Journal of Computational Physics 474 (2023) 111722

Contents lists available at ScienceDirect

Journal of Computational Physics

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

Self-adaptive physics-informed neural networks

Levi D. McClenny, Ulisses M. Braga-Neto *

Texas A&M University, United States of America

ARTICLE INFO

Article history:
Received 3 April 2022
Received in revised form 16 September 2022
Accepted 21 October 2022
Available online 14 November 2022

Keywords:
Physics-informed neural networks
Scientific machine learning
Numerical methods for PDE

ABSTRACT

Physics-Informed Neural Networks (PINNs) have emerged recently as a promising application of deep neural networks to the numerical solution of nonlinear partial differential equations (PDEs). However, it has been recognized that adaptive procedures are needed to force the neural network to fit accurately the stubborn spots in the solution of "stiff" PDEs. In this paper, we propose a fundamentally new way to train PINNs adaptively, where the adaptation weights are fully trainable and applied to each training point individually, so the neural network learns autonomously which regions of the solution are difficult and is forced to focus on them. The self-adaptation weights specify a soft multiplicative soft attention mask, which is reminiscent of similar mechanisms used in computer vision. The basic idea behind these SA-PINNs is to make the weights increase as the corresponding losses increase, which is accomplished by training the network to simultaneously minimize the losses and maximize the weights. In addition, we show how to build a continuous map of self-adaptive weights using Gaussian Process regression, which allows the use of stochastic gradient descent in problems where conventional gradient descent is not enough to produce accurate solutions. Finally, we derive the Neural Tangent Kernel matrix for SA-PINNs and use it to obtain a heuristic understanding of the effect of the self-adaptive weights on the dynamics of training in the limiting case of infinitely-wide PINNs, which suggests that SA-PINNs work by producing a smooth equalization of the eigenvalues of the NTK matrix corresponding to the different loss terms. In numerical experiments with several linear and nonlinear benchmark problems, the SA-PINN outperformed other state-of-the-art PINN algorithm in L2 error, while using a smaller number of training epochs.

1. Introduction

As part of the burgeoning field of scientific machine learning [1], physics-informed neural networks (PINNs) have emerged recently as an alternative to traditional numerical methods for partial different equations (PDE) [2–5]. Typical data-driven deep learning methodologies do not take into account physical understanding of the problem domain. The PINN approach is based on a strong physics prior that constrains the output of a deep neural network by means of a system of PDEs. The potential of using neural networks as universal function approximators to solve PDEs had been recognized since the 1990's [6]. However, PINNs promise to take this approach to a different level by using deep neural networks, which is made possible by the vast advances in computational capabilities and training algorithms since that time [7,8], as well as the availability of automatic differentiation methods [9,10].

* Corresponding author.
E-mail address: ulisses@tamu.edu (U.M. Braga-Neto).

https://doi.org/10.1016/j.jcp.2022.111722
0021-9991/

A great advantage of PINNs over traditional time-stepping PDE solvers is that it is possible to obtain the solution over the entire spatial-temporal domain at once, using training points distributed irregularly across the domain, obviating the need of constructing computationally-expensive grids. In addition, the PINN solution defines a function over the continuous domain, rather that a discrete solution on a grid as in traditional methods. Finally, PINNs allow sample data assimilation in a natural and efficient way.

The continuous PINN algorithm proposed in [2], henceforth referred to as the "baseline PINN" algorithm, is effective at estimating solutions that are reasonably smooth with simple boundary conditions, such as the viscous Burgers, Poisson and Schrödinger PDEs. On the other hand, it has been observed that the baseline PINN has convergence and accuracy problems when solving "stiff" PDEs [11], with solutions that contain sharp space transitions or fast time evolution [12]. This is known to be the case, for example, when attempting to solve the nonlinear Allen-Cahn equation with the baseline PINN [4]. As we will see in this paper, this may occur even in the case of the linear wave and advection PDEs.

This paper introduces Self-Adaptive PINNs (SA-PINNs), a fundamentally new method to train PINNs adaptively, which addresses the issues mentioned previously. SA-PINNs applies trainable weights on each training point, in a way that is reminiscent of soft multiplicative attention masks used in computer vision [13,14]. The adaptation weights are trained concurrently with the network weights. As a result, initial, boundary or residue points in difficult regions of the solution are automatically weighted more in the loss function, forcing the approximation to improve on those points. The basic principle in SA-PINNs is to make the weights increase as the corresponding losses do, which is accomplished by training the network to simultaneously minimize the losses and maximize the weights, i.e., to find a saddle point in the cost surface.

We also propose a methodology to build a continuous map of self-adaptive weights based on Gaussian Process regression, in order to allow the use of stochastic gradient descent in training self-adaptive PINNs. This is illustrated by application to a 1-D wave PDE that is challenging to non-SGD training.

Finally, we derive the Neural Tangent Kernel matrix for self-adaptive PINNs and use it to obtain a heuristic understanding of the effect of the self-adaptive weights on the dynamics of training in the limiting case of infinitely-wide PINNs. We examine the effect of the self-adaptive weights on the eigenvalues of the NTK matrix in the solution of a linear advection PDE, and observe that it not only equalizes the magnitudes between the different loss components, but also smooths the shape of the distribution of eigenvalues. This provides preliminary theoretical justification of the success of self-adaptive PINNs.

Comprehensive experimental results presented throughout the test, based on several well-known benchmarks, show that self-adaptive PINNs can solve "stiff" PDEs with significantly better accuracy than other state-of-the-art PINN algorithms, while using a smaller number of training epochs.

2. Background

2.1. Physics-informed neural networks

Consider the initial-boundary value problem:

$$\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] = f(\boldsymbol{x},t)\,,\ \ \boldsymbol{x}\in\Omega\,,\ t\in(0,T]\,, \tag{1}$$

$$\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] = g(\boldsymbol{x},t)\,,\ \ \boldsymbol{x}\in\partial\Omega,\ t\in(0,T]\,, \tag{2}$$

$$u(\boldsymbol{x},0) = h(\boldsymbol{x})\,,\ \ \boldsymbol{x}\in\overline{\Omega}\,. \tag{3}$$

Here, the domain $\Omega\subset R^d$ in a open set, $\overline{\Omega}$ is its closure, $u:\overline{\Omega}\times[0,T]\to R$ is the desired solution, $\boldsymbol{x}\in\Omega$ is a spatial vector variable, t is time, and $\mathcal{N}_{\boldsymbol{x},t}$ and $\mathcal{B}_{\boldsymbol{x},t}$ are spatial-temporal differential operators. The problem *data* is provided by the forcing function $f:\Omega\to R$, the boundary condition function $g:\partial\Omega\times(0,T]$, and the initial condition function $h:\overline{\Omega}\to R$. Additionally, sensor data in the interior of the domain may be available. In any case, we assume that the data are sufficient and appropriate for a well-posed problem. Time-independent problems and other types of data can be handled similarly, so we will use the equations (1)-(3) as a model.

Following [2], let $u(\boldsymbol{x},t)$ be approximated by the output $u(\boldsymbol{x},t;\boldsymbol{w})$ of a deep neural network with inputs $\boldsymbol{x}$ and t (in the case of a PDE system, this would be a neural network with multiple outputs). The value of $\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t;\boldsymbol{w})]$ and $\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x},t;\boldsymbol{w})]$ can be computed quickly and accurately using reverse-mode *automatic differentiation* [9,10].

The network weights $\boldsymbol{w}$ are trained by minimizing a loss function that penalizes the output for not satisfying (1)-(3):

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w}) + \mathcal{L}_0(\boldsymbol{w})\,, \tag{4}$$

where $\mathcal{L}_s$ is the loss term corresponding to sample data (if any), while $\mathcal{L}_r$, $\mathcal{L}_b$, and $\mathcal{L}_0$ are loss terms corresponding to not satisfying the PDE (1), the boundary condition (2), and the initial condition (3), respectively:

$$\mathcal{L}_s(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_s}|u(\boldsymbol{x}_s^i,t_s^i;\boldsymbol{w}) - y_s^i|^2, \tag{5}$$

$$\mathcal{L}_r(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_r} |\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] - f(\boldsymbol{x}_r^i, t_r^i)|^2, \tag{6}$$

$$\mathcal{L}_b(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_b} |\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w})] - g(\boldsymbol{x}_b^i, t_b^i)|^2, \tag{7}$$

$$\mathcal{L}_0(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_0} |u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}) - h(\boldsymbol{x}_0^i)|^2, \tag{8}$$

where $\{\boldsymbol{x}_s^i, t_s^i, y_s^i\}_{i=1}^{N_s}$ are sensor data (if any), $\{\boldsymbol{x}_0^i\}_{i=1}^{N_0}$ are initial condition points, $\{\boldsymbol{x}_b^i, t_b^i\}_{i=1}^{N_b}$ are boundary condition points, $\{\boldsymbol{x}_r^i, t_r^i\}_{i=1}^{N_r}$ are residue ("collocation") points randomly distributed in the domain Ω, and N_s, N_0, N_b and N_r denote the total number of sensor, initial, boundary, and residue points, respectively. The network weights $\boldsymbol{w}$ can be tuned by minimizing the total training loss $\mathcal{L}(\boldsymbol{w})$ via standard gradient descent procedures used in deep learning.

2.2. Related work

The baseline PINN algorithm described in the previous section, though remarkably successful in the solution of many linear and nonlinear PDEs, can produce inaccurate approximations, or fail to converge entirely, in the solution of certain "stiff" PDEs. A large amount of evidence has accumulated indicating that this happens due to the shortcomings of gradient descent applied to the multi-part or multi-objective loss function (4); e.g., see [4,15,12,5]. This occurs because gradient descent is a greedy procedure that may latch on some of the components at the expense of others, which creates imbalance in the rate of descent among the different loss components and prevents convergence to the correct solution. The standard approach in the literature of PINNs to try to correct the imbalance is the introduction of weights in (4):

$$\mathcal{L}(\boldsymbol{w}) = \lambda_s\mathcal{L}_s(\boldsymbol{w}) + \lambda_r\mathcal{L}_r(\boldsymbol{w}) + \lambda_b\mathcal{L}_b(\boldsymbol{w}) + \lambda_0\mathcal{L}_0(\boldsymbol{w})\,. \tag{9}$$

Several methods, from very simple to complex, have been advanced to set the values of these weights; we mention a few below.

Nonadaptive weighting. In [4], it was pointed out that a premium should be put on forcing the neural network to satisfy the initial conditions closely, especially for PDEs describing time-irreversible processes, where the solution has to be approximated well early. Accordingly, a loss function of the form $\mathcal{L}(\theta) = \mathcal{L}_r(\theta) + \mathcal{L}_b(\theta) + C\,\mathcal{L}_0(\theta)$ was suggested, where $C \gg 1$ is a hyperparameter.

Learning rate annealing. In [12], it is argued that the optimal value of the weight C in the previous scheme may vary wildly among different PDEs so that choosing its value would be difficult. Instead they propose to use weights that are tuned during training using statistics of the backpropagated gradients of the loss function. It is noteworthy that the weights themselves are not adjusted by backpropagation. Instead, they behave as learning rate coefficients, which are updated after each epoch of training.

Adaptive resampling. In [4], a strategy to adaptively resample the residual collocation points based on the magnitude of the residual is proposed. While this approach improves the approximation, the training process must be interrupted and the MSE evaluated on the residual points to deterministically resample the ones with the highest error. After each resampling step, the number of residual points grows, increasing computational complexity. In [16], resampling of the collocation points for solving the steady-state Fokker-Planck PDE is performed using an approximate density function, while in [17], this work is extended to time-evolution problems by means of an adaptive density approximation method based on normalizing flows.

Neural tangent kernel (NTK) weighting. Recently, [5] derived the NTK kernel matrix for PINNs, and used a heuristic argument to set the weights adaptively based on the evolution of the eigenvalues of the NTK matrix during training.

Mimimax weighting. In [18], a methodology was proposed to update the weights during training using gradient descent for the network weights, and gradient ascent for the loss weights, seeking to find a saddle point in weight space. Loss components that do not decrease are assigned larger weights.

More generally, the need to use weighting to correct imbalance in multi-part loss functions has been recognized in the general deep learning literature [19–21]. Note that the multi-part loss (9) corresponds to a *linear scalarization* of this multiple-objective problem [22].

All the previous methods employ a linearly-scalarized function such as (9), the only difference among them being the way the weights are updated. The self-adaptive weighting method proposed in this paper is fundamentally different in that the weights apply to individual training points in the different loss components, rather than the entire loss component. The previous methods can be seen as a special case of this, when all self-adaptive weights for a particular loss component

are updated in tandem. Among the previous methods, the independently-developed Minimax weighting scheme [18] is the closest to SA-PINNs, as it also updates its weights via gradient ascent; however, these weights still apply to the whole loss components. This paper presents empirical and theoretical evidence that having the flexibility of weighting each training point in the various loss terms brings additional flexibility that can lead to better performance.

3. Self-adaptive physics-informed neural networks

While previously proposed weighting methods produce improvements in stability and accuracy over the baseline PINN, they are either nonadaptive or introduce inflexible adaptation. Here we propose a simple procedure that applies fully-trainable weights to produce a multiplicative soft attention mask, in a manner that is reminiscent of attention mechanisms used in computer vision [13,14]. Instead of hard-coding weights at particular regions of the solution, the proposed method is in agreement with the neural network philosophy of self-adaptation, where the weights in the loss function are updated by gradient descent side-by-side with the network weights.

Using the PDE in (1)-(3) as reference, the proposed self-adaptive PINN utilizes the following loss function

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\lambda}_r) + \mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) + \mathcal{L}_0(\boldsymbol{w}, \boldsymbol{\lambda}_0)\,, \tag{10}$$

where $\boldsymbol{\lambda}_r = (\lambda_r^1, \ldots, \lambda_r^{N_r})$, $\boldsymbol{\lambda}_b = (\lambda_b^1, \ldots, \lambda_b^{N_b})$, and $\boldsymbol{\lambda}_0 = (\lambda_0^1, \ldots, \lambda_0^{N_0})$ are trainable, nonnegative *self-adaptation weights* for the initial, boundary, and residue points, respectively, and

$$\mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\lambda}_r) = \frac{1}{2}\sum_{i=1}^{N_r} m(\lambda_r^i)\,|\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] - f(\boldsymbol{x}_r^i, t_r^i)|^2 \tag{11}$$

$$\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) = \frac{1}{2}\sum_{i=1}^{N_b} m(\lambda_b^i)\,|\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] - g(\boldsymbol{x}_b^i, t_b^i)|^2 \tag{12}$$

$$\mathcal{L}_0(\boldsymbol{w}, \boldsymbol{\lambda}_0) = \frac{1}{2}\sum_{i=1}^{N_0} m(\lambda_0^i)\,|u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}) - h(\boldsymbol{x}_0^i)|^2, \tag{13}$$

where the *self-adaptation mask function* $m(\lambda)$ defined on $[0, \infty)$ is a nonnegative, differentiable on $(0, +\infty)$, strictly increasing function of λ. A key feature of self-adaptive PINNs is that the loss $\mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0)$ is minimized with respect to the network weights $\boldsymbol{w}$, as usual, but is *maximized* with respect to the self-adaptation weights $\boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0$. The corresponding gradient descent/ascent steps are:

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - \eta_k \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \tag{14}$$

$$\boldsymbol{\lambda}_r^{k+1} = \boldsymbol{\lambda}_r^k + \rho_r^k \nabla_{\boldsymbol{\lambda}_r} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \tag{15}$$

$$\boldsymbol{\lambda}_b^{k+1} = \boldsymbol{\lambda}_b^k + \rho_b^k \nabla_{\boldsymbol{\lambda}_b} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \tag{16}$$

$$\boldsymbol{\lambda}_0^{k+1} = \boldsymbol{\lambda}_0^k + \rho_0^k \nabla_{\boldsymbol{\lambda}_0} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k)\,, \tag{17}$$

where $\eta^k > 0$ is the learning rate for the neural network weights at step k, $\rho_p^k > 0$ is a separate learning rate for the self-adaption weights, for $p = r, b, 0$, and

$$\nabla_{\boldsymbol{\lambda}_r} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) = \frac{1}{2}\begin{bmatrix} m'(\lambda_r^{k,1})\left|\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w}^k)] - f(\boldsymbol{x}_r^1, t_r^1)\right|^2 \\ \cdots \\ m'(\lambda_r^{k,N_r})\left|\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w}^k)] - f(\boldsymbol{x}_r^{N_r}, t_r^{N_r})\right|^2 \end{bmatrix}, \tag{18}$$

$$\nabla_{\boldsymbol{\lambda}_b} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) = \frac{1}{2}\begin{bmatrix} m'(\lambda_b^{k,1})\left|\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w}^k)] - g(\boldsymbol{x}_b^1, t_b^1)\right|^2 \\ \cdots \\ m'(\lambda_b^{k,N_b})\left|\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w}^k)] - g(\boldsymbol{x}_b^{N_b}, t_b^{N_b})\right|^2 \end{bmatrix}, \tag{19}$$

$$\nabla_{\boldsymbol{\lambda}_0} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) = \frac{1}{2}\begin{bmatrix} m'(\lambda_0^{k,1})\left|u(\boldsymbol{x}_0^1, 0; \boldsymbol{w}^k) - h(\boldsymbol{x}_0^1, t_0^1)\right|^2 \\ \cdots \\ m'(\lambda_0^{k,N_0})\left|u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}^k)] - h(\boldsymbol{x}_0^{N_0})\right|^2 \end{bmatrix}. \tag{20}$$

Hence, since $m'(\lambda) > 0$ (the mask function is strictly increasing, by assumption), then $\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_b}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_0}\mathcal{L} \geq 0$, and any gradient component is zero if and only if the corresponding unmasked loss is zero. This shows that the sequences of weights $\{\boldsymbol{\lambda}_r^k; k = 1, 2, \ldots\}$, $\{\boldsymbol{\lambda}_b^k; k = 1, 2, \ldots\}$, $\{\boldsymbol{\lambda}_0^k; k = 1, 2, \ldots\}$ (and the associated mask values) are monotonically increasing, provided

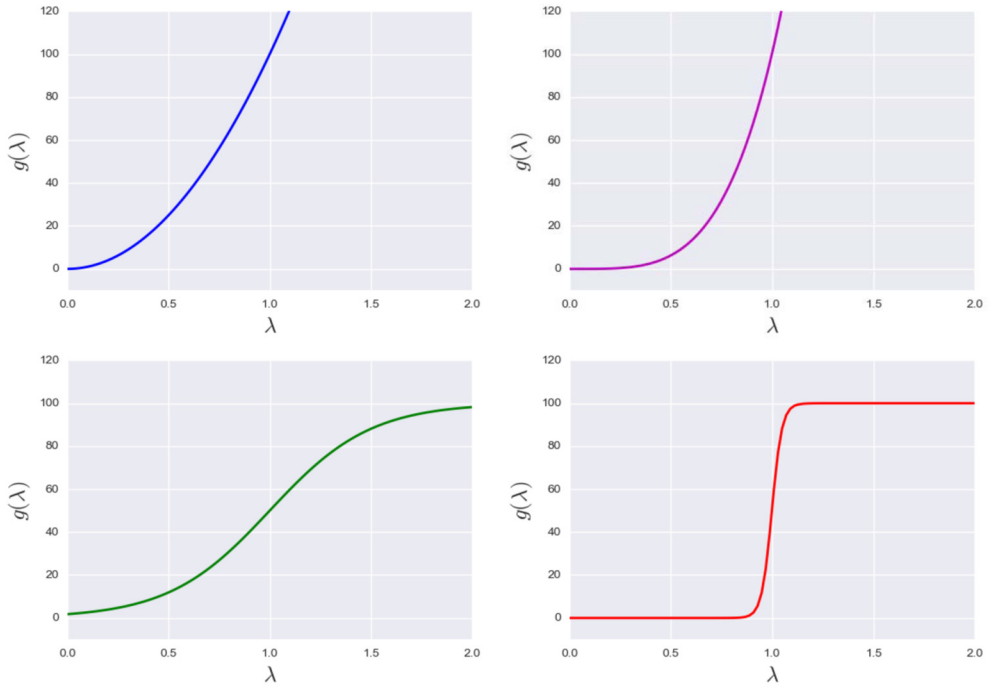

Fig. 1. Mask function examples. From the upper left to the bottom right: polynomial mask, $q = 2$; polynomial mask, $q = 4$; smooth logistic mask; sharp logistic mask.

that the corresponding unmasked losses are nonzero. Furthermore, the magnitude of the gradients $\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_b}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_0}\mathcal{L}$, and therefore of the updates, are larger if the corresponding unmasked losses are larger. In addition, the magnitude of updates can be controlled by specifying a schedule for the learning rates ρ_p^k, for $p = r, b, 0$, adding extra flexibility. This progressively penalizes the network more for not fitting the residual, boundary, and initial points closely.

We remark that any of the weights can be set to fixed, non-trainable values, if desired. For example, by setting $\lambda_b^k \equiv 1$, only the weights of the initial and residue points would be trained. The sensor data loss is not masked in this formulation, since if these data consist of noisy observations, weighting them requires extra care to avoid overfitting (though this remains an open research problem).

Notice that the self-adaptive weights need to be initialized at the beginning of training. They could be initialized to 1 (no weighting) or to a different value, depending on the problem. They could also be initialized randomly over an interval, similarly as to how neural network weights are often initialized. Here, prior knowledge plays an important role; e.g., if it is known that the initial conditions in a problem are hard to fit, then the initial condition weights could be initialized to a larger value than the other weights (alternatively, it could be initialized at the same value as the other weights, but employ a larger learning rate).

The shape of the function g affects mask sharpness and training of the PINN. Examples include polynomial masks $m(\lambda) = c\lambda^q$, for $c, q > 0$, and sigmoidal masks. See Fig. 1 for a few examples. In practice, the polynomial mask functions have to be kept below a suitable (large) value, to avoid numerical overflow. The sigmoidal masks do not have this issue, and can be used to produce sharp masks. For example, in the bottom right example in Fig. 1, the mask is essentially binary; it starts small for small starting values of the self-adaptive weight λ, and after these exceed a certain threshold, the mask value will quickly take on the upper saturation value. Similarly to neural network nonlinearities, sigmoid mask functions can suffer from vanishing gradients during training. This is particularly a problem at the lower starting value. Therefore, excessively sharp sigmoidal mask functions should be avoided.

The gradient ascent/descent step can be implemented easily using off-the-self neural network software, by simply flipping the sign of $\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}$, $\nabla_{\boldsymbol{\lambda}_b}\mathcal{L}$, and $\nabla_{\boldsymbol{\lambda}_0}\mathcal{L}$. In our implementation of SA-PINNs, we use Tensorflow 2.3 with a fixed number of iterations of Adam [23]. In some case, these are followed by another fixed number of iterations of the L-BFGS quasi-newton method [24]. This is consistent with the baseline PINN formulation in [2], as well as follow-up literature [4]. However, the adaptive weights are only updated in the Adam training steps, and are held constant during L-BFGS training, if any. A full implementation of the methodology described here has been made publicly available by the authors[1] and it is included in the open-source software *TensorDiffEq* [25]. Finally, we remark that there are some similarities between SA-PINN training and penalty methods in optimization, which introduce a sequence of increasing penalty costs [26].

[1] https://github.com/levimcclenny/SA-PINNs.

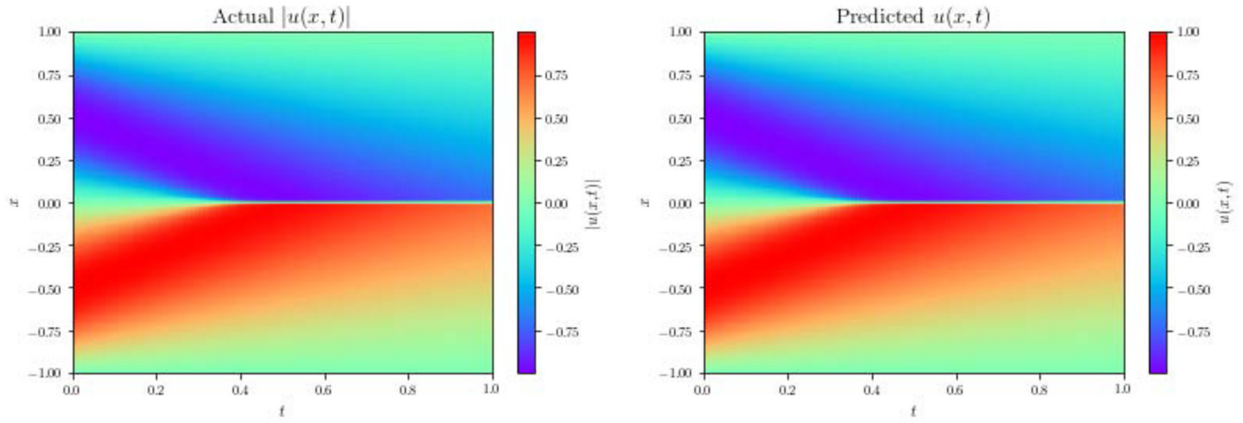

Fig. 2. High-fidelity (*left*) vs. predicted (*right*) solutions for the viscous Burgers PDE. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

4. Numerical examples

In this section we present numerical experiments demonstrating the SA-PINN performance on various benchmarks. The main figure of merit used is the L2-error:

$$L_2 \text{ error} = \frac{\sqrt{\sum_{i=1}^{N_U} |u(x_i,t_i) - U(x_i,t_i)|^2}}{\sqrt{\sum_{i=1}^{N_U} |U(x_i,t_i)|^2}}, \tag{21}$$

where $u(x,t)$ is the trained approximation, and $U(x,t)$ is a high-fidelity solution over a fine mesh $\{x_i,t_i\}$ containing N_U points. In all cases below, we repeat the training process over 10 random restarts and report the average L2 error and its standard deviation.

4.1. Viscous Burgers equation

The viscous Burgers PDE considered here is

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0\,, \quad x \in [-1,1],\ t \in [0,1]\,, \tag{22}$$

$$u(0,x) = -\sin(\pi x)\,, \tag{23}$$

$$u(t,-1) = u(t,1) = 0\,. \tag{24}$$

All results for the viscous Burgers PDE were generated from a fully-connected network with input layer size 2 corresponding to the x and t inputs, 8 hidden layers of 20 neurons each, and an output layer of size 1 corresponding to the output of the approximation $u(x,t)$. This directly mimics the setup of the viscous Burgers PDE result presented in [2]. All training is done for 10k iterations of Adam, followed by 10k iterations of L-BFGS to fine tune the network weights, consistent with related work. Additionally, the number of points selected for the trials shown are N_0 = 100, N_b = 200, and N_r = 10000. Training with this architecture took 96 ms/iteration on a single Nvidia V100 GPU. We initialize the self adaptive weights on the IC and the residual points to be $U(0,1)$ and the learning rates for all self-adaptive weights were set to $5e{-}3$.

We achieved an L2-error of $4.80e{-}4 \pm 1.01e-4$, which is smaller value than the $6.7e{-}4$ L2 error reported in [2], while using only 20% as many training iterations and an identical neural network architecture. The high-fidelity and predicted solutions are displayed in Fig. 2. Fig. 3 demonstrate the accuracy of the proposed approach, using a significantly shorter training horizon than the baseline PINN.

Fig. 4 shows that the sharp discontinuity at $x = 0$ in the solution has correspondingly large weights, indicating that the model must pay extra attention to those particular points in its solution, resulting in an increase in approximation accuracy and training efficiency.

4.2. Helmholtz equation

The Helmholtz PDE is typically used to describe the behavior of wave and diffusion processes, and can be employed to model evolution in a spatial domain or combined spatial-temporal domain. Here we study a particular Helmholtz PDE existing only in the spatial (x, y) domain, described as:

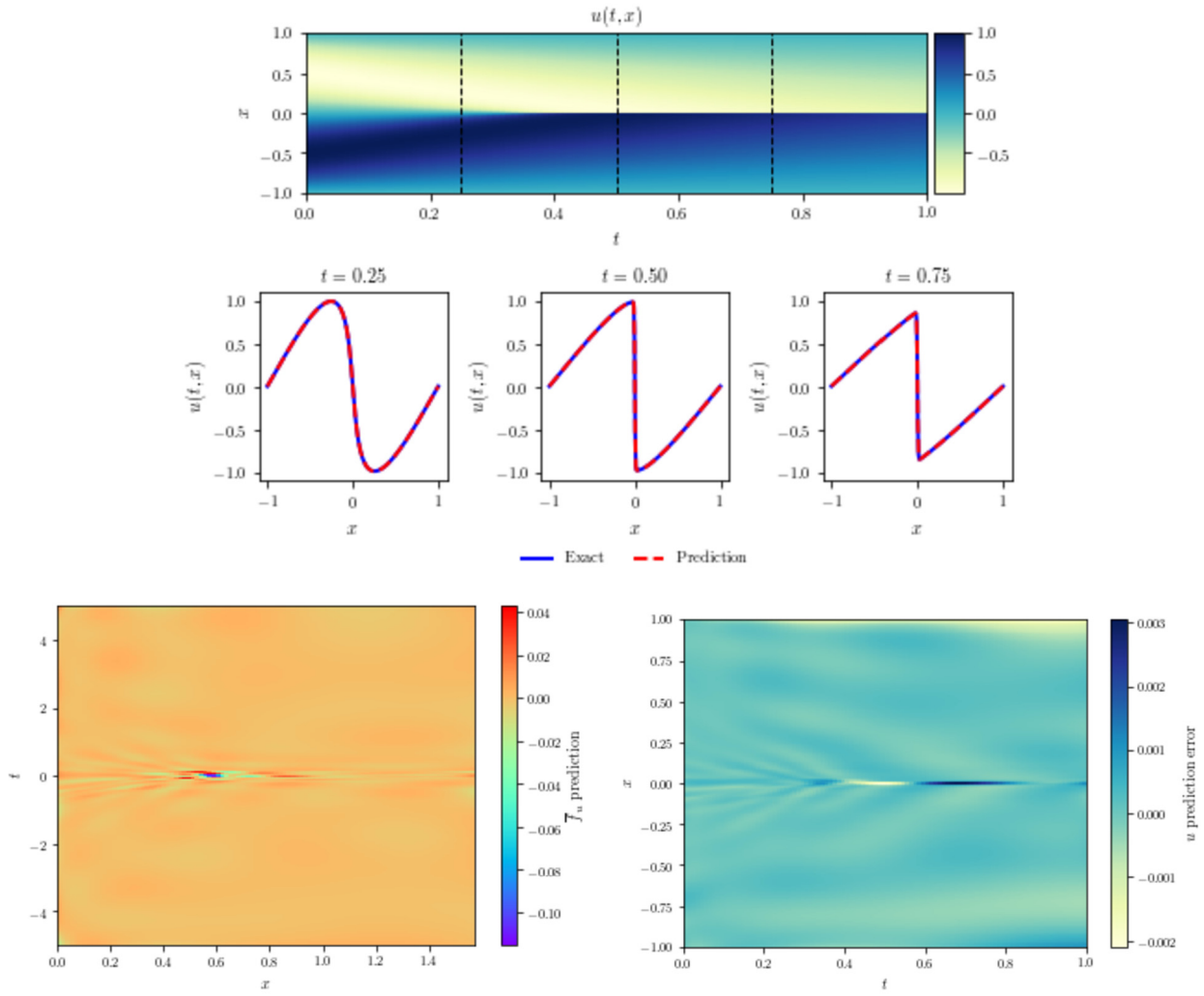

Fig. 3. *Top:* predicted solution of the viscous Burgers PDE. *Middle:* Cross-sections of the approximated vs. actual solutions for various x-domain snapshots. *Bottom left:* Residual $r(x,t)$ across the spatial-temporal domain. *Bottom right:* Absolute error between prediction and high-fidelity solution across the spatial-temporal domain.

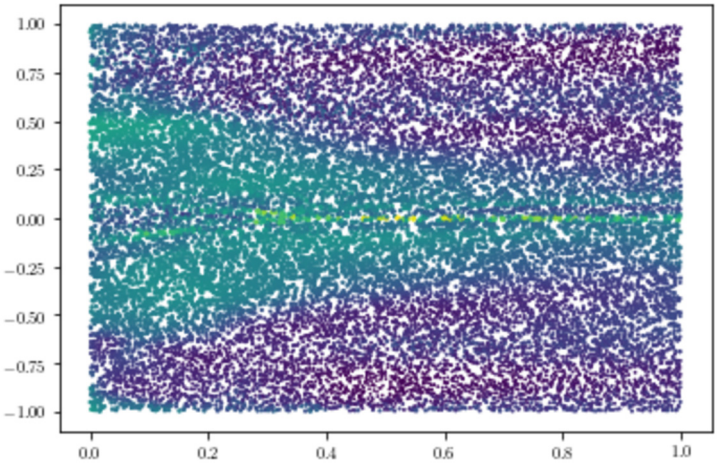

Fig. 4. Trained weights for residue points across the domain Ω. Larger/brighter colored points correspond to larger weights.

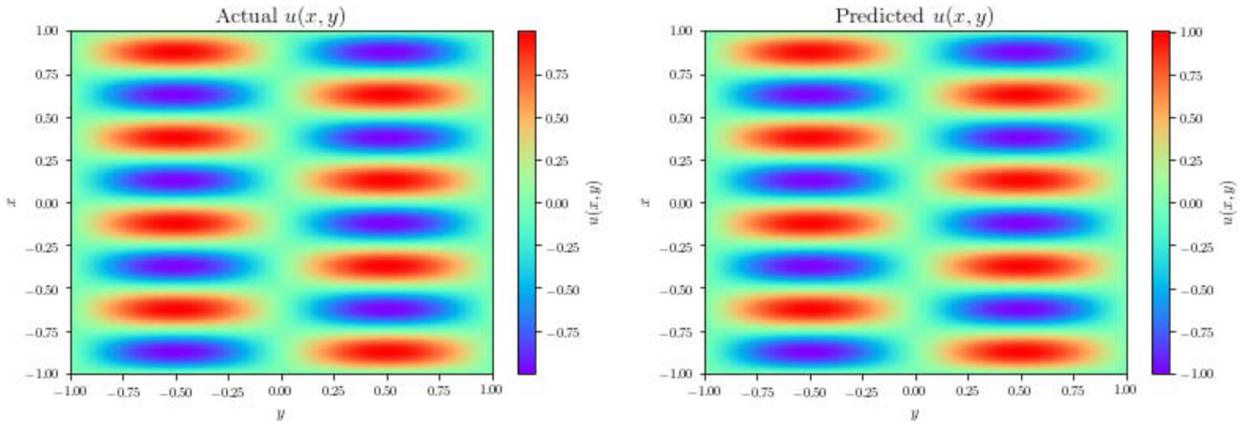

Fig. 5. Exact (*left*) vs. predicted (*right*) solutions for the Helmholtz PDE.

$$u_{xx} + u_{yy} + k^2 u - q(x, y) = 0 \tag{25}$$

$$u(-1, y) = u(1, y) = u(x, -1) = u(x, 1) = 0 \tag{26}$$

where $x \in [-1, 1]$, $y \in [-1, 1]$ and

$$\begin{aligned} q(x, y) = &- (a_1\pi)^2 \sin(a_1\pi x) \sin(a_2\pi y) \\ &- (a_2\pi)^2 \sin(a_1\pi x) \sin(a_2\pi y) \\ &+ k^2 \sin(a_1\pi x) \sin(a_2\pi y) w \end{aligned} \tag{27}$$

is a forcing term that results in a closed-form analytical solution

$$u(x, y) = \sin(a_1\pi x) \sin(a_2\pi y)\,. \tag{28}$$

To allow a direct comparison to the Helmholtz PDE result reported in [12], we take a_1 = 1 and a_2 = 4 and use the same neural network architecture with layer sizes [2, 50, 50, 50, 50, 1]. Our architecture is trained for 10k Adam and 10k L-BFGS iterations, again keeping the self-adaptive mask weights constant through the L-BFGS training iterations and only allowing those to train via Adam. We sample N_b = 400 (100 points per boundary). Given the steady-state initialization and constant forcing term, there is no applicable initial condition and consequently no N_0. We create a mesh of size (1001,1001) corresponding to the $x \in [-1, 1]$, $y \in [-1, 1]$ range, yielding 1,002,001 total mesh points, from which we select N_r=100k residue points. We initialize the self adaptive weights on the BC and the residual points to be $U(0, 1)$ and the learning rates for all self-adaptive weights were set to 5e-3.

We can see in Fig. 5 that the SA-PINN prediction is very accurate and indistinguishable from the exact solution, with an L2 error of $3.2e-3 \pm 2.2e-4$. With a larger neural network, the results reported in [12] (row 5 of Table 2) are $1.4e-1$ for the baseline PINN, and between $2.54e-3$ and $2.74e-2$ for various learning-rate annealing weighted schemes proposed in that paper. We would add that the SA-PINN is trained for a smaller number of iterations (10k Adam and 10k L-BFGS) with respect to that in [12] (40k Adam).

Fig. 6 shows individual cross-sections of the Helmholtz solution, demonstrating the SA-PINN's ability to accurately approximate the sinusoidal solution on the whole domain. Fig. 7 shows that the Self-Adaptive PINN largely ignores the flat areas in the solution, while focusing its attention on the nonflat areas.

4.3. Allen-Cahn reaction-diffusion equation

In this section, we report experimental results obtained with the Allen-Cahn PDE, which contrast the performance of the proposed SA-PINN algorithm against the baseline PINN and two of the PINN algorithms mentioned in Section 2.2, namely, the nonadaptive weighting and time-adaptive schemes (for the latter, Approach 1 in [4] was used).

The Allen-Cahn reaction-diffusion PDE is typically encountered in phase-field models, which can be used, for instance, to simulate the phase separation process in the microstructure evolution of metallic alloys [27–29]. The Allen-Cahn PDE considered here is specified as follows:

$$u_t - 0.0001 u_{xx} + 5u^3 - 5u = 0\,, \quad x \in [-1, 1],\ t \in [0, 1]\,, \tag{29}$$

$$u(x, 0) = x^2 cos(\pi x)\,, \tag{30}$$

$$u(t, -1) = u(t, 1)\,, \tag{31}$$

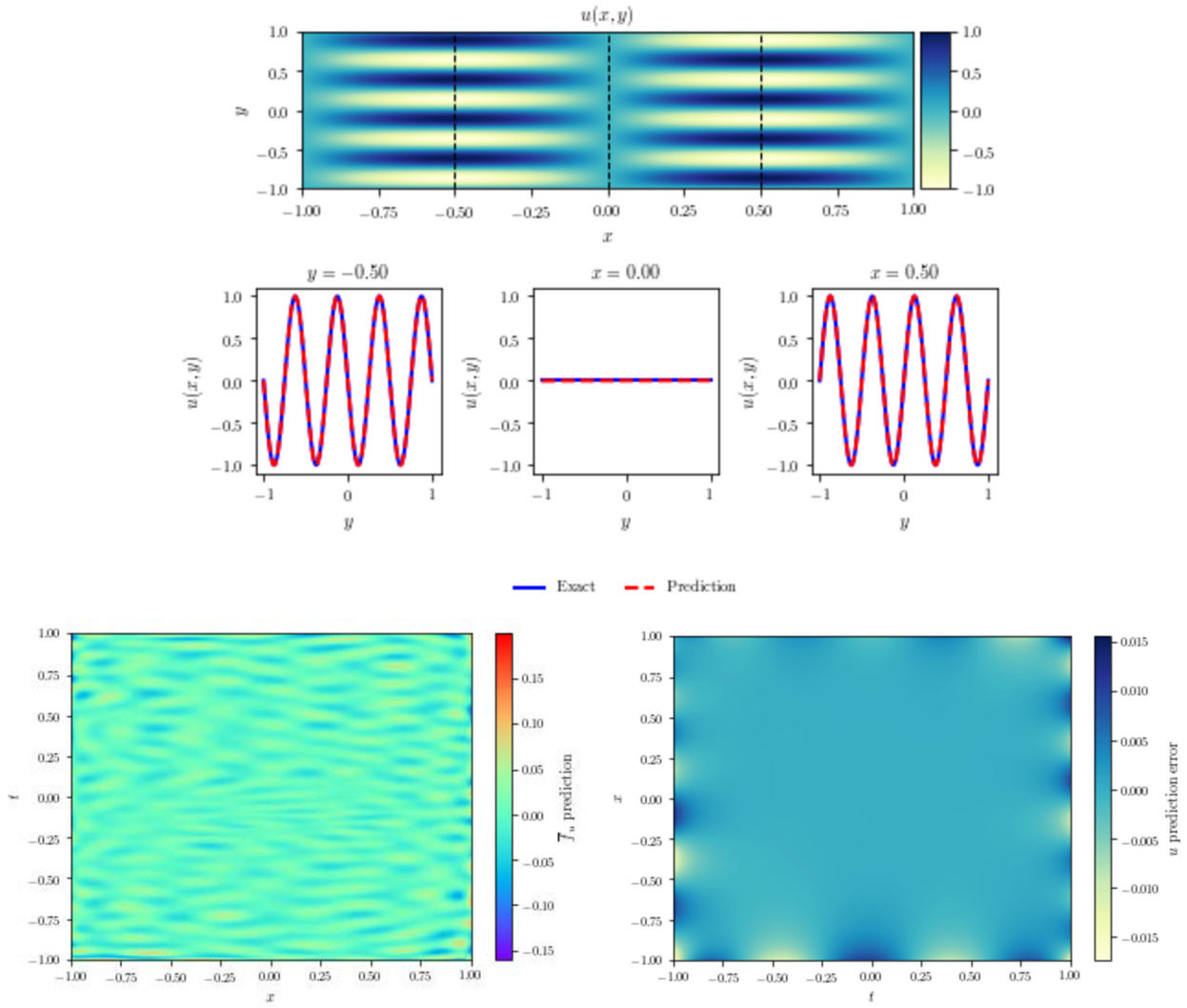

Fig. 6. *Top* predicted solution of Helmholtz equation. *Bottom* Cross-sections of the approximated vs. actual solutions for various x-domain snapshots.

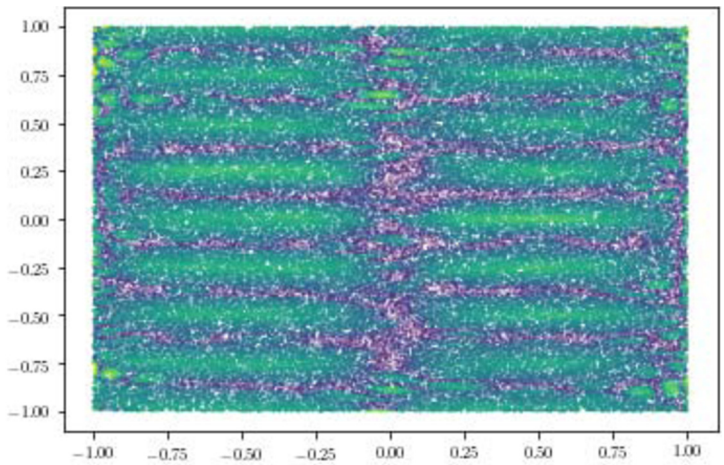

Fig. 7. Self-learned weights after training via Adam for the Helmholtz system. Brighter/larger points correspond to larger weights.

$$u_x(t,-1) = u_x(t,1)\,. \tag{32}$$

The Allen-Cahn PDE is an interesting benchmark for PINNs for multiple reasons. It is a "stiff" PDE that challenges PINNs to approximate solutions with sharp space and time transitions, and is also introduces periodic boundary conditions (31), (32). In order to deal with the latter, the boundary loss function $\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b)$ in (12) is replaced by

$$\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) = \frac{1}{N_b}\sum_{i=1}^{N_b} g(\lambda_b^i)(|u(1,t_b^i) - u(-1,t_b^i)|^2 + |u_x(1,t_b^i) - u_x(-1,t_b^i)|^2) \tag{33}$$

The neural network architecture is fully connected with layer sizes $[2, 128, 128, 128, 128, 1]$. This architecture is identical to the one used in the Allen-Cahn PDE result reported in [4], in order to allow a direct comparison of performance. We set the number of residue, initial, and boundary points to $N_r = 20,000$, $N_0 = 100$ and $N_b = 100$, respectively (due to the periodic boundary condition, there are in fact 200 boundary points). Here we hold the boundary weights w_b^i at 1, while the initial weights w_0^i and residue weights w_r^i are trained. The initial and residue weights are initialized from a uniform distribution in the intervals $[0, 100]$ and $[0, 1]$, respectively. Training took 65 ms/iteration on a single Nvidia V100 GPU.

Numerical results obtained with the SA-PINN are displayed in Fig. 8. The average L2-error across 10 runs with random restarts was $2.1e{-}2 \pm 1.21e{-}2$, while the L2-error on 10 runs obtained by the time-adaptive approach in [4] was $8.0e{-}2 \pm 0.56e{-}2$. Neither the baseline PINN nor the nonadaptive weighted scheme, with initial condition weight $C = 100$, were able to solve this PDE satisfactorily, with L2 errors $96.15e{-}2 \pm 6.45e{-}2$ and $49.61e{-}2 \pm 2.50e{-}2$, respectively (these numbers matched almost exactly those reported in [4]).

Fig. 9 is unique to the proposed SA-PINN algorithm. It displays the trained self-adaptive weights for the residue points across the spatio-temporal domain. These are the weights of the multiplicative soft attention mask self-imposed by the PINN. This plot stays remarkably constant across different runs with random restarts, which is an indication that it is a property of the particular PDE being solved. We can observe that in this case, more attention is needed early in the solution, but not uniformly across the space variable. In [4], this observation was justified by the fact that the Allen-Cahn PDEs describes a time-irreversible diffusion-reaction processes, where the solution has to be approximated well early. However, here this fact is "discovered" by the SA-PINN itself.

In order to study the behavior of the SA-PINN more closely, we plot in Fig. 10 the average value of the residue weights from various partitions of the solution domain. While all the weights are increasing, as must be the case since the mask function is required to be monotone, the rate of increase is of importance. Notice that the initial condition weights grow much faster than the residue weights, as expected, since the initial condition tends to be neglected by the PINN, otherwise. As for the residue weights, we see that, for small values of t, they increase faster than for large values of t. This shows that the SA-PINN has learned that the early part of the evolution is the most critical part of the solution. (This agrees with what was seen in the map of Fig. 9.) In contrast with traditional time-marching approaches, where earlier time steps are solved prior to later ones, the SA-PINN solves the PDE over the entire space-time domain at once; however, the self-adaptive weights allow it to concentrate on the early part of the evolution.

Finally, Fig. 12 displays the training loss for the baseline PINN and SA-PINN as a function of training iteration. For the SA-PINN, the weights were removed from the loss value to provide a direct comparison to the baseline. These plots are generated from 10 random restarts of the SA and baseline PINN training cycles over 10k Adam training iterations with consistent learning rates. We can see that the SA-PINN achieves significantly lower initial condition loss than the baseline PINN for the initial. Indeed, this is the major issue faced by the baseline PINN in the AC problem. As for the residual loss, we see that the baseline PINN decreases it fast (at the expense of the initial condition loss), but that eventually the SA PINN is able to achieve a loss two orders of magnitude smaller. The oscillatory behavior of the residue loss in the SA-PINN reveals the dynamics of the competing self-adaptive weighted initial condition and residue loss terms.

4.4. 2D Burgers equation

Here we demonstrate the efficacy of SA-PINN in the solution of a three-dimensional problem (two spatial dimensions plus time), namely, a 2D viscous Burgers nonlinear PDE system with velocity fields $u(x, y, t)$ and $v(x, y, t)$, which satisfy:

$$u_t + uu_x + vu_y = \nu(u_{xx} + u_{yy})\,, \tag{34}$$

$$v_t + uv_x + vv_y = \nu(v_{xx} + v_{yy})\,, \tag{35}$$

in the domain $(x, y, t) \in (0, 1)^3$, where the ν is the kinematic viscosity (in this example, $\nu = 0.002$). Any velocity fields such that $u(x, y, t) + v(x, y, t)$ is constant provide a solution of this PDE system. Following [30], we take:

$$u(x, y, t) = \frac{3}{4} - \frac{1}{4}\left[1 + \exp\left(\frac{-4x + 4y - t}{32\nu}\right)\right]^{-1}, \tag{36}$$

$$v(x, y, t) = \frac{3}{4} + \frac{1}{4}\left[1 + \exp\left(\frac{-4x + 4y - t}{32\nu}\right)\right]^{-1}, \tag{37}$$

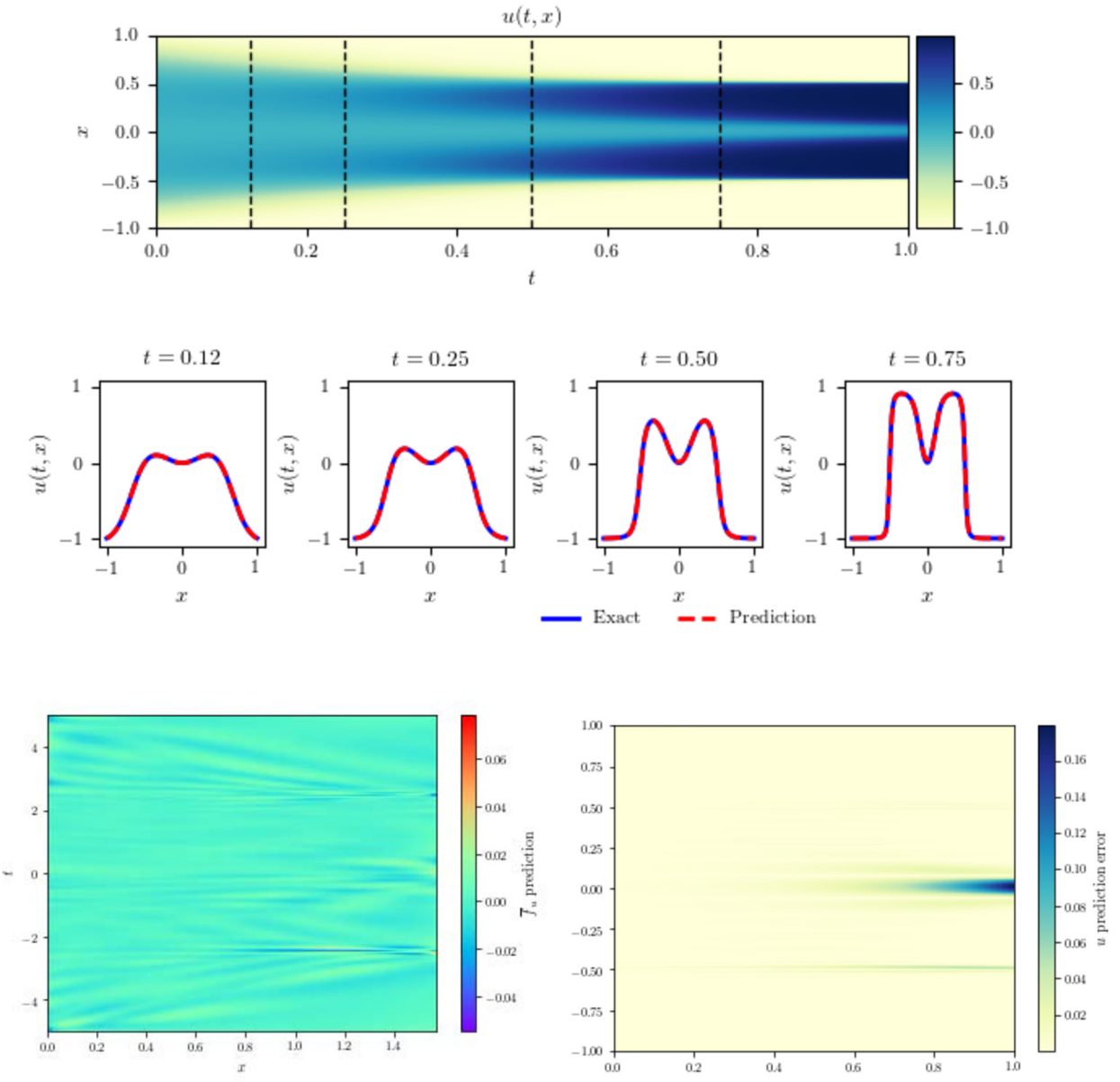

Fig. 8. *Top:* Plot of the approximation $u(x,t)$ via the SA-PINN. *Middle:* Snapshots of the approximation $u(x,t)$ vs. the high-fidelity solution $U(x,t)$ at various time points through the temporal evolution. *Bottom left:* Residual $r(x,t)$ across the spatial-temporal domain. As expected, it is close to 0 for the whole domain Ω. *Bottom right:* Absolute error between approximation and high-fidelity solution across the spatial-temporal domain.

with matching initial and Dirichlet boundary conditions. We apply the SA-PINN without enforcing the boundary conditions, i.e., only the initial conditions are enforced. Despite this, the SA-PINN was able to capture the attenuated shock effectively as shown in Fig. 11. These results were obtained with 35k Adam iterations at a learning rate of $1e-5$, followed by 20k L-BFGS iterations to fine-tune the solution. Training of the baseline PINN for the same amount of iterations was not successful, as the L-BFGS optimizer failed consistently to converge. Notably, SA-PINN appears to stabilize the training process, allowing the L-BFGS optimizer to converge.

4.5. Self-adaptive weight training hyperparameters

We comment here on the choice of hyperparameter settings made in the experiments reported in the previous sections. In all experiments, a constant learning rate of 5e-3 for the gradient ascent of the self-adaptive learning rates, and Adam optimization was used. Better results could potentially be obtained by using learning rate scheduling.

Empirically, we observed that effective training strategies for self-adaptive PINNs tend to require smaller values of learning rate for the neural network weights (i.e., 1e-5), and larger values for the self-adaptive weights (i.e., 1e-3 to 1e-1).

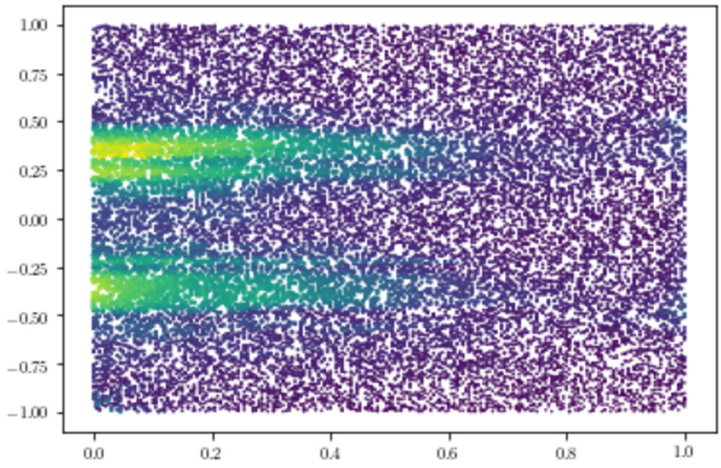

Fig. 9. Learned self-adaptive weights across the spatio-temporal domain. Brighter colors and larger points indicate larger weights.

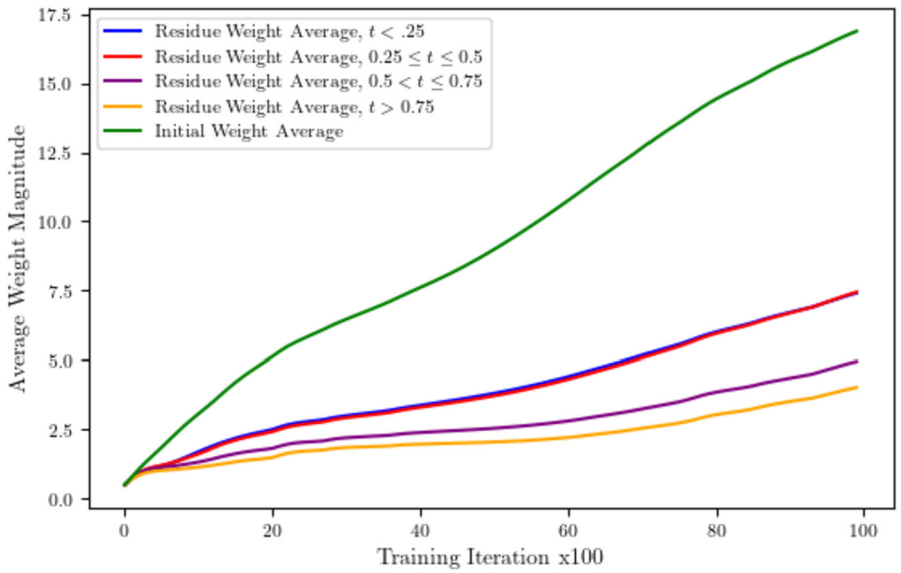

Fig. 10. Average learned residue weights across various partitions of the solution domain. Note that earlier times require heavier weighting, with the highest average weights being the initial condition weights. This is consistent with the rationale that earlier solutions must be correctly learned for time-diffusive processes.

As specified in Section 4.3, the results shown in Fig. 8 employ random initialization of the initial condition and residue self-adaptive weights in the intervals $[0, 100]$ and $[0, 1]$, respectively, and the learning rates are held constant and equal. This choice is dictated by the prior knowledge that heavier weighting of the initial condition is needed in the AC problem [4]. On the other hand, in the results displayed in Fig. 10, all weights were initialized randomly in the interval $[0, 1]$. In this case, it is observed that the initial conditions increase faster on their own, even though all learning rates are held constant and equal.

5. Self-adaptive PINNs with stochastic gradient descent

Stochastic gradient descent (SGD) [31] uses randomly sampled subsets of the training data to compute approximations to the gradient for training neural networks by gradient descent [32]. It has been claimed that the empirical superior performance of stochastic gradient descent over large-batch training is due to a tendency of the latter to converge to "sharp" minima in the loss surface, which have poor performance, while SGD with small batches converge to better "flat" minima [33].

The issue has not been well studied in the context of PINNs at the time of writing, though there is some empirical evidence that SGD can indeed improve the L_2 performance of PINNs with some PDEs. It should be pointed out that PINNs

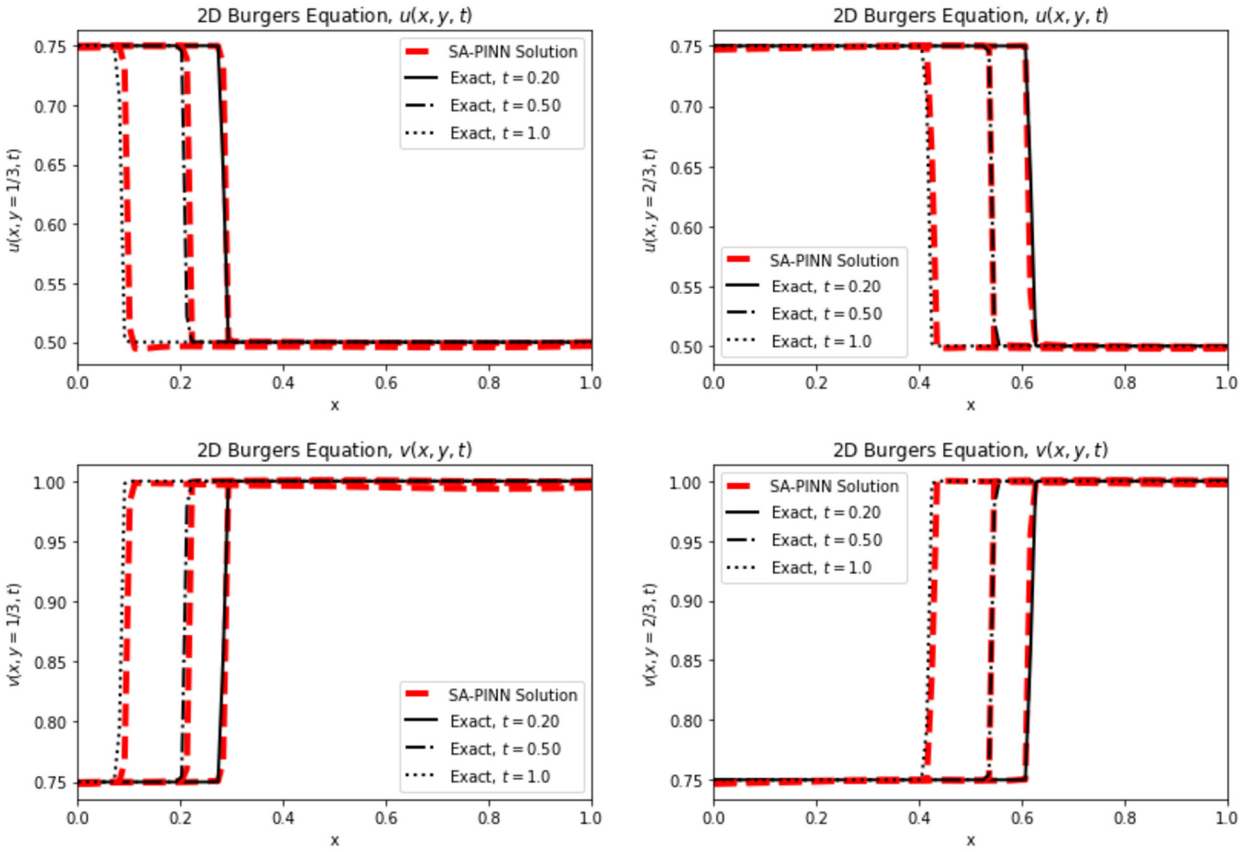

Fig. 11. SA-PINN solution of 2D Burgers equation example. The L2 error for both u and v at $t = 1/3$ and $t = 2/3$ is approximately $3e-3$.

are well-suited to SGD since a new set of residue, initial and boundary points can be sampled each time rather than subsampling a given set of training data points as in conventional machine learning.

The baseline SA-PINN algorithm described previously cannot take advantage of small-batch SGD since the self-adaptive weights are attached to specific training points. In this section, we examine an extension of SA-PINN that allows the use of SGD. The basic idea is to use a spatial-temporal predictor of the value of self-adaptive weights for the newly sampled points. Here we use standard Gaussian process regression due to its predictive power. (However, simpler regression approaches could be equally used, in cases where GP regression is unwieldy, e.g., due to large sample size.)

A problem where SGD has been found empirically to have a strong impact is the 1D wave equation:

$$u_{tt}(x,t) - 4u_{xx}(x,t) = 0\,, \quad x \in [0,1]\,,\ t \in [0,1]\,, \tag{38}$$

$$u(0,t) = 0\,, u(1,t) = 0,\ t \in [0,1]\,, \tag{39}$$

$$u_t(x,0) = 0\,,\ x \in [0,1]\,, \tag{40}$$

$$u(x,0) = \sin(\pi x) + \frac{1}{2}\sin(4\pi x)\,,\ x \in [0,1]\,. \tag{41}$$

This problem was considered in [5] to study their NTK weighting scheme. The problem has an analytical solution:

$$u(x,t) = \sin(\pi x)\cos(2\pi t) + \frac{1}{2}\sin(4\pi x)\cos(8\pi t)\,, \quad x \in [0,1]\,,\ t \in [0,1]\,. \tag{42}$$

The baseline PINN struggles in this problem due to its stiffness. Here, we investigate the improvement provided by SGD, fixed weights, and self-adaptive weights. The architecture of the neural network consists of 5 layers of 500 neurons each with the tanh nonlinearity, and the number of residue, initial, and boundary points were set to 300, 100, and 100, respectively (these are the same hyperparameters used in [5]). The small sample sizes are appropriate to study the impact of SGD.

In all experiments, the learning rate for the neural network weights is kept fixed at 10^{-5} for a total of 80,000 iterations. The self-adaptive weights are all initialized to 1.0, with learning rate 0.01 for the residue points, 0.05 for the initial condition on u_t, and 0.25 for all other initial and boundary conditions on u. In the fixed-weight experiment, the weights were kept

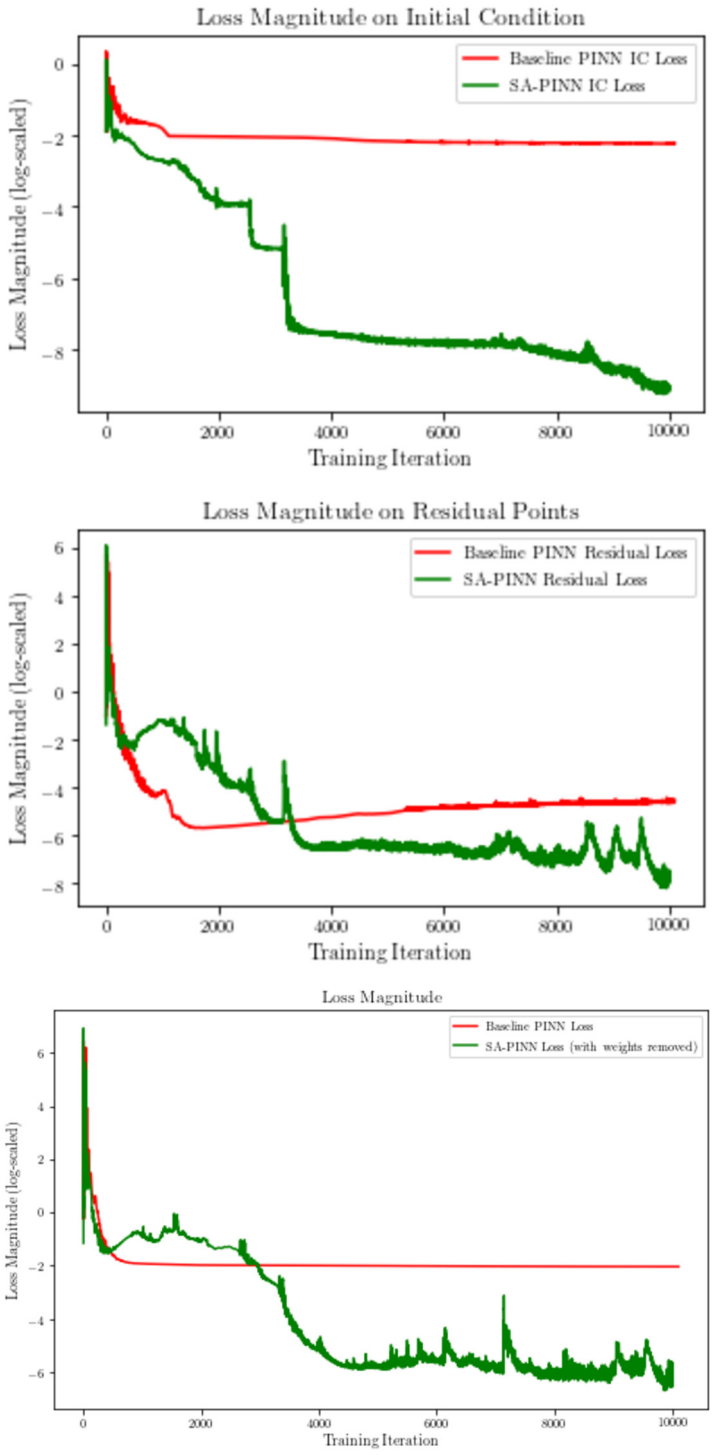

Fig. 12. Average values for the initial condition loss, residue loss, and total loss, over 10k Adam training iterations. For the SA-PINN, the weights were removed from the loss value to provide a direct comparison to the baseline.

constant at 1.0 for the residue points, 5.0 for the initial condition on u_t, and 50.0 for all other initial and boundary conditions on u. These values make the final average values taken by the self-adaptive weights at the end of training approximately match the fixed weights. SGD is applied by resampling all training points every 100 iterations. The GPs were trained using fixed hyperparameters (no automatic tuning is performed).

Table 1
Wave PDE results. The L2 error mean and standard deviation are based on 10 independent runs. The training time is an average over the 10 runs.

PINN method	No SGD		SGD	
	L2 error	time (sec)	L2 error	time (sec)
baseline	0.3792 ± 0.0162	879.97	0.4513 ± 0.0255	1057.04
fixed weights	0.7296 ± 0.1421	850.75	0.2079 ± 0.0624	1012.06
self-adaptive	0.8105 ± 0.1591	961.27	$\mathbf{0.0295 \pm 0.0070}$	1207.85

Results based on 10 independent random initializations of the neural network weights are displayed in Table 1. We can observe that all methods fail in the absence of SGD. On the other hand, while SGD is not able to improve the performance of the baseline PINN, it produces a significant improvement to the fixed-weight PINN, and a large improvement to the SA-PINN. In fact, the SA-PINN achieves an average L2-error of 2.95%, which is an order of magnitude better than the fixed-weight result. This L2-error is however larger than the one reported in [5]. Optimizations to the SGD process, including adaptive tuning of the GP hyperparameters, will be part of future work, and are expected to improve performance.

These results can perhaps be better appreciated in the plots in Fig. 13. As an extra feature, the Gaussian Process predictor produces a *continuous* self-adaptive weight map. Fig. 14 displays the self-adaptive weight GP maps for the initial condition and residue points. We can observe in the 1D map that the self-adaptive weights become larger at the (high and low) peaks of the initial condition $u(x, 0)$, which is where the curvature is maximum in magnitude; these are the most difficult regions to approximate with the neural network. In the 2D map, we can see that the self-adaptive weights are larger in the initial time, once again indicating the importance of approximating the solution early in time-evolution problems.

6. Neural tangent kernel analysis of self-adaptive PINNs

In this section, we investigate the dynamics of SA-PINN training by studying its neural tangent kernel (NTK). We derive the expression for the NTK matrix for self-adaptive PINNs and then use it to obtain a heuristic understanding of the effect of the self-adaptive weights on the dynamics of training in the limiting case of infinitely-wide PINNs. We examine the effect of the self-adaptive weights on the eigenvalues of the NTK matrix in the solution of a linear advection PDE.

First, note that (14) can be written as

$$\frac{\boldsymbol{w}^{k+1} - \boldsymbol{w}^k}{\eta_k} = -\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k)\,. \tag{43}$$

In the limit as the learning rate η_k tends to zero, the previous expression yields the *gradient flow* differential equation [34]:

$$\frac{d\boldsymbol{w}(\tau)}{d\tau} = -\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}(\tau), \boldsymbol{\lambda}_r(\tau), \boldsymbol{\lambda}_b(\tau), \boldsymbol{\lambda}_0(\tau))\,, \tag{44}$$

where $\tau \geq 0$ denotes the (continuous) training time. Notice that the usual gradient descent step corresponds to a forward Euler discretization of (44). It follows that the properties of gradient descent optimization can be investigated by studying this differential equation.

Under this vanishing learning-rate limit, the *neural tangent kernel* (NTK) [35] characterizes the training dynamics of the neural network, i.e., the evolution of the output $u(\boldsymbol{x}, t; \boldsymbol{w}(\tau))$ as a function of training time τ. In [5], the NTK for PINNs was derived and its properties were studied. Here we show how that a simple modification to their derivation produces the NTK for SA-PINNs.

For definiteness, consider the PDE problem:

$$\mathcal{N}_{\boldsymbol{x}}[u(\boldsymbol{x})] = f(\boldsymbol{x}, t)\,, \quad \boldsymbol{x} \in \Omega\,, \tag{45}$$

$$u(\boldsymbol{x}) = g(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Gamma \subseteq \partial\Omega\,. \tag{46}$$

For a time-evolution problem, t becomes one of the components of $\boldsymbol{x}$, and the set Γ typically includes an initial condition at $t = 0$. More complex boundary conditions and sample data can be added to the analysis below in a straightforward way.

Given residue points $\{\boldsymbol{x}_r^i\}_{i=1}^{N_r}$ and boundary condition points $\{\boldsymbol{x}_b^i\}_{i=1}^{N_b}$, let the response vectors be

$$\mathbf{u}_r(\tau) = [N_x[u(\boldsymbol{x}_r^1; \boldsymbol{w}(\tau))], \ldots, N_x[u(\boldsymbol{x}_r^{N_r}; \boldsymbol{w}(\tau))]]^T, \tag{47}$$

$$\mathbf{u}_b(\tau) = [u(\boldsymbol{x}_b^1; \boldsymbol{w}(\tau)), \ldots, u(\boldsymbol{x}_r^{N_b}; \boldsymbol{w}(\tau))]^T. \tag{48}$$

Likewise, the data vectors are denoted by

$$\mathbf{v}_r = [f(\boldsymbol{x}_r^1), \ldots, f(\boldsymbol{x}_r^{N_r})]^T, \tag{49}$$

$$\mathbf{v}_b = [g(\boldsymbol{x}_b^1), \ldots, g(\boldsymbol{x}_b^{N_b})]^T. \tag{50}$$

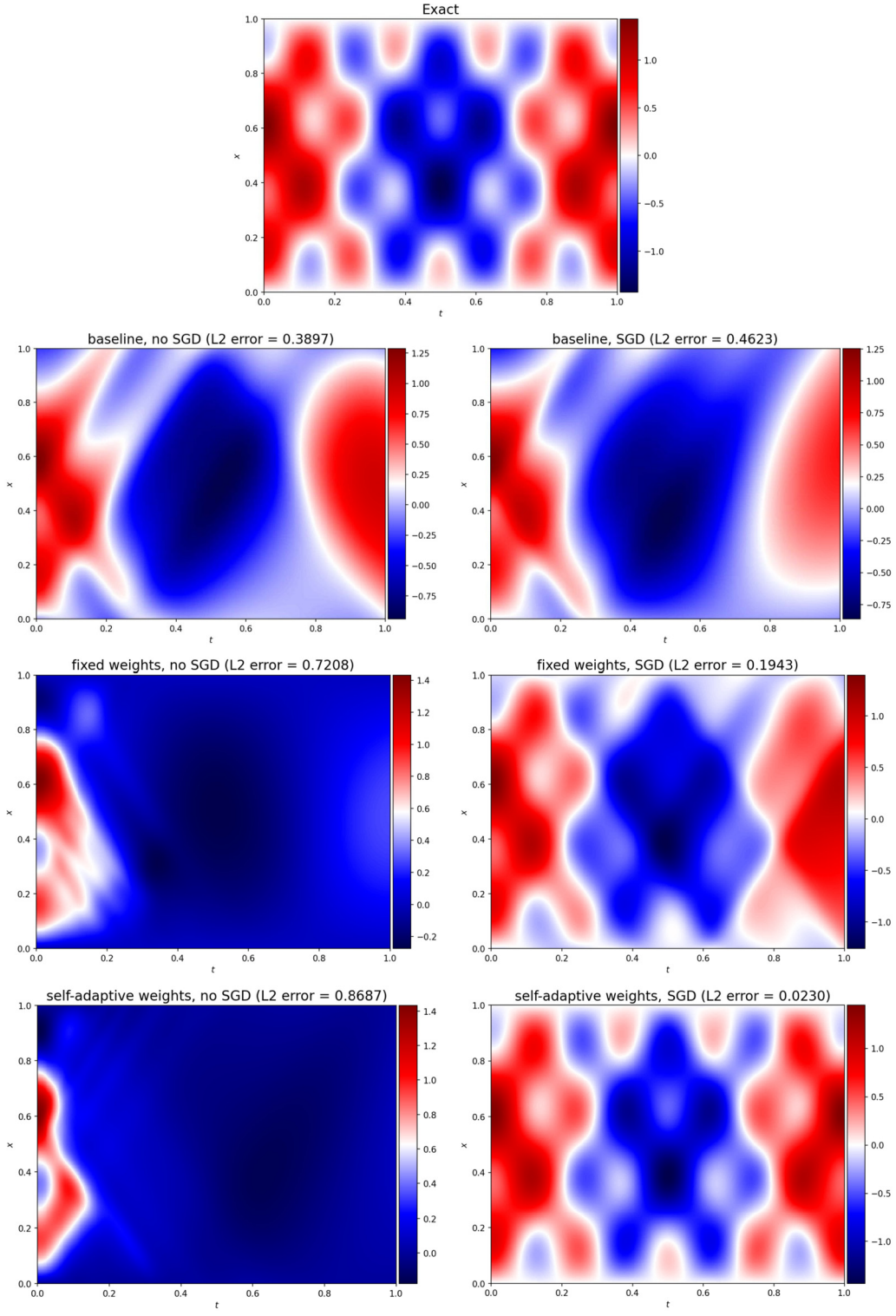

Fig. 13. *Top:* Exact solution of the wave problem. *Left:* Approximations obtained without SGD. *Right:* Approximations with SGD. From top to bottom: baseline, fixed weights, and self-adaptive weights.

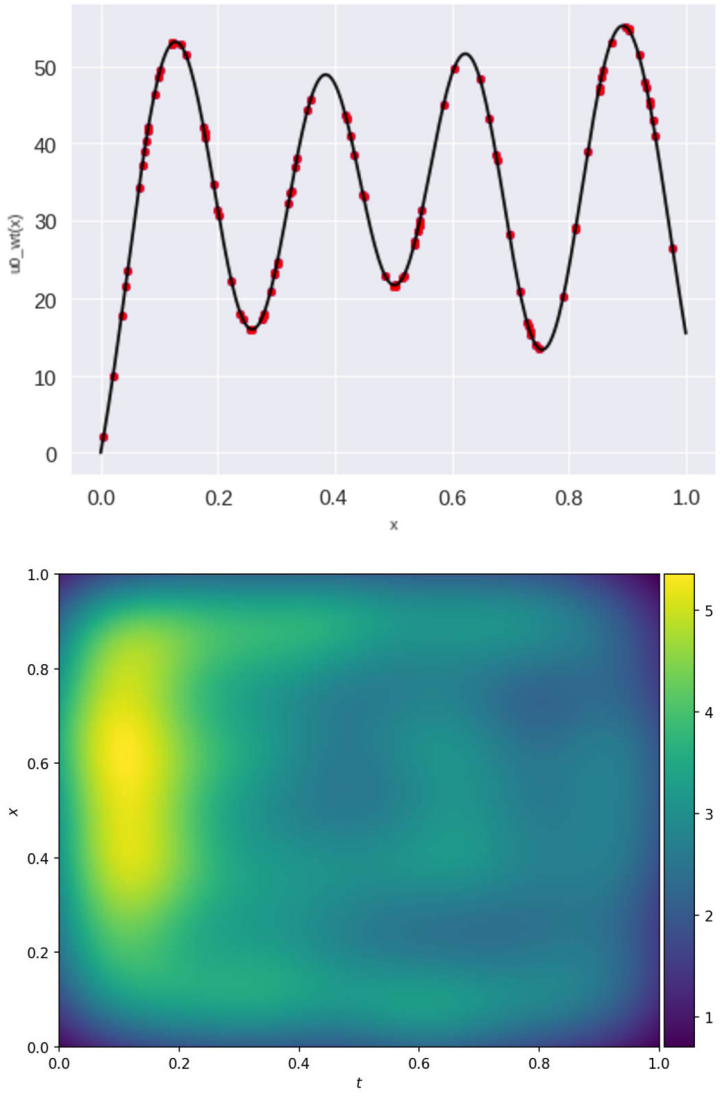

Fig. 14. Gaussian-Process maps of self-adaptive weights. *Top:* 1D map of initial condition weights. The red dots indicate the values of the weights at the actual data locations. *Bottom:* 2D map of PDE residue weights.

We write $\mathbf{u}_p(\tau) = (u_p^1(\tau), \ldots, u_p^{N_p}(\tau))$ and $\mathbf{v}_p = (v_p^1, \ldots, v_p^{N_p})$ to identify the individual responses $u_p^i(\tau)$ and data point v_p^i, for $p = r, b$.

The loss function at training time τ can be written similarly to (11)–(13):

$$\mathcal{L}(\boldsymbol{w}(\tau), \boldsymbol{\lambda}_r(\tau), \boldsymbol{\lambda}_b(\tau)) = \frac{1}{2} \sum_{q=r,b} \sum_{j=1}^{N_q} m(\lambda_q^j(\tau)) \, |u_q^j(\tau) - v_q^j|^2 \tag{51}$$

Hence, the gradient flow in (44) becomes

$$\frac{d\mathbf{w}}{d\tau} = -\sum_{q=r,b} \sum_{j=1}^{N_q} \nabla_{\boldsymbol{w}} u_q^j(\tau) m(\lambda_q^j(\tau)) \, (u_q^j(\tau) - v_q^i) \tag{52}$$

$$= -\sum_{q=r,b} \mathbf{J}_q^T(\tau) \boldsymbol{\Gamma}_q(\tau) (\mathbf{u}_q(\tau) - \mathbf{v}_q) \tag{53}$$

where $\mathbf{J}_q(\tau)$ is the Jacobian of $\mathbf{u}_q(\tau)$ with respect to $\boldsymbol{w}$, for $q = r, b$, and $\boldsymbol{\Gamma}_q(\tau)$ is a diagonal matrix of dimension $N_q \times N_q$ containing the self-adaptive mask values $m(\lambda_q^1(\tau)), \ldots, m(\lambda_q^{N_q}(\tau))$ in the diagonal, for $q = r, b$.

It follows that

$$\frac{d\mathbf{u}_p(\tau)}{d\tau} = \mathbf{J}_p(\tau)\cdot\frac{d\mathbf{w}(\tau)}{d\tau} = -\sum_{q=r,b}\mathbf{J}_p(\tau)\mathbf{J}_q^T(\tau)\mathbf{\Gamma}_q(\tau)(\mathbf{u}_q(\tau)-\mathbf{v}_q)\,, \tag{54}$$

for $p=r,b$.

Now define $\mathbf{K}_{pq}(\tau)=\mathbf{J}_p(\tau)\mathbf{J}_q^T(\tau)$, for $p,q=r,b$. Notice that these are matrices of dimensions $N_p\times N_q$, with i,j elements

$$\left(\mathbf{K}_{pq}\right)_{ij}(\tau) = \nabla_{\boldsymbol{w}}u_p^i(\tau)^T\cdot\nabla_{\boldsymbol{w}}u_q^j(\tau) = \sum_{w\in\mathbf{w}}\frac{du_p^i(\tau)}{dw}\cdot\frac{du_q^j(\tau)}{dw}\,. \tag{55}$$

It is clear from the definition that the matrices $\mathbf{K}_{pp}(\tau)$ are symmetric and positive semi-definite, and that $\mathbf{K}_{pq}(\tau)=\mathbf{K}_{qp}(\tau)^T$, for $p,q=r,b$.

This allows us to collect the previous results in the following differential equation describing the evolution of the output of the SA-PINN in the vanishing learning-rate limit:

$$\frac{d\mathbf{u}(\tau)}{d\tau} = -\mathbf{K}(\tau)\cdot(\mathbf{u}(\tau)-\mathbf{v})\,, \tag{56}$$

where

$$\mathbf{u}(\tau) = \begin{bmatrix}\mathbf{u}_r(\tau)\\ \mathbf{u}_b(\tau)\end{bmatrix}, \qquad \mathbf{v} = \begin{bmatrix}\mathbf{v}_r\\ \mathbf{v}_b\end{bmatrix}, \tag{57}$$

and

$$\mathbf{K}(\tau) = \begin{bmatrix}\mathbf{K}_{rr}(\tau)\mathbf{\Gamma}_r(\tau) & \mathbf{K}_{rb}(\tau)\mathbf{\Gamma}_b(\tau)\\ \mathbf{K}_{br}(\tau)\mathbf{\Gamma}_r(\tau) & \mathbf{K}_{bb}(\tau)\mathbf{\Gamma}_b(\tau)\end{bmatrix} \tag{58}$$

is the *empirical neural tangent kernel* matrix for the SA-PINN. (When all the mask values are 1, this reduces to the expression in Lemma 3.1 of [5].)

Next, we employ the previous result to perform a heuristic analysis of the self-adaptive weights through their effects in the gradient flow ODE system in (56). The analysis is based on the infinite-width limit of neural networks, when it can be shown that, under the vanishing learning rate regime, the NTK matrix converges to a constant deterministic value throughout training [35]. Under certain regularity conditions, it is shown in [5] that this result still holds in the case of PINNs, i.e., the matrices $\mathbf{K}_{rr}(\tau)$, $\mathbf{K}_{rb}(\tau)=\mathbf{K}_{br}^T(\tau)$ and $\mathbf{K}_{bb}(\tau)$ are constant and equal to their respective values at initialization ($\tau=0$) throughout training. In [5], this was proved for PINNs with one hidden layer and linear PDEs, though the authors conjectured that this result also holds for multiple-layer PINNs and nonlinear PDEs.

We thus make the assumption that for a wide PINN under a small learning rate, the NTK matrix and self-adaptive weights change little during training, i.e., $\mathbf{K}_{pq}(\tau)\approx\mathbf{K}_{pq}$ and $\mathbf{\Gamma}_p(\tau)\approx\mathbf{\Gamma}_p$, for $p,q=r,b$, $\tau\geq 0$. In addition, we make the simplifying approximation that the ODE system (56) can be decoupled, so that

$$\frac{d\mathbf{u}_p(\tau)}{d\tau} \approx -\mathbf{K}_{pp}\mathbf{\Gamma}_p\cdot(\mathbf{u}_p(\tau)-\mathbf{v})\,, \tag{59}$$

for $p=r,b$. (This approximation is also made, implicitly, in Section 7.3 of [5].) Some justification for the decoupling approximation comes from empirical evidence (not shown) that the matrix norms of the cross-terms $\mathbf{K}_{rb}\mathbf{\Gamma}_b$ and $\mathbf{K}_{br}\mathbf{\Gamma}_r$ are smaller than those of $\mathbf{K}_{rr}\mathbf{\Gamma}_r$ and $\mathbf{K}_{bb}\mathbf{\Gamma}_b$ and, in some cases, much smaller. This approximation allows us to gain a qualitative understanding of the importance of the residual and boundary loss components separately.

For $p=r,b$, matrix $\mathbf{K}_{pp}$ is real symmetric and positive semi-definite, and thus diagonalizable with nonnegative eigenvalues. However, matrix $\mathbf{K}_{pp}\mathbf{\Gamma}_p$ is not symmetric and not diagonalizable, in general. Fortunately, with the extra minor assumption that $\mathbf{K}_{pp}$ is positive definite, and thus invertible, $\mathbf{K}_{pp}\mathbf{\Gamma}_p$ is diagonalizable. To see this, note that

$$\mathbf{K}_{pp}^{-\frac{1}{2}}\mathbf{K}_{pp}\mathbf{\Gamma}_p\mathbf{K}_{pp}^{\frac{1}{2}} = \mathbf{K}_{pp}^{\frac{1}{2}}\mathbf{\Gamma}_p\mathbf{K}_{pp}^{\frac{1}{2}}\,. \tag{60}$$

But $\mathbf{K}_{pp}^{\frac{1}{2}}\mathbf{\Gamma}_p\mathbf{K}_{pp}^{\frac{1}{2}}$ is a product of symmetric matrices, and thus symmetric itself. Hence, $\mathbf{K}_{pp}\mathbf{\Gamma}_p$ is similar to a real symmetric matrix, and thus diagonalizable. Furthermore, it is fairly simple fact of matrix theory that if $\gamma_p^1\geq\cdots\geq\gamma_p^{N_p}$ and $\mu_p^1\geq\cdots\geq\mu_p^{N_p}$ are the ordered eigenvalues of $\mathbf{K}_{pp}$ and $\mathbf{K}_{pp}\mathbf{\Gamma}_p$, respectively, and $\lambda_p^1,\geq\cdots\geq\lambda^{N_p}$ are the self-adaptive weights sorted by magnitude, then

$$\mu_p^1 \leq m(\lambda_1)\,\gamma_p^1\,, \tag{61}$$

$$\mu_p^n \geq m(\lambda_n)\,\gamma_p^n\,. \tag{62}$$

In particular, (62) implies that all eigenvalues of $\mathbf{K}_{pp}\mathbf{\Gamma}_p$ are nonnegative (and positive, if $\mathbf{K}_{pp}$ is positive definite).

It follows that, under the assumption that $\mathbf{u}_p(0) \approx \mathbf{0}$ (this can be achieved with proper initialization of the neural network weights), the solution of the ODE (59) is given by

$$\mathbf{u}_p(\tau) = (\mathbf{I} - e^{-\mathbf{K}_{pp}\mathbf{\Gamma}_p t}) \cdot \mathbf{v}_p \,, \tag{63}$$

which can be rewritten as

$$\mathbf{u}_p(\tau) - \mathbf{v}_p = -\mathbf{Q}^T e^{-\mathbf{M}t}\mathbf{Q} \cdot \mathbf{v}_p \,, \tag{64}$$

that is

$$\mathbf{Q} \cdot (\mathbf{u}_p(\tau) - \mathbf{v}_p) = -e^{-\mathbf{M}t}\mathbf{Q} \cdot \mathbf{v}_p \,, \tag{65}$$

where $\mathbf{Q}$ is the matrix of eigenvectors and $\mathbf{M}$ is the diagonal matrix of eigenvalues $\mu_p^1, \ldots, \mu_p^{N_p}$ of $\mathbf{K}_{pp}\mathbf{\Gamma}_p$, for $p = r, b$. This implies that the training error $u_p^i(\tau) - v_p^i$ decreases at a rate $e^{-mu_p^i}$ rate. Large variation among the eigenvalues $\mu_p^1, \ldots, \mu_p^{N_p}$, both across the different loss terms $p = r, b$ *and* the different data points in each loss term, will potentially lead to training imbalances and loss of convergence.

The standard weighted loss function in (9) corresponds to the case when all the self-adaptive weights for each loss component are equal, with $\mathbf{\Gamma}_p = \lambda_p I$, in which case the eigenvalues of the NTK matrix are simply scaled by λ_p: $\mu_p^i = \lambda_p^i \gamma_p^i$, for $i = 1, \ldots, N_p$. On the other hand, the transformation effected on the eigenvalues of the NTK matrix by the self-adaptive weights is nonlinear. In general, little can be said about it other than the transformed eigenvalues are in the interval determined by $m(\lambda_1)\gamma_p^1$ and $m(\lambda_n)\gamma_p^n$, as stated in (61)–(62). The simple linear scaling introduced by traditional weighting can certainly help reduce the imbalance among the various terms, but it is less flexible than the transformation introduced by the pointwise self-adaptive weights, which can also change the *shape* of the eigenvalue distribution.

Next, we illustrate this analysis with the classical univariate advection PDE: [36]:

$$q_t(x,t) + \bar{u} q_x(x,t) = 0 \,, \quad x \in [0, L] \,, \; t \in [0, T] \,, \tag{66}$$

$$u(0,t) = u(L,t) = 0, \; t \in [0, T] \,, \tag{67}$$

$$u(x,0) = g(x) \,, \; x \in [0, L] \,, \tag{68}$$

where $q(x,t)$ is for example the concentration of a tracer being transported in a fluid in a tube of length L, where $\bar{u} > 0$ is the fluid constant velocity. For simplicity, it is assumed that $g(x) = 0$ outside an interval in $[0, L]$, and that T is short enough that the Dirichlet boundary condition is satisfied. (This could be changed at the expense of more complex boundary conditions.) In this scenario, the problem has a simple solution:

$$q(x,t) = g(x - \bar{u}t) \,, \quad x \in [0, L] \,, \; t \in [0, T] \,, \tag{69}$$

i.e., the initial concentration profile is simply translated to the right at constant speed $\bar{u}$. Here, we adopt a fairly complex initial condition, containing several discontinuities, which makes the problem rather difficult to solve with the baseline PINN — or indeed numerical methods in general [36].

The results presented in Figs. 15, 16, and 17 are generated with a neural network architecture of [2, 400, 400, 400, 400, 1], trained for 10k Adam iterations with a neural network weight learning rate of 0.001 (hence, a wide PINN with a small learning rate, as required by the theory). The learning rate for all self-adaptive weight was set at 0.1. Glorot Normal initialization was utilized, and all training was completed in Tensorflow on a single V100 GPU with an average training time of 7 seconds for 10k iterations. At the end of 10k training iterations, the baseline PINN failed to grasp even the rough structure of the solution, while the SA-PINN was able to approximate the solution within 5% L2 error. (More accurate results could have obtained by using more training epochs and a decreasing learning rate schedule.)

An analysis of NTK eigenvalues similar to that performed in [5] is demonstrated in Fig. 17. (In this example, the boundary condition weights were fixed and equal to 1.0, and we disregarded this component in the analysis.) We can see that the eigenvalues become closely matched in scale between K_{uu} and K_{rr}, removing the imbalance between these two loss components and enabling convergence to the solution (here we are only looking at the initial and. Importantly, the shape of the eigenvalue distribution is also nicely equalized, as opposed to simply being scaled up as would be the case with traditional weighting of the entire loss component.

7. Conclusion

In this paper, we introduced Self-Adaptive Physics-Informed Neural Networks, a novel class of physics-constrained neural networks. This approach uses a similar conceptual framework as soft self-attention mechanisms in computer vision, in that the network identifies which inputs are most important to its own training. It was shown that training of the SA-PINN is formally equivalent to solving a PDE-constrained optimization problem using penalty-based method, though in a way where

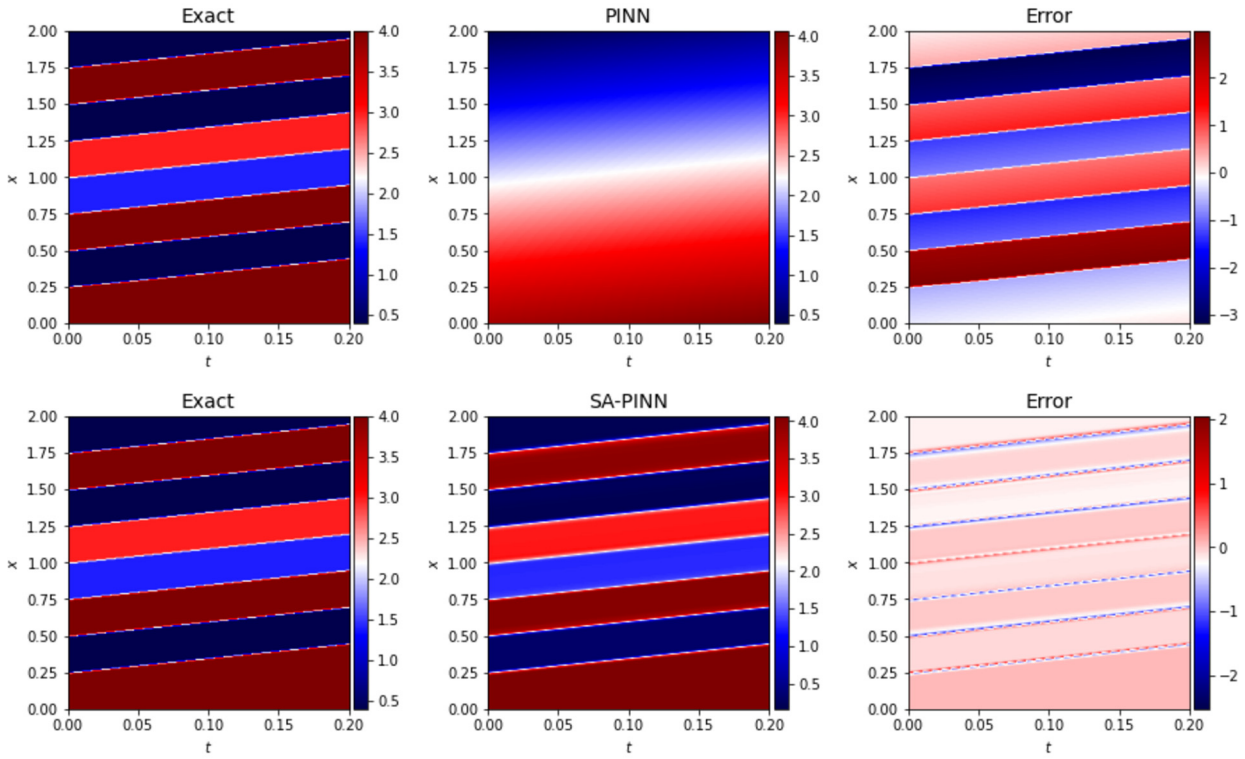

Fig. 15. *Top:* Plot of the approximation $u(x,t)$ via the baseline PINN, showing the exact solution vs predicted solution vs absolute error. *Bottom:* The SA-PINN results, L2 error decreases by an order of magnitude and the SA-PINN closely captures the exact solution.

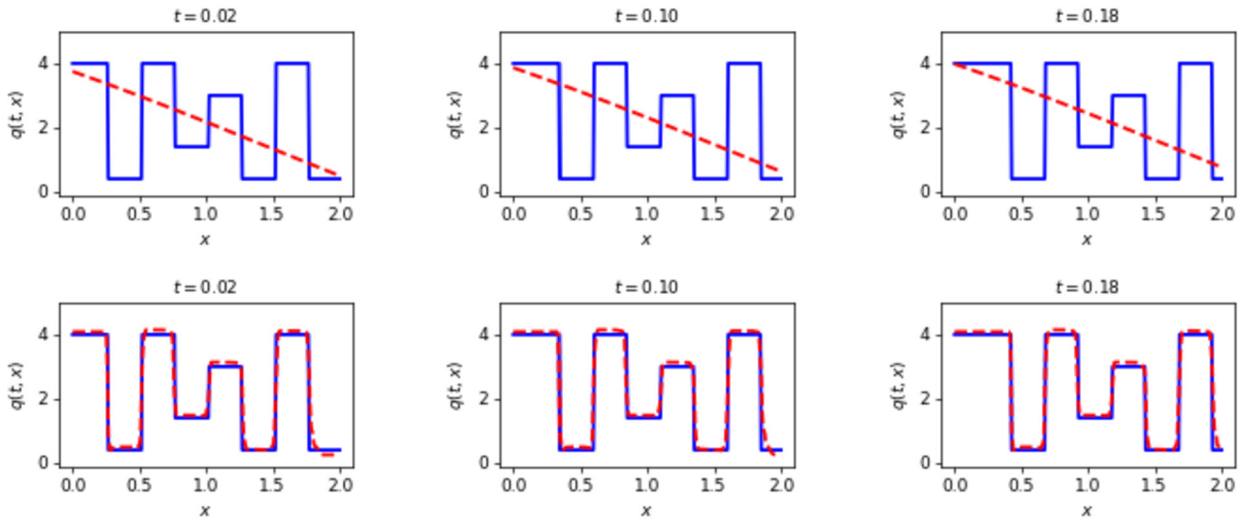

Fig. 16. *Top:* Plot of the approximation $u(x,t)$ via the baseline PINN, showing cross sections of the spatial domain at $t = 0.02, 0.10, 0.18$ *Bottom:* The SA-PINN results at the same time steps, with the same number of epochs (10k Adam) and all other parameters held constant.

the monotonically-nondecreasing penalty coefficients are trainable. Experimental results with several linear and nonlinear PDE benchmarks indicate that SA-PINNs produce more accurate solutions than other state-of-the-art PINN algorithms. It was seen that SA-PINNs can employ stochastic gradient training, through continuous Gaussian-process interpolated self-adaptive maps, which allows the solution of a difficult wave PDE. These experimental results were complemented by a theoretical analysis based on the Neural Tangent Kernel for SA-PINNs.

We believe that SA-PINNs open up new possibilities for the use of deep neural networks in forward and inverse modeling in engineering and science. However, there is much that is not known yet about this class of algorithms. While the empirical results observed here show significant promise, and despite the fact that an initial analysis based on the NTK is provided,

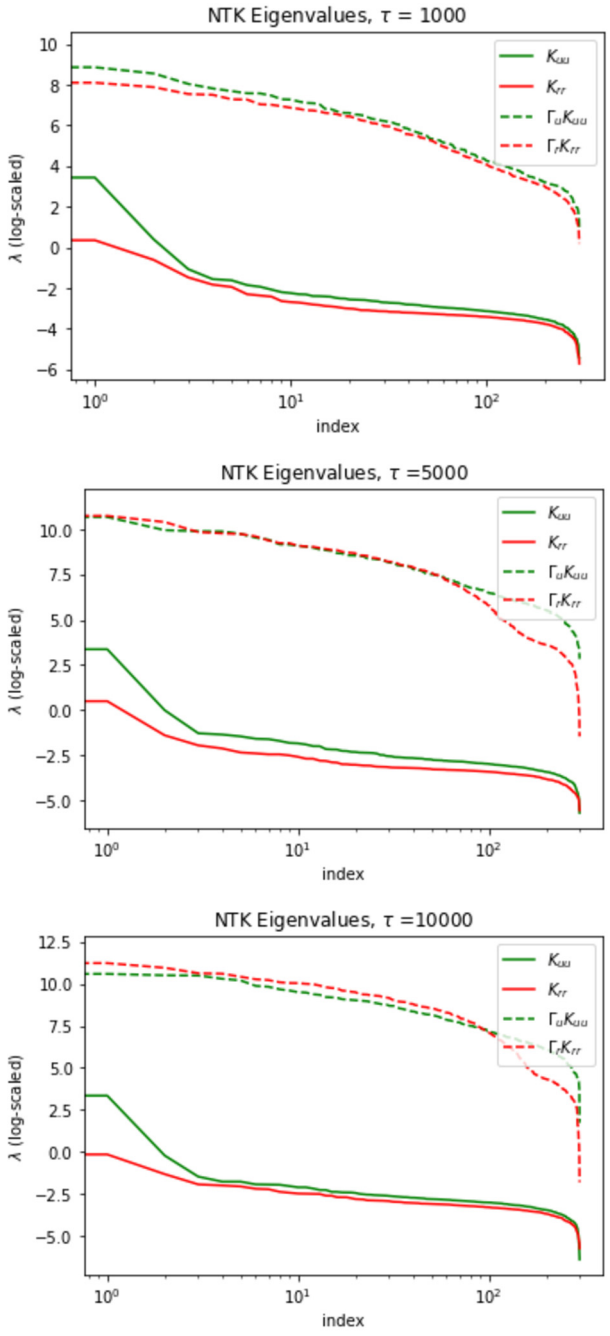

Fig. 17. NTK eigenvalues of the baseline PINN (solid) vs. the SA-PINN (dashed) for τ =1000, 5000, and 10000 training iterations. It can be observed that the SA-PINN accurately matches the magnitudes of the NTK eigenvalues between terms of the loss function, in this case the initial condition K_{uu} and the residual loss K_{rr}.

more theoretical justification is desirable, which will be part of future work. In addition, the use of standard off-the-shelf optimization algorithms for training deep neural networks, such as Adam, may not be appropriate, since those algorithms were mostly developed for traditional deep learning applications; obtaining optimization algorithms specifically tailored to Self-Adaptive PINNs, and indeed PINNs in general, in an open problem. Finally, the relationship between Self-Adaptive PINNs and constrained-optimization problems is likely a fruitful topic of future study.

CRediT authorship contribution statement

Levi McClenny co-authored the idea of SA-PINNs, co-wrote the manuscript, wrote most of the code, to include the proposed SA-PINN code/algorithm, and performed the bulk of the numerical experiments presented in the paper.

Ulisses Braga-Neto co-authored the idea of SA-PINNs, co-wrote the manuscript, performed most of the mathematical analyses presented in the paper, to include deriving the modifications to the PINN NTK based on the addition of self-adaptive weights and the mathematical analysis of the SA-PINN optimization, as well as provided commentary in the manuscript and numerical experiments in an advisory capacity. All authors read and approved the manuscript.

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

No data was used for the research described in the article.

Acknowledgements

The authors would like to acknowledge the support of the D^3EM program funded through NSF Award DGE-1545403. The authors would further like to thank the US Army CCDC Army Research Lab (CA number W911NF-16-2-0008) for their generous support and affiliation.

References

[1] Nathan Baker, Frank Alexander, Timo Bremer, Aric Hagberg, Yannis Kevrekidis, Habib Najm, Manish Parashar, Abani Patra, James Sethian, Stefan Wild, Karen Willcox, Steven Lee, Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence, 2 2019.
[2] 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.
[3] Maziar Raissi, Forward-backward stochastic neural networks: deep learning of high-dimensional partial differential equations, arXiv preprint arXiv:1804.07010, 2018.
[4] Colby L. Wight, Jia Zhao, Solving Allen-Cahn and Cahn-Hilliard equations using the adaptive physics informed neural networks, arXiv preprint arXiv:2007.04542, 2020.
[5] Sifan Wang, Xinling Yu, Paris Perdikaris, When and why pinns fail to train: a neural tangent kernel perspective, arXiv preprint arXiv:2007.14527, 2020.
[6] M.W.M.G. Dissanayake, N. Phan-Thien, Neural-network-based approximations for solving partial differential equations, Commun. Numer. Methods Eng. 10 (3) (1994) 195–201.
[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., Tensorflow: a system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 265–283.
[8] Jarrett Revels, Miles Lubin, Theodore Papamarkou, Forward-mode automatic differentiation in Julia, arXiv preprint arXiv:1607.07892, 2016.
[9] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind, Automatic differentiation in machine learning: a survey, J. Mach. Learn. Res. 18 (1) (2017) 5595–5637.
[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, Adam Lerer, Automatic Differentiation in Pytorch, 2017.
[11] Richard L. Burden, Douglas J. Faires, Numerical Analysis, 1985.
[12] Sifan Wang, Yujun Teng, Paris Perdikaris, Understanding and mitigating gradient pathologies in physics-informed neural networks, arXiv preprint arXiv:2001.04536, 2020.
[13] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, Xiaoou Tang, Residual attention network for image classification, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 3156–3164.
[14] Yanwei Pang, Jin Xie, Muhammad Haris Khan, Rao Muhammad Anwer, Fahad Shahbaz Khan, Ling Shao, Mask-guided attention network for occluded pedestrian detection, in: Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 4967–4975.
[15] Yeonjong Shin, Jerome Darbon, George Em Karniadakis, On the convergence and generalization of physics informed neural networks, arXiv preprint arXiv:2004.01806, 2020.
[16] Kejun Tang, Xiaoliang Wan, Qifeng Liao, Adaptive deep density approximation for Fokker-Planck equations, J. Comput. Phys. 457 (2022) 111080.
[17] Xiaodong Feng, Li Zeng, Tao Zhou, Solving time dependent Fokker-Planck equations via temporal normalizing flow, arXiv preprint arXiv:2112.14012, 2021.
[18] Dehao Liu, Yan Wang, A dual-dimer method for training physics-constrained neural networks with minimax architecture, Neural Netw. 136 (2021) 112–125.
[19] Softadapt: Techniques for adaptive loss weighting of neural networks with multi-part loss functions, arXiv preprint arXiv:1912.12355, 2019.
[20] Conrado Silva Miranda, Fernando José Von Zuben, Multi-objective optimization for self-adjusting weighted gradient in machine learning tasks, arXiv preprint arXiv:1506.01113, 2015.
[21] Haowen Xu, Hao Zhang, Zhiting Hu, Xiaodan Liang, Ruslan Salakhutdinov, Eric Xing, Autoloss: learning discrete schedules for alternate optimization, arXiv preprint arXiv:1810.02442, 2018.
[22] Michael Emmerich, André H. Deutz, A tutorial on multiobjective optimization: fundamentals and evolutionary methods, Nat. Comput. 17 (3) (2018) 585–609.
[23] Diederik P. Kingma, Jimmy Ba, Adam: a method for stochastic optimization, arXiv preprint arXiv:1412.6980, 2014.
[24] Dong C. Liu, Jorge Nocedal, On the limited memory bfgs method for large scale optimization, Math. Program. 45 (1–3) (1989) 503–528.

[25] Levi D. McClenny, Mulugeta A. Haile, Ulisses M. Braga-Neto, Tensordiffeq: scalable multi-gpu forward and inverse solvers for physics informed neural networks, arXiv preprint arXiv:2103.16034, 2021.
[26] David G. Luenberger, Yinyu Ye, Linear and Nonlinear Programming, 3rd edition, Springer, 2008.
[27] Nele Moelans, Bart Blanpain, Patrick Wollants, An introduction to phase-field modeling of microstructure evolution, Calphad 32 (2) (2008) 268–294.
[28] Jie Shen, Xiaofeng Yang, Numerical approximations of Allen-Cahn and Cahn-Hilliard equations, Discrete Contin. Dyn. Syst., Ser. A 28 (4) (2010) 1669.
[29] Courtney Kunselman, Vahid Attari, Levi McClenny, Ulisses Braga-Neto, Raymundo Arroyave, Semi-supervised learning approaches to class assignment in ambiguous microstructures, Acta Mater. 188 (2020) 49–62.
[30] 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.
[31] Herbert Robbins, Sutton Monro, A stochastic approximation method, Ann. Math. Stat. (1951) 400–407.
[32] Sebastian Ruder, An overview of gradient descent optimization algorithms, arXiv preprint arXiv:1609.04747, 2016.
[33] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, Ping Tak, Peter Tang, On large-batch training for deep learning: generalization gap and sharp minima, arXiv preprint arXiv:1609.04836, 2016.
[34] Andrew M. Saxe, James L. McClelland, Surya Ganguli, Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, arXiv preprint arXiv:1312.6120, 2013.
[35] Arthur Jacot, Franck Gabriel, Clément Hongler, Neural tangent kernel: convergence and generalization in neural networks, Adv. Neural Inf. Process. Syst. 31 (2018).
[36] Randall J. LeVeque, et al., Finite Volume Methods for Hyperbolic Problems, vol. 31, Cambridge University Press, 2002.