DEEP LEARNING METHODS FOR SOLVING PARTIAL DIFFERENTIAL EQUATIONS

Jules Berman* **Dan Berenberg*** **Arvi Gjoka*** **Tymor Hamamsy***

ABSTRACT

Partial differential equations (PDEs) are ubiquitous throughout the natural sciences. Due to their prevalence, the development of numerically driven PDE solvers has been an active research area for decades. Recent methodological developments in Deep Learning attempt to approximate traditional the performance of solvers by engaging with the differential system using a data-driven paradigm rather than an analytical one. In this work, we will provide an overview of recent developments in the application of deep learning methods for solving partial differential equations. We provide an exposition of these methods as well as contrast them with the established numerical solvers. To this end we hope to give a clear picture of deep learning's successes and shortcomings in this area while suggesting possible ways that current research might continue to make progress.

1 Introduction

Partial differential equations (PDEs) arise in all natural phenomena where it is possible to ascertain an equation involving differential operators that governs the phenomena at large scales. A few examples of these are the Poisson equation for electrostatics and gravitation ($u_{xx} = \rho$), the heat equation ($u_t = u_{xx}$), the wave equation ($u_{tt} = u_{xx}$), Schrodinger's equation ($iu_t = u_{xx}$) and Einstein field equations. These equations can form systems of equations (multiple coupled equations), involve nonlinear operators, and may depend on time.

Generally, these equations prescribe families of solutions that have a certain behavior. For example, the solutions to the wave equations are superpositions of waves travelling right and left. While finding these families of solutions generally involves mathematical analysis of the equations, we generally want to find solutions to specific instances of the problem. A well posed problem is one where there is a unique solution to the differential equation based on the differential form and some initial condition or boundary condition.

1.1 Some Families of PDEs and Approaches to Solving Them

Solving a differential equation in a domain typically involves discretizing the domain and solving the differential equation or some form of it at each point. We will consider techniques below for solving PDEs in Euclidean geometries.

Non time dependent PDEs can be stated as a boundary value problem, where boundary conditions must be set and the solution is static in time. One example is the Poisson equation, $\nabla^2 u = f$ with some boundary conditions. Common approaches to such problems include:

*Equal Contribution — correspondence to: {jmb1174, db3783, ag4571, tch362}@nyu.edu

- Using properties of Fourier transform to diagonalize a linear system, such as $\partial_{xx}\hat{u}_j = -j^2\hat{u}_j$. For the Laplace equation, this transforms the problem into a diagonal one $-j^2\hat{u}_j = \hat{f}_j$ which can be solved in $O(n\log(n))$ using the FFT.
- Discretizing the Laplacian operator ∇^2 by a linear finite difference operator based on approximations to the second derivatives, such as $\partial^2 u = (1/(\Delta x)^2)(u(x+h) - u(x-h))$. Then solving the problem through the linear system $Lu = f$. This also can be solved as a nonlinear system if there are nonlinear terms in the PDE.
- More complex approaches that cast the differential equation as an integral equation.

Time dependent PDEs require discretization in time as well as space. A popular method to solve these problems is to use a marching scheme, where the discretization in time allows the solution to be computed from one time step to the next as a linear or nonlinear system. For a PDE which can be written as $\partial_t u = f(t, u(t))$, where $f(t, u(t))$ may contain other differential operators, a simple first order marching scheme might look like

$$u(t + \Delta t) = u(t) + \Delta t f(t, u(t)) \tag{1}$$

A simple ODE might contain no differential operators in $f(t, u(t))$ and would not require a system of equations to be solved. If $f(t, u(t))$ contains differential operators, as is the case for most complicated PDEs, several approaches exist:

- A finite difference approach which discretizes the differential operators along a grid, producing a linear or nonlinear system of equations. For example, a numerical scheme for solving the heat equation could look like

$$u(t + \Delta t, x) = u(t, x) + \frac{\Delta t}{(\Delta x)^2}(u(t, x + h) - u(t, x - h)) \tag{2}$$

- A finite difference approach which discretizes the spatial domain into a possibly irregular mesh. Then one must declare local basis functions of the solution centered on the vertices of discretization and use the weak formulation of the PDE with the aforementioned local basis functions as test functions. This method works very well for complicated geometries.
- An approach which exploits equations that can be written in the form of a continuity equation. This implies a conservation law such that a finite volume method can be used. This uses the divergence theorem to equate the rate of change inside each cell with the inflow or outflow at the boundary of the cell. This requires creating cells over the domain and works for equations which can be cast as $u_x + f(u)_x = f$.

These methods have various drawbacks and special cases depending on the equations they are applied to. For example, stiffness is a characteristic of certain parabolic equations with large eigenvalue ratios and leads to preference for implicit methods. Advection is a property of solutions to hyperbolic PDEs, such as the wave equation, which require special treatment at boundaries to avoid numerical artifacts. More complicated problems, like the Navier Stokes equations which model fluid flow, exhibit combinations of advective and diffusive terms that make solving them more complicated. Nevertheless, there is good treatment of these complicated equations in literature and industrial solvers.

1.2 Numerical Techniques

At the center of most numerical solvers is some system of equations. These equations might originate from a finite difference scheme, a Fourier transform, or some other process. Once a system of equations is formed, solving it could be approached in a number of ways,

1. By doing matrix vector multiplication, as in the case of many explicit methods for linear systems.
2. Setting up a linear system to be solved by a direct or indirect solver.
3. Setting up a nonlinear system of equations to be solved by a Newton iteration type method.

There are, of course, regimes where each of these is preferred.

All of these methods impose a discretization of the domain at some point. This means there is an inherent trade-off between accuracy and running time. A higher order finite difference scheme can be applied to get a more asymptotically accurate answer, but it requires more operations. Likewise, a smaller discretization will make the solution more accurate (assuming that the method is convergent) at the cost of a larger system of equations.

In general, explicit matrix vector multiplication is quadratic in the dimension, so these methods are typically faster when the formulation can be cast in this way. However, explicit methods also have problems with stiff systems of ODEs, so they may not give accurate answers for larger step sizes. In this case, it is actually faster to set up a linear system with a much larger time step for the same error. Some explicit systems cannot be cast as matrix vector multiplications to begin with, so a linear solve is needed. Other times, linear systems involve tridiagonal matrices, so the solution is asymptotically linear (same as sparse matrix vector multiplication) even for an implicit method. Finally, favorable eigenvalue distribution or size of the system makes iterative solvers for linear problems more ideal. Nonlinear systems of equations have to be solved by Newton iteration style algorithms.

We can summarize some of the benefits of traditional solvers which include

- Time-tested, off-the-shelf software for solving all kinds of different systems.
- Direct control over the error through adjustment of the mesh resolution.
- Well developed theory around the stability and possible failure modes of the methods.
- Adaptive techniques and well developed heuristics to manage trade-offs between speed, accuracy, and stability.

However, there remain some difficulties with these methods

- Complicated domains are hard to solve with many of the simpler methods. Typically, they require the use of finite element or finite volume methods.
- The curse of dimensionality becomes problematic past 3 dimensions, as the space that you need to discretize increases exponentially.
- Unavoidable trade-off between speed and accuracy.

1.3 Paper outline

This paper is structured as follows. We first discuss learning based stencil approximation as a natural evolution of numerical methods. We then introduce two major classes of deep learning solvers for PDEs: Physics Informed Neural Networks (PINNs) (Section 3) and Neural Operators (Section 5). The motivating mathematics and implementations of each class are described in detail. After each section, we review the successes and shortcomings of each model considering issues such as theoretical grounding, generalization, speed, and accuracy. Finally, we conclude the discussion with a practical assessment of the field of solving PDEs with neural networks and describe what remains unaddressed at the time of writing this work.

As a disclaimer, we note that the usage of the word "physics" throughout this paper is informal. As will be shown, some classes of PDE-solving neural networks (PINNs) are said to require *physical* knowledge. However, these methods are not limited to physical problems and are instead applicable to PDE systems in general.

2 Neural Network Stencil Approximation

One approach to applying neural networks for solving PDEs is to generalize the finite difference method by learning the difference stencil. This is laid out explicitly in [2]. For example, take the FDM second order centered difference of the spatial derivative $\partial_x u(x)|_j = (1/2\Delta x)u_{j+1} - (1/2\Delta x)u_{j-1} + O(\Delta x^2)$. If we wanted to increase the accuracy, we could change the stencil such that a Taylor expansion of the stencil produces the required error. This is an expression of the form

$$\partial_x u(x)|_j = \sum_{i=\ldots,j,\ldots} \alpha_i^{(n)} u_i \tag{3}$$

In addition to precomputed coefficients α, we could also replace these by neural networks which are functions of the node values u_i. This is an easy way to produce adaptive discretizations, especially when the solution is not locally smooth. If the solution is not locally smooth up to the discretization, existing finite difference method stencils which do polynomial interpolation may introduce errors as the underlying solution at that scale cannot be modelled by polynomials. Even for a relatively smooth solution, as in the case of the Burgers equation, we can take a very large discretization where the underlying function we are trying to discretize cannot be accurately expressed through polynomial interpolation. In this regime, neural network stencils still recover an accurate solution.

The training regime involves training on very fine data on small problems and using this method for inference on coarser discretizations but larger problems. So for example, once you have solved the solution accurately at a small scale Δx, you can learn the stencil necessary to solve the solution well at a larger scale ΔX using that larger stencil in the learning. There is some hope by the authors that this will allow using existing discretization sizes to resolve the physics of systems at smaller size, since training of these networks can be done on smaller problems where one could discretize with a smaller step size. Some of this has been shown on simple problems, however at the end of the day, the above statement is an extrapolation based on small experiments.

The authors do state some problems that exist with this method:

- Computation is slow compared to the traditional stencil methods. This is obvious, because it requires generating ground truth data, training networks and running inference as opposed to precomputed stencils. Even compared to locally adaptive stencils, they are still expensive. Training on small problems and running inference on larger ones could change this result if the latter problem is sufficiently large, but this requires further analysis.
- Similar to Finite Difference Methods, this method does not naturally extend to higher dimensions or complex geometries. However, there is a variety of future work that can be done.

3 Physics Informed Neural Networks

3.1 History

Researchers have discussed physics informed neural networks (PINNs) for a long time. In 1997, Lagaris et al. in their seminal: "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations", presented a method to solve partial differential equations using NNs [9].

In their work, NNs are used to approximate functions, and the cost term that gets minimized is represented by the sum of the initial/boundary conditions of the PDE and the feedforward NN that is trained to solve the differential equation. Lagaris et al. employed a multilayer perceptron with one hidden layer to approximate the solution functions.

Fast forward a few decades, with faster computers and GPU parallel processing, the idea of using machine learning and NNs to solve PDEs came back in vogue. In this next wave, the first machine learning methods to tackle PDEs however were not NNs. This earlier work included from Raissi et al., "Machine Learning of Linear Differential Equations using Gaussian Processes", [19] in which they used probabilistic machine learning, specifically Gaussian Processes, to solve "inverse problems", essentially inferring the parameters governing PDEs from data observations of the underlying systems.

3.2 Physics Informed Deep Learning

The core of the PINN work that we will discuss is "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations", [20] and "Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations", [21] comes from some of the same team that employed GPs to solve PDEs, Raissi et al. As the paper names suggest, the authors distinguish between data-driven solutions of nonlinear PDEs and data-driven discovery of nonlinear PDEs. For each case, they further divide their efforts between continuous time and discrete time domains.

Work involving NNs to solve PDEs theoretically relies on the universal approximation theorem of NNs.[5] PINNs use NNs as universal function approximators, by applying NNs to nonlinear PDEs, and using the automatic differentiation (i.e. autograd) built into NN computation to get any partial differentials of functions.

3.2.1 PINNs for data-driven solutions to PDEs

NNs have historically done better in big data than small data regimes, because issues of robustness and overfitting arise when overparameterized models are trained on small datasets. However, in the case of PDEs where the functional form is known, there is prior information that can be encoded into NNs, and NN approaches can effectively regularize the search space of possible solutions and work with small data.

The basic setup equation is: $u_t + \mathcal{L}[u] = 0, x \in R^D, t \in [0, T]$. In this equation, $u(t, x)$ is the latent solution and $\mathcal{L}[u]$ is a nonlinear operator where the parameters are known. The goal of this setup is to learn $u(t, x)$ of the system given the known parameters of the nonlinear operator. Furthermore, the authors define the residual: $R(t, x) := u_t + \mathcal{L}[u]$, and they approximate $u(t, x)$ with a deep neural network, using the fact that NNs are universal approximators. As a result, the authors have a physics informed neural network (PINN), $u(t, x)$, and they can use automatic differentiation to find the differentials of functions and compute the residual at any point $R(t, x)$.[3]

Example: Burgers' Equation

Burgers' Equation is a common equation found in applied mathematics, and is derived from the Navier-Stokes equations. In one dimension with Dirichlet boundary conditions, Burger's equation is as follows:

$$
\begin{aligned}
&u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1, 1], \quad t \in [0, 1], \\
&u(0, x) = -\sin(\pi x) \\
&u(t, -1) = u(t, 1) = 0
\end{aligned}
$$

Then, the residual of the PDE is: $R(t,x) := u_t + uu_x - (0.01/\pi)u_{xx}$, where $u(t,x)$ is a Neural Network. We can take derivatives of a neural network with respect to the inputs in an analytical sense since we can write the network as an analytical function $\text{MLP}(x) = \sigma(W\sigma(\ldots Wx + b\ldots)) + b)$. The parameters of the network are learned by minimizing the mean squared error loss from the residual and the error with respect to the boundary conditions. Formally we have,

$$MSE = MSE_R + MSE_\partial \tag{4}$$

$$MSE_\partial = \frac{1}{N_{\text{boundary}}} \sum_{i \in \text{boundary}} |u(t_i, x_i) - u_i|^2 \tag{5}$$

$$MSE_R = \frac{1}{N_f} \sum_{i=1}^{N} |R(t_i, x_i)|^2 \tag{6}$$

The MSE_∂ loss will relate to the initial data and boundary conditions of u, while MSE_R to the residual of the PDE.

As mentioned earlier, MSE_∂ acts as a regularizer for the NN, and allows the network to learn reasonable parameters in the small data regime ($N_{\text{boundary}} = 100$ and $N = 10,000$). The authors found that they were able to achieve competitive results for predicting $u(t,x)$ on this simple problem, and distinguish their approach to prior GP work, and found that their error is approximately two orders of magnitude lower using PINNs. The actual architecture of the NN, $u(t,x)$ was not fundamentally important, however they found that increasing the number of neurons and layers improved performance.
The authors also employ Runge Kutta methods and PINNs to predict their intermediate steps as a way to deal with initial value problems and learn the solution at intermediate steps.

3.2.2 PINNs for data-driven discovery to PDEs

PINNs can also be applied to the data-driven discovery of partial differential equations. In this case, the equation becomes: $u_t + \mathcal{L}[u;\lambda] = 0, x \in R^D, t \in [0,T]$. Where $u(t,x)$ again is the latent solution, but now $\mathcal{L}[u;\lambda]$ is a nonlilnear operator that is parametrized by λ. The goal of the PINN for data-driven solutions then, is to learn the parameters, λ, of the PDE, using data from the system.

Example: Burger's equations for data-driven discovery

The Burger's equations for data-driven discovery are given by,

$$R(x,t,\lambda_1,\lambda_2) := u_t + \lambda_1 uu_x - \lambda_2 u_{xx}$$

The new objective is to learn the parameters λ_1, λ_2, that best fit a dataset for the latent $u(t,x)$ of a system. Again, $u(x,t)$ is approximated with a NN, and one can use automatic differentiation to find its differentials. The objective is the minimize the mean squared error loss given by

$$MSE = MSE_R + MSE_{\text{data}}$$

where MSE_R is defined in (6) and

$$MSE_{\text{data}} = \sum_{i \in \text{data}} |u(t_i, x_i) - u_i|^2$$

Now, the MSE_{data} will relate to the loss with respect to the training data on $u(t,x)$, while MSE_R regularizes the solution with respect to the PDE structure. In this paper, the authors make a dataset by generating 2,000 points from the solution of the PDE that has $\lambda_1 = 1.0$ and $\lambda_2 = 0.01/\pi$. When they train the network, they are able to accurately predict $u(t,x)$ and also learn the unknown parameters, even in the case of corrupted training data. One advantage over GPs that the authors state, is the robustness of their system up to 10% uncorrelated noise corruption.

4 Review of PINNs

PINNs are presented here as a simple and effective computational modelling approach to find the parameters of PDEs or to solve PDEs when parameters are known, working in the small data regime where other NN approaches fail. PINNs have been shown to converge on simple problems like Burger's equation, the Shrodinger equation, Navier Stokes equations, Kortewegde Vries Equations, and other such PDEs, and are a general approach that can be used when the PDE can be written explicitly or a loss involving the residual can be computed. While these equations have been tested mostly in 1 and some in 2 dimensions, this method naturally extends to higher dimensional PDEs. The challenge in testing this, however, is that it is hard to get baseline data with traditional methods for high dimensional PDEs. Another thing worth noting, at least in the forward case, is that the training loss is a direct measure of the error (unlike in other machine learning applications).

There are several limitations of PINNs that are worth discussing. Firstly, while PINNs lend themselves to simple implementations, problems in complex domains still require a lot of work in order to sample domains effectively. Intuitively you'd want to sample optimization points in parts of the domain where there are complex features in the solution, so sampling can even get adaptive. While there are papers that apply PINNs to more complex applications, they don't directly compare against state of the art non-learning methods very effectively, so their competitiveness compared to other methods is not yet well established. The PINNs presented here do not handle or present the uncertainty of their predictions, while other approaches, like GPs, naturally do this. There are however, newer versions of PINNs, B-PINNs: Bayesian Physics-Informed Neural Networks, that do present uncertainty intervals for predictions, and could address this problem. [23]

With regards to generalization, for both forward and inverse PDE problems PINNs have been shown to perform well. [17, 18] In the forward case, these authors find that PINNs accurately approximate the solutions to the forward problem for PDEs in a wide range of PDE problems, and they also introduce bounds on the generalization error of PINNs - how well PINNs can approximate solutions on new data. Their generalization error for forward problems depends on three factors: the training error, the number of training points, and the stability bounds of the PDE. Furthermore, they find that the generalization error will be low as long as the training error of the PINN is low (it was properly trained), there are enough training points (big enough training dataset), and the solution of the PDE is stable (not always the case for nonlinear PDEs). These authors also find that the generalization error for inverse problems depends on the same three factors. For many inverse problems, PINNs have been shown robust performance. [15, 22]

5 Neural Operators

A few recent lines of work have investigated a class of learning models formulated around operators as their primary mathematical object. Coined neural operators or operator networks, these methods are trained agnostic to the underlying mathematical system by being exposed to supervised, possibly noisy examples, e.g., the output of a costly simulation. Operator networks differentiate themselves from standard neural networks in that the latter is generally understood to approximate a function mapping between finite-dimensional, generally Euclidean spaces, and the former approximates a mapping between infinite-dimensional function spaces. This extension to support on functional spaces relies on an analogous universal approximation result for operators, reviewed in Appendix A.

5.1 DeepONet and DeepM&MNet

An extension of PINNs (from the same group at Brown), one approach is to use the operator universal approximation theorem and use neural networks to make the constants learnable. This theorem is not specific to PDEs, but one can think of an operator that maps from a forcing function of a PDE system

to the solution. After all, it is the forcing function that produces different behaviors in the solution. We can define the following problem and corresponding network to learn the operator:

Let $f_1(x), f_2(x), \ldots$ be the different forcing functions of a PDE, which are sampled at nonuniform nodes within the domain (but the same nodes for all forcing functions). Presumably, these nodes would be enough to resolve the forcing function. For each of these, we sample the true solution at points y_i, where $i = 1 \ldots n$ (n can be arbitrarily selected to choose the size of training data). That is, we have y_i and the value of the true solution $u(y_i)$ for each forcing function $f_j(x)$.

The network these define, along with the theorem in A, is laid out in [13]. We have two MLPs, one which takes in a vector of $[f_j(x_1) \quad f_j(x_2) \quad \ldots]^T$ and the other takes in a point where the solution is sampled y_i. The inner product of the output of the MLPs, then, predicts the value of the true solution $u_j(x)$ at y_i. Since we know the true solution here, we can set up a loss function and optimize the network to learn the operator. During inference, we can supply the network with an unseen forcing function $f^*(x)$ at a point y and expect the network to predict the value of the true solution at that point. The authors show that this technique converges for some simple operators, such as the 1d antiderivative operator and the operator for mapping a forcing function to a solution for a simple ODE.

An extension of the work uses DeepONets as building blocks for larger networks that try to work with larger systems of PDEs with coupled parameters. Some explorations into using these networks to solve larger, real world PDE systems can be found in [16] and [4].

5.2 Li's Neural Operator

A group of researchers at Caltech with lead author Zongyi Li have recently put out a series of papers which formulate their general approach to operator learning [12, 11, 10]. They formulate this by first establishing separable Banach spaces of functions $\mathcal{A}$ and $\mathcal{U}$. The fact that these spaces are Banach simply requires that despite being possibly infinitely-dimensional, there is still a notion of distance via a defined norm. Separability means that within $\mathcal{A}$ and $\mathcal{U}$, there resides a countable, dense subset. These requirements of $\mathcal{A}$ and $\mathcal{U}$ simply require that they contain Banach manifolds [8], infinitely-dimensional extensions of manifolds.

By defining a nonlinear map

$$G^\dagger : \mathcal{A} \mapsto \mathcal{U}, \tag{7}$$

which takes parameterizations of an underlying PDE system to their solutions, we can further define a parametric approximate operator G_θ to $G^\dagger$ where $\theta \in \Theta$ is a finite-dimensional set of parameters. The goal of the neural operator then is to learn an optimal $G_\theta^\dagger$, presumably capturing the topology of the underlying Banach manifolds within the input and output spaces, by minimizing the expectation of the norm defined on $\mathcal{U}$ between each $G_\theta(a_j)$ and $G(a_j)$ for some finite set of a_j. Note that so far, the approximate solution operator G_θ has not yet been cast as a neural network. Neural networks however are a sensible implementation choice for such a method as there is a somewhat unappreciated universal approximation result for operators [5, 14].

In the context differential equations, it is clear why neural operators are of interest. The solution to a well-posed differential equation is a function, and more generally a solver for a differential equation can be thought of as an operator. In the theory of differential equations these solution operators are typically formulated as Green's functions which provide some of the conceptual framework for the methods described in Li, et al [10, 11, 12]. We review briefly this theory briefly in Section 5.3.

The basic learning task for these neural operators is set up as follows. Some differential equation is chosen to solve. Then some parameters over this form are chosen as variables. This might be the coefficients on the differential equation, initial or boundary conditions, or some other parameter one is interested in. Many different values of this variable parameter are then drawn from a normal distribution and this set of values constitutes the input, A, of the training data. Then a numeric solver

is applied over the PDEs problems defined by each of these inputs. The solutions, U, generated by the solver constitute the output training data. The model then trained on this data, attempting to learn a map from A to U.

5.3 Inspiration from Green's Functions

Suppose we are given a linear, inhomogeneous differential equation which is defined on a domain Ω. We can succinctly write this in terms of a differential operator $\mathcal{L}$ which maps a function to some linear combination of its derivatives. If u and f are two function then we may write the differential equation as

$$(\mathcal{L}u)(x) = f(x) \tag{8}$$

In most applications we want to solve for the function u. Theoretically, this is as simple as inverting $\mathcal{L}$ and applying it to f. That is

$$u(x) = (\mathcal{L}^{-1}f)(x). \tag{9}$$

Indeed, for many differential operators (9) is often possible. The inverse of a differential operator is an integral operator (this should align with our intuition from the fundamental theorem of calculus) which we denote as M and write as,

$$(Mf)(x) = \int_{\Omega} k(x,y)f(y)\mathrm{d}y \tag{10}$$

where $k(x,y)$ is called the *kernel* of the operator. In the case of the differential equation, we have

$$u(x) = (\mathcal{L}^{-1}f)(x) = \int_{\Omega} G(x,y)f(y)\mathrm{d}y \tag{11}$$

where $G(x,y)$ is the kernel, called the Green's function. Thus it should be clear why the Green's functions is of particular interest. If we know the Green's function for a particular differential equation, then we should easily be able to solve for the solution $u(x)$ via this integral equation. Notably, (11) looks similar to a standard convolution, this is discussed further in Section 5.5 and 6.1.

The Green's function is presented here (and in Li's papers) as a foundational aspect of PDE theory which motives the idea of learning a solution operator and the subsequent construction of these operators using integral equations.

5.4 Neural Operators as Integral Operators

Formally, neural networks are a composition of alternating linear transformations and element-wise non-linearities, where each linear transformation is parameterized by a learnable weight matrix. In the case of neural operators we swap the linear function for some linear operator. For some input $a \in \mathcal{A}$ and output $u \in \mathcal{U}$, we may write a neural operator of depth d as,

$$u = \left(K_d \circ \sigma_d \circ \cdots \circ \sigma_1 \circ K_1\right) a \tag{12}$$

where each K_l is the linear operator and each σ_l is an element-wise non-linearity. Li, et al [12] choose to formulate the linear operator as an integral operator. Such a choice is loosely motivated by some of the previously discussed PDE theory in that integral equations are useful for solving PDEs. We can then write down one neural operator layer, $(\sigma_l \circ K_l)(\cdot)$, as

$$v_{l+1}(x) = \sigma_l\left(\int_{\Omega} \kappa_\phi(x,y)v_t(y)\mathrm{d}y + Wv_l(x)\right). \tag{13}$$

Here the layer is shown to be a function of two objects: the kernel κ, parameterized by ϕ and the linear operator W. The kernel κ is responsible for obtaining a new representation of the input, while

W ensures the network carries through relevant information from the previous layer, analogous to residual connections [7].

As shown in (12), the layers in (13) are composed in succession to form a network of arbitrary depth.

Additionally, the authors choose to "lift" the input space into a higher dimension in order to learn more descriptive features. Thus the entire compositional form of the neural operator is given by

$$Q(K_d \circ \sigma_d \circ \cdots \sigma_1 \circ K_1)Pa \tag{14}$$

where P transforms a into a higher input dimension and Q returns the output to the original domain.

By noting the fact that each layer of the architecture itself is approximates an operator, as well as the single-layer universal operator approximation result due to Chen & Chen [5], the authors draw an analogy of this class of neural operators to classic iterative solvers. In the same way that successive iterations of a classic solver incur less error, it is assumed that each operator layer works in a similar way.

Evaluating the integral in (13) is left as an implementation detail; in [12] and [11] for example, the authors formulate κ as a graph neural network. These earlier methods showed competitive performance, but still left something to be desired. Recently however, a new version of the neural operator [10], deemed Fourier Neural Operator (FNO) (described in Section 5.5), which explicitly decomposes the input function into a truncated set of Fourier coefficients, has shown pique performance on an array of classical PDE problems.

5.5 Fourier Neural Operators

The Fourier Neural Operator defines the integral operator as a convolution. Formally, this is done by requiring $\kappa_\phi(x, y) = \kappa_\phi(x - y)$. With this formulation, we can apply the well known Convolution Theorem (Appendix B) which states that convolution in physical space is equivalent to element-wise multiplication in Fourier space. In terms of implementation, what this looks like is that each layer first maps the input into the frequency domain via the Fast Fourier Transform (FFT), second amplifies task-relevant frequencies of the representation via element-wise multiplication with a learnable weight tensor, and finally transforms the the amplified representation back to the spatial domain via the inverse FFT (IFFT).

Formally one FNO layer is written as,

$$\mathcal{F}^{-1}\left(R_\phi \cdot (\mathcal{F}v_t)\right)(x) \tag{15}$$

This is then fed into the network architecture described in Section 5.4 so that the update of the representation from one layer to another is given by,

$$v_{l+1}(x) = \sigma\Big(\mathcal{F}^{-1}\left(R_\phi \cdot (\mathcal{F}v_t)\right)(x) + Wv_t(x)\Big) \tag{16}$$

We see from the description that R, parameterized by ϕ, and W are the learnable parts of the layer. The FFT and IFFT can be thought of as fixed, differentiable transformations. In practice, the Fourier modes in the frequency representation are truncated to a fixed number, keeping only the most relevant components of the function. Empirically, strong performance on a few selected problems was achieved by truncating to just 12 modes.

With this formulation, we have clear insight into one main feature of the FNO: *resolution invariance*. Since the transformation of the function via the R_ϕ operator takes place in Fourier space and acts only on the Fourier modes, the Fourier Operator can be evaluated on functions sampled at an arbitrary number of points, regardless of the dimensionality of the training set.

This means we can train the FNO on functions discretized at one resolution and then evaluate it at another resolution. The authors call this property *resolution invariance* and it enables what they call

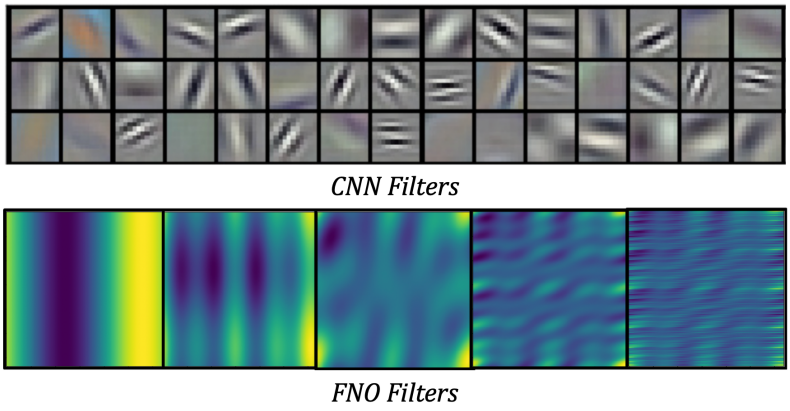

Figure 1: CNN filters vs FNO filters

zero-shot super-resolution. Being highly unconstrained, we cannot analytically verify such a claim, however empirically the method has shown to generalize well in such settings. These claims are examined further in Section 6.2.

6 Review of Neural Operators

6.1 Connection to Convolutions

As discussed previously the FNO is formulated by defining the kernel of an integral operator to be a convolution kernel, that is $\kappa_\phi(x, y) = \kappa_\phi(x - y)$. Then via Convolution Theorem (Appendix B), this convolution is approximated by discrete element-wise multiplication in Fourier space. So a natural question arises, how is this operation different from a Convolution Neural Network (CNN) which are based around a similar convolution operation. CNNs perform their convolutions by applying successive filters to inputs locally in physical space. By contrast, in FNOs the R matrix acts on the transformed input in Fourier space. This means that from the perspective of physical space, the FNO is applying a global convolution. This is local vs global dichotomy is a property Li observes in a blog post on this topic (Figure 6.1).

With knowledge of this difference between CNNs and FNOs, we can then ask what when we might prefer a global vs a local convolution scheme. In the case of computer vision, the canonical use case of CNNs, it is clear why local filters might be useful property. In many computer vision problems the relevant features we want to extract from an image are often independent and localized in space. By contrast, most PDEs describe the global dynamics of a system. For example, it is well known that the heat equation has an infinite domain of dependence. Explicitly what this means is that heat at one point in a domain affects the heat of every other point in the domain as the system evolves in time. So in this sense global transformations might be preferable in order to capture this property. This general idea is formalized in PDE theory as the CFL condition (Appendix C). While some aspects of this are not directly applicable here it still suggests that the domain of dependence of a numerical solution to a PDE and the domain of dependence of the PDE itself should coincide. While not all PDEs have a global domain of dependence, many do. This gives some theoretical justification as to why we might want to use FNOs over CNNs as a general architecture for solving PDEs. This intuition also reflects the empirical results as Li finds that FNOs out perform CNNs on a variety of PDE problems.

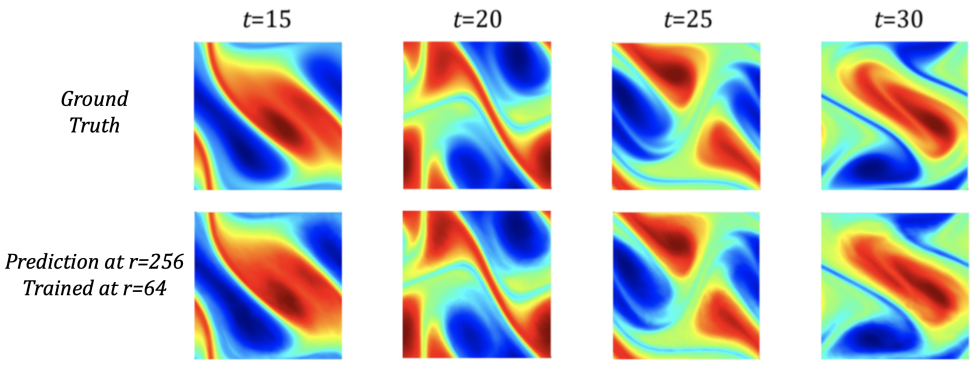

Figure 2: Evidence of zero-shot super-resolution provided by Li et al.

6.2 Resolution Invariance and “zero-shot super-resolution"

As discussed earlier Li’s FNO can be trained on functions discretized at one resolution and then evaluated on functions discretized at another resolution. This property which the authors term resolution invariance comes from the fact the transformation of the function happens in Fourier space. Practically speaking it is obvious that this is a desirable property. If one invested significant time and resources in training a successful PDE solver, one would not want to have to retrain the network when presented with a problem with a different discretization than the training data.

But aside from avoiding retraining, resolution invariance enables another property which the authors claim is a virtue of their method. That is, zero-shot super-resolution. This is simply the property of resolution invariance in the case that the problem we are evaluating is of much higher resolution than the training data. While at first this might seem like an exciting property, upon closer inspection it should be clear that what we really want to know is how the error behaves as we scale the problem resolution beyond the training resolution. This is because if the operator network converged to a very good approximation of the true solution operator during training, we might expect the error to decrease as we increased the resolution up to a certain threshold. Unfortunately this data is not presented. We are given images of a solution to the Navier-Stokes Equation generated at a resolution of $256 \times 256 \times 80$ from a network trained at a resolution of $64 \times 64 \times 80$ (Figure 2). The reader is meant to conclude from visual inspection that these solutions are good approximations to the ground truth. There is no mention of whether the error increased, decreased or stayed the same as the resolution was increased. Without any explicit data with regards to the error, it is difficult to assess what “zero-shot super-resolution" really means. As of now, it seems to be a repackaging of the resolution invariance property discussed previously.

6.3 FNOs Generalization

In the FNO paper the experiments are done by varying some parameters of a PDE problem and then using numerical solvers over these parameters to generate the training data. For example in the Burgers equation, the functional form of the PDE is fixed while a set of initial conditions are drawn from a normal distribution. Specifically u_0 is generated according to $u_0 \sim \mu$ where $\mu = \mathcal{N}\left(0, 625(-\Delta + 25I)^{-2}\right)$. The FNO does show good generalization error on test examples generated from this same distribution, but there is no analysis given with regard to how the test error is affected as we increase the variance of $\mathcal{N}$. These are key questions as more robust generalization

outside the training distribution would suggest that the FNO are truly approximating solutions operators. Whereas bad generalization might suggest that the model is just performing some type of naive interpolation. Ultimately the ability of FNOs to extrapolate to unseen problem parameters is necessary for its use in many practical applications. It is difficult to regard a pretrained model as a solver if it can only produce solutions within some small section of the problem space.

Another significant concern in the generalizability of neural operators is that it is not clear how to extend the method to multiple independent parameterizations (say we are interested in a system the behavior of which changes in response to two independent temperature and speed parameters), which is often the type of system we are interested in.

7 Conclusions

Here we have presented three classes of PDE solvers: traditional methods, physics informed neural networks, and neural operator networks, the latter two being of the most interest as they are deep learning approaches. Each class of method engages with mathematical theory differently. On one side of this spectrum, traditional solvers rely on decades of careful results that afford them analytical and numerical guarantees in a variety of contexts. In almost direct contrast, neural operators formulated in [12, 11, 10] rely almost entirely on the universal approximation theorem for operators and while drawing inspiration from a few fundamental results in PDE theory, their learned solution is entirely data driven and entirely deprived of a priori physical knowledge. That is, in addition to lacking rigorous empirical error analysis, it is not currently verifiable whether the operator network even represents the mathematical object it is modeled after, namely the Green's function. Physics informed deep learning lies somewhere between these poles. On one hand, PINNs are trained methods, meaning they could generalization poorly if in the training stage they are exposed to a biased set of inputs. On the other hand, assuming the training points sample the input space in such a way to resolve the underlying solution, PINNs approximate the solution function to the system directly by virtue of their design.

The manner in which these methods adhere to their mathematical design also functions as a guide to their general applicability. For real-world applications in which near exact solutions are necessary such as structural engineering or aerodynamics, traditional solvers are clearly the best choice; any claimed gain in speed or generalizability afforded through machine learning is entirely eclipsed by the provable error guarantees. That being said, a variety of real-world applications do not require such stringent assurance or become computationally intractable for traditional solvers when the problem grows sufficiently large. For these settings, such as large-scale many-body simulations, physics informed deep learning becomes more attractive. In this case properly trained PINNs will approximate the underlying physical rules in a tractable number of parameters while achieving relatively low amortized cost at inference time. Finally, for such cases in which the underlying system is not fully understood or unknown, neural operators may serve a valuable purpose in that they can yield a first-pass solution in an entirely data-driven manner. Of course finding a neural operator solution to such problems requires expressing the input and output data in an amenable format, which itself could be a nontrivial task. To date however, such an application of neural operators to a non-toy example has not yet been shown, suggesting their general applicability is still unknown. Additionally, while neural operator methods may be the newest, their novelty is still questionable.As highlighted throughout this work, operator networks formulated as integral operators are remarkably similar in form and function to convolution.

As hinted previously, a major limitation of the machine learning methods is the inability to accurately state error guarantees. These limitations however are not unique in that neural networks in general cannot do this. Though it is not a complete substitute, one way to address this obstacle by building uncertainty quantification into the network; this is done by reformulating the model as a Bayesian model and approximating a distribution of solutions (or solution operators in the case of neural

operators) and sampling from it at inference time conditioned on the new inputs. This paradigm has begun to emerge in PINNs [23], but has not yet been attempted for operator networks. The lack of Bayesian operator methods could be due to a variety of factors, the first that operator networks are still relatively new methods, the second being that it may be problematic to extend approximating a distribution over an infinitely-dimensional function space with the current variational methods. These lines of work in uncertainty quantification could become vital to the field of deep learning for PDEs, as it would inform practitioners of the biases or limitations of their networks.

In this work we have explored examples of neural PDE solvers. Although to varying degrees the methods suffer from major drawbacks with respect to their real-world feasibility, concerns of some of their mathematical coherence, and lack of strong empirical evidence of their generalizability, the methods remain as interesting lines of research for both theoretical and practical reasons. Practically, we'd like to replace costly methods with cheaper ones, possibly without access to the underlying governing rules. Theoretically, success in this field would indicate an important result that artificial neural networks can approximate arbitrarily complex systems in a controllably complex way. Additionally, in scaling to problems of contemporary scientific interest, this field could potentially elucidate active research areas by requiring only limited access to the system. Overall, although these works will require extensive testing and perhaps modifications to their formulation, they stand out as unique and exciting applications of deep neural networks.

Appendix A Universal Approximation Theorem for Operators

The universal approximation theorem for operators was first proved in [5] and is re-stated as follows in [14]:

Theorem 1. *Suppose that σ is a continuous polynomial function, X is a Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$ respectively, V is a compact set in $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, $i = 1, \ldots, n, k = 1, \ldots, p, j = 1, \ldots, m$, such that*

$$\left| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_i^k \sigma\left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma(w_k \dot{y} + \zeta_k) \right| < \epsilon \tag{17}$$

Notably, the only learnable parameters of in the expression are the $w_k \in \mathbb{R}^d$, implying that in fact such operators can be approximated to arbitrary accuracy by a single layer neural network followed by a nonlinearity.

Appendix B Convolution Theorem

The Convolution theorem is a foundational theorem in signal processing and Fourier analysis which states,

Theorem 2. *Given two functions $f(x)$ and $g(x)$ with Fourier transform G and H, their convolution is equal to the pointwise product of their Fourier transforms. Formally,*

$$r(x) = (g * h)(x) = \mathcal{F}^{-1}(G \odot H) \tag{18}$$

where $\odot$ is pointwise multiplication and $*$ is the convolution operator defined as,

$$(g * f)(x) = \int_{-\infty}^{\infty} g(\tau) f(x - \tau) d\tau = \int_{-\infty}^{\infty} f(\tau) g(x - \tau) d\tau \tag{19}$$

Appendix C The CFL Condition

The CFL condition is named for its authors Courant, Friedrichs, and Lewy, who first stated the idea in [6]. Here we restate a modern version given in [1]:

Definition 1. *CFL condition. A numerical method can be convergent only if its numerical domain of dependence contains the true domain of dependence of the PDE, at least in the limit as k and h go to zero.*

The values k and h refer to the incremental time and space steps respectively taken in a numeric solver applied to some PDE. While in our context there are no explicit time or space steps, or even a formal definition of convergence, we can still infer a general principle. That is, if a PDE solver does not capture the true domain of dependence of the PDE in its architecture, it will be inherently limited in its accuracy.

References

[1] "10. Advection Equations and Hyperbolic Systems." In: *Finite Difference Methods for Ordinary and Partial Differential Equations*, pp. 201–231. DOI: 10.1137/1.9780898717839.ch10. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717839.ch10. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898717839.ch10.

[2] Yohai Bar-Sinai et al. "Learning data-driven discretizations for partial differential equations." In: *Proceedings of the National Academy of Sciences* 116.31 (2019), pp. 15344–15349. ISSN: 0027-8424. DOI: 10.1073/pnas.1814058116. eprint: https://www.pnas.org/content/116/31/15344.full.pdf. URL: https://www.pnas.org/content/116/31/15344.

[3] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767 [cs.SC].

[4] Shengze Cai et al. "Flow over an espresso cup: inferring 3-D velocity and pressure fields from tomographic background oriented Schlieren via physics-informed neural networks." In: *Journal of Fluid Mechanics* 915 (Mar. 2021). ISSN: 1469-7645. DOI: 10.1017/jfm.2021.135. URL: http://dx.doi.org/10.1017/jfm.2021.135.

[5] Tianping Chen and Hong Chen. "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems." In: *IEEE Transactions on Neural Networks* 6.4 (1995), pp. 911–917. DOI: 10.1109/72.392253.

[6] R. Courant, K. Friedrichs, and H. Lewy. "On the Partial Difference Equations of Mathematical Physics." In: *IBM Journal of Research and Development* 11.2 (1967), pp. 215–234. DOI: 10.1147/rd.112.0215.

[7] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[8] David W. Henderson. "Infinite-dimensional manifolds are open subsets of Hilbert space." In: *Bulletin of the American Mathematical Society* 75.4 (1969), pp. 759–762. DOI: bams/1183530637. URL: https://doi.org/.

[9] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. ISSN: 1045-9227. DOI: 10.1109/72.712178. URL: http://dx.doi.org/10.1109/72.712178.

[10] Zongyi Li et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. 2021. arXiv: 2010.08895 [cs.LG].

[11] Zongyi Li et al. *Multipole Graph Neural Operator for Parametric Partial Differential Equations*. 2020. arXiv: 2006.09535 [cs.LG].

[12] Zongyi Li et al. *Neural Operator: Graph Kernel Network for Partial Differential Equations*. 2020. arXiv: 2003.03485 [cs.LG].

[13] Lu Lu, Pengzhan Jin, and George Em Karniadakis. "DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators." In: *CoRR* abs/1910.03193 (2019). arXiv: 1910.03193. URL: http://arxiv.org/abs/1910.03193.

[14] Lu Lu, Pengzhan Jin, and George Em Karniadakis. *DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators*. 2020. arXiv: 1910.03193 [cs.LG].

[15] Zhiping Mao, Ameya D. Jagtap, and George Em Karniadakis. "Physics-informed neural networks for high-speed flows." In: *Computer Methods in Applied Mechanics and Engineering* 360 (2020), p. 112789. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2019.112789. URL: https://www.sciencedirect.com/science/article/pii/S0045782519306814.

[16] Zhiping Mao et al. *DeepMMnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators*. 2020. arXiv: 2011.03349 [physics.comp-ph].

[17] Siddhartha Mishra and Roberto Molinaro. *Estimates on the generalization error of Physics Informed Neural Networks (PINNs) for approximating a class of inverse problems for PDEs*. 2021. arXiv: 2007.01138 [math.NA].

[18] Siddhartha Mishra and Roberto Molinaro. *Estimates on the generalization error of Physics Informed Neural Networks (PINNs) for approximating a class of inverse problems for PDEs*. 2021. arXiv: 2007.01138 [math.NA].

[19] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Machine learning of linear differential equations using Gaussian processes." In: *Journal of Computational Physics* 348 (Nov. 2017), pp. 683–693. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2017.07.050. URL: http://dx.doi.org/10.1016/j.jcp.2017.07.050.

[20] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*. 2017. arXiv: 1711.10561 [cs.AI].

[21] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. *Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations*. 2017. arXiv: 1711.10566 [cs.AI].

[22] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. *Hidden Fluid Mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data*. 2018. arXiv: 1808.04327 [cs.CE].

[23] Liu Yang, Xuhui Meng, and George Em Karniadakis. "B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data." In: *Journal of Computational Physics* 425 (Jan. 2021), p. 109913. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.109913. URL: http://dx.doi.org/10.1016/j.jcp.2020.109913.