Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 393 (2022) 114823
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems

Jeremy Yu[a], Lu Lu[b,*], Xuhui Meng[c], George Em Karniadakis[c,d]

[a] St. Mark's School of Texas, Dallas, TX 75230, USA
[b] Department of Chemical and Biomolecular Engineering, University of Pennsylvania, Philadelphia, PA 19104, USA
[c] Division of Applied Mathematics, Brown University, Providence, RI 02912, USA
[d] School of Engineering, Brown University, Providence, RI 02912, USA

Received 1 November 2021; received in revised form 24 February 2022; accepted 26 February 2022
Available online 18 March 2022

Abstract

Deep learning has been shown to be an effective tool in solving partial differential equations (PDEs) through physics-informed neural networks (PINNs). PINNs embed the PDE residual into the loss function of the neural network, and have been successfully employed to solve diverse forward and inverse PDE problems. However, one disadvantage of the first generation of PINNs is that they usually have limited accuracy even with many training points. Here, we propose a new method, gradient-enhanced physics-informed neural networks (gPINNs), for improving the accuracy of PINNs. gPINNs leverage gradient information of the PDE residual and embed the gradient into the loss function. We tested gPINNs extensively and demonstrated the effectiveness of gPINNs in both forward and inverse PDE problems. Our numerical results show that gPINN performs better than PINN with fewer training points. Furthermore, we combined gPINN with the method of residual-based adaptive refinement (RAR), a method for improving the distribution of training points adaptively during training, to further improve the performance of gPINN, especially in PDEs with solutions that have steep gradients.
© 2022 Elsevier B.V. All rights reserved.

Keywords: Deep learning; Partial differential equations; Physics-informed neural networks; Gradient-enhanced; Residual-based adaptive refinement

1. Introduction

Deep learning has achieved remarkable success in diverse applications; however, its use in solving partial differential equations (PDEs) has emerged only recently [1]. As an alternative method to traditional numerical PDE solvers, physics-informed neural networks (PINNs) [2,3] solve a PDE via embedding the PDE into the loss of the neural network using automatic differentiation. The PINN algorithm is mesh-free and simple, and it can be applied to different types of PDEs, including integro-differential equations [4], fractional PDEs [5], and stochastic PDEs [6,7]. Moreover, one main advantage of PINNs is that from the implementation point of view, PINNs solve inverse PDE problems as easily as forward problems. PINNs have been successfully employed to solve diverse problems in different fields, for example in optics [8,9], fluid mechanics [10], systems biology [11], and biomedicine [12].

* Corresponding author.
E-mail address: lulu1@seas.upenn.edu (L. Lu).

https://doi.org/10.1016/j.cma.2022.114823
0045-7825/© 2022 Elsevier B.V. All rights reserved.

Despite promising early results, there are still some issues in PINNs to be addressed. One open problem is how to improve the PINN accuracy and efficiency. There are a few aspects of PINNs that can be improved and have been investigated by researchers. For example, the residual points are usually randomly distributed in the domain or grid points on a lattice, and other methods of training point sampling and distribution have been proposed to achieve better accuracy when using the same number of training points, such as the residual-based adaptive refinement (RAR) [4] and importance sampling [13]. The standard loss function in PINNs is the mean square error, and Refs. [14,15] show that a properly-designed non-uniform training point weighting can improve the accuracy. In PINNs, there are multiple loss terms corresponding to the PDE and initial/boundary conditions, and it is critical to balance these different loss terms [16,17]. Domain decomposition can be used for problems in a large domain [18–20]. Neural network architectures can also be modified to satisfy automatically and exactly the required Dirichlet boundary conditions [21–23], Neumann boundary conditions [24,25], Robin boundary conditions [26], periodic boundary conditions [9,27], and interface conditions [26]. In addition, if some features of the PDE solutions are known a-priori, it is also possible to encode them in network architectures, for example, multi-scale and high-frequency features [11,28–31]. Moreover, the constraints in PINNs are usually soft constraints, and hard constraints can be imposed by using the augmented Lagrangian method [9].

In PINNs, we aim to train a neural network to minimize the PDE residual for each PDE, and thus we only use the PDE residual as the corresponding loss for each PDE. This idea is straightforward and used by many researchers in the area, and no attention has been paid to other types of losses for a PDE yet. However, if the PDE residual is zero, then it is clear that the gradient of the PDE residual should also be zero.

In this work, we develop the gradient-enhanced PINN (gPINN), which uses a new type of loss functions by leveraging the gradient information of the PDE residual to improve the accuracy of PINNs. The general idea of using gradient information has been demonstrated to be useful for function regression such as via Gaussian process [32] and neural networks (see the review article [33]). However, these methods were only developed for function regression by using the function gradients in addition to function values, and they cannot be used to solve forward or inverse PDEs. The only related technique is the input gradient regularization, which uses the gradient of the loss with respect to the network input as an additional loss [34]. However, input gradient regularization is an alternative technique of L^1 and L^2 regularization used to improve classification accuracy [34], adversarial robustness [35–37], and interpretability [35,38,39]. Hence, as we use the gradient of the PDE residual term with respect to the network inputs, our method is fundamentally different from these prior approaches which use gradient information for solving different objectives. In addition, we combine gPINN with the aforementioned RAR method to further improve the performance.

The paper is organized as follows. In Section 2, after introducing the algorithm of PINN, we present the extension to gPINN and gPINN with RAR. In Section 3, we demonstrate the effectiveness of gPINN and RAR for eight different problems, including function approximation, forward problems of PDEs, and inverse PDE-based problems. We systematically compare the performance of PINN, gPINN, PINN with RAR, and gPINN with RAR. Finally, we conclude the paper in Section 5.

2. Methods

We first provide a brief overview of physics-informed neural networks (PINNs) for solving forward and inverse partial differential equations (PDEs) and then present the method of gradient-enhanced PINNs (gPINNs) to improve the accuracy of PINNs. Next we discuss how to use the residual-based adaptive refinement (RAR) method to further improve gPINNs.

2.1. PINNs for solving forward and inverse PDEs

We consider the following PDE for the solution $u(\mathbf{x}, t)$ parametrized by the parameters $\boldsymbol{\lambda}$ defined on a domain Ω:

$$f\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right) = 0, \quad \mathbf{x} = (x_1, \ldots, x_d) \in \Omega, \tag{1}$$

with the boundary conditions

$$\mathcal{B}(u, \mathbf{x}) = 0 \quad \text{on} \quad \partial\Omega.$$

We note that in PINNs the initial condition is treated in the same way as the Dirichlet boundary condition.

To solve the PDE via PINNs, we first construct a neural network $\hat{u}(\mathbf{x};\boldsymbol{\theta})$ with the trainable parameters $\boldsymbol{\theta}$ to approximate the solution $u(x)$. We then use the constraints implied by the PDE and the boundary conditions to train the network. Specifically, we use a set of points inside the domain ($\mathcal{T}_f$) and another set of points on the boundary ($\mathcal{T}_b$). The loss function is then defined as [3,4]

$$\mathcal{L}(\boldsymbol{\theta};\mathcal{T}) = w_f\mathcal{L}_f\left(\boldsymbol{\theta};\mathcal{T}_f\right) + w_b\mathcal{L}_b\left(\boldsymbol{\theta};\mathcal{T}_b\right),$$

where

$$\mathcal{L}_f\left(\boldsymbol{\theta};\mathcal{T}_f\right) = \frac{1}{\left|\mathcal{T}_f\right|}\sum_{\mathbf{x}\in\mathcal{T}_f}\left|f\left(\mathbf{x};\frac{\partial\hat{u}}{\partial x_1},\ldots,\frac{\partial\hat{u}}{\partial x_d};\frac{\partial^2\hat{u}}{\partial x_1\partial x_1},\ldots,\frac{\partial^2\hat{u}}{\partial x_1\partial x_d};\ldots;\boldsymbol{\lambda}\right)\right|^2, \tag{2}$$

$$\mathcal{L}_b\left(\boldsymbol{\theta};\mathcal{T}_b\right) = \frac{1}{|\mathcal{T}_b|}\sum_{\mathbf{x}\in\mathcal{T}_b}\left|\mathcal{B}(\hat{u},\mathbf{x})\right|^2, \tag{3}$$

and w_f and w_b are the weights.

One main advantage of PINNs is that the same formulation can be used not only for forward problems but also for inverse PDE-based problems. If the parameter $\boldsymbol{\lambda}$ in Eq. (1) is unknown, and instead we have some extra measurements of u on the set of points $\mathcal{T}_i$. Then we add an additional data loss [3,4] as

$$\mathcal{L}_i\left(\boldsymbol{\theta},\boldsymbol{\lambda};\mathcal{T}_i\right) = \frac{1}{|\mathcal{T}_i|}\sum_{\mathbf{x}\in\mathcal{T}_i}\left|\hat{u}(\mathbf{x}) - u(\mathbf{x})\right|^2$$

to learn the unknown parameters simultaneously with the solution u. Our new loss function is then defined as

$$\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\lambda};\mathcal{T}) = w_f\mathcal{L}_f\left(\boldsymbol{\theta},\boldsymbol{\lambda};\mathcal{T}_f\right) + w_b\mathcal{L}_b\left(\boldsymbol{\theta},\boldsymbol{\lambda};\mathcal{T}_b\right) + w_i\mathcal{L}_i\left(\boldsymbol{\theta},\boldsymbol{\lambda};\mathcal{T}_i\right).$$

In this study, we choose the weights $w_f = w_b = w_i = 1$. In some PDEs, it is possible to enforce the boundary conditions exactly and automatically by modifying the network architecture [5,9,21,26], which eliminates the loss term of boundary conditions.

2.2. Formulation of gradient-enhanced PINNs (gPINNs)

In PINNs, we only enforce the PDE residual f to be zero; because $f(\mathbf{x})$ is zero for any $\mathbf{x}$, we know that the derivatives of f are also zero. Here, we assume that the exact solution of the PDE is smooth enough such that the gradient of the PDE residual $\nabla f(\mathbf{x})$ exists, and then propose the gradient-enhanced PINNs to enforce the derivatives of the PDE residual to be zero as well, i.e.,

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1},\frac{\partial f}{\partial x_2},\ldots,\frac{\partial f}{\partial x_d}\right) = \mathbf{0}, \quad \mathbf{x}\in\Omega.$$

Then the loss function of gPINNs is:

$$\mathcal{L} = w_f\mathcal{L}_f + w_b\mathcal{L}_b + w_i\mathcal{L}_i + \sum_{i=1}^{d} w_{g_i}\mathcal{L}_{g_i}\left(\boldsymbol{\theta};\mathcal{T}_{g_i}\right),$$

where the loss of the derivative with respect to x_i is

$$\mathcal{L}_{g_i}\left(\boldsymbol{\theta};\mathcal{T}_{g_i}\right) = \frac{1}{\left|\mathcal{T}_{g_i}\right|}\sum_{\mathbf{x}\in\mathcal{T}_{g_i}}\left|\frac{\partial f}{\partial x_i}\right|^2. \tag{4}$$

Here, $\mathcal{T}_{g_i}$ is the set of residual points for the derivative $\frac{\partial f}{\partial x_i}$. Although $\mathcal{T}_f$ and $\mathcal{T}_{g_i}$ $(i = 1,\ldots,d)$ can be different, in this study we choose $\mathcal{T}_{g_i}$ to be the same as $\mathcal{T}_f$. In this study, we choose the weights $w_{g_1} = w_{g_2} = \cdots = w_{g_d}$. We determine the optimal value of the weight by grid search. We find empirically from our numerical results that the performance of gPINN is sensitive to the weight value in some PDEs (e.g., the Poisson equation in Section 3.2.1), while it is not sensitive for other PDEs (e.g., the diffusion–reaction equation in Section 3.2.2).

For example, for the Poisson's equation $\Delta u = f$ in 1D, the additional loss term is

$$\mathcal{L}_g = w_g\frac{1}{\left|\mathcal{T}_g\right|}\sum_{\mathbf{x}\in\mathcal{T}_g}\left|\frac{d^3u}{dx^3} - \frac{df}{dx}\right|^2.$$

For the Poisson's equation in 2D, there are two additional loss terms:

$$\mathcal{L}_{g_1} = w_{g_1} \frac{1}{|\mathcal{T}_{g_1}|} \sum_{\mathbf{x} \in \mathcal{T}_{g_1}} \left| \frac{\partial^3 u}{\partial x^3} + \frac{\partial^3 u}{\partial x \partial y^2} - \frac{\partial f}{\partial x} \right|^2,$$

$$\mathcal{L}_{g_2} = w_{g_2} \frac{1}{|\mathcal{T}_{g_2}|} \sum_{\mathbf{x} \in \mathcal{T}_{g_2}} \left| \frac{\partial^3 u}{\partial x^2 \partial y} + \frac{\partial^3 u}{\partial y^3} - \frac{\partial f}{\partial y} \right|^2.$$

As we will show in our numerical examples, by enforcing the gradient of the PDE residual, gPINN improves the accuracy of the predicted solutions for u and requires less training points. Moreover, gPINN improves the accuracy of the predicted solutions for $\frac{\partial u}{\partial x_i}$. One motivation of gPINN is that the PDE residual of PINNs usually fluctuates around zero, and penalizing the slope of the residual would reduce the fluctuation and make the residual closer to zero.

However, theoretical analysis of gPINN and the benefits of incorporating gradient information requires extensive work, which is beyond the scope of this study. Here we provide some justification from the works of input gradient regularization [34] for classification problems. Although gPINN and input gradient regularization are developed in different setups for different objectives, they both use the gradient of the original loss with respect to the network input as an additional loss term. It has been observed empirically that the input gradient regularization improves the network generalization [34]. It has also been demonstrated both empirically [35–37] and theoretically [37] that input gradient regularization improves the network robustness against adversarial samples. The benefits of generalization and robustness may also apply to gPINN. For PINNs, using more training points leads to a smaller generalization error and thus a better accuracy [40]. Compared to PINNs, better generalization of gPINN means that gPINN would be more generalizable to the regions without training points and thus have a better accuracy by using less training points. While there has not been theoretical study of the robustness of PINNs, intuitively if gPINN has better robustness, then for inverse problems, it would tolerate larger noise of the data measurements, especially for a small number of training points.

2.3. Formulation of gPINN with residual-based adaptive refinement (RAR)

The residual points $\mathcal{T}_f$ of PINNs are usually randomly distributed in the domain, and in Ref. [4] a residual-based adaptive refinement (RAR) method is developed to improve the distribution of residual points during the training process. In RAR, we adaptively add more residual points in the locations where the PDE residual is large during the network training. Here we combine RAR and gPINN to further improve the accuracy and training efficiency (Algorithm 1).

Algorithm 1: gPINN with RAR.

Step 1 Train the neural network using gPINN on the training set $\mathcal{T}$ for a certain number of iterations.

Step 2 Compute the PDE residual $\left| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right|$ at random points in the domain.

Step 3 Add m new points to the training set $\mathcal{T}$ where the residual is the largest.

Step 4 Repeat Steps 1, 2, and 3 for n times, or until the mean residual falls below a threshold $\mathcal{E}$.

3. Results

We apply our proposed gPINNs and gPINNs with RAR to solve several forward and inverse PDE problems. In all examples, we use the tanh as the activation function, and the other hyperparameters for each example are listed in Table 1. All the codes in this study are implemented by using the library DeepXDE [4] with TensorFlow 1 backend and are available in GitHub at https://github.com/lu-group/gpinn.

Table 1
Hyperparameters for the 10 problems tested in this study. The Brinkman–Forchheimer problem in Section 3.3.1 has 3 sub-problems.

Section number and problem	Depth	Width	Optimizer	Learning rate	# Iterations
3.1 Function approximation	4	20	Adam	0.001	10 000
3.2.1 Poisson	4	20	Adam	0.001	20 000
3.2.2 Diffusion–reaction (forward)	4	20	Adam	0.0001	100 000
3.2.3 Poisson 2D	4	20	Adam	0.001	20 000
3.3.1 Brinkman–Forchheimer	4	20	Adam	0.001	50 000
3.3.2 Diffusion–reaction (inverse)	4	20	Adam	0.0001	200 000
3.4.1 Burgers'	4	32	Adam+L-BFGS	0.001	20 000
3.4.2 Allen–Cahn	5	64	Adam+L-BFGS	0.001	20 000

3.1. Function approximation via a gradient-enhanced neural network (gNN)

We first use a pedagogical example of function approximation to demonstrate the effectiveness of adding gradient information. We consider the following function

$$u(x) = -(1.4 - 3x)\sin(18x), \quad x \in [0, 1],$$

from the training dataset $\{(x_1, u(x_1)), (x_2, u(x_2)), \ldots, (x_n, u(x_n))\}$, where $(x_1, x_2, \ldots, x_n)$ are equispaced points in $[0, 1]$. The standard loss function to train a NN is

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} |u(x_i) - \hat{u}(x_i)|^2,$$

and we also consider the following gradient-enhanced NN with the extra loss function of the gradient as

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} |u(x_i) - \hat{u}(x_i)|^2 + w_g \frac{1}{n}\sum_{i=1}^{n} |\nabla u(x_i) - \nabla \hat{u}(x_i)|^2.$$

We performed the network training using different values of the weight w_g, including 1, 0.1, and 0.01, and found that the accuracy of gNN is insensitive to the value of w_g. Hence, here we will only show the results of $w_g = 1$.

When we use more training points, both NN and gNN have smaller L^2 relative error of the prediction of u, and gNN performs significantly better than NN with about one order of magnitude smaller error (Fig. 1A). In addition, gNN is more accurate than NN for the prediction of the derivative $\frac{du}{dx}$ (Fig. 1B). As an example, the prediction of u and $\frac{du}{dx}$ from NN and gNN using 15 training data points is shown in Figs. 1C and D, respectively. The standard NN has more than 10% error for u and $\frac{du}{dx}$, while gNN reaches about 1% error.

3.2. Forward PDE problems

After demonstrating the effectiveness of adding the gradient loss on the function approximation, we apply gPINN to solve PDEs.

3.2.1. Poisson equation in 1D

We first consider a 1D Poisson equation as

$$-\Delta u = \sum_{i=1}^{4} i \sin(ix) + 8\sin(8x), \quad x \in [0, \pi],$$

with the Dirichlet boundary conditions $u(x = 0) = 0$ and $u(x = \pi) = \pi$. The analytic solution is

$$u(x) = x + \sum_{i=1}^{4} \frac{\sin(ix)}{i} + \frac{\sin(8x)}{8}.$$

Instead of using a loss function $\mathcal{L}_b$ for the Dirichlet boundary conditions, we enforce it by choosing the surrogate of the solution as

$$\hat{u}(x) = x(\pi - x)\mathcal{N}(x) + x, \tag{5}$$

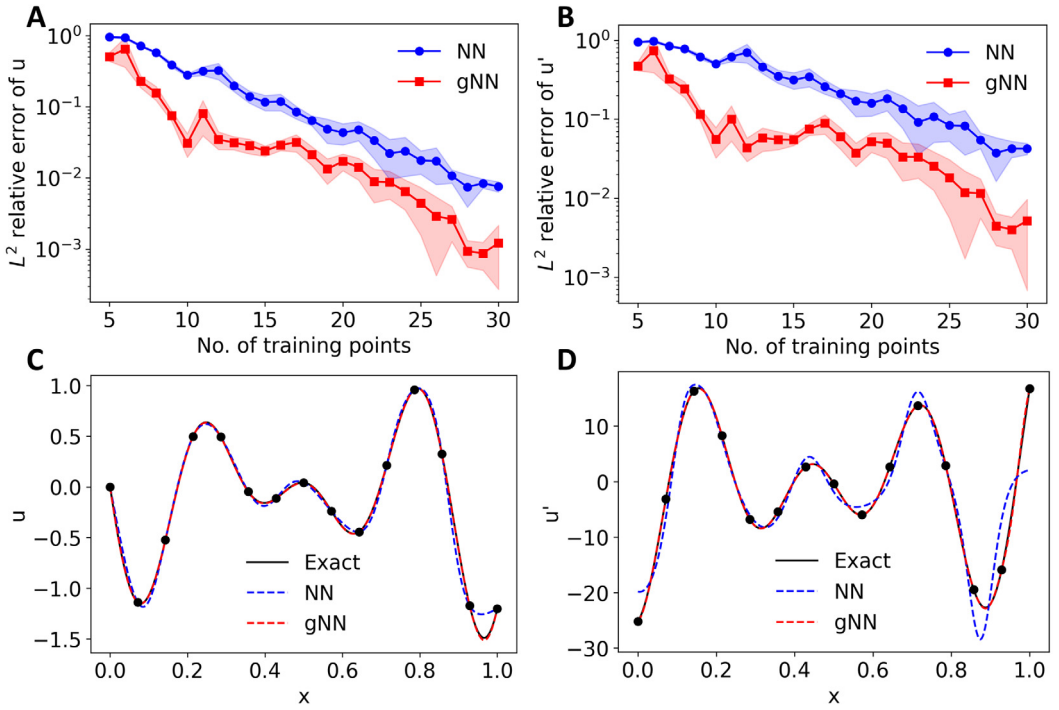

Fig. 1. Example in Section 3.1: Comparison between NN and gNN. (**A** and **B**) L^2 relative error of NN and gNN for (A) u and (B) $\frac{du}{dx}$ using different number of training points. The line and shaded region represent the mean and one standard deviation of 10 independent runs. (**C** and **D**) Example of the predicted (C) u and (D) $\frac{du}{dx}$, respectively. The black dots show the locations of the 15 training data points.

where $\mathcal{N}(x)$ is a neural network. Hence, the loss function is

$$\mathcal{L} = \mathcal{L}_f + w\mathcal{L}_g,$$

where $\mathcal{L}_f$ and $\mathcal{L}_g$ are defined in Eqs. (2) and (4), respectively.

When increasing the number of residual points from 10 to 20, as the baseline, the L^2 relative error of PINN for u decreases from 26% to 0.48% (Fig. 2A). The performance of gPINN depends on the choice of the weight w. For $w = 0.01$, gPINN thoroughly outperforms PINN in terms of the L^2 relative error of u (Fig. 2A), L^2 relative error of $\frac{du}{dx}$ (Fig. 2B), and the mean absolute value of the PDE residual (Fig. 2C). When using 20 residual points, the L^2 relative error of gPINN for u is about one order of magnitude smaller than PINN (Fig. 2A). An even greater improvement by gPINN can be seen in the L^2 relative error of $\frac{du}{dx}$ (about two orders of magnitude, Fig. 2B). gPINN outperforms PINN, because gPINN utilizes the information of the gradient and thus has a much faster convergence rate than PINN. The results of PINN and gPINN for the example of using 15 residual points are shown in Figs. 2D and E.

However, we note that if the value of w is not chosen properly, gPINN may not perform well. For example, when we choose $w = 1$, the accuracy of gPINN is even worse than PINN (Figs. 2A, B and C). We systematically investigated the performance of gPINN for different values of w when using 20 residual points (Figs. 2F and G). When w is small and close to 0, then gPINN becomes a standard PINN (the black horizontal line in Figs. 2F and G). When w is very large, the error of PINN increases. There exists an optimal weight at around $w = 0.01$. When w is smaller than 1, gPINN always outperforms PINN.

In the results above, we enforce the boundary condition exactly as a hard constraint through the network architecture of Eq. (5). Here we also compare PINN and gPINN for soft constraints of boundary conditions by choosing $\hat{u}(x) = \mathcal{N}(x)$ and using the additional loss term of Eq. (3). Compared to hard constraints, by using soft constraints, the errors of PINN for both u and $\frac{du}{dx}$ increase (Figs. 2H and I), although the PDE residual is similar (Fig. 2J). However, the errors of gPINN with hard or soft constraints are almost the same (Figs. 2H, I and J). Therefore, when using soft constraints, gPINN is even more accurate than PINN.

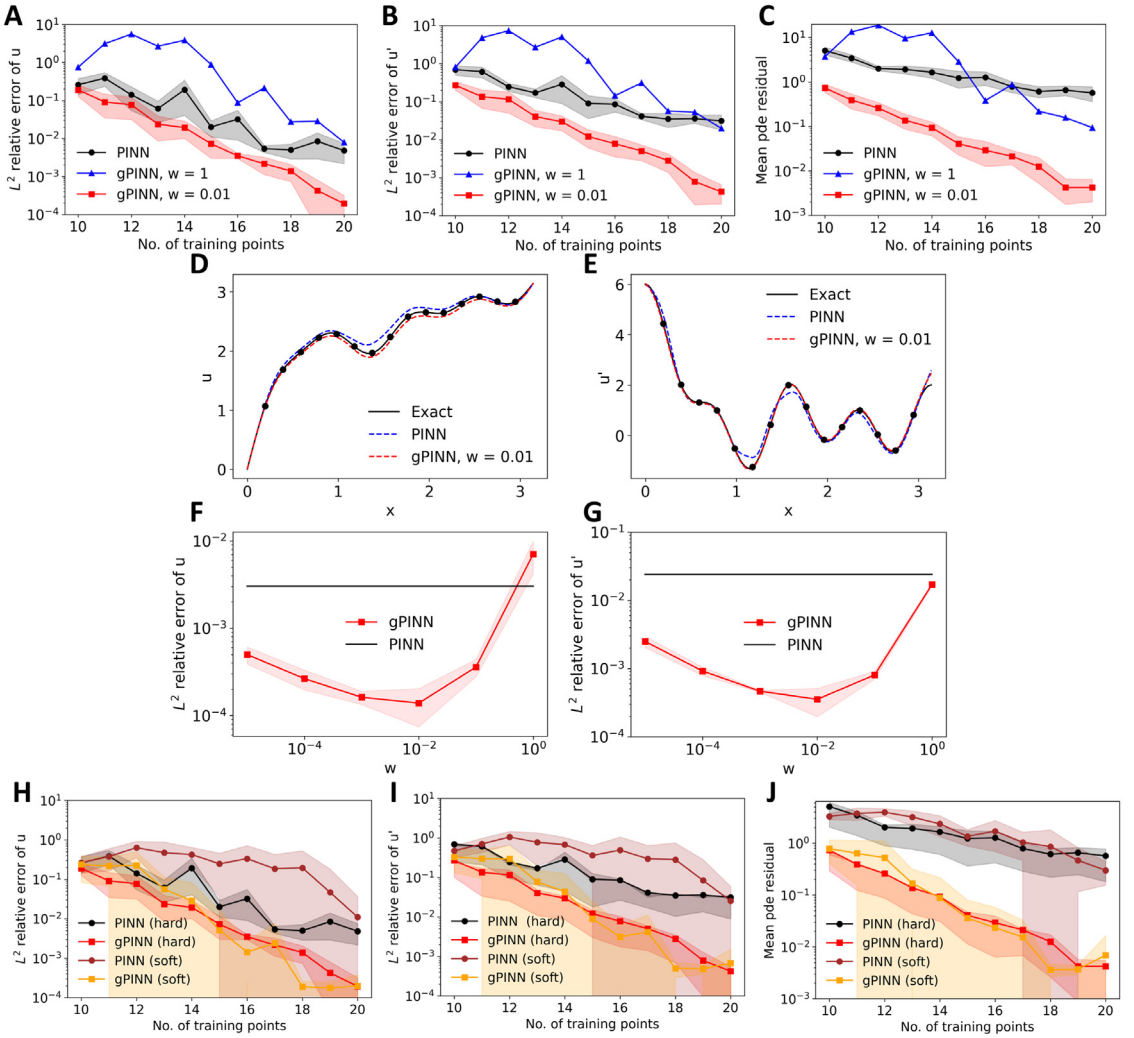

Fig. 2. Example in Section 3.2.1: Comparisons between PINN and gPINNs with the loss weight $w = 1$ and 0.01 (**A**) L^2 relative error of u. (**B**) L^2 relative error of u'. (**C**) The mean value of the PDE residual after training. (**D** and **E**) Example of the predicted u and u', respectively, when using 15 training points. The black dots show the locations of the residual points for training. (**F** and **G**) L^2 relative errors of gPINN for u and u' with different values of the weight w when using 20 training points. The shaded regions represent the one standard deviation of 10 random runs. (**H**, **I**, **J**) Comparison between hard and soft constraints for PINN and gPINN. (**H**) L^2 relative error of u. (**I**) L^2 relative error of u'. (**J**) The mean value of the PDE residual after training.

3.2.2. *Diffusion–reaction equation*

Next we consider a time-dependent PDE of a diffusion–reaction system described as

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + R(x,t), \quad x \in [-\pi, \pi], \ t \in [0,1],$$

where u is the solute concentration, $D = 1$ represents the diffusion coefficient, and R is the chemical reaction as

$$R(x,t) = e^{-t}\left[\frac{3}{2}\sin(2x) + \frac{8}{3}\sin(3x) + \frac{15}{4}\sin(4x) + \frac{63}{8}\sin(8x)\right].$$

The initial and boundary conditions are as follows:

$$u(x,0) = \sum_{i=1}^{4}\frac{\sin(ix)}{i} + \frac{\sin(8x)}{8},$$

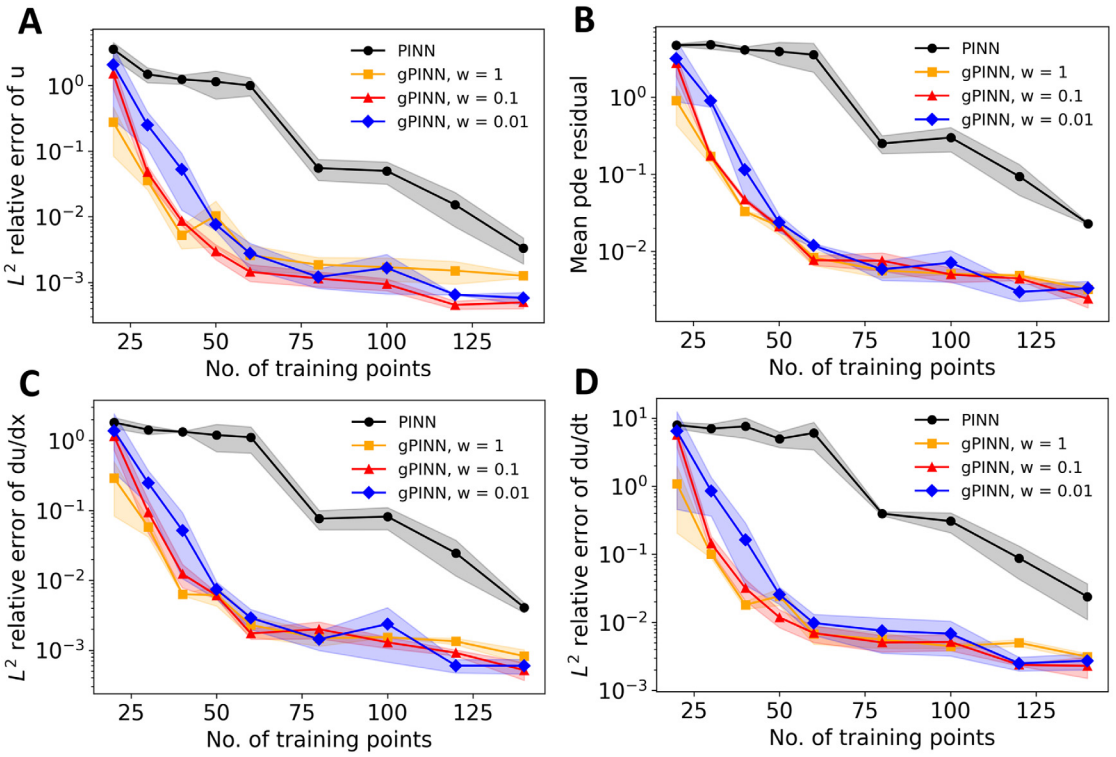

Fig. 3. Example in Section 3.2.2: Comparison between PINN and gPINN. (**A**) L^2 relative error of u for PINN and gPINN with $w = 1$, 0.1, and 0.01. (**B**) Mean absolute value of the PDE residual. (**C**) L^2 relative error of $\frac{du}{dx}$. (**D**) L^2 relative error of $\frac{du}{dt}$.

$$u(-\pi, t) = u(\pi, t) = 0,$$

which yields the analytic solution for u as

$$u(x, t) = e^{-t}\left[\sum_{i=1}^{4} \frac{\sin(ix)}{i} + \frac{\sin(8x)}{8}\right]. \tag{6}$$

Similar as the previous example, we also choose a proper surrogate of the solution to satisfy the initial and boundary conditions automatically:

$$\hat{u}(x) = (x^2 - \pi^2)(1 - e^{-t})\mathcal{N}(x) + u(x, 0),$$

where $\mathcal{N}(x)$ is a neural network. Here, we have two loss terms of the gradient, and the total loss function is

$$\mathcal{L} = \mathcal{L}_f + w\mathcal{L}_{g_x} + w\mathcal{L}_{g_t},$$

where $\mathcal{L}_{g_x}$ and $\mathcal{L}_{g_t}$ are the derivative losses with respect to x and t, respectively.

In the Poisson equation, the performance of gPINN depends on the value of the weight, but in this diffusion–reaction system, gPINN is not sensitive to the value of w. gPINN with the values of $w = 0.01$, 0.1, and 1 all outperform PINN by up to two orders of magnitude for the L^2 relative errors of u, $\frac{du}{dx}$ and $\frac{du}{dt}$, and the mean absolute error of the PDE residual (Fig. 3). gPINN reaches 1% L^2 relative error of u by using only 40 training points, while PINN requires more than 100 points to reach the same accuracy.

As an example, we show the exact solution, the predictions, and the error of PINN and gPINN with $w = 0.01$ in Fig. 4 when the number of the residual points is 50. The PINN prediction has a large error of about 100%. However, the gPINN prediction's largest absolute error is around 0.007 and the L^2 relative error of 0.2%.

We note that gPINN appears to plateau at around 130 training points (Fig. 3), which is due to network optimization. We show that a better accuracy is achieved using a smaller learning rate of 10^{-6} and more iterations

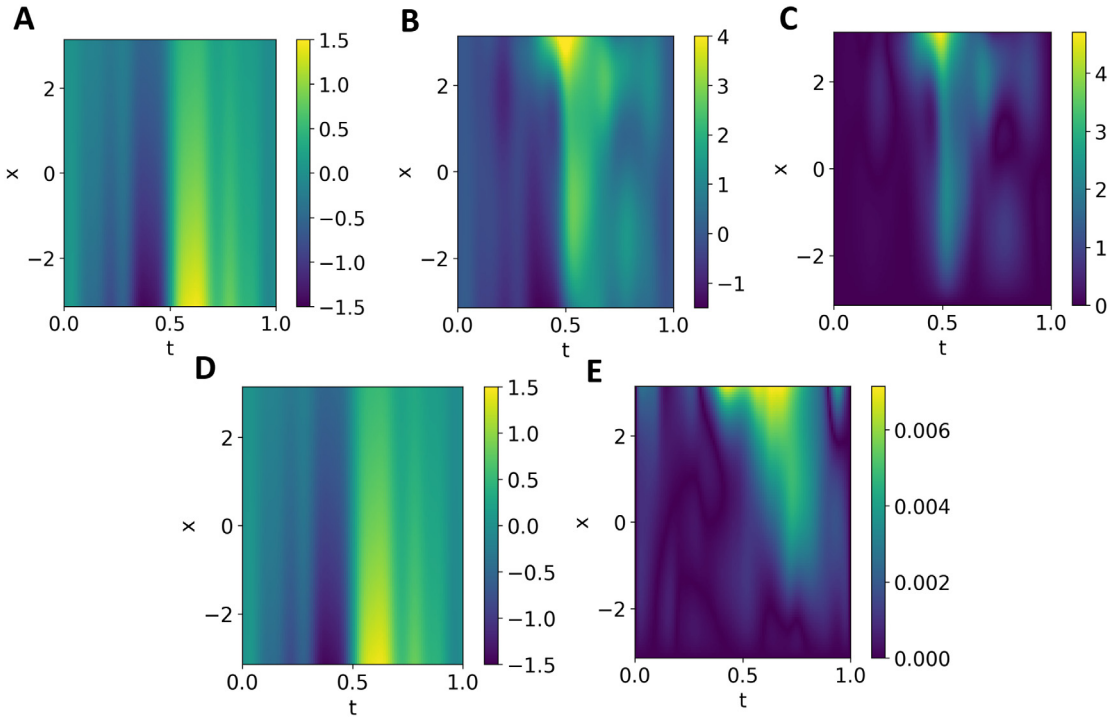

Fig. 4. Example in Section 3.2.2: Comparison between PINN and gPINN using 50 residual points for training. (**A**) The exact solution in Eq. (6). (**B** and **C**) The (B) prediction and (C) absolute error of PINN. (**D** and **E**) The (D) prediction and (E) absolute error of gPINN with $w = 0.1$.

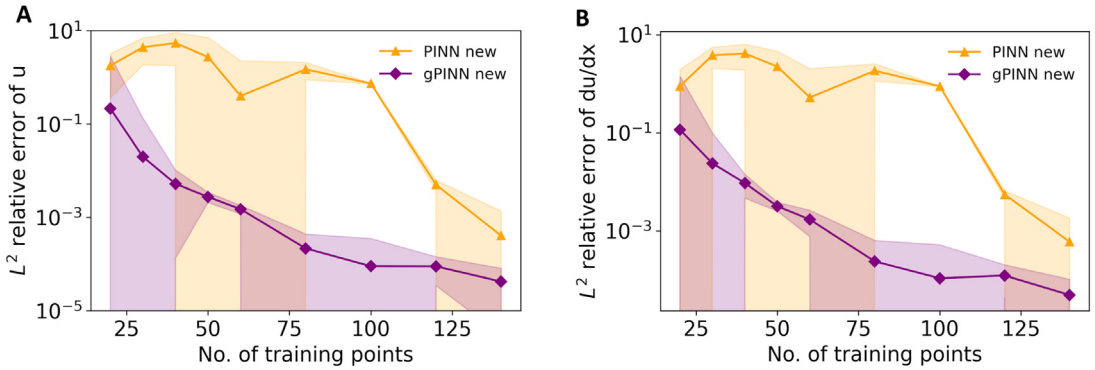

Fig. 5. Example in Section 3.2.2: PINN and gPINN are trained with a smaller learning rate and more iterations. L^2 relative error of (**A**) u and (**B**) $\frac{du}{dx}$ for PINN and gPINN with $w = 0.1$.

(5×10^6) in Fig. 5. The L^2 relative error of gPINN for u and $\frac{du}{dx}$ decreases to less than 0.01% using 140 training points and does not saturate. In this case, the accuracy of PINN is also worse than gPINN, especially when the number of training points is small.

3.2.3. *Poisson equation in 2D*

We also consider a 2D Poisson problem

$$-\Delta u = f, \quad (x, y) \in [0, 1]^2,$$

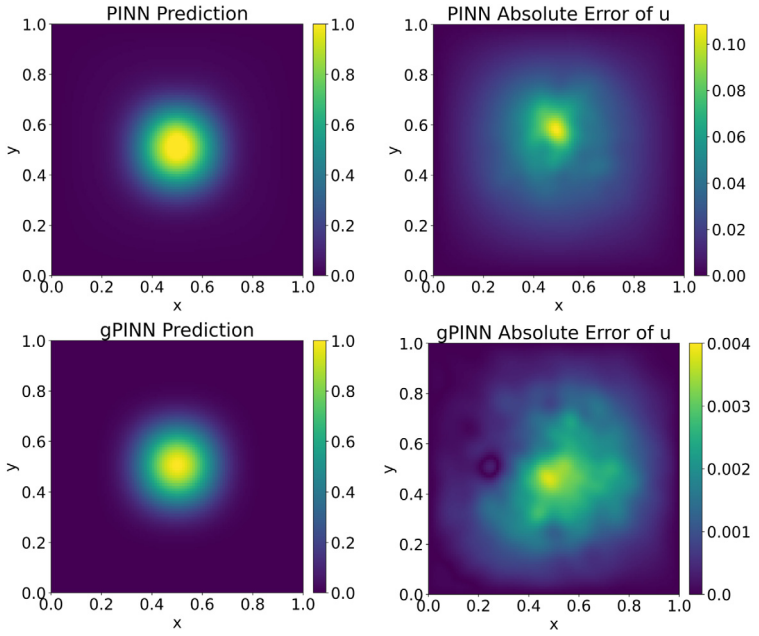

Fig. 6. Example in Section 3.2.3: Comparison between PINN and gPINN for a 2D Poisson problem.

with zero Dirichlet boundary conditions. We choose f according to the following exact solution:

$$u(x, y) = 2^{4a} x^a (1-x)^a y^a (1-y)^a$$

with $a = 10$.

We use hard boundary constraints for both PINN and gPINN, and train PINN and gPINN with 400 residual points. The weight of the additional loss terms in gPINN is chosen as 10^{-5}. The solutions and errors of PINN and gPINN are shown in Fig. 6. The solution of PINN is accurate in most part of the domain. However, because the solution has a peak in the center of the domain, PINN has a large error near the center. By contrast, gPINN is very accurate in the entire domain.

3.3. Inverse problems of PDEs

In addition to solving forward PDE problems, we also apply gPINN for solving inverse PDE problems.

3.3.1. Inferring the effective viscosity and permeability for the Brinkman–Forchheimer model

The Brinkman–Forchheimer model can be viewed as an extended Darcy's law and is used to describe wall-bounded porous media flows:

$$-\frac{\nu_e}{\epsilon}\nabla^2 u + \frac{\nu u}{K} = g, \quad x \in [0, H],$$

where the solution u is the fluid velocity, g denotes the external force, ν is the kinetic viscosity of fluid, ϵ is the porosity of the porous medium, and K is the permeability. The effective viscosity, ν_e, is related to the pore structure and hardly to be determined. A no-slip boundary condition is imposed, i.e., $u(0) = u(1) = 0$. The analytic solution for this problem is

$$u(x) = \frac{gK}{\nu}\left[1 - \frac{\cosh\left(r\left(x - \frac{H}{2}\right)\right)}{\cosh\left(\frac{rH}{2}\right)}\right]$$

with $r = \sqrt{\nu\epsilon/\nu_e K}$. We choose $H = 1$, $\nu_e = \nu = 10^{-3}$, $\epsilon = 0.4$, and $K = 10^{-3}$, and $g = 1$. To infer ν_e, we collect the data measurements of the velocity u in only 5 sensor locations.

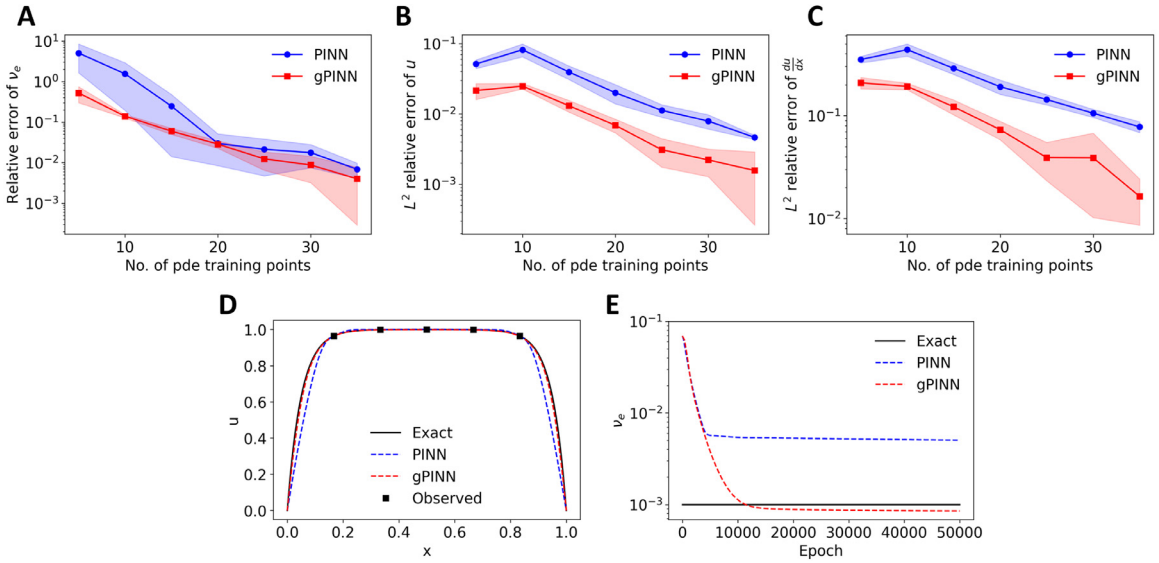

Fig. 7. Example in Section 3.3.1: Inferring ν_e by using PINN and gPINN from 5 measurements of u. (**A**) Relative error of ν_e. (**B**) L^2 relative error of u. (**C**) L^2 relative error of $\frac{du}{dx}$. (**D**) Example of the predicted u using 5 observations of u and 10 PDE residual points. The black squares in D show the observed locations. (**E**) The convergence of the predicted value for ν_e throughout training.

In PINN and gPINN, we simultaneously optimize the network and the unknown value of ν_e. The loss weight in gPINN is chosen as $w = 0.1$. Similar as what we observed in the forward PDE problems, gPINN outperforms PINN in this case (Fig. 7). Specifically, the error of the predictions of gPINN for u and $\frac{du}{dx}$ is about one order of magnitude smaller than PINN (Figs. 7B and C). Also, the inferred ν_e from gPINN is more accurate than that from PINN (Fig. 7A). We also show the example using only 10 PDE residual points. While PINN failed to predict u near the boundary with a steep gradient, gPINN can still have a good accuracy (Fig. 7D). During the training, the predicted ν_e in PINN did not converge to the true value, while in gPINN the predicted ν_e is more accurate (Fig. 7E).

To further test the performance of gPINN, we next infer both ν_e and K still from 5 measurements of u. Similarly, the PINN solution of u struggles with the regions near the boundary, but gPINN can still achieve a good accuracy (Fig. 8A). Both PINN and gPINN converge to an accurate value of K (Fig. 8C), but gPINN converges to much more accurate value for ν_e than PINN (Fig. 8B).

Next, we add Gaussian noise (mean 0 and standard deviation 0.05) to the observed values and infer both ν_e and K using 12 measurements of u (Fig. 9). Both PINN and gPINN converge to an accurate value for K. Whereas PINN struggles with the added noise and struggles to learn u and ν_e, gPINN performs very well in both. However, after doubling the number of PDE training points from 15 to 30 ("PINN 2x" in Fig. 9), PINN can also perform well at inferring ν_e, though the performance is still slightly worse than that of gPINN.

3.3.2. Inferring the space-dependent reaction rate in a diffusion–reaction system

We consider a one-dimensional diffusion–reaction system in which the reaction rate $k(x)$ is a space-dependent function:

$$\lambda \frac{\partial^2 u}{\partial x^2} - k(x)u = f, \quad x \in [0, 1],$$

where $\lambda = 0.01$ is the diffusion coefficient, u is the solute concentration, and $f = \sin(2\pi x)$ is the source term. The objective is to infer $k(x)$ given measurements on u. The exact unknown reaction rate is

$$k(x) = 0.1 + \exp\left[-0.5\frac{(x - 0.5)^2}{0.15^2}\right].$$

In addition, the condition $u(x) = 0$ is imposed at $x = 0$ and 1.

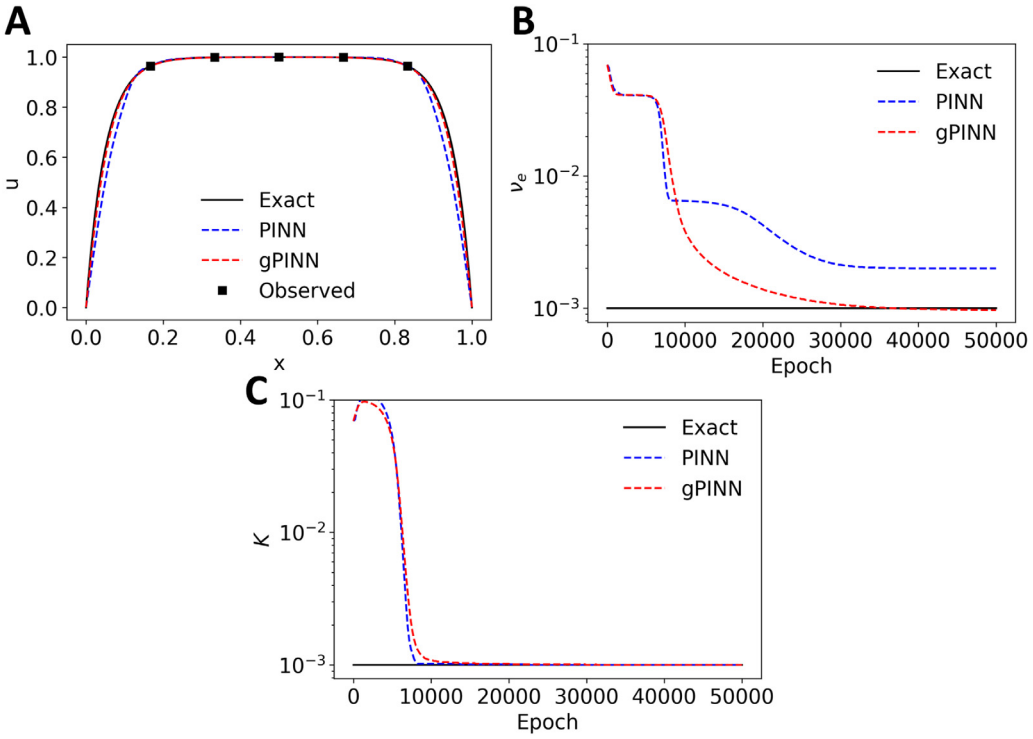

Fig. 8. Example in Section 3.3.1: Inferring both ν_e and K. (**A**) The predicted u from PINN and gPINN. (**B**) The convergence of the predicted value for ν_e throughout training. (**C**) The convergence of the predicted value for K. The black squares in A show the observed locations of u.

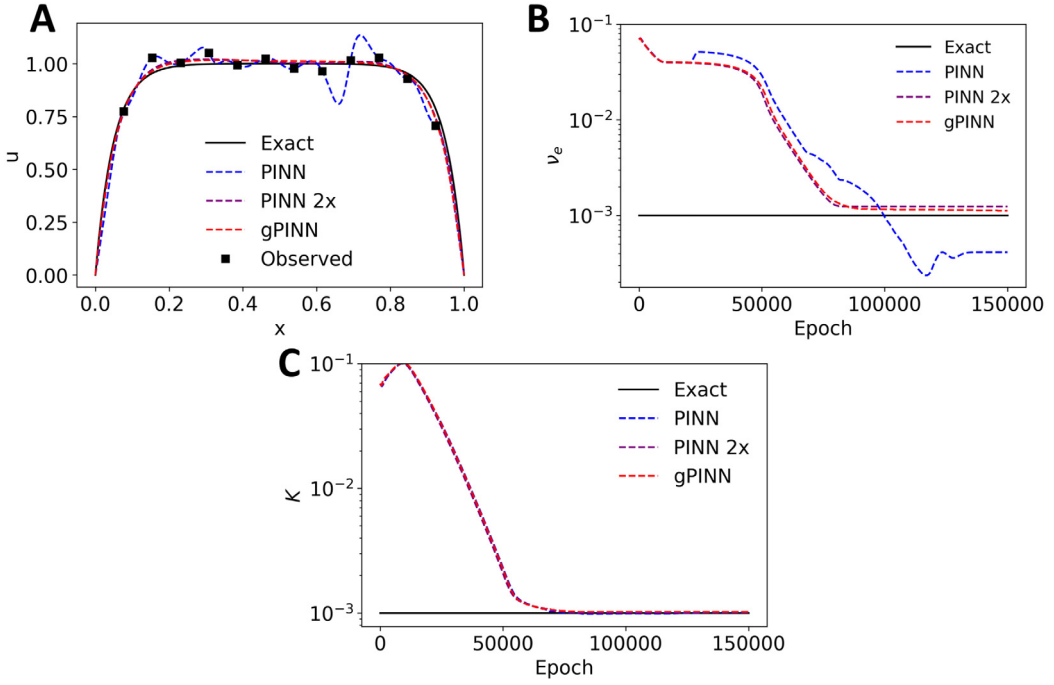

Fig. 9. Example in Section 3.3.1: Inferring both ν_e and K from noise data. (**A**) The predicted u from PINN and gPINN. "PINN 2x" is PINN with twice more PDE training points. The black squares show the observed measurements of u. (**B**) The convergence of the predicted value for ν_e throughout training. (**C**) The convergence of the predicted value for K.

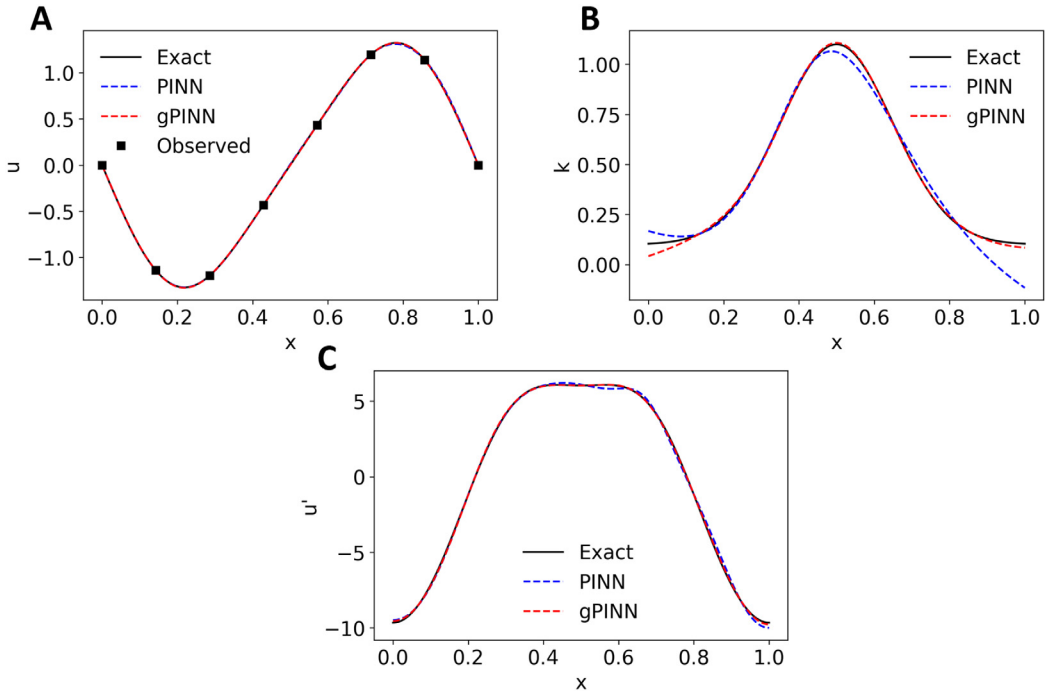

Fig. 10. Example in Section 3.3.2: Comparison between PINN and gPINN. (**A**) The prediction of u. (**B**) The prediction of k. (**C**) The prediction of $\frac{du}{dx}$. We used 8 observations of u (the black squares in A) and 10 residual points for training.

As the unknown parameter k is a function of x instead of just one constant, in addition to the network of u, we use another network to approximate k. We choose the weight $w = 0.01$ in gPINN. We test the performance of PINN and gPINN by using 8 observations of u and 10 PDE residual points for training. Both PINN and gPINN perform well in learning the solution u, though the PINN solution slightly deviates from the exact solution around $x = 0.8$ (Fig. 10A). However, for the inferred function k, gPINN's prediction was much more accurate than PINN (Fig. 10B). Also, the prediction of $\frac{du}{dx}$ by gPINN is more accurate than the prediction of PINN.

3.4. gPINN enhanced by RAR

To further improve the accuracy and training efficiency of gPINN for solving PDEs with a stiff solution, we apply RAR to adaptively improve the distribution of residual points during the training process.

3.4.1. Burgers' Equation

We consider the 1D Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1, 1], t \in [0, 1],$$

with the initial and boundary conditions

$$u(x, 0) = -\sin(\pi x), \quad u(-1, t) = u(1, t) = 0,$$

with $\nu = 0.01/\pi$.

We first test PINN and gPINN for this problem. PINN converges very slowly and has a large L^2 relative error, while gPINN achieves one order of magnitude smaller error ($\sim$0.2%) (the blue and red lines in Fig. 11), as we expected.

The solution to this 1D Burgers' equation is very steep near $x = 0$, so intuitively there should be more residual points around that region. We first show the effectiveness of PINN with RAR proposed in Ref. [4]. For RAR, we

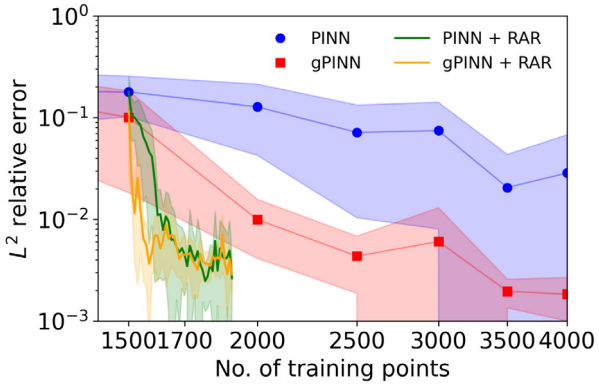

Fig. 11. Example in Section 3.4.1: L^2 relative errors of PINN, gPINN, PINN with RAR, and gPINN with RAR. For RAR, we started from 1500 uniformly-distributed residual points and added 400 extra points. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

first train the network using 1500 uniformly-distributed residual points and then gradually add 400 more residual points during training. We added 10 new points at a time, i.e., $m = 10$ in Algorithm 1. For the Burgers' equation, the solution has a steep gradient around $x = 0$, and after the initial training of 1500 residual points, the region around $x = 0$ has the largest error of u and the PDE residual (Fig. 12). RAR automatically added new points near the largest error, as shown in the left column in Fig. A.17, and then the errors of u and the PDE residual consistently decrease as more points are added. By using RAR, the error of PINN decreases very fast, and PINN achieves the L^2 relative error of $\sim$0.3% by using only 1900 residual points for training (the green line in Fig. 11).

We also used gPINN together with RAR. gPINN with RAR also added new points near $x = 0$, and the errors of u and the PDE residual consistently decrease when more points are added adaptively (Fig. 12), similarly to PINN with RAR. The final accuracy of PINN with RAR and gPINN with RAR is similar, but the error of gPINN with RAR drops much faster than PINN with RAR when using only 100 extra training points. Therefore, by pairing gPINN together with the RAR, we can achieve the best performance.

3.4.2. Allen–Cahn equation

We also consider the following Allen–Cahn equation:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + 5(u - u^3), \quad x \in [-1, 1], \ t \in [0, 1],$$

with the initial and boundary conditions:

$$u(x, 0) = x^2 \cos(\pi x),$$
$$u(-1, t) = u(1, t) = -1,$$

where $D = 0.001$. The solution to this Allen–Cahn equation has multiple very steep regions similar to that of the Burgers' equation.

First, comparing PINN and gPINN, we can once again observe that gPINN has better accuracy than PINN (the blue and red lines in Fig. 13). gPINN requires around 2000 training points to reach 1% error, while PINN requires around 4000 training points to reach that same accuracy.

We next show the behavior and effectiveness of PINN with RAR and gPINN with RAR again. We first train the network using 500 uniformly-distributed residual points and then gradually add 3000 more residual points during training with 30 training points added at a time. The solution u has two peaks around $x = -0.5$ and $x = 0.5$, where the largest error occurs (Fig. 14B). The added points by RAR also fall on these two regions of high error, as shown in Fig. 14G. The error becomes nearly uniform with 1200 added points (Fig. 14H), and then the added points start to become more and more uniform (Figs. 14J and M). By using RAR, the error of gPINN decreases drastically fast, and by adding only 200 additional points (i.e., 700 in total), gPINN reaches 1% error. However, gPINN with RAR begins to plateau at about 1500 training points with approximately 0.1% error, which could be resolved by using a smaller learning rate as we show in Section 3.2.2.

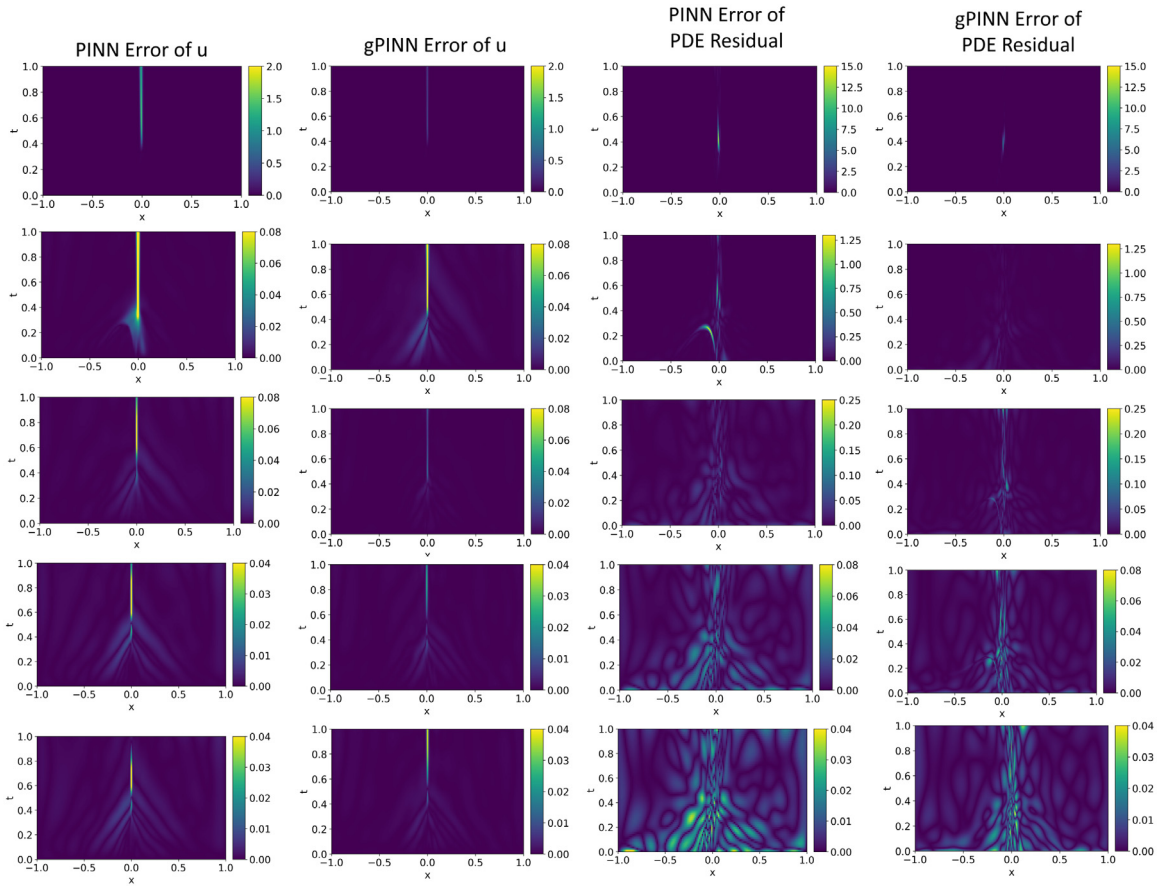

Fig. 12. Example in Section 3.4.1: Comparison between PINN with RAR and gPINN with RAR. (First column) The absolute error of PINN for u. (Second column) The absolute error of gPINN for u. (Third column) The absolute error of PINN for the PDE residual. (Fourth column) The absolute error of gPINN for the PDE residual. (First row) No extra points have been added. (Second row) 100 extra points have been added. (Third row) 200 extra points have been added. (Fourth row) 300 extra points have been added. (Fifth row) 400 extra points have been added.

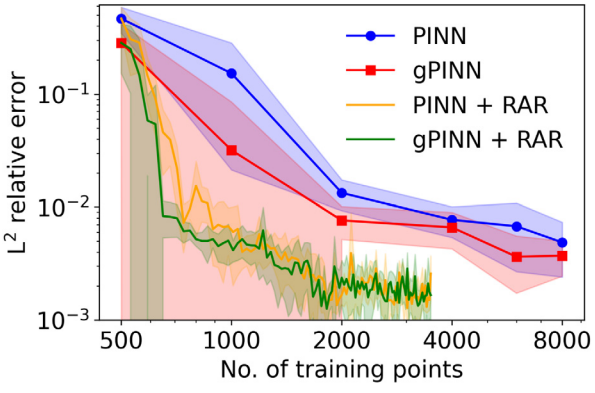

Fig. 13. Example in Section 3.4.2: L^2 relative errors of PINN, gPINN, PINN with RAR, and gPINN with RAR. For RAR, there were 500 initial points and 3000 added points. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

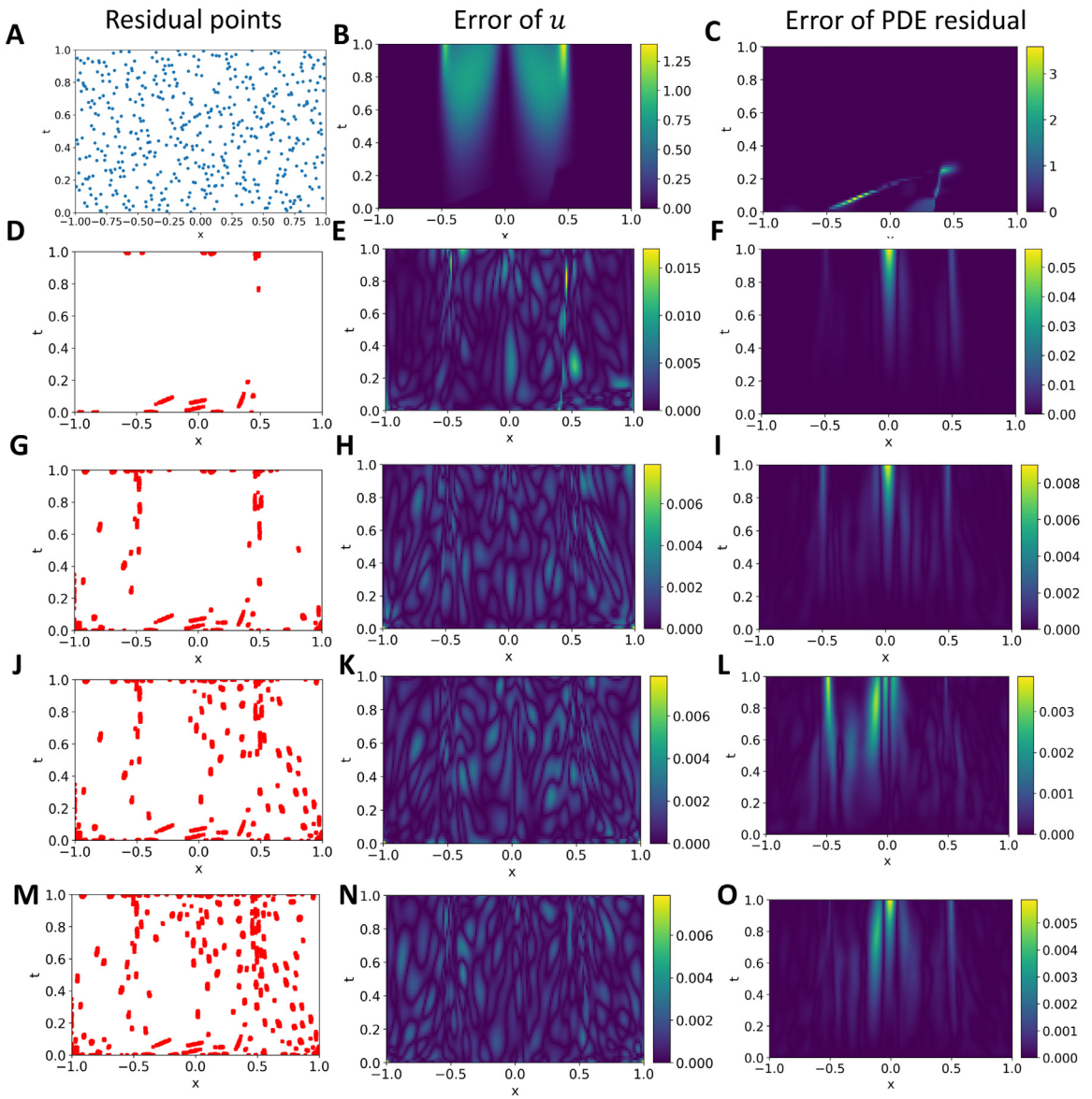

Fig. 14. Example in Section 3.4.2: gPINN with RAR. (**A**, **B**, **C**) No extra points have been added. (A) The initial distribution of the 500 residual points. (B) The absolute error of u. (C) The absolute error of the PDE residual. (**D**, **E**, **F**) 300 extra points (point locations shown in D) have been added. (**G**, **H**, **I**) 1200 extra points (point locations shown in G) have been added. (**J**, **K**, **L**) 2100 extra points (point locations shown in J) have been added. (**M**, **N**, **O**) 3000 extra points (point locations shown in M) have been added.

4. Discussion

We have showed that gPINN achieves improved accuracy over PINN using the same number of training points for forward problems and learns the unknown parameters more accurately in inverse problems. Here we discuss the computational cost of gPINN and a further extension by using higher-order derivatives of the PDE residual.

4.1. Computational cost of gPINN

When using the same number of residual points (each loss term is evaluated with the entire training set), gPINN achieves better accuracy than PINN, but the computational cost of gPINN is higher than PINN because of the

Table 2
Relative computational cost of gPINN to PINN for the Burgers' equation in Section 3.4.1. The number of training points for both PINN and gPINN varies from 1500 to 4000.

No. of training points	1500	2000	2500	3000	3500	4000	Average
Relative cost	1.85	2.01	1.91	1.99	1.92	1.95	1.94

Table 3
Relative computational cost of gPINN to PINN for all the problems.

Problem	Relative cost
3.1 Function approximation	1.43
3.2.1 Poisson	1.69
3.2.2 Diffusion–reaction (forward)	1.74
3.3.1 Brinkman–Forchheimer	1.83
3.3.2 Diffusion–reaction (inverse)	1.45
3.4.1 Burgers'	1.94
3.4.2 Allen–Cahn	1.97

additional loss terms with higher order derivatives. To quantify the computational overhead of gPINN, we define the *relative computational cost* of gPINN to PINN as the training time of gPINN divided by the training time of PINN, which depends on the specific problem such as the highest derivative order and the complexity of the PDE.

For example, for the Burgers' equation in Section 3.4.1, the relative cost for different number of training points in one trial is shown in Table 2, ranging from 1.85 to 2.01. The average relative cost is ∼1.94, i.e., gPINN is about twice as computationally expensive as PINN. However, even if we take the relative cost into consideration, gPINN is still more accurate than PINN in some cases. For example, the cost of PINN with 4000 training points is twice the cost of PINN with 2000 points, and thus the cost of PINN with 4000 training points is comparable to the cost of gPINN with 2000 training points, but gPINN with 2000 training points is more accurate than PINN with 4000 training points, as shown in Fig. 11. All computations are performed using GPU on a workstation with an Intel Core i9-11900F CPU and a NVIDIA GeForce RTX 3090 GPU.

We list the relative computational cost of gPINN to PINN for all the problems in Table 3. The relative cost is between ∼1.5 to ∼2.0. We note that due to the computational overhead, gPINN is not always more computationally efficient than PINN. For example, for the Poisson equation (Figs. 2A and B), the relative cost is 1.69, and gPINN with 12 training points is less accurate than PINN with 20 training points.

We also note that in this work, we compute higher-order derivatives by applying automatic differentiation (AD) of first-order derivative recursively. However, this nested approach results in combinatorial amounts of redundant work and is not efficient [41,42], and other methods, e.g., Taylor-mode AD, can be used for better computational performance. As shown in [43], if we implement higher-order derivatives efficiently, we can reduce the computational cost by at least one order of magnitude, by using which the computational overhead of gPINN will be greatly reduced. A more efficient implementation of gPINN needs to be conducted in the future, but this heavily relies on the development of the deep learning libraries such as TensorFlow and PyTorch.

4.2. gPINN with higher-order derivatives of the PDE residual

We have used the first-order derivatives of the PDE residual as additional loss terms. As an extension, we may also consider higher-order derivatives. For example, in addition to the first-order derivatives, we can also add the second-order derivatives of the PDE residual

$$\left| \frac{\partial^2 f}{\partial x_i \partial x_j} \right|^2,$$

i.e., both first-order and second-order derivatives are in the loss. In this section, we show that higher-order derivatives have the following two effects:

- Accuracy (Section 4.2.1): Adding higher-order derivatives does not always improve accuracy.
- Computational cost (Section 4.2.2): The cost of gPINN increases exponentially with respect to the order of derivatives.

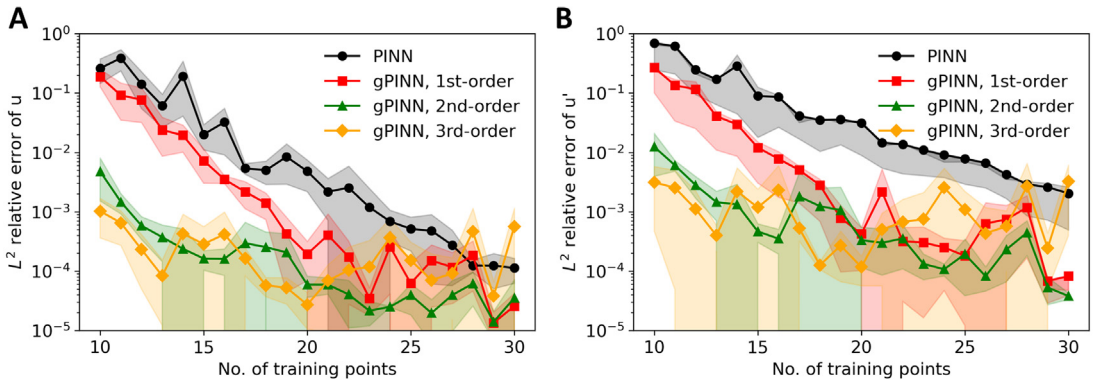

Fig. 15. Comparisons between gPINN with first-, second- and third-order derivatives for the Poisson equation in Section 3.2.1. (**A**) L^2 relative error of u. (**B**) L^2 relative error of u'.

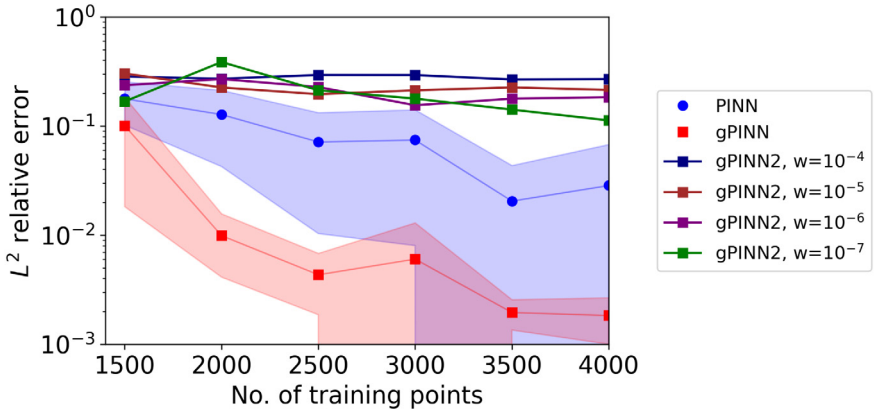

Fig. 16. L^2 relative error of PINN and gPINN with first-order and second-order derivatives for the Burgers' equation in Section 3.4.1. "gPINN2" denotes gPINN with the second-order derivatives. The standard deviations for gPINN2 are not shown, to keep the figure readable.

4.2.1. Effect of higher-order derivatives on accuracy

We first test gPINN with higher-order derivatives on the Poisson equation in Section 3.2.1. For the loss term of the first-order derivative, we use the same weight 0.01 as we used in Section 3.2.1. For the loss term of the second-order derivative, we tried different weights and obtained the best results by using the weight 10^{-4}. When the number of training points is small, gPINN with both first- and second-order derivatives is more accurate than gPINN with only the first-order derivative (Fig. 15).

We further add the loss term of the third-order derivative, i.e., gPINN with all the first-, second- and third-order derivatives, and the best result is obtained with the weight 10^{-6} for the loss term of third-order derivative. However, by adding the third-order derivative, there is no significant improvement compared to gPINN with second-order derivative (Fig. 15). In fact, adding the third-order derivative makes the error larger when the number of training points is more than 20. Therefore, for the Poisson equation, the highest order of derivative that we can use is the second order.

We also test the Burgers' equation in Section 3.4.1 by adding the second-order derivatives. Because the weight of the loss term of the first-order derivatives is 10^{-4}, we try different weights of the second-order derivatives from 10^{-4} to 10^{-7}. We find that by adding the second-order derivatives, gPINN does not converge at all (Fig. 16). The reason is that gPINN with second-order derivatives is very difficult to train — the training loss is only decreased by one order of magnitude.

From the results of the Poisson equation and the Burgers' equation, we find that adding higher-order derivatives does not always improve accuracy. One possible reason could be that the higher-order derivatives of the PDE residual may have bad regularity, which makes the network optimization more difficult.

Table 4
Relative computational cost of gPINN with different order of the highest derivative to PINN.

Highest derivative order	First	Second	Third
3.2.1 Poisson	1.69	3.14	6.93
3.4.1 Burgers'	1.94	4.56	14.90

4.2.2. Effect of higher-order derivatives on computational cost

We list the relative computational cost of gPINN with different order of the highest derivative for the two problems in Table 4. The computational cost increases significantly (exponentially) with respect to the order of derivatives. This exponential growth is consistent with the observation in [43], and thus using higher-order derivatives may not be efficient due to the computational overhead, although the overhead could be reduced by using an efficient implement of AD as we discussed in Section 4.1.

5. Conclusion

In this paper, we proposed a new version of physics-informed neural networks (PINNs) with gradient enhancement (gPINNs) for improved accuracy. We demonstrated the effectiveness of gPINN in both forward and inverse PDE problems, including Poisson equation, diffusion–reaction equation, Brinkman–Forchheimer model, Burgers' equation, and Allen-Cahn equation. Our numerical results from all of the examples show that gPINN clearly outperforms PINN with the same number of training points in terms of the L^2 relative errors of the solution and the derivatives of the solution. For the inverse problems, gPINN learned the unknown parameters more accurately than PINN. In addition, we combined gPINN with residual-based adaptive refinement (RAR) to further improve the performance. For the PDEs with solutions that had especially steep gradients, such as Burgers' equation and the Allen-Cahn equation, RAR allowed gPINN to perform well with much fewer residual points.

In our numerical results, we showed the improved accuracy of gPINN compared to PINN, and future theoretical work should be done to understand the benefits of incorporating gradient information. Compared to PINN, in gPINN we have an extra hyperparameter—the weight coefficient of the gradient loss. In some problems, the performance of gPINN is not sensitive to this weight, but in some cases, there exists an optimal weight to achieve the best accuracy, and thus we need to tune the weight. In this study, we determine the optimal value of the weight by grid search, and it is an interesting research topic in the future to automatically determine an optimal weight such as through the analysis of neural tangent kernel [17]. Moreover, it is possible to combine gPINN with other extensions of PINN to further improve the performance, such as extended PINN (XPINN) [19] and parareal PINN (PPINN) [18]. In this paper, we have not tested gPINN for high-dimensional problems, which will be a future work. We note that the number of the additional loss terms is the same as the dimension, and thus gPINN becomes more expensive for high-dimensional problems.

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.

Acknowledgments

This work was supported by the DOE PhILMs project, USA (no. DE-SC0019453) and OSD/AFOSR MURI, USA grant FA9550-20-1-0358. J.Y. and L.L. thank MIT's PRIMES-USA program.

Appendix A. RAR for Burgers' equation

See Fig. A.17.

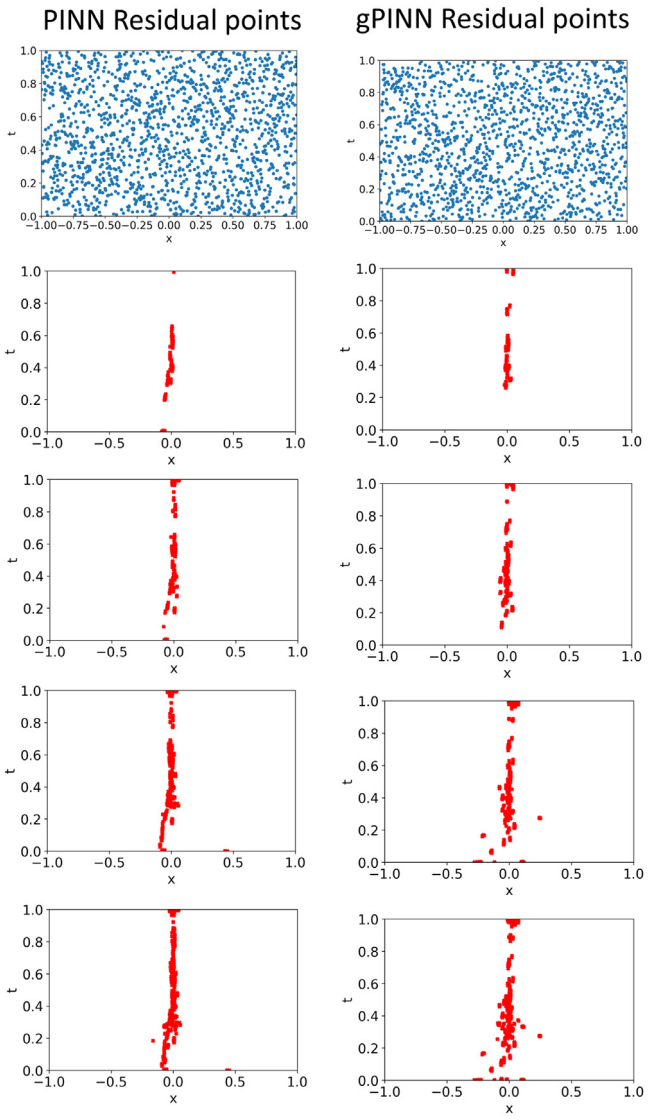

Fig. A.17. Locations of added points in PINN with RAR and gPINN with RAR for Burgers' equation. (Left) PINN. (Right) gPINN. (First row) The initial distribution of the 1500 residual points. No extra points have been added. (Second row) 100 extra points have been added. (Third row) 200 extra points have been added. (Fourth row) 300 extra points have been added. (Fifth row) 400 extra points have been added.

References

[1] G.E. Karniadakis, I.G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nat. Rev. Phys. 3 (6) (2021) 422–440.

[2] M. Dissanayake, N. Phan-Thien, Neural-network-based approximations for solving partial differential equations, Commun. Numer. Methods. Eng. 10 (3) (1994) 195–201.

[3] M. Raissi, P. Perdikaris, G.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.

[4] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, DeepXDE: A deep learning library for solving differential equations, SIAM Rev. 63 (1) (2021) 208–228.

[5] G. Pang, L. Lu, G.E. Karniadakis, fPINNs: Fractional physics-informed neural networks, SIAM J. Sci. Comput. 41 (4) (2019) A2603–A2626.

[6] D. Zhang, L. Lu, L. Guo, G.E. Karniadakis, Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems, J. Comput. Phys. 397 (2019) 108850.

[7] D. Zhang, L. Guo, G.E. Karniadakis, Learning in modal space: Solving time-dependent stochastic PDEs using physics-informed neural networks, SIAM J. Sci. Comput. 42 (2) (2020) A639–A665.
[8] Y. Chen, L. Lu, G.E. Karniadakis, L. Dal Negro, Physics-informed neural networks for inverse problems in nano-optics and metamaterials, Opt. Express 28 (8) (2020) 11618–11633.
[9] L. Lu, R. Pestourie, W. Yao, Z. Wang, F. Verdugo, S.G. Johnson, Physics-informed neural networks with hard constraints for inverse design, SIAM J. Sci. Comput. 43 (6) (2021) B1105–B1132.
[10] M. Raissi, A. Yazdani, G.E. Karniadakis, Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations, Science 367 (6481) (2020) 1026–1030.
[11] A. Yazdani, L. Lu, M. Raissi, G.E. Karniadakis, Systems biology informed deep learning for inferring parameters and hidden dynamics, PLoS Comput. Biol. 16 (11) (2020) e1007575.
[12] F. Sahli Costabal, Y. Yang, P. Perdikaris, D.E. Hurtado, E. Kuhl, Physics-informed neural networks for cardiac activation mapping, Front. Phys. 8 (2020) 42.
[13] M.A. Nabian, R.J. Gladstone, H. Meidani, Efficient training of physics-informed neural networks via importance sampling, Comput.-Aided Civ. Infrastruct. Eng. (2021).
[14] Y. Gu, H. Yang, C. Zhou, Selectnet: Self-paced learning for high-dimensional partial differential equations, 2020, arXiv preprint arXiv:2001.04860.
[15] L. McClenny, U. Braga-Neto, Self-adaptive physics-informed neural networks using a soft attention mechanism, 2020, arXiv preprint arXiv:2009.04544.
[16] S. Wang, Y. Teng, P. Perdikaris, Understanding and mitigating gradient pathologies in physics-informed neural networks, 2020, arXiv preprint arXiv:2001.04536.
[17] S. Wang, X. Yu, P. Perdikaris, When and why PINNs fail to train: A neural tangent kernel perspective, 2020, arXiv preprint arXiv:2007.14527.
[18] X. Meng, Z. Li, D. Zhang, G.E. Karniadakis, PPINN: Parareal physics-informed neural network for time-dependent PDEs, Comput. Methods Appl. Mech. Engrg. 370 (2020) 113250.
[19] A.D. Jagtap, G.E. Karniadakis, Extended physics-informed neural networks (XPINNs): A generalized space-time domain decomposition based deep learning framework for nonlinear partial differential equations, Commun. Comput. Phys. 28 (5) (2020) 2002–2041.
[20] V. Dwivedi, B. Srinivasan, Physics informed extreme learning machine (PIELM)–a rapid method for the numerical solution of partial differential equations, Neurocomputing 391 (2020) 96–118.
[21] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Trans. Neural Netw. 9 (5) (1998) 987–1000.
[22] H. Sheng, C. Yang, PFNN: A penalty-free neural network method for solving a class of second-order boundary-value problems on complex geometries, 2020, arXiv preprint arXiv:2004.06490.
[23] N. Sukumar, A. Srivastava, Exact imposition of boundary conditions with distance functions in physics-informed deep neural networks, 2021, arXiv preprint arXiv:2104.08426.
[24] K.S. McFall, J.R. Mahan, Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions, IEEE Trans. Neural Netw. 20 (8) (2009) 1221–1233.
[25] R.S. Beidokhti, A. Malek, Solving initial-boundary value problems for systems of partial differential equations using neural networks and optimization techniques, J. Franklin Inst. B 346 (9) (2009) 898–913.
[26] P.L. Lagari, L.H. Tsoukalas, S. Safarkhani, I.E. Lagaris, Systematic construction of neural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions, Int. J. Artif. Intell. Tools 29 (5) (2020).
[27] S. Dong, N. Ni, A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks, 2020, arXiv preprint arXiv:2007.07442.
[28] W. Cai, X. Li, L. Liu, A phase shift deep neural network for high frequency approximation and wave problems, SIAM J. Sci. Comput. 42 (5) (2020) A3285–A3312.
[29] B. Wang, W. Zhang, W. Cai, Multi-scale deep neural network (mscalednn) methods for oscillatory Stokes flows in complex domains, 2020, arXiv preprint arXiv:2009.12729.
[30] Z. Liu, W. Cai, Z.J. Xu, Multi-scale deep neural network (mscalednn) for solving Poisson-Boltzmann equation in complex domains, 2020, arXiv preprint arXiv:2007.11207.
[31] S. Wang, H. Wang, P. Perdikaris, On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks, 2020, arXiv preprint arXiv:2012.10047.
[32] Y. Deng, G. Lin, X. Yang, Multifidelity data fusion via gradient-enhanced Gaussian process regression, 2020, arXiv preprint arXiv:2008.01066.
[33] L. Laurent, R. Le Riche, B. Soulier, P.-A. Boucard, An overview of gradient-enhanced metamodels with applications, Arch. Comput. Methods Eng. 26 (1) (2019) 61–106.
[34] H. Drucker, Y. Le Cun, Improving generalization performance using double backpropagation, IEEE Trans. Neural Netw. 3 (6) (1992) 991–997.
[35] A.S. Ross, F. Doshi-Velez, Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018.
[36] A.G. Ororbia II, D. Kifer, C.L. Giles, Unifying adversarial training algorithms with data gradient regularization, Neural Comput. 29 (4) (2017) 867–887.
[37] C. Finlay, A.M. Oberman, Scaleable input gradient regularization for adversarial robustness, Mach. Learn. Appl. 3 (2021) 100017.
[38] A.S. Ross, M.C. Hughes, F. Doshi-Velez, Right for the right reasons: Training differentiable models by constraining their explanations, 2017, arXiv preprint arXiv:1703.03717.

[39] A. Ross, I. Lage, F. Doshi-Velez, The neural LASSO: Local linear sparsity for interpretable explanations, in: Workshop on Transparent and Interpretable Machine Learning in Safety Critical Environments, 31st Conference on Neural Information Processing Systems, 2017.
[40] Y. Shin, J. Darbon, G.E. Karniadakis, On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs, 2020, arXiv preprint arXiv:2004.01806.
[41] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, Automatic differentiation in machine learning: a survey, J. Mach. Learn. Res. 18 (2018).
[42] C.C. Margossian, A review of automatic differentiation and its efficient implementation, Wiley Interdiscip. Rev: Data Min. Knowl. Discov. 9 (4) (2019) e1305.
[43] J. Bettencourt, M.J. Johnson, D. Duvenaud, Taylor-mode automatic differentiation for higher-order derivatives in JAX, 2019.