Univerza *v Ljubljani*
Fakulteta *za matematiko in fiziko*

Physics-Informed Neural Networks

Author: Simon Perovnik
Mentor: doc. dr. Simon Čopar

Seminar I - year I, II. cycle

Ljubljana, 2023

Abstract

Physics-informed neural networks are machine learning algorithms that are able to provide a numerical solutions to laws of physics described by differential equations and corresponding boundary conditions. Their key feature is the loss function that is composed in such a way, that it penalises deviations from what we describe with differential equations. It has proved to be a reliable and effective method to solve common differential equations such as the heat equation [1], Schrödinger equation [2] and others. We start with a less technical description of neural networks and then proceed to introduce the idea of physics-informed neural networks and provide some examples of its use. We conclude with a brief discussion about the advantages and disadvantages of these methods.

Contents

1 Introduction

Most fundamental laws of physics can be formulated as (partial) differential equations. We have developed several techniques how to solve them, majority of them being some sort of analytical approach or a variety of finite differences methods. In this seminar we would like to describe another method that has some roots in the late 90s, but was just recently properly developed. This method is called physics-informed neural networks and is a special type of neural network that is able to produce solutions that are in agreement with physical laws. To understand how it works, we must first understand the fundamentals of such networks.

2 What is a Neural Network?

As the name suggests, the idea behind neural networks (NN) is inspired by the way our nervous system (to be more precise, nerve cells or neurons) functions. To gain insight into what are the similarities between a biological system and a machine learning algorithm we must first be able to answer this two questions - what are neurons and how do they connect together to form a network?

A neuron is a fundamental unit of nervous system which is excitable by electric impulses and can also produce electric signals of its own to communicate with other neurons via connections that are called synapses [3]. A neuron is at all times receiving electrical impulses from (input) neurons to which it is connected. Whenever the sum of this signals is high enough the neuron sends an impulse to all the (output) neurons it is connected to.

Inspired by this biological structure we can construct an artificial neural network which is composed of several neurons that are interconnected with each other. We arrange them in layers so that the input and output neurons for a given artificial nerve cell consist of all the neurons in the layer before and after, respectively. We call such a network feedforward (the electrical signal travels only in forward direction) and dense (every neuron of a given layer is connected to every neuron of its preceding layer). We define first layer as the input one, the last as an output one and all layers in between as hidden layers.

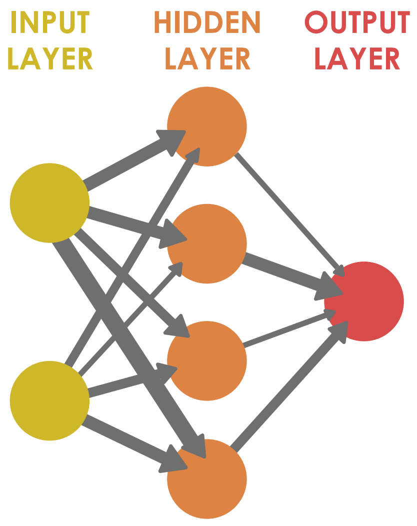

Figure 1: Schematic representation of a neural network with one hidden layer. Adapted from [4].

We can then propose a few modifications to this system in order to transform it into a powerful computational tool. The first one is the activation of neurons. In a biological system the output signal is binary (if the sum of input signals is above a certain threshold, the neuron sends a pulse or doesn't fire at all if the sum isn't sufficiently high) but in an artificial neuron we can model different functions as the response functions of a neuron. In particular, we can make use of universal approximation theorem which states that we can model any function with a large enough composition of non-linear functions and use a non-linear activation function. A few commonly used functions are shown in figure (2). Along with non-linearity it is also beneficial that we use functions that are continuous and differentiable, the former being important for smoothness of output, so that similar inputs produce similar results, and the latter so they can be used along with backpropagation algorithm, which is a vital element of NN and will be described later on.

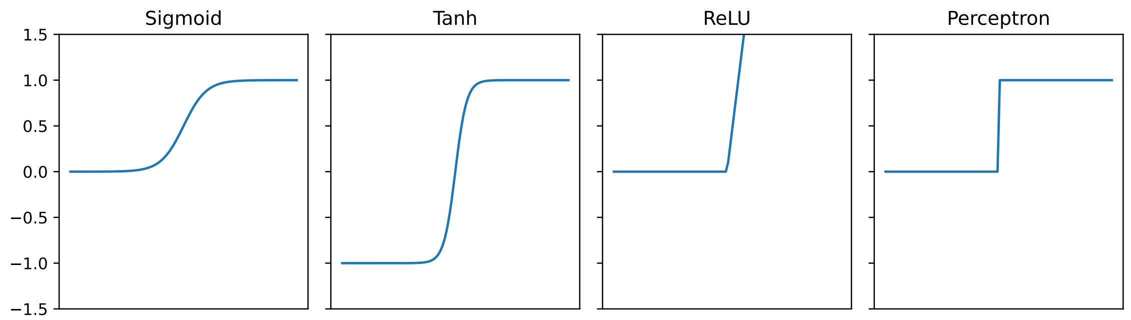

Figure 2: Commonly used activation functions in comparison to the one in biological neurons, often labeled perceptron.

Next alteration can be made by attributing each connection between neurons a weight (a multiplication factor of the signal) which corresponds to the importance of that linkage and a each neuron a bias, a constant input that is related to how often the neuron is activated. With this changes we can describe i-th neuron's response as

$$y_i = \sigma \left(\sum_{j=1}^{N} x_j w_j + b_i \right) \tag{1}$$

where σ is a non-linear activation function, N is the number of neurons in preceding layer and x_j along with w_j are the signal and the corresponding weight for a certain connection. We can compact the notation by writing out outputs from all neurons in i-th layer and combining them into a vector $\mathbf{y}^{(i)}$ as well as constructing a matrix $\mathbf{W}$ which carries all the information about network's weights.

$$\mathbf{y}^{(i)} = \sigma \left(\mathbf{W}\mathbf{y}^{(i-1)} + \mathbf{b}^{(i)} \right) \tag{2}$$

It is important to note that even though in the biological system we talk about electric pulses that transmit information between neurons, this concept can be generalized to form a purely mathematical construct, where each neuron essentially represents a number (called activation of a neuron) that is determined with respect to weights, bias and activations of neurons in the previous layer.

It is evident that NN is essentially a layered system of equations that takes some inputs and produces an output with respect to the weights and biases. Another way to look at it is to picture it as a universal function (making use of the non-linear activation functions) that can be used to describe any desired function by appropriately setting the weights and biases. How to set weights and biases so NN models an arbitrary function is the topic of next segment.

2.1 How Does a Neural Network Learn?

In the previous chapter our description of what functions we can model with NN may have seemed limited to functions that we can describe mathematically, where in fact the word function should be perceived in a more general way, which would be that it is simply a construct that takes in some input values and returns some output values. In the common example of classification of handwritten digits (depicted in figure (3)) the input of such a function is a 16×16 grayscale image and the output is a list of values that correspond to how certain the network is, that the portrayed image is in fact the corresponding digit – for example, the first value in the list would represent the likelihood that digit 0 is written in the input image.

Figure 3: Some examples of colorized greyscale images of handwritten digits from MNIST set.

The process of calculating optimal weights and biases for a given function we want to model is conventionally called "training", because it is similar to the process of training an individual in a skill or task. In the same way that a person's abilities can be improved through repetition and feedback, the weights of a neural network are improved through exposure to many examples and adjusting based on the error in the network's predictions. In order to train NN to model the appropriate function we must be able to provide lots of labeled data (data for which we know what the output should be) about what we want the function output to be according to certain input. The key part of this process is the feedback that the NN then gains by comparing its output to the target's one. With this comparison we can construct a loss function that is a measure of how inaccurate was the output with respect to the target one. Commonly used loss function is mean-square error function that we can write out as

$$\mathcal{L}(\mathbf{W}) = \sum_{i=0}^{N_L} (y_i - \hat{y}_i)^2, \tag{3}$$

where we emphasized that the loss function is a function of weights ($\mathbf{W}$) of the NN and used y_i and $\hat{y}_i$ to express the i-th output component of NN and the target value, respectively. N_L denotes the number of neurons in the output (L-th) layer. As we will see later when discussing physics-informed NN, the loss function will serve as a feature through which we will assure that the output given by NN complies with (certain) laws of physics.

As implied, the goal of training is to find the set of weights and biases that results in the lowest error, or best fit, between the network's predictions and the true target values. This process allows the network to generalize to new, unseen examples and improve its accuracy. This learning capability is achieved through the use of algorithms such as backpropagation and stochastic gradient descent.

The basic idea behind backpropagation is to propagate the error from the network's output layer to the input layer, computing the gradient of the loss function with respect to each weight in the network. The gradients are then used to update the weights in a way that reduces the loss function, resulting in improved predictions by the network.

The backpropagation algorithm works by first using the labeled (also called training) examples to make a forward pass through the network, producing an output and a corresponding loss. The error is then backpropagated through the network, using the chain rule of differentiation to compute the gradients of the loss with respect to each weight. The gradients are then used to update the weights using an optimization algorithm, such as stochastic gradient descent which is essentially a tool to search for a global minimum by analysing a change of what parameters (weights and biases) reduces the loss function the most. Training of NN is therefore in its essence a numerical minimisation algorithm. In every training instance the weights get updated with respect to the magnitude of loss function and gradient

$$\Delta\mathbf{W} \propto -\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W}). \tag{4}$$

The gradients of lost function in relation to internal parameters will prove to be of significant importance in constructing physics-informed NN. The main reason being the fact, that many physics laws are described with some derivatives which are easily computed and accessible in the context of NN.

3 Physics-Informed NN

As we learned so far NNs are machine learning algorithms that receive an input $\mathbf{x}$ and generate $\mathbf{y}$ according to currently set weights. This is schematically portrayed in picture (4). By comparing $\mathbf{y}$ and $\hat{\mathbf{y}}$ (target output) we can propagate this error backwards and find which corrections to the weights will reduce the loss function the most. We use that information to calculate a new set of weights. If multiple inputs share some common features or patterns we should be able to produce less incorrect outputs next time the NN is exposed to a similar input.

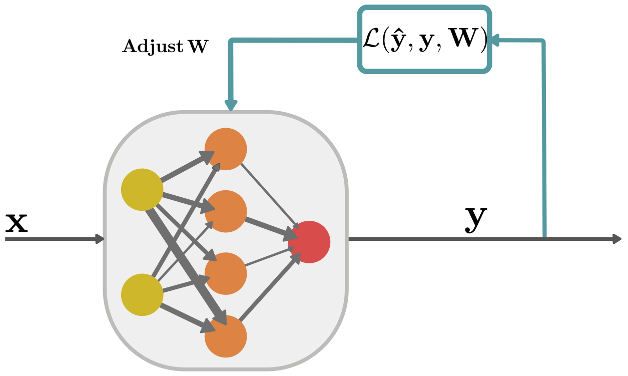

Figure 4: Graphic corresponding to how a NN adjusts its weights W according to the loss function $\mathcal{L}$.

While we could use this construction to predict, for example, how a certain physical system might evolve in time, we would have no guarantee that NN will in fact pick up on laws of physics that our data obeys. This is the setup where one can make use of physics-informed neural networks (PINNs).

3.1 General Case

Laws of physics are often described in form of differential equations (DEs). A simple example would be a second order DE with some boundary conditions

$$\begin{gathered} A\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + B\frac{\mathrm{d}y}{\mathrm{d}x} + Cy = 0, \\ y(0) = y_0, \qquad y(1) = y_1. \end{gathered} \tag{5}$$

In this simple case the solution to such an equation can be found analytically, but we can also try to solve it using PINN. In this instance we want to train the NN in such a way, that it will resemble a function that will numerically solve equation (5). In other words, we wish that our network would learn the laws of physics that are described in the before-mentioned equation and would provide a correct prediction of y to any given x. This can be done very cleverly by modifying the loss function. We can add additional terms that will penalise the output if the equation (5) will not equal to zero. This is only possible due to automatic differentiation that is commonly already build into NN deep learning models. This method is described with great detail in [5]. In its essence, automatic differentiation is closely related to backpropagation and provides a way of computing derivatives with respect to input coordinates (temporal or spatial). It also provides a crucial advantage over other similar methods which make use of numerical differentiation and are more prone to numerical errors and are more time consuming [1].

Analogously we can also add a term to the loss function that assure that the boundary (or initial) conditions are met, or at least, that the residual is close to zero. All those terms are combined together in equation

$$\mathcal{L} = \mathcal{L}_d + \mathcal{L}_r + \mathcal{L}_b + \mathcal{L}_0 \tag{6}$$

where we denoted the residuals of the DE itself with $\mathcal{L}_r$, residuals of boundary and initial conditions with $\mathcal{L}_b$ an $\mathcal{L}_0$, respectively, and the inconsistency with target data with $\mathcal{L}_d$. To physicists, loss function should be somewhat similar to Lagrange function which is defined in a way that it offers a way of finding local minimums. With that in mind, we could see PINNs as a numerical variation of solving the variational problem.

It is important to highlight the fact, that NN constructed in this manner does not inherently require any data points on which it is trained, so the $\mathcal{L}_d$ could be omitted. This means that it is not necessary to obtain a certain part of solution via other numerical methods in order to train the network and then predict the full solution. That being said, this factor could represent measurements that were acquired experimentally.

Even though there is no theoretical guarantee that this method will in fact converge the empirical evidence has shown that it can successfully tackle a variety of problems involving partial differential equations [1] [2]. In the next chapter we will present some examples of how this method performs with respect to analytical solutions.

4 Solving Burger's Equation Using PINN

Burger's equation is a simplification of well-known Navier-Stokes equation where the pressure term is dropped. It describes motion of a fluid by modeling its speed ($u(x,t)$) with two general effects – motion governed by conservation laws (left side of equation (7)) and the diffusion associated with heat equation (right side of equation (7)). It can be written out in one (space) dimension and a Direchlet boundary condition as

$$\begin{gathered} u_t + uu_x = (0.01/\pi)u_{xx}, \quad x \in [-1,1], \;\; t \in [0,1], \\ u(0,x) = -\sin(\pi x), \\ u(t,-1) = u(t,1) = 0. \end{gathered} \tag{7}$$

Finding the numerical solution for $u(x,t)$ using PINN was done in [2], where they constructed a loss function so that it minimised the residuals of both the equation and the boundary conditions. Results are shown in (5). As indicated with black cross marks on the plot, the training data consisted only of points corresponding to initial and boundary conditions. The network had 7 hidden layers with 20 neurons in each layer and had hyperbolic tangent as an activation function. It is remarkable that using as few as 100 data points (which are trivial to provide) yields great agreement with the exact solution, even though non-linearity gets especially prominent around $t = 0.4$. To arrive to this solution with classical approaches one would have to go through laborious spatio-temporal discretization of equation (7), which is not needed when dealing with PINNs.

We should highlight the fact that in the given example, we took the parameters of equation (7) for ground truth and

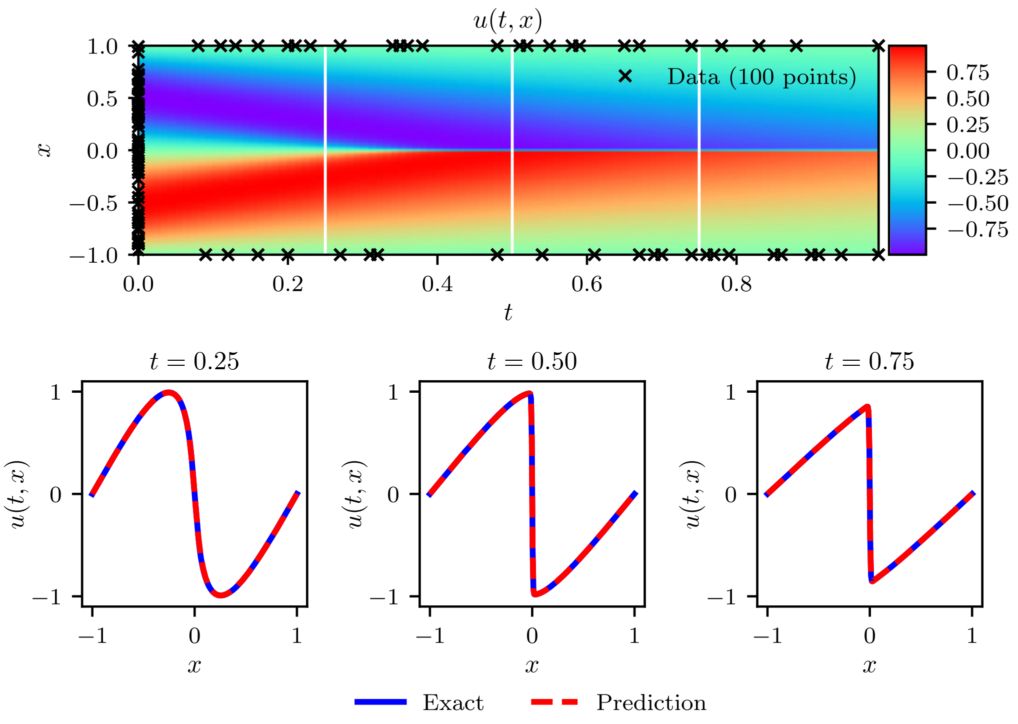

Figure 5: Solution of Burger's equation (top) and three snapshots that correspond to the white vertical lines and show the similarities between numerical and exact solution. The relative squared difference between prediction and exact solution was $6.7 \cdot 10^{-4}$. Figure is from [2].

went on to compute the solution $u(x,t)$, but as was shown in [2] it is possible to turn the paradigm around and try to model the correct parameters of DE that best fit your data. This is also called the data-driven discovery of DE.

A potential downside to this method is the number of points at which the network needs to be evaluated in order to give a representation in the entire spatio-temporal domain. Even though this very property could serve as an advantage since solutions at every coordinate are available and do not demand (computationally slow) interpolation between already generated solutions, it could also pose a problem, especially when dealing with more spatial dimensions, where the number of points needed to enforce a physics informed constrain will increase exponentially. This problem can be addressed and circumvented by making use of classical Runge–Kutta time-stepping schemes in a way that we essentially train the PINN to calculate the parameters of Runge-Kutta methods.

5 Solving Allen–Cahn Equation Using PINN

Allen-Cahn equation describes the process of phase separation in multi-component alloy systems, but for the purpose of this seminar we will not focus on the physical interpretation, but rather on mathematical description, which is given by

$$\begin{gathered} u_t - 0.0001u_{xx} + 5u^3 - 5u = 0, \quad x \in [-1,1], \; t \in [0,1], \\ u(0,x) = x^2 \cos(\pi x), \\ u(t,-1) = u(t,1), \quad u_x(t,-1) = u_x(t,1), \end{gathered} \tag{8}$$

where we assumed periodic boundary conditions. In the case represented in figure (6) it was solved using PINN with 4 hidden layers that each consisted of 200 neurons. The solution was acquired a bit differently than in the example before. The network was trained on the data that corresponded to one time-instance of full solution (at $t = 0.1$) with numerical spectral method. To calculate the solution at a later time with conventional methods one would have to make many in-between time steps due to stability constraints, but using a PINN it is possible to predict the solution in a single time step. This is possible due to clever construction of NN that takes in coordinates x and outputs q number of function values corresponding to q number of stages used for Runga-Kuta integration, as well as a solution at final time $t = 0.9$. Stages represent function evaluation in between initial time $t = 0.1$ and the final one. In this example q was set to 100 so that the desired accuracy was reached. Snapshot at final time $t = 0.9$, portrayed in (6), is the direct output of NN. This property of being able to integrate over a large time gap without significantly sacrificing accuracy is unique to this implementation of PINN.

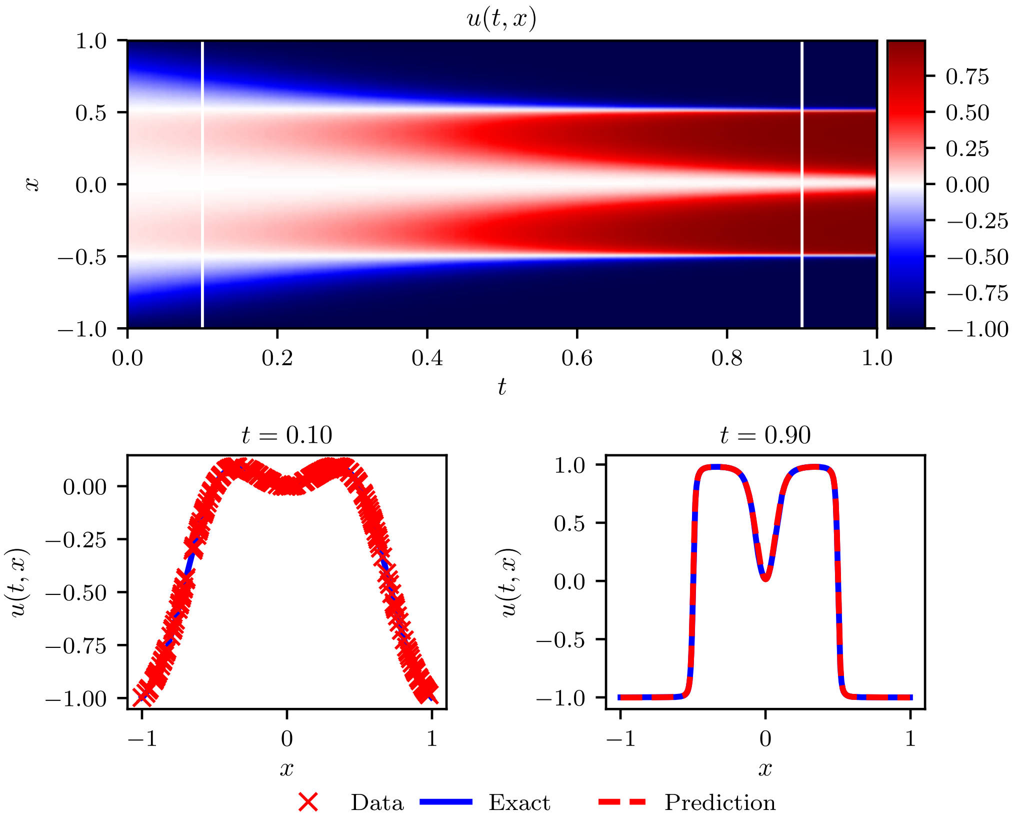

Figure 6: Solution to Allen-Cahn equation on the top and a snapshot of the training data along side with a predicted snapshot at $t = 0.9$. Figure is from [2].

6 Limitations and Advantages of PINNs

In this last section we would like to summarize the upsides and downsides of using PINNs and put those in the context of alternative finite element methods (FEM). Most of the limitations stem from intrinsic characteristics of NN:

- Since NN are generally not very interpretative, it is hard to ensure that the method has indeed found a general solution, when the solution is not yet available.
- There is no obvious way to quantify the uncertainty of results.
- As with most NN, there is no theoretical approach that would be able to predict the optimal network architecture, so this has to be done empirically using intuition. This might change with emerging meta-learning techniques that can

be used to automate this search.

- Most studies [2], [6], [7], [8] agree solving DEs using PINNs is currently slower, or at most comparable to using FEM techniques. In most cases they also offer less accurate results.

All that being said, PINNs have some clear advantages over other methods, the main one being the fact, that they are mesh-free. Since there is no integral discretisation, acquiring solutions in any domain point is therefore merely a question of evaluating NN which is typically much faster than numerical interpolation, which would be used when working with FEM methods. Similarly, they may provide a useful tool to use when dealing with complex geometries where constructing proper grid is not trivial.

Another useful feature that stems from intrinsic property of NN is the simplicity to solve inverse problems, once the general problem solution has been found, as the inverse model can be described relatively simply once the network has been trained.

7 Conclusion

In recent years PINNs have emerged as a viable tool to solve various (partial) differential equations. Despite impressive results, these methods should not be viewed as a complete replacement for already established numerical methods like FEM, but rather a complementary tool to use with the classical solvers. Their main value lies within the fact, that they are grid-less so the solution at any given point in the domain is easily accessed. They also offer a unique numerical method that can combine experimental data with the theoretical governing equations.

References

[1] Shengze Cai, Zhicheng Wang, Sifan Wang, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks for heat transfer problems. *Journal of Heat Transfer*, 143(6), 2021.

[2] 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.

[3] Kevin Gurney. *An introduction to neural networks.* CRC press, 2018.

[4] Wikipedia: Neural network (https://en.wikipedia.org/wiki/neural_network), last accessed on 5.2. 2023.

[5] 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.

[6] 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.

[7] Tamara G Grossmann, Urszula Julia Komorowska, Jonas Latz, and Carola-Bibiane Schönlieb. Can physics-informed neural networks beat the finite element method? *arXiv preprint arXiv:2302.04107*, 2023.

[8] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. Deepxde: A deep learning library for solving differential equations. *SIAM review*, 63(1):208–228, 2021.