Communications in Information and Systems
Volume 22, Number 2, 223–245, 2022

Neural-PDE: a RNN based neural network for solving time dependent PDEs

Yihao Hu, Tong Zhao, Shixin Xu, Lizhen Lin, and Zhiliang Xu

Partial differential equations (PDEs) play a crucial role in studying a vast number of problems in science and engineering. Numerically solving nonlinear and/or high-dimensional PDEs is frequently a challenging task. Inspired by the traditional finite difference and finite elements methods and emerging advancements in machine learning, we propose a sequence-to-sequence learning (Seq2Seq) framework called Neural-PDE, which allows one to automatically learn governing rules of any time-dependent PDE system from existing data by using a bidirectional LSTM encoder, and predict the solutions in next n time steps. One critical feature of our proposed framework is that the Neural-PDE is able to simultaneously learn and simulate all variables of interest in a PDE system. We test the Neural-PDE by a range of examples, from one-dimensional PDEs to a multi-dimensional and nonlinear complex fluids model. The results show that the Neural-PDE is capable of learning the initial conditions, boundary conditions and differential operators defining the initial-boundary-value problem of a PDE system without the knowledge of the specific form of the PDE system. In our experiments, the Neural-PDE can efficiently extract the dynamics within 20 epochs training and produce accurate predictions. Furthermore, unlike the traditional machine learning approaches for learning PDEs, such as CNN and MLP, which require great quantity of parameters for model precision, the Neural-PDE shares parameters among all time steps, and thus considerably reduces computational complexity and leads to a fast learning algorithm.

1. Introduction

The research of time-dependent partial differential equations (PDEs) is regarded as one of the most important disciplines in applied mathematics. PDEs appear ubiquitously in a broad spectrum of fields including physics, biology, chemistry, and finance, to name a few. Despite their fundamental importance, most PDEs can not be solved analytically and have to rely on numerical solving methods. Developing efficient and accurate numerical

schemes for solving PDEs, therefore, has been an active research area over the past few decades [1, 2, 3, 4, 5, 6]. Still, devising stable and accurate schemes with acceptable computational cost is a difficult task, especially when nonlinear and(or) high-dimensional PDEs are considered. Additionally, PDE models emerged from science and engineering disciplines usually require huge empirical data for model calibration and validation, and determining the multi-dimensional parameters in such a PDE system poses another challenge [7].

Deep learning is considered to be the state-of-the-art tool in classification and prediction of nonlinear inputs, such as image, text, and speech [8, 9, 10, 11, 12]. Recently, considerable efforts have been made to employ deep learning tools in designing data-driven methods for solving PDEs [13, 14, 15, 16]. Most of these approaches are based on fully-connected neural networks (FCNNs), convolutional neural networks(CNNs) and multilayer perceptron (MLP). These neural network structures usually require an increment of the layers to improve the predictive accuracy [16], and subsequently lead to a more complicated model due to the additional parameters. Recurrent neural networks (RNNs) are another type of neural network architectures. RNNs predict the next time step value by using the input data from the current and previous states and share parameters across all inputs. This idea [17] of using current and previous step states to calculate the state at the next time step is not unique to RNNs. In fact, it is ubiquitously used in numerical PDEs. Almost all time-stepping numerical methods applied to solve time-dependent PDEs, such as Euler's, Crank-Nicolson, high-order Taylor and its variance Runge-Kutta [18] time-stepping methods, update numerical solution by utilizing solution from previous steps.

This motivates us to think what would happen if we replace the previous step data in the neural network with numerical solution data to PDE supported on grids. It is possible that the neural network behaves like a time-stepping method, for example, forward Euler's method yielding the numerical solution at a new time point as the current state output [19]. Since the numerical solution on each of the grid point (for finite difference) or grid cell (for finite element) computed at a set of contiguous time points can be treated as neural network input in the form of one time sequence of data, the deep learning framework can be trained to predict any time-dependent PDEs from the time series data supported on some grids if the bidirectional structure is applied [20, 21]. In other words, the supervised training process can be regarded as a practice of the deep learning framework to learn the numerical solution from the input data, by learning the coefficients on neural network layers.

Long Short-Term Memory (LSTM) [22] is a neural network built upon RNNs. Unlike vanilla RNNs, which suffer from losing long term information and high probability of gradient vanishing or exploding, LSTM has a specifically designed memory cell with a set of new gates such as input gate and forget gate. Equipped with these new gates which control the time to preserve and pass the information, LSTM is capable of learning long term dependencies without the danger of having gradient vanishing or exploding. In the past two decades, LSTM has been widely used in the field of natural language processing (NLP), such as machine translation, dialogue systems, question answering systems [23].

Inspired by numerical PDE schemes and LSTM neural network, we propose a new deep learning framework, denoted as Neural-PDE. It simulates multi-dimensional governing laws, represented by time-dependent PDEs, from time series data generated on some grids and predicts the next n time steps data. The Neural-PDE is capable of intelligently processing related data from all spatial grids by using the bidirectional [21] neural network, and thus guarantees the accuracy of the numerical solution and the feasibility in learning any time-dependent PDEs. The detailed structures of the Neural-PDE and data normalization are introduced in Section 3.

The rest of the paper is organized as follows. Section 2 briefly reviews finite difference method and finite element method for solving PDEs. Section 3 contains detailed description of designing the Neural-PDE. In Section 4, we apply the Neural-PDE to solve four different PDEs, including the 1-dimensional(1D) wave equation, the 2-dimensional(2D) heat equation, and two systems of PDEs: the invicid Burgers' equations and a coupled Navier Stokes-Cahn Hilliard equations, which widely appear in multiscale modeling of complex fluid systems. We demonstrate the robustness of the Neural-PDE, which achieves accuracy within 20 epochs with an admissible mean squared error, even when we add Gaussian noise in the input data.

2. Preliminaries

2.1. Time dependent partial differential equations

A time-dependent partial differential equation is an equation of the form:
(2.1.1)

$$u_t = f(x_1, \cdots, u, \frac{\partial u}{\partial x_1}, \cdots, \frac{\partial u}{\partial x_n}, \frac{\partial^2 u}{\partial x_1 \partial x_1}, \cdots, \frac{\partial^2 u}{\partial x_1 \partial x_n}, \cdots, \frac{\partial^n u}{\partial x_1 \cdots \partial x_n}) \ ,$$

where $u = u(x_1, ..., x_n, t)$ is known, $x_i \in \mathbb{R}$ are spatial variables, and the operator f maps $\mathbb{R}^N \mapsto \mathbb{R}$. For example, consider the parabolic heat equation: $u_t = \alpha^2 \Delta u$, where u represents the temperature and f is the Laplacian

operator Δ. Eq. (2.1.1) can be solved by finite difference methods, which are briefly reviewed below for the self-completeness of the paper.

2.2. Finite difference method

Consider using a finite difference method (FDM) to solve a two-dimensional second-order PDE of the form:

$$u_t = f(x, y, u_x, u_y, u_{xx}, u_{yy}), \quad (x, y) \in \Omega \subset \mathbb{R}^2, \quad t \in \mathbb{R}^+ \cup \{0\} \ , \tag{2.2.1}$$

with some proper boundary conditions. Let Ω be $\Omega = [x_a, x_b] \times [y_a, y_b]$, and

$$u^n_{i,j} = u(x_i, y_j, t_n) \tag{2.2.2}$$

where $t_n = n\delta t$, $0 \le n \le N$, and $\delta t = \frac{T}{N}$ for some large integer N. $x_i = i\delta x$, $0 \le i \le N_x$, $\delta x = \frac{x_a - x_b}{N_x}$. $y_j = j\delta y$, $0 \le j \le N_y$, $\delta y = \frac{y_a - y_b}{N_y}$. N_x and N_y are integers.

The central difference methods approximate the spatial derivatives as follows [5]:

$$u_x(x_i, y_j, t) = \frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j}) + \mathcal{O}(\delta x^2) \ , \tag{2.2.3}$$

$$u_y(x_i, y_j, t) = \frac{1}{2\delta y}(u_{i,j+1} - u_{i,j-1}) + \mathcal{O}(\delta y^2) \ , \tag{2.2.4}$$

$$u_{xx}(x_i, y_j, t) = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + \mathcal{O}(\delta x^2) \ , \tag{2.2.5}$$

$$u_{yy}(x_i, y_j, t) = \frac{1}{\delta y^2}(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}) + \mathcal{O}(\delta y^2) \ . \tag{2.2.6}$$

To this end, the explicit time-stepping scheme to update next step solution u^{n+1} is given by:

$$u^n_{i,j} \approx U^{n+1}_{i,j} = U^n_{i,j} + \delta t f(x_i, y_j, U^n_{i,j}, U^n_{i,j-1}, U^n_{i,j+1}, U^n_{i+1,j}, U^n_{i-1,j}) \ , \tag{2.2.7}$$

$$\equiv \mathbf{F}(x_i, y_j, \delta x, \delta y, \delta t, U^n_{i,j}, U^n_{i,j-1}, U^n_{i,j+1}, U^n_{i+1,j}, U^n_{i-1,j}) \ , \tag{2.2.8}$$

where $U^n_{i,j}$ is the numerical solution at grid point (x_i, y_j, t_n).

Apparently, the finite difference method (2.2.7) for updating u^{n+1} on a grid point relies on the previous time steps' solutions, supported on the grid point and its neighbours. The scheme (2.2.7) updates $u^{n+1}_{i,j}$ using five points of u^n values (see Figure 1).

Similarly, the finite element method (FEM) approximates the new solution by calculating the corresponded mesh cell coefficient, which is updated by its related nearby coefficients on the mesh.

From this perspective, one may regard the numerical schemes for solving time-dependent PDEs as methods catching the information from neighbourhood data of interest.

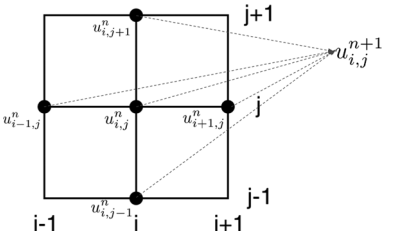


Figure 1: Updating scheme for central difference method.

2.3. Finite element method

Finite element method (FEM) is a powerful numerical method in solving PDEs. Consider a 1D wave equation of $u(x,t)$:

$$u_{tt} - v^2 u_{xx} = f, \quad x \in [a,b] \equiv \Omega \subset \mathbb{R}, \quad t \in \mathbb{R}^+ \cup \{0\} \ , \tag{2.3.1}$$

$$u_x(a,t) = u_x(b,t) = 0 \ . \tag{2.3.2}$$

The function u is approximated by a FEM function u_h:

$$u(x,t) \approx u_h(x,t) = \sum_{i=1}^{N} a_i(t)\psi_i(x) \tag{2.3.3}$$

where $\psi_i \in V$ is the basis functions of some FEM space V, and a_i^n denotes the coefficients. N denotes the degrees of freedom.
Multiply the equation with an arbitrary test function ψ_j and integral over the whole domain we have:

$$\int_\Omega u_{tt}\psi_j \ dx + v^2 \int_\Omega \nabla u \nabla \psi_j \ dx = \int_\Omega f\psi_j \ dx \tag{2.3.4}$$

and approximate $u(x,t)$ by u_h:

$$\sum_i^N \frac{\partial^2 a_i(t)}{\partial t^2} \underbrace{\int_\Omega \psi_i\psi_j \ dx}_{\mathbf{M}_{i,j}} + v^2 \sum_i^N a_i(t) \underbrace{\int_\Omega \nabla\psi_i \nabla\psi_j \ dx}_{\mathbf{A}_{i,j}} = \underbrace{\int_\Omega f\psi_j}_{\mathbf{b}} \ dx \ , \tag{2.3.5}$$

$$\equiv \mathbf{M}^T \mathbf{a}_{tt} + v^2 \mathbf{A}^T \mathbf{a} = \mathbf{b} \ . \tag{2.3.6}$$

Here $\mathbf{M}$ is the mass matrix and $\mathbf{A}$ is the stiffness matrix, $\mathbf{a} = (a_1, .., a_N)^t$ is a N vector of the coefficients at time t. The central difference method for time discretization indicates that [6]:

$$\mathbf{a}^{n+1} = 2\mathbf{a}^n - \mathbf{a}^{n-1} + \mathbf{M}^{-1}(\mathbf{b} - v^2\mathbf{A}^T\mathbf{a}^n) \ . \tag{2.3.7}$$

This leads to

$$u^{n+1} \approx u_h^{n+1} = \sum_i^N a_i^{n+1}\psi_i(x) \ . \tag{2.3.8}$$

2.4. Long short-term memory

Long Short-Term Memory networks (LSTM) [22, 24] are a class of artificial recurrent neural network (RNN) architecture that is commonly used for processing sequence data, and can overcome the gradient vanishing issue in RNN. Similar to most RNNs [25], LSTM takes a sequence $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_t\}$ as input and learns hidden vectors $\{\boldsymbol{h}_1, \boldsymbol{h}_2, \cdots, \boldsymbol{h}_t\}$ for each corresponding input. In order to better retain long distance information, LSTM cells are specifically designed to update the hidden vectors. The computation process of the forward pass for each LSTM cell is defined as follows:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\mathbf{W}_i^{(x)}\boldsymbol{x}_t + \mathbf{W}_i^{(h)}\boldsymbol{h}_{t-1} + \mathbf{W}_i^{(c)}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i) \ , \\
\boldsymbol{f}_t &= \sigma(\mathbf{W}_f^{(x)}\boldsymbol{x}_t + \mathbf{W}_f^{(h)}\boldsymbol{h}_{t-1} + \mathbf{W}_f^{(c)}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f) \ , \\
\boldsymbol{c}_t &= \boldsymbol{f}_t\boldsymbol{c}_{t-1} + \boldsymbol{i}_t \tanh(\mathbf{W}_c^{(x)}\boldsymbol{x}_t + \mathbf{W}_c^{(h)}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \ , \\
\boldsymbol{o}_t &= \sigma(\mathbf{W}_o^{(x)}\boldsymbol{x}_t + \mathbf{W}_o^{(h)}\boldsymbol{h}_{t-1} + \mathbf{W}_o^{(c)}\boldsymbol{c}_t + \boldsymbol{b}_o), \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \tanh(\boldsymbol{c}_t) \ ,
\end{aligned}$$

where σ is the logistic sigmoid function, $\mathbf{W}$s are weight matrices, $\boldsymbol{b}$s are bias vectors, and subscripts $\boldsymbol{i}$, $\boldsymbol{f}$, $\boldsymbol{o}$ and $\boldsymbol{c}$ denote the input gate, forget gate, output gate and cell vectors respectively, all of which have the same size as hidden vector $\boldsymbol{h}$.

This LSTM structure is used in the paper to simulate the numerical solutions of partial differential equations.

3. Proposed method

3.1. Mathematical motivation

Recurrent neural network including LSTM is an artificial neural network structure of the form [23]:

(3.1.1)
$$\boldsymbol{h}^t = \sigma(\mathbf{W}^{hx}\boldsymbol{x}^t + \mathbf{W}^{hh}\boldsymbol{h}^{t-1} + \boldsymbol{b}_h) \equiv \sigma_a(\boldsymbol{x}^t, \boldsymbol{h}^{t-1}) \equiv \sigma_b(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2, \cdots, \boldsymbol{x}^t) \ ,$$

where $\boldsymbol{x}^t \in \mathbb{R}^d$ is the input data of the t^{th} state and $\boldsymbol{h}^{t-1} \in \mathbb{R}^h$ denotes the processed value in its previous state by the hidden layers. The output $\boldsymbol{y}^t$ of the current state is updated by the current state value $\boldsymbol{h}^t$:

(3.1.2)
$$\boldsymbol{y}^t = \sigma(\mathbf{W}^{hy}\boldsymbol{h}^t + \boldsymbol{b}_y)$$

(3.1.3)
$$\equiv \sigma_c(\boldsymbol{h}^t) \equiv \sigma_d(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2, \cdots, \boldsymbol{x}^t) \ .$$

Here $\mathbf{W}^{hx} \in \mathbb{R}^{h\times d}$, $\mathbf{W}^{hh} \in \mathbb{R}^{h\times h}$, $\mathbf{W}^{hy} \in \mathbb{R}^{h\times h}$ are the matrix of weights, vectors $\boldsymbol{b}_h, \boldsymbol{b}_y \in \mathbb{R}^h$ are the coefficients of bias, and $\sigma, \sigma_a, \sigma_b, \sigma_c, \sigma_d$ are corresponded activation and mapping functions. With proper design of input and forget gate, LSTM can effectively yield a better control over the gradient flow and better preserve useful information from long-range dependencies [24].

Now consider a temporally continuous vector function $\boldsymbol{u} \in \mathbb{R}^n$ given by an ordinary differential equation with the form:

(3.1.4)
$$\frac{d\boldsymbol{u}(t)}{dt} = g(\boldsymbol{u}(t)) \ .$$

Let $\boldsymbol{u}^n = \boldsymbol{u}(t = n\delta t)$, a forward Euler's method for solving $\boldsymbol{u}$ can be easily derived from the Taylor's theorem which gives the following first-order accurate approximation of the time derivative:

(3.1.5)
$$\frac{d\boldsymbol{u}^n}{dt} = \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\delta t} + \mathcal{O}(\delta t) \ .$$

Then we have:

(3.1.6)
$$\frac{d\boldsymbol{u}}{dt} = g(\boldsymbol{u}) \xrightarrow{(3.1.5)} \boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \delta t \ g(\boldsymbol{u}^n) + \mathcal{O}(\delta t^2)$$

$$\rightarrow \hat{\boldsymbol{u}}^{n+1} = f_1(\hat{\boldsymbol{u}}^n) = \underbrace{f_1 \circ f_1 \circ \cdots f_1(\hat{\boldsymbol{u}}^0)}_{n}$$

Here $\hat{\boldsymbol{u}}^n \approx \boldsymbol{u}(n\delta t)$ is the numerical approximation and $f_1 \equiv \boldsymbol{u}^n + \delta t\ g(\boldsymbol{u}^n) : \mathbb{R}^n \to \mathbb{R}^n$. Combining equations (3.1.1) and (3.1.6) one may notice that the residual networks, recurrent neural network and also LSTM networks can be regarded as a numerical scheme for solving time-dependent differential equations if more layers are added and smaller time steps are taken. [19]

Canonical structure for such recurrent neural network usually calculates the current state value by its previous time step value $\boldsymbol{h}^{t-1}$ and current state input $\boldsymbol{x}^t$. Similarly, in numerical PDEs, the next step data at a grid point is updated from the previous (and current) values on its nearby grid points (see Eq. 2.2.7).

Thus, what if we replace the temporal input $\boldsymbol{h}^{t-1}$ and $\boldsymbol{x}^t$ with spatial information? A simple sketch of the upwinding method for a $1D$ example of $u(x,t)$:

$$u_t + \nu u_x = 0 \tag{3.1.7}$$

will be:

$$u_i^{n+1} = u_i^n - \nu\frac{\delta t}{\delta x}(u_i^n - u_{i-1}^n) + \mathcal{O}(\delta x, \delta t) \to \hat{u}_i^{n+1} = f_2(\hat{u}_{i-1}^n, \hat{u}_i^n) \tag{3.1.8}$$

$$\equiv f_\theta\big(f_\eta(\boldsymbol{x}_i, \boldsymbol{h}_{i-1}(u))\big) = f_{\theta,\eta}\big(\hat{u}_0^n, \hat{u}_1^n, \cdots, \hat{u}_{i-1}^n, \hat{u}_i^n\big) = v_i^{n+1} \tag{3.1.9}$$

$$\boldsymbol{x}_i = \hat{u}_i^n,\ \boldsymbol{h}_{i-1}(\hat{u}) = \sigma(\hat{u}_{i-1}^n, \boldsymbol{h}_{i-2}(\hat{u})) \equiv f_\eta(\hat{u}_0^n, \hat{u}_1^n, \hat{u}_2^n, \cdots, \hat{u}_{i-1}^n). \tag{3.1.10}$$

Here we use v_i^{n+1} to denote the prediction of $\hat{u}_i^{n+1}$ processed by neural network. We replace the temporal previous state $\boldsymbol{h}^{t-1}$with spacial grid value $\boldsymbol{h}_{i-1}$ and input the numerical solution $\hat{u}_i^n \approx u(i\delta x, n\delta t)$ as current state value, which indicates the neural network could be seen as a forward Euler method for equation 3.1.7 [26]. Function $f_2 \equiv \hat{u}_i^n - \nu\frac{\delta t}{\delta x}(\hat{u}_i^n - \hat{u}_{i-1}^n) : \mathbb{R}^2 \to \mathbb{R}$ and the function f_θ represents the dynamics of the hidden layers in decoder with parameters θ, and f_η specifies the dynamics of the LSTM layer [22, 24] in encoder withe the parameters η. The function $f_{\theta,\eta}$ simulates the dynamics of the Neural-PDE with parameters θ and η. By applying Bidirectional neural network, all grid data are transferred and it enables LSTM to simulate the PDEs as:

$$v_i^{n+1} = f_\theta\big(f_\eta(\boldsymbol{h}_{i+1}(\hat{\hat{u}}), \hat{u}_i^n, \boldsymbol{h}_{i-1}(\hat{u}))\big) \tag{3.1.11}$$

$$\boldsymbol{h}_{i+1}(\hat{u}) \equiv f_\eta(\hat{u}_{i+1}^n, \hat{u}_{i+2}^n, \hat{u}_{i+3}^n, \cdots, \hat{u}_k^n). \tag{3.1.12}$$

For a time-dependent PDE, if we map all our grid data into an input matrix which contains the information of $\delta x, \delta t$, then the neural network would

regress such coefficients as constants and will learn and filter the physical rules from all the k mesh grids data as:

$$v_i^{n+1} = f_{\theta,\eta}\big(\hat{u}_0^n, \hat{u}_1^n, \hat{u}_2^n, \cdots, \hat{u}_k^n\big) \tag{3.1.13}$$

The LSTM neural network is designed to overcome the vanishing gradient issue through hidden layers, therefore we use such recurrent structure to increase the stability of the numerical approach in deep learning. The highly nonlinear function $f_{\theta,\eta}$ simulates the dynamics of updating rules for u_i^{n+1}, which works in a way similar to a finite difference method (section 2.2) or a finite element method.

3.2. Neural-PDE

In particular, we use the bidirectional LSTM [22, 24] to better retain the state information from data on grid points which are neighbourhoods in the mesh but far away in input matrix.

The right frame of Figure 2 shows the overall design of the Neural-PDE. Denote the time series data at collocation points as $\boldsymbol{a}_1^N, \boldsymbol{a}_2^N, \cdots, \boldsymbol{a}_k^N$ with $\boldsymbol{a}_i^N = [\hat{u}_i^0, \hat{u}_i^1, \cdots, \hat{u}_i^N]$ at i^{th} point. The superscript represents different time points. The Neural-PDE takes the past states $\{\boldsymbol{a}_1^N, \boldsymbol{a}_2^N, \cdots, \boldsymbol{a}_k^N\}$ of all collocation points, and outputs the predicted future states $\{\boldsymbol{b}_1^M, \boldsymbol{b}_2^M, \cdots, \boldsymbol{b}_k^M\}$, where $\boldsymbol{b}_i^M = [v_i^{N+1}, v_i^{N+2}, \cdots, v_i^{N+M}]$ is the Neural-PDE prediction for the i^{th} collocation point at time points from $N+1$ to $N+M$. The data from time point 0 to N are the training data set.

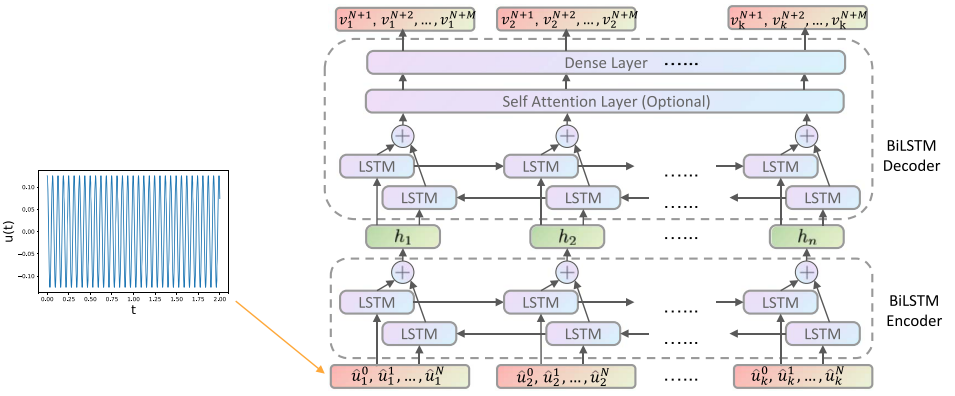


Figure 2: Neural-PDE.

The Neural-PDE is an encoder-decoder style sequence model that first maps the input data to a low dimensional latent space that

$$\boldsymbol{h}_i = \overrightarrow{\text{LSTM}}(\boldsymbol{a}_i) \oplus \overleftarrow{\text{LSTM}}(\boldsymbol{a}_i), \tag{3.2.1}$$

where $\oplus$ denotes concatenation and $\boldsymbol{h}_i$ is the latent embedding of point $\boldsymbol{a}_i$ under the environment.

One then decoder, another bi-lstm with a dense layer:

$$v_i = \left(\overrightarrow{\text{LSTM}}(\boldsymbol{h}_i) \oplus \overleftarrow{\text{LSTM}}(\boldsymbol{h}_i)\right) \cdot \mathbf{W}, \tag{3.2.2}$$

where $\mathbf{W}$ is the learnable weight matrix in the dense layer. Moreover, the final decode layers could also have an optional self attention [27] layer, which makes the model easier to learn long-range dependencies of the mesh grids.

During training process, mean squared error (MSE) loss $\mathcal{L}$ is used as we typically don't know the specific form of the PDE.

$$\mathcal{L} = \sum_{t=N+1}^{N+M} \sum_{i=1}^{k} ||\hat{u}_i^t - v_i^t||^2 \ , \tag{3.2.3}$$

3.3. Data initialization and grid point reshape

In order to feed the data into our sequence model framework, we map the PDE solution data onto a $K \times N$ matrix, where $K \in \mathbb{Z}^+$ is the dimension of the grid points and $N \in \mathbb{Z}^+$ is the length of the time series data on each grid point. There is no regularization for the input order of the grid points data in the matrix because of the bi-directional structure of the Neural-PDE. For example, a $2D$ heat equation at some time t is reshaped into a $1D$ vector (See Fig. 3). Then the matrix is formed accordingly.

For a n-dimensional time-dependent partial differential equation with K collocation points, the input and output data for $t \in (0, T)$ will

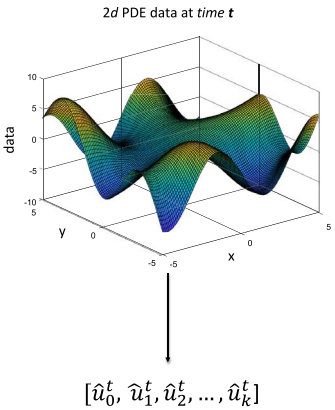


Figure 3: An example of maping $2d$ data matrix into $1d$ vector where $k = N_x \times N_y$ and N_x and N_y are the numbers of grid points on x and y, respectively.

be of the form:

$$\boldsymbol{A}(K,N) = \begin{bmatrix} \boldsymbol{a}_0^N \\ \vdots \\ \boldsymbol{a}_\ell^N \\ \vdots \\ \boldsymbol{a}_K^N \end{bmatrix} = \begin{bmatrix} \hat{u}_0^0 & \hat{u}_0^1 & \cdots & \hat{u}_0^n & \cdots & \hat{u}_0^N \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{u}_\ell^0 & \hat{u}_\ell^1 & \cdots & \hat{u}_\ell^n & \cdots & \hat{u}_\ell^N \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{u}_K^0 & \hat{u}_K^1 & \cdots & \hat{u}_K^n & \cdots & \hat{u}_K^N \end{bmatrix} \tag{3.3.1}$$

$$\boldsymbol{B}(K,M) = \begin{bmatrix} \boldsymbol{b}_0^M \\ \vdots \\ \boldsymbol{b}_\ell^M \\ \vdots \\ \boldsymbol{b}_K^M \end{bmatrix} = \begin{bmatrix} v_0^{N+1} & v_0^{N+2} & \cdots & v_0^{N+m} & \cdots & v_0^{N+M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ v_\ell^{N+1} & v_\ell^{N+2} & \cdots & v_\ell^{N+m} & \cdots & v_k^{N+M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ v_K^{N+1} & v_K^{N+2} & \cdots & v_K^{N+m} & \cdots & v_K^{N+M} \end{bmatrix} \tag{3.3.2}$$

Here $N = \frac{T}{\delta t}$ and each row ℓ represents the time series data at the ℓ^{th} mesh grid, and M is the time length of the predicted data.

By adding Bidirectional LSTM encoder in the Neural-PDE, it will automatically extract the information from the time series data as:

$$\boldsymbol{B}(K,M) = PDESolver(\boldsymbol{A}(K,N)) = PDESolver(\boldsymbol{a}_0^N, \boldsymbol{a}_1^N, \cdots \boldsymbol{a}_i^N, \cdots, \boldsymbol{a}_K^N) \tag{3.3.3}$$

4. Computer experiments

Table 1: Error analysis models

	Wave	Heat	Burgers'
Equation	$u_{tt} = \frac{1}{16\pi^2} u_{xx}$	$u_t = u_{xx}$	$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0.1\frac{\partial^2 u}{\partial x^2}$
IC	$\sin(4\pi x)$	$6\sin(\pi x)$	$u(0 \le x \le L, t = 0) = 0.9$
BC	*periodic*	*periodic*	*periodic*

Table 2: L^2 error for model evaluation

$\Delta x = 0.1$	Wave	Heat	Burgers'
$\Delta t = 0.1$	4.385×10^{-3}	6.912×10^{-5}	9.450×10^{-4}
$\Delta t = 0.01$	3.351×10^{-5}	5.809×10^{-5}	5.374×10^{-3}
$\Delta t = 0.001$	1.311×10^{-5}	3.757×10^{-5}	1.244×10^{-3}

Since the Neural-PDE is a sequence to sequence learning framework which allows to predict within any time period by the given data. One may

Table 3: L^2 error for model evaluation

$\Delta t = 0.1$	Wave	Heat	Burgers'
$\Delta x = 0.1$	2.190×10^{-5}	1.162×10^{-4}	2.561×10^{-4}
$\Delta x = 0.01$	6.059×10^{-5}	7.706×10^{-4}	4.206×10^{-4}
$\Delta x = 0.001$	1.498×10^{-5}	1.400×10^{-5}	3.700×10^{-4}

test the Neural-PDE using different permutations of training and predicting time periods for its efficiency, robustness and accuracy. In the following examples, the whole dataset is randomly splitted in 80% for training and 20% for testing. We will predict the next $t_p \in [31 \times \delta t, 40 \times \delta t]$ PDE solution by using its previous $t_{tr} \in [0, 30 \times \delta t]$ data as:

$$\boldsymbol{B}(K, 10) = PDESolver(\boldsymbol{A}(K, 30)) \tag{4.0.1}$$

We tested Neu-PDE using three classical PDE models with different Δx and Δt, Table 1 summarizes the information of these models. Table 2 and Table 3 show the experimental results of the Neural-PDE model solving the above three different PDEs. We used the Neural-PDE which only consists of 3 layers: 2 bi-lstm (encoder-decoder) layers with 20 neurons each and 1 dense output layer with 10 neurons and achieved MSEs from $\mathcal{O}(10^{-3})$ to $\mathcal{O}(10^{-5})$ within 20 epochs, a MLP based neural network such as Physical Informed Neural Network [16] usually will have more layers and neurons to achieve similar L^2 errors. Additional examples are also discussed in this section.

Example: wave equation

Consider the $1D$ wave equation:

$$u_{tt} = cu_{xx},\ x \in [0, 1],\ t \in [0, 2]\ , \tag{4.0.2}$$

$$u(x, 0) = sin(4\pi x) \tag{4.0.3}$$

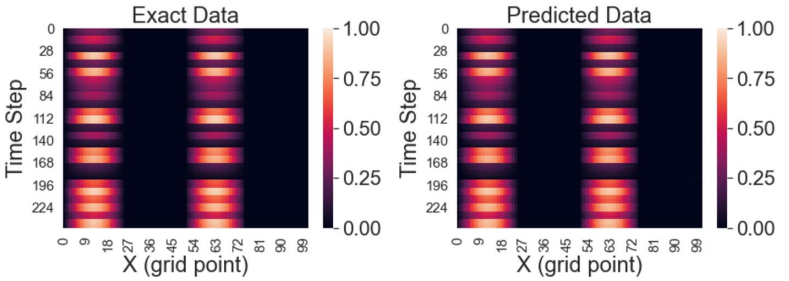
$$u(0, t) = u(1, t) \tag{4.0.4}$$

Let $c = \frac{1}{16\pi^2}$ and use the analytical solution given by the characteristics for the training and testing data:

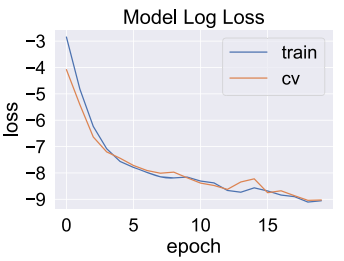
$$u(x, t) = \frac{1}{2}(sin(4\pi x + t) + sin(4\pi x - t))\ . \tag{4.0.5}$$

Here we used $\delta x = 1 \times 10^{-2}$, $\delta t = 1 \times 10^{-2}$, and the mesh grid size is 101. We obtained a MSE 3.5401×10^{-5}. The test dataset batch size is 25 and thus the total discrete testing time period is 250. Figures 4(a) and 4(b) are the heat map for the exact test data and our predicted test data. Figure 4(c) shows both training and cross-validation errors of Neural-PDE convergent within 20 epochs.

We selected the final four states for computation and compared them with analytic solutions. The result indicates that the Neural-PDE is robust in capturing the physical laws of wave equation and predicting the sequence time period. See Figure 5.



(a) Exact Test Dataset (b) Predicted Test Dataset

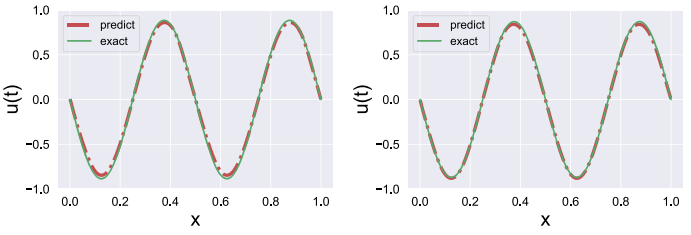


(c) Training Metrics

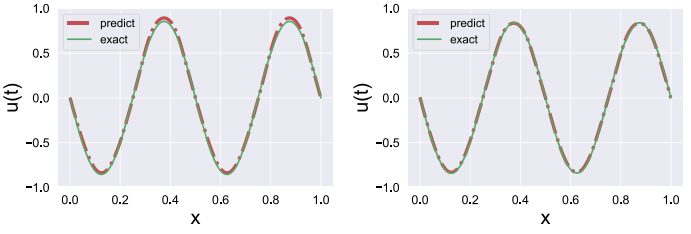
Figure 4: The Neural-PDE for solving the wave equation.

Example: heat equation

The 1D wave equation case maps the data into a matrix (3.3.1) with its original spatial locations. In this test, we solve the 2D heat equation describing how the motion or diffusion of a heat flow evolves over time. Here the 2-dimensional PDE grid in space is mapped into matrix without regularization of the position. The experimental results show that the Neural-PDE



(a) $t = 1.991$ (b) $t = 1.994$



(c) $t = 1.997$ (d) $t = 2$

Figure 5: Comparison between exact solution and Neural-PDE prediction of the 1D wave equation at various time points.

is able to capture the valuable features regardless of the order of the grid points in the matrix. Let's start with a 2D heat equation as follows:

$$u_t = u_{xx} + u_{yy} \ , \tag{4.0.6}$$

$$u(x, y, 0) = \begin{cases} 0.9, & \text{if } (x-1)^2 + (y-1)^2 < 0.25 \\ 0.1, & \text{otherwise} \end{cases} \tag{4.0.7}$$

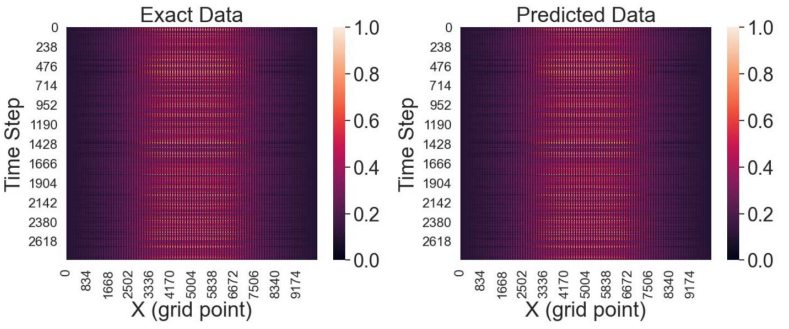
$$\Omega = [0, 2] \times [0, 2], \ t \in [0, 0.15] \ . \tag{4.0.8}$$

Figures 6 and 7 show the test of the Neural-PDE using the 2D heat equation. We obtained a MSE $\mathcal{O}(10^{-6})$.

Example: inviscid Burgers' equation

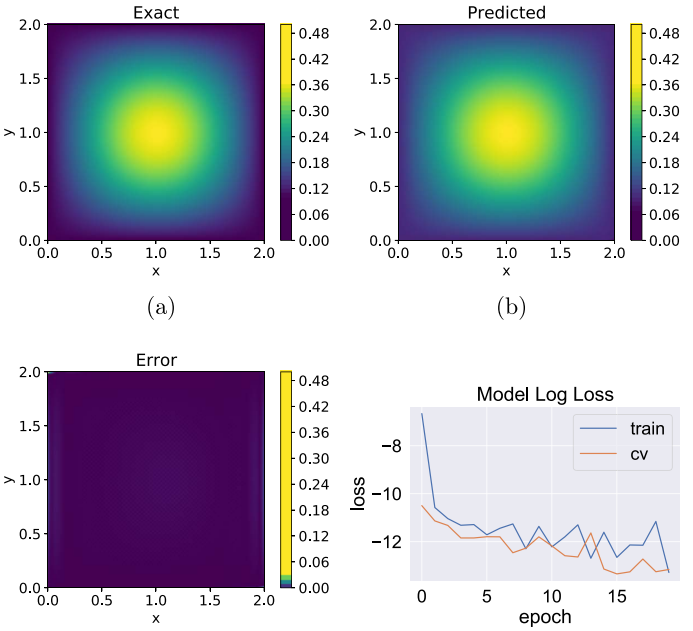
Inviscid Burgers' equation is a classical nonlinear PDE in fluid dynamics. In this example, we consider a $2D$ invicid Burgers' equation which has the following hyperbolic form:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = 0 \ , \quad \frac{\partial v}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = 0 \ , \tag{4.0.9}$$



(a) Exact Test Dataset (b) Predicted Test Dataset

Figure 6: Heatmaps of the heat equation data. $\delta x = 0.02, \delta y = 0.02$, $\delta t = 10^{-4}$, MSE: 2.1551×10^{-6}.



(a) (b)

(c) (d) Training Metrics

Figure 7: The Neural-PDE for solving the 2D heat equation. (a) is the exact solution $u(x, y, t = 0.15)$ at the final state. (b) is the Neural-PDE prediction. (c) is the corresponding error map and (d) shows the training and cross-validation errors.

$$\Omega = [0,1] \times [0,1], t \in [0,1] \ , \tag{4.0.10}$$

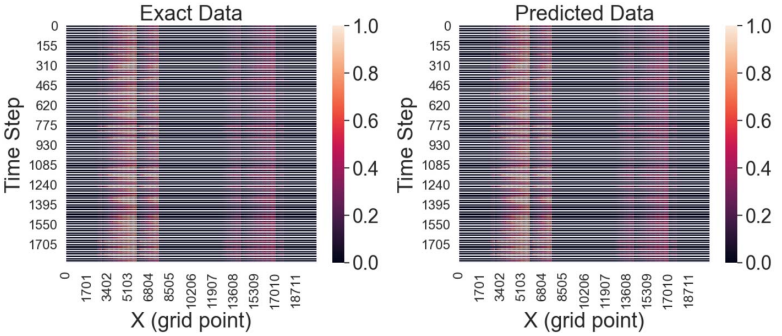
and with the initial and boundary conditions:

$$u(0.25 \leq x \leq 0.75,\ 0.25 \leq y \leq 0.75, t = 0) = 0.9 \ , \tag{4.0.11}$$

$$v(0.25 \leq x \leq 0.75,\ 0.25 \leq y \leq 0.75, t = 0) = 0.5 \ , \tag{4.0.12}$$

$$u(0, y, t) = u(1, y, t) = v(x, 0, t) = v(x, 1, t) = 0 \ . \tag{4.0.13}$$

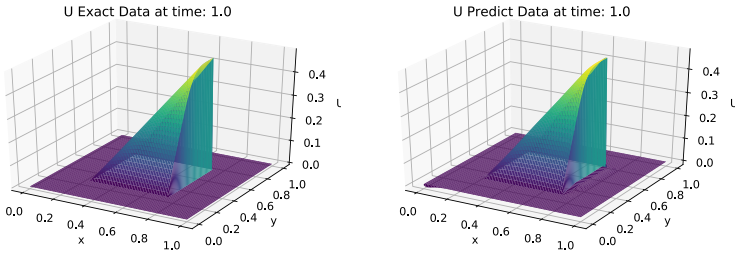
The invicid Burgers' equation is difficult to solve due to the discontinuities (shock waves) in the solutions. We use a upwinding finite difference scheme to create the training data and put the velocity u, v in to the input matrix. Let $\delta x = \delta y = 10^{-2}, \delta t = 10^{-3}$, our empirical results (see Figure 9) show that the Neural-PDE is able to learn the shock waves, boundary conditions and the rules of the equation, and predict u and v simultaneously with an overall MSE of 2.3070×10^{-6}. The heat maps of exact solution and predicted solution are shown in Figure 8.



(a) Exact Test Dataset (b) Predicted Test Dataset

(c) Training Metrics

Figure 8: Neural-PDE prediction on the $2D$ Burgers' equation.



(a) Exact $u(t=1)$ (b) Predicted $u(t=1)$

(c) Exact $v(t=1)$ (d) Predicted $v(t=1)$

Figure 9: Neural-PDE shows accurate prediction on Burgers' equation.

Example: multiscale modeling: Coupled Cahn–Hilliard–Navier–Stokes system

Finally, let's consider the following $2D$ Cahn–Hilliard–Navier–Stokes system widely used for modeling complex fluids:

$$\boldsymbol{u}_t + \boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p + \nu\Delta\boldsymbol{u} - \phi\nabla\mu \ , \tag{4.0.14}$$

$$\phi_t + \nabla\cdot(\boldsymbol{u}\phi) = M\Delta\mu \ , \tag{4.0.15}$$

$$\mu = \lambda(-\Delta\phi + \frac{\phi}{\eta^2}(\phi^2-1)) \ , \tag{4.0.16}$$

$$\nabla\cdot\boldsymbol{u} = 0 \ . \tag{4.0.17}$$

In this example we use the following initial condition:

$$\phi(x,y,0) = (\frac{1}{2} - 50\tanh(f_1 - 0.1)) + (\frac{1}{2} - 50\tanh(f_2 - 0.1)) \ , \text{ where} \tag{4.0.18}$$

$$f_1(x,y) = \sqrt{(x+0.12)^2 + (y)^2}, \ f_2(x,y) = \sqrt{(x-0.12)^2 + (y)^2} \tag{4.0.19}$$

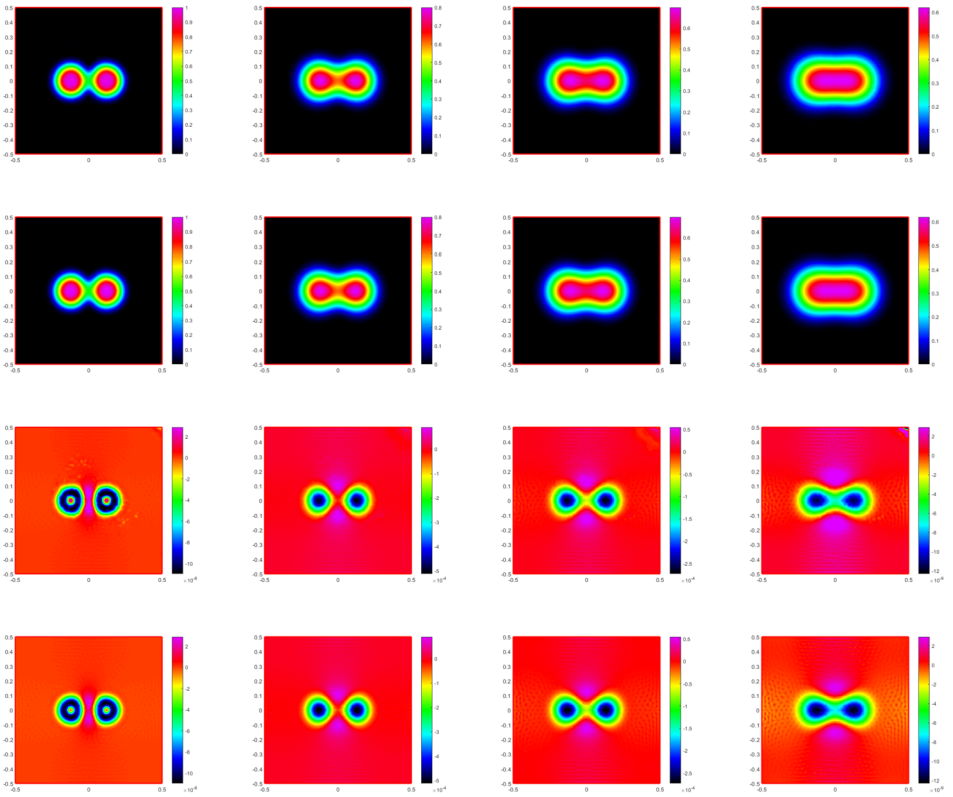
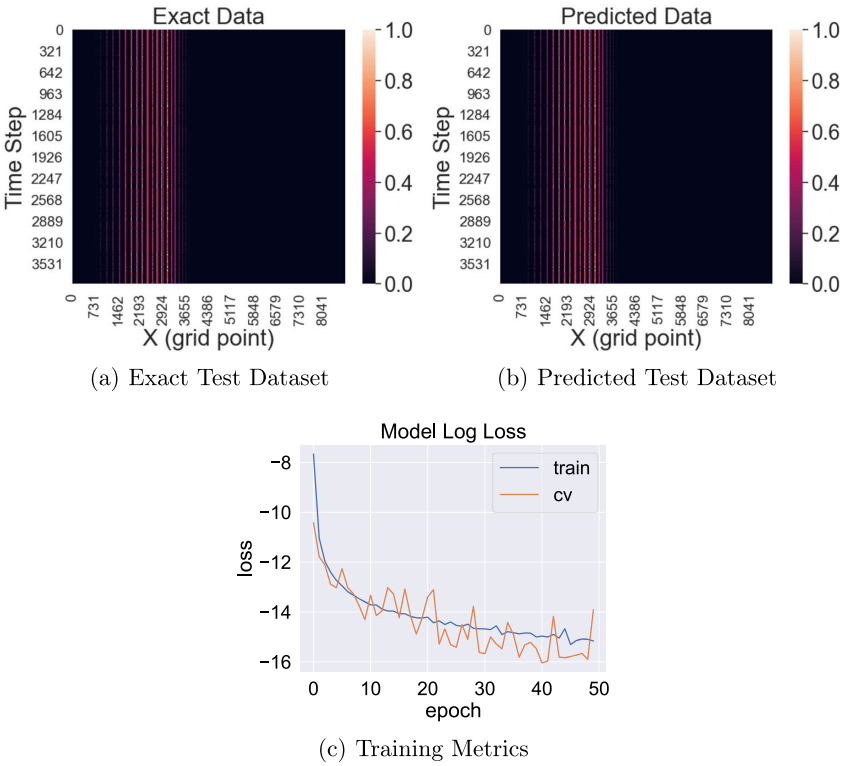


Figure 10: Predicted data by the Neural-PDE (1^{st} row) and the exact data (2^{nd} row) of volume fraction ϕ, predicted pressure p (3^{rd} row) and exact pressure(4^{th} row). The graphs of columns 1-4 represent the time states of t_1, t_2, t_3, t_4, respectively, where $0 \leq t_1 < t_2 < t_3 < t_4 \leq 1$.

(4.0.20)
with $x \in [-0.5, 0.5],\ y \in [-0.5, 0.5],\ t \in [0, 1],\ M = 0.1,\ \nu = 0.01,\ \eta = 0.1.$

This fluid system can be derived by the energetic variational approach [28]. Here $\boldsymbol{u}$ is the velocity and $\phi(x, y, t) \in [0, 1]$ is the labeling function of the fluid phase. M is the diffusion coefficient, and μ is the chemical potential of ϕ. Equation (4.0.17) indicates the incompressibility of the fluid. Solving such PDE system is notorious because of its high nonlinearity and multi-physical and coupled features. A challenge of deep learning in solving a system like this is how to process the data to improve the learning efficiency when the input matrix consists of variables such as $\phi \in [0, 1]$ with large



(a) Exact Test Dataset

(b) Predicted Test Dataset

(c) Training Metrics

Figure 11: Neural-PDE shows ideal prediction on Fluid System.

magnitude value and variable of very small values such as $p \sim 10^{-5}$. For the Neural-PDE to better extract and learn the physical features of variables in different spatial-temporal scales, we normalized the p data with a *sigmoid* function. We set $\delta t = 5 \times 10^{-4}$. Here the training dataset is generated by the FEM solver FreeFem++ [29] using a Crank-Nicolson in time C^0 finite element scheme. Our Neural-PDE prediction shows that the physical features of p and ϕ have been successfully captured with an overall MSE: 6.1631×10^{-7} (see Figure 10). In this example, we only coupled p and ϕ together to show the learning ability of the Neural-PDE. Another approach is to couple p, ϕ and the velocity $\boldsymbol{u}$ together in the training data to predict all the related variables $(p, \phi, \boldsymbol{u})$, which would need more normalization and regularization, techniques such as batch normalization would be helpful, please see recent research on PINN based neural network in solving such system [30].

5. Conclusions

In this paper, we proposed a novel sequence recurrent deep learning framework: Neural-PDE, which is capable of intelligently filtering and learning solutions of time-dependent PDEs. One key innovation of our method is that the time marching method from the numerical PDEs is applied in the deep learning framework, and the neural network is trained to explore the accurate numerical solutions for prediction.

Our experiments show that the Neural-PDE is capable of simulating from 1D to multi-dimensional scalar PDEs to highly nonlinear and coupled PDE systems with their initial conditions, boundary conditions without knowing the specific forms of the equations. Solutions to the PDEs can be either continuous or discontinuous.

The state-of-the-art researches have shown the promising power of deep learning in solving high-dimensional nonlinear problems in engineering, biology and finance with efficiency in computation and accuracy in prediction. However, there are still unresolved issues in applying deep learning in PDEs. For instance, the stability and convergence of the traditional numerical algorithms have been rigorously studied by applied mathematicians. Due to the high nonlinearity of the neural network system, theorems guiding stability and convergence of solutions predicted by the neural network are yet to be revealed.

Lastly, it would be helpful and interesting if one can theoretically characterize a numerical scheme from the neural network coefficients and learn the forms or mechanics from the scheme and prediction. We leave these questions for future study. The code and data for this paper will become available at https://github.com/YihaoHu/Neural_PDE upon publication.

References

[1] R. Courant, K. Friedrichs, H. Lewy, On the partial difference equations of mathematical physics, IBM journal of Research and Development 11 (2) (1967) 215–234. MR0213764

[2] S. Osher, J. A. Sethian, Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations, Journal of computational physics 79 (1) (1988) 12–49. MR0965860

[3] R. J. LeVeque, Numerical methods for conservation laws, Vol. 3, Springer, 1992. MR1153252

[4] B. Cockburn, G. E. Karniadakis, C.-W. Shu, Discontinuous Galerkin methods: theory, computation and applications, Vol. 11, Springer Science & Business Media, 2012. MR1842160

[5] J. W. Thomas, Numerical partial differential equations: finite difference methods, Vol. 22, Springer Science & Business Media, 2013. MR1367964

[6] C. Johnson, Numerical solution of partial differential equations by the finite element method, Courier Corporation, 2012. MR0911477

[7] G. C. Peng, M. Alber, A. B. Tepole, W. R. Cannon, S. De, S. Dura-Bernal, K. Garikipati, G. Karniadakis, W. W. Lytton, P. Perdikaris, et al., Multiscale modeling meets machine learning: What can we learn?, Archives of Computational Methods in Engineering (2020) 1–21. MR4246233

[8] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, C. I. Sánchez, A survey on deep learning in medical image analysis, Medical image analysis 42 (2017) 60–88.

[9] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).

[10] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.

[11] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.

[12] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, IEEE Signal processing magazine 29 (6) (2012) 82–97.

[13] J. Han, A. Jentzen, E. Weinan, Solving high-dimensional partial differential equations using deep learning, Proceedings of the National Academy of Sciences 115 (2018) 8505–8510. MR3847747

[14] Z. Long, Y. Lu, X. Ma, B. Dong, Pde-net: Learning pdes from data, in: International Conference on Machine Learning, 2018, pp. 3208–3216.

[15] J. Sirignano, K. Spiliopoulos, Dgm: A deep learning algorithm for solving partial differential equations, Journal of computational physics 375 (2018) 1339–1364. MR3874585

[16] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, Journal of Computational Physics 378 (2019) 686–707. MR3881695

[17] A. Sherstinsky, Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network, Physica D: Nonlinear Phenomena 404 (2020) 132306. MR4057560

[18] U. M. Ascher, S. J. Ruuth, R. J. Spiteri, Implicit-explicit runge-kutta methods for time-dependent partial differential equations, Applied Numerical Mathematics 25 (2-3) (1997) 151–167. MR1485812

[19] R. T. Chen, Y. Rubanova, J. Bettencourt, D. K. Duvenaud, Neural ordinary differential equations, in: Advances in neural information processing systems, 2018, pp. 6571–6583.

[20] Z. Huang, W. Xu, K. Yu, Bidirectional lstm-crf models for sequence tagging, arXiv preprint arXiv:1508.01991 (2015).

[21] M. Schuster, K. K. Paliwal, Bidirectional recurrent neural networks, IEEE transactions on Signal Processing 45 (11) (1997) 2673–2681.

[22] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (8) (1997) 1735–1780.

[23] Z. C. Lipton, J. Berkowitz, C. Elkan, A critical review of recurrent neural networks for sequence learning, arXiv preprint arXiv:1506.00019 (2015).

[24] A. Graves, J. Schmidhuber, Framewise phoneme classification with bidirectional lstm and other neural network architectures, Neural networks 18 (5-6) (2005) 602–610.

[25] T. Mikolov, S. Kombrink, L. Burget, J. Černockỳ, S. Khudanpur, Extensions of recurrent neural network language model, in: 2011 IEEE international conference on acoustics, speech and signal processing (ICASSP), IEEE, 2011, pp. 5528–5531.

[26] Y. Lu, A. Zhong, Q. Li, B. Dong, Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations, in: International Conference on Machine Learning, 2018, pp. 3276–3285.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in neural information processing systems, 2017, pp. 5998–6008.

[28] J. Forster, Mathematical modeling of complex fluids, Master's, University of Wurzburg (2013).

[29] F. Hecht, New development in freefem++, J. Numer. Math. 20 (3-4) (2012) 251–265. URL https://freefem.org/ MR3043640

[30] C. L. Wight, J. Zhao, Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks, arXiv preprint arXiv:2007.04542 (2020). MR4203116

Yihao Hu
Department of Applied and Computational Mathematics and Statistics
University of Notre Dame
Notre Dame, IN 46545, USA
E-mail address: yhu5@nd.edu

Tong Zhao
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46545, USA
E-mail address: tzhao2@nd.edu

Shixin Xu
Duke Kunshan University
Kunshan, Jiangsu 215316, P.R. China
E-mail address: shixin.xu@dukekunshan.edu.cn

Lizhen Lin
Department of Applied and Computational Mathematics and Statistics
University of Notre Dame
Notre Dame, IN 46545, USA
E-mail address: lizhen.lin@nd.edu

Zhiliang Xu
Department of Applied and Computational Mathematics and Statistics
University of Notre Dame
Notre Dame, IN 46545, USA
E-mail address: zxu2@nd.edu

Received October 26, 2021