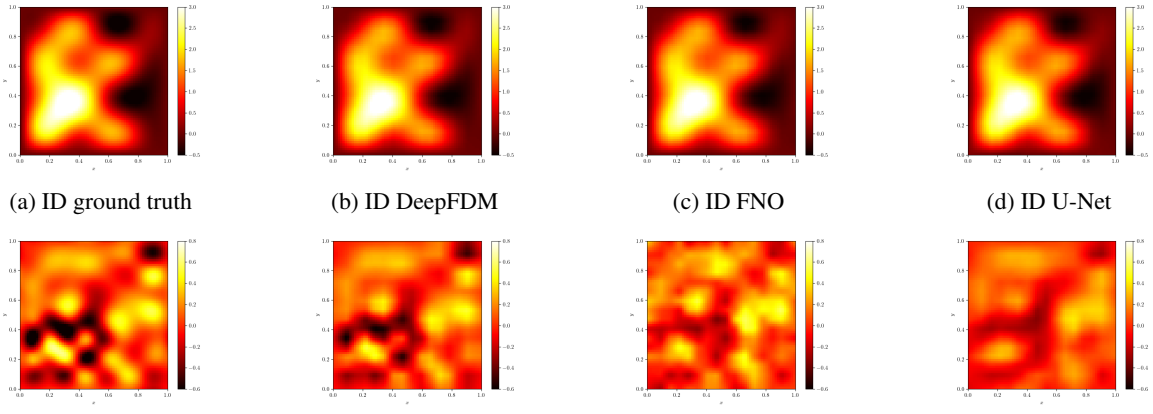
Figure 1: DeepFDM leverages the extra assumption of a PDE structure to learn the PDE coefficients and to implement the corresponding numerical PDE solver. Learned coefficients: (a) 1D diffusion process, (b) and (c) 2D diffusion process ground truth and learned coefficients.

We also introduce a principled method for measuring out-of-distribution (OOD) performance of neural operators. We generate PDE solution data by choosing families of orthogonal functions with random coefficients for the initial conditions and passing these functions through a high-accuracy numerical PDE solver. Using different distributions on the coefficients allows us to quantify the distance between the distribution (over functions), in terms of the Hellinger distance. These results are illustrated in Figure 2, and quantified in Figure 4a.

Together, the scientific computing-based benchmark solver, DeepFDM, and the quantitative generation of OOD datasets provide a foundation for future development of neural PDE operators.

	ResNet	U-Net	FNO	DeepFDM
Diffusion	0.6149	0.0640	0.0266	**0.0024**
Advection	0.7039	0.0618	0.0251	**0.0007**
Advection-Diffusion	0.6286	0.0692	0.0307	**0.0017**
Reaction	0.9119	0.0521	0.0319	**0.0016**
Burgers	0.8517	0.0790	0.0379	**0.0045**

Table 1: Test error (normalized MSE) of our model and various benchmarks on a diverse set of PDE problems; by leveraging the additional assumption of a PDE structure, the more restrictive DeepFDM method outperforms neural operators. The results reported are the average over three runs.



(a) ID ground truth (b) ID DeepFDM (c) ID FNO (d) ID U-Net

(e) OOD ground truth (f) OOD DeepFDM (g) OOD FNO (h) OOD U-Net

Figure 2: Two-dimensional modelled solutions for a diffusion equation for both in distribution (ID) data (top) and out-of-distribution (OOD) data (bottom). All models are visually similar on the in-distribution data. For OOD data, FNO and U-Net lose accuracy, as can be seen from the visualization of the solution.

2 RELATED WORK

2.1 NEURAL PDES

Early machine learning methods (Rudy et al., 2017) focused on discovering the form of a PDE from solution data, without building solution operators. Physics-informed neural networks (PINNs) (Karniadakis et al., 2021; Shin et al., 2020) were among the first models to leverage neural networks to solve PDEs. These methods corresponded to solving a single PDE from incomplete data, rather than learning PDE solution operators from a dataset.

Key Neural PDE Operator learning papers Lu et al. (2019) propose the DeepONet architecture, which learns PDE solution operators. However, in this case, the PDE is fully known and the PDE residual is included in the loss. Subsequently, Li et al. (2020b) proposed a similar approach by learning Green's function for a given PDE. This method gave rise to the Fourier neural operator (Li et al., 2020a), which takes advantage of certain assumptions about Green's function to solve the problem in the Fourier space. Liu et al. (2022) build neural network models that integrate PDE operators directly in the model's architecture while retaining the large capacity neural network architecture.

There are too many recent works on neural PDEs to mention. Zhang (2024) lists over seven hundred articles, most of them from the last three years, including surveys and benchmark articles. We mention a few relevant papers: Takamoto et al. (2022), provide a benchmark dataset and an interface for learning PDEs. Solver in the loop Um et al. (2020) integrate NN methods with a PDE solver. ClimODE (Verma et al., 2024) solves an advection equation with source.

Neural Networks architectures related to differential equations Several works connect neural network architectures and solution operators for differential equations. Chen et al. (2018) proposed a neural network architecture based on ODE solvers and Haber & Ruthotto (2017) focused on the stability aspects of the architecture. Also we have neural SDEs (Tzen & Raginsky, 2019). Ruthotto & Haber (2020) proposed network architectures based on discretized PDE solvers; however, they do not learn PDE operators. Long et al. (2018) represents a PDE solution operator as a feed-forward neural network, and learns both an approximation to the PDE coefficients and the solution operator from the data, however, this early contribution had low accuracy.

2.2 INVERSE PROBLEMS

Neural PDE operators aim to learn to solve a given PDE from data, without assuming that the form of the PDE is known. In contrast, the PDE inverse problem approach assumes that a specific form of the PDE is known, but that the coefficients are unknown. More precisely, inverse PDE problems (Taler & Duda, 2006) aim to infer unknown parameters of a PDE with a known form, using a dataset of PDE solutions. Although the inverse problem approach is compatible, in theory, with any numerical or neural PDE method, the drawback is the specialized nature of each solver, with custom code and custom optimization routines.

Numerical inverse problems We first discuss the approach that uses forward PDE solvers to learn the PDE coefficients. The PDE approach uses numerical PDE solvers, such as finite element methods (FEM) and finite difference methods (FDM) (Larsson & Thomée, 2009). These methods discretize the PDEs on the domain into a system of equations that can be solved numerically. These solvers require knowledge of the full equation governing the process of interest and operate on structured input data, which can limit their applicability and adaptability to a wider range of scientific problems. For inverse problems, numerical PDE solvers are often used in combination with gradient-based optimization techniques. These are implemented in packages such as (COMSOL, 2023; Logg et al., 2012; Virtanen, 2020; Ruthotto et al., 2017). These methods are computationally intensive and often require customized code for each problem formulation. However, when combined with proper regularization and optimization strategies, they provide accurate and reliable solutions.

Neural inverse problems There are a number of works on neural inverse problems. (These works differ somewhat from ours, in that they focus on a single problem at a time, rather than developing a methodology for solving a wide class of inverse problems - they have not applied the same method to a number of benchmark problems). Zhao et al. (2022) solve PDE inverse problems, such as waveform inversion. In this case, the forward solver is given by a graph neural network or by a U-Net. They report faster solution times, compared to using the Finite Element Method for the forward solver. Even using neural networks, their approach has the limitation that changes to the PDEs require training a new forward solver. Huang et al. (2022) treat inverse problems for Darcy and Navier Stokes. Jiao et al. (2024) use DeepONets as a solver in a Bayesian Markov Chain Monte Carlo (MCMC) approach to PDE inverse problems, to learn from noisy solutions of a diffusion equation. Zhang et al. (2024) solves inverse problems using a PINN approach for the forward solver.

A second approach is the Bayesian inference approach to inverse problems (Stuart, 2010): this one is more appropriate for problems with uncertainty in the model and noise in the data, and does not require PDE solvers. Cao et al. (2023) solve Bayesian Inverse Problems and find that using neural networks for the forward solver is faster but less accurate, compared to traditional scientific computing solvers, so they implement a hybrid approach.

3 Neural Operators and PDE inverse problems

In this section, we explain the difference between neural PDE operators and our approach. In this work, we consider a known family of time dependent PDEs, written as

$$\frac{\partial}{\partial t}u(x,t) = P(u(\cdot), a(x)) \tag{PDE}$$

along with initial condition, $u(x,0) = u_0(x)$. We implemented periodic boundary conditions for convenience. Here $P(u(\cdot), a(x))$ is a PDE operator, parameterized by $a(x)$, a vector function, representing the PDE coefficients. We choose the following operator, $P(u(\cdot), a(x))$,

$$\begin{aligned} P(u(\cdot), a(x)) = a_0(x) + a_1(x)u(x,t) + a_2(x)\cdot\nabla u(x,t) + a_3(x)\Delta u(x,t) \\ + a_4(x)u(x,t)(1-(u(x,t)) + a_5(x)u(x,t)\nabla u(x,t) \end{aligned}$$

The PDE above includes many benchmark PDEs, from Li et al. (2020a;b) and Takamoto et al. (2022), as a special case, by setting some coefficients to zero. In particular, it includes each of the following PDEs: advection, diffusion, advection-diffusion, reaction diffusion equations, and the Burgers equation. We focus mainly on the two-dimensional case, but we also include a one-dimensional implementation.

The benchmarks which were excluded were the PDEs which fell outside the class: the two-dimensional Navier-Stokes equations, because it is a vector PDE, and the Darcy equation, because it is not time-dependent. In the case of a numerical solver, we expect to know the type of the data (e.g., time dependent, or time independent, vector, or scalar), so these cases were excluded from the equation.

3.1 Learning PDE solution operators

Given a dataset, $S^m = \{U_1, \ldots, U_m\}$ where each element

$$U_i(X,T) = (U_i(X,0), U_i(X,t_1), \ldots, U_i(X,t_k))$$

consists of a vector of grid values of a PDE, one for each time, $t \in T = (0, t_1, \ldots, t_k)$. The dataset can come from benchmark datasets. Below we show how it can also be generated by a numerical PDE solver solution (2), along with initial conditions sampled from a distribution. In this case, the solution is generated on a higher resolution grid, and then coarsened (upsampled), in order to better approximate the PDE solution.

Learning PDE operator with neural networks The neural PDE solver corresponds to a neural network architecture with weights W. A forward pass (fixed W), maps initial grid data U_0, to a vector of time slices $U(X,T)$.

$$U(X,T) = \text{NNOperator}(U_0; W) \tag{1}$$

The neural network learns the solution operator by fitting the data, using mean squared loss,

$$\min_W \sum_{U_i \in S} \|U_i - \text{NNOperator}(U_i(X,0); W)\|_X^2$$

Once the neural network is trained, the final weights, $\widehat{W}$, lead to the approximate solution operator, $\text{NNOperator}(U_i(X,0); \widehat{W})$. In the results section, we show empirically that neural solvers can be biased.

Learning PDE coefficients with numerical PDE solvers The parametric PDE learning problem (which is a type of PDE inverse problem) corresponds to the following. The input data set is assumed to be a solution of (PDE) with unknown but bounded coefficients. A given benchmark problem would have most of the coefficients set to zero. However, each training run assumes all the coefficients can be nonzero.

The numerical solution operator is written,

$$U(X,T) = \text{NumPDESolve}(U_0; A(X)) \tag{2}$$

Given the dataset S^m, training the inverse problem corresponds to fitting the coefficients to the data.

$$\min_{A\in\mathcal{A}} \sum_{U_i\in S} \|U_i - \text{NumPDESolve}(U_i(X,0); A)\|_X^2 \tag{3}$$

The method (3) is *interpretable by design*: the learned parameters of the model, $\widehat{A}$, correspond to the coefficients of the PDE. It can be interpreted as a vector regression problem for parameter identification. The method (3) is more tractable and easier to analyze than the neural network problem (1). Thus, we expect that overfitting will not be a problem. Moreover, since a forward pass corresponds to an accurate numerical solution of the PDE, we expect the solution to be accurate. However, there will still be errors associated with finite data, so we expect $\widehat{A}$ to approximate, but not be equal, to A^*, the grid values of the coefficients.

In the next section, we have a theorem that characterizes the numerical inverse problem in terms of parametric regression.

3.2 Parametric Regression problem for the numerical PDE solver

For a given grid, X, let $h(X)$ be the grid resolution. Regarding a function on a grid as an approximation, we define $\|U\|_X = h(X)^2\|U\|_2$ (in two dimensions). The scaling factor is a normalization that ensures that constant functions have the same norm, regardless of the grid resolution.

Theorem 3.1. *Let A^* be the grid values of the true PDE parameters, $A^* = a(X)$. The numerical PDE learning problem corresponds to the parametric vector regression learning problem*

$$\min_{A\in\mathcal{A}} \sum_{U_i\in S} \|\textit{NumPDESolve}(U_i(X,0); A) - \textit{NumPDESolve}(U_i(X,0); A^*) - \epsilon_i\|_X^2$$

with noise vector, ϵ_i, whose norm goes to zero with the grid resolution, $\max_i \|\epsilon_i\|_X = \mathcal{O}(h(X))$.

Proof. Given (PDE), write $u(x,t) = \text{PDESolve}(u_0, a(x))$ for the solution of the PDE, with initial data $u_0(x)$.

Define

$$\epsilon_i = \text{NumPDESolve}(U_0; A(X)) - \text{PDESolve}(u_0, a(x))(X)$$

Standard PDE finite difference numerical approximation bounds Larsson & Thomée (2009) can then be expressed as $\|\epsilon_i\|_X^2 = \mathcal{O}(h(X))$, where we assume first order accuracy. The PDE solution, $u(x,t)$, when evaluated on the grid, corresponds to U_i. Thus

$$U_i - \text{NumPDESolve}(U_i(X,0); A^*) = \epsilon_i$$

where ϵ_i represents the numerical solver error, which has a norm on the order of the grid resolution $\|\epsilon_i\|_X = \mathcal{O}(h(X))$, as desired. □

The small amount of noise means there can be a small error in learning the parameters, but still we expect that the model learn a close approximation of the correct parameters, and should generalize. Thus, using standard results about regression, this theorem tells us that we expect a nearly unbiased approximation to the true parameters of the model, with better results as the grid resolution improves. In many cases, for an inverse problem, there is a theory that ensures machine learning *consistency:* with enough data that the solution operator converges to the correct one. With additional assumptions, the coefficients also converge, $\widehat{A} \to A^*$.

4 DeepFDM model design and model architecture

Equation (3) described a general purpose inverse problem solver, where a forward pass corresponds to solving a PDE with fixed coefficients, and where the optimization step corresponds to learning the coefficients. Normally, the inverse problem (3) is implemented using a scientific computing package, along with a user-defined optimization code.

In our case, we build the inverse problem in a neural network architecture, DeepFDM. DeepFDM can be interpreted as neural network architecture, which implements a finite difference method for solving a PDE, as a forward pass.

DeepFDM is implemented as a feedforward convolutional neural network, where each forward pass corresponds to an implementation of a scientific computing solver, $\text{NumPDESolve}(U; A(X))$, of (PDE), where $A(X)$ corresponds to the unknown vector of coefficients. In other words, for a given vector of coefficients, a forward pass is a numerical

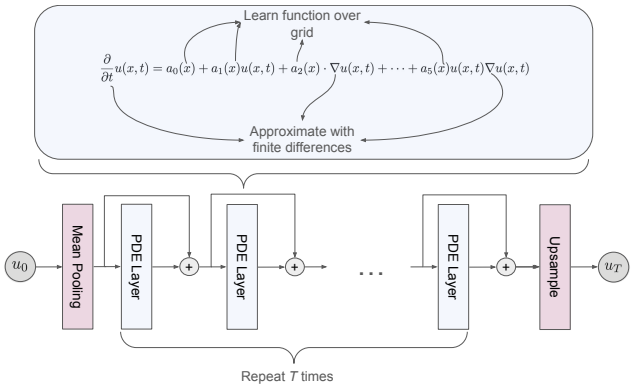


Figure 3: Example network architecture for the PDE solver. The mean pooling layer is used to reduce the resolution of the input and the upsampling layer is used to bring the output back up to size.

solution of the corresponding PDE. The numerical PDE operator is implemented using finite differences, and the PDE is parameterized by the grid values, $A(X)$, of the coefficients $a(x)$. The details of the implementation and the use of finite difference schemes are not required to understand the main work, but are included in Appendix C, for reference.

Using a neural network architecture is very convenient, since we can take advantage of built-in optimization routines, rather than implementing optimization as is more typical with inverse problems. Moreover, a forward pass is very computationally efficient since we are working with a deep but small convolutional neural network. Training is also faster than for benchmark neural solvers; see the results section below.

The model architecture is an implementation of the finite difference solver for (PDE). In this case, the finite difference operators are implemented as convolution with fixed (predefined, non-learnable) operators. The coefficients correspond to model weights passed through a sigmoid nonlinearity (to make them bounded). This allows the finite difference solver to be implemented as a differentiable model and trained using standard SGD implementations (see the results section below). The architecture is illustrated in Figure 3, and more details of the architecture can be found in Appendix A.

5 Dataset generation and OOD quantification

In this section, we describe the procedure used to generate the synthetic data we use for testing. We also explain the definition of an out-of-distribution (OOD) shift we consider in this paper. To the best of our knowledge, no other work employs the OOD quantification scheme used in our paper, making it a novel contribution.

Data generation process To characterize and benchmark DeepFDM against existing architectures, we train on synthetic data generated by PDE solvers.

1. Sample some Fourier coefficients $c \sim \mathcal{N}(0, \Sigma)$ from a Fourier spectrum with at most N modes and compute the resulting function, U_0, with coefficients multiplied by the Fourier basis functions.
2. Use a standard scientific computing solver to compute the solution to the PDE problem with initial condition U_0 for the required number of time steps.

Generating initial conditions The initial conditions are generated as follows. Let $\mathcal{N}(0, \Sigma)$ be a mean zero normal distribution with diagonal covariance matrix, Σ. Sample a coefficient vector, $c_i \sim \mathcal{N}(0, \Sigma_{ii})$. Let $\Phi(x) = (\phi_1(x), \ldots \phi_N(x))$ be an orthonormal family of functions defined on the grid. We used a Fourier basis, sines and cosines in one dimension, and products of sines and cosines in two dimensions. Then for each sample of coefficients, c, set

$$u(x) = c \cdot \Phi(x) = \sum_{i=1}^{N} c_i \phi_i(x) \tag{4}$$

This generates samples of function $u(x)$ defined on the grid. Since the basis functions are orthonormal, the covariance of the functions (using the standard L_2 inner product) is also given by Σ.

To generate OOD data, we employ the same procedure, using a different distribution $\tilde{\rho} = \mathcal{N}(0, \tilde{\Sigma})$.

Measuring dataset shift We generate functions, using the orthonormal basis Φ and coefficients $c \sim N(0, \Sigma)$. Using a different $\tilde{\Sigma}$, keeping the same basis functions Φ, we can measure the distance between the two distributions. A convenient choice of distance is given by the Hellinger distance (Cramér, 1999), chosen because it has a formula for the distance between two multivariate normal distributions. The Hellinger distance between two multivariate mean zero normal distributions is given by

$$H^2(\mathcal{N}(0,\Sigma),\mathcal{N}(0,\tilde{\Sigma})) = 1 - \frac{\det(\Sigma)^{\frac{1}{4}}\det(\tilde{\Sigma})^{\frac{1}{4}}}{\det\left((\Sigma+\tilde{\Sigma})/2\right)^{\frac{1}{2}}}.$$

Figure 6 shows examples of initial conditions generated using different Fourier spectra and gives the Hellinger distances between the distributions.

Generating accurate Numerical PDE solutions To generate PDE solutions from the initial data, we implemented a finite difference solver (Larsson & Thomée, 2009). The PDE solver uses the forward Euler method, with a small time step, calculated from the coefficients and spatial resolution, to ensure stability and convergence. This solver can be used to generate high-resolution PDE solutions, with a given distribution of coefficients. High-resolution PDE solutions are then projected onto a coarser grid, which gives an accurate approximation to the exact solution of the PDEs. Projecting the finer grid solutions avoids biasing the solution towards the particular choice of numerical solver.

6 Results

In this section, we compared the performance of our proposed benchmark method, DeepFDM, again the state-of-the-art neural operator methods, with the primary goal of comparing the accuracy for data generated both in and out of distribution (OOD). We also report training times and show how DeepFDM is interpretable. Finally, we illustrate how the benchmark method can be used to study the bias of neural solvers.

As described above, the training procedure was the same for the neural operators and for DeepFDM, since DeepFDM is implemented as a neural network. However, since DeepFDM also corresponds to an inverse problem method (3), for (PDE), we expect that it will have higher accuracy and better generalization performance. Moreover, since it has many fewer parameters and a simpler architecture, we expect that it will train faster.

To measure accuracy, we used the normalized mean squared error of the predicted solutions. The normalization factor is designed to set the variance of the initial data (as a function of x) to one and allows for a fair comparison between different distributions, which may have different coefficient norms.

For DeepFDM, we used the model architecture described in section 4, designed so that a forward pass corresponds to a numerical solution of (PDE) with all coefficients allowed to be non-zero. The coefficients were assumed to have magnitude at most 2.5. (Making the bounds on the coefficients larger did not have a significant effect since in either case, the model achieved high training accuracy).

For in-distribution data, we used the available solution dataset, obtained from (Liu et al., 2022). To measure the performance of the models on data from a different distribution, we used the method described in section 5 to generate synthetic PDE solution data, sampled from different distributions.

Considered benchmarks **U-Net:** We use a U-Net architecture, popular for image-to-image tasks, such as segmentation. We consider a 2D U-Net with 4 blocks (Ronneberger et al., 2015). **ResNet**: We use an 18 block ResNet with residual connections (He et al., 2016). **FNO:** We use an FNO with 12 modes for all channels and all experiments (Li et al., 2020a). Our results for the ResNet and U-Net matched the results reported for these models in Li et al. (2020a).

Training dynamics All models are trained using the MSE loss function. All models were trained with the Adam optimizer without weight decay. The training, validation and test data samples were split as 75%, 12.5%, and 12.5%, respectively. All models were run on a Tesla T4, GPU with a batch size of 32. DeepFDM trained faster, and with a smaller generalization gap than the other methods. See Figure 5.

Parameter count The number of parameters in DeepFDM is on the order of the number of grid points (spatial data points) as shown in Table 2, which is hundreds of times less than FNO and U-Net.

Grid resolution	4,096	1,024	256	64
Parameters in DeepFDM	20,484	5,124	1,284	324
Parameters in FNO	184,666	184,666	184,666	184,666
Parameters in U-Net	7,762,762	7,762,762	7,762,762	7,762,762
Parameters in Res-Net	3,960	3,960	3,960	3,960

Table 2: Model parameters for DeepFDM and the benchmark models tested.

Model accuracy We see from Table 1 that, as expected, DeepFDM is more accurate by a factor of 10 in all equations considered. Furthermore, as expected, DeepFDM is more accurate in OOD performance; the solutions given by DeepFDM are visually accurate and have lower errors than the other benchmarked models. Figure 2 shows example modeled solutions in two dimensions. We note that in all figures we exclude ResNet, since errors were higher than 60%.

Out-of-distribution generalization To quantify the generalizability of DeepFDM to distinct data distributions, we tested several distributions, each one further apart from the training distribution. Figure 4a shows the relative error of the models tested as a function of the Hellinger distance between the training and the test distribution. We can see that as the test distribution is further apart from the training distribution, all models start losing accuracy but DeepFDM still achieves under 1% relative error while both U-Net and FNO approach errors of 10% for the furthest distributions. We tested this for all equations shown in Table 1.

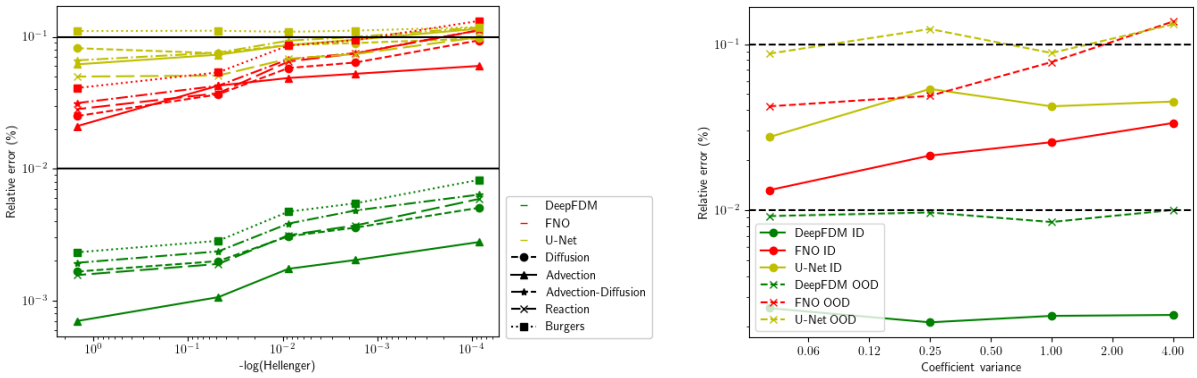
Interpretable models: learning coefficients In most cases, DeepFDM successfully learns a set of parameters that match the ground-truth process. Figure 1, shows the case of a diffusion equation, where the coefficients are recovered from the model with high accuracy.

Coefficient variance explains FNO errors In Figure 4b, we report the relative error for different coefficient values. The variance of the coefficient on the x-axis corresponds to the amplitudes of the sine waves used to generate the coefficients (larger amplitude correlating to larger variance). We note that both FNO and U-Net see a degradation in performance as the coefficient variance increases, while DeepFDM has nearly constant performance. By design, DeepFDM is able to learn variable-coefficient PDEs accurately.

For the FNO, this can be explained by one of the underlying hypotheses of their model architecture; in order to perform computation in Fourier space, the authors make the assumption that the Green's function they learn is *translation invariant*. Since variable coefficients are not translation-invariant, as the variance of the coefficients grows, this hypothesis becomes less valid. Thus, we illustrated the bias of FNO towards translation invariant solutions.

7 Conclusion

In this work, we introduced DeepFDM, a benchmark framework for comparing neural Partial Differential Equation (PDE) operators with traditional numerical solvers. Although DeepFDM is not intended as a replacement for neural PDE solvers, it takes advantage of the inherent structure of PDEs to offer improved accuracy, generalization, and interpretability, particularly in out-of-distribution (OOD) scenarios. Furthermore, we proposed a method for generating and quantifying distribution shifts using the Hellinger distance, enabling robust performance evaluation across diverse PDE problems. Our results show that DeepFDM consistently outperforms neural operator methods when applied to learning PDEs from the class it was designed for. This makes it a valuable tool for benchmarking and advancing neural PDE operator research.



(a) Relative error of different models (y-axis) in terms of the Hellinger distance between the training and test distributions (x-axis). DeepFDM is the most accurate, achieving under 1% relative error even under the largest dataset shift. On the other hand, FNO and U-Net significantly decrease their performance on distinct distributions, with relative errors approaching 10%.

(b) Relative error as a function of coefficient variance. The coefficient variance corresponds to the amplitude of the sine waves used to generate the coefficients (bigger variance means bigger amplitude). Our model shows constant error across coefficient size while FNO and U-Net see a performance drop as variance increases.

Figure 4: Comparison of relative errors of different models under the shift of the data set (left) and coefficient variance (right).

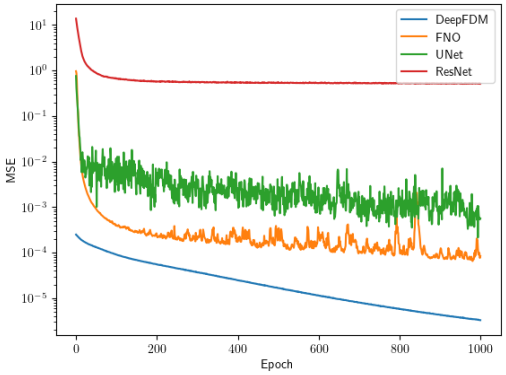


Figure 5: Training dynamics of the models tested. With fewer parameters, DeepFDM trains to an error of 10^{-3} in just epoch, and to an error of 10^{-4} in less than 100 epochs, compared to FNO and U-Net which take longer. The ResNet model trains more slowly and has a higher loss.

REFERENCES

L. Cao, T. O'Leary-Roseberry, P.K. Jha, and J.T. Oden. Residual-based error correction for neural operator accelerated infinite-dimensional bayesian inverse problems. *Journal of Computational Physics*, 2023. URL https://www.sciencedirect.com/science/article/pii/S0021999123001997.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

COMSOL. *COMSOL Multiphysics® v. 6.1*. COMSOL AB, Stockholm, Sweden, 2023. URL https://www.comsol.com. Accessed: 2024-09-26.

Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische annalen*, 100(1):32–74, 1928.

Richard Courant, Kurt Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM journal of Research and Development*, 11(2):215–234, 1967.

Harald Cramér. *Mathematical Methods of Statistics*, volume 26. Princeton University Press, 1999.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1):014004, dec 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Daniel Zhengyu Huang, Jiaoyang Huang, Sebastian Reich, and Andrew M Stuart. Efficient derivative-free bayesian inference for large-scale inverse problems. *Inverse Problems*, 38(12):125006, 2022.

A. Jiao, Q. Yan, J. Harlim, and L. Lu. Solving forward and inverse pde problems on unknown manifolds via physics-informed neural operators. *arXiv preprint arXiv:2407.05477*, 2024. URL https://arxiv.org/abs/2407.05477.

George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, may 2021.

Stig Larsson and Vidar Thomée. *Partial Differential Equations With Numerical Methods*, volume 45. Springer, Chalmers University of Technology and University of Gothenburg 412 96 Göteborg Sweden, 2009.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 10 2020a.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*, 03 2020b.

Xin-Yang Liu, Hao Sun, Min Zhu, Lu Lu, and Jian-Xun Wang. Predicting parametric spatiotemporal dynamics by multi-resolution pde structure-preserved deep learning. *arXiv preprint arXiv:2205.03990*, 2022.

Anders Logg, Kent-Andre Mardal, Garth Wells, et al. Automated solution of differential equations by the finite element method: The fenics book. *Lecture Notes in Computational Science and Engineering*, 84, 2012. doi: 10.1007/978-3-642-23099-8. URL https://fenicsproject.org/. Accessed: 2024-09-26.

Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. Pde-net: Learning pdes from data. In *International conference on machine learning*, pp. 3208–3216. PMLR, 2018.

Lu Lu, Pengzhan Jin, and George Em Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.

Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM review*, 63(1):208–228, 2021.

Adam M Oberman. Convergent difference schemes for degenerate elliptic and parabolic equations: Hamilton–jacobi equations and free boundary problems. *SIAM Journal on Numerical Analysis*, 44(2):879–895, 2006.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pp. 234–241. Springer, 2015.

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.

Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, 62(3):352–364, Apr 2020.

Lars Ruthotto, Eran Treister, and Eldad Haber. jinv–a flexible julia package for pde parameter estimation. *SIAM Journal on Scientific Computing*, 39(5):S702–S722, 2017. doi: 10.1137/16M1081063. URL https://doi.org/10.1137/16M1081063.

Yeonjong Shin, Jerome Darbon, and George Em Karniadakis. On the convergence and generalization of physics informed neural networks. *arXiv e-prints*, pp. arXiv–2004, 2020.

Andrew M Stuart. Inverse problems: a bayesian perspective. *Acta numerica*, 19:451–559, 2010.

Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Daniel MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. Pdebench: An extensive benchmark for scientific machine learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/0a9747136d411fb83f0cf81820d44afb-Abstract-Datasets_and_Benchmarks.html.

Jan Taler and Piotr Duda. *Solving direct and inverse heat conduction problems*. Springer, 2006.

Belinda Tzen and Maxim Raginsky. Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit. *arXiv preprint arXiv:1905.09883*, 2019.

Kiwon Um, Robert Brand, Yun (Raymond) Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/43e4e6a6f341e00671e123714de019a8-Abstract.html.

Yogesh Verma, Markus Heinonen, and Vikas Garg. Climode: Climate and weather forecasting with physics-informed neural odes. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=xuY33XhEGR.

Pauli et al Virtanen. Scipy 1.0: Fundamental algorithms for scientific computing in python, Feb 2020. URL https://doi.org/10.1038/s41592-019-0686-2.

Chengyang Zhang. Neural-pde-solver. https://github.com/bitzhangcy/Neural-PDE-Solver, 2024. Version 1.0.

R.Z. Zhang, X. Xie, and J. Lowengrub. Bilo: Bilevel local operator learning for pde inverse problems. *arXiv preprint arXiv:2404.17789*, 2024. URL https://arxiv.org/abs/2404.17789.

Qingqing Zhao, David B. Lindell, and Gordon Wetzstein. Learning to solve pde-constrained inverse problems with graph networks. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (eds.), *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pp. 26895–26910. PMLR, 2022. URL https://proceedings.mlr.press/v162/zhao22d.html.

A FULL MODEL ARCHITECTURE

DeepFDM is formulated to take some initial condition U_0 (defined as a function on a grid) and iterate in time for some given number of time steps T. For $k = 0, 1, \ldots, T-1$, the iterative update is defined as

$$U_{k+1} := U_k + c_t \left(\sum_{i=1}^{n_{\text{lin}}} \sigma(\theta_i^{\text{lin}}) \odot (\text{conv}(W_i, U_k)) + \sum_{j=1}^{n_{\text{quad}}} N_j(\theta_j^{\text{quad}}, U_k, U_{k-1}) \right)$$

where $\odot$ represents the componentwise product. Here c_t is a constant representing the time-step interval, and each W_i is a predetermined (nontrainable) convolution kernel corresponding to a finite difference operator. The componentwise $\sigma(\cdot)$ is the sigmoid function. $n_{\text{lin}} = 4$ and $n_{\text{quad}} = 2$ represent the number of linear and nonlinear terms, respectively. θ_j^{lin} and θ_j^{quad} are the corresponding parameters.

The linear terms corresponds to standard upwind finite difference discretizations of the derivatives, pointwise multiplied by nonlinearly scaled coefficient terms.

The expressions N_j represent nonlinear terms. For the reaction operator, for example, this corresponds to a quadratic reaction term and takes the form

$$N_1(\theta_1^{\text{quad}}, U) = \sigma(\theta_1^{\text{quad}}) U \odot (1 - U),$$

The nonlinear term corresponding to a non-constant coefficient Burgers operator takes the form

$$N_2(\theta_2^{\text{quad}}, U_k, U_{k-1}) = \sigma(\theta_2^{\text{quad}}) \odot U_{k-1} \left(\text{conv}(W_{adv}, U_k)\right)$$

where the first term corresponds to a linear advection term, multiplied by U_{k-1} (we used the previous time step value for stability).

More PDE terms can be added by widening the network with the corresponding discretization.

By bounding the coefficients with a sigmoid function scaled by the time step interval, our model corresponds by design, for fixed parameter values θ, to a stable finite difference method consistent with a PDE with coefficients given by the model parameters.

Each layer is repeated T times, and corresponds to the Forward Euler method, as can be seen in Figure 3.

B VISUALIZATION OF OOD SAMPLES

C TRADITIONAL NUMERICAL PDE SOLVER BACKGROUND: FINITE DIFFERENCE METHODS

In this section, we demonstrate the finite difference operator in a simple case and give an idea of how to build learnable finite difference operators in the general case.

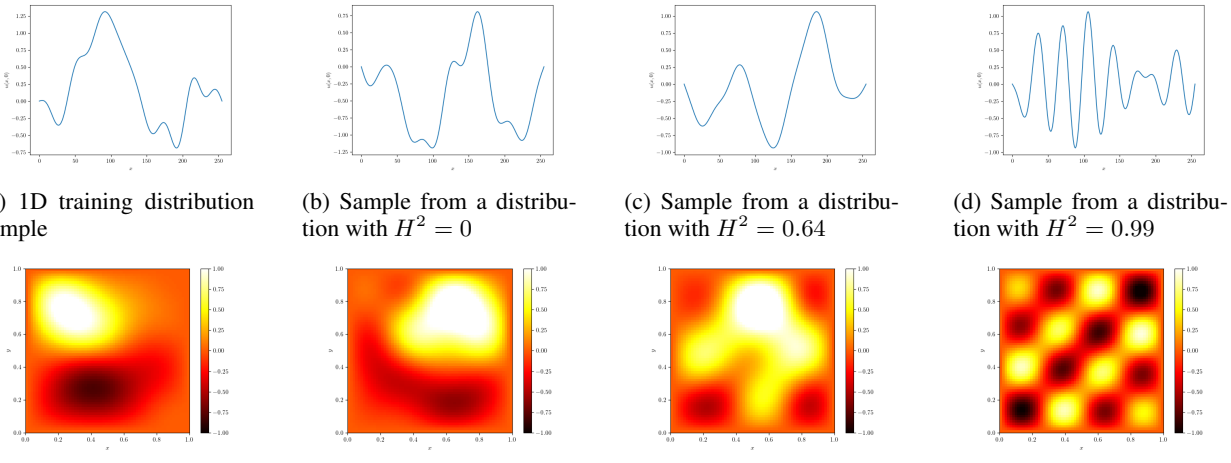
A fundamental result in numerical approximation of linear PDEs (Courant et al., 1928; Larsson & Thomée, 2009) provides conditions on the time and grid discretization parameters, c_t, c_x, in terms of bounds on the coefficients $a(\boldsymbol{x})$ which ensure that the method is numerically stable, and convergent. Oberman (2006), extended the family of stable finite difference operators to a wide class of diffusion-dominated PDEs. The convergence theory states that as the resolution of the data increases, the solution operator converges to the PDE solution operator. $\lim_{\epsilon \to 0} \|h^\epsilon - h^*\| = 0$, in the appropriate operator norm.

C.1 FINITE DIFFERENCE HEAT EQUATION SOLVER

An intuitive way to approximate a derivative is by a finite difference. For example, for $\boldsymbol{x}$ in one dimension,

$$u_{\boldsymbol{x}}(\boldsymbol{x}) \approx \frac{u(\boldsymbol{x} + \epsilon) - u(\boldsymbol{x})}{\epsilon}.$$

A more careful analysis shows that the second derivative operator, $u_{\boldsymbol{xx}}$, is approximated by the finite difference $u_{\boldsymbol{xx}}(\boldsymbol{x}) \approx \frac{u(\boldsymbol{x}+\epsilon) - 2u(\boldsymbol{x}) + u(\boldsymbol{x}-\epsilon)}{\epsilon^2}$.



(a) 1D training distribution sample

(b) Sample from a distribution with $H^2 = 0$

(c) Sample from a distribution with $H^2 = 0.64$

(d) Sample from a distribution with $H^2 = 0.99$

(e) 2D training distribution sample

(f) Sample from a distribution with $H^2 = 0$

(g) Sample from a distribution with $H^2 = 0.64$

(h) Sample from a distribution with $H^2 = 0.99$

Figure 6: Examples of functions, $u(x)$, randomly samples from the distribution according to (4). By changing the distribution of coefficients, we obtain different functions with a different Fourier spectrum, visible from the scale of oscillations in the functions. The first two columns are sampled from the same distribution. The second two are from different distributions. The last column has the finest scale oscillations. Top: $x - y$ plot of functions of one variable. Bottom: color map plots of functions of two variables.

C.2 Finite difference operators on grids

A finite difference operator on a grid is an approximation of a differential operator. The finite difference operator corresponding to $u_{\boldsymbol{x}}$ on a uniform grid is represented by a convolution operator, with kernel $W = \frac{1}{c_x}[-1, 1]$, where we have replaced ϵ with the grid spacing parameter, c_x. We have a similar operator in two dimensions.

Finite difference approximations of the Laplacian in one dimension, in two dimensions, correspond, respectively, to convolutions with the following kernels

$$W_{Lap,1} = \frac{1}{c_x^2}[1, -2, 1], \qquad W_{Lap,2} = \frac{1}{c_x^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

These operators are linearly combined to approximate each of the linear terms, $L(u, a)$, in the linear part of the PDE.

To approximate nonlinear terms, we use upwind nonlinear finite difference opertators, Oberman (2006), which showed that it is possible to build *numerically stable* finite difference approximations for a wide class of nonlinear elliptic and parabolic PDEs. (For example, a stable approximation of the the eikonal operator, $|u_x|$, is given by the maximum of the upwind finite difference schemes for u_x and $-u_x$, respectively.)

C.3 Stable discretization

Each layer of the operator corresponds to a discretization of a PDE. We need this discretization to be convergent, which puts requirements on the hyperparameters in the model, and how they relate to the possible coefficients. Here we discuss the special case of the heat equation, for clarity of exposition.

When solving any PDE numerically, we are bound by some stability constraints that are necessary for obtaining a convergent solution. For the heat equation, assuming we take space intervals of c_x (and equal in all dimensions) and time intervals of c_t, we are bound by the stability constraint $0 \leq a(\boldsymbol{x}) \cdot \frac{c_t}{c_x^2} \leq \frac{1}{2 \cdot D}$ where D is the dimension of the data, (Courant et al., 1967). Thus when one knows the coefficients $a(\boldsymbol{x})$ then one can simply pick c_t and c_x to satisfy the stability constraint.

In this case, we take the opposite approach. Given fixed values of c_x and c_t, we can bound the coefficients themselves by

$$0 \leq a(\boldsymbol{x}) \leq C_a = \frac{c_x^2}{2D \cdot c_t}$$

This is a crucial constraint since the parameters of the model will take the place of the coefficients of the equation being modelled. In this way, we design DeepFDM precisely with the aim of learning the physical process that is trying to approximate.

In order to satisfy the stability constraint, we bound the raw parameters learned by the model with a scaled sigmoid function. This is, if the model's parameters are θ, then the values that we multiply with the convolution layer (corresponding to the Laplace operator) are given by $C_a \cdot \sigma(\theta)$. This ensures that the parameters are bounded by the stability region of the PDE and thus forces DeepFDM to find a solution in the parameter space in which the PDE itself is stable.