Physics/Math (2022), 1–8

CAPSTONE PROJECT

LAWRENCE UNIVERSITY
APPLETON, WISCONSIN

Using Scientific Machine Learning to solve Partial Differential Equations

Miles Cochran-Branson[1]

[1]Department of Physics and Mathematics, Lawrence University, Appleton, WI.

Keywords: SciML, Scientific Computing, ML/AI, PINN, PDEs

Abstract

Machine learning and artificial intelligence have been used with huge success in data science, mathematics, and physics as well as numerous other fields to solve complex or previously impossible computational problems. Computationally intensive problems in physics such as data analysis using a complex model and little data, or finding numerical solutions to differential equations have traditionally been solved with classical scientific computing techniques such as Taylor-series expansions or approximations using polynomials. Recent efforts have gone into replacing these techniques with neural networks and machine learning. Neural networks have the potential to fit any complex model or differential equation solution very accurately and can solve these problems quickly. In the following paper, I describe the basic principles of the growing field of Scientific Machine Learning (SciML). I will use the techniques I describe to solve the Einstein field equations to obtain the Schwarzschild metric using the package `NeuralPDE.jl` in the `Julia` programming language. This example demonstrates the power a physics-informed neural network has in solving large, highly coupled, systems of partial differential equations.

Contents

1. Introduction

Machine learning and scientific computing are two prominent fields with applications in physics, mathematics, and computer science. An initiative laid out by the DOE in 2019 outlines the importance and potential of the ever growing field of SciML [1]. In this report, they indicate the challenges that SciML could potentially solve: lack of data in analysis, and solving differential equations. In the first case, in problems where data is either expensive or impossible to collect, implementations of machine learning become impossible. This can be overcome if we can incorporate physics knowledge of the system into the neural network. Such implementations make use of *Physics-Informed Neural Networks*

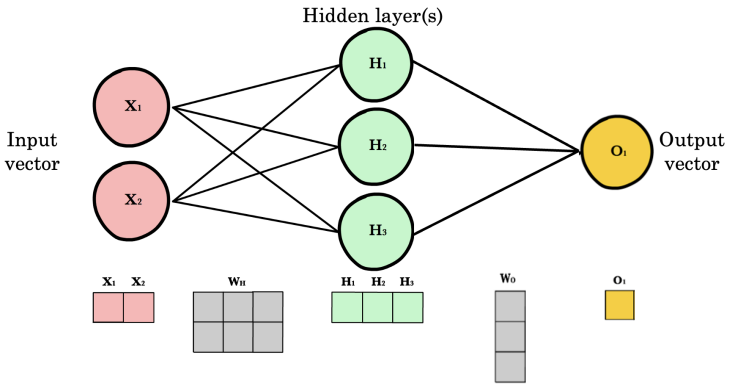


Figure 1. *An example of a simple neural network with two inputs, one hidden layer with three nodes, and a single output. The network is defined by the weights connecting the layers which are represented as matrices. At each node, the weights are passed to a function called the* activation function *which scales the weights appropriately. In principle, the size of the network can change in every dimension. .*

(PINN) [2]. It turns out that PINN can also solve differential equations by slight modification of the process used above. In fact, we can include data into solving of differential equations to better fit a model or aid in solving the equation(s).

Below, I will describe how machine learning in tandem with scientific computing can solve both of these problems elegantly and efficiently. I will begin by outlining the basics of what machine learning is in section two. In sections three and four I will describe how to solve ODEs and PDEs with Physics-Informed Neural Networks (PINN). Finally, in section five I present a specific problem: solving Einstein's field equations to obtain the Schwarzschild Metric using the `Julia` library `NeuralPDE.jl`. In the conclusions section I describe some future directions of this research and hopefully some suggestions? We shall see.

2. Machine Learning

Machine learning and artificial intelligence are prominent features of our everyday life from personalized advertisements to facial recognition software. Machine learning is also a powerful analytical tool in scientific research, in particular with model development and validation. Below, I will describe how neural networks can be trained to make predictions and give some motivation for why neural networks are well suited for function approximation.

2.1. Neural Networks

A neural network is simply a function composed of layers of matrix multiplication. As input, a neural networks take a matrix with size of number of obervables by number of data points and gives as an output a vector of weights. Thus, our network takes the form

$$NN : \mathbb{R}^{n\times m} \to \mathbb{R}^{\ell} \tag{2.1}$$

where n is the number of parameters for each data point m and ℓ is the number of desired output features. Networks have a fixed size defined by the number of *hidden layers* and the number of *nodes* each hidden layer has.

Hidden layers cannot be directly accessed in the training process and are thus so named. A node within a layer describes a connection point between layers. Connections are achieved via a multiplication by a weight stored in the network. Modification of these weights changes the way in which nodes are

connected to each other and thus how inputs are interpreted by the network. At each node, the weight passed to the node is modified by an *activation function*. These typically modify the weight such that each weight is not linearly connected to each other. This allows us to describe a more complex input space more accurately [3]. Some examples of activation functions include the sigmoid defined by

$$\sigma(w) = \frac{1}{1 + e^{-w}}, \tag{2.2}$$

$\tanh(w)$, and LSTM defined by $f(w) = \max\{0, x\}$ where in each case w is the weight provided by the previous layer. In the work that follows, we typically use a sigmoid activation function.

An example of a simple neural network with one hidden layer is shown in Fig. 1. This network has one input dimension and, as shown in the graphic, we feed one data point X with two features x_1, x_2 into the network. Parameters of the network are stored in the matrix W_0 such that when we do the multiplication XW_0 we get a vector of three values corresponding to the three nodes of the single hidden layer. The activation function is now applied to these values. We then multiply by the weights W_1 to get our output of a single value which is passed to one final activation function. In most cases, the activation in the final node is the identity function $f(x) = x$. Notice that we can have as many inputs, layers, nodes, and output nodes as we would like. Scaling up the amount of hidden layers will increase the weights of the matrix significantly. The network described above is typically called a *feed-forward* neural network as there are no cycles between nodes and no reference to nodes previously touched. There are more complex neural networks such as recurrent neural networks which have these features, however, we will not use these in our work.

2.2. Training Neural Networks

Now that we have built some neural network, we would like to have it make some useful predictions! This involves picking a set of special weights through a process called training.

In order to train a neural network, we need a way in which to evaluate how well the network is doing. This is achieved with a *loss function*. The loss function takes the output of the network given a certain input and returns a number, thus, our loss function is of the form

$$L : \mathbb{R}^{\ell} \to \mathbb{R}. \tag{2.3}$$

Such a function is defined such that the smaller the output value, the better the network performs. Thus, training a neural network becomes an optimization problem where we want to minimize the function $L(w)$ where w are the weights of our network.

2.3. The Universal Approximation Theorem

Thang that makes everything work!

Theorem 2.1. *Consider $x \in \Omega$ and let $u(x)$ be a regular function and $N(x, w)$ be a neural network with weights w. Then we can find some arbitrarily small $\epsilon \in \Omega$ such that*

$$||N(x, w) - u(x)|| < \epsilon. \tag{2.4}$$

3. Solving ODEs with Physics-Informed Neural Networks

4. Solving PDEs with Physics-Informed Neural Networks

We now turn out attention to solving partial different equations using techniques from scientific machine learning. We will use the package `NeuralPDE.jl` [4] to solve some examples.

Consider a partial differential equation given by

$$f(u(x);\lambda) = 0 \tag{4.1}$$

where $x \in \Omega$ are the independent variables in the space Ω, $u(x)$ is the solution, f is some non-linear function acting on u and λ are the parameters of the equation. To solve with a neural network $N(x, w)$ where N is a neural network with weights w, we want to fulfill

$$f(N(x, w);\lambda) \approx 0. \tag{4.2}$$

The universal approximation theorem tells us that this should be possible. The error of the above equation is then given by

$$L(w) = \int_{\Omega} ||f(N(x, w);\lambda)||\, dx. \tag{4.3}$$

We have suggestively named the error L as we can use this function as a loss function in training our neural network N by attempting to minimize L. Our computational problem then reduces to the problem of evaluating the integral in Eqn. 4.3. Notice that in practical problems, we apply boundary conditions b_i on $\partial\Omega \in \Omega$. In order to use these in our neural network, we add in these conditions to our loss function giving us

$$L(w) = \sum_i \int_{\Omega\setminus\partial\Omega} ||f_i(N(x, w);\lambda)||\, dx + \sum_i \int_{\partial\Omega} ||b_i(N(x, w);\lambda)||\, dx. \tag{4.4}$$

where we have generalized the problem to a system of coupled differential equations f_i.

There are several different methods for evaluating the above integral namely: grid approximation techniques, stochastic grid approximation techniques, and machine learning (quadrature) techniques. A simple grid approximation takes the space Ω and divides it into units of volume ΔV with a specific length. The PDE is evaluated at each of this points and scaled by the volume, thus our integral is computed via

$$\int_{\Omega} ||f(N(x, w);\lambda)||\, dx = \sum_i \Delta V ||f(x_i)||. \tag{4.5}$$

This method has two main disadvantages: 1) as the dimension of the problem increases, the number of points we sample increases exponentially in order to maintain the same granularity and 2) no evaluation of the integral is done between grid points, thus information can be quickly lost. To solve the second problem, we can use stochastic methods of sampling—i.e., use Monte Carlo techniques to evaluate the integral. This can be described by

$$\int_{\Omega} ||f(N(x, w);\lambda)||\, dx = \alpha \sum_i ||f(x_i)||. \tag{4.6}$$

for some scaler α. As we simply want to minimize this integral, there is no need to specify α. This problem still suffers from dimensionality exponentially increasing. Thus, we turn to a third method: quadrature training with a neural network. Several processes for specifying quadrature are described in [5]. These are typically of the form

$$\int_{\Omega} ||f(N(x, w);\lambda)||\, dx = \sum_i \alpha_i ||f(x_i)||\, dx. \tag{4.7}$$

In the implementation via `NeuralPDE.jl`, a neural network is solved via `Integrals.jl` which calls the `Julia` differential equation solver [6] to find the correct sampling points x_i and weights α_i in an implementation of Gaussian quadrature rules.

4.1. Example using `NeuralPDE.jl`

We show how `NeuralPDE.jl` can be used to solve a simple example and describe the important components of the code.

Example 4.1. Consider the system of PDEs given by

$$\begin{aligned} \partial_t u_1 &= \partial_x^2 u_1 - u_2 \partial_x u_1 + u_1^2 - 2\int_0^1 u_1^2\, dx \\ 0 &= \partial_x u_2 - u_1 \end{aligned} \tag{4.8}$$

for $0 < x < 1$ and $t > 0$. We emply the following initial condition

$$u_1(0, x) = \cos \pi x \tag{4.9}$$

and the boundary conditions

$$\partial_x u_1(t, 0) = \partial_x u_1(t, 1) = u_2(t, 0) = u_2(t, 1) = 0. \tag{4.10}$$

This system is solved analytically in the paper [7] where they find the exact solution

$$\begin{aligned} u_1(t, x) &= e^{-\pi^2 t} \cos \pi x \\ u_2(t, x) &= \frac{1}{\pi} e^{-\pi^2 t} \sin \pi x. \end{aligned} \tag{4.11}$$

Let's now solve this problem using `NeuralPDE.jl`. We begin by importing the required packages from `Julia` , defining the variables and parameters of the system, and defining the equation, boundary conditions, and domains over which we want to solve.

```
using NeuralPDE, ModelingToolkit, Lux, Domainsets
using Optimization, OptimizationOptimisers
import ModelingToolkit: Interval

@parameters t x
@variables u1(..) u2(..)

Dx = Differential(x)
Dxx = Differential(x)^2
Dt = Differential(t)
Ix = Integral(x in DomainSets.ClosedInterval(0, 1))

eqns = [Dt(u1(t,x)) ~ Dxx(u1(t,x)) - u2(t,x)*Dx(u1(t,x)) +
                                     (u1(t,x))^2 - 2*Ix((u1(t,x))^2),
        0 ~ Dx(u2(t,x)) - u1(t,x)]

bcs = [u1(0,x) ~ cos(π*x),
       Dx(u1(t,0)) ~ 0,
       Dx(u1(t,1)) ~ 0,
       u2(t,0) ~ 0,
       u2(t,1) ~ 0
]

domains = [x ∈ Interval(0.0, 1.0),
```

```
            t ∈ Interval(0.0, 1.0)]

@named pde_sys = PDESystem(eqns, bcs, domains, [t,x], [u1(t,x), u2(t,x)])
```

Notice that with the aid of the package `ModelingToolkit.jl`, we can symbolically represent the set-up of our problem. This is a feature of the SciML ecosystem of `Julia`.

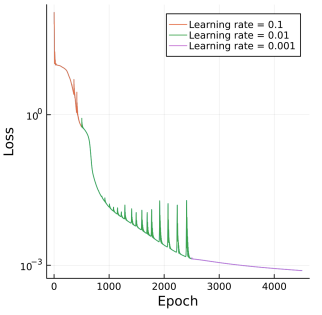
We then create a neural network using `Lux.jl` and pick a discretization to use. In this example we use the sigmoid as our activation function and build a network with one hidden layer of fifteen nodes. The input layer of our network must be the length of the number of parameters defined in our problem. Additionally, we use a different neural network for each solution u_1 and u_2 of our system, thus we define two neural networks with the set-up described above. The discretization we choose is the quadrature training method described above.

```
dim = length(domains) # number of dimensions
n = 15
chains = [Lux.Chain(
            Dense(dim, n, Lux.σ),
            Dense(n, n, Lux.σ),
            Dense(n, 1)) for _ in 1:2]

strategy = QuadratureTraining()
discretization = PhysicsInformedNN(chains, strategy)
@time prob = discretize(pde_sys, discretization)
```

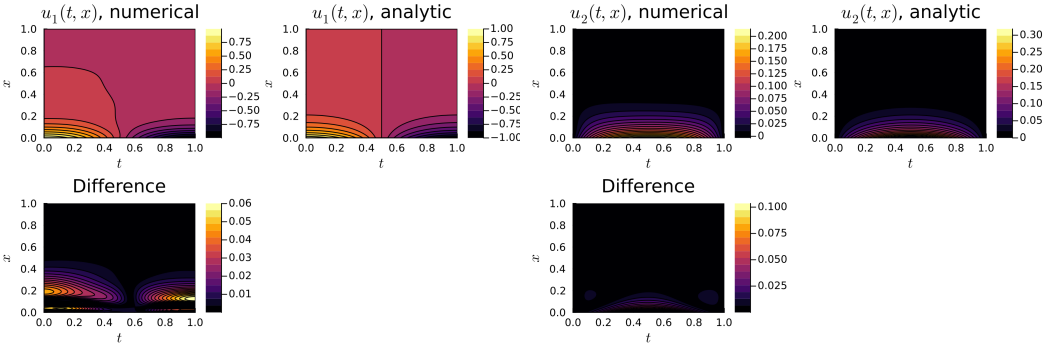
We finally define a function to record the loss of our training and train the network

```
i = 0
loss_history = []

callback = function (p,l)
    global i += 1
    if i % 100 == 0
        println("Current loss is: $l")
    end
    append!(loss_history, l)
    return false
end

res = @time Optimization.solve(prob, ADAM(0.1);
                               callback = callback, maxiters=500)
prob = remake(prob, u0=res.minimizer)
res = @time Optimization.solve(prob, ADAM(0.01);
                               callback = callback, maxiters=2000)
prob = remake(prob, u0=res.minimizer)
res = @time Optimization.solve(prob, ADAM(0.0001);
                               callback = callback, maxiters=2000)
```

For optimization we use the ADAM optimizer and vary the learning rate starting with a rate of 0.1 over 500 epochs, then 0.01 over 2000 epochs, then finally 10^{-4} over a final 2000 epochs. The loss over these epochs and learning rates is shown below



We can then plot the result of this training as shown below:



and we see that our numerical solution very closely matches the analytical solution.

For more information, see the data availibility statement.

5. Solving Einstein's Field Equations to obtain the Schwarzschild Metric using `NeuralPDE.jl`

OG solution: [8]. Video I use: [9].

6. Conclusion

Acknowledgments. Grateful for support of Pickett and Heaton + physics and mathematics EDU at LU! Huzzah!!

Data Availability Statement. All code presented above can be found here in GitHub at the following link `https://github.com/lvb5/solve_PDEs_with_PINN`.

References

[1] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, "Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence," Tech. Rep. 1478744, Feb. 2019.

[2] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, pp. 422–440, June 2021.

[3] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92–108, Sept. 2022.

[4] K. Zubov, Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe, N. Vinchhi, K. Balakrishnan, D. Upadhyay, and C. Rackauckas, "NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations," July 2021.

[5] J. A. Rivera, J. M. Taylor, Á. J. Omella, and D. Pardo, "On quadrature rules for solving Partial Differential Equations using Neural Networks," *Computer Methods in Applied Mechanics and Engineering*, vol. 393, p. 114710, Apr. 2022.

[6] C. Rackauckas and Q. Nie, "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia," *Journal of Open Research Software*, vol. 5, p. 15, May 2017.

[7] B. Benhammouda and H. Vazquez-Leal, "Analytical solutions for systems of partial differential–algebraic equations," *SpringerPlus*, vol. 3, p. 137, Mar. 2014.
[8] K. Schwarzschild, "On the gravitational field of a mass point according to Einstein's theory," May 1999.
[9] eigenchris, "(1) Relativity 108a: Schwarzschild Metric - Derivation - YouTube." https://www.youtube.com/.