Chapter 5
Physics-Informed Neural Networks: Theory and Applications

Cosmin Anitescu, Burak İsmail Ateş, and Timon Rabczuk

5.1 Introduction

Machine learning (ML) methods based on artificial neural networks (ANNs) have become increasingly used, particularly in data-rich fields such as text, image, and audio processing, where they have achieved remarkable results, greatly surpassing the previous state-of-the-art algorithms. Typically, ML methods are most efficient in applications where the patterns are difficult to describe by clear-cut rules, such as handwriting recognition. In these cases, it may be more efficient to generate the rules by a kind of high-dimensional regression between a sufficiently large number of input–output pairs. However, other techniques based on ANNs have also been successful in domains where the rules are relatively easy to describe, such as AlphaZero (Silver et al. 2017) for game playing and AlphaFold (Jumper et al. 2021) for protein folding. Many of these advancements have been driven by an increase in computational capabilities, in particular with regard to Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) (Jouppi et al. 2017), but also by theoretical advances related to the initialization and architecture of the ANNs. In the scientific community, there has also been increased interest in applying the new developments in ANNs and ML to solve partial differential equations (PDEs) and other engineering problems of interest.

C. Anitescu · T. Rabczuk (✉)
Institut für Strukturmechanik, Bauhaus-Universität Weimar, Weimar, Germany
e-mail: timon.rabczuk@uni-weimar.de

C. Anitescu
e-mail: cosmin.anitescu@uni-weimar.de

B. İsmail Ateş
School of Engineering and Design, Technical University of Munich, Munich, Germany
e-mail: burak.ismail.ates@tum.de

T. Rabczuk and K. J. Bathe (eds.), *Machine Learning in Modeling and Simulation*, Computational Methods in Engineering & the Sciences,
https://doi.org/10.1007/978-3-031-36644-4_5

One can distinguish between supervised, unsupervised, and reinforcement learning. In the former, the aim is to find the mapping between a set of inputs and outputs, such as images of hand-written digits and the actual digit they represent, so that when a new input is presented, the correct output can be predicted by the ML algorithm. A prerequisite for the application of these methods is the availability of *labeled data*. In engineering applications, such approaches can be used e.g. for predicting the solution from the boundary conditions for a given PDE based on a large set of inputs/solutions pairs of similar problems; see also operator-approximation methods (Li et al. 2020a, b; Lu et al. 2021). However, a drawback is the requirement for possibly large amounts of labeled data (i.e. solved examples) drawn from the same distribution as the problems that we like to solve in the first place. On the contrary, in unsupervised learning the algorithm aims to find patterns in the input data to produce useful output based on some hard-coded rules or objectives. In classical ML, such tasks include image segmentation, dimensionality reduction (such as principal component analysis or PCA), or different types of clustering (grouping unlabeled data based on similarities or differences). Furthermore, there is a middle ground category of semi-supervised learning, where a mixture of labeled and unlabeled data is used in an attempt to overcome some of the shortcomings of the first two categories. Related to this, is the concept of reinforcement learning, where an agent-based system seeks to learn the actions that maximize a reward function.

Physics-informed neural networks (PINNs) are more closely related to the unsupervised or semi-supervised learning, whereby satisfying the governing equations, including the boundary conditions, at a given set of collocation points defines the objective function. This idea was originally proposed during the 1990s in Lagaris et al. (1998), Lagaris et al. (1997) and further extended for domains with irregular boundaries in Lagaris et al. (2000). As the cost of training neural networks became cheaper, further developments have been first reported in Raissi et al. (2019), Sirignano and Spiliopoulos (2018) among others, including the extension to time-dependent problems and model parameter inference (i.e. inverse problems). In Raissi et al. (2019), the term PINN is first used, along with the concept of combining (possibly noisy) experimental data with the governing equation in a *small data* or semi-supervised setting. Since then, several improvements have been suggested, such as adaptively choosing the collocation points (Anitescu et al. 2019; Wight and Zhao 2020), variational formulations (Yu et al. 2018; Samaniego et al. 2020; Kharazmi et al. 2019), and domain decomposition approaches (Shukla et al. 2021). Moreover, PINNs have been applied to a wide variety of problems, such as hyperelasticity (Nguyen-Thanh et al. 2020), multiphase poroelasticity (Haghighat et al. 2022), Kirchhoff plates (Zhuang et al. 2021), eikonal equation (bin Waheed et al. 2021), biophysics (Kissas et al. 2020), quantum chemistry (Pfau et al. 2020), materials science (Shukla et al. 2020) and others.

In this chapter, we give a concise overview of the main ideas of PINNs, focusing on the implementation and potential applications to forward and inverse PDEs. In Sect. 5.2, we introduce the building blocks required to create and train a neural network model, while, in Sect. 5.3, we present the collocation and energy minimization approaches, along with a discussion of enforcing the boundary conditions. In Sect.

5.4, we present some numerical examples, focusing on some pedagogical examples of standard PINNs for problems that are feasible to compute on regular desktops or even mobile computers, followed by some concluding remarks in Sect. 5.5.

5.2 Basics of Artificial Neural Networks

An artificial neural network (ANN) is loosely modeled after the structure of the brain, which is made up of a large number of cells (neurons) which communicate with their neighbors through electrical signals. Mathematically, an ANN can be seen as a function $u_{NN} : \mathbb{R}^n \to \mathbb{R}^m$, which maps n inputs into m outputs. An ANN is a universal function approximator (Hornik et al. 1989). Therefore, u_{NN} can be used to interpolate some unknown function from the data given at certain points or to approximate the solution of a partial differential equation. The function u_{NN} depends on a collection of parameters (called *trainable parameters*) which are obtained by an optimization procedure with the goal of minimizing some user-defined objective or *loss* function.

In an ANN, the neurons, or computational units, are organized in *layers* which are connected by composition with an *activation function* as detailed below. Different types of layers (and activation functions) can be assembled together, according to the application and the information known about the function to be approximated. There are several types of ANNs, which include fully connected feed-forward networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), residual neural networks (ResNets), transformers, and others. In the following, we will focus mostly on the feed-forward neural networks which are among the simplest and can also be used as building blocks of more complicated architectures.

5.2.1 Feed-Forward Neural Networks

In this type of network, also called multi-layer perceptron (MLP), the output is obtained by successive compositions of a linear transformation and a nonlinear activation function. The network consists of an *input layer*, an *output layer*, and any number of intermediate *hidden layers*. The function u_{NN} for a network with an n-dimensional input, and m-dimensional output and k hidden layers can be written as

$$u_{NN} = L_k \circ L_{k-1} \circ \ldots \circ L_0 \tag{5.1}$$

with

$$L_i(\mathbf{x}_i) = \sigma_i\left(\mathbf{W}_i\mathbf{x}_i + \mathbf{b}_i\right) = \mathbf{x}_{i+1} \text{ for } i = 0, \ldots, k. \tag{5.2}$$

Here, $\mathbf{W}_i$ are matrices of size $m_i \times n_i$, with $n_0 = n$, $n_{i+1} = m_i$, and $m_k = m$, $\mathbf{x}_{i+1}$ and $\mathbf{b}_i$ are column vectors of size m_i, and the activation functions σ_i are applied

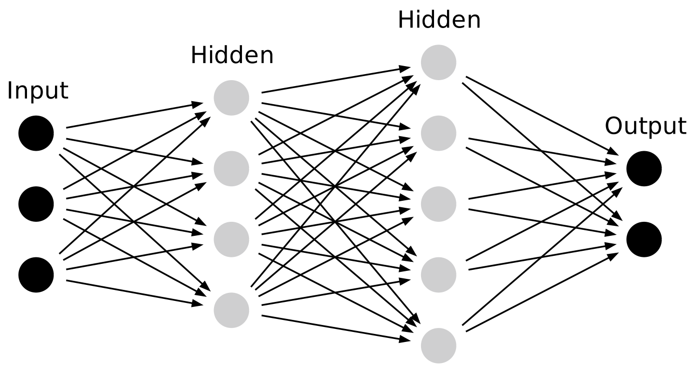


Fig. 5.1 A fully connected feed-forward neural network with the input, hidden, and output layers

element-wise to the vectors $\mathbf{W}_i \mathbf{x}_i + \mathbf{b}_i$. The entries of the matrices $\mathbf{W}_i$ are called *weights* and those of the vectors $\mathbf{b}_i$ are called *biases*, and together they represent the trainable parameters of the neural network. For $k > 0$, the values $m_0, \ldots, m_{k-1}$ can be chosen freely and represent the number of neurons in each hidden layer. If the number of hidden layers $k > 1$, we say that u_{NN} is a *deep* neural network. A schematic of a feed-forward network with 3 neurons in the input layer, two hidden layers with 4 and 5 neurons, respectively, and an output layer consisting of 2 neurons is shown in Fig. 5.1.

In a typical application, many inputs are collected in a *batch* and evaluated together. Evaluating the output of the neural network involves mainly linear algebra operations (such as matrix and vector products) which can be easily parallelized. In machine learning frameworks, such as TensorFlow (Abadi et al. 2015), PyTorch (Paszke et al. 2019), or JAX (Bradbury et al. 2018), a *computational graph* is built to record the different operations. This allows for efficient evaluation and also for computing the gradients by automatic differentiation methods as will be detailed in Sect. 5.2.3.

5.2.2 Activation Functions

Several types of activation functions can be considered depending on the task at hand. We will briefly describe a few popular ones in the following subsections.

5.2.2.1 Linear Activation

The simplest activation function is the linear activation, which means that σ is simply the identity function:

$$\sigma(x) = x. \tag{5.3}$$

On a network with no hidden layers, a linear activation function between the input and output layers can be used to perform a linear regression between the input and output data. For networks with one or more hidden layers, stacking linear layers is not useful since a composition of linear activations is still linear. However, linear layers can be combined with other nonlinear activation functions. For example, linear layers can be used as the last layer to scale the output to arbitrary values. A non-trainable linear transformation is often used to *normalize* the input of a network to speed up the training (optimization) process, as will be detailed in Sect. 5.2.3.3.

5.2.2.2 Rectified Linear Units

One of the simplest nonlinear activation functions is the piece-wise linear rectified linear unit (ReLU) function, defined as

$$\sigma(x) = \max(0, x). \tag{5.4}$$

It can easily be seen that a single hidden layer with ReLU activation, followed by a linear activation layer, can approximate exactly piecewise linear functions in one dimension (Samaniego et al. 2020). Indeed, on a grid with nodes $x_0 < x_1 < \ldots < x_n$, the finite element linear hat function $N_i(x)$ can be written as

$$N_i(x) = \frac{1}{h_i} ReLU(x - x_{i-1}) - \left(\frac{1}{h_i} + \frac{1}{h_{i+1}}\right) ReLU(x - x_i) + \frac{1}{h_{i+1}} ReLU(x - x_{i+1}) \tag{5.5}$$

where $h_i = x_i - x_{i-1}$. This observation can be extended to higher dimensions, where two hidden layers are enough to approximate piecewise linear simplex elements in two and more dimensions (He et al. 2020). Further, error bounds for the approximation of ReLU networks in Sobolev norms are given, e.g. in Petersen and Voigtlaender (2018), Gühring et al. (2020).

5.2.2.3 Sigmoid

The sigmoid activation, also known as the logistic function, is defined as

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{5.6}$$

This function has a S-shaped form, as shown in Fig. 5.2b. The range of this function is the interval (0, 1), therefore it is often used in the output layer of neural networks used for binary classification tasks, where the output is a probability that the input

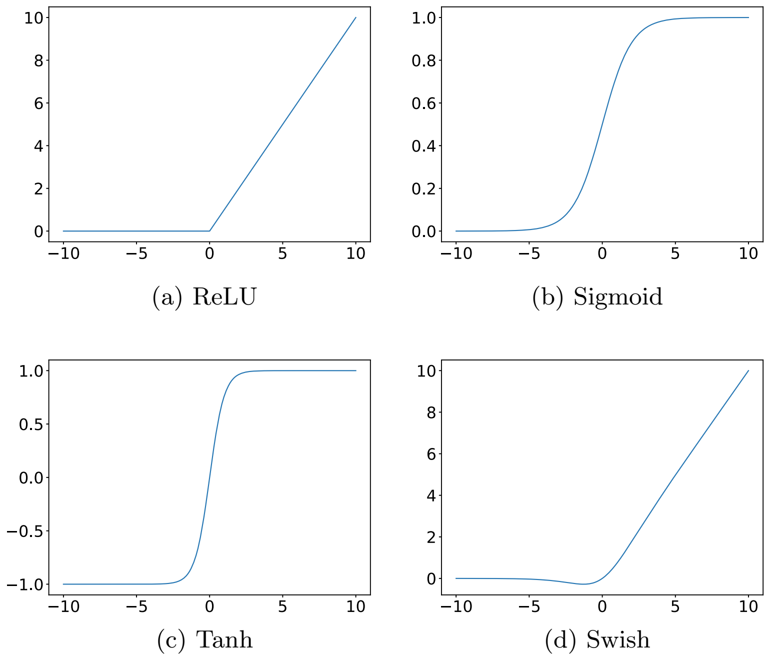


Fig. 5.2 Commonly used non-linear activation functions

belongs to a given class. The function is also differentiable infinitely many times, resulting in a smooth approximation which is desirable for many applications.

5.2.2.4 Hyperbolic Tangent

The hyperbolic tangent activation function is defined as

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}. \tag{5.7}$$

This function looks similar to the sigmoid activation, maintaining the overall S-shape and smoothness. An important difference is that the range of the outputs is $(-1, 1)$ which is centered at 0. This makes the tanh activation more suitable for deep networks without creating a bias toward positive outputs.

5.2.2.5 Swish

The swish activation function is defined as

$$\text{swish}(x) = \frac{x}{1 + \exp(-x)} = x \cdot \sigma(x), \tag{5.8}$$

where $\sigma(x)$ is the sigmoid activation. The plot of this function is shown in Fig. 5.2d. The swish function looks similar to the ReLU activation. However, like sigmoid and tanh, it is infinitely differentiable.

We note that there are several other activations that have been proposed which are similar to ReLU and swish, such as Leaky ReLU (Maas et al. 2013), exponential linear units (ELUs) (Clevert et al. 2015), Gaussian error linear units (GELUs) (Hendrycks and Gimpel 2016), Mish (Misra 2019), and others. These have been shown to remedy some of the drawbacks of the previously considered activation functions and provide a modest improvement on some machine learning tasks, particularly related to image-based classification and segmentation tasks (Li et al. 2021). However, from the point of view of function approximation where partial derivatives are involved, tanh or swish are also well-suited due to their smoothness properties.

5.2.2.6 Adaptive Activation Functions

In addition to the standard activation functions, which are fixed at each layer, the so-called *adaptive activations* have been proposed which depend on some model-dependent or trainable parameters. In particular, for a given activation function $\sigma(x)$, we can define the adaptive version by

$$\sigma_a(x) = \sigma(ax). \tag{5.9}$$

The idea of using trainable parameters in the activation function was proposed in Agostinelli et al. (2014), and further developed in the context of function and PDE solution approximation in Jagtap et al. (2020b, a, 2022), Shukla et al. (2020) among others. Some adaptive or trainable activation functions have a different form, for example, the original Swish activation proposed in Ramachandran et al. (2017) is of the form:

$$\sigma_\beta(x) = \frac{x}{1 + \exp(-\beta x)}, \tag{5.10}$$

where β is either a trainable or user-defined parameter. In some cases, using an adaptive activation can improve the results on classification tasks by a modest amount, usually an increase of 0.5–2% in the accuracy (Apicella et al. 2021). A similar improvement can be seen for function approximation, although the overall complexity of the architecture is increased.

5.2.3 Training

As mentioned earlier, the training process involves optimizing the network parameters (weights and biases) such that an objective function is minimized. Suppose the loss function is denoted by $\mathcal{L}(u_{NN}(\mathbf{x};\boldsymbol{\theta}))$, where u_{NN} is the neural network and $\boldsymbol{\theta}$ represents the trainable parameters, e.g. the matrices $\mathbf{W}_i$ and vectors $\mathbf{b}_i$ in (5.2). In the case of regression, a commonly used loss function is the *mean square error*, defined as

$$\mathcal{L}_{MSE}(u_{NN}(\mathbf{x}_j;\boldsymbol{\theta})) = \frac{1}{N}\sum_{j=1}^{N} |u_{NN}(\mathbf{x}_j) - \mathbf{y}_j|^2, \tag{5.11}$$

where $\mathbf{x}_j$, $j = 1, \ldots, N$ are input points at which the *ground truth* output values $\mathbf{y}_j$ are known. For the case of PDE approximations, more complicated loss functions which contain the partial derivatives of u_{NN} with respect to the inputs can be devised. Additional terms can be used to incorporate the governing equations and boundary conditions, as will be detailed in Sect. 5.3. Then the process of training a neural network can be described as

$$\text{Find } \boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(u_{NN}(\mathbf{x};\boldsymbol{\theta})). \tag{5.12}$$

We note that since $\mathcal{L}(u_{NN}(\mathbf{x};\boldsymbol{\theta})$ is usually based on the evaluation of u_{NN} or its derivatives at a finite number of points (called *training points*); therefore, $\boldsymbol{\theta}^*$ will depend in general on number and location of these points. A careful choice of selecting $\mathcal{L}$ and a proper weighting between its terms is therefore key to ensuring that the training is successful and the output generalizes well to new inputs.

5.2.3.1 Forward and Back Propagation

Finding the optimal weights is usually done by gradient-based methods, such as gradient descent. Parallelization and automatic differentiation methods are key ingredients in efficient implementations. The optimization method requires the gradients of a possibly large number of trainable parameters, with many networks containing tens of millions of parameters. Some are even larger, for example, the GPT-3 language model uses 175 billion parameters (Floridi and Chiriatti 2020). Therefore, reverse-mode differentiation, also known as back-propagation (Rumelhart et al. 1986), is commonly used to compute the gradients with respect to the trainable parameters.

The differentiation process involves a *forward pass*, during which the neural network output and the loss function are evaluated from a given input, and the operations involved are recorded in a graph. Then the derivatives are computed in reverse order of the evaluation, with the intermediate results obtained from the chain rule stored at the graph nodes (see e.g. Sect. 6.5 in Goodfellow et al. 2016 for details). The remarkable outcome of this procedure is that the partial derivatives of the loss function with respect to all the parameters can be evaluated at a cost that is proportional to the number of floating points operations involved in the forward evaluation.

Using forward mode differentiation, where the partial derivatives are computed in the order of evaluation, would result in a much higher cost that is also proportional to the number of parameters, although the memory requirements may be lower (López et al. 2021). In general, evaluating the partial derivatives (Jacobian) of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ requires $O(n)$ operations in forward mode, and $O(m)$ operations in reverse mode. In the context of PDEs, forward mode differentiation may be more efficient when computing the partial derivatives of the outputs with respect to the input coordinates, particularly for multiphysics models or other coupled problems where several solution fields are considered.

5.2.3.2 Network Initialization

When initializing the training process, particular care is needed for the selection of the initial value. For example, if all the weights and biases are set to zero, then the gradients with respect to the weights within a layer will have the same value. In a gradient descent update with a fixed step size, all the parameters will be updated by the same amount, resulting in the equivalent of a network with a single neuron per layer. Part of the recent success of deep neural networks in applications is owed to better techniques for initializing the values of the network parameters, such as Glorot (Xavier) (Glorot and Bengio 2010) and He et al. (2015) initialization.

While the initialization method can be seen as a hyperparameter which can be tuned according to the problem at hand, a commonly used one is *Glorot uniform*, where the weights are chosen from a uniform distribution $U[-l, l]$, where

$$l = \sqrt{\frac{6}{n_{in} + n_{out}}}, \tag{5.13}$$

with n_{in} and n_{out} being the number of input and output neurons for a given layer. The biases are initialized to zero. This is also the default initialization used in the TensorFlow deep learning framework.

5.2.3.3 Data Normalization

It can be observed that the nonlinear region of most activation functions $\sigma(x)$, such as the ones in Fig. 5.2, is centered in a small interval around $x = 0$. Therefore, if the input data is in a region far away from the origin, then the activation will be mostly constant or linear, which will hinder the performance of gradient descent methods (see also Sect. 5.2.4.2). To remedy this issue, it is essential to perform a normalization on the input data, which is just a linear transformation into the interval $[-1, 1]$. In particular, for each input neuron, the transformation is given by the formula:

$$T_{norm}(x) = \frac{2 \cdot (x - x_{min})}{x_{max} - x_{min}} - 1, \tag{5.14}$$

where x_{max} and x_{min} are the maximum and minimum input values, respectively. In the case where the input values are points in the computational domain, then x_{min} and x_{max} represent the *bounding box* of the domain. These values must be fixed for training and testing; otherwise, incorrect results will be obtained.

5.2.4 Testing and Validation

After a neural network is trained, it is expected to output useful results. However, in most cases, it is not feasible to train the network indefinitely or until the loss function stops decreasing (up to machine precision). Moreover, the number of training points and the number of layers and neurons must be correlated in the sense that, for optimal results, a larger number of parameters require a larger number of training points to avoid overfitting. The performance of the network is then measured by testing and validating the output.

In standard machine learning tasks, it is common to partition the available data into training/testing/validation subsets. The training data is used in the optimization procedure (5.12) for finding the optimal trainable parameters (weights and biases). The validation data is used to monitor the performance of the network by just evaluating the loss function. Tuning the network hyperparameters, such as the type of activation function, network size, and optimization algorithm may require some trial and error. Although the validation data is not used directly in the optimization process, it may indirectly create a bias in the process of hyperparameter tuning. Therefore, when the performance of the validation data is satisfactory, the network may be further validated using the test set. A typical split is to use 80% of the data for training and ca. 20% for testing and validation, although these ratios may vary depending on the problem at hand. For example, in the case of physics-informed neural networks, where training and testing data are just points in the domain, it may be useful to test the network by generating many more points from a higher resolution sample. We note that in most machine learning models, optimization is the most computationally intensive part. Therefore, the amount of training data is most closely related to the amount of random access memory (RAM) and numerical (floating point) operations required while testing (evaluating) the model is comparatively much cheaper.

In this section, we describe some of the pitfalls involved in training and testing a network, and the countermeasures that can be implemented.

5.2.4.1 Underfitting and Overfitting

Several types of approximation pathologies can be encountered in the process of training a neural network, among which underfitting and overfitting are some of the most common. Underfitting can occur when the neural network does not have enough approximation capability to satisfactorily fit the data or solve the problem at hand. It can also occur when the optimization has not converged, for example, because too few iterations have been performed, or because the learning rate is too low or too high. Underfitting can be typically identified when both the training and validation losses are higher than acceptable values.

Overfitting, on the other hand, can appear when the network capacity is larger than required. In this scenario, the training data is well approximated but other data points may be far off from the actual values, or in machine learning parlance, the model "does not generalize" well. A similar case where fitting exactly a small dataset does guarantee that the target function is well approximated occurs in interpolation by high-order polynomials, where the interpolant can oscillate wildly between the interpolation points. In this case, the training loss value decreases to a low value (even zero), while the validation loss can be much higher.

A good strategy to avoid overfitting or underfitting is to monitor both the training and validation losses and to stop the training when the testing loss begins to increase. To illustrate, the results for regression of the function $u(x) = \sin(\pi x)$ for $x \in [-1, 1]$ are shown in Fig. 5.3. A random uniform noise with magnitude in the interval (0, 0.1) was added to the training and validation data, which consists of 201 and 50 points, respectively. A neural network with two hidden layers consisting of 64 neurons has been used, together with the tanh activation function for the first two layers and linear

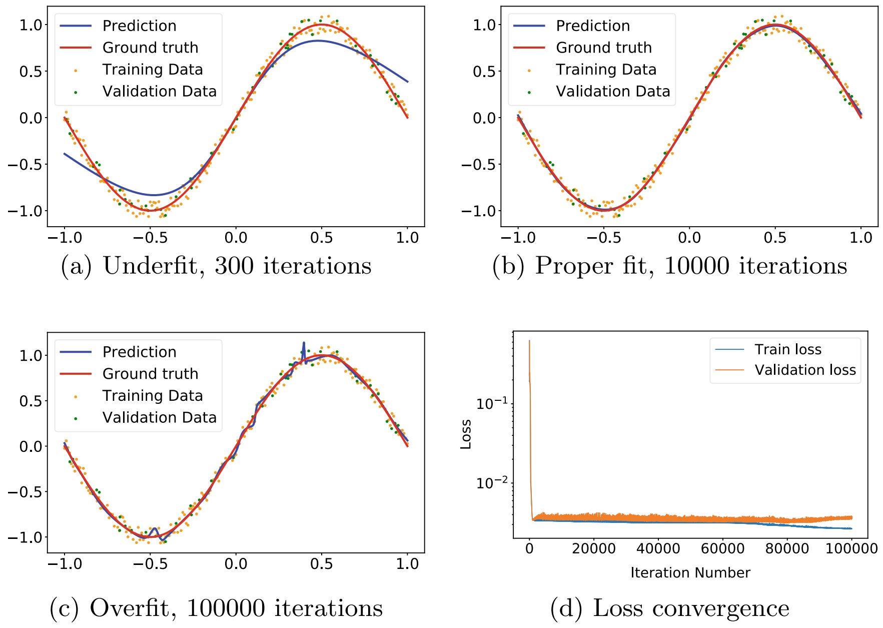


Fig. 5.3 Fitting a noisy function and the loss convergence history

activation in the last layer. The ADAM optimizer with the default parameters and learning rate of 0.001 is used to minimize the mean square error of the difference between the predicted and training values.

We observe from Fig. 5.3a that after 300 iterations, the neural network can start to approximate the sinusoidal function, but it is still quite far away from the actual shape (underfitting). The training loss value at this stage is 0.1129, while the validation loss is 0.1393. After 10000 iterations, the approximation is already quite good, with only a small error between the prediction and the actual function (without noise) as shown in Fig. 5.3b. Here, the training loss is 0.0033 and the validation loss is quite close at 0.0035. Next, if we continue to training, we start to observe that after many more iterations, the training and validation loss start to diverge (see Fig. 5.3d). After 100000 iterations, we notice that the predicted function has some oscillations and spikes as it tries to capture the noise in the data as shown in Fig. 5.3c. At this stage, the training loss is 0.0026 and the test loss is 0.0037.

5.2.4.2 Vanishing and Exploding Gradients

Two other types of problems encountered in training of artificial neural networks are those related to the magnitude of the gradients. The vanishing gradients phenomenon occurs when the derivative of the loss function with respect to the training variables is very small. This can mean that the objective function is very close to a stationary point, which can also be a saddle point or some other point far from the global minimum. The end result is very slow or no convergence of the loss function. A common remedy for this problem is to perform a *normalization* of the input data (see also Sect. 5.2.3.3). Otherwise, changing the network architecture or the activation function (for example using rectified activations like ReLU or Swish) may also be helpful, since S-shaped activations like sigmoid or hyperbolic tangent are particularly susceptible to vanishing gradients.

Exploding gradients, on the contrary, refer to the occurrence of too large derivatives of the loss function with respect to the trainable parameters. In extreme cases, the gradients can overflow, resulting in *not-a-number* (NaN) values for the loss. Another possible effect is unstable training, where the loss value oscillates without converging to the optimal value. Possible remedies for this problem include using a smaller learning rate, and adding residual (or skip) connections to the neural network (Philipp et al. 2018).

The ReLU activation function may suffer from a related problem known as “dying ReLU”, which occurs when some neurons become inactivated, in the sense that they always output zero for all the inputs. This can happen when a large negative bias value is learned for a particular neuron. Because the derivative of the constant zero function is also zero, it is not possible to recover a “dead” ReLU neuron, resulting in a diminished approximation capability.

5.2.5 Optimizers

We will now briefly describe the optimization algorithms commonly used to train (i.e. minimize the loss function) a neural network. First, we mention that two types of optimization strategies can be employed: full-batch training and mini-batch training. In the former, the entire data set is used during a forward pass through the network and the gradients with respect to all the data points are computed in one step. In mini-batch training, on the other hand, the training data is split into several subsets of (approximately) the same size called mini-batches. Then an optimization sub-step is taken with respect to each mini-batch. When the entire dataset is seen by the optimization algorithm once, then a training *epoch* is completed. In general, first-order optimization methods, like gradient descent, are commonly used with mini-batch training, while algorithms that make use of (approximations of) second derivative information use full-batch training. A detailed survey of optimization methods used in machine learning has been presented in Sun et al. (2019).

5.2.5.1 Stochastic Gradient Descent

The gradient descent method is the simplest gradient-based optimizer. The idea is to minimize the function in the direction of the gradient evaluated at the current guess by a fixed step size (also called the *learning rate*). If the objective function is $\mathcal{L}(\mathbf{w})$, then an optimization step can be written as

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^{(t)}), \tag{5.15}$$

where η is the learning rate. In the case of mini-batch training, since the mini-batches are typically randomly selected, the method is called *stochastic gradient descent* (SGD). Using mini-batches has been shown to improve the robustness, allowing the optimizer to find the global optima (or better local optima) even for non-convex problems (De Sa et al. 2015; Mertikopoulos et al. 2020).

5.2.5.2 Adaptive Momentum (ADAM)

This optimization method, proposed in Kingma and Ba (2014), replaces the fixed learning rate of the conventional SGD with a variable step-size based on the *momentum*, which can be seen as a linear combination of the gradients of the current and previous time steps.

An update of the ADAM optimizer from step t to step $t + 1$ has the form:

$$\mathbf{m}^{(t+1)} := \beta_1 \mathbf{m}^{(t)} + (1-\beta_1)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}) \quad \text{(biased first moment)} \tag{5.16}$$

$$\mathbf{v}^{(t+1)} := \beta_2 \mathbf{v}^{(t)} + (1-\beta_2)(\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}))^2 \quad \text{(biased second moment)} \tag{5.17}$$

$$\hat{\mathbf{m}} := \frac{\mathbf{m}^{(t+1)}}{1-\beta_1^{(t+1)}} \quad \text{(unbiased first moment)} \tag{5.18}$$

$$\hat{\mathbf{v}} := \frac{\mathbf{v}^{(t+1)}}{1-\beta_2^{(t+1)}} \quad \text{(unbiased second moment)} \tag{5.19}$$

$$\mathbf{w}^{(t+1)} := \mathbf{w}^{(t)} - \eta\frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}}+\epsilon} \quad \text{(weights update).} \tag{5.20}$$

Here, $\mathbf{m}$ and $\mathbf{v}$ are the moment vectors which are initialized to zeros, β_1, β_2, and ϵ are constants which are usually initialized to $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$, and η is the learning rate. β_1^t and β_2^t denote β_1 and β_2 to the power t, and $(\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}))^2$ denotes the element-wise squaring of the gradient vector. Because the momentum vectors are initialized to zeros, a bias-correction is introduced in (5.18) and (5.19). This technique can smooth out the oscillations in the gradients and usually improves the convergence compared to the standard SGD optimizer.

5.2.5.3 Quasi-Newton Methods

The gradient descent-based methods approximate the loss at each step by a linear function without taking into account the curvature information. Faster convergence can be obtained by using Newton algorithms, which involve computing the second derivatives. Nevertheless, for a large number of parameters, the cost of Newton's method in terms of memory storage and floating point operations can be prohibitive, since the Hessian matrix has size $n \times n$, where n is the number of parameters. A more feasible alternative is the family of quasi-Newton methods, like the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm (Broyden 1970; Fletcher 1970; Goldfarb 1970; Shanno 1970) or the limited memory version L-BFGS (Liu and Nocedal 1989), which are already implemented in machine learning frameworks like PyTorch or TensorFlow Probability (Dillon et al. 2017). Another algorithm that can be used for problems with a small number of parameters is the Levenberg–Marquardt algorithm (Levenberg 1944; Marquardt 1963), which can be seen as a combination of the Gauss–Newton method and gradient descent.

5.3 Physics-Informed Neural Networks

In the following, we focus on the physics-informed neural network, by which is meant an artificial neural network incorporates the residuals of the PDE to be solved into the loss function. In most cases, a simple, fully connected feed-forward network is used; however, some important differences can be noted in the form of the objective function, in particular regarding to whether the strong or weak form of the PDE is used.

5.3.1 Collocation Method

The classical PINNs are collocation-based, meaning that the neural network aims to approximate the strong form of the governing equation at a set of *collocation points*. Because the collocation points can be randomly distributed inside the domain and no mesh is needed, this method belongs to the category of mesh-free methods. Moreover, once the "building blocks" for constructing the neural network and evaluating the partial derivatives with respect to the inputs are obtained, the implementation is relatively simple.

In particular, suppose that the governing PDE is of the form:

$$\mathcal{F}(\mathbf{u}(\mathbf{x}), \frac{\partial \mathbf{u}(\mathbf{x})}{\partial x_1}, \ldots) = \mathbf{0} \qquad \text{for } \mathbf{x} \in \Omega \tag{5.21}$$

$$\mathcal{G}(\mathbf{u}(\mathbf{x}), \frac{\partial \mathbf{u}(\mathbf{x})}{\partial \mathbf{n}}, \ldots) = \mathbf{0} \qquad \text{for } \mathbf{x} \in \Gamma, \tag{5.22}$$

where $\mathcal{F}$ represents a differential operator for the domain interior, $\mathcal{G}$ is a differential operator for the boundary conditions, $\mathbf{u}$ is the unknown function, Ω and Γ are the computational domain and its boundary, and $\mathbf{n}$ is the outer normal vector to the boundary. The interior differential operator may contain any order of derivatives with respect to the inputs, while the boundary operator may contain any order of derivative with respect to the outer normal vector for Neumann-type boundary conditions.

The loss function for a neural network $\mathbf{u}_{NN}(\mathbf{x}; \boldsymbol{\theta})$ with trainable parameters $\boldsymbol{\theta}$ (which include the weights and biases for each layer) can be constructed based on the "mean square error" (MSE) evaluated at a set of N_{int} interior collocation points $\{\mathbf{x}_i^{int}\}$, $i = 1, \ldots, N_{int}$ and a set of N_{bnd} boundary collocation points $\{\mathbf{x}_j^{bnd}\}$, $j = 1, \ldots, N_{bnd}$ as

$$\begin{aligned}\mathcal{L}_{coll}(\boldsymbol{\theta}) = \frac{\lambda_1}{N_{int}} \sum_{i=1}^{N_{int}} \mathcal{F}(\mathbf{u}_{NN}(\mathbf{x}_i^{int}; \boldsymbol{\theta}), \frac{\partial \mathbf{u}_{NN}(\mathbf{x}_i^{int}; \boldsymbol{\theta})}{\partial x_1}, \ldots)^2 \\ + \frac{\lambda_2}{N_{bnd}} \sum_{j=1}^{N_{bnd}} \mathcal{G}(\mathbf{u}_{NN}(\mathbf{x}_j^{bnd}; \boldsymbol{\theta}), \frac{\partial \mathbf{u}_{NN}(\mathbf{x}_j^{bnd}; \boldsymbol{\theta})}{\partial \mathbf{n}}, \ldots)^2.\end{aligned} \tag{5.23}$$

Here λ_1 and λ_2 are weight terms; usually, choosing $\lambda_2 >> \lambda_1$ helps to speed up convergence by ensuring that the boundary conditions are satisfied. Adaptive methods for choosing the weights have also been proposed in Wang et al. (2022). In the case of time-dependent problems, the classical PINNs use a space–time discretization, where the time is considered as an additional dimension.

5.3.2 Energy Minimization Method

In the energy minimization method, we seek to minimize an energy functional, which is usually based on the weak (variational) form of the PDE. In many scientific modeling tasks, an energy functional appears naturally from the physical laws involved (for example, the principle of minimum potential energy in structural mechanics). Suppose the functional to minimize is denoted by $\mathcal{J}(\mathbf{u})$, which can be decomposed into an interior term and a boundary term:

$$\mathcal{J}(\mathbf{u}) = \int_{\Omega} \mathcal{H}_{int}(\mathbf{u})\, d\Omega + \int_{\Gamma} \mathcal{H}_{bnd}(\mathbf{u})\, d\Gamma, \tag{5.24}$$

with Γ denoting the portion of the boundary over which the boundary term is evaluated. Then we can define the loss function of the form:

$$\mathcal{L}_{energy}(\boldsymbol{\theta}) = \int_{\Omega} \mathcal{H}_{int}(\mathbf{u}_{NN})\, d\Omega + \int_{\Gamma} \mathcal{H}_{bnd}(\mathbf{u}_{NN})\, d\Gamma. \tag{5.25}$$

The integrals in (5.25) are usually approximated by numerical integration, using a finite set of quadrature points $\{\mathbf{q}_i^{int}\}$, and weights $\{w_i^{int}\}$ with $i = 1, \ldots, Q_{int}$ for the interior integral and quadrature points $\{\mathbf{q}_j^{bnd}\}$ and weights $\{w_j^{bnd}\}$ with $j = 1, \ldots, Q_{bnd}$ for the boundary integral, i.e.

$$\mathcal{L}_{energy}(\boldsymbol{\theta}) \approx \sum_{i=1}^{Q_{int}} \mathcal{H}_{int}(\mathbf{u}_{NN}(\mathbf{q}_i^{int}))w_i^{int} + \sum_{j=1}^{Q_{bnd}} \mathcal{H}_{bnd}(\mathbf{u}_{NN}(\mathbf{q}_j^{bnd}))w_j^{bnd}. \tag{5.26}$$

When additional constraints are needed, such as Dirichlet boundary conditions, additional terms can be added to the loss function, similarly to (5.23). Alternatively, one can impose the Dirichlet boundary conditions *strongly* (i.e. exactly) by modifying the output of the neural network to match the prescribed boundary data. In particular, we consider the computed solution to be $\tilde{\mathbf{u}}_{NN}$, satisfying $\tilde{\mathbf{u}}_{NN}(\mathbf{x}) = \mathbf{u}_D(\mathbf{x})$ for $\mathbf{x} \in \Gamma_D$, where $\mathbf{u}_D$ is the Dirichlet boundary condition specified on the boundary Γ_D to be of the form:

$$\tilde{\mathbf{u}}_{NN}(\mathbf{x}) = \mathbf{g}(\mathbf{x}) + \mathbf{d}(\mathbf{x})\mathbf{u}_{NN}(\mathbf{x}), \tag{5.27}$$

where $\mathbf{g}$ is a smooth extension of $\mathbf{u}_D$ such that $\mathbf{g}(\mathbf{x}) = \mathbf{u}_D(\mathbf{x})$ for $\mathbf{x} \in \Gamma_D$ and d is a distance function such as $d(\mathbf{x}) = 0$ for $\mathbf{x} \in \Gamma_D$ and $d(\mathbf{x}) > 0$ otherwise. When $\mathbf{u}_{NN}$ is vector-valued, then we multiply each of its components by $d(\mathbf{x})$. This ensures that the output $\tilde{\mathbf{u}}_{NN}$ satisfies exactly the boundary conditions (i.e. $\tilde{\mathbf{u}}_{NN}(\mathbf{x}) = \mathbf{u}_D(\mathbf{x})$ for $\mathbf{x} \in \Gamma_D$), although the choice of $\mathbf{g}(\mathbf{x})$ and $d(\mathbf{x})$ requires some care (Sukumar and Srivastava 2022).

In general, the energy minimization method tends to be less computationally demanding than the collocation-based PINNs, due to the fact that, e.g. only the first-order derivatives need to be computed for solving a second-order problem. However, it requires an integration mesh and it is more difficult to verify that the solution is correct within a certain tolerance, since the objective function should converge to some non-zero minimum which is not known in advance. A possible approach to overcome this problem is to compute the residual loss for validation, which can then also be used to adaptively adjust the number of integration points, as proposed in Goswami et al. (2020).

5.4 Numerical Applications

By using a small set of training or input data (e.g. initial and boundary conditions and/or measured data) as well as governing physical laws, PINNs attempt to approximate the solution of the problem. Complex nonlinear systems and phenomena in physics and engineering are described by differential equations.

PINNs have shown their capabilities to solve both forward and inverse problems in science and engineering. A forward problem can be defined as a problem of finding a particular effect of a given cause utilizing a physical or mathematical model, whereas an inverse problem refers to finding causes from the given effects (Vauhkonen et al. 2016). We can investigate the 1D steady-state heat equation with the source term to give more concrete examples of forward and inverse problems.

Let us consider a rod with unit length along the x-axis and the heat flowing through this rod with a heat source as our model. We can represent the temperature at location x on the rod as $T(x)$. Under certain assumptions, such as the rod being perfectly insulated, with the source term $q(x)$ being known, then the governing equation can be written as

$$\kappa \frac{d^2T}{dx^2} + q(x) = 0 \tag{5.28}$$

where $\kappa > 0$ is the thermal diffusivity constant. Finding temperature at any location on the rod is a forward problem. On the other hand, finding the constant κ, which is a rod feature, from observed temperature data is a good example of an inverse problem. These examples will be detailed in Sects. 5.4.1 and 5.4.2.

To summarize, the aforementioned procedures explained in the previous sections to solve differential equations with PINNs will become tangible with numerical applications in this section. The solution estimation of PINNs for both forward and inverse problems will be discussed by providing simple and complex examples.

5.4.1 Forward Problems

In the introductory part of this section, the definition of a forward problem is given as finding the particular effect of a given cause using a physical or mathematical model.

5.4.1.1 1D Steady-State Heat Equation

Let us remember the 1D steady-state heat conduction problem with a heat source. As we discussed before, the governing equation for this example is given in (5.28). Let the thermal diffusivity constant be $\kappa = 0.5$ and x denote the location on the rod. Here, the source term is given as $q(x) = 15x - 2$. We assume that the temperatures at both ends are 0. Then we can re-write (5.28) as

$$\begin{aligned} \frac{d^2T}{dx^2} + \frac{q(x)}{\kappa} &= 0, \ \ x \in [0, 1] \\ q(x) &= 15x - 2 \\ \kappa &= 0.5 \\ T(0) = T(1) &= 0 \end{aligned} \tag{5.29}$$

The first step to solve this problem is to discretize the domain with uniform or randomly sampled collocation points. Then the neural network will process these collocation points through its linearly connected layers consisting of neurons with nonlinear activation functions. Of course, the outcome of the first forward propagation will not be compatible with the true solution. Therefore, at this point, the physics and boundary knowledge will guide the neural network to approximate the ground truth by updating the weights and biases of the neural network. Let us elaborate on this step by step and reinforce these steps with code snippets. Note that these codes are written with TensorFlow version 2.x with the Keras API.

We first generate 100 equidistant points in our domain. Here, the choice of the number of points is up to the user. However, it should be noted that the number of points also has some influence on the number of iterations or network size required to have results with similar accuracy. The ADAM optimizer with a learning rate of 0.005 is used for this example. An input layer, three hidden layers with 32 neurons equipped with tanh activation function, and an output layer form the neural network (see Fig. 5.4). The input and output layers have one neuron each since the input for the network is only one spatial dimension, and the output is the temperature at these points. By setting the number of iterations to 1000 and introducing the boundary condition data in TensorFlow tensors, we complete the initial settings of our model (see Listing 1).

Listing 1: Initial settings for the heat equation

```
# We set seeds initially. This feature controls the randomization of
# variables (e.g. initial weights of the network).
# By doing it so, we can reproduce same results.
tf.random.set_seed(123)
# 100 equidistant points in the domain are created
x = tf.linspace(0.0, 1.0, 100)
# boundary conditions T(0)=T(1)=0 and \kappa are introduced.
bcs_x = [0.0, 1.0]
bcs_T = [0.0, 0.0]
```

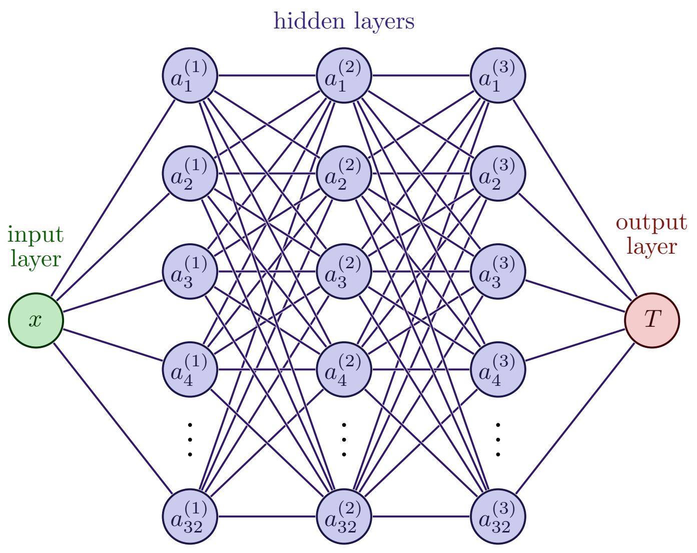


Fig. 5.4 The architecture of the feed-forward neural network for 1D steady-state heat conduction problem. The network consists of one input layer, one output layer, and three hidden layers with 32 neurons each. *a* is the activation function. Superscripted numbers denote the layer number, and subscripted ones denote the neuron number in the relevant layer

```
bcs_x_tensor = tf.convert_to_tensor(bcs_x)[:, None]
bcs_T_tensor = tf.convert_to_tensor(bcs_T)[:, None]
kappa = 0.5
# Number of iterations
N = 1000
# ADAM optimizer with learning rate of 0.005
optim = tf.keras.optimizers.Adam(learning_rate=0.005)

# Function for creating the model
def buildModel(num_hidden_layers, num_neurons_per_layer):
    tf.keras.backend.set_floatx("float32")
    # Initialize a feedforward neural network
    model = tf.keras.Sequential()

    # Input is one dimensional ( one spatial dimension)
    model.add(tf.keras.Input(1))

    # Append hidden layers
    for _ in range(num_hidden_layers):
        model.add(
            tf.keras.layers.Dense(
                num_neurons_per_layer,
                activation=tf.keras.activations.get("tanh"),
                kernel_initializer="glorot_normal",
            )
        )

    # Output is one-dimensional
    model.add(tf.keras.layers.Dense(1))

    return model


# determine the model size (3 hidden layers with 32 neurons each)
model = buildModel(3, 32)
```

Then we define our loss function, which is composed of two parts, the boundary loss, and physics loss, as formulated in (5.30). Here, the loss term tells us how far away our model is from 'reality'. For the measure of these loss terms, we will use mean square error formulation, which is mentioned in Sects. 5.2.3 and (5.11).

$$\mathcal{L}_{Loss} = \mathcal{L}_{BCs} + \mathcal{L}_{Physics} \tag{5.30}$$

Constructing the boundary loss is easier compared to the physics loss. Our model's assumptions should be compatible with the prescribed boundary conditions, which are $T(0) = 0$ and $T(1) = 0$ for our case. Thus, our goal should be to minimize the mean square error between our model's temperature prediction at both ends of the rod and the real temperature values at these points, which must be 0. The boundary condition loss is given by (5.31)

$$\mathcal{L}_{BCs} = \frac{\lambda_1}{N_B} \sum_{j=1}^{N_B=2} |T_{NN}(\mathbf{x}_j) - \mathbf{y}_j|^2, \tag{5.31}$$

where $N_B = 2$ since we have boundary condition data for two points which are $T(0) = T(1) = 0$. The regularization term λ_1 is taken as 1.

We also need to provide information about the interior points to get reasonable results. Although we do not know the temperature data for intermediate points on the rod, we know those points have to satisfy some physical laws that we derived in (5.28). Or in other words, our temperature prediction needs to satisfy (5.28). When we take the derivative of the temperature prediction of the network with respect to x two times and sum this result with the source term $q(x)$ divided by κ, this summation must yield 0. Thus, the physics loss for our example becomes

$$\mathcal{L}_{Physics} = \frac{\lambda_2}{N_P} \sum_{j=1}^{N_P=100} | \left. \frac{d^2 T_{NN}}{dx^2} \right|_{x=x_j} + \frac{q(x_j)}{\kappa} |^2, \tag{5.32}$$

Again, the regularization term λ_2 is taken as 1. Now we can combine the boundary conditions loss and physics loss functions to form our model's loss function (see (5.30)), which will guide the model to make better predictions in each iteration.

Listing 2: Loss function definition for the heat equation

```
# Boundary loss function
def boundary_loss(bcs_x_tensor, bcs_T_tensor):
    predicted_bcs = model(bcs_x_tensor)
    mse_bcs = tf.reduce_mean(tf.square(predicted_bcs - bcs_T_tensor))
    return mse_bcs

# the first derivative of the prediction
def get_first_deriv(x):
    with tf.GradientTape() as tape:
        tape.watch(x)
        T = model(x)
    T_x = tape.gradient(T, x)
    return T_x

# the second derivative of the prediction
def second_deriv(x):
    with tf.GradientTape() as tape:
        tape.watch(x)
        T_x = get_first_deriv(x)
    T_xx = tape.gradient(T_x, x)
    return T_xx

# Source term divided by \kappa
source_func = lambda x: (15 * x - 2) / kappa

# Function for physics loss
def physics_loss(x):
    predicted_Txx = second_deriv(x)
    mse_phys = tf.reduce_mean(tf.square(predicted_Txx + source_func(x)))
    return mse_phys

# Overall loss function
def loss_func(x, bcs_x_tensor, bcs_T_tensor):
    bcs_loss = boundary_loss(bcs_x_tensor, bcs_T_tensor)
    phys_loss = physics_loss(x)
    loss = bcs_loss + phys_loss
    return loss
```

TensorFlow records the operations on the trainable variables when computing the loss function and calculates the gradients by backpropagation. Then, the ADAM optimizer with a fixed learning rate of 0.005 minimizes the loss function. By performing one forward and one back propagation over the whole data set, one epoch is completed. This procedure is repeated the specified number of times, which is 1000 for our example.

Listing 3: Training of the heat equation model

```
# taking gradients of the loss function
def get_grad():
    with tf.GradientTape() as tape:
        # This tape is for derivatives with
        # respect to trainable variables
        tape.watch(model.trainable_variables)
        Loss = loss_func(x, bcs_x_tensor, bcs_T_tensor)
    g = tape.gradient(Loss, model.trainable_variables)
    return Loss, g


# optimizing and updating the weights of the model by using gradients
def train_step():
    # Compute current loss and gradient w.r.t. parameters
    loss, grad_theta = get_grad()
    # Perform gradient descent step
    optim.apply_gradients(zip(grad_theta, model.trainable_variables))
    return loss


# Training loop
for i in range(N + 1):
    loss = train_step()
    # printing loss amount in each 100 epoch
    if i
        print("Epoch {:05d}: loss = {:10.8e}".format(i, loss))
```

Once the training process is completed with the desired loss value, we can validate the output by performing one forward pass with a test dataset which is typically formed in the same domain as the training dataset. In our example, the training data was 100 equidistant points between 0 and 1. We can determine our test dataset as 200 equidistant points in the same domain. Figure 5.5 depicts that the model's prediction captures the analytical result.

5.4.1.2 2D Linear Elasticity Example

So far, we have seen the most straightforward application of PINNs to solve a 1D forward problem. The problem has been described with a linear second-order nonhomogeneous ordinary differential equation with Dirichlet boundary conditions. Then the equation was solved with a neural network with three hidden layers. This pedagogical example is supposed to help readers to understand the concept of using

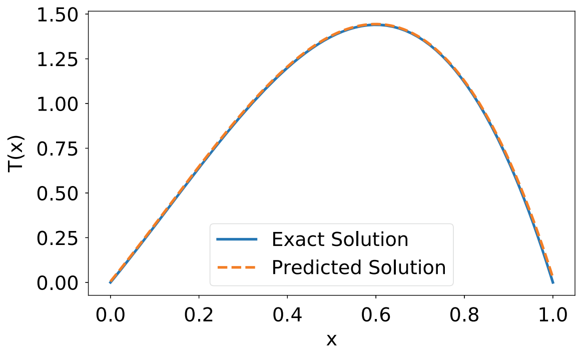


Fig. 5.5 Exact solution and the prediction of the model. The predicted solution coincides with the ground truth which is $T(x) = -5x^3 + 2x^2 + 3x$

neural networks in scientific problems. Now, let us proceed with a more complex application.

Therefore, consider the cantilever beam model (Wang et al. 2006; Otero and Ponta 2010; Wang and Feng 2009), a classical example in linear elasticity theory. This problem is governed by the well-known equilibrium equation (5.33) given as

$$-\nabla \cdot \boldsymbol{\sigma}(\mathbf{x}) = f(\mathbf{x}) \quad \text{for} \quad x \in \Omega \tag{5.33}$$

with the strain-displacement given by:

$$\boldsymbol{\epsilon} = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T) \tag{5.34}$$

and Hooke's law for a linear isotropic elastic solid:

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\epsilon} + \lambda(\nabla \cdot \mathbf{u})\mathbf{I}, \tag{5.35}$$

where μ and λ are the Lamé constants, and $\mathbf{I}$ is the identity tensor. The Dirichlet boundary conditions are $u(\mathbf{x}) = \hat{u}$ for $\mathbf{x} \in \Gamma_D$ and the Neumann boundary conditions are $\sigma\mathbf{n} = \hat{t}$ for $\mathbf{x} \in \Gamma_N$, where $\mathbf{n}$ is the normal vector.

For this example (see Fig. 5.6), Ω is a rectangle with corners at (0, 0) and (8, 2). Letting $\mathbf{x} = (x, y)$, and $\mathbf{u} = (u, v)$ the Dirichlet boundary conditions for $x = 0$ are

$$\begin{aligned} u(x, y) &= \frac{Py}{6EI}\left[(2+\nu)(y^2 - \frac{W^2}{4})\right] \\ v(x, y) &= -\frac{P}{6EI}(3\nu y^2 L) \end{aligned} \tag{5.36}$$

Commonly, a parabolic traction at $x = 8$

$$p(x, y) = P\frac{y^2 - yW}{2I} \tag{5.37}$$

is applied where $P = 2$ is the maximum traction, $E = 10^3$ is Young's modulus, $\nu = 0.25$ is the Poisson ratio and $I = b\frac{W^3}{12}$ is second moment of area of the cross section. The dimensions of the beam in x, y and z directions are $L = 8$, $W = 2$ and $b = 1$, respectively (Fig. 5.6).

The objective of this problem is to find the displacements on the beam in x and y directions. In order to solve the problem, firstly, uniform collocation points in the domain are created. The numbers of collocation points are 80 and 40 in x and y directions, respectively, as shown in Fig. 5.7. The prediction of the neural network shall satisfy the equilibrium equation (5.33) and the constitutive law (5.35) as well as the boundary conditions (5.36).

The strong imposition of the Dirichlet boundary conditions is explained in detail in Sect. 5.3.2. In this example, we strongly imposed Dirichlet boundary conditions

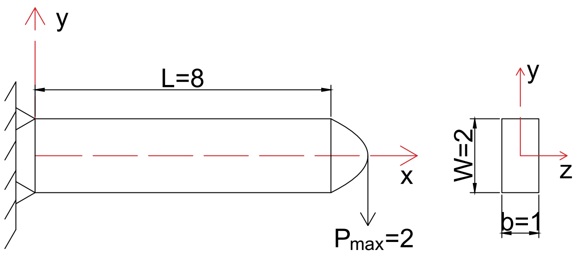


Fig. 5.6 The illustration of 2D elasticity problem

at $x = 0$. Therefore, there is no need to insert collocation points along the y-axis where $x = 0$ as is illustrated in Fig. 5.7.

In this example, a fully connected feed-forward neural network with three hidden layers and with 20 neurons per hidden layer is used with the *swish* activation function. The neural network constructed for this problem is depicted in Fig. 5.8. ADAM and L-BFGS optimizers are used together. ADAM optimizer was used for the first 15000 iterations, and then optimization of the parameters continued with L-BFGS optimizer for the successive 500 iterations.

After completing the training procedure, the model is tested with a new set of data. For the test data, the number of uniformly spaced collocation points was doubled in the same domain. One forward pass is performed to see the results for the test data. The solution obtained with the model and the exact solution for the displacements of the beam in x and y directions are plotted in Fig. 5.9.

The approximation obtained by the neural network is very close to the analytical solution. The relative L_2 error in the approximation is 5.899×10^{-5}. The error distribution can be seen in Fig. 5.10. The error for the displacements in x and y directions are of the order of 10^{-6} and 10^{-5}, respectively. The model makes less accurate predictions for the displacements around the beam's free end than the fixed end.

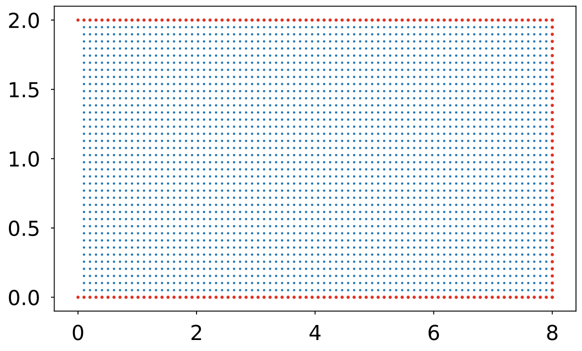


Fig. 5.7 Collocation points on the Timoshenko cantilever beam. 80 points in x direction and 40 points in y direction. Red points stand for boundary points whereas the blue points represent interior collocation points. Since Dirichlet boundary conditions are strongly imposed, the collocation points along the y-axis where $x = 0$ are not needed

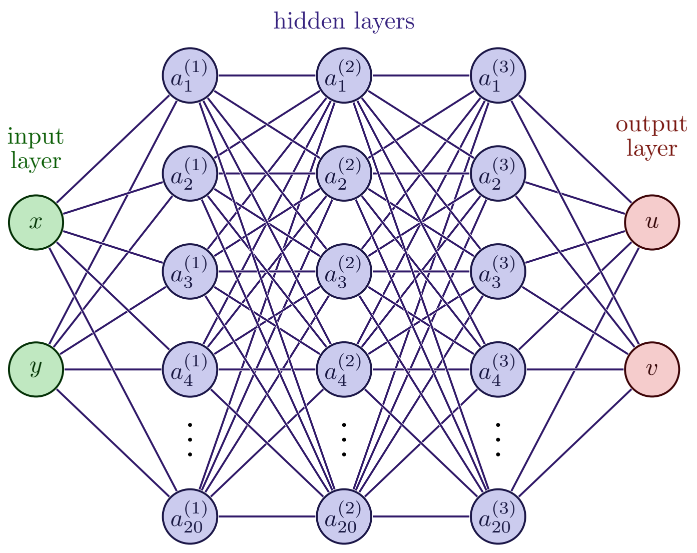


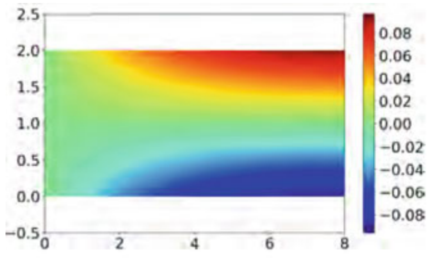
Fig. 5.8 The architecture of the feed-forward neural network for the Timoshenko beam problem. The network consists of one input layer, one output layer, and three hidden layers. There are 20 neurons per hidden layer. Two neurons in the input layer take x and y coordinates, and the output neurons give displacements in u and v directions. a is the activation function that is swish in this example. Superscripted numbers denote the layer number, and subscripted ones denote the neuron number in the relevant layer

5.4.1.3 3D Hyperelasticity

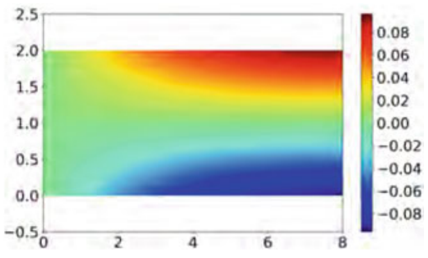
As the last example of this section, a hyperelasticity problem presented in Samaniego et al. (2020) will be discussed. We will solve this particular problem with the deep energy method shown in Sect. 5.3.2. Objective of the problem is obtaining the displacements for a 3D hyperelastic cuboid made of an isotropic, homogeneous material subjected to prescribed twisting, body forces, and traction forces. In order to obtain optimal parameters of the neural network, the potential energy formulation for the body will be used as the loss function. The governing equations and boundary conditions for this problem are written as

$$\begin{aligned} \nabla \cdot \boldsymbol{P} + \mathbf{f}_b &= 0, \\ \text{Dirichlet boundary} : \mathbf{u} = \bar{\mathbf{u}} \quad &\text{on} \quad \partial\Omega_D, \\ \text{Neumann boundary} : \boldsymbol{P} \cdot \mathbf{n} = \bar{\mathbf{t}} \quad &\text{on} \quad \partial\Omega_N, \end{aligned} \tag{5.38}$$

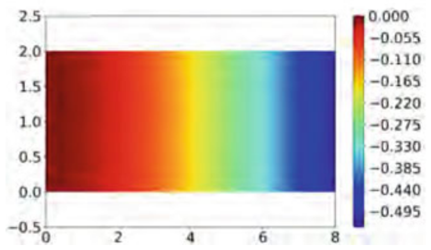
where $\bar{\mathbf{u}}$ is the prescribed displacement given on the Dirichlet boundary and $\bar{\mathbf{t}}$ is the prescribed traction at the Neumann boundary; $\mathbf{n}$ denotes the outward unit normal vector, $\boldsymbol{P}$ is the 1st Piola Kirchoff stress tensor and $\mathbf{f}_b$ is the body force. The potential energy functional of this problem is given by Samaniego et al. (2020)



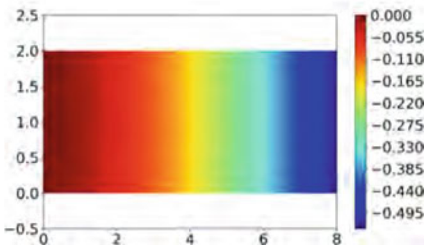
(a) Predicted displacements in x-direction



(b) Exact displacements in x-direction

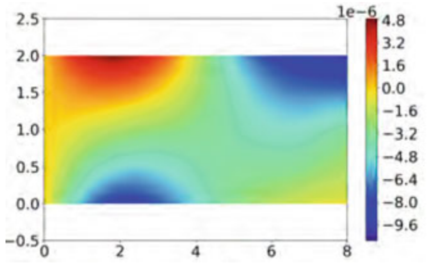


(c) Predicted displacements in y-direction

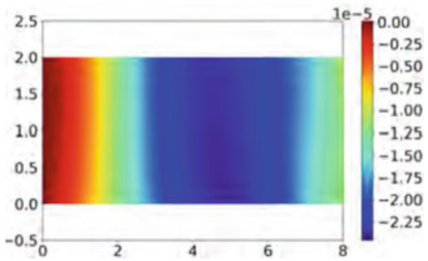


(d) Exact displacements in y-direction

Fig. 5.9 Predicted and exact values for displacements on a Timoshenko cantilever beam in x and y directions



(a) Estimation error for displacements in x-direction



(b) Estimation error for displacements in y-direction

Fig. 5.10 The difference between the exact solution and the predicted solution for displacements on the beam in x and y directions

$$\varepsilon(\varphi) = \int_{\Omega} \Psi dV - \int_{\Omega} \mathbf{f}_b \cdot \varphi dV - \int_{\partial\Omega_N} \bar{\mathbf{t}} \cdot \varphi dA, \tag{5.39}$$

where Ψ is the strain energy density and φ indicates the mapping of points on the body from the initial/undeformed to the deformed state.

In order to obtain optimal parameters of the neural network, the potential energy (5.39) is parameterized by the neural network's prediction for the displacements.

Thus, the loss function reads

$$\mathcal{L}(p) = \int_{\Omega} \Psi(\varphi(X; p))dV - \int_{\Omega} \mathbf{f}_b \cdot \varphi(X; p)dV - \int_{\partial\Omega_N} \bar{\mathbf{t}} \cdot \varphi(X; p)dA, \quad (5.40)$$

If we rewrite (5.40) in a discrete form, it becomes

$$\mathcal{L}(p) \approx \frac{V_\Omega}{N_\Omega} \sum_{i=1}^{N_\Omega} \Psi((\varphi_p)_i) - \frac{V_\Omega}{N_\Omega} \sum_{i=1}^{N_\Omega} (\mathbf{f}_b)_i \cdot (\varphi_p)_i - \frac{A_\partial \Omega_N}{N_\partial \Omega_N} \sum_{i=1}^{N_\partial \Omega_N} \bar{t}_i \cdot (\varphi_p)_i, \quad (5.41)$$

in which V_Ω is the volume and N_Ω is the number of data points within the solid; $N_\partial \Omega_N$ and $A_\partial \Omega_N$ denote the number of points on the surface subjected to the force and the surface area, respectively.

Let us consider now 3D cuboid of length $L = 1.25$, width $W = 1.0$, and depth $H = 1.0$. It is fixed at the left surface and twisted 60° counter-clockwise by boundary conditions $u_{|\Gamma_1}$ at the right-end surface. Also, at the lateral surfaces, a body force $\mathbf{f}_b = [0, -0.5, 0]^T$ and traction forces $\bar{\mathbf{t}} = [1, 0, 0]^T$ are applied(see Fig. 5.11).

The Dirichlet boundary conditions for this particular problem are

$$\begin{aligned} u_{|\Gamma_0} &= [0, 0, 0]^T, \\ u_{|\Gamma_1} &= \begin{bmatrix} 0 \\ 0.5[0.5 + (X_2 - 0.5)\cos(\pi/3) - (X_3 - 0.5)\sin(\pi/3) - X_2] \\ 0.5[0.5 + (X_2 - 0.5)\sin(\pi/3) + (X_3 - 0.5)\cos(\pi/3) - X_3] \end{bmatrix} \end{aligned} \quad (5.42)$$

The Neo-Hookean model is assumed in this problem. The material properties are shown in Table 5.1.

We now proceed with determining the network parameters. In each direction, 40 equally spaced points, 64000 points in total, are placed over the whole domain (see Fig. 5.12a). The neural network consists of three hidden layers, and each hidden

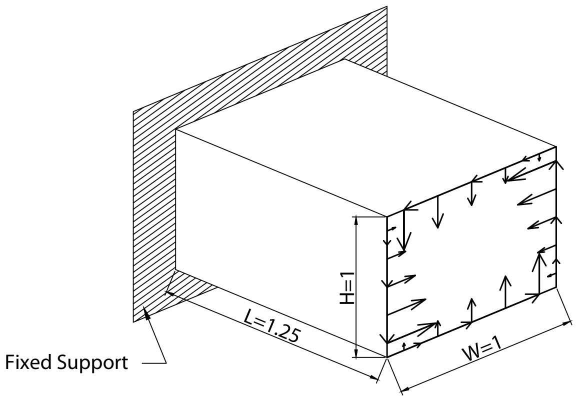
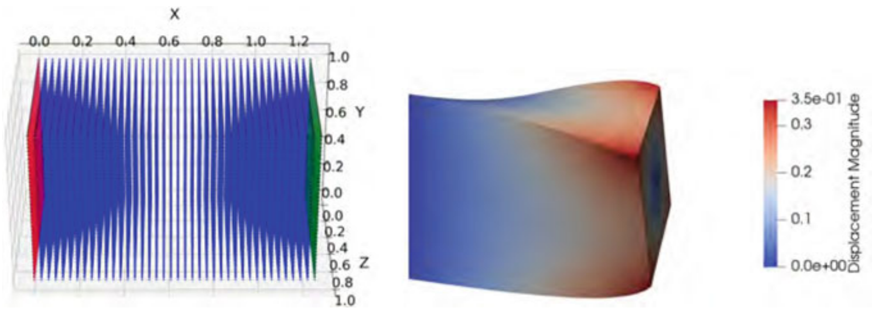


Fig. 5.11 The 3D hyperelastic cuboid is fixed at the left-hand side and twisted 60° counter-clockwise

Table 5.1 Material properties and parameters for hyperelastic cuboid

Description	Value
E—Young's modulus	10^6
ν—Poisson ratio	0.3
μ—Lame' parameter	$\dfrac{E}{2(1+\nu)}$
λ—Lame' parameter	$\dfrac{E\nu}{(1+\nu)(1-2\nu)}$



(a) 64000 equidistant points over the domain

(b) Deformed shape of 3D hyperelastic cuboid

Fig. 5.12 Training points on the cuboid and its predicted deformed shape after training

layer has 30 neurons with a *tanh* activation function. The input and output layers have three neurons corresponding to coordinates of the initial configuration of the designated points and their deformed coordinates after loading, respectively. The network is trained with 50 iterations and the parameters are optimized by the L-BFGS optimizer.

The predicted deformed shape of the cuboid is given in Fig. 5.12b. A line passing through two points on the cube $A(0.625, 1, 0.5)$ and $B(0.625, 0, 0.5)$ is drawn to compare the displacement predictions and the real displacements on the line. We showed in Samaniego et al. (2020), that a neural network with the same setup but trained with 25 steps has an error in the L_2 norm of 0.13210, whereas the finite element model has an error of 0.13275 for estimating the displacements on the line AB.

5.4.2 Inverse Problems

An inverse problem can be considered as inferring features of a model from the observed data. Finding the elasticity modulus of a beam from its displacement measurements under certain constraints or inferring the space-dependent reaction rate of a diffusion-reaction system (Yu et al. 2022) can be given as examples for the inverse problems. PINNs have already been used to tackle several inverse problems existing in unsaturated groundwater flow (Depina et al. 2022), nano-optics, and metamaterials (Chen et al. 2020). We will discuss two inverse problems in this section to demonstrate the application of PINNs to solve these problems.

5.4.2.1 Inverse Heat Equation

In Sect. 5.4.1, we examined a 1D steady-state heat equation with a source term. The goal was to find the temperature values along the rod with boundary data and governing physical laws. Now, let us reconsider that problem with a slightly different setup that converts the problem into an inverse one.

Assume we measured temperature at 100 equidistant points on the rod. Additionally, we have the same source term and the same boundary conditions and aim to find the thermal diffusivity constant κ in (5.28). Again, the neural network seeks to predict the temperature values at 100 equidistant points on the rod. However, the thermal diffusivity constant κ is unknown this time. Therefore, the model will be trained to obtain the optimum values of the diffusivity constant as well as the network parameters. Mean square error between measured temperatures and model prediction will be used as the loss function. Additionally, a physics loss term that guides the prediction of the network according to the governing equations will be added to the loss function.

At first, initial settings are applied (see Listing 4) similar to the forward heat conduction problem defined in Sect. 5.4.1. However, the constant κ is not known in advance for this problem. We have an initial guess of $\kappa = 0.1$ for the thermal diffusivity constant. The neural network has three hidden layers with 32 neurons each, and the *tanh* function is used as the activation function. The ADAM optimizer optimizes the network parameters with a fixed learning rate of 0.001. The number of epochs is designated as 6000.

Listing 4: Initial settings for the inverse heat equation

```python
# importing necessary libraries
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# We set seeds initially. This feature starts the model with same random
# variables (e.g. initial weights of the network).
# By doing it so, we have same results whenever the code is run
tf.random.set_seed(123)

# 100 equidistant points in the domain are created
x = tf.linspace(0.0, 1.0, 100)

# boundary conditions which are T(0)=T(1)=0 are introduced.
bcs_x = [0.0, 1.0]
```

```
bcs_T = [0.0, 0.0]
bcs_x_tensor = tf.convert_to_tensor(bcs_x)[:, None]
bcs_T_tensor = tf.convert_to_tensor(bcs_T)[:, None]
kappa = tf.Variable([0.1], trainable=True)

# Number of iterations
N = 6000

# ADAM optimizer with learning rate of 0.001
optim = tf.keras.optimizers.Adam(learning_rate=1e-3)

#The exact solution of the problem. It will be used to produce measured data
#and test data
solution = lambda x: -5 * x**3 + 2 * x**2 + 3 * x

def buildModel(num_hidden_layers, num_neurons_per_layer):
    tf.keras.backend.set_floatx("float32")
    # Initialize a feedforward neural network
    model = tf.keras.Sequential()

    # Input is one dimensional ( one spatial dimension)
    model.add(tf.keras.Input(1))

    # Append hidden layers
    for _ in range(num_hidden_layers):
        model.add(
            tf.keras.layers.Dense(
                num_neurons_per_layer,
                activation=tf.keras.activations.get("tanh"),
                kernel_initializer="glorot_normal",
            )
        )

    # Output is one-dimensional
    model.add(tf.keras.layers.Dense(1))

    return model

# determine the model size (3 hidden layers with 32 neurons each)
model = buildModel(3, 32)
```

After defining the model settings, we can proceed with constructing the loss function. The loss function (5.43) consists of three parts, namely, boundary loss, physics loss, and data loss.

$$\mathcal{L}_{Loss} = \mathcal{L}_{BCs} + \mathcal{L}_{Physics} + \mathcal{L}_{Data} \tag{5.43}$$

with

$$\begin{aligned}\mathcal{L}_{BCs} &= \frac{\lambda_1}{N_B}\sum_{i=1}^{N_B}|T_{NN}(\mathbf{x}_i) - \mathbf{y}_i|^2,\\ \mathcal{L}_{Physics} &= \frac{\lambda_2}{N_P}\sum_{j=1}^{N_P}|\left.\frac{d^2T_{NN}}{dx^2}\right|_{x=x_j} + \frac{q(x_j)}{\kappa}|^2,\\ \mathcal{L}_{Data} &= \frac{\lambda_3}{N_D}\sum_{j=1}^{N_D}|T_{NN}(\mathbf{x}_j) - \mathbf{y}_j|^2\end{aligned} \tag{5.44}$$

Here N_B, N_P, and N_D correspond to the number of data points for boundary loss, physics loss, and measured data loss, respectively. Regularization terms $\lambda_1, \lambda_2, \lambda_3$ are taken as 1 in this example; (5.43) and (5.44) are defined in the code as follows:

Listing 5: Loss function for the inverse heat equation

```
@tf.function
def boundary_loss(bcs_x_tensor, bcs_T_tensor):
    predicted_bcs = model(bcs_x_tensor)
    mse_bcs = tf.reduce_mean(tf.square(predicted_bcs - bcs_T_tensor))
    return mse_bcs

# the first derivative of the prediction
@tf.function
def get_first_deriv(x):
    with tf.GradientTape() as tape:
        tape.watch(x)
        T = model(x)
    T_x = tape.gradient(T, x)
    return T_x

# the second derivative of the prediction
@tf.function
def second_deriv(x):
    with tf.GradientTape() as tape:
        tape.watch(x)
        T_x = get_first_deriv(x)
    T_xx = tape.gradient(T_x, x)
    return T_xx

# Source term
def source_func(x): return (15 * x - 2)

@tf.function
def physics_loss(x):
    x = x[1:-1]
    predicted_Txx = second_deriv(x)
    mse_phys = tf.reduce_mean(
        tf.square(predicted_Txx * kappa + source_func(x)))
    return mse_phys

@tf.function
```

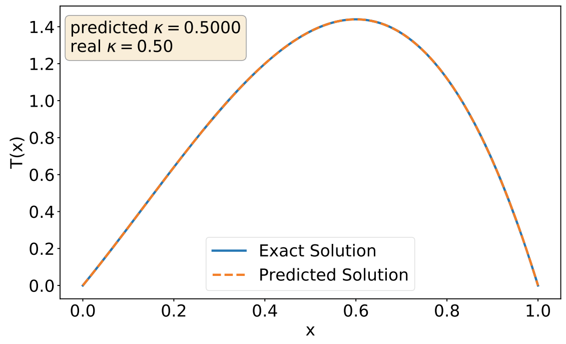
```
def data_loss(x):
    x = x[1:-1]
    ob_T = solution(x)[:, None]
    data_loss = tf.reduce_mean(tf.square(ob_T - model(x)))
    return data_loss

@tf.function
def loss_func(x):
    bcs_loss = boundary_loss(bcs_x_tensor, bcs_T_tensor)
    phys_loss = physics_loss(x)
    ob_loss = data_loss(x)
    loss = phys_loss + ob_loss + bcs_loss
    return loss
```

The training and testing procedures are the same as for the forward problem. Again, the gradients of the loss function with respect to κ and the trainable variables, which are weights and biases of the network, are determined with backpropagation. Then, the trainable variables and the κ value are updated by the ADAM optimizer using previously obtained gradients. This iterative procedure is repeated a number of epochs times, and eventually, it is expected to reach the possible minimum loss value.

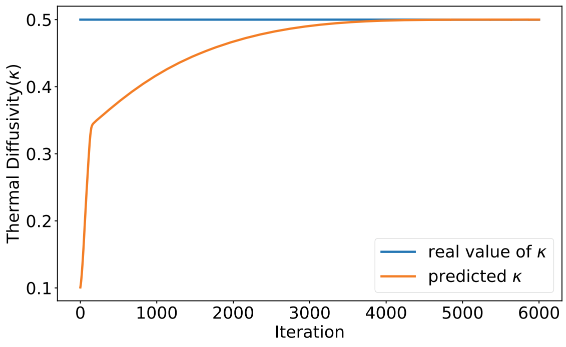
Listing 6: Training

```
# taking gradients of the loss function w.r.t. trainable variables
# and kappa
@tf.function
def get_grad():
    with tf.GradientTape(persistent=True) as tape:
        # This tape is for derivatives with
        # respect to trainable variables
        tape.watch(model.trainable_variables)
        tape.watch(kappa)
        Loss = loss_func(x)
    g = tape.gradient(Loss, model.trainable_variables)
    g_kappa = tape.gradient(Loss, kappa)
    return Loss, g, g_kappa

# optimizing and updating the weights and biases of the model and
# kappa by using the gradients
@tf.function
def train_step():
    # Compute current loss and gradient w.r.t. parameters
    loss, grad_theta, grad_kappa = get_grad()

    # Perform gradient descent step
    optim.apply_gradients(zip(grad_theta, model.trainable_variables))
    optim.apply_gradients([(grad_kappa, kappa)])
    return loss
```

The network parameters obtained at the last epoch form our model. We can test the model with a new data set in the same domain and plot the results to compare it with the ground truth (see Fig. 5.13a). The value for κ estimated by the neural



(a) The exact temperature and the predicted temperature.



(b) Change in the prediction of κ during the training.

Fig. 5.13 The temperature and thermal diffusivity constant prediction of the neural network and true values

network is equal to 0.5000, and the real value of κ is 0.5. Figure 5.13b illustrates that as the network is being trained, the value of κ converges to the true value. The relative L_2 error norm is 7.575×10^{-6}.

5.4.2.2 Inverse Helmholtz Equation

The second and also last example is the Helmholtz equation, which is a time-independent version of the wave equation. It is used for describing problems in electromagnetic radiation, acoustics, and seismology. The homogeneous form of the Helmholtz equation is written as :

$$\nabla^2 u + k^2 u = 0 \tag{5.45}$$

where ∇^2 is the Laplace operator and k is the wave number. The solution of the problem is $u(x, y)$ for $(x, y) \in \Omega$. An inverse acoustic duct problem, adopted from Anitescu et al. (2019), whose governing equation is a complex-valued Helmholtz equation such that k is unknown and $u(x, y)$ is known at some points in the domain, will be investigated.

We can write (5.45) with domain information and boundary conditions as:

$$\nabla^2 u(x, y) + k^2 u(x, y) = 0 \text{ where } (x, y) \in \Omega \text{ and } \Omega := (0, 2) \times (0, 1), \tag{5.46}$$

with the Neumann and Robin Boundary Conditions

$$\begin{aligned} \frac{\partial u}{\partial n} &= \cos(m\pi x) \quad \text{at} \quad x = 0 \\ \frac{\partial u}{\partial n} &= -iku \quad \text{at} \quad x = 2 \\ \frac{\partial u}{\partial n} &= 0 \quad \text{for} \quad y = 0 \quad \text{and} \quad y = 1 \end{aligned} \tag{5.47}$$

m being the number of modes which is taken as 1. The wave number k is unknown. The initial guess for k is 1, and the true value is chosen as $k = 6$. The exact solution for $u(x, y)$ can be written as:

$$\begin{aligned} &u(x, y) = \cos(m\pi y)(A_1 e^{-ik_x x})(A_2 e^{ik_x x}), \\ &\text{where} \quad k_x = \sqrt{k^2 - (\pi m)^2} \end{aligned} \tag{5.48}$$

where A_1 and A_2 are obtained by solving a 2×2 linear system:

$$\begin{bmatrix} ik_x & -ik_x \\ (k - k_x)e^{-2ik_x} & (k + k_x)e^{2ik_x} \end{bmatrix} \times \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{5.49}$$

Similar to the previous inverse problem in which we obtained the thermal diffusivity constant of a rod, the overall loss function is composed of boundary loss, physics loss and data loss. The boundary loss is constructed by Neumann and Robin boundary conditions specified in (5.47), and the physics loss is equal to the left-hand-side of (5.46). In addition, the data loss, in other words, the mean square error between observed $u(x, y)$ values and the prediction of the neural network is the last term in our overall loss function. These loss functions can be described as follows:

$$\mathcal{L}_{loss} = \mathcal{L}_{BCs} + \mathcal{L}_{Physics} + \mathcal{L}_{Data} \tag{5.50}$$

where $\mathcal{L}_{BCs}$, $\mathcal{L}_{Physics}$, $\mathcal{L}_{Data}$ are:

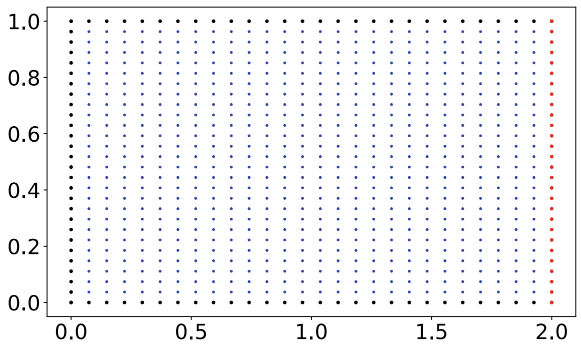


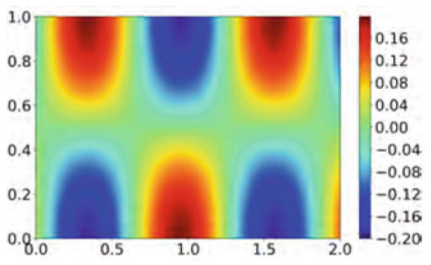
Fig. 5.14 Collocation points for 2D Helmholtz equation. Black points depict the boundary points where Neumann boundary conditions are valid whereas the red points show the Robin boundary points. In addition, blue points represent the inner collocation points where physics loss and data loss are computed

$$\begin{aligned}
\mathcal{L}_{BCs} &= \frac{\lambda_1}{N_B}\sum_{i=1}^{N_B}|\frac{\partial u_{NN}}{\partial n}(x_i^b, y_i^b) - \frac{\partial u}{\partial n}(x_i^b, y_i^b)|^2, \\
\mathcal{L}_{Physics} &= \frac{\lambda_2}{N_P}\sum_{j=1}^{N_P}|\nabla^2 u_{NN}(x_j^*, y_j^*) - k^2 u(x_j^*, y_j^*)|^2, \qquad (5.51) \\
\mathcal{L}_{Data} &= \frac{\lambda_3}{N_D}\sum_{j=1}^{N_D}|u_{NN}(x_j^*, y_j^*) - u(x_j^*, y_j^*)|^2
\end{aligned}$$

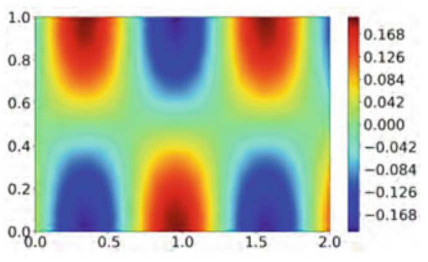
Here $\mathcal{L}_{BCs}, \mathcal{L}_{Physics}, \mathcal{L}_{Data}$ refer to the loss obtained from boundary conditions, governing equation, and measured data, respectively. The regularization term λ_1 is 100 whereas λ_2 and λ_3 are 1; N_B indicates the number of boundary points, N_P, N_D are the number of interior collocation points where physics loss is computed and the number of points where the observed data is available, respectively. In this problem, 784 equidistant points (28×28) such that N_P=N_D = 676 and N_B =108 are created (see Fig. 5.14).

The neural network consists of 5 hidden layers with the *tanh* activation function, and there are 30 neurons in each layer. The data is normalized to the interval $[-1, 1]$ before being processed. First, ADAM optimizer and, subsequently, the quasi-Newton method (L-BFGS) are employed to minimize the loss function. Five thousand iterations for ADAM and 6200 iterations with L-BFGS are applied. The estimated solution for $u(x, y)$ and the exact solution are shown in Fig. 5.15.

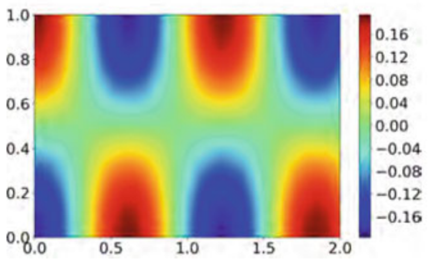
The initial guess for k was one, and the neural network's estimation for k after the training is 5.999. The relative L_2 error norm for the real part of the solution is 0.0015. A comparison between the predicted solution and the exact solution can be found in Fig. 5.16.



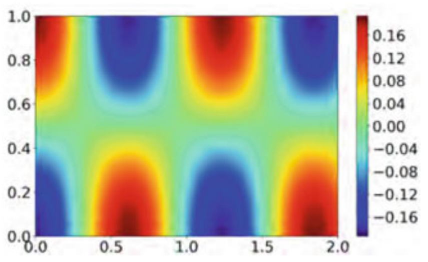
(a) Predicted solution for the real part of Helmholtz equation



(b) Exact solution for the real part of Helmholtz equation

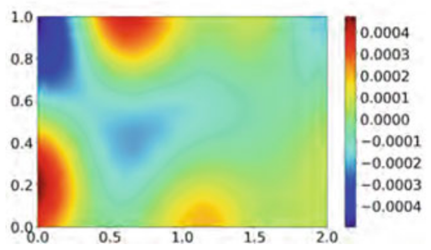


(c) Predicted solution for the imaginary part of Helmholtz equation

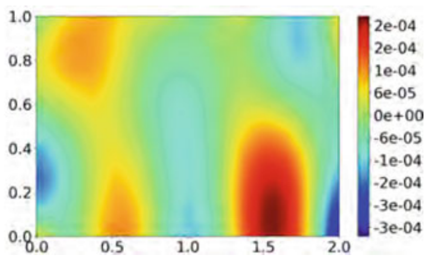


(d) Exact solution for the imaginary part of Helmholtz equation

Fig. 5.15 Predicted and exact values for real and imaginary parts of the Helmholtz equation



(a) Error distribution between predicted and exact solution for the real part



(b) Error distribution between predicted and exact solution for the imaginary part

Fig. 5.16 Error distribution for real and imaginary parts of the Helmholtz equation

5.5 Conclusions

In this chapter, we have introduced some of the main building blocks for PINNs. The main idea is to cast the process of solving a PDE as an optimization problem, where either the residual or some energy functional related to the governing equations is minimized. We showed the implementation of PINNs for both simple and more advanced inverse problems. First, a 1D steady-state heat conduction problem with a source term was solved for the unknown thermal diffusivity constant κ. Later, a complex-valued Helmholtz equation for an inverse acoustic duct problem was investigated. The wave number k is unknown in the beginning, and it is approximated by the PINN model. Unlike the forward problems, we have an additional term in the loss function, which is formed as the mean square error between the measured data and the model's prediction.

By taking advantage of modern machine learning libraries, it is possible to write fairly succinct programs that approximate the solution or some quantity of interest, while at the same time taking advantage of the built-in parallelization offered by multi-processor and GPU architectures. Nevertheless, solving PDEs by the optimization of parameters in a "standard" fully connected neural network is less efficient than current methods such as finite elements. More advances seem possible by combining machine learning algorithms with classical methods for solving PDEs which make use of the available knowledge for approximating the solutions or quantities of interest.

References

Abadi M, Agarwal A, Barham P, Brevdo E et al (2015) TensorFlow: large scale machine learning on heterogeneous systems. Software available from tensorflow.org. https://www.tensorflow.org/

Agostinelli F, Hoffman M, Sadowski P, Baldi P (2014) Learning acti vation functions to improve deep neural networks. arXiv:1412.6830

Anitescu C, Atroshchenko E, Alajlan N, Rabczuk T (2019) Artificial neural network methods for the solution of second order boundary value problems. Comput Mater Continua 59(1):345–359

Apicella A, Donnarumma F, Isgr'o F, Prevete R (2021) A survey on modern trainable activation functions. Neural Netw 138:14–32

Bin Waheed U, Haghighat E, Alkhalifah T, Song C et al (2021) PINNeik: Eikonal solution using physics-informed neural networks. Comput Geosci 155:104833

Bradbury J, Frostig R, Hawkins P, Johnson MJ et al (2018) JAX: compos able transformations of Python+NumPy programs. Version 0.2.5. http://github.com/google/jax

Broyden CG (1970) The convergence of a class of double-rank minimiza tion algorithms: 2. The new algorithm. IMA J Appl Math 6(3):222–231

Chen Y, Lu L, Karniadakis GE, Dal Negro L (2020) "Physics-informed neural networks for inverse problems in nano-optics and metamateri als. Opt Express 28(8):11618–11633

Clevert D-A, Unterthiner T, Hochreiter S (2015) Fast and accurate deep network learning by exponential linear units (ELUs). arXiv:1511.07289

De Sa C, Re C, Olukotun K (2015) Global convergence of stochastic gradient descent for some non-convex matrix problems. International conference on machine learning. PMLR, pp 2332–2341

Depina I, Jain S, Mar Valsson S, Gotovac H (2022) Application of physics-informed neural networks to inverse problems in unsaturated groundwater flow. Georisk: Assess Manag Risk Eng Syst Geohazards 16(1):21–36

Dillon JV, Langmore I, Tran D, Brevdo E et al (2017) Tensorflow dis tributions. arXiv:1711.10604

Fletcher R (1970) A new approach to variable metric algorithms. Comput J 13(3):317–322

Floridi L, Chiriatti M (2020) GPT-3: Its nature, scope, limits, and consequences. Minds Mach 30(4):681–694

Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics. JMLR Work shop and conference proceedings, pp 249–256

Goldfarb D (1970) A family of variable-metric methods derived by varia tional means. Math Comput 24(109):23–26

Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press

Goswami S, Anitescu C, Rabczuk T (2020) Adaptive fourth-order phase field analysis for brittle fracture. Comput Methods Appl Mech Eng 361:112808

Gühring I, Kutyniok G, Petersen P (2020) Error bounds for approxi mations with deep ReLU neural networks in Ws, p norms. Anal Appl 18(05):803–859

Haghighat E, Amini D, Juanes R (2022) Physics-informed neural net work simulation of multiphase poroelasticity using stress-split sequen tial training. Comput Methods Appl Mech Eng 397:115141

He J, Li L, Xu J, Zheng C (2020) Relu deep neural networks and linear finite elements. J Comput Math 38(3):502–527

Hendrycks D, Gimpel K (2016) Gaussian error linear units (GELUs). arXiv:1606.08415

He K, Zhang X, Ren S, Sun J (2015) Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE international conference on computer vision, pp 1026–1034

Hornik K, Stinchcombe M, White H (1989) Multilayer feedforward networks are universal approximators. Neural Netw 2(5):359–366

Jagtap AD, Kawaguchi K, Em Karniadakis G (2020a) Locally adap tive activation functions with slope recovery for deep and physics informed neural networks. Proc R Soc A 476(2239):20200334

Jagtap AD, Kawaguchi K, Karniadakis GE (2020b) "Adaptive acti vation functions accelerate convergence in deep and physics-informed neural networks. J Comput Phys 404:109136

Jagtap AD, Shin Y, Kawaguchi K, Karniadakis GE (2022) Deep Kronecker neural networks: A general framework for neural networks with adaptive activation functions. Neurocomputing 468:165–180

Jouppi NP, Young C, Patil N, Patterson D et al (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th annual international symposium on computer architecture, pp 1–12

Jumper J, Evans R, Pritzel A, Green T et al (2021) Highly accurate protein structure prediction with Alpha fold. Nature 596(7873):583–589

Kharazmi E, Zhang Z, Karniadakis GE (2019) Variational physics informed neural networks for solving partial differential equations. arXiv:1912.00873

Kingma DP, Ba J (2014) Adam: A method for stochastic optimiza tion. arXiv:1412.6980

Kissas G, Yang Y, Hwuang E, Witschey WR et al (2020) Machine learn ing in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4D flow MRI data using physics-informed neural net works. Comput Methods Appl Mech Eng 358:112623

Lagaris IE, Likas AC, Papageorgiou DG (2000) Neural-network methods for boundary value problems with irregular boundaries. IEEE Trans Neural Netw 11(5):1041–1049

Lagaris IE, Likas A, Fotiadis DI (1997) Artificial neural network methods in quantum mechanics. Comput Phys Commun 104(1–3):1–14, 40

Lagaris IE, Likas A, Fotiadis DI (1998) Artificial neural networks for solving ordinary and partial differential equations. IEEE Trans Actions Neural Netw 9(5):987–1000

Levenberg K (1944) A method for the solution of certain non-linear prob lems in least squares. Q Appl Math 2(2):164–168

Li A, Chen R, Farimani AB, Zhang YJ (2020a) Reaction diffusion system prediction based on convolutional neural network. Sci Rep 10(1):1-9

Li Z, Kovachki N, Azizzadenesheli K, Liu B et al (2020b) Fourier neural op erator for parametric partial differential equations. arXiv:2010.08895

Li Z, Liu F, Yang W, Peng S et al (2021) A survey of convolutional neural networks: analysis, applications, and prospects. IEEE Trans Neural Netw Learn Syst

Liu DC, Nocedal J (1989) On the limited memory BFGS method for large scale optimization. Math Program 45(1):503–528

López J, Anitescu C, Rabczuk T (2021) Isogeometric structural shape optimization using automatic sensitivity analysis. Appl Math Model 89:1004–1024

Lu L, Jin P, Pang G, Zhang Z et al (2021) Learning nonlinear opera tors via DeepONet based on the universal approximation theorem of operators. Nat Mach Intell 3(3):218–229

Maas AL, Hannun AY, Ng AY et al (2013) Rectifier nonlinearities improve neural network acoustic models. Proc icml 30(1):3. Citeseer

Marquardt DW (1963) An algorithm for least-squares estimation of non linear parameters. J Soc Indus Appl Math 11(2):431–441

Mertikopoulos P, Hallak N, Kavis A, Cevher V (2020) On the al most sure convergence of stochastic gradient descent in non-convex problems. Adv Neural Inf Process Syst 33:1117–1128

Misra D (2019) Mish: A self regularized non-monotonic activation function. arXiv:1908.08681

Nguyen-Thanh VM, Zhuang X, Rabczuk T (2020) A deep energy method for finite deformation hyperelasticity. Eur J Mech-A/Solids 80:103874

Otero AD, Ponta FL (2010) Structural analysis of wind-turbine blades by a generalized Timoshenko beam model

Paszke A, Gross S, Massa F, Lerer A et al (2019) PyTorch: an impera tive style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32. Curran Associates, Inc., 2019, pp 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Petersen P, Voigtlaender F (2018) Optimal approximation of piecewise smooth functions using deep ReLU neural networks. Neural Netw 108:296–330

Pfau D, Spencer JS, Matthews AGDG, Foulkes WMC (2020) Ab initio solution of the many-electron Schrödinger equation with deep neural networks. Phys Rev Res 2:033429

Philipp G, Song D, Carbonell JG (2018) Gradients explode—Deep Networks are shallow—ResNet explained

Raissi M, Perdikaris P, Karniadakis GE (2019) Physics-informed neu ral networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. J Comput Phys 378:686–707

Ramachandran P, Zoph B, Le QV (2017) Searching for activation functions. arXiv:1710.05941

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning rep resentations by back-propagating errors. Nature 323(6088):533–536

Samaniego E, Anitescu C, Goswami S, Nguyen-Thanh VM et al (2020) An energy approach to the solution of partial differential equations in computational mechanics via machine learning: Concepts, implementation and applications. Comput Methods Appl Mech Eng 362:112790

Shanno DF (1970) Conditioning of quasi-Newton methods for function minimization. Math Comput 24(111):647–656

Shukla K, Di Leoni PC, Blackshire J, Sparkman D et al (2020) Physics informed neural network for ultrasound nondestructive quantification of surface breaking cracks. J Nondestruct Eval 39(3):1–20

Shukla K, Jagtap AD, Karniadakis GE (2021) Parallel physics informed neural networks via domain decomposition. J Comput Phys 447:110683

Silver D, Hubert T, Schrittwieser J, Antonoglou I et al (2017) Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815

Sirignano J, Spiliopoulos K (2018) DGM: A deep learning algorithm for solving partial differential equations. J Comput Phys 375:1339–1364

Sukumar N, Srivastava A (2022) Exact imposition of boundary con ditions with distance functions in physics-informed deep neural net works. Comput Methods Appl Mech Eng 389:114333
Sun S, Cao Z, Zhu H, Zhao J (2019) A survey of optimization meth ods from a machine learning perspective. IEEE Trans Cybern 50(8):3668–3681
Vauhkonen M, Tarvainen T, Lähivaara T (2016) Inverse problems. In: Pohjolainen S (ed) Mathematical modelling. Springer International Publishing
Wang G-F, Feng X-Q (2009) Timoshenko beam model for buckling and vibration of nanowires with surface effects. J Phys D: Appl Phys 42(15):155411
Wang C, Tan V, Zhang Y (2006) Timoshenko beam model for vibra tion analysis of multi-walled carbon nanotubes. J Sound Vib 294(4–5):1060–1072
Wang S, Yu X, Perdikaris P (2022) When and why PINNs fail to train: A neural tangent kernel perspective. J Comput Phys 449:110768
Wight CL, Zhao J (2020) Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks. arXiv:2007.04542
Yu B et al (2018) The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems. Commun Math Stat 6(1):1–12
Yu J, Lu L, Meng X, Karniadakis GE (2022) Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems. Comput Methods Appl Mech Eng 393:114823
Zhuang X, Guo H, Alajlan N, Zhu H et al (2021) Deep autoencoder based energy method for the bending, vibration, and buckling anal ysis of Kirchhoff plates with transfer learning. Eur J Mech-A/Solids 87:104225