energies

MDPI

Article

Physics-Based Deep Learning for Flow Problems

Yubiao Sun [1,*], Qiankun Sun [2] and Kan Qin [3]

[1] Department of Engineering, University of Cambridge, Cambridge CB2 1TN, UK
[2] Investment Promotion and Enterprise Service Center of Yantian District, Shenzhen 518000, China; qksun1993@163.com
[3] School of Marine Science and Technology, Northwestern Polytechnical University, Xi'an 710060, China; kan.qin@nwpu.edu.cn
* Correspondence: ys572@cam.ac.uk

Abstract: It is the tradition for the fluid community to study fluid dynamics problems via numerical simulations such as finite-element, finite-difference and finite-volume methods. These approaches use various mesh techniques to discretize a complicated geometry and eventually convert governing equations into finite-dimensional algebraic systems. To date, many attempts have been made by exploiting machine learning to solve flow problems. However, conventional data-driven machine learning algorithms require heavy inputs of large labeled data, which is computationally expensive for complex and multi-physics problems. In this paper, we proposed a data-free, physics-driven deep learning approach to solve various low-speed flow problems and demonstrated its robustness in generating reliable solutions. Instead of feeding neural networks large labeled data, we exploited the known physical laws and incorporated this physics into a neural network to relax the strict requirement of big data and improve prediction accuracy. The employed physics-informed neural networks (PINNs) provide a feasible and cheap alternative to approximate the solution of differential equations with specified initial and boundary conditions. Approximate solutions of physical equations can be obtained via the minimization of the customized objective function, which consists of residuals satisfying differential operators, the initial/boundary conditions as well as the mean-squared errors between predictions and target values. This new approach is data efficient and can greatly lower the computational cost for large and complex geometries. The capacity and generality of the proposed method have been assessed by solving various flow and transport problems, including the flow past cylinder, linear Poisson, heat conduction and the Taylor–Green vortex problem.

Keywords: deep learning; physics-informed neural networks; partial differential equation; automatic differentiation; surrogate model

Citation: Sun, Y.; Sun, Q.; Qin, K. Physics-Based Deep Learning for Flow Problems. *Energies* **2021**, *14*, 7760. https://doi.org/10.3390/en14227760

Academic Editor: Dmitry Eskin

Received: 12 October 2021
Accepted: 16 November 2021
Published: 19 November 2021

1. Introduction

The solution of many physical equations in fluid dynamics is generally obtained by discretesizing them into finite difference equations and then solving the algebraic equations. For complicated problems with high dimension or for the sake of accurate solutions, computational costs soar when more discrete points are required. This difficulty can be overcome by the increasingly popular and powerful machine learning approach. Machine learning is superior in dealing with any nonlinear problems compared to the conventional discretization method, which often requires making appropriate prior assumptions, performing linearization, and considering restrictive local time-stepping. Machine learning makes use of multilayer neural network architectures to model various physical systems. It has been acknowledged that standard multilayer feed-forward network architectures with sufficiently many hidden units act as universal approximators as they can approximate virtually any function to any desired degree of accuracy [1]. The first attempt to apply neural networks (NNs) to solve PDEs was reported by Lee and Kang et al. [2]. The main idea is to approximate the solution of PDEs using the NN as continuous functions and

train the neural networks to minimize the solution residuals inside the domain and on the boundaries. Lagaris et al. [3] demonstrated this idea by solving a number of benchmark problems such as the Poisson equation, subject to both Dirichlet and Neumann boundary conditions.

However, to solve differential equations with high accuracy, great care has to be taken when designing the neural network structure. Another challenge is that huge amounts of data have to be fed into the neural network in the training stage, which is a time-consuming and computationally prohibitive process. This prompts researchers to seek room for improvement: Is it possible to inform the neural network from the beginning for performance enhancement? Multilayer perceptions (MLP) was reported to build a smart neural model based on predicting the proper orthogonal decomposition modes of the Kuramoto–Sivashinsky (KS) equation and the Navier–Stokes equation [4]. Karhunen–Loéve decomposition was applied to reduce the dataset for training of neural networks, after which the trained NN is capable of predicting the data coefficients at a future time. Similarly, MLP was also employed to compute the solution of the Stokes equation by decomposing it into multiple Poisson problems and then solving these Poisson equations with neural networks. The neural network approximates the Stokes equation using randomly sampled data points and delivers solutions that are in a differentiable and closed analytic form [5].

Big progress was made by Raissi et al. as they introduced a new concept of 'Physics Informed Neural Networks (PINNs)' to tackle PDEs defined in complex domains with a variety of boundary conditions [6,7]. PINN exploits structured prior information to construct neural networks with physical equation integration [8–10]. Gaussian process regression was employed to derive functional representations of linear and nonlinear problems. When dealing with nonlinear problems, locally linearization of nonlinear terms in time is required, thus limiting the methods applicable only to discrete-time domains. Furthermore, the method's robustness is compromised with the Bayesian nature of Gaussian process regression as some certain prior assumptions are introduced. Inspired by the Galerkin method's solution approximation strategy using linear combination of basis functions, K. Spiliopoulos et al. put forward that the solution can also been approached by the combined functions arising from vast number of neuron units in neural networks, and coined this approach the "Deep Galerkin Method (DGM)" [11]. They also proved that the neural network would converge to the solution of the partial differential equation as the number of hidden units increases.

This idea was furthered by Lu and Karniadakis as they released the "DeepXDE" library to handle a wide range of differential equations including partial- and integro-differential equations [12]. The concept of PINN was extended to solve problems with limited high-fidelity data and sufficient and readily low-fidelity data by constructing four fully-connected neural networks, which can learn both the linear and complex nonlinear correlations between high- and low-fidelity data. This work is pretty useful for high-dimensional regression and classification problems with large multi-fidelity data.

This paper is focused on developing a physics-informed, data-free deep neural network for surrogate modeling of various flow and heat transfer problems. A multi-layer neural network structure has been devised to approximate the solutions of physical equations, with the initial/boundary conditions being penalized in training stage. Compared with conventional data-driven machine learning approach, our devised neural network is advantageous as the training process is driven by minimizing the residuals of the governing equations, and no large labeled data from expensive numerical simulations are required. The generality and robustness of the proposed method are demonstrated in several flow dynamics and heat transfer problems. The rest of this paper is organized as follows. Section 2 gives a comprehensive introduction of deep learning. The proposed physics-informed neural networks is described in Section 3. In Section 4, the developed approach is applied to solve a variety of test problems governed by various differential

equations. Finally, the conclusion part in Section 5 summarizes this work and points out the limitations and future directions of our method.

2. General Description of Deep Learning

Deep learning, a state-of-the-art method, has demonstrated its success in solving complex problems in speech recognition [13], computer vision [14], natural language processing [15] and audio processing [16]. It was originally inspired by biological neural networks, as is shown in Figure 1. By mapping a large number of inputs into a target output, it can approximate and estimate highly complicated functions. As the neural networks go deeper, complicated features at various levels of abstraction can be learned and thus better predictability and higher accuracy are available. Increasing the number of layers enables neural networks to have superior generality as the level of features is enriched [17]. With sufficient layers and enough transformations, neural networks are capable of approximating any function to any desired degree of accuracy. A systematical study by Chen et al. [18] proved the universal capability of neural networks to approximate functionals, nonlinear operators and functions of multiple variables.

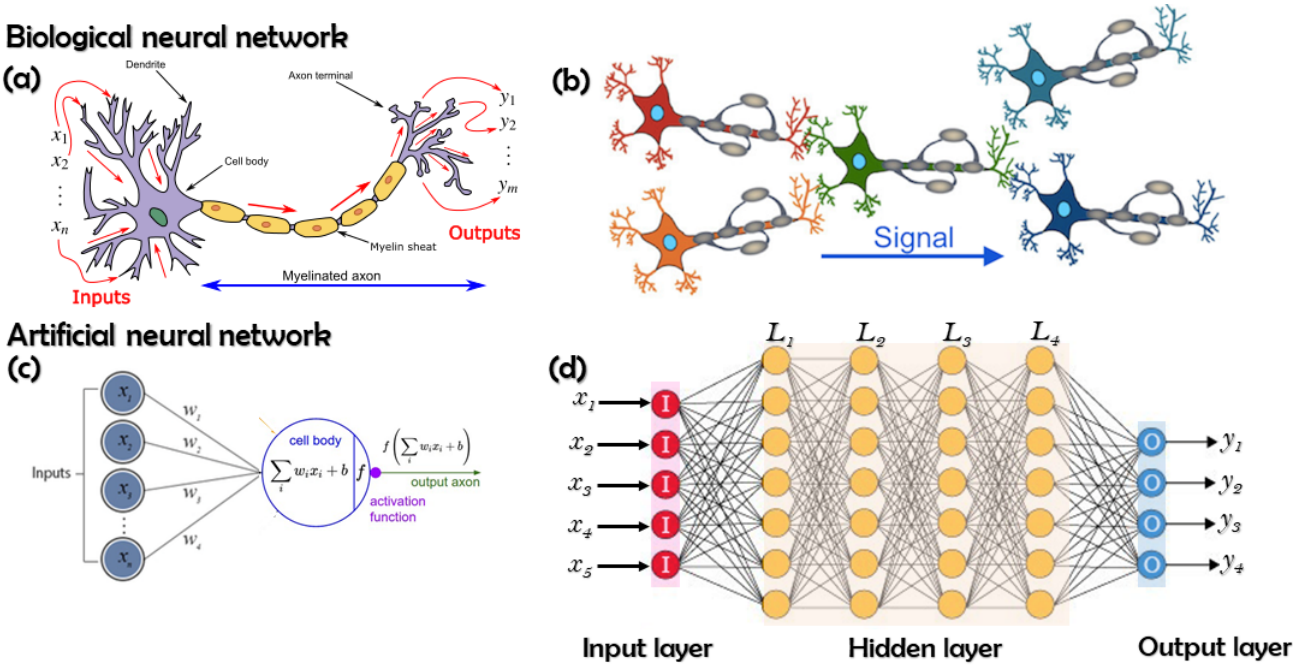


Figure 1. Biological (**a**,**b**) and artificial (**c**,**d**) neural networks.

2.1. Deep Neural Networks

A deep neural network (DNN), a multi-layer neural network, is essentially a "stack" of nonlinear operations where each operation is prescribed by some adjustable parameters. Compared with single-layer neural network, a DNN can learn hierarchical representations to represent sophisticated phenomena as it has more parameters, more complex functions and better inductive bias [19,20]. Stacked layer by layer, deep learning is realized via such multi-layer neural networks, which act as the composite of nonlinear functions (also called activation functions) to transform function representation from one primary level into a higher, more abstract level (Figure 1).

From the perspective of mathematicians, the multi-layered neural networks use the compositions of simple function to approximate complicated ones, and thus act as a compositional model regarding function representation. A deep neural network $f(\mathbf{x})$ with

N hidden layers is composed of a series of compositional functions, linear or nonlinear. A general DNN shown in Figure 1d can be mathematically expressed as

$$f(\mathbf{x}) = \mathcal{F}^{out} \circ \mathcal{F}^{N} \circ \mathcal{F}^{N-1} \circ \mathcal{F}^{N-2} \circ \cdots \mathcal{F}^{1} \circ \mathcal{F}^{in} \tag{1}$$

where the symbol "$\circ$" denotes function composition. Here $\mathcal{F}^N$ is the mapping from layer $N-1$ to N. The superscript "*in*" and "*out*" denote the input and output layer.

The N-hidden-layer DNN, denoted by $f(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^{\mathcal{D}}$, implements the learning procedure by intaking the d-dimensional data from the layer of input units at the bottom ($\mathcal{F}^{in}$), mapping the incoming data via certain number of intermediate layers ($\mathcal{F}^j$), and finally generating the k-dimensional output from a layer of output unit at the top ($\mathcal{F}^{out}$). The j-th layer has N_j neurons, with the associated weight matrix and bias vector being referred to as $\mathbf{W}^j \in \mathbb{R}^{N_j \times N_{j-1}}$ and $\mathbf{b}^j \in \mathbb{R}^{N_j}$, respectively. With the employment of activation function σ, the N-layer DNN is defined by:

$$\text{Input layer:} \quad \mathcal{F}^{in}(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^d \tag{2}$$

$$\text{Hidden layer:} \quad \mathcal{F}^{j} = \sigma(\mathbf{W}^j \mathcal{F}^{j-1}(\mathbf{x}) + \mathbf{b}^j) \in \mathbb{R}^j \text{ where } 1 \le j \le N \tag{3}$$

$$\text{Output layer:} \quad \mathcal{F}^{out} = \mathbf{W}^{out} \mathcal{F}^{N}(\mathbf{x}) + \mathbf{b}^{out} \in \mathbb{R}^D \tag{4}$$

The labeled training set $\mathcal{T} = \{(x_i, y_i), 1 \le i \le N\}$ consists of input vectors x_i and output vector y_i of length N. The free parameters are $\theta = \{\mathbf{W}^j, \mathbf{b}^j, j = 1, 2, \cdots, N\}$ and explicitly write the neural network function parameterized by θ as $f(\cdot, \theta)$.

2.2. Objective Function

In the training stage, the major task is to identify the optimal weights θ that produce accurate predictions via the optimization of predefined objective functions, i.e., explicitly minimize a cost function by gradually adjusting the free-parameter weights. DNN acts as a mapping function $f(\cdot, \theta)$ that approximates the true value y_i using the predicted value $\hat{y}_i = f(x_i, \theta)$. The cost is usually expressed as "Mean Squared Error (MSE)" defined in Euclidean space. The cost function is usually calculated as an average over all training examples, as is shown below:

$$C_{mse}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \|y_i - \hat{y}_i\|^2 \tag{5}$$

where C_{mse} is the mean squared error evaluated for the given set of N inputs $\mathbf{X} = \{x_1, x_2, \cdots, x_N\}$ and the corresponding output $\mathbf{Y} = \{y_1, y_2, \cdots, y_N\}$ from neural network prediction.

2.3. Activation Function

The activation maps an incoming input x to an outcoming output y using different activation functions. Some of the most popular activation functions include Sigmoid ($y = \sigma(x) = \frac{1}{1+e^{-x}}$), hyperbolic tangent ($y = tanh(x) = \frac{2}{1+e^{-2x}} - 1$), and Rectified Linear Unit (ReLU, $y = \max\{0, x\}$). The ReLU function accelerates the training process, making neural networks several times faster than their equivalents with tanh units [21]. Another merit of ReLUs is that they do not require input normalization to prevent saturating. However, for regression applications, ReLU function suffers from diminishing second and higher-order derivatives, lowering the accuracy for cases involving such higher-order derivatives [22]. Tanh or Sigmoid activations can overcome this issue brought by the second or higher-order PDEs. Moreover, sigmoids or Tanh are chosen for classification problems as they stretch the input space around a central point and can categorize elements into different classes.

2.4. Optimization Process

To train a neural network, the derivatives, mostly in the form of gradients and Hessians, need to be computed [23]. Derivatives can be manually addressed as analytical formula, or computed by symbolic differentiation, numerical differentiation, and automatic differentiation (also called algorithmic differentiation, AD). The exact analytical expressions of derivatives are pretty hard to get manually for complex problems. More often, even if an analytical solution exists, its derivation is mathematically challenging, time consuming, and error prone. While for symbolic and finite differentiation, they both suffer from poor performance for complex functions [24]. Hence, AD has become the mainstream for derivative calculations and serves as the real secret sauce that powers machine learning.

2.4.1. Automatic Diffraction for Derivative Evaluation

AD executes program codes and automatically computes derivatives using chain rule for accumulation of values instead of derivative expressions. Specifically, AD interprates computer programs by replacing the variable domains to incorporate derivative values and redefining the semantics of operators to propagate derivatives per the chain rule of differential calculus [24]. For symbolic differentiation, the goal is a complex and accurate expression, whereas for AD, the goal is the numerical derivative evaluations. The main advantages of AD lie in its capability to evaluate derivatives at machine precision in constant time, with only a small constant factor of overhead and ideal asymptotic efficiency [25]. Currently AD can be implemented in two distinct ways: Forward mode and reverse mode. Forward mode evaluates derivatives by transversing the chain rule from inside to outside. For instance, a general target function f expressed in the composite of k functions

$$y = f^k(f^{k-1}(f^{k-2}\cdots(f^2(f^1(x))))) = f^k \circ f^{k-1} \circ \cdots \circ f^1(x)$$

where $f^k \circ f^{k-1} = f^k(f^{k-1}(x))$.With the variable replacement as: $t_0 = x, t_1 = f^1(t_0), t_2 = f^2(t_1), \cdots t_k = f^k(t_{k-1}) = y$, the calculation of derivatives in forward mode is:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial t_{k-1}}\frac{\partial t_{k-1}}{\partial x} = \frac{\partial y}{\partial t_{k-1}}\left(\frac{\partial t_{k-1}}{\partial t_{k-2}}\frac{\partial t_{k-2}}{\partial x}\right) = \frac{\partial y}{\partial t_{k-1}}\left(\frac{\partial t_{k-1}}{\partial t_{k-2}}\left(\frac{\partial t_{k-2}}{\partial t_{k-3}}\frac{\partial t_{k-3}}{\partial x}\right)\right) = \cdots \quad (6)$$

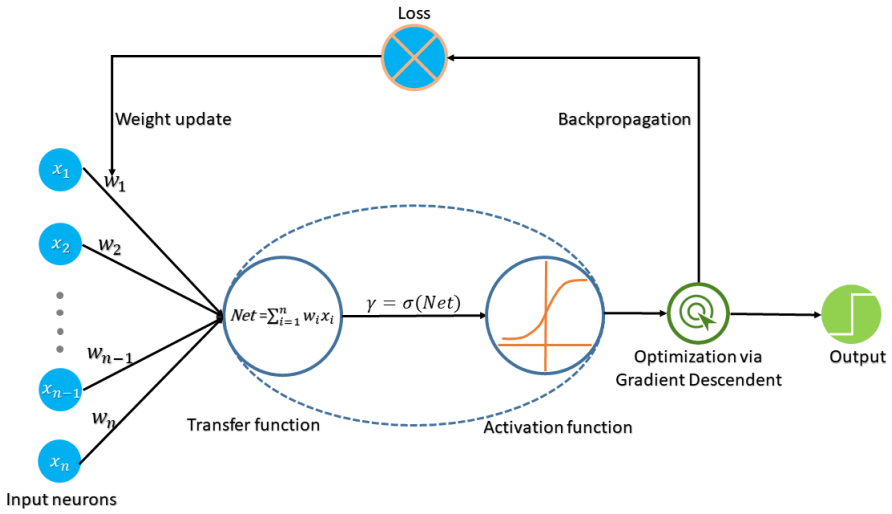
While in the reverse mode, the dependent variable to be differentiated is fixed and the derivatives are calculated from outside to inside, as is shown below:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial t_1}\frac{\partial t_1}{\partial x} = \left(\frac{\partial y}{\partial t_2}\frac{\partial t_2}{\partial t_1}\right)\frac{\partial t_1}{\partial x} = \left(\left(\frac{\partial y}{\partial t_3}\frac{\partial t_3}{\partial t_2}\right)\frac{\partial t_2}{\partial t_1}\right)\frac{\partial t_1}{\partial x} = \cdots \quad (7)$$

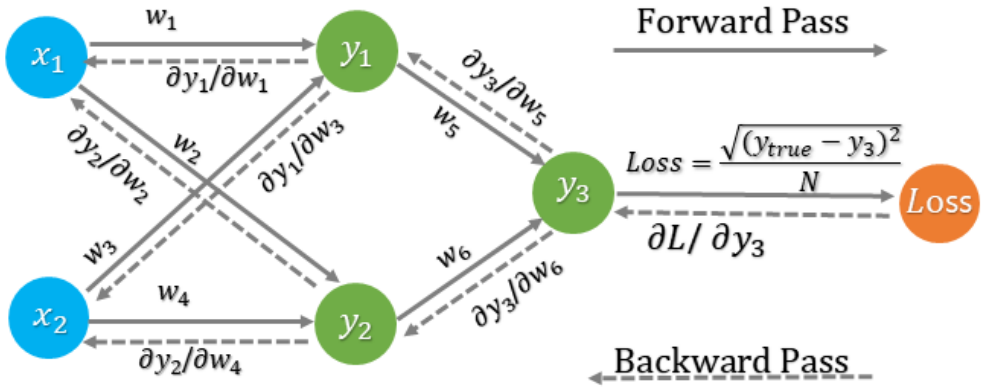
The backpropagation algorithm, a specialized counterpart of AD, is the backbone of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows us to reduce error rates and make the model reliable by increasing its generalization. First the sensitivity of the objective value at the output is computed as partial derivatives of the objective function with respect to each weight utilizing the chain rule; then the sensitivity is backpropopagated to derive the required gradients. The process is essentially equivalent to transforming the network evaluation function composed with the objective function under reverse mode AD. At the heart of backpropagation is the partial derivative of the objective function with respect to any weight (or bias) in the network, which gives detailed insights into how the changing weights and biases change the overall behaviour of the network.

Figure 2 shows the role and key procedures of backpropagation in a simple neural network. Figure 2b illustrates the phenomena, with an example describing both the forward and backward pass. In the forward direction, training inputs $x_1$and x_2 are transformed to generate corresponding output y_3. A loss function measuring the error between predicted output y_i and the true value y_3 is computed. For the backward propopaga-

tion, the sensitivity of objective function $J(\theta)$ with respect to different neuron weights $\nabla_{\theta_i} J = \left(\frac{\partial J}{\partial \theta_1}, \cdots, \frac{\partial J}{\partial \theta_6}\right)$ is used in a gradient-descent procedure for weights update.



(**a**)



(**b**)

Figure 2. An overview of backpropagation. (**a**) Role of backpropagation in a neural network; (**b**) Step-by-step backpropagation example.

2.4.2. Weight Update Using Gradient Descent

Gradient descent is commonly used to minimize an objective function $J(\theta)$ with a combination of varied neural network weights $\theta \in \mathbb{R}^d$. The minimization ends at a valley when following the downhill direction of the surface slope of the objective function. This minimization process updates the involved weights in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with an assigned learning rate η; a hyperparameter controls how much the model weights are updated in response to the estimated error at each iteration. A small η means a long training process that could get stuck, whereas a large value for the sake of accelerating the training process may prevent convergence due to the large fluctuation of loss function.

Based on the data used to compute the gradient of the objective function, the gradient descent algorithm can be classified into three variants: Batch gradient descent, stochastic gradient descent and mini-batch gradient descent. They show different levels of accuracy for the weight update and timescale at each iteration.

Batch gradient descent For the batch gradient descent (also referred to as vanilla gradient descent), the entire training datasets are used to compute the gradient of the cost

function for parameter update. It will converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces [26].

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{8}$$

Calculation of the gradients from the whole dataset makes the update quite slow and intractable for large datasets exceeding the memory limit. Batch gradient descent also does not support online model update, i.e., with new examples on the fly.

Stochastic gradient descent To prevent the slow convergence of batch gradient descent, stochastic gradient descent (SGD) has been introduced, where the fluctuations arising from the randomly selected points x_i allow jumps to new and potentially better local minima. This algorithm is a popular choice since it is fast, reliable, and has low susceptibility to bad local minima. In this algorithm, the weights are updated after the presentation of each example, according to the gradient of loss [27].

$$\theta = \theta - \eta \cdot \nabla_\theta J(x_i; \theta) \tag{9}$$

By performing the update at each iteration, SGD converges much faster than its batch-based counterpart and enables the model to update online. However, the fluctuation of SGD makes its jump to local minima, complicating the convergence to global minimum. The fluctuation causes a sharp change for a large learning rate. Decreasing the learning rate slows the convergence of SGD, and its convergence history is similar to the batch gradient descent approach.

Mini-batch gradient descent To strike a balance between batch gradient descent and SGD, the parameter θ is updated for every n training samples x_{i+n}, coined as mini-batch. This mini-batch gradient descent reduces the variance of parameter updates, shows a stable convergence and is compatible with many state-of-the-art matrix optimization approaches. The mini-batch sizes can range from 50 to 256, depending on the application scenario. It has become the algorithm of choice and the term SGD usually refers to mini-batch gradient descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(x_{i+n}; \theta) \tag{10}$$

It should be mentioned that this mini-batch gradient descent approach is also bounded by the challenge of getting trapped in some suboptimal local minima when minimizing highly non-convex error functions for many neural networks. This is because of the existence of saddle points which are usually surrounded by a plateau of the same error [28]. The saddle points have one dimension slope up and another slope down; thus their gradients are close to zero in all dimensions.

3. Physics-Driven Deep Learning

In fluid dynamics, many transport phenomena can be modeled by some partial differential equations (PDEs), which can be expressed as

$$\begin{aligned} \mathcal{L}(t, \boldsymbol{x}; u(t, \boldsymbol{x})) &= 0 \quad (t, \mathbf{x}) \in [0, T] \times \Omega \\ \mathcal{I}(\boldsymbol{x}; u(0, \boldsymbol{x})) &= 0 \quad x \in \Omega \\ \mathcal{B}(t, \boldsymbol{x}; u(t, \boldsymbol{x})) &= 0 \quad x \in [0, T] \times \partial\Omega \end{aligned} \tag{11}$$

where $\mathcal{L}(\cdot)$ denotes a general differential operator consisting of temporal and spatial derivatives, as well as some linear and nonlinear terms. The position vector $\boldsymbol{x}$ is defined on a bounded continuous spatial domain $\Omega \in \mathbb{R}^d$, $d \in \{1, 2, 3\}$ with the boundary denoted as $\partial\Omega$. The initial condition $\mathcal{I}(\cdot)$ and boundary condition $\mathcal{B}(\cdot)$ may contain differential, linear and nonlinear terms. $\mathcal{B}(\cdot)$ implements the Neumann, Dirichlet, Robin, or periodic boundary conditions. In view of that the true solution $u(t, \boldsymbol{x})$ is unknown or too costly to derive, an approximate one $\hat{u}_{NN}(t, \boldsymbol{x})$ can be obtained via the minimization of the cost function (usually the L^2 norm of errors) with the following formula:

$$
\begin{aligned}
R_{\mathcal{L}}(\theta) &= \int_{[0,T]\times\Omega} \|\mathcal{L}(t,\boldsymbol{x};\theta)\|^2 \,\mathrm{d}t\,\mathrm{d}\boldsymbol{x} \\
R_{\mathcal{I}}(\theta) &= \int_{\Omega} \|\mathcal{I}(t;\theta)\|^2 \,\mathrm{d}\boldsymbol{x} \\
R_{\mathcal{B}}(\theta) &= \int_{[0,T]\times\partial\Omega} \|\mathcal{B}(t,\boldsymbol{x};\theta)\|^2 \,\mathrm{d}t\,\mathrm{d}\boldsymbol{x}
\end{aligned} \tag{12}
$$

The training process of neural network produces a set of optimal $\theta^\star$, which is calculated based on

$$
\theta^\star = \underset{\theta}{\operatorname{argmin}}\, R_{\mathcal{L}}(\theta) \quad \text{subject to: } R_{\mathcal{I}} = 0 \text{ and } R_{\mathcal{B}} = 0 \tag{13}
$$

With the initial and boundary conditions posed as constraints, the solution of the general nonlinear PDE (Equation (11)) can be approximated as the outcome of the optimization problem defined by Equation (13). To solve this optimization problem, the constraints in Equation (13) are integrated into a sophisticated loss function that can be minimized by neural networks. For better illustration, the abstract PDE in Equation (11) is reformulated in a more expressive way, as is shown below:

$$
\partial_t u(t,\boldsymbol{x}) + \mathcal{L}u(t,\boldsymbol{x}) = 0 \qquad (t,\boldsymbol{x}) \in [0,T]\times\Omega \tag{14}
$$

$$
\text{subject to:} \quad u(0,\boldsymbol{x}) = u_0(\boldsymbol{x}) \qquad \boldsymbol{x} \in \Omega \tag{15}
$$

$$
u(t,\boldsymbol{x}) = g(t,\boldsymbol{x}) \qquad \boldsymbol{x} \in \partial\Omega \tag{16}
$$

where the dimensional variable $x \in \Omega \subset \mathbb{R}^d$. The unknown $u(t,x)$ is approximated by $\hat{u}(t,\boldsymbol{x};\theta)$ from a well-crafted deep neural network with adjustable weights θ. The accuracy of predictions is quantified by measuring the residual $J(\cdot;\theta)$ of the equation satisfaction under the constraint of boundary and initial conditions:

$$
\begin{aligned}
J(\cdot;\theta) = \Big\| \partial_t \hat{u}(t,\boldsymbol{x};\theta) + \mathcal{L}\hat{u}(t,\boldsymbol{x};\theta) \Big\|^2_{[0,T]\times\Omega,\xi_1} &+ \|\hat{u}(t,\boldsymbol{x};\theta) - g(t,\mathbf{x})\|^2_{[0,T]\times\partial\Omega,\xi_2} \\
&+ \|f(0,\boldsymbol{x};\theta) - u_0(\mathbf{x})\|^2_{\Omega,\xi_3}
\end{aligned} \tag{17}
$$

where $\|u(x)\|^2_{\Xi,\xi_1} = \int_\Xi |u(x)|^2 \xi(x)\mathrm{d}x$ and $\xi(x)$ is a positive probability density defined on the domain Ξ.

The true solution for Equation (14) can be identified under the condition of $J(u(t,\boldsymbol{x};\theta)) = 0$. However, in real situations, it is pretty hard to derive θ, especially for the high dimensional problems where the integral over the domain Ω is computationally intractable, hence a reliable approximate solution is sought at a reasonable cost. An approximate solution $\hat{u}(t,\boldsymbol{x};\theta)$ should minimize the error indicator $J(\cdot;\theta)$. A deep neural network uses the stochastic gradient descent (SGD), an iterative method, to implement the minimization task. A key strength of SGD lies in its ease. SGD is simple to implement and also fast for problems with substantial training data, which reduces the computational burden, achieving faster iterations in trade for a slightly lower convergence rate [29]. Instead of calculating the actual gradient from the entire dataset, the approximated gradient is generated by randomly selecting some data from the whole dataset [30].

The overview of the whole procedure is shown in Table 1. Key steps in solving the partial differential equations are listed below:

Table 1. Physics-informed deep learning approach.

Deep Learning Algorithm with Embedded Physics
1. Build the neural network architecture of DNN, i.e., setup the number of layers, number of neurons bounded to each layer and activation function.
2. Initialize neural network using provided parameters θ.
3. Construct the objective function for optimization using Equation (17), which accounts for the L^2 norm of the residual of the physical equation initial and boundary conditions represented by Equation (14), Equation (15) and Equation (16), respectively.
4. Implement the stochastic gradient descent algorithm within the mini-batch, specify optimizer hyper-parameters and batch size N.
5. Set iteration $n = 0$ and specify the maximum iteration number n_{max}.
6. Start the training in accordance with the following steps:
While $n < n_{max}$ do
• Design sampling strategy to generate random N input points $s_i = \{(t_i, x_i), (\tau_i, z_i), w_i\}$ from $[0, T] \times \Omega$ with $1 \leq i \leq N$
• Estimate the loss for each prediction loop n
• Update the weights from θ_n to θ_{n+1} according to stochastic gradient descent algorithm: $\theta_{n+1} = \theta_n - \eta_n \nabla J(s_n; \theta_n)$
• $n = n + 1$
end while

(1) Generate some random points (t_n, x_n) from $[0, T] \times \Omega$ the probability density ξ_1 to approximate the governing equation, Equation (14). Moreover, sample another set of points $(\tilde{t}_n, \tilde{x}_n)$ in $[0, T] \times \partial\Omega$ with the probability densities ξ_2 to capture boundary condition Equation (16) and pockets of random data w_n from Ω using possibility density ξ_3 to meet initial condition Equation (15). Lastly, distribute the random point $\widetilde{x}_n$ according to the probability density ξ_3. This sampling strategy avoids the lengthy and time-consuming mesh-generation process, and thus reduces computational cost.

(2) Calculate the objective function, i.e., the squared error $J(s_n; \theta_n)$ using the randomly sampled observation $s_n = \{(t_n, x_n), (\tilde{t}_n, \tilde{x}_n), \widetilde{x}_n\}$:

$$\begin{aligned} J(s_n; \theta_n) = (\partial_t \hat{u}(t_n, x_n; \theta_n) + \mathcal{L}\hat{u}(t_n, x_n; \theta_n))^2 + (\hat{u}(\tilde{t}_n, \tilde{x}_n; \theta_n) - g(\tilde{t}_n, \tilde{x}_n))^2 \\ + (\hat{u}(0, \widetilde{x}_n; \theta_n) - u_0(\widetilde{x}_n))^2 \end{aligned} \tag{18}$$

(3) Explicitly apply the gradient descent algorithm to update the weights of neural network. Each iteration involves drawing an example s_n at random and applying the parameter update rule:

$$\theta_{n+1} = \theta_n - \eta_n \nabla J(s_n; \theta_n) \tag{19}$$

The "learning rate" η_n decreases with increasing iteration n. It is either a positive scalar or a symmetric positive definite matrix. The step $\nabla G(s_n; \theta_n)$ is an unbiased estimate of $\nabla J_\theta(\hat{u}(\cdot; \theta_n))$:

$$\mathbb{E}[\nabla_\theta G(\theta_n, s_n) | \theta_n] = \nabla J(\hat{u}(\cdot; \theta_n)) \tag{20}$$

Therefore, the stochastic gradient descent algorithm will on average take steps in a descent direction for the objective function $J(\cdot; \theta)$. The descent direction diminishes the objective function so that $J(\hat{u}(\cdot; \theta_{n+1})) < J(\hat{u}(\cdot; \theta_n))$. As a result, the θ_{n+1} enables the neural network to produce a better estimation than θ_n.

As the iteration approaches to infinite (i.e., $n \to \infty$), the algorithm θ_n would ultimately converge to the critical point defined as:

$$\lim_{n\to\infty} \|\nabla_\theta J(\hat{u}(\cdot;\theta_n))\| = 0 \tag{21}$$

Since the stochastic gradient descent may converge to a local minimum for the non-convex optimization problems [11], a caution must be taken that θ_n is likely to converge to a local minimum rather than a global minimum for neural networks with non-convex nature.

4. Results

To illustrate the effectiveness of our proposed approach, four problems ranging from fluid dynamics to heat transfer are presented. These problems can be modeled by physical laws in the form of differential equations. To train the employed neural network, we use tanh as the activation function, and some other key hyperparameters are listed in Table 2. Here, "Adam, L-BFGS" represents that the neural network was first optimized under the Adam algorithm for 1000 iterations, and then we switch to Limited-memory Broyden–Fletcher–Goldfarb–Shanno(L-BFGS) [31] for the remaining iterations.

Table 2. Designed neural network architectures for different working examples.

Examples	Hidden Layers	Neurons on Each Layer	Optimizer	Learning Rate	Iterations
Potential Flow over Cylinder	5	50	Adam, L-BFGS	0.001	30,000
Taylor–Green Vortex	5	30	Adam, L-BFGS	0.001	3000
Poisson Problem	5	40	Adam, L-BFGS	0.001	5000
Thermal Conduction	5	40	Adam, L-BFGS	0.001	10,000

4.1. Potential Flow Over Circular Cylinder

In this section, we present the application of the aforementioned algorithms to address the potential flow past cylinder problem. For this data-free approach, we do not need CFD calculated flow data used as inputs to calibrate the predicted results from neural networks. Instead, by exploiting the governing equation modeling the physical phenomena, we can ask the neural network to do self-learning by minimizing the customized errors instead of feeding them prior data used for supervised learning. Hence, if we know the governing equation, this method enables us to get rid of the lengthy and expensive CFD computations. It spins out neuron-generated solutions for the governing equations in a cheap and efficient way. Another big advantage of this promising technique that facilitates the study of various physical phenomena lies in the removal of the necessity of generating structural or non-structural meshes used for geometry discretion, which is a big challenge for modeling of transport phenomena in complex geometries.

One of the most basic problems in elementary fluid dynamics is to find the velocity potential and streamlines associated with uniform irrotational flow past a cylindrical obstacle. This benchmark problem for stationary, inviscid, and incompressible flow has obvious application to simplified problems in aerodynamics. The test configuration considers a solid cylinder centered at (0, 0) with the radius $R = 0.5$ in a $L = 3$ by $h = 2$ rectangular channel. The fluid is assumed to have a constant density equal to 1.29. As the flow is assumed to be irrotational and steady; hence there exists a velocity potential $\varphi = \varphi(x, y)$ such that

$$\vec{V} = \nabla\varphi$$

where $\vec{V}$ is the velocity vector. The velocity components in the x and y directions can be obtained by

$$u_x = \frac{\partial\varphi}{\partial x} \quad \text{and} \quad v_y = \frac{\partial\varphi}{\partial y}$$

The relationship between potential and velocity can be expressed by Laplace's equation as:

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = 0 \tag{22}$$

For a uniform flow with $U_\infty = 1.0$, as shown in Figure 3, the analytical solutions in Polar and Cartesian coordinates are

$$\varphi = \left(r + \frac{R}{r}\right) U_\infty \cos\theta \tag{23}$$

$$u_r = \left(1 - \frac{R^2}{r^2}\right) U_\infty \cos\theta \tag{24}$$

$$v_\theta = -\left(1 + \frac{R^2}{r^2}\right) U_\infty \sin\theta \tag{25}$$

$$u_x = 2U_\infty \sin\theta \sin\theta \tag{26}$$

$$v_y = -2U_\infty \sin\theta \cos\theta \tag{27}$$

The designed neural network for this cylinder flow problem is shown in Figure 4. Here the network consists of five hidden layers with each layer being 50 neurons. The Sigmoid function was chosen as the activation function. R_{PDE} is the residues of Laplace's equation, which measures the difference between neural network predictions and the exact solution for each sampling data point. Meanwhile, the boundary conditions are considered by including the residual R_{BC}, which quantifies the closeness of neural network predicted flows at each boundary to the imposed boundary conditions. Small values of both R_{PDE} and R_{BC} are desirable. During the training process, both these residuals were minimized via the stochastic gradient descent algorithm. Ideally, R_{PDE} and R_{BC} are expected to infinitely approach zero. In reality, it is generally accepted that an extremely small value, say 10^{-3}, indicates the converged and reliable prediction results.

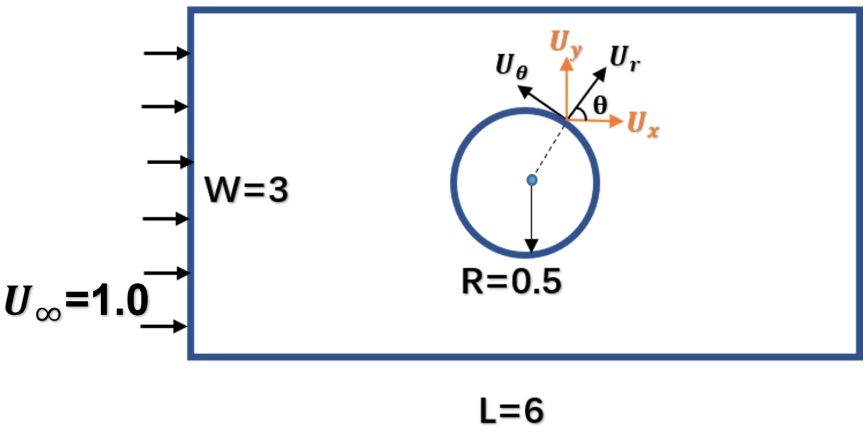


Figure 3. Schematic for potential flow over circular cylinder.

To approximate the flow field inside the computational domain, 2000 spatial points were randomly sampled inside the domain, as is shown in Figure 5. Here the predicted flow field is used to evaluate how well it satisfies the Laplacian equation. The residue R_{PDE} at each point was summed up to quantify the deviation of the prediction from true solutions. The satisfaction of the imposed boundary conditions was evaluated at the 400 data points sampled at the boundaries, i.e., the left, right, bottom and top boundary of the rectangular domain and the surface of the cylinder.

With 30,000 iterations, the training loss was reduced to the magnitude of 10^{-3}. To test the accuracy of the network's prediction capacity, 800 randomly sampled spatial points were used, as shown in Figure 5. The predicted flow potential is shown in Figure 6. The coincidence between blue and orange points indicates the predicted results approximate the

analytical results with high accuracy. The deviation between these two datasets is the source of the prediction error. As to our concern, the neural network gives a pretty good prediction, as indicated by the small deviations between these test points.

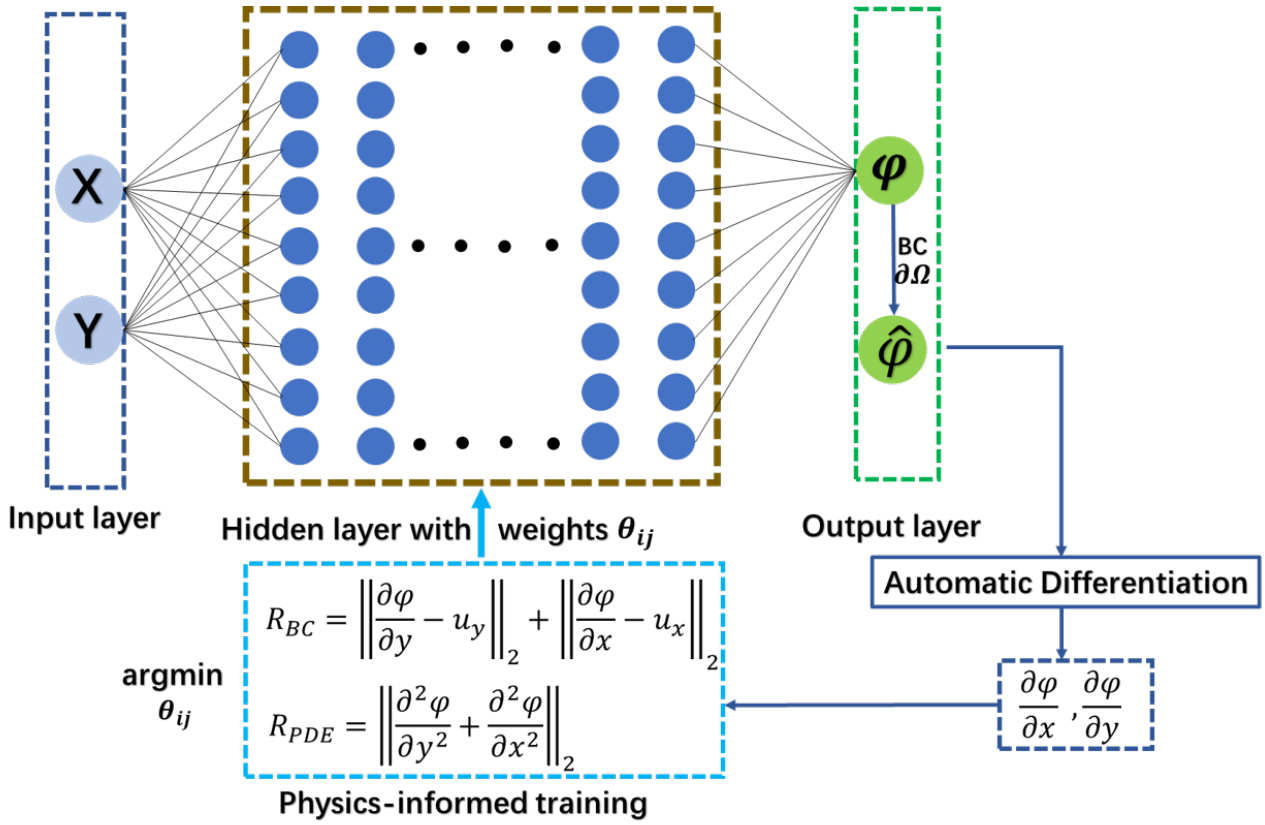


Figure 4. Schematic of constructed physics-based neural network architecture. The input layer has two neurons as placeholders for data coordinate (x, y), and the output layer has a single for the predicted velocity potential. N refers to the number of hidden layers, and each hidden layer is composed of N_n neurons. Each neuron (blue dot) is connected to the nodes of the previous layer with adjustable weights and bias. BC denotes the boundary conditions specified in Figure 3.

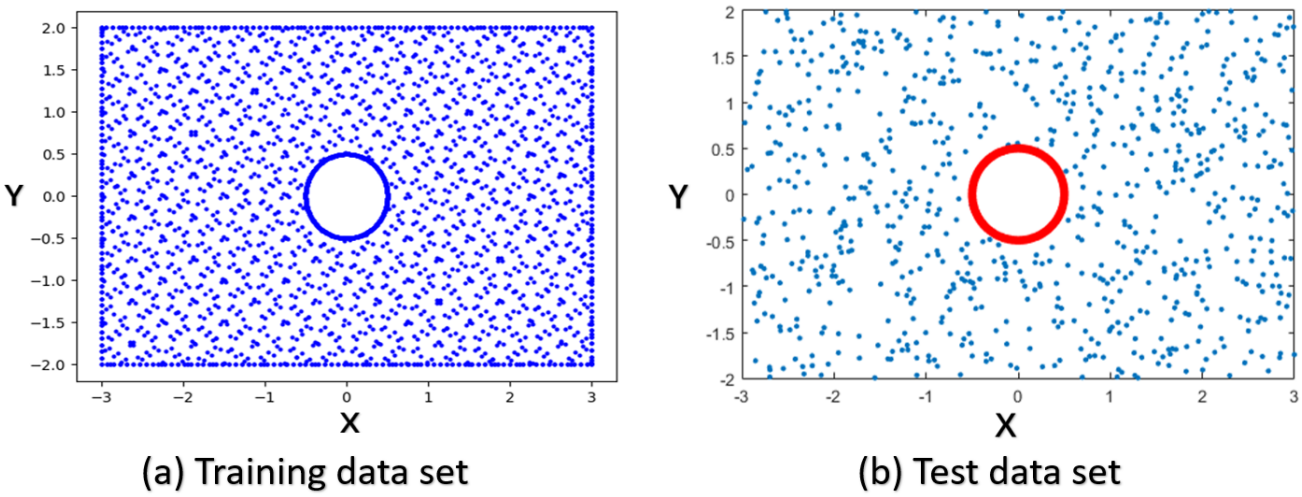


Figure 5. Randomly sampled data points across the computational domain for training (**a**) and test purpose (**b**).

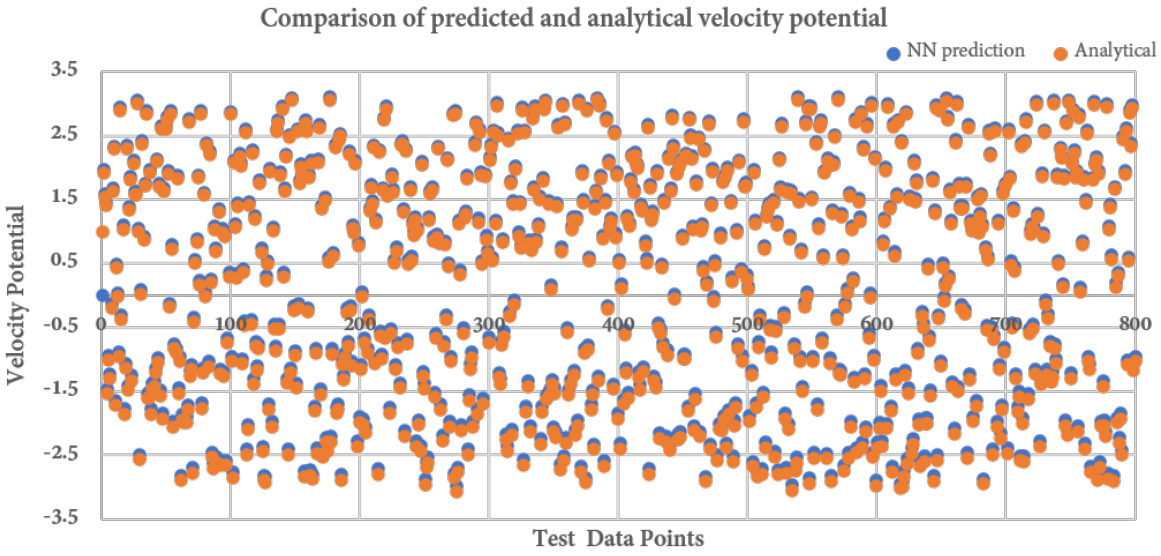


Figure 6. Velocity potential comparison between predicted and analytical results. The blue and orange dots denote the neural network predicted results and the analytical results. The horizontal axis shows the number ID of sampled points and vertical axis is the value of velocity potential.

Further evidence including streamline and vector field is also provided to double check the outcomes of the neural network predictions. Figures 7 and 8 show the streamline and velocity field predicted by the employed neural network. The velocity magnitude exhibits two lines of symmetry. A line drawn horizontally through the cylinder divides the velocity magnitude into upper and lower sides that are geometric mirror images. Note that the velocity itself is not symmetric about this line. These results lead to the conclusion that the employed neural network has excellent capacity to predict this flow phenomenon with high accuracy.

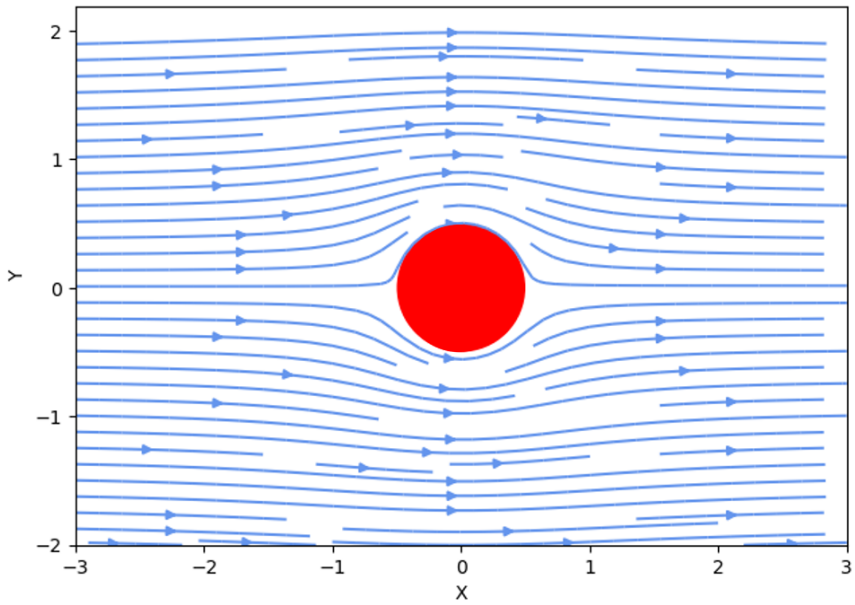


Figure 7. Neural network predicted streamline.

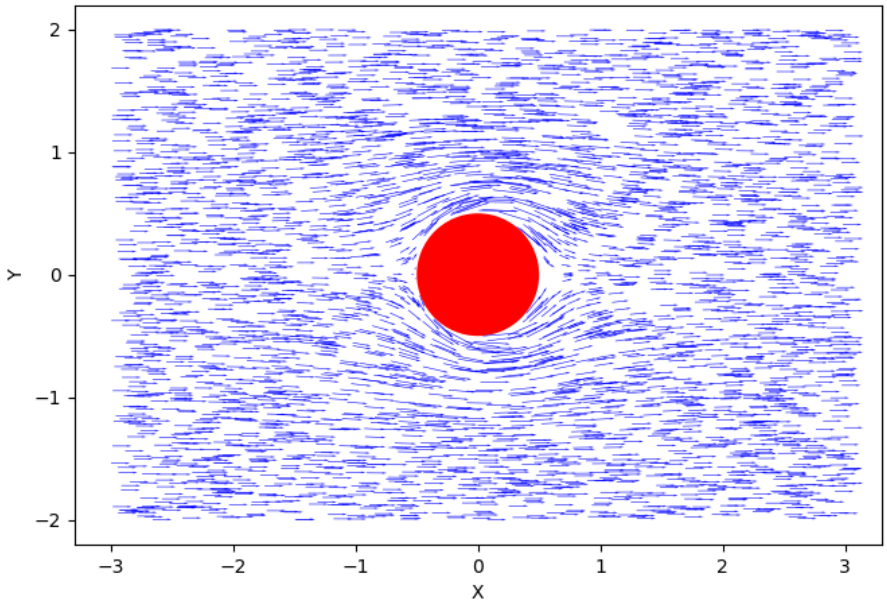


Figure 8. Neural network predicted velocity.

Using Bernoulli's equation, we can obtain the pressure coefficient

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho U_\infty^2}$$

Even if this inviscid flow case is simple, the predicted results can enable people to have good estimates of the pressure and velocity distribution as the pressure and velocity distribution are related via Bernoulli's equation. It should be noted that for real fluid flows past a cylinder, we have to consider viscous effects, which cause the flow to separate away from the cylinder, and the streamlines are no longer attached to the cylinder body.

Figure 8 shows predicted flow velocity. From this figure, we know that along the surface of the cylinder, flow velocity is in a tangential direction, i.e., parallel to the surface of the cylinder. As the mainstream flow gets close to the cylinder, fluid elements begin to decelerate. There is a stagnation point as the fluid element on the surface of the upstream side of the cylinder is stopped. Due to the zero velocity at stagnation point, pressure increases to its maximum value, while the pressure coefficient reaches its maximum. Figure 9 shows the pressure coefficient distribution along the cylinder surface. The estimated pressure coefficients from neural networks match well with the analytical results. For fluid elements passing either above or below the cylinder, their velocity magnitudes increase due to the narrowing flow path. For this inviscid flow, flow velocity decreases as flow continues around the downstream side of the cylinder, producing a second stagnation point at the downstream equator. Further downstream, flow velocity begins to increase and gradually returns to the free stream value.

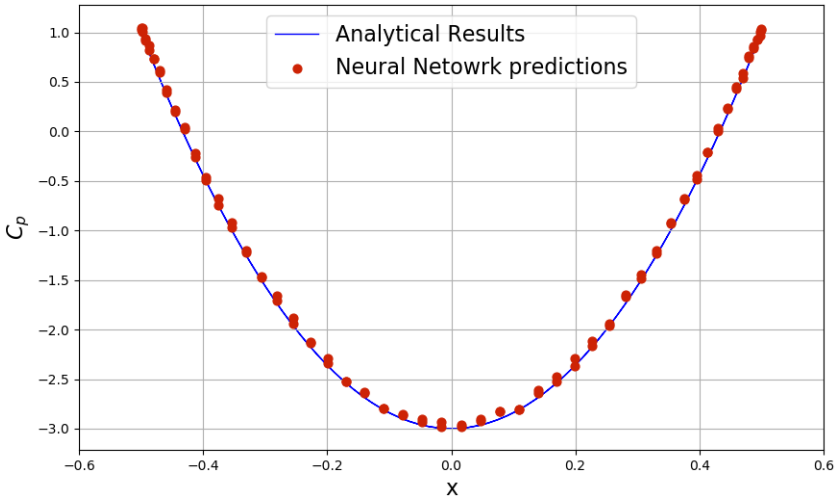


Figure 9. Predicted pressure coefficient.

4.2. Taylor–Green Vortex

The Taylor–Green Vortex (TGV) is an unsteady flow of a decaying vortex and was firstly solved by Taylor and Green by a perturbation series in time to explain the creation of small scales by vortex-stretching, diffusion, and dissipation in a three-dimensional (3D) flow field [32]. It has an analytical solution based on the incompressible Navier–Stokes equations in Cartesian coordinates and thus is widely used as a benchmark problem in validating solvers and formulations in numerical computations. Here the two-dimensional decaying vortex defined in the square domain, $0 \leq x, y \leq 2\pi$ serves as a benchmark problem for testing and validation of incompressible Navier–Stokes codes.

Without the presence of body force, the incompressible Navier–Stokes equation in the Cartesian coordinate system is given by:

$$\begin{aligned} &\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \\ &\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \\ &\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \end{aligned} \tag{28}$$

In the domain $0 \leq x, y \leq 2\pi$, the solution is given by [33]:

$$\begin{aligned} u &= e^{-2\nu t}\cos x \sin y \\ v &= -e^{-2\nu t}\sin x \cos y \end{aligned} \tag{29}$$

where ν is the kinematic viscosity of the fluid. The pressure field p can be obtained by substituting the velocity solution in the momentum equations and is given by

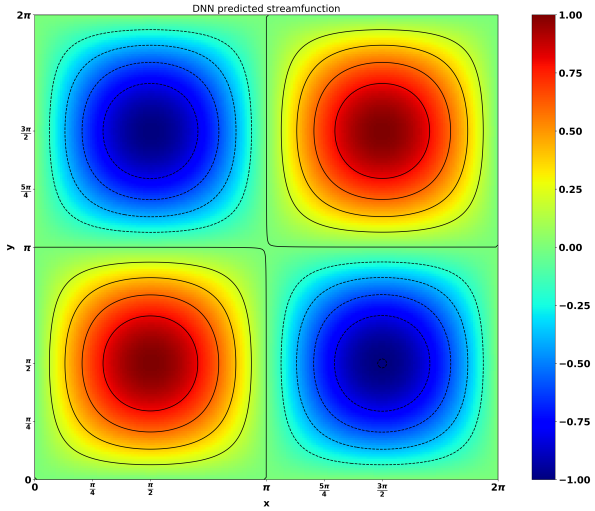
$$p = -\frac{\rho}{4}(\cos 2x + \cos 2y)e^{-4\nu t} \tag{30}$$

The stream function satisfying $v = \nabla \times \psi$ and the vorticity governed by $\omega = \nabla \times v$ can be expressed by the following equations:

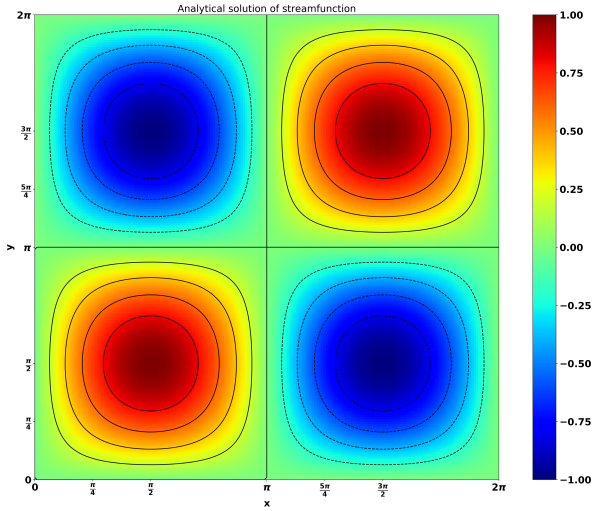
$$\begin{aligned} \psi &= e^{-2\nu t}\cos x \cos y \\ \omega &= -2e^{-2\nu t}\cos x \cos y \end{aligned} \tag{31}$$

The neural network predictions were trained using 2000 residual points that are randomly sampled in the spatio-temporal domain, and 300 and 600 points for the initial

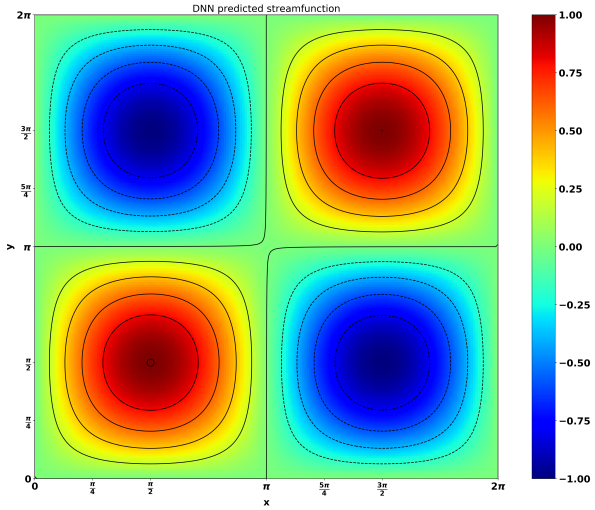
and boundary conditions, respectively. Figure 10 shows the result obtained by neural network predictions and analytical solutions of the viscous Navier–Stokes equation for the Taylor–Green vortex flow. The time evolutions (t = 1.0, 3.0 and 5.0) of the stream function and vorticity are presented in Figure 11a,b, respectively. There is a negligible difference between the neural network predicted results and the analytical solutions. This great consistency showcases the excellent capacity of neural networks for flow predictions.



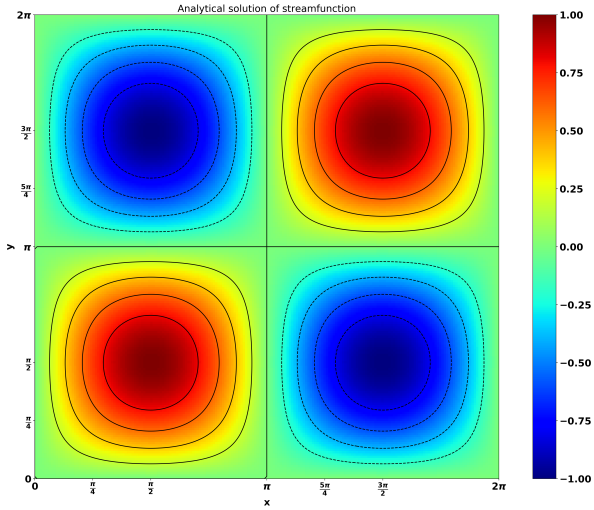
(**a**) PINNs predicted streamfunction at t = 1.0



(**b**) Analytical solution of streamfunction at t = 1.0

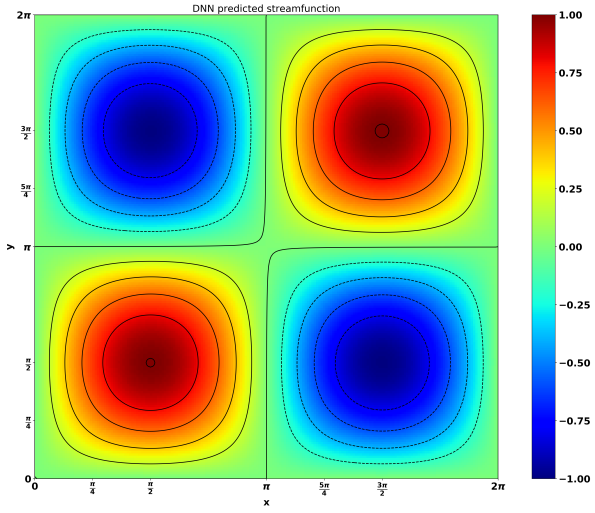


(**c**) PINNs predicted streamfunction at t = 3.0

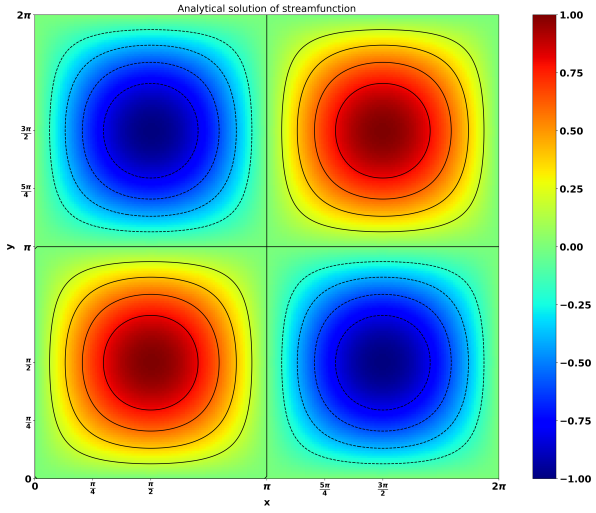


(**d**) Analytical solution of streamfunction at t = 3.0

Figure 10. *Cont.*



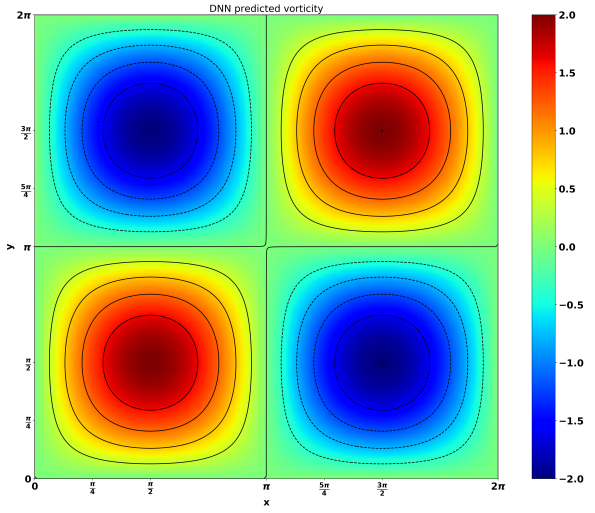
(**e**) PINNs predicted streamfunction at t = 5.0



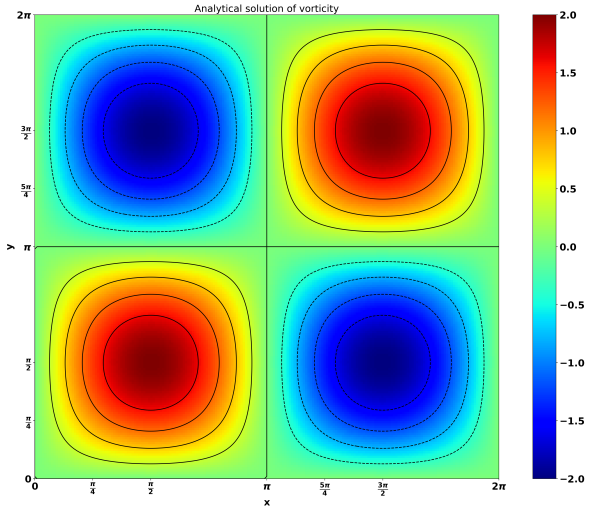
(**f**) Analytical solution of streamfunction at t = 5.0

Figure 10. Predicted stream functions from PINNs (**left**) and analytical results (**right**).

The above figures also show multiple well-defined laminar vortices and their interactions and evolutions in time. The TGV flow is initially characterized by the set of laminar, well-defined, and symmetric vortices, which evolve and interact in time, leading to vortex stretching mechanisms generating vortex sheets which gradually get closer. In summary, this study reinforces the potential of the proposed machine learning method to calculate transitional flows of practical interest efficiently [34]. Our approach provides an efficient computational alternative to allow scientists and engineers to use this TGV flow as a test-case to study more complicated transition to turbulence driven by vortex-stretching and reconnection mechanisms.

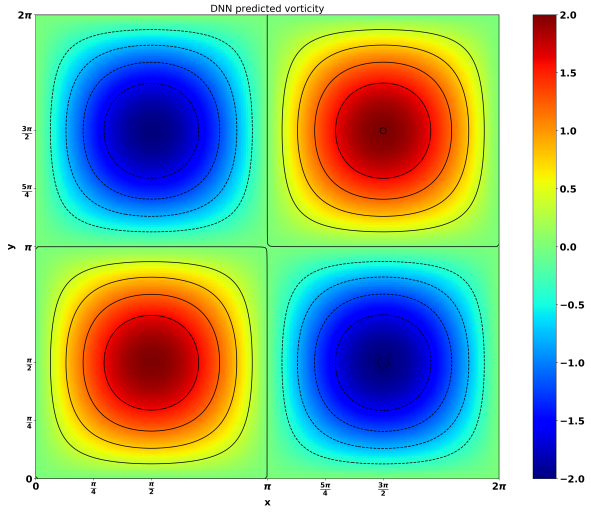


(**a**) PINNs predicted vorticity at t = 1.0

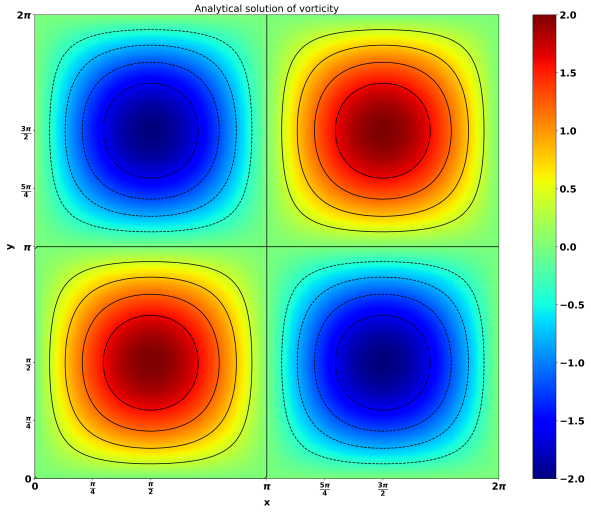


(**b**) Analytical solution of voriticity at t = 1.0

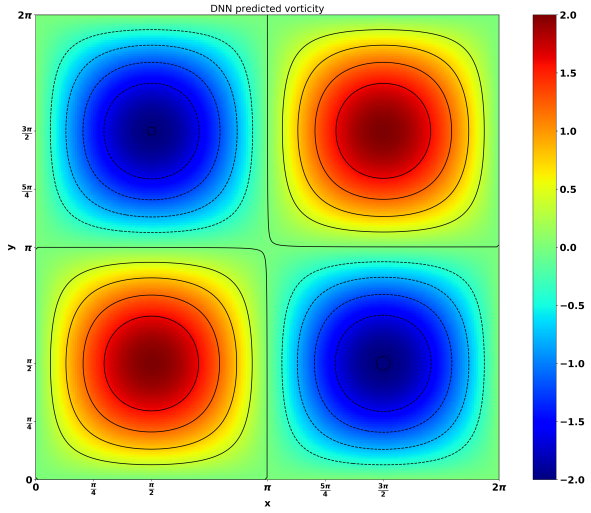
Figure 11. *Cont.*



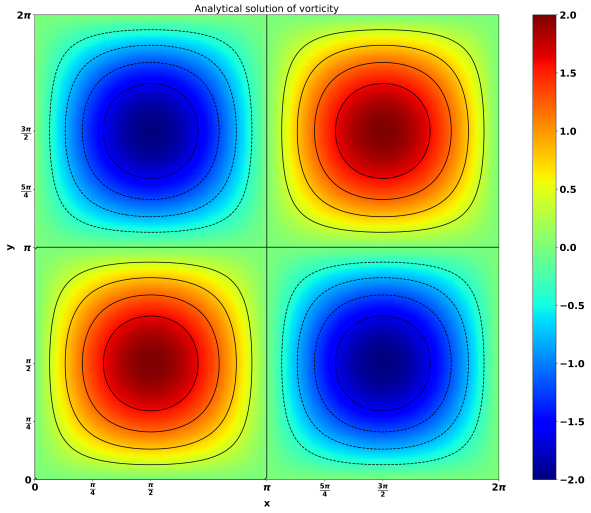
(**c**) PINNs predicted vorticity at t = 3.0



(**d**) Analytical solution of voriticity at t = 3.0



(**e**) PINNs predicted vorticity at t = 3.0



(**f**) Analytical solution of vorticity at t = 5.0

Figure 11. Predicted vorticity from PINNs (**left**) and analytical results (**right**).

4.3. Linear Poisson Problem

The accuracy and efficiency of the proposed technique were tested in the following two-dimensional inhomogeneous partial differential equation:

$$\begin{aligned} & \nabla^2\phi = \sin(\pi x)\sin(\pi y) \\ \text{subject to}: \quad & 0 \leq x \leq 1 \\ & 0 \leq y \leq 1 \end{aligned} \tag{32}$$

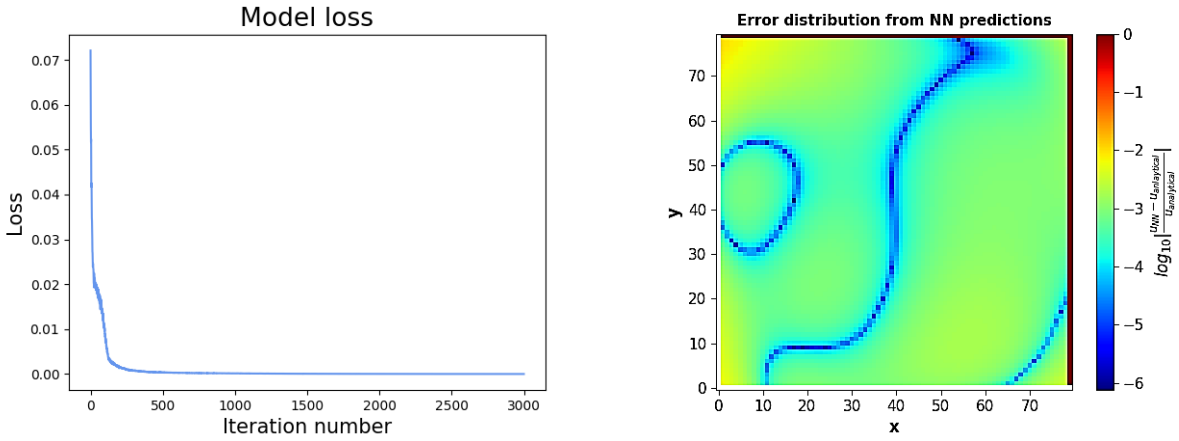
$\phi = 0$ along the whole squared boundary. In this case, we use a network of five layers with 30 neurons on each layer to predict flow potential ϕ. The physical law and boundary conditions (Equation (32)) were incorporated into the designed neural networks. After training, the loss history and prediction error distribution are shown in Figure 12. This error plot is calculated based on the relative difference between the PINN predictions and the analytical solutions presented in Equation (33). The error contours show that the achieved mean errors are around 10^{-3} and the minimal error can be as small as 10^{-6}.

The analytic solution is found to be

$$\phi = -\frac{1}{2\pi^2}\sin(\pi x)\sin(\pi y) \tag{33}$$

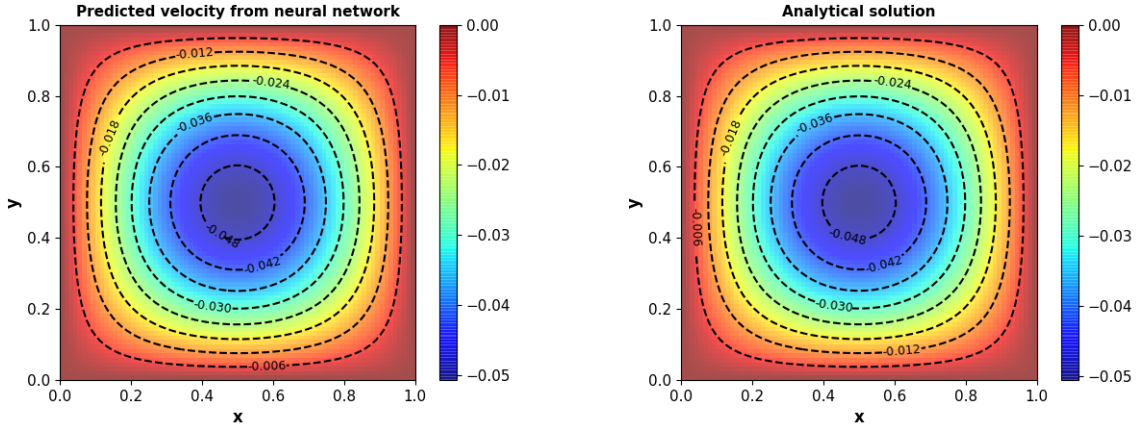
The accuracy of network predictions can be clearly illustrated as in Figure 13 by comparing the contour plot of PINN predicted unknown u and analytical solutions. Differences can be hardly spotted as the prediction accuracy is high enough. Meanwhile, the isovalue lines of u value, as represented by the dashed dark lines, are superposed onto the contour plot to quantify the parameter distribution as well as for better visualization.

In this problem, ϕ is a scalar potential which is to be determined, and the right-hand side of Equation (32) is a specific source function. Poisson's equation shows linear property in both the potential and the source term; hence its solutions are completely superposable. Moreover, Figure 13 shows that equipotential sets of the solution graph become smoother as the potential increases.



(**a**) Decreasing loss over iteration (**b**) Error distribution

Figure 12. The loss history (**a**) and error contour (**b**) of neural network predictions for Poisson equation.



(**a**) PINNs prediction (**b**) Analytical solution

Figure 13. Comparisons between the PINN prediction (**a**) and the analytical solution (**b**).

4.4. Thermal Conduction with Non-Linear Heat Generation

The capability of the developed neural network scheme for non-linear problems is also illustrated by a non-linear heat generation problem, where an unsteady temperature

distribution in a homogeneous solid is predicted. Temperature field is governed by the following equation:

$$\begin{aligned} \rho c_p \frac{\partial T}{\partial t} &= \frac{\partial}{\partial x}\left(\kappa(T)\frac{\partial T}{\partial x}\right) \\ \text{subject to:} \quad & u(a,t) = \phi(t), u(b,t) = \psi(t), \quad \forall t > 0 \\ & u(x,0) = u_0(x), \quad x \in [a,b] \end{aligned} \tag{34}$$

where the $T(x,t)$ is the temperature at point x and time t, ρ is the density, c_p is the heat capacity under constant pressure, and κ is the thermal conductivity of the selected media. Here c_p and ρ are assumed to be constants while κ varies with medium temperature. After some differential operation, Equation (34) is transformed into the following form:

$$\rho c_p \frac{\partial T}{\partial t} = \partial_T \kappa(T)\left(\frac{\partial T}{\partial x}\right)^2 + \kappa(T)\frac{\partial^2 T}{\partial x^2} \tag{35}$$

If κ takes a constant value, i.e., $\partial_T \kappa(T) = 0$, then Equation (35) becomes a linear (parabolic) partial differential equation. In contrast, when $\partial_T \kappa(T) \neq 0$, the Equation (35) is nonlinear, which can be solved numerically. For this initial-value (Cauchy) problem, the finite difference method is used for solution derivation with the implicit Euler scheme employed for temporal discretization [35].

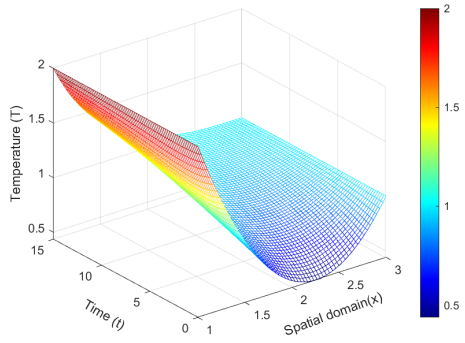
For illustration, the thermal conductivity is assumed to have the form of $\kappa = \kappa_0 exp(\chi T)$. The constant parameters have the value of $\kappa_0 = 0.1, \rho = c_p = 1$. The spatial domain $x \in [1,3]$ includes a boundary condition

$$T(1,t) = 2,\ T(3,t) = 1 \qquad \forall t > 0 \tag{36}$$

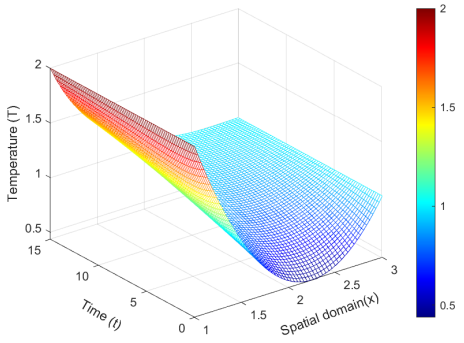
The initial condition of temperature is

$$T(x,0) = 2 - \frac{x-1}{2} + (x-3)(x-1) \tag{37}$$

With the aforementioned boundary and initial conditions, the time evolution of Equation (35) can be solved. Figure 14 shows both the results from neural network predictions and the numerical computations using finite difference (FD) method [35].

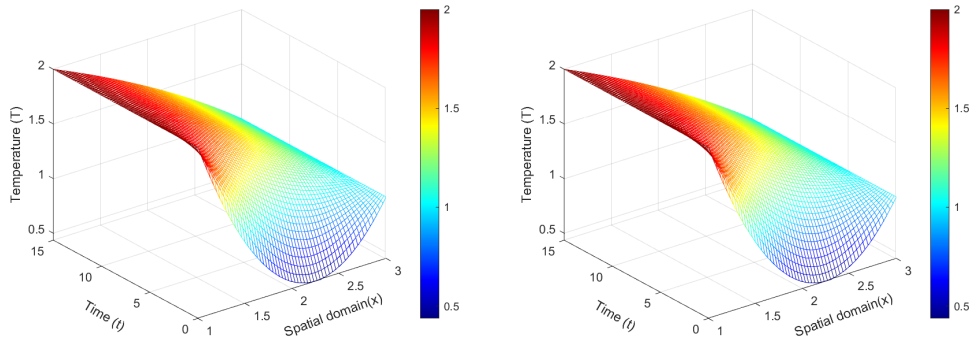


(**a**) PINNs prediction for $\chi = -1.0$

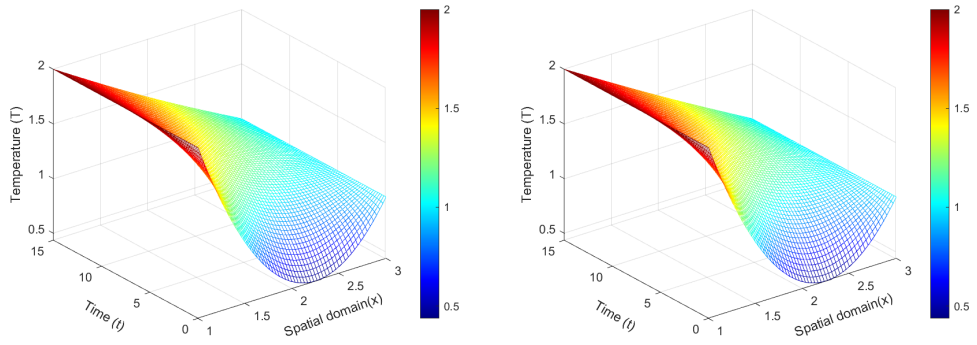


(**b**) FD calculation for $\chi = -1.0$

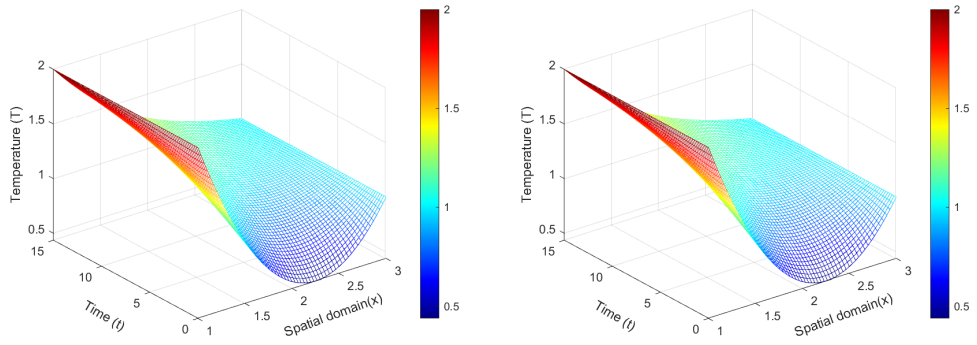
Figure 14. *Cont.*



(**c**) PINNs prediction for $\chi = -0.5$ (**d**) FD calculation for $\chi = -0.5$



(**e**) PINNs prediction for $\chi = 0$ (**f**) FD calculation for $\chi = 0$



(**g**) PINNs prediction for $\chi = 0.5$ (**h**) FD calculation for $\chi = 0.5$

Figure 14. Comparisons between PINN prediction and numerical calculations.

5. Conclusions

This paper presents a new solution framework based on physics-constrained machine learning that can be used to solve partial differential equations in fluid dynamics and thermodynamics. By leveraging prior physical laws, our feed-forward fully-connected neural networks are capable of solving physical equations commonly seen in fluid dynamics efficiently. Instead of using collocation points to discretize the spatial and temporal domains to find solutions, randomly sampled points were employed to evaluate their satisfactions of the desired physical equations. Automatic differentiation was adopted to handle differential operators, enabling this approach to be mesh free and time efficient. Since random points are generated on spatial domains, this randomness helps to capture complex physics in irregular computational domains. Thus, this mesh-free method is particularly attractive for problems with complex computations domains. The approximate solutions satisfying the differential operator can be obtained via tuning the deep neural network parameters, which are trained by minimizing the squared residuals over the entire

computational domain. In particular, the initial and boundary conditions are satisfied in a weak sense by imposing related penalty terms to the loss function. This modified loss function is used as the objective for minimization. The effectiveness and robustness of this proposed method have been illustrated via solving the flow past cylinder problem, linear Poisson problem, heat conduction and Taylor–Green vortex problems. The proposed method is relatively simple to implement, and provides a good tool for engineers and scientists to develop, test, and analyze their ideas.

While the proposed PINNs have great potential as an alternative solver for some physical problems, there is a long way to go to replace the traditional numerical method. For real problems with a large and complex computational domain, PINNs are still slower than the finite element method. Another challenge lies in their incapacity to address some multi-physics and multi-scale problems, which typically require heavy computations. Last but not the least, the design and construction of effective neural network architectures remain to be a headache as different users may build different neural network structures, which unquestionably imposes substantial impacts on the performance of PINNs as well as the accuracy of the predicted results. As further extensions to this work, we aim to tackle these challenges to facilitate this physics-based machine learning approach to tackle more complex problems in thermodynamic fluid dynamics. We will use emerging meta-learning techniques to automate the design of more efficient neural network structures and propose some customized loss functions for different tasks.

Author Contributions: Conceptualization, Y.S. and K.Q.; methodology, Y.S.; code development, Y.S. and Q.S.; validation, Y.S.; formal analysis, Y.S. and K.Q.; writing—original draft preparation, Y.S. and Q.S.; writing—review and editing, Y.S. and K.Q.; visualization, Q.S. and Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

List of Abbreviations

The following abbreviations are used in this manuscript:

Acronym	Full name
NN	Neural network
PINN	Physics-informed neural network
BC	Boundary condition
PDE	Partial differential equation
DGM	Deep Galerkin method
MLP	Multilayer perceptions
KS	Kuramoto-Sivashinsky
DNN	Deep neural network
MSE	Mean squared error
ReLU	Rectified linear unit
AD	Automatic differentiation
SGD	Stochastic gradient descent
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno
FD	Finite difference
TGV	Taylor–Green vortex
CFD	Computational fluid dynamics
3D	Three-dimensional

References

1. Hornik, K.; Stinchcombe, M.; White, H. Multilayer feedforward networks are universal approximators. *Neural Netw.* **1989**, *2*, 359–366. [CrossRef]
2. Lee, H.; Kang, I.S. Neural algorithm for solving differential equations. *J. Comput. Phys.* **1990**, *91*, 110–131. [CrossRef]
3. Lagaris, I.E.; Likas, A.; Fotiadis, D.I. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. Neural Netw.* **1998**, *9*, 987–1000. [CrossRef] [PubMed]
4. Smaoui, N.; Al-Enezi, S. Modelling the dynamics of nonlinear partial differential equations using neural networks. *J. Comput. Appl. Math.* **2004**, *170*, 27–58. [CrossRef]
5. Baymani, M.; Kerayechian, A.; Effati, S. Artificial Neural Networks Approach for Solving Stokes Problem. *Appl. Math.* **2010**, *1*, 288–292. [CrossRef]
6. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Physics Informed Deep Learning (Part I): Data-Driven Discovery of Nonlinear Partial Differential Equations. 2017. pp. 1–22. Available online: https://arxiv.org/abs/1711.10561 (accessed on 8 November 2021).
7. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Physics Informed Deep Learning (Part II): Data-Driven Discovery of Nonlinear Partial Differential Equations. 2017. pp. 1–19. Available online: https://arxiv.org/abs/1711.10566 (accessed on 8 November 2021).
8. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* **2019**, *378*, 686–707. [CrossRef]
9. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Inferring solutions of differential equations using noisy multi-fidelity data. *J. Comput. Phys.* **2017**, *335*, 736–746. [CrossRef]
10. Raissi, M.; Perdikaris, P.; Karniadakis, G.E. Machine learning of linear differential equations using Gaussian processes. *J. Comput. Phys.* **2017**, *348*, 683–693. [CrossRef]
11. Sirignano, J.; Spiliopoulos, K. DGM: A deep learning algorithm for solving partial differential equations. *J. Comput. Phys.* **2018**, *375*, 1339–1364. [CrossRef]
12. Lu, L.; Meng, X.; Mao, Z.; Karniadakis, G.E. DeepXDE: A deep learning library for solving differential equations. *arXiv* **2019**, arXiv:1907.04502.
13. Deng, L.; Hinton, G.; Kingsbury, B. New types of deep neural network learning for speech recognition and related applications: An overview. In Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, 26–31 May 2013; pp. 8599–8603. [CrossRef]
14. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the Inception Architecture for Computer Vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016.
15. Hirschberg, J.; Manning, C.D. Advances in natural language processing. *Science* **2015**, *349*, 261–266. [CrossRef] [PubMed]
16. Purwins, H.; Li, B.; Virtanen, T.; Schlüter, J.; Chang, S.Y.; Sainath, T. Deep Learning for Audio Signal Processing. *IEEE J. Sel. Top. Signal Process.* **2019**, *13*, 206–219. [CrossRef]
17. Ba, J.; Caruana, R. Do deep networks really need to be deep? *arXiv* **2014**, arXiv:1312.6184.
18. Chen, T.; Chen, H. Universal Approximation to Nonlinear Operators by Neural Networks with Arbitrary Activation Functions and Its Application to Dynamical Systems. *IEEE Trans. Neural Netw.* **1995**, *6*, 911–917. [CrossRef]
19. Dauphin, Y.N.; Bengio, Y. Big neural networks waste capacity. In Proceedings of the 1st International Conference on Learning Representations (ICLR 2013—Workshop Track Proceedings), Scottsdale, AZ, USA, 2–4 May 2013; pp. 1–5.
20. Erhan, D.; Courville, A.; Bengio, Y.; Vincent, P. Why does unsupervised pre-training help deep learning? *J. Mach. Learn. Res.* **2010**, *9*, 201–208.
21. Eckle, K.; Schmidt-Hieber, J. A comparison of deep networks with ReLU activation function and linear spline-type methods. *Neural Netw.* **2019**, *110*, 232–242. [CrossRef]
22. Viquerat, J.; Hachem, E. A supervised neural network for drag prediction of arbitrary 2D shapes in low Reynolds number flows. *arXiv* **2019**, arXiv:1907.05090.
23. Wickham, H. *ggplot2: Elegant Graphics for Data Analysis*, 3rd ed.; Springer: Berlin/Heidelberg, Germany, 2010.
24. Margossian, C.C. A review of automatic differentiation and its efficient implementation. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **2019**, *9*, 1–32. [CrossRef]
25. Güneş Baydin, A.; Pearlmutter, B.A.; Andreyevich Radul, A.; Mark Siskind, J. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.* **2018**, *18*, 1–43.
26. Ruder, S. An overview of gradient descent optimization algorithms. *arXiv* **2016**, arXiv:1609.04747.
27. Bottou, L. Stochastic Gradient Learning in Neural Networks. *Proc. Neuro-Nımes* **1991**, *91*, 12.
28. Darken, C.; Chang, J.; Moody, J. Learning rate schedules for faster stochastic gradient search. In Proceedings of the Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop, Helsingoer, Denmark, 31 August–2 September 1992.
29. Zinkevich, M.; Weimer, M.; Li, L.; Smola, A.J. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 23*; Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A., Eds.; Curran Associates, Inc.: New York, NY, USA, 2010; pp. 2595–2603.
30. Johnson, R.; Zhang, T. Accelerating stochastic gradient descent using predictive variance reduction. *Adv. Neural Inf. Process. Syst.* **2013**, *1*, 1–9.
31. Le, Q.V.; Ngiam, J. On Optimization Methods for Deep Learning Quoc. In Proceedings of the 28th International Conference on Machine Learning, Bellevue, WA, USA, 28 June–2 July 2011; pp. 129–132. [CrossRef]

32. Sharma, N.; Sengupta, T.K. Vorticity dynamics of the three-dimensional Taylor–Green vortex problem. *Phys. Fluids* **2019**, *31*, 035106. [CrossRef]
33. Kim, J.; Moin, P. Application of a fractional-step method to incompressible Navier–Stokes equations. *J. Comput. Phys.* **1985**, *59*, 308–323. [CrossRef]
34. Pereira, F.S.; Grinstein, F.F.; Israel, D.M.; Rauenzahn, R.; Girimaji, S.S. Modeling and simulation of transitional Taylor–Green vortex flow with partially averaged Navier–Stokes equations. *Phys. Rev. Fluids* **2021**, *6*. [CrossRef]
35. Filipov, S.; Faragó, I. Implicit Euler Time Discretization and Fdm With Newton Method in Nonlinear Heat Transfer Modeling. *Math. Model.* **2018**, *2*, 94–98.