TUTORIAL | AUGUST 19 2024

Tutorials: Physics-informed machine learning methods of computing 1D phase-field models

Wei Li ; Ruqing Fang ; Junning Jiao ; Georgios N. Vassilakis ; Juner Zhu

APL Mach. Learn. 2, 031101 (2024)
https://doi.org/10.1063/5.0205159

Tutorials: Physics-informed machine learning methods of computing 1D phase-field models

Cite as: APL Mach. Learn. 2, 031101 (2024); doi: 10.1063/5.0205159
Submitted: 24 February 2024 • Accepted: 24 July 2024 •
Published Online: 19 August 2024

Wei Li,[1,a] Ruqing Fang,[1,b] Junning Jiao,[1,c] Georgios N. Vassilakis,[2,d] and Juner Zhu[1,e]

AFFILIATIONS

[1] Department of Mechanical and Industrial Engineering, Northeastern University, Boston, Massachusetts 02115, USA
[2] Department of Physics, Northeastern University, Boston, Massachusetts 02115, USA

[a] **Authors to whom correspondence should be addressed:** we.li@northeastern.edu and j.zhu@northeastern.edu
[b] r.fang@northeastern.edu
[c] ju.jiao@northeastern.edu
[d] vassilakis.g@northeastern.edu
[e] https://www.zhujuner.com/

ABSTRACT

Phase-field models are widely used to describe phase transitions and interface evolution in various scientific disciplines. In this Tutorial, we present two neural network methods for solving them. The first method is based on physics-informed neural networks (PINNs), which enforce the governing equations and boundary/initial conditions in the loss function. The second method is based on deep operator neural networks (DeepONets), which treat the neural network as an operator that maps the current state of the field variable to the next state. Both methods are demonstrated with the Allen–Cahn equation in one dimension, and the results are compared with the ground truth. This Tutorial also discusses the advantages and limitations of each method, as well as the potential extensions and improvements.

I. INTRODUCTION

Phase-field models are a powerful and versatile class of mathematical equations that have found wide applications in different scientific disciplines such as materials science,[1] electrochemistry,[2] and fracture mechanics.[3] These theories employ continuous, order-parameter fields to describe phase transitions and evolving interfaces, offering a unified approach to investigate a diverse range of phenomena, from solidification and grain growth to fluid dynamics and chemical and biological pattern formation. Phase-field models are typically solved numerically using the spectral method (SM),[4] finite difference method (FDM),[5] or finite element method (FEM).[6] SM offers a relatively fast convergence rate, while FDM is straightforward to implement. However, both SM and FDM are primarily applicable to regular domains. For high-dimensional problems involving complex geometries and boundary conditions, the FEM is often employed. Nevertheless, the FEM requires substantial computational resources due to the exceedingly fine mesh resolutions necessary to capture the intrinsic small length-scales associated with phase boundaries. Since machine learning (ML) is known to have a rapid inference speed after training, it can potentially accelerate the computation of phase-field models. Substantial prior work has successfully demonstrated the concept of leveraging machine learning to accelerate simulations across various domains, including fluid dynamics,[7,8] crack propagation,[9] molecular dynamics,[10] photodynamics,[11] and phase-field dynamics.[12–15] These approaches often train purely data-driven machine learning models as surrogates, which need a large experimental or simulation dataset, with significant effort in data preparation.

Alternatively, surrogate models can be trained with limited or no data by incorporating physics into the machine learning framework, a paradigm often referred to as physics-informed machine learning.[16] In this Tutorial, we specifically focus on training surrogates without any data, namely purely physics-informed, which is essentially solving equations. Using ML to solve equations is a promising approach to addressing the trade-off between computational efficiency and accuracy. The original concept of solving equations with ML methods can be traced back to the pioneering work of Lagaris *et al.*,[17] where they proposed to use fully connected neural networks to approximate solutions of ordinary and partial

differential equations (ODEs and PDEs). An important advancement surfaced with the Physics-Informed Neural Networks (PINNs) by Raissi *et al.*,[18] wherein physical laws and neural networks are integrated into a unified ML framework. Around the same time, Neural Ordinary Differential Equations (Neural ODEs) were proposed to model dynamic systems with neural networks.[19] Both Neural ODEs and PINNs build upon the foundational principle introduced by Lagaris *et al.*[17] where governing equations and boundary/initial conditions are introduced into the loss function. The neural network then approximates the solution through optimization (training) rather than solving a set of discretized algebraic equations as conventional numerical methods (e.g., FEM) do. The recent advent of deep operator learning marks a further advancement in this field.[20–22] It can significantly improve the generalizability of neural networks by developing innovative network architectures inspired by computational science, such as Deep Operator Network (DeepONet)[20] and Fourier Neural Operator (FNO),[22] thereby broadening their applicability.

We focus on the machine learning-based solution of partial differential equations (PDEs), with a particular emphasis on phase-field models. The existing literature has demonstrated several successful implementations of this approach.[23–25] The primary objective of this Tutorial is to summarize and introduce the fundamental concepts and methodologies of this approach. This Tutorial is organized as follows: Sec. II introduces the general phase-field method and the specific phase-field equations that will be solved. Section III then presents the overarching PIML framework, which is the foundation for the two approaches presented in the following. Sections IV–VI introduce two machine learning approaches to solve phase-field equations within the PIML paradigm: the PINN-based framework and the DeepONet-based framework. Numerical examples involving the one-dimensional Allen–Cahn equation will be implemented in detail to demonstrate the effectiveness of the two approaches. The complete code is available in the GitHub repository (https://github.com/weili101/Phase-Field_DeepONet/tree/main/Tutorial). Finally, Sec. VIII discusses the computation efficiency and provides a short outlook on potential improvements.

II. PHASE-FIELD METHOD

We first outline the phase-field method. For a single field variable u, the total standard free energy functional $\mathcal{G}$ for an inhomogeneous system, proposed by van der Waals[26] and Cahn and Hilliard,[27] is defined as

$$\mathcal{G} = \int_\Omega G(u)\,\mathrm{d}x = \int_\Omega \left[g(u) + \frac{1}{2}\kappa_u (\nabla u)^2 \right] \mathrm{d}x, \tag{1}$$

where $G(u)$ denotes the total free-energy density, $g(u)$ is the homogeneous free-energy density, and the second term on the right-hand side represents the gradient energy at phase interfaces with the gradient coefficient κ_u. Generally, the field variables evolve in the direction where the free energy continuously decreases. For a conserved field variable, the dynamics can be expressed as a conservation law for gradient-driven fluxes. The Cahn–Hilliard equation can then be obtained as follows:

$$\frac{\partial u}{\partial t} = \nabla \cdot M \nabla \frac{\delta \mathcal{G}}{\delta u}, \tag{2}$$

where M is a transport coefficient (the product of the mobility and the concentration field variable[28]) and the functional derivative (diffusional chemical potential) is given by

$$\frac{\delta \mathcal{G}}{\delta u} = \frac{\mathrm{d}g(u)}{\mathrm{d}u} - \kappa_u \nabla^2 u. \tag{3}$$

For a non-conserved field variable, we have the Allen–Cahn equation as follows:

$$\frac{\partial u}{\partial t} = -M \frac{\delta \mathcal{G}}{\delta u}, \tag{4}$$

where the functional derivative is also given by Eq. (3). A detailed physical interpretation of the two equations can be found in Refs. 2 and 28.

The above phase-field theory is general and can be applied to various scientific disciplines. As an example, we assume a commonly used double-well homogeneous free energy $g(u) = \frac{1}{4\epsilon^2}(1-u^2)^2$, where ϵ is the length scale that characterizes the width of the phase boundary. Both κ_u and M are set to 1 for simplicity. Substituting $g(u)$ into Eq. (1), we get the total free energy of the system as follows:

$$\mathcal{G} = \int_\Omega \left[\frac{1}{4\epsilon^2}(1-u^2)^2 + \frac{1}{2}(\nabla u)^2 \right] \mathrm{d}x. \tag{5}$$

We can then derive the corresponding Allen–Cahn equation with Eqs. (3)–(5),

$$\frac{\partial u}{\partial t} = \nabla^2 u - \frac{1}{\epsilon^2}(u^3 - u), \tag{6}$$

and the Cahn–Hilliard equation with Eqs. (2), (3), and (5),

$$\frac{\partial u}{\partial t} = \nabla^2 \left(-\nabla^2 u + \frac{1}{\epsilon^2}(u^3 - u) \right). \tag{7}$$

In this Tutorial, we focus our investigation on the one-dimensional Allen–Cahn equation, since this formulation provides a tractable framework to elucidate the fundamental concepts. Conventionally, this PDE can be solved by many existing numerical methods such as the finite difference method (see the Appendix). In the following, we will solve the equation with machine learning methods. The solution with the finite difference method will be treated as the ground truth to evaluate the prediction accuracy of machine learning methods.

III. PHYSICS-INFORMED MACHINE LEARNING

We then introduce the machine learning methods to solve the above phase-field models. Although machine learning has demonstrated remarkable success in many areas such as computer vision[29] and natural language processing,[30] it is often perceived as a black box when applied to scientific problems that require

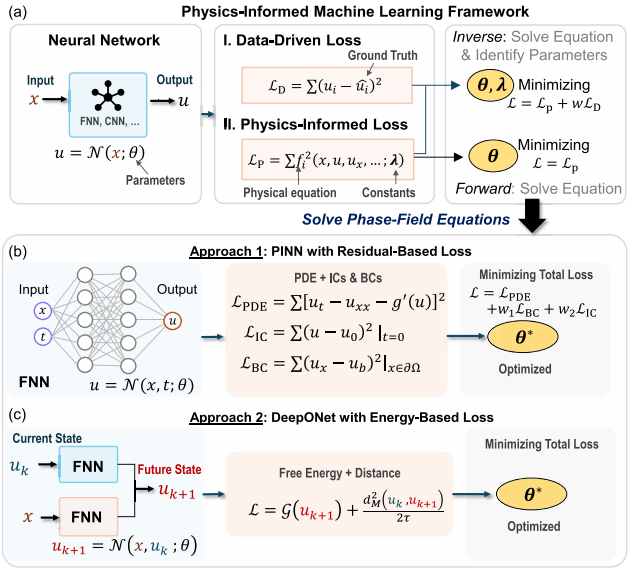

FIG. 1. (a) The overall framework of physics-informed machine learning. It consists of three parts: a machine learning algorithm such as neural networks, a loss function incorporating both data and physical laws, and a training strategy. Two approaches to solve phase-field equations within this framework: (b) PINN-based framework that consists of a FNN representing the solution and a physics-informed loss incorporating the governing equation, IC, and BCs, and (c) DeepONet-based framework that consists of a DeepONet as the time stepper predicting the evolution of the phase variable and a free-energy-based loss function following the minimizing movement scheme.

interpretable and physics-grounded solutions. The emergence of Scientific Machine Learning (SciML) represents a concerted effort to overcome this challenge.[31] The overarching objective of SciML is to improve the domain awareness, interpretability, and robustness of machine learning in scientific applications. This burgeoning field finds application in many areas[32] such as enhancing multiphysics and multiscale modeling.[33]

One successful example of SciML is physics-informed machine learning (PIML)[16] that combines both data and physics in a unified ML framework. Like most ML frameworks, PIML consists of three main components: a machine learning model, a loss function, and an optimization algorithm, as illustrated in Fig. 1(a). For readers unfamiliar with these concepts and frameworks, it is helpful to treat machine learning as an optimization process. A machine learning model such as a neural network is a function approximator with many unknown parameters. The loss function that conventionally characterizes the discrepancy between prediction and ground truth is the objective function. Training neural networks is essentially an optimization process to find the optimal parameters of the neural network that minimize the loss function. We will give a more detailed explanation of the three aspects in the following.

A. Neural network

The first component of a PIML framework is the machine learning model responsible for approximating the solution to the problem of interest. Generally, this can take the form of any type of machine learning algorithm, although in this Tutorial, we focus specifically on neural network architectures. Neural networks have emerged as a versatile class of machine learning models capable of representing complex functions and capturing intricate patterns in data. Some of the prominent neural network architectures include feedforward neural networks (FNNs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers. The choice of neural network architecture is often guided by the specific characteristics of the problem domain. For instance, FNNs are well-suited for modeling one-dimensional problems, whereas CNNs are a better choice for problems involving spatial or grid-structured data, such as images. Meanwhile, RNNs can well handle time-dependent or sequential data. These fundamental neural network building blocks can also be combined and extended to construct more advanced architectures capable of tackling increasingly complex problems. One such example is DeepONet, which we will introduce in Secs. V and VI. Mathematically, we can regard a neural network as a highly nonlinear approximation function,

$$u = \mathcal{N}(x, \theta), \tag{8}$$

where $u \in \mathbb{R}^M$ and $x \in \mathbb{R}^D$ represent the input and output vectors, $\mathcal{N}$ denotes the overall mapping from the inputs to the outputs, and θ denotes all the unknown parameters (weights and biases) of the neural network.

B. Loss function

The construction of the loss function is a crucial element that distinguishes physics-informed machine learning (PIML) from conventional purely data-driven machine learning approaches. In conventional ML, the loss function is typically defined only based on the observed data, with the goal of minimizing the discrepancy between the model predictions and the training targets. In contrast, the PIML loss function adopts a hybrid formulation that incorporates both data-driven and physics-informed components. This hybrid loss function enables the machine learning model to not only fit the observed data but also respect the underlying physical laws governing the problem of interest. The data-driven loss term ensures that the ML model accurately reflects the observed data, which can be defined as the mean square error between the model prediction u and the observed data $\tilde{u}$,

$$\mathcal{L}_D = \frac{1}{N_D} \sum_{i=1}^{N_D} \left[u(x^i) - \tilde{u}^i \right]^2, \tag{9}$$

where $(x^i, \tilde{u}^i)(i = 1, 2, \ldots N_D)$ are the N_D training data pairs, representing the sampled inputs and corresponding outputs. The physics-informed loss term integrates the relevant physical laws into the learning process. Suppose we have a physical law $F(x, u, u_x, u_t, u_{xx}, \ldots; \lambda) = 0$, where u_x and u_t denote the first derivatives of u with respect to x and t, u_{xx} indicates the second derivative, and λ represents the unknown physical parameters. The corresponding physics-informed loss function can be constructed as

$$\mathcal{L}_P = \frac{1}{N_P}\sum_{i=1}^{N_P} F(x^i, u^i, u_x^i, u_t^i, u_{xx}^i, \ldots; \lambda)^2. \tag{10}$$

Unlike the data-driven loss, the physics-informed loss term does not require any observed data; it only requires the input locations x_i as the training data. This is because the physics-informed loss enforces the physical constraints directly, without the need to fit the observed data.

C. Training

The last component of PIML is an optimization algorithm minimizing the total loss. The commonly used optimization algorithms include stochastic gradient descent (SGD), Adam, and L-BFGS. The total loss to be minimized during training in a PIML framework can be tailored to the studied problem. Depending on the available information and the nature of the problem, the total loss can include either or both the data-driven and physics-informed terms. For inverse problems, where the underlying physics is partially known but supplemented by some experimental data, it is beneficial to include both the data-driven and physics-informed loss terms. By minimizing this hybrid loss function, the PIML framework can simultaneously find the optimized physical parameters λ^* and network parameters θ^* to approximate the solution. This can be mathematically written as

$$\theta^*, \lambda^* = \arg\min_{\theta,\lambda} \mathcal{L}(\theta;\lambda), \quad \text{where } \mathcal{L} = \mathcal{L}_D + w\mathcal{L}_P, \tag{11}$$

in which w is the weight to balance the contribution of data-driven and physical-informed loss terms. For forward problems, where the physical laws are fully understood but experimental data are limited or unavailable, we can rely only on the physics-informed loss term. In this case, the neural network is trained to approximate the solution by enforcing the physical constraints, without the need for any observed data,

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta), \quad \text{where } \mathcal{L} = \mathcal{L}_P. \tag{12}$$

It is important to note that the PIML framework presented above is general and can be applied to various problem domains. Below, we will introduce two ML approaches to solve phase-field equations within this PIML paradigm: the PINN-based framework and the DeepONet-based framework. The PINN-based approach consists of a FNN and a loss function formulated based on the residuals of the governing partial differential equations (PDEs), boundary conditions (BCs), and initial conditions (ICs) [see Fig. 1(b)]. In contrast, the DeepONet-based framework comprises a DeepONet architecture and a loss function defined in terms of the total free energy of the system [see Fig. 1(c)]. Both approaches will be detailed in Secs. IV–VI with numerical examples.

IV. SOLVING ALLEN–CAHN EQUATION WITH PINN

This section demonstrates solving the Allen–Cahn equation in 1D with the PINN-based framework as illustrated in Fig. 1(b). The governing PDE, IC, and BCs to be solved are given as follows:

$$\begin{aligned} &u_t = u_{xx} - \frac{1}{\varepsilon^2}(u^3 - u), \quad x \in [-1,1], t \in [0, 0.05], \\ &u(x, t=0) = \sin(\pi x) x^2 (1-x)^2, \\ &u_x(x = \pm 1, t) = 0, \end{aligned} \tag{13}$$

where u represents a non-conserved phase field variable that depends on time and space and ε is set to 0.1. The IC is selected to be consistent with the BC.

To solve this problem with PINN, we have three major steps: (1) construct an appropriate neural network to represent the solution; (2) construct the loss function with the residuals of PDEs, ICs, and BCs; and (3) train the neural network to approximate the solution. We will explain each step in detail.

Step 1: Neural Network Construction. Since the solution $u(x,t)$ is both time- and space-dependent, we take the spatial coordinate x and the time t as the inputs of the neural network. The output is then the approximated solution u. We use a fully connected neural network, consisting of three hidden layers with 100 nodes each. For the activation function, we utilize the hyperbolic tangent (tanh) function, which has stable and easy-to-compute gradients and a bounded output range $[-1, 1]$, which aligns well with the physical bounds of the phase-field variable in this example. However, it is important to note that for more general cases, where the phase-field variable may not be inherently bounded within $[-1, 1]$, the output layer can be set to use a linear activation function. This would allow the neural network to output values outside the $[-1, 1]$ range, providing greater flexibility in representing the solution. The network can be represented mathematically as follows:

$$\begin{aligned} \mathbf{u} &= \mathcal{N}(\mathbf{x};\theta) \\ &= \mathbf{W}_4^T \tanh\left(\mathbf{W}_3^T \tanh\left(\mathbf{W}_2^T \tanh\left(\mathbf{W}_1^T \mathbf{x} \right.\right.\right. \\ &\quad \left.\left.\left. + \mathbf{b}_1\right) + \mathbf{b}_2\right) + \mathbf{b}_3\right) + \mathbf{b}_4, \end{aligned} \tag{14}$$

where $\mathbf{W}_i$ and $\mathbf{b}_i (i = 1, 2, 3, 4)$ are the weight matrices and bias vectors of the neural network, respectively. We use $\theta = \cup\{\mathbf{W}_i, \mathbf{b}_i\}$ $(i = 1, 2, 3, 4)$ to denote the collection of all the parameters of the neural network.

Listing 1. Define a fully connected feedforward neural network with three hidden layers.

```
# Step 1. Create a Neural Network Model
model = tf.keras.Sequential([
  tf.keras.layers.Dense(100,activation='tanh'),
  tf.keras.layers.Dense(100,activation='tanh'),
  tf.keras.layers.Dense(100,activation='tanh'),
  tf.keras.layers.Dense(1,activation=None)
])
```

To implement this in code, there are several popular options, such as TensorFlow,[34] PyTorch,[35] MXNet,[36] and JAX.[37] Here, we use TensorFlow as an example. The same workflow also applies to other packages. To define this neural network, we can use the code in Listing 1.

Step 2: Loss Function Construction In this example, we only consider the physics-informed loss term. It is constructed as follows:

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_{\mathrm{PDE}} + w_1\mathcal{L}_{\mathrm{BC}} + w_2\mathcal{L}_{\mathrm{IC}},\\ \mathcal{L}_{\mathrm{PDE}} &= \frac{1}{N_{\mathrm{P}}}\sum_{i=1}^{N_{\mathrm{P}}}\left(u_t^i - u_{xx}^i + \frac{1}{\varepsilon^2}(u^i-(u^i)^3)\right)^2,\\ \mathcal{L}_{\mathrm{BC}} &= \frac{1}{N_{\mathrm{B}}}\sum_{i=1}^{N_{\mathrm{B}}}(u_x^i)^2,\\ \mathcal{L}_{\mathrm{IC}} &= \frac{1}{N_{\mathrm{I}}}\sum_{i=1}^{N_{\mathrm{I}}}(u^i-\sin(\pi x^i)(x^i+1)^2(x^i-1)^2)^2,\end{aligned} \tag{15}$$

where $\mathcal{L}_{\mathrm{PDE}}$, $\mathcal{L}_{\mathrm{BC}}$, and $\mathcal{L}_{\mathrm{IC}}$ are the loss terms corresponding to the PDE, BC, and IC in Eq. (13). These loss terms are constructed as the mean squared error of the residuals, where the residual is defined as the difference between the left-hand side and right-hand side of each equation. N_{P}, N_{B}, and N_{I} represent the number of sampled collocation points used for training. The relative weights w_1 and w_2 can be used to balance the contributions of the PDE, BC, and IC terms in the overall loss function. Here, we have set $w_1 = w_2 = 1$, giving equal importance to all components. Notably, these weights would largely influence the training results and should be carefully tuned according to specific problems. We implement this loss function in TensorFlow using the code in Listing 2, where both the first and second derivatives are calculated to evaluate the loss. This can be achieved by the automatic differentiation technique implemented in most ML packages.[34,38]

Listing 2. Construct the total loss function, including three terms for PDE, IC, and BC.

```
# Step 2. Construct loss function
u0 = lambda x: np.sin(np.pi*x)*(x+1)**2*(x-1)**2  # IC
def loss(model, Xp, Xi, Xb):
    # Xp, Xi, Xb are training data for PDE, IC, BC
    # Calculate the loss of PDE
    with tf.GradientTape(persistent=True) as g:
        g.watch(Xp)
        u = model(Xp)
        u_x = g.gradient(u, Xp)[:, 0:1]
    u_t  = g.gradient(u, Xp)[:, 1:2]
    u_xx = g.gradient(u_x, Xp)[:, 0:1]
    loss_PDE = tf.reduce_mean(((u_t - u_xx)
        - (u - u**3))**2/eps**2 )
    # Calculate loss of IC
    loss_IC = tf.reduce_mean((model(Xi)
        - u0(Xi[:, 0:1]))**2)
    # Calculate the loss of BC
    with tf.GradientTape() as g2:
        g2.watch(Xb)
        ub = model(Xb)
    ub_x = g2.gradient(ub, Xb)[:, 0:1]
    loss_BC = tf.reduce_mean(ub_x**2)
return loss_PDE, loss_IC, loss_BC
```

Step 3: Data Preparation and Training Finally, we need to generate a dataset and perform training and testing. As introduced in Sec. III C, we only need the collocation points for the PDEs, ICs, and BCs, without the corresponding outputs (ground truth or observed data) as the training data. This can be done by randomly sampling the spatial and temporal coordinates from the domain $\{(x,t)|x\in[-1,1], t\in[0,0.05]\}$, the boundary $\{(x,t)|x=\pm1, t\in[0,0.05]\}$, and the initial state $\{(x,t)|x\in[-1,1], t=0\}$. In TensorFlow, we can implement it with the code in Listing 3. All the training data are sampled randomly following a uniform distribution. To speed up the training, the dataset is usually fed into the neural network in small batches (a subset of the total dataset). To simplify this process, we can convert the data into the TensorFlow `Dataset` format, which has the built-in function to feed data in batches.

Listing 3. Prepare training datasets for evaluating the loss terms of PDE, IC, and BC.

```
# Step 3. Prepare Datasets and Training
# 3.0. Generate training data
def BC_data(Nt):
    # Training data for BC
    x = np.random.choice([-1, 1], size=(Nt, 1))
    t = np.random.rand(Nt, 1)*0.05
    X = np.hstack((x, t))
    return tf.constant(X, dtype=tf.float32)

def IC_data(Nx):
    # Training data for IC
    x = np.random.rand(Nx, 1)*2 - 1
    t = np.zeros((Nx, 1))
    X = np.hstack((x, t))
    return tf.constant(X, dtype=tf.float32)

def PDE_data(Np):
    # Training data for PDE
    x = np.random.rand(Np, 1)*2 - 1
    t = np.random.rand(Np, 1)*0.05
    X = np.hstack((x, t))
    return tf.constant(X, dtype=tf.float32)

Nb = 100  # number of training data for BC
Ni = 200  # number of training data for IC
Np = 4000 # number of training data for PDE

Xb = BC_data(Nb)
Xi = IC_data(Ni)
Xp = PDE_data(Np)

# Create a dataset object for Xp
Xp_DS = tf.data.Dataset.from_tensor_slices(Xp)
# Shuffle and batch data
Xp_DS = Xp_DS.shuffle(buffer_size=Np).batch(256)
```

After all the above steps, we can now start the training process. This training process comprises the following key steps: (1) We initialize an Adam optimizer[39] for updating the network parameters; (2) we implement an outer training loop to perform training for multiple epochs; and (3) we introduce an inner training loop for iterating over all batches. In each batch, we input the training data for the PDE, BC, and IC to evaluate the total loss. Subsequently, we compute the gradients of this loss with respect to all the training variables

(i.e., weights and biases). These gradients are then used by the Adam optimizer to update and optimize the network's weights and biases. The complete code implementation in TensorFlow can be found in the code in Listing 4.

Listing 4. Train neural network to find the optimal parameters for minimizing the total loss.

```
# Step 3.1. Define optimizer
optimizer=tf.keras.optimizers.Nadam(learning_rate=0.001)
# Step 3.2. Define outer loop
epochs = 3000
# Loop over epochs, each epoch contains all batches
# Each batch contains batch_size data
for epoch in range(epochs):
  # Step 3.3 Define inner loop
  for Xp_batch in Xp_DS:
    # a) read data from Xp dataset
    with tf.GradientTape() as g:
      # b) calculate loss
      loss_total, loss_PDE, loss_IC, loss_BC =
      loss(model, Xp_batch, Xi, Xb)
      # c) calculate gradient
      grads = g.gradient(loss_total,
                model.trainable_variables)
      # d) update weights
      optimizer.apply_gradients(zip(grads,
                model.trainable_variables))

  if (epoch+1) % 100 == 0:
    print('Epoch: %d, Loss: %f, Loss_PDE: %f,
    Loss_IC: %f, Loss_BC: %f'
    % (epoch+1, loss_total, loss_PDE, loss_IC, loss_BC))
```

It is worth noting that although the optimizer automatically updates the weights and biases of a neural network, determining other parameters—such as network size, activation function, batch size, optimizer type, and learning rate—poses a significant challenge. These parameters, known as hyperparameters, play a crucial role in training. A common and straightforward method for identifying hyperparameters is grid searching or trial-and-error, where various combinations are tested to select the most effective one. As an alternative, Bayesian optimization offers an automated solution for finding an optimal set of hyperparameters.[40] Some open-source tools such as Optuna can be used for this purpose.

After the training process, the neural network can be viewed as an analytical model that approximates the true solution of the problem. Figure 2 compares the predictions of PINN and the actual ground truth at different time instances. The ground truth is obtained by the finite different methods (FDMs). Despite slight discrepancies near the boundaries, the overall trend of phase separation is well captured, which indicates the effectiveness of the proposed framework. In this Tutorial, we will not delve into error analysis. Such an analysis can be conducted by creating a separate testing dataset and evaluating various error metrics. In addition, by repeating the training multiple times, the uncertainty associated with the predictions can be assessed.

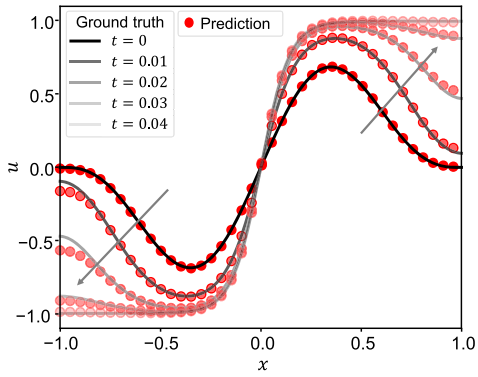

FIG. 2. Comparison of PINN predictions and ground truth at different time steps.

V. DEEP OPERATOR LEARNING FOR PHASE-FIELD MODELS

The above example showcases the capability of PINNs to solve problems given any PDEs, BCs, and ICs. A notable limitation of PINNs, however, is their requirement for retraining whenever the BCs or ICs are changed. To address this challenge, we present an alternative approach within the realm of SciML: namely DeepONet. DeepONet offers solutions to PDEs that can be adaptable to any BCs and ICs without the need for retraining. This section introduces the DeepONet-based framework to solve phase-field equations as shown in Fig. 1(c). We will particularly focus on customizing the DeepONet structure and incorporating a physics-informed loss function to effectively solve phase-field models. We will also provide a numerical example to illustrate the practical implementation of the proposed framework in Sec. VI.

A. Structure of DeepONet

While PINN focuses on constructing the loss function to enforce physical laws, DeepONet focuses on the neural network architecture to better capture complex solutions. DeepONet was first proposed by Lu *et al.*[20] The key idea behind it is to treat the neural network as an operator that approximates the unknown mapping between functions. DeepONet is a high-level network structure with two sub-networks, the "branch" and the "trunk" networks, as shown in Fig. 3. The trunk network takes the coordinates $x \in \mathbb{R}^D$ as the input, while the branch network takes the discretized values or features extracted from a continuous function $u \in \mathbb{R}^M$ as the input. The mapping $\mathcal{N} : (x, u) \to \mathbb{R}$ is given by multiplying the outputs of both networks,

$$v = \mathcal{N}(x, u; \theta) = B(u; \theta_b) \cdot T(x; \theta_t) + b_0, \tag{16}$$

where $v \in \mathbb{R}$ is the scalar output; $B \in \mathbb{R}^P$ and $T \in \mathbb{R}^P$ represent the branch network and trunk network, respectively; the outputs of both networks have the same dimension P to enable the dot product; and $b_0 \in \mathbb{R}$ is a bias. θ_b and θ_t represent the unknown parameters in the branch and trunk nets, respectively; $\theta = \theta_b, \theta_t$ denotes all the unknown parameters.

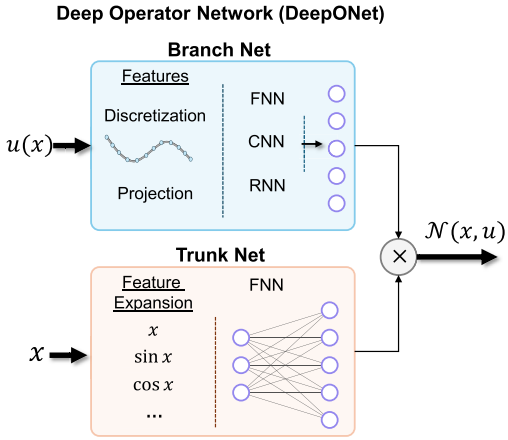

FIG. 3. Illustration of DeepONet. It consists of two sub-networks: branch net and trunk net. Feature expansion or extraction can be performed before feeding into the networks to potentially enhance the generalizability.

DeepONets have demonstrated superior generalization capabilities over traditional Feedforward Neural Networks (FNNs) as highlighted by Lu *et al.*[20] Moreover, DeepONet has several key attributes that align well with our specific problem requirements. First, by taking spatial coordinates as the input of the trunk network, DeepONet enables continuous predictions across any point within the domain. This further makes it possible to get the gradients through automatic differentiation, which is fundamental for the construction of physics-informed loss functions. Second, DeepONet's design offers exceptional versatility, making it suitable for various phase-field models across different dimensions. This flexibility stems from its multi-network architecture, which separates vector and function inputs into the trunk and branch networks, respectively, thus facilitating adaptation to a wide range of applications. Within the trunk network, spatial coordinates can undergo feature expansion [e.g., transforming (x) into $(x, \sin(x), \cos(x))$], enhancing the model's input representation. The branch network allows for feature extraction from function inputs, such as magnitude and phase in the frequency domain after Fourier-related transformations. Furthermore, both sub-networks can adopt any neural network architecture, including FNN, CNN, and RNN, offering additional adaptability.

B. DeepONet as a time-stepper

DeepONet can be tailored to capture the temporal evolution of the field variable in phase-field models. The goal is to predict its future states based on the current state. This is achieved by treating the current distribution of a field variable as the input and its future distribution as the output, essentially creating a mapping from u_k to u_{k+1}, which can be written as $u_{k+1} = \mathcal{N}(x, u_k; \theta)$ [see Fig. 1(c)]. This setup allows the network to function like an explicit time-stepper: given the state of the field variable at the kth time step u_k, it predicts the state at the $(k+1)$th time step u_{k+1}. This prediction can then serve as the new input for predicting u_{k+2} and so on. Through this iterative process, we can generate a sequence of predicted distributions $[u_k, u_{k+1}, u_{k+2}, \ldots]$ for successive time steps.

C. Free energy-guided training

Our strategy utilizes physics-informed training to find the solution. One method involves training the DeepONet with an Euler explicit time-stepper. This can be done by reducing the discrepancy between DeepONet's predictions and those produced by the Euler explicit method. However, this technique necessitates a very small time step to guarantee solution convergence and requires the evaluation of higher-order derivatives. As an alternative, we employ a free energy-based minimizing movement scheme to solve the problem, as suggested in our previous work.[25] With this approach, given a small fixed time step $\tau > 0$ and a free energy functional $\mathcal{G}(u)$, we can generate a time sequence of u, denoted as $[u_1, u_2, \ldots, u_n]$, starting from the initial condition u_0, through an iterative process known as the minimizing movement scheme,

$$u_{k+1} \in \min_u \left[\mathcal{G}(u) + \frac{d^2(u, u_k)}{2\tau} \right], \tag{17}$$

where $u_k = u(x, t = k\tau)$ and $u_{k+1} = u(x, (k+1)\tau)$ represent the field variable distribution at time steps k and $k+1$, respectively; $d(.,.)$ is a two-element operator calculating the distance between two functions. Different definitions of this distance will result in different physical laws. More details can be found in our previous work.[25] Intuitively, the total free energy will be eventually minimized when approaching the equilibrium state (distance approaches zero), and distance controls the path (speed and direction) of the decreasing free energy. The loss function can therefore be constructed following Eq. (17),

$$\mathcal{L} = \mathcal{G}(u_{k+1}) + \frac{d^2(u_{k+1}, u_k)}{2\tau}. \tag{18}$$

The training process will minimize the above total loss, which is essentially the minimizing movement scheme that can predict the field evaluation of phase-field models. The evaluation of the above loss function requires numerical integration, as well as a properly sampled training dataset. We will explain the detailed training process in the following example.

VI. SOLVING ALLEN–CAHN EQUATION WITH DeepONet

We apply the above DeepONet-based framework to the Allen–Cahn equation, as previously introduced in Eq. (13), with a modification to the BC. Dirichlet BCs $u(x = \pm 1) = 0$ are applied instead of Neumann BCs. Unlike the previous example for PINN, the IC is not specified, since we aim to find the solution of the PDE with any given random ICs. Figure 4 depicts the entire framework, which consists of three fundamental steps: (1) developing a suitable network architecture, (2) constructing a physics-informed loss function, and (3) preparing a training dataset and training the network. Each of these steps will be detailed in the following discussion. The complete code is accessible in the GitHub repository.

A. Neural network

We first construct a DeepONet consisting of two FNNs as sub-networks (Fig. 4). The trunk net (three layers, 100 nodes for each layer) and branch net (two layers, 100 nodes for each layer) take

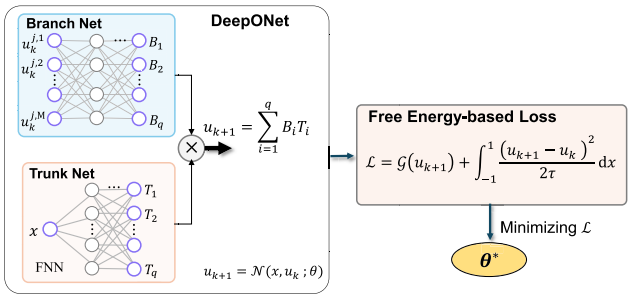

FIG. 4. DeepONet-based framework for solving 1D Allen–Cahn equations. It consists of a DeepONet with two FNNs, a free-energy-based loss function, and an optimizer.

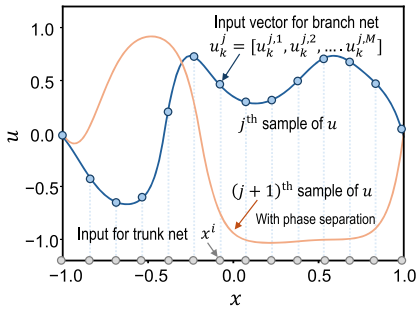

FIG. 5. Randomly sampled distributions of the field variable. The blue and orange solid lines indicate two different continuous distributions of u without and with significant phase separation, respectively. The gray circles indicate the collocated data points for the input of the trunk net, and the red circles are the discretized input vector for the branch net.

the spatial coordinate $x \in \mathbb{R}$ and discretized field variable $u_k \in \mathbb{R}^M$ as the inputs, respectively. We set $M = 20$ (dimension of the input for branch net) and $P = 100$ (dimension of the output for both branch and trunk nets) in this case. The final output $u_{k+1} \in \mathbb{R}$ is the field value at the next time step. It can be written as $u_{k+1} = \mathcal{N}(x, u_k; \theta)$, where θ denotes all the weights and biases of the network.

B. Loss function

We then construct the loss function. It is important to note that while the free energy-based loss inherently accounts for the PDE and Neumann type BCs, it does not cover Dirichlet type BCs.[25,33] Therefore, we must explicitly incorporate the Dirichlet BC. One method is to add the Dirichlet BC residue directly to the total loss function as in Eq. (15). Alternatively, we can modify the network's output to $u_{k+1} = \mathcal{N}(x, u_k; \theta)(1+x)(1-x)$. This ensures that the BCs are automatically met ($u_{k+1} = 0$, when $x = \pm 1$). We then construct the loss function based on Eq. (18),

$$\mathcal{L}_{\text{ONet}} = \frac{1}{N_u} \sum_{j=1}^{N_u} \left[\mathcal{G}(u_{k+1}) + \frac{d^2(u_{k+1}, u_k^j)}{2\tau} \right], \tag{19}$$

where u_k^j represents the jth sample of the discretized u at the current state, with N_u denoting the total number of sampled u_k. The time step τ is set to 0.002. The total loss is computed as the average loss across all N_u sampled u_k. For each sample, the free energy term $\mathcal{G}$ [see Eq.(5)] and the distance term d are calculated as follows:

$$\begin{aligned} \mathcal{G}(u_{k+1}) &= \int_{-1}^{1} \left[g(u_{k+1}) + \frac{1}{2} \frac{\partial u_{k+1}}{\partial x} \right] dx \\ &= \int_{-1}^{1} \left[\frac{1}{4\varepsilon^2} \left(1 - u_{k+1}^2\right)^2 + \frac{1}{2} \left(\frac{\partial u_{k+1}}{\partial x} \right)^2 \right] dx \\ &\approx \frac{2}{N_x} \sum_{i=1}^{N_x} \left[\frac{1}{4\varepsilon^2} \left(1 - u_{k+1}(x^i, u_k^j)^2\right)^2 + \frac{1}{2} \left(\frac{\partial u_{k+1}}{\partial x} \right)^2 \right] \end{aligned} \tag{20}$$

and

$$\begin{aligned} d^2(u_{k+1}, u_k^j) &= \int_{-1}^{1} (u_{k+1} - u_k^j)^2 dx \\ &\approx \frac{2}{N_x} \sum_{i=1}^{N_x} \left[u_{k+1}(x^i, u_k^j) - u_k^j \right]^2. \end{aligned} \tag{21}$$

The integrals in Eqs. (20) and (21) are approximated using the Monte Carlo integration method, where x^j represents the ith sampled location within space domain $[-1, 1]$.

C. Data preparation and training

The training data for DeepONet consist of input pairs (x^i, u_k^j). For ease of numerical integration, x^j is uniformly sampled within the interval $[-1, 1]$, and the discretized u_k^i is extracted at these same x^j locations, as illustrated in Fig. 5. Generating random distributions for u poses a challenge, which is addressed by employing a Gaussian random process with subsequent modifications to meet BCs. For further details, readers are directed to the GitHub repository. It is important to note that the efficacy of training is significantly impacted by the quality of the training dataset. We recommend that the dataset encompasses a wide range of representative cases to ensure comprehensive physical relevance. In this example, we should generate samples with and without distinct phase separation (see Fig. 5) to capture the whole phase transition process.

Figure 6 compares the DeepONet predictions with the ground truth at different times. A phase separation can be observed with increasing time and well captured by the DeepONet. Figure 6(a) shows the predicted phase evolution given a sinusoidal initial distribution, which is outside the training dataset. More importantly, given any random initial distribution, the trained DeepONet can well predict the phase separation dynamics [see Fig. 6(b)]. In other words, DeepONet does not require retraining when the initial condition changes.

While the complete code is available in the GitHub repository, we want to emphasize a key aspect of code implementation. In machine learning code, variables are conceptualized as tensors (or matrices), and their dimensions are critical to successfully implementing machine learning algorithms. When constructing the network structure, it is essential to recognize that the inputs and outputs represent a single general sample. For instance, in Fig. 4, the input of the trunk net, denoted as $x \in \mathbb{R}$, is a zero-dimensional (0-D) tensor (scalar). The input of the branch net is a 1D tensor (vector), and the output is a 0D tensor representing the value of u evaluated at location x. During training, however, these tensors accommodate

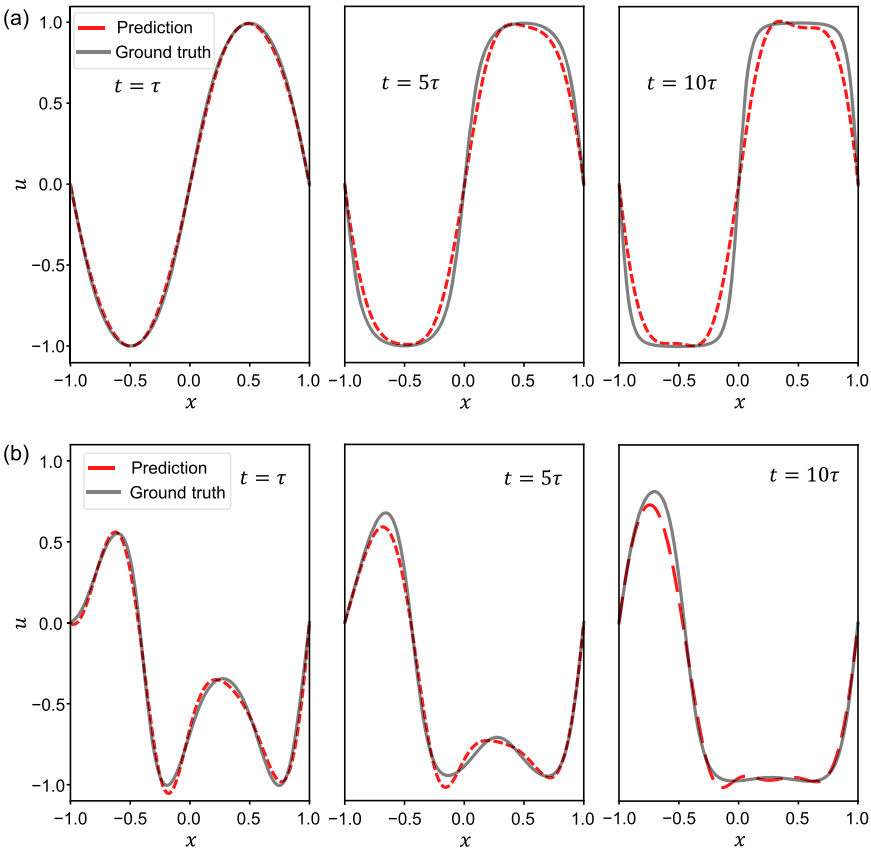

FIG. 6. DeepONet predictions for 1D Allen–Cahn equations given a sinusoidal (a) and a random (b) initial condition. The phase separation can be captured in both cases.

multiple samples within a batch, introducing an additional dimension. In the example above, the input of the trunk net becomes $x = [x^1, x^2, \ldots, x^{N_x}]^T$, constituting a 1D tensor ($N_x \times 1$ matrix), where each element (each row) corresponds to a distinct training sample. Similarly, the tensor $u_k = u_k^{j,l}$, $j = 1, 2, .., N_u$, $l = 1, 2, \ldots, M$, becomes a 2D tensor ($N_u \times M$ matrix), with each row representing a training sample for the branch net and each column signifying one feature (discretized value of u_k). The output is then a 2D tensor ($N_u \times N_x$ matrix), where each row is the spatial distribution of u at the N_x locations corresponding to a given sample of u_k^j. In addition, it is important to highlight that the number and dimensions of weights and biases of a neural network are fixed once the structure is determined and remain unaffected by the number of samples.

VII. DISCUSSION

A. Comparison of computation efficiency

Our goal of employing ML algorithms in scientific computing is to achieve high computational efficiency. To demonstrate this, we compare the computation time of ML-based methods with traditional numerical approaches using the Allen–Cahn equation as an example. Before making this comparison, it is crucial to note that the training process for ML models, essentially an optimization procedure, often requires significant time and involves a trade-off in prediction accuracy. For the 1D Allen–Cahn equation, we observed a training time of ~12 and 20 min for the PINN-based and DeepONet-based methods, respectively, using a standard desktop with an NVIDIA 4070 GPU. This training time is substantially longer than the computation time of traditional numerical methods. However, once trained, these neural networks offer faster prediction times, as they only require forward passes (evaluation of the neural network). The PINN-based framework, being continuous in time, can provide predictions at any time point with a single pass, resulting in extremely rapid computations. In contrast, the DeepONet-based framework, discretized in the time domain, requires multiple passes for predictions over an extended time span. For instance, DeepONet trained with a time step of 0.002 necessitates 100 evaluations to predict over a time span of 0.2. This is similar to traditional time-steppers, where an explicit time-stepper (e.g., explicit Euler) often requires a much smaller time step for convergence, while an implicit time-stepper (e.g., implicit Euler) can use a larger time step due to its unconditional stability. In practice, the largest stable time

TABLE I. Comparison of computation efficiency of DeepONet and traditional numerical methods (explicit Euler and Crank–Nicolson).

	Explicit Euler	Crank–Nicolson	DeepONet
Stable time step (s)	5×10^{-5}	2×10^{-3}	2×10^{-3}
Number of time steps	4000	100	100
Computation time (s)	2.09	0.19	0.28

step can be determined by starting with a conservative (small) time step and gradually increasing it to a threshold at which the solution begins to exhibit unstable behaviors. Table I compares the stable time step and computation time of DeepONet, explicit Euler, and implicit Crank–Nicolson methods. DeepONet emerges as the fastest for two primary reasons: (1) It incorporates a minimizing movement scheme during training, equivalent to implicit methods, allowing for larger stable time steps. Consequently, it requires fewer evaluations (time steps) compared to explicit time-steppers for predicting an extended time span. (2) Unlike implicit methods that solve systems of nonlinear algebraic equations at each time step (often requiring multiple iterations), DeepONet needs only one forward pass evaluation per time step.

However, it is important to note that the relative performance of ML approaches may vary depending on problem complexity, specific training processes, and the particular numerical methods used for comparison. Therefore, while ML methods show great promise for enhancing computational efficiency in scientific applications, their performance should be carefully evaluated for each specific use case.

B. Extension to high-dimensional problems

While the outlined workflow can be applied to a wide range of problems, it is crucial to acknowledge the significant challenges that arise when extending these methods to more complex scenarios such as high-order and high-dimensionality problems. These challenges, which remain active areas of research, include several aspects:

1. Network Architectures. When extending to higher-dimensional domains (2D and 3D), we need suitable network architectures that effectively capture complex solutions. For example, 2D or 3D problems with a regular domain may benefit from incorporating convolutional layers to capture spatial correlations.[25] However, the network architecture should be further innovated to accommodate irregular domains.[41] In the specific case of phase-field models, network structures should also be designed to effectively represent sharp transitions along phase boundaries.
2. Training Strategy. As network size increases to accommodate these complexities, the training process becomes more computationally expensive. The purely physics-informed training process, in particular, can be inefficient due to the gradient calculation in the training loss, especially for high-order equations. To address this, hybrid training processes combining data-driven and physics-informed approaches can be explored to improve efficiency.
3. Model Reliability. As these methods are applied to more complex problems, it becomes increasingly important to evaluate the uncertainty and robustness of the trained models to ensure reliable predictions. This involves developing robust validation techniques and uncertainty quantification methods specific to physics-informed machine learning models.

Addressing these challenges is crucial for the broader application of machine learning methods in scientific computing, particularly in fields involving complex physical phenomena such as phase-field modeling.

VIII. SUMMARY

Using machine learning algorithms to solve scientific governing equations is an emerging technique. This Tutorial introduces two neural network methods for solving phase-field models, exemplified by their implementation in code for the 1D Allen–Cahn equation. In a broader context, the application of neural network methods to scientific or engineering problems can be guided by the following steps: (1) Identify independent (inputs) and dependent (outputs) variables, and select or design a neural network structure that best represents the relationship between inputs and outputs. (2) Construct a loss function to guide the training process, which can be purely data-driven, fully physics-informed, or a hybrid of both depending on the problem. (3) Train network and validation. The training dataset should be well-prepared to encompass a comprehensive range of cases with physical relevance.

The two frameworks presented, the PINN-based and the DeepONet-based approaches, differ in the first two steps: (1) For the network structure, the PINN-based approach utilizes a fully connected neural network architecture, whereas the DeepONet-based approach employs the DeepONet structure, which is designed to serve as a time-stepper. (2) For the loss function, the PINN-based approach adopts a residual-based loss function, while the DeepONet-based approach utilizes a loss function based on the total free energy of the system. These lead to differences in the types of problems that each framework is best suited to address. The PINN-based approach is well-equipped to handle small-scale problems with known BCs and ICs. In contrast, the DeepONet-based framework is more suitable for tackling more complex scenarios where the BCs or ICs may be random.

Furthermore, we compared the computational efficiency between ML-based methods and traditional numerical methods. Our findings indicate that neural networks, once trained, potentially offer faster execution times. However, this advantage comes with notable trade-offs: (1) ML models require a significant initial time investment for dataset preparation and training, and (2) there may be a compromise in accuracy compared to well-established numerical methods.

Finally, we explored the challenges and opportunities in extending these ML frameworks to more complex problems. Key considerations include (1) designing appropriate network architectures for higher-dimensional and irregular domains, (2) developing efficient training strategies, and (3) ensuring model reliability through uncertainty quantification techniques.

ACKNOWLEDGMENTS

W.L. and J.Z. acknowledge the support of the present work through the NASA 19-TTT-0103 project (Award No. 80NSSC24M0009). They are also supported by the Northeastern University and College of Engineering startup funds.

AUTHOR DECLARATIONS

Conflict of Interest

The authors have no conflicts to disclose

Author Contributions

Wei Li: Conceptualization (equal); Formal analysis (equal); Project administration (equal); Writing – original draft (equal); Writing – review & editing (equal). **Ruqing Fang**: Writing – review & editing (equal). **Junning Jiao**: Writing – review & editing (equal). **Georgios N. Vassilakis**: Writing – review & editing (supporting). **Juner Zhu**: Conceptualization (equal); Funding acquisition (equal); Project administration (equal).

DATA AVAILABILITY

The data that support the findings of this study are openly available in the GitHub repository (https://github.com/weili101/Phase-Field_DeepONet/tree/main/Tutorial).

APPENDIX: FINITE DIFFERENCE METHOD SOLVING ALLEN–CAHN EQUATION

We discretize the space and time domain with an equal space and time interval of h and τ, respectively. The solution is then represented by discrete values u_k^j, where $k = 1, 2, \ldots, N_x$ indicates the N_x discretized spatial locations (nodes) and $j = 1, 2, \ldots, N_t$ denotes the N_t time instances. We want to point out that the notation here is different from what we use in the main context, where the subscript represents the discretized time domain. The second spatial derivative in Eq. (6) is approximated by

$$\frac{\partial^2 u_k^j}{\partial x^2} = \frac{u_{k+1}^j - 2u_k^j + u_{k-1}^j}{h^2}. \tag{A1}$$

We then introduce three approaches to approximate the first-order time derivative, namely the explicit Euler, implicit Euler, and Crank–Nicolson methods. With the explicit Euler method (explicit time-stepper), we have

$$\frac{\partial u^j}{\partial t} \approx \frac{u_k^{j+1} - u_k^j}{\tau}. \tag{A2}$$

The Allen–Cahn equation can then be discretized as

$$\frac{u_k^{j+1} - u_k^j}{\tau} = \frac{u_{k+1}^j - 2u_k^j + u_{k-1}^j}{h^2} + \frac{u_k^j - (u_k^j)^3}{\varepsilon^2}. \tag{A3}$$

For Dirichlet BCs, we set $u_0^j = u_{N_x}^j = 0$ for all j. Given the initial condition u_k^0, we can then iteratively obtain the unknown u_k^{j+1} for the next time step by substituting the known $u_{k-1}^j, u_k^j, u_{k+1}^j$ at the current step. This process is straightforward and computationally efficient; however, it requires a very small time step to guarantee solution convergence.

With the implicit Euler time-stepper, the time derivative is approximated slightly differently by

$$\frac{\partial u^{j+1}}{\partial t} \approx \frac{u_k^{j+1} - u_k^j}{\tau}. \tag{A4}$$

The Allen–Cahn equation can then be discretized as

$$\frac{u_k^{j+1} - u_k^j}{\tau} = \frac{u_{k+1}^{j+1} - 2u_k^{j+1} + u_{k-1}^{j+1}}{h^2} + \frac{u_k^{j+1} - (u_k^{j+1})^3}{\varepsilon^2}. \tag{A5}$$

We then need to solve the above system of nonlinear algebraic equations at each time step for u_k^{j+1} given u_k^j. This can be done by using the iterative Newton–Raphson method. With the Taylor expansion, the above-mentioned equation can be linearized as

$$F(\tilde{u}_{k-1}, \tilde{u}_k, \tilde{u}_{k+1}) + \frac{\partial F}{\partial u_{k-1}^{j+1}}\Delta\tilde{u}_{k-1} + \frac{\partial F}{\partial u_k^{j+1}}\Delta\tilde{u}_k + \frac{\partial F}{\partial u_{k+1}^{j+1}}\Delta\tilde{u}_{k+1} = 0, \tag{A6}$$

where F is the residual of Eq. (A5) (left-hand side minus right-hand side), and the partial derivatives can be derived as

$$\begin{aligned} \frac{\partial F}{\partial u_{k-1}^{j+1}} &= -\frac{1}{h^2}, \\ \frac{\partial F}{\partial u_k^{j+1}} &= \frac{1}{\tau} + \frac{2}{h^2} - \frac{1 - 3(\tilde{u}_k)^2}{\varepsilon^2}, \\ \frac{\partial F}{\partial u_{k+1}^{j+1}} &= -\frac{1}{h^2}. \end{aligned} \tag{A7}$$

We use $\tilde{u}_k$ to denote the trial solution of u_k^{j+1} and $\delta\tilde{u}_k$ to denote the increment from the trial solution. After solving the linearized system, we can update the solution as $\tilde{u}_k \leftarrow \tilde{u}_k + \Delta\tilde{u}_k$. The above process is repeated to get the converged solution ($u_k^{j+1} = \tilde{u}_k$) when the residual is smaller than a predefined tolerance. Compared with the explicit Euler method, the implicit method requires solving a system of nonlinear algebraic equations at each time step, which can be computationally expensive. However, the implicit method is unconditionally stable and can use a larger time step that facilitates computation efficiency.

The Crank–Nicolson method is a semi-implicit method that combines the explicit Euler method and the implicit Euler method. The time derivative is approximated by the average of the explicit and implicit Euler methods. The Allen–Cahn equation can be discretized as

$$\frac{u_k^{j+1} - u_k^j}{\tau} = \frac{1}{2}(F^{j+1} + F^j), \tag{A8}$$

where F^{j+1} and F^j are the right-hand side of the Allen–Cahn equation at time steps $j+1$ and j, respectively. Similar to the implicit Euler method, we need to solve a system of nonlinear algebraic equations with the same linearization process. The complete code for solving the Allen–Cahn equation with the finite difference method can be found in the GitHub repository.

REFERENCES

[1] I. Steinbach, "Phase-field models in materials science," Modell. Simul. Mater. Sci. Eng. **17**, 073001 (2009).
[2] M. Z. Bazant, "Thermodynamic stability of driven open systems and control of phase separation by electro-autocatalysis," Faraday Discuss. **199**, 423–463 (2017).
[3] J.-Y. Wu, V. P. Nguyen, C. T. Nguyen, D. Sutula, S. Sinaie, and S. P. Bordas, "Phase-field modeling of fracture," Adv. Appl. Mech. **53**, 1–183 (2020).
[4] L. Q. Chen and J. Shen, "Applications of semi-implicit Fourier-spectral method to phase field equations," Comput. Phys. Commun. **108**, 147–158 (1998).
[5] S. M. Wise, C. Wang, and J. S. Lowengrub, "An energy-stable and convergent finite-difference scheme for the phase field crystal equation," SIAM J. Numer. Anal. **47**, 2269–2288 (2009).
[6] M. R. Tonks, D. Gaston, P. C. Millett, D. Andrs, and P. Talbot, "An object-oriented finite element framework for multiphysics phase field simulations," Comput. Mater. Sci. **51**, 20–29 (2012).
[7] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, "Machine learning–accelerated computational fluid dynamics," Proc. Natl. Acad. Sci. U. S. A. **118**, e2101784118 (2021).
[8] R. Vinuesa and S. L. Brunton, "Enhancing computational fluid dynamics with machine learning," Nat. Comput. Sci. **2**, 358–366 (2022).
[9] S. Goswami, M. Yin, Y. Yu, and G. E. Karniadakis, "A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials," Comput. Methods Appl. Mech. Eng. **391**, 114587 (2022).
[10] V. Botu and R. Ramprasad, "Adaptive machine learning framework to accelerate *ab initio* molecular dynamics," Int. J. Quantum Chem. **115**, 1074–1083 (2015).
[11] J. Li and S. A. Lopez, "Machine learning accelerated photodynamics simulations," Chem. Phys. Rev. **4**, 031309 (2023).
[12] D. Montes de Oca Zapiain, J. A. Stewart, and R. Dingreville, "Accelerating phase-field-based microstructure evolution predictions via surrogate models trained by machine learning methods," npj Comput. Mater. **7**, 3 (2021).
[13] G. H. Teichert and K. Garikipati, "Machine learning materials physics: Surrogate optimization and multi-fidelity algorithms predict precipitate morphology in an alternative to phase field dynamics," Comput. Methods Appl. Mech. Eng. **344**, 666–693 (2019).
[14] K. Alhada-Lahbabi, D. Deleruyelle, and B. Gautier, "Machine learning surrogate model for acceleration of ferroelectric phase-field modeling," ACS Appl. Electron. Mater. **5**, 3894–3907 (2023).
[15] V. Oommen, K. Shukla, S. Goswami, R. Dingreville, and G. E. Karniadakis, "Learning two-phase microstructure evolution using neural operators and autoencoder architectures," npj Comput. Mater. **8**, 190 (2022).
[16] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," Nat. Rev. Phys. **3**, 422–440 (2021).
[17] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," IEEE Trans. Neural Networks **9**, 987–1000 (1998).
[18] 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," J. Comput. Phys. **378**, 686–707 (2019).
[19] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, 31 (Curran Associates, 2018).
[20] 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," Nat. Mach. Intell. **3**, 218–229 (2021).
[21] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, and G. E. Karniadakis, "A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data," Comput. Methods Appl. Mech. Eng. **393**, 114778 (2022); arXiv:2111.05512.
[22] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, "Fourier neural operator for parametric partial differential equations," arXiv:2010.08895 [cs, math] (2021).
[23] C. L. Wight and J. Zhao, "Solving Allen-Cahn and Cahn-Hilliard equations using the adaptive physics informed neural networks," arXiv:2007.04542 [cs, math] (2020).
[24] H. J. Hwang, C. Kim, M. S. Park, and H. Son, "The deep minimizing movement scheme," arXiv:2109.14851 [cs, math] (2021).
[25] W. Li, M. Z. Bazant, and J. Zhu, "Phase-field DeepONet: Physics-informed deep operator neural network for fast simulations of pattern formation governed by gradient flows of free-energy functionals," Comput. Methods Appl. Mech. Eng. **416**, 116299 (2023).
[26] J. D. Van der Waals, "The thermodynamic theory of capillarity under the hypothesis of a continuous variation of density," J. Stat. Phys. **20**, 200–244 (1979).
[27] J. W. Cahn and J. E. Hilliard, "Free energy of a nonuniform system. I. Interfacial free energy," J. Chem. Phys. **28**, 258–267 (1958).
[28] M. Z. Bazant, "Theory of chemical kinetics and charge transfer based on nonequilibrium thermodynamics," Acc. Chem. Res. **46**, 1144–1160 (2013).
[29] A. Voulodimos, N. Doulamis, A. Doulamis, E. Protopapadakis *et al.* "Deep learning for computer vision: A brief review," Comput. Intell. Neurosci. **2018**, 1.
[30] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing [review article]," IEEE Comput. Intell. Mag. **13**, 55–75 (2018).
[31] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild *et al.*, "Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence," Tech. Rep., USDOE Office of Science (SC), Washington, DC (USA), 2019.
[32] S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli, "Scientific machine learning through physics–informed neural networks: Where we are and what's next," J. Sci. Comput. **92**, 88 (2022).
[33] Z. Li, H. Zheng, N. Kovachki, D. Jin, H. Chen, B. Liu, K. Azizzadenesheli, and A. Anandkumar, "Physics-informed neural operator for learning partial differential equations," arXiv:2111.03794 [cs, math] (2021).
[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," arXiv:1603.04467 (2016).
[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 32 (Curran Associates, 2019).
[36] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," arXiv:1512.01274 (2015).
[37] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: Composable transformations of Python+NumPy programs," https://github.com/google/jax (2018).
[38] 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," https://openreview.net/forum?id=BJJsrmfCZ (2017).
[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv:1412.6980 (2014).
[40] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on Bayesian optimization," J. Electron. Sci. Technol. **17**, 26–40 (2019).
[41] J. He, S. Koric, D. Abueidda, A. Najafi, and I. Jasiuk, "Geom-DeepONet: A point-cloud-based deep operator network for field predictions on 3D parameterized geometries," Comput. Methods Appl. Mech. Engin. **429**, 117130 (2024).