Simulating Partial Differential Equations with Neural Networks

Alina Chertock and Christopher Leonard

Abstract In this paper, we present a novel approach for simulating solutions of partial differential equations using neural networks. We consider a time-stepping method similar to the finite-volume method, where the flux terms are computed using neural networks. To train the neural network, we collect 'sensor' data on small subsets of the computational domain. Thus, our neural network learns the local behavior of the solution rather than the global one. This leads to a much more versatile method that can simulate the solution to equations whose initial conditions are not in the same form as the initial conditions we train with. Also, using sensor data from a small portion of the domain is much more realistic than methods where a neural network is trained using data over a large domain.

Keywords Neural networks · Partial differential equations · Finite-volume methods

1 Introduction

Consider the time-dependent partial differential equation (PDE)

$$\boldsymbol{U}_t + \nabla_{\boldsymbol{x}} \cdot \boldsymbol{F}(\boldsymbol{U}) = \boldsymbol{\varepsilon} \Delta \boldsymbol{U}, \tag{1}$$

where $\boldsymbol{U} = (u_1(\boldsymbol{x}, t), \ldots, u_{N_e}(\boldsymbol{x}, t))^\top$ is a vector function of the spatial variable $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ and the time variable $t \geq 0$, $\boldsymbol{F} = (F_1, \ldots, F_d)^\top$ is the nonlinear convection flux, and $\boldsymbol{\varepsilon} = \text{diag}(\varepsilon_1, \ldots, \varepsilon_{N_e})$ is a constant diagonal matrix with positive entries. Among others, Eq. (1) is used to describe hyperbolic systems of conservation laws ($\boldsymbol{\varepsilon} \equiv 0$) and systems of convection-diffusion equations. Models spanned by (1) are widely used to describe a variety of phenomena in physical, astrophysical, geophysical, meteorological, biological, chemical, financial, social, and other scientific areas.

A. Chertock (✉) · C. Leonard
North Carolina State University, Raleigh, NC, USA
e-mail: chertock@math.ncsu.edu

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

C. Parés et al. (eds.), *Hyperbolic Problems: Theory, Numerics, Applications. Volume II*,
SEMA SIMAI Springer Series 35, https://doi.org/10.1007/978-3-031-55264-9_4

In recent years, machine learning techniques for solving PDEs and learning about their solutions have been a growing field of interest. This is in part due to the success machine learning has had in other fields, such as computer vision and speech recognition. Along with these successes, another motivation to use neural networks (NNs) is due to the universal approximation theorem, which proves that, under certain conditions, an artificial NN could approximate any continuous function, [3]. There are many different approaches to finding numerical solutions to PDEs with the help of NNs. One of the most popular methods is known as a physics-informed neural network (PINN), which can be trained to satisfy the differential equation and the initial and boundary conditions; see, e.g., [5, 9]. While PINNs have been successfully applied to various problems, one of its major drawbacks is that they need to be retrained for any new initial or boundary conditions. Other approaches use NNs alongside standard numerical methods and utilize the benefits of physics-informed and deep learning-based techniques.

In this paper, we train NNs to simulate the solution to PDEs when the underlying dynamics are unknown. The method of solving systems of ordinary differential equations (ODEs) with NNs without information about the governing equations has been introduced in [8]. Then, in [1], a similar method was implemented to simulate PDEs. Whereas their method uses the entire discrete solution at a given time step as input into a NN, our approach only considers local data for the NN input. Thus we can train the NN with data collected on small areas in space and use the NN to simulate solutions on arbitrarily big domains. This is a much more realistic data collection setting, allowing one to simulate a wide range of initial value problems.

The outline of this paper is as follows. In Sect. 2, we briefly introduce NNs. Then, in Sect. 3, we describe our NN method for solving systems of PDEs in the form of (1). In Sect. 4, we illustrate the proposed approach's performance on several numerical examples. Lastly, in Sect. 5, we conclude the paper.

2 Neural Networks

An artificial NN is an operator $\boldsymbol{N}_\Theta(\cdot)$ with a parameter set $\Theta = (\Theta_1, \ldots, \Theta_{H+1})$, where Θ_η is itself a parameter set for each layer $\eta = 1, 2, \ldots, H+1$ (see Eq. (3)). The set Θ can have hundreds, thousands, or even millions of parameters, which allows $\boldsymbol{N}_\Theta$ to represent a wide range of functions depending on which parameters are chosen. Typically, the network is provided with a set of input-output pairs $\{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^M$, to "learn" which parameters to use by a training process. This training process tries to find the parameter set Θ such that $\boldsymbol{U}_m^{\text{out}} \approx \boldsymbol{N}_\Theta(\boldsymbol{U}_m^{\text{in}}),\ m = 1, \ldots, M$ by minimize a given loss function

$$L_\Theta = L(N_\Theta, \{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^M) \tag{2}$$

that depends on the NN and the supplied set of data.

Training a NN from a set of input-output data pairs is known as supervised learning. Other training processes include unsupervised learning and reinforcement learn-

ing. Unsupervised learning is a training algorithm that finds underlying patterns in the data, such as clusters, and reinforcement learning is an algorithm that uses a reward system to learn the best parameters for the machine learning model. While all three learning processes have found success in different applications, supervised learning has been the most successful approach to training NNs and is used for all of the models in this paper.

Artificial NNs are built from compositions of smaller functions (see Eq. (3)). Each of these smaller functions corresponds to what is known as a layer of the NN. Feedforward NNs are a specific form of a NN where the information is passed in one direction without looping back. This is opposed to, say, recurrent NNs that contain cycles in their NN architecture. All examples in this paper will use feedforward NNs, although many could be extended to other types of architectures, such as recurrent NNs.

A feedforward NN with H hidden layers can be represented as the composition of parameterized functions

$$\boldsymbol{N}_\Theta = \boldsymbol{N}_{\Theta_{H+1}} \circ \boldsymbol{N}_{\Theta_H} \circ \ldots \circ \boldsymbol{N}_{\Theta_1} \quad \text{with} \quad \boldsymbol{N}_{\Theta_\eta}(z) = \sigma_\eta(A_\eta),\ \eta = 1, 2, \ldots, H+1, \tag{3}$$

where A_η is the connection between layers $\eta - 1$ and η, and σ_η is the activation function for the layer η. Here, $\eta = 0$ corresponds to the input layer and $\eta = H + 1$ corresponds to the output layer. A layer is said to be fully connected if $A_\eta(\boldsymbol{z}) = W_\eta \boldsymbol{z} + \boldsymbol{b}_\eta$, where each element of $W_\nu \in \mathbb{R}^{\omega_\nu \times \omega_{\nu-1}}$ is nonzero, $\boldsymbol{b}_\eta \in \mathbb{R}^{\omega_\eta}$. Here, ω_η is the number of nodes for layer η, W_η is known as the weight matrix, and $\boldsymbol{b}_\eta$ is the bias vector. Another type of layer is a convolutional layer, which in the simplest case can be defined as $A_\eta(\boldsymbol{z}) = \boldsymbol{c}_\eta \star \boldsymbol{z} + \boldsymbol{b}_\eta$, where $\boldsymbol{c}_\eta \in \mathbb{R}^{\kappa_\eta}$, $\boldsymbol{b}_\eta \in \mathbb{R}$, and $\star$ is the cross-correlation operator. The vector $\boldsymbol{c}_\eta$ is known as a convolution kernel with a kernel size of κ_η. While the value of $\boldsymbol{b}_\eta$ is often a scalar value for convolution layers (this is the default setting in PyTorch [7], and what is used in this paper), it can also be set such that $\boldsymbol{b} \in \mathbb{R}^{\omega_\eta}$. The parameters in the sets Θ_η are the element values for W_η, $\boldsymbol{b}_\eta$, and $\boldsymbol{c}_\eta$. The activation functions σ_η, $\eta = 1, \ldots, H + 1$, are functions prescribed before the training process. Usually, these functions are nonlinear so the NN can learn nonlinear relationships from the data.

Training the Neural Network

Before training a NN, the following hyperparameters need to be defined:

(a) Loss function: L_Θ (see (2)).
(b) Optimizer: The algorithm used to find the global minimum of the loss function.
(c) Initial learning rate: The initial step size that the optimization algorithm takes. Depending on the optimization algorithm, the step sizes may change between steps.
(d) Number of epochs: The number of times the optimization algorithm goes through the entire training data set. Denote the number of epochs as M_e.
(e) Batch size: The number of samples from the training data that propagates through the network for each update of the parameters. Denote the batch size as B_s.

(f) Number of hidden layers: H.
(g) Linear function for connection between layers A_η (W_η or $\boldsymbol{c}_\eta$), and their sizes, i.e. we need to define ω_η and $\omega_{\eta-1}$ for fully connected layers and κ_η for convolutional layers.
(h) Activation functions: $\sigma_\eta,\ \eta = 1, \ldots, H+1$.

We will identify the specific hyperparameters used for each example in their respective sections. We train $\boldsymbol{N}_\Theta$ using the following algorithm.

Algorithm

Input: Data set $\{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^{M}$, initial NN model $\boldsymbol{N}_{\Theta^{(0)}}$, and hyperparameters (a)-(h) from above.
Output: Trained $\boldsymbol{N}_\Theta$.

1. *Start*: Randomly split the data set $\{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^{M}$ into three disjoint sets, the training set of size M_{tr}, $\{\boldsymbol{U}_{m_i}^{\text{in}}, \boldsymbol{U}_{m_i}^{\text{out}}\}_{i=1}^{M_{tr}}$, the validation set of size M_{val}, $\{\boldsymbol{U}_{m_l}^{\text{in}}, \boldsymbol{U}_{m_l}^{\text{out}}\}_{l=1}^{M_{val}}$, and the test set of size M_{test}, $\{\boldsymbol{U}_{m_r}^{\text{in}}, \boldsymbol{U}_{m_r}^{\text{out}}\}_{r=1}^{M_{val}}$, where $M = M_{tr} + M_{val} + M_{test}$.
   a. The optimization algorithm uses the training set to find the parameter set Θ that minimizes the loss function L_Θ.
   b. The validation set checks that the NN can generalize to new data.
   c. The test set is used to test the NN on data that does not influence the training of the NN at all.
2. *Iterate*: For $e = 1, 2, \ldots, M_e$
   a. Let $M_{tr} = (\beta - 1)B_s + r_b$, where $\beta, r_b \in \mathbb{N}$ and $r_b \leq B_s$ and randomly split the training set into β separate batches each of size B_s except the last, which is of size r_b.
   (b) *Iterate*: For each batch = 1,2,...,β
      - Update the NN parameters using one step of the optimization algorithm.
   (c) With the current parameter set $\Theta^{(e)}$, calculate the loss value $L_{\Theta^{(e)}}$ using the new model $\boldsymbol{N}_{\Theta^{(e)}}$ and the validation set $\{\boldsymbol{U}_{m_l}^{\text{in}}, \boldsymbol{U}_{m_l}^{\text{out}}\}_{l=1}^{M_{val}}$ as the input into the model (see (2)).
   (d) If $L_{\Theta^{(e)}} < L_{\Theta^{(i)}}$ for all $i = 1, \ldots, e-1$, set $\Theta = \Theta^{(e)}$, i.e. if this loss value is less than the loss value at the end of every other epoch before it, update Θ.

Remark 1 Finding good hyperparameters is often done by running multiple trials of the training algorithm using different hyperparameters and identifying which results in the best outcome.

Remark 2 We will use the PyTorch [7] machine learning framework for all the NN models in this paper. The initial parameters in the set $\Theta^{(0)}$ are the random parameters set by PyTorch.

3 Simulating PDEs with the Help of Neural Networks

In this section, we assume that Eq. (1) is discretized using a finite-volume (FV) approach and demonstrate how its numerical solution can be evolved in time using NNs. For the rest of this section, we consider (1) in one spatial dimension ($D = 1$), but note that the method is very similar for other dimensions.

3.1 The Finite-Volume Method

Consider the one-dimensional (1-D) version of the system (1) rewritten as

$$\boldsymbol{U}_t + \boldsymbol{H}(\boldsymbol{U})_x = \boldsymbol{0}, \quad \boldsymbol{H}(\boldsymbol{U}) = \boldsymbol{F}(\boldsymbol{U}) - \boldsymbol{\varepsilon}\boldsymbol{U}_x, \tag{4}$$

Assume that the computational domain is divided into uniform cells $C_j = (x_{j-\frac{1}{2}}, x_{j+\frac{1}{2}})$ of size Δx centered at x_j with $x_{j+\frac{1}{2}} - x_{j-\frac{1}{2}} \equiv \Delta x$. Then, the computed discrete quantities are the cell averages, $\overline{\boldsymbol{U}}_j(t) \approx \frac{1}{\Delta x}\int_{C_j} \boldsymbol{U}(x,t)\,\mathrm{d}x$, that are evolved in time by solving the following system of ODEs:

$$\frac{\mathrm{d}}{\mathrm{d}t}\overline{\boldsymbol{U}}_j(t) = -\frac{\mathcal{F}_{j+\frac{1}{2}}(t) - \mathcal{F}_{j-\frac{1}{2}}(t)}{\Delta x}, \tag{5}$$

where $\mathcal{F}_{j+\frac{1}{2}}$ are numerical fluxes across cell interfaces $x_{j+\frac{1}{2}}$. The numerical fluxes $\mathcal{F}_{j+\frac{1}{2}}$ in (5) are typically computed using point values of the computed solution at the cell interfaces, that is, $\mathcal{F}_{j+\frac{1}{2}} = \mathcal{F}(\boldsymbol{U}^-_{j+\frac{1}{2}}, \boldsymbol{U}^+_{j+\frac{1}{2}})$, where $\boldsymbol{U}^{\pm}_{j+\frac{1}{2}} \approx \boldsymbol{U}(x_{j+\frac{1}{2}} \pm 0, t)$, and the accuracy of the the method depends on the accuracy with which these point values are reconstructed. Finally, the semi-discrete FV scheme in (5) is a system of ODEs, which is to be integrated numerically by an accurate ODE solver with a suitable time step Δt, chosen to satisfy a proper CFL stability condition.

3.2 Time-Stepping Neural Network

In this section, a time stepping NN, $\boldsymbol{F}_\Theta : \mathbb{R}^{N_f \times N_e} \to \mathbb{R}^{N_e}$ to solve (5) numerically is introduced. The NN is trained such that for stencils of size N_f in the spatial domain, $\boldsymbol{F}_\Theta(\overline{\boldsymbol{U}}^n_{j+N_f/2}, ..., \overline{\boldsymbol{U}}^n_{j-N_f/2+1}) \approx \mathcal{F}^n_{j+\frac{1}{2}}$, where $\overline{\boldsymbol{U}}^n_{j+N_f/2}$ is the numer-

ical solution at time $t = t^n$ and Θ is the NN's internal parameter set. Denote $\widehat{\boldsymbol{F}}^n_{j+\frac{1}{2}} := \boldsymbol{F}_{\Theta_x}(\overline{\boldsymbol{U}}^n_{j+N_f/2}, ..., \overline{\boldsymbol{U}}^n_{j-N_f/2+1})$ and set

$$\widehat{\boldsymbol{U}}^{n+1}_j(\Theta) = \bar{\boldsymbol{U}}^n_j - \Delta t\, \frac{\widehat{\boldsymbol{F}}^n_{j+\frac{1}{2}} - \widehat{\boldsymbol{F}}^n_{j-\frac{1}{2}}}{\Delta x}.$$

The NNs are trained to find the parameter sets Θ such that $\widehat{\boldsymbol{U}}^{n+1}_j \approx \bar{\boldsymbol{U}}^{n+1}_j$.

To prepare the neural network for training, it's essential to gather the data it will learn from. To do so, assume the initial condition to (1) is some parameterized function, $\boldsymbol{U}_0(\boldsymbol{x}; \hat{\boldsymbol{p}})$ with parameters $\hat{\boldsymbol{p}} = [\hat{p}_1, \ldots, \hat{p}_{N_p}]$. Randomly draw N_s parameter vectors $\hat{\boldsymbol{p}}^i, i = 1 \ldots, N_s$, from a uniform distribution, with each parameter $\hat{p}^i_\ell \sim \mathcal{U}[\alpha_\ell, \beta_\ell)$, $i = 1, \ldots, N_s$ and $\ell = 1, \ldots, N_p$. With the N_s initial conditions, simulate the solution of (1) for N_t time steps, using highly accurate solvers to collect data at discrete points in space and time. To collect data, assume there are 'sensors' in the spatial domain, which collect stencil data located around the discrete points for every time step of the simulation. Then, collect the data $\overline{\boldsymbol{U}}^n_{j_s+\alpha},\ \alpha = -N_f/2, \ldots, N_f/2,\ s = 1, \ldots, S,\ n = 0, \ldots, N_t$ at S sensor locations $\{x_{j_s}\}_{s=1}^S$.

Once the data is collected, train the NNs to minimize a loss function of the form

$$L_\Theta = \sum_{m=1}^{M} \|\widehat{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m} - \overline{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m}\|_1 + \mu \sum_{m=1}^{M} \|\widehat{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m}\|_1, \tag{6}$$

where $\|\cdot\|_1$ is the l_1 norm, and the term on the right is a regularization term with constant μ used to help stabilize the numerical method. Here, $\widehat{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m}$ and $\overline{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m}$ correspond to the NN solution and reference solution, respectively, from the mth data in our data set, where $n_m \in \{0, \ldots, N_t - 1\}$, $j_m \in \{1, \ldots, N_x\}$, and $i_m \in \{1, \ldots, N_s\}$.

4 Numerical Examples

In this section, we illustrate the performance of the method described above when applied to a 1-D scalar nonlinear equation, as well as a two-dimensional (2-D) nonlinear system of equations.

To train the NNs, a feedforward NN with four hidden layers is used, alternating between fully connected and convolutional layers. For each NN, the convolutional layers have a kernel of length 5. We use $M = 200{,}000$ data points for the NNs, splitting it into training, testing, and validation sets with sizes of $M_{train} = 120{,}000$, $M_{test} = 40{,}000$, and $M_{val} = 40{,}000$ for each of the respective sets. During the training procedure, a batch size of 32 is used and the NNs are trained for 1000 epochs

using the Adam optimizer [7] with an initial learning rate of 10^{-4}. The loss function for our training algorithm is the one given in (6) with $\mu = 0.1$. The only changes between the two examples are the number of nodes for each fully connected layer and the number of inputs and outputs for the NN. We will state these values in the section of their respective examples.

To collect accurate data to train the NN with, we compute the numerical solution using the second-order semi-discrete central-upwind (CU) scheme from [4] on a fine grid with $\tilde{N}_x = m_x N_x$, where $m_x > 1$ and then map the fine grid solution to the coarser grid the NN is trained on.

Once the NN is trained, to simulate the solution from time t_n to time t_{n+1}, we use all the appropriate stencils from the solution

$$\widehat{\boldsymbol{U}}^n = [\widehat{\boldsymbol{U}}_1^n, \ldots, \widehat{\boldsymbol{U}}_{N_x}^n] \in \mathbb{R}^{N_x},$$

as one input batch into the NN, F_Θ. Thus we benefit from the machine learning libraries' parallelization capabilities, instead of looping through the data points manually to calculate every flux value.

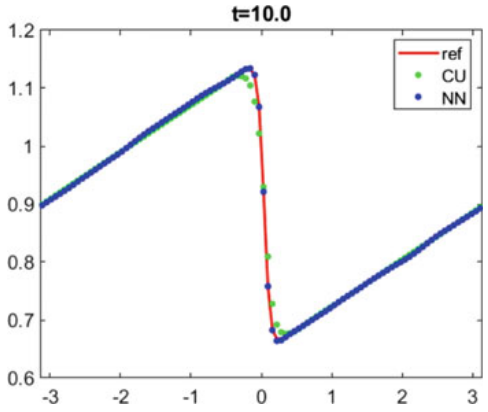
4.1 Burgers Equation

In this example, we consider the 1-D viscous Burgers equation with $\boldsymbol{U} = u$ and $\boldsymbol{F}(\boldsymbol{U}) = \frac{1}{2}u^2$ in (4), subject to the initial conditions

$$u(x, 0) = a_0 + a_1 \cos(x) + b_1 \sin(x), \tag{7}$$

where $a_0, a_1, b_1 \in [-1, 1)$, and $\varepsilon = 0.01$. To collect training data for our NN, the computational domain $x \in [-\pi, \pi]$ is discretized into $N_x = 100$ cells of uniform size $\Delta x = \frac{2\pi}{N_x}$. We then simulate $N_s = 1000$ solutions where the parameters $a_0^i, a_1^i, b_1^i \sim \mathcal{U}[-1, 1)$, $i = 1, \ldots, N_s$, are used to define the initial conditions. To simulate the data, we use a fine grid with $\tilde{N}_x = 1000$ using the second-order semidiscrete CU scheme [4] for spatial discretization and the IMEX-SSP2(3,3,2) method [6] for time integration of the ODE system (5). We ran the simulation for $N_t = 500$ time steps collecting the data at every $t_n = n\Delta t$, with $\Delta t = 0.02$ (note that this time step satisfied the CFL condition on the known solution for all steps. To train the NN, we collect the data $u_{j,in}^{n,\hat{\boldsymbol{p}}^i} = [\overline{u}_{j-2}^{n,\hat{\boldsymbol{p}}^i}, \overline{u}_{j-1}^{n,\hat{\boldsymbol{p}}^i}, \overline{u}_j^{n,\hat{\boldsymbol{p}}^i}, \overline{u}_{j+1}^{n,\hat{\boldsymbol{p}}^i}, \overline{u}_{j+2}^{n,\hat{\boldsymbol{p}}^i}]$, $u_{j,out}^{n+1,\hat{\boldsymbol{p}}^i} = \bar{u}_j^{n+1,\hat{\boldsymbol{p}}^i}$, for $j = 10, 20, \ldots, 90$ and $n = 0, \ldots N_t - 1$, where $\bar{u}_j^{n,\hat{\boldsymbol{p}}_i}$ is the cell average approximation at (x_j, t_n) of the ith simulation. Note that $\{x_{10}, x_{20}, \ldots, x_{90}\}$ are our sensor locations.

We use 80 nodes for each fully connected layer for this NN. The number of inputs into the NN is 4 data points around the cell interfaces and only one output, the NN flux around that cell interface.



t	NN	CU Scheme
2	0.0013	0.0018
4	0.0017	0.0045
6	0.0015	0.0044
8	0.0017	0.0039
10	0.0018	0.0034

Fig. 1 Burgers equation: Solution computed with initial condition $u_0(x) = 0.9 - 0.2\cos(x) + 0.4\sin(x)$ (left) and relative l_1 errors for both NN and CU scheme (right).

In Fig. 1, we plot the solution computed at time $t = 10$ with the initial data corresponding to $(a_0, a_1, b_1) = (0.9, -0.2, 0.4)$ in (7) by both the NN approach and CU scheme, along with the reference solution computed by the latter on a finer grid. As one can see, the NN solution lines up very well with the reference solution and can even produce a more accurate resolution near the sharp slope than the CU scheme. The run time for the NN solution is 0.392 seconds with a constant time step $\Delta t = 0.02$. The run time for the CU solution is 0.602 when implemented with a variable time step determined by the CFL condition and 0.653 when using the same fixed time step as in the NN simulations. In Fig. 1, we also show the relative l_1 errors for the CU scheme and the NN method when the numerical solutions computed by both approaches are compared with the reference solution.

As we can see, the NN can produce an accurate result when the initial condition is the same as the initial condition in the training data. Below we illustrate that our method can be generalized to initial conditions that look different from those in the training data. We consider two different scenarios. In both scenarios, we use the same NN used to produce the example in Fig. 1; thus, the NN is trained with the initial data of the form (7). The first initial condition we consider is a periodic Gaussian function with a period of 2π, on the computational domain $[-\pi, \pi]$. Thus, it is smooth, just like the initial conditions of the training data. Another example is a piece-wise constant function, where the solution maintains a sharp slope.

From Fig. 2, we can see that our method can generalize to new initial conditions. This is because the NN does not look at the values of the equation on the whole computational domain but instead uses only local values for its input. Thus, our method should work if the local values come from the same distribution as the training data. As a result, our method can simulate the solution for a wider range of initial conditions than the initial conditions it is trained with.

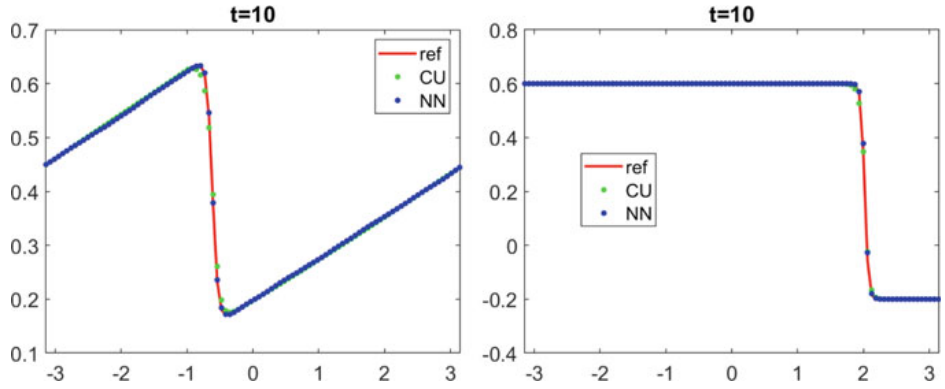


Fig. 2 Burgers equation: Solutions computed at time $t = 10$ with initial condition (i) $u_0(x) = 0.9e^{-0.4x^2}$ (left) and (ii) $u_0(x) = 0.6$ for $x \leq 0$ and $u_0(x) = -0.2$ for $x > 0$ (right).

4.2 2-D Navier-Stokes Equations

In this final example, we consider the 2-D isentropic Navier-Stokes (N-S) equation

$$\begin{cases} \rho_t + (\rho u)_x + (\rho v)_y = 0, \\ (\rho u)_t + (\rho u^2 + \rho^\gamma)_x + (\rho uv)_y = \varepsilon u_{xx}, \qquad x, y \in \mathbb{R},\ t > 0, \\ (\rho v)_t + (\rho uv)_x + (\rho v^2 + \rho^\gamma)_y = \varepsilon v_{yy}, \end{cases}$$

subject to the initial condition

$$\rho(x, y, 0) = \begin{cases} \rho_1, & x < 0.5 \text{ and } y < 0.5, \\ \rho_2, & x < 0.5 \text{ and } y \geq 0.5, \\ \rho_3, & x \geq 0.5 \text{ and } y < 0.5, \\ \rho_4, & x \geq 0.5 \text{ and } y \geq 0.5 \end{cases}, \quad u(x, y, 0) = v(x, y, 0) = 0,$$

where $\gamma = 1.4$, $\varepsilon = 0.005$, and $\rho_1, \rho_2, \rho_3, \rho_4 \in [.5, 1.5)$. Here, ρ, u, and v are the density, x- and y-components of a fluid's velocity, respectively, and we denote $q_1 = \rho u$, $q_2 = \rho v$, and $\boldsymbol{U} = [\rho, q_1, q_2]$. To simulate the solution, the numerical domain $[0, 1] \times [0, 1]$ is discretized into $N_x = N_y = 80$ cells, and $N_s = 1000$ simulations are run for $N_t = 150$ time steps. We collect data after every time step of size $\Delta t = 0.001$, and the parameters for the initial conditions are drawn from the uniform distribution $\rho_1^i, \rho_2^i, \rho_3^i, \rho_4^i \sim \mathcal{U}[0.5, 1.5)$, for all $i = 1 \ldots, N_s$. The data collected is

$$\begin{aligned} \boldsymbol{U}^{n,\hat{\boldsymbol{p}}^i}_{j,k,in_x} &= [\bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j-2,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j-1,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j+1,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j+2,k}] \\ \boldsymbol{U}^{n,\hat{\boldsymbol{p}}^i}_{j,k,in_y} &= [\bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k-2}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k-1}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{k,j+1}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{k,j+2}] \end{aligned}, \qquad \boldsymbol{U}^{n+1,\hat{\boldsymbol{p}}^i}_{j,k,out} = \bar{\boldsymbol{U}}^{n+1,\hat{\boldsymbol{p}}^i}_{j,k},$$

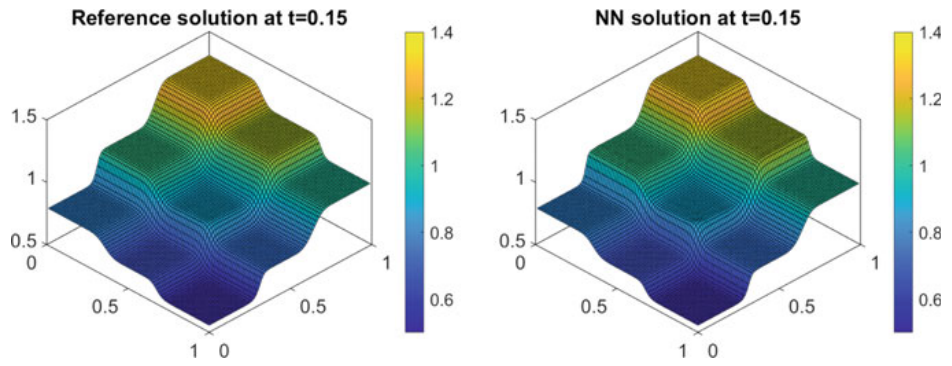


Fig. 3 Isentropic N-S equations: Density ρ at $t = 0.15$ for the reference solution (left) and NN solution (right) with $\rho_1 = 0.79$, $\rho_2 = 1.32$, $\rho_3 = 0.55$, and $\rho_4 = 0.99$.

Table 1 Isentropic N-S equations: relative l_1 errors

t	NN			CU scheme		
	ρ	ρu	ρv	ρ	ρu	ρv
0.05	0.0016	0.0207	0.0497	0.0031	0.0687	0.0700
0.10	0.0019	0.0173	0.0317	0.0035	0.0395	0.0415
0.15	0.0021	0.0147	0.0243	0.0037	0.0289	0.0308

for $j, k = 10, 20, \ldots, 70$, $n = 0, 1, \ldots N_t - 1$, and $i = 1, \ldots, N_s$. Here $\bar{\boldsymbol{U}}_{j,k}^{n,\hat{\boldsymbol{p}}^i}$ is the cell average approximation of $\boldsymbol{U}(x, y, t_n; \hat{\boldsymbol{p}}^i)$ over the cell $[x_{j-1/2}, x_{j+1/2}] \times [y_{k-1/2}, y_{k+1/2}]$, which is computed using a fine grid simulation with $\tilde{N}_x = \tilde{N}_y = 240$, using the CU scheme to find the numerical fluxes and the third-order strong stability-preserving Runge-Kutta method [2] for the time integration.

We use 300 nodes for each fully connected layer of this NN. The number of inputs into the NN is 12 data points, 4 for each equation around the cell interfaces, and there are 3 outputs, the NN fluxes around the cell interfaces. In Fig. 3, we consider the case with $\rho_1 = 0.79$, $\rho_2 = 1.32$, $\rho_3 = 0.55$, and $\rho_4 = 0.99$. The reference solution is found using a fine grid simulation with 800×800 cells. The plots show almost no discernible difference between the NN and reference solutions. We also compute relative l_1 errors for the NN and the CU solution when comparing them with the reference solution and depict the results in Table 1, indicating that the NN solution is comparable to the solution obtained by the CU scheme. The run time for the NN solution is 3.45 seconds using the GPU and 7.05 seconds using the CPU. For the CU scheme, the run time is 36.88 second. The NN solution uses a time step $\Delta t = 0.001$, which, based on observed simulations, is a much smaller time step than the CFL condition requires. The CU scheme is run using an adaptive time step based on the CFL condition.

5 Conclusion

In this paper, we have shown that we the solution to time-dependent PDEs can be simulated using only solution data and NNs. Specifically, we demonstrated that a NN could be used to calculate numerical flux values. Since the flux terms do not explicitly depend on the solution's spatial location, data from a small subset of the computational domain was used to train the NNs. Because the NN only learns from local data, the presented method can generalize to initial conditions not seen in the training set, allowing one to solve a wide range of problems.

Acknowledgements The work of A. Chertock and C. Leonard were supported in part by NSF grants DMS-1818684 and DMS-2208438.

References

1. Chen, Z., Churchill, V., Wu, K., Xiu, D.: Deep neural network modeling of unknown partial differential equations in nodal space. J. Comput. Phys. **449**, Paper No. 110782, 20 (2022)
2. Gottlieb, S., Shu, C.-W., Tadmor, E.: Strong stability-preserving high-order time discretization methods. SIAM Rev. **43**, 89–112 (2001)
3. Hornik, K., Stinchombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Netw. **2**, 359–366 (1989)
4. Kurganov, A., Noelle, S., Petrova, G.: Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton-Jacobi equations. SIAM J. Sci. Comput. **23**, 707–740 (2001)
5. Lagaris, I.E., Likas, A., Fotiadis, D.I.: Artificial neural networks for solving ordinary and partial differential equations. IEEE Trans. Neural Netw. **9**, 987–1000 (1998)
6. Pareschi, L., Russo, G.: Implicit-Explicit Runge-Kutta schemes and applications to hyperbolic systems with relaxation. J. Sci. Comput. **25**, 129–155 (2005)
7. PyTorch. https://pytorch.org
8. Qin, T., Wu, K., Xiu, D.: Data driven governing equations approximation using deep neural networks. J. Comput. Phys. **395**, 620–635 (2019)
9. Raissi, M., Perdikaris, P., Karniadakis, G.E.: 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)