Physics-Informed Deep Learning Frameworks for Solving Partial Differential Equations

Cody Rucker
Computer Science
University of Oregon

Advisory Committee:

Dr. Brittany Erickson, Dr. Boyana Norris, Dr. Daniel Lowd

Abstract—**A Physics-Informed Neural Network (PINN) is a Deep Learning (DL) framework for approximating solutions to Partial Differential Equations (PDE). Though PINNs benefit from the power and flexibility of DL, they lack the robust approximation theory of traditional numerical methods and, in the absence of real-world data, traditional methods outperform PINNs when solving PDE except in very high dimensions. However, the network architecture of PINNs is amenable to PDE for which traditional methods cannot be employed. This study highlights ways that the traditional and DL methods can be hybridized to exploit advantages inherent to both frameworks. We focus on the class of neural nets trained to satisfy Initial Boundary Value Problems (IBVP) (starting with the PINN framework) and investigate an approximation theory that is emerging in response to this class of physics-informed networks. We also present three areas of current research utilizing DL methods: First, using implicitly-computed antiderivatives, a mesh-free integration technique is presented on a 2D square domain as a starting point for approximating integrals in higher dimensions. Then, by investigating both continuous and discrete time formulations we may develop an efficient network for accelerating iterative methods. Lastly, the DeepONet solution for parametric PDE could allow us to create efficient and persistent physical systems which can generate data in a coupled system via a network forward pass.**

I. Introduction

In mathematics, PDE are used to encode physical phenomena in terms of differential relationships. Solutions to PDE are functions which exhibit the desired physical property, but such solutions are not guaranteed to have an analytic form so we often turn to numerical schemes like the Finite Difference Method(FDM) or Finite Element Method(FEM) to approximate solutions. However, an emergent Deep Learning(DL) framework has recently demonstrated a surprising capacity for approximating PDE solutions. Though this DL framework lacks the robust mathematical theory of the traditional methods, it shows particular promise in solving problems for which traditional numerical methods are ill suited.

Traditional numerical methods are typically solved on a mesh which, due to the curse of dimensionality [1], leads to computationally intractable problems in higher dimensions. Additionally, traditional approaches are not amenable to real-world observation and ultimately cannot be efficiently generalized to solutions of parametric PDE. This is contrasted by the DL framework which produces a closed, analytic form for the solution. Because of this, the solution is continuous and defined at every point in the domain allowing one to evaluate the solution 'off-grid' without having to solve the PDE again. While the DL framework cannot compete with traditional methods in lower dimensions, the 2017 foundational framework, PINNs, [2] suggests that both approaches may be used in tandem to offset specific weaknesses of a method.

The rest of this paper is organized as follows: In section II we provide brief history of Artificial Neural Networks (ANN) in solving PDE. section III Introduces the PINN framework and we give details for handling forward and inverse learning problems along with a formulation for discretizing the network with respect to time. Once we have established the basic PINNs framework section IV covers some ways that the PINN framework may be altered to enhance certain model features. section V introduces operator methods which provide a means by which PINNs can be generalized to a family of PDE solutions. section VI provides a look at the current state of an emerging approximation theory for physics-informed networks. Finally, I will conclude the paper with section VII in which I give my thoughts on the framework and carve out future areas of research.

II. Background

We begin by introducing the Feed Forward Neural Network (FFNN) architecture and give its Universal Approximation Theorems (UAT). FFNN also inherit regularity from their activation functions and, along with UAT this allows them to be infinitely differentiable function approximators, a property that is shared with polynomials. This makes FFNN a valid option in solving PDE and we will show how Machine Learning (ML) can be constrained by PDE.

A. *Feed Forward Neural Network (FFNN)*

Let $\mathbf{x} \in \mathbb{R}^n, W \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. Then a single hidden layer of a neural network can be expressed as

$$\ell(\mathbf{x};\theta) = \varphi.(W\mathbf{x} + b), \quad \theta = (W, b)$$

where φ is an activation function. That is, φ is bounded and

$$\lim_{x\to+\infty} \varphi(x) = \alpha$$
$$\lim_{x\to-\infty} \varphi(x) = \beta$$

with $\alpha \neq \beta$. The notation $\varphi.(*)$ signifies component-wise application of the activation function. We construct a deep neural network by repeated composition of hidden layers. Let $L \in \mathbb{N}_{>0}$ be the network depth and let $\{\phi_i\}_{i=1}^{L}$ be a collection of activation functions along with a sequence of trainable network parameters $\{\theta_i\}_{i=1}^{L}$ where $\theta_k = (W_k, b_k)$ for each $0 < k \leq L$. Take $\mathbf{x} \in \mathbb{R}^n$ to be the network input and network parameters $(W_1, b_1) \in \mathbb{R}^{m\times n} \times \mathbb{R}^m$, $(W_k, b_k) \in \mathbb{R}^{m\times m} \times \mathbb{R}^m$ for $1 < k < L$, and $(W_L, b_L) \in \mathbb{R}^{d\times m} \times \mathbb{R}^d$. Then the recursive composition

$$\ell_0 = \mathbf{x}, \tag{1a}$$
$$\ell_k = \varphi_k.(W_k\ell_{k-1} + b_k) \quad \text{for } 0 < k < L \tag{1b}$$

defines a neural network $\mathcal{N} : \mathbb{R}^n \to \mathbb{R}^d$ (of depth L and width m) with network parameters θ by

$$\mathcal{N}(\mathbf{x};\theta) = W_L\ell_{L-1} + b_L. \tag{2}$$

B. Universal Approximation Theorems

When considering such neural networks as function approximators there are two primary avenues to consider: arbitrary width and arbitrary depth. The arbitrary width formulation considers a continuous activation function $\phi : \mathbb{R} \to \mathbb{R}$ and a collection of feed-forward neural networks $\mathcal{N}_n^{\phi}$ mapping $\mathbb{R}^n$ to $\mathbb{R}$ which have a single hidden layer with an arbitrary number of neurons. Then we get the following universal approximation theorem (UAT):

Theorem 1 (Arbitrary-Width UAT[3])**:**
For compact subsets $K \subseteq \mathbb{R}^n$, $\mathcal{N}_n^{\phi}$ is dense in $C(K)$ if and only if ϕ is nonpolynomial.

Theorem 1 can be trivially extended to networks with a bounded number of hidden layers by including identity mapping layers [4], [5], [6], [7], [3].

The arbitrary-depth formulation, on the other hand, considers the approximation capabilities of networks with an arbitrary number of hidden layers each with fixed width. This formulation has been explored for Rectified Linear Units(ReLU) [8], [9], [10] and a minimal network size has been established as a necessary condition for deep networks to be universal approximators. However, as we will see later in the paper, the ReLU activation function cannot be utilized when constraining a network to satisfy PDE. More recently, a more general theorem has been proposed which shows the following: Let $\phi : \mathbb{R} \to \mathbb{R}$ and $n, d, m \in \mathbb{N}$. Denote by $\mathcal{N}_{n,d,m}^{\phi}$ the class of functions mapping $\mathbb{R}^n$ to $\mathbb{R}^d$ described by feed forward neural networks with n input neurons, d output neurons, and an arbitrary number of hidden layers, each with m neurons and activation function ϕ. Each neuron in the output layer uses the identity activation function. Then we have the following theorem:

Theorem 2 (Arbitrary-Depth UAT[3])**:**
For any non-affine continuous function $\phi : \mathbb{R} \to \mathbb{R}$ which is continuously differentiable at at least one point (with nonzero derivative at that point), and compact subset $K \subseteq \mathbb{R}^n$, the set $\mathcal{N}_{n,d,n+d+2}^{\phi}$ is uniformly dense in $C(K;\mathbb{R}^d)$.

Note that this theorem is quite general as far as the allowable activation functions. It includes polynomial activation functions which distinguishes it significantly from Theorem 1. Moreover, [3] proves that Theorem 2 can be extended to networks with activation functions which are bounded and continuous, but differentiable nowhere.

Theorem 3:
Let $w : \mathbb{R} \to \mathbb{R}$ be any bounded continuous nowhere differentiable function. Let $\phi(x) = \sin(x) + w(x)e^{-x}$, which is also nowhere differentiable. Let $K \subseteq \mathbb{R}^n$ be compact. Then $\mathcal{N}_{n,d,n+d+1}^{\phi}$ is dense in $C(K;\mathbb{R}^d)$ with respect to the uniform norm.

The oddity of this result lies in the fact that the set of differentiable functions is contained within the set of continuous functions. This means that we can uniformly approximate differentiable functions using activation functions which are differentiable nowhere. Thus deep neural networks maintain their universal approximation property for a pretty non-restrictive class of activation functions. Considering that gradient descent will require almost-everywhere differentiable activation functions, the choice of activation functions is limited by network training. The universal approximation theorems (in addition to the Neural Net's closed, analytic form) establishes ANN as potential PDE solvers. To train a Network to satisfy a PDE we first need to augment the ML optimization problem to incorporate information from IBVP. Inclusion of the IBVP is what makes a network *Physics Informed*.

C. Enforcing PDE in ML Training

One of the earlier appearances of a physics-informed network was proposed by [11] as a numerical solution for PDE on orthogonal box domains and was later expanded to domains of complex geometry by [12]. They use the compact and differentiable closed form of an Artificial Neural Network (ANN) solution to construct a trial function consisting of two terms: One term has no trainable parameters and satisfies a desired Boundary Condition (B.C) while the second term is a trainable feed-forward network which does not contribute to B.C but can be trained to solve a minimization problem. While this approach is fairly simple on orthogonal box domains, it becomes more involved on complex domains. Additionally, the manual computation of the loss function derivative terms is prone to error and becomes quite burdensome on deep neural networks.

More recently, [13] presents a unified framework for approximating PDE solutions using artificial neural networks and gradient-based optimization. This approach captures the idea proposed in [11] and provides a good basis for understanding physics-informed networks. We start with the general form of a PDE

$$\mathcal{O}[u](\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega, \tag{3a}$$
$$\mathcal{B}[u](\mathbf{x}) = g(\mathbf{x}), \quad \mathbf{x} \in \Gamma \subset \partial\Omega, \tag{3b}$$

where $\mathcal{O}$ is a differential operator, f a forcing function, $\mathcal{B}$ a boundary operator and g the boundary data. Here we take $\Omega = \mathbb{R}^n$ and Γ is the region on the boundary $\partial\Omega$ where boundary conditions are imposed. We approximate the solution by $u(\mathbf{x}) \approx \mathcal{N}(\mathbf{x};\ \theta)$ and consider $N_\Omega, N_\Gamma \in \mathbb{N}_{>0}$ collocation points taken from Ω and Γ respectively. Then we may define the constrained optimization problem

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N_\Omega} \frac{1}{2}\frac{1}{N_\Omega} \left|\left|\mathcal{O}[\mathcal{N}](\mathbf{x}_i;\theta) - f(\mathbf{x}_i)\right|\right|^2, \tag{4}$$

subject to constraints at the boundary defined by

$$\mathcal{B}[\mathcal{N}](\mathbf{x}_i) = g(\mathbf{x}_i) \quad \forall i = \{1, \dots, N_\Gamma\}. \tag{5}$$

This can be reformulated as an unconstrained optimization problem by training a network $G = G(\mathbf{x})$ to be a smooth extension of boundary data g and training a distance network $D = D(\mathbf{x})$ to be a smooth distance function that computes the distance from an internal point $\mathbf{x} \in \Omega$ to the nearest boundary point in Γ. Then, define a new network approximation for u by

$$\widetilde{\mathcal{N}}(\mathbf{x};\theta) = G(\mathbf{x}) + D(\mathbf{x})\mathcal{N}(\mathbf{x};\theta) \tag{6}$$

and the unconstrained optimization problem becomes

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N_\Omega} \frac{1}{2}\frac{1}{N_\Omega} \left|\left|\mathcal{O}[\widetilde{\mathcal{N}}](\mathbf{x}_i;\theta) - f(\mathbf{x}_i)\right|\right|^2. \tag{7}$$

The particular form of Equation 6 guarantees that the boundary conditions are satisfied on Γ. By building the boundary data in to the approximate solution, we are implementing a *hard* enforcement of boundary conditions. Now, we point out that the unconstrained optimization problem given by Equation 7 is very similar to the framework proposed in [11], [12] but the trial solution is vastly simplified by training boundary and distance networks. Additionally, manual differentiation of the loss function is no longer necessary as Automatic Differentiation (AD) can be used. Using AD, it is now possible to efficiently construct loss functions that constrain a network's derivatives without needing to massively restrict the network's depth. This allows for deeper, more expressive networks to be used when approximating PDE with ANN.

III. Physics-Informed Neural Networks

The 2017 framework, PINNS, establishes a class of neural networks that are trained to respect the physical laws encoded by general nonlinear PDE [2]. By constraining the network using an Initial Boundary Value Problem (IBVP), we effectively reduce the complexity of the network solution space which allows the network to be trained on a smaller set of data points. While this kind of network constraint is not novel [11], the framework proposes a model set up that can be minimally modified to handle two main classes of problems: data driven solutions and data-driven discovery of PDE. Furthermore, temporal evolution can be handled continuously or discretely. This gives the framework significant flexibility and efficiently exploits the predictive power of neural networks [14]. In this section we will first detail the model set up for data-driven solutions and data-driven discovery before showing how to handle the discrete-time formulation.

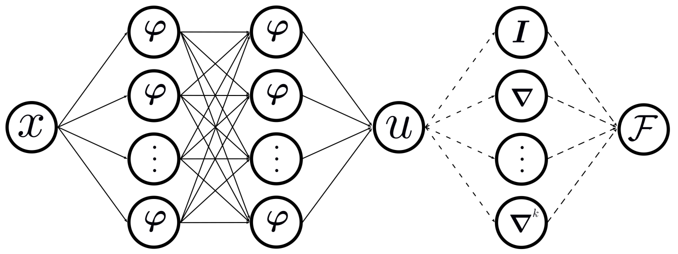

Fig. 1: Network diagram for a generic PINN with activations φ, input x, output u and PDE $\mathcal{F}$. Network connections shown with dashed lines represent non-trainable parameters.

Consider a spatial domain $\Omega \subseteq \mathbb{R}^n$ and time interval $\tau = [0, T]$ for some $T > 0$. Then $\hat{\Omega} = \Omega \times \tau$ is a spatio-temporal domain with boundary $\partial\hat{\Omega}$ on which we may define the general boundary value problem

$$u_t + \mathcal{O}[u;\lambda] = f, \quad (x,t) \in \hat{\Omega} \tag{8a}$$
$$\mathcal{B}[u] = g, \quad (x,t) \in \hat{\Gamma} \subseteq \partial\hat{\Omega} \tag{8b}$$

where $\mathcal{O}[\cdot;\lambda]$ is a nonlinear differential operator parametrized by λ, while f, $\mathcal{B}$, and g are similarly defined as in Equation 3. When the operator parameters λ are known apriori, we can solve the *forward* problem which approximates a solution u to Equation 8.

A. *Data-Driven Solutions*

We start by approximating the solution to Equation 8 using a feed-forward, multilayer neural network $u(x,t) \approx \mathcal{N}(x,t;\theta)$ and define a physics-informed neural network $\mathcal{F}$ by

$$\mathcal{F} := \mathcal{N}_t + \mathcal{O}[\mathcal{N};\lambda] - f. \tag{9}$$

Figure 1 shows the network diagram for a general PINN. Note that $\mathcal{F}$ and $\mathcal{N}$ will have the same set of trainable network parameters which can be learned by minimizing a mean squared error loss. To define this loss, we will consider N_∂ many initial/boundary points $\{x_\partial^i, t_\partial^i, g^i\}_{i=1}^{N_\partial}$ where $g^i = g(x_\partial^i, t_\partial^i)$ for each $1 \le i \le N_\partial$ and a set of $N_{\hat{\Omega}}$ internal collocation points $\{x_{\hat{\Omega}}^i, t_{\hat{\Omega}}^i\}_{i=1}^{N_{\hat{\Omega}}}$. Then for a set of N_{data} points given by

$\{x^i, t^i, u^i\}_{i=1}^{N_{\text{data}}}$ we define the mean squared loss as

$$MSE_{\text{data}} = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} |\mathcal{N}(x^i, t^i) - u^i|^2 \tag{10a}$$

$$MSE_{\partial} = \frac{1}{N_{\partial}} \sum_{i=1}^{N_{\partial}} \left|\mathcal{B}[\mathcal{N}](x^i_{\partial}, t^i_{\partial}) - g^i\right|^2 \tag{10b}$$

$$MSE_{\hat{\Omega}} = \frac{1}{N_{\hat{\Omega}}} \sum_{i=1}^{N_{\hat{\Omega}}} |\mathcal{F}(x^i_{\hat{\Omega}}, t^i_{\hat{\Omega}})|^2 \tag{10c}$$

$$MSE = \omega_d MSE_{\text{data}} + \omega_{\mathcal{B}} MSE_{\partial} + \omega_{\mathcal{F}} MSE_{\hat{\Omega}} \tag{10d}$$

where $\omega_d, \omega_{\mathcal{B}}, \omega_{\mathcal{F}}$ are, respectively, penalty weights associated with enforcement of the data, boundary, and differential conditions. The penalty terms can be learned by the network or set manually. Solving Equation 8 is done by minimizing Equation 10d with respect to the network parameters θ. That is, we employ a gradient descent algorithm to approximate the set of solution parameters θ^* such that

$$\theta^* = \arg\min_{\theta} MSE. \tag{11}$$

This results in the network $\mathcal{N}$ becoming an approximate solution to Equation 8. Moreover, the PINN given by Equation 9 includes the velocity term $\mathcal{N}_t$, so the resulting solution is continuous in time as well as in space. The data-driven solution can actually be carried out in the absence of any labelled data. In the event that $\{u^i\}_{i=1}^{N_{\text{data}}} = \varnothing$, the data-driven solution problem reduces to an unsupervised learning task [15]. This is one of the small differences to consider when dealing with data-driven discovery which requires that we have training data.

B. Data-Driven Discovery

The setup of a data-driven discover PINN is nearly the same as in subsection III-A but with a minor change. Instead of knowing the parameters λ of the nonlinear differential operator, we establish them as trainable network weights in Equation 9. This means that $\mathcal{N}$ and $\mathcal{F}$ no longer share the same set of trainable parameters as $\mathcal{F}$ is composed of the parameter set $\theta' = \theta \cup \lambda$. In order for the network to learn the operator weights we must have $\{u^i\}_{i=1}^{N_{\text{data}}} \neq \varnothing$, ideally with N_{data} sufficiently large so that the network accurately learns the operator parameters. This is all that is different between the two problem set ups. The collocation and boundary points are the same in each case and so is the definition of the loss function MSE. It is worth noting that the inverse problem of inferring system parameters is just as simple to set up as the forward problem of solution approximation. For both the solution and the discovery class of problems we have used a continuous-time formulation. While this formulation is fine for a small number of spatial dimensions, it can cause a significant bottleneck in higher dimensional problems as global enforcement of the PDE condition would require an exponential increase in collocation points [2]. The PINN framework circumvents this issue by introducing numerical quadrature along the time axis.

C. Discrete Time Model

For dynamic problems like Equation 8, PINNs may implement either continuous or discrete time evolution. The continuous model arises when we include a loss term approximating the velocity u_t. While this results in a solution that is continuous in time, the continuous model can be challenging to train. Alternatively, we can use a more traditional approach and discretize the time-axis. Perfmorning a time discretization leads to a solution that will be continuous in space but discrete in time. Moreover, we show how PINNs can exploit implicit Runge-Kutta methods to perform large time steps without accumulating temporal error. Let $r > 0$ and consider the $r-$stage general Runge-Kutta discretization of Equation A.3 defined for each $i = \{1, 2, \ldots, r\}$ at time t_η

$$u_i^\eta := u^{\eta+c_i} - \Delta t \sum_{j=1}^{r} a_{ij} \mathcal{O}[u^{\eta+c_j}], \tag{12a}$$

$$u_{r+1}^\eta := u^{\eta+1} - \Delta t \sum_{j=1}^{r} b_j \mathcal{O}[u^{\eta+c_j}]. \tag{12b}$$

This approach uses two sub-networks. The first of which uses input $x \in \mathbb{R}$ to approximate a vector of evaluations for u at each temporal RK-stage. Let $\mathcal{N}^\eta$ be a neural net defined by

$$\mathcal{N}^\eta(x;\theta) \approx \begin{bmatrix} u^{\eta+c_1}(x) \\ u^{\eta+c_2}(x) \\ \vdots \\ u^{\eta+c_r}(x) \\ u^{\eta+1}(x) \end{bmatrix}. \tag{13}$$

The second sub-net uses Equation 13 as its input along with Equation 12 to define a network encoding the PDE derivative conditions. To simplify notation, take $\mathbf{A} = (a_{ij})$, $\mathbf{b} = (b_i)$, and $\mathbf{c} = (c_i)$. Define the network $\mathcal{H}$ and composite network $\widehat{\mathcal{N}}$ by

$$\mathcal{H}[\mathcal{N}^\eta](x;\theta') = \mathcal{N}^\eta(x) - \Delta t \begin{bmatrix} \mathbf{A} \\ \mathbf{b}^\intercal \end{bmatrix} \mathcal{O}[\mathcal{N}^\eta](x), \tag{14}$$

$$\widehat{\mathcal{N}} = \mathcal{H}[\mathcal{N}^\eta]. \tag{15}$$

Let $\{x^k, u^{\eta,k}\}_{k=1}^{N_\eta}$ be system data associated with time t_η and take enforced boundary points $\{x^k_\partial\}_{k=1}^{N_\partial}$. Then the network parameters common to $\mathcal{N}$ and $\mathcal{H}$ can be learned by minimizing the sum of square errors

$$SSE_\eta = \sum_{j=1}^{r+1} \sum_{i=1}^{N_\eta} \left|\widehat{\mathcal{N}}_j^\eta(x^i) - u^{\eta,i}\right|^2, \tag{16a}$$

$$SSE_b = \sum_{i=j}^{r+1} \sum_{i=1}^{N_\partial} \left|\mathcal{B}[\widehat{\mathcal{N}}_j^\eta](x^i_\partial) - g(x^i_\partial)\right|^2, \tag{16b}$$

$$SSE = SSE_\eta + SSE_b. \tag{16c}$$

This formulation may seem a bit odd at first. The neural net is set up so that we can use the known solution at each time step (starting with the initial condition) to train the intermediate network to approximate the solution along the Runge-Kutta quadrature points in time. A network diagram is given in Figure 3 that shows how the RK PINN is connected. Because the new network is trained only on the known solution at the current time we no longer need any collocation points and $MSE_{\mathcal{F}}$ is dropped from the loss function. It is important to note that this formulation has a distinct advantage over classical Runge-Kutta methods. Namely, the number of RK stages can be increased without significantly increasing the training cost of the network. For each additional stage we just need to add a neuron to the output layer and include its associated network parameters. So the network parameters will only increase linearly with the number of Runge-Kutta stages [14], [2].

Note that if we choose $\mathbf{A}, \mathbf{b}, \mathbf{c}$ according to the Gauss-Legendre quadrature and Equation A.5 then the RK method is an implicit method with truncation error $\varepsilon = \mathcal{O}(\Delta t^{2r})$ for time step Δt and an $r-$stage method. This gives us a standard way of determining how many stages are necessary to bound the numerical approximation by some tolerance ε when Δt is known. That is, we may choose the number of RK stages by

$$r = \frac{1}{2} \frac{\log \varepsilon}{\log (\Delta t)}. \tag{17}$$

and since the Gauss-Legendre RK method is implicit and therefore unconditionally stable we may use large time steps and only need concern ourselves with the method's accuracy. Though the number of stages becomes unreasonable close to $\Delta t = 1$, we are permitted to take a time step of $\Delta t = 0.85$ and only need $r = 110$ to ensure the time-component of error is below ε. This means that the DL framework is capable of utilizing arbitrarily high order implicit RK methods for a relatively small cost. Considering the recent advancements that have been made in computing the GL quadrature weights and nodes [16], [17], along with the presentation of simple forms for the RK coefficient matrices [18], this particular feature of the DL method shows great potential as a numerical tool in scientific computing.

The use of the implicit RK method in tandem with PINNs suggests that hybridizing discrete and DL methods could yield powerful numerical tools. It is not yet clear how effective the hybridized approach is but we are compelled to investigate further. It seems natural to then wonder if the PINN framework could benefit from hybridization with other traditional numerical techniques.

IV. Alternate PINN Formulations

Since the PINN was proposed in [2] there has been a massive uptick in publications relating to deep neural network applications in solving PDE [15], [19]. PINNs have been implemented in solutions for a variety of physical problems like Navier-Stokes [20], [21], [22], convection heat transfer [23], solid mechanics, [24], [25] and the Euler equations [26]. Many of these studies apply the PINN framework on a very specific class of problems that range in complexity. We are concerned with PINNs as a framework so rather than delving too far into

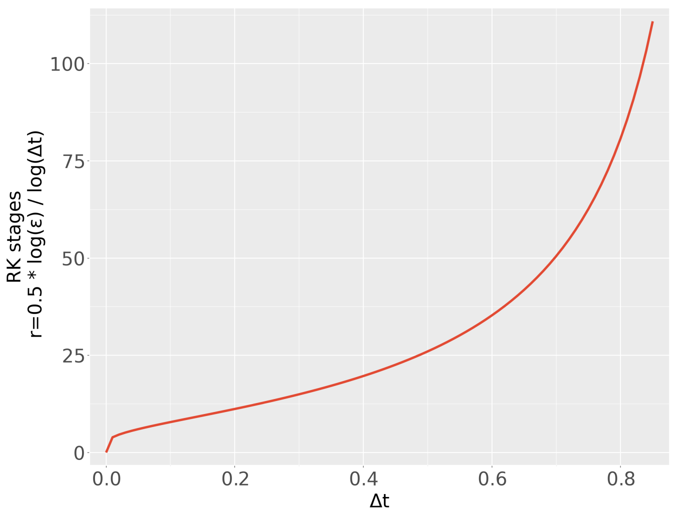

Fig. 2: Number of Runge-Kutta stages necessary to bound the temporal error accumulation (for GLRK) below machine precision $\varepsilon = 2.220e-16$.

specific problems we consider a more general approach that utilizes alternate forms of the PINN framework.

Though the standard PINN outlined in section III is the most commonly used formulation, there are a number of modified approaches that draw inspiration from more traditional numerical methods. While these alternate formulations may provide some computational advantage[27], [28], [29] they also highlight the flexibility of the deep neural network model and how it can be effectively hybridized with traditional numerical methods.

A. Variational Form (vPINN)

The vPINN, presented in [27], draws inspiration from the Petrov-Galerkin method for finite element approximations for solving PDE. We proceed by defining test and trial function spaces and using integration-by-parts to reduce the differential order of the governing problem [30], [31]. Consider the following initial boundary value problem,

$$\mathcal{O}[\mathcal{N};\lambda](\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega \tag{18a}$$
$$\mathcal{B}[\mathcal{N}](\mathbf{x}) = h(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega, \tag{18b}$$

where $\mathcal{O}$ is a (potentially) non-linear differential operator with parameters λ, $\mathcal{B}$ a boundary operator, and f some external forcing function. Equation 18 is known as the strong-form of the mathematical problem. By considering an appropriate test function $\phi(\mathbf{x})$, we may multiply Equation 18a by ϕ and integrate over Ω to get

$$\big(\mathcal{O}[\mathcal{N};\lambda](\mathbf{x}),\, \phi(\mathbf{x})\big)_{\Omega} = \big(f(\mathbf{x}),\, \phi(\mathbf{x})\big)_{\Omega} \tag{19a}$$
$$\mathcal{B}[\mathcal{N}](\mathbf{x}) = h(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega, \tag{19b}$$

where we use $(\cdot,\cdot)_{\Omega}$ to denote the usual inner product of functions on Ω. Though the boundary residual term is essentially the same as it is in the standard PINN formulation, the differential residual is no longer trained on collocation points within the domain. Instead we consider a collection

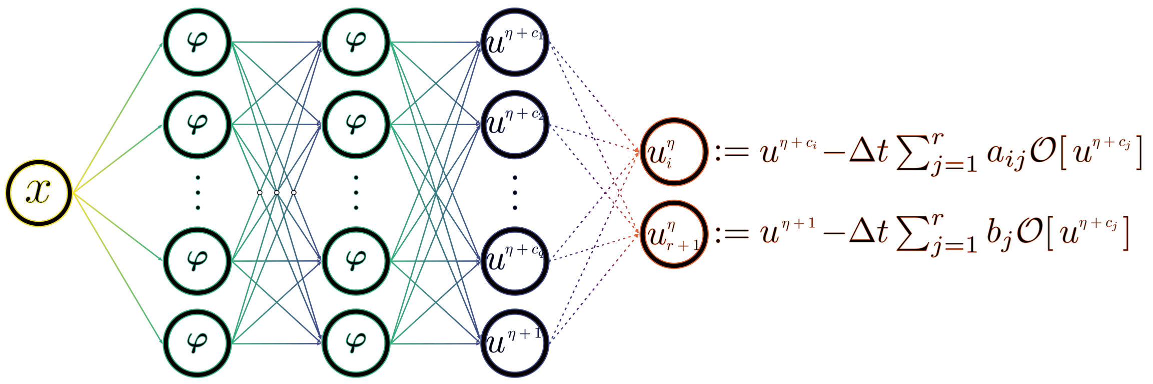
x
φ
$u^{\eta+c_1}$
$u^{\eta+c_2}$
$u^{\eta+c_q}$
$u^{\eta+1}$
$u_i^\eta := u^{\eta+c_i} - \Delta t \sum_{j=1}^r a_{ij}\mathcal{O}[u^{\eta+c_j}]$
$u_{r+1}^\eta := u^{\eta+1} - \Delta t \sum_{j=1}^r b_j\mathcal{O}[u^{\eta+c_j}]$

Fig. 3: A physics-informed neural net diagram for a discrete-time model using a general Runge-Kutta method. Dashed connections represent non-trainable network parameters.

$\Phi = \{\phi_k, k = 1, \ldots, K\}$ of admissable test functions and construct the residuals

$$MSE^{\nu} = \frac{1}{K}\sum_{k=1}^{K}\left|\left(\mathcal{O}[\mathcal{N}], \phi_k\right)_{\Omega} - \left(F, \phi_k\right)_{\Omega}\right|^2, \tag{20a}$$

$$MSE_{\partial} = \tau\frac{1}{N_{\partial}}\sum_{i=1}^{N_{\partial}}\left|\mathcal{B}[\mathcal{N}](\mathbf{x}_{\partial}^{i}) - h(\mathbf{x}_{\partial}^{i})\right|^2, \tag{20b}$$

$$MSE = MSE^{\nu} + MSE_{\partial}, \tag{20c}$$

where $\{\mathbf{x}_{\partial}^{i}\}_{i=1}^{N_{\partial}}$ is a set of boundary points of Ω and τ is a penalty term. The penalty term helps enforce data at the boundary points and is typically chosen based on the problem but can also be included in the set of learned parameters[27]. It is worth noting that the integration required in this approach only has an analytic representation for very simple networks and requires traditional numerical integration for deep neural networks. Note that this formulation changes the space of training data. Here one must define a set of domain points to be fixed as quadrature points while the variational PINN is trained using test functions. The spatial integral reduces the order of the PDE by one, allowing for the network to be more easily trained.

Though I have not seen it mentioned in any literature thus far, the variational problem allows us yet another option for handling the boundary conditions. One can build the B.C into the network(hard enforcement) or you can include it as a term in the loss function (soft enforcement). Because we are solving a variational problem over a space of test functions, the original boundary conditions of the problem may be enforced *essentially* or *naturally* depending on the quantity enforced by the B.C. Since we are approximating the solution using a space of functions we can choose to use functions which inherently satisfy the boundary conditions. The *essential* b.c are those which are imposed explicitly on the solution while *natural* b.c are imposed on solution derivatives. As an example, suppose we have the following boundary value Poisson problem:

$$-\Delta u = f \quad \text{on } \Omega, \tag{21a}$$

$$u = 0 \quad \text{on } \Gamma \subset \partial\Omega, \tag{21b}$$

$$\frac{\partial u}{\partial n} = 0 \quad \text{on } \partial\Omega \setminus \Gamma, \tag{21c}$$

where $\frac{\partial u}{\partial n}$ is the derivative of u in the direction normal to the boundary $\partial\Omega$. Then we construct a variational space to incorporate the essential boundary condition Equation 21b

$$V := \{v \in H^1(\Omega) \colon v|_{\Gamma} = 0\} \tag{22}$$

where $H^1(\Omega)$ is the space of L_2 functions on Ω whose first order derivatives are also in L_2. Additionally, $v|_{\Gamma}$ is interpreted using the trace theorem in $L_2(\partial\Omega)$, so we consider it to mean $v \cdot \chi_{\Gamma} = 0$ where χ is the usual characteristic function. With the choice of test space V, the *natural* boundary condition Equation 21c is automatically met for a $u \in V$ satisfying the variational form $a(u, v) = (f, v)$ for every $v \in V$ [32], [33].

Then the loss function consists solely of the term MSE^{ν} and boundary conditions are enforced exactly. Though this approach requires more manual overhead for the user, we can see how the variational form along with function space theory can reduce the cost of training the network. The drawback of this approach lies in establishing a numerical quadrature and being able to precisely choose a valid test space of functions.

B. *Conservative Form (cPINN)*

cPINNs, presented in [28], partitions the computational domain and defines a separate PINN on each sub-domain. Each PINN is tasked with approximating a solution locally on the sub-domain while preserving global requirements through interface conditions.

Recall the general IBVP given by Equation 8

$$\begin{aligned} u_t + \mathcal{O}[u; \lambda] &= 0, \quad (x, t) \in \hat{\Omega} \\ \mathcal{B}[u] &= g, \quad (x, t) \in \hat{\Gamma} \subseteq \partial\hat{\Omega} \end{aligned}$$

where this formulation refers to $\mathcal{O}$ as a nonlinear flux parametrized my λ. On each sub-domain we define a neural net training problem constrained by the IBVP for the system along with a conservative flux at sub-domain interfaces which maintain global conditions.

Let on sub-domain p we let $\mathcal{N}_p(x, y, t; \theta) \approx u(x, y, t)$ and define the local loss function

$$MSE_{\partial p} = \frac{1}{N_{\partial p}} \sum_{i=1}^{N_{\partial p}} \left| g^i - \mathcal{B}[\mathcal{N}_p](x^i_{\partial p}, y^i_{\partial p}, t^i_{\partial p}) \right|^2 \tag{23a}$$

$$MSE_{\mathcal{F}_p} = \frac{1}{N_{\mathcal{F}_p}} \sum_{i=1}^{N_{\mathcal{F}_p}} \left| \mathcal{F}_p(x^i_{\mathcal{F}_p}, y^i_{\mathcal{F}_p}, t^i_{\mathcal{F}_p}) \right|^2 \tag{23b}$$

$$\begin{aligned} MSE_I = \frac{1}{N_I} \sum_{i=1}^{N_I} \big| &\mathcal{O}_p[\mathcal{N}_p](x^i_I, y^i_I, t^i_I) \cdot \mathbf{n} \\ &- \mathcal{O}^+_p[\mathcal{N}_p](x^i_I, y^i_I, t^i_I) \cdot \mathbf{n} \big|^2 \end{aligned} \tag{23c}$$

$$\begin{aligned} MSE_{\text{avg}} = \frac{1}{N_I} \sum_{i=1}^{N_I} \big| &\mathcal{N}_p(x^i_I, y^i_I, t^i_I) \\ &- \{\mathcal{N}(x^i_I, y^i_I, t^i_I)\} \big|^2 \end{aligned} \tag{23d}$$

$$\begin{aligned} MSE_p = &\omega_\partial MSE_{\partial p} + \omega_{\mathcal{F}} MSE_{\mathcal{F}_p} \\ &+ \omega_I MSE_I + \omega_a MSE_{\text{avg}} \end{aligned} \tag{23e}$$

where $\mathcal{F}$ is the residual for the governing PDE, $\mathcal{O}_p \cdot \mathbf{n}$ and $\mathcal{O}^+_p \cdot \mathbf{n}$ are interface fluxes across subdomains. $N_{\partial p}$, $N_{\mathcal{F}_p}$, and N_I are the number of boundary/initial data, number of collocation points, and the number of interface points, respectively, within sub-domain p. Additionally, the average of $\mathcal{N}$ is

$$\{\mathcal{N}\} = \mathcal{N}_{\text{avg}} := \frac{\mathcal{N}_p + \mathcal{N}^+_p}{2} \tag{24}$$

This formulation allows for meticulous control over the total network. Since each subnetwork only interacts with its neighbors, training the network is very amenable to parallelization and would only require a small number of data transfers between neighboring networks. Additionally, each subnetwork need not be the same. Then shallower networks could be used in areas where the solution is well-behaved and more complex networks used on less behaved regions of the domain. Individual networks on each subdomain also increase the network expressivity while the flux conditions at interfaces reduce error propagation across subdomains [28].

V. Parametric PDE

Up until now we have been primarily concerned with solutions for IBVP in which every parameter of the problem is fixed and the resulting numerical output can only describe the fixed-state system. Instead, we may wish the solve the PDE over a range of possible scenarios (such as varying boundary/initial conditions, operator parameters, or domain geometries) so that our trained network represents a family of PDE solutions. This generalized problem is referred to as a parametric PDE and it cannot be solved using the traditional finite difference/element methods. Because traditional numerical methods would require a simulation to be carried out for every single scenario, the computational cost (along with the need to store each solution) quickly becomes prohibitive. While it is possible to adjust the PINNs framework for this task by introducing new input variables, this results in a high-dimensional PDE that will drive up the computational cost of higher-order derivatives in the loss function [34]. A number of techniques known as Neural Operator Methods have been proposed for this kind of problem [35], [36], [37], but there exists a fairly simple architecture which is established using a universal approximation theorem for operators.

A. DeepONet

The 2020 neural net architecture, DeepONet[38], employs a lesser known universal approximation theorem that established neural networks as universal approximators of functionals and operators[39], [40]. Like with Neural Operator methods, this shift in learning focuses on the approximation of infinite-dimensional function spaces rather than a finite-dimensional map between Euclidean spaces. We outline the approach in this section.

Consider the operator G with input function u. Then, for y in the domain of $G(u)$, $G(u)(y)$ is a real number. The DeepONet architecture consists of two sub-networks each trained on a separate space of inputs. The *branch* network is trained on different function inputs while the *trunk* network is trained on inputs from the PDE's Euclidean domain. However, we cannot simply train a newtwork on function inputs but rather must train the network on the image of said inputs. To this end, we define a set of *sensor* points $\{\xi_i\}_{i=1}^s$ which are to remain fixed across different function inputs. Then the *branch* network $\mathcal{N}_B : \mathbb{R}^s \to \mathbb{R}^p$ is defined as

$$\mathcal{N}_B\big([u(\xi_1), \ldots, u(\xi_s)]^T; \theta_B\big) = [b_1, \ldots, b_p]^T, \tag{25}$$

and the *trunk* network $\mathcal{N}_T : \mathbb{R}^n \to \mathbb{R}^p$ is defined as

$$\mathcal{N}_T(y; \theta_T) = [t_1, \ldots, t_p]^T. \tag{26}$$

The two networks are combined by using the output of the branch network as weights to the output of the trunk network in a straightforward linear combination the defines an operator network $\mathcal{N}_\theta$ by

$$\mathcal{N}_\theta[u](y) = \mathcal{N}_B\big(u(\boldsymbol{\xi}); \theta_B\big) \cdot \mathcal{N}_T\big(y; \theta_T\big) = \sum_{i=1}^{p} b_i t_i \tag{27}$$

where $\theta = \theta_B \cup \theta_T$. This formulation is shown to exhibit an approximation error which is dependent on the number of fixed sensors, $\{\xi_i\}_{i=1}^s$, and input function type while also demonstrating a reduction in generalization error when compared to a standard fully-connected network [38]. Moreover, [41] demonstrates how the DeepONet architecture can be used alongside PINNs to train an operator across various initial and boundary conditions. This may allow for the construction of persistent and efficient families of solutions for PDE. To develop the appropriate loss function let $\{u^i\}_{i=1}^N$ be a collection of N separate input functions and for each u^i take $\{y^i_{\partial j}\}_{j=1}^P$ to be P locations determined from data, initial, or boundary conditions and $\{y^i_{\Omega j}\}_{j=1}^Q$ to be a set of collocation points sampled from the domain of $G[\mathbf{u}^i]$. Here we use $\mathbf{u}^i = [u^i(\xi_1), \ldots, u^i(\xi_s)]^T$. Then the relevant loss function is given by

$$\mathcal{L}_{\partial}(\theta) = \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|\mathcal{B}\left[G[\mathbf{u}^i]\right](y^i_{\partial j}) - u^i(y^i_{\partial j})\right|^2, \quad (28a)$$

$$\mathcal{L}_{\Omega}(\theta) = \frac{1}{NQs}\sum_{i=1}^{N}\sum_{j=1}^{Q}\sum_{k=1}^{s}\left|\mathcal{O}\left[G_\theta[\mathbf{u}^i]\right](y^i_{\Omega j})\right|^2, \quad (28b)$$

$$\mathcal{L}(\theta) = \mathcal{L}_{\partial}(\theta) + \mathcal{L}_{\Omega}(\theta). \quad (28c)$$

Figure 4 gives a network connectivity diagram for a physics-informed deep operator network formulated across multiple initial/boundary data functions.

VI. Approximation Theory

Statistical Learning Framework for PINN Error Analysis

Consider $\{x_i\}_{i=1}^N$ collocation points of $\bar{\Omega} = \Omega \cup \partial\Omega$.

Let $\mathcal{N}_\theta$ be a neural net approximation (realized with parameters θ) approximating a solution to Equation 8. Define the residuals

$$\mathcal{R}_{\text{PDE}}[v] = v_t + \mathcal{O}[v] - f \quad (29)$$

$$\mathcal{R}_{\text{bdry}}[v] = \mathcal{B}[v] - g \quad (30)$$

The risk for a PINN can be defined as

$$E^2_G(\theta) = \int_{\bar{\Omega}} \left|\mathcal{R}_{\text{PDE}}[\mathcal{N}_\theta](x,t)\right|^2 dx\, dt + \int_{\partial\bar{\Omega}} |\mathcal{R}_{\text{bdry}}[\mathcal{N}_\theta](x,t)|^2 dx\, dt. \quad (31)$$

While we cannot compute the risk, we can approximate it using a discrete integral. The empirical risk is given by

$$E^2_T(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left|\mathcal{R}_{\text{PDE}}[\mathcal{N}_\theta](x_i,t_i)\right|^2 + \frac{1}{N_\partial}\sum_{i=1}^{N_\partial}|\mathcal{R}_{\text{bdry}}[\mathcal{N}_\theta](x_i,t_i)|^2. \quad (32)$$

Here, the training error E_T approximates the generalization error E_G by a simple 'rectangular' quadrature. Let θ^* be the final parameter set achieved by training the PINN. Then the optimization error $\mathcal{E}_O$ is given by

$$\mathcal{E}_O := E^2_T(\theta^*) - \inf_{\theta\in\Theta} E^2_T(\theta) \quad (33)$$

which measures the empirical risk of the final approximation against the smallest attainable empirical risk. Next, measuring the separation between risk and empirical risk we define $\mathcal{E}_G$ by

$$\mathcal{E}_G := \sup_{\theta\in\Theta}\left|E^2_G(\theta) - E^2_T(\theta)\right|. \quad (34)$$

Some of the literature involving PDE refers to $\mathcal{E}_G$ as the generalization error but I have opted to use generalization error as it is used it ML. Namely, E_G is the generalization error and $\mathcal{E}_G$ the risk approximation error. Lastly, the approximation error $\mathcal{E}_A$ given by

$$\mathcal{E}_A := \inf_{\theta\in\Theta} E^2_G(\theta). \quad (35)$$

These errors give rise to an upper bound on the global error between a trained network and the exact solution. The global error bound

$$E^2_G(\theta^*) \le \mathcal{E}_O + 2\mathcal{E}_G + \mathcal{E}_A \quad (36)$$

also indicates how each error term is vital in constraining the global error and define key research areas to improve mathematical applications for deep neural networks [42], [15]. While standard DL models can take advantage of ReLU activation functions to establish bounds on approximation error $\mathcal{E}_A$ [43], solving PDE requires networks(and by extension activation functions) which are sufficiently differentiable. While [44] presents an error analysis framework for PINNs when applied to linear PDE, a more general bound on approximation error can be established on Sobolev spaces[45], [15].

A. Approximation Error $\mathcal{E}_A$

While universal approximation theorems Theorem 1 and Theorem 2 can help identify a minimal network size for universal approximation they do not provide any sufficiency conditions to guarantee accuracy of the network. Specifying bounds on approximation error requires one to choose an appropriate network architecture and error is typically quantified as an increase to network complexity as $\epsilon \to 0$(asymptotic approximation rates). However, we instead would like to know how large a network must be to guarantee a specified accuracy will be reached. Such an error bound has recently been proposed for networks with two hidden layers which utilize hyperbolic tangent activation functions[45]. The choice to use a shallow but wide network is backed by empirical observation that shallow/wide networks perform better on scientific computing problems where training data may be scarce [46]. To give the approximation theorem we first define $R > 0$ such that $|\varphi^{(m)}|$ is decreasing on $[R,\infty)$ for every $1 \le m \le k$ and define

$$|P_n, d| = \binom{n+d-1}{n}. \quad (37)$$

Theorem 4 (tanh NN Sobolev Approximation [45])**:** *Let $d, s \in \mathbb{N}, R > 0$, $\delta > 0$ and $f \in W^{s,\infty}([0,1]^d)$. There exist constants $c(d,k,s,f), N_0(d) > 0$, such that for every $N \in \mathbb{N}$ with $N > N_0(d)$ there exists a $\tanh$ neural network $\widehat{f}^N$ with two hidden layers, one of width at most $3\lceil\frac{s}{2}\rceil|P_{s-1,d+1}| + d(N-1)$ and another of width at most $3\lceil\frac{d+2}{2}\rceil|P_{d+1,d+1}|N^d$, such that*

$$\left\|f - \widehat{f}^N\right\|_{L^\infty([0,1]^d)} \le (1+\delta)\frac{c(d,0,s,f)}{N^s}, \quad (38)$$

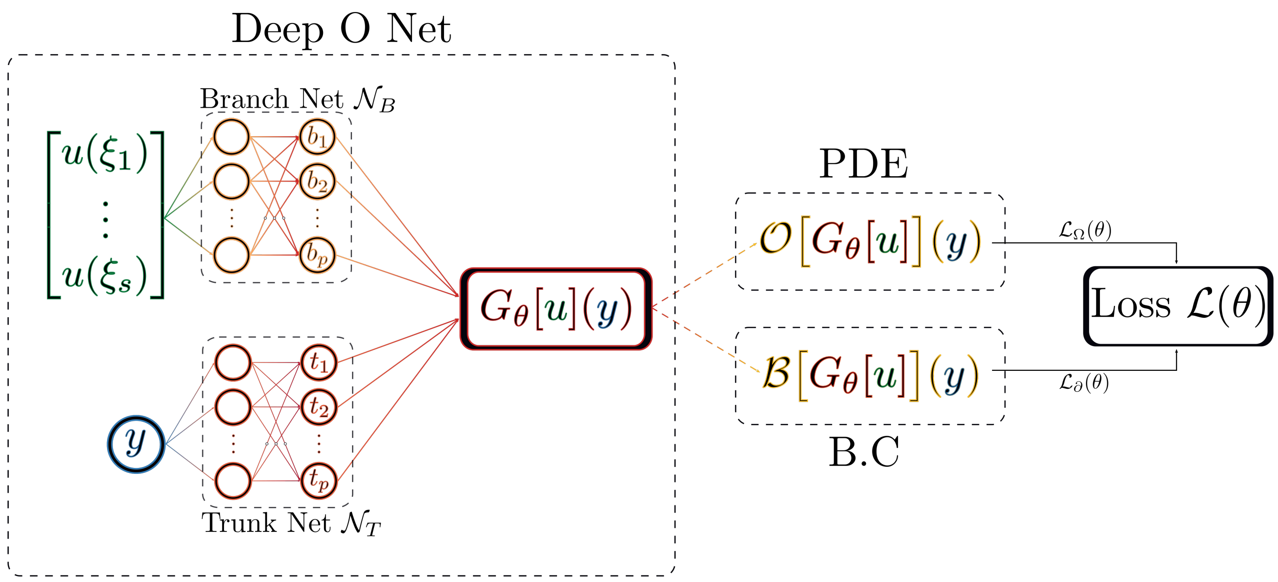
Deep O Net
Branch Net $\mathcal{N}_B$
$u(\xi_1)$
$u(\xi_s)$
b_1
b_2
b_p
$G_\theta[u](y)$
y
t_1
t_2
t_p
Trunk Net $\mathcal{N}_T$
PDE
$\mathcal{O}[G_\theta[u]](y)$
$\mathcal{B}[G_\theta[u]](y)$
B.C
$\mathcal{L}_\Omega(\theta)$
$\mathcal{L}_\partial(\theta)$
Loss $\mathcal{L}(\theta)$

Fig. 4: Presents the network connectivity graph for a physics-informed deep operator network formulated to learn the PDE solution operator across various initial and boundary data functions. The DeepONet consists of a branch network $\mathcal{N}_B : \mathbb{R}^s \to \mathbb{R}^p$ and trunk network $\mathcal{N}_T : \mathbb{R}^n \to \mathbb{R}^p$. The branch network $\mathcal{N}_B$ is trained using input functions evaluated along a fixed set of sensors $\{\xi_i\}_{i=1}^s$ while the trunk network $\mathcal{N}_T$ is trained on the spatio-temporal domain $\mathbb{R}^n$ of the PDE. Dashed connection signify non-trainable parameters. This figure is a recreation of Fig. 1 in [41].

and for $k = 1, \ldots, s-1$,

$$\left\| f - \widehat{f}^N \right\|_{W^{k,\infty}([0,1]^d)} \leq 3^d (1+\delta)\big(2(k+1)\big)^{3k} \times \max\left\{R^k, \ln^k(\beta N^{s+d+2})\right\} \times \frac{c(d,k,s,f)}{N^{s-k}} \quad (39)$$

where we define

$$\beta = \frac{k^3 2^d \sqrt{d} \max\{1, \|f\|^{\frac{1}{2}}_{W^{k,\infty}([0,1]^d)}\}}{\delta \min\{1, \sqrt{c(d,k,s,f)}\}}. \quad (40)$$

If $f \in C^s([0,1]^d)$, *then it holds that* $N_0(d) = \frac{3d}{2}$ *and*

$$c(d,k,s,f) = \max_{0 \leq \ell \leq k} \frac{1}{(s-\ell)!}\left(\frac{3s}{2}\right)^{s-\ell} |f|_{W^{s,\infty}([0,1]^d)}, \quad (41)$$

otherwise $N_0(d) = 5d^2$ *and*

$$c(d,k,s,f) = \max_{0 \leq \ell \leq k} \frac{\pi^{\frac{1}{4}}\sqrt{s}}{(s-\ell-1)!}(5d^2)^{s-\ell} |f|_{W^{s,\infty}([0,1]^d)}, \quad (42)$$

The weights of $\widehat{f}^N$ scale as

$$O\Big(c(d,k,s,f)^{-s/2}\, N^{d(d+s^2+k^2)/2}\, \big(s(s+2)\big)^{3s(s+2)}\Big). \quad (43)$$

Equation 4 provides explicit approximation bounds characterized by d, problem dimension, k, order of the approximation Sobolev space, s, order of the target Sobolev space, f, the target function, and N, the number of training points. This result pushes back against the idea that depth is a necessary feature for highly expressive networks. Additionally, the theorem extends to a wider class of activation functions including logistic functions. However, approximation error is only part of the total error and more work is required to bound the generalization error.

B. Generalization Error $\mathcal{E}_G$

Recent work has provided bounds on PINN generalization error for linear second-order PDE [47] (later extended to all linear problems [44]) and some specific cases like Navier-Stokes[48]. Of particular note, however, is the abstract framework for PINNs in which generalization error E_G is estimated by the training error E_T and number of training points for both forward[49] and inverse[50] problems. Moreover, the abstract framework leverages stability of PDE to provide conditions under which generalization error is small whenever training error is small. The formulation of the bounds on generalization error is quite intricate and requires substantial use of functional analysis. This falls outside of the scope of this study but we include this section as it is important in the development of PINNs.

VII. Future Research Areas

A. Mesh-Free Integration

It may be useful to have a DL process for integrating functions over domains in higher dimensions. This issue briefly appears in subsection IV-A where the variational form gives rise to a numerical quadrature, which becomes numerically intractable for high dimensional problems[51]. I have not encountered any literature directly addressing this but I have seen some mention that the numerical quadrature is necessary unless we get access to DL integral methods [27]. However, in solving differential equations we have already seen that a mesh-free integral technique may be plausible. This is because constraining network derivatives results in a somewhat implicit computation of an antiderivative. To be explicit, consider the function $f(x,y) = \cos(x^2 - y^2)$ for which we wish to compute the integral

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx\,dy = \int_{-1}^{1}\int_{-1}^{1} \cos(x^2-y^2)\,dx\,dy. \quad (44)$$

Let $\mathcal{N}(x,y;\theta_1)$, $\mathcal{M}(y;\theta_2)$ each be Feed-forward deep neural networks with parameters θ_1 and θ_2 respectively. The idea here is to approximate an antiderivative of f(w.r.t x) using $\mathcal{N}$ and then use $\mathcal{M}$ to approximate the antiderivative of $\mathcal{N}$(w.r.t y). Let $\{(x_i,y_i)\}_{i=1}^{N}$ be collocation points chosen from $[-1,1]\times[-1,1]$ and define the loss function

$$\mathcal{L}_{\partial x}(\theta) = \frac{1}{N}\sum_{i=1}^{N} |\mathcal{N}_x(x_i,y_i) - f(x_i,y_i)|^2, \quad (45a)$$

$$\mathcal{L}_{\partial y}(\theta) = \frac{1}{N}\sum_{i=1}^{N} |\mathcal{M}_y(y_i) - F^x(y_i)|^2, \quad (45b)$$

$$\mathcal{L}(\theta) = \mathcal{L}_{\partial x}(\theta) + \mathcal{L}_{\partial y}(\theta), \quad (45c)$$

where $\theta = \theta_1 \cup \theta_2$ and $F^x(y) = \mathcal{N}(1,y) - \mathcal{N}(-1,y)$ computes the internal integral w.r.t the x variable. The top row of Figure 5 gives the exact integrand plot along with $\mathcal{N}_x \approx f$ and the error between the two while the bottom row plots $\mathcal{M}_y$ and F^x along the $y - axis$ to track how well the internal integral is approximated. By minimizing Equation 45c we train the network $\mathcal{M}$ to approximate a mixed antiderivative and thus the integral

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dx\,dy \approx \mathcal{M}(1) - \mathcal{M}(-1). \quad (46)$$

Comparing the neural net integral with the known exact integral yields an error $\epsilon = 2.207e-4$.

This approach is fairly straight forward and easily generalizes to integrals over domains of higher dimension. Rather than using a network that is differentiated multiple times it seems to be more efficient to use a separate network to approximate each necessary antiderivative. Additionally, this approach only works on simple, rectangular domains and some care will be needed to extend this mesh-free integration to arbitrary domains. It may be worthwhile to consider a DeepONet architecture as well as it may be necessary to learn a more general antiderivative operator. We believe a careful formulation of a DL technique for integration on generalized domains could be a beneficial tool in computational mathematics. It would provide another avenue of exploration for DL techniques and could allow for a more general class of solutions to PDE.

B. *Initial Guesses for Iterative Methods*

The existence of continuous and discrete formulations for time evolution in PINNs leads to two primary approaches for implementing an iterative method accelerator. It has been shown that using an implicit RK method of extremely high precision in a discrete time model can provide a good initial guess for Newton's method [18]. The question then is about efficiently learning these initial guesses. For the discrete-time case, we must train the network at every time step and can iterate using Newton's method once the training process is done. Alternatively, in the continuous-time model the network could be trained across a time interval and results in a solution which is continuous in time. If the continuous approximation happens to provide a good initial guess, then an iterative method could call the continuous PINN at each time step to ensure convergence in a small number of iterations. It is unclear which of these approaches would be more efficient or if either is capable of outperforming a stand alone iterative approach.

C. *Persistent, General Solutions*

When creating physical models it is common to identify systems in which various physical models are coupled together to simulate complex dynamical systems. In this case, each coupled system is usually described by its own set of differential equations and the result is a system of coupled PDE. Such problems can become computationally expensive for relatively simple physical systems. The idea behind this area of research is to assess whether the DeepONet solution for parametric PDE can be leveraged to build more efficient coupled physical systems. Because PINN learn solutions to PDE, we believe that training individual systems on a variety of boundary or initial conditions, it may be possible to replace parts of a coupled PDE system with forward passes of a PINN. To illustrate this, consider the incredibly simplified magma reservoir problem. Given a magma chamber that exerts uniform pressure P on the surrounding medium let $\mathcal{F}(P) = \mathbf{u}$ represent the resulting displacements induced by pressure P. Furthermore, suppose that the change in pressure is determined by some change in volume

$$\frac{dP}{dt} = \alpha\frac{dV}{dt} \quad (47)$$

This defines a coupled system where the external problem uses a pressure to compute Displacements and, by extension, change in reservoir volume. This change in volume then causes the internal system to change the exerted pressure. So the two systems are coupled along the reservoir boundary and exchange pressure and volume data.

Using the DeepONet architecture, solution for pressure can be learned that is a function of time and $\frac{dV}{dt}$. Then the system evolves from time t_n to time t_{n+1} by

1) $\mathcal{F}(P_n) = \mathbf{u}_n$
2) $\Delta V = \int_\Gamma \mathbf{u}\cdot\mathbf{n}$
3) $P_{n+1} = \mathcal{P}(t_{n+1}, \frac{\Delta V}{\Delta t})$

where Γ is the reservoir boundary, $\mathcal{P}$ the DeepONet pressure approximation, and $\Delta t = t_{n+1} - t_n$. This way, it might be possible to use pre-trained networks in coupled systems rather than needing to solve the coupled system all at once.

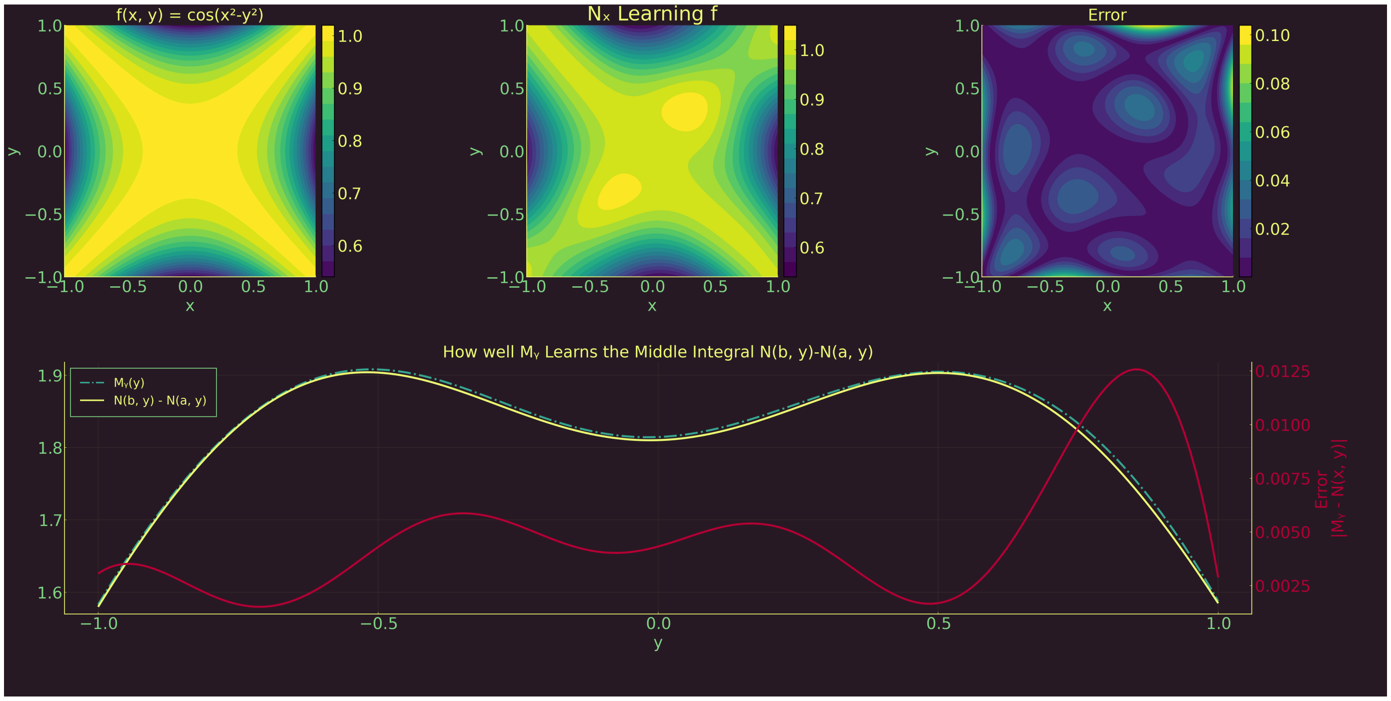

Fig. 5: For networks $\mathcal{N}$, and $\mathcal{M}$, each having 2 hidden layers with 25 neurons, approximate the integral of $\cos(x^2 - y^2)$ over the 2D domain $[-1,1] \times [-1,1]$ by minimizing Equation 45c using stochastic gradient descent. The above figure tracks intermediate training goals but method error is determined by how well the integral is approximated. This network configuration gave error $\epsilon = 2.207e-4$.

VIII. Final Remarks

DL shows incredible potential as a tool for numerical solutions to PDE and recent developments show a promising, emerging framework in the form of PINNs. In this study we see how PINNs can be bolstered using traditional numerical methods while also showing how some traditional methods can be augmented with DL to overcome computational challenges. This positions DL as a means to extend traditional methods to problems they could not previously address, and in the case of Parametric PDE we see how DL can perform where mesh-based methods cannot. While DL approaches have typically shown to perform well empirically, the recent progress in PINN approximation theory is beginning to legitimize PINNs as PDE solutions. Being able to predictably bound the error of a method is critical in developing numerical solutions. As PINNs become more reliable tools, we may apply them in unique ways to expand the reach of physical models.

Appendix A
Gauss-Legendre Runge-Kutta Method

Let $t > 0$ and consider the ordinary differential equation

$$\frac{d}{dt}u(t) = f(t,u) \tag{A.1}$$

where $u : \mathbb{R} \to \mathbb{R}^d$ and $f : \mathbb{R}^{d+1} \to \mathbb{R}^d$ are vector-valued functions. The Gauss-Legendre quadrature approximates the definite integral of a function over the symmetric interval $[-1,1]$ on n sample points

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{A.2}$$

where $\{w_i\}_{i=1}^n$ are the quadrature weights, x_i the roots of the n^{th} degree Legendre polynomial, $P_n(x)$, and $\{k_i\}_{i=1}^n$ give the sample velocities $k_i = f(x_i)$ for $1 \le i \le n$. For an ODE of the form Equation A.1 the Runge-Kutta method approximates a single time step $t_\eta \to t_{\eta+1}$ using a sequence of q stages by $u^{\eta+c_i} = u(t_\eta + \Delta t c_i)$, $i = \{1, 2, \dots, q\}$

$$u^{\eta+c_i} = u^\eta + \Delta t \sum_{j=1}^{q} A_{ij} f(t_\eta + \Delta t c_j, u^{\eta+c_j}) \tag{A.3a}$$

$$u^{\eta+\Delta t} = u^\eta + \Delta t \sum_{i=1}^{q} b_i f(t_\eta + \Delta t c_i, u^{\eta+c_i}) \tag{A.3b}$$

where $\mathbf{A} \in \mathbb{R}^{q\times q}$ is the Runge-Kutta coefficient matrix, $\mathbf{b} \in \mathbb{R}^q$ the stage combination coefficients and $\mathbf{c} \in \mathbb{R}^q$ the vector of abscissa obtained by shifting the Legendre polynomial roots from $[-1,1] \to [t_\eta, t_{\eta+1}]$. This gives the general form of RK methods and selection of $\mathbf{A}$ and $\mathbf{b}$ will determine a precise numerical scheme. The method will typically be an implicit method except if $\mathbf{A}$ satisfies $A_{ij} = 0$ for $i \le j$ in which case the result is an explicit method. The Gauss-Legendre Runge-Kutta method defines an implicit method where $\mathbf{A}, \mathbf{b}, \mathbf{c}$ are determined using Legendre polynomials. The choice of quadrature weights and nodes in Equation A.2 gives the GL quadrature the capacity to perform an exact integration of polynomials with degree $2n-1$. While this approach offers a very nice approximation with truncation error $O(\Delta t^{2q})$ when q stages are used, it requires one to solve a system of algebraic equations at each time step which becomes costly. The exact point at which the methods shows diminishing returns will vary between problems but it is common to use a 3-stage formulation [52].

To properly construct the $q-$stage GLRK method, we need the roots of $P_q(x)$ along with the derivative $P_q'(x)$. Though the roots can be found using a root finding method like the Netwon method, there has been a recent development that provides explicit asymptotic forms of the GL nodes and weights which are accurate up to double-precision machine ϵ for $q \geq 21$ [53], [16], [17]. If $\{x_i\}_{i=1}^{q}$ are the roots of the Legendre polynomial (with $P_q(x)$ normalized to so that $P_q(1) = 1$), then the Gauss-Legendre quadrature weights are given by

$$w_i = \frac{2}{(1 - x_i^2)[P_q'(x_i)]^2}. \tag{A.4}$$

Earlier this year, it was shown that by considering a transformation of the integration domain from $[-1, 1] \to [t_n, t_{n+1}]$ by $\xi = \frac{\Delta t}{2}(t - \bar{t}_m)$ for step size $\Delta t \equiv t_{n+1} - t_n$ and mid-point $\bar{t}_n \equiv \frac{1}{2}(t_{n+1} + t_n)$ we may use the the node points $\{\xi_i\}_{i=0}^{q-1}$ (roots of the transformed Legendre polynomial) to compute

$$A_{ij} = \frac{w_j}{2}\left(1 + \sum_{s=0}^{q-1} P_s(\xi_j)\frac{P_{s+1}(\xi_i) - P_{s-1}(\xi_i)}{2}\right), \tag{A.5a}$$

$$b_i \equiv \frac{w_i}{2}, \tag{A.5b}$$

demonstrating how the Gauss-Legendre Runge-Kutta method can be constructed to arbitrarily high order without much difficulty [18]. In Equation A.5 we use $P_{-1}(\xi) \equiv 1$. With the RK parameters properly defined, all that is left is to solve the implicit method given by using said parameters in Equation A.3.

The choice of an implicit method means Equation A.3 will be solved using an iterative root-finding method like the Newton–Raphson method. To do this, it helps to create stacks of vectors and represent the problem as a matrix-vector problem. We then define the vector stacks

$$\mathcal{U}^{\eta} := \bigoplus_{i=1}^{q} u^{\eta}, \tag{A.6a}$$

$$\mathcal{U}^{\eta+c} := \bigoplus_{i=1}^{q} u^{\eta+c_i}, \tag{A.6b}$$

$$F(\mathcal{U}^{\eta+c}) := \bigoplus_{i=1}^{q} \Delta t f\big(t_\eta + c_i \Delta t, u^{\eta+c_i}\big), \tag{A.6c}$$

allowing Equation A.3a to be represented as $\mathcal{D}\big(\mathcal{U}^{\eta+c}\big) = 0$ with

$$\mathcal{D}\big(\mathcal{U}^{\eta+c}\big) = \mathcal{U}^{\eta} - \mathcal{U}^{\eta+c} + \Delta t\big(A \otimes I\big)F\big(\mathcal{U}^{\eta+c}\big). \tag{A.7}$$

Then, the k^{th} step of the Newton method applied to Equation A.3 involves the following matrix problem and iteration update

$$\mathcal{D}'\big(\mathcal{U}_k^{\eta+c}\big)\Delta\mathcal{U}_k^{\eta+c} = \mathcal{D}\big(\mathcal{U}_k^{\eta+c}\big), \tag{A.8}$$

$$\mathcal{U}_{k+1}^{\eta+c} = \mathcal{U}_k^{\eta+c} + \Delta\mathcal{U}_k^{\eta+c}, \tag{A.9}$$

where the derivative $\mathcal{D}'$ is given by

$$\mathcal{D}'\big(\mathcal{U}^{\eta+c}\big) = \Delta t\big(A \otimes I\big)F'\big(\mathcal{U}^{\eta+c}\big) - I, \tag{A.10}$$

and the derivative of F is the block-diagonal matrix

$$F'\big(\mathcal{U}^{\eta+c}\big) := \bigoplus_{i=1}^{q} \mathcal{J}_f\big(u^{\eta+c_i}\big) \tag{A.11}$$

consisting the the Jacobian of f evaluated at different stages. With $f : \mathbb{R}^{d+1} \to \mathbb{R}^d$ and the choice of q stages in the RK method, the resulting system of algebraic equations from Equation A.8 is $qd \times qd$. Thus, a solution (probably via LU factorization) will need $\mathcal{O}\big((qd)^3\big)$ operations. Luckily it is possible to re-use the LU factorization across the Newton iterations but the system will need a new factorization computed for each time step [54], [55].

Appendix B
Special Considerations

There are some network limitations that arise when approximating solutions for differential equations. A popular activation function choice becomes unviable in this problem domain. The Rectified Linear Unit (ReLU) can be used in bounded-width networks to create universal approximators [8], [10] (a fact that is generalized in [3] but requires a deeper network for arbitrary activation functions) and approximation error for such network can be bounded in terms of the network width and depth [56]. Now, considering a simple neural network $\mathcal{N}(x; W, b) = \phi.(Wx + b)$ we can see that its first two input derivatives are

$$\frac{\partial}{\partial x}\mathcal{N}(x; , W, b) = \phi'.(Wx + b) * . W, \tag{B.1a}$$

$$\frac{\partial^2}{\partial x^2}\mathcal{N}(x; , W, b) = \phi''.(Wx + b) * . W \; * . W. \tag{B.1b}$$

That is derivatives of the network require derivatives of the activation function. This generalizes to derivatives of order k requiring the order k derivatives for the activation function [11]. Because physics-based neural net methods utilize network derivatives in construction of the loss function, ReLU activation function is not generally usable. To approximate a PDE involving derivatives of order k, the activation function must be differentiable of order $k + 1$ to account for the additional derivative taken during backpropogation.

While both of these are susceptible to the vanishing gradient problem, tanh tends to be more stable due to a larger gradient range. While we want an activation function with sufficient regularity we would also like a network that can be trained efficiently. Rather than using a fixed activation function we can modify the network to learn activation functions for us. One approach to this is to learn coefficients that approximate an optimal activation function with respect to a chosen basis

[57]. While such an approach does learn the activation function it does require you to choose a basis for the approximate activation. Instead we will focus on a method that works by learning slopes of the activation functions and which have been applied to PINN [58], [59], [60].

A. *Adaptive Activation Functions*

Begin by modify an activation function ϕ with trainable slope $a \in \mathbb{R}$ and scaling parameter $n \geq 1$ to get a new activation function,

$$\tilde{\phi}(x) = \phi(nax), \tag{B.2}$$

where a is now included in the set of trainable parameters. This approach can improve the efficiency of DNN [58] along with improving the robustness, and accuracy of neural net approximations for PDE [59]. Furthermore, adaptive coefficients can be included at the global, layer, and neuron levels allowing for finer tuning of the network during training. Equation B.2 shows a globally-adaptive activation as it uses a single shared value of a for each layer in the network.

1) Layer-Wise Locally Adaptive Activation: You could also define a scalar for each hidden layer $\{a_i\}_{i=1}^{L-1}$ where $a_i \in \mathbb{R}$ for each i and specify activation function for layer k as

$$\tilde{\phi}_k(x) = \phi(n\, a_k\, x) \quad \text{for } k = \{1, 2, \ldots, L-1\}. \tag{B.3}$$

Since the set of training parameters is now $\theta \cup \{a_i\}_{i=1}^{L-1}$ the number of parameters needing optimization is $|\theta| + L - 1$.

2) Neuron-Wise Locally Adaptive Activation: This time each neuron is given a trainable weight. So let $\{a_i\}_{i=1}^{L-1}$ where $a_i \in \mathbb{R}^m$ and specify activation function for layer k by

$$\tilde{\phi}_k(x) = \phi(n\, a_k\, * . x) \quad \text{for } k = \{1, 2, \ldots, L-1\}. \tag{B.4}$$

Since each layer comes with m hidden neurons we get $|\theta| + m(L-1)$ trainable network parameters.

While these adaptive activation functions do improve training efficiency, [60] includes an extra term in the loss function which can contribute to the loss gradient without vanishing. The slope recovery term $\mathcal{S}$ is defined as

$$\mathcal{S}(a) := \begin{cases} \frac{L-1}{\sum_{i=1}^{L-1} \exp(a_i)} & \text{if } a_i \in \mathbb{R}, \\ \frac{L-1}{\sum_{i=1}^{L-1} \exp\left(\frac{\sum_{k=1}^{m} (a_i)_k}{m}\right)} & \text{if } a_i \in \mathbb{R}^m, \end{cases} \tag{B.5}$$

and the appended term for the loss function is

$$SRT = W_a \mathcal{S}(a) \tag{B.6}$$

with trainable weight W_a. It was observed that networks utilizing adaptive activation functions exhibited more rapid decay of the loss function over networks using fixed activation functions [58]. Moreover, locally-adaptive networks outperformed both fixed-activation and global-adaptive networks[60]

References

[1] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[2] 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," *Journal of Computational physics*, vol. 378, pp. 686–707, 2019.

[3] P. Kidger and T. Lyons, "Universal approximation with deep narrow networks," in *Conference on learning theory*. PMLR, 2020, pp. 2306–2327.

[4] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[5] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[6] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[7] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

[8] B. Hanin and M. Sellke, "Approximating continuous functions by relu nets of minimal width," *arXiv preprint arXiv:1710.11278*, 2017.

[9] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width," *Advances in neural information processing systems*, vol. 30, 2017.

[10] B. Hanin, "Universal function approximation by deep neural nets with bounded width and relu activations," *Mathematics*, vol. 7, no. 10, p. 992, 2019.

[11] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE transactions on neural networks*, vol. 9, no. 5, pp. 987–1000, 1998.

[12] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou, "Neural-network methods for boundary value problems with irregular boundaries," *IEEE Transactions on Neural Networks*, vol. 11, no. 5, pp. 1041–1049, 2000.

[13] J. Berg and K. Nyström, "A unified deep artificial neural network approach to partial differential equations in complex geometries," *Neurocomputing*, vol. 317, pp. 28–41, 2018.

[14] S. Kollmannsberger, D. D'Angella, M. Jokeit, L. Herrmann *et al.*, *Deep Learning in Computational Mechanics*. Springer, 2021.

[15] S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific machine learning through physics-informed neural networks: Where we are and what's next," *arXiv preprint arXiv:2201.05624*, 2022.

[16] I. Bogaert, "Iteration-free computation of gauss–legendre quadrature nodes and weights," *SIAM Journal on Scientific Computing*, vol. 36, no. 3, pp. A1008–A1026, 2014.

[17] I. Bogaert, B. Michiels, and J. Fostier, "O(1) computation of legendre polynomials and gauss–legendre nodes and weights for parallel computing," *SIAM Journal on Scientific Computing*, vol. 34, no. 3, pp. C83–C101, 2012.

[18] D. W. Crews, "Arbitrarily high order implicit ode integration by correcting a neural network approximation with newton's method," *arXiv preprint arXiv:2203.00147*, 2022.

[19] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021.

[20] B. Wang, W. Zhang, and W. Cai, "Multi-scale deep neural network (mscalednn) methods for oscillatory stokes flows in complex domains," *arXiv preprint arXiv:2009.12729*, 2020.

[21] L. Sun, H. Gao, S. Pan, and J.-X. Wang, "Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data," *Computer Methods in Applied Mechanics and Engineering*, vol. 361, p. 112732, 2020.

[22] X. Jin, S. Cai, H. Li, and G. E. Karniadakis, "Nsfnets (navier-stokes flow nets): Physics-informed neural networks for the incompressible navier-stokes equations," *Journal of Computational Physics*, vol. 426, p. 109951, 2021.

[23] S. Cai, Z. Wang, S. Wang, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks for heat transfer problems," *Journal of Heat Transfer*, vol. 143, no. 6, 2021.

[24] E. Haghighat, M. Raissi, A. Moure, H. Gomez, and R. Juanes, "A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 379, p. 113741, 2021.

[25] S. Goswami, C. Anitescu, S. Chakraborty, and T. Rabczuk, "Transfer learning enhanced physics informed neural network for phase-field modeling of fracture," *Theoretical and Applied Fracture Mechanics*, vol. 106, p. 102447, 2020.

[26] Z. Mao, A. D. Jagtap, and G. E. Karniadakis, "Physics-informed neural networks for high-speed flows," *Computer Methods in Applied Mechanics and Engineering*, vol. 360, p. 112789, 2020.

[27] E. Kharazmi, Z. Zhang, and G. E. Karniadakis, "Variational physics-informed neural networks for solving partial differential equations," *arXiv preprint arXiv:1912.00873*, 2019.

[28] A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis, "Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 365, p. 113028, 2020.

[29] A. A. Ramabathiran and P. Ramachandran, "Spinn: Sparse, physics-based, and partially interpretable neural networks for pdes," *Journal of Computational Physics*, vol. 445, p. 110600, 2021.

[30] P. G. Ciarlet, *The finite element method for elliptic problems*. Siam, 2002, vol. 40.

[31] A. Ern and J.-L. Guermond, *Theory and practice of finite elements*. Springer, 2004, vol. 159.

[32] S. C. Brenner, L. R. Scott, and L. R. Scott, *The mathematical theory of finite element methods*. Springer, 2008, vol. 3.

[33] S. Larsson and V. Thomée, *Partial differential equations with numerical methods*. Springer, 2003, vol. 45.

[34] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *Journal of Computational Physics*, vol. 375, pp. 1339–1364, 2018.

[35] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, "Fourier neural operator for parametric partial differential equations," *arXiv preprint arXiv:2010.08895*, 2020.

[36] ——, "Neural operator: Graph kernel network for partial differential equations," *arXiv preprint arXiv:2003.03485*, 2020.

[37] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, A. Stuart, K. Bhattacharya, and A. Anandkumar, "Multipole graph neural operator for parametric partial differential equations," *Advances in Neural Information Processing Systems*, vol. 33, pp. 6755–6766, 2020.

[38] L. Lu, P. Jin, and G. E. Karniadakis, "Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators," *arXiv preprint arXiv:1910.03193*, 2019.

[39] T. Chen and H. Chen, "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems," *IEEE Transactions on Neural Networks*, vol. 6, no. 4, pp. 911–917, 1995.

[40] A. D. Back and T. Chen, "Universal approximation of multiple nonlinear operators by neural networks," *Neural Computation*, vol. 14, no. 11, pp. 2561–2566, 2002.

[41] S. Wang, H. Wang, and P. Perdikaris, "Learning the solution operator of parametric partial differential equations with physics-informed deeponets," *Science advances*, vol. 7, no. 40, p. eabi8605, 2021.

[42] G. Kutyniok, "The mathematics of artificial intelligence," *arXiv preprint arXiv:2203.08890*, 2022.

[43] D. Elbrächter, D. Perekrestenko, P. Grohs, and H. Bölcskei, "Deep neural network approximation theory," *IEEE Transactions on Information Theory*, vol. 67, no. 5, pp. 2581–2623, 2021.

[44] Y. Shin, Z. Zhang, and G. E. Karniadakis, "Error estimates of residual minimization using neural networks for linear pdes," *arXiv preprint arXiv:2010.08019*, 2020.

[45] T. De Ryck, S. Lanthaler, and S. Mishra, "On the approximation of functions by tanh neural networks," *Neural Networks*, vol. 143, pp. 732–750, 2021.

[46] L. Lu, Y. Su, and G. E. Karniadakis, "Collapse of deep and narrow neural nets," *arXiv preprint arXiv:1808.04947*, 2018.

[47] Y. Shin, J. Darbon, and G. E. Karniadakis, "On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type pdes," *arXiv preprint arXiv:2004.01806*, 2020.

[48] T. De Ryck, A. D. Jagtap, and S. Mishra, "Error estimates for physics informed neural networks approximating the navier-stokes equations," *arXiv preprint arXiv:2203.09346*, 2022.

[49] S. Mishra and R. Molinaro, "Estimates on the generalization error of physics-informed neural networks for approximating pdes," *IMA Journal of Numerical Analysis*, 2022.

[50] ——, "Estimates on the generalization error of physics-informed neural networks for approximating a class of inverse problems for pdes," *IMA Journal of Numerical Analysis*, vol. 42, no. 2, pp. 981–1022, 2022.

[51] A. Hinrichs, E. Novak, M. Ullrich, and H. Woźniakowski, "The curse of dimensionality for numerical integration of smooth functions," *Mathematics of Computation*, vol. 83, no. 290, pp. 2853–2863, 2014.

[52] A. Iserles, *A first course in the numerical analysis of differential equations*. Cambridge university press, 2009, no. 44.

[53] N. Hale and A. Townsend, "Fast and accurate computation of gauss–legendre and gauss–jacobi quadrature nodes and weights," *SIAM Journal on Scientific Computing*, vol. 35, no. 2, pp. A652–A674, 2013.

[54] G. Cooper, "On the implementation of singly implicit runge-kutta methods," *Mathematics of computation*, vol. 57, no. 196, pp. 663–672, 1991.

[55] R. J. LeVeque, *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[56] D. Yarotsky, "Error bounds for approximations with deep relu networks," *Neural Networks*, vol. 94, pp. 103–114, 2017.

[57] M. Goyal, R. Goyal, and B. Lall, "Learning activation functions: A new paradigm for understanding neural networks," *arXiv preprint arXiv:1906.09529*, 2019.

[58] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," *arXiv preprint arXiv:1412.6830*, 2014.

[59] A. D. Jagtap, K. Kawaguchi, and G. E. Karniadakis, "Adaptive activation functions accelerate convergence in deep and physics-informed neural networks," *Journal of Computational Physics*, vol. 404, p. 109136, 2020.

[60] A. D. Jagtap, K. Kawaguchi, and G. Em Karniadakis, "Locally adaptive activation functions with slope recovery for deep and physics-informed neural networks," *Proceedings of the Royal Society A*, vol. 476, no. 2239, p. 20200334, 2020.