TINNS: TIME-INDUCED NEURAL NETWORKS FOR SOLVING TIME-DEPENDENT PDES

Chen-Yang Dai[1], **Che-Chia Chang**[2], **Te-Sheng Lin**[1,3], **Ming-Chih Lai**[1], **Chieh-Hsin Lai**[1]
[1]Department of Applied Mathematics, National Yang Ming Chiao Tung University
[2]Institute of Artificial Intelligence Innovation, National Yang Ming Chiao Tung University
[3]National Center for Theoretical Science, National Taiwan University

ABSTRACT

Physics-informed neural networks (PINNs) solve time-dependent partial differential equations (PDEs) by learning a mesh-free, differentiable solution that can be evaluated anywhere in space and time. However, standard space–time PINNs take time as an input but reuse a single network with shared weights across all times, forcing the same features to represent markedly different dynamics. This coupling degrades accuracy and can destabilize training when enforcing PDE, boundary, and initial constraints jointly. We propose *Time-Induced Neural Networks (TINNs)*, a novel architecture that parameterizes the network weights as a learned function of time, allowing the effective spatial representation to evolve over time while maintaining shared structure. The resulting formulation naturally yields a nonlinear least-squares problem, which we optimize efficiently using a Levenberg–Marquardt method. Experiments on various time-dependent PDEs show up to $4\times$ improved accuracy and $10\times$ faster convergence compared to PINNs and strong baselines. Code is available at `https://github.com/CYDai-nycu/TINN`.

1 INTRODUCTION

Time-dependent partial differential equations (PDEs) are central to modeling transport, fluid flows, and reaction–diffusion phenomena. While classical numerical solvers based on discretization (e.g., finite differences and finite elements (LeVeque, 2007; Brenner & Scott, 2008)) are highly accurate, they can become costly when the domain geometry is complex, when solutions must be queried at arbitrary locations, or when the effective dimension grows due to parameters, uncertainty, or inverse settings. Physics-informed neural networks (PINNs) provide an appealing alternative by learning a surrogate solution that satisfies the governing equations through a constrained training objective (Raissi et al., 2019). By minimizing a weighted combination of PDE residuals and initial/boundary constraints at collocation points, PINNs yield a mesh-free, differentiable solution operator that can be queried anywhere in the domain (Cuomo et al., 2022; Luo et al., 2025).

Despite these successes, standard space–time PINNs still struggle to accurately model dynamics whose spatial complexity evolves over time. In the prevailing formulation, time is treated as an additional input coordinate and a single neural network $u_{\boldsymbol{\theta}}(\mathbf{x}, t)$ is trained over the entire space–time domain with a shared set of parameters $\boldsymbol{\theta}$. As a result, time influences the model only through input conditioning, while the deeper representation is reused across all temporal regimes. We refer to this structural limitation as the *time-entanglement problem*: qualitatively different regimes, such as smooth early-time behavior and sharp later-time transitions, must be represented by the same shared features. This causes representation interference and makes optimization difficult when jointly enforcing the PDE residual, boundary conditions, and initial conditions.

To build intuition, consider a toy parameterization in one spatial dimension with a single affine space–time feature,

$$u_{\boldsymbol{\theta}}(x, t) := U(wx + vt + b),$$

where U represents the remaining (possibly deep) nonlinear network, and $\boldsymbol{\theta} = (w, v, b)$ consists of three learnable scalars. Its spatial derivative is

$$\partial_x u_{\boldsymbol{\theta}}(x, t) = U'(wx + vt + b)\, w.$$

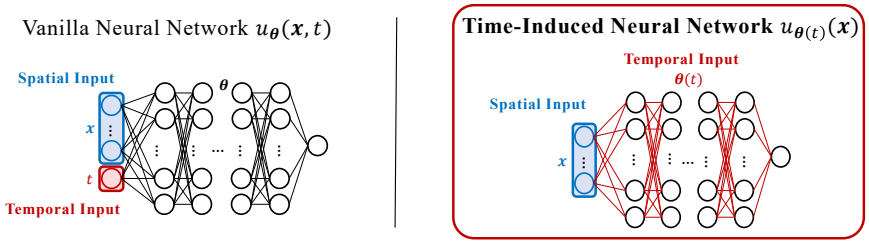

Figure 1: **Space–time PINN (time as input) vs. TINN (time in parameter space).** The left shows how a vanilla neural network structure deals with time-dependent PDEs. The time is incorporated into the input dimension. On the right-hand side, the time information in TINN is integrated into the parameter space, making it easier to capture the complex dynamics of the system.

This exposes the bottleneck: time affects the network only through an additive shift vt, while the spatial scaling w is fixed for all t. Therefore, the model cannot directly increase spatial steepness over time by rescaling spatial features. Instead, any steepening must be achieved indirectly by shifting the argument of $U'(\cdot)$. For example, at $x = 0$ the slope changes from $U'(b)w$ at $t = 0$ to $U'(v + b)w$ at $t = 1$. This shift-only control is brittle in practice, since large changes in spatial gradients must be produced by moving across activation regimes. Deeper networks may increase expressivity, but the same coupling persists: time still modulates the solution through shared features, so evolving spatial scales remain implicit rather than explicitly controlled. Existing methods address these issues mainly via optimization (Wang et al., 2023; Bihlo, 2024; Wang et al., 2025) and training heuristics, including adaptive sampling, loss reweighting, and causality-inspired curricula that prioritize earlier times (McClenny & Braga-Neto, 2023; Wang et al., 2024a;b; Bischof & Kraus, 2025). While often effective, they retain the global form $u_{\boldsymbol{\theta}}(\mathbf{x}, t)$, forcing one shared representation across all times and handling temporal nonstationarity only through input conditioning.

To address the *time-entanglement problem*, we propose *Time-Induced Neural Networks (TINNs)*, which represent a time-dependent solution as a trajectory in *parameter space*:

$$u_{\boldsymbol{\theta}(t)}(\mathbf{x}),$$

with smoothly varying parameters $\boldsymbol{\theta}(t)$. Unlike standard space–time PINNs that fit all time with a single parameter vector, TINNs learn a mapping $t \mapsto \boldsymbol{\theta}(t)$, producing a family of time-indexed spatial networks (see in Figure 1). This explicit temporal evolution allows spatial features and gradients to adapt over time, reducing representation interference and enabling accurate solutions with compact models. Empirically, TINNs achieve strong accuracy with compact models without uniformly increasing capacity.

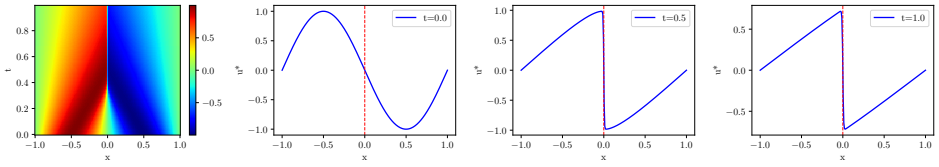

Figure 2: **Motivation for TINNs on viscous Burgers. Left:** space–time contour of the exact solution on $x \in [-1, 1]$, $t \in [0, 1]$. **Right:** solution profiles at $t \in \{0, 0.5, 1.0\}$. The solution evolves from a smooth profile to a thin transition layer near $x = 0$, which is challenging for a single space–time network with shared parameters and motivates TINN's time-modulated parameterizations.

Building on TINNs' flexibility, we adopt an efficient training procedure for the resulting objective. Because the loss is a nonlinear least-squares problem (PDE residual with boundary/initial constraints), we use a Gauss–Newton–style optimizer via Levenberg–Marquardt (LM) (Levenberg, 1944; Marquardt, 1963). This second-order update improves stability and balances competing constraints better than first-order methods.

Our contributions are: (i) we propose *Time-Induced Neural Networks (TINNs)*, a time-induced parameterization with practical constructions of $\boldsymbol{\theta}(t)$ to deal with *time-entanglement problem* in space–time PINNs; (ii) we employed an LM-based optimizer that exploits the resulting nonlinear least-squares structure; and (iii) we validate TINNs on benchmark time-dependent PDEs, showing improved accuracy and stability over strong baselines. The rest of the paper is organized as: Sec-

tion 2 reviews PINNs and related work. Section 3 introduces TINNs and our parameterization of $\boldsymbol{\theta}(t)$. Section 4 presents experiments, and Section 5 discusses mechanisms. Section 6 concludes.

2 Preliminary and Related Work

2.1 Physics-Informed Neural Networks (PINNs)

Consider a time-dependent PDE posed on a spatial domain $\Omega \subset \mathbb{R}^d$ and a time interval $[0, T]$:

$$\begin{cases} \mathcal{L}(u) = 0, & \text{in } \Omega \times [0,T], \\ \mathcal{B}(u) = 0, & \text{on } \partial\Omega \times [0,T], \\ \mathcal{I}(u) = 0, & \text{on } \Omega \times \{0\}, \end{cases}$$

where $\mathcal{L}$ is a differential operator (possibly nonlinear), and $\mathcal{B}$ and $\mathcal{I}$ denote the boundary and initial condition operators, respectively. Physics-informed neural networks (Raissi et al., 2019) approximate the solution by a neural network $u_{\boldsymbol{\theta}}(\mathbf{x}, t)$ and enforce these constraints by minimizing a weighted sum of squared residuals evaluated at collocation points:

$$L(\boldsymbol{\theta}) := \frac{\lambda_{\mathrm{r}}}{N_{\mathrm{r}}} \sum_{i=1}^{N_{\mathrm{r}}} \left\| \mathcal{L}(u_{\boldsymbol{\theta}})(\mathbf{x}_i^{\mathrm{r}}, t_i^{\mathrm{r}}) \right\|_2^2 + \frac{\lambda_{\mathrm{b}}}{N_{\mathrm{b}}} \sum_{i=1}^{N_{\mathrm{b}}} \left\| \mathcal{B}(u_{\boldsymbol{\theta}})(\mathbf{x}_i^{\mathrm{b}}, t_i^{\mathrm{b}}) \right\|_2^2 + \frac{\lambda_{\mathrm{ic}}}{N_{\mathrm{ic}}} \sum_{i=1}^{N_{\mathrm{ic}}} \left\| \mathcal{I}(u_{\boldsymbol{\theta}})(\mathbf{x}_i^{\mathrm{ic}}, 0) \right\|_2^2, \quad (1)$$

where $\lambda_{\mathrm{r}}, \lambda_{\mathrm{b}}, \lambda_{\mathrm{ic}} \geq 0$ are penalty weights for the residual, boundary condition, and initial condition terms of the PDE, respectively. The collocation sets $\{(\mathbf{x}_i^{\mathrm{r}}, t_i^{\mathrm{r}})\}_{i=1}^{N_{\mathrm{r}}} \subset \Omega \times [0,T]$, $\{(\mathbf{x}_i^{\mathrm{b}}, t_i^{\mathrm{b}})\}_{i=1}^{N_{\mathrm{b}}} \subset \partial\Omega \times [0,T]$, and $\{(\mathbf{x}_i^{\mathrm{ic}}, 0)\}_{i=1}^{N_{\mathrm{ic}}} \subset \Omega \times \{0\}$ correspond to the constraint of the residual, boundary, and initial condition of the PDE, respectively.

For time-dependent problems, standard space–time PINNs treat t as an additional input dimension and parameterize the solution with a single time-independent parameter vector $\boldsymbol{\theta}$ shared across all time steps, i.e., a network $u_{\boldsymbol{\theta}} : \mathbb{R}^{d+1} \to \mathbb{R}$, trained by minimizing Equation (1) over the entire space–time domain.

2.2 Related Work

For time-dependent PDEs, several PINN variants have been developed to improve training stability over long time horizons. Causal PINNs (Wang et al., 2024b) address error propagation by applying a time-adaptive loss reweighting that prioritizes early-time accuracy, since inaccuracies near small t can accumulate and degrade the solution at later times. Despite this improvement in optimization, the method still relies on a single space–time network that treats time as an additional input coordinate, which can limit representational efficiency for complex temporal dynamics.

Another direction explicitly separates space and time (Datar et al., 2024) by using a fixed set of spatial features (often randomly initialized) and learning only time-varying coefficients on top of them. The temporal evolution is then advanced with a classical ODE solver, which can be fast and accurate; however, this hybrid design departs from fully end-to-end PINN training and typically requires additional mechanisms to enforce initial and boundary conditions.

In contrast, we adopt a space–time separation viewpoint while retaining continuous, physics-informed, end-to-end learning. We parameterize the solution with a time-dependent neural representation and optimize it directly using the standard PINN objective, without freezing spatial features or relying on discrete time integration.

3 Time-Induced Neural Networks (TINNs)

3.1 Limitation with Vanilla Network Parametrizations

As discussed in the introduction, standard space–time PINNs suffer from the *time-entanglement problem*: temporal evolution is entangled with spatial features via additive shifts in activation arguments, making evolving spatial scales difficult to represent explicitly. Here, we illustrate this limitation on a concrete time-dependent PDE, and show how it can be addressed by our proposed TINN, which introduces time-dependent network parameters.

To illustrate the missing degree of freedom, we consider a toy TINN in one spatial dimension with a single affine space–time feature,

$$u_{\boldsymbol{\theta}(t)}(x) := U\big(w(t)x + b(t)\big),$$

where U denotes the remaining network and the parameters $\boldsymbol{\theta}(t)$ vary smoothly with time. Its spatial derivative is

$$\partial_x u_{\boldsymbol{\theta}(t)}(x) = w(t)\, U'\big(w(t)x + b(t)\big).$$

Unlike vanilla space–time networks, temporal evolution can now modify spatial steepness directly through the time-dependent scaling $w(t)$, rather than indirectly through shifts inside $U'(\cdot)$. This explicit control of spatial scales is precisely what is absent in standard PINN parameterizations.

The time-entanglement problem becomes evident, for instance, in viscous Burgers' equation (defined in Appendix A): the solution evolves from a smooth profile into a thin yet continuous transition layer. As shown in Figure 2, the overall shape remains similar while spatial gradients increase markedly over time. Since time affects the representation only through input conditioning, standard space–time PINNs can express this steepening only implicitly via temporal shifts, which is difficult in practice.

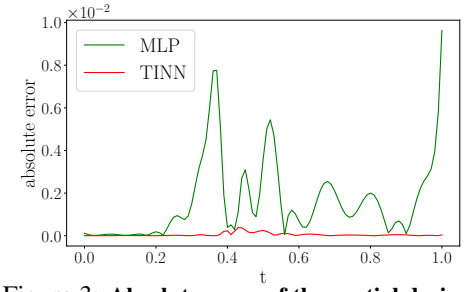

Figure 3: **Absolute error of the spatial derivative at $x = 0$ over $t \in [0, 1]$: vanilla MLP vs. TINN.** With comparable parameter counts, TINN yields smaller and more stable errors. See details in Appendix D.1.

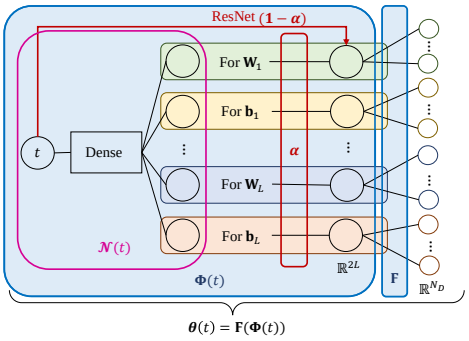

Figure 4: **Architecture for time-dependent parameters $\boldsymbol{\theta}(t)$ in TINN.** A small dense network $\boldsymbol{\mathcal{N}}(t)$ outputs a $2L$-dimensional code $\boldsymbol{\Phi}(t)$, which is lifted to the full parameter vector $\boldsymbol{\theta}(t) \in \mathbb{R}^{N_D}$ via an entrywise affine map, avoiding direct prediction of an N_D-dimensional output.

We additionally report the absolute error of the spatial derivative at $x = 0$ over $t \in [0, 1]$ in Figure 3. Since no closed-form derivative is available, we compute a high-accuracy reference solution of the viscous Burgers equation using `Chebfun` package (Driscoll et al., 2014) and obtain its derivative via spectral differentiation. For both the vanilla multilayer perceptron (MLP) and TINN with comparable model sizes, we compute derivatives via automatic differentiation.

As shown in Figure 3, the derivative error for the MLP increases sharply after the shock-like transition forms, reflecting its difficulty in resolving increasingly steep spatial gradients at later times. In contrast, TINN maintains a substantially smaller and more stable error throughout the entire time interval. This behavior highlights a structural limitation of standard space–time PINNs: with a single shared representation across time, they struggle to adapt to evolving spatial scales. Allowing spatial features to evolve explicitly over time, as in TINN, alleviates this issue. Indeed, the time-entanglement problem can also arise in modern space–time parameterizations beyond MLPs (He et al., 2016; Wang et al., 2021; Cho et al., 2023); we defer the discussion to Appendix B.1.

3.2 Designing Time-Embedding for Efficient Training

To separate temporal evolution from spatial representation, we remove time from the network input and instead model the solution trajectory through time-dependent parameters. Specifically, we parameterize a time-dependent solution as

$$u(\mathbf{x}, t) := u_{\boldsymbol{\theta}(t)}(\mathbf{x}), \tag{2}$$

where $u_{\boldsymbol{\theta}} : \Omega \subset \mathbb{R}^d \to \mathbb{R}$ is a spatial neural network and $\boldsymbol{\theta}(t)$ varies smoothly over time. In this view, the backbone $u_{\boldsymbol{\theta}}$ captures spatial structure, while $\boldsymbol{\theta}(t)$ encodes the temporal dynamics as a trajectory in parameter space.

Why Naive Hypernetworks of $\boldsymbol{\theta}(t)$ Are Expensive? A key challenge in Equation (2) is to construct $\boldsymbol{\theta}(t)$ without introducing a prohibitive number of additional parameters. To make this concrete, consider an L-layer MLP,

$$u_{\boldsymbol{\theta}(t)}(\mathbf{x}) = \mathbf{W}_L \sigma\Big(\mathbf{W}_{L-1}\sigma\big(\cdots\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\big) + \mathbf{b}_{L-1}\Big) + \mathbf{b}_L,$$

parameterized by $\boldsymbol{\theta}(t) = \{(\mathbf{W}_\ell, \mathbf{b}_\ell)\}_{\ell=1}^{L}$, where $\sigma(\cdot)$ is a component-wise applied nonlinear activation, $\mathbf{W}_\ell \in \mathbb{R}^{l_\ell \times l_{\ell-1}}$ and $\mathbf{b}_\ell \in \mathbb{R}^{l_\ell}$ are time-dependent, with l_0 denoting the input (spatial) dimension. The total number of parameters is

$$N_D := \sum_{\ell=1}^{L} \big(l_{\ell-1} l_\ell + l_\ell\big).$$

A naive implementation of Equation (2) uses a fully-connected network to output $\boldsymbol{\theta}(t) \in \mathbb{R}^{N_D}$. If this time network has hidden width h, then its final layer alone requires $\mathcal{O}(N_D h)$ parameters. Since N_D grows with the width of the spatial backbone (and typically increases with the PDE dimension through the input layer), this overhead can quickly dominate the overall model size and training cost.

This motivates restricting the temporal parameterization of $\boldsymbol{\theta}(t)$ to reduce dimensionality while retaining sufficient expressive power. A lightweight baseline is to impose a simple functional form on $\boldsymbol{\theta}(t)$. For example, one may assume a *linear trajectory* $\boldsymbol{\theta}(t) = \boldsymbol{w}t + \boldsymbol{b}$, or a *one-neuron trajectory* $\boldsymbol{\theta}(t) = \boldsymbol{w}_2 \sigma(\boldsymbol{w}_1 t + \boldsymbol{b})$, where $\boldsymbol{w}, \boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{b} \in \mathbb{R}^{N_D}$. These choices substantially reduce the parameter count (roughly $2N_D$ and $3N_D$, respectively), but are often too restrictive to capture heterogeneous temporal behaviors across PDEs. As shown in Table 1, the linear trajectory works well for Allen–Cahn, while both linear and one-neuron behave similarly for Burgers, motivating a more flexible yet still compact design.

Table 1: **Simple parametric forms for $\boldsymbol{\theta}(t)$ vs. TINN.** For fair comparison, we fix training points, iterations, optimizer settings, and random seeds, and match parameter budgets across methods. Linear and one-neuron trajectories are similar on Burgers, while linear is better on Allen–Cahn. Our proposed compact layer-wise construction of $\boldsymbol{\theta}(t)$ (Equation (3)) achieves the best performance.

Case	Rel L^2-Error (↓)	# Params.	# Neurons
Burgers			
linear	2.65E-06	1144	22
one-neuron	2.93E-06	1188	18
TINN's $\boldsymbol{\theta}(t)$	**5.67E-07**	1145	20
Allen–Cahn			
linear	3.25E-06	1188	22
one-neuron	1.47E-05	1242	18
TINN's $\boldsymbol{\theta}(t)$	**2.73E-06**	1185	20

Taken together, these observations suggest two modeling requirements for the design of $\boldsymbol{\theta}(t)$. First, it should exhibit *macro-level coherence*: If the exact solution varies sharply around $t^\star$, then $\boldsymbol{\theta}(t)$ should be able to represent a correspondingly sharp, coordinated change near $t^\star$. Second, it should allow *micro-level diversity*: even near $t^\star$, layers should remain distinguishable and be allowed to evolve differently over time.

Proposed Alternative: Compact Layer-wise Time Embedding. To achieve both goals while avoiding a fully-connected network with an N_D-dimensional output, we propose a compact layer-wise code $\boldsymbol{\Phi}(t) \in \mathbb{R}^{2L}$ ($2L-1$ when omitting the output bias), and lift it to the full parameter vector $\boldsymbol{\theta}(t) \in \mathbb{R}^{N_D}$ via an entrywise affine map. This yields *macro-level coherence* (many parameters change in a coordinated manner when the solution varies sharply in time) while preserving *micro-level diversity* (different layers follow different temporal scalings). More precisely, we define the layer-wise embedding $\boldsymbol{\Phi}(t)$ as:

$$\boldsymbol{\Phi}(t) = (\mathbf{1} - \boldsymbol{\alpha})\, t \;+\; \boldsymbol{\alpha} \odot \boldsymbol{\mathcal{N}}(t),$$

where $\boldsymbol{\mathcal{N}} : \mathbb{R} \to \mathbb{R}^{2L}$ is a small learnable network, $\boldsymbol{\alpha} \in \mathbb{R}^{2L}$ is a learnable gate, $\odot$ denotes element-wise multiplication, and $\mathbf{1}$ is the all-ones vector. We associate $\boldsymbol{\Phi}_{2\ell-1}(t)$ and $\boldsymbol{\Phi}_{2\ell}(t)$ with the ℓ-th layer weight and bias groups $\mathbf{W}_\ell(t)$ and $\mathbf{b}_\ell(t)$, respectively. Given $\boldsymbol{\Phi}(t)$, we construct the full parameter trajectory via a linear-affine lifting map $\mathbf{F}$ applied within each parameter group. Within a group, all entries share the same temporal coordinate of $\boldsymbol{\Phi}(t)$, while retaining independent affine coefficients. For example, for $\mathbf{W}_1(t) = \{w_1^{ij}(t)\}_{i,j}$, we define

$$w_1^{ij}(t) = a_{w_1}^{ij}\, \boldsymbol{\Phi}_1(t) + b_{w_1}^{ij},$$

and define the remaining weights and biases analogously. Equivalently, these entrywise affine rules together define a global mapping

$$\boldsymbol{\theta}(t) \;=\; \mathbf{F}\big(\boldsymbol{\Phi}(t)\big), \tag{3}$$

where $\mathbf{F}$ stacks all affine coefficients $\{a^{ij}, b^{ij}\}$ across layers and parameter entries. Thus, the learnable variables consist of the embedding network $\boldsymbol{\mathcal{N}}$, the gate $\boldsymbol{\alpha}$, and the affine map $\mathbf{F}$. To avoid ambiguity, we denote by $\boldsymbol{\psi}$ the collection of all trainable parameters in $\{\boldsymbol{\mathcal{N}}, \boldsymbol{\alpha}, \mathbf{F}\}$, and by $\boldsymbol{\theta}(t)$ the induced time-dependent parameters of the spatial backbone. Once $\boldsymbol{\psi}$ is trained, we can instantiate $\boldsymbol{\theta}(t)$ (and hence $u_{\boldsymbol{\theta}(t)}$) at any time t. Figure 4 illustrates the proposed construction.

Parameter Efficiency: Layer-wise Embedding vs. Naive Hypernetworks. We now compare the parameter complexity of these two approaches. For a naive hypernetwork with hidden width h, the final layer alone requires $\mathcal{O}(N_D h)$ parameters to output $\boldsymbol{\theta}(t) \in \mathbb{R}^{N_D}$.

In contrast, the TINN time network $\boldsymbol{\mathcal{N}}(t)$ outputs the layer-wise code $\boldsymbol{\Phi}(t) \in \mathbb{R}^{2L}$ and has $\mathcal{O}(Lh)$ parameters (for fixed depth). The subsequent entrywise affine lift $\boldsymbol{\theta}(t) = \mathbf{F}(\boldsymbol{\Phi}(t))$ introduces two coefficients per backbone entry, contributing $2N_D$ additional parameters. The total number of trainable parameters therefore scales as

$$\#\,\text{Params.} \;=\; 2N_D \;+\; \mathcal{O}(Lh),$$

which is substantially smaller than $\mathcal{O}(N_D h)$ in practice, since typically $N_D \gg L$ and $h > 2$. A concrete comparison is provided in Appendix D.1. In this work, we use an MLP backbone, following common practice in the PINN literature. Our approach is not specific to MLPs and can be extended to other architectures; see Appendix B.2.

3.3 Fast TINN Training via Levenberg–Marquardt

Training PINNs naturally yields a *nonlinear least-squares* problem, in which the objective is a weighted sum of squared residuals of the governing PDE together with the initial and boundary conditions, evaluated at collocation points. TINN adopts the same physics-informed loss as standard space–time PINNs; the only difference is the parameterization $u_{\boldsymbol{\theta}(t)}(\mathbf{x})$, ensuring a fair comparison across methods.

Most PINN approaches rely on first-order optimizers such as Adam (Kingma & Ba, 2015) or quasi-Newton methods like L-BFGS (Liu & Nocedal, 1989), which do not explicitly exploit the least-squares structure and can be sensitive to scale mismatches among residual terms. While second-order methods are appealing in principle, naive Newton updates are often unstable due to poor conditioning.

We therefore adopt the Levenberg–Marquardt (LM) algorithm (Levenberg, 1944; Marquardt, 1963; Hu et al., 2024), a standard second-order method for nonlinear least-squares problems that interpolates between gradient descent and Gauss–Newton updates via adaptive damping. In our setting, LM operates on the physics-informed objective by forming and solving a sequence of damped linearized subproblems based on the stacked PDE, boundary, and initial residuals (see Appendix C for details).

Although LM can in principle be applied to general PINN parameterizations, its per-iteration cost scales with the number of trainable parameters, making it impractical for the large networks commonly used in recent PINN baselines (Duan et al., 2025; Wang et al., 2024a). By contrast, TINN is parameter-efficient yet expressive, keeping LM tractable for the moderately sized models considered here. Empirically, this combination yields faster and more stable convergence.

4 Numerical Results

4.1 Experimental Setup

To evaluate TINNs on complex temporal dynamics, we consider five time-dependent PDEs: Burgers, Allen–Cahn, Klein–Gordon, Korteweg–De Vries, and the wave equation (see Appendix A for details). All models are trained by minimizing the PINN objective in Equation (1).

Implementation of TINNs. To isolate the effect of the TINN parameterization, we use a unified training protocol across methods and avoid problem-specific heuristics. In particular, we do not employ random Fourier features (Tancik et al., 2020), constraint-preserving initializations (Wang et al., 2024a), causal/time-marching training (Wang et al., 2024b), or related techniques. This enables a direct and fair comparison to standard PINNs and baselines.

We instantiate TINN with a compact spatial backbone to improve parameter efficiency relative to standard space–time PINNs. The spatial network is a fully connected MLP with two hidden layers of width 20. Time dependence is introduced via an auxiliary network $\boldsymbol{\mathcal{N}}(t)$, implemented as a two-hidden-layer MLP with width 10, which outputs the layer-wise embedding used to modulate backbone parameters. Both networks use $\tanh$ activations, and we omit the bias in the final output layer. The output dimension of $\boldsymbol{\mathcal{N}}(t)$ matches the number of parameter groups in the backbone, which is five in our setup (weights and biases of the two hidden layers, and the output-layer weights).

The backbone input dimension matches the spatial domain dimension. For periodic boundary conditions (Allen–Cahn and Korteweg–De Vries), we additionally apply periodic input embeddings,

Table 2: **Results on various time-dependent PDEs.** All experiments run on a single NVIDIA A6000 GPU. We report relative L^2-error (mean over 5 runs), training time, and trainable parameters. **Bold** and underlined denote the best and second-best errors, respectively.

Case	Rel L^2-Error (↓)	Time (↓)	# Params.	acc. IMP
Burgers				
PINN (Raissi et al., 2019)	2.19E-04 ± 1.65E-04	1.24hr	309440	318×
CoPINN* (Duan et al., 2025)	6.23E-05 ± 2.74E-05	0.78hr	336864	90×
PirateNet SOAP (Wang et al., 2025)	1.97E-06 ± 5.13E-07	1.70hr	534853	2.9×
TINN (ours)	**6.89E-07± 3.97E-07**	**0.75hr**	1145	–
Allen-Cahn				
PINN (Raissi et al., 2019)	4.65E-01 ± 3.62E-01	0.95hr	309760	1.21E+05×
CoPINN* (Duan et al., 2025)	1.29E-04 ± 4.33E-05	0.80hr	337464	33×
PirateNet SOAP (Wang et al., 2025)	8.32E-06 ± 3.00E-06	1.50hr	534981	2.2×
TINN (ours)	**3.85E-06 ± 1.48E-06**	**0.78hr**	1185	–
Klein-Gordon				
PINN (Raissi et al., 2019)	4.04E-03 ± 7.75E-03	1.53hr	309760	845×
CoPINN* (Duan et al., 2025)	6.61E-06 ± 4.84E-06	0.70hr	212832	1.4×
PirateNet SOAP (Wang et al., 2025)	1.88E-05 ± 6.11E-07	3.31hr	379281	3.9×
TINN (ours)	**4.78E-06 ± 2.63E-06**	**0.67hr**	1185	–
Korteweg-De Vries				
PINN (Raissi et al., 2019)	2.47E-02 ± 9.93E-03	1.91hr	309760	161×
CoPINN* (Duan et al., 2025)	1.05E-01 ± 5.45E-02	1.14hr	337464	686×
PirateNet SOAP (Wang et al., 2025)	4.26E-04 ± 5.29E-05	1.86hr	534981	2.8×
TINN (ours)	**1.53E-04 ± 4.73E-05**	**0.69hr**	1185	–
Wave				
PINN (Raissi et al., 2019)	5.01E-02 ± 1.89E-02	1.14hr	309440	7.47E+03×
CoPINN* (Duan et al., 2025)	3.89E-03 ± 1.67E-03	0.78hr	336864	580×
PirateNet SOAP (Wang et al., 2025)	2.88E-05 ± 8.89E-06	1.89hr	534853	4.3×
TINN (ours)	**6.71E-06 ± 7.65E-06**	**0.77hr**	1145	–

which add two neurons to the second layer of the spatial network. Additional implementation details are provided in Appendix D.3.

Baselines. We compare TINNs against several representative baselines. *PINNs* (Raissi et al., 2019) serve as the vanilla baseline and are trained with Adam (Kingma & Ba, 2015) using a decaying learning-rate schedule.

CoPINN (Duan et al., 2025) extends SPINN (Cho et al., 2023) by dynamically reweighting samples based on estimated difficulty. We denote our implementation as *CoPINN**, as the released code is not optimized for best performance; we add learning-rate decay and high-precision matrix multiplication to further improve results.

PirateNet (Wang et al., 2024a) combines deep residual architectures with training enhancements (e.g., random Fourier features, causal training, and PDE-informed initialization). Following prior work, we train PirateNet using the improved second-order optimizer SOAP (Wang et al., 2025).

Measurements. We evaluate the relative L^2-error, training runtime, network size, and the accuracy improvement (IMP). IMP measures how many times it is improved in relative L^2-error achieved by TINNs over other non-TINN baselines (see Appendix D.2).

4.2 Comparison with Baselines

The main results are summarized in Table 2. For a fair comparison, we implement all methods in `JAX` and evaluate them under a *matched wall-clock training budget*, set to the runtime of TINNs. For each method, we report the final model obtained within this time budget. Methods with lower per-iteration cost therefore execute more optimization steps under the same budget; this includes PINN and CoPINN*, the latter benefiting from a more efficient automatic differentiation structure. For PirateNet, we follow the original experimental protocol and use the iteration count reported in the corresponding work, as its training procedure is not directly comparable under a wall-clock–matched setting. Across all tested PDEs, TINNs achieve the highest accuracy while using

Table 3: **Ablation of TINNs trained with Adam.** TINNs achieve the best accuracy across all cases, while using fewer parameters than PirateNet.

Case	Rel L^2-Error	Time	# Params.
Burgers			
PirateNet	2.43E-05	1.16hr	534853
TINN	**2.15E-05**	1.12hr	283037
Allen-Cahn			
PirateNet	2.32E-03	0.99hr	534981
TINN	**8.29E-05**	0.91hr	283165
Klein-Gordon			
PirateNet	5.60E-05	2.90hr	379281
TINN	**4.98E-05**	2.78hr	107384

substantially fewer trainable parameters than competing methods. For instance, TINNs improve accuracy by $2.9\times$ on Burgers and $2.2\times$ on Allen–Cahn. Moreover, when trained with LM, TINNs converge markedly faster than the strongest baseline, PirateNet optimized with SOAP. On the Burgers equation, TINNs reach comparable or better accuracy with a $10.55\times$ speedup over PirateNet with SOAP, and achieve a $2.30\times$ speedup on Allen–Cahn (see more detail in Appendix D.3 and Table 8).

Finally, although $\boldsymbol{\theta}(t)$ is parameterized by a neural network, computing higher-order temporal derivatives incurs no noticeable overhead in practice. In particular, evaluating $\partial^2_{tt} u_{\boldsymbol{\theta}(t)}$ has comparable cost to $\partial_t u_{\boldsymbol{\theta}(t)}$: 30K training iterations require 0.75 hours for Burgers (using $\partial_t u_{\boldsymbol{\theta}(t)}$) and 0.77 hours for the wave equation (using $\partial^2_{tt} u_{\boldsymbol{\theta}(t)}$).

4.3 Ablation Study

We further evaluate TINNs with Adam to decouple architectural gains from optimizer choice. While LM is most practical for small architectures, larger TINNs can be trained with Adam. In Table 3, we train larger TINNs with Adam and compare against PirateNet under matched engineering choices. TINNs remain competitive on Burgers, Allen–Cahn, and Klein–Gordon, indicating that the gains stem from the time-induced parameterization and are not specific to LM.

Table 4: **Ablation of architectural choices under LM training.** TINNs achieve the best accuracy across all PDEs under similar model size and training time, highlighting the benefit of the time-induced parameterization.

Case	Rel L^2-Error	Time	# Params.
Burgers			
subMLP	7.11E-05	0.75hr	500
MLP	1.92E-05	0.82hr	1160
TINN	**6.89E-07**	0.75hr	1145
Allen-Cahn			
subMLP	1.76E-03	0.80hr	520
MLP	7.14E-06	0.79hr	1202
TINN	**3.85E-06**	0.78hr	1185
Klein-Gordon			
subMLP	2.84E-05	0.68hr	520
MLP	5.11E-06	0.68hr	1202
TINN	**4.78E-06**	0.67hr	1185

Next, we run an LM-based ablation over compact architectural choices; results are summarized in Table 4. We include a baseline, *subMLP*, which uses the same spatial backbone as TINNs but treats time t as an additional input, i.e., a standard space–time MLP. Given TINNs' greater representational flexibility, we expect better performance under the same optimizer (see Section 5). Implementation details are deferred to Appendix D.4.

5 Mechanism of TINNs

PINNs as a Restricted Case of TINNs. Standard PINNs for time-dependent PDEs parameterize the solution with a single MLP $u_{\mathrm{M}}(\mathbf{x}, t)$ taking $(d{+}1)$-dimensional inputs, where d is the spatial dimension and time is treated as an additional coordinate. This can be viewed as a special case of our TINN framework, as formalized below.

Proposition 5.1. *Let $u_M(\mathbf{x}, t)$ be a MLP with $(d+1)$-dimensional input. Then u_M can be viewed as a special case of a TINN in which time dependence appears only in the bias of the first layer.*

This can be interpreted as follows: for a fixed model size, the TINN function class strictly contains that of a vanilla space–time MLP; hence, TINNs are strictly more expressive. To see this, consider a standard space–time PINN parameterized by an L-layer MLP with input $(\mathbf{x}, t) \in \mathbb{R}^{d+1}$. Its first hidden layer takes the form

$$\mathbf{z} = \sigma\big(\mathbf{W}_1^x \mathbf{x} + \mathbf{W}_1^t t + \mathbf{b}_1\big)\,,$$

where $\mathbf{W}_1^x \in \mathbb{R}^{l_1 \times d}$ is the spatial weight matrix, $\mathbf{W}_1^t \in \mathbb{R}^{l_1}$ is the time weight vector, and $\mathbf{b}_1 \in \mathbb{R}^{l_1}$ is the bias. The temporal contribution can be absorbed into a time-dependent bias, $\mathbf{b}_1(t) := \mathbf{b}_1 + \mathbf{W}_1^t t$, while all weight matrices remain time-independent and all subsequent layers receive no explicit temporal input.

In a TINN, each scalar parameter is parameterized in the form $a\,\boldsymbol{\Phi}_k(t) + b$. By setting $\boldsymbol{\Phi}_k(t) = t$ for the first-layer bias and choosing $a = 0$ for all weights and for all parameters in layers $\ell > 1$, the TINN reduces to a vanilla space–time PINN. Therefore, standard PINNs are a highly restricted instance of TINNs, where temporal dependence is confined to the first-layer bias term.

TINNs Provide Flexibility in Minimizing Equation (1). In standard space–time PINNs, the PDE residual, boundary conditions, and initial conditions in Equation (1) are enforced simultaneously using a single network $u_{\boldsymbol{\theta}}(\mathbf{x}, t)$. As seen in the toy example, time enters only as an input, so all

time slices share the same parameterization $\boldsymbol{\theta}$. This entangles temporal variation with spatial features, making time-varying spatial scales/gradients difficult to represent. We now further show that the same coupling also makes it challenging to satisfy the PDE residual and the initial/boundary constraints simultaneously.

To illustrate this limitation, consider the PDE $\mathcal{L}u = 0$ on $(x,t) \in [-1,1] \times [0,T]$ with the following initial and boundary conditions:

$$\begin{cases} u(x,0) = u_{\text{ic}}(x), & x \in [-1,1], \\ u(x,t) = u_{bc}(x,t), & (x,t) \in \{-1,1\} \times [0,T], \end{cases}$$

where $\mathcal{L}$ is a differential operator. Suppose a vanilla neural network takes the form $U(\mathbf{W}_1^x x + \mathbf{W}_1^t t + \mathbf{b}_1)$, where $\mathbf{W}_1^x$, $\mathbf{W}_1^t$, $\mathbf{b}_1$ are the parameters of the first layer and U represents the remaining network. Imposing the initial and boundary conditions requires

$$\begin{cases} U(\mathbf{W}_1^x x + \mathbf{b}_1) = u_{\text{ic}}(x), & x \in [-1,1], \\ U(\mathbf{W}_1^x + \mathbf{W}_1^t t + \mathbf{b}_1) = u_{\text{bc}}(1,t), & t \in [0,T], \\ U(-\mathbf{W}_1^x + \mathbf{W}_1^t t + \mathbf{b}_1) = u_{\text{bc}}(-1,t), & t \in [0,T]. \end{cases}$$

The distinction between the initial condition and the boundary conditions (and between $x = 1$ and $x = -1$) is introduced only through the first-layer affine inputs $\mathbf{W}_1^x x + \mathbf{W}_1^t t + \mathbf{b}_1$. All subsequent layers share the same parameters and must jointly satisfy all constraints through a single representation, limiting the model's flexibility to accommodate different components of the PINN loss.

In contrast, a TINN represents the solution as $u_{\boldsymbol{\theta}(t)}(x)$ with time-dependent parameters $\boldsymbol{\theta}(t)$. The initial and boundary constraints become

$$u_{\boldsymbol{\theta}(0)}(x) = u_{\text{ic}}(x), \qquad u_{\boldsymbol{\theta}(t)}(\pm 1) = u_{\text{bc}}(\pm 1, t).$$

Unlike a space–time PINN, which must satisfy all constraints using a single shared parameterization, TINN allows *all* layers to adapt over time through $\boldsymbol{\theta}(t)$. This decoupling provides additional flexibility to fit the initial and boundary constraints while simultaneously reducing the PDE residual, leading to a more flexible optimization of Equation (1).

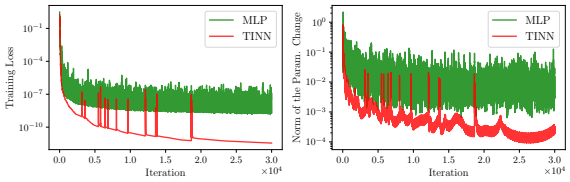

Figure 5: **Training stability on viscous Burgers: vanilla PINN vs. TINN. Left:** training loss. **Right:** ℓ_2-norm of parameter updates. Despite resampling spikes, vanilla PINN shows high-variance loss and irregular parameter updates, whereas TINN converges more smoothly with more regular update magnitudes.

TINNs Exhibit Substantially Reduced Overfitting. On the viscous Burgers equation, whose solution develops a sharp shock-like transition, we monitor generalization using a validation loss evaluated on held-out collocation points. While vanilla space–time PINNs exhibit a persistent train–validation gap and frequent loss spikes—even with collocation resampling—TINNs maintain a small gap, with validation loss tracking training loss closely throughout optimization (Figure 5). Correspondingly, TINNs show smoother loss decay and more stable parameter updates.

We attribute this behavior to a structural mismatch in standard space–time PINNs: a single network with shared parameters must fit distinct temporal regimes, which can cause representation interference and destabilize optimization. TINNs mitigate this by allowing the network parameters to evolve with time, better matching nonstationary dynamics and improving robustness. Although illustrated on Burgers' equation, the same issue is expected to arise broadly in time-dependent PDEs with strongly nonuniform temporal dynamics.

6 Conclusion

We identify *time entanglement* in standard space–time PINNs: a single shared-parameter network must fit heterogeneous temporal regimes, often harming optimization and generalization. We propose *TINNs*, which represent evolution as a smooth trajectory in parameter space to decouple spatial structure from temporal variation. Across time-dependent PDE benchmarks, TINNs achieve higher accuracy and stability than strong PINN baselines while converging faster.

ACKNOWLEDGEMENTS

T.-S. Lin acknowledge the supports by National Science and Technology Council, Taiwan, under research grants 111-2628-M-A49-008-MY4 and 114-2124-M-390-001. M.-C. Lai acknowledge the support by National Science and Technology Council, Taiwan, under research grant 113-2115-M-A49-014-MY3.

REFERENCES

Alex Bihlo. Improving physics-informed neural networks with meta-learned optimization. *The Journal of Machine Learning Research*, 25(14):755–780, 2024.

Rafael Bischof and Michael A. Kraus. Multi-objective loss balancing for physics-informed deep learning. *Computer Methods in Applied Mechanics and Engineering*, 439, 2025.

Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*, volume 15 of *Texts in Applied Mathematics*. Springer Science & Business Media, New York, NY, USA, 3 edition, 2008.

Junwoo Cho, Seungtae Nam, Hyunmo Yang, Seok-Bae Yun, Youngjoon Hong, and Eunbyung Park. Separable physics-informed neural networks. In *Advances in Neural Information Processing Systems*, volume 36, pp. 23761–23788. Curran Associates, Inc., 2023.

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics-informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(88), 2022.

Chinmay Datar, Taniya Kapoor, Abhishek Chandra, Qing Sun, Erik Lien Bolager, Iryna Burak, Anna Veselovska, Massimo Fornasier, and Felix Dietrich. Fast training of accurate physics-informed neural networks without gradient descent. *arXiv preprint arXiv:2405.20836*, 2024.

T. A. Driscoll, N. Hale, and L. N. Trefethen (eds.). *Chebfun Guide*. Pafnuty Publications, Oxford, 2014.

Siyuan Duan, Wenyuan Wu, Peng Hu, Zhenwen Ren, Dezhong Peng, and Yuan Sun. CoPINN: Cognitive physics-informed neural networks. In *Forty-second International Conference on Machine Learning*, 2025.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Wei-Fan Hu, Yi-Jun Shih, Te-Sheng Lin, and Ming-Chih Lai. A shallow physics-informed neural network for solving partial differential equations on static and evolving surfaces. *Computer Methods in Applied Mechanics and Engineering*, 418, 2024.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2007.

Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

Kuang Luo, Jingshang Zhao, Yingping Wang, Jiayao Li, Junjie Wen, Jiong Liang, Henry Soekmadji, and Shaolin Liao. Physics-informed neural networks for pde problems: a comprehensive review. *Artificial Intelligence Review*, 58(323), 2025.

Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal on Applied Mathematics*, 11(2):431–441, 1963.

Levi D. McClenny and Ulisses M. Braga-Neto. Self-adaptive physics-informed neural networks. *Journal of Computational Physics*, 474, 2023.

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.

Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. In *NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing Systems*, number 632, pp. 7537–7547, 2020.

Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

Sifan Wang, Bowen Li, Yuhan Chen, and Paris Perdikaris. Piratenets: physics-informed deep learning with residual adaptive networks. *The Journal of Machine Learning Research*, 25(402):19707–19757, 2024a.

Sifan Wang, Shyam Sankaran, and Paris Perdikaris. Respecting causality for training physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 421, 2024b.

Sifan Wang, Ananyae Kumar bhartari, Bowen Li, and Paris Perdikaris. Gradient alignment in physics-informed neural networks: A second-order optimization perspective. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yicheng Wang, Xiaotian Han, Chia-Yuan Chang, Daochen Zha, Ulisses Braga-Neto, and Xia Hu. Auto-PINN: Understanding and optimizing physics-informed neural architecture. In *NeurIPS 2023 AI for Science Workshop*, 2023.

Contents

A Time-Dependent PDEs **13**
A.1 Viscous Burgers Equation 13
A.2 Allen-Cahn Equation 13
A.3 Klein-Gordon Equation 14
A.4 Korteweg-De Vries Equation 14
A.5 Wave Equation 14

B TINNs: A General Remedy for Time Entanglement **14**
B.1 Time-Entanglement Problem for Other Architectures 14
B.2 TINNs Generalize to Other Backbones 15

C LM optimizer **16**

D Implementation Details and Experimental Settings **16**
D.1 Comparison between MLP and TINN (Figure 3) 16
D.2 Evaluation Metric 17
D.3 Implementation Details for Main Results (Table 2) 17
D.4 Setting for Ablation Studies (Tables 3 and 4) 20

A TIME-DEPENDENT PDES

This section summarizes the partial differential equations used in our numerical experiments. Figure 6 shows spatiotemporal heatmaps of the reference solutions for the 1D PDE benchmarks (Burgers, Allen–Cahn, KdV, and Wave).

A.1 VISCOUS BURGERS EQUATION

We consider the one-dimensional viscous Burgers equation on the space–time domain $[-1, 1] \times [0, 1]$. The solution $u(x, t)$ satisfies

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

where the viscosity parameter is set to $\nu = 0.01/\pi$. We use the same reference solution as in Raissi et al. (2019) for quantitative evaluation.

A.2 ALLEN-CAHN EQUATION

We consider the one-dimensional Allen–Cahn equation on the space–time domain $[-1, 1] \times [0, 1]$. The solution $u(x, t)$ satisfies

$$\begin{cases} u_t - 0.0001 u_{xx} + 5u^3 - 5u = 0, & (x,t) \in (-1,1) \times (0,1), \\ u(x,0) = x^2 \cos(3\pi x) + x^2, & x \in [-1,1], \end{cases}$$

with periodic boundary conditions in space. The reference solution is generated using the Chebfun package (Driscoll et al., 2014) with 512 Fourier modes and a time-step size of 10^{-6}.

In several prior works (Raissi et al., 2019; Wang et al., 2024a; 2025), the Allen–Cahn equation is initialized with $u(x, 0) = x^2 \cos(\pi x)$, which is incompatible with periodic boundary conditions. This mismatch introduces an L^∞ error of order 10^{-3} near the initial time, with the largest error occurring near $t = 0$. Consequently, all methods incur a relative L^2-error of order 10^{-5}–10^{-6} regardless of their approximation accuracy, resulting in an artificial error floor. In this regime, accuracy comparisons among different methods become uninformative. To avoid this issue, we adopt the periodic-compatible initial condition above.

A.3 KLEIN-GORDON EQUATION

We consider the two-dimensional Klein–Gordon equation on the space–time domain $[-1,1] \times [-1,1] \times [0,10]$. The solution $u(x,y,t)$ satisfies

$$\begin{cases} u_{tt} - \Delta u + u^2 = f, & (x,y,t) \in (-1,1) \times (-1,1) \times (0,10), \\ u(x,y,0) = x + y, & (x,y) \in [-1,1] \times [-1,1], \\ u_t(x,y,0) = 2xy, & (x,y) \in [-1,1] \times [-1,1], \\ u(x,y,t) = u_{\mathrm{bc}}(x,y,t), & (x,y,t) \in \partial([-1,1] \times [-1,1]) \times [0,10], \end{cases}$$

where f denotes a source term and u_{bc} specifies the boundary condition. The exact solution is given by

$$u(x,y,t) = (x+y)\cos(2t) + xy\sin(2t),$$

from which both the source term f and the boundary data u_{bc} are derived.

A.4 KORTEWEG-DE VRIES EQUATION

We consider the one-dimensional Korteweg–de Vries (KdV) equation on the space–time domain $[-1,1] \times [0,1]$. The solution $u(x,t)$ satisfies

$$\begin{cases} u_t + u\,u_x + 0.022^2 u_{xxx} = 0, & (x,t) \in (-1,1) \times (0,1), \\ u(x,0) = \cos(\pi x), & x \in [-1,1], \end{cases}$$

with periodic boundary conditions in space. We use the same reference solution as in Wang et al. (2024a) for quantitative evaluation.

A.5 WAVE EQUATION

We consider the one-dimensional wave equation on the space–time domain $[0,1] \times [0,1]$. The solution $u(x,t)$ satisfies

$$\begin{cases} u_{tt} - 4u_{xx} = 0, & (x,t) \in (0,1) \times (0,1), \\ u(x,0) = \sin(\pi x) + \frac{1}{2}\sin(4\pi x), & x \in [0,1], \\ u_t(x,0) = 0, & x \in [0,1], \\ u(0,t) = u(1,t) = 0, & t \in [0,1]. \end{cases}$$

The exact solution is given by

$$u(x,t) = \sin(\pi x)\cos(2\pi t) + \frac{1}{2}\sin(4\pi x)\cos(8\pi t).$$

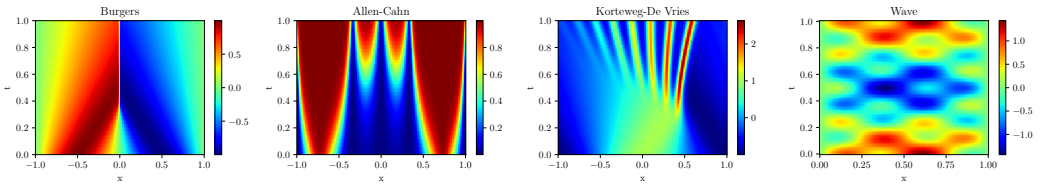

Figure 6: **PDEs solution.** The reference solutions for 1D PDEs are displayed.

B TINNS: A GENERAL REMEDY FOR TIME ENTANGLEMENT

B.1 TIME-ENTANGLEMENT PROBLEM FOR OTHER ARCHITECTURES

We introduced the *time-entanglement problem* using multilayer perceptrons (MLPs) as an illustrative example. However, this issue is not specific to MLPs and can arise in a broad class of neural architectures when time enters only through input conditioning while the internal representation remains shared across all temporal regimes. In this section, we briefly discuss several representative alternatives.

Consider a one-block ResNet (He et al., 2016) in one spatial dimension,

$$u_{\boldsymbol{\theta}}(x,t) := U(wx + vt + b) + \alpha^x x + \alpha^t t.$$

The spatial derivative is

$$\partial_x u_{\boldsymbol{\theta}}(x,t) = U'(wx + vt + b)\, w + \alpha^x,$$

which introduces an additional affine degree of freedom through α^x compared to a vanilla MLP. While this increases expressivity, temporal variation still acts only through a shift in the argument of U', and the underlying spatial representation remains fixed across time. As a result, the time-entanglement problem persists.

Next, consider a modified MLP (Wang et al., 2021) of the form

$$u_{\boldsymbol{\theta}}(x,t) := U(wx + vt + b)\, \sigma(w_1 x + v_1 t + b_1) + \big(1 - U(wx + vt + b)\big)\, \sigma(w_2 x + v_2 t + b_2).$$

The resulting spatial derivative is more structured and can partially alleviate interference between temporal regimes. Nevertheless, temporal dependence still enters exclusively through shifts in activation arguments, while the functional form of the spatial representation remains unchanged. Consequently, temporal evolution cannot be represented as flexibly as in TINNs.

Finally, consider separable physics-informed neural networks (SPINNs) (Cho et al., 2023), in which

$$u_{\boldsymbol{\theta}}(x,t) := U(x)\, V(t).$$

Here,

$$\partial_x u_{\boldsymbol{\theta}}(x,t) = U'(x)\, V(t),$$

allowing time-dependent scaling of spatial features. While this avoids pure activation shifts, the spatial representation $U'(x)$ itself remains time-independent and cannot adapt its structure as the solution evolves.

In contrast, TINNs explicitly parameterize temporal evolution as a trajectory in parameter space, enabling the spatial representation itself to change over time. This removes the shared-representation constraint underlying time entanglement and yields a more general and flexible modeling framework for time-dependent PDEs.

B.2 TINNs Generalize to Other Backbones

The TINN framework is not restricted to multilayer perceptrons as spatial backbones. More generally, TINNs incorporate temporal dependence by parameterizing the network weights as smooth functions of time, rather than treating time as an additional input coordinate. This construction can be applied to a wide range of neural architectures.

For example, given a ResNet (He et al., 2016), a time-induced formulation takes the form

$$u_{\boldsymbol{\theta}(t)}(x) := U(w(t)x + b(t)) + \alpha^x(t)x,$$

where all parameters evolve with time. Compared to a space–time ResNet with fixed weights, this allows the spatial representation itself to adapt dynamically.

Similarly, for a modified MLP (Wang et al., 2021), we may define

$$u_{\boldsymbol{\theta}(t)}(x) := U(w(t)x + b(t))\sigma(w_1(t)x + b_1(t)) + (1 - U(w(t)x + b(t)))\sigma(w_2(t)x + b_2(t)),$$

again allowing all parameters to vary with time. This yields a time-adaptive mixture structure while preserving the original architectural design.

For SPINNs (Cho et al., 2023), temporal dependence is already factorized from spatial variables. In the one-dimensional spatial case, applying TINNs recovers the same effective representation as using an MLP backbone with time-dependent parameters. In higher spatial dimensions, e.g., $d = 2$, a natural extension is

$$u_{\boldsymbol{\theta}(t)}(x,y) := U_{\boldsymbol{\theta}_1(t)}(x) U_{\boldsymbol{\theta}_2(t)}(y),$$

where each spatial component evolves independently over time.

These examples illustrate that the core idea of TINNs—representing temporal evolution as a trajectory in parameter space—is architecture-agnostic. As a result, TINNs can be readily integrated with a broad class of neural backbones beyond standard MLPs.

C LM OPTIMIZER

For clarity, we first describe the Levenberg–Marquardt (LM) method for a single least-squares loss term arising from a PDE residual. Consider

$$\mathcal{L}u_{\boldsymbol{\psi}} = f,$$

where $\mathcal{L}$ is a (possibly nonlinear) differential operator, f is a given source term, and $u_{\boldsymbol{\psi}}$ denotes a neural network parameterized by $\boldsymbol{\psi}$.

Let $\{\mathbf{x}_i\}_{i=1}^N$ be a set of collocation points. Define the residual vector $\mathbf{L} \in \mathbb{R}^N$ and the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{N\times P}$ by

$$\mathbf{J} = \begin{bmatrix} \frac{1}{\sqrt{N}}\nabla_{\boldsymbol{\psi}}\mathcal{L}u_{\boldsymbol{\psi}}(\mathbf{x}_1) \\ \vdots \\ \frac{1}{\sqrt{N}}\nabla_{\boldsymbol{\psi}}\mathcal{L}u_{\boldsymbol{\psi}}(\mathbf{x}_N) \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} \frac{1}{\sqrt{N}}(\mathcal{L}u_{\boldsymbol{\psi}}(\mathbf{x}_1) - f(\mathbf{x}_1)) \\ \vdots \\ \frac{1}{\sqrt{N}}(\mathcal{L}u_{\boldsymbol{\psi}}(\mathbf{x}_N) - f(\mathbf{x}_N)) \end{bmatrix},$$

where P is the number of trainable parameters.

With this notation, the loss can be written compactly as

$$\ell(\boldsymbol{\psi}) = \frac{1}{2}\|\mathbf{L}\|^2.$$

When multiple loss components are present—such as PDE residuals, initial conditions, and boundary conditions—the corresponding residual vectors and Jacobians are stacked vertically to form a single augmented least-squares system.

At each iteration, the LM method computes an update $\delta\boldsymbol{\psi}$ by solving the damped normal equations

$$(\mathbf{J}^\top\mathbf{J} + \mu I)\delta\boldsymbol{\psi} = \mathbf{J}^\top\mathbf{L},$$

where $\mu > 0$ is the damping parameter. The parameter vector is then updated as

$$\boldsymbol{\psi} \leftarrow \boldsymbol{\psi} - \delta\boldsymbol{\psi},$$

with μ adjusted adaptively according to the standard LM acceptance criterion.

We summarize the complete LM procedure used in our experiments in Algorithm 1.

D IMPLEMENTATION DETAILS AND EXPERIMENTAL SETTINGS

D.1 COMPARISON BETWEEN MLP AND TINN (FIGURE 3)

When introducing the Time-Induced Neural Network (TINN) in Section 3, we compared the ability of a vanilla MLP and a TINN to represent spatial derivatives of the solution to the Burgers equation. As shown in Figure 3, the two models are constructed with comparable numbers of parameters to ensure a fair comparison (MLP: 1160; TINN: 1145).

The MLP consists of two hidden layers with 20 and 50 neurons, respectively. In contrast, TINN employs a spatial network with two hidden layers of 20 neurons each, while temporal dependence is introduced through time-varying parameters.

To make the parameter count explicit, we detail the construction used for the Burgers equation in Table 1. The spatial backbone network has architecture $1\times20\times20\times1$ and does not include an output bias. This corresponds to $L = 3$ layers and a time-dependent feature vector $\boldsymbol{\Phi}(t) \in \mathbb{R}^{2L-1} = \mathbb{R}^5$, with a total of $N_D = 480$ backbone parameters.

The temporal network $\boldsymbol{\mathcal{N}}(t)$ is instantiated as $1\times10\times10\times5$ (again without an output bias), contributing 180 parameters. In addition, we introduce a gating vector $\boldsymbol{\alpha} \in \mathbb{R}^5$.

The total number of trainable parameters in TINN is therefore

$$2N_D + 180 + 5 = 1145,$$

Algorithm 1 LM Algorithm

Require: Training epochs E; loss construction $\mathcal{L}$ induced by PDE/BC/IC residuals; initial damping μ_0; update factors $\gamma_\uparrow > 1$, $\gamma_\downarrow > 1$; bounds $\mu_{\min}, \mu_{\max}$; safeguard factor $\eta \geq 1$ (for updating $\mu_{\min}$)
Ensure: Trained network $u_{\boldsymbol{\psi}}(\mathbf{x})$
  Initialize $\boldsymbol{\psi} \leftarrow \boldsymbol{\psi}_0$, $\mu \leftarrow \mu_0$, $\ell_{\text{old}} \leftarrow 10^5$
  for $k = 1$ to E **do**
    Assemble residual vector $(\mathbf{L})$, Jacobian $(\mathbf{J})$ and calculate loss (ℓ)
    Proposed update: $\boldsymbol{\psi} \leftarrow \boldsymbol{\psi} - (\mathbf{J}^\top \mathbf{J} + \mu I)^{-1} \mathbf{J}^\top \mathbf{L}$
    if $\ell < \ell_{\text{old}}$ **then**
      $\mu \leftarrow \max(\mu/\gamma_\downarrow, \mu_{\min})$ ▷ accept; decrease damping
    else
      $\mu \leftarrow \min(\mu \cdot \gamma_\uparrow, \mu_{\max})$ ▷ reject; increase damping
    end if
    $\ell_{\text{old}} \leftarrow \ell$
    if $\ell/\mu > 10^5$ **then**
      $\mu \leftarrow \ell/10$ ▷ safeguard for ill-conditioning / poor scaling
      $\mu_{\min} \leftarrow \eta\, \mu_{\min}$
    end if
  end for

where $2N_D$ arises from the affine lift $\mathbf{F}$, 180 from $\boldsymbol{\mathcal{N}}(t)$, and 5 from $\boldsymbol{\alpha}$.

For comparison, consider a naive parameterization of $\boldsymbol{\theta}(t)$ using the same spatial backbone with $N_D = 480$. If $\boldsymbol{\theta}(t)$ is generated by a fully connected network of architecture $1{\times}10{\times}10{\times}480$, sharing the same hidden width ($h = 10$) as $\boldsymbol{\mathcal{N}}(t)$, the total number of trainable parameters increases to 4930. This comparison illustrates how TINN introduces temporal flexibility with substantially fewer parameters than a naive time-dependent parameterization.

D.2 Evaluation Metric

To quantitatively compare accuracy across methods, we report the relative L^2-error, defined as

$$\sqrt{\frac{\sum_{i=1}^{N_{\text{test}}} (u_{\boldsymbol{\theta}(t_i)}(\mathbf{x}_i) - u^*(\mathbf{x}_i, t_i))^2}{\sum_{i=1}^{N_{\text{test}}} u^*(\mathbf{x}_i, t_i)^2}},$$

where $u_{\boldsymbol{\theta}(t)}(\mathbf{x})$ denotes the neural network prediction and $u^*(\mathbf{x}, t)$ denotes the reference solution. The test set $\{(\mathbf{x}_i, t_i)\}_{i=1}^{N_{\text{test}}} \subset \Omega \times [0, T]$ is sampled independently of the training data.

To quantify relative accuracy gains, we report the accuracy improvement (IMP):

$$\text{acc. IMP} = \frac{e_{\text{baseline}}}{e_{\text{TINN}}},$$

where e_{TINN} denotes the relative L^2-error achieved by the proposed method, and e_{baseline} denotes the relative L^2-error among all competing methods excluding TINNs.

D.3 Implementation Details for Main Results (Table 2)

We adopt the experimental settings proposed in prior works whenever they address the same benchmark PDEs, and apply minor refinements when necessary to ensure fair and stable comparisons. For example, for CoPINN, which reports results on the Klein–Gordon equation, we follow their original setup and apply additional implementation refinements, achieving a relative error of 10^{-6} compared to the 10^{-4} reported in their paper. For PirateNet, we use the hyperparameter configurations provided by the authors for the Burgers, Allen–Cahn, Korteweg–De Vries, and wave equations. Due to hardware constraints, the number of neurons is reduced in our implementation; nevertheless, the resulting models still contain substantially more parameters than the other baselines. All experiments are conducted on an NVIDIA A6000 GPU.

Table 5 summarizes the network architectures, training iterations, optimizers, and optimizer-specific hyperparameters used in all experiments. When using Adam, we apply a learning-rate decay schedule specified by the tuple (learning rate, decay rate, warmup, decay step); the same parameterization is adopted for SOAP. For the Levenberg–Marquardt (LM) optimizer, the damping parameter μ is adjusted dynamically during training according to the configuration (μ_0, $\gamma_\uparrow$, $\gamma_\downarrow$, $\mu_{\max}$, $\mu_{\min}$, η). The update mechanism is described in detail in Appendix C.

All methods except TINN involve a large number of parameters; consequently, these models are trained in single precision, consistent with the original implementations. In contrast, TINN employs a more compact parameterization, and LM training requires solving linear systems; therefore, we train TINN in double precision for improved numerical stability.

Table 5: Training configurations for each method and PDE.

Case	Main Structure	iteration	Optimizer	Hyperparameters
Burgers				
PINN	Layer: 4, Neuron: 320	250K	Adam	(10^{-3}, 0.9, 10000, 5000)
CoPINN*	Layer: 5, Neuron: 200	1050K	Adam	(10^{-3}, 0.9, 10000, 21000)
PirateNet SOAP	Block: 3, Neuron: 200	100K	SOAP	(10^{-3}, 0.9, 5000, 2000)
TINN	Layer: 2, x-neu: 20, t-neu: 10	30K	LM	(10, 1.7, 1.3, 10^{8}, 10^{-12}, 1.0)
Allen-Cahn				
PINN	Layer: 4, Neuron: 320	250K	Adam	(10^{-3}, 0.9, 10000, 5000)
CoPINN*	Layer: 5, Neuron: 200	1050K	Adam	(10^{-3}, 0.9, 10000, 21000)
PirateNet SOAP	Block: 3, Neuron: 200	100K	SOAP	(10^{-3}, 0.9, 5000, 2000)
TINN	Layer: 2, x-neu: 20, t-neu: 10	30K	LM	(10, 1.7, 1.3, 10^{8}, 10^{-12}, 1.0)
Korteweg-De Vries				
PINN	Layer: 4, Neuron: 320	250K	Adam	(10^{-3}, 0.9, 10000, 5000)
CoPINN*	Layer: 5, Neuron: 200	1050K	Adam	(10^{-3}, 0.9, 10000, 21000)
PirateNet SOAP	Block: 3, Neuron: 200	100K	SOAP	(10^{-3}, 0.9, 5000, 2000)
TINN	Layer: 2, x-neu: 20, t-neu: 10	25K	LM	(10, 1.7, 1.3, 10^{8}, 10^{-12}, 1.0)
Klein-Gordon				
PINN	Layer: 4, Neuron: 320	125K	Adam	(10^{-3}, 0.9, 10000, 2500)
CoPINN*	Layer: 5, Neuron: 128	200K	Adam	(10^{-3}, 0.9, 10000, 4000)
PirateNet SOAP	Block: 3, Neuron: 150	100K	SOAP	(10^{-3}, 0.9, 5000, 2000)
TINN	Layer: 2, x-neu: 20, t-neu: 10	10K	LM	(10, 1.7, 1.3, 10^{8}, 10^{-12}, 1.0)
Wave				
PINN	Layer: 4, Neuron: 320	250K	Adam	(10^{-3}, 0.9, 10000, 5000)
CoPINN*	Layer: 5, Neuron: 200	1100K	Adam	(10^{-3}, 0.9, 10000, 22000)
PirateNet SOAP	Block: 3, Neuron: 200	100K	SOAP	(10^{-3}, 0.9, 5000, 2000)
TINN	Layer: 2, x-neu: 20, t-neu: 10	30K	LM	(10, 1.27, 1.3, 10^{8}, 5×10^{-7}, 2.0)

All experiments are repeated using five random seeds $\{0, 1, 2, 3, 4\}$. For PINNs- and TINN-based methods, we monitor a validation loss to detect overfitting. Specifically, if the validation loss exceeds five times the training loss, the collocation points in the training dataset are resampled.

In Table 6, we report the number of training points used for each method and PDE. The number of collocation points for the PDE residual is denoted by N_c, while N_{ic} and N_{bc} correspond to the initial and boundary conditions, respectively. Entries marked as N/A indicate that the corresponding training points are not required. For example, the Allen–Cahn and Korteweg–De Vries equations are equipped with periodic boundary conditions; therefore, we employ embedding techniques to enforce these constraints exactly, eliminating the need for boundary collocation points when minimizing the boundary loss. For CoPINN* and PirateNet, we use the same training-point configurations as reported in CoPINN (Duan et al., 2025) and PirateNet (Wang et al., 2025), respectively.

In all experiments that use the Levenberg–Marquardt optimizer, we apply penalty weights to balance the different loss components. Let λ_r, λ_{ic}, and λ_b denote the penalty terms for the PDE residual, initial condition, and boundary condition, respectively. Specifically, we set λ_{ic}^{-1} of the initial-condition function values (average of 60% small values) to capture the small dynamics effectively. The resulting penalty settings for each PDE are listed in Table 7.

For PINN, we use uniform penalty weights, i.e., $\lambda_* = 1$. In CoPINN, although a pointwise "difficulty" measure is computed for each training point, the corresponding penalty weights are also fixed

Table 6: Number of training points for each method and PDE.

Case	N_c	$N_{\mathbf{ic}}$	N_{bc}
Burgers			
PINN	10000	500	400
CoPINN*	65536	256	512
PirateNet SOAP	8192	256	200
TINN	10000	500	400
Allen-Cahn			
PINN	10000	500	N/A
CoPINN*	65536	256	N/A
PirateNet SOAP	8192	512	N/A
TINN	10000	500	N/A
Korteweg-De Vries			
PINN	10000	500	N/A
CoPINN*	65536	256	N/A
PirateNet SOAP	8192	512	N/A
TINN	10000	500	N/A
Klein-Gordon			
PINN	15000	4000	8000
CoPINN*	16777216	65536	262144
PirateNet SOAP	8192	10201	81204
TINN	15000	4000	8000
Wave			
PINN	10000	500	400
CoPINN*	65536	256	512
PirateNet SOAP	8192	128	400
TINN	10000	500	400

to one. In contrast, PirateNet balances the loss components by normalizing their norms, which leads to penalty weights that vary dynamically during training.

Table 7: Penalty weights for each loss component in TINN training

Case	$\lambda_{\mathbf{r}}$	$\lambda_{\mathbf{ic}}$	λ_b
Burgers	1	2	1
Allen-Cahn	1	20	N/A
Korteweg-De Vries	1	2	N/A
Klein-Gordon	1	3	1
Wave	1	2	10

In Section 4, we claim that TINN reaches comparable or better accuracy with a $10.55\times$ speedup over PirateNet with SOAP on the Burgers equation. Specifically, TINN uses 0.16hr to achieve the accuracy $1.46E-06$, which is the error performance of the PirateNet with SOAP. On the other hand, TINN uses 0.65hr to achieve the accuracy $4.72E-06$, which is the error performance of the PirateNet with SOAP (see in Table 8).

Table 8: Comparison for the convergence speed between PirateNet and TINN. The error threshold is the error performance obtained from PirateNet with SOAP. All of the experiments are with random seed 4.

Case	Error Threshold	PirateNet	TINN	Speed Improvement
Burgers	1.46E-06	1.70hr	0.16hr	$10.55\times$
Allen-Cahn	4.72E-06	1.50hr	0.65hr	$2.30\times$

D.4 Setting for Ablation Studies (Tables 3 and 4)

Results are averaged over five runs (seeds 0–4).

Adam Ablation: Impact of Model Capacity. Table 9 reports the experimental configurations and the standard deviation of the relative L^2-error across trials for TINNs with large numbers of parameters trained using the Adam optimizer. In the network specification, $\{n\}^4$ denotes four hidden layers with n neurons each, while the leading number indicates the dimension of the random Fourier feature embedding (e.g., 256 or 50).

Following PirateNet, we employ parameter initialization and causal training for TINN. During training, the penalty terms are dynamically adjusted by normalizing the gradient magnitudes, as proposed in Wang et al. (2024a). Since the total number of training iterations differs between the two methods, the penalty normalization is updated every 1000 iterations for PirateNet and every 1500 iterations for TINN. All experiments in this subsection are conducted using single-precision arithmetic.

Table 9: Experimental settings and performance of large-capacity TINNs and PirateNet trained using the Adam optimizer.

Case	Rel L^2-Error	Iteration	Hyperparameters	t-structure	Main structure
Burgers					
PirateNet	2.43E-05 $\pm$ 4.01E-06	100K	(10^{-3}, 0.9, 5000, 2000)	–	block: 3, neuron: 200
TINN	**2.15E-05 $\pm$ 3.75E-06**	160K	(10^{-3}, 0.9, 10000, 2000)	$1 \times \{150\}^4 \times 9$	$1 \times 256 \times \{150\}^4 \times 1$
Allen-Cahn					
PirateNet	2.32E-03 $\pm$ 9.60E-04	100K	(10^{-3}, 0.9, 5000, 2000)	–	block: 3, neuron: 200
TINN	**8.29E-05 $\pm$ 2.80E-05**	160K	(10^{-3}, 0.9, 10000, 2000)	$1 \times \{150\}^4 \times 9$	$1 \times 2 \times 256 \times \{150\}^4 \times 1$
Klein-Gordon					
PirateNet	5.60E-05 $\pm$ 5.11E-06	100K	(10^{-3}, 0.9, 5000, 2000)	–	block: 3, neuron: 150
TINN	**4.98E-05 $\pm$ 8.31E-06**	200K	(10^{-3}, 0.9, 10000, 2000)	$1 \times 50 \times \{100\}^4 \times 9$	$2 \times 50 \times \{100\}^4 \times 1$

LM Ablation: Impact of Network Structure. Table 10 compares the performance of three network architectures—two MLP-based models and one TINN—trained using the LM optimizer. subMLP is the special case of TINN sharing the same backbone. For instance, if the spatial backbone of TINN is $1 \times 20 \times 20 \times 1$, where the parameters are time-dependent. Then the structure of subMLP is $2 \times 20 \times 20 \times 1$, where the parameters are time-independent, and the additional input dimension is time t. Since the parameters in subMLP are time-independent, the number of parameters is roughly half of the TINN. This causes a distinct number of parameters under a similar model size. MLP shares a similar backbone (two hidden layers with 20 neurons in the first hidden layer) with TINN, and uses a comparable number of parameters. We report the mean and standard deviation of the relative L^2-error across multiple trials. All experiments are conducted in double precision.

Table 10: Accuracy and variability of two MLP architectures and one TINN with similar parameter counts under LM training.

Case	Rel L^2-Error	Iteration	Hyperparameters	t-structure	Main structure
Burgers					
subMLP	7.11E-05 $\pm$ 4.10E-05	110K	(10, 1.7, 1.3, 10^8, 10^{-9}, 1.0)	–	$2 \times 20 \times 20 \times 1$
MLP	1.92E-05 $\pm$ 1.03E-05	34K	(10,1.7,1.3,10^8,10^{-12}, 1.0)	–	$2 \times 20 \times 50 \times 1$
TINN	**6.89E-07 $\pm$ 3.97E-07**	30K	(10,1.7,1.3,10^8, 10^{-12}, 1.0)	$1 \times 10 \times 10 \times 5$	$1 \times 20 \times 20 \times 1$
Allen-Cahn					
subMLP	1.76E-03 $\pm$ 1.10E-03	110K	(10, 1.7, 1.3, 10^8, 10^{-9}, 1.0)	–	$2 \times 3 \times 20 \times 20 \times 1$
MLP	7.14E-06 $\pm$ 8.57E-07	34K	(10,1.7,1.3,10^8, 10^{-12}, 1.0)	–	$2 \times 3 \times 20 \times 51 \times 1$
TINN	**3.85E-06 $\pm$ 1.48E-06**	30K	(10,1.7,1.3,10^8, 10^{-12}, 1.0)	$1 \times 10 \times 10 \times 5$	$1 \times 2 \times 20 \times 20 \times 1$
Klein-Gordon					
subMLP	2.84E-05 $\pm$ 1.24E-05	35K	(10, 1.7, 1.3, 10^8, 10^{-8}, 1.0)	–	$3 \times 20 \times 20 \times 1$
MLP	5.11E-06 $\pm$ 3.87E-06	11.1K	(10,1.7,1.3,10^8, 10^{-12}, 1.0)	–	$3 \times 20 \times 51 \times 1$
TINN	**4.78E-06 $\pm$ 2.63E-06**	10K	(10,1.7,1.3,10^8, 10^{-12}, 1.0)	$1 \times 10 \times 10 \times 5$	$2 \times 20 \times 20 \times 1$