GPT-PINN: Generative Pre-Trained Physics-Informed Neural Networks toward non-intrusive Meta-learning of parametric PDEs*

Yanlai Chen†, Shawn Koohy‡

Abstract

Physics-Informed Neural Network (PINN) has proven itself a powerful tool to obtain the numerical solutions of nonlinear partial differential equations (PDEs) leveraging the expressivity of deep neural networks and the computing power of modern heterogeneous hardware. However, its training is still time-consuming, especially in the multi-query and real-time simulation settings, and its parameterization often overly excessive. In this paper, we propose the Generative Pre-Trained PINN (GPT-PINN) to mitigate both challenges in the setting of parametric PDEs. GPT-PINN represents a brand-new meta-learning paradigm for parametric systems. As a network of networks, its outer-/meta-network is hyper-reduced with only one hidden layer having significantly reduced number of neurons. Moreover, its activation function at each hidden neuron is a (full) PINN pre-trained at a judiciously selected system configuration. The meta-network adaptively "learns" the parametric dependence of the system and "grows" this hidden layer one neuron at a time. In the end, by encompassing a very small number of networks trained at this set of adaptively-selected parameter values, the meta-network is capable of generating surrogate solutions for the parametric system across the entire parameter domain accurately and efficiently.

1   Introduction

The need to efficiently and accurately understand the behavior of the system under variation of a large number of underlying parameters is ubiquitous in *many query* type of applications e.g. uncertainty quantification, (Bayesian) inverse problems, data assimilation or optimal control/design. The parameters of interest may include material properties, wave frequencies, uncertainties, boundary conditions, the shape of the domain, etc. A rigorous study of the behavior of the system and its dependence on the parameters requires thousands, perhaps millions of simulations of the underlying partial differential equations (PDE). Each accurate and robust simulation of the underlying complex physical phenomena is often time consuming, and the massively repeated simulations needed become computationally challenging, if not entirely untenable, when using traditional numerical methods. Two techniques stand out in addressing this challenge, the more traditional and rigorous reduced order modeling and the more nascent deep neural networks.

The reduced basis method (RBM) [29, 37, 17, 43, 22, 15], a projection-based model order reduction approach [4], belongs to the first category. It was developed to generate a computational

emulator for parameterized problems whose error compared to the full problem is certifiable; this rigorous accuracy guarantee is a relatively unique ability among reduced order model algorithms. Once generated, the RBM emulator, using the results of the original method at carefully prese-lected parameter values, can typically compute an accurate solution with orders-of-magnitude less computational cost than the original method. This is achieved through an offline-online decomposition, where the parameter values are selected and reduced solution space constructed in an offline preparation/training phase via a greedy algorithm, allowing the rapid online computation of the surrogate solution for any parameter values.

Deep learning algorithms are increasingly popular in this context as well. Using data generated by many queries of the underlying system, one can train a deep neural network (DNN, a highly nonlinear function composed of layers of parameterized affine linear functions and simple nonlinear operations) to provide a surrogate for the parameter to solution map. Physics-informed neural networks (PINNs), popularized by [40], adopt DNNs to represent the approximate solutions of PDEs. Unlike typical data-driven deep learning methods that do not build in physical understanding of the problem, PINNs incorporate a strong physics prior (i.e. PDEs) that constrains the output of the DNN. The key advantages of PINNs, over traditional numerical solvers, include that they are able to solve the PDE without discretizing the problem domain, that they define a function over the entire continuous spatial-temporal domain, and that they can rely on automatic differentiation [3, 28] toward residual minimization. Thanks also to the enormous advances in computational capabilities in recent years [1, 42], PINNs have emerged as an increasingly popular alternative to traditional numerical methods for PDEs.

However, issues of PINNs remain [45]. Among them, vanilla PINNs are usually significantly slower than the classic numerical methods due to the training of the, what is usually substantially parameterized, neural network. The main purpose of this paper is to use strategies inspired by the classical and mathematically rigorous RBM techniques to significantly shrink the size of PINNs and accelerate solving parametric PDEs with PINNs. Just like RBM, the proposed solvers have an initial investment cost. However, they are capable of providing significant computational savings in problems where a PDE must be solved repeatedly or in real-time thanks to the fact that their marginal cost is of orders of magnitude lower than that of each PINN solve.

The jump from the vanilla PINN to the proposed Generative Pre-Trained PINN (GPT-PINN) parallels that from the traditional Finite Element Method (FEM) to RBM. To the best of our knowledge, it represents a first-of-its-kind meta-learning approach for parametric systems. Its infrastructure is a network of networks. The inner networks are the full PINNs. Its outer-/meta-network is hyper-reduced, in comparison to the inner networks, with only one hidden layer where the inner networks are pre-trained and serve as activation functions. The meta-network adaptively "learns" the parametric dependence of the system and "grows" this hidden layer one neuron/network at a time. In the end, by encompassing a very small number of networks trained at this set of adaptively-selected parameter values, the meta-network is capable of generating surrogate solutions for the parametric system across the entire parameter domain accurately and efficiently, with a cost inde-pendent of the size of the full PINN. The design of network architecture represent the first main novelty of the paper. To the best of our knowledge, this is the first time whole (pre-trained) networks are used as the activation functions of another network. The adoption of the training loss of the meta-network as an error indicator, inspired by the residual-based error estimation for traditional numerical solvers such as FEM, represents the second main novelty.

The rest of the paper is organized as follows. In Section 2, we review the RBM and PINN. In Section 3, we detail our design of GPT-PINN, and also remark on recent efforts about accelerating PINNs in the parametric PDE setting. We present numerical results on three parametric PDEs in Section 4 demonstrating the accuracy and efficiency of the proposed GPT-PINN. Finally, concluding

remarks are given in Section 5.

2 Background

2.1 Reduced Basis Method

RBM is a linear reduction method that has been a popular option for rigorously and efficiently simulating parametric PDEs. Its hallmark feature is a greedy algorithm embedded in an offline-online decomposition procedure. The offline (i.e. training) stage is devoted to a judicious and error estimate-driven exploration of the parameter-induced solution manifold. It selects a number of representative parameter values via a mathematically rigorous greedy algorithm [5]. During the online stage, a *reduced* solution is sought in the terminal surrogate space for each unseen parameter value. Moreover, unlike other reduction techniques (e.g. proper orthogonal decomposition (POD)-based approaches), the number of full order inquiries RBM takes offline is minimum, i.e. equal to the dimension of the surrogate space. To demonstrate the main ideas, we consider a generic parameterized PDE as follows,

$$\mathcal{F}(u; \mathbf{x}, \boldsymbol{\mu}) = f, \quad x \in \Omega \subseteq \mathbb{R}^d. \tag{1}$$

Here $\mathcal{F}$ encodes a differential operator parameterized via $\boldsymbol{\mu} \in \mathcal{D} \subset \mathbb{R}^{d_s}$ together with necessary boundary and initial conditions. The parameter can be equation coefficients, initial values, source terms, or uncertainties in the PDE for the tasks of the uncertainty quantification, etc. $\mathcal{F}$ can depend on the solution and its (space- and time-) derivatives of various orders. We assume that we have available a numerical solution $u(\mathbf{x}; \boldsymbol{\mu}) \in X_h$ obtained by a high fidelity solver, called Full Order Model (FOM) and denoted as $\text{FOM}(\boldsymbol{\mu}, X_h)$, and X_h is the discrete approximation space the numerical solution u belongs to.

A large number of queries of $u(\cdot; \boldsymbol{\mu})$ can be prohibitively expensive because the $\text{FOM}(\boldsymbol{\mu}, X_h)$ has to be called many times. Model order reduction (MOR) aims to mitigate this cost by building efficient surrogates. One idea is to study the map

$$\boldsymbol{\mu} \mapsto u(\cdot, \boldsymbol{\mu}) \in X_h$$

and devise an algorithm to compute an approximation $u_N(\cdot, \mu)$ from an N-dimensional subspace X_N of X_h, such that

$$u_N(\cdot, \boldsymbol{\mu}) \approx u(\cdot, \boldsymbol{\mu}) \text{ for all } \boldsymbol{\mu} \in \mathcal{D}$$

This reduced order model (ROM) formulation at a given $\boldsymbol{\mu}$ is denoted by $\text{ROM}(\boldsymbol{\mu}, X_N)$, and is much cheaper to solve than $\text{FOM}(\boldsymbol{\mu}, X_h)$ and can be conducted during the Online stage.

Algorithm 1 Classical RBM for parametric PDE (1): Offline stage

---

Input: A (random or given) $\boldsymbol{\mu}^1$, training set $\Xi \subset \mathcal{D}$.
Initialization: Solve $\text{FOM}(\boldsymbol{\mu}^1, X_h)$ and set $X_1 = \text{span} \{u(\cdot; \boldsymbol{\mu}_1)\}$, $n = 2$.
 1: **while** *stopping criteria not met,* **do**
 2:     Solve $\text{ROM}(\boldsymbol{\mu}, X_{n-1})$ for all $\boldsymbol{\mu} \in \Xi$ and compute error indicators $\Delta_{n-1}(\boldsymbol{\mu})$.
 3:     Choose $\boldsymbol{\mu}^n = \underset{\boldsymbol{\mu} \in \Xi}{\arg\max} \, \Delta_{n-1}(\boldsymbol{\mu})$.
 4:     Solve $\text{FOM}(\boldsymbol{\mu}_n, X_h)$ and update $X_n = X_{n-1} \bigoplus \{u(\cdot; \boldsymbol{\mu}_n)\}$.
 5:     Set $n \leftarrow n + 1$.
 6: **end while**
Output: Reduced basis set X_N, with N being the terminal index.

The success of RBM relies on the assumption that $u(\cdot; \mathcal{D})$ has small *Kolmogorov N-width* [33], defined as

$$d_N \left[u \left(\cdot; \mathcal{D} \right) \right] := \inf_{\substack{X_N \subset X_h \\ \dim X_N = N}} \sup_{\boldsymbol{\mu} \in \mathcal{D}} \inf_{v \in X_N} \| u(\cdot, \boldsymbol{\mu}) - v \|_X .$$

A small d_N means that the solution to eq. (1) for any $\boldsymbol{\mu}$ can be well-approximated from X_N that represents the outer infimum above. The identification of a near-infimizing subspace X_N is one of the central goals of RBM, and is obtained in the so-called Offline stage. RBM uses a greedy algorithm to find such X_N. The main ingredients are presented in Algorithm 1. The method explores the training parameter set $\Xi \subset \mathcal{D}$ guided by an error estimate or an efficient and effective error indicator $\Delta_n(\boldsymbol{\mu})$ and intelligently choosing the parameter ensemble $\{\boldsymbol{\mu}^n\}_{n=1}^N$ so that

$$X_N := \mathrm{span} \left\{ u(\cdot; \boldsymbol{\mu}^n) \right\}_{n=1}^N, \text{ and } u_N(\cdot, \boldsymbol{\mu}) = \sum_{n=1}^N c_n(\boldsymbol{\mu}) u(\cdot, \boldsymbol{\mu}^n). \tag{2}$$

An offline-online decomposed framework is key to realize the speedup.

Equipped with this robust and tested greedy algorithm, physics-informed reduced solver, rigorous error analysis, and certifiable convergence guarantees, RBM algorithms have become the go-to option for efficiently simulating parametric PDEs and established in the modern scientific computing toolbox [34, 29, 37, 17] and have benefited from voluminous research with theoretical and algorithmic refinement [27, 32, 2, 26, 43, 22]. One particular such development was the empirical error indicator of the L1-based RBM by Chen and his collaborators [8] where $\Delta_{n-1}(\mu)$ was taken to be $\|\mathbf{c}(\mu)\|_1$. Here $\mathbf{c}(\mu)$ is the coefficient vector of $u_N(\cdot, \mu)$ under the basis $\{u(\cdot; \mu_n)\}_{n=1}^N$ and $\|\cdot\|_1$ represents the ℓ^1-norm. As shown in [8, 7], $\mathbf{c}(\mu)$ represents a Lagrange interpolation basis in the parameter space implying that the indicator Δ_n represents the corresponding Lebesgue constant. The L1 strategy to select the parameter samples then controls the growth of the Lebesgue constants and hence is key toward accurate interpolation. This strategy, "free" to compute albeit not as traditionally rigorous, inspires the greedy algorithm of our GPT-PINN, to be detailed in Section 3.

2.2 Deep neural networks

Deep neural networks (DNN) have seen tremendous success recently when serving as universal approximators to the solution function (or certain quantity of interest (QoI) / observable) [19, 31, 11, 18, 38, 9, 48, 44]. First proposed in [19] on an underlying collocation approach, it has been successfully used recently in different contexts. See [39, 41, 20, 25, 11, 10, 16] and references therein. For a nonparametrized version (e.g. eq. (1) with a fixed parameter value), we search for a neural network $\Psi_{\mathsf{NN}}(\mathbf{x})$ which maps the coordinate $\mathbf{x} \in \mathbb{R}^d$ to a surrogate of the solution, that is $\Psi_{\mathsf{NN}}(\mathbf{x}) \approx u(\mathbf{x})$.

Specifically, for an input vector $\mathbf{x}$, a feedforward neural network maps it to an output, via layers of "neurons" with layer k corresponding to an affine-linear map C_k composed with scalar non-linear activation functions σ [14]. That is,

$$\Psi_{\mathsf{NN}}^{\theta}(\mathbf{x}) = C_K \circ \sigma \circ C_{K-1} \ldots \ldots \circ \sigma \circ C_1(\mathbf{x}).$$

A justifiably popular choice is the *ReLU* activation $\sigma(z) = \max(z, 0)$ that is understood as component-wise operation when z is a vector. For any $1 \le k \le K$, we define

$$C_k z_k = W_k z_k + b_k, \quad \text{for } W_k \in \mathbb{R}^{d_{k+1} \times d_k}, z_k \in \mathbb{R}^{d_k}, b_k \in \mathbb{R}^{d_{k+1}}.$$

To be consistent with the input-output dimension, we set $d_1 = d$ and $d_K = 1$. We concatenate the tunable weights and biases for our network and denote them as

$$\theta := \{W_k, b_k\}, \quad \forall \, 1 \le k \le K.$$

We have $\theta \in \Theta \subset \mathbb{R}^M$ with $M := \sum_{k=1}^{K-1} (d_k + 1) d_{k+1}$. We denote this network by

$$\mathsf{NN}(d_1, d_2, \cdots, d_K). \tag{3}$$

Learning $\Psi_{\mathsf{NN}}^\theta(\mathbf{x})$ then amounts to generating training data and determining the weights and biases θ by optimizing a loss function using this data.

2.3 Physics-Informed Neural Network

We define our problem on the spatial domain $\Omega \subset \mathbb{R}^d$ with boundary $\partial\Omega$, and consider time-dependent PDEs with order of time-derivative $k = 1$ or 2.

$$
\begin{aligned}
\frac{\partial^k}{\partial t^k} u(\mathbf{x}, t) + \mathcal{F}\left[u(\mathbf{x}, t)\right] = 0 \qquad & \mathbf{x} \in \Omega, \qquad && t \in [0, T], \\
\mathcal{G}(u)(\mathbf{x}, t) = 0 \qquad & \mathbf{x} \in \partial\Omega, \qquad && t \in [0, T], \\
u(\mathbf{x}, 0) = u_0(\mathbf{x}) \qquad & \mathbf{x} \in \Omega.
\end{aligned} \tag{4}
$$

Here $\mathcal{F}$ is a differential operator as defined in Section 2.1 and $\mathcal{G}$ denotes a boundary operator. The goal of a PINN is to identify an approximate solution $u(\mathbf{x}, t)$ via a neural network $\Psi_{\mathsf{NN}}^\theta(\mathbf{x}, t)$. Learning $\theta \in \mathbb{R}^M$ requires defining a loss function whose minimum θ^* leads to $\Psi_{\mathsf{NN}}^{\theta^*}$ approximating the solution to the PDE over the problem domain. PINN defines this loss as a sum of three parts, an integral of the local residual of the differential equation over the problem domain, that over the boundary, and the deviation from the given initial condition,

$$\mathcal{J}(u) = \int_\Omega \left\| \frac{\partial^k}{\partial t^k} u(\mathbf{x}, t) + \mathcal{F}(u)(\mathbf{x}, t) \right\|_2^2 + \|u(\mathbf{x}, 0) - u_0(\mathbf{x})\|_2^2 \, dx + \int_{\partial\Omega} \|\mathcal{G}(u)(\mathbf{x}, t)\|_2^2 \, dx.$$

During training, we sample collocation points in certain fashion from the PDE space domain Ω, space-time domain $\Omega \times (0, T)$, and boundary $\partial\Omega \times [0, T]$, $\mathcal{C}_o \subset \Omega \times (0, T)$ and $\mathcal{C}_\partial \subset \partial\Omega \times [0, T]$ and $\mathcal{C}_i \subset \Omega$, and use them to form an approximation of the true loss.

$$
\begin{aligned}
\mathcal{L}_{\mathrm{PINN}}(\Psi_{\mathsf{NN}}^\theta) = & \frac{1}{|\mathcal{C}_o|} \sum_{(\mathbf{x}, t) \in \mathcal{C}_o} \left\| \frac{\partial^k}{\partial t^k} (\Psi_{\mathsf{NN}}^\theta)(\mathbf{x}, t) + \mathcal{F}(\Psi_{\mathsf{NN}}^\theta)(\mathbf{x}, t) \right\|_2^2 + \\
& \frac{1}{|\mathcal{C}_\partial|} \sum_{(\mathbf{x}, t) \in \mathcal{C}_\partial} \left\| \mathcal{G}(\Psi_{\mathsf{NN}}^\theta)(\mathbf{x}, t) \right\|_2^2 + \frac{1}{|\mathcal{C}_i|} \sum_{\mathbf{x} \in \mathcal{C}_i} \left\| \Psi_{\mathsf{NN}}^\theta(\mathbf{x}, 0) - u_0(\mathbf{x}) \right\|_2^2.
\end{aligned} \tag{5}
$$

When the training converges, we expect that $\mathcal{L}_{\mathrm{PINN}}(\Psi_{\mathsf{NN}}^\theta)$ should be nearly zero.

3 The GPT-PINN framework

Inspired by the RBM formulation eq. (2), we design the GPT-PINN. Its two components and design philosophy are depicted in Figure 1. As a hyper-reduced feedforward neural network $\mathsf{NN}(2, n, 1)$ with $1 \le n \le N$ (see eq. (3) for the notation), we denoted it by $\mathsf{NN}^{\mathrm{r}}(2, n, 1)$. A key feature is

that it has customized activation function in the neurons of its sole hidden layer. These activation functions are nothing but the pre-trained PINNs for the corresponding PDEs instantiated by the parameter values $\{\boldsymbol{\mu}^1, \boldsymbol{\mu}^2, \cdots, \boldsymbol{\mu}^n\}$ chosen by a greedy algorithm that is specifically tailored for PINNs but inspired by the classical one adopted by RBM in Algorithm 1. The design of network architecture represents the first main novelty of the paper. To the best of our knowledge, this is the first time a whole (pre-trained) network is used as the activation function of one neuron.
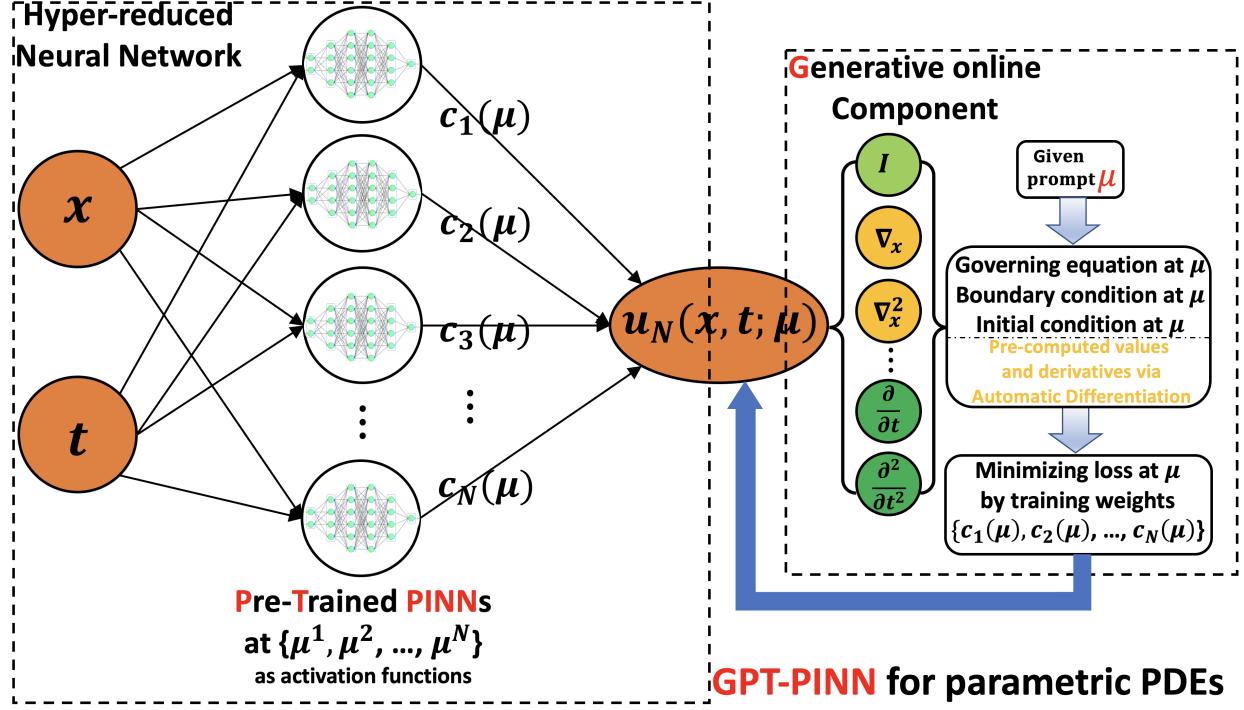

Figure 1: The GPT-PINN architecture. A hyper-reduced network adaptively embedding pre-trained PINNs at the nodes of its sole hidden layer. It then allows a quick online generation of a surrogate solution at any given parameter value.

3.1 The online solver of GPT-PINN

We first present the online solver, i.e. the training of the reduced network $\mathsf{NN}^{\mathrm{r}}(2, n, 1)$, for any given $\boldsymbol{\mu}$. With the next subsection detailing how we "grow" the GPT-PINN offline from $\mathsf{NN}^{\mathrm{r}}(2, n, 1)$ to $\mathsf{NN}^{\mathrm{r}}(2, n+1, 1)$, we have a strategy of adaptively generating the terminal GPT-PINN, $\mathsf{NN}^{\mathrm{r}}(2, N, 1)$. Indeed, given the simplicity of the reduced network, to train the weights $\{c_1(\boldsymbol{\mu}), \cdots, c_n(\boldsymbol{\mu})\}$, no backpropagation is needed. The reason is that the loss function, similar to eq. (5), is a simple function containing directly and explicitly $\{c_1(\boldsymbol{\mu}), \cdots, c_n(\boldsymbol{\mu})\}$ thanks to the reduced network structure of GPT-PINN. In fact, we denote by $\Psi_{\mathsf{NN}}^{\theta^i}(x, t)$ the PINN approximation of the PDE solution when $\boldsymbol{\mu} = \boldsymbol{\mu}^i$. Given that $u_n(\mathbf{x}, t; \boldsymbol{\mu}) \approx \sum_{i=1}^{n} c_i(\boldsymbol{\mu}) \Psi_{\mathsf{NN}}^{\theta^i}(x, t)$, we can calculate the GPT-PINN loss as a function of the weights $\mathbf{c}(\boldsymbol{\mu})$ as follows.

$$\mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c}(\boldsymbol{\mu})) = \frac{1}{|\mathcal{C}_o^r|} \sum_{(\mathbf{x},t)\in\mathcal{C}_o} \left\| \frac{\partial^k}{\partial t^k} \left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},t) + \mathcal{F} \left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},t) \right\|_2^2 +$$

$$\frac{1}{|\mathcal{C}_\partial^r|} \sum_{(\mathbf{x},t)\in\mathcal{C}_\partial} \left\| \mathcal{G} \left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},t) \right\|_2^2 + \frac{1}{|\mathcal{C}_i^r|} \sum_{\mathbf{x}\in\mathcal{C}_i} \left\| \sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i}(\mathbf{x},0) - u_0(\mathbf{x}) \right\|_2^2 . \tag{6}$$

The online collocation sets $\mathcal{C}_o^r \subset \Omega \times (0,T)$, $\mathcal{C}_\partial^r \subset \partial\Omega \times [0,T]$ and $\mathcal{C}_i^r \subset \Omega$ are used, similar to eq. (5), to generate an approximation of the true loss. They are taken to be the same as their full PINN counterparts $\mathcal{C}_o, \mathcal{C}_\partial, \mathcal{C}_i$ in this paper but we note that they can be fully independent. The training of $\mathsf{NN}^r(2,n,1)$ is then simply

$$\mathbf{c} \leftarrow \mathbf{c} - \delta_r \nabla_{\mathbf{c}} \mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c}) \tag{7}$$

Here $\mathbf{c} = (c_1(\boldsymbol{\mu}), \cdots, c_n(\boldsymbol{\mu}))^T$ and δ_r is the online learning rate. The detailed calculations of eq. (6) and eq. (7) are given in A for the first numerical example. Those for the other examples are very similar and thus omitted. We make the following three remarks to conclude the online solver.

1. Precomputation for fast training of $\mathsf{NN}^r(2,n,1)$: Due to the linearity of the derivative operations and the collocation nature of loss function, a significant amount of calculations of eq. (6) can be precomputed and stored. These include the function values and all (spatial and time) derivatives involved in the operators $\mathcal{F}$ and $\mathcal{G}$ of the PDE eq. (4):

$$\Psi_{\mathsf{NN}}^{\theta^i}(\mathcal{C}), \; \frac{\partial^k}{\partial t^k}\left(\Psi_{\mathsf{NN}}^{\theta^i}\right)(\mathcal{C}) \; (k=1 \text{ or } 2), \; \nabla_{\mathbf{x}}^\ell \Psi_{\mathsf{NN}}^{\theta^i}(\mathcal{C}) \; (\ell=1,2,\cdots) \text{ for } \mathcal{C} = \mathcal{C}_o^r, \mathcal{C}_\partial^r, \mathcal{C}_i^r. \tag{8}$$

Once these are precomputed, updating $\mathbf{c}$ according to eq. (7) is very efficient. It can even be made independent of $|\mathcal{C}|$.

2. Non-intrusiveness of GPT-PINN: It is clear that, once the quantities of eq. (8) are extracted from the full PINN, the online training of $\mathsf{NN}^r(2,n,1)$ is independent of the full PINN. GPT-PINN is therefore non-intrusive of the Full Order Model. One manifestation of this property is that, as shown in our third numerical example, the full PINN can be adaptive while the reduced PINN may not be.

3. The error indication of $\mathsf{NN}^r(2,n,1)$. One prominent feature of RBM is its *a posteriori* error estimators/indicators which guides the generation of the reduced solution space and certifies the accuracy of the surrogate solution. Inspired by this classical design, we introduce the following quantity that measures how accurate $\mathsf{NN}^r(2,n,1)$ is in generating a surrogate network at a new parameter $\boldsymbol{\mu}$.

$$\Delta_{\mathsf{NN}}^r(\mathbf{c}(\boldsymbol{\mu})) \triangleq \mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c}(\boldsymbol{\mu})). \tag{9}$$

We remark that this quantity is essentially free since it is readily available when we train $\mathsf{NN}^r(2,n,1)$ according to eq. (7). The adoption of the training loss of the meta-network as an error indicator, inspired by the residual-based error estimation for traditional numerical solvers such as FEM, represents the second main novelty of this paper.

3.2 Training the reduced network GPT-PINN: the greedy algorithm

With the online solver described in Section 3.1, we are ready to present our greedy algorithm. Its main steps are outlined in Algorithm 2 with its flowchart provided in Figure 2. The meta-network adaptively "learns" the parametric dependence of the system and "grows" its sole hidden layer one

---
Algorithm 2 GPT_PINN for parametric PDE: Offline stage

Input: A (random or given) $\boldsymbol{\mu}^1$, training set $\Xi_{\text{train}} \subset \mathcal{D}$, full PINN.

1: Train a full PINN at $\boldsymbol{\mu}^1$ to obtain $\Psi_{\text{NN}}^{\theta^1}$. Precompute quantities necessary for $\nabla_{\mathbf{c}}\mathcal{L}_{\text{PINN}}^{\text{GPT}}$ at collocation nodes $\mathcal{C}_o^r$, $\mathcal{C}_{\partial}^r$, and $\mathcal{C}_i^r$, see eq. (8). Set $n = 2$.
2: **while** *stopping criteria not met,* **do**
3:     Train $\text{NN}^{\text{r}}(2, n-1, 1)$ at $\boldsymbol{\mu}$ for all $\boldsymbol{\mu} \in \Xi_{\text{train}}$ and record the indicator $\Delta_{\text{NN}}^r(\boldsymbol{\mu})$.
4:     Choose $\boldsymbol{\mu}^n = \arg \max\limits_{\boldsymbol{\mu} \in \Xi_{\text{train}}} \Delta_{\text{NN}}^r(\boldsymbol{\mu})$.
5:     Train a full PINN at $\boldsymbol{\mu}^n$ to obtain $\Psi_{\text{NN}}^{\theta^n}$. Precompute quantities necessary for $\nabla_{\mathbf{c}}\mathcal{L}_{\text{PINN}}^{\text{GPT}}$ at collocation nodes $\mathcal{C}_o^r$, $\mathcal{C}_{\partial}^r$, and $\mathcal{C}_i^r$, see eq. (8).
6:     Update the GPT_PINN by adding a neuron to the hidden layer to construct $\text{NN}^{\text{r}}(2, n, 1)$.
7:     Set $n \leftarrow n + 1$.
8: **end while**

Output: GPT_PINN $\text{NN}^{\text{r}}(2, N, 1)$, with N being the terminal index.
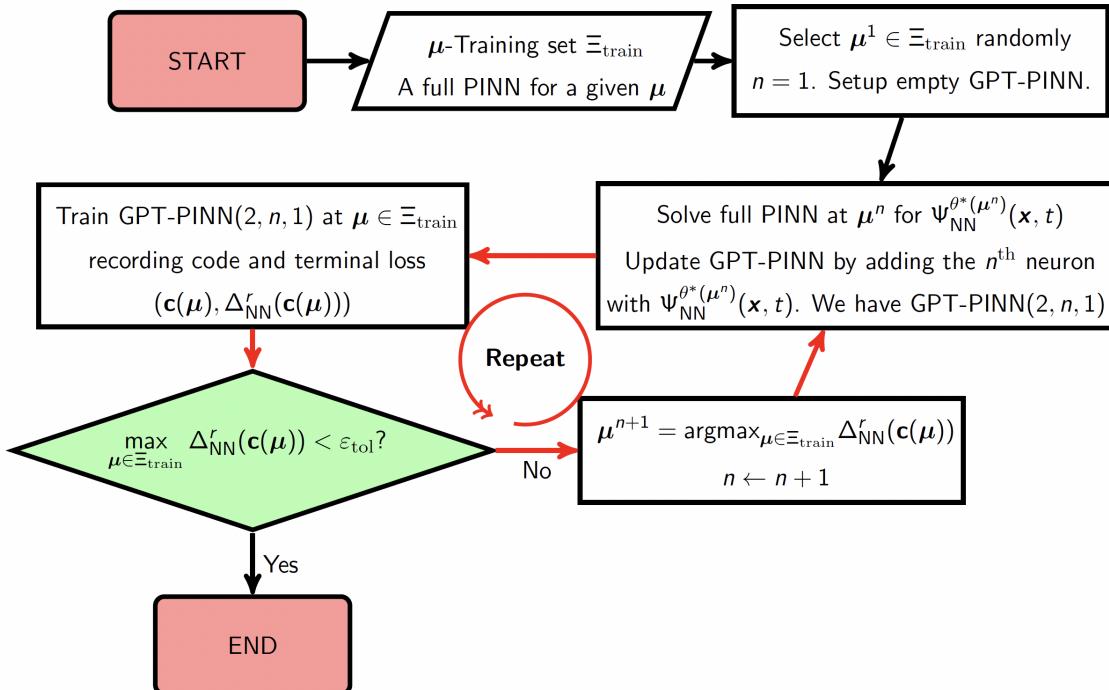---

Figure 2: Flowchart of the GPT-PINN Offline training stage.

neuron/network at a time in the following fashion. We first randomly select, in the discretized parameter domain Ξ_{train}, one parameter value $\boldsymbol{\mu}^1$ and train the associated (highly accurate) PINN $\Psi_{\text{NN}}^{\theta^1}$. The algorithm then decides how to "grow" its meta-network by scanning the entire discrete parameter space Ξ_{train} and, for each parameter value, training this reduced network (of 1 hidden layer with 1 neuron $\Psi_{\text{NN}}^{\theta^1}$). As it scans, it records an error indicator $\Delta_{\text{NN}}^r(\mathbf{c}(\boldsymbol{\mu}))$. The next parameter value $\boldsymbol{\mu}^2$ is the one generating the largest error indicator. The algorithm then proceeds by training a full PINN at $\boldsymbol{\mu}^2$ and therefore grows its hidden layer into two neurons with customized (but pre-trained) activation functions $\Psi_{\text{NN}}^{\theta^1}$ and $\Psi_{\text{NN}}^{\theta^2}$. This process is repeated until the stopping criteria is met which can be either that the error indicator is sufficiently small or a pre-selected size of the reduced network is met. At every step, we select the parameter value that is approximated most badly by the current meta-network. We end by presenting how we initialize the weights $\mathbf{c}(\boldsymbol{\mu})$ when we train $\text{NN}^r(2, n-1, 1)$ on Line 3 of Algorithm 2. They are initialized by a linear interpolation of up to 2^{d_s} closest neighbors of $\boldsymbol{\mu}$ within the chosen parameter values $\{\boldsymbol{\mu}^1, \ldots, \boldsymbol{\mu}^N\}$. Recall that d_s is the dimension of the parameter domain.

3.3   Related work

The last two to three years have witnessed an increasing level of interest toward metalearning of (parameterized or unparameterized) PDEs due to the need of repeated simulations and the remarkable success of PINNs in its original form or adaptive ones. Here we mention a few representative ones and point out how our method differentiates from theirs.

Metalearning via PINN parameters. In [30], the authors adopt statistical (e.g. regression) and numerical (e.g. RBF/spline interpolation) methods to build a surrogate for the map from the PDE parameter $\boldsymbol{\mu}$ to the PINN parameter (weights and biases, θ). They are shown to be superior than MAML [12] for parameterized PDEs which was shown to outperform LEAP [13] in [36]. Both are general-purpose meta-learning methods. However, the online solver (i.e. regression or interpolation) of [30] ignores the physics (i.e. PDE). The method assumes that the $\boldsymbol{\mu}$-variation of the PINN weights and biases is analogous to that of the PDE solution.

DeepONet. Aiming to learn nonlinear operators, a DeepONet [21] consists of two sub-networks, a branch net for encoding the input function (e.g source/control term, as opposed to PDE coefficients) at a fixed number of sensors, and a trunk net for encoding the locations for the output functions. It does not build in the physics represented by the dynamical system or PDE for a new input. Moreover, it is relatively data-intense by having to scan the entire input function space such as Gaussian random field or orthogonal polynomial space.

Metalearning loss functions. Authors of [35] concern the definition of the PINN loss functions. While it is in the parameterized PDE setting, the focus is a gradient-based approach to discover, during the offline stage, better PINN loss functions which are parameterized by e.g. the weights of each term in the composite objective function. The end goal is therefore improved PINN performance e.g. at unseen PDE parameters, due to the learned loss function configuration.

Metalearning initialization. In [50], the authors study the use of a meta network, across the parameter domain of a 1-D arc model of plasma simulations, to better initialize the PINN at a new task (i.e. parameter value).

MetaNO. The recent meta-learning approach for transferring knowledge between neural operators [49] aims to transfer the learned network parameters $\theta(\boldsymbol{\mu})$ between different $\boldsymbol{\mu}$ with only the first layer being retrained. Its resulting surrogate is fully data-driven, i.e. with no physics built in for a new value $\boldsymbol{\mu}$.

PRNN. The physics-reinforced neural network approach [6] builds the map $\boldsymbol{\mu} \mapsto \mathbf{c}(\boldsymbol{\mu})$ via regression (i.e. no physics during the online evaluation for a new $\boldsymbol{\mu}$) although PDE residuals were

considered during the supervised learning of the map via labelled data.

Our proposed GPT-PINN exploits the $\boldsymbol{\mu}$-variation of the PDE solution directly which may feature a Kolmogorov N-width friendlier to MOR approaches, see Figure 3, than the weights and biases. This is, in part, because that the weights and biases lie in a (much) higher dimensional space. Moreover, the meta-network of our approach, being a PINN itself, has physics automatically built in in the same fashion as the underlying PINNs. Lastly, our approach provides a surrogate solution to the unseen parameter values in addition to a better initialization transferred from the sampled PINNs. Most importantly, our proposed GPT-PINN embodies prior knowledge that is mathematically rigorous and PDE-pertinent into the network architecture. This produces strong inductive bias that usually leads to good generalization.
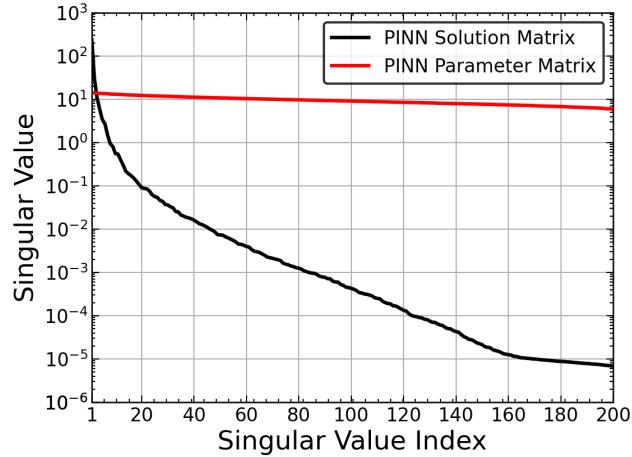

Figure 3: A motivating example showing that the solution matrix of a parametric PDE $\{u(\cdot, \boldsymbol{\mu}^n)\}_{n=1}^{200}$ exhibits fast decay in its singular values (indicating fast decay of the Kolmogorov N-width of the solution manifold) while the network weights and biases manifold $\{\theta(\boldsymbol{\mu}^n)\}_{n=1}^{200}$ does not.

4 Numerical results

In this section, we present numerical results of the GPT-PINN applied to three families of equations, the Klein-Gordon equation, the Burgers' equation, and the Allen-Cahn equation. All simulations are run on a desktop with AMD Ryzen 7 2700X CPU clocked at 4.0GHz, an NVIDIA GeForce RTX 2060 SUPER GPU, and 32 GB of memory. Python version 3.9.12 was used along with common numerical packages and machine learning frameworks such as NumPy (v1.23.4), PyTorch (v1.11.0), TensorFlow (v2.10.0), and for GPU support CUDA v11.6 was installed. Previous literature [46, 23, 24, 47] has shown common difficulties in the use of baseline (non-adaptive) PINNs for the approximation of the Allen-Cahn equations. We have therefore adopted the Self-Adaptive PINNs (SA-PINNs) formulated by [24] in section 4.3 to acquire accurate approximations by the full PINN, later used by the GPT-PINN. The tuned hyperparameters of the full PINNs include the activation functions and the learning rates. The code for all these examples are published on GitHub at https://github.com/skoohy/GPT-PINN. Throughout the experiments of sections 4.1 to 4.3, we calculate and report various losses and errors. They are defined in Table 1.

Largest Loss	The worst-case training loss $\max\limits_{\boldsymbol{\mu}\in\Xi_{\text{train}}} \mathcal{L}_{\text{PINN}}^{\text{GPT}}(\mathbf{c}(\boldsymbol{\mu}))$ of $\mathsf{NN}^{\text{r}}(2,n,1)$
Terminal Losses	Training losses $\mathcal{L}_{\text{PINN}}^{\text{GPT}}(\mathbf{c}(\boldsymbol{\mu}))$ of $\mathsf{NN}^{\text{r}}(2,n,1)$ (for more statistics)
Largest Error	The worst-case testing error $\max\limits_{\boldsymbol{\mu}\in\Xi_{\text{test}}} \dfrac{\left\|\mathsf{NN}^{\text{r}}(2,n,1)(x,t) - \Psi_{\mathsf{NN}}^{\theta(\boldsymbol{\mu})}(x,t)\right\|_2}{\left\|\Psi_{\mathsf{NN}}^{\theta(\boldsymbol{\mu})}(x,t)\right\|_2}$
Terminal Errors	Testing errors $\dfrac{\left\|\mathsf{NN}^{\text{r}}(2,n,1)(x,t) - \Psi_{\mathsf{NN}}^{\theta(\boldsymbol{\mu})}(x,t)\right\|_2}{\left\|\Psi_{\mathsf{NN}}^{\theta(\boldsymbol{\mu})}(x,t)\right\|_2}$ (for more statistics)
Point-wise Error	Absolute error for a given $\boldsymbol{\mu}$, $\left|\mathsf{NN}^{\text{r}}(2,N,1)(x,t) - \Psi_{\mathsf{NN}}^{\theta}(x,t)\right|$

Table 1: Exact meaning of the loss and error quantities reported in Section 4.

4.1 The parametric Klein-Gordon Equation

We first test the Klein-Gordon equation parameterized by $(\alpha,\beta,\gamma) \in [-2,-1] \times [0,1] \times [0,1]$,

$$
\begin{aligned}
u_{tt} + \alpha u_{xx} + \beta u + \gamma u^2 + x\cos(t) - x^2\cos^2(t) &= 0, \quad (x,t) \in [-1,1] \times [0,5], \\
u(-1,t) = -\cos(t), \quad u(1,t) &= \cos(t), \\
u(x,0) &= x, \\
u_t(x,0) &= 0.
\end{aligned}
\tag{10}
$$

The full PINN is a $[2,40,40,1]$-fully connected network with activation function $\cos(z)$ that is trained using uniformly distributed collocation points with $|\mathcal{C}_o| = 10{,}000$, $|\mathcal{C}_\partial| = 512$, $|\mathcal{C}_i| = 512$. A learning rate of 0.0005 is used with the ADAM optimizer and the maximum number of epochs being 75,000. The parameter training set Ξ_{train} is a tensorial grid of size $10 \times 10 \times 10$ for a total of 1000 parameter values. Up to 15 neurons are generated by the greedy algorithm producing GPT-PINNs of sizes $[2,1,1]$ to $[2,15,1]$. The GPT-PINNs are trained at the same set of collocation points as the full PINN (i.e. $\mathcal{C}_{pos}^r = \mathcal{C}_{pos}$ for $pos \in \{o,\partial,i\}$) but with a learning rate of 0.025 and (much smaller) 2000 epochs.
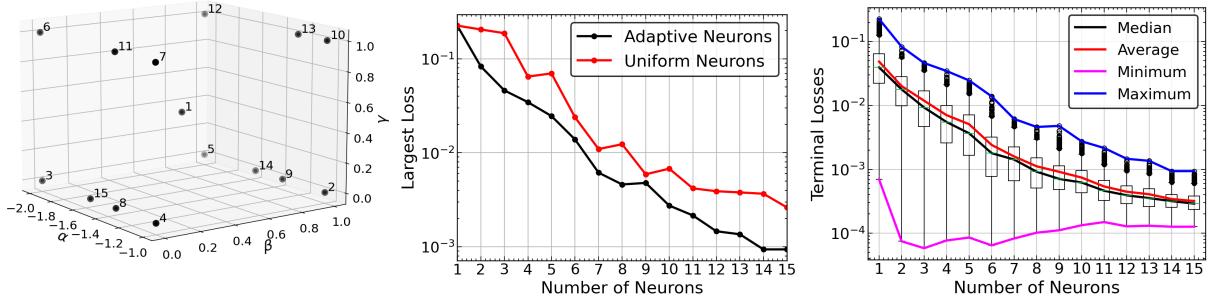

Figure 4: Klein-Gordon Equation training: The adaptively chosen parameter values (Left), worst-case GPT-PINN training losses (Middle), and the Box and Whisker plot of all adaptive GPT-PINN training losses (Right) during the outer-layer greedy training.
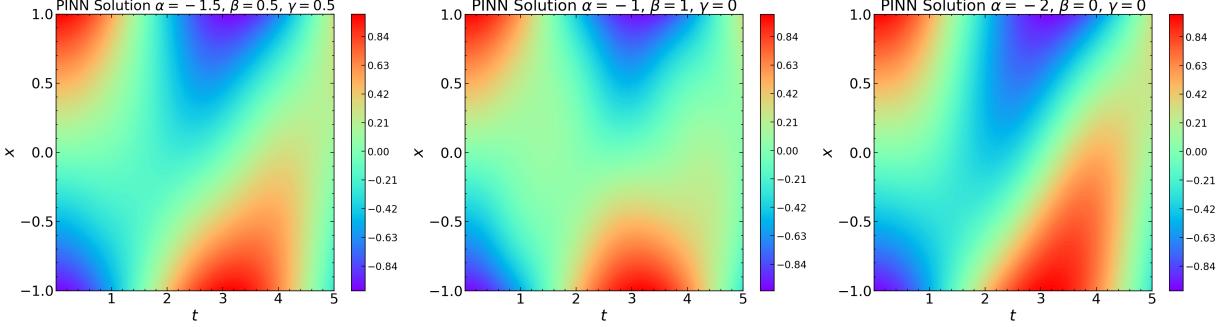
11

Figure 5: Klein-Gordon Equation: First three full PINN solutions found by the GPT-PINN that are used as the activation functions.

The GPT-PINN generates 15 neurons, i.e. full PINNs at $\{(\alpha_i, \beta_i, \gamma_i)\}_{i=1}^{15}$. These parameter values and the worse-case offline training loss $\mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c}(\boldsymbol{\mu}))$ after 2000 epochs as we increase the number of neurons (i.e. size of $\mathbf{c}(\boldsymbol{\mu})$) in the hidden layer of GPT-PINN are shown in Figure 4. Figure 5 shows the first three PINN solutions adaptively selected by GPT-PINN. It is clear that the sampled parameter values are toward the boundaries of the domain and that the decrease in training loss is exponential. Both features are consistent with typical RBM results. Moreover, we emphasize that to achieve 3 digits of accuracy across the parameter domain, we only need to train the full PINN 15 times. This contrast with pure data-driven approaches inherits that of RBM with POD approaches in that RBM requires much less full-order solves. In comparison, we sample the parameter domain uniformly (i.e. without the greedy approach of GPT-PINN), it is clear from Figure 4 Middle that the adaptive "learned neurons" performs 2 to 3 times better than the non-adaptive "uniform neurons". The fact that the latter performs reasonably well underscores the power of our novel idea of using pre-trained PINNs as activation functions.

Next, we test the GPT-PINN on Ξ_{test} consisting of 200 randomly selected parameter values distinct from the adaptively chosen "learned neurons" and "uniform neurons". Figure 6 displays the largest error for each size of the GPT-PINN. The trend is again exponential. Finally, to show the efficiency of the method, we plot in Figure 6 Right the cumulative run-time when both the full PINN and the (reduced) GPT-PINN are repeatedly called. The starting point of the GPT-PINN line reflects all offline preparation time. It is clear that the GPT-PINN line increases very slowly reflecting the fact that its marginal cost is tiny. In fact, it is about 0.0022 of that of the full PINN. The intersection points reflect how many simulations would it be worthwhile to invest in GPT-PINN. We remark that future work includes driving the intersection point down to essentially comparable to the number of neurons in GPT-PINN, which is the absolute minimum it could be.

Last but not least, we show the training losses as functions of epochs in Figure 7 for both the full PINN and GPT-PINN. We note the interesting phenomenon that the GPT-PINN loss decreases more smoothly than the full PINN. To give a sense of the error distribution, we also plot the point-wise error of the GPT-PINN solution.

4.2 The parametric viscous Burgers' Equation

Next, we test GPT-PINN on the Burgers' equation with one parameter, the viscosity $\nu \in [0.005, 1]$.

$$
\begin{aligned}
u_t + u u_x - \nu u_{xx} &= 0, \quad (x, t) \in [-1, 1] \times [0, 1], \\
u(-1, t) = u(1, t) &= 0, \\
u(x, 0) &= -\sin(\pi x).
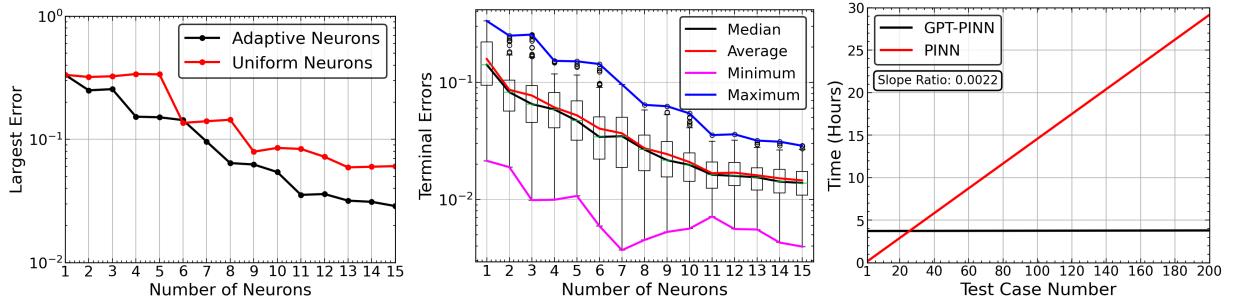\end{aligned}
\tag{11}
$$

12

Figure 6: Klein-Gordon Equation testing: Worst-case test error of the GPT-PINN of various sizes (Left), Box and Whisker plot of all adaptive GPT-PINN testing errors (Middle), and cumulative run time of the full PINN versus the GPT-PINN (Right).
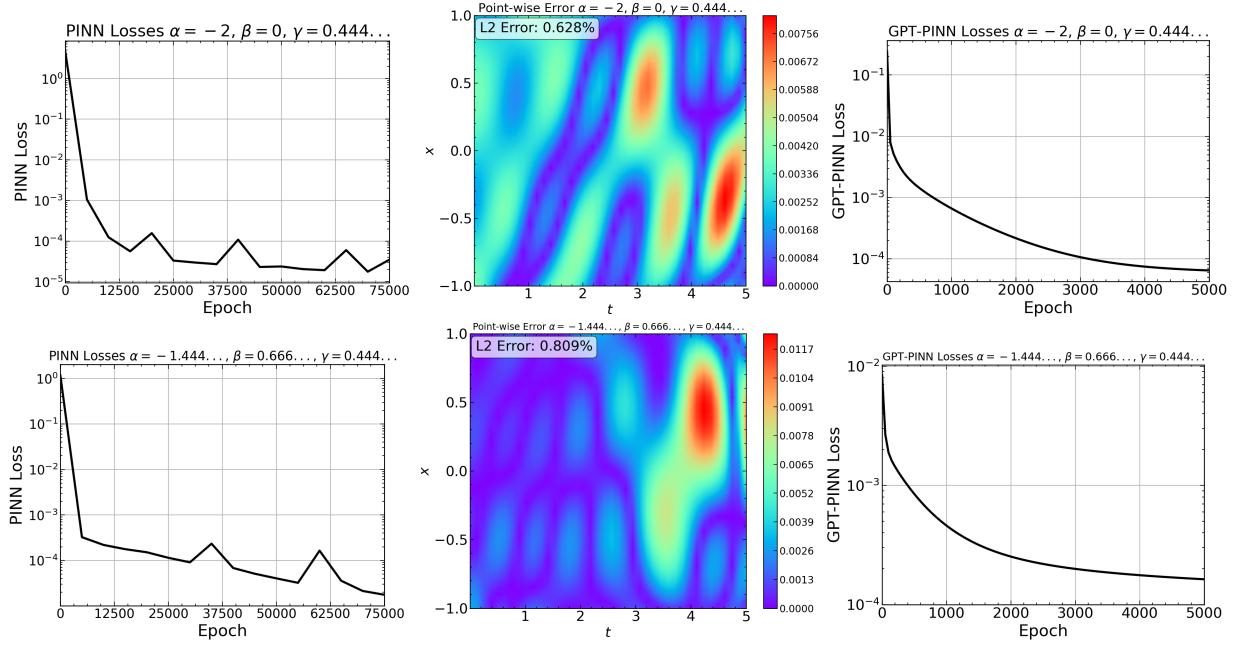

Figure 7: Klein-Gordon Equation: Full PINN training loss (Left) and GPT-PINN training loss (Right) as functions of epochs for various parameters. Plotted in the middle are the point-wise errors of the corresponding GPT-PINN solution.

The full PINN is a $[2, 20, 20, 20, 20, 1]$-fully connected network with activation function $\tanh(z)$ that is trained using uniformly distributed collocation points with $|\mathcal{C}_o| = 10,000$, $|\mathcal{C}_\partial| = 100$, $|\mathcal{C}_i| = 100$. A learning rate of 0.005 is used with the ADAM optimizer. A maximum number of $60,000$ epochs is run with a stopping criteria of 2×10^{-5} implemented on the loss values. The parameter training set is a uniform grid of size 129 in the ν-domain. Up to 9 neurons are generated by the greedy algorithm producing the reduced GPT-PINNs of sizes $[2, 1, 1]$ to $[2, 9, 1]$. The GPT-PINNs are trained at the same set of collocation points as the full PINN but with a learning rate of 0.02 and 2000 epochs. The solutions of eq. (11) develop near-discontinuities as time evolves when ν is small. In this scenario, $\left(\Psi_{\mathsf{NN}}^{\theta^i}\right)_{xx}$ is of little value in the training of GPT-PINN when x is close to these large gradients. We therefore exclude the collocation points where $\left|\left(\Psi_{\mathsf{NN}}^{\theta^i}\right)_{xx}\right|$ is within the top 20% of all such values. That is

$$\mathcal{C}_{pos}^r = \mathcal{C}_{pos} \setminus \left\{ x : \left|\left(\Psi_{\mathsf{NN}}^{\theta^i}\right)_{xx}(x)\right| > 0.8 \max_x \left|\left(\Psi_{\mathsf{NN}}^{\theta^i}\right)_{xx}(x)\right| \right\}, \quad pos \in \{o, \partial, i\}.$$
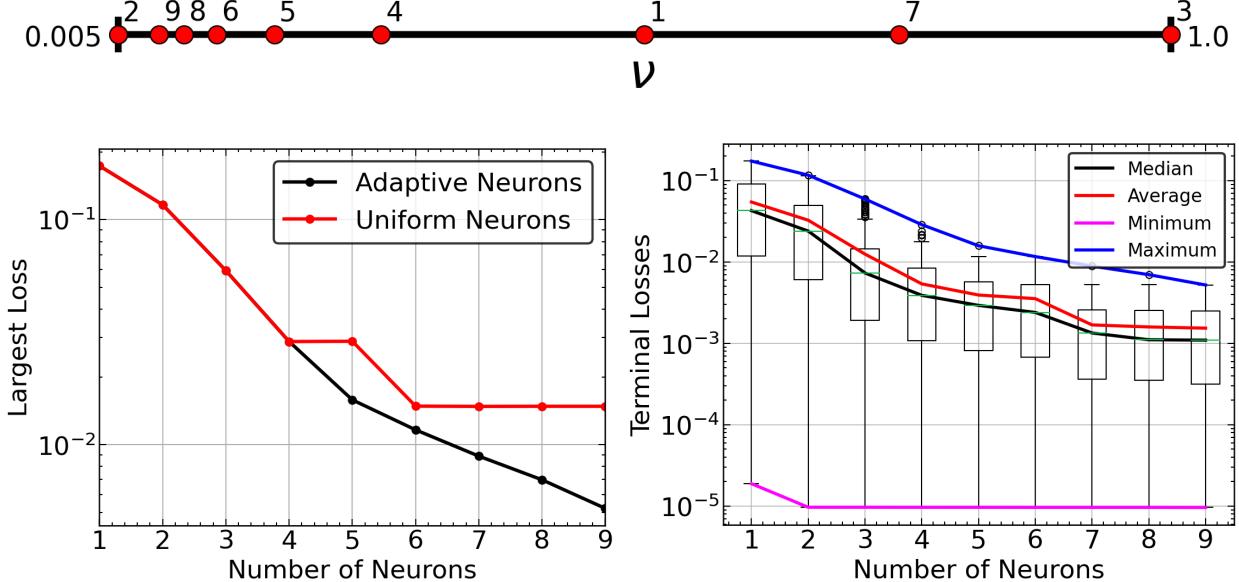

Figure 8: Burgers' Equation training: The adaptively chosen parameter values (Top), worst-case GPT-PINN training losses (Bottom Left), and the Box and Whisker plot of all GPT-PINN training losses (Bottom Right) during the outer-layer greedy training.

The GPT-PINN generates 9 neurons, i.e. full PINNs at $\{(\nu_i,\}_{i=1}^9$. These parameter values and the worse-case offline training loss $\mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c}(\boldsymbol{\mu}))$ after 2000 epochs as we increase the number of neurons (i.e. size of $\mathbf{c}(\boldsymbol{\mu})$) in the hidden layer of GPT-PINN are shown in Figure 8. Figure 9 shows the first three PINN solutions adaptively selected by GPT-PINN. We observe behavior that is similar to the Klein-Gordon case and consistent with typical RBM results. The adaptive "learned neurons" again perform 3 to 4 times better than the non-adaptive "uniform neurons" which already perform reasonably well, underscoring the power of our novel idea of using pre-trained PINNs as activation functions.

Next, we test the GPT-PINN on 25 parameter values. Figure 10 displays the largest error for each size of the GPT-PINN. The trend is again exponential. Finally, to show the efficiency of the
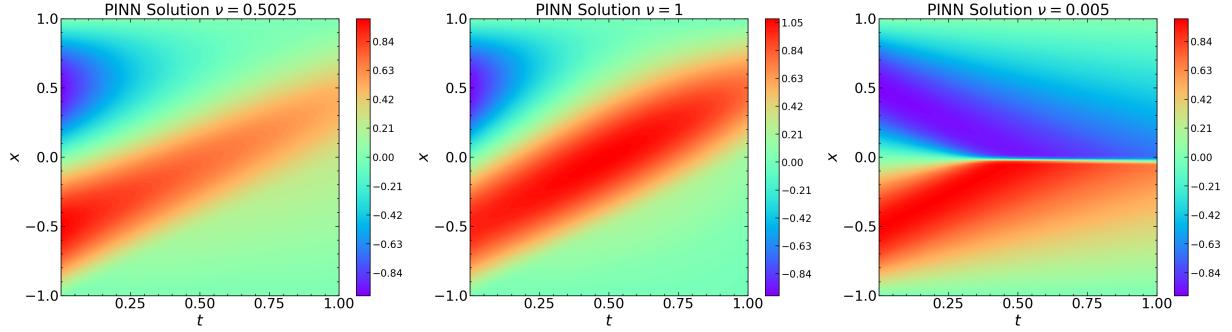
Figure 9: Burgers' Equation: First three full PINN solutions found by the GPT-PINN that are used as the activation functions.
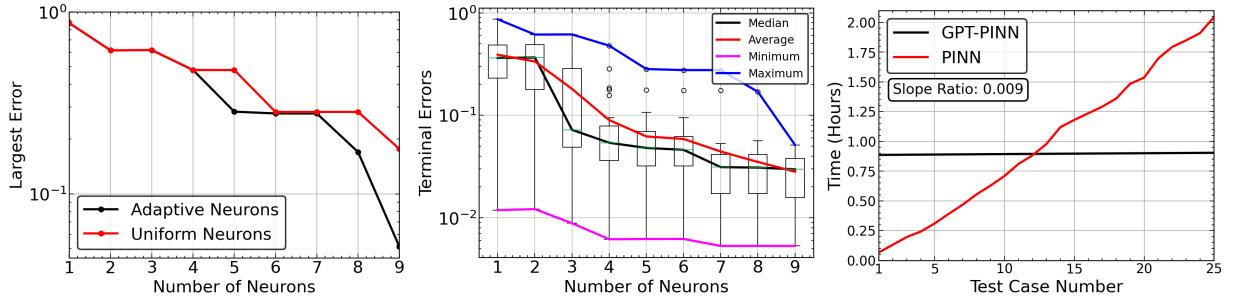

Figure 10: Burgers' Equation testing: Worst-case test error of the GPT-PINN of various sizes (Left), Box and Whisker plot of all adaptive GPT-PINN testing errors (Middle), and cumulative run time of the full PINN versus the GPT-PINN (Right).
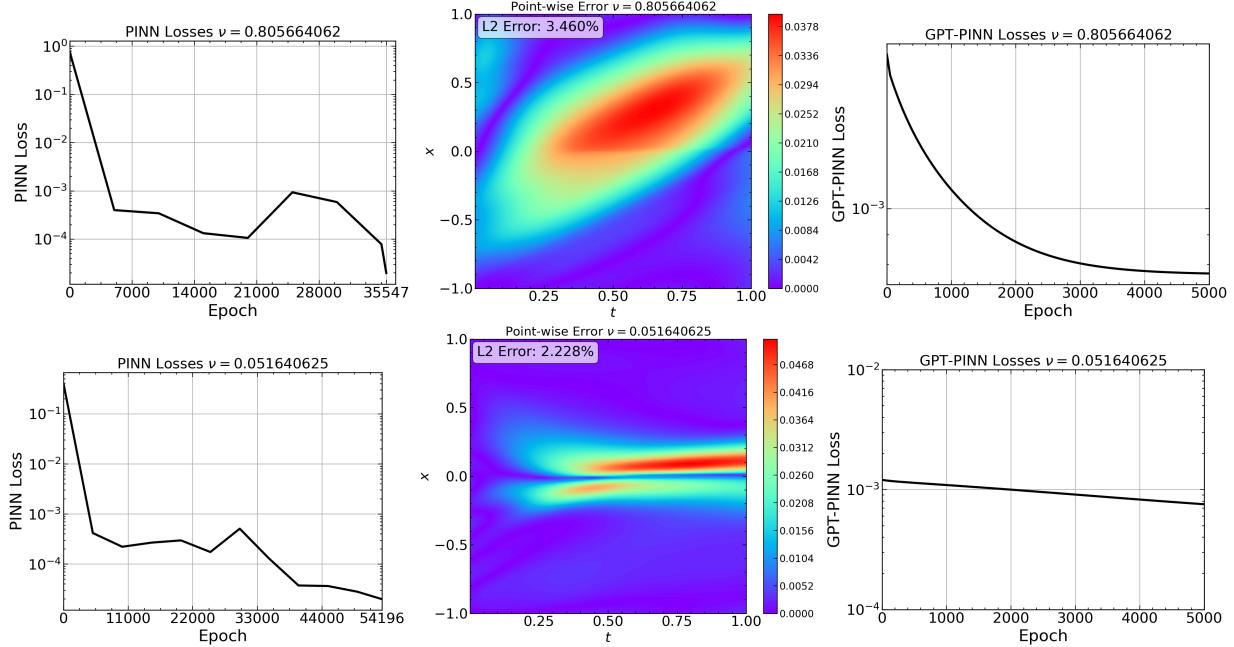

Figure 11: Burgers' Equation: Full PINN training loss (Left) and GPT-PINN training loss (Right) as functions of epochs for various parameters. Plotted in the middle are the point-wise errors of the corresponding GPT-PINN solution.

15

method, we plot in Figure 10 Right the cumulative run-time when both the full PINN and the (reduced) GPT-PINN are repeatedly called. It is clear that the GPT-PINN line increases very slowly (a relative speed of 0.009 in comparison to the full PINN) and that it is worthwhile to invest in GPT-PINN for a very modest number (12) of queries. We again show the training losses as functions of epochs in Figure 11 for both the full PINN and GPT-PINN. We note again that the GPT-PINN loss decreases more smoothly than the full PINN. This result also verifies the efficacy of our initialization strategy since the starting loss of the GPT-PINN is already very low.

4.3 The parametric Allen-Cahn Equation

Finally, we test the Allen-Cahn equation parameterized by $(\lambda, \epsilon) \in [0.0001, 0.001] \times [1, 5]$

$$
\begin{aligned}
u_t - \lambda u_{xx} + \epsilon(u^3 - u) = 0, \quad & (x, t) \in [-1, 1] \times [0, 1] \\
u(-1, t) = u(1, t) = -1 & \\
u(x, 0) = x^2 \cos(\pi x). &
\end{aligned}
\tag{12}
$$

The SA-PINN [24] is a $[2, 128, 128, 128, 128, 1]$-fully connected network with activation function $\tanh(z)$ that is trained on collocation points distributed by a Latin hypercube sampling with $|\mathcal{C}_o| = 20,000$, $|\mathcal{C}_\partial| = 100$, $|\mathcal{C}_i| = 512$. A learning rate of 0.005 with 10,000 epochs of ADAM optimization followed by 10,000 epochs of L-BFGS optimization with a learning rate of 0.8 is used. The parameter training set is a grid of size 121 uniform parameter values. Up to 9 neurons are generated by the greedy algorithm producing the reduced GPT-PINNs of sizes $[2, 1, 1]$ to $[2, 9, 1]$. The GPT-PINNs are trained at the same set of collocation points as the SA-PINN but with a learning rate of 0.0025 and 2000 epochs.
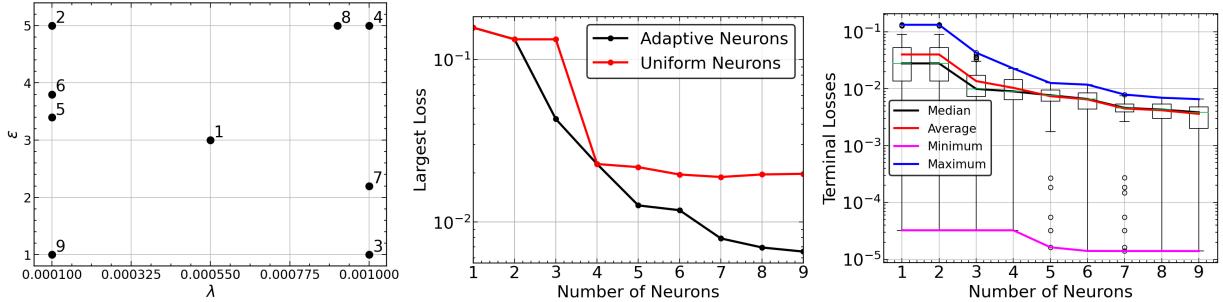

Figure 12: Allen-Cahn Equation training: The chosen parameter values (Left), worst-case GPT-PINN training losses (Middle), and the Box and Whisker plot of all GPT-PINN training losses (Right) during the outer-layer greedy training

The GPT-PINN generates 9 neurons, i.e. SA-PINNs at $\{(\epsilon_i, \lambda_i)\}_{i=1}^9$. These parameter values and the worse-case offline training loss $\mathcal{L}_{\text{PINN}}^{\text{GPT}}(\mathbf{c}(\boldsymbol{\mu}))$ after 2000 epochs as we increase the number of neurons (i.e. size of $\mathbf{c}(\boldsymbol{\mu})$) in the hidden layer of GPT-PINN are shown in Figure 12. Figure 13 shows the first three PINN solutions adaptively selected by GPT-PINN.

Next, we test the GPT-PINN on 25 parameter values. Figure 14 displays the largest error for each size of the GPT-PINN and the cumulative run-time when both the SA-PINN and the GPT-PINN are repeatedly called. It is clear that the GPT-PINN line increases very slowly (at a relative speed of 0.0006) and that it is again worthwhile to invest in GPT-PINN for a very modest number (9-10) of queries. We show the training losses as functions of epochs in Figure 15 for both the SA-PINN and GPT-PINN, with the latter again decaying more smoothly.
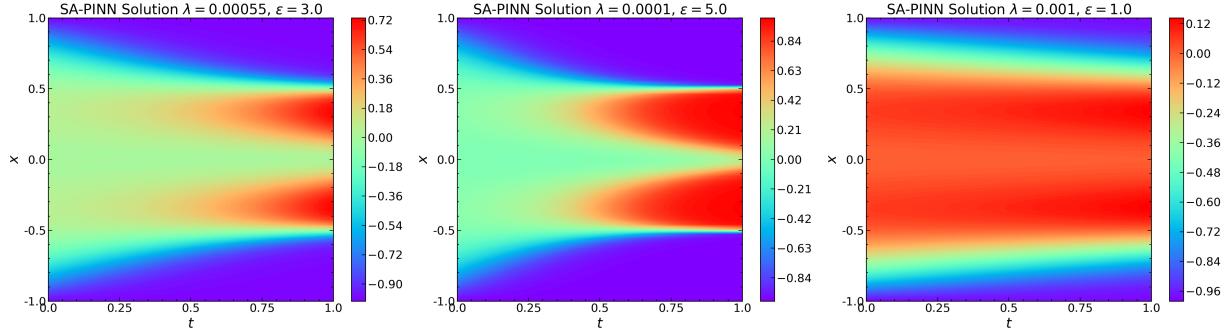
Figure 13: Allen-Cahn Equation: First three SA-PINN solutions found by the GPT-PINN that are used as the activation functions.
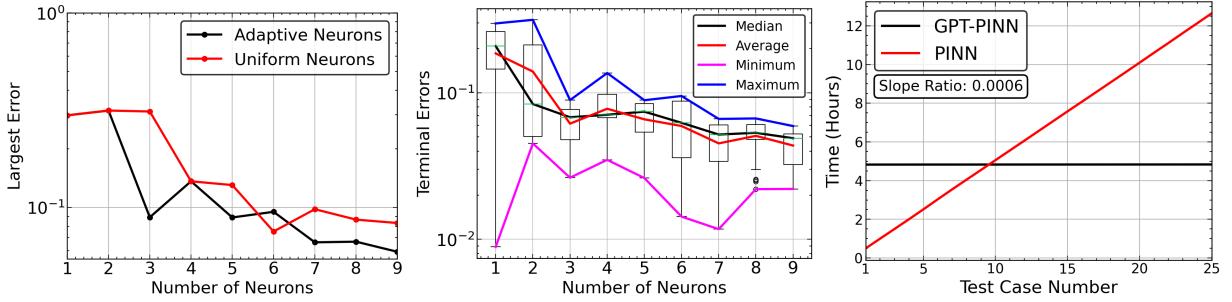

Figure 14: Allen-Cahn Equation testing: Worst-case test error of the GPT-PINN of various sizes (Left), Box and Whisker plot of all (Middle), and cumulative run time of the full PINN versus the GPT-PINN (Right)
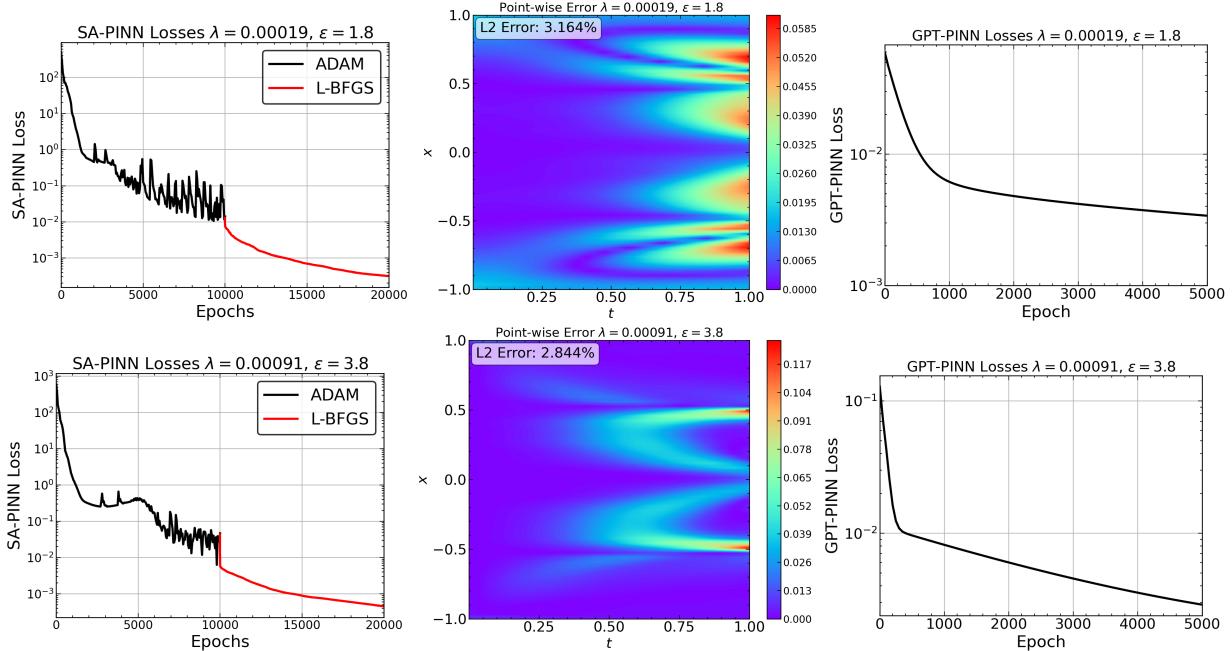

Figure 15: Allen-Cahn Equation: SA-PINN training loss (Left) and GPT-PINN training loss (Right) as functions of epochs for various parameters. Plotted in the middle are the point-wise errors of the corresponding GPT-PINN solution.

17

5 Conclusion

The proposed Generative Pre-Trained PINN (GPT-PINN) is shown to mitigate two challenges faced by PINNs in the setting of parametric PDEs, namely the cost of training and over-parameterization. Being a hyper-reduced network with activation functions pre-trained full PINNs, GPT-PINN represents a brand-new meta-learning paradigm for parametric systems. With two main novelties, the design of network architecture including its special activation functions and the adoption of the training loss of the meta-network as an error indicator, and via tests on three differential families of parametric equations, we have shown that encompassing a very small number of well-chosen networks can generate surrogate PINNs across the entire parameter domain accurately and efficiently.

A Detailed gradient of loss function for the Klein-Gordon case GPT-PINN

With the GPT-PINN formulation and considering the types of boundary and initial conditions for the equation given by eq. (10), the loss function eq. (6) becomes

$$
\mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c}(\boldsymbol{\mu})) = \frac{1}{|\mathcal{C}_o^r|} \sum_{(\mathbf{x},t)\in\mathcal{C}_o^r} \left\| \frac{\partial^2}{\partial t^2}\left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},t) + \alpha\frac{\partial^2}{\partial x^2}\left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},t) + \right.
$$

$$
\left. \beta\left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},t) + \gamma\left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)^2(\mathbf{x},t) + \mathbf{x}\cos(t) - \mathbf{x}^2\cos^2(t) \right\|_2^2
$$

$$
+ \frac{1}{|\mathcal{C}_\partial^r|} \sum_{(\mathbf{x},t)\in\mathcal{C}_\partial^r} \left\| \sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i}(\mathbf{x},t) - u(\mathbf{x},t) \right\|_2^2
$$

$$
+ \frac{1}{|\mathcal{C}_i^r|} \sum_{\mathbf{x}\in\mathcal{C}_i^r} \left\| \sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i}(\mathbf{x},0) - u(\mathbf{x},0) \right\|_2^2 + \frac{1}{|\mathcal{C}_i^r|} \sum_{\mathbf{x}\in\mathcal{C}_i^r} \left\| \frac{\partial}{\partial t}\left(\sum_{i=1}^n c_i(\boldsymbol{\mu})\Psi_{\mathsf{NN}}^{\theta^i} \right)(\mathbf{x},0) - u_t(\mathbf{x},0) \right\|_2^2
$$

with given $u(\mathbf{x},t)$ for $(\mathbf{x},t)\in\mathcal{C}_\partial^r$ and $u(\mathbf{x},0)$ and $u_t(\mathbf{x},0)$ when $\mathbf{x}\in\mathcal{C}_i^r$. The mth component of $\nabla_{\mathbf{c}}\mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c})$ needed for the GPT-PINN training eq. (7) then reads:

$$
\frac{\partial\mathcal{L}_{\mathrm{PINN}}^{\mathrm{GPT}}(\mathbf{c})}{\partial c_m} = \frac{2}{|\mathcal{C}_o^r|} \sum_{(\mathbf{x},t)\in\mathcal{C}_o^r} \left(\left(\sum_{i=1}^n \left(c_i P_{tt}^i + \alpha c_i P_{xx}^i + \beta c_i P^i \right) + \gamma\left(\sum_{i=1}^n c_i P^i \right)^2 + \mathbf{x}\cos(t) - \mathbf{x}^2\cos^2(t) \right) \right.
$$

$$
\left. \cdot \left(P_{tt}^m + \alpha P_{xx}^m + \beta P^m + 2\gamma\left(\sum_{i=1}^n c_i P^i \right)P^m \right) \right) + \frac{2}{|\mathcal{C}_\partial^r|} \sum_{(\mathbf{x},t)\in\mathcal{C}_\partial^r} \left(\left(\sum_{i=1}^n c_i P^i - u(\mathbf{x},t) \right)P^m \right)
$$

$$
+ \frac{2}{|\mathcal{C}_i^r|} \sum_{\mathbf{x}\in\mathcal{C}_i^r} \left(\left(\sum_{i=1}^n c_i P^i - u(\mathbf{x},0) \right)P^m \right) + \frac{2}{|\mathcal{C}_i^r|} \sum_{\mathbf{x}\in\mathcal{C}_i^r} \left(\left(\sum_{i=1}^n c_i P_t^i - u_t(\mathbf{x},0) \right)P_t^m \right)
$$

for $m = 1,\ldots,n$. Here, for shortness of notation, we denote $\Psi_{\mathsf{NN}}^{\theta^i}(\mathbf{x},t)$ by $P^i(\mathbf{x},t)$ and omit $(\mathbf{x},t)$. For every full PINN P^i identified by GPT-PINN, we would then just need to store the values of

$$
P^i(\mathcal{C}_o^r \cup \mathcal{C}_\partial^r \cup (\mathcal{C}_i^r \times \{0\})), \quad P_{xx}^i(\mathcal{C}_o^r), \quad P_{tt}^i(\mathcal{C}_o^r), \quad P_t^i(\mathcal{C}_i^r \times \{0\})
$$

for efficient online GPT-PINN training step of eq. (7).

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] A. Barrett and G. Reddien. On the reduced basis method. *Z. Angew. Math. Mech.*, 75(7):543–549, 1995.

[3] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[4] P. Benner, S. Gugercin, and K. Willcox. A Survey of Projection-Based Model Reduction Methods for Parametric Dynamical Systems. *SIAM Review*, 57(4):483–531, jan 2015.

[5] P. Binev, A. Cohen, W. Dahmen, R. Devore, G. Petrova, and P. Wojtaszczyk. Convergence Rates for Greedy Algorithms in Reduced Basis Methods. *SIAM J. MATH. ANAL*, pages 1457–1472, 2011.

[6] W. Chen, Q. Wang, J. S. Hesthaven, and C. Zhang. Physics-informed machine learning for reduced-order modeling of nonlinear problems. *Journal of Computational Physics*, 446:110666, 2021.

[7] Y. Chen, S. Gottlieb, L. Ji, and Y. Maday. An eim-degradation free reduced basis method via over collocation and residual hyper reduction-based error estimation. *Journal of Computational Physics*, 444:110545, 2021.

[8] Y. Chen, J. Jiang, and A. Narayan. A robust error estimator and a residual-free error indicator for reduced basis methods. *Computers & Mathematics with Applications*, 77:1963–1979, 2019.

[9] G. Cybenko. Approximation by superpositions of a sigmoidal function. 2:303–314, 1989.

[10] W. E, J. Han, and A. Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. 5:349–380, 2017.

[11] W. E and B. Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. 6:1–12, 2018.

[12] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[13] S. Flennerhag, P. G. Moreno, N. D. Lawrence, and A. Damianou. Transferring knowledge across learning processes. *arXiv preprint arXiv:1812.01054*, 2018.

[14] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

[15] B. Haasdonk. *Chapter 2: Reduced Basis Methods for Parametrized PDEs - A Tutorial Introduction for Stationary and Instationary Problems*, pages 65–136.

[16] J. Han, A. Jentzen, and W. E. Solving high-dimensional partial differential equations using deep learning. 115:8505–8510, 2018.

[17] J. S. Hesthaven, G. Rozza, and B. Stamm. *Certified reduced basis methods for parametrized partial differential equations.* SpringerBriefs in Mathematics. Springer, Cham; BCAM Basque Center for Applied Mathematics, Bilbao, 2016. BCAM SpringerBriefs.

[18] Y. Khoo, J. Lu, and L. Ying. Solving for high-dimensional committor functions using artificial neural networks. 6, 2019.

[19] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. 9:987–1000, 1998.

[20] L. Lu, P. Jin, and G. E. Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv e-prints*, page arXiv:1910.03193, Oct. 2019.

[21] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature machine intelligence*, 3(3):218–229, 2021.

[22] Y. Maday, A. T. Patera, and D. V. Rovas. A blackbox reduced-basis output bound method for noncoercive linear problems. In *Nonlinear partial differential equations and their applications. Collège de France Seminar, Vol. XIV (Paris, 1997/1998)*, volume 31 of *Stud. Math. Appl.*, pages 533–569. North-Holland, Amsterdam, 2002.

[23] R. Mattey and S. Ghosh. A novel sequential method to train physics informed neural networks for allen cahn and cahn hilliard equations. *Computer Methods in Applied Mechanics and Engineering*, 390:114474, 2022.

[24] L. McClenny and U. Braga-Neto. Self-adaptive physics-informed neural networks using a soft attention mechanism. *arXiv preprint arXiv:2009.04544*, 2020.

[25] T. P. Miyanawala and R. K. Jaiman. An efficient deep learning technique for the navier-stokes equations: Application to unsteady wake flow dynamics. *arXiv e-prints*, page arXiv:1710.09099, Oct. 2017.

[26] D. A. Nagy. Modal representation of geometrically nonlinear behaviour by the finite element method. *Computers and Structures*, 10:683–688, 1979.

[27] A. K. Noor and J. M. Peters. Reduced basis technique for nonlinear analysis of structures. *AIAA Journal*, 18(4):455–462, apr 1980.

[28] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[29] A. T. Patera and G. Rozza. *Reduced Basis Approximation and A Posteriori Error Estimation for Parametrized Partial Differential Equations.* MIT, version 1.0 edition, 2007.

[30] M. Penwarden, S. Zhe, A. Narayan, and R. M. Kirby. Physics-informed neural networks (pinns) for parameterized pdes: a metalearning approach. *arXiv preprint arXiv:2110.13361*, 2021.

[31] P. Perdikaris, M. Raissi, A. Damianou, N. D. Lawrence, and G. E. Karniadakis. Nonlinear information fusion algorithms for data-efficient multi-fidelity modelling. 473:20160751, 2017.

[32] J. S. Peterson. The Reduced Basis Method for Incompressible Viscous Flow Calculations. *SIAM Journal on Scientific and Statistical Computing*, 10(4):777–786, 1989.

[33] A. Pinkus. *N-widths in approximation theory.* Springer, 1985.

[34] C. Prud'homme, D. Rovas, K. Veroy, Y. Maday, A. T. Patera, and G. Turinici. Reliable real-time solution of parametrized partial differential equations: Reduced-basis output bound methods. *Journal of Fluids Engineering*, 124(1):70–80, mar 2002.

[35] A. F. Psaros, K. Kawaguchi, and G. E. Karniadakis. Meta-learning pinn loss functions. *Journal of Computational Physics*, 458:111121, 2022.

[36] T. Qin, A. Beatson, D. Oktay, N. McGreivy, and R. P. Adams. Meta-pde: Learning to solve pdes quickly without a mesh. 2022.

[37] A. Quarteroni, A. Manzoni, and F. Negri. *Reduced Basis Methods for Partial Differential Equations*, volume 92 of *UNITEXT*. Springer International Publishing, Cham, 2016.

[38] M. Raissi. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *J. Mach. Learn. Res.*, 19:25:1–25:24, 2018.

[39] M. Raissi and G. E. Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, Mar. 2018.

[40] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[41] M. Raissi, A. Yazdani, and G. E. Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. 367:1026–1030, 2020.

[42] J. Revels, M. Lubin, and T. Papamarkou. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

[43] G. Rozza, D. B. P. Huynh, and A. T. Patera. Reduced basis approximation and a posteriori error estimation for affinely parametrized elliptic coercive partial differential equations: Application to transport and continuum mechanics. *Arch Comput Methods Eng*, 15(3):229–275, 2008.

[44] T. D. Ryck, S. Mishra, and D. Ray. On the approximation of rough functions with deep neural networks. Dec. 2019.

[45] S. Wang, S. Sankaran, and P. Perdikaris. Respecting causality is all you need for training physics-informed neural networks. *arXiv preprint arXiv:2203.07404*, 2022.

[46] C. L. Wight and J. Zhao. Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks. *arXiv preprint arXiv:2007.04542*, 2020.

[47] J. Xu, J. Zhao, and Y. Zhao. Numerical approximations of the allen-cahn-ohta-kawasaki (acok) equation with modified physics informed neural networks (pinns). *arXiv preprint arXiv:2207.04582*, 2022.

[48] D. Yarotsky. Error bounds for approximations with deep relu networks. 94:103–114, 2017.

[49] L. Zhang, H. You, T. Gao, M. Yu, C.-H. Lee, and Y. Yu. Metano: How to transfer your knowledge on learning hidden physics, 2023.

[50] L. Zhong, B. Wu, and Y. Wang. Accelerating physics-informed neural network based 1d arc simulation by meta learning. *Journal of Physics D: Applied Physics*, 56(7):074006, feb 2023.