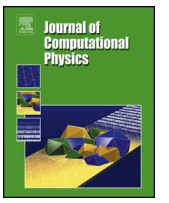
Journal of Computational Physics 504 (2024) 112865

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp



Multi-stage neural networks: Function approximator of machine precision

Yongji Wang [a,b,*], Ching-Yao Lai [b,*]

[a] Department of Mathematics, New York University, New York, NY 10012, United States of America
[b] Department of Geophysics, Stanford University, Stanford, CA 94305, United States of America

ARTICLE INFO

Keywords:
Scientific machine learning
Neural networks
Physics-informed neural networks
Multi-stage training

ABSTRACT

Deep learning techniques are increasingly applied to scientific problems, where the precision of networks is crucial. Despite being deemed as universal function approximators, neural networks, in practice, struggle to reduce the prediction errors below $O(10^{-5})$ even with large network size and extended training iterations. To address this issue, we developed the multi-stage neural networks that divides the training process into different stages, with each stage using a new network that is optimized to fit the residue from the previous stage. Across successive stages, the residue magnitudes decreases substantially and follows an inverse power-law relationship with the residue frequencies. The multi-stage neural networks effectively mitigate the spectral biases associated with regular neural networks, enabling them to capture the high frequency feature of target functions. We demonstrate that the prediction error from the multi-stage training for both regression problems and physics-informed neural networks can nearly reach the machine-precision $O(10^{-16})$ of double-floating point within a finite number of iterations. Such levels of accuracy are rarely attainable using single neural networks alone.

1. Introduction

Deep learning techniques [1] have been well developed in the fields of computer vision [2,3] and natural language processing [4–6]. More recently, neural networks have been increasingly applied to the mathematical and physical sciences [7–9], where the demand for precision is high. In particular, physics-informed neural networks (PINNs) [10,11] have emerged as a new class of numerical solver for partial differential equations, where computing high-precision solutions becomes an intrinsic requirement of the method.

Neural networks have been proven to be universal function approximators [12,13]. However, in practice, neural network training often falls into local minima [14,15], causing the training loss to plateau after a certain number of iterations n_{iters}. This issue cause the failure modes of PINNs [15]. Advanced methods focusing on different aspects, such as activation function selection [16–19], network configuration [20–23], optimization methods [24–26], trainable weights [27], physical causality [28], weighting of terms in the loss function [29–31], and hierarchical training strategy [32,33], have been developed to effectively enhance the convergence rate of the loss function for various problems. However, few of these methods manage to reduce the training error less than $O(10^{-5})$, or operate effectively in restrictive settings [23]. In contrast, classical numerical methods (e.g., finite difference) can systematically

* Corresponding authors at: Department of Geophysics, Stanford University, Stanford, CA 94305, United States of America.
E-mail addresses: yw8211@nyu.edu (Y. Wang), cyaolai@stanford.edu (C.-Y. Lai).

https://doi.org/10.1016/j.jcp.2024.112865
Received 11 July 2023; Received in revised form 10 February 2024; Accepted 14 February 2024

Available online 19 February 2024

enhance solution's accuracy by simply reducing the grid size [34]. This is a major shortcoming of neural networks for solving many problems within mathematical and physical sciences.

In this work, we propose the multi-stage neural networks (MSNN) that effectively addresses this limitation. Our novel method involves dividing the network training into multiple stages, where each stage incorporates a separate neural network. The settings of each network in a given stage are optimized based on the residues from the preceding stage. By executing training stage by stage, we significantly enhance the convergence rate, ensuring that it remains consistently high throughout the iterations. As a result, the combined neural networks from different stages can approximate the target function with remarkable accuracy, with the error approaching the machine precision $O(10^{-16})$ for double-floating point numbers. Furthermore, the multistage training scheme addresses the challenges faced by regular PINNs in solving multi-scale problems. This is because the stage-by-stage training is capable of capturing the spectrum of the target function, from low to sufficiently high frequencies, across a suitable number of stages.

We begin with the introduction of the multi-stage neural network for regression problems in Section 2. By exploring the limitations of classical neural network training, we highlight the benefits of the multi-stage training in overcoming these constraints. We then propose and substantiate the optimal settings for each training stage. In Section 3, we extend the method to physics-informed neural networks (PINNs) for solving differential equations. Unlike regression problems, the optimal settings for PINNs in each stage are implicitly tied to the equation residues of previous stages. Both theoretical investigation and practical algorithmic solutions are presented to address this challenge. Additional techniques that can expedite the multi-stage training for PINNs are also discussed. In Section 4, we generalize the multi-stage training scheme to solve combined-forward-and-inverse problems, which are of great importance in mathematical and physical sciences. Lastly, we provide further discussions on the challenges and potential development of the MSNN method in Section 5 and conclude the paper in Section 6.

2. Multi-stage training scheme for regression problems

We first illustrate the multistage idea with regression problems that involve predicting a continuous output variable u as a function of the input variable x. We train a neural network that represents $u(x)$ to fit N_d data points, denoted (x_i, u_i). The loss function for a regression problem is typically the mean squared error (MSE), defined as

$$\mathcal{L} = \frac{1}{N_d}\sum_{i=1}^{N_d}[u(x_i) - u_i]^2 . \tag{2.1}$$

In this study, we consider all training data with no noise.

2.1. Limitation of regular neural network training

To illustrate the limitation of neural network's function approximation capacity, we consider a target function,

$$u_g(x) = \sin(2x+1) + 0.2e^{1.3x}, \tag{2.2}$$

we created training data by sampling 300 data points from it with no noise, uniformly distributed within the domain $x \in [-1, 1]$. To fit the training data, we create a fully-connected neural network made of three hidden layers with 20 units in each layer and use hyperbolic tangent as the activation function for each unit. Using Adam [35] optimizer, Fig. 1(a) shows that the trained neural network $u_0(x)$ captures the target function $u_g(x)$ well. During the iterations, the training loss $\mathcal{L}$ based on mean squared error (MSE) between the data and network, decreases significantly at the early stage (Fig. 1b). However, after 5000 iterations, it reaches a plateau around $O(10^{-7})$ with very small convergence rate. The error function $e_1(x)$ between u_g and u_0 across the training domain is, thus, trapped around 10^{-4} (inset of Fig. 1a). Further experiments, elucidated in Appendix A, affirm that this plateau value of the error remains consistent even with larger networks and additional data, and not optimizer-specific.

Neural networks are known for their spectral biases [36], also referred to as the frequency principle [37]. Utilizing the tool of neural tangent kernels [38], prior studies [39,40] demonstrated that a standard multi-layer neural network struggles to learn the high frequencies of target functions in both theory and practice. The plateau of training loss corresponds to a mismatch between the trained network $u_0(x)$ and target function $u_g(x)$ at high frequencies. Fig. 1(b) demonstrates that the error function $e_1(x) = u_g(x) - u_0(x)$ within the domain is indeed a high-frequency function.

2.2. Key settings of multi-stage training scheme

Since training a single neural network struggles with learning the high frequencies of the target function, an intuitive approach is to train a second neural network to capture the error function $e_1(x)$, or the residue, between the training data and the first trained network [41]. The original training data from (2.2) is denoted with $(x^{(i)}, u_g^{(i)})$. The training data for the second neural network would be $(x^{(i)}, e_1(x^{(i)}))$, where $e_1(x^{(i)})$ denotes the error of network at $x^{(i)}$. At this point, extra care should be taken when setting up the second neural network, particularly concerning two key aspects.

2.2.1. Magnitude of the second neural network

Considering that the original training data has a magnitude of $O(1)$, then the training data for the second neural network, which is the residue $e_1(x)$, would be much smaller than 1 (Fig. 1b). We observe that a neural network employing regular weight initialization

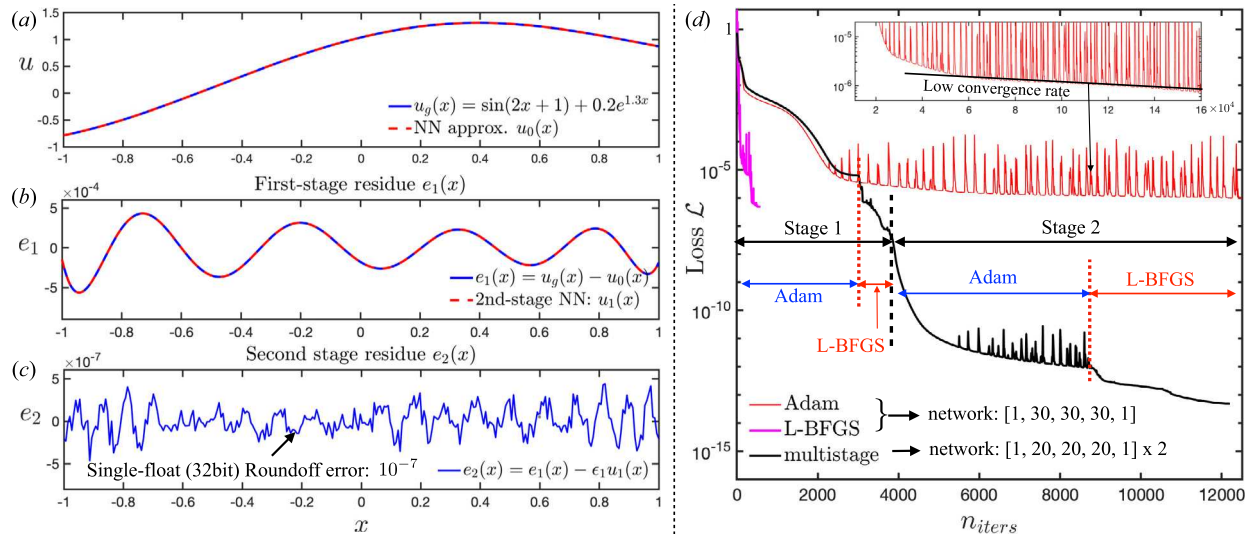


Fig. 1. Comparison of single-stage with multi-stage training. (*a*) Fitting of a neural network $u_0(x)$ with `tanh` activation function to the data from (2.2). (*b*) Fitting of second-stage neural network to the error $e_1(x)$ between the data from (2.2) and the first-stage trained network $u_0(x)$ as shown in (*a*). (*c*) The error $e_2(x)$ between the data and the sum of two-stage networks, which reaches the machine precision of a single float (32-bit). (*d*) Comparison of the loss convergence between a single-stage training (pink and red) and a two-stage training (black). For a single-stage training, the convergence rate of loss suddenly reduces (for Adam) after the loss reaches $O(10^{-6})$ or terminates (for L-BFGS). For a two-stage training, even with less number of weights and biases, the convergence rate is significantly faster than that for the single-stage training. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

methods, such as Xavier [42], often struggles to capture training data whose magnitude is significantly larger or smaller than 1 (see Appendix B). A straightforward solution to this issue is to normalize the training data by its root mean square value ϵ_1, defined as

$$\epsilon_1 = \mathrm{RMS}(e_1(x)) = \sqrt{\frac{1}{N_d}\sum_{i=1}^{N_d}[e_1(x^{(i)})]^2} = \sqrt{\frac{1}{N_d}\sum_{i=1}^{N_d}[u_g^{(i)} - u_0(x^{(i)})]^2}\,. \tag{2.3}$$

Then, the normalized training data for the second neural network becomes $(x^{(i)}, e_1(x^{(i)})/\epsilon_1)$. Denoting the second trained network as $u_1(x)$, the combined networks for the original data become

$$u_c^{(1)}(x) = u_0(x) + \epsilon_1 u_1(x)\,. \tag{2.4}$$

Subsequently, we can continue training the third or even further neural networks to reach higher accuracy for our model. The training data for the $(n+1)$-th neural network u_n is the residue e_n between the original training data u_g and the output of the previously combined n neural networks, $u_c^{(n-1)}(x^{(i)})$, normalized by its own magnitude (root mean square value) ϵ_n, namely $(x^{(i)}, e_n(x^{(i)})/\epsilon_n)$. Then, the final model that combines all the $(n+1)$ neural networks reads,

$$u_c^{(n)}(x) = \sum_{j=0}^{n} \epsilon_j u_j(x)\,, \tag{2.5}$$

where ϵ_i stands for the magnitude for the i-th neural network. When the original training data u_g is normalized, ϵ_0 is set to be 1.

2.2.2. *Frequency of the second neural network*

Even with normalization, the second neural network, if initialized with regular weights, could still struggle to fit the high-frequency data due to the inherent spectral biases of neural networks. To illustrate this, we consider a target function

$$u(x) = \left(1 - \frac{x^2}{2}\right)\cos\left[m\left(x + 0.5x^3\right)\right], \tag{2.6}$$

with m the free parameter related to the frequency of the function. Fig. 2 shows the function (2.6) for $m = 3$, 15, and 30, respectively. For each m, we generate 300 sample points (x_i, u_i) that satisfy (2.6) as our training data, with x_i uniformly distributed in the domain $[-1, 1]$. Fig. 2 shows that the neural network, using regular weight initialization, fits the data well for $m = 3$, partially misses the data for $m = 15$, and completely fails to fit the data for $m = 30$.

To understand the challenge in fitting high-frequency data, let's consider a shallow neural network with a single input, single output, and one hidden layer that uses the hyperbolic tangent as its activation function:

$$u(x) = \sum_{i=1}^{N} w_i^{(1)} \tanh\left(w_i^{(0)} x + b_i^{(0)}\right) + b_0\,, \tag{2.7}$$

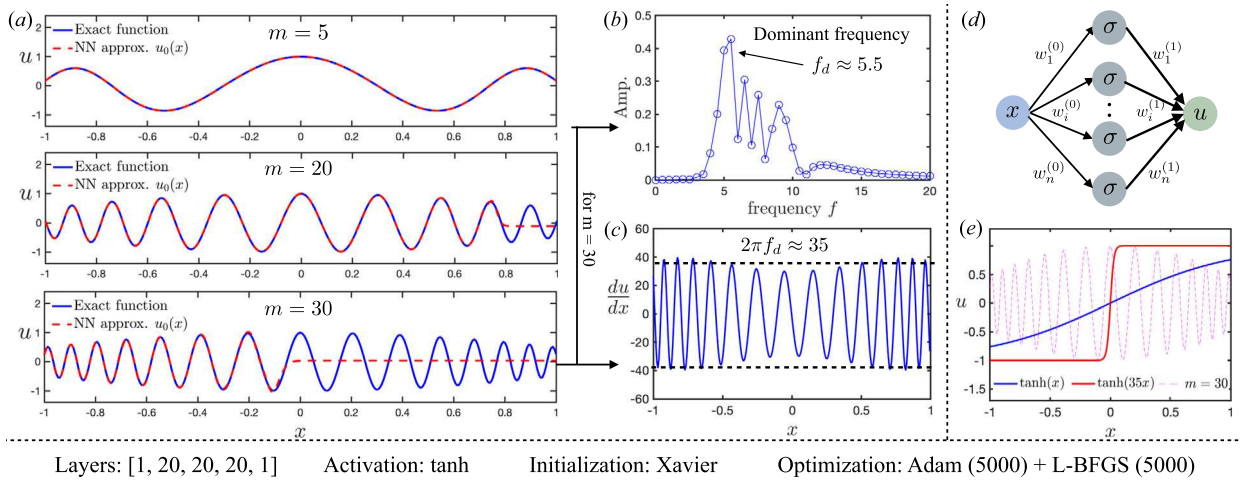


Fig. 2. Spectral biases of neural networks. (*a*) Fitting of neural networks with `tanh` activation function to the data from (2.6) for different m. Under regular settings, neural networks have difficulty fitting high-frequency functions. (*b*) Frequency domain of the function (2.6) for $m = 30$ with the dominant frequency $f_d = 5.5$. (*c*) Derivative of the function (2.6) du/dx for $m = 30$, which scale as $O(2\pi f_d)$. (*d*) Schematic diagram of a single-hidden layer neural network. (*e*) Comparison between single-neuron outputs for different weights $w^{(0)}$ within the `tanh` activation function and the function (2.6) for $m = 30$. To capture high-frequency functions, it shows that the weight within the activation function needs to increase from $O(1)$ to $O(2\pi f_d)$.

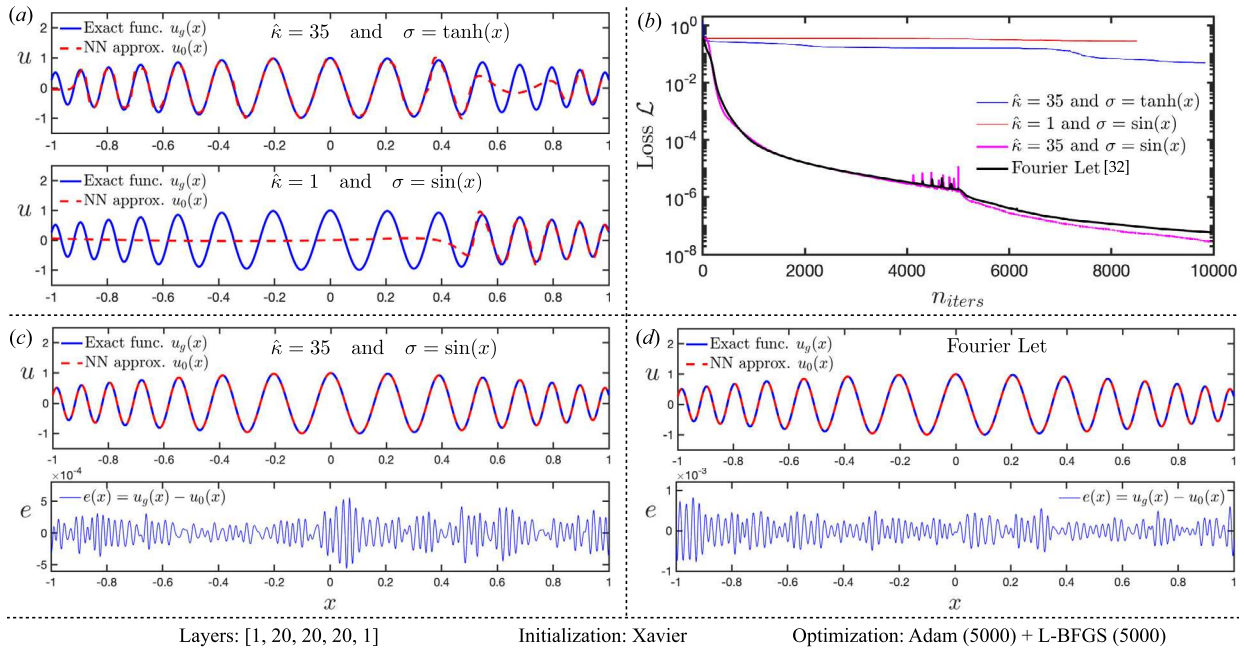


Fig. 3. Neural network settings for high-frequency functions. (*a*) Fitting of a neural network to the data from (2.6) with $m = 30$, by either changing the activation function for the first hidden layer to $\sin(x)$, or multiplying the weight $w_i^{(0)}$ before the first hidden layer by a scale factor κ. None of them captures the data. (*b*) Comparison of training loss for the neural networks with different settings. (*c*) Neural network using $\sin(x)$ activation function and modified scale factor $\hat{\kappa} = 35$ fits well the high-frequency data (2.6) with $m = 30$, reaching the same accuracy as (*d*) Fourier Let network [39].

where $w_i^{(0)}$ denote the weights between the input and hidden layers, and $w_i^{(1)}$ are the ones between the hidden and output layers. $b_i^{(0)}$ is the bias for the hidden units and b_0 is the bias for the output unit. The magnitude of the output function is determined by $w_i^{(1)}$, while $w_i^{(0)}$, within the activation function, influence the local gradient of the function (Fig. 2*e*). Common practice involves initializing the weights of the network to follow a Gaussian distribution with zero mean and a specified variance. For example, Xavier initialization uses a specified variance $V_{ar} = 2/(N_{l-1} + N_l)$, where N_{l-1} and N_l are the number of units in the preceding and succeeding layers, respectively. This initialization ensures that the variance of the sum of all unit outputs in each hidden layer remains $O(1)$, which prevents gradient vanishing or explosion during the training. However, a side effect of this approach is that the neural network becomes a slowly varying function with respect to normalized inputs.

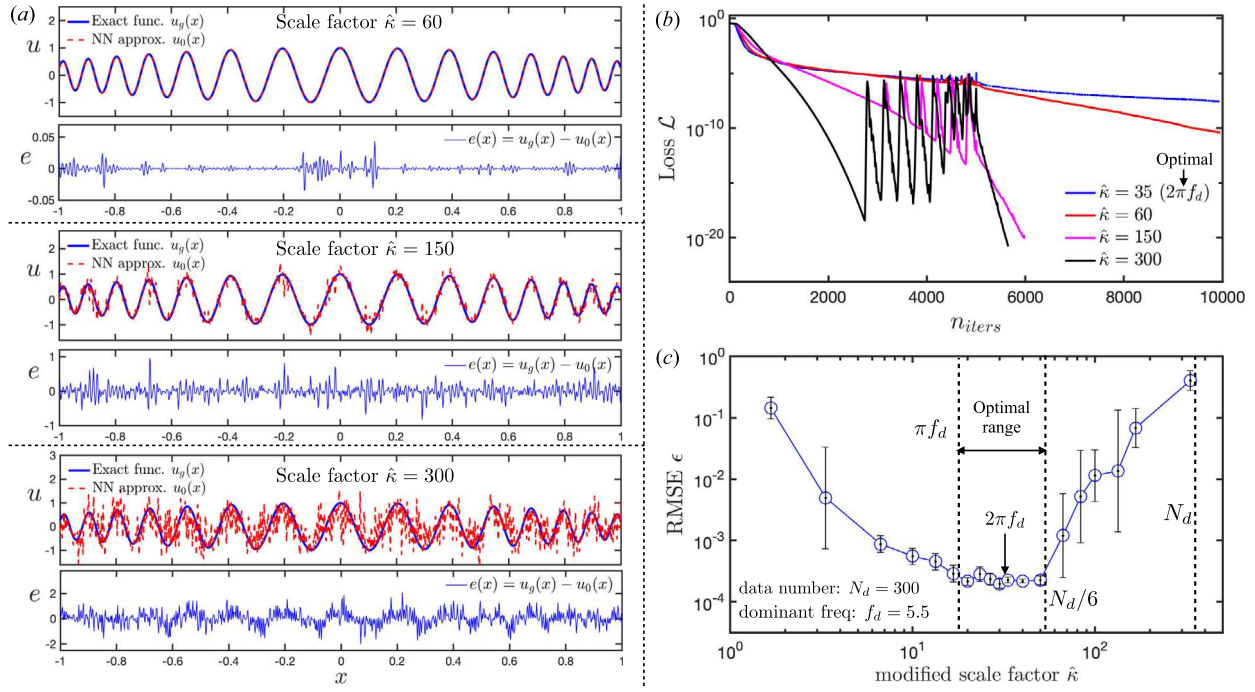


Fig. 4. Importance of the modified scale factor $\hat{\kappa}$. (*a*) Fitting of neural networks to the data from (2.6) for $m = 30$ using different modified scale factor $\hat{\kappa}$. The networks start overfitting the data when $\hat{\kappa} \geq 60$. (*b*) Training loss for the neural networks with different modified scale factors $\hat{\kappa}$. When $\hat{\kappa} \geq 60$, the training loss decreases significantly fast due to over-fitting. (*c*) Relation of the root mean square value ϵ of the error $e(x)$ between the trained network $u(x)$ and target function $u_g(x)$ with the modified scale factor $\hat{\kappa}$. The minimal error is reached when $\pi f_d < \hat{\kappa} < N_d/6$, where f_d denotes the dominant frequency and N_d the total number of training data points.

For a high-frequency function with normalized input and output, and a dominant frequency f_d, the magnitude of its gradient scales as $O(2\pi f_d)$ (Fig. 2*c*). To capture these large gradients, considering a one-hidden layer network with single input and output (2.7), the weights $w_i^{(0)}$ within the activation function need to increase from their initialized value of $O(\sqrt{V_{ar}})$ to $O(2\pi f_d)$ during training. This large shift in weight values, particularly for large f_d, leads to slower convergence during training or an inaccurate approximation of the data.

To address this issue, we multiply weights within the activation function by a large scale factor κ [18] to expedite the convergence of weights towards their optimal high value when fitting high-frequency data. We only multiply the scale factor κ to the weights between the input and the first hidden layer, rather than all weights, to prevent gradient explosion [43] during the training.

Besides large gradient, high-frequency functions also have a large amount of inflection points. In contrast, the hyperbolic tangent function, being a monotonic function, struggles to capture this feature. Periodic functions, such as the sine or cosine function, are more suitable choices for activation functions in this case [16]. In our approach, we use the sine function solely for the first hidden layer while retaining the hyperbolic tangent function for the remaining layers. This combination allows us to capture both low and high-frequency data effectively.

Fig. 3 illustrates the impact of the scale factor κ and the choice of activation function on improving the fit for high-frequency data. Using a combination of the scale factor κ and sine function for the first hidden layer yields the best training result. This combination equates to applying a Fourier feature mapping (i.e. Fourier let network) [39] to the input before it is passed through the multi-layer network. Fig. 3(*c* & *d*) compares the convergence rate and final error of both methods when fitting the same high-frequency data. The results are consistently good, verifying the efficacy of both methods in fitting high-frequency data.

To expedite the convergence of weights from their initialized value $O(\sqrt{V_{ar}})$ to the high gradient value $O(2\pi f_d)$ of a high frequency function, the optimal value of κ is expected to depend on the variance V_{ar}, which is related to the weight initialization approaches, and the size of neural network. To isolate the impact of V_{ar} on determining the optimal value of the scale factor, we introduce a modified scale factor $\hat{\kappa}$ as,

$$\kappa = \hat{\kappa}/\sqrt{V_{ar}} \qquad \Longrightarrow \qquad \hat{\kappa} = \kappa\sqrt{V_{ar}}\,. \tag{2.8}$$

Fig. 4(*c*) shows that the minimal fitting error is achieved when the modified scale factor is

$$\hat{\kappa} > \pi f_d\,, \qquad \text{namely} \qquad \kappa > \pi f_d/\sqrt{V_{ar}}\,, \tag{2.9}$$

where f_d denotes the dominant frequency of the data. This finding is intuitive, as a scale factor $\hat{\kappa}$ that meets the criterion (2.9) allows the neural network to directly capture the large gradient $O(2\pi f_d)$ of the high-frequency data.

However, setting $\hat{\kappa}$ too high, close to the number of data points N_d, results in overfitting of the neural network. Fig. 4(a) shows that a neural network trained with a scale factor $\hat{\kappa} = 300$ to fit the training data ($N_d = 300$) sampled from the high-frequency function (2.6) with $m = 30$ overfits the data. While the training loss is significantly small (Fig. 4b), the validation error is extremely large. To mitigate overfitting, Fig. 4(c) suggests that the modified scale factor $\hat{\kappa}$ should be less than one-sixth of the total number of data points N_d. As a rule of thumb, for the optimal fitting of high-frequency data, besides satisfying (2.9), the number of training data points N_d should also meet the criterion

$$N_d/6 > \pi f_d \qquad \Longrightarrow \qquad N_d > 6\pi f_d. \tag{2.10}$$

Given that the training domain is often normalized within $[-1, 1]$, which contains $2f_d$ dominant periods, the criterion (2.10) essentially requires a minimum of $3\pi \approx 10$ data points within each dominant period $1/(2f_d)$, to ensure optimal fitting of the neural network to the high-frequency data. Without specific clarification, the criterion (2.10) is applied to all the example problems in this section.

2.3. Algorithm of multi-stage training scheme for regression problems

Incorporating these key settings for higher-stage neural network training, we summarize a complete procedure of multi-stage training scheme for regression problems as shown in Algorithm 1.

Algorithm 1 Multi-stage training scheme for regression problems.

1: Normalize both the input and output data.
2: Build the first-stage neural network with regular weight initialization.
3: Train the neural network to fit the normalized data.
4: Obtain the output of the trained neural network $u_0(x)$.
5: Calculate the error $e_1(x) = u_g - u_0$ between the data u_g and the trained network $u_0(x)$.
6: Normalize the error $e_1(x)$ by its root mean square value ϵ_1
7: Build the second neural network with the scale factor κ obtained from the dominant frequency f_d of the error $e_1(x)$.
8: Train the neural network to fit the normalized error $e_1(x)/\epsilon_1$.
9: Repeat Step 4-9 for certain times until $e_{n+1} = e_n - \epsilon_n u_n$ is smaller enough.
10: Generate the final model, $u(x) = \sum_{i=0}^{n} \epsilon_i u_i(x)$, by combining all trained neural networks at different stages u_n. This final model can approximate the original data u_g with exceptionally high accuracy.

Following Algorithm 1, Fig. 1(b) shows the result of the second-stage training, which fits the residue $e(x)$ between the first-trained network $u_0(x)$ (Fig. 1a) and the original training data $u_g(x)$ from (2.2). In comparison to the single-stage training where the loss plateaued $O(10^{-8})$, the two-stage training dramatically reduced loss to $O(10^{-14})$ (Fig. 1c), resulting in the fitting error $e(x)$ between the two-stage trained networks and the data reaches the machine precision $O(10^{-7})$ for a 32-bit single float (Fig. 1c). Unless double-precision (64-bit) is employed, further reduction of loss is unattainable. Moreover, we note that the total number of weights used (Fig. 1c) in the two-stage neural networks (around $2 \times 3 \times 20^2 \approx 2400$) is less than that in the single-stage network (around $3 \times 30^2 \approx 2700$). This underscores that a larger neural network is not inherently advantageous; an appropriate training scheme is more vital and efficient for the reduction of training loss.

In fact, the power of multi-stage training scheme lies not only in boosting the convergence of training, but also fundamentally enabling neural networks to approximate a target function with arbitrary accuracy as required. We now convert the weights, biases and training data from single-float precision to double precision, and create the third and fourth-stage neural networks in accordance with Algorithm 1. Fig. 5(d) shows that the error $e_4(x)$ between the sum of four-staged networks and the data successfully approaches the machine precision of a 64-bit double float. As long as higher-precision floating-point is used, the error can be further reduced with additional stages of training.

Fig. 5(c) shows that the overall convergence rate of the root mean square value ϵ of the error $e(x)$ between the network and data using multi-stage training scheme follows $\epsilon \sim \exp(-\sqrt{n_{iters}}/25)$, closely approximating exponential decay. In contrast, the regular single-stage training only exhibits a linear decay, $\epsilon \sim 1/n_{iters}$. That is to say, without considering the risk of being trapped in local minima, it would take at least $O(10^{10})$ iterations for a single-stage training to reach an error of $O(10^{-10})$. With the multi-stage training scheme, it only requires $250^2 \approx 6 \times 10^5$ iterations to reach the same error, which is *four orders of magnitude faster*.

Moreover, we note that the number of data points required for higher-stage training also needs to increase following the criterion (2.10). Fig. 5(d) shows that the dominant frequency of residue after three-stage training can reach $f_d = 150$. This implies that the minimal number of training data to guarantee the success of the fourth-stage training needs to be $N_d > 6\pi f_d = 2830$. Fig. 5(c) shows that the relation between the dominant frequency f_d and magnitude ϵ of the residue after different stages of training empirically follows a power law

$$f_d \approx f_0 \epsilon^{-\alpha}, \quad \text{with the exponent } \alpha = 1/6, \tag{2.11}$$

where f_0 denotes the dominant frequency of the original training data. In practice, we anticipate a gradual increase in the frequency of the residue $e(x)$ corresponding to the decrease in its magnitude, namely the exponent α should be close to 0. Considering the error

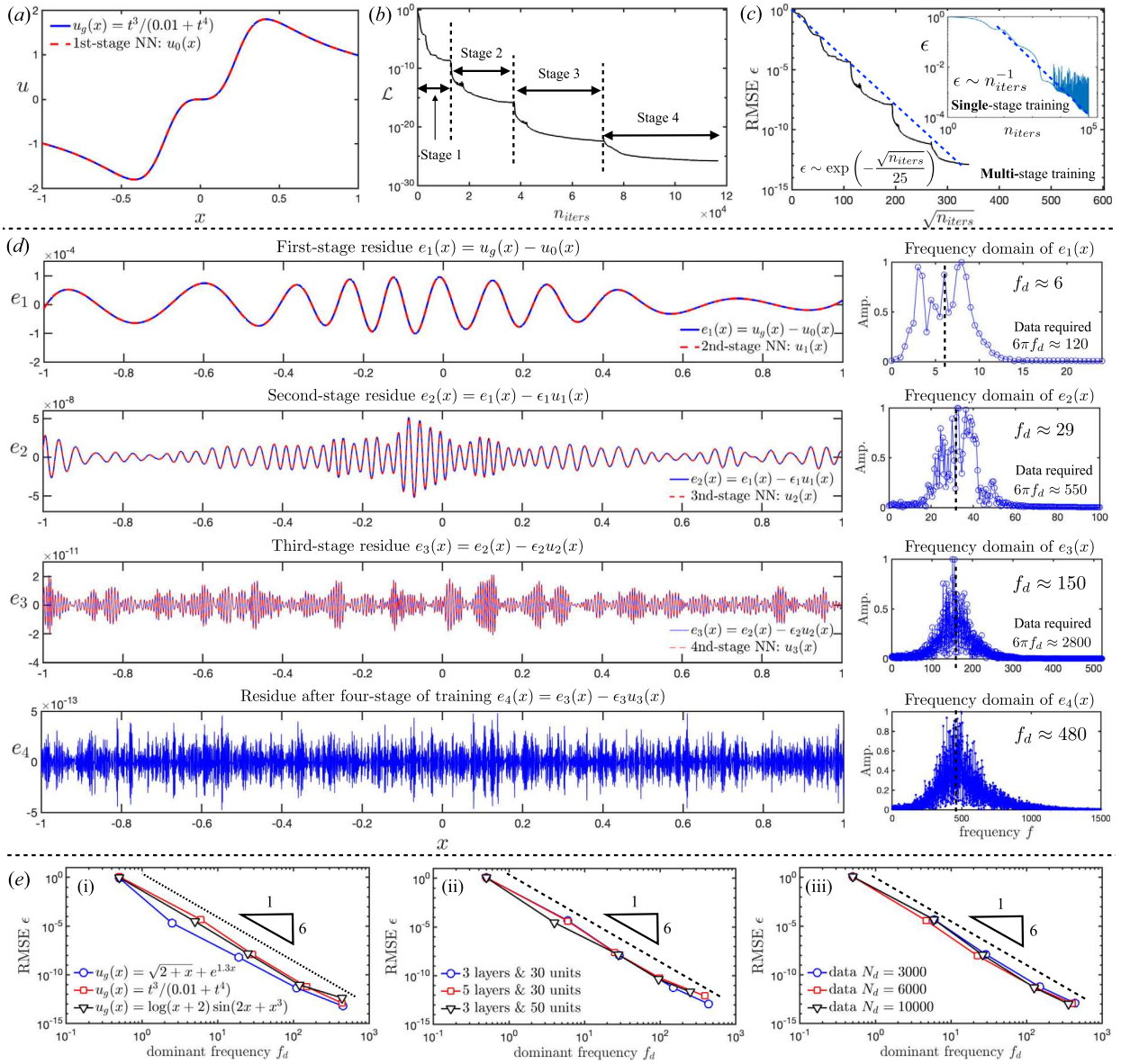


Fig. 5. Multi-stage neural networks. (*a*) Fitting of the first-stage neural network (red dashed curve) to the data from a given target function (blue curve). (*b*) Training loss $\mathcal{L}$ over the iterations based on multi-stage training scheme. (*c*) Evolution of the root mean square value ϵ of the error $e_n(x)$ over the iterations, which follows $\epsilon \sim \exp(-\sqrt{n_{iters}}/25)$, close to an exponential decay. However, for single-stage training (*c*-inset), the error convergence only follows a linear decay, $\epsilon \sim 1/n_{iters}$. (*d*) Fitting of higher-stage networks to the error of lower-stage training. Frequency domain of the error $e_n(x)$ for different stages are shown in the right column. After four stages of training, the error between the data and combined networks is close to the machine precision of a double float (64-bit) (*e*) Relation of the dominant frequency f_d and the root mean square value ϵ of the error $e_n(x)$ after different stages of training follows a power law (2.11) with an exponent α independent of (i) target functions, (ii) neural network size, (iii) and the number of data points.

between a neural network $u(x)$ and target function $u_g(x)$ of magnitude ϵ_0 and dominant frequency f_d, the error between the m-th derivative of $\hat{u}(x)$ and $u_g(x)$ becomes,

$$\frac{d^m}{dx^m}u(x) - \frac{d^m}{dx^m}u_g(x) = \frac{d^m}{dx^m}e(x) \sim (2\pi f_d)^m \epsilon_0 \sim \epsilon_0^{1-\alpha m}, \tag{2.12}$$

where we derive the last expression using (2.11). We find that when $1 - \alpha m > 0$, even if the magnitude of error ϵ_0 is small, the error at high derivatives $m > 1/\alpha$ can still exceed 1. This indicates that the trained neural network with high-frequency error tends to miss the high-derivative ($m > 1/\alpha$) information of the target function underlying the training data. Hence, our goal is to achieve a smaller α value during training, which enables the neural network to learn the high-derivative information from the data more accurately.

However, Fig. 5(*e*) shows that, for regression problems, the exponent α from multi-stage training scheme appears to be universal, independent of both target functions and neural network settings. To further reduce α, high-derivative information about the target

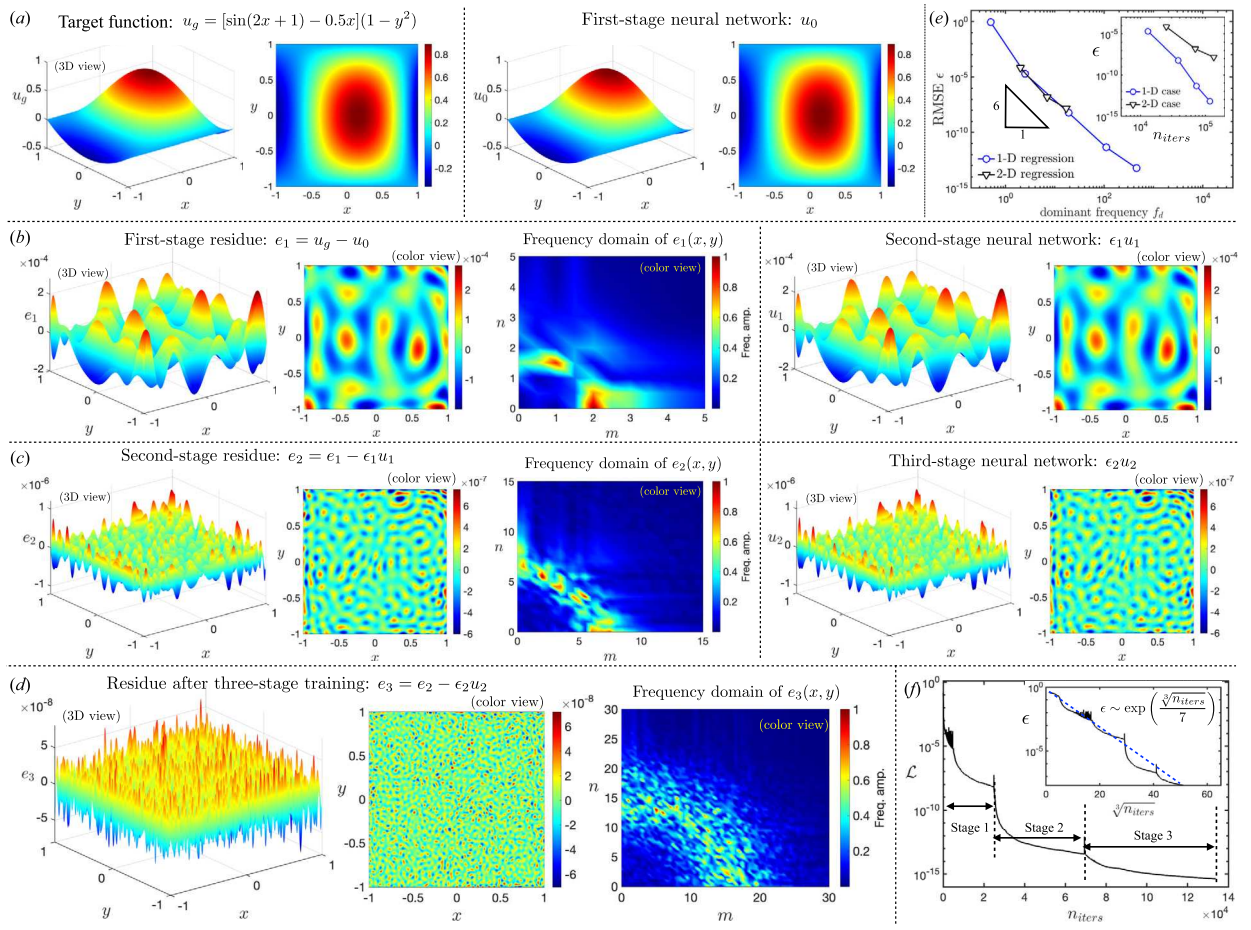


Fig. 6. Multi-stage neural networks for a 2D target function. (*a*) Fitting of first-stage neural network $u_0(x,y)$ to the data from a 2D target function $u_g(x,y)$. (*b-d*) Fitting of higher-stage networks $u_i(x,y)$ to the error $e_i(x,y)$ of lower-stage training. Frequency domain of the error at each stage is given. (*e*) Relation of the dominant frequency f_d with the root mean square value ϵ of the error $e_n(x,y)$ after different stages of training follows the same power law with the 1D problem, of which the exponent $\alpha = 1/6$. (*f*) Training loss $\mathcal{L}$ over iterations of the multi-stage neural networks. The inset shows that the evolution of the root mean square error ϵ over iterations for the 2D regression problem follows $\epsilon \sim \exp(-\sqrt[3]{n_{iters}}/7)$, which is slightly slower than that for the 1D problem (see inset of (*e*).

function would be required for the training. However, this information is often absent in regression problems, while it is readily available for physics-informed neural networks. The methodology of reducing the exponent α for PINNs will be addressed in a later section (§3.4).

Fig. 6(*a-d*) shows that the multi-stage training scheme is equally applicable for 2D regression problems. The convergence rate of the loss function for the 2D problem roughly follows $\epsilon \sim \exp(-\sqrt[3]{n_{iters}}/7)$ (Fig. 6*f*), slower than that for the 1D problem, but still much faster than the linear decay seen with regular single-stage training. Fig. 6(*e*) shows that the relation between the dominant frequency f_d and the root mean square value ϵ of the 2D residue $e(x,y)$ follows the same power law (2.11) with the exponent $\alpha \approx 1/6$.

3. Multi-stage training for physics-informed neural network

The multi-stage training scheme is particularly critical when we use neural network to approximate solutions governed by equations, where the demand for precision is high and essential for the usefulness of the solution. Here we apply the multistage idea to the physics-informed neural networks (PINNs) to improve their accuracy to machine precision. Unlike classical numerical method (i.e. finite difference) which can steadily enhance the accuracy of solution by reducing the grid size, PINNs cannot efficiently reduce solution errors merely by adding more collocation points or enlarging the neural network size, similar to the issue seen with regression problems (see Appendix A). This has made PINNs a less favored method for many scientific research that demands high-precision prediction. In this section, we show that the multi-stage training scheme can be extended to address this limitation of PINNs.

The general procedure of multi-stage training scheme for physics-informed neural networks (PINNs) mirrors that for regression problems (Algorithm 1). However, *two* new challenges emerge when applying multi-stage training to PINNs. *First*, for regression problems, we can directly determine the magnitude ϵ and dominant frequency f_d of the target function for each stage of training

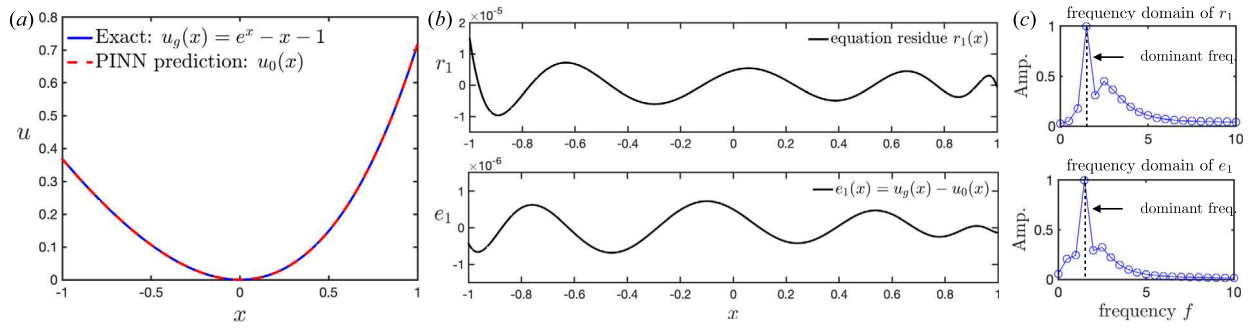


Fig. 7. Comparison of prediction error with equation residue of PINNs. (*a*) Exact solution $u_g(x)$ and neural network prediction $u_0(x)$ to equation (3.3). (*b*) Comparison of the equation residue $r_1(x, u_0)$ associated with the neural network prediction $u_0(x)$ with the prediction error $e_1(x)$ between $u_0(x)$ and the exact solution $u_g(x)$, which has different magnitude. (*c*) The frequency domain of the equation residue $r_1(x, u_0)$ and the prediction error $e_1(x)$, which has the same dominant frequency.

from the residue of lower-stage training. However, for PINNs, these two quantities are not readily obtainable because we lack the exact solution required to estimate the error of lower-stage training.

In addition, the loss function of PINNs involves both data loss and equation loss, defined as

$$\mathcal{L} = (1-\gamma)\mathcal{L}_d + \gamma \mathcal{L}_e \qquad \text{with} \tag{3.1}$$

$$\mathcal{L}_d = \frac{1}{N_d}\sum_{i=1}^{N_d}[u(x_i) - u_i]^2 \quad \text{and} \quad \mathcal{L}_e = \frac{1}{N_e}\sum_{j=1}^{N_e}[r(x_j, u(x_j))]^2, \tag{3.2}$$

where N_d represents the number of data points, commonly employed as the boundary condition, and N_e is the number of collocation points, which are utilized to examine the equation residue $r(x, u)$ at various positions within the domain. In comparison to regression problems, γ, known as the equation weight, is the additional hyper-parameter that balances the significance of the two losses during training. How to determine an appropriate value of γ for higher stages of training becomes the *second* challenge. Using a simple example, we will demonstrate new algorithms to address these challenges and develop a modified multi-stage training scheme for PINNs.

3.1. First challenge: magnitude and frequency of higher-stage network

As discussed in Section 2.2, the effectiveness of multi-stage training scheme depends largely on the optimal setting of the higher-stage neural networks u_n, which is based on the magnitude and frequency of the residue e_n between the combined lower-stage networks and the ground truth u_g. However, these pieces of information are not directly accessible for PINNs because we don't have the exact solution $u_g(x)$ to the equation that is required to estimate the error $e(x) = u_g - u_0$ of the lower-stage trained networks. Instead, the only information we have is the equation residue $r(x, u_0)$ associated with the trained first-stage network u_0. Thus, understanding the relation between the equation residue $r(x, u_0)$ and the error $e(x)$ of the lower-stage networks with the exact solution is crucial for determining the settings for the higher-stage training of PINNs.

3.1.1. A simple example

We consider a first-order ordinary differential equation with the boundary condition

$$\frac{du}{dx} = u + x \qquad \text{with} \qquad u(0) = 0\,, \tag{3.3}$$

which has the exact solution $u_g(x) = e^x - x - 1$. Fig. 7(*a*) shows the single-stage trained network $u_0(x)$ to solve the equation (3.3) via PINN, which matches the exact solution $u_g(x)$ well. The equation residue $r_1(x, u_0)$ associated with the network $u_0(x)$ gives

$$r_1(x, u_0) = \frac{du_0}{dx} - (u_0 + x)\,, \tag{3.4}$$

which has the same dominant frequency with the error $e_1(x) = u_g(x) - u_0(x)$ between the trained network $u_0(x)$ and the exact solution $u_g(x)$. However, the magnitude of equation residue $r_1(x, u_0)$ is one-order of magnitude larger than that of the error $e_1(x)$. To elucidate their relations, we introduce the ansatz,

$$u_g(x) = u_0(x) + \epsilon_1 u_1(x) \qquad \text{with} \quad e_1(x) = \epsilon_1 u_1(x)\,, \tag{3.5}$$

where ϵ_1 denotes the magnitude of the error $e_1(x)$, and $u_1(x)$ becomes the normalized function within the domain. Substituting the ansatz (3.5) into (3.3) and re-arranging the equation gives

$$-\epsilon_1\left(\frac{du_1}{dx} - u_1\right) = \frac{du_0}{dx} - (u_0 + x)\,. \tag{3.6}$$

Recalling (3.4), the right-hand side of (3.6) is the equation residue. Thus, the relation between the prediction error $e_1(x)$ and the equation residue $r_1(x,u_0)$ gives

$$-\epsilon_1\left(\frac{du_1}{dx}-u_1\right)=r_1(x,u_0), \tag{3.7}$$

which also becomes the governing equation for the second-stage training with $u_1(x)$ the second-stage neural network. The boundary condition of u_1, based on (3.3) and (3.5), gives

$$\epsilon_1 u_1(0)=1-u_0(0) \qquad \Longrightarrow \qquad u_1(0)=\frac{1-u_0(0)}{\epsilon_1}. \tag{3.8}$$

With the appropriate setting of the equation weight γ (as discussed in a later section §3.2), the data loss of the first-stage training should be much smaller than that of the equation loss. This indicates that the error $e_1(x)$, as well as $u_1(x)$, has much smaller value at the boundary than within the domain. Namely, the boundary condition of $u_1(0)$ can be considered as 0.

With zero boundary conditions, the magnitude and frequency of the solution $u_1(x)$ are governed by the source function. For a linear equation, the dominant frequency of $u_1(x)$ must be equal to that of the source function, namely the equation residue $r_1(x,u_0)$. Otherwise, the equation cannot be balanced in the frequency domain.

The magnitude of the term $du_1(x)/dx$ in (3.7) scales as $O(2\pi f_d)$, as discussed in Section 2.2, where f_d is the dominant frequency of the function. Given that $u_1(x)$ shares the same dominant frequency with the equation residue $r_1(x,u_0)$, the magnitude ϵ_1 of the error $e_1(x)$ between the network $u_0(x)$ and exact solution $u_g(x)$ can be determined by equating the magnitudes of the leading-order terms on both sides of the equation (3.7), which gives,

$$2\pi f_d\epsilon_1\sim\epsilon_{r_1} \qquad \Longrightarrow \qquad \epsilon_1=\frac{\epsilon_{r_1}}{2\pi f_d} \quad \text{with} \quad \epsilon_{r_1}=\mathrm{RMS}(r_1(x,u_0)) \tag{3.9}$$

where we use the root mean square (RMS) value ϵ_{r_1} to represent the magnitude of $r_1(x,u_0)$. Fig. 7(*c*) shows that the dominant frequency for the equation residue $r_1(x,u_0)$ and prediction error $e_1(x)$ are the same, around $f_d\approx 1.5$. Based on (3.9), the magnitude ϵ_1 of the error should be $2\pi f_d\approx 10$ times less than that of the equation residue, consistent with the result shown in Fig. 7(*b*).

3.1.2. *Magnitude and frequency estimation for general differential equations*

To extend the relations between the properties of equation residue $r_1(x,u_0)$ and prediction error $e_1(x)$ for general equations, we now consider a general form of ordinary differential equations

$$\mathcal{N}\left(x,u,u^{(1)},\ldots u^{(m)}\right)=F(x) \qquad \text{with} \quad u^{(i)}=\frac{d^iu}{dx^i} \quad \text{for} \quad i=1,2,\ldots,m \tag{3.10}$$

where $\mathcal{N}$ is a nonlinear differential operator that involves x, u and its derivative $u^{(i)}$ at different orders. m represents the highest order of derivative of u in the equation. $F(x)$ is a source function with known expression. We denote $u_g(x)$ as the exact solution to the equation and u_0 the first-stage neural network prediction. Substituting the ansatz (3.5) into (3.10) gives

$$\mathcal{N}\left(x,(u_0+\epsilon_1u_1),[u_0+\epsilon_1u_1]^{(1)},\ldots,[u_0+\epsilon_1u_1]^{(m)}\right)=F(x) \tag{3.11}$$

Considering that the first-stage trained network u_0 captures the main variation of the exact solution u_g, the magnitude ϵ_1 of the error $e_1(x)$ between u_g and u_0 would be much smaller than one, namely $\epsilon_1\ll 1$. In that case, the equation (3.11) can be rewritten in terms of a Taylor expansion of the nonlinear function $\mathcal{N}$. After re-arrangement, it gives

$$-\epsilon_1\left(\left.\frac{\partial\mathcal{N}}{\partial u}\right|_{u=u_0}u_1+\left.\frac{\partial\mathcal{N}}{\partial u^{(1)}}\right|_{u=u_0}u_1^{(1)}+\ldots+\left.\frac{\partial\mathcal{N}}{\partial u^{(m)}}\right|_{u=u_0}u_1^{(m)}\right)+O(\epsilon_1^2)=\mathcal{N}\left(x,\ldots,u_0^{(m)}\right)-F(x) \tag{3.12}$$

where u and its derivative $u^{(i)}$ at different orders are considered as separate independent variables of the function $\mathcal{N}(x,u,\ldots,u^{(m)})$. Because $\epsilon_1\ll 1$, all the nonlinear terms of u_1 fall into the high-order $O(\epsilon_1^2)$ term, and can generally be disregarded. This suggests that regardless of the nonlinearity of original equations, the governing equations for higher-stage networks essentially become *linear* equations. This is a key factor that ensures the *success* of multi-stage training scheme for PINNs.

Given that the right-hand side of (3.12) is the equation residue $r_1(x,u_0)$ of the first-stage training, the final equation for u_1 reads

$$\epsilon_1\left(\left.\frac{\partial\mathcal{N}}{\partial u}\right|_{u=u_0}u_1+\left.\frac{\partial\mathcal{N}}{\partial u^{(1)}}\right|_{u^{(1)}=u_0^{(1)}}u_1^{(1)}+\ldots+\left.\frac{\partial\mathcal{N}}{\partial u^{(m)}}\right|_{u^{(m)}=u_0^{(m)}}u_1^{(m)}\right)=r_1(x,u_0) \tag{3.13}$$

or, in a short form,

$$-\epsilon_1\sum_{k=0}^{m}\beta_ku_1^{(k)}=r_1(x,u_0) \qquad \text{with} \quad \beta_k=\left.\frac{\partial\mathcal{N}}{\partial u^{(k)}}\right|_{u=u_0} \quad \text{and} \quad u_1^{(k)}=\frac{d^ku_1}{dx^k} \tag{3.14}$$

As mentioned earlier, if u_0 is correctly trained, the boundary condition for u_1 should be close to 0. Given that (3.14) is linear, the magnitude and frequency of u_1 should be determined from the equation residue $r_1(x,u_0)$ by matching the magnitude and frequency of the dominant term (the term with the largest magnitude) on the left-hand side of (3.14) with that of $r_1(x,u_0)$.

Considering a physical equation with coefficients of similar scale before each term, and assuming u_1 to be a high-frequency function with a dominant frequency far exceeding that of u_0, the dominant term on the left-hand side of (3.14) is expected to be the one involving the highest-order derivative of u_1, namely $\epsilon_1 \beta_m u_1^{(m)}$. We denote the dominant frequency of β_m and $r_1(x, u_0)$ as $f_d^{(\beta)}$ and $f_d^{(r)}$, respectively. Then, the dominant frequency $f_d^{(1)}$ of u_1 gives

$$f_d^{(\beta)} + f_d^{(1)} = f_d^{(r)} \qquad \Longrightarrow \qquad f_d^{(1)} = f_d^{(r)} - f_d^{(\beta)}. \tag{3.15}$$

Given that β_m is a function only with respect to the lower-order network u_0, and the dominant frequency $f_d^{(1)}$ of the higher-stage network u_1 is much larger than that of u_0. Thus, even if β_m is a highly-nonlinear function, we have $f_d^{(1)} \gg f_d^{(\beta)}$. Combined with (3.15), we have

$$f_d^{(1)} = f_d^{(r)} - f_d^{(\beta)} \approx f_d^{(r)} \tag{3.16}$$

where the dominant frequency of u_1 is mainly governed by that of equation residue. With the dominant frequency determined, the magnitude ϵ_1 of the error between u_g and u_0 can be derived by matching the magnitude of the term $\epsilon_1 \beta_m u_1^{(m)}$ and $r_1(x, u_0)$, which gives

$$\epsilon_1 \cdot \epsilon_\beta \left[2\pi f_d^{(r)}\right]^m = \epsilon_{r_1} \qquad \Longrightarrow \qquad \epsilon_1 = \frac{\epsilon_{r_1}}{\left[2\pi f_d^{(r)}\right]^m \epsilon_\beta} \tag{3.17}$$

$$\text{with} \qquad \epsilon_\beta = \mathrm{RMS}(\beta_m) \quad \text{and} \quad \epsilon_{r_1} = \mathrm{RMS}(r_1(x, u_0))$$

where we use the root mean square (RMS) value ϵ_β and ϵ_{r_1} to represent the magnitude of β_m and equation residue $r_1(x, u_0)$, respectively.

The relations (3.16) and (3.17) can also be generalized to *partial* differential equations, for which we need to calculate the dominant frequency of u_1 with respect to each independent variable x_i, namely

$$f_d^{(1,x_i)} \approx f_d^{(r,x_i)} \qquad \text{for} \quad i = 1, 2, ..., N \tag{3.18}$$

where N is the total number of independent variables of the equation. Then, the magnitude ϵ_1 of the error between the first-stage network u_0 and the exact solution u_g would be

$$\epsilon_1 = \frac{\epsilon_{r_1}}{\epsilon_\beta \left[2\pi f_d^{(r,x_1)}\right]^{m_1} \cdot \left[2\pi f_d^{(r,x_2)}\right]^{m_2} ... \left[2\pi f_d^{(r,x_N)}\right]^{m_N}} \tag{3.19}$$

where $m_1 + m_2 + ... + m_N$ represents the highest order of partial derivative of u_1 in the equation. For most equations, the relations (3.16)-(3.19) are sufficiently accurate to estimate the magnitude and frequency of the network for higher-stage PINN training. However, there are two types of nonlinear equations where these relations may not hold exactly. They are discussed in Appendix C.

3.1.3. *Importance of magnitude and frequency for higher-stage PINN training*

The proper setting of the magnitude and frequency of a neural network, as shown to be essential for regression problems in Section 2.2, is critical for physics-informed neural networks, especially during higher-stage training. To illustrate this, we use a Poisson equation with a high-frequency source function to represent the equation residue, and zero boundary conditions. This setup effectively mimics the governing equation for higher-stage PINN training. The equation reads

$$u_{xx} + u_{yy} = -\sin(6\pi x)\sin(6\pi y) \qquad \text{with} \quad u(x, \pm 1) = u(\pm 1, y) = 0 \tag{3.20}$$

At first glance, one might assume that the solution u has the same order of magnitude with the source function, which is $O(1)$. However, based on the analysis (3.18) in Section 3.1.2, the solution should have a dominant frequency $f_d = 3$ with respect to both independent variables x and y. Given that the highest order of derivative in (3.20) is $m = 2$, the magnitude of the solution u can, then, be derived from (3.19), as $1/[2\pi f_d]^2 \sim O(10^{-3})$, as shown in the exact solution,

$$u_g(x, y) = \frac{1}{2(6\pi)^2} \sin(6\pi x)\sin(6\pi y). \tag{3.21}$$

Fig. 8 shows the neural network predicted solution u_0 to (3.20) via PINNs under different setting of magnitude (via magnitude prefactor ϵ_0) and frequency (via modified scale factor $\hat{\kappa}_0$). In these cases, we assume that the correct value of the equation weight γ is used. As shown in Fig. 8(*b*), only when both magnitude and frequency are correctly set in accordance with (3.18) and (3.19) does the neural network successfully converge to the exact solution at a rapid convergence rate.

3.1.4. *Algorithm for determining the solution magnitude for higher-stage PINN training*

Besides the theoretical relations (3.16)-(3.19) derived in Section 3.1.2, we also develop a general algorithm to determine the magnitude of the solution to linear differential equations with high-frequency source functions and zero boundary conditions to mimic the higher-stage training of PINN. The algorithm can subsequently be combined with Algorithm 1 to extend the multi-stage training scheme for PINNs. The specific steps of the algorithm are given in Algorithm 2.

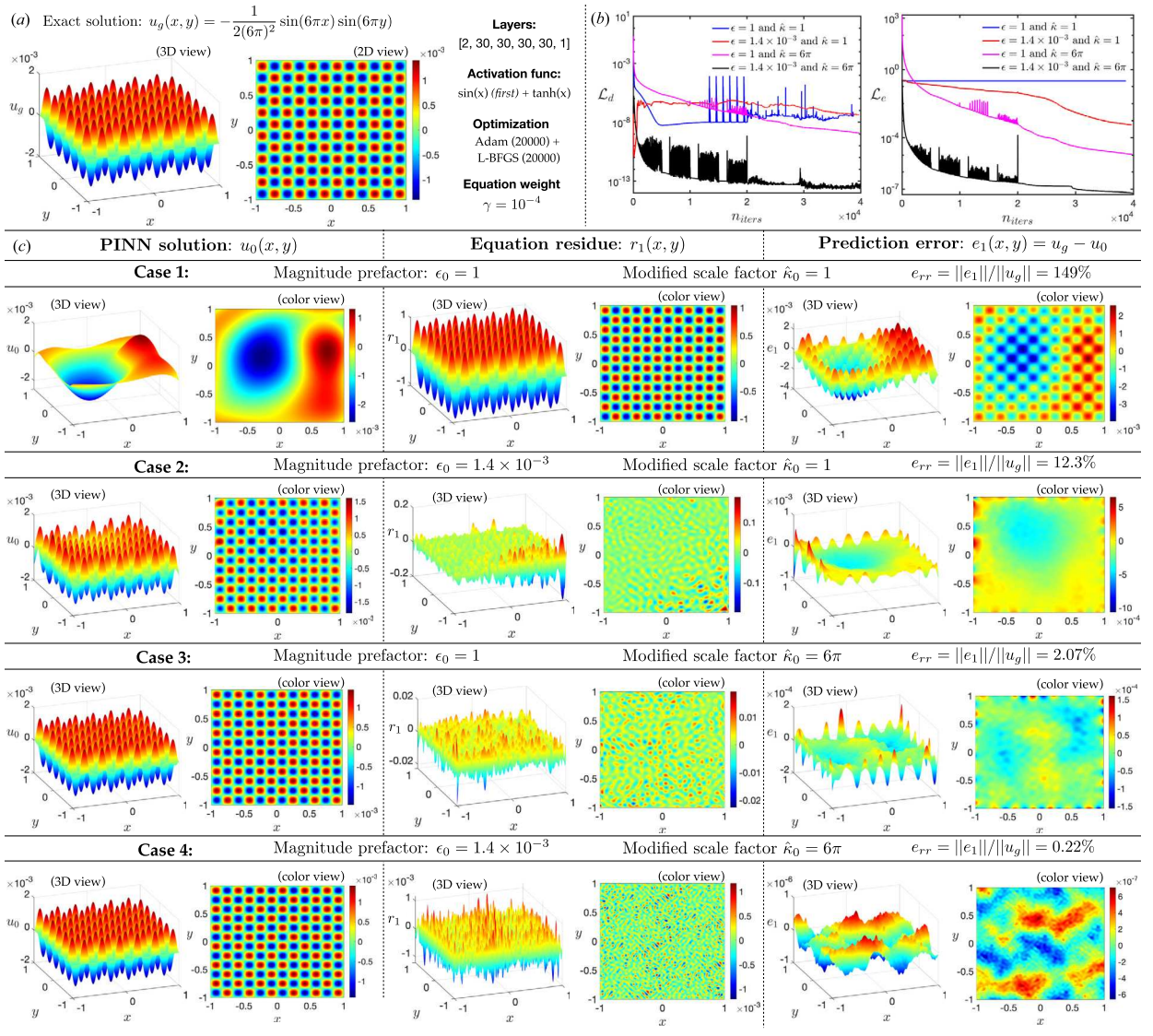


Fig. 8. Importance of rescaling PINN magnitude and frequency. (*a*) Exact solution $u_g(x,y)$ to equation (3.20) and the general setting of PINNs for 4 cases. (*b*) Evolution of the data loss and equation loss over the training of solving (3.20) via PINNs for different magnitude prefactor ϵ_0 and modified scale factors $\hat{\kappa}_0$. (*c*) Trained network $u_0(x,y)$ of the solution to (3.20), the associated equation residue $r_1(x,y,u_0)$ and prediction error $e_1(x,y)$ under different magnitude prefactor ϵ_0 and modified scale factor $\hat{\kappa}_0$ (Case 1-4). The trained network with ϵ_0 and $\hat{\kappa}_0$ from (3.18) and (3.19) gives the prediction with the lowest relative error e_{rr}.

The principle underlying the algorithm is based on the fact that the dominant frequency of the solution $u(\mathbf{x})$ mirrors that of the source function $s(\mathbf{x})$. Therefore, the amplification effect of the derivative, attributable to the high-frequency property of the solution, can be well-estimated by taking $s(\mathbf{x})$ as the guess solution. We define the ratio R as the magnitude of the differential operator $\mathcal{N}_s$ relative to that of the source function $s(\mathbf{x})$.

If R is larger than 10, the magnitude of the differential operator $\mathcal{N}_s$ associated with the guess solution (3.23) much larger than the source function $s(\mathbf{x})$. In that case, the magnitude of the guess solution should be reduced by decreasing α. Conversely, when R is less than 0.1, it suggests that differential operator $\mathcal{N}_s$ associated with the guess solution (3.23) is too small. Hence, we should increase the magnitude of the solution by increasing α. The recursive relation (3.27) in Algorithm 2 is designed to achieve this objective. Here, the learning rate η is a user-defined positive hyper-parameter, which determines the rate at which α and $R(\alpha)$ converges to satisfy the criterion (3.26). Finally, the magnitude of the solution ϵ can be estimated using (3.28).

Applying Algorithm 2 to equation (3.20), we obtain that $\epsilon = 1.41\times10^{-3}$, which is very close to the magnitude of the exact solution, $\epsilon = 1/[2(6\pi)^2] = 1.43\times10^{-3}$. Here, we note that Algorithm 2 is mainly applicable to linear differential equations with a single dependent variable. For nonlinear equations or a group of differential equations with multiple dependent variables, a more advanced algorithm may be required to determine the magnitude for each variable, which is beyond the scope of this paper.

Algorithm 2 Determine the magnitude ϵ of solution to a linear equation with *high-frequency* source function and **zero** boundary condition.

1: Write a linear equation in terms of the source function $s(\mathbf{x})$ (function without dependent variable) as

$$\mathcal{N}[\mathbf{x}, u(\mathbf{x})] = s(\mathbf{x}), \tag{3.22}$$

where $\mathcal{N}[\cdot]$ indicates the differential operator involved in a given equation and $\mathbf{x} = (x_1, x_2, \ldots)$ represents the independent variables.

2: Define a guess solution based on source function $s(\mathbf{x})$ as

$$u_s(\mathbf{x}, \alpha) = \alpha s(\mathbf{x}), \tag{3.23}$$

where the coefficient α can be initially set to be 1.

3: Substitute the guess solution u_s (3.23) and calculate the associated differential operator $\mathcal{N}_s$

$$\mathcal{N}_s(\mathbf{x}, \alpha) = \mathcal{N}[\mathbf{x}, u_s(\mathbf{x}, \alpha)] \tag{3.24}$$

In practice, for complicated equations, the differential operator $\mathcal{N}_s$ can be more easily calculated by adding the equation residue $r_s(\mathbf{x}, u_s)$ associated with the guess solution with the source term $s(\mathbf{x})$, namely

$$\mathcal{N}_s(\mathbf{x}, \alpha) = r_s[\mathbf{x}, u_s(\mathbf{x})] + s(\mathbf{x}) \tag{3.25}$$

4: Introduce an iteration process and define the criterion for the coefficient α

$$0.1 < R(\alpha) = \frac{||N_s(\mathbf{x}, \alpha)||}{||s(\mathbf{x})||} < 10 \tag{3.26}$$

where $||\cdot||$ represents the l_2-norm. If the ratio R falls outside the criterion (3.26), update α with learning rate η by

$$\alpha \leftarrow \alpha / R^{\eta} \tag{3.27}$$

and re-calculate $R(\alpha)$ based on the updated value of α until R meets the criterion (3.26).

5: Compute the solution magnitude ϵ by multiplying α from Step 4 by the root mean square ϵ_s of the source function $s(\mathbf{x})$, namely

$$\epsilon = \alpha \cdot \epsilon_s \qquad \text{where} \quad \epsilon_s = \text{RMS}(s(x, y)) \tag{3.28}$$

3.2. Second challenge: equation weight for higher-stage network

Equation weight γ, as shown in (3.1), is a hyper-parameter to balance the contribution of data loss and equation loss in the loss function for physics-informed neural networks (PINNs). In the context of PINNs as a differential equation solver, boundary conditions are often implemented as data loss and the governing equations constitute the equation loss. Given that boundary conditions determine the uniqueness of the solution, a general rule of thumb is weighting the data loss more than the equation loss in the loss function [27]. This ensures that the boundary conditions are prioritized and satisfied during the training. Several previous papers had proposed optimal ways to weigh different terms in the loss function [28–31]. For example, Wang et al. (2022) [30] demonstrated large discrepancies in the convergence rates between different terms of the loss function and developed algorithms to determine the optimal value of γ using the neural tangent kernel of PINNs.

The relative contribution of data loss and equation loss in the loss function (3.1) is the ratio of the first term, $I_1 = (1-\gamma)\mathcal{L}_d$, to the second term, $I_2 = \gamma\mathcal{L}_e$ in (3.1), i.e. I_1/I_2. For normalized linear differential equations that have low-frequency solutions, such as (3.3), the equation loss, $\mathcal{L}_e$, remains the same order of magnitude with the data loss $\mathcal{L}_d$, around $O(1)$. In that case, by setting $0.1 < \gamma < 0.5$, we can ensure that the contribution of data loss $I_1 > I_2$ in the loss function (3.1). However, this setting of γ does not hold for differential equations with high-frequency solutions. A systematic mathematical justification was provided by a prior study [31], showing that the magnitude of the equation loss increases with the frequency of the solution. When considering the same Poisson equation (3.20) with zero boundary conditions and a high-frequency source function, similar to the equation we solve during higher-stage training, the magnitude of equation loss at the beginning of the training can be estimated as

$$\mathcal{L}_e = \frac{1}{N_e}\sum_{i=0}^{N_e}\left[u_{xx} + u_{yy} + \sin(6\pi x_i)\sin(6\pi y_i)\right]^2 \sim O(1), \tag{3.29}$$

which is determined by the magnitude of the source function $\sin(6\pi x)\sin(6\pi y)$. The magnitude of data loss at the beginning of training reads

$$\mathcal{L}_d = \frac{1}{N_1}\sum_{i=0}^{N_1}[u(x_i, \pm 1)]^2 + \frac{1}{N_2}\sum_{j=0}^{N_2}[u(\pm 1, y_j)]^2 \sim O(u^2) \tag{3.30}$$

which is determined by the initial magnitude of the solution. Considering that the magnitude ϵ_0 of the solution has been estimated from the relation (3.19) or Algorithm 2, which gives $\epsilon_0 \approx 1/(6\pi)^2$ for the solution to (3.20), the magnitude of the data loss thus becomes

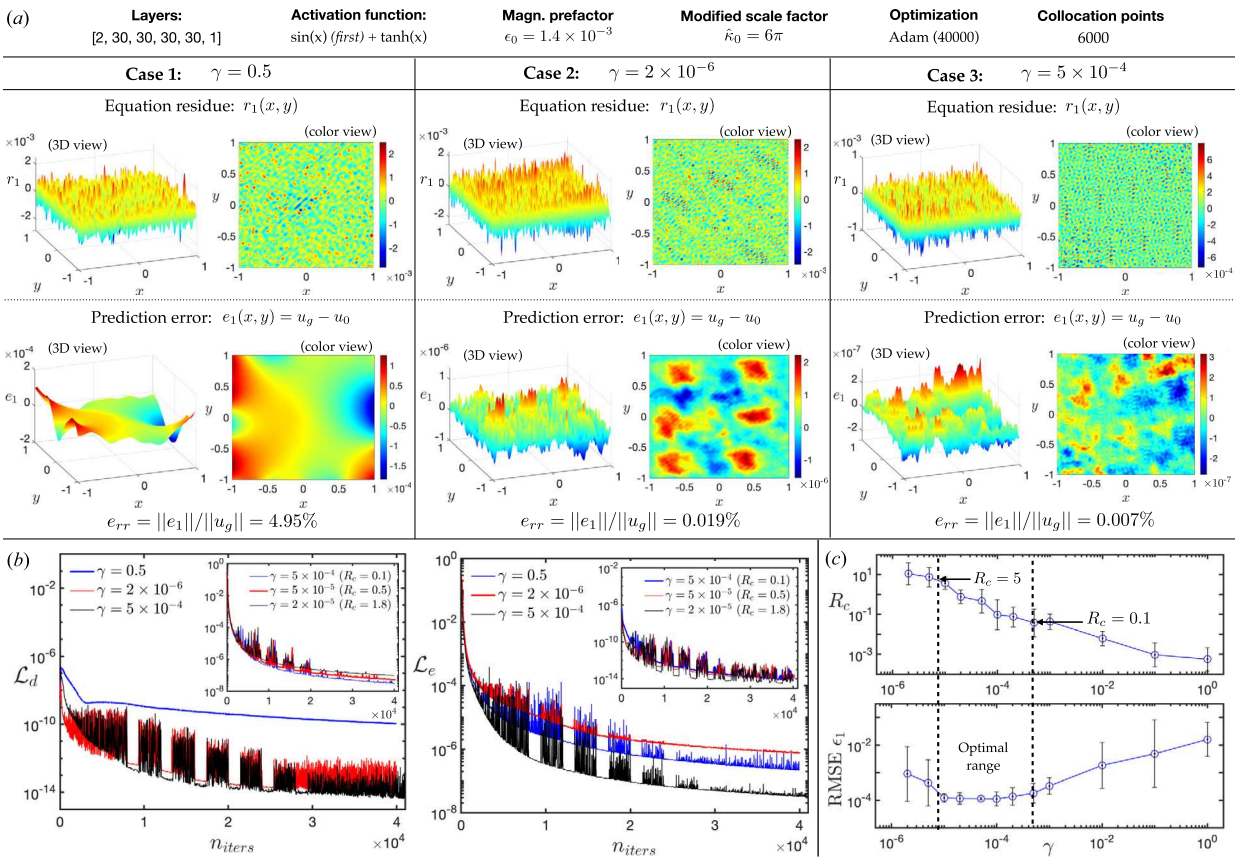


Fig. 9. Importance of the equation weight γ of PINNs. (*a*) Equation residue $r_1(x, y)$ and prediction error $e_1(x, y)$ of solving (3.20) via PINN for different γ. (*b*) Evolution of data loss and equation loss over the training for different γ shown in (*a*). The inset shows the loss evolution for different γ that satisfy the criterion (3.37), which are close to each other. (*c*) The root mean square value ϵ_1 of the prediction error $e_1(x, y)$ (lower panel), and the corresponding ratio R_c of the data loss convergence rate over that of equation loss (upper panel) as a function of γ. The optimal range of γ with minimal prediction error corresponds to $0.1 < R_c < 5$. Error bars show the standard deviation of five repetitive experiments with different random initialization.

$$\mathcal{L}_d \sim O(\epsilon_0^2) \sim O(10^{-6}) \tag{3.31}$$

which is six orders of magnitude smaller than the equation loss. If we still use $\gamma \sim O(0.1)$, as $I_2 \gg I_1$, the optimization process will primarily focus on minimizing the I_2 during the training, largely neglecting the contribution from the data loss. Utilizing the appropriate values of the magnitude prefactor ϵ_0 and the modified scale factor $\hat{\kappa}_0$ from Section 3, Fig. 9(*b*) shows the evolution of the data loss and equation loss over iterations by setting $\gamma = 0.5$. Compared with the equation loss, which was reduced by seven orders of magnitude in total, the data loss decays at a much slower rate, significantly limiting the errors of the trained neural network 9(*c*) to be round 5%.

3.2.1. *Theoretical approach of determining equation weight*

To improve the accuracy of trained network via PINN, we need to minimize the data loss prior to the equation loss. Therefore, the optimal value of γ should yield a larger contribution from the data loss larger than from the equation loss, namely,

$$I_1 = (1-\gamma)\mathcal{L}_d \geq \gamma \mathcal{L}_e = I_2 \qquad \Longrightarrow \qquad \gamma \leq \frac{\mathcal{L}_d}{\mathcal{L}_e + \mathcal{L}_d}, \tag{3.32}$$

which is consistent with the expression proposed in a prior study [31]. For equation (3.20), with the magnitude of the equation loss and data loss determined from (3.29) and (3.30) respectively, equation (3.32) yields $\gamma = 2 \times 10^{-6}$. With this γ, Fig. 9(*b*) shows that both data loss and equation loss rapidly decrease over the training by more than five orders of magnitude. This suggests that both the equation and boundary condition are progressively satisfied by the network throughout the training. Although the reduction in equation loss is slightly less than in the case with $\gamma = 0.5$, the relative error e_{rr} between the trained neural network and the exact solution is reduced by more than one hundred times (Fig. 9*a*). However, generally without prior knowledge of the solution we do not know its corresponding $\mathcal{L}_d$ and $\mathcal{L}_e$, so estimating γ theoretically is difficult. Therefore below we develop an alternative approach.

3.2.2. Algorithm for determining equation weight for general equations

Besides the theoretical expression (3.32), we also develop a more general algorithm to determine γ through a pre-training process. This approach provides higher accuracy and adaptability for a broad range of problems. As mentioned previously, the optimal value of γ should result in similar convergence rates for the data loss and equation loss over the course of training. We propose a heuristic approach for determining the optimal γ, which is outlined in Algorithm 3.

Algorithm 3 Determine the optimal equation weight γ for general PINN training.

Prerequisite: normalize the equation with the magnitude of the largest term around $O(1)$ and apply the scale factor and the magnitude prefactor estimated from Algorithm 2 to the neural network.

1: Set the initial value of γ and calculate the initial data loss and equation loss

$$\mathcal{L}_d^{(0)} = \mathcal{L}_d(i=0) \qquad \text{and} \qquad \mathcal{L}_e^{(0)} = \mathcal{L}_e(i=0) \tag{3.33}$$

where $\mathcal{L}_d^{(i)}$ and $\mathcal{L}_e^{(i)}$ represent the data and equation loss at the i-th iteration, respectively.

2: Pre-train the neural network for N_0 iterations and calculate the minimal data loss and equation loss within the last $N_1 = 0.1N_0$ iterations as

$$\mathcal{L}_d^{(m)} = \min_{i=0}^{N_1-1} \mathcal{L}_d(N_0 - i) \qquad \text{and} \qquad \mathcal{L}_e^{(m)} = \min_{i=0}^{N_1-1} \mathcal{L}_e(N_0 - i) \tag{3.34}$$

3: Quantify the convergence rate of the data loss and equation loss with

$$C_d = \frac{\mathcal{L}_d^{(0)}}{\mathcal{L}_d^{(m)}} \qquad \text{and} \qquad C_e = \frac{\mathcal{L}_e^{(0)}}{\mathcal{L}_e^{(m)}} \tag{3.35}$$

4: Introduce an iteration process and define the criterion for the coefficient γ

$$O(0.1) < R_c(\gamma) = \frac{C_d}{C_e} < O(10) \tag{3.36}$$

5: If the ratio R_c of the two convergence falls outside the criterion (3.36) , update γ with learning rate η by

$$\gamma \leftarrow \gamma \cdot R_c^{\eta} \tag{3.37}$$

and rerun Step 3 - 6 with the new updated γ until the criterion (3.36) is satisfied.

The principle underlying the algorithm lies in estimating the *convergence rates* of both data loss C_d and equation loss C_e. This estimation involves calculating the ratio of the initial loss $\mathcal{L}_d^0$ and $\mathcal{L}_e^0$, to the respective losses $\mathcal{L}_d^m$ and $\mathcal{L}_e^m$ after a short period of pre-training. Here, we use the minimal value during the last 10% of the pre-training iterations to calculate $\mathcal{L}_d^m$ and $\mathcal{L}_e^m$ to counteract any potential spikes in the loss evolution.

If the convergence rate of the data loss C_d is substantially lower than that of the equation loss C_e, it indicates that the γ used in training is too large and needs to be reduced. Vice versa. The recursive relation (3.37) is designed to reach this goal. η can be considered as the learning rate, a hyper-parameter that determines how fast $R_c(\gamma)$ meets the criterion (3.36). We note that, when γ is updated, one should re-train the neural network from the beginning to compute the updated $R_c(\gamma)$, instead of continuing the previous training.

Apply Algorithm 3 to the equation (3.20) with N_0 set to be 500, one gives $\gamma \approx 10^{-4}$. Fig. 9(a) shows the trained network using this γ, which reaches further higher accuracy than that using $\gamma = 2 \times 10^{-6}$ from (3.32). Compared with the case of $\gamma = 0.5$ and $\gamma = 2 \times 10^{-6}$, the convergence rate of both data loss and equation loss when using γ from Algorithm 3 are maximized (Fig. 9b), leading to the smallest errors of the neural network prediction.

The criterion range in (3.36) suggests that the training accuracy is not overly sensitive to the value of γ, provided that the convergence rate of data loss C_d and equation loss C_d remains within the same order of magnitude. Fig. 9(c) shows the optimal range of γ that yields the minimal root mean square value ϵ_1 of the error $e_1(x)$ between the trained network and the exact solution to (3.20). The corresponding range of R_c is found to be $0.1 < R_c < 5$, aligning with the range in criterion (3.37). The inset of Fig. 9(b) shows that the evolution of both data loss $\mathcal{L}_d$ and equation loss $\mathcal{L}_e$ for different $R_c(\gamma)$ with this range (3.36) are closely matched. For $\gamma = 0.5$, the ratio R_c, based on Algorithm 3, is found to be $R_c(\gamma = 0.5) = 10^{-4}$, which largely deviates from the criterion, thus, resulting in large prediction error.

3.3. Additional setting of PINN training for higher-stage network

Besides the most critical settings, i.e. magnitude prefactor ϵ, scale factor κ of frequency, and the equation weight γ, there are other settings and advanced algorithm developed in the literature that can ensure the success of PINN training for the high-stage networks.

3.3.1. Optimization method and re-sampling collocation points

Two other critical settings in the training of high-frequency function includes the selection of optimization method and the number of collocation point. Common choices for PINN training optimizer include Adam and L-BFGS, a second-order quasi Newton method. For general equations with low-frequency solution, L-BFGS is often the preferred optimization method. However, for equations with

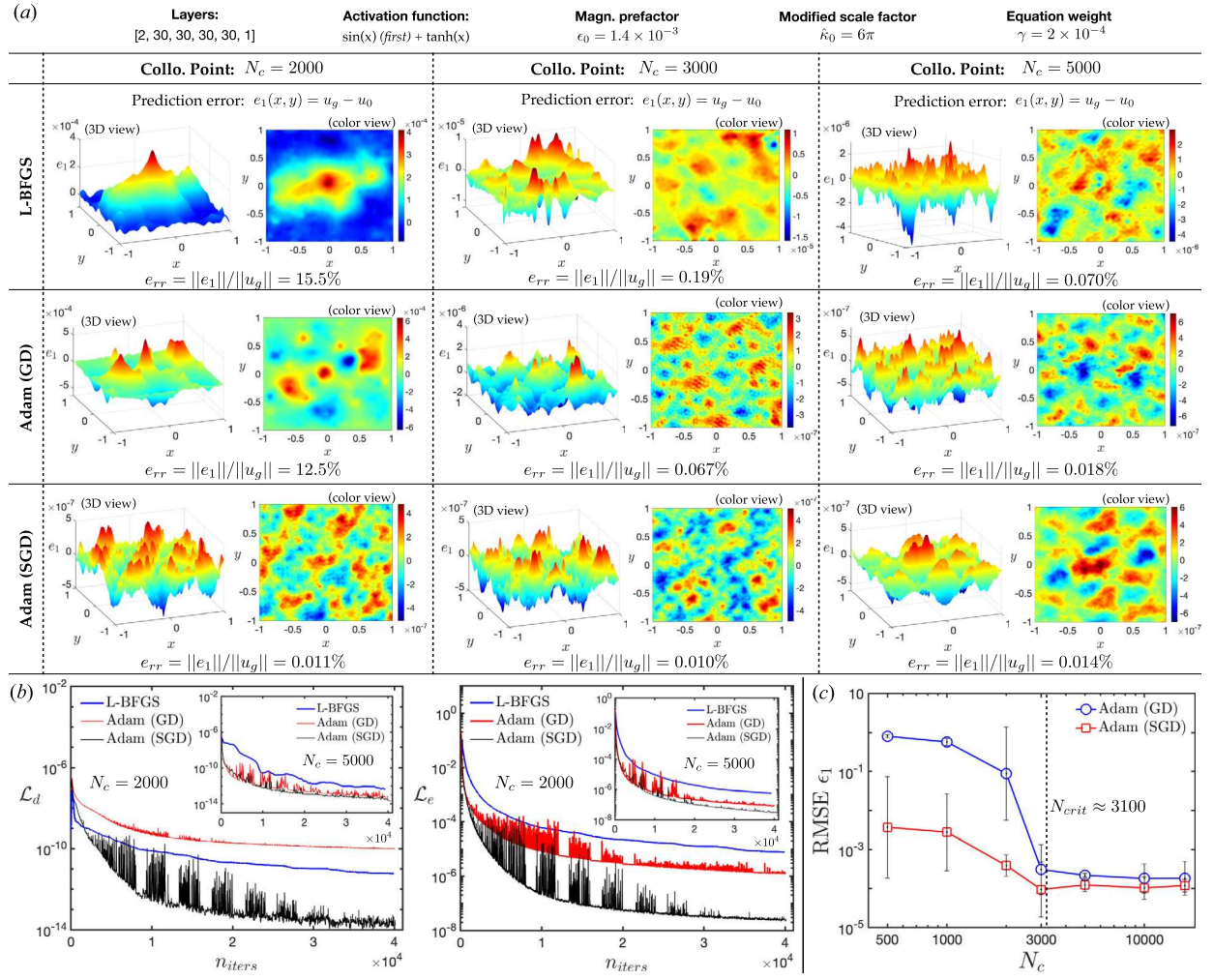


Fig. 10. Importance of re-sampling collocation points of PINNs. (*a*) Comparison of prediction error $e_1(x, y)$ of solving (3.26) via PINNs for different number of collocation points N_c and using different optimizer, including L-BFGS, Adam with gradient descend (GD) (fixed collocation points) and Adam with stochastic gradient descent (re-sample collocation points over the iterations). (*b*) The evolution of data loss and equation loss over the iterations for different optimizer using the number of collocation points N_c below or above (inset) the critical value N_{crit}. (*c*) The relation of the root mean square value ϵ_1 of the prediction error $e_1(x, y)$ with the number of collocation points N_c for Adam (GD) and Adam (SGD). When the number of collocation points N_c is less than the critical value $N_{crit} \approx 3100$, stochastic gradient descend can reach better performance for predicting high-frequency solutions. Error bars show the standard deviation of five repetitive experiments with different random initialization.

high-frequency solutions, this is not always the case. Fig. 10(*a*) presents a comparison of the loss evolution and final prediction error of trained network between using Adam and L-BFGS for solving the equation (3.20), where Adam shows a better overall convergence rate. Furthermore, Adam has the added advantage of utilizing stochastic gradient descent (SGD) by re-sampling the collocation points every few iterations [44,45], which shows a even higher convergence rate.

Collocation points in PINN training is as important as data points in regression problems. As discussed in Section 2.2.2, when training the neural network to fit high-frequency data, a sufficient number of data points ($3\pi \approx 10$ per dominant period) are needed to ensure accurate predictions. This principle remains valid for PINN training. Unlike regression problems, which are limited by the availability of finite data points, PINN could potentially utilize as many collocation points as computationally feasible. For equation (3.20), the dominant frequency $f_d = 3$ in each dimension. Given that the domain is defined in $(x, y) \in [-1, 1]$, there are 6 dominant periods in each dimension. Thus approximately $N_{crit} = (3\pi \times 6)^2 \approx 3100$ collocation points are required. Fig. 10(*a*) compares the accuracy of the trained network for different number of collocation points. For L-BFGS and Adam (GD) with fixed and small number of collocation points, the neural network predictions significantly deviate from the exact solution. When the number of collocation points reach the criterion N_{crit}, the prediction error drops sharply and only improves marginally with the addition of more collocation points.

Compared with using fixed collocation points, predictions using Adam with stochastic gradient descent (SGD) are less sensitive to the number of collocation points. Fig. 10(*a*) shows that the prediction error using Adam (SGD) can attain optimal precision even when the number N_c of collocation points falls below the critical value N_{crit}. Fig. 10(*c*) further compare the root mean square error

(RMSE) ϵ of the trained network between utilizing Adam (GD) and Adam (SGD) for different number of collocation points N_c. It confirms that SGD is an essential tool in PINN training for predicting high-frequency solution.

3.3.2. Advanced methods from the literature: RAR and gPINNs

Having discussed the essential settings, we note that many advanced algorithms developed in the literature can also largely improve the PINN training of higher-stage networks. Two of most useful methods we found are the adaptive residual-based collocation refinement (RAR) method [46,47] and the gradient-enhanced physics-informed networks (gPINNs) [48].

A common practice in PINN training is the uniform distribution of collocation points across the domain. However, this approach proves inadequate for equations whose solutions present steep gradients [49]. As discussed in Section 2.2, high-frequency solutions exhibit large gradients throughout the domain. Despite applying a large scale factor to match the gradient, there are still areas where the local gradient exceeds the averaged gradient $O(2\pi f_d)$, with f_d the dominant frequency of the solution. This makes it difficult to minimize the equation residue in those areas. To address this issue, we employ the residual-based refinement (RAR [46]) of collocation points. By continuing adding collocation points in areas of high equation residue throughout the training, the equation residue across the entire domain can be efficiently reduced. This technique thus becomes a vital tool for optimizing PINN training.

An additional method to boost the training performance of PINNs involves incorporating the gradient of the equation residue function $r(x,u)$ into the loss function $\mathcal{L}$, known as the gradient-enhanced physics-informed network (gPINN) [48]. Thus, the loss function can be expressed as,

$$\mathcal{L} = (1-\gamma)\mathcal{L}_d + \gamma(\mathcal{L}_e + \gamma_g \mathcal{L}_g) \qquad \text{with} \quad \mathcal{L}_g = \frac{1}{N_g}\sum_{j=1}^{N_g} |\nabla r(x_j, u(x_j))|^2, \tag{3.38}$$

where N_g is the number of collocation points used to examine the gradient of the equation residue $r(x,u)$ within the domain. γ_g is an additional hyper-parameter, akin to γ, to control the balance between the equation loss $\mathcal{L}_e$ and gradient loss $\mathcal{L}_g$ during the training. By incorporating the gradient loss $\mathcal{L}_g$, we obligate the neural networks to learn the high-derivative information of the solution involved in the gradient of the equation. This can significantly improve the convergence rate of the training loss, provided we choose the appropriate value of the weight γ_g for the gradient loss $\mathcal{L}_g$. Analogous to (3.32), the value of γ_g can be estimated by

$$\mathcal{L}_e \geq \gamma_g \mathcal{L}_g \qquad \Longrightarrow \qquad \gamma_g \leq \frac{\mathcal{L}_e}{\mathcal{L}_g} \sim \frac{||r||^2}{||\nabla r||^2} \tag{3.39}$$

where $||\cdot||^2$ represents the l_2-norm. As discussed in Section 3.2.2, for the equation with high-frequency solutions, the equation residue has roughly the same frequency with the solution. Thus, the magnitude ratio of the equation residue $||r||$ with its gradient should scale as $||r||/||\nabla r|| \sim O(2\pi f_d)^{-1}$, where f_d is the dominant frequency of the solution. Thus, the optimal value of γ_g can be selected as

$$\gamma_g = \frac{||r||^2}{||\nabla r||^2} \sim O(2\pi f_d)^{-2} \tag{3.40}$$

The effect of the gPINN on the higher-stage training is shown and discussed in Section 3.4.

3.4. Algorithm of multi-stage training for PINNs

Leveraging the multi-stage training algorithm for regression problems and incorporating the results discussed in the previous sections, we have extended the multistage training scheme to physics informed neural networks (PINNs). The details of the algorithm are provided in Algorithm 4.

Here, we note that the primary distinction between the multistage training scheme for PINNs and that for regression problems lies in the fact that we *lack* training data for the solution itself for PINNs. Contrasting with the multistage framework for regression problems, where the second network is trained directly using the error $e_1 = u_g - u_0$ between the first trained network u_0 with the data u_g, we don't necessarily have access to the error of the first trained network in the context of PINNs. Thus, the method of training the second network u_1 for PINNs involves creating a combined network $u_k^{(c)}$ (3.41) that involves the previously trained network $u_{k-1}^{(c)}$ and a new network $u_k(x,\kappa_k)$, with an appropriately-estimated magnitude prefactor ϵ_k and scale factor κ_k. A key advantage of this approach is that it circumvents the need to derive a new equation, as shown in (3.13), for each higher-stage network. By fixing the trained weights and biases in the previous networks, the training process for solving the original equation becomes mathematically equivalent to solving the higher-stage governing equation (3.13) with the high-frequency source function from the equation residue of the lower-stage training.

Using Algorithm 4, Fig. 11 shows the three-staged PINN training for solving the ordinary differential equation (2.2). For the first two stages, we employ a combination of Adam and L-BFGS for training, which maximizes the convergence rate. However, given the high-frequency residue from the second stage of training, it indicates a high-frequency solution for the third stage of training. Thus, we only use Adam with stochastic gradient descent (SGD) to optimize the performance of the third-stage training, in accordance with the suggestions made in Section 3.3.1. By combining all the optimal settings as discussed in the previous sections (§3.1-§3.3), the prediction error at each stage can be reduced by 3-5 orders of magnitude within 10^5 iterations.

Compared with single-stage training, Fig. 12(*a*&*b*) shows that multi-stage training can reduce both the data loss and equation loss by more than 20 orders of magnitude within the same number of iterations. In this instance, the number of weights in the single-stage

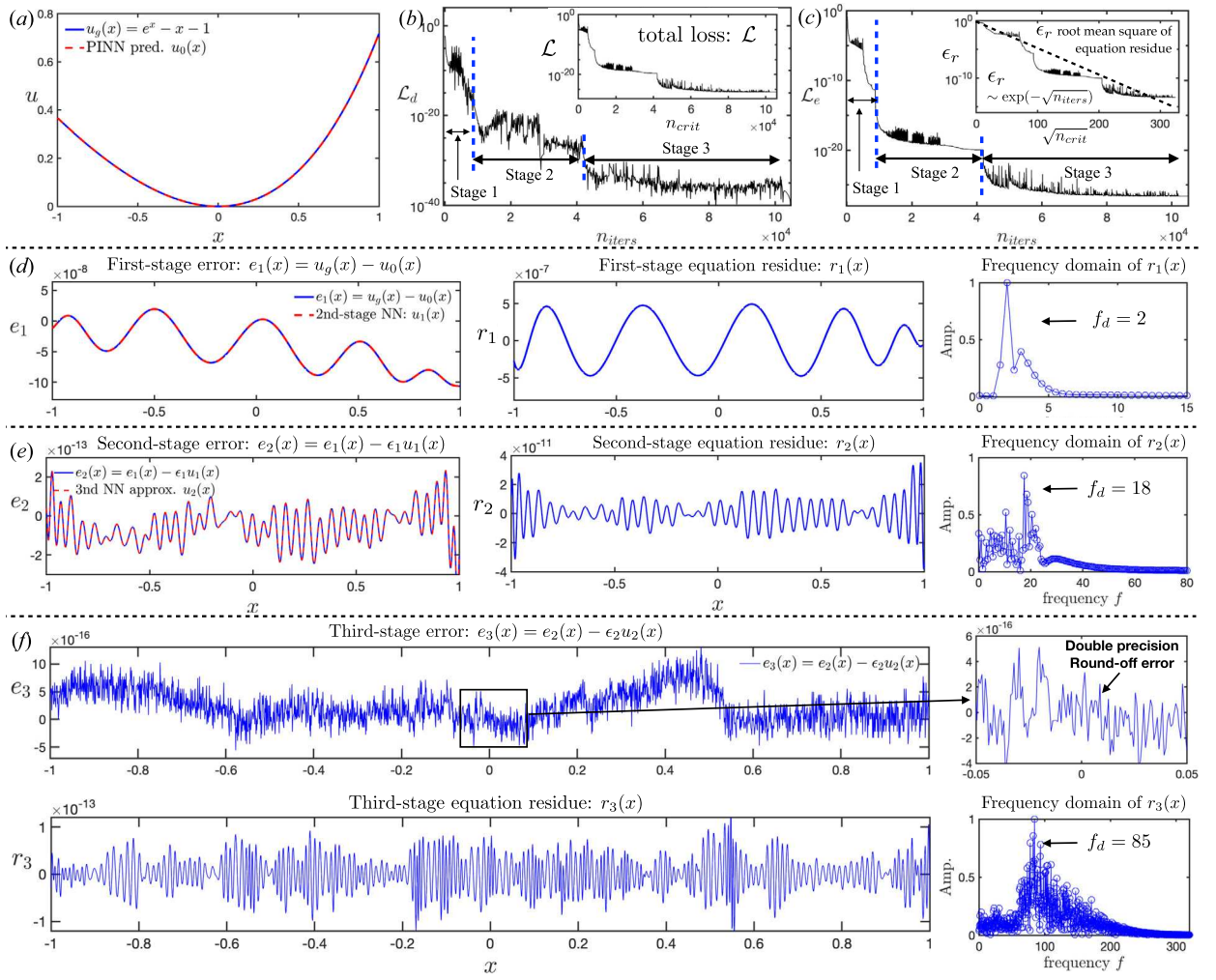


Fig. 11. Multi-stage gPINNs for 1D equations. (*a*) Comparison of the first-stage trained neural network (red dashed curve) with the exact solution $u_g(x)$ (blue curve) to equation (3.20). (*b*) Data loss and (*c*) equation loss over iterations of three-stage training. The inset of (*b*) shows the evolution of the total loss $\mathcal{L}$ over iterations. The inset of (*c*) shows that the evolution of the root mean square value ϵ_r of the equation residue $r(x,u)$ of the multi-stage neural networks follows $\epsilon_r \sim \exp(-\sqrt{n_{iters}})$, which is consistent with that for regression problems (Fig. 5*c*). (*d* & *e*) Comparison of the higher-stage trained network with the error of lower-stage training is shown in the left column. The equation residue $r_n(x)$ for different stages of training is in the middle. Frequency domain of the equation residue $r_n(x)$ at each stage is shown in the right column. (*f*) Prediction error $e_3(x)$ and the equation residue $r_3(x)$ after the third-stage of the training. The zoom-in figure (on the right) shows fluctuations in the prediction error $e_3(x)$, which is caused by the round-off error of the machine-precision of double-floating points.

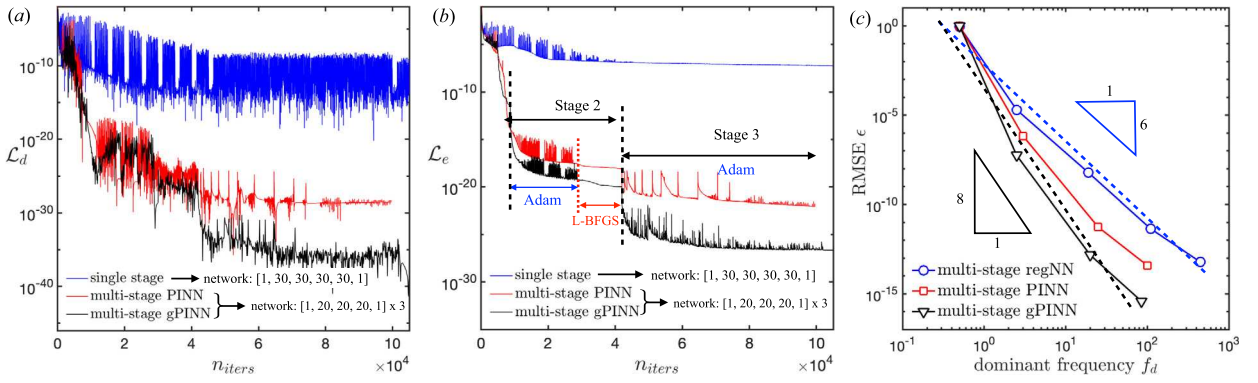


Fig. 12. Comparison of single-stage with multi-stage PINN training. (*a* & *b*) Comparison of the data loss (*a*) and equation loss (*b*) evolution over iterations between the single-stage training, multistage training of PINN, and multi-stage training of gPINN. (*c*) Relation of the dominant frequency f_d with the root mean square value ϵ of the error $e_n(x)$ after different stages of training for regression problems (blue), PINNs (red) and gPINNs (black).

Algorithm 4 Multi-stage training scheme for PINNs.

Prerequisite: normalize the equation with the magnitude of the largest term around $O(1)$.

1: Build the first neural network $u_0(x)$ using regular weight initialization.

2: Sample collocation points or data points e(i.e. boundary conditions) and training the neural network.

3: Calculate the output of the trained neural network $u_0(x)$ and corresponding equation residue $r_1(x, u_0)$.

4: Estimate the scale factor κ_1, and magnitude prefactor ϵ_1 of the prediction error e_1 for the first trained network u_0, based on the dominant frequency $f_d^{(r)}$ and magnitude ϵ_{r_1} of the equation residue $r_1(x, u_0)$, using the relations (3.18) and (3.19) or Algorithm 2.

5: Generate the ansatz of the solution for the second stage of training as

$$u_1^{(c)} = u_0(x) + \epsilon_1 u_1(x, \kappa_1), \tag{3.41}$$

where $u_1(x, \kappa_1)$ represents the second neural network utilizing the $sin(x)$ activation function in the first hidden layers and multiplying the scale factor κ_1 from Step 4 to the weight between the input layer and first hidden layer, we note that u_1 has normalized output value.

6: Substitute the ansatz (3.41) into the original equation. We note that, in the second stage, only the weight and biases in the network u_1 are trainable. The first trained network u_0 in the second stage is considered as a known function with *fixed* parameters.

7: Determine the optimal value of the equation weight γ using the relation (3.32) or Algorithm 3. Determine the optimal value of the weight γ_g for gradient constraint if the gradient-enhanced PINN is used.

8: Conduct the second stage of training and calculate the corresponding equation residue $r_2(x, u_1^{(c)})$.

9: Repeat Step 3 - 8 for higher stages of training until the equation residue $r_n(x, u_{n-1}^{(c)})$ is small enough. For each higher stage of training, the ansatz of the solution can be expressed in a recursive relation based on the previous trained network as

$$u_k^{(c)} = u_{k-1}^{(c)}(x) + \epsilon_k u_k(x, \kappa_k), \tag{3.42}$$

where u_k is the new network added at the stage k and $u_{k-1}^{(c)}(x)$ is the combined network from the previous stage of training. ϵ_k and κ_k are the magnitude prefactor and scale factor, respectively, estimated from the equation residue for the previous stage of training in Step 4.

10: Combine the neural networks from all n-stages of training to generate the final solution

$$u(x) = u_{n-1}^{(c)} = u_0 + \sum_{k=1}^{n-1} \epsilon_k u_k(x, \kappa_k). \tag{3.43}$$

The magnitude of error of the final solution $u(x)$ can be estimated from the equation residue $r_n(x, u_{n-1}^{(c)})$ for the last stage of training by the relation (3.19) or Algorithm 2.

network has been selected to be approximately equivalent to the total number across all three-stage networks. These results suggest that employing appropriate network settings and an effective training scheme plays a more essential role in successful training than simply increasing the size of neural network and the number of collocation points. Additionally, when combined with gPINN, the multistage training demonstrates an accelerated convergence rate. Fig. 11(f) shows that, after the three stages of gradient-enhanced PINN training, the prediction error of the final trained network with the exact solution reaches the machine precision of double floating points The observed oscillation in e_3 is primary attributable to round-off error.

The right panel of Fig. 11(d-f) displays the spectrum and dominant frequency of the equation residue after each stage of training. Fig. 12(c) further shows the relation of the dominant frequency f_d with the root mean square value ϵ of the prediction error $e_n(x)$ over the stages, which follows a power law $f_d \sim \epsilon^{-\alpha}$ for both regression problems and PINNs. We recall that the power law exponent α for regression problems is around 1/6. Compared with that, the power law exponent α for PINNs becomes noticeably smaller, around 1/7 for multi-stage training with regular PINNs and reduced further to 1/8 when using gradient-enhanced PINNs. As discussed in Section 2.2, this indicates that trained neural networks in PINNs achieve higher accuracy in capturing higher-order derivatives compared to regression problems. This is reasonable as PINNs involve differential equations that contain the derivatives of the solution. By minimizing the equation loss, PINNs constrain both the neural network and its derivatives to approach the exact solution, enhancing the capture of the high derivative information of the solution. The same reasoning applies to the gradient-enhanced PINNs, which result in an even lower exponent α since the gradient of equation residue involved further higher derivatives of the solution.

3.5. Application to 2D partial differential equations

Multi-stage training scheme for PINNs can also be applied to solve partial different equations (PDE). Fig. 13(a) shows the three-stage training to solve the diffusion equation

$$\frac{\partial u}{\partial t} = \frac{1}{2}\frac{\partial^2 u}{\partial x^2} + (1 - x^2 + t) \qquad \text{with} \quad u(x, 0) = u(\pm 1, x) = 0, \tag{3.44}$$

which has the exact solution,

$$u_g(t, x) = t(1 - x^2). \tag{3.45}$$

Consistent with regression problems, the convergence rate of multistage PINN method for solving 2D problem is slightly slower than that for 1D problem (inset of Fig. 13e). After three stage of training using RAR method and gradient-enhanced PINN, the predic-

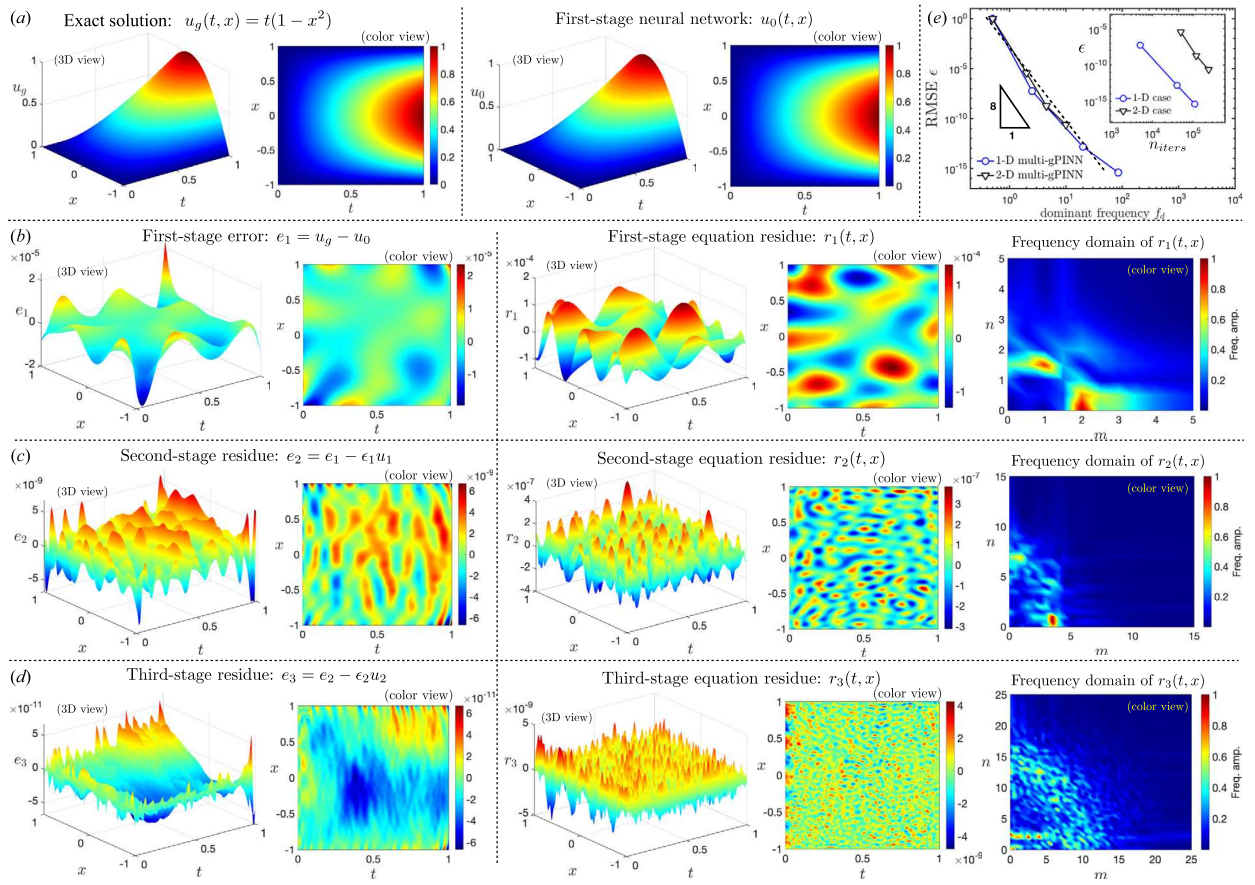


Fig. 13. Multi-stage gPINNs for 2D equations. (*a*) Comparison of the first-stage trained neural network with the exact solution $u_g(x,t)$ to the equation (3.44). (*b-d*) The error $e_n(x,t)$ of higher-stage trained network $u_n(x,t)$ with the exact solution $u_g(x,t)$ is shown in the left panel. The equation residue $r_n(x,t)$ for different stages of training and their frequency domains are shown in the right panel. (*e*) Relation of the dominant frequency f_d with the root mean square value ϵ of the error $e_n(x,t)$ after different stages of training follows the same power law as 1D problems, of which the exponent $\alpha = 1/8$. The inset shows that the number of iterations required for 2D problems to reach the same accuracy ϵ is more than that for 1D problems.

tion error $e_3(t,x)$ of the combined trained networks with the exact solution $u_g(t,x)$ is around $O(10^{-11})$. The accuracy of multistage training is still seven order-of-magnitude higher than that of the single-stage training. Fig. 13(*e*) shows that, when employing the multi-stage training scheme with gPINNs, the relation of the dominant frequency f_d with the root mean square value ϵ of the prediction error $e(x,y)$ for solving both 1D and 2D equations, follows the same power law (2.11) with an exponent of $\alpha \approx 1/8$. This observation is consistent with the results observed in regression problems (Fig. 6*e*).

We note that achieving a low prediction error of $O(10^{-11})$ for solving 2D partial differential equations via classical numerical method, such as finite difference, would require an extensive number of grid points. For instance, considering the central difference method along the x-direction, to reach 10^{-11}, we would need a grid size in the x-direction of $h_{(x)} \sim O(\sqrt{10^{-11}}) \sim O(10^{-5})$, namely 10^6 grid points for each time step. Even with a 4th-order Runge-Kutta method along the t direction, the step size in t-direction would need to be $h_{(x)} \sim O(\sqrt{(10^{-11})^{1/4}} \sim O(10^{-3})$, requiring 10^3 time steps. Consequently, the total number of grid points needed to achieve this accuracy across the entire domain would be on the order of $O(10^9)$.

In contrast, our approach utilizes fully connected neural network with 4-hidden layers consisting of 30 units for each stage of the training. Thus, the total number of weights and biases used to express the solution is only around $3\times4\times30^2 \approx 10^4$, which is five order of magnitude less than the number of grid points used in a discretized solution. This demonstrates the efficiency and effectiveness of the multi-stage PINN in achieving accurate solution with significantly fewer parameters compared to classical numerical methods.

4. Generalization of multistage PINN to a combined forward-and-inverse problem

Here we investigate a specific class of problems in mathematics that requires solving equations (forward problem) and simultaneously inferring unknown parameters in the equation (inverse problem) with a high demand for accuracy, for example, finding self-similar blow-up solutions for nonlinear fluid equations [9]. The physical significance of the problem was explained in Eggers (2015) [50], and a prior study [9] has developed the implementation of PINNs to solve it. In these problems, the multistage PINN method can play a critical role in achieving accurate results.

Here we focus on the 1D inviscid Burgers' equation for which we know the exact solutions. In Appendix D, we provide a summary of the background knowledge and PINN implementation. The task for the PINN involves discovering the *smooth* solution to the nonlinear self-similar Burgers' equation,

$$-\lambda U + [(1+\lambda)y + U]\partial_y U = 0, \qquad \text{with} \quad U(-2) = 1\,, \tag{4.1}$$

where the solution U should be an odd function with respect to the independent variable y, and λ is the unknown parameter to be predicted by PINNs. Smooth solutions to (4.1) exist when $\lambda = 1/(2+2i)$ for $i = 1, 2, 3....$ For other λ values, the solution is non-smooth, exhibiting discontinuity at certain order of its derivatives at the origin $y = 0$. Finding the smooth solution with the correct value of λ numerically is challenging.

To address the issue, a prior study [9] leveraged PINNs and introduced an additional *smoothness* constraint into the loss function, which penalizes the higher-order derivative of the equation residue around the non-smooth position. We note that the minimal order of derivative needed for the smoothness constraint depends on specific problems. In general, it should be larger than the order of smoothness for the non-smooth solution (see Appendix D). Any higher derivatives with the order larger than the minimal value can be involved in the smoothness constraint as long as it remains computationally feasible. Here we focus on the first smooth solution of the self-similar Burgers' equation (4.1). The loss function can be expressed as

$$\mathcal{L} = (1-\gamma)\mathcal{L}_d + \gamma(\mathcal{L}_e + \gamma_g \mathcal{L}_g) + \gamma_s \mathcal{L}_s \qquad \text{with} \tag{4.2}$$

$$\mathcal{L}_d = (U(y=-2) - 1)^2 \qquad \text{and} \qquad \mathcal{L}_e = \frac{1}{N_c}\sum_{i=1}^{N_c} \left| r\left(y_i, U(y_i)\right)\right|^2 \tag{4.3}$$

$$\mathcal{L}_g = \frac{1}{N_c}\sum_{i=1}^{N_c} \left| \frac{\partial r}{\partial y}\left(y_i, U(y_i)\right)\right|^2 \qquad \text{and} \qquad \mathcal{L}_s = \frac{1}{N_s}\sum_{j=1}^{N_s} \left| \frac{\partial^3 r}{\partial y^3}\left(y_j, U(y_j)\right)\right|^2 \tag{4.4}$$

$$\text{with} \qquad r(y, U) = -\lambda U + [(1+\lambda)y + U]\partial_y U \tag{4.5}$$

where $\mathcal{L}_d$ and $\mathcal{L}_e$ are the data loss and equation loss, respectively. $\mathcal{L}_g$ is the gPINNs implementation, which involves the first-order gradient of the equation residue $r(y, U)$. $\mathcal{L}_s$ is the smoothness constraint that incorporates the third derivative of the equation residue. While the equation loss $\mathcal{L}_e$ and gradient loss $\mathcal{L}_g$ are examined at N_c random collocation points y_i across the entire domain, the smoothness constraint $\mathcal{L}_s$ is calculated at y_j close to the origin (e.g. $|y_j| < 0.1$) with number $N_s \ll N_c$. Although the smoothness constraint depends on the equation residue, it can be viewed as an additional boundary condition for the solution to determine the value of λ.

Following Algorithm 4, Fig. 14(*a-d*) shows the first two stages of training for solving the self-similar Burgers' equation (4.1). We observe that the second-stage training successfully improves both the prediction error $e_2(y)$ of the trained network and the inferred lambda λ_2 by *four orders of magnitude*. However, in addition to the high-frequency error as previously seen for the higher-stages, we observed that the prediction error $e_2(y)$ from the second-stage training contains a low-frequency profile, which dominates over the high-frequency error. This disparity hinders the further reduction of the error by adding more stages of training based on Algorithm 4.

To understand the issue, we first study the occurrence of the high-frequency error in $e_2(y)$. The middle panel in Fig. 14(*e*) reveals that the equation residue $r_2(y)$ after the second stage of training exhibits a similar dominant frequency f_d to the high-frequency error in the prediction error $e_2(y)$. Using (3.9), we estimate the magnitude ϵ_2 of the prediction error $e_2(y)$ based on the magnitude ϵ_{r_2} of the equation residue $r_2(y)$ as $\epsilon_2 = \epsilon_{r_2}/(2\pi f_d) \sim O(10^{-13})$. This is consistent with the magnitude of the high-frequency error in $e_2(y)$. Then, following Algorithm 4, we create a new network $U_2(y)$ multiplied by the magnitude of $O(10^{-13})$ for the third-stage training. Fig. 14(*f*) shows that the high-frequency error in $e_2(y)$ after the third-stage training does vanish in $e_3(y)$. However, the magnitude of the prediction error $e_3(y)$ and the inferred λ_3 after the third-stage of training (Fig. 14*f*) remains nearly the same as those of the second-stage training (Fig. 14*e*).

The issue appears to be related to the existence of the low-frequency profile in $e_2(y)$. We recall that the prediction error of the training is estimated by comparing the trained networks at the inferred λ_2 with the exact smooth solution at $\lambda_g = 0.5$. Therefore, the error of the trained network is influenced not only by the equation residue, but also by the error ϵ_λ of the inferred λ. To assess the impact of the inference error ϵ_λ on the prediction error $e_2(y)$, we perform a similar analysis as discussed in Section 3.1.1, introducing the ansatz of the exact solution U_g and exact value of λ_g as

$$U_g(y) = U_0(y) + \epsilon U_e(y) \qquad \text{and} \quad \lambda_g = \lambda_0 + \epsilon_\lambda \tag{4.6}$$

where U_0 represents the lower-stage trained network and λ_0 is the inferred λ from the lower-stage training. $\epsilon U_e(y)$ represents the prediction error of the trained network and ϵ_λ is the error of inferred λ. Both ϵ and ϵ_λ are much smaller than 1. Substituting (4.6) into (4.1) and removing higher-order small terms $O(\epsilon^2)$, we have

$$\epsilon\left\{(\partial_y U_0 - \lambda_0)U_e + [(1+\lambda_0)y + U_0]\partial_y U_e\right\} = \underbrace{(\lambda U_0 - [(1+\lambda_0)y + U_0]\partial_y U_0)}_{\text{equation residue: } -r_0} + \underbrace{\epsilon_\lambda (U_0 - y\partial_y U_0)}_{\text{term from } \epsilon_\lambda:\, r_\lambda} \tag{4.7}$$

which can be viewed as the governing equation for the higher-stage network. In addition to the equation residue $r_0(y)$ from the lower-stage training, the higher-stage equation (4.7) involves a new source function $r_\lambda(y)$ that is associated with the error ϵ_λ of the

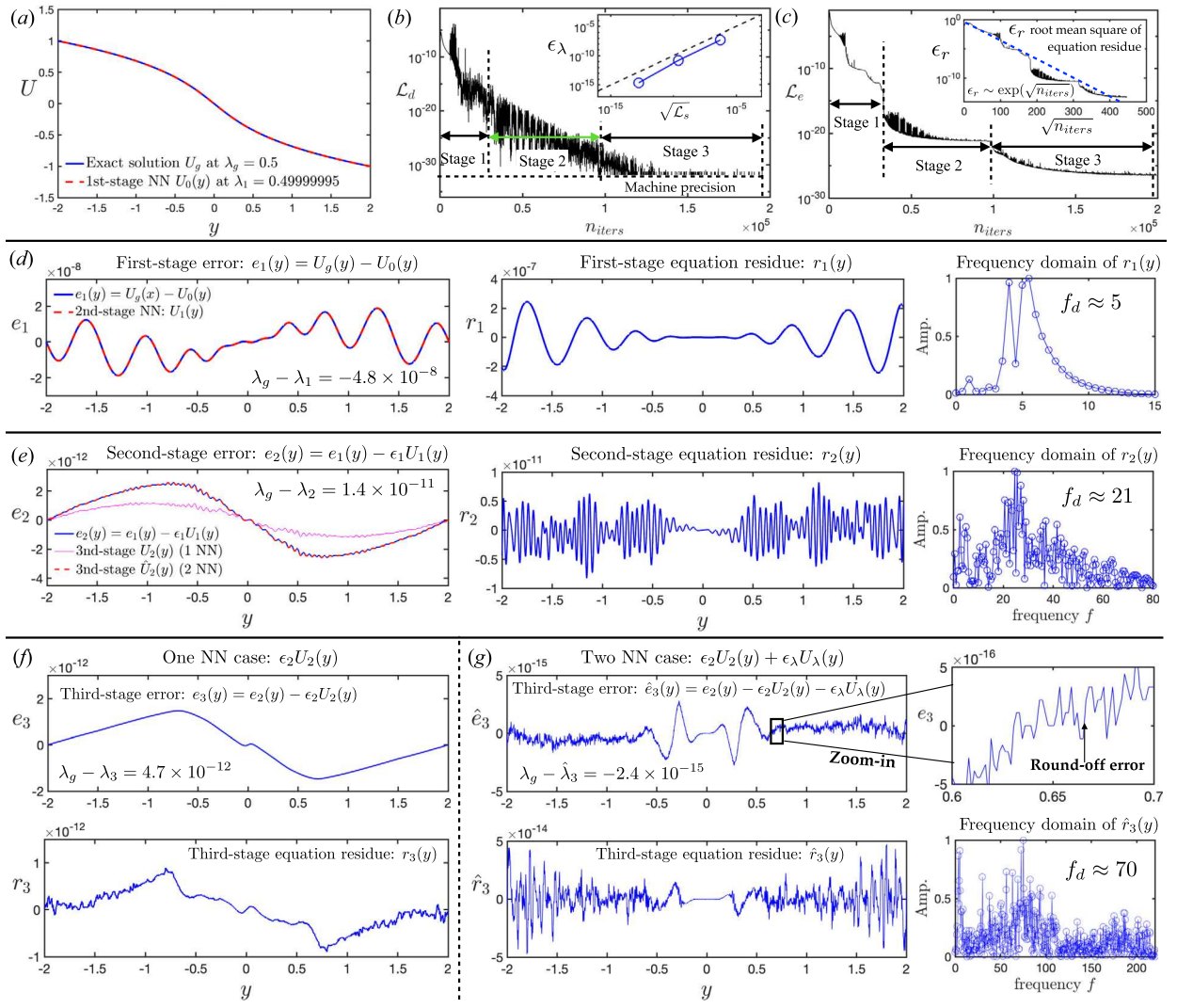


Fig. 14. Multi-stage gPINNs for a combined forward and inverse problem. (*a*) Comparison of the first-stage trained network $U_0(y)$ at the inferred λ_1 (red dashed curve) with the exact profile of the first smooth solution $U_g(y)$ with $\lambda_g = 0.5$ to the self-similar Burgers' equation (4.1). (*b*) Data loss and (*c*) equation loss over iterations of three-stage training. The inset of (*b*) shows the relation of the error ϵ_λ of inferred λ with the loss of the smoothness constraint $\mathcal{L}_s$ after different stages of training. The dash line indicates the relation $\epsilon = \sqrt{\mathcal{L}_s}$. The inset of (*c*) shows that the evolution of the root mean square value ϵ_r of the equation residue $r(y,u)$ over iterations of the multi-stage neural networks follows $\epsilon_r \sim \exp(-\sqrt{n_{iters}})$, consistent with that of regular forward problem (Fig. 11*c*). (*d* & *e*) The prediction error $e_n(y)$ (left), equation residue $r_n(y)$ (right) and its frequency domain (right) for the first (*d*) and second (*e*) stages of training. Comparison of higher-stage trained networks with the lower-stage prediction error is shown in the left panel. (*f*) Prediction error $e_3(y)$ and equation residue $r_3(y)$ for the third-stage training using only one additional neural network $U_3(y)$. It successfully reduce the high-frequency error from the second stage but fails to reduce its low-frequency error. (*g*) Prediction error $\hat{e}_3(y)$ and equation residue $\hat{r}_3(y)$ for the third-stage training using two neural networks $U_3(y)$ and $U_\lambda(y)$, which successfully reduces both the high-frequency error associated with the lower-stage equation residue $r_2(y)$ and the low-frequency error associated with the error ϵ_λ of the inferred λ_2. The zoom-in figure shows that the prediction after three stages of training approaches the machine precision of double-floating points.

inferred λ. While the equation residue $r_0(y)$ exhibits high-frequency behavior, the source function $r_\lambda(y)$ is influenced by the profile of the trained network $U_0(y)$, exhibiting the low-frequency profile in the prediction error $e_2(y)$ (Fig. 14*e*).

Considering the low frequency nature of the source function $r_\lambda(y)$, the magnitude of prediction error ϵ in (4.7) associated with $r_\lambda(y)$ is expected to be similar to the error ϵ_λ of the inferred λ, which is approximately $O(10^{-12})$, consistent with our results (Fig. 14*e*). In contrast, the prediction error associated with the high-frequency equation residue r_0, as discussed earlier, is only around $O(10^{-13})$. This explains why the low-frequency profile dominates the prediction error $e_2(y)$.

Here, we note that the error ϵ_λ of inferred λ is calculated using the known exact value $\lambda_g = 0.5$. However, in many other problems, the exact value of λ_g is unknown. Thus, an alternative way to quantify the inference error ϵ_λ is from the loss $\mathcal{L}_s$ of the smoothness constraint. The inset of Fig. 14(*b*) shows that the inference error ϵ_λ after different stages of training is proportional to $\sqrt{L_s}$, i.e. $\epsilon_\lambda = \sqrt{L_s}$ (dashed line). This suggests that we can use $\sqrt{L_s}$ to estimate the inference error ϵ_λ, as well as the magnitude prefactor for the higher-stage network U_λ associated with $r_\lambda(y)$.

Since the prediction error is dominated by the low-frequency source function $r_\lambda(y)$, one might intuitively consider creating a single low-frequency network multiplied by the error ϵ_λ of the inferred λ for the third-stage training. However, this approach is not effective because the smoothness constraint (4.4) depends on the higher-order derivative of the equation residue. By using only a low-frequency network, it would be challenging to reduce the high-frequency equation residue. Therefore, our proposed solution is to create two networks for both source functions in (4.7) at the third-stage training, namely

$$\epsilon_2 U_2(y) + \epsilon_\lambda U_\lambda(y), \tag{4.8}$$

where the magnitude prefactor ϵ_2 and modified scale factor $\hat{\kappa}$ (for frequency) for the high-frequency network $U_2(y)$ associated with the equation residue r_2 can be determined by the relations (3.17) and (3.16) or Algorithm 2. The low-frequency network U_λ associated with the error ϵ_λ of the inferred λ_2 can be directly multiplied by the inference error ϵ_λ. Fig. 14(g) shows that, using combined two networks for the third-stage training, the prediction error $\hat{e}_3(y)$ is successfully reduced by another three orders of magnitude, eventually approaching the machine precision of double-floating points.

5. Discussion

We note that the principle of multi-stage neural networks is similar to that of Fourier series, which combines a series of sine or cosine functions, ranging from low to high frequencies, to approximate functions. Provided that the series converge, the error between the Fourier series expansion of a given order and the target function possesses lower magnitudes but higher frequency than any terms in the series. To further minimize the error, higher-order sine or cosine functions need to be incorporated into the series, leading to additional higher-frequency error.

The introduction of new neural networks in multi-stage neural networks (MSNNs) is analogous to the inclusion of higher-order trigonometric functions in the Fourier series expansions. However, in contrast to sines and cosines, deep neural networks with appropriate settings offer stronger function representation capacity. Our finding indicates that the magnitude ϵ of error after each stage of training follows a inverse power law relation with the dominant frequency f_d of the error, i.e. $\epsilon \sim f_d^{-\alpha}$, with the exponent $\alpha \approx 1/6$ for regression problems, $\alpha \approx 1/7$ for regular PINNs, and $\alpha \approx 1/8$ for gradient-enhanced PINNs (gPINNs).

In comparison, the power law exponent α for Fourier series is roughly around $\alpha \approx 0.5$ [51], much larger than that of MSNNs. This indicates that, to achieve the same error magnitude, the error frequency generated by MSNNs could be several orders of magnitude lower than that by Fourier series. This observation confirms that MSNNs serve as a superior tool capable of accurately approximating target functions, as well as their high derivative information.

The multi-stage neural networks (MSNNs) developed in this work remains in their *early* stage, and mainly serve as a proof of concept to demonstrate that neural networks can practically achieve high accuracy. It is crucial to recognize that MSNNs should not be regarded as a substitute for classical numerical methods, but rather as a complementary approach. In fact, there remains several challenges that need to be addressed in the MSNN method. One of the primary challenges pertains to high-dimension problems. As shown in Fig. 6 and 13, the convergence rate of MSNNs for both 2D regression problems and PINNs are consistently slower than that for 1D problems. It is expected that this challenges will become more pronounced in higher-dimensional problems.

The second major challenge pertains to approximating functions or predict solutions with steep gradients. Near the regions where the target function exhibits steep gradients, neural networks often encounter local peaks in the error or the equation residue during training. The presence of these peaks hinder the reduction of error in successive stages, necessitating their removal before proceeding to the next stage of training. We note that functions with steep gradients are commonly encountered in differential equations, such as stiff equations, nonlinear equations, or singularly-perturbed equations (see Appendix C). How to effectively solve these types of equations via PINNs is critical but beyond the scope of this paper.

Additional investigations can be conducted to further improve the MSNN method, particularly regarding the optimal timing for transitioning between training stages. As training progresses, the rate of convergence for loss typically diminishes. Determining whether to advance to the next stage for improved convergence rates, or to extend the current stage until loss plateaus for maximal error reduction, requires careful analysis and additional research. Moreover, the multi-stage training reduces the necessity for overly large networks to achieve high accuracy within a single stage of training. Identifying the most efficient neural network size for each stage that can minimize the total number of training parameters (weights and biases) and, consequently, the computational cost of MSNN training presents another valuable avenue for future investigation.

6. Conclusion

We introduced the multi-stage neural networks (MSNNs) for both regression problems and physics-informed neural networks. Inspired by perturbation theory, we sequentially introduced new stages of training with new neural networks to capture the residue from the previous stage of training. This enable MSNNs to reach unprecedentedly high accuracy over stages. We showed that three stages of MSNN training can reach machine precision, making neural networks truly universal function approximators in practice. This new method can be widely applied to many scientific domains, such as mathematical and nonlinear physical science where the precision matters.

The success of MSNNs lies in two aspects. The first is the idea of staged training itself. Deep neural networks often suffer from spectral biases, making it challenging to capture the full spectrum of the target function in a single stage of training, even when employing large-sized networks with an increased number of data or collocation points. As a result, the training loss tends to plateau

after a certain number of iterations. However, by employing multi-stage training, the previously plateaued error can be substantially reduced in each successive stage, which enables MSNNs to progressively capture finer details of the target function.

The second aspect for the success of MSNNs is the specific design of each new-stage network based on the training error from the previous stage. The neural-network predictions in successive stages exhibits significantly small magnitude and high frequency compared to the previous stages. We showed that, by employing optimal magnitude prefactor ϵ and scale factor κ with `sin` activation function, accurate predictions of functions with small magnitudes and high frequencies can be achieved. This enables the effective capture of intricate features in each successive stage.

To maximize the performance of each stage of training, we also studied the optimal value of ϵ_n and κ_n for each stage. For regression problems, the ϵ_n is equal to the magnitude (root mean square value) of the error $e_n(\mathbf{x})$ between the trained networks in the previous stage and the ground truth $u_g(\mathbf{x})$, and κ_n is proportional to the dominant frequency f_d of the error $e_n(\mathbf{x})$.

However, for physics-informed neural networks (PINNs), the prediction error $e_n(\mathbf{x})$ is not directly available and needs to be inferred from the equation residue $r_n(\mathbf{x})$ from the previous stage of training. Considering the fact that the governing equations for higher-stage training are essentially *linear*, regardless of the nonlinearity of original equations, the theoretical relations between the magnitude and frequency of the prediction error $e_n(\mathbf{x})$ and the equation residue $r_n(\mathbf{x})$ were derived in Section 3. We also presented an algorithm that can effectively estimate the magnitude of the prediction error from the equation residue. The *linear* property for the higher-stage training ensures the effectiveness and generality of multi-stage training scheme for PINNs.

Moreover, we discussed several other optimal settings that can enhance the efficiency of multi-stage PINN training. These include the equation weight γ, number of collocation points N_c, choice of optimizer, and advanced PINN techniques in the literature, such as RAR method and gPINNs.

Leveraging all the optimal settings discussed in this work, we showed that multi-stage neural networks (MSNNs) can significantly reduce the prediction error for both regression problems and PINNs, approaching machine precision. Furthermore, MSNNs showcases their capability in solving combined-forward-and-inverse problems to machine precision, a task typically challenging for classical numerical methods, but of great importance in mathematical and physical sciences. However, there remains many questions and challenges to be addressed for further enhancing the MSNN method.

CRediT authorship contribution statement

Yongji Wang: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Ching-Yao Lai:** Conceptualization, Funding acquisition, Investigation, Project administration, Resources, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The codes associated with this work are available on Github, https://github.com/YaoGroup/MultistageNN

Acknowledgements

We thank T. Buckmaster and J. Gómez-Serrano for helpful discussions regarding the application of multi-stage neural networks to critical mathematical questions. We acknowledge the Office of the Dean for Research at Princeton University for partial funding support via the Dean for Research Fund for New Ideas in the Natural Sciences. C.-Y.L. acknowledge the National Science Foundation for funding via Grant No. DMS-2245228. We also gratefully acknowledges financial support from the Schmidt Data X Fund at Princeton University made possible through a major gift from the Schmidt Futures Foundation.

Appendix A. Neural network error under different settings

Systematic experiments (Fig. 15) show that the root mean square value (RMS) ϵ of the error $e(x)$ between the trained network $u_0(x)$ and the data from the target function $u_g(x)$ remains unchanged even when the number of either layers (Fig. 15*a*) or units (Fig. 15*b*) is increased. Although the RMS error ϵ does slightly decrease with an increase in training data (Fig. 15*c*), this reduction is smaller than the standard deviation of eight repetitive experiments conducted with different random initializations, and thus is negligible. These results suggest that the plateau in training loss is not due to insufficient neural network size or lack of training data, but instead arises from inherent limitations of the training process itself. Fig. 15(*d*) presents the training loss for two different optimization methods. Compared to Adam [35], a first-order gradient descent method, L-BFGS [52], a quasi-Newton method, exhibits a higher convergence rate. However, training with L-BFGS quickly falls into a local minimum after reaching the same plateau as Adam. This suggests that the loss plateau is not optimizer-specific.

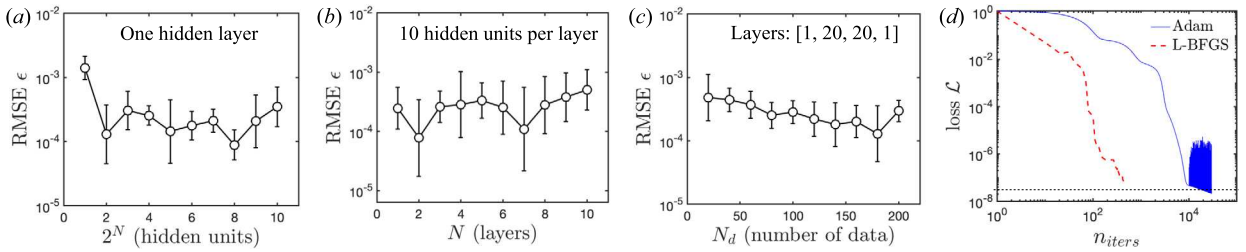


Fig. 15. Root mean square (RMS) value ϵ of the error between the target function and trained network using different number of (a) hidden units, (b) layers, and (c) training data, and (d) different types of optimizers, which show no big difference. Error bars show the standard deviation of eight repetitive experiments with different random initialization.

Appendix B. Effect of data magnitude on neural network training

Fig. 16 shows that a neural network using regular weight initialization methods has difficulty capturing training data either much larger or smaller than 1.

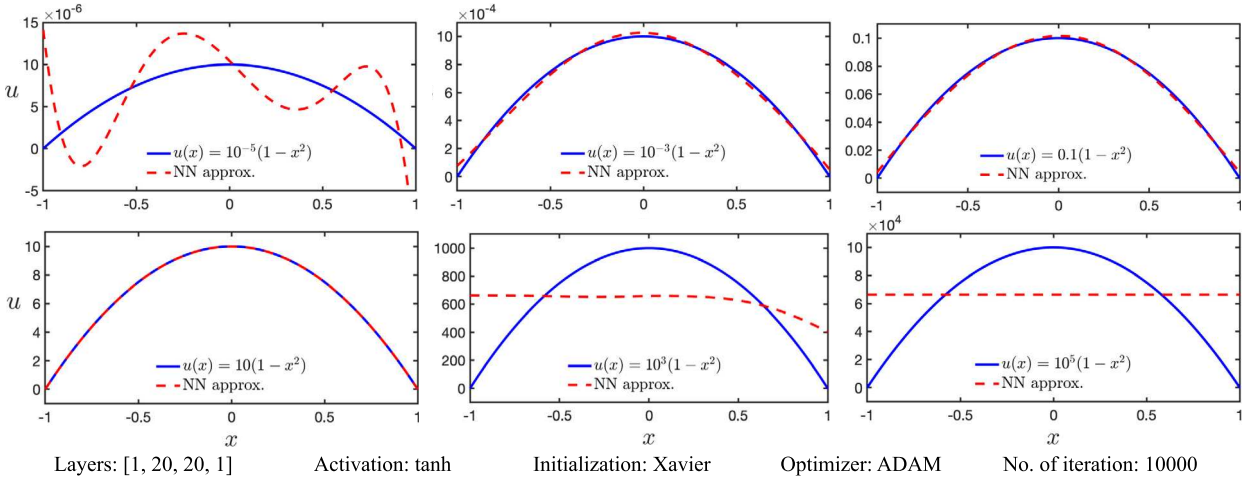


Fig. 16. Fitting of neural networks to the data with different magnitudes without normalization. It shows that the network is hard to fit data with magnitude either too much larger or smaller than 1.

Appendix C. Two extreme types of equations

There are two extreme types of equation where the general settings of networks derived in (3.19) and (3.18) for the high-stage training do not strictly hold. The *first* case is when the equation involves nonlinear term with high-order of derivatives, for example,

$$\left(\frac{d^8u}{dx^8}\right)^2 - u = F(x). \tag{C.1}$$

Substituting the ansatz (3.5) into (C.1) gives

$$-\epsilon\left(2\frac{d^8u_0}{dx^8}\frac{d^8u_1}{dx^8} - u_1\right) - \epsilon^2\left(\frac{d^8u_1}{dx^8}\right)^2 = r(x,u_0) = \left(\frac{d^8u}{dx^8}\right)^2 - u_0 - F(x) \tag{C.2}$$

where $r(x,u_0)$ is the equation residue of u_0. When u_1 is a high-frequency function with dominant frequency f_d satisfying the criterion,

$$(2\pi f_d)^8\epsilon > 1 \qquad \Longrightarrow \qquad f_d > \epsilon^{-1/8}, \tag{C.3}$$

the nonlinear term of u_1 on the left-hand side of (C.2) is no longer negligible and becomes the dominant term in the equation. The magnitude and frequency of $u_1(x)$, thus, need to be reassessed by balancing the nonlinear term with the equation residue $r(x,u_0)$. This results in the dominant frequency $f_d^{(1)}$ of $u_1(x)$ to be $f_d^{(1)} = f_d^{(e)}/2$, rather than $f_d^{(1)} = f_d^{(e)}$ from (3.16), where $f_d^{(e)}$ represents the dominant frequency of the equation residue $r(x,u_0)$.

However, we note that, although the dominant frequency $f_d^{(e)}$ of equation residue is larger, it still captures the order of magnitude of the actual frequency of $u_1(x)$. We recall from Fig. 4(c) that neural networks with modified scale factor $\hat{\kappa}$ larger than the criterion

$\hat{\kappa} > \pi f_d$ can reach the same high-accuracy when fitting high-frequency functions. This indicates that setting the scale factor κ based on a larger dominant frequency $f_d^{(e)}$ for the network of u_1 remains a good option to solve (C.2).

The *second* case is when the equation involves singular perturbation terms, for example

$$\alpha \frac{d^4u}{dx^4} + \frac{d^2u}{dx^2} - u = F(x) \qquad \text{with} \quad \alpha \ll 1, \tag{C.4}$$

where the coefficient before the highest-order derivative term is much smaller than the others. This type of equation is very common in physical sciences, such as boundary-layer problems. Substituting the ansatz (3.5) into (C.4) gives

$$-\epsilon \left(\alpha \frac{d^4u_1}{dx^4} + \frac{d^2u_1}{dx^2} - u_1 \right) = r(x, u_0) = \alpha \frac{d^4u_0}{dx^4} + \frac{d^2u_0}{dx^2} - u_0 - F(x) \tag{C.5}$$

where $r(x, u_0)$ is the equation residue of u_0. Based on (C.5), the dominant frequency $f_d^{(1)}$ of $u_1(x)$ remains equal to that of the equation residue $r(x, u_0)$. However, when $\alpha < (2\pi f_d)^{-2}$, the dominant term on the right-hand side of (C.5) is not the one with the highest-order derivative of u_1, but the term with the second-order derivative. The magnitude ϵ of the error is, then, determined by

$$\epsilon = \frac{\epsilon_r}{\left[2\pi f_d^{(1)}\right]^2}, \qquad \text{rather than} \quad \frac{\epsilon_f}{\left[2\pi f_d^{(1)}\right]^4 \alpha} \quad \text{based on (3.17).} \tag{C.6}$$

In fact, the actual challenges of solving singular perturbation equation via PINNs is more than the violation of the expression (3.17) for setting the higher-stage neural network. According to asymptotic analysis, the existence of singular perturbation term in the equation indicates that the solution to the equation has a narrow inner region where local gradient is very large. This property makes both the first-stage and higher-stage training of networks difficult. More discussion of the challenge is given in the Discussion (Section 6) of the paper. However, the solution to this challenge is beyond the scope of this paper.

Appendix D. 1D inviscid Burgers' equation

This section summarizes the exact self-similar blow-up solutions to the 1D inviscid Burgers' equation [50] and the PINN implementation developed in Wang et al. [9] to find it numerically.

Without viscous dissipation, the 1D inviscid Burgers' equation is given as

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0, \tag{D.1}$$

which has a shock wave solution where the velocity becomes discontinuous at a finite time, exhibiting a singularity where the solution blow up. However, right before the time when the shock/singularity is formed, the velocity profile remains smooth and continuous, and follow a self-similar structure near the singularity formation. We suppose the singularity occurs at $t = t_0$ and $x = x_0$. The self-similar coordinates can be written as

$$s = -\log(t_0 - t), \qquad y = \frac{x - x_0}{(t_0 - t)^{1+\lambda}}, \tag{D.2}$$

where s and y are the local time and spatial coordinates, respectively. When s goes to infinity, time t approaches to the blow-up time t_0, but can never go beyond that. In the meantime, the new self similar coordinate y allow us to zoom into and examine the solution profile around the singularity as time approaches t_0. The solution u follows the ansatz [9]

$$u(x, t) = (t - t_0)^{\beta} U(y, s) \tag{D.3}$$

where $U(y, s)$ indicates the self-similar profile near the singularity with β to be determined. Substituting the ansatz (D.3) into the equation (D.1) gives $\beta = \lambda$. Thus, the self-similar form of the Burgers' equation becomes

$$(\partial_s - 1)\lambda U + [(1 + \lambda)y + U]\partial_y U = 0. \tag{D.4}$$

We assume that when approaching the blow-up time t_0, namely s goes to infinity, time derivative term in (D.4) vanishes, and the self-similar profile $U(y, s)$ reaches steady state. Then, the steady state profile $\tilde{U}(y)$ is governed by

$$-\lambda \tilde{U} + [(1 + \lambda)y + \tilde{U}]\partial_y \tilde{U} = 0. \tag{D.5}$$

For simplicity, we consistently use U to represent the steady state solution in the rest of the section. The parameter λ (D.2), the rate at which singularity forms, remains unknown and is the key parameter to be inferred via the multi-stage neural networks.

To guarantee the equation (D.5) is well-posed globally in the local coordinates, the self-similar solution U must be an odd function. Theoretically, there exist solutions to (D.5) for each value of λ. The analytic solutions to the self-similar Burgers' equation (D.5) are

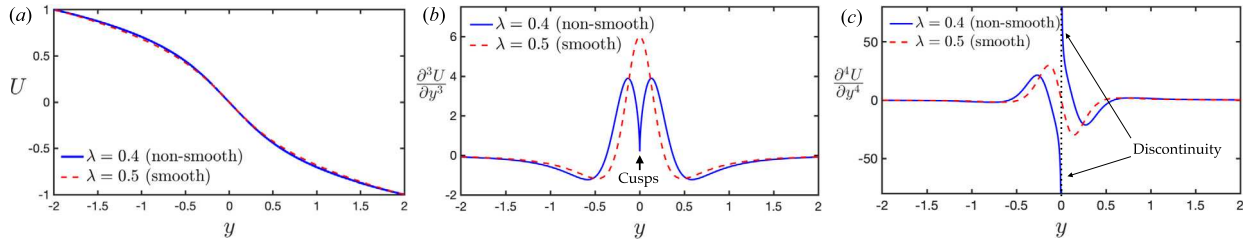


Fig. 17. Exact solutions to the Burgers' equation. (*a*) Exact first smooth and nearby non-smooth solutions to the self-similar Burgers' equation (D.5). (*b*) the third derivative of the non-smooth solution ($\lambda = 0.4$) forms cusps at the origin which indicates its fourth derivative (*c*) at the origin become discontinuous. In comparison, the smooth solution ($\lambda = 0.5$) is continuous at all orders of derivative everywhere in the domain.

$$y = \begin{cases} -U - CU^{1+\frac{1}{\lambda}} & \text{for} \quad x \geq 0 \\ -U + C(-U)^{1+\frac{1}{\lambda}} & \text{for} \quad x < 0 \end{cases} \tag{D.6}$$

where C is a constant determined by the boundary condition. Here, we use $U(y=2) = -1$, which gives $C = 1$. From the analytic expression (D.6), we can see that λ, in fact, determines the smoothness of the solution. Here, the smoothness indicates the solution is continuous at all its derivative. When $\lambda = 1/(2+2i)$ with $i = 0, 1, 2, \ldots$, the solution is smooth everywhere in the domain. However, when $\lambda \neq 1/(2+2i)$, the expression (D.6) involves fractional power, causing the solution to be non-smooth at the origin. For example, Fig. 17(*c*) shows that the fourth derivative of the solution for $\lambda = 0.4$ is discontinuous at the origin. Here, we note that the non-smooth solutions have no physical meaning. Thus, finding the smooth solutions to (4.1) is the goal.

Prior study by Wang et al. [9], leveraged the continuous property of neural networks, showing that PINN can discover the smooth solution with associated λ by imposing the high-order derivative constraint at the non-smooth position, known as the *smoothness* constraint. Additionally, we impose odd symmetry of the solution U by constructing the function form $U = y[\mathrm{NN}_u(y) + \mathrm{NN}_u(-y)]$, where NN_u indicates a fully-connected neural network created for U. The *data loss* and *equation loss* for solving the Burgers' equation (D.5) are given as

$$\mathcal{L}_d = (U(y=-2) - 1)^2 \qquad \text{and} \qquad \mathcal{L}_e = \frac{1}{N_c}\sum_{i=1}^{N_c} \left| r\left(y_i, U(y_i)\right)\right|^2 \tag{D.7}$$

$$\text{with} \qquad r(y, U) = -\lambda U + ((1+\lambda)y + U)\partial_y U \tag{D.8}$$

where y_i indicates the random collocation points in the training domain $y \in [-2, 2]$ and N_c is their total number. Here we focus on finding the first smooth solution with known $\lambda_g = 1/2$. Utilizing the fact that the non-smooth solutions in the neighborhood of $\lambda = 0.5$ has unbounded fourth-order derivative, which appears in the third order of derivative of equation residue, the smoothness constraint is given as

$$\mathcal{L}_s = \frac{1}{N_s}\sum_{i=1}^{N_s} \left| \frac{d^3 r}{dy^3}\left(y_i,\, U(y_i)\right)\right|^2, \tag{D.9}$$

where y_i indicates the random collocation points close to the origin (e.g. $|y_i| < 0.1$) and N_s is their total number. Although the smoothness constraint depends on the equation residue, it can be simply considered as a boundary condition for the solution that help determine the value of λ.

References

[1] Yann LeCun, Yoshua Bengio, Geoffrey Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[2] O. Ronneberger, P. Fischer, T. Brox, U-net: convolutional networks for biomedical image segmentation, in: Medical Image Computing and Computer-Assisted Intervention (MICCAI), in: LNCS, vol. 9351, Springer, 2015, pp. 234–241.

[3] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng, Nerf: representing scenes as neural radiance fields for view synthesis, Commun. ACM 65 (1) (2021) 99–106.

[4] Ronan Collobert, Jason Weston, A unified architecture for natural language processing: deep neural networks with multitask learning, in: Proceedings of the 25th International Conference on Machine Learning, 2008, pp. 160–167.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, Bert: pre-training of deep bidirectional transformers for language understanding, arXiv preprint, arXiv:1810.04805, 2018.

[6] K.R. Chowdhary, Natural language processing, Fundam. Artif. Intell. (2020) 603–649.

[7] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, Stephan Hoyer, Machine learning–accelerated computational fluid dynamics, Proc. Natl. Acad. Sci. USA 118 (21) (2021) e2101784118.

[8] Pablo Lemos, Niall Jeffrey, Miles Cranmer, Shirley Ho, Peter Battaglia, Rediscovering orbital mechanics with machine learning, arXiv preprint, arXiv:2202.02306, 2022.

[9] Yongji Wang, Ching-Yao Lai, Javier Gómez-Serrano, Tristan Buckmaster, Self-similar blow-up profile for the Boussinesq equations via a physics-informed neural network, arXiv preprint, arXiv:2201.06780, 2022.

[10] Maziar Raissi, Paris Perdikaris, George E. Karniadakis, Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.

[11] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, Liu Yang, Physics-informed machine learning, Nat. Rev. Phys. 3 (6) (2021) 422–440.
[12] Kurt Hornik, Maxwell Stinchcombe, Halbert White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (5) (1989) 359–366.
[13] Kurt Hornik, Maxwell Stinchcombe, Halbert White, Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks, Neural Netw. 3 (5) (1990) 551–560.
[14] Pierre Baldi, Kurt Hornik, Neural networks and principal component analysis: learning from examples without local minima, Neural Netw. 2 (1) (1989) 53–58.
[15] Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, Michael W. Mahoney, Characterizing possible failure modes in physics-informed neural networks, Adv. Neural Inf. Process. Syst. 34 (2021) 26548–26560.
[16] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, Gordon Wetzstein, Implicit neural representations with periodic activation functions, Adv. Neural Inf. Process. Syst. 33 (2020) 7462–7473.
[17] Vishwanath Saragadam, Daniel LeJeune, Jasper Tan, Guha Balakrishnan, Ashok Veeraraghavan, Richard G. Baraniuk, Wire: wavelet implicit neural representations, arXiv preprint, arXiv:2301.05187, 2023.
[18] Ameya D. Jagtap, Kenji Kawaguchi, George Em Karniadakis, Adaptive activation functions accelerate convergence in deep and physics-informed neural networks, J. Comput. Phys. 404 (2020) 109136.
[19] Honghui Wang, Lu Lu, Shiji Song, Gao Huang, Learning specialized activation functions for physics-informed neural networks, arXiv preprint, arXiv:2308.04073, 2023.
[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[21] Ameya D. Jagtap, George E. Karniadakis, Extended physics-informed neural networks (XPINNs): a generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations, in: AAAI Spring Symposium: MLPS, 2021, pp. 2002–2041.
[22] Ben Moseley, Andrew Markham, Tarje Nissen-Meyer, Finite basis physics-informed neural networks (FBPINNs): a scalable domain decomposition approach for solving differential equations, arXiv preprint, arXiv:2107.07871, 2021.
[23] Mark Ainsworth, Justin Dong, Galerkin neural network approximation of singularly-perturbed elliptic systems, in: A Special Issue in Honor of the Lifetime Achievements of J. Tinsley Oden, Comput. Methods Appl. Mech. Eng. 402 (2022) 115169.
[24] Chun-Chen Tu, Paishun Ting, Pin-Yu Chen, Sijia Liu, Huan Zhang, Jinfeng Yi, Cho-Jui Hsieh, Shin-Ming Cheng, Autozoom: autoencoder-based zeroth order optimization method for attacking black-box neural networks, in: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 742–749.
[25] Pao-Hsiung Chiu, Jian Cheng Wong, Chinchun Ooi, My Ha Dao, Yew-Soon Ong, Can-PINN: a fast physics-informed neural network based on coupled-automatic–numerical differentiation method, Comput. Methods Appl. Mech. Eng. 395 (2022) 114909.
[26] Johannes Müller, Marius Zeinhofer, Achieving high accuracy with PINNs via energy natural gradient descent, in: International Conference on Machine Learning, PMLR, 2023, pp. 25471–25485.
[27] Levi McClenny, Ulisses Braga-Neto, Self-adaptive physics-informed neural networks using a soft attention mechanism, arXiv preprint, arXiv:2009.04544, 2020.
[28] Sifan Wang, Shyam Sankaran, Paris Perdikaris, Respecting causality is all you need for training physics-informed neural networks, arXiv preprint, arXiv:2203.07404, 2022.
[29] Sifan Wang, Yujun Teng, Paris Perdikaris, Understanding and mitigating gradient flow pathologies in physics-informed neural networks, SIAM J. Sci. Comput. 43 (5) (2021) A3055–A3081.
[30] Sifan Wang, Xinling Yu, Paris Perdikaris, When and why PINNs fail to train: a neural tangent kernel perspective, J. Comput. Phys. 449 (2022) 110768.
[31] Remco van der Meer, Cornelis W. Oosterlee, Anastasia Borovykh, Optimally weighted loss functions for solving PDEs with neural networks, J. Comput. Appl. Math. 405 (2022) 113887.
[32] Nathaniel Trask, Amelia Henriksen, Carianne Martinez, Eric Cyr, Hierarchical partition of unity networks: fast multilevel training, in: Mathematical and Scientific Machine Learning, PMLR, 2022, pp. 271–286.
[33] Amanda A. Howard, Sarah H. Murphy, Shady E. Ahmed, Panos Stinis, Stacked networks improve physics-informed training: applications to neural networks and deep operator networks, arXiv preprint, arXiv:2311.06483, 2023.
[34] Anthony Ralston, Philip Rabinowitz, A First Course in Numerical Analysis, Courier Corporation, 2001.
[35] Diederik P. Kingma, Jimmy Ba, Adam: a method for stochastic optimization, arXiv preprint, arXiv:1412.6980, 2014.
[36] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, Aaron Courville, On the spectral bias of neural networks, in: International Conference on Machine Learning, PMLR, 2019, pp. 5301–5310.
[37] Zhi-Qin John Xu, Yaoyu Zhang, Tao Luo, Yanyang Xiao, Zheng Ma, Frequency principle: Fourier analysis sheds light on deep neural networks, arXiv preprint, arXiv:1901.06523, 2019.
[38] Arthur Jacot, Franck Gabriel, Clément Hongler, Neural tangent kernel: convergence and generalization in neural networks, Adv. Neural Inf. Process. Syst. 31 (2018).
[39] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, Ren Ng, Fourier features let networks learn high frequency functions in low dimensional domains, Adv. Neural Inf. Process. Syst. 33 (2020) 7537–7547.
[40] Sifan Wang, Hanwen Wang, Paris Perdikaris, On the eigenvector bias of Fourier feature networks: from regression to solving multi-scale PDEs with physics-informed neural networks, Comput. Methods Appl. Mech. Eng. 384 (2021) 113938.
[41] Eric J. Michaud, Ziming Liu, Max Tegmark, Precision machine learning, Entropy 25 (1) (2023) 175.
[42] Xavier Glorot, Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.
[43] Charu C. Aggarwal, et al., Neural Networks and Deep Learning, vol. 10(978), Springer, 2018, p. 3.
[44] C. Cowen-Breen, Navigating Physically-Informed Loss Landscapes with Stochastic Gradient Descent, Bachelor's thesis, Princeton University, Princeton, NJ, May 2022.
[45] Yunona Iwasaki, Ching-Yao Lai, One-dimensional ice shelf hardness inversion: clustering behavior and collocation resampling in physics-informed neural networks, J. Comput. Phys. 492 (2023) 112435.
[46] Lu Lu, Xuhui Meng, Zhiping Mao, George Em Karniadakis, Deepxde: a deep learning library for solving differential equations, SIAM Rev. 63 (1) (2021) 208–228.
[47] Shu-Mei Qin, Min Li, Tao Xu, Shao-Qun Dong, Rar-PINN algorithm for the data-driven vector-soliton solutions and parameter discovery of coupled nonlinear equations, arXiv preprint, arXiv:2205.10230, 2022.
[48] Jeremy Yu, Lu Lu, Xuhui Meng, George Em Karniadakis, Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems, Comput. Methods Appl. Mech. Eng. 393 (2022) 114823.
[49] Chenxi Wu, Min Zhu, Qinyang Tan, Yadhu Kartha, Lu Lu, A comprehensive study of non-adaptive and residual-based adaptive sampling for physics-informed neural networks, Comput. Methods Appl. Mech. Eng. 403 (2023) 115671.
[50] Jens Eggers, Marco Antonio Fontelos, Singularities: Formation, Structure, and Propagation, vol. 53, Cambridge University Press, 2015.
[51] Erwin Kreyszig, Advanced Engineering Mathematics, 9th edition, John Wiley & Sons, 2007.
[52] Dong C. Liu, Jorge Nocedal, On the limited memory BFGS method for large scale optimization, Math. Program. 45 (1–3) (1989) 503–528.