Practical Aspects on Solving Differential Equations Using Deep Learning: A Primer

Georgios Is. Detorakis[1]
[1]Independent Researcher, Irvine, CA, USA
[1]gdetor@protonmail.com

Abstract

Deep learning has become a popular tool across many scientific fields, including the study of differential equations, particularly partial differential equations. This work introduces the basic principles of deep learning and the Deep Galerkin method, which uses deep neural networks to solve differential equations. This primer aims to provide technical and practical insights into the Deep Galerkin method and its implementation. We demonstrate how to solve the one-dimensional heat equation step-by-step. We also show how to apply the Deep Galerkin method to solve systems of ordinary differential equations and integral equations, such as the Fredholm of the second kind. Additionally, we provide code snippets within the text and the complete source code on Github. The examples are designed so that one can run them on a simple computer without needing a GPU.

Contents

1 Introduction

Deep learning (DL) [44] has advanced significantly in recent years, providing algorithms that can solve complex problems varying from image classification [12] to playing games such as Go [58] at a human level or even better. More recent advances include the development of Large Language Models (LLMs), which are substantial deep learning models (*i.e.*, they usually have billions of parameters) such as ChatGPT [62, 21], and Llama [65] that have been trained on enormous data sets. Deep learning has naturally affected many scientific fields and has been applied to many complex problems. One such problem is the numerical solution of differential equations, where (deep) neural networks approximate the solution of partial or ordinary differential equations. The idea is not new since there are works from the 90s like Lee et al. [45] and Wang et al. [66] have relied on Hopfield neural networks [33] to solve differential equations using linear systems after discretizing them. Meade and Fernandez have proposed solvers for nonlinear differential equations based on combining neural networks and splines [47]. Lagaris et al. introduced in [42] an efficient method that relies on trial solutions and neural networks to solve boundary and initial values problems. They form a loss function using the differential operator of the problem, and a neural network approximates the solution by minimizing that loss function. These methods rely on the idea that feed-forward neural networks operate as universal approximators [17, 34], meaning that a neural network with one hidden layer wide enough can arbitrarily approximate any continuous function defined on a compact subset of $\mathbb{R}^n$.

Another more recent method is the deep Galerkin (DG), introduced by Sirignano and Spiliopoulos [59], based on the Galerkin methods. Galerkin methods are typically used to solve differential equations by dividing the domain of the problem into finite subdomains and approximating the solution within each subdomain using piecewise trial functions (basis functions) [43]. Sirignano and Spiliopoulous relied on Galerkin's methods, and they proposed to minimize loss functions that include the domain, the boundary, and the initial conditions of a boundary value problem by sampling from distributions defined on the domain, boundary, and initial conditions, and using deep neural networks to approximate the solution. The reader can find more details on the DG method applications in Raissi et al. [54], Cuomo et al. [16], Karniadakis et al. [38], and in [6, 10, 2].

In this work, we focus on the DG method. We introduce basic concepts on neural networks and deep learning and briefly introduce the DG method, skipping the theory behind it and concentrating on its practical aspects. We explain how to apply the DG method to solve differential equations via concrete examples. More precisely, we show how to build a neural network, construct the loss function, and implement the example in Python using Pytorch [51]. The examples we present here are (i) the one-dimensional heat equation, (ii) a simple ordinary differential equation (ODE), (ii) a system of ODEs, and (iii) a Fredholm integral equation of the second kind.

Before continuing with the rest of this work, let us summarize a few notable methods that share characteristics similar to those of the DG method. The first method is the deep splitting approximation for semi-linear PDEs [4], which solves semi-linear PDE problems using deep neural networks. The main idea behind this method is to split up the time interval of a semi-linear PDE into sub-intervals where one obtains a linear PDE and then apply a deep learning approximation method on that linear PDE (see Section 2 of [6]. In [5], the authors introduced a numerical method based on deep learning for solving the Kolmogorov PDE. Furthermore, the authors of [57] have proposed an unbiased deep solver for linear parametric (Kolmogorov) PDEs. Han et al. [28, 27] developed deep learning methods for solving problems involving backward stochastic differential equations (BSDE). Another method that relies on deep learning is the one proposed by [39] for scattering problems associated with linear PDEs of the Helmholtz type. The list of methods goes on since research in the field of solving differential equations using deep neural networks is still active, and of course, there are more methods available. However, it is out of the scope of the present work to list all the existing methods. Hence, we defer the reader to [6] for an exhaustive list of methods and to [10] for a thorough introduction to the basic methods mentioned earlier.

Despite the initial enthusiasm about DG and all the similar methods (see for instance [10]), there are still a few weak points that render those methods complementary to the traditional PDE solvers, such as finite element methods (see [43]), and not superior. For instance, in the field of computational fluid dynamics (CFD), recent studies [14, 15] of methods based on deep neural networks (physics-informed neural networks or PINNs) show that when the PINN solver does not have specific training data available, then the approximated solutions are not the expected ones (*i.e.*, the ones provided by a traditional CFD solver) [15].

We organize the rest of this work as follows: First, we introduce fundamental machine learning concepts and then describe the Deep Galerkin (DG) method. We show how to apply the DG method to the one-dimensional heat equation and explore some of its parameters. We also show how to apply the DG method to ordinary differential and integral equations.

2 Notation & Terminology

To begin with we will give some basic notations used in this work. $\mathbf{x} \in \mathbb{R}^q$, $q \in \mathbb{N}$, and $t \in \mathbb{R}_+$ are independent variables representing spatial and temporal variables, respectively. The dependent variable is y for ordinary and partial differential equations reflecting the unknown function(s). The domain of the integral and partial differential equations (PDEs) is $\Omega \in \mathbb{R}^q$, where $q \in \mathbb{N}$, and the boundary of the PDEs is $\partial\Omega$. We use the notation $\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}, t)$ when we refer to a neural network with parameters $\boldsymbol{\theta}$ (weights $\mathbf{W}$ and biases $\mathbf{b}$), and inputs $\mathbf{x}$ and t. The output of a neural network is $\hat{\mathbf{y}} \in \mathbb{R}^p$, where $p \in \mathbb{N}$. $\mathcal{L}(\boldsymbol{\theta}; \ldots)$ represents the loss function. Because we are about to use Pytorch throughout this work we provide some basic definitions and terminology regarding Pytorch:

- **Tensors** are data structures that represent and hold the data or provide methods to modify them. They are multidimensional arrays (not to be confused with tensors used in physics or mathematics) with a shape defined by a Python tuple. The first dimension is the size of the batch, and the second, third, and so on represent the actual input shape. For instance, if we have a mini-batch of ten (10) two-dimensional matrices of dimension 3×3, we can store them in a tensor of shape $(10, 3, 3)$. Figure 1 shows a schematic representation of tensors.

- **Batch** is a small subset of the training (or test) data set. We do offline batch training when we use batches to train our neural networks. To generate a batch, we first define its size n and then sample the train (or test) data set n times. We will interchange the words mini-batch or batch for the rest of this work.

- **Criterion** is a function that evaluates the loss function based on a target input $\mathbf{y}$ and the output of the neural network, $\hat{\mathbf{y}}$. It can generally be the mean squared error (MSE), the mean absolute error (MSE), the hinge loss, and many others. The reader can find an exhaustive list of functions on the documentation page of Pytorch[1]. However, when using the deep Galerkin method, one has to define custom loss functions based on the boundary or initial value problem they want to solve.

- **Optimizer** is an algorithm that minimizes a loss function or model's error by searching for a set of parameters $\boldsymbol{\theta}$ that will increase the neural network's performance. We can choose an optimizer to be one of the following: Stochastic Gradient Descent (SGD), Adam, Adagrad [18], RMSProp, or LBFGS [46]. Again, the reader will find a complete list of all Pytorch available optimizers on its documentation page[2].

- **Layers** In this work, we will implement fully connected (FC) layers using the `Linear` class of Pytorch. Moreover, whenever we refer to input layers, we imply a placeholder for the input data. On the other hand, the actual layers of a feed-forward neural network are the hidden, FC^{h}, and the output, FC^{out}ones. Furthermore, we will refer to a Deep Galerkin layer as DGML.

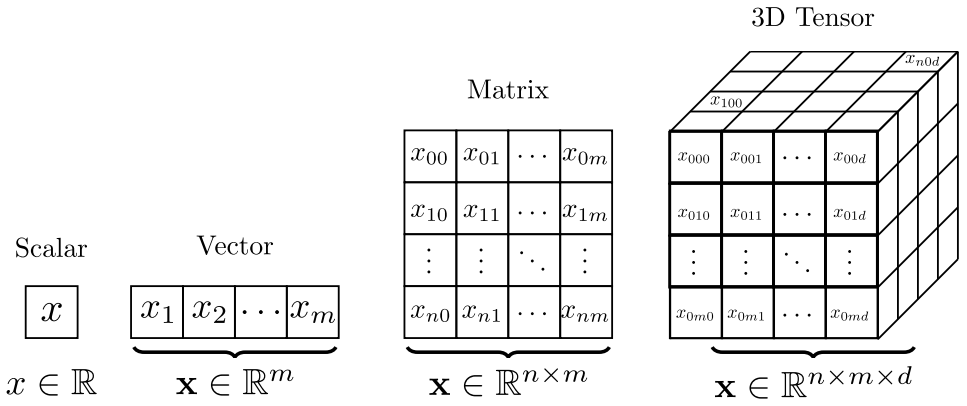

Figure 1: **Visual representation of Pytorch tensors**. From left to right we see a scalar, $x \in \mathbb{R}$, a one-dimensional tensor (or vector) of dimension d, $\mathbf{x} \in \mathbb{R}^d$, a two-dimensional tensor (or matrix) $\mathbf{X} \in \mathbb{R}^{n \times m}$, and finally a three-dimensional tensor $\mathbf{X} \in \mathbb{R}^{n \times m \times d}$. The index starts from zero, following Python's convention.

[1] `https://pytorch.org/docs/stable/nn.html#loss-functions`
[2] `https://pytorch.org/docs/stable/optim.html`

3 Deep Learning

This section briefly introduces deep learning concepts that are useful for understanding and using the Deep Galerkin method. We explain the basics of neural networks, universal approximation theorem, automatic differentiation, and backpropagation. We skip many aspects of deep learning since they are not used in this work. However, if the reader wants to learn more about deep learning, they are referred to [8, 25, 11, 22]. In addition, since we are primarily interested in the practical aspects of the DG method, we provide code snippets to explain more in-depth what happens under the hood. The complete source code with implementations of all neural networks and examples from this work is available on GitHub (see Conclusions).

3.1 Feed-forward neural network

Neural networks, or artificial neural networks, are at the heart of deep learning [25]. In the literature, we find a wide variety of neural networks, like feed-forward networks, convolutional neural networks [25], recurrent neural networks [56], long short-term memory (LSTM) networks [31], and ResNets [30]. Of course, there are many more, but they are out of the scope of this work.

Let's delve into the practical application of feed-forward neural networks, also known as multi-layer perceptron (MLP). These networks are not just theoretical concepts but powerful tools that map some input $\mathbf{x} \in \mathbb{R}^m$ to some output $\mathbf{y} \in \mathbb{R}^p$. In simpler terms, a neural network defines a mapping $\mathbf{y} = \mathcal{F}(\boldsymbol{\theta}; \mathbf{x})$ between an input and an output, making it a key component in the field of machine learning and artificial intelligence.

A neural network is defined by its parameters $\boldsymbol{\theta}$, namely its weights and biases. It consists of an input layer where we provide the input $\mathbf{x}$ and an output layer where we get the neural network's response, $\mathbf{y}$. A feed-forward neural network has one or more hidden layers between the input and output layers, and the number of hidden layers determines whether the network is deep. We call a neural network deep when it has more than two hidden layers [25, 11]. Moreover, in the practitioner's parlor, each layer in a feed-forward neural network is called a fully connected layer (FC) or linear layer, and the latter is the term Pytorch uses.

Computational units or neurons lie within each layer, performing affine transformations. We can write that in a vector form as $\mathbf{y}_l = f_l(\mathbf{z}) = g_l(\mathbf{W}_l \cdot \mathbf{z} + \mathbf{b}_l)$, where $\mathbf{W}_l$ contains the weights that connect neurons from layer $l-1$ with neurons in layer l and $\mathbf{b}_l$ are the biases of the neurons at layer l, and number of layers l runs $l = 1, \ldots, k$. The function $g(x)$ is called the activation function and can be (i) a Sigmoid $g(x) = \frac{1}{1+\exp(-x)}$, (ii) a hyperbolic tangent $g(x) = \tanh(x)$, a (iii) rectified linear unit (ReLU) $g(x) = \max\{x, 0\}$, any custom function, or any of the functions that the reader can find in Pytorch documentation page[3].

Figure 2 **A** shows a typical graph of a feed-forward neural network, where we see the input (green) and output (red) layers as well as the hidden (gray boxes) layers. We can also see the connectivity matrix from the input to the first hidden layer. We omit the rest of the connections due to simplicity. The feed-forward neural network in Figure 2 **A** has three hidden layers, one input, and one output layer. Hence, we can write the operations it performs as $\mathbf{y} = f(\mathbf{x}) = f^{(4)} \circ f^{(3)} \circ f^{(2)} \circ f^{(1)}(\mathbf{x}) = f^{(4)}(f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))))$, where f_1 is the mapping from the input to the first hidden layer, f_2 is the mapping from the first hidden to the second and so on so forth. If we are to expand the composition by using affine transformations, we have $\mathbf{y} = f(\mathbf{x}) = \mathbf{W}_4 \cdot (g_3(\mathbf{W}_3 \cdot g_2(\mathbf{W}_2 \cdot g_1(\mathbf{W}_1 \dot{\mathbf{x}} + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{b}_3) + \mathbf{b}_4$, where g_l represents the activation functions in each layer l. Notice that the output layer L has no activation function.

Like all the modern deep learning libraries, Pytorch offers all the necessary tools to build feed-forward neural networks and define affine transformations easily. The reader can find a complete list on the PyTorch documentation page[4]. Moreover, code listing 1 shows a basic implementation of a Pytorch Multi-Layer Perceptron neural network (a feed-forward neural network) with linear layers and tanh activation functions. We must define the layers and the activation functions used in the constructor `__init__` method. The same method receives as arguments the input and output dimensions, which are the number of independent and dependent variables in our case, respectively. The argument `input_dim` defines the number of input units (independent variables such as t and $\mathbf{x}$), and on the other hand, the argument `output_dim` determines the number of units in the output layer (*e.g.*, y). Moreover, it receives the number of hidden layers (`num_layers`) and the number of units in each hidden layer (`hidden_size`). In this implementation, we assume that all hidden layers will have the same number of units. In line 11 of code listing 1, we define the input layer; in lines 14–15, the hidden layers; and in line 18, the output layer. The class `Linear` of Pytorch is an implementation of an affine transformation ($\mathbf{y} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$). Notice that in line 14, we use a `ModuleList` to instantiate the hidden layers. We do so because the number of hidden layers is not known *a priori*; instead, it is passed as an argument to the constructor when we instantiate the `MLP` class. The `forward` method takes the input tensor as an argument (it might accept more arguments, as we will see later) and applies all the transformations and activation

[3] `https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity`

[4] `https://pytorch.org/docs/stable/nn.html#linear-layers`

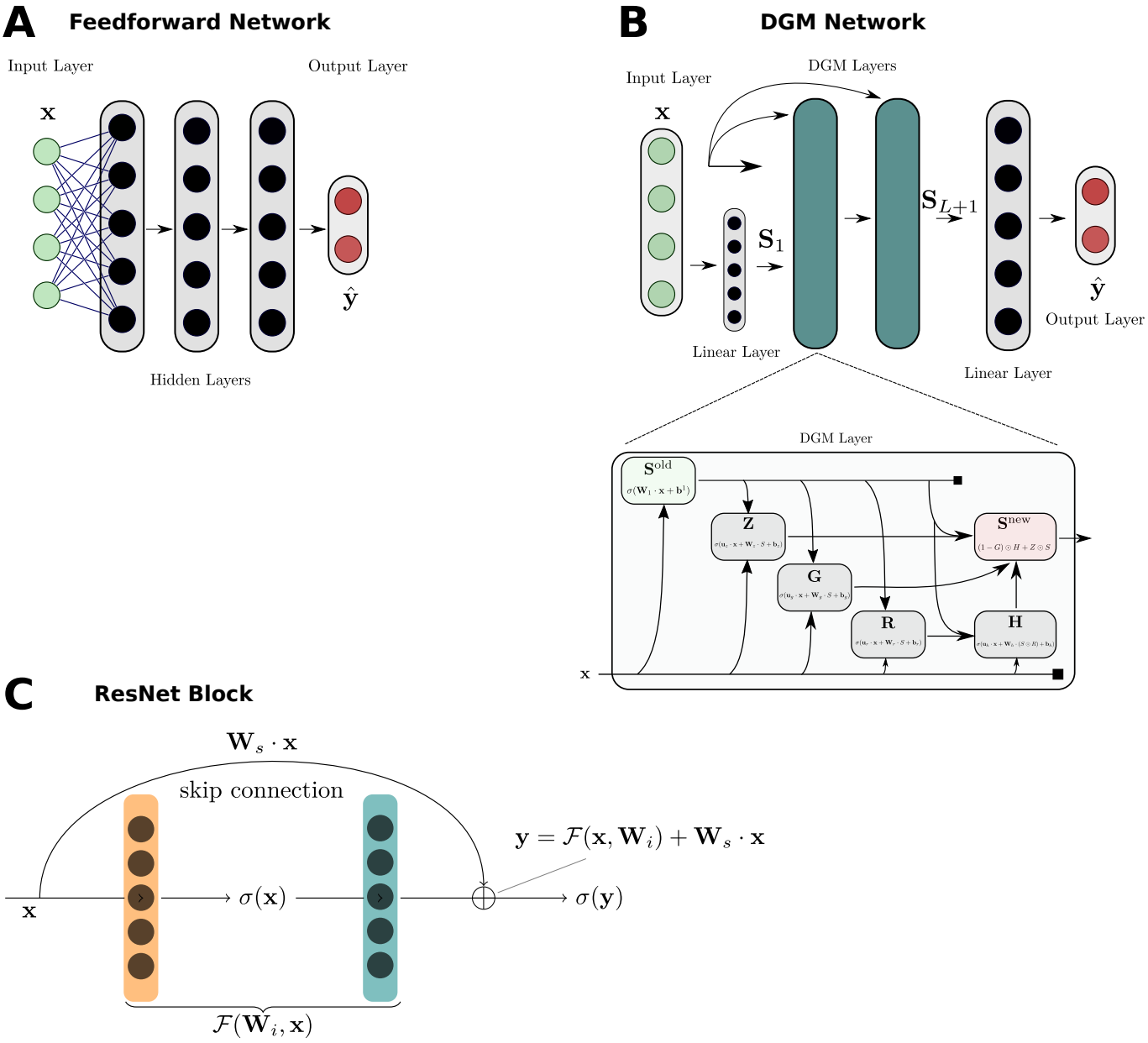

Figure 2: **Neural Network Architectures**. **A** A feed-forward neural network with three hidden layers (gray color), one input (green color), and one output (red color) layer. The connections from the input to the first hidden layer are visible in the graph. **B** A DG-like neural network with two DG layers (teal color) and two fully connected (linear) layers (gray color). The inset shows the flow of information and the transformations within a DG layer. **C** A residual block of a ResNet. The input is distributed to the residual mapping (orange and teal colors) and to the output of the block via a skip connection.

functions we have already defined. Essentially, this method runs the forward pass of our neural network, which is the part of the code where we determine the structure of our neural network.

```
class MLP(nn.Module):
    def __init__(self,
                 input_dim=2,
                 output_dim=1,
                 hidden_size=50,
                 num_layers:=1,
                 batch_norm=False):
        super(MLP, self).__init__()

        # Input layer
        self.fc_in = nn.Linear(input_dim, hidden_size)

        # Hidden layers
        self.layers = nn.ModuleList([nn.Linear(hidden_size, hidden_size)
                                     for _ in range(num_layers)])

        # Output layer
        self.fc_out = nn.Linear(hidden_size, output_dim)

        # Non-linear activation function
        self.act = nn.Tanh()

        # Batch normalization
        if batch_norm:
            self.bn = nn.BatchNorm1d(hidden_size)
        else:
            self.bn = nn.Identity()

    def forward(self, x):
        out = self.act(self.bn(self.fc_in(x)))

        for i, layer in enumerate(self.layers):
            out = self.act(self.bn(layer(out)))
        out = self.fc_out(out)
        return out
```

Listing 1: **Pytorch Implementation of a Multi-Layer Perceptron class.**

3.2 DGM neural network

A second type of neural network usually used in solving differential equations is the neural network proposed by [59]. We will call this type of neural network a DGM-like neural network. The main idea behind those neural networks relies on the Long Short-Term Memory (LSTM) neural networks introduced by Hochreiter and Schmidhuber in [31] and are similar to the Highway Networks introduced in [61]. LSTM is a special kind of recurrent neural network equipped with input, output, and forget gates, as well as with a memory cell. LSTMs learn to balance the memory and forget to learn long sequences, avoiding the gradient vanishing problem (we will explain the vanishing gradient problem after introducing the gradient descent and the backpropagation algorithms).

To build a DGM-like neural network, we must first construct a DG layer. A DG layer receives an input $\mathbf{x}$ and the output of the previous DG layer, and then it propagates the information through four affine transformations. The following equations describe the operations that take place within a DG layer.

$$\begin{aligned} S_1 &= \sigma(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}^1), \\ Z_l &= \sigma(\mathbf{u}_{z,l} \cdot \mathbf{x} + \mathbf{W}_{z,l} \cdot S_l + \mathbf{b}_{z,l}), \\ G_l &= \sigma(\mathbf{u}_{g,l} \cdot \mathbf{x} + \mathbf{W}_{g,l} \cdot S_l + \mathbf{b}_{g,l}), \\ R_l &= \sigma(\mathbf{u}_{r,l} \cdot \mathbf{x} + \mathbf{W}_{r,l} \cdot S_l + \mathbf{b}_{r,l}), \\ H_l &= \sigma(\mathbf{u}_{h,l} \cdot \mathbf{x} + \mathbf{W}_{h,l} \cdot (S_l \odot R_l) + \mathbf{b}_{h,l}), \\ S_{l+1} &= (1 - G_l) \odot H_l + Z_l \odot S_l, \\ f(t; \mathbf{x}; \boldsymbol{\theta}) &= \mathbf{W} \cdot S_{L+1} + \mathbf{b}, \end{aligned} \tag{1}$$

where $l = 1, \ldots, L$ is the number of layers, σ is an activation function, and $\odot$ is the element-wise multiplication (Hadamard product). Figure 2 **B** shows a DG-like neural network with two DG layers (teal boxes) and two fully

connected (linear) layers (gray boxes), one input and one output layer. The inset illustrates the operations and the flow of information within a DG layer.

```
class DGMLayer(nn.Module):
    def __init__(self, input_dim=1, hidden_size=50):
        super().__init__()

        self.Z_wg = nn.Linear(hidden_size, hidden_size)
        self.Z_ug = nn.Linear(input_dim, hidden_size, bias=False)

        self.G_wz = nn.Linear(hidden_size, hidden_size)
        self.G_uz = nn.Linear(input_dim, hidden_size, bias=False)

        self.R_wr = nn.Linear(hidden_size, hidden_size)
        self.R_ur = nn.Linear(input_dim, hidden_size, bias=False)

        self.H_wh = nn.Linear(hidden_size, hidden_size)
        self.H_uh = nn.Linear(input_dim, hidden_size, bias=False)

        # Non-linear activation function
        self.sigma = nn.Tanh()

    def forward(self, x, s):
        Z = self.sigma(self.Z_wg(s) + self.Z_ug(x))
        G = self.sigma(self.G_wz(s) + self.G_uz(x))
        R = self.sigma(self.R_wr(s) + self.R_ur(x))
        H = self.sigma(self.H_wh(s * R) + self.H_uh(x))
        out = (1 - G) * H + Z * s
        return out
```

Listing 2: **Pytorch Implementation of a DG Layer.**

In code listing 2, we see the Pytorch implementation of a neural network layer proposed by [59]. The notation of the layers follows the Equation (1); thus, the implementation is a straightforward mapping of the Equation (1) to Pytorch code. Notice, that the `forward` method receives two arguments, the input `x` and the state of the previous DG layer `s`.

3.3 Feed-forward ResNet

He et al. introduced the Residual Neural Network (ResNet) with convolutional layers for image recognition [30] in 2016. The main idea behind a ResNet is to use residual learning. To do that, first, we must build our blocks, consisting of layers and activation functions. A minimal block performs $\mathbf{y} = \mathcal{F}(\mathbf{x}, W_i) + \mathbf{W}_s \cdot \mathbf{x}$, where $\mathbf{x}$ and $\mathbf{y}$ are the input and output of the block, respectively. The $\mathcal{F}(\mathbf{x}, W_i)$ is a trainable mapping called residual, and it represents multiple convolutional layers in the original version of ResNet. The crucial component of the block, though, is the skip connection from the input of the block to the output, which propagates the input to the output of the block, skipping the residual mapping. When the dimension of the input $\mathbf{x}$ is the same as the residual mapping, then $\mathbf{W}_s$ is the identity; otherwise, it has to be learned such that the dimension of $\mathbf{W}_s \cdot \mathbf{x}$ matches the one of residual mapping. Figure 2 **C** shows a building block of a feed-forward ResNet, where we replace the convolutional with linear layers (fully connected) for our purposes.

An example of implementing a linear ResNet block in Pytorch is given in code listing 3. The method `__init__` takes the number of units in the input and output layers as arguments, respectively. The argument `downsample` defines a transformation (*i.e.*, $\mathbf{W}_s$) at the skip connections that we will apply when the input's $\mathbf{x}$ dimensions mismatches the size of the output of the residual block (see Figure 2 **C**). Finally, the argument `num_features` indicates if the input tensor will contain an extra dimension besides the batch size and the independent variables. Notice that the linear layers here are defined using the Pytorch `Sequential` container in lines 10–14. Within each container, we instantiate a Pytorch linear layer followed by a batch normalization layer (see Section 3.7). In the forward method in lines 23–33, we apply two linear layers (see lines 20 and 21 in code listing 3) and linear rectifications on their outputs. Then we check if `downsample` is not `None` (see lines 23 and 24), and if it's not, then we apply the downsample transformation to the input $\mathbf{x}$, otherwise our residual remains the original input $\mathbf{x}$. Once we have the residual values, we add them to the output, apply linear rectification, and return the output.

```python
class ResidualBlock(nn.Module):
    def __init__(self,
                 input_dim=2,
                 output_dim=1,
                 downsample=None,
                 num_features=None):
        super().__init__()
        self.downsample = downsample

        if num_features is None:
            self.num_features = output_dim
        else:
            self.num_features = num_features

        self.fc1 = nn.Sequential(nn.Linear(input_dim, output_dim),
                                 nn.BatchNorm1d(self.num_features))

        self.fc2 = nn.Sequential(nn.Linear(output_dim, output_dim),
                                 nn.BatchNorm1d(self.num_features))

        self.relu = nn.ReLU()

    def forward(self, x):
        residual = x
        out = self.relu(self.fc1(x))
        out = self.relu(self.fc2(out))

        if self.downsample is not None:
            residual = self.downsample(x)
        out = out + residual
        out = self.relu(out)

        return out
```

Listing 3: **Pytorch Implementation of a Linear ResNet Block.**

3.4 Neural Network Initialization

So far, we have discussed the neural networks we will use in this work and their Pytorch implementations. Next, we explore how to initialize their parameters $\boldsymbol{\theta}$ (weights and biases). The simplest way to initialize a neural network's parameters is with random values drawn from a distribution, such as uniform or normal. The parameter's initialization is crucial to the optimization process since the optimizer requires a starting point in the parameters space. Therefore, the choice of the initial values of the network's parameters affects the optimization and, hence, the ability of the neural network to solve the problem at hand.

A major issue we might face when we initialize the parameters is that we might use huge or tiny initial values, leading to a situation like the exploding or vanishing gradients, respectively (see Section 3.6). Ideally, we would like to initialize our parameters such that the mean value of the activations is zero and the variance stays the same across every layer. To do so, we can use some existing methods in the literature, like the Xavier initialization [23]. In this case, the weights and the biases receive values based on

$$\begin{aligned} \mathbf{W}_l &= \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{k_{l-1}}), \\ \mathbf{b}_l &= \mathbf{0}, \end{aligned} \tag{2}$$

where $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weights and the biases of layer l, $\mathcal{N}$ is a normal distribution centered at zero, and k is the number of neurons of layer l. Xavier initialization will thus set the initial values of the weights close to one so it will overcome the vanishing or exploding gradient problem. Moreover, it will accelerate the optimization and minimize potential oscillations around the minimima. The Xavier initialization works well with activation functions such as tanh and sigmoid. However, authors in [29] argue that Xavier initialization does not work well when we use ReLU or LeakyReLU as activation functions. They propose a slightly different initialization method, called He or Kaiming initialization, where they replace the variance in Equation (2) with $\frac{2}{k_{l-1}}$ [41, 25].

Finally, Pytorch (like most of the modern deep learning frameworks) supports Xavier initialization with normal and uniform distributions, and nonetheless, it provides a method to calculate a gain depending on the activation function

of each layer in the neural network that will multiply the variance and improve the initialization[5].

3.5 Backpropagation, Optimizers and Automatic Differentiation

For a neural network to perform a task or solve a problem, it has to be trained on that particular task. Therefore, we have to optimize the parameters $\boldsymbol{\theta}$ (weights and biases) of a neural network with respect to a given problem (*e.g.*, solving a differential equation). The training process involves two algorithms: the backpropagation [56] or backprop and an optimization algorithm (or optimizer). The backprop estimates gradients and propagates the error from the output layer to the input. On the other hand, an optimizer will use those gradients to solve a minimization problem and find the nearly optimal network parameters.

The backprop algorithm consists of three steps [7]:

1. The forward pass (propagation), where the input $\mathbf{x}$ to the neural network activates the units at the input layer, and that activation propagates forward to the output layer via the hidden layers.

2. The backward pass, where the backprop uses the neural network layers' activations, starting from the output layer and moving backward to the input, evaluates the gradients (derivatives) of the loss function with respect to those activations. The backprop evaluates the gradients at this step using the calculus chain rule.

3. The final step involves evaluating derivatives with respect to network parameters, $\boldsymbol{\theta}$, by forming products of activations from the forward pass and gradients from the backward pass.

Once the backprop has finished evaluating the derivatives and has propagated the error backward, an optimizer will be responsible for minimizing the loss with respect to the network parameters. The Gradient Descent (GD) is such an optimizer [55]. GD begins with an initial guess of the parameters $\boldsymbol{\theta}_0$ (see Section 3.4) and uses the gradient at that point to obtain the direction of the most significant increase of the loss function. Then, it moves in the opposite direction to minimize the loss based on the following equation:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_n), \tag{3}$$

where η is the learning rate. When we use mini-batches, we use a modified version of the GD called Stochastic Gradient Descend (SGD), which looks like $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i)$, where n is the size of the batch and L_i is the loss per training example [55, 25].

The SGD or any other optimizer we might use requires an extensive computation of partial derivatives (gradients) with respect to the loss function. All modern machine learning and deep learning frameworks implement the backprop algorithm using Automatic Differentiation (AD). AD is a collection of methods to compute and evaluate partial derivatives of functions specified by computer programs [3]. AD should not be confused with numerical or symbolic differentiation. AD builds a computational graph of all the operations within a neural network based on primitive (basic) operations and elementary functions. Applying the chain rule over and over on the computational graph can evaluate all the partial derivatives needed by the SGD. For more details on optimizers, backprop, and AD, see [25, 11].

Let's put all the pieces together now and see how to train a neural network. First, we initialize the network's parameters. Then, we specify the optimization algorithm (or optimizer), the loss function or criterion, and we set the learning rate and any learning rate scheduler we might use. Then, we feed the neural network with the entire training data set at each epoch, looping over shuffled mini-batches (see Section 2) that comprise the training set. For each input or minibatch, the network returns an output or a batch of outputs, which we use to evaluate the loss, backpropagate the errors, minimize the loss, and eventually update the neural network's parameters.

Algorithm 1 summarizes the learning steps, and code listing 4 shows the backbone of training a neural network in Pytorch. In general, we first instantiate our neural network class (line 2 in code listing 4, and then we specify the optimizer (in this case, at line 4, we use the `Adam` optimizer [40]). In this work, we define custom loss functions per equation; we will see in Section 4.1 how to implement a custom loss function. Then, for each iteration (line 7), we sample our independent variables (*e.g.*, time t, space x, or both) based on predefined distributions. Once we get the samples, we zero the grads (line 10), then we approximate the solution `yhat` using the neural network and the independent variables. Finally, we pass the approximation to the loss function, do a backward step (lines 13-18), and finally minimize it (line 21).

```
1    # Initialize the neural network
2    net = ...
3    # Define the optimizer and the criterion
4    optimizer = Adam(net.parameters(), lr=0.0001)
```

[5]The reader can find more details on Pytorch Xavier initialization here: `https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.xavier_normal_`

Algorithm 1: Training Loop. An algorithmic representation of neural network's $\mathcal{F}$ training. η is the learning rate, n is the minibatch size.

Data: $\eta, \mathcal{F}, n$, epochs
Result: parameters $\boldsymbol{\theta}$
Initialize $\mathcal{F}(\boldsymbol{\theta})$;
Set Optimizer (*e.g.*, Adam);
Set Criterion (*e.g.*, MSE);
Set learning rate scheduler (*e.g.*, MultiStepLR);
for $e \leq epochs$ **do**
  $\hat{\mathbf{y}} = \mathcal{F}(\boldsymbol{\theta}; \mathbf{x})$;
  $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}) = \text{Criterion}(\mathbf{y}, \hat{\mathbf{y}})$;
  Compute $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathbf{x})$;
  Update parameters $\boldsymbol{\theta}$ using Optimizer;
end

```

    loss = ...  # To be discussed later

    for _ in range(n_iters):
            # Sample the independent variables
            t, x = torch.rand([batch_size, 1]), torch.rand([batch_size, 1])

            # Reset the gradients of all optimized tensor
            optimizer.zero_grad()

            # Forward pass
            y_hat = net(x, t)

            # Compute loss
            loss = DELoss(y_hat, x, t)

            # Backward pass
            loss.backward()

            # Minimize loss taking one step
            optimizer.step()
```

Listing 4: **Pytorch Learning (Training) Loop**

3.6 Vanishing & Exploding Gradients

As we described in the previous paragraph, the training of a neural network can be affected by two major factors. The first factor is the initialization of the neural network's parameters, and more precisely, the magnitudes of the initial guess, $\boldsymbol{\theta}_0$. If the initial guess is too high or too low, it can cause an explosion or a vanishing of the gradients because the increments of the update will be too big or too small, respectively. The second factor is the choice of the activation functions since they directly affect the behavior of the gradients, as we saw in the previous paragraph. If an activation function squeezes the gradients toward low values, such as the tanh function, then the magnitude of the gradients will start diminishing. On the other hand, if an activation function, such as the ReLU, that is not bounded allows the gradients to increase too much, they will probably explode. The former problem is known as vanishing gradients, and the latter as the exploding gradients.

To solve the exploding gradient, we can use either a gradient clipping, where we clip the gradients based on some norm, or batch normalization (see Section 3.7). On the other hand, we can solve the vanishing gradient problem by using activation functions that do not squeeze gradients toward smaller and smaller values. One such function is the linear rectification (ReLU) function [24], which returns zero for negative values and the identity function for the positive ones. Another way to overcome the problem of vanishing gradient is to use a recurrent neural network such as long-short-term memory (LSTM) [31] or gated recurrent unit(GRU) [13].

3.7 Batch Normalization

In the previous section, we discussed the vanishing and exploding gradients problem that might arise when training a neural network. Generalization errors are another potential problem we face while training a neural network. The

generalization error is the error a neural network commits when operating on data it has not previously trained on [25]. Typically, we want a neural network to perform well even on data that has not been seen before (provided that the data still comes from the same distribution as the training data set). When a neural network performs well even on unseen data, it generalizes well. Usually, we split the data set into three disjoint subsets: training, test, and validation data sets, and we evaluate the generalization error on the test data set. The problem arises when we end up with a simple neural network that cannot "learn" the primary trends and statistics of the data and has a significant generalization error. In this case, we say that the neural network is underfitting the data. On the other hand, we may end up with a complex neural network, or we do not have enough training data to train the network, leading to a model that "learns" the training data too well without generalizing [11]. In this case, we say that the neural network is overfitting the data.

We can use a regularization method to mitigate the problem of underfitting or overfitting. Regularization refers to machine learning methods that reduce errors in generalization (or test). Two of the most common regularization techniques are the L_1 and L_2, where we add an extra term, known as the penalty term, to the loss function. This term has a unique role depending on whether it's L_1 or L_2. In L_1 regularization, the penalty term makes the model sparse, meaning most of the model's parameters are zeros. On the other hand, in L_2 regularization, the penalty term pushes the model's weights to admit small values close to zero, preventing the model from becoming too complex and overfitting the data. Dropout is another commonly used method, where we randomly choose several units within a neural network's layer (or layers) and drop them along their connections [60]. Finally, batch normalization provides another regularizer where the input to a neural network layer is re-centered and re-scaled. Batch normalization can speed learning and make it more stable [37]. The reader can find more details and regularization techniques in Chapter 7 of [25].

In this work, we will focus on batch normalization, an algorithm that receives a mini-batch as input, computes its mean and variance, and then applies a normalization step followed by a scale and shift transform [37]. Batch normalization can speed up learning and render it more stable. In the literature on differential equations and deep learning, researchers have used batch normalization to speed learning when they solve differential equations with deep neural networks [10]. Thus, we briefly introduce the concept of batch normalization (BN) and later show how to use it when we train a neural network.

Let $\mathcal{B} = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n\}$ be a minibatch of size n with $\mathbf{x}^i = (\mathbf{x}^i_1, \ldots, \mathbf{x}^i_d)$. The batch normalization transform is given by

$$\begin{aligned} \hat{\mathbf{x}}^i_j &= \frac{\mathbf{x}^i_j - \mu^i_{\mathcal{B}}}{\sqrt{(\sigma^i)^2_{\mathcal{B}} + \epsilon}}, \quad i \in [1, n],\ j \in [1, d], \\ \hat{\mathbf{y}}^{i,l}_j &= \alpha^i \hat{\mathbf{x}}^i_j + \beta^i, \quad l \in [1, L]. \end{aligned} \tag{4}$$

where $\mu^i_{\mathcal{B}}$ and $\sigma^i_{\mathcal{B}}$ are the per-dimension mean and variance, respectively. ϵ is a tiny constant added to the denominator for numerical stability, and α and β are learnable parameters. $\hat{\mathbf{y}}^{i,l}$ here is the transformation output, and it refers to the output of a layer l. Equation (4) is used in training only. The population statistics replace the mean and the variance during the inference. The reader can find more details about BN in [37].

3.8 Universal Approximation Theorem

So far, we have seen some fundamental aspects of deep learning, such as different types of neural networks and how to initialize them properly, the backprop algorithm, and batch normalization. Nevertheless, feed-forward neural networks are valuable tools in approximating functions. For instance, we can use a straightforward neural network of one hidden layer with three units to approximate function $f(x) = \sin(3x)$ in the interval $[-1, 1]$. Figure 3 shows in black dots the data points we used to train the neural network,

$$\text{Input}(1) \rightsquigarrow \text{FC}^{\text{h}}(3) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{out}}(1),$$

and the orange line shows the approximation the neural network achieved after training. The neural network, here, has one hidden (FC^{h}) and one output (FC^{out}) layer implemented in Pytorch as `Linear` layers.

Neural networks with one hidden layer can, in theory, approximate any continuous function defined in a compact subspace of $\mathbb{R}$ per the Universal Approximation Theorem [17, 35]. More precisely, and according to [17], I_q is the q-dimensional unit cube, $[0, 1]^q$, and $C(I_q)$ is the space of continuous functions on I_q, and $\alpha_j, \beta_j \in \mathbb{R}$ are fixed parameters.

Theorem 1 *Let σ be any continuous sigmoidal function. Then, the finite sums of the form*

$$G(\mathbf{x}) = \sum_{j=1}^{M} \alpha_j \sigma(\mathbf{y}_j^T \mathbf{x} + \beta_j), \tag{5}$$

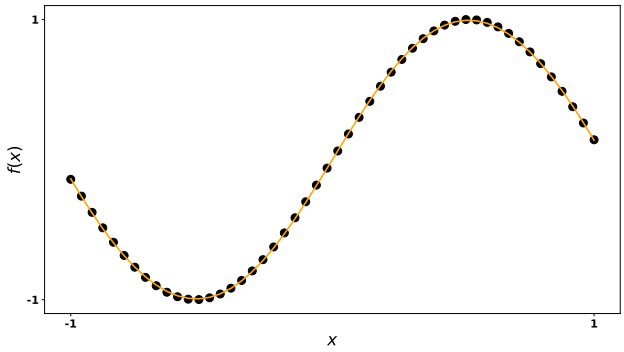

Figure 3: **Universal Approximation Theorem.** A neural network with one hidden layer, three hidden units, and a tanh activation function approximates the function $f(x) = \sin(3x)$ in the interval $[-1, 1]$. Fifty samples of the function $f(x)$ used to train the neural network are shown here as black dots. The orange line indicates the approximation, $\hat{f}(x)$ provided by the neural network.

are dense in $C(I_n)$*. In other words, given any* $f \in C(I_n)$ *and* $\epsilon > 0$*, there is a sum,* $G(\mathbf{x})$ *of the above form, for which*

$$|G(\mathbf{x}) - f(\mathbf{x})| < \epsilon \quad \textit{for all } \mathbf{x} \in I_n. \tag{6}$$

In simple words, a feed-forward neural network with a single hidden layer followed by a nonlinear activation function such as a sigmoid ($\sigma(x) = \frac{1}{1+\exp(-x)}$) and a linear output layer can approximate a continuous function defined on a compact subset of $\mathbb{R}^q$. Although the theorem states that it is possible to approximate any such function, it becomes apparent in practice that it needs to discuss the number of hidden units we will have to use to approximate a function of interest. However, deep feed-forward neural networks seem to work in practice and approximate functions defined on compact subsets of $\mathbb{R}^q$. Furthermore, a variation of the Universal Approximation theorem regarding the first derivatives of continuous functions has been proposed by Hornik et al. [36].

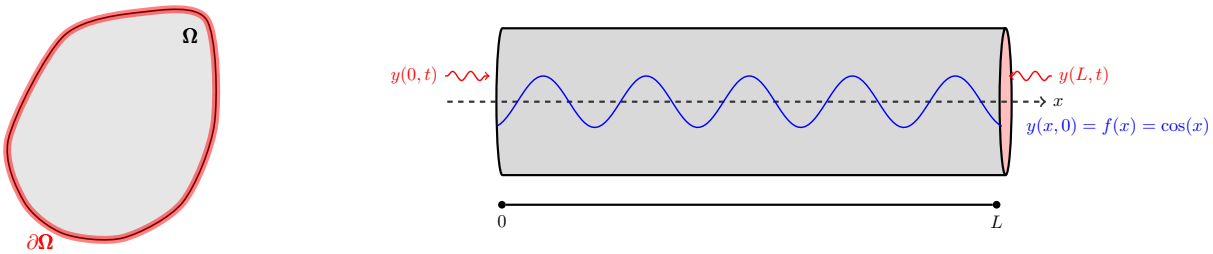

Figure 4: **Domain and boundary schematics.** Left Panel shows a domain Ω of a partial differential equation in gray color and its boundary $\partial\Omega$ in red color. The right panel shows a rod of size L (gray color), which is used to simulate heat diffusion (see main text). The boundary conditions $y(x, 0)$ and $y(x, L)$ are depicted with red color fonts, and the initial condition $y(x, 0) = f(x) = \cos(x)$ with a blue sinusoidal line.

4 Solving Partial Differential Equations

Before we delve into the deep Galerkin method and how to numerically approximate solutions to partial differential equations using deep neural networks, here are a few words about PDEs. PDEs typically involve partial derivatives of the dependent variable with respect to time and space, and they describe distributed physical systems (compared to lumped systems described by ordinary differential equations, and the dependent variables are a function of time alone). In general, a PDE is defined as a system of order $m \in \mathbb{N}$ by the equation:

$$\mathbf{F}(\mathbf{x}, D\mathbf{y}, D^2\mathbf{y}, \ldots, D^m\mathbf{y}) = 0, \tag{7}$$

Where $\mathbf{x} \in \Omega \subseteq \mathbb{R}^q$, $q \geq 2$ are the independent variables, and $\mathbf{y} = (y_1, y_2, \ldots, y_p)$ are the unknown functions, y_i, $i = 1, \ldots, p$ are functions of time and space, and $D = \frac{\partial^\alpha y}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \cdots \partial x_q^{\alpha_q}}$, describes all the partial derivatives. Finally, when $p \geq 2$, then we have a system of PDEs; otherwise, we have a single PDE or a scalar PDE.

PDEs are usually studied as initial-boundary-value-problems (IBVP), where one defines a domain, Ω, a boundary $\partial\Omega$, and some initial conditions when $t = 0$. Boundary Conditions are, in essence, constraints imposed on the boundary of a PDE's domain, and the solution of a PDE must satisfy these boundary and initial conditions. The left panel of Figure 4 shows a schematic representation of a domain Ω in gray color and its boundary $\partial\Omega$ in red.

Imagine a rod for which we want to study heat diffusion when we apply a heat source on one of its ends at time $t = 0$. The inside of the rod is the domain, Ω, where we define and solve the heat equation to obtain the diffusion inside the rod (see the gray area in the right panel of Figure 4). On the other hand, the boundary is the surface of the rod and its left and right edges (see again the left panel of Figure 4, red color). On the boundary, we can assume that different things are happening. For example, suppose we place the heat source on the right side of the rod and think that the rest is insulated. In that case, we will impose a boundary condition on the right side that will follow some function of time and provide the necessary heat. Moreover, we can assume that the values of the heat equation on the rest of the boundary are constant and equal to zero. Therefore, we see that boundary conditions will tell us how the heat equation behaves on the surface of the cylinder and at its end sides. Overall, there are five basic types of boundary conditions:

- **Dirichlet** is the simplest boundary condition (BC), where $y(\mathbf{x}, t) = g(\mathbf{x}, t)$, when $\mathbf{x} \in \partial\Omega$. We use this type of BC when a physical quantity, such as temperature, is kept constant on the boundary.
- **Neumann** is described by $\frac{\partial y(\mathbf{x},t)}{\partial \mathbf{n}} = g(\mathbf{x}, t)$, where $\mathbf{n}$ is the (exterior) normal to the boundary, and $\frac{\partial y(\mathbf{x},t)}{\partial \mathbf{n}}$ is the normal derivative. We use the Neumann boundary conditions when there is a reflection, a flux, or a gradient at the boundary.
- **Robin** follows the equation $ay(\mathbf{x}, t) + b\frac{\partial y(\mathbf{x},t)}{\partial \mathbf{n}} = g(\mathbf{x}, t)$, where $a, b \in \mathbb{R}$ constants, and $\mathbf{n}$ is the normal vector on the boundary.
- **Mixed** conditions are combinations of any of the previous types. For instance, on one end of a rod, we might have constant temperature, thus a Dirichlet boundary condition, and on the other end, a flux of heat, hence a Neumann boundary condition.
- **Cauchy** guarantees the uniqueness of a solution by imposing a Dirichlet and a Neumann boundary condition simultaneously.

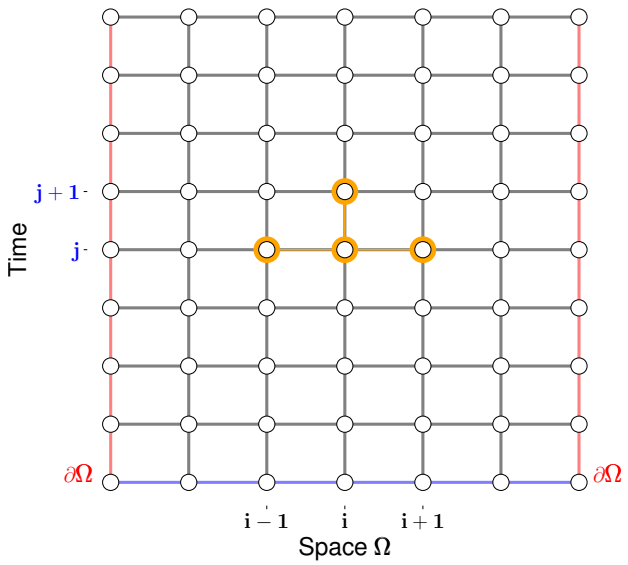

Figure 5: **One-dimensional Finite Differences Scheme** for a one-dimensional problem. The spatio-temporal discrete grid appears as circles and and the stencil of the finite differences is shown in orange color. The blue line indicates the initial conditions, and the red one shows the boundary conditions. The blue js and black is reflect the temporal and spatial discrete steps, respectively.

The reader can find more information on PDEs in [63, 19] since we will not go into more detail on PDEs since that is outside the scope of this work.

Let's return to Equation (7) and rewrite it as an initial-boundary-value-problem (IVBP) of an unknown spatiotemporal function $y(\mathbf{x}, t)$ defined on the domain $\Omega \times [0, T]$, where $\Omega \in \mathbb{R}^d$, and $\partial\Omega$ is the boundary of Ω. Then y satisfies the IBVP:

$$\begin{aligned} (\partial_t + \mathcal{T})y(\mathbf{x}, t) &= 0, && (\mathbf{x}, t) \in \Omega \times [0, T] \\ y(\mathbf{x}, 0) &= y_0(\mathbf{x}), && \mathbf{x} \in \Omega \\ y(\mathbf{x}, t) &= g(\mathbf{x}, t), && (\mathbf{x}, t) \in \partial\Omega \times [0, T], \end{aligned} \tag{8}$$

where $\mathcal{T}$ is an operator that describes the PDE, for instance, when $\mathcal{T}$ is a linear operator, then the PDE described by Equation (8) is a linear PDE. The function $y_0(\mathbf{x})$ represents the initial conditions, and function $g(\mathbf{x}, t)$ indicates the type of boundary conditions such as Dirichlet or Neumann. One common way to numerically solve the IVBP given by Equation (8) is to use the finite differences (FD) method. The main idea behind the FD methods is to discretize the domain of Equation (8) using regular grids, temporal and spatial. On the spatial nodes of the grid, the FD will estimate the spatial derivatives, and on the temporal nodes, the time derivatives. However, there are problems that we want to solve in strange or exotic domains that are hard to discretize using the FD method. Thus, we employ a different family of Galerkin methods to convert our numerical approximation problem into a variational one.

Finite Differences For example, let the differential operator $\mathcal{T}$ be $-\frac{\partial^2}{\partial x^2}$ and take as boundary conditions $y(0, t) = y(1, t) = 0$, and initial conditions $y(x, 0) = y_0(x)$, then we can rewrite Equation (8) in one-dimension as:

$$\begin{aligned} \frac{\partial y(x,t)}{\partial t} &= \frac{\partial^2 y(x,t)}{\partial x^2}, \\ y(x, 0) &= y_0(x), \\ y(0, t) &= y(1, t) = 0. \end{aligned} \tag{9}$$

Then, we can numerically solve the problem given by Equation (9) by discretizing the temporal and spatial derivatives using finite differences. One such method is the forward explicit scheme, where we discretize the temporal and spatial derivatives using M discrete nodes for the time derivatives and N for the spatial. Hence, the temporal and spatial nodes are given by $t_j = j\Delta_t$, where $j = 0, \ldots, M$ and $x_i = i\Delta x$, where $i = 0, \ldots, N$, respectively. Then we obtain the grid points (or nodes) (x_i, t_j) on which we will approximate the solution $y(x_i, t_j)$. Figure 5 shows the spatio-temporal discrete grid and the stencil (orange color) of the finite differences we use in this example. The blue color indicates

initial conditions, and the red one shows the boundary conditions. Finally, we approximate the temporal derivative using a forward difference approximation and the spatial derivative with a central difference approximation:

$$\frac{\partial y(x,t)}{\partial t} \approx \frac{y_{i,j+1} - y_{i,j}}{\Delta_t}, \qquad \frac{\partial^2 y(x,t)}{\partial x^2} \approx \frac{y_{i+1,j} - 2y_{i,j} + y_{i-1,j}}{\Delta_x^2}. \tag{10}$$

Now, we plug the approximated derivatives of Equation (10) to Equation (9) and we obtain

$$y_{i,j+1} = y_{i,j} + \alpha\Big(y_{i+1,j} - 2y_{i,j} + y_{i-1,j}\Big), \tag{11}$$

where $\alpha = \frac{\Delta t}{\Delta x^2}$. We apply a similar discretization on the boundary conditions, and we can then either iterate Equation (11) over i and j and obtain the solution, or we can rewrite Equation (11) as a system of linear equations and solve it using numerical methods from computational linear algebra. The reader can find more information about finite difference methods in [49, 64], such as the implicit and Crank-Nicolson methods.

Galerkin Methods As we already saw, FD methods discretize the continuous differential operator and solve the boundary problem using a discrete spatio-temporal domain. On the other hand, we can use Galerkin methods, which use linear combinations of basis functions to approximate the solution of the PDE. Let's formulate the general idea behind Galerkin methods through an example. Take the one-dimensional Poisson equation:

$$\begin{aligned} \frac{\partial^2 y}{\partial x^2} &= f, \quad \text{on } \Omega, \\ y &= 0, \quad \text{on } \partial\Omega, \end{aligned} \tag{12}$$

and we want to find a numerical approximation $\hat{y}$. To do so, we first define a finite-dimensional approximation space $\mathcal{X}_0^K$ and an associated set of basis functions $\{\phi_i \in \mathcal{X}_0^K\}$, where $i = 1, \ldots, k$. In addition, we require the basis functions to satisfy the boundary conditions such that $\phi_i = 0$ on $\partial\Omega$. We want to find a numerical approximation $\hat{y}(\mathbf{x}) = \sum_{j=1}^{k} \hat{y}_j \phi_j(\mathbf{x})$ that satisfies

$$-\int_\Omega \nabla v \cdot \nabla \hat{y}\, dV = \int_\Omega v f dV, \quad \forall v \in \mathcal{X}_0^K, \tag{13}$$

where $v \in \mathcal{X}_o^N$ is a test function. The problem described by Equation (13) is a vatiational problem, where we are looking to optimize functionals such that we obtain a numerical approximation, $\hat{y}$, of the solution y of Equation (12). Equation (13) constitutes the Galerkin formulation and from that equation, one can derive the Finite Element, Spectral Element, and Spectral Formulations. Here, we will not go into more details on those methods since they are out of the scope of this work. However, the reader can find more details in [43, 9].

4.1 Deep Galerkin Method

As we have already seen, the Deep Galerkin (DG) method was introduced by Sirignano and Spiliopoulos in [59], which can be seen as an extension of the Galerkin Methods (see Section 4 Galerkin Methods paragraph). Again, we will solve an optimization problem with the DG method as we did with the Galerkin method. However, only this time will we use neural networks to approximate the solution to the PDE instead of basis and test functions. The main idea behind the DG method is to define a loss function based on a given initial-boundary-value-problem, minimize that function by sampling points from the domain, boundary, and initial conditions of the IVBP, and pass them through a neural network that will approximate the solution.

In Section 3.8, we saw that a feed-forward neural network can approximate any continuous function defined on a compact subset of $\mathbb{R}^q$. DG takes advantage of the universal approximation theorem and uses a neural network $\mathcal{F}$ with parameters $\boldsymbol{\theta}$ to approximate the solution to Equation (8) by minimizing the objective function:

$$\mathcal{J}(\mathcal{F}(\boldsymbol{\theta};\mathbf{x},t)) = \underbrace{||(\partial_t + \mathcal{T})\mathcal{F}(\boldsymbol{\theta};\mathbf{x},t)||^2_{[0,T]\times\Omega,\nu_1}}_{\text{Differential Operator}} + \underbrace{||\mathcal{F}(\boldsymbol{\theta};\mathbf{x},t) - g(\mathbf{x},t)||^2_{[0,T]\times\partial\Omega,\nu_2}}_{\text{Boundary Conditions}} + \underbrace{||\mathcal{F}(\boldsymbol{\theta};\mathbf{x},0) - y_0(\mathbf{x})||^2_{\Omega,\nu_3}}_{\text{Initial Conditions}}, \tag{14}$$

where the first term tells us how good our approximation of the differential operator is, and the second and third terms indicate how well we approximate the boundary and the initial values, respectively. The norm is an L^2 given by $||f(y)||^2_{\mathcal{Y},\nu} = \int_{\mathcal{Y}} |f(y)|^2 \nu(y) dy$ with ν being a density defined on $\mathcal{Y}$. In Equation (15), $\mathcal{Y}$ becomes $[0,T]\times\Omega$, $[0,T]\times\partial\Omega$, and Ω in the three terms of the right-hand side, respectively. By minimizing Equation (15), we obtain an approximated solution to the IVBP described by Equation (8) (*i.e.*, when $\mathcal{J}(\mathcal{F}(\boldsymbol{\theta};\mathbf{x},t)) = 0$, then the solution $\hat{y}$ provided by the

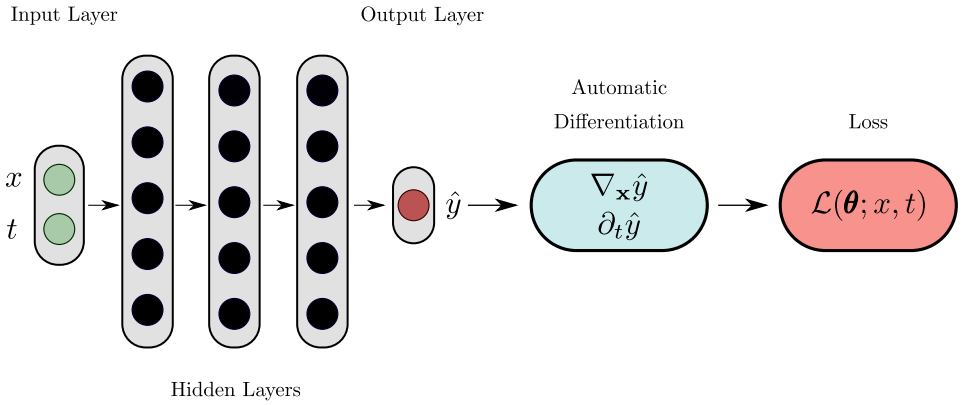

Figure 6: **Deep Galerkin Method Schematic**. A feed-forward neural network (MLP) takes an input (x, t) (here we assume that the $x \in \mathbb{R}$) and returns an output $\hat{y}$. Then, we use automatic differentiation to compute the derivatives $\partial_t\hat{y}$ and any spatial derivative $\nabla_x\hat{y}$. Then, we plug the derivatives in the loss function (15) and use the result to train the neural network. See main text and algorithm 2 for more details on the DG method.

neural network $\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}, t)$ will satisfy the IVBP (8)). Therefore, our main objective is to find a set of parameters $\boldsymbol{\theta}$ of a neural network $\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}, t)$) that will minimize the loss function described by Equation (15). However, and accordinly to [59], this is not always possible since when the dimension q (q is the dimension of domain and variables, see Section 2) of the problem increases, the integrals over Ω become intractable. To overcome this problem we employ a machine learning method where we can randomly draw samples from distributions on Ω and $\partial\Omega$ and estimate the squared error:

$$\mathcal{L}(\boldsymbol{\theta}) = \Big(\partial_t\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}_\Omega, t_\Omega) + \mathcal{T}\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}, t_\Omega)\Big)^2 + \Big(\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}_{\partial\Omega}, t_{\partial\Omega}) - g(\mathbf{x}_{\partial\Omega}, t_{\partial\Omega})\Big)^2 + \Big(\mathcal{F}(\boldsymbol{\theta}; \mathbf{x}_{t_0}, 0) - y_0(\mathbf{x}_{t_0})\Big)^2, \tag{15}$$

where $(\mathbf{x}_\Omega, t_\Omega) \sim \mathcal{U}(\Omega \times [0, T])$, $(\mathbf{x}_{\partial\Omega}, t_{\partial\Omega}) \sim \mathcal{U}(\partial\Omega \times [0, T])$, and $\mathbf{x}_{t_0} \sim \mathcal{U}(\Omega)$. Thus, we have to find the optimum parameters $\boldsymbol{\theta}$ that minimize Equation (15), to find an approximated solution to problem (8). Figure 6 depicts the main idea of the DG method, where a neural network (left side of the picture) receives as input (olive green nodes), $\mathbf{x}$, the independent variables (*i.e.*, time, and spatial components) of the IVBP, and the output layer (red node) returns the approximation to the solution, $\hat{y}$. Automatic differentiation uses the inputs and the output of the neural network to estimate the temporal (*i.e.*, $\partial_t\hat{y}$) and the spatial (*i.e.*, $\nabla_{\mathbf{x}}\hat{y}$) derivatives (teal colored box). More precisely, we use the `autograd` package of Pytorch to estimate the derivatives. Finally, we estimate the loss function and proceed with its minimization (red box). The neural network parameters are updated based on the loss function and through the backprop and the Adam optimizer (see below for more details on implementing the method).

Algorithm 2: Deep Galerkin Method.

Data: $\eta, \nu_1, \nu_2, \nu_3, \mathcal{F}, n$, Iterations
Result: parameters θ
Initialize $\mathcal{F}(\boldsymbol{\theta})$;
for $i \leq$ *Iterations* **do**
  Generate $(\mathbf{x}, t)_{\Omega\times[0,T]} \sim \nu_1$;
  Generate $(\mathbf{x}, t)_{\partial\Omega\times[0,T]} \sim \nu_2$;
  Generate $(\mathbf{x}, 0)_\Omega \sim \nu_3$;
  Compute $y_{\Omega\times[0,T]} = \mathcal{F}(\boldsymbol{\theta}; (\mathbf{x}, t)_{\Omega\times[0,T]})$;
  Compute $y_{\partial\Omega\times[0,T]} = \mathcal{F}(\boldsymbol{\theta}; (\mathbf{x}, t)_{\partial\Omega\times[0,T]})$;
  Compute $y_{\Omega,t=0} = \mathcal{F}(\boldsymbol{\theta}; (\mathbf{x}, 0)_\Omega)$;
  Compute $\partial_t\hat{y}$ and $\nabla_{\mathbf{x}}\hat{y}$;
  Compute $\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, t)$ based on equation (15);
  Compute $\Delta\boldsymbol{\theta} = \nabla_\theta\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}, t)$;
  Update parameters $\boldsymbol{\theta}$;
  if *learning rate schedule condition met* **then**
    Adjust η;
end

Finally, algorithm 2 outlines the steps we must follow to minimize the loss function (15). First, we initialize the

parameters $\boldsymbol{\theta}$ (weights and biases) of the neural network and set the number of learning iterations. For each iteration, we generate samples by randomly drawing from the distributions ν_1, ν_2, and ν_3 of the domain Ω, the boundary $\partial\Omega$, and the initial conditions, respectively. Then we pass the samples as input to the neural network to compute the corresponding values $y_{\Omega\times[0,T]}$, $y_{\partial\Omega\times[0,T]}$, and $y_{\Omega,t=0}$ for solving the problem in the domain, on the boundary, and at the initial time $t = 0$, respectively. In the next step, we compute the loss based on Equation (15), we backpropagate the errors and then update the parameters $\boldsymbol{\theta}$ using an optimization algorithm. In addition, if we use a learning rate scheduler, we check if the condition(s) of the scheduler are satisfied, and if yes, then we proceed to update the learning rate η accordingly. However, if we use a Pytorch scheduler, we don't have to check any condition; the scheduler will take care of that for us.

4.2 Evaluate approximations

After we have minimized the loss and obtained the approximated solution, $\hat{\mathbf{y}}$, we compare it against either the exact solution, $\mathbf{y}$, of the differential equation if it's available or against a numerical solution provided by a traditional differential equation numerical solver. To this end, we estimate the mean absolute error:

$$\text{MAE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|. \tag{16}$$

We use the mean absolute error because we use batches. Thus, we can compute the errors between entire batches at once and get an estimate of how close or how far the two solutions are. To this end, we employ the function `mean_absolute_error` of the `Scikit-learn` Python package [52].

4.3 Heat Equation

This section shows the DG method for solving the one-dimensional heat equation. The unknown function that describes the heat at time t and at spatial point x is $y(x,t)$ and the equation describes the boundary value problem:

$$\begin{cases} \partial_t y(x,t) = \kappa \Delta y(x,t), \quad (x,t) \in [0,\pi] \times (0,3] & \\ u(x,0) = \sin(x) & \text{(Initial conditions)} \\ u(0,t) = u(1,t) = 0 & \text{(Boundary conditions)} \end{cases} \tag{17}$$

where the Laplacian of y is $\Delta y = \frac{\partial^2 y}{\partial x^2}$, and κ is the thermal conductivity constant. Equation (17) has an exact solution in closed form $y(x,t) = \sin(x)\exp(-\kappa t)$. Notice that Equation 17 has constant boundary conditions (Dirichlet), and the initial conditions indicate that at time $t = 0$, the heat spatial distribution is sinusoidal.

To numerically solve the IVBP (17) using the DG method, we first must define a loss function that we will minimize using a neural network to approximate the solution $y(x,t)$ of $\partial_t y(x,t) - \kappa\frac{\partial^2 y(x,t)}{\partial x^2} = 0$. Therefore, we start with the equation (15), and we plug in the terms given by our boundary value problem in equation (17):

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) = &\frac{1}{n}\sum_{i=1}^{n}\Big(\frac{\partial \hat{y}(x,t)}{\partial t} - \kappa\frac{\partial^2 \hat{y}(x,t)}{\partial x^2}\Big)^2 \\ &+ \frac{1}{n}\sum_{i=1}^{n}(\hat{y}(x,0) - \sin(x))^2 \\ &+ \frac{1}{n}\sum_{i=1}^{n}(\hat{y}(0,t))^2 + \frac{1}{n}\sum_{i=1}^{n}(\hat{y}(1,t))^2. \end{aligned} \tag{18}$$

The last term in Equation (18) includes only the solution approximated by a neural network on the boundaries. Since the boundary conditions are zero, we have nothing to subtract as we did for the initial conditions. Remember that we average over the minibatch of size n.

The following code listing shows the Pytorch implementation of the loss function based on Equation (18). The first step is to approximate the solution within the domain Ω using a neural network (line 3 of the snippet). Then, it computes the temporal and second spatial derivatives using Automatic Differentiation. Finally, it computes all three terms of Equation (18) and returns the mean over the batches.

```
1 def heat1d_loss_func(net, x, x0, xbd1, xbd2, x_bd1, x_bd2):
2     kappa = 1.0          # Thermal conductivity constant
3     y = net(x)           # Obtain a neural  network approximation of heat eq. solution
```

```
4
5       # Compute the gradient (derivatives)
6       dy = torch.autograd.grad(y,
7                                x,
8                                grad_outputs=torch.ones_like(y),
9                                create_graph=True,
10                               retain_graph=True)[0]
11      dydt = dy[:, 1].unsqueeze(1)     # Get the temporal derivative
12      dydx = dy[:, 0].unsqueeze(1)     # Get the spatial first derivative
13
14      # Compute the second partial derivative
15      dydxx = torch.autograd.grad(dydx,
16                                  x,
17                                  grad_outputs=torch.ones_like(u),
18                                  create_graph=True,
19                                  retain_graph=True)[0][:, 0].unsqueeze(1)
20
21      # Compute the loss within the domain
22      L_domain = ((dydt - kappa * dydxx)**2)
23
24      # Compute the initial conditions loss term
25      y0 = net(x0)
26      L_init = ((y0 - torch.sin(x0[:, 0].unsqueeze(1)))**2)
27
28      # Compute the boundary conditions loss terms
29      y_bd1 = net(xbd1)
30      y_bd2 = net(xbd2)
31      L_boundary = ((y_bd1 - x_bd1)**2 + (y_bd2 - x_bd2)**2)
32
33      # return the mean (over batches)
34      return torch.mean(L_domain + L_init + L_boundary)
```

Listing 5: **1D Heat Equation Loss Function**

Once we have defined the loss function, we can choose a neural network to approximate the solution. For this particular problem, we will use a multilayer feed-forward neural network (MLP) with four hidden layers with 128 units in each one, initializing their parameters with Xavier's uniform distribution. We apply a tanh function as non-linearity after each layer except the output one. The architecture of the neural network is:

$$\text{Input}(2) \rightsquigarrow \text{FC}^{\text{h}}(128) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{h}}(128) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{h}}(128) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{h}}(128) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{out}}(1).$$

As we can see, the neural network receives a two-dimensional input as the heat equation receives a temporal, t, and a spatial, x argument. The neural network's output is one-dimensional and reflects the temperature we get as a solution of the heat equation at point x and at time t. FC means fully connected layer, and it implements an affine transformation as we already have seen ($\mathbf{y} = \mathbf{W}\mathbf{x}+\mathbf{b}$). In Pytorch, we define fully connected layers using the `Linear()` class. In addition, the output layers are also fully connected layers, and only the input layer serves as a placeholder. Code listing 1 provides the implementation of the MLP neural network used in this work.

Since we have determined the neural network's type and architecture, we proceed by setting the mini-batch size $n = 64$. A mini-batch holds, in this case, 64 pairs of variables (x, t), and thus the neural network can provide approximations to 64 spatial points at 64 different time steps at once. Thus, we can accelerate the minimization process and make it smoother since we average the loss function over all the 64 points at each iteration. Finally, the number of iterations we will use to minimize the loss function (18) is set to $15,000$. At each iteration, and based on algorithm 2 we draw 64 samples from two uniform distributions, one for the temporal and another for the spatial component. In this case, we have $t \sim \mathcal{U}(0, 3)$ and $x \sim \mathcal{U}(0, \pi)$.

Finally, we have to pick an optimizer for our problem. We use the Adam optimizer based on what we discussed in Section 3.5. In Pytorch, one imports the Adam optimizer from the `optim` package. The code listing 4 shows how one can initialize (line 4) and call the optimizer (line 23) to minimize the loss function. In addition, the learning rate for this problem is set to 0.0001 and does not change during training.

After completing training, we obtain the approximated solution shown in Figure's 7 panel **B**. From visually inspecting it, it looks very similar to the analytical solution given in panel **A** in Figure 7. In addition, in Figure 7**C**, we see that the loss, indeed, drops close to zero after about $1,000$ iterations, indicating that our neural network successfully approximated the solution of Equation (17) and that the initial number of iterations we chose is too much for this particular problem. Finally, the mean absolute error (MAE) is 0.0529, reflecting the proximity of the analytical and the approximated solutions.

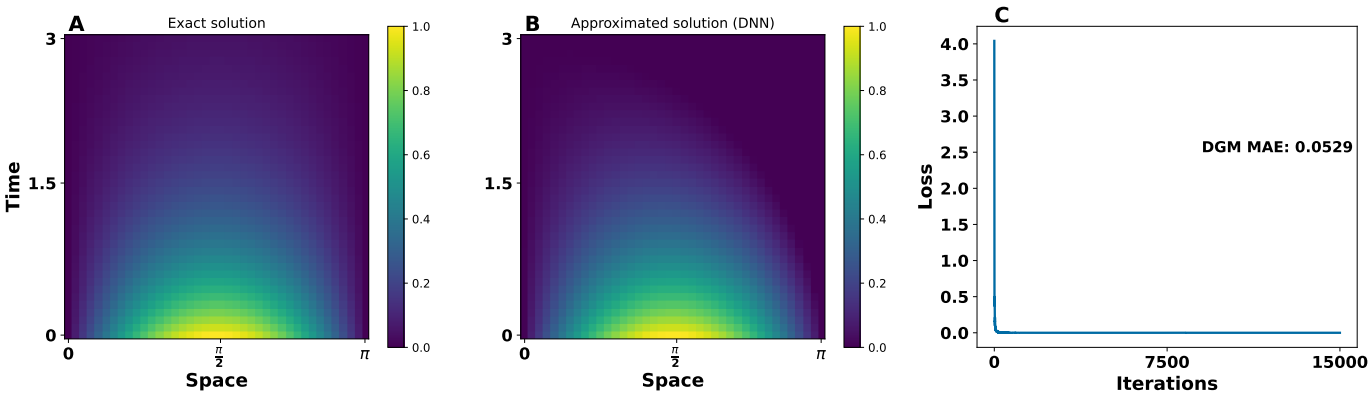

Figure 7: 1**D Heat equation solution.** Panel **A** shows the exact solution of Equation (17), **B** illustrates the approximated solution by the DG method and a deep feed-forward neural network, and panel **C** displays the loss over training iterations.

4.3.1 Effects of Batch Size on Minimization

As we already have seen, a minimization problem, especially one involving neural networks, has many parameters that can affect the results. For instance, the size of the neural network, the number of hidden units, the learning rate, and many others are called hyperparameters to distinguish from the actual parameters of a neural network, which are the weights and the biases. We will discuss the hyperparameters later and see how to search for good sets of them using specialized software. For now, we will focus on one of those hyperparameters, the mini-batch size. We will investigate how the mini-batch size affects the minimization process and, thus, the loss function of our problem. To this end, we solve the problem of the one-dimensional heat Equation (17) using the loss function (18), and we follow the methodology we described in the previous section. Only this time, we vary the size of the mini-batches using powers of two, starting from $2^0 = 1$ and going up to $2^{10} = 1024$. We reduce the number of iterations to $1,000$, and we run for each mini-batch size the script that solves the one-dimensional heat equation. As we see in Figure 8, the slowest loss decrease occurs when the batch size is set to one. Once we start increasing the size of a mini-batch, the convergence of the minimization takes place faster. However, after a batch size of 4, there is no practical convergence difference.

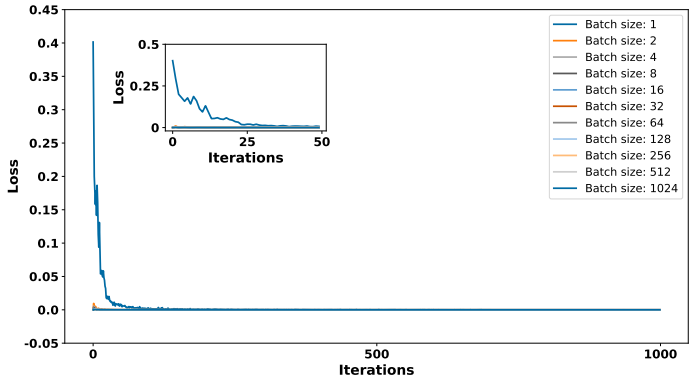

Figure 8: **Effect of Batch Size on Minimization** We see the evolution of the loss function (18) throughout $1,000$ iterations when we solve the initial-boundary-value-problem described by Equation (17). Different batch sizes as powers of two have been used in this case ($n \in \{2^k | \, k \in \mathbb{N} \text{ and } 0 \leq k \leq 10\}$). The inset shows the first 50 iterations of the minimization problem.

4.3.2 Effects of Batch Normalization on Minimization

In Section 3.7, we saw how batch normalization can improve and stabilize learning speed. To this end, we will examine how batch normalization affects the minimization process in our problem described by Equation (17), using the same parameters and neural network as in the previous sections with a batch size of 64. The code snippet 1 (set the flag `batcn_norm` to `True`) shows how we instantiate and use the one-dimensional batch normalization in Pytorch.

Figure 9 (top) shows the loss function throughout $1,000$ iterations, and as we see, the batch normalization that was applied after the activation function (non-linearity), which in this case is a tanh function, deteriorates the convergence of the minimization process (gray line). On the other hand, the batch normalization that was applied before the activation function smoothly minimizes the loss function given by Equation (18) as we see in Figure 9 (bottom). However, it's not better than the case where no batch normalization is used. We observe a similar effect even when we replace the tanh with a ReLU non-linearity. The fact that batch normalization does not improve convergence speed in this example does not mean that one does not have to use it. Every time we have to solve a problem, we should try different approaches because no general recipe works best for all problems. The heat equation we solve here is probably an *easy* problem, and thus, batch normalization unnecessarily increases the neural network's complexity. However, the take-home message here is always to use different tools and approaches since they might help solve a problem faster.

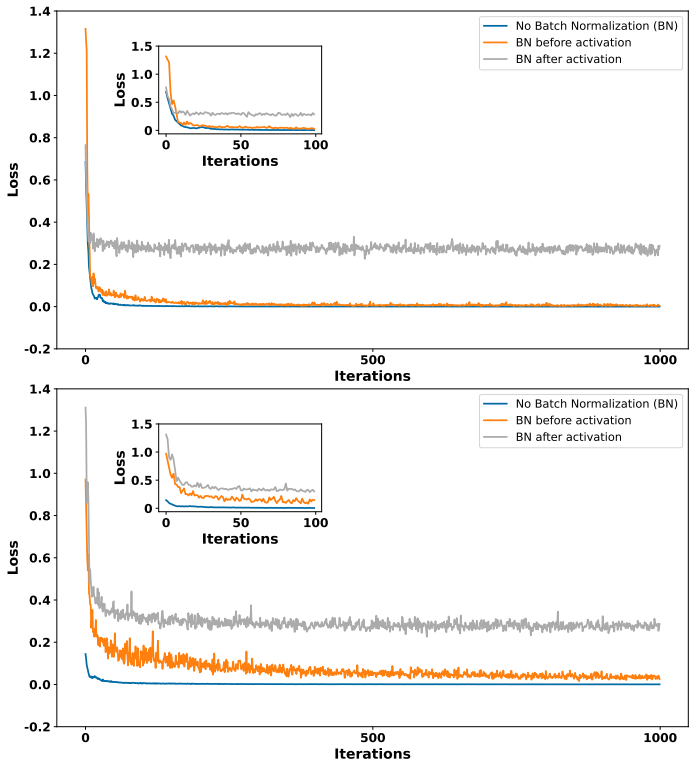

Figure 9: **Effect of Batch Normalization on Minimization** The top panel shows the loss function (Equation (18)) over time for $1,000$ iterations when we solve the initial-boundary-value-problem described by Equation (17), and the activation function of the neural network is a tanh function. The blue line indicates the loss when we do not use batch normalization. The orange and gray lines show the loss when we use batch normalization before and after the non-linearity (activation function), respectively. The bottom panel shows the effect of batch normalization on the same neural network when we replace the tanh with a *ReLU* function as non-linearity. The inset shows the loss function in all three cases for 100 iterations.

4.4 Search for the Hyperparameters

When we use neural networks, we usually face the problem of finding the correct hyperparameters. For instance, how many hidden layers do we need to solve a problem, how many units are in each layer, how many epochs, and what is the correct learning rate value? One way to solve this problem is to perform a manual hyperparameters search, although it takes time and effort. Another approach would be using a grid or random search algorithm. However, a more appropriate, clean, and less time-consuming method is to use a software package that does the searching (tuning) for us in a distributed and intelligent way. Such a package is the Ray Tune package.

Ray [48] is an open-source computing framework for scaling machine and deep learning applications. It provides many tools for implementing parallel and distributed processing without requiring the user to be an expert in parallel computing. Moreover, it comes with many packages, such as the Ray Tune for tuning machine learning models' hyperparameters. Ray Tune allows the user to define the search space and provides all the necessary means to search that space for the optimal hyperparameters by minimizing or maximizing a cost function provided by the user. In this

work, we combine the Ray Tune with the Optuna [1] package to search a space comprised of the number of learning iterations, the learning rate, the number of hidden layers, and the units within each layer. The cost function we try to minimize returns the MAE as defined by Equation (16).

To use the Ray Tune, first, we have to write our objective function (the one we will maximize/minimize). In our case, we are looking for the hyperparameters that will minimize the loss function defined by Equation (18). The hyperparameters we aim to optimize are the batch size, the learning rate, and the number of iterations. Of course, we could search for the number of layers and units per layer along with the other three parameters. However, we would like to keep things as simple as possible. Thus, our objective function takes as argument a Python dictionary `config` that contains all the hyperparameters we are optimizing for. Then, we define a neural network (lines 3–6 in code listing 6), and we use that neural network to minimize our loss function (18). Finally, we transmit the last value of the loss to Ray Tune (line 16) to be used in the hyperparameters optimization process. In this case, we are trying to find the set of hyperparameters that minimize the loss.

```
1  def objective(config):
2      # Define the neural network
3      net = MLP(input_dim=2,
4                output_dim=1,
5                hidden_size=128,
6                num_layers=3).to(device)
7
8      # Train the neural network to approximate the DE
9      _, loss_dgm = minimize_loss_dgm(net,
10                                     iterations=config["n_iters"],
11                                     batch_size=config["batch_size"],
12                                     lrate=config["lrate"],
13                                     )
14
15     # Report loss
16     session.report({"loss": loss_dgm[-1]})
```

Listing 6: **Objective function used with Ray Tune**

In the code listing 7, we show how to use the `objective` function to search for the three hyperparameters: batch size (`batch_size`), number of iterations (`n_iters`), and learning rate (`lrate`). First, we define the search space and specify the distributions from which we will draw candidate values for our hyperparameters (lines 3–6). Then, we choose the search algorithm (line 9), in this case, the Optuna [1] algorithm, and which scheduler we will use. Finally, we instantiate the Tuner class (lines 14–26) and pass as arguments the resources we will use (1 GPU and 10 CPU cores). We pass the objective function (`objective`) and the configuration, where we define the metric (loss in this case) and a minimization. Moreover, we pass the Optuna algorithm as an argument, the scheduler, and the number of samples for which the algorithm will perform the hyperparameters space search. Finally, we pass the search space (which is a Python dictionary), and we call the `fit` method. One can find more details on using Ray Tune on its documentation page[6].

[6] `https://docs.ray.io/en/latest/tune/index.html`

```python
def optimizeHeat():
    # Define the search space of hyperparameters
    search_space = {"batch_size": tune.randint(lower=1, upper=512),
                    "n_iters": tune.randint(1000, 50000),
                    "lrate": tune.loguniform(1e-4, 1e-1),
                    }

    # Set the Optuna optimization algorithm, and the scheduler
    algo = OptunaSearch()
    algo = ConcurrencyLimiter(algo, max_concurrent=5)
    scheduler = AsyncHyperBandScheduler()

    # Instantiate the Tuner class
    tuner = tune.Tuner(
                tune.with_resources(
                    objective,
                    resources={"cpu": 10,
                               "gpu": 1}),
                tune_config=tune.TuneConfig(metric="loss",
                                            mode="min",
                                            search_alg=algo,
                                            scheduler=scheduler,
                                            num_samples=10,
                                            ),
                param_space=search_space,
                )
    # Run the optimization
    results = tuner.fit()
```

Listing 7: **Hyperparameters tuning with Ray Tune**

5 Ordinary Differential Equations

In the previous section, we saw how to solve PDEs using the DG method, a powerful tool readily applied to partial differential equations. Importantly, the DG method is equally effective in solving ordinary differential equations (ODEs), systems of ODEs, and integral equations. Therefore, we continue this primer by showing how to solve ODEs and systems of ODEs.

5.1 Exponential Decay

Our first example is about a first-order linear ODE with initial conditions $y(0) = 2$, described by the following initial conditions problem:

$$\begin{aligned} \frac{dy(t)}{dt} &= y(t), \\ y(0) &= 2. \end{aligned} \tag{19}$$

Equation (19) admits an exponential solution, $y(t) = 2\exp(-t)$, that decays towards zero as time approaches infinity. Following the same recipe we used for the heat equation, we first define a loss function to solve Equation (19) with the DG method. We notice that Equation (19) does not have any boundary conditions; thus, we neglect the boundary condition term in equation (15), and hence we have

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d\hat{y}}{dt} + \hat{y}\right)^2 + \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_0 - 2\right)^2, \tag{20}$$

where $\hat{y}$ is the output of the neural network $\mathcal{F}(\boldsymbol{\theta}; t)$. In this case, the neural network receives a one-dimensional input, the time t, and returns a one-dimensional output y. The neural network has two hidden layers with 32 neurons each, one input layer, and one output layer with one neuron each. The overall architecture is

$$\text{Input}(1) \rightsquigarrow \text{FC}^{\text{h}}(32) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{h}}(32) \rightsquigarrow \text{Tanh} \rightsquigarrow \text{FC}^{\text{out}}(1).$$

```
def simple_ode_loss(y,  # Approximated solution (neural network)
                    y0, # Approximated solution (neural network) at t = 0
                    t,  # Input variable - time
                    y_ic):  # Initial condition value

    # Compute the temporal derivative
    dydt = torch.autograd.grad(y,
                               t,
                               grad_outputs=torch.ones_like(y),
                               create_graph=True,
                               retain_graph=True)[0]

    L_domain = ((dydt + y)**2)

    L_init = ((y0 - y_ic)**2)
    return torch.mean(L_domain + L_init)
```

Listing 8: **Loss function for the FitzHugh-Nagumo system.**

We discretize the temporal dimension using 64 time steps. The input to the neural network is organized into mini-batches of size $n = 64$ and thus has a $(n, 1)$ shape. We populate the tensor at each iteration with 64 data points sampled from a uniform distribution $\mathcal{U}(0, 1)$ (since $t \in [0, 1]$). By using mini-batches, we can accelerate the convergence of learning and speed up the computations since we exploit GPU's architecture more efficiently due to parallel processing of mini-batches and make it more stable [25]. We minimize the loss for $5,000$ iterations with a fixed learning rate $\eta = 0.0001$, and using the Adam optimizer. Figure 10 **A** shows the solution obtained by the neural network in orange. We see that it is very close to the analytic solution (blue solid line), something that we can confirm by estimating the MAE, which is MAE $= 0.0017$. In addition, panel **C** shows the loss of the DG method. The DG method converges to zero at about 500 iterations and stabilizes without oscillating or diverging.

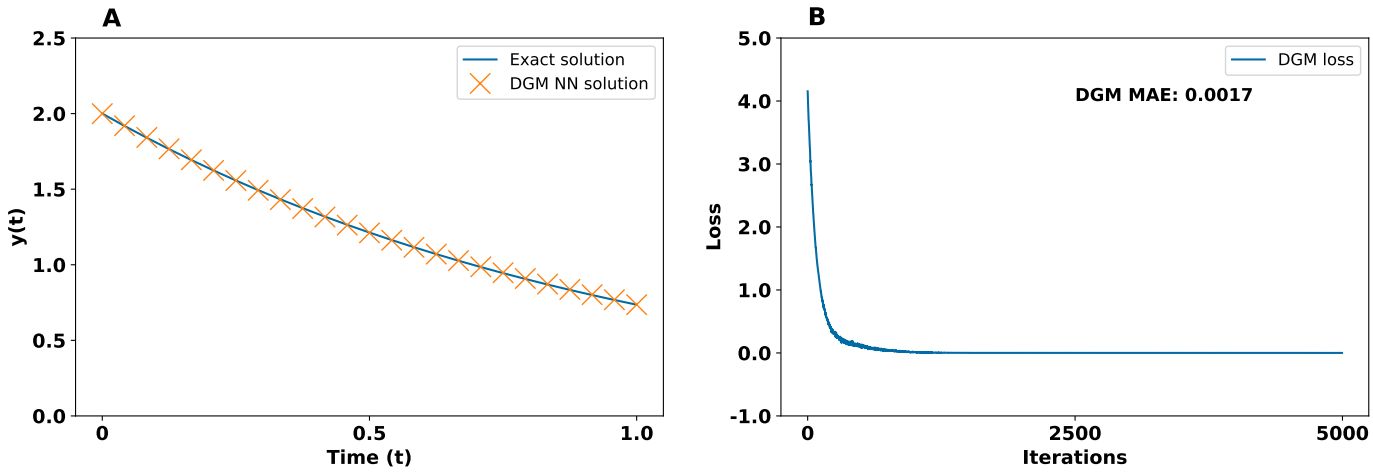

Figure 10: **First-order ordinary differential equation solution. A** Solution of Equation (19) in the interval $[0, 1]$. The blue solid line shows the analytic solution $y(t) = 2\exp(-t)$, and the orange crosses represent the solution obtained by the Deep Galerkin (DG) method. The approximated solution is close to the analytic one (mean absolute error MAE= 0.0017). **B** Evolution of loss for the DG method.

5.2 FitzHugh-Nagumo Model

The second example is the FitzHugh-Nagumo model [20, 50], a system of two coupled ordinary differential equations (two-dimensional system) that describes the dynamics of neuronal excitability. It reduces the Hodgkin-Huxley model of action potential generation in the squid giant axon [32]. The following equations describe the FitzHugh-Nagumo model:

$$\begin{aligned} \frac{dy(t)}{dt} &= y(t) - \frac{y(t)^3}{3} - w(t) + I_{\text{external}}, \\ \tau\frac{dw(t)}{dt} &= y(t) + \alpha - \beta w(t), \end{aligned} \tag{21}$$

where $y(t)$ is the membrane potential, $w(t)$ is a recovery variable, I_{external} is the magnitude of the external current applied on the membrane, α and β are constants, and τ is a time decay constant. The variable y allows for regenerative self-excitation via positive feedback, and the w follows linear dynamics, as seen in equation (21), and provides a slower negative feedback that acts as a recovery mechanism. Figure 11 illustrates a typical solution (blue line) of Equation (21) when $\alpha = 0.7$, $\beta = 0.8$, $\tau = 2.5$, and $I_{\text{external}} = 0.5$. Panels **A** and **B** show the y and w solutions in blue, respectively. We solve numerically Equation (21) using the `odeint` class of `Scipy`. The initial conditions are $y(0) = 0$ and $w(0) = 0$, and we solve for $t = 30\,\text{s}$ discretized with 50 nodes.

On the other hand, we can solve the problem given by Equation (21) using the DG method. The major difference between the current problem and the simple ODE of the previous example is that we have to add an extra term in the loss function to describe the dynamics of $w(t)$ and adjust the output of the neural network since we have two dependent variables, $y(t)$ and $w(t)$, in this case. Therefore we proceed in defining the loss function based on equation (15), and we get

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) = &\frac{1}{n}\sum_{i=1}^{n}\left(\frac{d\hat{y}}{dt} + (\frac{\hat{y}^3}{3} + \hat{y} - I_{\text{external}} - \hat{w})\right)^2 \\ &+ \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d\hat{w}}{dt} + (\beta\hat{w} - \alpha - \hat{y})/\tau\right)^2 \\ &+ \frac{1}{n}\sum_{i=1}^{n}\left(\hat{\mathbf{s}}_0 - \mathbf{y}_{\text{IC}})\right)^2, \end{aligned} \tag{22}$$

where $\hat{y}$ and $\hat{w}$ are the solutions approximated by the neural network for y and w, respectively. The tensor $\mathbf{y}_{\text{IC}}$ contains the initial conditions and it has a shape $(n, 2)$. The output tensor of the neural network at time $t = 0$ is $\hat{\mathbf{s}}_0$ and has a shape $(n, 2)$, where $n = 256$ is the batch size. The neural network's output is two-dimensional since we have two dependent variables, y and w, and that's why we use an extra tensor $\hat{\mathbf{s}}$ to store the output. In Pytorch we assign

`y = s[:, 0]` and `w = s[:, 1]`. Generally, when we have a q-dimensional system, the neural network's output would be a q-dimensional tensor. Moreover, remember that the neural network takes a one-dimensional input, the argument t in equation (21). All the parameters of the FitzHugh-Nagumo equations receive the same values as before.

For this particular problem, we choose a DGM-type neural network with one input unit that represents the time variable t, two output units that reflect the solutions $y(t)$ and $w(t)$, and finally, four hidden DG layers with 128 units each. We decided to go with a DMG-type neural network because of the rapidly changing dynamics of Equation (21). The code snippet 2 gives the implementation of the DGM layer used here. Moreover, all the activation functions of the DGM neural network are linear rectifiers (ReLU). The overall architecture of the neural network is

$$\text{Input}(1) \rightsquigarrow \text{FC}(128) \rightsquigarrow \text{ReLU} \rightsquigarrow \text{DGML}(128) \rightsquigarrow \text{DGML}(128) \rightsquigarrow \text{DGML}(128) \rightsquigarrow \text{DGML}(128) \rightsquigarrow \text{FC}^{\text{out}}(2).$$

Finally, we use the `grad` method of Pytorch to compute the temporal derivatives $\frac{d\hat{y}}{dt}$ and $\frac{d\hat{w}}{dt}$. An alternative is to use the `jacobian` method of the `autograd` since the neural network's output is not scalar (one-dimensional). However, the `jacobian` is generally slower than the `grad`, which is applied on scalar neural network outputs. Thus, we separately apply the `grad` method on each neural network output. The code listing 9 below shows the implementation of the loss function for the FitzHugh-Nagumo problem.

```
def fitzhugh_nagumo_loss(y,     # Solution approximated by a neural network
                         y0,    # Solution approximated by a neural network at t = 0
                         t,     # Input variable - time
                         y_ic): # Initial conditions
    Iext = 0.5      # External current
    alpha, beta, tau = 0.7, 0.8, 2.5    # FitzHugh-Nagumo model parameters

    # Get the variables y, w
    Y, W = y[:, 0].unsqueeze(1), y[:, 1].unsqueeze(1)

    dY = torch.autograd.grad(Y,
                             t,
                             grad_outputs=torch.ones_like(Y),
                             create_graph=True,
                             retain_graph=True)[0]

    dW = torch.autograd.grad(W,
                             t,
                             grad_outputs=torch.ones_like(W),
                             create_graph=True,
                             retain_graph=True)[0]

    Lx = torch.sum((dY + (Y**3/3.0 - Y - Iext + W))**2)
    Ly = torch.sum((dW + (beta * W - alpha - Y) / tau)**2)
    L0 = torch.sum((y0 - y_ic)**2)

    return Lx + Ly + L0
```

Listing 9: **Loss function for the FitzHugh-Nagumo system.**

The optimizer of choice here is the Adam optimizer with its default parameters. We run the training loop for $150,000$ iterations, drawing at each iteration a random sample from a uniform distribution $\mathcal{U}(0, 30)$ since we are interested in solving for $t \in [0, 30]$. We start our training with a learning rate $\eta = 0.0001$ and reduce it to 0.00001 after $35,000$ iterations. All the other parameters concerning Equation (21) are the same as the ones we previously used with the `odeint` solver.

Figure 21 shows the solution we obtained after training the neural network. We see in panels **A** and **B** in orange color the solutions $y(t)$ and $w(t)$, respectively. We observe that the numerical solution we obtained from the `odeint` solver matches the one we got from the neural network, and the MAE is 0.0088, meaning that the two solutions are close. Panel **C** shows the loss function dropping to zero after $10,000$ iterations and stabilizing after about $90,000$ iterations.

6 Fredholm Integral Equations

Integral equations, as their name reveals, involve an unknown function $y : [a, b] \to \mathbb{R}$, and its integrals. In their general form, integral equations are described by the following equation:

$$\alpha(x)y(x) = g(x) + \lambda \int_{a(x)}^{b(x)} K(x, t)y(t)dt, \tag{23}$$

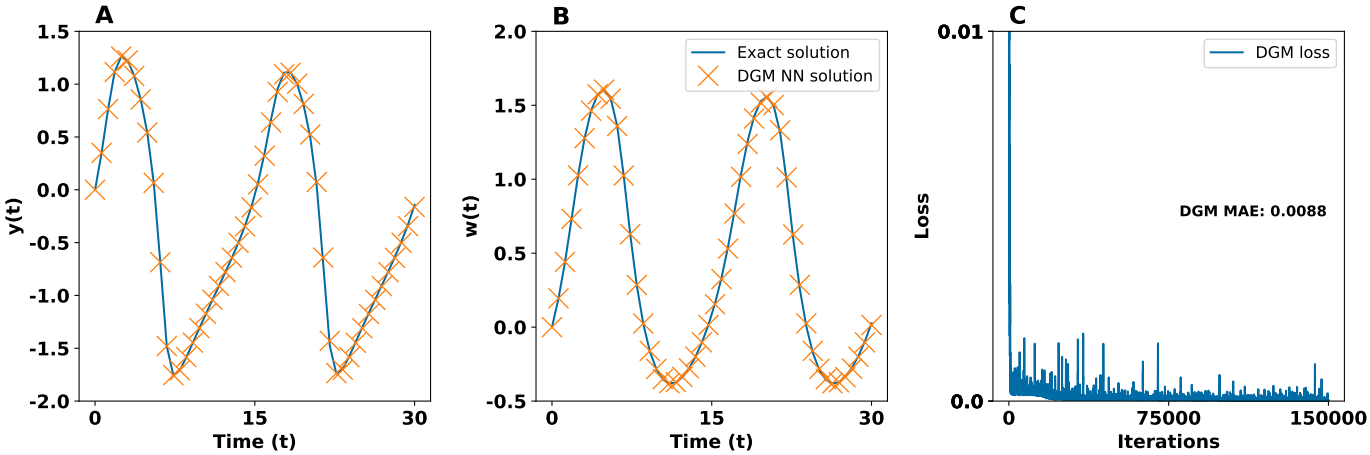

Figure 11: **FitzHugh-Nagumo model's solution. A** The solid blue line shows the solution $y(t)$ of equation (21) solved numerically using `odeint` and the orange crosses indicate the approximated solution by a neural network of DGM-type (see main text). We see that the neural network approximates the solution provided by `odeint` well enough. The mean absolute error (MAE) is MAE $= 0.0088$. **B** Similarly, the blue line indicates the numerical solution for $w(t)$ of Equation (21), and the orange crosses the solution obtained by the neural network. **C** Evolution of loss over learning iterations $(150,000)$. The loss initially oscillates and, after about $90,000$, iterations converges with some small oscillations. Finally, after $150,000$ iterations, the loss has been minimized, and the neural network has learned a set of parameters $\boldsymbol{\theta}$ that solves problem (22).

where $\alpha(x)$ and $g(x)$ are given functions (the function $g(x)$ usually describes a source or external forces), and λ is a parameter. And the function $K : [a,b] \times [a,b] \to \mathbb{R}$ is called the integral kernel or just kernel. An integral equation described by Equation (23), for which $\alpha(x) = 0$, and only the unknown function y appears under the integral sign is of the *first kind*, otherwise it is the *second kind*. In addition, if the known function $g(x) = 0$, we call the integral equation *homogeneous*. Finally, when the integral limits $a(x)$ and $b(x)$ are constants, then the integral equation is called the Fredholm equation; otherwise, it is called the Volterra equation. For instance, the following integral equation is a Fredholm of the first kind:

$$g(x) = \int_a^b K(x,t)y(t)dt. \tag{24}$$

And the following equation is an inhomogeneous Fredholm's equation of the second kind

$$y(x) = g(x) + \int_a^b K(x,t)y(t)dt. \tag{25}$$

We refer the reader to [53] for more information about integral equations and their solutions. For the rest of this section, we will focus on Fredholm's equation of the second kind. We will solve Equation (25) with a kernel function $K(x,t) = \sin(x)\cos(t)$, and $g(x) = \sin(x)$. Thus we have

$$y(x) = \sin(x) + \int_0^{\frac{\pi}{2}} \sin(x)\cos(t)y(t)dt, \tag{26}$$

which admits an analytic solution $y(x) = 2\sin(x)$. Let's begin with defining the loss function for the Equation (26). Based on Equation (15) and Equation (26), we get

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\Big(\hat{y} - \sin(x) - \mathcal{I}\Big)^2, \tag{27}$$

where $\hat{y}$ is the output of the neural network $\mathcal{F}(\boldsymbol{\theta}, \mathbf{t})$ and $\mathcal{I}$ is the integral term in equation (26). We will compute the integral term using Monte Carlo integration similar to [26]. Hence,

$$\mathcal{I} = \int_a^b K(x,t)y(t)dt \approx \frac{\#[a,b]}{k}\sum_{j=1}^{k} K(x_i,t_j)\hat{y}(t_j) = \frac{\#[0,\frac{\pi}{2}]}{50}\sum_{j=1}^{50}\sin(x_i)\cos(t_j)\hat{y}(t_j) \tag{28}$$

where $\#[a,b]$ is the length of $[a,b]$, which is $\frac{pi}{2}$ for equation (26), and the number of samples (random points) we draw from a uniform distribution $\mathcal{U}(a,b) = \mathcal{U}(0,\frac{\pi}{2})$ is $k = 50$. So for every point $j \in k$, we draw a random sample, we pass it to the neural network $\mathcal{F}$ to obtain the $\hat{y}$ and then we plug that in equation (28) to get the integral term $\mathcal{I}$. After we evaluate the integral term, we proceed in minimizing the loss function (27). The code listing 10 shows the Pytorch implementation of the loss function for this problem, where in lines 8-12, we evaluate the integral part described by Equation (28), and in line 15, we compute the solution $\hat{y}$ using the DGM-like neural network. Finally, in line 17, we estimate the loss function based on Equation (27).

Monte Carlo Integration It is a numerical method for estimating multidimensional integrals such as $\mathcal{I} = \int_\Omega f(\mathbf{x})d\mathbf{x}$, where $\Omega \subseteq \mathbb{R}^d$. The naive algorithm for the Monte Carlo integration consists of the following steps:

1. Sample k vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k$ from a uniform distribution on Ω, $\mathbf{x}_j \sim \mathcal{U}(\Omega)$,
2. Estimate the term $\frac{\#\Omega}{k}\sum_{j=1}^{k} f(\mathbf{x}_j)$, where $\#\Omega = \int_\Omega d\mathbf{x}$.

Due to the law of large numbers the sum eventually will converge to $\mathcal{I}$.

```
1  def fredhold_loss(net,  # Here we pass the neural network (Pytorch object) to the loss function
2                    x,    # Independent variable
3                    k=50):    # Number of random points we use in the Monte Carlo integration
4
5      # Monte Carlo integration
6      dr = np.pi / (2 * k)        # Integration step
7
8      integral = 0.0
9      for i in range(k):
10         t = np.pi/2.0 * torch.rand_like(x)
11         integral += torch.sin(x) * torch.cos(t) * net(t)
12     integral *= dr
13
14     # Evaluate the neural network on input x
15     yhat = net(x)
16
17     L = ((yhat - torch.sin(x) - integral)**2)
18     return torch.mean(L)
```

Listing 10: **Loss function for the second kind Fredholm equation.**

The neural network, in this case, is a DGM-type (see Section 3.2 and code listing 2) with one unit input and one output unit since equation (26) has one input argument, t, and the function $y(t)$ returns a scalar output. One hidden layer with 32 units is sufficient to solve the problem. Again, we choose a DGM-type since equation (26) involves trigonometric functions that would cause sudden turns. ResNet is another type of neural network used successfully in [26] to approximate solutions of integral equations in high dimensions. In this work, we will show how a DGM-type can approximate solutions of equation (26). The architecture of the neural network is

$$\text{Input}(1) \rightsquigarrow \text{FC}(32) \rightsquigarrow \text{ReLU} \rightsquigarrow \text{DGML}(32) \rightsquigarrow \text{FC}^{\text{out}}(2).$$

We use the Adam optimizer with its default parameters and learning rate $\eta = 0.0001$. We train the neural network for $3,000$ iterations. In each iteration we uniformly draw $n = 32$ (batch size) random numbers from $[0, \frac{\pi}{2}]$ representing the independent variable x and we feed the mini-batch to the neural network. Figure 12 **A** shows the analytic solution of equation (26) (blue line) and the approximated solution evaluated with the trained neural network on the interval $[0, \frac{\pi}{2}]$ with 50 grid points. We see that the approximated solution overlays with the analytic one, and the MAE=0.0134. Panel 12 **B** shows the loss during training. We observe how the loss drops until convergence after about 500 iterations.

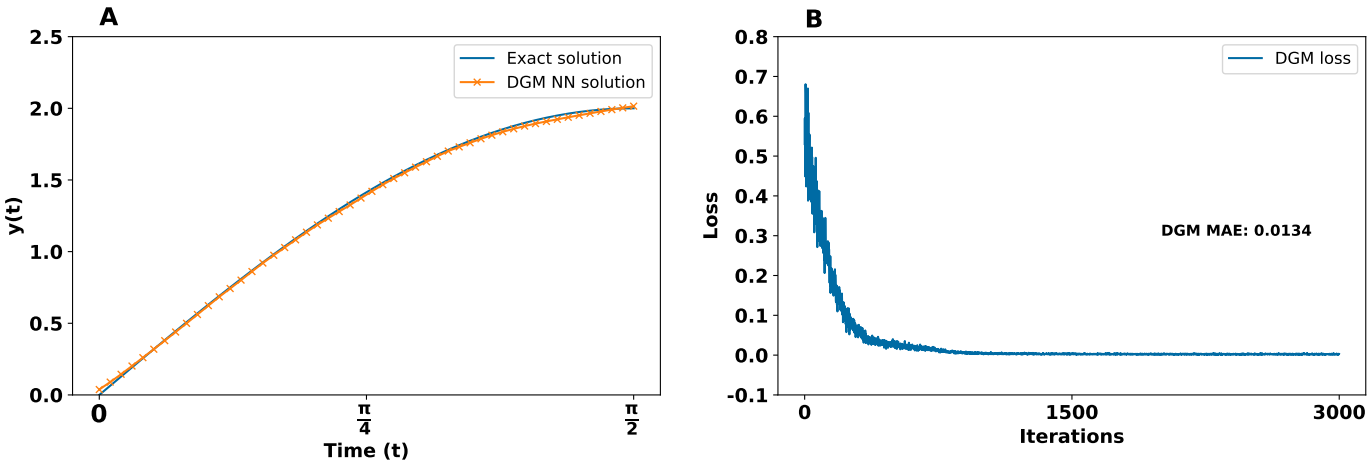

Figure 12: **Solution of a Fredholm equation of the second-kind.** **A** The blue curve is the analytic solution of equation (26), and the orange one is the approximated solution provided by a DGM-like neural network. The mean absolute error (MAE = 0.0134) indicates that the two solutions are close. **B** The loss as a function of training iterations (3,000 total iterations) shows a transient phase for about 500 iterations and then a convergence towards zero.

7 Summary & Implementations

We have presented many practical aspects of the Deep Galerkin method [59] and successfully applying it to solving partial differential equations (PDEs), such as the one-dimensional heat equation. More precisely, we have introduced basic concepts of deep learning useful to understand and implement the DG method. We provided a step-by-step practical example and code listings (snippets) on how to apply the DG method on the one-dimensional heat equation. In addition, we demonstrated how one can use Ray [48] to tune the hyperparameters of deep neural networks. Finally, we provided examples on how to use the DG method on solving ordinary differential equations (ODEs) and integral equations such as the Fredholm of the second kind.

We left uncovered other deep learning methods such as [42, 38] and focused on the DG method [59], because, in our view, if one understands and can implement the DG method, then they can easily move their focus to the rest of the methods as well. Moreover, in this work, we provided methodology and practical tips that one can apply to other methods that use deep neural networks to solve differential equations. For a complete and more concrete review about many deep learning methods for differential equations the reader is referred to [58, 6].

Last but not least, we provided many code snippets that accompanied the theory we presented in this work. In addition, the endeavouring reader can find the source code for all the examples of this work for free and under the `GPL v3.0` license on Github (`https://github.com/gdetor/differential_equations_dnn`). The source code is written in Python using Pytorch, Numpy, Scipy, Sklearn, Ray, and Matplotlib. On the same URL, readers can find more details about the software/hardware specifications and instructions on how to use the source code. All the experiments presented here ran on a desktop computer equipped with an Intel i7 13th Generation CPU, 64 GB physical memory, and an Nvidia GeForce RTX 3060 with 12 GB memory. Nevertheless, the examples are designed such that the computational requirements are minimal and can run even on a computer that is not equipped with a GPU.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[2] Ali Al-Aradi, Adolfo Correia, Danilo Naiff, Gabriel Jardim, and Yuri Saporito. Solving nonlinear and high-dimensional partial differential equations via deep learning. *arXiv preprint arXiv:1811.08782*, 2018.

[3] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[4] Christian Beck, Sebastian Becker, Patrick Cheridito, Arnulf Jentzen, and Ariel Neufeld. Deep splitting method for parabolic pdes. *SIAM Journal on Scientific Computing*, 43(5):A3135–A3154, 2021.

[5] Christian Beck, Sebastian Becker, Philipp Grohs, Nor Jaafari, and Arnulf Jentzen. Solving the kolmogorov pde by means of deep learning. *Journal of Scientific Computing*, 88:1–28, 2021.

[6] Christian Beck, Martin Hutzenthaler, Arnulf Jentzen, and Benno Kuckuck. An overview on deep learning-based approximation methods for partial differential equations. *arXiv preprint arXiv:2012.12348*, 2022.

[7] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Springer, 2024.

[8] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[9] Philippe Blanchard and Erwin Brüning. *Variational methods in mathematical physics: a unified approach*. Springer Science & Business Media, 2012.

[10] Jan Blechschmidt and Oliver G Ernst. Three ways to solve partial differential equations with neural networks—a review. *GAMM-Mitteilungen*, 44(2):e202100006, 2021.

[11] Andriy Burkov. *The hundred-page machine learning book*, volume 1. Andriy Burkov Quebec City, QC, Canada, 2019.

[12] Adam Byerly, Tatiana Kalganova, and Richard Ott. The current state of the art in deep learning for image classification: A review. In *Intelligent Computing: Proceedings of the 2022 Computing Conference, Volume 2*, pages 88–105. Springer, 2022.

[13] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[14] Pi-Yueh Chuang and Lorena A Barba. Experience report of physics-informed neural networks in fluid simulations: pitfalls and frustration. *arXiv preprint arXiv:2205.14249*, 2022.

[15] Pi-Yueh Chuang and Lorena A Barba. Predictive limitations of physics-informed neural networks in vortex shedding. *arXiv preprint arXiv:2306.00230*, 2023.

[16] Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

[17] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[18] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[19] Stanley J Farlow. *Partial differential equations for scientists and engineers*. Courier Corporation, 1993.

[20] Richard FitzHugh. Impulses and physiological states in theoretical models of nerve membrane. *Biophysical journal*, 1(6):445–466, 1961.

[21] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR, 2023.

[22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* " O'Reilly Media, Inc.", 2022.

[23] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[24] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[26] Yu Guan, Tingting Fang, Diankun Zhang, and Congming Jin. Solving fredholm integral equations using deep learning. *International Journal of Applied and Computational Mathematics*, 8(2):87, 2022.

[27] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[28] Jiequn Han, Arnulf Jentzen, et al. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in mathematics and statistics*, 5(4):349–380, 2017.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[32] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[33] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[34] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[35] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[36] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[37] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[38] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[39] Yuehaw Khoo and Lexing Ying. Switchnet: a neural network model for forward and inverse scattering problems. *SIAM Journal on Scientific Computing*, 41(5):A3182–A3201, 2019.

[40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] Siddharth Krishna Kumar. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863*, 2017.

[42] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[43] Hans Petter Langtangen and Kent-Andre Mardal. *Introduction to numerical methods for variational problems*, volume 21. Springer Nature, 2019.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[45] Hyuk Lee and In Seok Kang. Neural algorithm for solving differential equations. *Journal of Computational Physics*, 91(1):110–131, 1990.

[46] DC Liu and J Nocedal. On the limited memory method for large scale optimization: Mathematical programming b. 1989.

[47] Andrew J Meade Jr and Alvaro A Fernandez. Solution of nonlinear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling*, 20(9):19–44, 1994.

[48] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[49] Keith W Morton and David Francis Mayers. *Numerical solution of partial differential equations: an introduction*. Cambridge university press, 2005.

[50] Jinichi Nagumo, Suguru Arimoto, and Shuji Yoshizawa. An active pulse transmission line simulating nerve axon. *Proceedings of the IRE*, 50(10):2061–2070, 1962.

[51] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[53] Polyanin Polyanin and Alexander V Manzhirov. *Handbook of integral equations*. Chapman and Hall/CRC, 2008.

[54] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[55] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[56] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[57] Marc Sabate Vidales, David Šiška, and Lukasz Szpruch. Unbiased deep solvers for linear parametric pdes. *Applied Mathematical Finance*, 28(4):299–329, 2021.

[58] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[59] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

[60] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[61] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[62] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.

[63] Walter A Strauss. *Partial differential equations: An introduction*. John Wiley & Sons, 2007.

[64] John C Strikwerda. *Finite difference schemes and partial differential equations*. SIAM, 2004.

[65] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. corr, abs/2302.13971, 2023. doi: 10.48550. *arXiv preprint arXiv.2302.13971*, 2023.

[66] Lixin Wang and Jerry M Mendel. Structured trainable networks for matrix algebra. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 125–132. IEEE, 1990.