Discovering PDEs with Deep Learning

Jack Arnott

January 2025

Introduction

Owing to the recent technological advancements and the abundance of labelled data available online, deep learning is an exciting and evolving mathematical field. *PDE discovery*, a subfield of both system identification and physics-informed machine learning, will be the subject of this paper. At a very high level, the field aims to find governing equations from datasets using machine learning techniques. This is relevant to modern scientific inquiry as classical methods for finding PDEs, such as use of conservation laws, have failed for more complex systems. These include drug discovery, financial markets, and power grids [1]. It should be noted that PDEs are not unique in regard to this methodology. Indeed, similar approaches have been taken to find Green's functions [2].

In this paper, we shall use the subject of PDE discovery to discuss some important concepts in the theories of deep learning. First, we shall ask why we can use deep learning to find PDEs from data. We then explore a new type of neural network called a Rational Neural Network and establish their theoretical advantage over the ever popular ReLU activation function. Afterwards, we take a more detailed look at a recent PDE discovery algorithm, called `PDE-LEARN` which uses rational neural networks. We finish by performing some numerical simulations with this algorithm, bringing together all the concepts discussed.

Why Deep Learning?

Before we take a look at some more specific examples, we shall first take some time to ask *why* we can use neural networks to find governing equations from data. While it can be tempting to throw as much data as possible at a network until we see a favourable result, as mathematicians, we require more rigour.

In 2022, Stephany et al. introduced the PDE discovery algorithm `PDE-READ` [1]. In the introduction of this paper, the authors briefly outline an argument for why Neural Networks can be used to find PDEs from data. We will now explore this argument in more detail, concluding that we can use deep learning to find governing equations. First, we shall introduce some definitions.

Definition 1 (Neural Network [3])**.** *A fully-connected neural network of depth N with weight matrix $W_i \in \mathbb{R}^{n_{i+1} \times n_i}$, bias $b_i \in \mathbb{R}^{n_i}$, trainable parameters $\theta = \{W_i, b_i\}_{i=1}^{N}$ and input $h_0 \in \mathbb{R}^{n_0}$ is a series of N compositions of functions of the form*

$$h_{i+1} = \phi_i(W_i h_i + b_i)$$

where ϕ is a nonlinear activation function.

As we shall see, the form which ϕ takes influences the network's approximation ability.

Definition 2 (L^p-space [4])**.** *Let $(X, \mathcal{A}, \mu)$ be a measure space and $1 \leq p < \infty$. The space $L^p(X)$ consists of measurable functions such that*

$$\int_X |f|^p d\mu < \infty$$

We define the $L^p(X)$ norm of f as

$$\|f\|_{L^p} \coloneqq \left(\int |f|^p d\mu \right)^{\frac{1}{p}}$$

We include these definitions for completeness, though without an overly measure-theoretic approach.

Definition 3 (L^p Dense [5])**.** *Let μ be an arbitrary finite measure and let $S \subset L^p(\mu)$. We say S is dense on L^p if for an arbitrary function $f \in L^p$ and $\epsilon > 0$, there exists $g \in S$ such that*

$$\|f - g\|_{L^p} < \epsilon$$

In other words, if a set of functions A is dense on some other set of functions B, we can approximate any function in B with a linear combination of functions from A to within ϵ-accuracy in the L^p norm. The definition of density for continuous functions is very similar to that of Definition 3.

In his 1989 paper [6], Cybenko used the Hahn-Banach Theorem to prove that shallow ($N = 1$ layers) neural networks with a sigmoidal activation function are dense in $C([0, 1]^n)$, meaning we can use a neural network to approximate any continuous function in $C([0, 1]^n)$. This result was expanded on in 1991 by Hornik [5], who proved two important theorems:

Theorem 1 (Hornik)**.** *Let H be the set of neural networks with bounded nonconstant activation functions. Then for any finite measure μ, H is dense in $L^p(\mu)$, provided a sufficient number of hidden units is available.*

Theorem 2 (Hornik)**.** *Let G be the set of neural networks with continuous, bounded, non-constant activation functions and let $C^m(\mathbb{R}^k)$ denote the space of functions which, along with their partial derivatives $D^\alpha f$ of order $|\alpha| \leq k$, are continuous on $\mathbb{R}^k$. Then, G is dense in $C^m(\mathbb{R}^k)$ for all compactly supported subsets[1] in R^k, provided a sufficient number of hidden units is available.*

Theorem 1 means we can use neural networks to approximate any function in L^p. Theorem 2 means we can approximate any continuous function and it's derivatives with a neural network. Leshno et al. [7] further generalised these results:

[1]Compact support means the function is non-zero within a finite interval on the domain and zero elsewhere.

Theorem 3 (Leshno et al.)**.** *Shallow Neural networks with non-polynomial activation functions are dense in the set of compactly supported continuous functions.*

These results are generally regarded as the *universal approximation theorem*. It means that the only requirement for a single-layer neural network to be able to approximate a function is for the activation function to not be polynomial. It is important to note that these results are purely theoretical and in no way constructive. We discuss methodology later. We now state a standard result from measure theory.

Theorem 4. *Compactly supported continuous functions are dense in L^p.*

Theorem 4 bridges approximation theory and PDE theory, demonstrating how neural networks can approximate PDE solutions. As continuous functions are dense in L^p and neural networks with non-polynomial activation functions are dense in the set of continuous functions, it then follows by topological transitivity that neural networks with non-polynomial activation functions are dense in L^p.

This result gives us a good indication that PDE discovery can be used to find PDEs from data using deep learning. If, as claimed by the authors [1], both classical and weak PDE solutions live in L^p, it would suggest we can use neural networks to approximate any PDE solution. We shall devote the remainder of this section to exploring this claim.

First, consider the classical solution u of an n^{th}-dimensional PDE of order m defined on some open domain $\Omega \subset \mathbb{R}^n$. By definition, $u \in C^m(\Omega) \cap C^{m_b}(\overline{\Omega})$ where $m_b \leq m - 1$ [8]. As $u \in C^{m_b}(\overline{\Omega})$, u is uniformly continuous on the closed domain $\overline{\Omega}$ and therefore is bounded on $\overline{\Omega}$. Consequently $|u|^p$ for $1 \leq p < \infty$ will also be bounded on $\overline{\Omega}$ and thus the integral $\int_{\overline{\Omega}} |u(x)|^p dx$ is finite, meaning $u \in L^p$. However, the process is more involved for weak solutions.

Consider first elliptic equations. We define a Hölder space $C^{0,\alpha}(\Omega)$ as the space of all functions which are Hölder continuous on some bounded Ω for some α [9], that is

$$C^{0,\alpha}(\Omega) \coloneqq \left\{ u : \Omega \to \mathbb{R} : \sup_{\substack{x,y \in \Omega \\ x \neq y}} \frac{|u(x) - u(y)|}{|x - y|^\alpha} < \infty; 0 < \alpha \leq 1 \right\}$$ [2]

[2]The '0' in the superscript of the Hölder space $C^{0,\alpha}(\Omega)$ denotes that only u is Hölder continuous;

We also need to introduce the Sobolev space $\mathcal{W}^{n,p}$, the space of functions with weak derivatives in L^p, that is

$$\mathcal{W}^{n,p}(\Omega) \coloneqq \{u \in L^p(\Omega) : D^a u \in L^p(\Omega), |a| \leq n\}$$

where $D^a u$ denotes a weak derivative of order a. By definition, a weak solution u to second-order elliptic boundary value problems is contained in $\mathcal{W}^{1,2}(\Omega)$. We can then use De Giorgi's regularity theorem ([10], Theorem 3.2), which states that as $u \in \mathcal{W}^{1,2}(\Omega)$, u is also a member of the Hölder space $C^{0,\alpha}(\Omega')$, for any subdomain $\Omega' \subset \Omega$ compactly contained in Ω. As Hölder continuity is stronger than uniform continuity, we can thus say $u \in C(\overline{\Omega})$ and our argument proceeds as in the case of the classical solution.

John Nash proved a similar result for parabolic equations in 1958 [11]. Hyperbolic equations require more extensive analysis, and their behaviour is not always as straightforward as that of elliptic or parabolic equations. However, in practice, PDE discovery algorithms do not struggle to find hyperbolic equations. This analysis indicates that neural networks can be effective tools for finding solutions to PDEs.

Depth

The above results were all proved for a single-layer network. We now discuss deep networks, which exhibit some different behaviours. The definition of what makes a network 'deep' is debated. Some claim a deep network has $N > 1$ hidden layers [12], whilst others claim that $N \gg 1$ for depth [13]. We shall use the first definition of depth as this will be more appropriate for the subsequent discussions. In different contexts, such as in industry, there are merits to a distinction.

The theoretical work of Telgarsky [14] compared deep networks to shallow ones. He showed that the number of parameters required to exactly approximate a given function scales linearly with depth, but exponentially with width ([14], Theorem 1.1). This work was extended by Yarotsky [15], who proved a form of the universal

none of its derivatives are, necessarily.

approximation theorem for a ReLU network on the unit ball

$$F_{n,d} \coloneqq \{f \in \mathcal{W}^{n,\infty}([0,1]^d) : \|f\|_{\mathcal{W}^{n,\infty}} \leq 1\},$$

where $\|f\|_{\mathcal{W}^{n,\infty}} \coloneqq \max_{|a|\leq n} \operatorname{ess\,sup}_{x\in[0,1]^d} |D^a f|$.

Theorem 5 (Yarotsky)**.** *Let $\epsilon \in (0,1)$. For any n and d, there exists a ReLU architecture that*

1. *is dense on $F_{n,d}$*
2. *has depth at most $C(\ln(1/\epsilon)+1)$ with at most $c\epsilon^{-d/n}(\ln(1/\epsilon)+1)$ weights*

The difference between a *network* and a *network architecture* is that when talking about networks, we talk about them for some arbitrary number of weights. Speaking about a network architecture implies that it is defined with some rule on the number of weights.

More recently, Kidger et al. considered the universal approximation result for networks with an arbitrary depth but bounded width, dubbed *narrow* neural networks [16]. They considered this as the 'dual' of the classical approximation result, as they extend this result to a bounded depth with arbitrary width by using the surplus hidden layers to approximate the identity function. Notably, Kidger et al. showed for the dual problem on the space of continuous functions, all activation functions are permissible choices for density, including polynomial functions. This contrasts to the single layer results which stipulate the activation must be strictly non-polynomial. These results proved that deep networks can approximate functions more efficiently than shallow ones, with fewer restrictions on the architecture.

Rational Neural Networks

We shall now explore Rational Neural Networks. These networks were introduced by Boulle, Nakatsukasa and Townsend [17] and are used in several PDE-Discovery algorithms [1, 18, 19].

A Rational function $f(x)$ is a function of the form

$$f(x) = \frac{\sum_{i=1}^{P} a_i x^i}{\sum_{j=1}^{Q} b_j x^j}.$$

We say f is of type (P, Q) and degree $\max(P, Q)$. A Rational Neural Network is a neural network with rational activation functions, although these do not necessarily have to be the same function between layers. In their paper, Boulle et al. establish two interesting results: (1) Rational Neural Networks are themselves rational functions and (2) Rational Neural Networks require less nodes and a shallower depth to get the same results as ReLU activation functions (ReLU: $\phi(x) = \max(0, x)$). ReLU networks are of interest as they are a very popular choice of activation function used today.

Rational functions also circumnavigate what has been dubbed the 'vanishing gradient problem' where the gradients of traditional activation functions such as $\phi(x) = \tanh(x)$ or $\phi(x) = (1+e^{-x})^{-1}$ become very small for large inputs. Calculating gradients is an important part of gradient descent, a common training algorithm in deep learning. Bengio et al. discuss in [20] how vanishing gradients cause significant problems for training. The vanishing gradient problem may be solved by using polynomial activation functions, as this class of functions generally have well behaved derivatives. However, Boulle et al. note that these functions can lead to numerical instability [17].

Boulle et al. claim that Rational Neural Networks are themselves rational functions. We now provide an original proof of this claim for the case of type $(3, 2)$ rational functions, which are used by Boulle et al. in their experiments and in `PDE-LEARN` discussed later.

Let $\mathcal{R}_{(P,Q)}$ denote the set of all rational functions of type (P, Q), that is

$$\mathcal{R}_{(P,Q)} \coloneqq \left\{ f : \mathbb{R}^n \to \mathbb{R}^n : f(x) = \frac{\sum_{i=1}^{P} a_i x^i}{\sum_{j=1}^{Q} b_j x^j}; P, Q \in \mathbb{N} \right\}$$

Theorem 6. *A neural network of depth N composed of rational activation functions*

ϕ of type $(3,2)$ is a rational function of type $(3^N, 2 \cdot 3^{N-1})$, and thus degree 3^N.

Proof. We proceed by induction. Consider the base case of $N = 1$, with input $x_0 \in \mathbb{R}^n$

$$x_1 = \phi(W_1^T x_0 + b_1) = \phi(x_0^{'}) = \frac{P(x_0^{'})}{Q(x_0^{'})} \in \mathcal{R}_{(3,2)},$$

As $N = 1$ means we have a single-hidden-layer network, x_1 is our output and we are done. We now state our inductive hypothesis for $N = k$

$$x_k = \phi(W_k^T x_{k-1} + b_k) \in \mathcal{R}_{(3^k, 2 \cdot 3^{k-1})}.$$

Now performing the inductive step for $N = k+1$ is simple and our conclusion follows

$$x_{k+1} = \phi(W_{k+1}^T x_k + b_{k+1}) = \phi(x_k^{'}) = \frac{P(x_k^{'})}{Q(x_k^{'})} \in \mathcal{R}_{(3^{k+1}, 2 \cdot 3^k)}$$

as $x_k^{'} \in \mathcal{R}_{(3^k, 2 \cdot 3^{k-1})}$. □

This means Rational Neural Networks benefit from both the low cost of using low-degree functions for the individual layers of the network, as well as the high accuracy which high degree rational functions provide (see [21] for an interesting exposition of why the advent of the *AAA-Algorithm* [22] makes rational functions more appealing for approximation). In the field of deep learning, where computational cost is a important factor, this gives Rational Neural Networks an edge over traditional activation functions. We shall now consider the parameter costs of these activation functions.

Theorem 7 (Boulle et al.)**.** *Let $0 < \epsilon < 1$ and $\| \cdot \|_1$ be the vector 1-norm.*

1. *Let $H : [-1,1]^d \to [-1,1]$ be a rational neural network with N layers and at most k nodes per layer. Each node computes $x \mapsto r(a^T x + b)$ where $r : [-1,1]^d \to [-1,1]$ is rational funciton with Lipschitz constant L and $\|a\|_1 + |b| \leq 1$; a, b and r can be distinct across nodes. Then there exists a ReLU network F :*

$[-1,1]^d \to [-1,1]$ *of size* [3] $\mathcal{O}\left(kN\log\left(\frac{NL^N}{\epsilon}\right)^3\right)$ *such that* $\max_{x\in[-1,1]^d}|H(x)-F(x)|\leq\epsilon$.

2. *Let* $F:[-1,1]^d\to[-1,1]$ *be a ReLU network with* N *layers and at most* k *nodes per layer. Each node computes* $x\mapsto ReLU(a^Tx+b)$ *and* $\|a\|_1+|b|\leq 1$*;* a, b *and* r *can be distinct across nodes. Then there exists a rational network* $H:[-1,1]^d\to[-1,1]$ *of size* $\mathcal{O}\left(kN\log\left(\log\left(\frac{N}{\epsilon}\right)\right)\right)$ *such that* $\max_{x\in[-1,1]^d}|F(x)-H(x)|\leq\epsilon$.

This theorem means that given a rational neural network with M layers and at most k nodes per layer, the ReLU network required to approximate this network to within ϵ-accuracy is of size $\mathcal{O}\left(kN\log\left(NL^N/\epsilon\right)^3\right)$. In the opposite case, the rational network required to approximate a ReLU network of M layers and at most k nodes per layer to within ϵ-accuracy is of size $\mathcal{O}\left(kN\log\left(\log\left(\frac{N}{\epsilon}\right)\right)\right)$. A graphical argument shows that the parameter requirements for using ReLU networks in approximation is greater than that of rational networks, provided that $L\geq 1$. This gives rational networks an advantage over ReLU networks, as the cost of training will be lower for the same results. The proof of this theorem is interesting and we shall discuss it now. Before we begin, we require the following lemma

Lemma 8 ([17], Lemma 2)**.** *If* $H:[-1,1]^d\to[-1,1]$ *is a rational function, there exists a ReLU network* $F:[-1,1]^d\to[-1,1]$ *with* $\mathcal{O}(\log\left(\frac{1}{\epsilon}\right)^3)$ *parameters such that* $\max_{x\in[-1,1]^d}|F(x)-H(x)|\leq\epsilon$*, where* $0<\epsilon<1$.

In other words, we can approximate any rational function with a ReLU network of size $\mathcal{O}(\log\left(\frac{1}{\epsilon}\right)^3)$. We are now ready to tackle the proof of Theorem 7.

Proof. Due to space constraints, we only prove (1), the statement about ReLU networks. The proof for (2) is much shorter and builds upon a result proved by Telegarsky [23]. We begin by defining a subnetwork and the error term. We define the subnetwork G which is made up of all the layers of H up to and including the J^{th} layer, where $1\leq J<N$. We then define G_{ReLU}, which is the network we obtain by

[3]The size of a network refers to the number of trainable parameters of the network.

replacing each rational function $r_{i,j}$[4] in G with a ReLU approximation, denoted $f_{r_{i,j}}$ such that $\max_{x\in[-1,1]^d} |f_{r_{i,j}} - r_{i,j}| \leq \epsilon_j$ (Lemma 8) for a given tolerance ϵ_j, defined in a way which means $|G_{\text{ReLU}}(x)| \leq 1 \; \forall x \in [-1,1]$. We now seek to find a term for the error of the output of the i^{th} node at the $(J+1)^{\text{st}}$ layer in G and G_{ReLU}, of which we can bound. The output of the $(J+1)^{\text{st}}$ layer of H at the i^{th} node is

$$\Phi_{i,J+1}(x) \coloneqq r_{i,J+1}(a_{i,J+1}^T G(x) + b_{i,J+1}).$$

as we are passing the network $G(x)$ as an input. Now approximating $\Phi_{i,J+1}(x)$ with a ReLU network is

$$\Psi_{i,J+1}(x) \coloneqq f_{r_{i,J+1}}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1}).$$

We define the error as

$$E_{i,J+1} \coloneqq |\Psi_{i,J+1}(x) - \Phi_{i,J+1}(x)|.$$

We now seek a bound on this term. Using the elementary identity $|a-b| = |a-c+c-b| \leq |a-c| + |c-b|$ and setting $a = \Psi_{i,J+1}(x), b = \Phi_{i,J+1}(x)$ and $c = r_{i,J+1}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1})$, we find

$$E_{i,J+1} \leq |f_{r_{i,J+1}}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1}) - r_{i,J+1}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1})|$$

$$+|r_{i,J+1}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1}) - r_{i,J+1}(a_{i,J+1}^T G(x) + b_{i,J+1})|$$

Let

$$\psi \coloneqq |f_{r_{i,J+1}}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1}) - r_{i,J+1}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1})|.$$

Consider the inner product of vectors $x, y \in \mathbb{R}^n$ and let $|x_m| = \max_{i\in\{1,\dots,n\}} |x_i|$. It

[4] $r_{i,j}$ is the rational function for the i^{th} node in the j^{th} layer of G, where $1 \leq i \leq k_j$, $1 \leq j \leq J$

then follows

$$|x^T y| = \left| \sum_{i=1}^{n} x_i y_i \right| \leq \sum_{i=1}^{n} |x_i||y_i| \leq \sum_{i=1}^{n} |x_m||y_i| = |x_m| \sum_{i=1}^{n} |y_i| = \|x\|_\infty \|y\|_1$$

where we define $\|x\|_\infty = \max_{i \in \{1,\ldots,n\}} |x_i|$. Using this result, we can see that

$$|a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1}| \leq |a_{i,J+1}^T G_{\text{ReLU}}(x)| + |b_{i,J+1}|$$

$$\leq \|a_{i,J+1}\|_1 \|G_{\text{ReLU}}(x)\|_\infty + |b_{i,J+1}| \leq \|a_{i,J+1}\|_1 + |b_{i,J+1}| \leq 1$$

using our assumptions that $\|G_{\text{ReLU}}(x)\|_\infty \leq 1$ and $\|a_{i,J+1}\|_1 + |b_{i,J+1}| \leq 1$. As $a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1} \in [-1, 1]$, ψ is bounded from above by the maximum of $f_{r_{i,J+1}} - r_{i,J+1}$ over $x \in [-1, 1]$. Thus we conclude

$$\psi \leq \max_{x \in [-1,1]^d} |r_{i,J+1}(x) - f_{r_{i,J+1}}| \leq \epsilon_{J+1},$$

using Lemma 8. For the second term, by Lipschitz, we can say

$$\phi \coloneqq |r_{i,J+1}(a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1}) - r_{i,J+1}(a_{i,J+1}^T G(x) + b_{i,J+1})|$$

$$\leq L|a_{i,J+1}^T G_{\text{ReLU}}(x) + b_{i,J+1} - a_{i,J+1}^T G(x) - b_{i,J+1}| = L|a_{i,J+1}^T (G_{\text{ReLU}}(x) - G(x))|$$

where L is our Lipschitz constant. Using the above identity again, it follows

$$L|a_{i,J+1}^T (G_{\text{ReLU}}(x) - G(x))| \leq L\|a_{i,J+1}\|_1 \|G_{\text{ReLU}}(x) - G(x)\|_\infty$$

Further, we see

$$L\|a_{i,J+1}\|_1 \|G_{\text{ReLU}}(x) - G(x)\|_\infty \leq L\|G_{\text{ReLU}}(x) - G(x)\|_\infty$$

using $\|a_{i,J+1}\|_1 \leq 1$ which follows from our assumption that $\|a_{i,J+1}\|_1 \leq \|a_{i,J+1}\|_1 +$

$|b_{i,J+1}| \leq 1$. We then maximise the remaining expression over $[-1,1]^d$ to get

$$\phi \leq L \max_{x\in[-1,1]^d} \|G_{\text{ReLU}}(x) - G(x)\|_\infty$$

Thus our error is bounded as

$$E_{i,J+1} \leq L \max_{x\in[-1,1]^d} \|G_{\text{ReLU}}(x) - G(x)\|_\infty + \epsilon_{J+1}$$

We established that on the $(J+1)^{\text{st}}$ layer, at the i^{th} neuron, the rational network H takes in the subnetwork $G(x)$ as an input which returns $r_{i,J+1}(a_{i,J+1}^T H(x) + b_{i,J+1})$. We then approximate the output on the $(J+1)^{\text{st}}$ layer given by the rationals with a ReLU network and compare this error (this was our original $E_{i,J+1}$). As G_{ReLU} is a ReLU approximation to $G(x)$ and these are the inputs of the previous layer, it is equivalent to compare the maximum error of these functions. Thus

$$E_{i,j} = \max_{x\in[-1,1]^d} |G_{\text{ReLU}}(x) - G(x)|_j$$

for the j^{th} layer. Maximising this error over i yields

$$\max_{1\leq i\leq k_j} E_{i,j} = \|G_{\text{ReLU}}(x) - G(x)\|_\infty$$

Combining the inequalities, we find

$$\max_{1\leq i\leq k_{j+1}} E_{i,j+1} \leq L \max_{1\leq i\leq k_j} E_{i,j} + \epsilon_{j+1},$$

noting $E_{i,0} = 0$ as this is the input layer. Thus we can form the following bound

$$\begin{aligned}
\max_{1\leq i\leq k_1} E_{i,1} &\leq L \max_{1\leq i\leq k_0} E_{i,0} + \epsilon_1 = \epsilon_1 \\
\max_{1\leq i\leq k_2} E_{i,2} &\leq L \max_{1\leq i\leq k_1} E_{i,1} + \epsilon_2 \leq \epsilon_2 + \epsilon_1 \\
&\vdots \\
\max_{1\leq i\leq k_j} E_{i,j} &\leq \epsilon_1 + \epsilon_2 + \cdots + \epsilon_j = \sum_{n=1}^{j} \epsilon_n
\end{aligned}$$

Choosing $\epsilon_j = \epsilon L^{j-J-1}/(J+1)$ and by recalling that $1 \leq j \leq J+1$, it follows

$$\sum_{n=1}^{j} \epsilon_n = \frac{\epsilon}{L^{J+1}(J+1)} \sum_{n=1}^{j} L^n \leq \frac{\epsilon}{L^{J+1}(J+1)} \sum_{n=1}^{J+1} L^n \leq \frac{\epsilon L^{J+1}}{L^{J+1}(J+1)} \sum_{n=1}^{J+1} 1 = \epsilon$$

using $1 \leq n \leq j \leq J+1$. Thus $\max_{1\leq i\leq k_j} E_{i,j} \leq \epsilon$. At $J = N-1$, the network $G_{\mathrm{ReLU}}(x)$ is the ReLU approximation of the original rational network H, as the $J+1^{\mathrm{st}}$ layer is the N^{th} layer. Consider a single node in the j^{th} layer in H as a rational function approximated by the corresponding layer ReLU in $G_{\mathrm{ReLU}}(x)$. By lemma 8, we know that for ϵ accuracy, we require the ReLU network to be of size $\mathcal{O}\left(\log\left(1/\epsilon_j\right)^3\right)$. For the full ReLU approximation to H, it follows $\epsilon_j = \epsilon L^{j-N}/N$ and at the j^{th} layer, $G_{\mathrm{ReLU}}(x)$ satisfies $\mathcal{O}\left(k_j \log\left(\frac{1}{\epsilon_j}\right)^3\right) = \mathcal{O}\left(k \log\left(\frac{N}{L^{j-N}\epsilon}\right)^3\right)$, recalling $k_j \leq k$. Summing over the layers, we find the network $G_{\mathrm{ReLU}}(x)$ has size $\mathcal{O}\left(k \sum_{j=1}^{N} \log\left(\frac{N}{L^{j-N}\epsilon}\right)^3\right) = \mathcal{O}\left(k \sum_{j=1}^{N} \left(\log\left(\frac{NL^N}{\epsilon}\right) - j\log(L)\right)^3\right)$. Provided $L \geq 1$, $\log(L) \geq 0$ and so as j increases, the terms of the sum becomes smaller, meaning $\sum_{j=1}^{N} \left(\log\left(\frac{NL^N}{\epsilon}\right) - j\log(L)\right)^3 \leq N \log\left(\frac{NL^N}{\epsilon}\right)^3$. Thus we can say that

$$\mathcal{O}\left(k \sum_{j=1}^{N} \left(\log\left(\frac{NL^N}{\epsilon}\right) - j\log(L)\right)^3\right) = \mathcal{O}\left(kN \log\left(\frac{NL^N}{\epsilon}\right)^3\right)$$

which concludes the proof.

□

PDE–LEARN

Having discussed that deep learning can theoretically be used to mine governing equations from data, we shall now discuss a modern algorithm used to mine equations from data. This algorithm uses Rational Neural Networks to find these equations, which will allow us to see the discussed benefits of these networks. The algorithm we shall discuss is called *PDE discovery via L^0 Error Approximation and Rational Neural networks* – or `PDE-LEARN` for short, introduced in 2024 by Stephany et al. in [18], who also created the aforementioned `PDE-READ` [1].

We now define the problem statement for the algorithm. Consider spatial domain Ω and temporal domain $(0, T]$, $T > 0$. We assume there exists a *system response function* u such that $u : (0, T] \times \Omega \to \mathbb{R}$ describes a state of the system. Indeed, $u(t, x)$ denotes the system state at a time $t \in (0, T]$ at a position $x \in \Omega$ and we assume u to govern some physical process. We then assume there exists a *hidden PDE* satisfied by u given

$$f_0(\partial^{\alpha(0)}u, \ldots, \partial^{\alpha(M)}u) = \sum_{k=1}^{K} c_k f_k(\partial^{\alpha(0)}u, \ldots, \partial^{\alpha(M)}u) \tag{1}$$

where $\partial^{\alpha(0)}u, \ldots, \partial^{\alpha(M)}u$ denotes an enumeration of partial derivatives of u of order $\leq m$, where $M = D^m$, where D is the number of spatial dimensions being considered and m is the highest order of partial derivative. We refer to the functions $f_0, f_1, \ldots, f_K$ as *library terms*. The library terms are defined by the user based on intuition of what terms may occur in the final PDE. The algorithm has access to all terms which are defined in the library. We also have data points $\{(t_j, x_j)\}_{j=1}^{N_{Data}} \subseteq (0, T] \times \Omega$. The algorithm assumes we have access to noisy measurements $\tilde{u}$ such that

$$\{\tilde{u}(t_j, x_j)\}_{j=1}^{N_{Data}} \subseteq (0, T] \times \Omega$$

We assume the noise is normally distributed, such that

$$u(t_j, x_j) - \tilde{u}(t_j, x_j) \sim \mathcal{N}(0, \sigma^2)$$

for some standard deviation $\sigma > 0$.

The goal of the algorithm is to learn the coefficients $c_1 \dots c_K$ in equation (1) by training a Rational Neural Network $U : (0, T] \times \Omega \to \mathbb{R}$ to approximate u which satisfies the hidden PDE. The rational activation function is of type $(3, 2)$ and has trainable coefficients which are different between layers of U. These coefficients are learned along with the bias and weights during backpropagation. `PDE-LEARN` trains U and vector ξ simultaneously, where ξ is used to approximate the coefficients in (1).

Data and Architecture

`PDE-LEARN` is unique in the way in which it uses its unique training algorithm to find U and ξ which satisfy (1). In terms of data and architecture, the algorithm is not novel. The data, as discussed, is assumed to be noisy, sparse scientific data, which does not have to be confined to any prescribed dimension. Furthermore, an architecture is not prescribed to the algorithm but is specified by the user in terms of both depth and width. In the numerical experiments the authors present, they do not experiment with changing the depth for the same dataset, choosing to use four or five hidden layers for each dataset. This is something we shall explore in the subsequent numerical simulations.

Training Algorithm

The training algorithm of `PDE-LEARN` is by far the most interesting element of the algorithm. The authors take strong inspiration from the `DeepMoD` algorithm, another PDE discovery algorithm [24]. `DeepMoD` uses a three part loss-function, which is very similar to the one employed by Stephany et al. The loss function in `PDE-LEARN` is given

$$\mathcal{L}(U, \xi) = \omega_{\text{Data}} \mathcal{L}_{\text{Data}}(U) + \omega_{\text{Col}} \mathcal{L}_{\text{Col}}(U, \xi) + \omega_{L^p} \mathcal{L}_{L^p}(\xi) \tag{2}$$

where $\omega_{\text{Data}}, \omega_{\text{Col}}$ and ω_{L^p} are hyperparameters. The individual loss functions are

$$\mathcal{L}_{\text{Data}}(U) = \frac{1}{N_{\text{Data}}} \sum_{j=1}^{N_{\text{Data}}} \left| U(t_j, x_j) - \tilde{u}(t_j, x_j) \right|^2 \tag{3}$$

$$\mathcal{L}_{\text{Col}}(U) = \frac{1}{N_{\text{Col}}} \sum_{j=1}^{N_{\text{Col}}} \left| R_{\text{PDE}}(U, \hat{t}_j, \hat{x}_j) \right|^2 \tag{4}$$

$$\mathcal{L}_{L^p}(\xi) = \sum_{k=1}^{K} a_k \xi_k^2 \tag{5}$$

where

$$R_{\text{PDE}}(U, \hat{t}_j, \hat{x}_j) = f_0(\hat{t}_j, \hat{x}_j) - \sum_{k=1}^{K} c_k f_k(\hat{t}_j, \hat{x}_j)^5 \tag{6}$$

is the *PDE residual*, analogous to the residual $Ax - b$ from linear algebra. The points $\{(\hat{t}_j, \hat{x}_j)\}_{j=1}^{N_{\text{Col}}}$ are the *collocation points*. The values N_{Data} and N_{Col} represent the number of data and collocation points respectively.

The aim of `PDE-LEARN` is to find both the vector $\xi \in \mathbb{R}^K$ and $\tilde{u}$ which satisfy the hidden PDE (1). The network U is trained to approximate $\tilde{u}$ by use of the three-part loss function. The authors stipulate that U must satisfy both the noisy measurements $\{\tilde{u}(t_j, x_j)\}_{j=1}^{N_{\text{Data}}}$ and the hidden PDE. This is achieved by minimising the data loss function (3) which is simply the mean squared error of the difference between the network U and the noisy data $\tilde{u}$. They require ξ to also satisfy the hidden PDE, which is done by minimising (5). The collocation loss is the mean squared error of the PDE residual. It is this functions which links U and ξ by forcing them both to satisfy the hidden PDE (1). The training process is enhanced by use of the Adam optimisation algorithm [25].

The coefficients a_k in (5) are determined using the rule

$$a_k = \left(\frac{1}{\min(\delta, |\xi_k|^{2-p})} \right) \tag{7}$$

where $p \in (0, 2)$ is a hyperparameter and $\delta > 0$ is small to prevent poles. At the start of each training epoch, the coefficients $a_1, \ldots, a_K$ are determined according to

[5]Notation: $f_k(\hat{t}_j, \hat{x}_j) \coloneqq f_k(\partial^{\alpha(0)} U(\hat{t}_j, \hat{x}_j), \ldots, \partial^{\alpha(M)} U(\hat{t}_j, \hat{x}_j))$

(7). If the values of ξ_k are sufficiently large, then

$$a_k = |\xi_k|^{p-2} \implies \mathcal{L}_{L^p}(\xi) = \sum_{k=0}^{K} |\xi_k|^p$$

These coefficients are held constant during backpropagation, meaning the loss function is convex in ξ. We can see why convexity is necessary by considering if the p-norm was used instead as the loss function. For $0 < p < 1$, the function $\|\xi\|_p^p$ is non-convex (and consequently not a norm as it fails to satisfy the triangle inequality). This causes problems in training as the function has a sharp corner at the origin which poses issues when trying to compute gradients and can prevent the Adam optimiser from converging to a minimum.

The algorithm consists of three stages. First, *Burn-In*. During this stage, we set $\omega_{L^p} = 0$ and $\omega_{\text{Data}} = \omega_{\text{Col}} = 1$. `PDE-LEARN` initialises U and ξ and the algorithm begins training U to approximate $\tilde{u}$ for a user defined number of epochs. As $\omega_{L^p} = 0$, we are not minimising $\mathcal{L}_{L^p}$, so ξ is largely non-zero. As a result, the PDE identified at this stage is not very accurate. Once training has finished, ξ is pruned, setting all values below a tolerance to zero. Second, *Sparsification*. Here $\omega_{L^p} = \rho$, where ρ is some small, non-zero value. We begin training again. As $\omega_{L^p} \neq 0$, the $\mathcal{L}_{L^p}$ loss function will influence the training process, causing ξ to become sparse, only containing terms which correspond to the RHS of the hidden PDE. This step identifies which coefficients are non-zero but does not prescribe the final values. In the final step, called *Fine-Tuning*, we set $\omega_{L^p} = 0$ again and proceed with a final round of training. When active, the $\mathcal{L}_{L^p}$ loss function causes coefficients to go to zero during training, as it is being minimised. However, as we have set $\omega_{L^p} = 0$, the only loss function which influences ξ is the collocation loss function, so ξ converges to the hidden PDE.

Numerical Simulation

We shall conclude this paper with some numerical simulations. These simulations will bring together all concepts we have discussed so far. As `PDE-LEARN` is available

online to use, it seems appropriate to run the algorithm on some classical examples of PDEs, namely the KdV equation and Burger's equation. The authors provide functionality to create data compatible with the algorithm [26, 27]. There are three main areas we would like to see in practice. First, accuracy: Do we see the PDEs being approximated correctly by the algorithms? This will of course be partly contingent on how well `PDE-LEARN` works, but the theoretical analysis we conducted in the first section is underlying this process. Second, depth: do we see a difference in the results when we use shallower networks? From the analysis discussed, comparing a shallow network to a deep one, we would expect to see less accurate results in the same length of time. And finally, activation functions: `PDE-LEARN` allows for other activation functions to be used, namely $\phi(x) = \tanh(x)$. We shall compare this function to the rational activation functions, expecting to see faster results with the rational functions, due to the low cost of the low degree functions. Unfortunately, `PDE-LEARN` does not currently support ReLU activation functions, which would have served as a more direct comparison for rational networks. We choose to use noise free data for simplicity and the parameter values can be found in Appendix A.1.

Burgers' Equation

Burgers' equation is a nonlinear PDE which often arises in fluid dynamical contexts, in particular in gas dynamics. For $u(t, x)$, it is given

$$u_t = \nu u_{xx} - u u_x \tag{8}$$

First we shall investigate the effects of different activation functions on data generated from solving Burgers' equation subject to the initial condition $u(0, x) = -\sin\left(\frac{\pi x}{8}\right)$ on the domain $(t, x) \in (0, 10] \times [-8, 8]$ (Figure 1) with 10000 data points. The results are shown in Table 1. We trained a network with five layers with 20 neurons on each layer, taking $\nu = 0.1$.

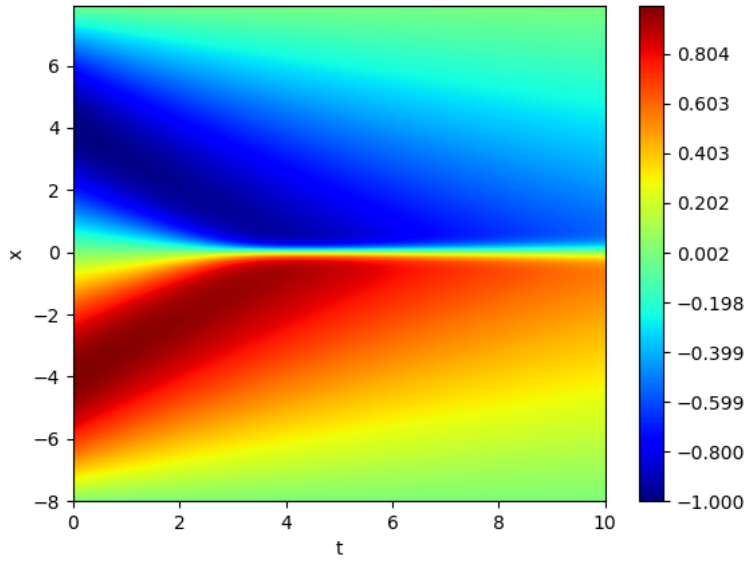

Figure 1: Burger's Equation given initial condition $u(0, x) = -\sin\left(\frac{\pi x}{8}\right)$

Activation Function	PDE
$\tanh(x)$	$u_t = -0.8004uu_x + 0.2108u^2u_{xx} + 0.0740uu_x^3$
Rational	$u_t = 0.0986u_{xx} - 0.9792uu_x - 0.0026u_xu^3$

Table 1: Results for Burgers' Equation Dataset

As expected, in 1000 epochs of each stage, `PDE-LEARN` does not manage to converge to the PDE with a $\tanh(x)$ activation function, even when the data is completely noise free. Conversely, the rational activation function converges much more precisely on the PDE, with only a minor error term. These results are in line with our expectations.

The KdV Equation

The Korteweg–De Vries (KdV) equation is a nonlinear PDE which models shallow water waves. In one spatial dimension x, it is given

$$u_t = -uu_x - u_{xxx} \tag{9}$$

We test `PDE-LEARN` on data obtained from solving (9) subject to condition $u(0, x) = -\sin\left(\frac{\pi x}{20}\right)$ on the domain $(t, x) \in [0, 40] \times [-20, 20]$. The data is shown in Figure 1.

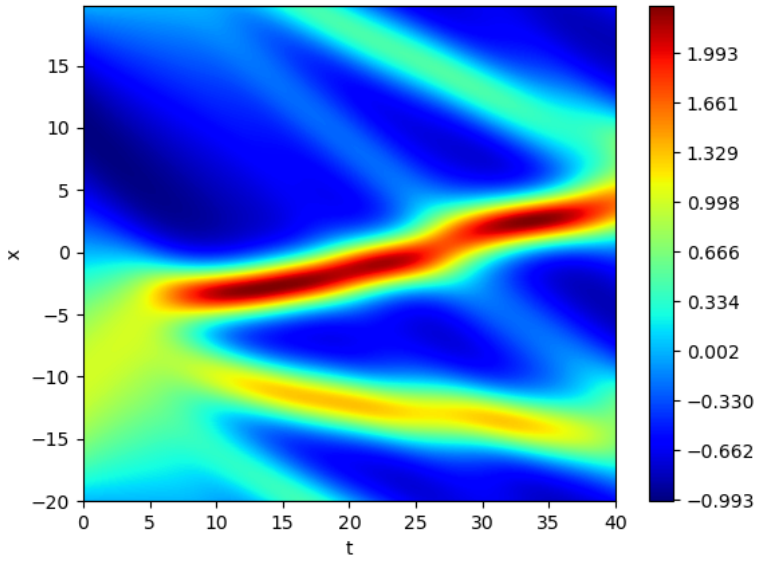

Figure 2: The KdV equation given initial condition $u(0, x) = -\sin\left(\frac{\pi x}{20}\right)$

Now we investigate the effects of depth in more detail. First, we use a network of five layers, each with 40 nodes. Next, we try a single-hidden-layer network, with 100 nodes on that layer. We repeat, increasing the number of nodes to 200. Our final experiment uses a three-layer network with 60 nodes on each layer.

Layers	Neurons	PDE
5	40	$u_t = -0.8831u_{xxx} - 0.8941uu_x$
1	100	$u_t = -0.0725u_x$
1	200	$u_t = -0.1385uu_x$
3	60	$u_t = -0.7445u_{xxx} - 0.7717uu_x$

Table 2: Results for the KdV Equation Dataset

The results, given in Table 2, are in line with our expectations. We see that the deeper networks are capable of a far higher level of expressivity. Even the three-layer network with 60 neurons, which has fewer overall neurons than the single-layer network with 200, manages a result much closer to the KdV equation.

Conclusion

We have discussed many of the theoretical concepts underlying PDE discovery, such as density and depth of networks. We explored how rational functions appear to be a sensible alternative to ReLU networks and saw how these are applied by studying `PDE-LEARN`. These theoretical concepts were seen in practice with our numerical experiments performed using `PDE-LEARN`, the results of which confirmed the theoretical results about depth which we had discussed. Given more space, it would have been insightful to perform a direct comparison between ReLU and rational activation functions and to discuss training algorithms and optimisation techniques further.

References

[1] Robert Stephany and Christopher Earls. "PDE-READ: Human-readable partial differential equation discovery using deep learning". In: *Neural Networks* 154 (2022), pp. 360–382. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2022.07.008. URL: https://www.sciencedirect.com/science/article/pii/S0893608022002660.

[2] Nicolas Boullé, Christopher J. Earls, and Alex Townsend. "Data-driven discovery of physical laws with human-understandable deep learning". In: *CoRR* abs/2105.00266 (2021). arXiv: 2105.00266. URL: https://arxiv.org/abs/2105.00266.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] John K. Hunter. *Measure Theory Notes Chapter 7*. Department of Mathematics, University of California at Davis. URL: https://www.math.ucdavis.edu/~hunter/measure_theory/measure_notes_ch7.pdf.

[5] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. URL: https://www.sciencedirect.com/science/article/pii/089360809190009T.

[6] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[7] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80131-5. URL: https://www.sciencedirect.com/science/article/pii/S0893608005801315.

[8] Arian Novruzi. "Partial differential equations". In: *A Short Introduction to Partial Differential Equations*. Cham: Springer Nature Switzerland, 2023, pp. 19–24. ISBN: 978-3-031-39524-6. DOI: 10.1007/978-3-031-39524-6_2. URL: https://doi.org/10.1007/978-3-031-39524-6_2.

[9] John K. Hunter. *Measure Theory Notes Chapter 7*. Department of Mathematics, University of California at Davis. URL: https://www.math.ucdavis.edu/~hunter/pdes/pde_notes.pdf.

[10] Gaspard Jankowiak. *Partial Differential Equations III: The regularity theory of DE GIORGI, NASH and MOSER for elliptic partial differential equations*. 2023. URL: https://gaspard.janko.fr/s/enseignement/konstanz/SoSe2023/PDE3/PDE3-DeGiorgiNashMoser.pdf.

[11] J. Nash. "Continuity of Solutions of Parabolic and Elliptic Equations". In: *American Journal of Mathematics* 80.4 (1958), pp. 931–954. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2372841 (visited on 12/16/2024).

[12] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL: http://udlbook.com.

[13] Abdulhamit Subasi. "Chapter 3 - Machine learning techniques". In: *Practical Machine Learning for Data Analysis Using Python*. Ed. by Abdulhamit Subasi. Academic Press, 2020, p. 179. ISBN: 978-0-12-821379-7. DOI: https://doi.org/10.1016/B978-0-12-821379-7.00003-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128213797000035.

[14] Matus Telgarsky. "Representation Benefits of Deep Feedforward Networks". In: *CoRR* abs/1509.08101 (2015). arXiv: 1509.08101. URL: http://arxiv.org/abs/1509.08101.

[15] Dmitry Yarotsky. "Error bounds for approximations with deep ReLU networks". In: *CoRR* abs/1610.01145 (2016). arXiv: 1610.01145. URL: http://arxiv.org/abs/1610.01145.

[16] Patrick Kidger and Terry J. Lyons. "Universal Approximation with Deep Narrow Networks". In: *CoRR* abs/1905.08539 (2019). arXiv: 1905.08539. URL: http://arxiv.org/abs/1905.08539.

[17] Nicolas Boullé, Yuji Nakatsukasa, and Alex Townsend. "Rational neural networks". In: *CoRR* abs/2004.01902 (2020). arXiv: 2004.01902. URL: https://arxiv.org/abs/2004.01902.

[18] Robert Stephany and Christopher Earls. "PDE-LEARN: Using deep learning to discover partial differential equations from noisy, limited data". In: *Neural Networks* 174 (2024), p. 106242. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2024.106242. URL: https://www.sciencedirect.com/science/article/pii/S0893608024001667.

[19] Hao Xu, Junsheng Zeng, and Dongxiao Zhang. "Discovery of Partial Differential Equations from Highly Noisy and Sparse Data with Physics-Informed Information Criterion". In: *Research* 6 (Jan. 2023). ISSN: 2639-5274. DOI: 10.34133/research.0147. URL: http://dx.doi.org/10.34133/research.0147.

[20] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181.

[21] Lloyd N. Trefethen. *Rational Approximation*. Submitted to Notices Amer. Math. Soc. 2024. URL: https://people.maths.ox.ac.uk/trefethen/ratfun.pdf.

[22] Yuji Nakatsukasa, Olivier Sète, and Lloyd N. Trefethen. "The AAA Algorithm for Rational Approximation". In: *SIAM Journal on Scientific Computing* 40.3 (Jan. 2018), A1494–A1522. ISSN: 1095-7197. DOI: 10.1137/16m1106122. URL: http://dx.doi.org/10.1137/16M1106122.

[23] Matus Telgarsky. *Neural networks and rational functions*. 2017. arXiv: 1706.03301 [cs.LG]. URL: https://arxiv.org/abs/1706.03301.

[24] Gert-Jan Both et al. "DeepMoD: Deep learning for model discovery in noisy data". In: *Journal of Computational Physics* 428 (Mar. 2021), p. 109985. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2020.109985. URL: http://dx.doi.org/10.1016/j.jcp.2020.109985.

[25] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv e-prints*, arXiv:1412.6980 (Dec. 2014), arXiv:1412.6980. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs.LG].

[26] Robert Stephany and Christopher Earls. *PDE-READ Github Repository*. 2022. URL: https://github.com/punkduckable/PDE-READ.

[27] Robert Stephany and Christopher Earls. *PDE-LEARN Github Repository*. 2024. URL: https://github.com/punkduckable/PDE-LEARN.

A Appendix

A.1 Parameters

Parameter	Value
ρ	0.002
Learning Rate	0.001
Training Collocation Points	3000
Testing Collocation Points	1000