**CAPSTONE PROJECT**

# Using Scientific Machine Learning to solve Partial Differential Equations

Miles Cochran-Branson

Department of Physics and Mathematics, Lawrence University, Appleton, WI.

**Abstract**

Machine learning and artificial intelligence have been used with great success in data science, mathematics, and physics as well as numerous other fields to solve complex or previously impossible 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, we describe the basic principles of the growing field of Scientific Machine Learning (SciML). (THIS WILL LIKELY CHANGE WITH MATURITY OF THE PROJECT): We will use the techniques we 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

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 computational challenges 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* (PINN) [2]. Moreover, PINN can additionally solve differential equations by slight modification of the lack of data problem. We can even combine these two techniques by including data into solving differential equations to better fit a model or aid in solving the equation(s). A robust implementation of many of these techniques has been underway in the `Julia` programming language. Packages that make use of machine learning are part of the `SciML.jl` universe which includes a basic syntax for code and basic functions in using SciML. Further information on the evolution of the SciML universe can be found in last year's SciML Con [3].

Because of the rapidly growing nature of the field of SciML, documentation on how the techniques are used and the new tools available are not easy to access. This paper seeks to fill in some of the gaps, describing succinctly how SciML can be used to solve differential equations. In particular, we put the newly-released package `NeuralPDE.jl` to the test to solve several complex differential equations and compare the computational solutions to known analytical solutions. In section two, we will outline the basics of what machine learning is. In sections three and four we will describe how to solve ODEs and PDEs with Physics-Informed Neural Networks (PINN). Finally, in section five we present a more complex problem: solving Einstein's field equations to obtain the Schwarzschild Metric using PINN.

ADD HERE ONCE COMPLETED: In the conclusions section I describe some future directions of this research and hopefully some suggestions? We shall see. Also findings?

## 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, in particular differential equation approximation.

### 2.1. Neural Networks

A neural network is simply a function composed primarily of layers of matrix multiplication, but also of non-linear components—activation functions, and optionally addition of bias vectors within the function [4, 5]. As input, a neural network can take a scalar, vector, matrix, or sometimes tensor with a user-specified size / dimension and gives as an output a vector or scalar. Thus, our network $\mathcal{N}$ takes the form

$$\mathcal{N} : \mathbb{R}^{\Theta} \to \mathbb{R}^{\ell} \tag{2.1}$$

where $\Theta$ is the user-defined input space, and $\ell$ 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 are composed of nodes which cannot be directly accessed in the training process. A node within a layer describes a connection point between layers. Connections are achieved via 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 a non-linear function called the *activation function*. This allows us to
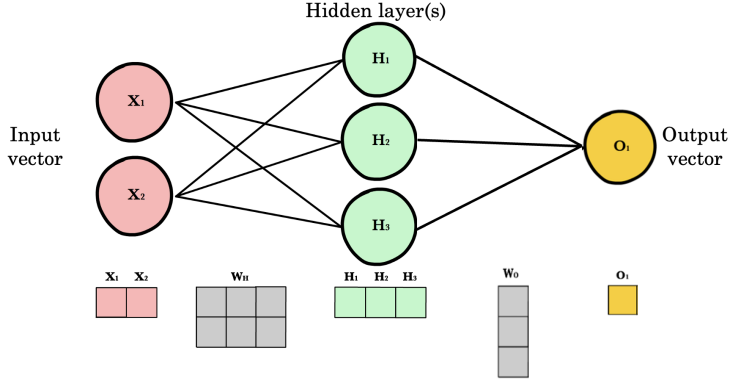
*Figure 1.* *An example of a simple neural network with two inputs, one hidden layer with three nodes, and a single output. The network is defined by the weights connecting the layers which are represented as matrices. At each node, the weights are passed to a function called the activation function which scales the weights appropriately. In principle, the size of the network can change in every dimension.*

describe a complex input space more accurately [6]. Some examples of activation functions include the sigmoid defined by

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

the hyperbolic tangent $\tanh(w)$, and LSTM defined by $f(w) = \max\{0, w\}$ where in each case $w$ are a vector of weights of the network at a node. Typically, the sigmoid is well suited for classification problems, while LSTM is best used in regression problems due to its simplicity. In examples presented later, 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 and optionally a bias vector can be added to these modified weights. 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. The network described above is typically called a *feed-forward* neural network as there are no cycles between nodes, no reference to nodes previously touched and each node from the previous layer is connected to each node of the subsequent layer. 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 given a specific task. This is achieved with a *loss function*. The loss function takes an input of all of the parameters of the network that can change, $X$, and outputs a single number, thus, our loss function is of the form

$$\mathcal{L} : \mathbb{R}^{\text{size}(X)} \to \mathbb{R}. \tag{2.3}$$

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 $\mathcal{L}(X)$. Typically, the space $X$ is composed of the weights and biases of the network.

For a very basic idea of how this works, imagine the space defined by $\mathcal{L}$ as a surface with some collection of minima. We want to find a minimum, so we need to travel in the direction of one. Intuitively, we can take the gradient and then travel in the direction of the negative gradient to go "downhill". More specifically, we travel in the $-\nabla \mathcal{L}(X)$ direction taking discrete steps of size $\Delta x$. After each step we update the network parameters, re-evaluate the loss, compute $-\nabla \mathcal{L}(X)$ and repeat. We continue in this way for as many steps as the user desires. This process is called gradient descent and the algorithm for finding the desired weights is called backpropagation [8]. In most cases, it is not necessary to find the global minimum and may even be computationally disadvantageous as finding global minimum can lead to over-fitting [7]. Because in some cases, learning can get stuck in a "saddle point" or small local minimum, we often use stochastic methods which allow us to get out of such cases and continue learning to a minimum. This process is called stochastic gradient descent. In the following examples, we will use the ADAM optimizer which uses stochastic methods to find a minimum [9].

### 2.3. The Universal Approximation Theorem

A natural question when posing machine learning as a solution to PDEs is why this should work in the first place. The answer comes down to a famous theorem in machine learning: the Universal Approximation Theorem (UAT):

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

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

Proofs of this theorem are beyond the scope of this report, but are readily availible for several different contexts including for classical feed-forward networks [10, 11], finding probability distributions [12], and complex-valued neural networks [13].

There is, however, one big caveat to this theorem: nothing about the problem tells us how big our network should be in order to fulfill the theorem. In some cases, the size of the network may be computationally infeasible to model. Nonetheless, it is this theorem which allows us to be able to have some hope that differential equations can be arbitrarily approximated by neural networks.

In the rest of the paper, we take these theoretical concepts and apply them to the problem of solving differential equations, starting with ODEs and then moving to PDEs.

## 3. Solving ODEs with Physics-Informed Neural Networks

As an illustration of the application of PINN, consider an ordinary differential equation of the form

$$u' = f(u, t). \tag{3.1}$$

The UAT tells us that we should be able to achieve $N(w, t) \approx u(t)$ where $N(w, t)$ is a neural network with weights $w$. Consequently, our neural network must satisfy $N'(w, t) \approx f(N(w, t), t)$. To solve, we discretize the space $\Omega$ where $t \in \Omega$. Subsequently, we define the loss function by

$$\mathcal{L}(w) = \sum_i \left( \frac{d\mathcal{N}(w, t_i)}{dt} - f(\mathcal{N}(w, t_i), t_i) \right)^2. \tag{3.2}$$

In order to find a unique solution, we apply the initial condition $u(0) = u_0$. We can simply add this to our loss function as

$$\mathcal{L}(w) = (\mathcal{N}(w,0) - u_0)^2 + \sum_i \left( \frac{d\mathcal{N}(w,t_i)}{dt} - f(\mathcal{N}(w,t_i),t_i), \right)^2 \qquad (3.3)$$

or we can parametrize the function to learn on, encoding the initial condition in our neural network solution by defining the function

$$g(w,t) = u_0 - t\mathcal{N}(w,t). \qquad (3.4)$$

This ensures that the initial condition is satisfied and allows us to write down a simpler version of the loss function:

$$\mathcal{L}(w) = \sum_i \left( \frac{dg(w,t_i)}{dt} - f(g(w,t_i),t_i) \right)^2. \qquad (3.5)$$

Now that we have defined our problem, we can use standard minimization techniques to find a solution. For more information and further examples see [14].

### 3.1. Example ODE problem solved with a PINN

Consider the equation

$$u' = e^t \cos t \qquad (3.6)$$

with the initial condition $u(0) = 0.1$. Notice that we can find the solution to this equation via integration, i.e., the solution is given by

$$u(t) = \int e^t \cos t \, dt. \qquad (3.7)$$

Applying integration by parts twice, we find the solution

$$u(t) = \frac{1}{2}e^t(\sin t + \cos t) + C \qquad (3.8)$$

where $C = -0.4$ via use of the initial condition.

Let's now solve using the techniques defined above using `Julia` and the machine learning package `Flux.jl`. We begin by defining a neural network with two hidden layers one of 64 nodes the other with 32 nodes using the tanh activation function and a linear output:

```
using Flux
NNODE = Chain(x -> [x], # Take in a scalar and transform it into an array
        Dense(1,64, tanh),
        Dense(64, 32, tanh),
        Dense(32,1),
        first) # Take first value, i.e. return a scalar
```

We then parametrize the solution and define the loss function in which the differentiation is computed via discretization:

```
g(t)  = t*NNODE(t) + 0.1

using Statistics
```

```
ε = sqrt(eps(Float32))
loss() = mean(abs2(((g(t+ε)-g(t))/ε) - (exp(t) * cos(t))) for t in 0:1f-2:5f0)
```

Notice that we use machine $\epsilon$ here to discretize our integral as this is the smallest possible discretization before floating point error disrupts the solution. We will use gradient descent to solve with a learning rate of $10^{-4}$ over $10^4$ epochs. Because there is no data we are providing to train on, we feed the network empty data. Instead, all of the learning information comes from the loss function defined above. This is accomplished with the following code:

```
epochs = 10_000
learning_rate = 1e-4

opt = Flux.Descent(learning_rate) # use gradient Descent
data = Iterators.repeated((), epochs) # no data here : repeat empty data
iter = 0
loss_history = []
cb = function () #callback function to observe training
  global iter += 1
  push!(loss_history, loss())
  if iter % 500 == 0
    display(loss())
  end
end
display(loss())
Flux.train!(loss, Flux.params(NNODE), data, opt; cb=cb)
```

We can then see how our network does in comparison to the true solution as shown in the below plot:
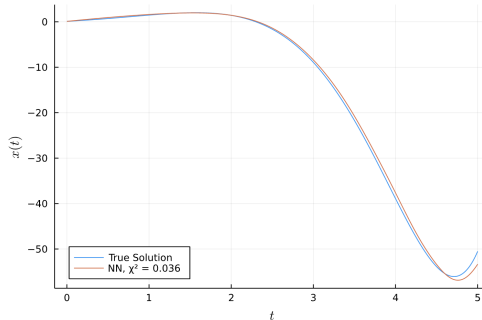


and we see that our network does pretty well! We could, of course, improve performance by training longer or playing with the size of the network or learning rate, however the above remains a good example of how to utilize neural networks to solve ODEs as is.

## 4. Solving PDEs with Physics-Informed Neural Networks

We now turn our attention to solving partial different equations using techniques from scientific machine learning. We will use the package `NeuralPDE.jl` [15] 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 $\Omega$, $u(x)$ is the solution, $f$ is some non-linear function acting on $u$, and $\lambda$ are the parameters of the equation. To solve with a neural network $\mathcal{N}(x, w)$ where $\mathcal{N}$ is a neural network with weights $w$, we want to fulfill

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

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

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

We have suggestively named the error $\mathcal{L}$ as we can use this function as a loss function in training our neural network $\mathcal{N}$ by minimizing $\mathcal{L}$. Our computational problem then reduces to the problem of evaluating the integral in Eqn. 4.3 in order to perform minimization.

Notice that in practical problems, we apply boundary conditions $b_i$ on $\partial\Omega \in \Omega$ in order to obtain, ideally, a unique solution. In order to include this information in the network, we add in these conditions to our loss function giving us

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

where we have generalized the problem to a system of coupled differential equations $f_i$. If we can find a way to efficiently evaluate this integral, then we can apply standard minimization algorithms to solve.

### 4.1. Methods of discretization

There are several different methods for evaluating the integral in Eqn. 4.4 namely: grid approximation techniques, stochastic grid approximation techniques, and machine learning (quadrature) techniques. A simple grid approximation takes the space $\Omega$ and divides it into units of volume $\Delta V$ with a specific length. The PDE is evaluated at each of this points and scaled by the volume, thus our integral is computed via

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

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., Monte Carlo techniques to evaluate the integral. This can be described by

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

for some scaler $\alpha$. As we simply want to minimize this integral, there is no need to specify $\alpha$. 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 [16]. These are typically of the form

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

In the implementation via `NeuralPDE.jl`, this is accomplised using the `Integrals.jl` package which calls the `Julia` differential equation solver [17] to find the correct sampling points $x_i$ and weights $\alpha_i$ in an implementation of Gaussian quadrature rules.

### 4.2. 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.

Consider the homogeneous system of linear PDEs given by

$$\partial_t u + \partial_x v + (v + u) = 0$$
$$\partial_x u + \partial_t v + (v + u) = 0 \qquad (4.8)$$

subject to the initial conditions

$$u(0, x) = \sinh x, \qquad v(0, x) = \cosh x. \qquad (4.9)$$

By means of the so-called Homotopy analysis method, the authors in [18] find the analytical solution to this system to be given by

$$u(x, t) = \sinh(x - t), \qquad v(x, t) = \cosh(x - t). \qquad (4.10)$$

We now solve with `NeuralPDE.jl`, describing important aspects of the code. To begin, we set up the system symbolically with the aid of `Modelingtoolkit.jl` as follows:

```
@parameters t x
@variables u(..) v(..)

Dx = Differential(x)
Dt = Differential(t)

eqns = [Dt(u(t,x)) - Dx(v(t,x)) + (u(t,x) + v(t,x)) ~ 0,
        Dt(v(t,x)) - Dx(u(t,x)) + (u(t,x) + v(t,x)) ~ 0]

bcs = [u(0,x) ~ sinh(x), v(0,x) ~ cosh(x)]

domains = [x ∈ Interval(0.0, 1.0),
           t ∈ Interval(0.0, 1.0)]

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

For systems of equations, store each equation as an element of the vector `eqn` as with the boundary conditions, `bcs`. We now define a neural network to solve the problem using the machine learning library `Lux.jl`. Our network has one hidden layer with 30 nodes, and takes two input variables—one for each parameter of the problem. We output a single number. Notice that each variable $u, v$ of our problem gets its own network. This is implemented simply as

```
dim = length(domains) # number of dimensions
n = 30
chains = [Lux.Chain(
            Dense(dim, n, Lux.σ),
            Dense(n, n, Lux.σ),
            Dense(n, 1)) for _ in 1:2]
```

We then discretize the system using quadrature training as follows

```
strategy = QuadratureTraining()
discretization = PhysicsInformedNN(chains, strategy)
@time prob = discretize(pde_sys, discretization)
```
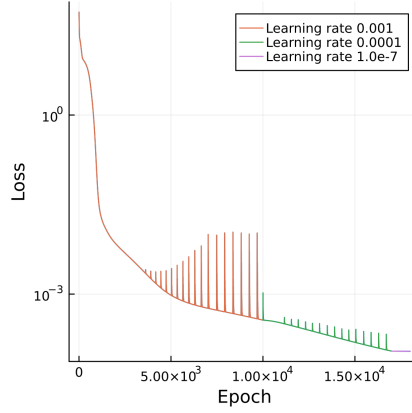
*Figure 2. Value of loss as a function of epoch for the linear system of homogenous PDEs. Three stages of learning are represented with the learning rate labeled. While there are some instabilities in the loss, we ultimately find a nice solution with the final learning rate of $10^{-7}$.*

After this is set-up and a loss function is defined through calling `PhysicsInformedNN`, we simply need to train the network to find a nice solution. This is done with the `Optimization.jl` library. We choose the ADAM optimizer over three learning rates: $10^{-3}, 10^{-4}$, and $10^{-7}$ over 10000, 7000, and 1000 epochs respectively. By training the network with successively smaller learning rates, we can improve the accuracy of the solution. This is accomplished in the following code:

```
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

learning_rates = [1e-3, 1e-4, 1e-7]
res = @time Optimization.solve(prob, ADAM(learning_rates[1]); callback =
    callback, maxiters=10000)
loss_history1 = loss_history
loss_history = []
prob = remake(prob, u0=res.minimizer)
res = @time Optimization.solve(prob, ADAM(learning_rates[2]); callback =
    callback, maxiters=7000)
loss_history2 = loss_history
loss_history = []
prob = remake(prob, u0=res.minimizer)
res = @time Optimization.solve(prob, ADAM(learning_rates[3]); callback =
    callback, maxiters=1000)
loss_history3 = loss_history
loss_history = vcat(loss_history1, loss_history2, loss_history3)
phi = discretization.phi
```

The loss is displayed in Fig. 2 and the results of the above learning are shown in Fig. 3. Notice the instability of the loss at higher learning rates. Fixing these instabilities is accomplished with a lower learning rate as this requires that over each epoch movement in loss-space is limited. The results demonstrate that our learning was successful.
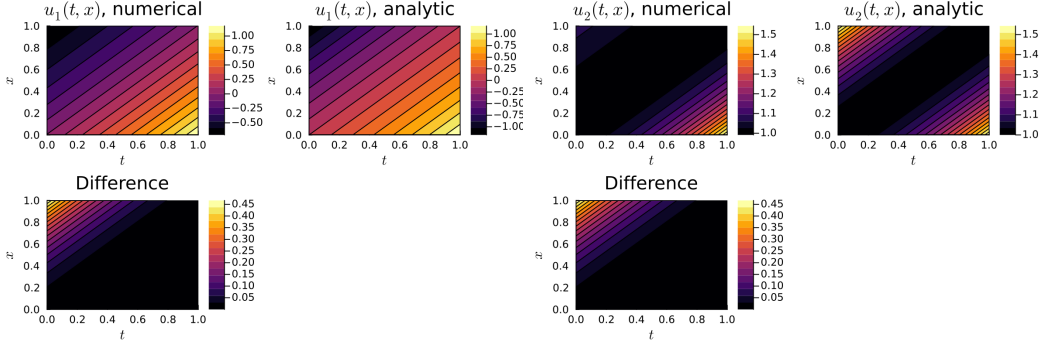
**Figure 3.** *Results of training a neural network to learn the solution of the linear homogenous PDE. Solution u is represented by $u_1$ and v by $u_2$. In both cases, numerical and analytic solutions are shown as well as the difference between numerical and analytic defined by $|u_{numerical} - u_{analytic}|$.*

### 4.3. Solving Similarity Solution of Navier-Stokes using `NeuralPDE.jl`

In the 1994 paper investigating the Blow-up of solutions to Navier-Stokes equations, Budd *et al.* consider the system of PDEs arising from the Navier-Stokes equations under the geometry of an infinite channel [19]. In particular, they explore the parabolic partial differential equations perturbed from a similarity solution of the Navier-Stokes given by

$$\partial_t u_1 + \mu u_2 \partial_x u_1 = \partial_x^2 u_1 + u_1^2 - K^2, \quad 0 < x < 1, 0 < t < T$$
$$\partial_x u_2 = u, \quad 0 < x < 1, 0 < t < T \tag{4.11}$$

with the boundary conditions

$$\partial_x u_1(t, 0) = \partial_x u_1(t, 1) = u_2(t, 0) = u_2(t, 1) = 0 \tag{4.12}$$

and initial condition

$$u_1(x, 0) = u_0(x). \tag{4.13}$$

These equations are parameterized by $\mu$, which perturbs the equations from Navier-Stokes. It is the goal of the paper to determine for which values of $\mu$ the solutions "blow up." We will consider exclusively the case where $\mu = 1$, as this has a nice solution and the resulting system arises from the Navier Stokes equations as discussed below:

#### 4.3.1. Problem Derivation
We consider the incompressible Navier-Stokes equations in an infinitely long channel in the $x$- and $z$-direction and finite in the $y$ direction; namely we bound $y$ by $0 < y < 1$. Let $u_1, u_2, u_3$ be velocity fields, $1/v$ be the Reynolds number, and $p$ be the pressure. Then, the Navier-Stokes equations read

$$\partial_t u_1 - u_1^2 + u_2 \partial_y u_1 = v \partial_y^2 u_1 - c(t)$$
$$\partial_t u_2 + u_2 \partial_y u_2 = v \partial_y^2 u_2 - \partial_y p$$
$$\partial_t u_3 - u_3^2 + u_2 \partial_y u_3 = v \partial_y^2 u_3 - e(t)$$
$$\partial_y u_2 = u_1 + u_3. \tag{4.14}$$

In order to simplify, let $u_3(y, t) = 0$, $e(t) = 0$ and $v = 1$ so that our system becomes

$$\partial_t u_1 - u_1^2 + u_2 \partial_y u_1 = v \partial_y^2 u_1 - c(t)$$
$$\partial_t u_2 + u_2 \partial_y u_2 = v \partial_y^2 u_2 - \partial_y p \qquad (4.15)$$
$$\partial_y u_2 = u_1.$$

Now note that $p(t)$ can be determined once we know $u_1, u_2$ from the above first and last equation as the middle equation of Eqn. 4.15 is only dependent on $u_2$. Hence, we write our system as

$$\partial_t u_1 + u_2 \partial_y u_1 = \partial_y^2 u_1 + u_1^2 - c(t)$$
$$\partial_y u_2 = u_1. \qquad (4.16)$$

which matches Eqn. 4.11 when $\mu = 1$ and under the condition that $y \to x$. In order to have a complete solution, we need only finding appropriate inital conditions. The condition $u_2(0, t) = u_2(1, t)$ follows naturally to disallow flow through the walls of the channel. In [19], they also employ the boundary condition $\partial_y u_1 = 0$ to obtain the above complete initial conditions. Finally, we require $c(t) \geq 0$ so that we can write $c(t) = K^2$.

### 4.3.2. Solution with PINN and comparison to analytical solution

Let's now solve this system! Notice that via integration by parts of the first equation in 4.11 we get

$$K^2 = 2 \int_0^1 u_1^2 \, dx. \qquad (4.17)$$

Thus, we solve the system

$$\partial_t u_1 = \partial_x^2 u_1 - u_2 \partial_x u_1 + u_1^2 - 2 \int_0^1 u_1^2 \, dx$$
$$0 = \partial_x u_2 - u_1 \qquad (4.18)$$

for $0 < x < 1$ and $0 < t < 1$. Using the same initial conditions as above, we set $u_0(x) = \cos \pi x$, coinciding with the analytical solution given in [20]. Using power-series methods, the authors find the analytical solution to the above system to be

$$u_1(t, x) = e^{-\pi^2 t} \cos \pi x$$
$$u_2(t, x) = \frac{1}{\pi} e^{-\pi^2 t} \sin \pi x. \qquad (4.19)$$

We can then solve the problem using `NeuralPDE.jl`. This differs from the above code only in the definition of an integral through `Modelingtoolkit.jl` which can simply be done with

```
Ix = Integral(x in DomainSets.ClosedInterval(0, 1))
```

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
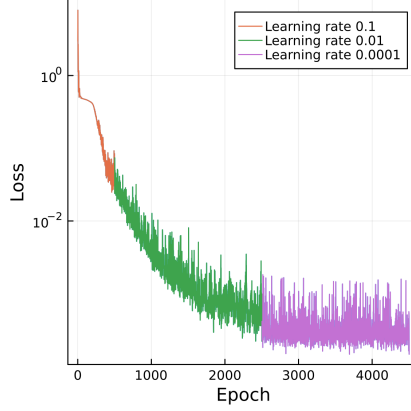
**Figure 4.** *Loss as a function of epoch in training a neural network to learn the solution of the integral PDE presented above. Learning is broken up into three sections with differing learning rate as shown.*
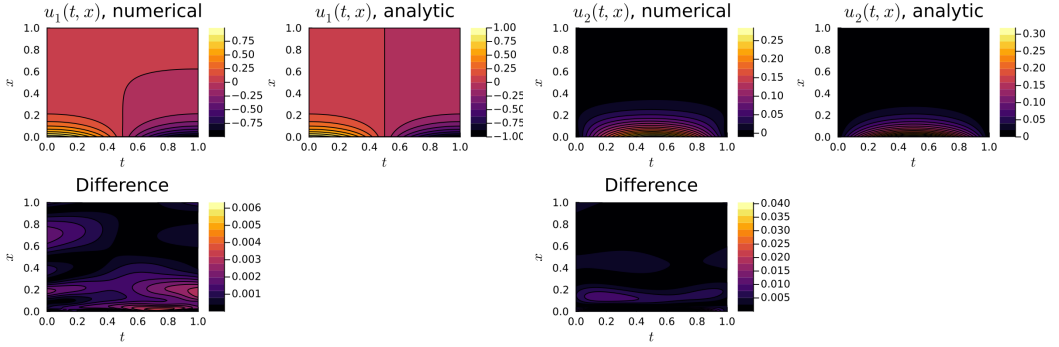


**Figure 5.** *Results of training a network to learn the solution of the integral PDE. Both numerical and analytical solutions are given with the difference between the two. We see that the numerical solution matches the analytic solution very well..*

our system, thus we define two neural networks with the set-up described above. The discretization we choose is the quasi random training with 100 points (ADD MORE DESCRIPTION!!).

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 can be seen in Fig. 4 and the solution can be seen in Fig. 5.

To see the complete implementation of the above, see the data availability statement.

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

We find a numerical solution to Einstein's field equations under a few assumptions to yield the Schwarzschild Metric as first described by Schwarzschild in 1916 [21].

### 5.1. Physics Background and Problem Set-up

Einsteins field equations are given by

$$R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu} \tag{5.1}$$

.

Under our assumptions (Define) this reduces to

$$R_{\mu\nu} = 0. \tag{5.2}$$

We define connecting coefficients as

$$\Gamma_{\mu\nu}^{\alpha} = -\frac{1}{2}\sum_{\beta} g^{\alpha\beta}\left(\frac{\partial g_{\mu\beta}}{\partial x_{\nu}} + \frac{\partial g_{\nu\beta}}{\partial x_{\mu}} - \frac{\partial g_{\mu\nu}}{\partial x_{\beta}}\right) \tag{5.3}$$

The solutions outlined in Swarzschild's paper yields the following differential equations to solve

$$\sum_{\alpha}\frac{\partial \Gamma_{\mu\nu}^{\alpha}}{\partial x_{\alpha}} + \sum_{\alpha\beta}\Gamma_{\mu\beta}^{\alpha}\Gamma_{\mu\alpha}^{\beta} = 0 \tag{5.4}$$

where we also require

$$|g_{\mu\nu}| = -1. \tag{5.5}$$

A full solution to this problem yielding the analytic solution in spherical coordinates

$$g_{\mu\nu} = \begin{pmatrix} 1 - \frac{r_s}{r} & 0 & 0 & 0 \\ 0 & -(1 - \frac{r_s}{r})^{-1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2\theta \end{pmatrix}, \tag{5.6}$$

where

$$r_s = \frac{2GM}{c^2}, \tag{5.7}$$

is given in [22].

### 5.2. Technical Implementation

### 5.3. Results and Discussion

## 6. Conclusion

# 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] "SciMLCon 2022." https://scimlcon.org/2022/.

[4] G. Strang, *Linear Algebra and Learning from Data*. Wellesley: Wellesley-Cambridge press, 2019.

[5] C. M. Bishop, *Pattern Recognition and Machine Learning*. Information Science and Statistics, New York: Springer, 2006.

[6] 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.

[7] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The Loss Surfaces of Multilayer Networks," Jan. 2015.

[8] Y. Chauvin and D. E. Rumelhart, eds., *Backpropagation: Theory, Architectures, and Applications*. New York: Psychology Press, Feb. 1995.

[9] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," Jan. 2017.

[10] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, Jan. 1989.

[11] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, pp. 251–257, Jan. 1991.

[12] Y. Lu and J. Lu, "A Universal Approximation Theorem of Deep Neural Networks for Expressing Probability Distributions," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 3094–3105, Curran Associates, Inc., 2020.

[13] F. Voigtlaender, "The universal approximation theorem for complex-valued neural networks," Dec. 2020.

[14] C. Rackauckas, "SciML/SciMLBook: Parallel Computing and Scientific Machine Learning (SciML): Methods and Applications (MIT 18.337J/6.338J)." https://github.com/SciML/SciMLBook.

[15] 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.

[16] 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.

[17] 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.

[18] A. Sami Bataineh, M. S. M. Noorani, and I. Hashim, "Approximate analytical solutions of systems of PDEs by homotopy analysis method," *Computers & Mathematics with Applications*, vol. 55, pp. 2913–2923, June 2008.

[19] C. J. Budd, J. W. Dold, and A. M. Stuart, "Blow-up in a System of Partial Differential Equations with Conserved First Integral. Part II: Problems with Convection," *SIAM Journal on Applied Mathematics*, vol. 54, pp. 610–640, June 1994.

[20] B. Benhammouda and H. Vazquez-Leal, "Analytical solutions for systems of partial differential–algebraic equations," *SpringerPlus*, vol. 3, p. 137, Mar. 2014.

[21] K. Schwarzschild, "On the gravitational field of a mass point according to Einstein's theory," May 1999.

[22] eigenchris, "(1) Relativity 108a: Schwarzschild Metric - Derivation - YouTube." https://www.youtube.com/.