Chapter 5
Physics-Informed Neural Networks

Generating an accurate surrogate model of a complex physical system usually requires a large amount of solution data about the problem at hand. However, data acquisition from experiments or simulations is often infeasible or too costly. With this in mind, Raissi et al. proposed an approach, that augments surrogate models with existing knowledge about the underlying physics of a problem [RPK19]. In many cases, the governing equations or empirically determined rules defining the problem are known a priori. For instance, an incompressible flow has to satisfy the law of conservation of mass. By incorporating this information, the solution space is drastically reduced and, as a result, less training data are needed to learn the solution.

The idea of adding prior knowledge to a machine learning algorithm is not completely new. As mentioned in Chap. 4, the studies by Raissi et al. were inspired by papers of Psichogios and Ungar [PU92], Lagaris et al. [LLF98], and more recent developments by Kondor [Kon18], Hirn et al. [HMP17] and Mallat [Mal16]. Nevertheless, the solutions proposed by Raissi et al. extended existing concepts and introduced fundamentally new approaches like a discrete time-stepping scheme, that efficiently exploits the predictive power of neural networks. Furthermore, they demonstrated their method on a variety of examples that are of interest in a physics and engineering context [RPK19]. The accompanying code was written in Python and utilizes the popular GPU-accelerated machine learning framework Tensorflow. Additionally, the code is publicly available on GitHub allowing others to explore physics-informed neural networks and contribute to their development [Rai20].

The paper "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations" [RPK19] by Raissi et al. was referenced in different reviews [BNK20, FDC20]

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-030-76587-3_5) contains supplementary material, which is available to authorized users.

S. Kollmannsberger et al., *Deep Learning in Computational Mechanics*,
Studies in Computational Intelligence 977,
https://doi.org/10.1007/978-3-030-76587-3_5

and marks a starting point for further research on physics-enriched surrogate models as outlined in Sect. 5.4.

5.1 Overview

The general idea of physics-informed neural networks (PINNs) is to solve problems where only limited data are available, e.g. noisy measurements from an experiment. To compensate for the data scarcity, the algorithm is enriched with physical laws governing the problem at hand. Typically, those laws are described by parameterized non-linear partial differential equations of the form

$$\frac{\partial u}{\partial t} + \mathcal{N}[u; \lambda] = 0, \ x \in \Omega, \ t \in \mathcal{T}. \tag{5.1}$$

Here, the latent solution $u(t, x)$ depends on time $t \in [0, T]$ and a spatial variable $x \in \Omega$, where Ω refers to a space in $\mathbb{R}^D$. Further, $\mathcal{N}[u; \lambda]$ represents a non-linear differential operator with coefficients λ. This description covers a wide variety of problems ranging from advection–diffusion reaction of chemical or biological systems to the governing equations of continuum mechanics.

The following sections distinguish between two different use-cases for physics-informed neural networks, namely data-driven inference and data-driven identification of partial differential equations. The first case addresses forward problems where the coefficients λ are known and the hidden solution $u(t, x)$ is computed based on initial and boundary data. The second case solves an inverse problem. Given scattered data of the solution $u(t, x)$, the goal is to identify the coefficients λ of the partial differential equation.

For stationary or transient problems, the physics-informed neural network computes the underlying partial differential equation. Therefore, a classic feed-forward neural network approximates the solution $u(t, x)$ and the physics-enriched part evaluates the corresponding partial derivatives forming the left-hand side of Eq. (5.1). Both the network approximating $u(t, x)$ as well as the whole physics-informed neural network depend on the same set of parameters. Those shared weights and biases are trained by minimizing a cost function which consists of multiple mean squared error losses. The cost function in its general form can be written as

$$C = MSE_u + MSE_f. \tag{5.2}$$

The first term, here denoted as MSE_u, computes the error of the approximation $u(t, x)$ at known data points. In case of a forward problem, this term entails data representing the boundary and initial conditions while for inverse problems the solution at different points inside the domain is provided. The other term in the cost function, MSE_f, enforces the partial differential equation on a large set of randomly

chosen collocation points inside the domain. In particular, this term penalizes the error between the approximated left-hand side and the known right-hand side of the partial differential equation at every collocation point.

Generally, there are two ways to enforce boundary conditions of a forward problem. In case of a weak enforcement MSE_u, Eq. (5.2) entails multiple terms that enforce the solution at the boundary. Alternatively, regular and constant boundary conditions can be enforced in a strong sense. To this end, the output of the network is adapted so that the predicted solution automatically satisfies the boundary condition for any given input. This simplifies the cost function since the boundary terms can be neglected. As with other deep learning algorithms, the optimal network parameters are found by minimizing the custom cost function (cf. Eq. (5.2)). In the context of physics-informed neural networks gradient-based optimization algorithms like Adam or L-BFGS are often used [RPK19, Sam+19].

Even though the proposed approach is not guaranteed to converge to a global minimum, and thus, an accurate solution $u(x)$, Raissi et al. [Rai20, RPK19] showed empirically that their method achieves accurate results for different problems and architectures. Assuming that the partial differential equation has a unique solution and is well-posed, the physics-informed neural network is able to predict the underlying solution. Further requirements are a network architecture with adequate representational power and a sufficient number of collocation points N_f. In addition to the collocation-based method, Raissi et al. proposed a discrete-time model for both forward and inverse problems. This approach makes use of a discrete time-stepping scheme to predict the solution $u(x)$ at time t^{n+1} from limited snapshot data at time t^n. An advantage of the discrete method is that it omits the use of collocation points since only a solution for the initial state at t^n and boundary conditions must be provided.

The following sections explain the different types of physics-informed neural networks on forward (cf. Sect. 5.2) and backward problems (cf. Sect. 5.3). After beginning with an introductory example of a static bar, each section illustrates the continuous and discrete models with a non-linear heat problem. The implementation of the presented examples is based on the code accompanying the paper by Raissi et al. However, the code has been adapted and now utilizes the deep learning library PyTorch instead of TensorFlow.

5.2 Data-Driven Inference

In general, the problem of inference can be phrased as: find the hidden solution $u(t, x)$ for given coefficients λ. Since all values of λ are known, Eq. (5.1) simplifies to

$$\frac{\partial u}{\partial t} + \mathcal{N}[u] = 0, \quad x \in \Omega, \quad t \in \mathcal{T}. \tag{5.3}$$

5.2.1 Static Model

A very simple application of the physics-informed neural network is the one-dimensional linear elastic static bar, which is governed by an ordinary differential equation and the corresponding boundary conditions. A solution in equilibrium state is sought so that $\frac{\partial u}{\partial t} = 0$ from Eq. (5.3). Therefore, the differential equation only applies to the spatial domain $\Omega \in \mathbb{R}^1$. The differential equation operator is then given as $\mathcal{N}[u] = \frac{d}{dx}(EA\frac{du}{dx})$. Additionally, there is an inhomogeneous term p, which defines a distributed load. The system can thereby be expressed as

$$\frac{d}{dx}\left(EA\frac{du}{dx}\right) + p = 0 \qquad \text{on } \Omega, \tag{5.4}$$

$$EA\frac{du}{dx} = F \qquad \text{on } \Gamma_N, \tag{5.5}$$

$$u = g \qquad \text{on } \Gamma_D. \tag{5.6}$$

The Neumann and the Dirichlet boundaries are represented by Γ_N and Γ_D, respectively, where F denotes a concentrated load on Γ_N, and g prescribes a displacement on Γ_D. The Young's modulus $E(x)$ and the cross-sectional area $A(x)$ may vary with respect to x.

In the physics-informed neural network, a deep neural network is used to approximate the unknown solution $u(x)$. So far, this approach does not differ from the surrogates introduced in Sect. 4.3. However, these models have to rely on a tremendous amount of labeled training data closely related to the solution being approximated. Instead of having labeled training data in the whole domain, the solution for the example at hand is only known at the boundaries, for example, at $x = 0$ and $x = 1$. Additionally, Eq. (5.4) has to be satisfied for every point inside the domain. In order to verify that every input fulfills this condition, the network is extended to compute the left-hand-side of Eq. (5.4)

$$f := \frac{d}{dx}\left(EA\frac{du}{dx}\right) + p. \tag{5.7}$$

Since a neural network is fully differentiable, it is not only possible to compute the derivatives with respect to the parameters necessary for training. Likewise, the automatic differentiation capabilities of libraries such as PyTorch or Tensorflow allow the fast computation of derivatives with respect to the input variable x (cf. Sect. 3.9).

All together, this extended network architecture can be interpreted as a physics-informed neural network with outputs $u_{NN}(x)$ and $f_{NN}(x)$. As depicted in Fig. 5.1, the first part simply approximates the solution $u(x)$ with a feed-forward fully connected neural network. The second part represents the computation of Eq. (5.7), $f(x)$ using the corresponding derivatives d_x and d_{xx} with respect to u, and the identity I of u. It should be noted that both networks $u_{NN}(x)$ and $f_{NN}(x)$ depend on the same set of parameters, namely the weights and biases of the first network. The dashed

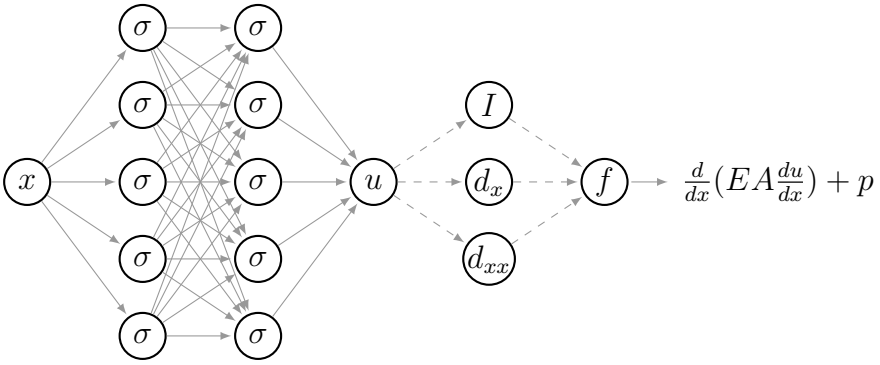


Fig. 5.1 Conceptual physics-informed neural network for the static bar equation. The left part shows the feed-forward neural network and the right part represents the physics-informed neural network. The dashed lines denote non-trainable weights

connections in Fig. 5.1 are simply introduced to visualize the composition of f and do not represent trainable parameters of the network.

To give a better idea of how a physics-informed neural network is implemented, a specific example is now introduced. Let us define a domain $\Omega = [0, 1]$ with $\Gamma_D = \{x \mid x = 0,\ x = 1\}$ and $\Gamma_N = \emptyset$. The material parameters are set to $EA = 1$. A solution for the displacement is chosen as

$$u(x) = \sin(2\pi x), \tag{5.8}$$

which, after insertion into the differential equation (5.4), results in the distributed load

$$p(x) = 4\pi^2 \sin(2\pi x). \tag{5.9}$$

Furthermore, the following Dirichlet boundary conditions apply

$$u(0) = u(1) = 0. \tag{5.10}$$

With this information it is now possible to define a cost function. In case of the static physics-informed neural network this custom cost function is assembled from two mean squared error losses Eq. (5.2)

$$C = MSE_b + MSE_f, \tag{5.11}$$

where

$$MSE_b = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(u_{NN}\left(x_b^i\right) - u_b^i \right)^2, \tag{5.12}$$

and

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(f_{NN}\left(x_f^i\right)\right)^2 . \tag{5.13}$$

The solution on the boundary is represented by N_b labeled data points $\{x_b^i, u_b^i\}_{i=1}^{N_b}$. The boundary loss MSE_b is computed by comparing the approximation u_{NN} with the labels u_b of the training data. The error of the approximation at known data points is given by $MSE_u = MSE_b$, as only the boundary data points are known. Note also, that here the boundary loss MSE_b solely includes Dirichlet boundary conditions, as only these were defined in the example. However, the application of Neumann boundary conditions works similarly and is discussed further in Sect. 5.2.2.

In order to enforce Eq. (5.3) in the whole spatial domain, a set of N_f collocation points $\{x_f^i\}_{i=1}^{N_f}$ is generated using a uniform distribution throughout the one-dimensional domain. The corresponding loss MSE_f from Eq. (5.13) is computed as the mean squared error of f_{NN} at all collocation points. Now, the physics-informed neural network can be trained by minimizing the cost function in Eq. (5.11). Instead of a classical stochastic gradient descent procedure, the L-BFGS optimizer is employed [LN89].

The physics-informed network is implemented in PyTorch. At first, a neural network is created to predict the displacement $u(x)$. For simplicity's sake, a model consisting of a single hidden layer with the dimension `hidden_dim` is defined. The input and output dimensions are correspondingly given as `input_dim` and `output_dim`. The linear transformations from one layer to the next are defined by `torch.nn.Linear` and the non-linear activation function is chosen as the hyperbolic tangent, `torch.nn.Tanh`. There is no activation function in the output layer meaning that the range of the outputs is not limited.

```
def buildModel(input_dim, hidden_dim, output_dim):
    model = torch.nn.Sequential(torch.nn.Linear(input_dim, hidden_dim),
    torch.nn.Tanh(),
    torch.nn.Linear(hidden_dim, output_dim))
    return model
```

With the neural network, the first part of the PINN illustrated in Fig. 5.1 is defined. A displacement prediction $\hat{u}$ at $x = 0.5$ can be made with the following code.

```
model = buildModel(1, 10, 1)
u_pred = model(torch.tensor([0.5]))
```

The second part of the network requires the computation of the derivatives with respect to the input x. This is achieved with PyTorch's built-in automatic differentiation, which creates and retains a computational graph to store all information necessary for calculating the gradients.

```
def get_derivative(y, x):
    dydx = grad(y, x, torch.ones(x.size()[0], 1),
        create_graph=True,
        retain_graph=True)[0]
    return dydx
```

Finally, $f(x)$ from Eq. (5.7) is computed with the following code.

```
def f(model, x, EA, p):
    u = model(x)
    u_x = get_derivative(u, x)
    EAu_xx = get_derivative(EA(x) * u_x, x)
    f = EAu_xx + p(x)
    return f
```

Now, to calculate the loss function of the differential equation MSE_f, the example values are inserted into the model.

```
model = buildModel(1, 10, 1)
x = torch.linspace(0, 1, 10, requires_grad=True).view(-1, 1)
EA = lambda x: 1 + 0 * x
p = lambda x: 4 * math.pi**2 * torch.sin(2 * math.pi * x)

f = f(model, x, EA, p)
MSE_f = torch.sum(f**2)
```

In combination with the loss function of the boundary conditions MSE_b, allowing to compute the cost function C from Eq. (5.11). The implementation for this example is shown in the code snippet below. Note that only the Dirichlet boundary conditions are imposed here. Neumann boundary conditions can be imposed in a similar way with the small addition of computing the derivatives with the `get_derivative` function. This is described in more detail in the upcoming Sect. 5.2.2.

```
model = buildModel(1, 10, 1)
u0 = 0
u1 = 0

u0_pred = model(torch.tensor([0.]))
u1_pred = model(torch.tensor([1.]))
MSE_b = (u0_pred - u0)**2 + (u1_pred - u1)**2
```

It is now possible to compute and minimize the cost function C via a stochastic gradient approach or more sophisticated methods such as the L-BFGS optimizer. When the cost function is sufficiently small, an accurate prediction of the displacement

$u(x)$ is to be expected. In principle, this is how the physics informed neural network works. The entire training procedure is summarized in Algorithm 4.

Algorithm 4 Training a physics-informed neural network for the static solution of the problem described in Eq. (5.4).

Require: training data for boundary condition $\{x_b^i, u_b^i\}_{i=1}^{N_b}$

generate N_f collocation points with a uniform distribution $\{x_f^i\}_{i=1}^{N_f}$
define network architecture (input, output, hidden layers, hidden neurons)
initialize network parameters $\boldsymbol{\Theta}$: weights $\{\boldsymbol{W}^l\}_{l=1}^{L}$ and biases $\{\boldsymbol{b}^l\}_{l=1}^{L}$ for all layers L
set hyperparameters for L-BFGS optimizer ($epochs$, learning rate α, ...)

for all $epochs$ **do**
  $\hat{\boldsymbol{u}}_b \leftarrow \boldsymbol{u}_{NN}(\boldsymbol{x}_b; \boldsymbol{\Theta})$
  $\boldsymbol{f} \leftarrow \boldsymbol{f}_{NN}(\boldsymbol{x}_f; \boldsymbol{\Theta})$
  compute MSE_b, MSE_f ▷ cf. Eqs. (5.12) and (5.13)
  compute cost function: $C \leftarrow MSE_b + MSE_f$
  update parameters: $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{\partial C}{\partial \boldsymbol{\Theta}}$ ▷ L-BFGS
end for

Alternatively, it is possible to use a strong enforcement of the Dirichlet boundary conditions to simplify the minimization problem. For that, the output of the neural network must be adapted to automatically satisfy the boundary conditions for any given input. Then, the mean squared error loss of the Dirichlet boundary conditions can be dropped. The strong enforcement of boundary conditions is demonstrated on a transient heat problem in Sect. 5.2.3.

The static bar example is now used to build up a PINN. The corresponding results are presented in Fig. 5.2. Here, the boundary conditions are enforced via the cost function. The plot on the left shows that the estimated displacements and the analytic displacements coincide. Additionally, the training history is illustrated on the right, showing the cost function for every training epoch. One observes that there is a difference in the order of magnitudes between the cost function of the differential equation and the cost function of the boundary conditions. Adjusting this difference with a weighting factor on the boundary condition term is possible, yet difficult to determine a priori. An alternative would be to enforce the Dirichlet boundary conditions strongly, which in this case leads to an unconstrained optimization problem, as only the differential equation cost has to be minimized.

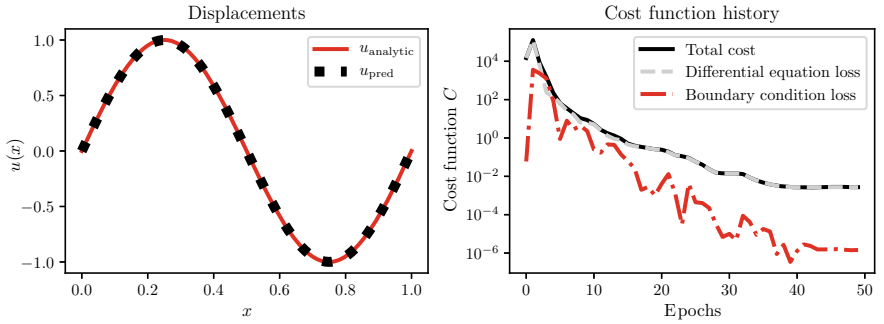


Fig. 5.2 Linear elastic static bar example. Left: the displacements $u(x)$ computed with a physics-informed neural network and the analytic solution are compared. Right: the training history is illustrated by showing the cost function for each epoch

5.2.2 *Continuous-Time Model*

To demonstrate the application of PINNs to transient problems, an initial-boundary value problem described by a one-dimensional non-linear heat equation is introduced. The evolution of temperature $u(t, x)$ as a function of time t and space x is described by a partial differential equation and boundary conditions of the following form

$$c\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left(\kappa\frac{\partial u}{\partial x}\right) - s = 0 \qquad \text{on } \mathcal{T} \times \Omega, \tag{5.14}$$

$$\kappa\frac{\partial u}{\partial x} = h \qquad \text{on } \mathcal{T} \times \Gamma_N, \tag{5.15}$$

$$u = g \qquad \text{on } \mathcal{T} \times \Gamma_D, \tag{5.16}$$

$$u(x, 0) = u_0 \qquad \text{on } \Omega. \tag{5.17}$$

Here, $\Omega \subset \mathbb{R}^1$ represents the one-dimensional spatial domain, whereas $\mathcal{T}$ denotes the temporal domain. Together they form the time-space domain $\mathcal{T} \times \Omega$ with Neumann and Dirichlet boundaries represented by Γ_N and Γ_D, respectively. In engineering applications the heat capacity $c(u)$ or the thermal conductivity $\kappa(u)$ are often temperature-dependent which introduces a non-linearity to Eq. (5.3).

The specific problem investigated in the following is defined on the time-space domain $\mathcal{T} \times \Omega = t \in [0, 0.5] \times x \in [0, 1]$ with $\Gamma_D = \emptyset$ and $\Gamma_N = \{x \mid x = 0,\ x = 1\}$. In particular, the problem is subject to homogeneous Neumann boundary conditions

$$\frac{\partial u(t, 0)}{\partial x} = \frac{\partial u(t, 1)}{\partial x} = 0, \tag{5.18}$$

and the initial condition

$$u(0, x) = u_0. \tag{5.19}$$

Both the heat capacity $c(u)$ and the thermal conductivity $\kappa(u)$ are temperature-dependent and defined as [Kol+18]

$$c(u) = 1/2000\, u^2 + 500, \tag{5.20}$$
$$\kappa(u) = 1/100\, u + 7. \tag{5.21}$$

In order to verify the network predictions, a manufactured solution of the following form is proposed

$$u = \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right), \tag{5.22}$$

where $\sigma = 0.02$. Normally, such Gaussian bell formulations are used to model a laser-induced heat source s [Kol+19] which is traveling along a path p

$$p(t) = \frac{1}{4}\cos\left(\frac{2\pi t}{t_{\max}}\right) + \frac{1}{2}. \tag{5.23}$$

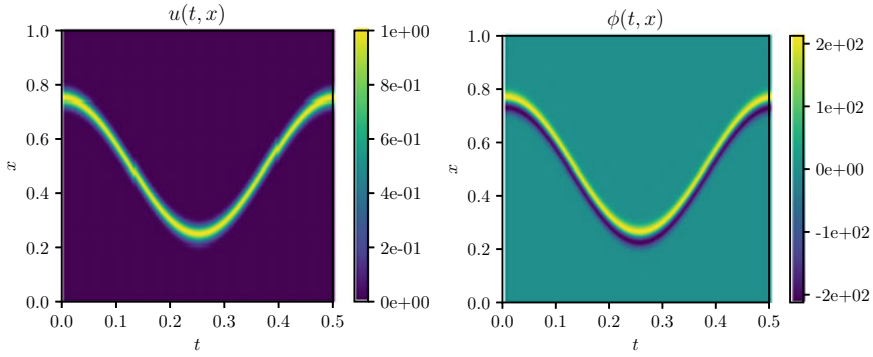
Here, the Gaussian bell term is chosen as the manufactured solution to the problem instead. Plugging Eq. (5.22) into (5.14) yields

$$s = \frac{\kappa u}{\sigma^2} + u\frac{x-p}{\sigma^2}\left[c\frac{\partial p}{\partial t} - \frac{x-p}{\sigma^2}\left(\kappa + u\frac{\partial \kappa}{\partial u}\right)\right]. \tag{5.24}$$

The method of manufactured solutions is commonly used to verify numerical solvers on sufficiently complex examples [Roa02]. Nonetheless, the manufactured solution from Eq. (5.22) should by no means be interpreted as an realistic example. It simply serves as a fast and direct way to benchmark the predictions of the physics-informed neural network. For the example at hand the benchmark solution u and the corresponding heat flux ϕ were generated with MATLAB at 201×256 discrete points in the time-space domain (cf. Fig. 5.3).

The implementation that generated the following results is based on the code accompanying the paper by Raissi et al. [Rai20, RPK19], but was adapted to employ the PyTorch framework. Like in the static example, a feed-forward neural network $u_{NN}(t, x; \boldsymbol{\Theta})$ is used to predict the hidden solution of the problem specified in Eqs. (5.14), (5.19), (5.18). The inputs of the network are the temporal variable t and the spatial variable x. The function `build_model` (cf. Sect. 5.2.1) determines the layer and neuron architecture of the feed-forward neural network u_{NN} which approximates the temperature distribution $u(t, x)$. To define the output of the physics-informed neural network $f_{NN}(t, x; \boldsymbol{\Theta})$ (cf. Fig. 5.4), the left-hand side of Eq. (5.14) is recalled as

$$f := c\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left(\kappa\frac{\partial u}{\partial x}\right) - s = c\frac{\partial u}{\partial t} - \frac{\partial \kappa}{\partial u}\frac{\partial u}{\partial x}\frac{\partial u}{\partial x} - \kappa\frac{\partial^2 u}{\partial x^2} - s. \tag{5.25}$$



(a) Manufactured solution u. **(b)** Manufactured heat flux ϕ.

Fig. 5.3 Manufactured solution u and corresponding flux ϕ for the one-dimensional heat transfer problem. The solution was generated using MATLAB with a resolution of $t \times x = 201 \times 256$ points

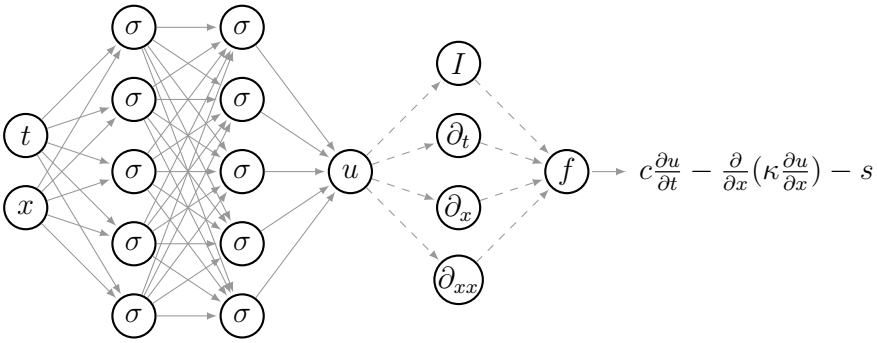


Fig. 5.4 Conceptual physics-informed neural network for the heat equation. The left side shows the feed-forward neural network and the right side represents the physics-informed part. The dashed lines denote non-trainable weights

The code corresponding to Eq. (5.25) is given as

```
def f_nn(self, t, x):
    u = self.u_nn(t, x)
    u_t = get_derivative(u, t, 1)
    u_x = get_derivative(u, x, 1)
    u_xx = get_derivative(u, x, 2)
    k = 0.01 * u + 7
    k_u = 0.01
    c = 0.0005 * u ** 2 + 500
    s = self.source_term(t,x)
    f = c * u_t - k_u * u_x * u_x - k * u_xx - s
    return f
```

Here, `self.u_nn(t,x)` calls the neural network $u_{NN}(t, x; \boldsymbol{\Theta})$ to return the predicted temperature `u`. Then, the previously introduced function `get_derivative` (cf. Sect. 5.2.1) is extended to recursively compute the necessary first- and second-order partial derivatives.

```
def get_derivative(y, x, n):
    if n == 0:
        return y
    else:
        dy_dx = grad(y, x, torch.ones_like(y), create_graph=True,
                     retain_graph=True, allow_unused=True)[0]
        return get_derivative(dy_dx, x, n - 1)
```

Furthermore, `self.source_term(t,x)` returns the source term s according to Eq. (5.24).

```
def source_term(self, t, x):
    t_max = 0.5
    sigma = 0.02
    u_max = 1
    p = 0.25 * torch.cos(2 * np.pi * t / t_max) + 0.5
    p_t = -0.5 * torch.sin(2 * np.pi * t / t_max) * np.pi / t_max
    u_sol = u_max * torch.exp(-(x - p) ** 2 / (2 * sigma ** 2))
    k_sol = 0.01 * u_sol + 7
    k_u_sol = 0.01
    c_sol = 0.0005 * u_sol ** 2 + 500
    factor = 1/(sigma ** 2)
    s = factor * k_sol * u_sol
        + u_sol * (x - p) * factor * (c_sol * p_t
        - (x - p) * factor * (k_sol + u_sol * k_u_sol))
    return s
```

In order to train the network, the following cost function is defined

$$C = MSE_0 + MSE_b + MSE_f. \tag{5.26}$$

Here, the term MSE_0

$$MSE_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} \left(u_{NN}(0, x_0^i; \boldsymbol{\Theta}) - u_0^i \right)^2, \tag{5.27}$$

enforces the initial condition (cf. Eq. (5.19)) by penalizing the error between the network prediction $\{u_{NN}(0, x_0^i, \boldsymbol{\Theta})\}_{i=1}^{N_0}$ and the initial solution $\{u_0^i\}_{i=1}^{N_0}$ at N_0 points randomly drawn from a uniform distribution. The term MSE_b

$$MSE_b = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(\frac{\partial}{\partial x} u_{NN}\left(t_b^i, 0; \boldsymbol{\Theta}\right) \right)^2 + \frac{1}{N_b} \sum_{i=1}^{N_b} \left(\frac{\partial}{\partial x} u_{NN}\left(t_b^i, 1; \boldsymbol{\Theta}\right) \right)^2, \tag{5.28}$$

enforces the Neumann boundary condition according to Eq. (5.18) at N_b random samples $\{t_b, x_b^i\}_{i=1}^{N_b}$ on each boundary $x = 0$ and $x = 1$. Finally, adding the error of

the residual MSE_f

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(f_{NN}\left(t_f^i, x_f^i\right)\right)^2 \tag{5.29}$$

ensures that the solution satisfies the governing Eq. (5.14). As proposed by Raissi et al. [RPK19], the N_f collocation points $\{t_f, x_f^i\}_{i=1}^{N_f}$ are generated by a Latin-hypercube sampling technique [Ste87]. Following Eqs. (5.27)–(5.29) the cost function is implemented as follows.

```
def cost_function(self, t0, x0, t_lb, x_lb, t_ub, x_ub, t_f, x_f, u0):
    u0_pred = self.u_nn(t0, x0)
    u_lb_pred = self.u_nn(t_lb, x_lb)
    u_x_lb_pred = get_derivative(u_lb_pred, x_lb, 1)
    u_ub_pred = self.u_nn(t_ub, x_ub)
    u_x_ub_pred = get_derivative(u_ub_pred, x_ub, 1)
    f_pred = self.f_nn(t_f, x_f)
    mse_0 = torch.mean((u0 - u0_pred)**2)
    mse_b = torch.mean(u_x_lb_pred**2) + torch.mean(u_x_ub_pred**2)
    mse_f = torch.mean((f_pred)**2)
    return mse_0, mse_b, mse_f
```

Subsequently, a full-batch gradient-based optimization procedure searches for the optimal network parameters $\boldsymbol{\Theta}^*$ by minimizing the cost function in Eq. (5.26). In addition to the L-BFGS method, a preceding minimization with the Adam optimizer is employed [KB17]. This combination was adapted from [Rai20] and has empirically proven to be the most robust approach throughout this study.

Algorithm 5 summarizes the previously introduced training procedure employed for a continuous prediction of the temperature distribution $u(t, x)$. Figure 5.5a shows the history of the cost function terms. After the algorithm terminated, the network $u_{NN}(t, x; \boldsymbol{\Theta})$ with trained parameters $\boldsymbol{\Theta}$ is used to predict the continuous temperature distribution over the whole domain $\mathcal{T} \times \Omega = [0, 0.5] \times [0, 1]$. The result along with a comparison between the prediction and the manufactured solution is presented in Fig. 5.6.

When optimizing a cost function that consists of multiple terms, it is important to balance the influence of each term. Figure 5.5b shows an example where no balancing is applied. The term MSE_f is magnitudes larger than the two other terms MSE_0 and MSE_b. Here, the initial and boundary conditions are underrepresented in the cost function and the algorithm does not converge to a correct solution (cf. Fig. B.1). Nabian and Meidani addressed this issue by introducing weight factors adjusting the relative importance of each term in the cost function [NM19]. In the example at hand, an empirically determined factor of 5×10^{-4} must be added to the term `mse_f` in `cost_function` to gain satisfactory results (cf. Figs. 5.5a and 5.6).

```
mse_f = torch.mean((5e-4*f_pred)**2)
```

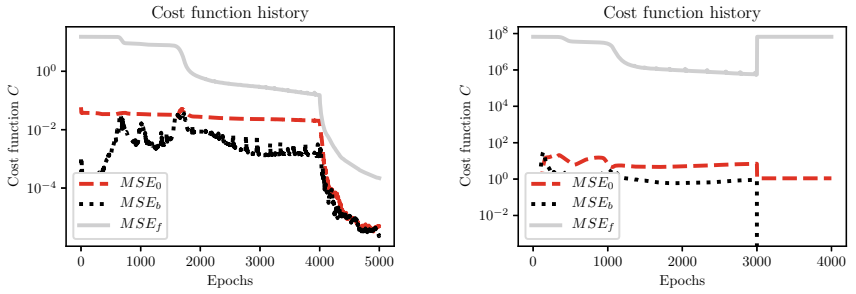
Algorithm 5 Training a physics-informed neural network for the continuous solution of the problem described in Eq. (5.14).

Require: training data for initial condition $\{0, x_0^i, u_0^i\}_{i=1}^{N_0}$
Require: training data for boundary condition $\{t_b, x_b^i, u_b^i\}_{i=1}^{N_b}$

generate N_f collocation points with Latin-hypercube sampling $\{t_f, x_f^i\}_{i=1}^{N_f}$
define network architecture (input, output, hidden layers, hidden neurons)
initialize network parameters $\boldsymbol{\Theta}$: weights $\{\boldsymbol{W}^l\}_{l=1}^{L}$ and biases $\{\boldsymbol{b}^l\}_{l=1}^{L}$ for all layers L
set hyperparameters for Adam optimizer (Adam-*epochs*, learning rate α, ...)
set hyperparameters for L-BFGS optimizer (L-BFGS-*epochs*, convergence criterion, ...)

procedure TRAIN
compute $\{u_{NN}(0, x_0^i, \boldsymbol{\Theta})\}_{i=1}^{N_0}$
compute $\{\frac{\partial}{\partial x} u_{NN}(t_b^i, x_b^i; \boldsymbol{\Theta})\}_{i=1}^{N_b}$
compute $\{f_{NN}(t_f^i, x_f^i; \boldsymbol{\Theta})\}_{i=1}^{N_f}$
compute MSE_0, MSE_b, MSE_f ▷ cf. Eqs. (5.27) to (5.29)
evaluate cost function: $C \leftarrow MSE_0 + MSE_b + MSE_f$
update parameters: $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{\partial C}{\partial \boldsymbol{\Theta}}$ ▷ Adam or L-BFGS
end procedure

for all Adam-*epochs* **do**
run TRAIN with Adam optimizer
end for
for all L-BFGS-*epochs* **do**
run TRAIN with L-BFGS optimizer
end for



(a) Balanced cost function terms plotted over the number of training iterations.

(b) Unbalanced cost function terms plotted over the number of training iterations.

Fig. 5.5 Balanced and unbalanced cost function history for 3000 Adam epochs and 1000 L-BFGS epochs

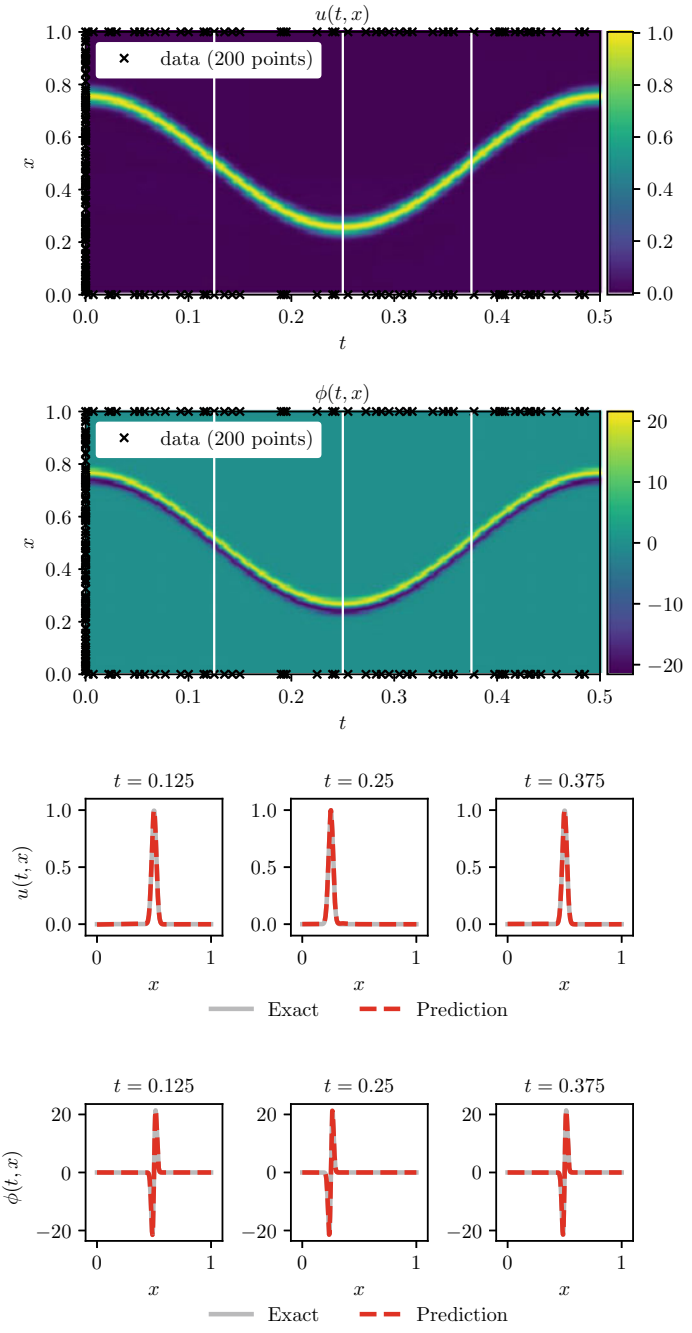


Fig. 5.6 Prediction of the temperature distribution u and corresponding heat flux ϕ. Top: approximated solution and location of time snapshots (white lines). Bottom: comparison of predicted and exact solution at distinct snapshots

5.2.3 Discrete-Time Model

To circumvent the need for collocation points Raissi et al. proposed an alternative solution-inference approach based on a Runge–Kutta time-stepping scheme. Contrary to a continuous prediction in time, the solution is only predicted at specific time steps. Assuming the solution at time step t^n is known, then the proposed method is able to predict the solution at the next time step $t^{n+1} = t^n + \Delta t$, where Δt denotes the step size.

To solve the problem

$$\frac{\partial u}{\partial t} = g[u], \tag{5.30}$$

the general form of the Runge–Kutta method with q stages is [Ise08]

$$\begin{aligned} u^{n+c_i} &= u^n + \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}\right], \quad i = 1, \ldots, q, \\ u^{n+1} &= u^n + \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}\right], \end{aligned} \tag{5.31}$$

where

$$u^{n+c_j}(x) = u(t^n + c_j \Delta t, x) \quad j = 1, \ldots, q. \tag{5.32}$$

Writing the Runge–Kutta method in this general form allows the usage of both explicit and implicit time-stepping schemes. The type of method depends on the choice of parameters a_{ij}, b_j, and c_j that are organized in a so-called Butcher table [Ise08]. The advantage of explicit methods is that they are fast and easy to implement. After the solution at the first stage is obtained, it is substituted into the equation at the second stage and so on. As the name suggests, implicit methods can not simply be solved by substitution. They form a system of dependent equations that requires the use of an iterative solution process. However, implicit methods exhibit excellent stability properties, which make them especially suitable for stiff systems.

Assuming a feed-forward neural network $\boldsymbol{u}_{NN}^{n+1}$ is able to predict the solution u^{n+1} at time t^{n+1} and the intermediate solutions u^{n+c_i} at all stages $i = 1, \ldots, q$ from an input x, then its output can be written as

$$\left[u^{n+c_1}(x), \ldots, u^{n+c_q}(x), u^{n+1}(x)\right] \leftarrow \boldsymbol{u}_{NN}^{n+1}. \tag{5.33}$$

In particular, the neural network predicts the left-hand side of Eq. (5.31). Rearranging Eq. (5.31) yields

$$
\begin{aligned}
u^n &= u^{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}\right], \quad i = 1, \ldots, q, \\
u^n &= u^{n+1} - \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}\right].
\end{aligned}
\tag{5.34}
$$

Now, all the terms dependent on the prediction of the neural network stand on the right-hand side and the solution at time t^n is found on the left. In other words, the time-stepping scheme is reversed to get an estimate of u^n that is dependent on the neural network prediction $\boldsymbol{u}_{NN}^{n+1}$. The error between this estimate and the known solution at time t^n is later used to formulate a cost function for training the network parameters. To assign a unique identifier to each equation in Eq. (5.34), the following nomenclature is introduced

$$
\begin{aligned}
u^n &= u_i^n, \quad i = 1, \ldots, q, \\
u^n &= u_{q+1}^n,
\end{aligned}
\tag{5.35}
$$

where

$$
\begin{aligned}
u_i^n &= u^{n+c_i} - \Delta t \sum_{j=1}^{q} a_{ij} g\left[u^{n+c_j}\right], \quad i = 1, \ldots, q \\
u_{q+1}^n &= u^{n+1} - \Delta t \sum_{j=1}^{q} b_j g\left[u^{n+c_j}\right].
\end{aligned}
\tag{5.36}
$$

Eventually, Eqs. (5.35) and (5.36) define the output of the physics-informed neural network $\boldsymbol{u}_{NN}^n$ for an input x as follows

$$
\left[u_1^n(x), \ldots, u_q^n(x), u_{q+1}^n(x)\right] \leftarrow \boldsymbol{u}_{NN}^n. \tag{5.37}
$$

Like in the continuous-time model, the physics-informed neural network consists of two parts (cf. Fig. 5.7). The first part is a deep neural network with multiple outputs as shown in (5.33). The second part transforms the output of the first network according to Eqs. (5.35) and (5.36) and returns the quantities in (5.37) for comparison with the known solution u^n at time t^n.

All in all, the network learns to predict the solution at time t^{n+1} based on the known solution at time t^n and the boundary conditions in the time interval $[t^n, t^{n+1}]$ by minimizing a suitable cost function. This step can be repeated to predict the solution at the following time steps $u(t^{n+2}, x)$, $u(t^{n+3}, x)$, and so on. When explicit methods are used, the step size Δt is chosen to be small in order to prevent stability issues. Implicit schemes are stable even for larger time steps. In classic, implicit discretization schemes, this simultaneously leads to a costly increase in the number of required stages q. However, what makes the proposed method distinct from the classical Runge–Kutta time-stepping schemes is the fact that the number of stages q can be increased without a significant increase in computational effort. Adding

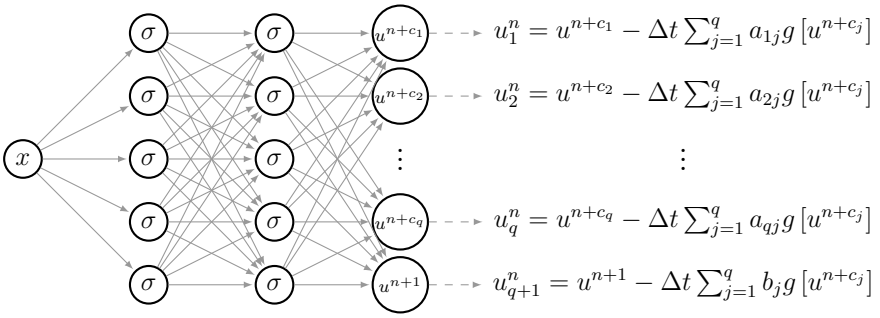


Fig. 5.7 Discrete physics-informed neural network with one input and $q+1$ outputs. The feed-forward neural network generates a prediction of u^{n+1} and the intermediate solutions u^{n+c_i}, $i = 1, \ldots, q$ for q stages, which are then used in Eqs. (5.35) and (5.36) to compute an output that can be compared to the solution u^n at initial time t^n

another stage to the model simply extends the output layer of the neural network by an extra neuron and the corresponding parameters. Overall, the parameters only increase linearly with the total number of stages.

To study the discrete solution method, the heat problem from the previous Sect. 5.2.2 is revisited. The governing partial differential equation is observed in the time-space domain $\mathcal{T} \times \Omega = [0, 0.5] \times [0, 1]$ with $\Gamma_D = \{x \mid x = 0,\ x = 1\}$ and $\Gamma_N = \emptyset$ and is written as

$$c\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left(\kappa\frac{\partial u}{\partial x}\right) - s = 0 \quad t \in [0, 0.5],\ x \in [0, 1]. \tag{5.38}$$

The temperature-dependent coefficients, namely, the heat capacity $c(u)$ and thermal conductivity $\kappa(u)$ are defined as [Kol+18]

$$c(u) = 1/2000\, u^2 + 500, \tag{5.39}$$
$$\kappa(u) = 1/100\, u + 7. \tag{5.40}$$

In contrast to the case studied in Sect. 5.2.2, the problem is now subject to homogeneous Dirichlet boundary conditions

$$u(0, t) = u(1, t) = 0. \tag{5.41}$$

Further, the problem is subject to the following initial condition

$$u(0, x) = u_0. \tag{5.42}$$

To validate the computation and to provide an initial solution snapshot at time t^n, the manufactured solution (cf. Eq. (5.22)) and the corresponding source term (cf. Eq. (5.24)) from Sect. 5.2.2 are reused. The implementation of the discrete method is inspired by the code from Raissi [Rai20], but uses the PyTorch library to construct the physics-informed neural network.

At first, the architecture of the physics-informed surrogate model for a desired time step $t^{n+1} = t^n + \Delta t$ and q stages is specified. Since a discretization in time is applied, the feed-forward neural network $\boldsymbol{u}_{NN}^{n+1}(x; \boldsymbol{\Theta})$ only takes the spatial variable x as an input. Next to the desired solution $u^{n+1}(x)$ at time t^{n+1}, the network predicts the intermediate solutions u^{n+c_i} for all stages $i = 1, \ldots, q$. Here, the parameters a_{ij}, b_j, and c_j for the implicit Runge–Kutta time-stepping scheme are taken from a Butcher table corresponding to the number of stages [Ise08]. To apply the temporal discretization, the problem described in Eq. (5.38) is rearranged in accordance with Eq. (5.30)

$$\frac{\partial u}{\partial t} = \frac{1}{c}\left(\frac{\partial \kappa}{\partial u}\frac{\partial u}{\partial x}\frac{\partial u}{\partial x} + \kappa\frac{\partial^2 u}{\partial x^2} + s\right). \tag{5.43}$$

Thus, $g\left[u^{n+c_j}\right]$ can be defined as

$$g\left[u^{n+c_j}\right] = \frac{1}{c}\left(\frac{\partial \kappa}{\partial u^{n+c_j}}\frac{\partial u^{n+c_j}}{\partial x}\frac{\partial u^{n+c_j}}{\partial x} + \kappa\frac{\partial^2 u^{n+c_j}}{\partial x^2} + s\right), \tag{5.44}$$

where

$$c(u) = 1/2000\,(u^{n+c_j})^2 + 500, \tag{5.45}$$

$$\kappa(u) = 1/100\,u^{n+c_j} + 7. \tag{5.46}$$

Following the modified Runge–Kutta time-stepping scheme from Eqs. (5.35) and (5.36) and using Eq. (5.44), the physics-informed neural network $\boldsymbol{u}_{NN}^n(x; \boldsymbol{\Theta})$ with outputs $\left[\hat{u}_1^n(x), \ldots, \hat{u}_q^n(x), \hat{u}_{q+1}^n(x)\right]$ is defined.

```
def U0_nn(self, x):
    U1 = self.U1_nn(x)
    U = U1[:, :-1]
    U_x = torch.zeros_like(U)
    U_xx = torch.zeros_like(U)
    for i in range(U.size(1)):
        U_x[:,i:i+1] = get_derivative(U[:,i], x, 1)
        U_xx[:,i:i+1] = get_derivative(U[:,i], x, 2)

    t = self.t0 + self.dt * self.IRK_times.T
    s = self.source_term(t, x)

    c = 0.0005 * U ** 2 + 500
```

```
        k = 0.01 * U + 7
        k_u = 0.01
        F = (k_u * U_x * U_x + k * U_xx + s) / c
        U0 = U1 - self.dt * torch.matmul(F, self.IRK_weights.T)
        return U0
```

Here, the feed-forward neural network `self.U1_nn` predicts the solution at time t^{n+1}. Its implementation is explained in more detail below. The variable `self.t0` stores the initial time t^n while `self.dt` corresponds to the time step size Δt. The parameters for the implicit Runge–Kutta time-stepping scheme a_{ij}, b_j are arranged in the matrix `self.IRK_weights`, while the vector `self.IRK_times` contains the parameters c_j.

So far, the initial and boundary conditions have been enforced weakly in the cost function. However, it is also possible to apply constraints in a strong sense. To do so, the solution u is modified to satisfy the boundary conditions for any given input. Following the generalized approach of Lagaris et al. [LLF98] for a strong enforcement, the network output is multiplied by $(1-x)x$ to consider for the homogeneous Dirichlet boundary conditions as prescribed in Eq. (5.41). Since the discrete solution is only dependent on the spatial variable x, the output can be written as

$$\left[\hat{u}^{n+c_1}, \ldots, \hat{u}^{n+c_q}, \hat{u}^{n+1}(x)\right] \leftarrow (1-x)x\, \boldsymbol{u}_{NN}^{n+1}(x;\boldsymbol{\Theta}) \quad i = 1, \ldots, q, \tag{5.47}$$

which easily translates into the following code.

```
def U1_nn(self, x):
    U1 = (1-x)*x*self.model(x)
    return U1  # N x (q+1)
```

Due to the discretization in time, no collocation points are needed to train the physics-informed neural network. Hence, the term MSE_f from Eq. (5.2) can be dropped. Moreover, the introduction of a strong boundary enforcement obviates the boundary term MSE_b as part of MSE_u. As a result, the cost function for training the network simplifies to a single loss term eliminating the need to determine suitable weighting factors. With the output of the physics-informed neural network $\left[\hat{u}_1^n(x), \ldots, \hat{u}_q^n(x), \hat{u}_{q+1}^n(x)\right] \leftarrow \boldsymbol{u}_{NN}^n(x;\boldsymbol{\Theta})$ (cf. (5.37)), the resulting cost function is written as

$$C = MSE_n, \tag{5.48}$$

where

$$MSE_n = \sum_{j=1}^{q+1} \sum_{i=1}^{N_n} \left(\hat{u}_j^n\left(x^i\right) - u^{n,i}\right)^2. \tag{5.49}$$

The term MSE_n computes the prediction error of the physics-informed neural network at N_n randomly sampled points $\{x^{n,i}, u^{n,i}\}_{i=1}^{N_n}$ of the solution at initial time t^n.

This could be the initial condition of the problem at hand or any other snapshot of the solution, e.g. at time $t^n = 0.05$. According to Eq. (5.49), the cost function can be implemented as follows:

```
def cost_function(self, x0, u0):
    U0_pred = self.U0_nn(x0)
    return torch.mean((U0_pred - u0)**2)
```

To learn the shared set of optimal parameters $\boldsymbol{\Theta}^*$, the combination of Adam optimizer and L-BFGS method is employed to minimize the cost function introduced in Eq. (5.48).

Finally, Algorithm 6 summarizes the previously elaborated steps for training the discrete-time physics-informed neural network. After the algorithm terminated, the network $\boldsymbol{u}_{NN}^{n+1}(x;\boldsymbol{\Theta})$ with the trained parameters $\boldsymbol{\Theta}$ is used to predict the temperature at time t^{n+1} for a given input x. For a prediction of the temperature distribution $u(x)$ at time $t^{n+1} = 0.3$ given a snapshot of $N_n = 200$ random data points at time $t^n = 0.05$, the cost function history and the results are shown in Figs. 5.8 and 5.9, respectively.

Algorithm 6 Training a discrete physics-informed neural network for a single time step $t^{n+1} = t^n + \Delta t$ and q stages.

Require: N_n training samples for initial snapshot $\{x^i, u^{n,i}\}_{i=1}^{N_n}$ at time t^n
define time step size Δt
define number of stages q
define network architecture (input, hidden layers, hidden neurons)
initialize output layer with $q+1$ neurons
initialize network parameters $\boldsymbol{\Theta}$: weights $\{\boldsymbol{W}^l\}_{l=1}^{L}$ and biases $\{\boldsymbol{b}^l\}_{l=1}^{L}$ for all layers L
set hyperparameters for Adam optimizer (Adam-*epochs*, learning rate α, ...)
set hyperparameters for L-BFGS optimizer (L-BFGS-*epochs*, convergence criterion, ...)

procedure TRAIN
  compute $\{\left[\hat{u}_1^n(x^i), \ldots, \hat{u}_q^n(x^i), \hat{u}_{q+1}^n(x^i)\right] \leftarrow \boldsymbol{u}_{NN}^n(x^i;\boldsymbol{\Theta})\}_{i=1}^{Nn}$
  compute MSE_n ▷ cf. Eq. (5.49)
  $C \leftarrow MSE_n$
  update parameters: $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \alpha \frac{\partial C}{\partial \boldsymbol{\Theta}}$ ▷ Adam or L-BFGS
end procedure
for all Adam-*epochs* **do**
  run TRAIN with Adam optimizer
end for
for all L-BFGS-*epochs* **do**
  run TRAIN with L-BFGS optimizer
end for

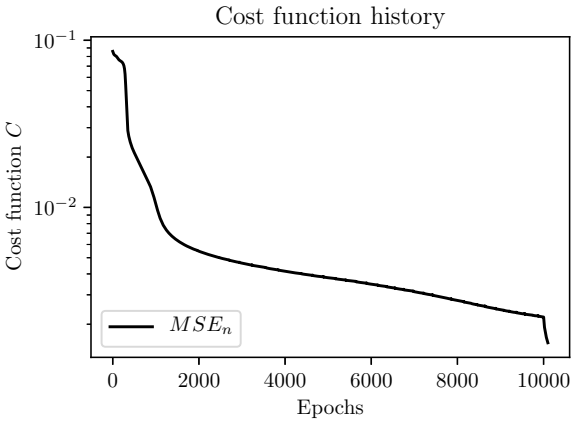


Fig. 5.8 Cost function history of discrete-time inference physics-informed neural network

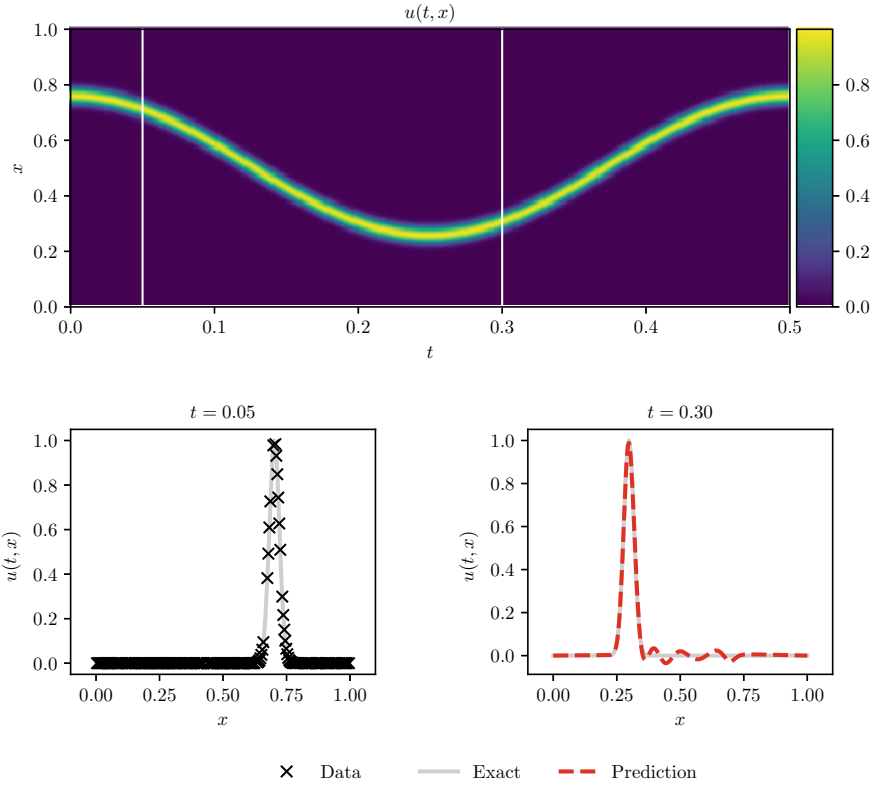


Fig. 5.9 Predictions of the inference physics-informed neural network. Top: manufactured solution u and time snapshots $t^n = 0.05$ and $t^{n+1} = 0.3$ (white lines). Bottom: training data at $t^n = 0.05$ and prediction at $t^{n+1} = 0.3$

5.3 Data-Driven Identification

System identification from sparse data is another class of problems often encountered in physics or engineering applications. In the second part of their article, Raissi et al. addressed the problem of data-driven discovery with the help of the previously introduced physics-informed neural networks. In other words, the task of system identification can be phrased as: find the parameters λ that describe the observed data best. The main idea is to use the non-linear partial differential equations (5.1) together with a solution $u(t, x)$ to solve the inverse problem of identifying the parameters λ.

A neural network is used to predict the parameter λ instead of the solution $u(t, x)$. The remaining steps are almost identical to the data-driven inference. The partial differential equation is represented by the gradients of the solution u and parameter λ with respect to the input parameters x and t. These are used to compute the mean squared error loss of the partial differential equation MSE_f, boundary conditions MSE_b and initial conditions MSE_0. The sum of the mean squared error losses is then used as a cost function as in Eq. (5.2). If the boundary or initial conditions are independent of the parameter λ, their corresponding loss terms can be neglected. Finally, the cost function is minimized by the procedure previously explained in Sect. 5.2.

5.3.1 Static Model

To illustrate the data-driven identification using a physics-informed neural network, the example presented in Sect. 5.2.1 is revisited. However, now it is assumed that the displacement $u(x)$ is known while the cross-sectional properties $EA(x)$ are to be determined. As before, the domain $\Omega = [0, 1]$ is investigated. To increase the complexity of the problem, a cubic variation in the cross-sectional properties is defined as

$$EA(x) = x^3 - x^2 + 1. \tag{5.50}$$

Again, the manufactured solution $u(x)$ is chosen as

$$u(x) = \sin(2\pi x), \tag{5.51}$$

which leads to the following distributed load $p(x)$ after insertion into the differential equation (5.4)

$$p(x) = -2(3x^2 - 2x)\pi \cos(2\pi x) + 4(x^3 - x2 + 1)\pi^2 \sin(2\pi x). \tag{5.52}$$

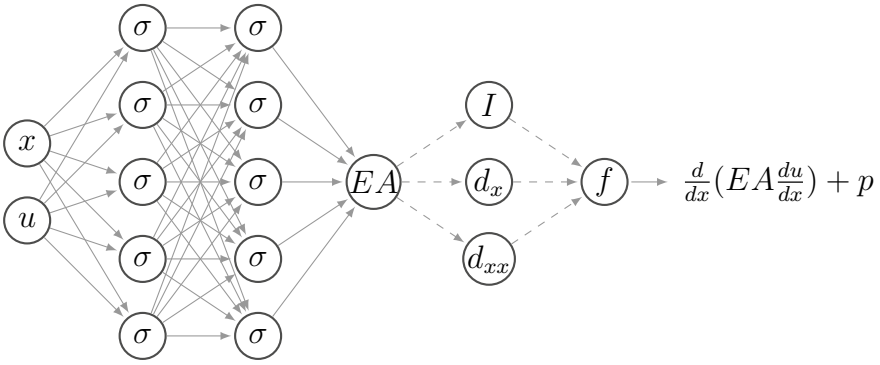


Fig. 5.10 Conceptual physics-informed neural network for data-driven identification of the static bar equation. The left part shows the feed-forward neural network and the right part represents the physics-informed neural network. The dashed lines denote non-trainable weights

The chosen solution leads to the boundary conditions

$$u(0) = u(1) = 0. \tag{5.53}$$

However, the boundary conditions do not have any influence on the learning of the model parameters, as the model parameters are not part of the boundary conditions. Therefore, the boundary loss MSE_b can be omitted.

The network architecture to solve this inverse problem, illustrated in Fig. 5.10 is similar to the one presented in Sect. 5.2.1, Fig. 5.1. The difference lies in the inputs being the coordinates x as well as the displacements u. Additionally, the left feed-forward neural network predicts the stiffness EA instead of the displacements u. The differential equation loss is identical to Eq. (5.13), while the boundary loss is omitted.

Given this information, an inverse physics-informed neural network can be constructed to predict the cross-sectional properties $EA(x)$ of the bar. This is shown in Fig. 5.11. Here, it is seen that the varying cross-section is approximated well. Additionally, the training history is shown along with the cost function.

Note that variations of this inverse network exist. Raissi et al. [RPK19] extend a neural network with a trainable parameter EA. Their neural network uses coordinates x as inputs and displacements u as output, similar to the forward-driven approach. The cost function is then defined as the sum of the mean squared error between the measurements $\hat{u}$ and the predictions u and the physics-informed loss. Tartakovsky et al. [Tar+18] use two neural networks. The first network predicts the displacement u from the coordinates x, and the second network predicts the stiffness EA from the predicted displacements u.

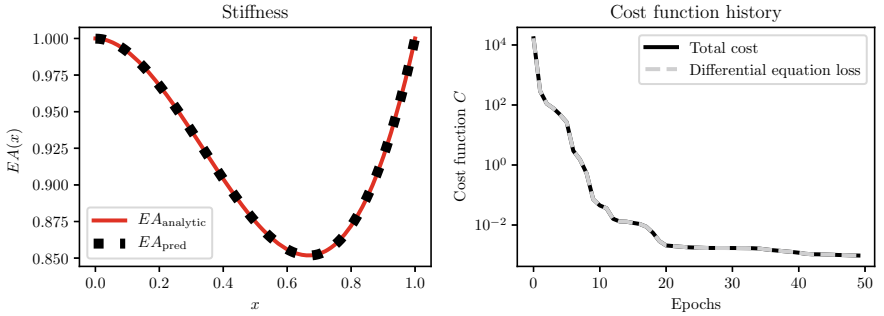


Fig. 5.11 Linear elastic static bar example. Left: the cross-sectional properties $EA(x)$ computed with a physics-informed neural network and the analytic expression are compared. Right: the training history is illustrated by showing the cost function for each epoch

5.4 Related Work

In earlier work [RPK17a, RPK17b], Gaussian processes were used for the inference and identification of differential equations. However, certain limitations imposed by the nature of Gaussian processes [RK18, RPK17c] led to the development of neural-network-based approaches [RPK19].

The introduction of physics-informed neural networks inspired fellow researchers to adapt and extend the proposed method. For instance, Pang et al. introduced a hybrid approach, that combines numerical discretization and physics-informed neural networks to solve space-time fractional advection–diffusion equations [PLK18]. Another extension is tailored to solve three-dimensional fluid flow problems by encapsulating the governing physics of the Navier–Stokes equation [RYK18]. Following classical numerical approximations from the family of Galerkin methods, Kharazmi et al. proposed a physics-enriched network, that is trained by minimizing the variational formulation of the underlying partial differential equation [KZK19]. Choosing a variational residual as the cost function reduces the order of the differential operator, and thus promises to simplify the optimization problem during training. Nevertheless, the use of a variational description does not come without limitations as the choice of integration points and the enforcement of Dirichlet boundaries require special treatment.

Since using the variational formulation of a problem is the typical approach for the numerical solution of partial differential equations, it is not surprising to find this idea also in other related publications. For example, Samaniego et al. followed this paradigm and translated the variational energy formulation of mechanical systems into the cost function of a deep learning model [Sam+19]. Next to a description of their physics-informed neural network implementation, the publication entailed several example applications from the field of computational solid mechanics, like phase-field modeling of fracture or bending of a Kirchhoff plate.

Motivated by the successful application of machine learning algorithms to the solution of high-order non-linear partial differential equations [BEJ19], E and Yu employed a deep residual neural network for solving variational problems [EY17]. Residual neural networks form an extension of feed-forward neural networks (cf. Sect. 3.1) which introduce additional connections between non-adjacent layers. In this way, some outputs can skip intermediate layers, which reduces the difficulty of training extremely deep networks [He+15]. Nabian and Meidani et al. also made use of this network architecture in order to build a surrogate model for high-dimensional random partial differential equations [NM19]. On the academic examples of diffusion and heat conduction they demonstrated that their implementation is able to deal with the strong and the variational formulation of the problem. The challenge of solving high-dimensional partial differential equations also motivated Sirignano and Spiliopoulos to investigate the applicability of deep learning in this context [SS18]. Their method can solve free-boundary partial differential equations in up to 200 dimensions. Apart from a special treatment of the free boundaries, they proposed a Monte Carlo-based method for the computation of second derivatives in higher dimensions.

The aforementioned papers dealt only with the inference of solutions to partial differential equations. An alternative approach for data-driven discovery of partial differential equations was presented by Rudy et al. [Rud+17]. They introduced a technique for discovering governing equations and physical laws by observing time-series measurements in the spatial domain. Their method allows the use of either an Eulerian or a Lagrangian reference frame.

A major drawback of the approaches introduced in the previous sections is that training a neural network is generally much more computationally expensive than applying a conventional numerical method. Since the physics-informed neural networks are learning the solution of a problem for specific initial conditions, boundary conditions and material parameters, a small change in one of them demands a complete re-training. Hence, they do not represent a viable alternative in terms of computational costs. In recent publications, Zhu et al. [Zhu+19], and Geneva and Zabaras [GZ20] tried to overcome this issue by building physics-constrained surrogate models that are able to accurately predict solutions for a range of initial conditions. Their goal was to construct a model that provides a time-discretized solution given only the initial state of the system. For that, they used an autoencoder (auto-regressive dense encoder-decoder) to produce a mapping from one time step to the other similar to a discrete time-stepping scheme. The autoencoder employed convolutional neural networks (cf. Sect. 3.10.1) that offer an effective alternative to fully connected feed-forward architectures when modeling transient partial differential equations [Zhu+19]. To train the network parameters, a cost function compares the autoencoder prediction with the result of a time integration step. Figure 5.12 shows the training procedure where the autoencoder predicts multiple time steps before the network parameters are updated with backpropagation. In contrast to physics-informed neural networks and related approaches, their method is not dependent on labeled training data, e.g. a prescribed solution at the boundaries. The initial training of the autoencoder network is still in a timely manner. However, after the training is

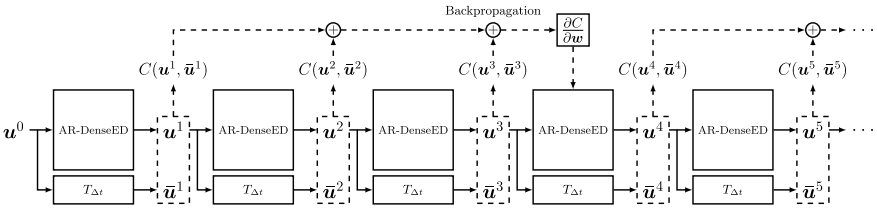


Fig. 5.12 Composition of cost function used for training the autoencoder (AR-DenseED) with backpropagation. Here, $\boldsymbol{w}$ denotes the learnable parameters and $\boldsymbol{u}^n$ represents the prediction at time step n. Further, $\bar{\boldsymbol{u}}^n$ describes the target computed with the numerical time integration scheme $T_{\Delta t}$, while $C(\boldsymbol{u}^n, \bar{\boldsymbol{u}}^n)$ is the physics-constrained cost function. In the shown example, three predictions are considered during the parameter update with backpropagation. Image adapted from Geneva and Zabaras [GZ20] with permission from Elsevier

completed the model can be used to predict the solution of non-linear partial differential equations for a wide range of initial conditions. Geneva and Zabaras showed that in case of non-linear dynamical systems their approach yielded results magnitudes faster than classical FEM or FDM solvers. In particular, the authors were able to solve and model the Kuramoto-Sivashinsky equation as well as the one- and two-dimensional Burgers' equations. Furthermore, their method achieved satisfactory generalization properties outside the training domain. Another interesting feature introduced is a Bayesian framework that allows to quantify the uncertainty of the predicted quantities. Even though promising results were presented, their approach still exhibited certain downsides. Since a discretization in time was used, the method is prone to instabilities typical for any kind of numerical approximation. According to the authors, those issues could be resolved with the same techniques used to stabilize classical discrete time-stepping schemes.

References

[PU92] Dimitris C. Psichogios and Lyle H. Ungar. "A hybrid neural network-first principles approach to process modeling". In: *AIChE J.* 38.10 (Oct. 1992), pp. 1499–1511. ISSN: 0001-1541, 1547-5905. DOI https://doi.org/10.1002/aic.690381003 (visited on 07/02/2020).

[LLF98] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Trans. Neural Netw.* 9.5 (Sept. 1998), pp. 987–1000. ISSN: 10459227. DOI https://doi.org/10.1109/72.712178 (visited on 01/08/2020).

[Kon18] Risi Kondor. "N-body Networks: a Covariant Hierarchical Neural Network Architecture for Learning Atomic Potentials". In: arXiv:1803.01588 [cs] (Mar. 5, 2018) (visited on 07/15/2020).

[HMP17] Matthew Hirn, Stéphane Mallat, and Nicolas Poilvert. "Wavelet Scattering Regression of Quantum Chemical Energies". In: *Multiscale Model. Simul.* 15.2 (Jan. 2017), pp. 827–863. ISSN: 1540-3459, 1540-3467. DOI https://doi.org/10.1137/16M1075454. arXiv:1605.04654 (visited on 07/15/2020).

[Mal16] Stéphane Mallat. "Understanding deep convolutional networks". In: *Phil. Trans. R. Soc. A* 374.2065 (Apr. 13, 2016), p. 20150203. ISSN: 1364-503X, 1471-2962. DOI https://doi.org/10.1098/rsta.2015.0203 (visited on 07/15/2020).

[RPK19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. ISSN: 00219991. DOI https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021999118307125 (visited on 01/08/2020).

[Rai20] Maziar Raissi. *maziarraissi/PINNs*. original-date: 2018-01-21T04:04:32Z. July 25, 2020. URL: https://github.com/maziarraissi/PINNs (visited on 07/27/2020).

[BNK20] Steven Brunton, Bernd Noack, and Petros Koumoutsakos. "Machine Learning for Fluid Mechanics". In: *Annu. Rev. Fluid Mech*. 52.1 (Jan. 5, 2020), pp. 477–508. ISSN: 0066-4189, 1545-4479. DOI https://doi.org/10.1146/annurev-fluid-010719-060214. arXiv:1905.11075 (visited on 06/26/2020).

[FDC20] Michael Frank, Dimitris Drikakis, and Vassilis Charissis. "Machine-Learning Methods for Computational Science and Engineering". In: *Computation* 8.1 (Mar. 3, 2020), p. 15. ISSN: 2079-3197. DOI https://doi.org/10.3390/computation8010015. URL: https://www.mdpi.com/2079-3197/8/1/15 (visited on 07/02/2020).

[Sam+19] Esteban Samaniego et al. "An Energy Approach to the Solution of Partial Differential Equations in Computational Mechanics via Machine Learning: Concepts, Implementation and Applications". In: arXiv:1908.10407 [cs, math, stat] (Sept. 2, 2019) (visited on 01/08/2020).

[LN89] Dong C. Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45.1 (Aug. 1989), pp. 503–528. ISSN: 0025-5610, 1436-4646. DOI https://doi.org/10.1007/BF01589116 (visited on 07/13/2020).

[Kol+18] S. Kollmannsberger et al. "A hierarchical computational model for moving thermal loads and phase changes with applications to selective laser melting". In: *Computers & Mathematics with Applications* 75.5 (Mar. 2018), pp. 1483–1497. ISSN: 08981221. DOI https://doi.org/10.1016/j.camwa.2017.11.014. URL: https://linkinghub.elsevier.com/retrieve/pii/S0898122117307289 (visited on 07/21/2020).

[Kol+19] Stefan Kollmannsberger et al. "Accurate Prediction of Melt Pool Shapes in Laser Powder Bed Fusion by the Non-Linear Temperature Equation Including Phase Changes: Model validity: isotropic versus anisotropic conductivity to capture AM Benchmark Test AMB2018-02". In: *Integr Mater Manuf Innov* 8.2 (June 2019), pp. 167–177. ISSN: 2193-9764, 2193-9772. DOI https://doi.org/10.1007/s40192-019-00132-9 (visited on 07/20/2020).

[Roa02] Patrick J. Roache. "Code Verification by the Method of Manufactured Solutions". In: *J. Fluids Eng* 124.1 (Mar. 1, 2002). Publisher: American Society of Mechanical Engineers Digital Collection, pp. 4–10. ISSN: 0098-2202. DOI https://doi.org/10.1115/1.1436090. URL: https://asmedigitalcollection.asme.org/fluidsengineering/article/124/1/4/462791/Code-Verification-by-the-Method-of-Manufactured (visited on 07/23/2020).

[Ste87] Michael Stein. "Large Sample Properties of Simulations Using Latin Hypercube Sampling". In: *Technometrics* 29.2 (May 1987), pp. 143–151. ISSN: 0040-1706, 1537-2723. DOI https://doi.org/10.1080/00401706.1987.10488205 (visited on 07/13/2020).

[KB17] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: arXiv:1412.6980 [cs] (Jan. 29, 2017) (visited on 07/30/2020).

[NM19] Mohammad Amin Nabian and Hadi Meidani. "A Deep Neural Network Surrogate for High-Dimensional Random Partial Differential Equations". In: *Probabilistic Engineering Mechanics* 57 (July 2019), pp. 14–25. ISSN: 02668920. DOI https://doi.org/10.1016/j.probengmech.2019.05.001. arXiv:1806.02957 (visited on 02/21/2020).

[Ise08] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Google-Books-ID: 3acgAwAAQBAJ. Cambridge University Press, Nov. 27, 2008. 481 pp. ISBN: 978-1-139-47376-7.

[Tar+18] Alexandre M. Tartakovsky et al. "Learning Parameters and Constitutive Relationships with Physics Informed Deep Neural Networks". In: (Aug. 2018). URL: https://arxiv.org/pdf/1808.03398.pdf.

[RPK17a] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Inferring solutions of differential equations using noisy multi-fidelity data". In: *Journal of Computational Physics* 335 (Apr. 2017), pp. 736–746. ISSN: 00219991. DOI https://doi.org/10.1016/j.jcp.2017.01.060. arXiv:1607.04805 (visited on 07/16/2020).

[RPK17b] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Machine learning of linear differential equations using Gaussian processes". In: *Journal of Computational Physics* 348 (Nov. 1, 2017), pp. 683–693. ISSN: 0021-9991. DOI https://doi.org/10.1016/j.jcp.2017.07.050. URL: http://www.sciencedirect.com/science/article/pii/S0021999117305582 (visited on 07/16/2020).

[RK18] Maziar Raissi and George Em Karniadakis. "Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations". In: *Journal of Computational Physics* 357 (Mar. 2018), pp. 125–141. ISSN: 00219991. DOI https://doi.org/10.1016/j.jcp.2017.11.039. arXiv:1708.00588 (visited on 07/16/2020).

[RPK17c] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Numerical Gaussian Processes for Time-dependent and Non-linear Partial Differential Equations". In: arXiv:1703.10230 [cs, math, stat] (Mar. 29, 2017) (visited on 07/16/2020).

[PLK18] Guofei Pang, Lu Lu, and George Em Karniadakis. "fPINNs: Fractional Physics-Informed Neural Networks". In: arXiv:1811.08967 [physics] (Nov. 19, 2018) (visited on 07/16/2020).

[RYK18] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. "Hidden Fluid Mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data". In: arXiv:1808.04327 [physics, stat] (Aug. 13, 2018) (visited on 04/09/2020).

[KZK19] E. Kharazmi, Z. Zhang, and G. E. Karniadakis. "Variational Physics- Informed Neural Networks For Solving Partial Differential Equations". In: arXiv:1912.00873 [physics, stat] (Nov. 27, 2019) (visited on 07/16/2020).

[BEJ19] Christian Beck, Weinan E, and Arnulf Jentzen. "Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations". In: *J Nonlinear Sci* 29.4 (Aug. 2019), pp. 1563–1619. ISSN: 0938-8974, 1432-1467. DOI https://doi.org/10.1007/s00332-018-9525-3. arXiv:1709.05963 (visited on 07/16/2020).

[EY17] Weinan E and Bing Yu. "The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems". In: arXiv:1710.00211 [cs, stat] (Sept. 30, 2017) (visited on 01/14/2020).

[He+15] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: arXiv:1512.03385 [cs] (Dec. 10, 2015) (visited on 07/16/2020).

[SS18] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations". In: *Journal of Computational Physics* 375 (Dec. 2018), pp. 1339–1364. ISSN: 00219991. DOI https://doi.org/10.1016/j.jcp.2018.08.029. arXiv:1708.07469 (visited on 01/08/2020).

[Rud+17] Samuel H. Rudy et al. "Data-driven discovery of partial differential equations". In: *Sci. Adv.* 3.4 (Apr. 2017), e1602614. ISSN: 2375-2548. DOI https://doi.org/10.1126/sciadv.1602614 (visited on 01/08/2020).

[Zhu+19] Yinhao Zhu et al. “Physics-Constrained Deep Learning for High-dimensional Surrogate Modeling and Uncertainty Quantification without Labeled Data”. In: *Journal of Computational Physics* 394 (Oct. 2019), pp. 56–81. ISSN: 00219991. DOI https://doi.org/10.1016/j.jcp.2019.05.024. arXiv:1901.06314 (visited on 07/06/2020).

[GZ20] Nicholas Geneva and Nicholas Zabaras. “Modeling the Dynamics of PDE Systems with Physics-Constrained Deep Auto-Regressive Networks”. In: *Journal of Computational Physics* 403 (Feb. 2020), p. 109056. ISSN: 00219991. DOI https://doi.org/10.1016/j.jcp.2019.109056. arXiv:1906.05747 (visited on 11/09/2020).