A Meshfree Deep Learning Approach for Numerical Solution of Differential Equations with Implementation in Python

Christopher Denq, Liang Kong

Department of Mathematical Sciences and Philosophy
University of Illinois Springfield
Springfield, IL 62703

Abstract

Differential equations provide powerful mathematical models across scientific disciplines, enabling researchers to describe complex dynamic systems. However, analytical solutions are often intractable, necessitating numerical techniques. This paper explores recent advances in solving differential equations using a scientific machine learning approach, physics-informed neural networks (PINNs). PINNs integrate scientific knowledge in the form of mathematical equations directly into the deep learning training process. By encoding physics-based constraints through automatic differentiation, PINNs can efficiently produce accurate solutions while avoiding limitations of traditional discretization-based methods. After introducing the mathematical foundations and architecture of PINNs, step-by-step Python implementation details are provided for two canonical examples. Results demonstrate PINNs' ability to incorporate both data and physical laws to solve differential equations. The emerging intersection of scientific computing and artificial intelligence holds immense potential to advance modeling and simulation across applications. This paper aims to introduce the PINN paradigm to applied mathematics and data science students, fostering intuition and preparing the next generation of researchers to leverage modern techniques. All code is available on GitHub to facilitate reproducibility and hands-on learning.

Keywords

Differential Equations, Numerical methods, Scientific machine learning, Physics-informed Machine learning

1 Introduction

Differential equations have a significant impact in science and engineering, providing mathematical models of dynamic processes and systems across disciplines. From Newton's law of cooling to

the wave equation in quantum mechanics, differential equations capture the interrelated change of quantities. While analytical solutions to differential equations enable precise analysis, numerical techniques are often necessitated by complexity. This paper explores recent advances in solving differential equations using machine learning, an emerging interdisciplinary approach.

Neural networks, inspired by biological neurons, can approximate solutions to nonlinear systems through iterative adjustment of weights and biases ((Higham & Higham, 2019; Poggio, Mhaskar, Rosasco, Miranda, & Liao, 2017)). Scientific machine learning synergistically integrates expertise from computational science and computer science to develop innovative machine learning techniques suited for complex problems in scientific and engineering settings (Lagaris, Likas, & Fotiadis, 1998). Physics-informed neural networks (PINNs) incorporate physical laws and constraints directly into the loss function, allowing incorporation of prior scientific knowledge during training ((Raissi, Perdikaris, & Karniadakis, 2019)). By integrating data with underlying governing equations, PINNs can rapidly produce accurate solutions even with limited data. These meshfree techniques forego traditional discretizations, circumventing dimensionality limitations of numerical methods.

Exposing students to emerging methods like PINNs is critical for preparing the next generation of researchers. PINNs reside at the intersection of scientific computing, differential equations, and deep learning - three pillars of modern applied mathematics. By mastering PINNs, students gain versatile skills that translate across disciplines.

PINNs also provide an engaging pedagogical opportunity. Unlike traditional mesh-based techniques like finite difference and finite element methods, deep learning can act as a mesh-free approach through automatic differentiation (Lu, Meng, Mao, & Karniadakis, 2021). This allows it to overcome the curse of dimensionality that affects other methods (Poggio et al., 2017; Grohs, Hornung, Jentzen, & von Wurstemberger, 2023). But PINNs enable hands-on experiments that illuminate core concepts like convergence, consistency, and stability. Students can tweak network architecture and training parameters and directly observe the impact on solution accuracy. Active learning deepens intuition about foundational ideas.

For educators, PINNs offer an interactive way to demonstrate foundational concepts like stability, convergence, and consistency. Rather than relying solely on theoretical explanations, instructors can have students actively experiment with modifying the PINN architecture and training parameters to observe the impact on model accuracy and error. This fosters intuitive understanding through hands-on exploration, which is crucial for differential equation learning (Lozada, Guerrero-Ortiz, Coronel, & Medina, 2021; Spooner, 2024). PINNs also promote connecting analytical methods, numerical techniques, data science, and machine learning in a unified framework. Students can gain a holistic perspective on modern applied mathematics techniques.

For students, PINNs provide an intuitive grasp of abstract concepts in numerical analysis, as limitations of discretization methods become tangible through direct comparisons to the mesh-free PINN approach. Coding PINNs from scratch in Python (Rossum & Jr, 1995) cements comprehension through experiential learning. Students are empowered to explore cutting-edge

techniques and contribute novel research prior to graduation.

By embracing PINNs as a core component of applied mathematics curriculum, instructors can stimulate enduring learning, promote research, and forge interdisciplinary connections. PINNs bring differential equations to life for students, preparing them to utilize modern techniques for modeling the world. The hands-on and research opportunities unlocked by PINNs are transformative for both differential equations teaching and learning.

One purpose of this paper is to make physics-informed neural networks (PINNs) accessible to students with minimal Python coding experience. Step-by-step Python implementation details are provided to demonstrate application of PINNs for solving canonical differential equations. By scaffolding the coding examples, readers with limited programming backgrounds will be able to follow along and gain hands-on experience with PINN modeling.

The remainder of this paper is structured as follows. Section 2 provides mathematical background on the foundations and general architecture of physics-informed neural networks (PINNs), as well as discussion of the strengths and limitations of this emerging technique. Section 3 contains detailed, line-by-line Python code examples that walk through using PINNs to solve two canonical differential equations. Section 4 presents results and concluding remarks, including a brief synopsis of potential directions for future work. For reproducibility and reader benefit, all code implementations from this paper will be publicly available on GitHub upon publication[1].

2 Physics-Informed Neural Networks (PINNs) solving differential equation

This section begins with a brief overview of deep neural networks and automatic differentiation, as well as presenting the algorithm and theory behind using PINNs to solve differential equations.

2.1 Mathematical Background

A deep neural network can be viewed as a special type of compositional function. The most basic deep neural network is the feed-forward neural network (FNN). This is also called a multilayer perceptron (MLP). It works by applying a series of linear and nonlinear transformations to the input data. Each layer transforms the output from the previous layer, doing this recursively through the network.

Define $\mathcal{N}^L(x) : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ as an L-layer neural network with N_ℓ neurons in the ℓth layer, here $N_0 = d_{\text{in}}$, $N_L = d_{\text{out}}$. We denote the weight matrix and bias vector in the ℓth layer by $W^\ell \in \mathbb{R}^{N_\ell} \times \mathbb{R}^{N_\ell - 1}$ and $b^{N_\ell} \in \mathbb{R}^{N_\ell}$, respectively. We involve an activation function, σ, to introduce non-linearity into the output of a neuron. The most commonly used activation functions are the sigmoid, the hyperbolic tangent, and the rectified linear unit. We define FNN as the following formula:

[1](GitHub repository: https://github.com/cdenq/pinns-teaching-uis)

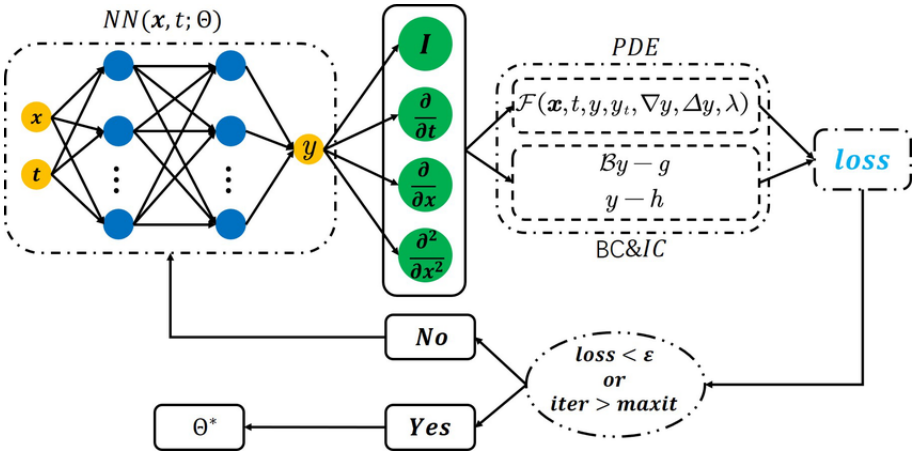

Figure 1: The architecture layout of a Physics-Informed Neural Network.

$$
\begin{aligned}
\text{input layer:} \quad & \mathcal{N}^0(x) = x \in \mathbb{R}^{d_{\text{in}}}, \\
\text{hidden layers:} \quad & \mathcal{N}^\ell(x) = \sigma(W^{(\ell)}\mathcal{N}^{\ell-1}(x) + b^\ell) \in \mathbb{R}^{N_\ell} \quad \text{for } 1 \le \ell \le L-1, \\
\text{output layer:} \quad & \mathcal{N}^L(x) = W^L\mathcal{N}^{L-1}(x) + b^L \in \mathbb{R}^{d_{\text{out}}}
\end{aligned}
$$

The neural network architecture in PINNs typically consists of multiple hidden layers with activation functions. The network takes inputs such as spatial coordinates or time and outputs the solution of the differential equation. The loss function is a combination of data-fitting terms and regularization terms that enforce the differential equation constraints. During training, the neural network is optimized to minimize the loss function. By incorporating the physics-based constraints, PINNs learn to approximate the solution to the differential equation, capturing the underlying dynamics. The algorithm of PINN (Raissi et al., 2019) is shown visually in the schematic of Figure 1.

2.2 Advantages and Limitations

PINNs have several advantages. They can handle noisy or sparse data, adapt to complex geometries, and learn from limited data. PINNs have been successfully applied to various problems, including fluid dynamics, heat transfer, structural mechanics, and image reconstruction. They offer an alternative to traditional numerical methods, providing accurate and efficient solutions to differential equations, especially in scenarios where analytical solutions are not available or computationally expensive.

Physics-Informed Neural Networks bridge the gap between data-driven machine learning and physics-based modeling. By integrating the governing physics into the learning process, PINNs combine the flexibility of neural networks with the constraints of the differential equations, enabling accurate and physics-consistent solutions to a wide range of differential equation problems.

3 Simulation in Python

Along with the selection of Python, Harmonic oscillator equation (Moseley, 2021) and residual-based adaptive refinement (RAR) method (Lu et al., 2021) on the Burgers equation serve as great case studies for PINNs in an applied setting. This paper will reimplement these examples and remark on their various educational teaching points.

In this section, we will briefly overview Python and its setup in Section 3.1, code a simulation of the harmonic oscillation in Section 3.2 by using PyTorch, and code a RAR of the Burgers equation in Section 3.3 by using DeepXDE.

3.1 Setting Up Python

3.1.1 What is Python?

Python is a popular programming language known for its versatility in areas like web development or software engineering, but most notably for machine learning and data science. Extensively adopted within academia and industry alike, Python boasts a vast library ecosystem that encompasses modules, datasets, and comprehensive documentation. The Python Package Index (PyPI) is the principal hub for Python packages, hosting over 300,000 libraries ready for deployment.

Python and its extensive library support present many opportunities to teach PINNs, deep learning, and differential equations. Along with its more human-legible syntax, Python also furnishes specialized libraries (e.g. Pandas, Numpy, Matplotlib, and PyTorch) that are optimized for the entire machine-learning workflow: from data pipelines to model training to deployment. By leveraging Python, educators can readily teach students both applied and theoretical aspects of topics in mathematics or computer science, easily visualize results through its extensive graphing capabilities, and even quickly abstract computer science processes into beginner-friendly functions. Educators can pursue any level of granularity with Python code, and such versatility coupled with its robust and widespread use, make Python an ideal choice for teaching these topics.

3.1.2 Python on Local Machines

Setting up Python and virtual environments on your local machine affords you full control over your development environment, ensuring privacy, persistent storage, and seamless integration with local files and hardware. It is particularly beneficial for long-term projects requiring significant customization, as you can maintain, modify, and optimize your environment without restrictions. However, the downside includes the manual management of dependencies, potential difficulty in replicating environments, and the need for your hardware to support certain computational tasks, which can be costly if dealing with high-performance requirements. If you prefer setting up your workspace on remote servers, then skip to Section 3.1.3.

For the rest of this section, the terms "packages", "libraries", and "dependencies" are used interchangeably to mean any external code modules or scripts. An example of an external code

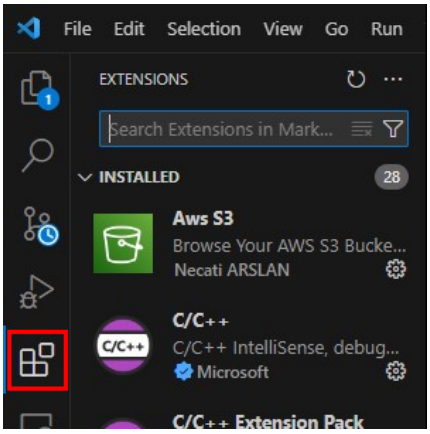

Figure 2: The icon boxed in red is the "Extensions" panel in Visual Studio Code. The image above is taken from a preexisting VS Code environment, so your image may not directly match this.

module or script would be "Pandas" or "DeepXDE", which are installed later in this section.

To get started, visit the official Python website to download Python and then run the integrated installer. Depending on your operating system (Windows, macOS, or Linux), you'll see a recommended version for download. During installation, check the box that says "Add Python 3.x to PATH"; this will allow you to run Python from the terminal. To verify a successful installation, open your terminal and run the following command.

Windows:

```
python --version
```

macOS and Linux:

```
python3 --version
```

If a version number is returned, then the installation was successful.

For code development and debugging, utilizing an integrated development environment (IDE) is heavily recommended. Visual Studio Code (VS Code) is a contemporary, extensible IDE that caters to multiple programming languages. Download the systems-appropriate version of the installer from its official website and proceed with its installation.

Once finished, navigate to "Extensions" within VS Code to unlock this IDE's full potential. Extensions such as IntelliSense are additional features created by other creators or organizations (e.g. Microsoft) that are officially supported and provide enhanced functionality for Python. You can navigate to the Extensions view by clicking on its icon in the Activity Bar on the left side of the VS Code window or the View: Extensions command `(Ctrl+Shift+X)` (Figure 2).

Two extensions recommended for installation are "Jupyter" and "Python" (Figure 3), both of which are developed and maintained by Microsoft. Click their installation buttons and let them run to completion. As part of this installation, other dependencies like "Pylance" are also installed. You may need to restart VS Code to complete this process.

Figure 3: "Jupyter" and "Python" extensions (February 2024).

With the IDE setup completed, virtual environments can be easily set up using conda (automatically installed when you installed Python), as well. Virtual environments help isolate project-specific dependencies, which in turn allow users to swap between these environments to fit the task at hand. For example, if project 1 requires Python packages A to D while project 2 requires packages C to F, having just one virtual environment containing packages A to F is unnecessary. It is more advisable to have separate environments tailored with exactly the necessary packages for each project. This reduces library bloat (since users only need to load the libraries they need), maintains project consistency, and reduces the potential for library-dependency conflicts.

To create a new virtual environment, open the Integrated Terminal within VS Code by using the `(Ctrl+‘)` command or going to "View" > "Terminal" at the top of the VS Code window (Figure 4). Type the following command to create a new virtual environment called `"example"`. You may optionally replace `[version]` with your specific Python version, though if you have no preference, you can omit `python=[version]` entirely to automatically install the most recent Python version. Both methods are shown below. When the command has been entered into the terminal, you will receive a prompt confirming the installation with a `(y/n)` prompt; type in `y` to continue.

```
conda create --name example python=3.11
```

```
conda create --name example
```

Conda, a package installer for Python, is typically included in modern distributions of Python. It simplifies the process of installing and managing additional libraries or dependencies. With conda, we can effortlessly incorporate essential deep-learning libraries. Note: these libraries are not the same as the Extensions we previously downloaded (though they sometimes bear the same name).

- Jupyter for working with `".ipynb"` files
- Numpy for numerical computations
- Matplotlib for data visualization
- PyTorch for modular customization of deep learning

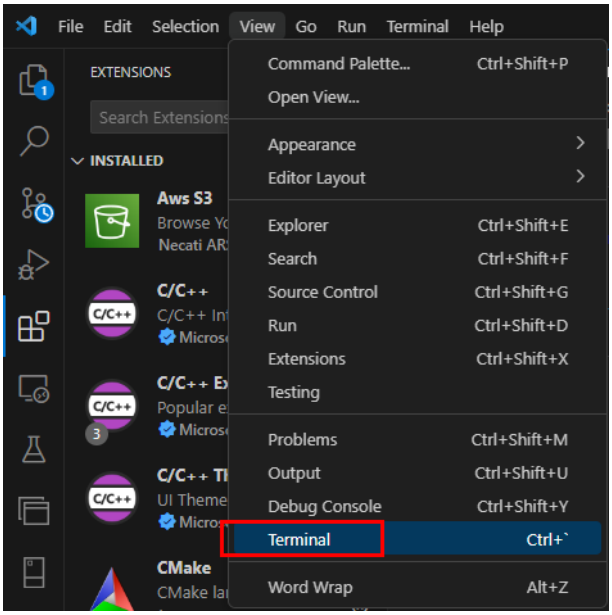

Figure 4: Manual method to access the Integrated Terminal within Visual Studio Code.

- DeepXDE for packaged functions in deep learning

To get started, activate your `example` virtual environment within your terminal. This virtual environment should be the environment in which you want to install the libraries.

```
conda init
conda activate example
```

If the above commands do not work on your machine, you can also try this command below.

```
source activate example
```

If this activation step is not done, conda will install the packages to the "base" environment, which represents the default overall working environment. It is generally a good practice to keep your packages within specific environments rather than all in "base", as it makes it easier to update old packages and troubleshoot inter-package conflicts should they arise in the future.

Next, input the following command, where the text following `install` and delimited by whitespaces represents the names of the various libraries being installed. Let the installations run to completion.

```
conda install jupyter numpy matplotlib
```

Next, we can create a starter file to begin coding. Navigate to the Explorer pane in VS Code or input the hotkey `(CTRL+SHIFT+E)`. Then, navigate to any location on your local machine in which you want the starter file to exist. For this demonstration, we will use an example folder titled "PINNs-Example" and open it; within the "PINNs-Example" folder, we can then right-click in the region to reveal a menu that would allow us to create a new file to house our Python code or click the "New File" icon (all steps shown in Figure 5). Create a new file, give it the

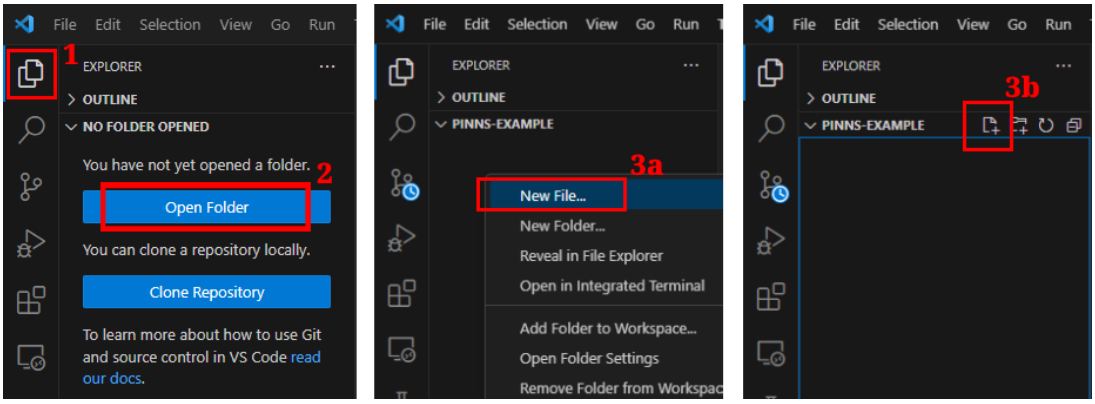

Figure 5: Navigate to the Explorer pane, open a folder (in this case the "PINNs-Example" folder), and then create the `example_file.ipynb` file within it by right-clicking within the folder pane (3a) or clicking the "New File" icon (3b).

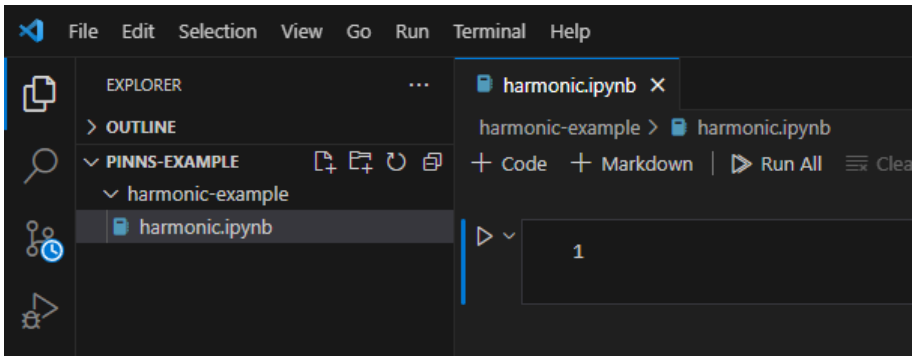

Figure 6: If done correctly, the file `harmonic.ipynb` has been created within the "harmonic-example" folder, which itself is contained by the "PINNs-Example" folder.

name `harmonic`, and give it the file extension `.ipynb` to designate it as an Interactive Python Notebook file (also called Juypter notebook file). More veteran users of Python may prefer `.py` for traditional Python files, but for this demonstration, we will stick to `.ipynb` for convenience. If done correctly, we should see our new file within the Explorer pane. Following similar steps to making a new file, we can also make a new folder within "PINNs-Example" called "harmonic-example" to better organize our files (this will be utilized later on in the paper) and place the recently created `harmonic.ipynb` file within. The resulting folder structure can be seen below (Figure 6).

To start programming in this new `harmonic.ipynb` file, we will need to designate the proper kernel within VS Code. VS Code works with many languages beyond just Python, and thus, it needs to know when to interpret a block of text as Python code, C++ code, LaTeX, or even just plain text. As such, we can define our code to be interpreted as Python code by selecting the virtual environment we just created as a kernel. Use the command `(Ctrl+Shift+P)` to open the Command Palette. Start typing "Python: Select Interpreter" in the Command Palette and

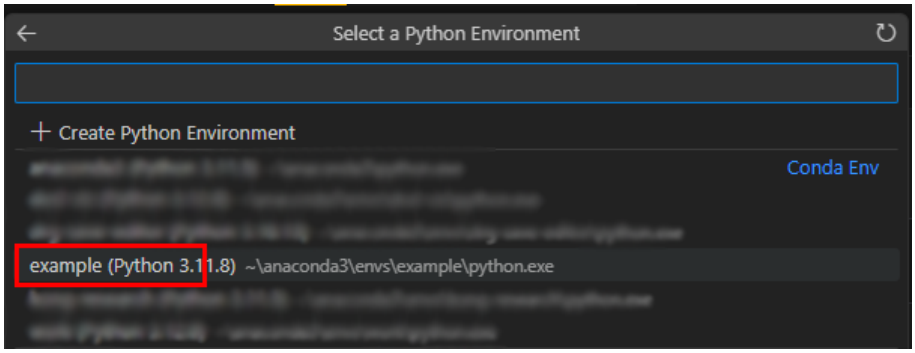

Figure 7: The kernel selection menu in VS Code. The image above is taken from a preexisting VS Code environment, so your menu may not directly match this.

select that option to open a menu of all available Python kernels (Figure 7). Select the desired virtual environment from the list, which in this case is the `example` virtual environment. If you do not see your newly created environment there, be sure to restart your VS Code.

You are now ready to start programming in Python on your local machine!

3.1.3 Python on Remote Servers

Leveraging servers for computational processes poses advantages over working on your local machine. Google Colab is a streamlined process because Colab provides a host of Google computing resources (like GPUs) for free, which are particularly useful for intensive computations required by ML processes. Google Colab simplifies the setup by providing a pre-configured environment, while simultaneously enhancing collaboration, version control, and environment sharing (everything is cloud-based). However, it is not without its limitations: you are subjected to usage quotas, limited session times, and potential internet dependency issues. Moreover, there is less control over the environment's specifics, and concerns regarding data security and privacy, especially with sensitive information, as your data is uploaded to the cloud. The temporary nature of the storage means additional steps to ensure your data and environment settings are consistently backed up if needed for future sessions.

Visit Google Colab and sign in with a Google account. If you use Gmail, this should be the same account. Once you are signed in, you can create a new notebook by clicking on "File" > "New notebook". You will see a new tab with a fresh notebook, where you can write and execute Python code in code cells.

If you need high-intensity computational resources (like a GPU or TPU): Click on "Runtime" in the menu, choose "Change runtime type", and then in the pop-up, select "GPU" or "TPU" from the "Hardware accelerator" dropdown menu. Click "Save". This step is optional if you anticipate large computational workloads.

Google Colab comes with many popular libraries automatically installed (like Numpy and Matplotlib). However, in the cases in which you would need more customized libraries, you can

install them using the `!pip` install command. For instance:

```
!pip install torch torchvision torchaudio
!pip install deepxde
```

Every time you run a Colab notebook, Google creates a new virtual environment for your run instance, which gets recycled when the session ends. This means there is no virtual environment setup needed (like the local machine case), but the tradeoff is that you will need to reinstall libraries like Tensorflow every time you want to run the code again.

You are now ready to work remotely in Colab!

3.2 Harmonic Oscillator Simulation

3.2.1 Process Overview

PINNs prove useful in approximating non-linear ordinary differential equations such as the harmonic oscillation equation. In the section below, we will generate the exact solution to the harmonic oscillation motion, extract a small sample of points from that solution, train both a conventional neural network and a PINN to approximate the harmonic oscillation solution, and then compare the results.

3.2.2 Code Setup and Model Definitions

Following the Moseley example, we first create our virtual environment, specifying that we want to use Python 3. Navigate to your terminal and create a new virtual environment with the commands below. (If you followed the steps in Section 3.1, the `example` environment will not be used in this section.)

```
conda create -n harmonic-pinns python=3
conda activate harmonic-pinns
```

Next, we will install the necessary libraries using conda.

```
conda install jupyter numpy matplotlib
conda install pytorch torchvision torchaudio -c pytorch
```

From Section 3.1.2 "Python on Local Machines", we should have the `harmonic.ipynb` file open and the `example` kernel selected. We can easily swap kernels to the `harmonic-pinns` virtual environment we just created by following the same steps outlined previously (Figure 7) or by clicking the kernel icon on the right-hand side of VS Code (Figure 8).

Within the first cell block, add the following imports and run the code. We can run a cell with these two methods: (1) We can type in the hotkey `(SHIFT+ENTER)` to run the currently highlighted cell block OR manually click the arrow boxed in red, or (2) manually click the "Run All" functionality boxed in red to run all cell blocks in the notebook (Figure 9).

```
from PIL import Image
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

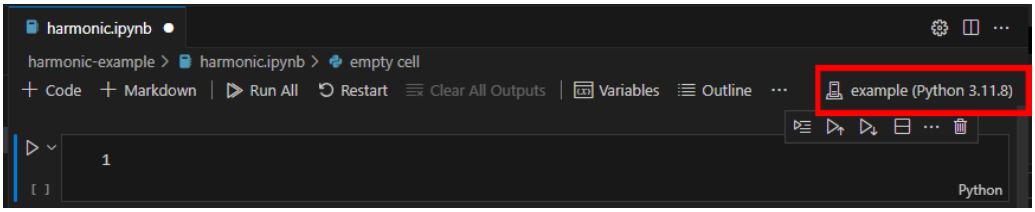

Figure 8: To swap kernels once you have one already loaded, you can also click the kernel icon on right-hand side of VS Code to bring up the menu of all found kernels. This icon also displays the currently loaded kernel.

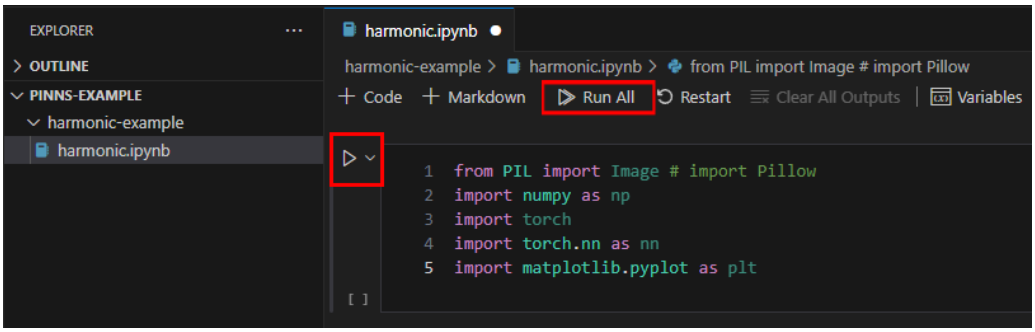

Figure 9: There are two methods to run code in a notebook: (1) running individual cell blocks using the arrow (in the red square box) or the hotkey `(SHIFT+ENTER)` OR (2) running the entire notebook by clicking "Run All".

Next, we will implement some custom functions and classes to assist with modeling. We can also add each segment of code shown in the paper into its own cell block within VS Code; a new cell block will be created automatically if the hotkey `(SHIFT+ENTER)` is used on a previous cell block or we can manually create a new cell block by clicking the `+ Code` icon at the top of our VS Code window (Figure 10).

Now, consider the mathematical equation for a one-dimensional damped harmonic oscillator:

$$m\frac{d^2y}{dx^2} + \mu\frac{dy}{dx} + ky = 0, \tag{3.1}$$

with initial conditions

$$y(0) = 1, \quad \frac{dy}{dx} = 0. \tag{3.2}$$

We can assert the system as under-damped with:

$$\delta < \omega_0, \quad \text{with} \quad \delta = \frac{\mu}{2m}, \quad \omega_0 = \sqrt{\frac{k}{m}}, \tag{3.3}$$

which then gives us the exact solution:

$$y(x) = e^{-\delta x}\left(2A\cos(\phi + \omega x)\right), \quad \text{with} \quad \omega = \sqrt{\omega_0^2 - \delta^2}. \tag{3.4}$$

where

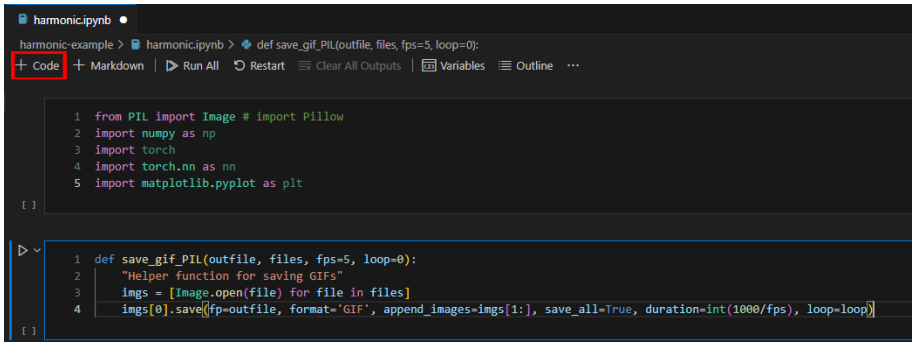

Figure 10: To create a new cell block, we can run the `(SHIFT+ENTER)` hotkey on a previous cell block or we can click the `+ Code` button.

- $y(x)$ is the displacement of the system from the equilibrium position as a function of time x. We use x to denote time (typically t) and y to denote displacement (typically x) due to coding conventions that will be more clear later in the code.
- δ is the damping coefficient, which quantifies the rate at which the oscillation amplitude decreases over time due to non-conservative forces like friction or air resistance.
- A is the amplitude of the oscillations, representing the maximum extent of the displacement from the equilibrium position.
- ϕ is the phase constant (or initial phase), which determines the oscillator's displacement at time $x = 0$.
- ω is the angular frequency of the oscillation, which is related to the natural frequency of the system and determines how rapidly it oscillates.

We can codify this equation in the following function, representing the first step in injecting physical laws into machine learning processes.

```
def oscillator(d, w0, x):
    assert d < w0
    w = np.sqrt(w0**2-d**2)
    phi = np.arctan(-d/w)
    A = 1/(2*np.cos(phi))
    cos = torch.cos(phi+w*x)
    exp = torch.exp(-d*x)
    y  = exp*2*A*cos
    return y
```

We first assert the system to be underdamped by setting the damping ratio `d` as less than the natural frequency `w0`. We then calculate the damped angular frequency `w`. This is derived from the square root of the difference between the square of the natural frequency and the square of the damping coefficient `d`. By taking the arctangent of the ratio of the negative damping coefficient to the damped angular frequency, we can also determine the phase constant `phi`. Through similar analytical methods, we can derive the amplitude `A` in the same way.

Then, we calculate the cosine of the sum of the phase constant and the product of the damped angular frequency and time to represent the oscillatory component of the system's displacement. The next line containing `exp` computes the exponential decay factor on the oscillation. The resulting displacement of the underdamped oscillator `y` is then returned.

We can now move on to the main modeling component for the harmonic oscillator demonstration. Specifically, the code here is implemented via an object-oriented programming (OOP) paradigm, which is a coding format or style that organizes functions underneath classes. Details of the OOP paradigm are beyond the scope of this paper, but what is relevant is that it is being used here and has a programming structure different from normal functions.

```
class FCN(nn.Module):
    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS):
        super().__init__()
        activation = nn.Tanh
        self.fcs = nn.Sequential(*[
                        nn.Linear(N_INPUT, N_HIDDEN),
                        activation()])
        self.fch = nn.Sequential(*[
                        nn.Sequential(*[
                            nn.Linear(N_HIDDEN, N_HIDDEN),
                            activation()]) for _ in range(N_LAYERS-1)])
        self.fce = nn.Linear(N_HIDDEN, N_OUTPUT)

    def forward(self, x):
        x = self.fcs(x)
        x = self.fch(x)
        x = self.fce(x)
        return x
```

The custom class `FCN` represents a fully connected neural network. It is built upon the `nn.Module` base class provided by PyTorch, which we had previously imported.

In the `__init__` method, the network's architecture is established. The parameters `N_INPUT`, `N_OUTPUT`, `N_HIDDEN`, and `N_LAYERS` specify the number of input features, the number of expected outputs, the number of hidden units in each layer, and the number of layers in the network, respectively. The `activation` variable defines the `Tanh` (hyperbolic tangent) as the activation function of choice, given its popularity in modeling non-linear learning patterns.

The `self.fcs` attribute creates the network's initial layer, transforming the input data to a higher-dimensional hidden space (`N_HIDDEN`). `self.fch` defines a sequence of intermediary fully connected layers (excluding the first and last) coupled with the chosen activation function, with the number of layers being `N_LAYERS-1` as the first layer is already defined by `self.fcs`. Finally, `self.fce` establishes the end layer of the network, mapping the representation from the hidden layers down to the output space dimension (`N_OUTPUT`).

The `forward` method defines the sequence of operations the model performs to generate predictions from input data `x`. Here, the data is passed through the initial layer `fcs`, propagating through the intermediary hidden layers `fch`, and finally through the end layer `fce`. Each stage applies its transformations and activation functions, distilling the raw input into a prediction

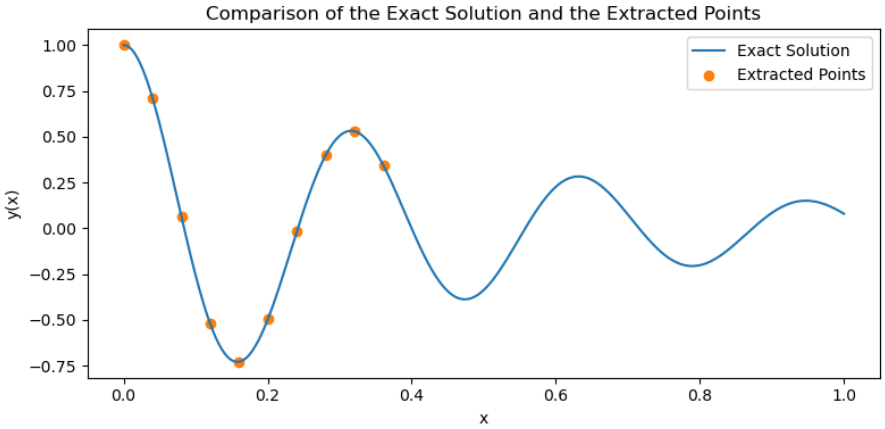

Figure 11: Our models will only train on a small subset of the points (orange) within the exact solution (blue).

output through the network's learned mappings. The method returns `x`, which, at this point, holds the network's predictions corresponding to the provided inputs.

With the modeling architecture established, we can now generate sample data to train our PINNs model.

```
d, w0 = 2, 20

x = torch.linspace(0,1,500).view(-1,1)
y = oscillator(d, w0, x).view(-1,1)

x_data = x[0:200:20]
y_data = y[0:200:20]
```

First, we can set the system's damping ratio and natural frequency, referenced by the variables `d` and `w0` respectively, to 2 and 20.

The `torch.linspace` function generates a sequence of 500 points evenly spaced between 0 and 1, representing time intervals. These points are then reshaped into a two-dimensional tensor (or matrix) `x` with 500 rows and 1 column, serving as the input for the `oscillator` function to solve the system over the specified domain. The `oscillator` function is called with `d`, `w0`, and `x` as arguments, and the resulting positional values `y` are similarly reshaped into a 500x1 tensor.

Next, in effort to demonstrate the predictive accuracy of PINNs, we want to extract a subset of the data points. This simulates the realistic scenario where complete continuous data is not available in live projects. Specifically, we select every 20th point from the first 200 points in the `x` and `y` tensors by using the slicing syntax. This operation significantly reduces the dataset size, resulting in `x_data` and `y_data` tensors with only 10 points each (effectively a 95% reduction from the original set). We can then visualize this data extraction (Figure 11). Note: as a reminder, the full codebase, including the code to construct this figure, can be found in our companion GitHub repository.

And finally, before training our models, we can make a custom function to visualize progress

at every training step.

```
def plot_result(x, y, x_data, y_data, yh, tag, xp=None):
    plt.figure(figsize=(8,4))
    plt.title(f"{tag} Modeling Performance")
    plt.ylabel("y(x)")
    plt.xlabel("x")
    plt.plot(x, y, color="gray", linewidth=2, alpha=0.8, label="Exact Solution")
    plt.plot(x, yh, color="tab:blue", linewidth=4, alpha=0.8, label="Model
    Prediction")
    plt.scatter(x_data, y_data, s=60, color="tab:orange", alpha=0.4, label="
    Training Data")
    if xp is not None:
        plt.scatter(xp, -0*torch.ones_like(xp), s=60, color="tab:green", alpha
    =0.4,
                    label="Physics Loss Training Locations")
    l = plt.legend(loc=(1.01, 0.34), frameon=False, fontsize="large")
    plt.setp(l.get_texts(), color="k")
    plt.xlim(-0.05, 1.05)
    plt.ylim(-1.1, 1.1)
    plt.text(1.065, 0.7, "Training step: %i"%(i+1), fontsize="xx-large", color="
    k")
```

When called, `plot_result` initiates a new figure with the specified dimensions of (8, 4), the plot title (which is annotated by the given tag attribute), and xy-axes labels. It then proceeds to plot the exact harmonic solution, the neural network's predictions, and the extracted training points.

The next portion of the code (containing the if-statement) will only be used during the PINN modeling. The set of points, represented by `xp`, is included to indicate to the model where specific locations in the input domain the PINNs loss function were applied during training. This helps steer the neural network's learning process using the underlying system's known physical laws.

The remaining code deals with the plot's aesthetics, such as legends, colors, transparency (`alpha`), and text annotations. Labels are used to identify the different datasets, and the legend is customized and positioned outside of the plot area to avoid obscuring the data.

3.2.3 Results: Comparing Traditional NN and the PINN

Now, we can finally implement a traditional neural network on your dataset to test performance. We start by setting a specific random seed for PyTorch to ensure consistency in results across different runs (essential for reproducibility).

```
random_seed = 123
```

Next, we can create our architecture, train the model, and plot its performance.

```
tag = "Traditional"
nn_errors = []
files = []

torch.manual_seed(random_seed)
model = FCN(1, 1, 32, 3)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
for i in range(1000):
    optimizer.zero_grad()
    y_nn = model(x_data)
    train_loss = torch.mean((y_nn - y_data)**2)
    train_loss.backward()
    optimizer.step()

    y_nn = model(x).detach()
    actual_loss = torch.mean((y_nn - y)**2)
    nn_errors.append(actual_loss.item())

    if (i+1) % 250 == 0:
        plt.show()
    else:
        plt.close("all")
```

We first define `tag` for visualization labeling, `nn_errors` for collecting our residuals, and `files` for creating our GIF. Next, the neural network model is created with the specified architecture parameters, and an optimization strategy (Adam optimizer) is set at the learning rate `lr` of 1e-3.

In the training phase, we enter a loop (in this case, 1000 training epochs) to incrementally improve the model. During each iteration, the model makes predictions, and the discrepancy between these predictions and the training data is calculated using mean squared error. This `train_loss` is used to adjust the model's internal workings slightly, aiming to reduce prediction error in subsequent iterations. At the same time, we will also measure and collect the `actual_loss` between the model and the exact solution.

Simultaneously, we can visually document the training process. At every 250th training step, we directly output the graph to the console for a real-time view of the progress. Selected snapshots from the training are seen below (Figure 12), while a modified version of this script containing code for storing and generating GIFs of the training process can be found in our companion GitHub repository.

Let us proceed to the PINNs counterpart.

```
x_physics = torch.linspace(0,1,30).view(-1, 1).requires_grad_(True)
mu, k = 2*d, w0**2

tag = "PINNs"
pinn_errors = []
files = []

torch.manual_seed(random_seed)
model = FCN(1, 1, 32, 3)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

for i in range(16500):
    optimizer.zero_grad()
    y_pinn = model(x_data)
    loss1 = torch.mean((y_pinn-y_data)**2)

    yhp = model(x_physics)
```

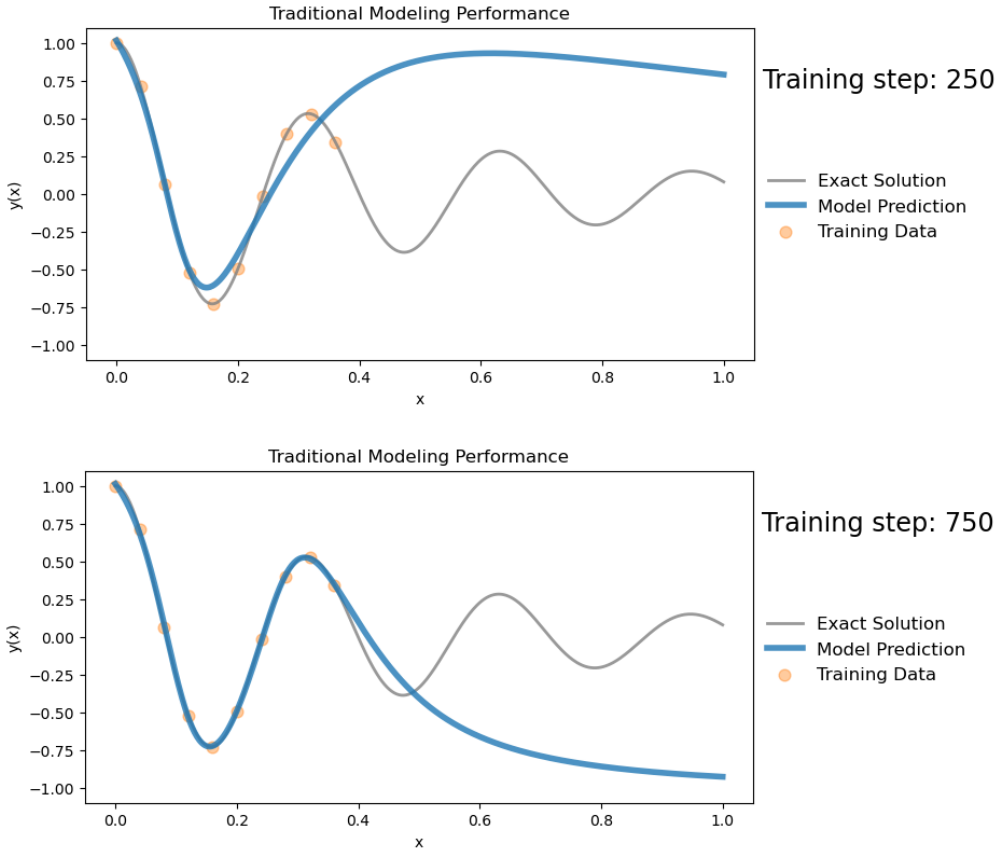

Figure 12: By training step 250, the traditional model shows promise in approximating the exact harmonic oscillation solution, but by training step 750, it converges to low-fitting approximation. Traditional neural networks struggle with modeling non-linear patterns far past their given training data.

```
    dx  = torch.autograd.grad(yhp, x_physics, torch.ones_like(yhp), create_graph
    =True)[0]
    dx2 = torch.autograd.grad(dx,  x_physics, torch.ones_like(dx),  create_graph
    =True)[0]
    physics = dx2 + mu*dx + k*yhp
    loss2 = (1e-4)*torch.mean(physics**2)

    loss = loss1 + loss2
    loss.backward()
    optimizer.step()

    y_pinn = model(x).detach()
    actual_loss = torch.mean((y_pinn - y)**2)
    nn_errors.append(actual_loss.item())

    if (i+1) % 500 == 0:
        plt.show()
    else:
        plt.close("all")
```

We start by setting our key physical parameters (`mu`, `k`) which are derived from the values of (`d`, `w0`) defined previously. A neural network model is then instantiated with the specific structure, and an algorithm (Adam optimizer) is prepared for its training with the specified learning rate `lr` of 1e-4.

In the training phase, we enter a loop (in this case, 16,500 training epochs) to incrementally improve the model with dual objectives: to learn from both the training data and the harmonic oscillation ODE.

First, we calculate our "data loss" `loss1` by comparing the model's predictions to the training data, aiming to minimize this discrepancy. Next, it calculates our "physical loss" `loss2` by computing the model's derivatives and evaluating the residual from the exact solution. Both types of loss are combined, signifying that the model should simultaneously fit the data and respect the physical rules. This is one key way in which PINNs integrate their specified physical law with their traditional machine-learning process. We then also calculate the actual loss and store it in `pinns_error` for future evaluation.

Similarly to the traditional model case, we visually document the training process. For every 500th step, we directly output the graph to the console for a real-time view of the progress. Selected snapshots from the training are seen below (Figure 13).

With the data graphed from both the NN and the PINN models, we can compare the results (Figure 14). We see great performance in the short run with the traditional model, but it eventually converges on a poor approximation in the long run. The PINN model, on the other hand, continually fits itself to the exact solution far after the initial training data points. PINNs thus have implications for not only model robustness but also for data efficiency, as they can get accurate results with much fewer data points.

Seen another way, we can compare the relative error rates of both models against the exact solution (Figure 15). Here, we see that the PINN model is able to continually decrease its error,

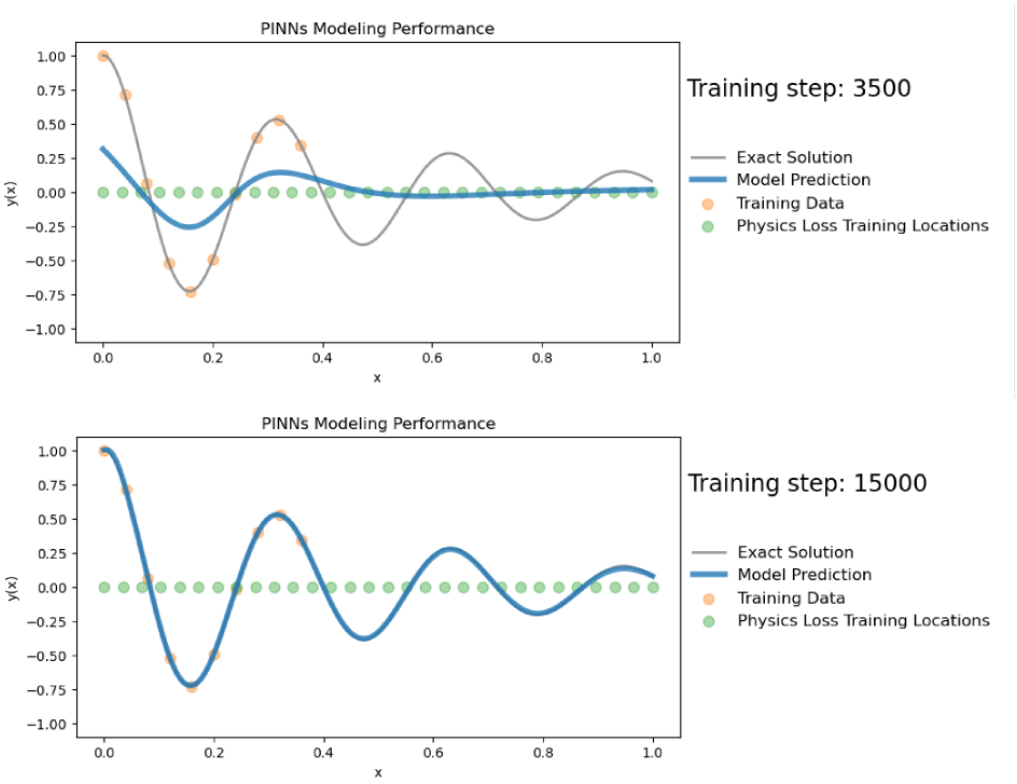

Figure 13: At training step 3500, the PINN model starts with a poorly fitted approximation, but by training step 15000, has perfectly approximated the exact solution far past the given training data.

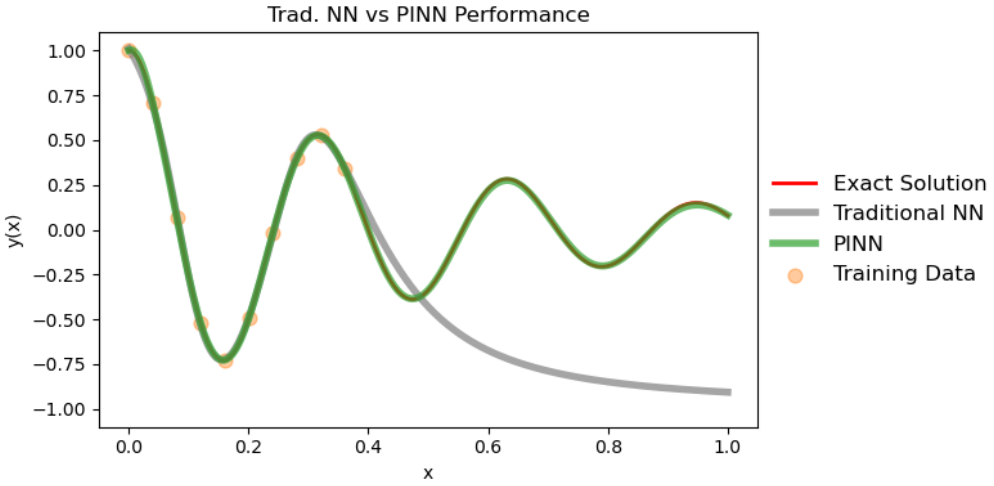

Figure 14: Traditional neural networks struggle to model non-linear patterns far outside their original training data, a problem that PINNs seem to handle more easily.

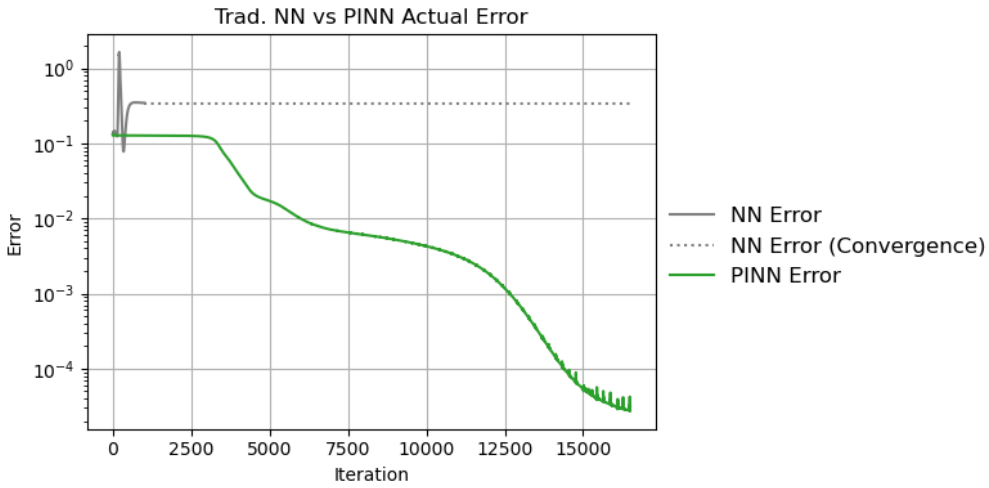

Figure 15: PINNs see continual error improvements while traditional models converge onto a high-error solution early on.

while the traditional neural network plateaus.

The harmonic oscillation example, inspired by Moseley, is both a great way to showcase the power of PINNs compared to traditional NNs and an entry point for educating others about PINNs through a coding framework.

3.3 Residual-Based Adaptive Refinement of the Burgers Equation

3.3.1 Process Overview

Another area in which PINNs prove useful is in approximating partial differential equations (PDEs), specifically in conjunction with residual-based adaptive refinement (RAR). RAR is a standard technique through which we can dynamically refine the neural network's approximation by increasing the density of collocation points in regions that have high error rates. In the section below, we will evaluate the impact of PINN architecture, Burgers' equation and then compare their performance.

3.3.2 Code Setup and Model Definitions

Following the Lu Lu et al. code, we first create our virtual environment. Since the DeepXDE example does not specify a Python version, we will omit this variable in our setup.

```
conda create -n burgers-pinns
conda activate burgers-pinns
```

Next, we install the following library packages. Since the DeepXDE package is still actively being developed, the second line is required to ensure all prerequisites of DeepXDE are installed.

```
conda install jupyter numpy matplotlib pandas
```

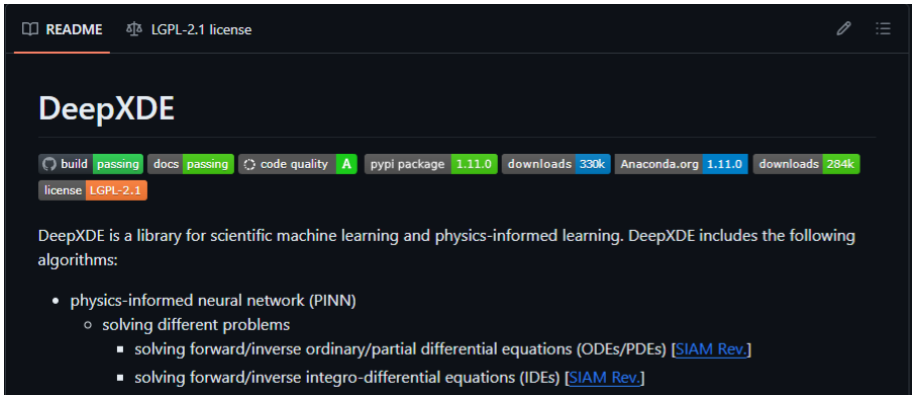

Figure 16: The Burgers dataset can be downloaded from Lu Lu et al's provided repository.

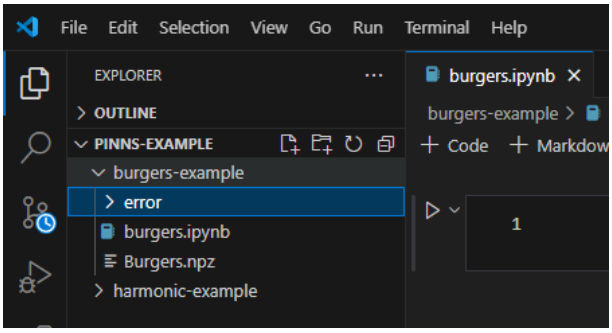

Figure 17: The new `Burgers.npz` dataset should be in the same folder as our other files.

```
conda install paddlepaddle==2.6.0 --channel https://mirrors.tuna.tsinghua.edu.cn
    /anaconda/cloud/Paddle/
conda install -c conda-forge deepxde
```

Following the same steps to create our "harmonic-example" folder and `harmonic.ipynb` file, we can create a new folder called "burgers-example". Within that folder, we can create another empty "error" folder and the `burgers.ipynb` file.

Next, we want to download the data provided by Lu Lu et al's example, which can be found in their GitHub repository [2] (Figure 16). Save this `Burgers.npz` file to the same location as the `burgers.ipynb` file. If done correctly, we should now see all our Burgers files in the same folder (Figure 17).

Once done, open `burgers.ipynb` and add the following imports within the first cell block.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import deepxde as dde
from deepxde.backend import tf
```

Next, we can add custom functions to assist with modeling. First, consider the mathematical equation for the Burgers' equation in one-dimensional space:

[2](GitHub repository: https://github.com/lululxvi/deepxde/blob/master/examples/dataset/Burgers.npz)

$$\frac{\partial y}{\partial t} + y\frac{\partial y}{\partial x} = \nu\frac{\partial^2 y}{\partial x^2}, \quad x \in [-1, 1], t \in [0, 1], \tag{3.5}$$

with initial and boundary conditions

$$y(x, 0) = -\sin(\pi x), y(-1, t) = 0, y(1, t) = 0. \tag{3.6}$$

where

- $y(x, t)$ is the velocity field in a fluid flow.
- ν is the kinematic viscosity coefficient quantifying the fluid's resistance to flow, which is set at $0.01/\pi$

We can then define the 1D Burgers' equation symbolically.

```
def pde(x, y):
    dy_x = dde.grad.jacobian(y, x, i=0, j=0)
    dy_t = dde.grad.jacobian(y, x, i=0, j=1)
    dy_xx = dde.grad.hessian(y, x, i=0, j=0)
    return dy_t + y * dy_x - 0.01 / np.pi * dy_xx
```

In the code above, we can leverage DeepXDE's built-in gradient operators to find the derivatives. Specifically, this function calculates the first spatial derivative (`dy_x`), the temporal derivative (`dy_t`), and the second spatial derivative (`dy_xx`). We then return the discretized form of the Burgers' equation.

Next, we define our modeling constraints.

```
geom = dde.geometry.Interval(-1, 1)
timedomain = dde.geometry.TimeDomain(0, 0.99)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)

bc = dde.icbc.DirichletBC(geomtime, lambda x: 0, lambda _, on_boundary:
    on_boundary)
ic = dde.icbc.IC(geomtime, lambda x: -np.sin(np.pi * x[:, 0:1]), lambda _,
    on_initial: on_initial)
```

The computational domain is defined within a spatial interval and a time domain, combined into a spatio-temporal domain (`geomtime`). Dirichlet boundary conditions (`bc`) ensure the solution remains zero at the spatial boundaries, and an initial condition (`ic`) sets the solution at the beginning of the time domain based on a sine function, aligning with the Burgers' equation's physical context. The function `dde.data.TimePDE` creates a data object for a time-dependent PDE, incorporating the geometry, PDE, boundary conditions, and initial condition defined earlier, effectively codifying this system for modeling.

At this point, we can implement some custom functions to assist with modeling. First, we can define the function to extract the exact solution from our downloaded dataset.

```
def gen_testdata():
    df = np.load("Burgers.npz")
    t, x, exact = df["t"], df["x"], df["usol"].T
```

```
    xx, tt = np.meshgrid(x, t)
    X = np.vstack((np.ravel(xx), np.ravel(tt))).T
    y = exact.flatten()[:, None]
    return X, y
```

A visualization of the exact solution of the Burgers' equation is shown below (Figure 18). The full codebase used to generate these images is contained the companion GitHub repo.

Secondly, we can define an export function to extract our errors from DeepXDE's model history and save it as a CSV file. This will assist with error visualization.

```
def export_error(input_file, output_file):
    steps = []
    loss_train = []
    loss_test = []
    with open(input_file, "r") as file:
        next(file)
        for line in file:
            values = line.split()
            steps.append(float(values[0]))
            loss_train.append(float(values[1]))
            loss_test.append(float(values[2]))
    df = pd.DataFrame({"Step": np.array(steps),
                       "Training Loss": np.array(loss_train),
                       "Test Loss": np.array(loss_test)})
    filepath = f"error/{output_file}.csv"
    df.to_csv(filepath, index=False)
```

This function starts by opening the input file specified by the `input_file` parameter in read mode (`"r"`). The function skips the header line of the input file using `next(file)` to move to the actual data. Then, it iterates through each line of the file, splitting the line into values and appending the first value (`step`), second value (`training loss`), and third value (`test loss`) to their respective lists. After reading all the data, the function creates a pandas DataFrame named `df` using the collected lists with specified three columns: "Step", "Training Loss", and "Test Loss". Finally, the function saves the DataFrame to the indicated file path, using the given `output_file` as the file name.

And lastly, we can define a function to graph these extracted errors.

```
def plot_error(title, df, output_file):
    plt.figure(figsize=(8,4))
    plt.title(f"{title} Error")
    plt.xlabel("Training Step")
    plt.ylabel("Error")
    plt.yscale("log")

    plt.plot(df["Step"], df["Training Loss"], linewidth=2, alpha=0.8, label="
    Training Error")
    plt.plot(df["Step"], df["Test Loss"], color="tab:orange", alpha=0.4, label="
    Test Error", linestyle="--")
    l = plt.legend(loc=(1.01, 0.34), frameon=False, fontsize="large")
    plt.setp(l.get_texts(), color="k")
    plt.show()
```

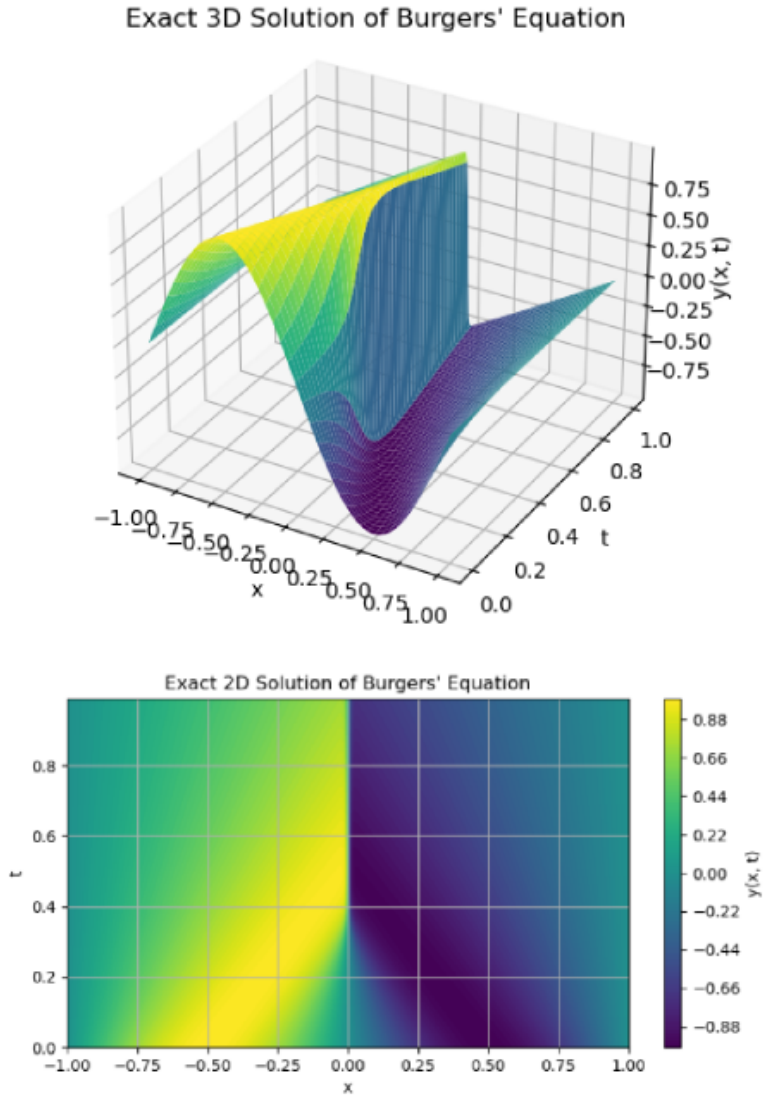

Figure 18: The 3D and 2D contour visualization of the Burgers' equation that the traditional NN and PINN will try to approximate.

This function operates very similarly to the graphing functions described in Section 3.2 on harmonic oscillation.

3.3.3 Results: Approximating the Burgers' Equation with PINNs

Let us first generate random spatial inputs for the Burgers' equation. Because DeepXDE's `geomtime` function leverages numpy's random number generation, we can specify an arbitrary random seed through the numpy's `random.seed` function to ensure reproducibility.

```
np.random.seed(123)
X = geomtime.random_points(100000)
```

Moving forward into training, we can now approximate the Burgers' equation with our first model: a PINN using the RAR method.

```
data = dde.data.TimePDE(geomtime, pde, [bc, ic], num_domain=2500, num_boundary
    =100, num_initial=100)
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "Glorot normal")
pinn_rar = dde.Model(data, net)
```

We proceed with the training.

```
pinn_rar.compile("adam", lr=1e-3)
losshistory, train_state = pinn_rar.train(iterations=10000)
pinn_rar.compile("L-BFGS-B")
losshistory, train_state = pinn_rar.train()

err = 1
while err > 0.005:
    f = pinn_rar.predict(X, operator=pde)
    err_eq = np.absolute(f)
    err = np.mean(err_eq)

    x_id = np.argmax(err_eq)
    data.add_anchors(X[x_id])

    early_stopping = dde.callbacks.EarlyStopping(min_delta=1e-4, patience=2000)

    pinn_rar.compile("adam", lr=1e-3)
    pinn_rar.train(iterations=10000, disregard_previous_best=True, callbacks=[
    early_stopping])
    pinn_rar.compile("L-BFGS")
    losshistory, train_state = pinn_rar.train()
```

We initially train the model with one "round" of the Adam and L-BFGS-B optimizers to approach a preliminary solution. Next, we set `err = 1` as the initial error metric (`err`) to control the iterative refinement loop. (The initial value is set arbitrarily high to ensure the loop's execution.) RAR then occurs within the while-loop, which will continue as long as the mean residual error across the domain exceeds the 0.005 predefined threshold or desired accuracy level. Within this loop, `f = model.predict(X)` uses the model to predict the solution at each point in `X`. The residual `f` represents the discrepancy between the left and right-hand sides of the Burgers' equation at each point, indicating areas where the model's solution does not satisfy the equation.

The line `err_eq = np.absolute(f)` computes the absolute value of each residual, transforming them into a measure of error regardless of sign, while the line `err = np.mean(err_eq)` calculates the mean of these absolute errors, providing a single metric (`err`) that quantifies the model's overall performance across the sampled points. We then take the point with the highest error as a new anchor point in the training data using `data.add_anchors(X[x_id])`. Anchor points are specific locations where the model is encouraged to improve its accuracy, effectively directing the model's learning efforts toward rectifying its most significant deficiencies–RAR codified in action.

The model is then recompiled with the Adam optimizer and a learning rate of 1e-3 and retrained for up to 10000 iterations, incorporating an early stopping callback (`early_stopping`) if necessary. This callback monitors the training process and halts it if the model's performance ceases to improve significantly, preventing overfitting and unnecessary computational expenditure. After the Adam optimization phase, the model is recompiled with the L-BFGS optimizer for fine-tuning—effectively reproducing the initial training regiment defined above.

The loop iterates, each time evaluating the model's residuals, adding the worst-performing point as an anchor, and retraining the model. This process continues until the mean residual error falls below the threshold, indicating that the model's predictions are consistently within the desired accuracy across the domain.

We can now graph the results (Figure 19).

```
dde.saveplot(losshistory, train_state, issave=True, isplot=False)
input_file = "loss.dat"
output_file = "pinns_rar"
export_error(input_file, output_file)

title = "PINN w/ RAR"
df = pd.read_csv(f"error/{output_file}.csv")
output_file = "Burgers-PINN-RAR"
plot_error(title, df, output_file)
```

For comparison's sake, we can also develop a PINN model without the RAR learning method. First, instantiate the model.

```
data = dde.data.TimePDE(geomtime, pde, [bc, ic], num_domain=2540, num_boundary
    =80, num_initial=160)
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh", "Glorot normal")
pinn = dde.Model(data, net)
```

And then train it on the same inputs.

```
pinn.compile("adam", lr=1e-3)
pinn.train(iterations=15000)
pinn.compile("L-BFGS")
losshistory, train_state = pinn.train()
```

By saving and plotting the results with similar code below (Figure 19), we can qualitatively see the differences that the RAR method has with PINN architecture (Figure 20).

```
dde.saveplot(losshistory, train_state, issave=True, isplot=False)
input_file = "loss.dat"
```

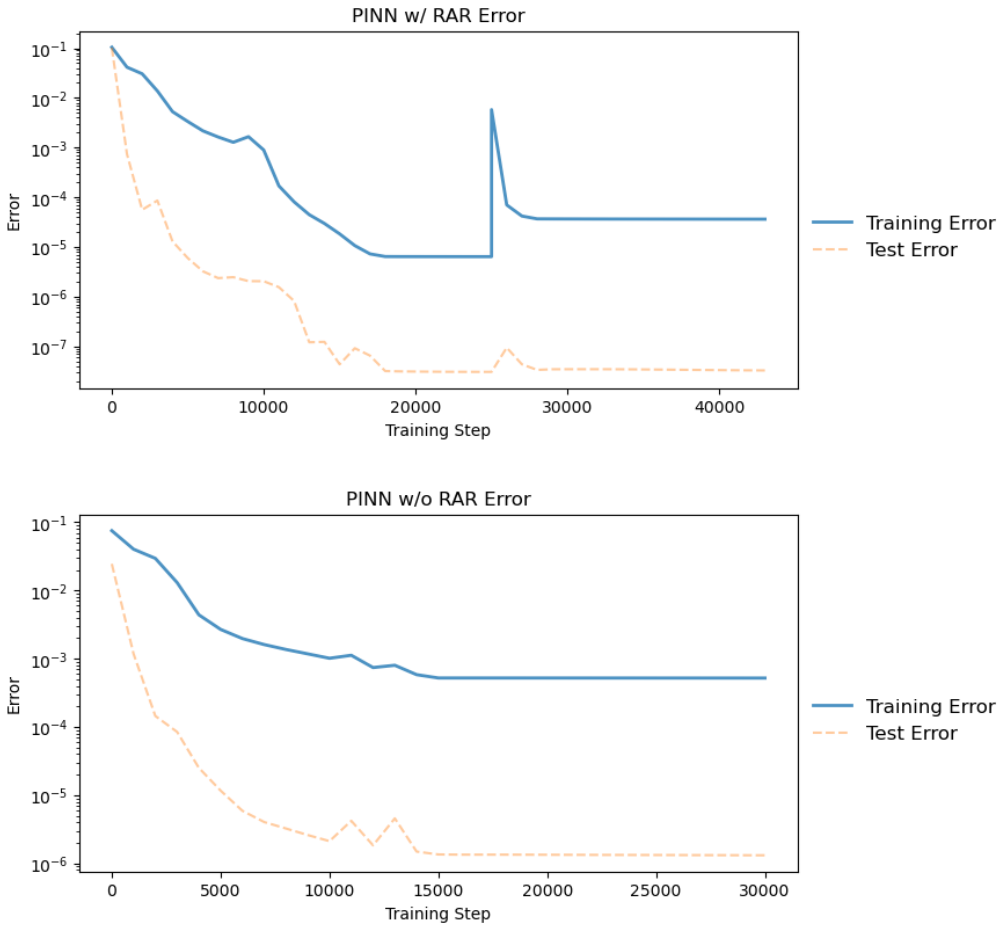

Figure 19: The errors from the PINN model with and without the RAR method.

```
output_file = "pinns"
export_error(input_file, output_file)

title = "PINN w/o RAR"
df = pd.read_csv(f"error/{output_file}.csv")
output_file = "Burgers-PINN"
plot_error(title, df, output_file)
```

This specifically demonstrates that the RAR method greatly augments the already powerful PINN architecture, and together they have great potential in modeling non-linear equations. When comparing the PINNs architecture with and without the RAR learning method against other traditional methods, we also see strong performances.

From the study presented by Lu Lu et al, they found that the PINN models all have a close fidelity to the reference solution of the Burgers equation (in teal, which is notably overlapped by the red line). Other traditional methods such as the finite difference method (FD) can achieve similar results but require many data points (the dotted blue line) lest it suffers in performance

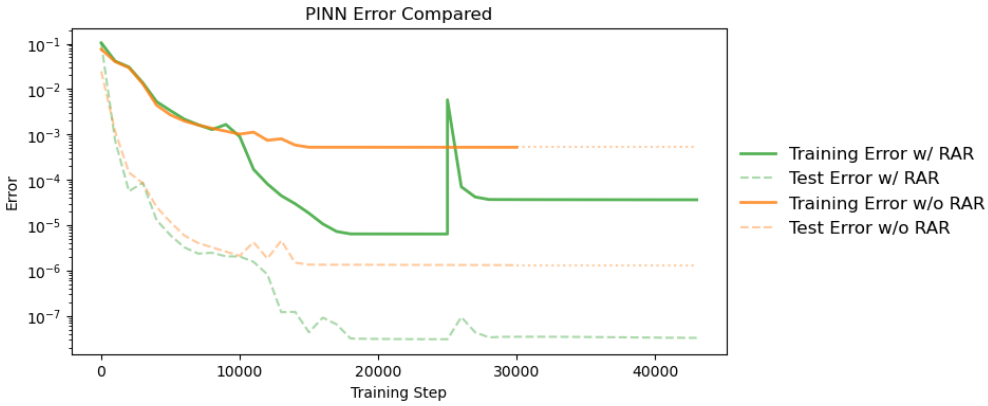

Figure 20: Burgers' equation errors graphed together. PINNs with RAR method perform noticeably better when approximating non-linear relationships.

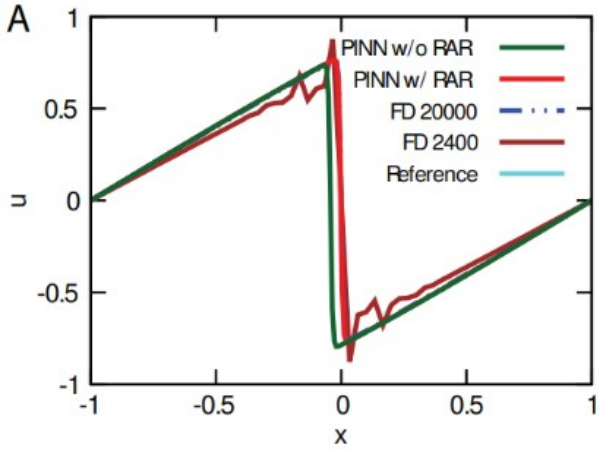

Figure 21: PINNs with RAR closely match the reference solution (red line overlaps the teal line), while more traditional approximation methods like FD requires many more data points to achieve the similar result. This image is taken from Lu Lu et al's paper.

(the brown line). This demonstrates that PINNs are much less affected by these data problems, even in higher dimensional problems.

Thus, as seen, the PINNs architecture is crucial for capturing the steep gradients and shock waves, especially those that are characteristic of the Burgers equation. PINNs enhance the solution's accuracy without uniformly increasing the computational cost across the entire domain.

4 Conclusion

In this study, we have showcased the remarkable capabilities of Physics-Informed Neural Networks (PINNs) in addressing two distinct yet equally challenging problems: model approximation in the harmonic oscillator model and Residual-based Adaptive Refinement (RAR) for the Burgers' equation. Our investigation highlights the versatility and efficiency of PINNs in modeling complex physical phenomena, leveraging the inherent structure of the governing equations to guide the learning process.

For the harmonic oscillator, a fundamental system in classical mechanics, we demonstrated how PINNs can effectively incorporate physical laws to correct for errors arising from noisy or incomplete data—a problem that is well-known from traditional machine learning models. The ability of PINNs to seamlessly blend data-driven insights with established physical principles led to significant improvements in the accuracy and reliability of the oscillator model, showcasing the potential of PINNs in enhancing traditional analytical and numerical approaches.

In the context of the Burgers' equation, a canonical problem in fluid dynamics known for its nonlinear and shock-forming characteristics, the application of RAR through PINNs proved to be a game-changer. By adaptively refining the solution in regions with high residuals, PINNs were able to capture the equation's complex dynamics with remarkable precision. This adaptive refinement approach, enabled by the PINN framework, not only improved solution accuracy but also optimized computational resources, focusing efforts on areas where the model's accuracy was most deficient.

These code implementations demonstrate the transformative potential of PINNs in computational physics and engineering. By intelligently leveraging the underlying physics of the problems, PINNs offer a powerful tool for solving a wide range of PDEs, from linear and time-invariant systems like the harmonic oscillator to highly nonlinear and time-dependent equations exemplified by the Burgers' equation.

Furthermore, the adaptability and data efficiency of PINNs, as demonstrated through the RAR technique, pave the way for tackling problems where traditional numerical methods falter due to computational constraints or lack of fine-grained data. In an era where data-driven approaches are increasingly critical, the integration of physical laws into the learning process presents a promising avenue for advancing scientific computing.

In conclusion, our exploration of physics-informed neural networks (PINNs) shows both their effectiveness in modeling scientific problems with limited data, as well as their potential to

transform computational strategies across scientific domains. This note aims to build intuition for PINNs in the context of differential equations and mathematical modeling, with the goal of equipping the next generation of scientists and engineers with leading-edge techniques at the intersection of artificial intelligence and domain expertise. Our guided examples and hands-on methodology can aid educators in teaching PINNs for tackling real-world systems, ultimately furthering the synergistic development of scientific machine learning.

References

Grohs, P., Hornung, F., Jentzen, A., & von Wurstemberger, P. (2023, 4). A proof that artificial neural networks overcome the curse of dimensionality in the numerical approximation of black–scholes partial differential equations. *Memoirs of the American Mathematical Society*, *284*. doi: 10.1090/memo/1410

Higham, C. F., & Higham, D. J. (2019, 1). Deep learning: An introduction for applied mathematicians. *SIAM Review*, *61*, 860-891. doi: 10.1137/18M1165748

Lagaris, I., Likas, A., & Fotiadis, D. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, *9*, 987-1000. doi: 10.1109/72.712178

Lozada, E., Guerrero-Ortiz, C., Coronel, A., & Medina, R. (2021, 3). Classroom methodologies for teaching and learning ordinary differential equations: A systemic literature review and bibliometric analysis. *Mathematics*, *9*, 745. doi: 10.3390/math9070745

Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021, 1). Deepxde: A deep learning library for solving differential equations. *SIAM Review*, *63*, 208-228. doi: 10.1137/19M1274067

Moseley, B. (2021, 8). *So, what is a physics-informed neural network?*

Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B., & Liao, Q. (2017, 10). Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing*, *14*, 503-519. doi: 10.1007/s11633-017-1054-2

Raissi, M., Perdikaris, P., & Karniadakis, G. (2019, 2). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686-707. doi: 10.1016/j.jcp.2018.10.045

Rossum, G. V., & Jr, F. D. (1995). Python reference manual. *Centrum voor Wiskunde en Informatica Amsterdam*.

Spooner, K. (2024, 2). Using mathematical modelling to provide students with a contextual learning experience of differential equations. *International Journal of Mathematical Education in Science and Technology*, *55*, 565-573. doi: 10.1080/0020739X.2023.2244472