GENERATING SYNTHETIC DATA FOR NEURAL OPERATORS

Erisa Hasani *

Department of Mathematics University of Texas at Austin ehasani@utexas.edu

Rachel A. Ward

Department of Mathematics University of Texas at Austin Microsoft Research rward@math.utexas.edu

ABSTRACT

Numerous developments in the recent literature show the promising potential of deep learning in obtaining numerical solutions to partial differential equations (PDEs) beyond the reach of current numerical solvers. However, data-driven neural operators all suffer from a similar problem: the data needed to train a network depends on classical numerical solvers such as finite difference or finite element, among others. In this paper, we propose a different approach to generating synthetic functional training data that does not require solving a PDE numerically. We draw a large number N of independent and identically distributed 'random functions' u_j from the underlying solution space (e.g., $H_0^1(\Omega)$) in which we know the solution lies according to classical theory. We then plug each such random candidate solution into the equation and get a corresponding right-hand side function f_j for the equation, and consider $(f_j, u_j)_{j=1}^N$ as supervised training data for learning the underlying inverse problem $f \to u$. This 'backwards' approach to generating training data only requires derivative computations, in contrast to standard 'forward' approaches, which require a numerical PDE solver, enabling us to generate many data points quickly and efficiently. While the idea is simple, we hope this method will expand the potential for developing neural PDE solvers that do not depend on classical numerical solvers.

Keywords Synthetic data · Numerical PDEs · Neural operators

1 Introduction

The use of deep learning to obtain numerical solutions to PDE problems beyond the reach of classical solvers shows promise in revolutionizing science and technology. Deep learning-based methods have overcome many challenges that classical numerical methods suffer from, among which are the curse of dimensionality and grid dependence.

Methods that attempt to solve PDE problems using deep learning can be split into two main classes: those that solve an instance of a PDE problem by directly approximating the solution (e.g. (6), (27), (21), (35), (34), (3)), and those that consider solutions to a family of PDE problems, also known in the literature as parametric PDEs, through operator learning. In the operator learning approach, the goal is to learn an operator that maps the known parameters to the unknown solution (e.g. (17), (2), (13), (10), (22)). In this paper, we focus on the second class, where we seek solutions to a class of PDE problems instead of an instance. Although the approach we describe is general, we focus on the Fourier Neural Operator (FNO) (10), which is a state-of-the-art neural operator learning method at the time that this paper is being written. We stress that our method is independent of the particular neural operator learning architecture and should remain applicable as a synthetic data generation plug-in as the state-of-the-art architecture evolves.

To the best of our knowledge, classical numerical methods, such as finite differences, finite element (29), pseudo-spectral methods, or other variants, have been used to obtain data for training purposes in operator learning. In particular, some works have used finite difference schemes (e.g. (17), (2), (14), (28), (20), (25), (22), (18)). In other works, data has been generated by constructing examples that have a closed-form explicit solution or by using schemes such as finite element, pseudo-spectral schemes, fourth-order Runge-Kutta, forward Euler, etc. (e.g. (16), (32), (8), (24), (12), (15),

^{*}Corresponding author

(26), (30), (4), (31), (19), (33)). While these works are a strong proof-of-concept for neural operators, it is critical to move away from using classical numerical solvers to generate training data for neural operator learning if we want to develop neural operators as a general-purpose PDE solver *beyond* the reach of classical numerical solvers.

Our approach. Our approach is conceptually simple: suppose we want to train a neural network to learn solutions to a parameterized class of PDE problems of the form (1). If we know that the solution for any value of the parameter belongs to a Sobolev space which has an explicit orthonormal basis of eigenfunctions and associated eigenvalues, we can generate a large number of synthetic training functions $\{u_{a_j}^k\}_{j,k}$ in the space as random linear combinations of the first M eigenfunctions, scaled by the corresponding eigenvalues (see Section 3 for more details). We can efficiently generate corresponding right-hand side functions $f_{a_j}^k$ by computing derivatives, $-L_{a_j}u_{a_j}^k=f_{a_j}^k$. We then use training data $(f_{a_i}^k, a_j, u_{a_i}^k)_{j=1}^N$ to train a neural operator to learn the class of PDEs.

The concept of first generating the 'unknown' function and then plugging this function into an equation is not new; one previous manifestation of this concept appeared under the name of *the method of manufactured solutions* (see e.g (23)). This concept has been widely used for code verification when developing numerical solvers, where after constructing an exact solution, one can compare how the numerical solution compares to the exact solution. The novelty in our work is to combine this simple idea with classical PDE theory in order to randomly draw unknown functions that generalize well in the context of operator learning.

Recall the standard supervised learning setting where the training data are input-output pairs (x_j,y_j) , where the input vectors x_j are independent and identical draws from an underlying distribution \mathcal{D} , and $y_j = \mathcal{G}(x_j)$, and the goal is to derive an approximation $\tilde{\mathcal{G}}$ with minimal test error $\mathbb{E}_{x \sim \mathcal{D}} |\tilde{\mathcal{G}}(x) - \mathcal{G}(x)|$. In our setting, the function to learn is the operator $\mathcal{G}: (a,f) \to u$. Our method of generating $(a_j,f_{a_j}^k)$ and our overall approach can be viewed as a best attempt within the operator learning framework to replicate training data within the classical supervised learning setting.

Organization of the paper. This paper is organized as follows: in Section we introduce the main idea in more detail, in Section 3 we discuss how to determine a space for the unknown functions depending on the problem, and in Section 4 we present some numerical experiments using our data in a known network architecture such as the Fourier Neural Operator (10) (FNO). The types of PDE problems we consider are elliptic linear and semi-linear second-order equations with Dirichlet and Neumann boundary conditions, starting with the Poisson equation as a first example and then considering more complicated equations. At the end of this paper, we include an appendix section with a description of the mathematical symbols used in this paper. Our data generation code can be found on GitHub under the repository name synthetic-data-for-neural-operators.

2 Set-up and Main Approach

Consider a class of PDE problems of the form

$$\begin{cases}
-L_a u = f & \text{in } \Omega \\
B(u) = 0 & \text{on } \partial\Omega,
\end{cases}$$
(1)

where $L=L_a$ denotes a differential operator parameterized by $a\in\mathcal{A},\,\Omega\subset\mathbb{R}^n$ is a given bounded domain, and B(u) denotes a given boundary condition. The goal is to find a solution u that solves (1) given L_a and f. So in a general setting, we wish to learn an operator of the form

$$\mathcal{G}: \mathcal{A} \times \mathcal{F} \longrightarrow \mathcal{U}$$

 $(a, f) \longmapsto u,$

where A, F and U are function spaces that depend on the specifics of the PDE problem.

For example, if we take $Lu = \Delta u$, and B(u) = u then (1) becomes the Poisson equation with zero Dirichlet boundary condition. In that case, we can take $\mathcal{F} = L^2(\Omega)$ and $\mathcal{U} = H_0^1(\Omega)$ and the operator we wish to learn is of the form

$$\begin{aligned} \mathcal{G}: L^2(\Omega) &\longrightarrow H^1_0(\Omega) \\ f &\longmapsto u, \end{aligned}$$

So instead of first fixing a function f and then solving (1) to obtain u to be used as input-output pairs (f, u), we instead generate u first, plug it into (1), and compute f by the specified rule.

The main innovation of our work is in determining the appropriate class of functions for the unknown function u in (1). While from the PDE theory we know that u lives in some Sobolev space (see e.g (7)) in the case of elliptic PDEs, such

space is infinite-dimensional and it is unclear at first how to generate functions that serve as good representatives of the full infinite-dimensional space. We propose to generate functions as random linear combinations of basis functions of the corresponding Sobolev space. In the case where we know from theory that the underlying Sobolev space is $H_0^1(\Omega)$ or $H^1(\Omega)$, then we can obtain explicit basis elements that can be obtained by the eigenfunctions of the Laplace operator with Dirichlet and Neumann boundary conditions, respectively.

3 Drawing synthetic representative functions from a Sobolev space

In this section, we discuss how to draw representative functions from the solution space in the case of elliptic problems so that they generalize well when used in numerical experiments. See the appendix for the definitions of the function spaces used in this section.

Let $\Omega \subset \mathbb{R}^n$ be a bounded open set. Consider the following eigenvalue problem

$$\begin{cases} -\Delta u = \lambda u & \text{in } \Omega \\ B(u) = 0 & \text{on } \partial \Omega, \end{cases} \tag{2}$$

which is called the Laplace-Dirichlet operator when B(u)=u. We say that $\lambda\in\mathbb{R}$ is an eigenvalue to the Laplace-Dirichlet operator if there exists $u\in H^1_0(\Omega)$ with $u\neq 0$ such that

$$\int_{\Omega}\nabla u(x)\nabla\varphi(x)dx=\lambda\int_{\Omega}u(x)\varphi(x)dx, \text{ for all }\varphi\in H^1_0(\Omega)$$

If such $u \neq 0$, we say that it is an eigenvector associated to the eigenvalue λ . The following theorem is well known in the analysis of PDEs and spectral theory (see Chapter 8 of (1)).

Theorem 1. The Laplace-Dirichlet operator has countably many eigenvalues $0 < \lambda_1 \le \lambda_2 \le \cdots \lambda_N \le \cdots$. There exists an orthonormal basis $(e_i)_{i=0}^{\infty}$ of $L^2(\Omega)$ such that e_i is an eigenvector of the Laplace-Dirichlet operator, i.e of problem (2), corresponding to the eigenvalue λ_i for each $i \in \mathbb{N}$. Moreover, $(e_i/\sqrt{\lambda_i})_{i=0}^{\infty}$ is an orthonormal basis of $H_0^1(\Omega)$ equipped with the scalar product $\langle u, \varphi \rangle = \int_{\Omega} \nabla u \cdot \nabla \varphi$.

This theory extends to more general Hilbert spaces, including different elliptic linear operators or different types of boundary value conditions such as Neumann or mixed (e.g. see Theorem 6.6.1 in (1)).

For Neumann boundary conditions, we have $B(u)=\partial u\cdot \nu$ where ν denotes the exterior unit normal vector to the boundary $\partial\Omega$ in problem (2), then we have a similar theorem for the Hilbert space $V=\{v\in H^1(\Omega): \int_\Omega v(x)dx=0\}$, where we can obtain an orthogonal basis for the functional space V. Notice that V here is essentially $H^1(\Omega)$ but functions that differ by adding or subtracting a constant are considered the same.

3.1 Representative functions in rectangular domains

Eigenvectors of the Laplace operator are known for the Dirichlet, Neumann, and Robin boundary conditions on rectangular domains of the form $(a_1,b_1)\times (a_2,b_2)\times \cdots \times (a_n,b_n)\subset \mathbb{R}^n$. They are also known for some non-rectangular domains, see 4.4 for an example on a triangular domain. To keep the presentation simple, we will mainly consider Dirichlet (B(u):=0) and Neumann $(B(u):=\nabla u\cdot \nu)$ boundary conditions on $\Omega:=(0,1)^2$.

For the Dirichlet case, the eigenvectors e_{ij} corresponding to the eigenvalues λ_{ij} of problem (2) are given by

$$e_{ij}(x,y) = \sin(i\pi x)\sin(j\pi y), \quad \lambda_{ij} = (i\pi)^2 + (j\pi)^2, \quad (x,y) \in (0,1)^2, i,j \in \mathbb{N}.$$
 (3)

For the Neumann case, they are given by

$$e_{ij}(x,y) = \cos(i\pi x)\cos(j\pi y), \quad \lambda_{ij} = (i\pi)^2 + (j\pi)^2, \quad (x,y) \in (0,1)^2, i,j \in \mathbb{N}.$$
 (4)

Further, normalizing appropriately, we define the following basis elements for $H_0^1(\Omega)$ and V, respectively

$$u_{ij}(x,y) := \frac{\sin(i\pi x)\sin(j\pi y)}{\sqrt{(i\pi)^2 + (j\pi)^2}}, \quad v_{ij}(x,y) = \frac{\cos(i\pi x)\cos(j\pi y)}{\sqrt{(i\pi)^2 + (j\pi)^2}}$$
 (5)

Finally, we generate the unknown functions u (which we assume are from $H^1_0(\Omega)$ or V) as truncated sums of random linear combinations basis functions with prescribed decay in the coefficients. More precisely, let M,K denote positive truncation numbers and let $a_{ij},b_{ij}\sim N(0,1/\sqrt{i^2+j^2})$ generate $u\in H^1_0(\Omega)$ and $v\in V$ as follows

$$u(x,y) = \sum_{i,j=1}^{M} a_{ij} u_{ij}(x,y), \quad v(x,y) = \sum_{i,j=1}^{K} b_{ij} v_{ij}(x,y)$$
(6)

Notice that by construction, functions of the form (6) satisfy zero Dirichlet and zero Neumann boundary conditions, respectively. In experiments, we draw M and K randomly in $\{1, 2, \cdots, 20\}$, that is to say we use up to the first 20 basis functions. While these spaces are infinite-dimensional and thus require an infinite number of basis functions, we observe that using only the first 20 is sufficient to achieve good generalizations to unseen non-trigonometric f functions. In our implementation of this method, it takes about one minute to generate f000 training data points.

3.2 Representative functions in non-rectangular domains

In our experiments we mainly focus on square domains; however, the eigenvectors and eigenvalues of the Laplacian are also known explicitly for certain specific non-rectangular domains. They are known for disks, circular annuli, spheres and spherical shells which can generally be described as $\Omega = \{x \in \mathbb{R}^n : r < |x| < R\}$, with n = 2, 3, as well as for ellipses and elliptical annuli. In addition, they are also known for equilateral triangles, that is when $\Omega = \{(x,y) \in \mathbb{R}^2 : 0 < x < 1, 0 < y < \sqrt{3}x, y < \sqrt{3}(1-x)\}$. For more details on eigenvectors of the Laplacian, see (9).

As for domains that are not of the above type, there could be ways to obtain the eigenvectors of the Laplacian numerically, however, in that case, we cannot easily take derivatives symbolically which is the main reason our method is computationally efficient. A potential way to generalize to any domain shape could be by passing the boundary values as an input during the training phase and ask for the right boundary condition after the training is finished to get a prediction.

4 Numerical Experiments using the Fourier Neural Operator

The architecture we use for numerical experiments is the Fourier Neural Operator (FNO) introduced in (10), which can learn mappings between function spaces of infinite-dimensions. The advantage of FNO is that it aims to approximate an operator that learns to solve a family of PDEs by mapping known parameters to the solution of that PDE, instead of only approximating one instance of a PDE problem. Due to the nature of FNO, this enables us to use our synthetic data in order to approximate an entire class of problems at once. The novelty of FNO is that the kernel function, which is learned from the data, is parameterized directly in Fourier space, leveraging the Fast Fourier Transform when computing the kernel function. We train FNO using Adam optimizer on batches of size 100, with a learning rate of 0.001, modes set to 12, and of width 64. We also use relative L_2 error to measure performance for both training and testing.

We focus on second-order semi-linear elliptic PDE equations in divergence form defined on $\Omega := (0,1)^2$, with zero boundary conditions, given by

$$\begin{cases}
-\operatorname{div}(A(x)\cdot\nabla u) + b\cdot\nabla u + cu + g(u) = f & \text{in } \Omega \\
B(u) = 0 & \text{on } \partial\Omega,
\end{cases}$$
(7)

where $A(x) \in \mathbb{R}^{2 \times 2}$, $b \in \mathbb{R}^2$, $c \in \mathbb{R}$ and g(u) is some nonlinear function in u and B(u) is either B(u) = u or $B(u) = \nabla u \cdot \nu$, where ν is the exterior unit normal vector to the boundary $\partial \Omega$ that correspond to zero Dirichlet or Neumann condition, respectively. Here we also assume that A is uniformly elliptic and each $a_{ij} \in L^{\infty}(\Omega)$ with $i, j \in \{1, 2\}$.

For the rest of the paper, we will denote by $H(\Omega)$ the corresponding Sobolev space depending on B(u), which is $H(\Omega) = H_0^1(\Omega)$ when B(u) = u and $H(\Omega) = H^1(\Omega)$ when $B(u) = \nabla u \cdot \nu$.

4.1 The Poisson Equation

We first consider a simple example of problem (7), the Poisson equation, by taking A(x) = I (the 2×2 identity matrix), b = (0,0), c = 0 and $g(u) \equiv 0$

$$\begin{cases}
-\Delta u = f & \text{in } \Omega \\
B(u) = 0 & \text{on } \partial\Omega,
\end{cases}$$
(8)

Our goal is to learn an operator of the form:

$$G:L^2(\Omega)\longrightarrow H(\Omega)$$

$$f\longmapsto u$$

Notice that the Poisson equation can be easily solved when fixing f, however, here we would like to demonstrate our method of generating data on this easy problem first. Later we will consider more complicated examples.

We generate data points of the form (f, u) where u is defined as in (6), depending on B(u), and f is computed by taking derivatives of u so that (8) holds. This way, we can generate a lot of data. We let M and K in (6) range between 1 and 20, so that we can get a variety of such functions and various oscillations. We perform experiments by training with 1000, 10000, and 100000 functional data points and testing with 100 data points for the Poisson problem with Dirichlet and then with Neumann boundary conditions. We report the relative L_2 errors in the following Table 1.

	Dirichlet			Neumann		
Training points	1,000	10,000	100,000	1,000	10,000	100,000
Training loss	0.02972	0.00731	0.00218	0.00926	0.00297	0.00239
Testing loss	0.09674	0.01484	0.00358	0.02373	0.00419	0.00231

Table 1: FNO performance on the Poisson equation using our synthetic data generated as in (6).

Testing on f **beyond finite trigonometric sums**. Notice that if u is represented as a finite linear sum of sines and cosines, as in (6), then f generated according to (8) also consists of a finite linear sum of sines or cosines depending on B. So it is important to test on f's that are not sums of sines or cosines to demonstrate that our method of generating data generalizes well.

Restricting our attention to the Dirichlet case, let us generate f so that it does not consist of sine or cosine functions. This is akin to out-of-distribution testing in the machine learning literature. We consider the following two example functions: $f_1(x,y)=x-y$, which is smooth, and $f_2(x,y)=|x-0.5||y-0.5|$, which is a not everywhere differentiable function. However, in each case, f_1, f_2 are in $L^2(\Omega)$, and approximation of L^2 functions by trigonometric functions is well studied, and error bounds are available (see (5)). So we expect to obtain approximate solutions to the Poisson equation (8) for any f function that is in $L^2(\Omega)$.

In Figures 1 and 2, we summarize the predicted solutions using FNO, when trained with 1,000, 10,000 and 100,000 synthetic data functions that consist of sine functions given by (6). We also record the relative mean squared errors (RMSE) for each example. The following demonstrates that the choice of functions constructed as in (6) generalizes well.

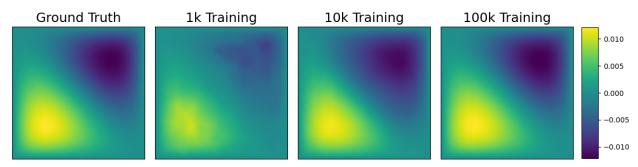


Figure 1: Predicted solutions of the Poisson equation with $f_1(x, y) = x - y$ as the right-hand side using FNO with 1,000, 10,000 and 100,000 training data points. Their RMSEs are 0.406, 0.116 and 0.024, respectively.

Note that FNO performs better when predicting a solution to the Poisson equation when the right-hand side is given by a smooth function, and has a harder time when the right-hand side is not smooth in Ω .

4.2 Second-order linear elliptic PDE

In problem (7), take g(u) = 0 and allow the matrix A and the lower-order terms to be of any form, possibly depending on (x, y). Then (7) becomes

$$\begin{cases} -\operatorname{div}(A \cdot \nabla u) + b \cdot \nabla u + cu = f & \text{in } \Omega \\ B(u) = 0 & \text{on } \partial \Omega, \end{cases}$$

$$(9)$$

In general, since we use derivatives in our computations, we assume that the entries of A are once differentiable in the corresponding variables.

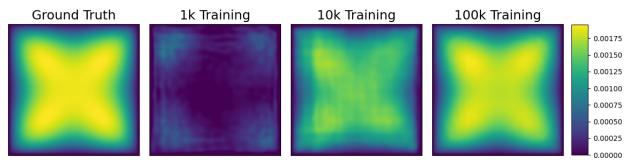


Figure 2: Predicted solutions of the Poisson equation with $f_2(x,y) = |x - 0.5||y - 0.5|$ as the right-hand side using FNO with 1,000,10,000 and 100,000 training data points. Their RMSEs are 0.620,0.162 and 0.042, respectively.

A as a fixed matrix. First, we look at the case where we fix a matrix A. Then we compute the derivatives involved for the components of A and save those as well. We generate a function u according to (6), plug it in to (9), and then compute f. As before, the goal is to learn the operator

$$G: L^2(\Omega) \longrightarrow H^1(\Omega)$$

 $f \longmapsto u$

For a numerical experiment, let A be as follows

$$A = \begin{pmatrix} x^2 & \sin(xy) \\ x+y & y \end{pmatrix} \tag{10}$$

In this case, FNO is learning a family of solutions for a fixed A defined above of the problem (9) for varying pairs of f and u functions. This choice of A is not particularly special, and the same process can be repeated for any positive definite A (so that (9) is elliptic). For the most accurate results, we can re-generate data points of the form (f,u) for each new matrix A and train different A-dependent neural networks. The following Table 2 summarizes the relative L_2 errors when using FNO to solve (9) when A is given by (10) and when training with 1,000, 10,000 and 100,000 data points and testing with 100 data points.

	Dirichlet			Neumann		
Training points	1,000	10,000	100,000	1,000	10,000	100,000
Training loss	0.01992	0.01147	0.00262	0.03452	0.00621	0.00229
Testing loss	0.04780	0.01523	0.00274	0.08926	0.00848	0.00215

Table 2: FNO performance on the problem (9) with A given by (10), using (6) functions.

A as a parametric matrix. As a more general-purpose approach to solving elliptic PDEs using FNO and synthetic data, we can also attempt to train a single neural network for an entire parameterized family of matrices A, by passing A as an input in the training data pair. That is, instead of fixing the matrix A in our synthetic data, we vary A within a parameterized class and pass it as input data together with f. In other words, the learning operator is of the form $G^{\dagger}:(f,A)\mapsto u$. For simplicity, we assume here that A is a diagonal matrix of the form

$$A(x,y) = \begin{pmatrix} \alpha(x,y) & 0\\ 0 & \delta(x,y) \end{pmatrix}$$

Here, we vary $\alpha(x,y)$ and $\delta(x,y)$. In other words, the operator we are trying to learn is given by

$$G: L^2(\Omega) \times L^{\infty}(\Omega) \times L^{\infty}(\Omega) \longrightarrow H(\Omega)$$

To further simplify, we assume the components of A are linear functions in x, y, that is

$$A(x,y) = \begin{pmatrix} m_1x + m_2y & 0\\ 0 & m_3x + m_4y \end{pmatrix}$$

where m_i 's are uniformly distributed in [0.1, 5] and u is generated according to (6) with $M \in \{1, 2, \dots, 10\}$. For each generated data point, we generate a matrix of the above form and a function u according to (6), then plug them both in equation (9) to compute f. Finally, the input data forms a triple (f, α, δ) , while the target is to predict u. This way, FNO learns how to solve a *family* of functions satisfying (9). We summarize the relative L_2 errors in Table 3 using FNO when training with 1,000,5,000 and 10,000 data points. As we can see below and as expected, the performance of the FNO with more degrees of freedom in the input data is worse compared to the case where the matrix A is considered fixed and held constant across all the input data.

	Dirichlet			Neumann		
Training points	1,000	5,000	10,000	1,000	5,000	10,000
Training loss	0.12266	0.07352	0.03134	0.14295	0.04639	0.01107
Testing loss	0.27257	0.10641	0.04885	0.25906	0.07611	0.05508

Table 3: FNO performance on the problem (9) with varying matrix A, using (6) functions.

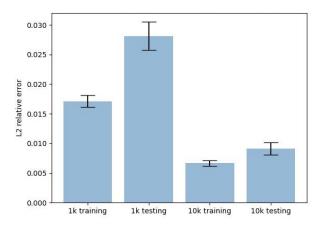


Figure 3: Relative L_2 errors with standard errors, over 10 experiments with fixed diagonal matrices linear in x and y.

4.3 Second-order semi-linear elliptic PDE

We take A = I, b = (0, 0), c = 0 and $g(u) = u^2$ in (7), in which case the problem becomes

$$\begin{cases}
-\Delta u + u^2 = f & \text{in } \Omega \\
B(u) = 0 & \text{on } \partial\Omega,
\end{cases}$$
(11)

In this problem we have a nonlinear term $g(u) = u^2$ added. It turns out that despite the nonlinear term, we get decent approximations of solutions when using our data generation with FNO.

As before, we generate u as specified in (6) and compute f by plugging into (11). From the theory, we know that the space of solutions is $H(\Omega)$. Numerical experiments show that despite the non-linearity in that term, FNO achieves low L_2 relative errors, as indicated in Table 4. We summarize the relative L_2 errors of training and testing loss in Table 4 when we train on 1,000,10,000 and 100,000 data points and test on 100 data points.

	Dirichlet			Neumann		
Training points	1,000	10,000	100,000	1,000	10,000	100,000
Training loss	0.01679	0.01763	0.00184	0.01017	0.00800	0.00237
Testing loss	0.03391	0.02693	0.00562	0.02992	0.01472	0.00295

Table 4: FNO performance on the problem (11) using (6) functions.

Testing on f **beyond finite trigonometric sums.** Notice that when we generate the unknown u to be of sums of sines or cosines, when plugging in equation (11), the f we compute still consists of sines or cosines, with some terms

squared. As before, after only training FNO with such u's, we are interested in seeing how well it generalizes when testing non-trigonometric $L^2(\Omega)$ functions. We demonstrate generalization through the following two examples: take $f_1(x,y) = xy$ and $f_2(x,y) = (x-0.5)^2 + (y-0.5)^2$.

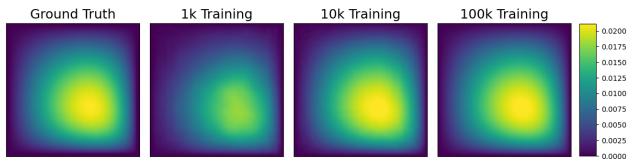


Figure 4: Predicted solutions of the semi-linear equation with $f_1(x, y) = xy$ as the right-hand side using FNO with 1,000, 10,000 and 100,000 training data points. Their RMSEs are 0.249, 0.064 and 0.06, respectively.

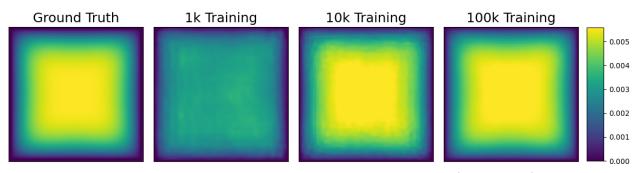


Figure 5: Predicted solutions of the semi-linear equation with $f_2(x,y) = (x-0.5)^2 + (y-0.5)^2$ as the right-hand side using FNO with 1,000, 10,000 and 100,000 training data points. Their RMSEs are 0.348, 0.086 and 0.085, respectively.

The error plateaus after a certain amount of training data and stops decreasing further, even though the predicted solution becomes smoother. We notice similar behavior across several right-hand sides in equation (11) that smooth, albeit not finite sums of sines or cosines.

4.4 Further examples

Second-order linear elliptic. We show an example of a linear second-order where we also include lower-order terms. For example take A = I, b = (3, 4), c = 1 and g(u) = 0. Then (7) becomes

$$\begin{cases} -\Delta u + 3u_x + 4u_y + u = f & \text{in } \Omega \\ B(u) = 0 & \text{on } \partial \Omega, \end{cases} \tag{12}$$

Once again, we would like to learn the operator

$$G:L^2(\Omega)\longrightarrow H^1(\Omega)$$

$$f\longmapsto u$$

Below we provide an example where the error increased when we went from 10k training data to 100k. While it is typically the case that more data is better, in some cases, like the one below, it could decrease the performance on a particular example.

Additional semi-linear examples Here, we consider problem (7) with A = I, b = (0,0), c = 0, $g(u) = \varepsilon e^u$ and B(u) = u, that is

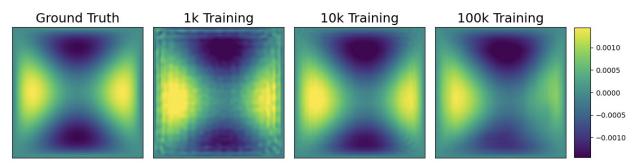


Figure 6: Predicted solutions of the linear equation (12) with $f(x,y) = (x-0.5)^2 - (y-0.5)^2$ as the right-hand side using FNO with 1,000, 10,000 and 100,000 training data points. Their RMSEs are 0.545, 0.487 and 0.501, respectively.

$$\begin{cases}
-\Delta u + \varepsilon e^u = f & \text{in } \Omega \\
u = 0 & \text{on } \partial \Omega,
\end{cases}$$
(13)

Notice that when $\varepsilon = 0$, (13) becomes the Poisson equation. In this experiment, we let $\varepsilon \in [0, 1]$ take values in increments of 0.1 starting from 0. The purpose of this is to demonstrate the performance of our method as we go from a linear to a more non-linear problem by recalling the nonlinear term in (13).

For each $\varepsilon=0,0.1,\cdots,0.9,1.0$ we generate 10k training data and test on 100, from which five examples are testing data where the right-hand side is first picked and we use a numerical solver to get the solution so we can test on whether we have good generalizations. We record the RMSEs of the predicted solution to problem (13) by fixing the right-hand side f and a $\varepsilon=0,0.1,\cdots,0.9,1.0$. We summarize the testing performance in the following plot and see that as we go from linear to non-linear, the performance improves, which at first seems surprising.

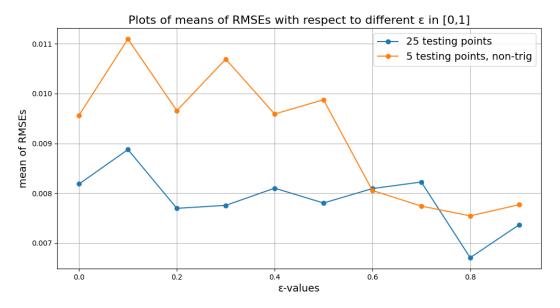


Figure 7: Plots of means of RMSEs of 25 testing data for problem (13) generated using our method and 5 testing data with non-trig right-hand sides for which we invoked numerical solvers to obtain u.

The Poisson equation on a triangular domain. Eigenvalues and eigenvectors of the Laplacian are also known in equilateral triangular domains given by $\Omega = \{(x,y) \in \mathbb{R}^2 : 0 < x < 1, 0 < y < \sqrt{3}x, y < \sqrt{3}(1-x)\}$, first discovered by Lamé (11) using reflection and symmetry arguments. We use the first 10 eigenvalues (some are with multiplicity two) and their corresponding eigenvectors in order to generate training data.

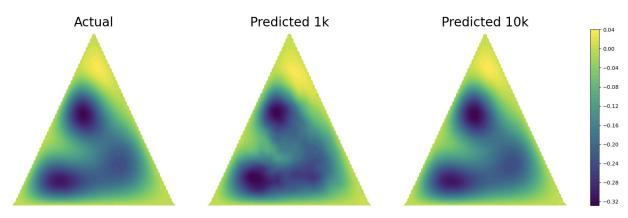


Figure 8: Predicted solutions of an example on a triangular domain using FNO with 1,000 and 10,000 training data. The RMSEs are respectively 0.10233 and 0.01424.

The Darcy flow equation. Here we present a case where using our method of generating data does not work very well compared to using the dataset provided in (10). The Darcy flow equation is given by

$$\begin{cases}
-\operatorname{div}(a(x)\cdot\nabla u) = f & \text{in }\Omega\\ B(u) = 0 & \text{on }\partial\Omega,
\end{cases}$$
(14)

In the paper (10), they fix $f \equiv 1$ and they are interested in learning the operator mapping the coefficients α into the solution u

$$G^{\dagger}: L^{\infty}(\Omega) \longrightarrow H_0^1(\Omega)$$

 $\alpha \longmapsto u$

In our setting, we are trying to learn the operator mapping the coefficients in α and the forcing term f into the solution u

$$G: L^{\infty}(\Omega) \times L^{2}(\Omega) \longrightarrow H_{0}^{1}(\Omega)$$

 $(\alpha, f) \longmapsto u$

Here, $\alpha \sim \mu$ where μ is the pushforward of a Gaussian measure with covariance $C = (-\Delta + 9I)^{-2}$ under the map

$$T: \mathbb{R} \longrightarrow \mathbb{R}_+$$

$$x \longmapsto \begin{cases} 12, & x \ge 0 \\ 3, & x \le 0 \end{cases}$$

Notice that by construction the coefficients are not smooth. We make some slight modifications to the FNO architecture so that it can take two functions (α, f) as an input and train FNO with 100,000 data points. For testing we use functions from the FNO dataset on the Darcy flow and and passing the input in the form $(\alpha, 1)$. Below we summarize performance of FNO trained with our data while testing is done with the FNO dataset. However, for examples of problems where the coefficients α are smoother, our method generalizes better.

5 Limitations, conclusion and future work

Limitations. Through experiments, we have observed that for certain "smooth" problems, our method generalizes well. However, as shown in the Darcy flow example (see Section 4.4), where the coefficients are non-smooth, our method did not generalize as effectively. While second-order elliptic PDEs represent a sizable class of problems, there are many other types of PDEs that fall outside this class, such as parabolic and hyperbolic. We have not yet tested the performance of our method on these other types of PDEs. We believe our method could apply to these other cases, but this requires further investigation. Finally, it is worth noting that the selected basis functions are not the only option, and different basis functions may be more suitable for certain problems. Alternative basis functions are worth investigating further.

Conclusion and future work. Using deep learning to solve PDEs has been very promising in recent years. Here, we propose a method that in some settings could eliminate the need to repeatedly solve a PDE for obtaining training data

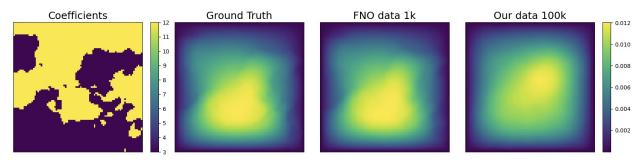


Figure 9: Predicted solutions of the Darcy flow equation using FNO with 1,000 of the FNO data set and training with 100,000 of our training data. The RMSEs are respectively 0.012 and 0.174.

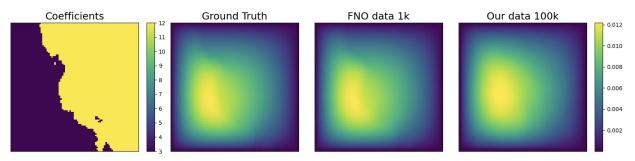


Figure 10: Predicted solutions of the Darcy flow equation using FNO with 1,000 of the FNO data set and training with 100,000 of our training data. The RMSEs are respectively 0.005 and 0.071.

used in training neural operators by first generating the unknown solution and then computing the right-hand side of the equation. Although we exclusively provide theoretical motivation and numerical experiments for second-order elliptic PDEs, this concept could be extended to other types of PDEs where the solution space is known beforehand, enabling the construction of representative functions for such solution spaces. This method could open up the possibility of obtaining good predictions for PDE problems using data-driven neural operators, for which the training data does not require classical numerical solvers to generate. We stress that our synthetic data generation approach is computationally efficient, particularly compared to solving new PDE problems numerically to generate training data for each new problem instance. We believe that our approach is an important step towards reaching the ultimate goal of *using deep learning to solve PDEs that are intractable using classical numerical solvers*. We also note that as a by-product, our method eliminates sources of error coming from numerically solving PDE problems; instead, our synthetic training data is of the form of exact solutions to a problem on a fixed-size grid.

Acknowledgments

EH and RW were supported in part by AFOSR MURI FA9550-19-1-0005, NSF DMS-1952735, NSF IFML grant 2019844, NSF DMS-N2109155, and NSF 2217033 and NSF 2217069.

References

- [1] H. Attouch, G. Buttazzo, and G. Michaille. *Variational Analysis in Sobolev and BV Spaces: Applications to PDEs and Optimization, Second Edition.* MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2014.
- [2] K. Bhattacharya, B. Hosseini, N. B. Kovachki, and A. M. Stuart. Model reduction and neural networks for parametric pdes. *CoRR*, abs/2005.03180, 2020.
- [3] O. Bilgin, T. Vergutz, and S. Mehrkanoon. GCN-FFNN: A two-stream deep model for learning solution to partial differential equations. *Neurocomputing*, 511:131–141, 2022.
- [4] Q. Cao, S. Goswami, and G. E. Karniadakis. LNO: laplace neural operator for solving differential equations. *CoRR*, abs/2303.10528, 2023.

- [5] R. A. DeVore and G. G. Lorentz. *Constructive approximation*, volume 303. Springer Science & Business Media, 1993.
- [6] W. E and B. Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *CoRR*, abs/1710.00211, 2017.
- [7] L. Evans. Partial Differential Equations. Graduate studies in mathematics. American Mathematical Society, 2010.
- [8] V. Fanaskov and I. V. Oseledets. Spectral neural operators. CoRR, abs/2205.10573, 2022.
- [9] D. S. Grebenkov and B.-T. Nguyen. Geometrical structure of laplacian eigenfunctions. *SIAM Review*, 55(4):601–667, 2013.
- [10] N. B. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. M. Stuart, and A. Anandkumar. Neural operator: Learning maps between function spaces. *CoRR*, abs/2108.08481, 2021.
- [11] G. Lamé. Leçons sur la théorie math ématique de l'élasticité des corps solides, Bachelier. Paris, 1852.
- [12] Z. Li, D. Z. Huang, B. Liu, and A. Anandkumar. Fourier neural operator with learned deformations for pdes on general geometries. *CoRR*, abs/2207.05209, 2022.
- [13] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar. Neural operator: Graph kernel network for partial differential equations. *CoRR*, abs/2003.03485, 2020.
- [14] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, A. M. Stuart, K. Bhattacharya, and A. Anandkumar. Multipole graph neural operator for parametric partial differential equations. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020.*
- [15] H. Lin, L. Wu, Y. Xu, Y. Huang, S. Li, G. Zhao, and S. Z. Li. Non-equispaced fourier neural solvers for pdes. *CoRR*, abs/2212.04689, 2022.
- [16] Z. Long, Y. Lu, X. Ma, and B. Dong. Pde-net: Learning pdes from data. In J. G. Dy and A. Krause, editors, Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, volume 80 of Proceedings of Machine Learning Research, pages 3214–3222. PMLR, 2018
- [17] L. Lu, P. Jin, and G. E. Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *CoRR*, abs/1910.03193, 2019.
- [18] R. Molinaro, Y. Yang, B. Engquist, and S. Mishra. Neural inverse operators for solving PDE inverse problems. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 25105–25139. PMLR, 2023.
- [19] F. Pichi, B. Moya, and J. S. Hesthaven. A graph convolutional autoencoder approach to model order reduction for parametrized pdes. *CoRR*, abs/2305.08573, 2023.
- [20] M. A. Rahman, Z. E. Ross, and K. Azizzadenesheli. U-NO: u-shaped neural operators. CoRR, abs/2204.11127, 2022.
- [21] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.*, 378:686–707, 2019.
- [22] B. Raonić, R. Molinaro, T. D. Ryck, T. Rohner, F. Bartolucci, R. Alaifari, S. Mishra, and E. de Bézenac. Convolutional neural operators for robust and accurate learning of pdes, 2023.
- [23] K. Salari, P. Knupp Code Verification by the Method of Manufactured Solutions. United States, 2000. Web. doi:10.2172/759450.
- [24] J. H. Seidman, G. Kissas, P. Perdikaris, and G. J. Pappas. NOMAD: nonlinear manifold decoders for operator learning. In *NeurIPS*, 2022.
- [25] W. Shi, X. Huang, X. Gao, X. Wei, J. Zhang, J. Bian, M. Yang, and T. Liu. Lordnet: Learning to solve parametric partial differential equations without simulated data. *CoRR*, abs/2206.09418, 2022.
- [26] J. Shin, J. Y. Lee, and H. J. Hwang. Pseudo-differential integral operator for learning solution operators of partial differential equations. *CoRR*, abs/2201.11967, 2022.
- [27] J. A. Sirignano and K. Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *J. Comput. Phys.*, 375:1339–1364, 2018.

- [28] L. Tan and L. Chen. Enhanced deeponet for modeling partial differential operators considering multiple input functions. *CoRR*, abs/2202.08942, 2022.
- [29] V. Thomée. From finite differences to finite elements: A short history of numerical analysis of partial differential equations. *Journal of Computational and Applied Mathematics*, 128(1):1–54, 2001. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.
- [30] A. Tran, A. P. Mathews, L. Xie, and C. S. Ong. Factorized fourier neural operators. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net, 2023.
- [31] A. Vadeboncoeur, I. Kazlauskaite, Y. Papandreou, F. Cirak, M. Girolami, and Ö. D. Akyildiz. Random grid neural processes for parametric partial differential equations. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 34759–34778. PMLR, 2023.
- [32] S. Wang, H. Wang, and P. Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed deeponets. *CoRR*, abs/2103.10974, 2021.
- [33] C. White, R. Tu, J. Kossaifi, G. Pekhimenko, K. Azizzadenesheli, and A. Anandkumar. Speeding up fourier neural operators via mixed precision. *CoRR*, abs/2307.15034, 2023.
- [34] L. Zhang, T. Luo, Y. Zhang, W. E, Z.-Q. John Xu, and Z. Ma. Mod-net: A machine learning approach via model-operator-data network for solving pdes. *Communications in Computational Physics*, 32(2):299–335, June 2022.
- [35] X. Zhang and K. C. Garikipati. Bayesian neural networks for weak solution of pdes with uncertainty quantification. *CoRR*, abs/2101.04879, 2021.

6 Appendix

6.1 Notation

Notation and descriptions used in this paper. In the next few sections we present some more examples.

Function Spaces					
$L^2(\Omega)$	space of Lebesgue-measurable functions $u:\Omega\to\mathbb{R}$ with finite norm $ u _{L^2}=\left(\int_\Omega u ^2dx\right)^{1/2}.$				
$L^{\infty}(\Omega)$	space of Lebesgue-measurable functions $u:\Omega\to\mathbb{R}$ that are essentially bounded.				
$H^1(\Omega)$	Sobolev space of functions $u \in L^2(\Omega)$ with $ \nabla u \in L^2(\Omega)$, equipped with the inner product $\langle u, v \rangle = \int_{\Omega} uv + \int_{\Omega} \nabla u \nabla v$ and induced norm $ u _{H^1} = u _{L^2} + \nabla u _{L^2}$.				
$H^1_0(\Omega)$	completion of $C_c^\infty(\Omega)$ in the norm $ u _{H^1}$. If Ω is bounded, we have the equivalent norm given by $ u _{H^1}= \nabla u _{L^2}$.				
$C_c^{\infty}(\Omega)$	space of smooth functions $u:\Omega\to\mathbb{R}$ that have compact support in $\Omega.$				