Physica D 451 (2023) 133773

Contents lists available at ScienceDirect

Physica D

journal homepage: www.elsevier.com/locate/physd

Identification of the flux function of nonlinear conservation laws with variable parameters

Qing Li [a], Jiahui Geng [b], Steinar Evje [a,*]

[a] University of Stavanger, Department of Energy and Petroleum, Group of Computational Engineering, Norway
[b] University of Stavanger, Department of Electrical Engineering and Computer Science, Stavanger, Norway

ARTICLE INFO

Article history:
Received 8 November 2022
Received in revised form 21 March 2023
Accepted 28 April 2023
Available online 9 May 2023
Communicated by R. Kuske

Keywords:
Nonlinear conservation law
Variable parameter flux function
Machine learning method
Entropy condition
Linear Regression Neural Network
Symbolic neural network

ABSTRACT

Machine learning methods have in various ways emerged as a useful tool for modeling the dynamics of physical systems in the context of partial differential equations (PDEs). Nonlinear conservation laws (NCLs) of the form $u_t+f(u)_x=0$ play a vital role within the family of PDEs. A main challenge with NCLs is that solutions contain discontinuities. That is, one or several jumps of the form $(u_L(t), u_R(t))$ with $u_L \neq u_R$ may move in space and time such that information about $f(u)$ in the interval associated with this jump is not present in the observation data. Moreover, the lack of regularity in the solution $u(x,t)$ prevents use of physics informed neural network (PINN) and similar methods. The purpose of this work is to propose a method to identify the nonlinear flux function $f(u,\beta)$ with variable parameters of an unknown scalar conservation law

$$u_t + f(u,\beta)_x = 0 \tag{$*$}$$

with u as the dependent variable and β as the parameter. In a recent work we introduced a framework coined ConsLaw-Net that combines a symbolic multi-layer neural network and an entropy-satisfying discrete scheme to learn the nonlinear, unknown flux function $f(u;\beta)$ for various fixed β. Learning the flux function with variable parameters, marked as $f(u,\beta)$, is more challenging since it requires an understanding of the relationship between the variable u and parameter β as well. In this work we demonstrate how to couple ConsLaw-Net to the Linear Regression Neural Network (LRNN) to learn the functional form of the two variable function $f(u,\beta)$. In addition, ConsLaw-Net is here further developed and made more generic by using a refined discrete scheme combined with a more general symbolic neural network (S-Net). We experimentally demonstrate that the upgraded ConsLaw-Net integrated with LRNN is well-suited for learning tasks, achieving more accurate identification than existing learning approaches when applied to problems that involve general classes of flux functions $f(u,\beta)$. The investigations of this work are restricted to the class of polynomial, rational functions.

1. Introduction

1.1. Background

Machine learning methods for solving partial differential equation problems have been widely used in science and engineering, for example, electromagnetism [1], aerodynamics [2,3], weather prediction [4], and geophysics [5]. Raissi et al. [6] introduced the physics informed neural networks (PINNs) for solving partial differential equations (PDEs) and learning parameters in PDEs. Long et al. [7,8] proposed PDE-Net that combines numerical approximation of differential operators and symbolic multi-layer neural networks. The authors reported that PDE-Net has the most flexible and expressive power by learning both differential operators and the nonlinear response function of the underlying PDE model. In this work, we focus on the problem of learning the unknown nonlinear flux function $f(u,\beta)$ that is involved in the general scalar nonlinear conservation law. Here we are restricted to the one-dimensional case, given by

$$\begin{aligned} u_t + f(u,\beta)_x &= 0, \qquad x \in (0,L) \\ u|_{t=0} &= u_0(x) \\ u_x|_{x=0} &= u_x|_{x=L} = 0 \end{aligned} \tag{1}$$

where $u = u(x,t)$ is the main variable and β is a parameter, typically, related to different forces that drive the process under consideration. Note that this extension to include dependence on a parameter β is highly relevant for many situations where different physical parameters naturally vary over certain intervals.

* Corresponding author.
E-mail addresses: quing.li@uis.no (Q. Li), jiahui.geng@uis.no (J. Geng), steinar.evje@uis.no (S. Evje).

https://doi.org/10.1016/j.physd.2023.133773

Displacement of several fluids is one classical example where quantities like fluid viscosity and density may strongly influences the displacement behavior and their roles are expressed in a β-like parameter [9,10]. In the context of learning $f(u;\beta)$ for a fixed β, we identify $f(u;\beta)$ with u as the only variable based on a series of observations $\{u(x_j,t_i)\}$ at specific time points $\{t_i\}$ described on a spatial grid $\{x_j\}$. However, when learning $f(u,\beta)$, we seek the analytical expression of $f(u,\beta)$ that includes both the variable u and the parameter β, given a set of observations $\{u(x_j,t_i)\}$ at various time points t_i and at different values of β over an interval of interest.

There are special challenges in identifying $f(u,\beta)$ in (1) as compared to previous PDE models as studied in, e.g., [6–8]. Firstly, in our scenario, the term $f(u,\beta)_x$ in (1) cannot be expressed by $f_u(u,\beta)u_x$ since u generally is not a differentiable function. Consequently, we cannot approximate differential operators using convolutions, as done in [7,8]. In our earlier work [11], we demonstrated that the proposed ConsLaw-Net for solving $f(u;\beta)$ for a fixed β can handle this problem. Secondly, variable u and parameter β are completely distinct. How to learn the functional relation between u and β? A natural idea is to try to discover their relationship by treating both as variables. I.e., feed u and β into the ConsLaw-Net at the same time. However, it seems difficult to control and minimize the error associated with both u and β simultaneously. The discrete scheme we use to represent the entropy solution of (1) would produce a cumulative error in the ConsLaw-Net, especially when the number of iterations is large. Incorporating learning of an unknown parameter into the ConsLaw-Net significantly increases the complexity of the calculations involved in minimizing the loss function. To overcome this, we introduce a Linear Regression Neural Network(LRNN) to distill the structure of $f(u,\beta)$ learned by relying on the ConsLaw-Net for a few selected values of β.

At this stage, it seems appropriate to highlight some essential features of the entropy solution associated with (1). It is well known that conservation laws of the form $(1)_{1,2}$ (Cauchy problem) do not in general possess classical solutions. Instead, one must consider weak solutions in the sense that the following integral equality holds [12–14]

$$\int_{\Omega_x}\int_0^T\Big[u\phi_t+f(u,\beta)\phi_x\Big]\,dx\,dt+\int_{\Omega_x}u_0(x)\phi(x,t=0)\,dx=0 \quad (2)$$

for all $\phi\in C^1$ such that $\phi(x,t):\Omega_x\times(0,T)\to\mathbb{R}$ and which is compactly supported, i.e., ϕ vanishes at $x\to\Omega_x$ and $t\to T$. It follows that if a discontinuity occurs in the solution, i.e., a left state u_L and a right state u_R, then the speed s satisfies the Rankine–Hugoniot jump condition [12,14], i.e.,

$$s=\frac{f(u_L,\beta)-f(u_R,\beta)}{u_L-u_R}. \quad (3)$$

However, direct calculations show that there are several weak solutions for one and the same initial data [13]. To overcome this issue of non-uniqueness of weak solutions, we need criteria to determine whether a proposed weak solution is admissible or not. This has led to the class of *entropy* solutions, which amounts to introducing an additional constraint which ensures that the unique physically relevant one is found among all the possible weak solutions. There are different ways to express the entropy condition for scalar nonlinear conservation laws. One variant is by introducing an entropy pair (η,q) where $\eta:\mathbb{R}\to\mathbb{R}$ is any strictly convex function and $q:\mathbb{R}\to\mathbb{R}$ is constructed as [14,15] (where $f'(s)$ represents derivative with respect to s since β is a passive parameter)

$$q(v,\beta)=\int_0^v f'(s,\beta)\eta'(s)\,ds \quad (4)$$

for any v. This implies that $q'=f'\eta'$. Then, u is an entropy solution of $(1)_{1,2}$ if (i) u is a weak solution in the sense of (2); (ii) u satisfies in a weak sense $\eta(u)_t+q(u,\beta)_x\le 0$ for any pair (η,q). This condition can also be formulated as the following characterization of a discontinuity (u_L,u_R) [14,16]: For all numbers v between u_L and u_R,

$$\frac{f(v,\beta)-f(u_L,\beta)}{v-u_L}\ge s\ge\frac{f(v,\beta)-f(u_R,\beta)}{v-u_R} \quad (5)$$

where s is given by (3). In particular, it gives a tool for constructing unique solutions of nonlinear scalar conservations laws as given by (1).

The ConsLaw-Net framework introduced in [11] relies on different symbolic neural networks, S-Net-M for multiplicative function and S-Net-D for division function to represent the flux function, and are combined with an entropy consistent discrete numerical scheme (ECDNS) to identify $f(u;\beta)$ for a fixed β. However, this ConsLaw-Net has two shortcomings: Firstly, the entropy consistent numerical scheme relies on a global estimate of $|f_u(u;\beta)|$ through $M=\max_u|f_u(u;\beta)|$. This facilitates the symbolic neural network to find the optimal solution quickly. However, it is also known to corrupt data accuracy. In this work we explore whether the flux function can be correctly learned when the symbolic neural network is coupled with a numerical scheme that relies on a local estimate of the form $M_{j+1/2}=\max\{|f_u(u_j^n;\beta)|,|f_u(u_{j+1}^n;\beta)|\}$ where u_j^n and u_{j+1}^n refer to u at spatial grid points x_j and x_{j+1}, to improve the overall accuracy of the method. Secondly, there are two different symbolic neural networks involved, S-Net-M and S-Net-D for multiplicative and division functions, respectively. We must decide which type of symbolic neural network to use by comparing the final losses of S-Net-M and S-Net-D, which obviously increases the computational burden.

1.2. Purpose of this work

In particular, our main contribution by this work includes:

- Provide a method for learning of the functional form of the nonlinear two-variable function $f(u,\beta)$ where u is the main variable and β is a parameter. To our knowledge, this represents the first approach to learn the unknown nonlinear flux function involved in a scalar conservation law, with respect to both its dependent variable u as well as the parameter β.
- We update the ConsLaw-Net proposed in [11] regarding the following two aspects:
  - We use a local rather than global estimate of the wave speed for the Rusanov scheme. Specifically, we use a local estimate $M_{j+1/2}=\max\{|f_u(u_j^n;\beta)|,|f_u(u_{j+1}^n;\beta)|\}$ instead of a global estimate of $M=\max_u|f_u(u;\beta)|$. This updated ECDNS provides a more accurate description of observation data versus the underlying flux function, but also increases the computational time for model optimization (i.e., identification of the unknown flux function). The success of applying the improved numerical scheme demonstrates that the methodology can be extended to a wider variety, i.e., the ConsLaw-Net model has some robustness;
  - We employ a unified Symbolic Multi-layer Neural Network (S-Net) to learn $f(u;\beta)$, regardless of whether it is a division or a multiplication function that is involved in the unknown underlying conservation law (1).

Hence, the updated ConsLaw-Net is a refinement and a generalization as compared to its first version presented

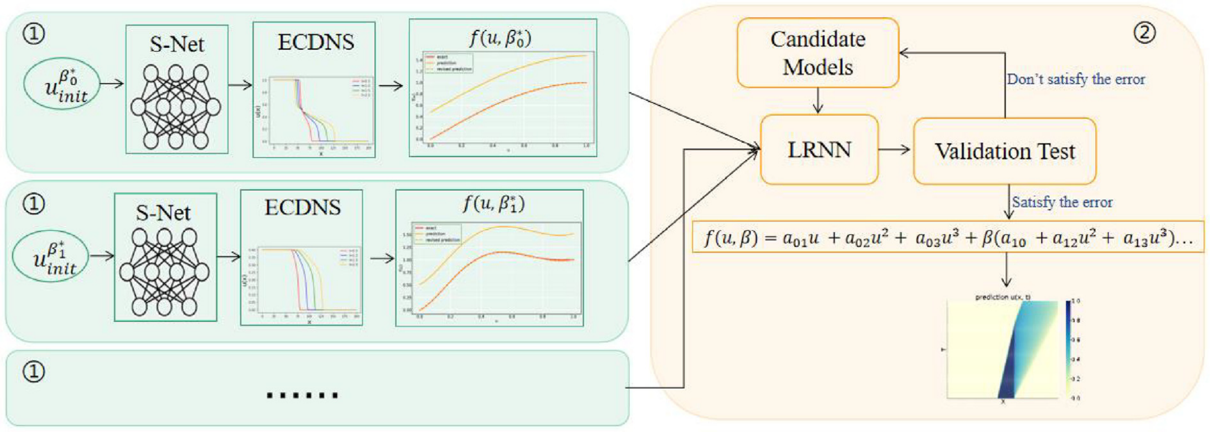


Fig. 1. Pipeline for our approach. There are two steps: ① using the improved ConsLaw-Net to learn $f(u;\beta)$ where $\beta \in \{\beta_0^*, \beta_1^*, \ldots, \beta_{N_\beta}^*\}$. We use the symbol β^* to distinguish the values of β used in the first step. The learned expression $f(u;\beta_i^*)$ in each subproblem i is referring to an equation with u as variable but not β_i^*. ② Given the results of the first step, we first construct a collection of candidate models of the two variable function $f(u,\beta)$, and then use LRNN to obtain the coefficients of the selected candidate model. If this candidate model can satisfy the error criterion, we will apply it to compute the solutions of (1). Otherwise, we reselect the candidate model and repeat the process in the second step.

in [11]. However, this comes with an additional challenge related to the optimization of (1). To handle this, we design a new loss function.

- Learned graph neural networks (GNNs) have recently been established as fast and accurate methods when applied to PDEs with regular solutions, such as the Wave equation, Poisson, or Navier–Stokes equations [17–19]. We compare our method with the GNNs method proposed in [17]. Experiments show that our method generates relative accurate predictions whereas this GNN does not seem to have a built-in capacity to handle the problem (1) well.

The proposed approach is summarized in Fig. 1. This method consists of two steps. The first step is to use the optimized ConsLaw-Net to learn $f(u;\beta)$ for a few values of β, whereas the second step is to use LRNN to learn $f(u,\beta)$ relying on the results of the first step. The challenge with lack of uniqueness, i.e., the fact that many different flux functions can give rise to the same observation data, is dealt with by using a combination of two different components: (i) we collect observation data corresponding to a set of different initial states; (ii) regularity information pertaining to the class of flux functions $f(u,\beta)$ we seek, is incorporated through the use of symbolic neural networks, as mentioned above.

1.3. Related work

Learning the flux function of a nonlinear conservation law from sparse data has important applications in industry. There are several examples of non-machine approaches to deal with this problem. Holden et al. used the front-tracking algorithm to reconstruct the flux function from observed solutions with suitable initial data [20] for a traffic flow relevant model. Recent studies on the reconstruction of the flux function for sedimentation problems involved the separation of a flocculated suspension into a clear fluid and a concentrated sediment [21,22]. In particular, Bürger and Diehl revealed that the inverse problem of identifying the batch flux density function has a unique solution, and derived an explicit formula for the flux function [23]. This method was recently extended to construct almost the entire flux function [24] using a cone-shaped separator. Diehl explored a direct inversion method using linear combinations of finite element hat functions to represent the unknown nonlinear function [25].

With the development of deep AI, more and more neural network methods are used to solve forward and inverse problems of PDEs and ODEs. These problems frequently involve recovering solutions to PDEs [6] or physical quantities, such as the density, viscosity, or other material parameters [26–28], from a sparse set of measurements. We refer to [29,30] for examples of methods based on assuming that the governing PDE is linear combination of a few differential terms in a prescribed dictionary, and the objective is to find the correct coefficients. Raissi et al. [6] introduced the physics informed neural networks (PINNs) for solving two types of problems: data-driven solutions and data-driven discovery of partial differential equations. Except for a few scalar learnable parameters, the explicit form of the PDEs is assumed to be known in the second problem. Therefore, the structure of the equation is used as part of the loss function to guide the optimization. However, PINNs seem not able to learn the nonlinear hyperbolic PDE that governs two-phase transport in porous media [10]. It was suggested that this shortcoming of PINNs for hyperbolic PDEs is due to the lack of regularity in the solution. Mesh-based simulations have recently made significant progress [17,18], enabling faster runtimes than principled solvers, and higher adaptivity to the simulation domain compared to the grid-based convolutional neural network (CNNs) [31,32]. Neural networks have been used also in combination with discrete schemes to accelerate computations of PDEs [33].

The rest of this paper is organized as follows: In Section 2 the updated version of the ConsLaw-Net proposed in [11] is described. In particular, it is described how LRNN is used to learn the two variable flux function $f(u,\beta)$. In Sections 3 and 4 we present different experimental results on identifying $f(u,\beta)$ using our method by exploring performance, respectively, for two different families of flux functions. In Section 5, we close with a discussion and outlook.

2. Method

2.1. The first step: using an improved ConsLaw-net to learn the fixed parameter problem $f(u;\beta)$, with $\beta \in \{\beta_0^*, \beta_1^*, \ldots, \beta_{N_\beta}^*\}$.

2.1.1. Entropy consistent discrete numerical scheme (ECDNS)

We investigate a discretization of the spatial domain $[0,L]$ in terms of $\{x_i\}_{i=0}^{N_x-1}$ where $x_i = (1/2+i)\Delta x$ for $i = 0,\ldots,N_x-1$ with $\Delta x = L/N_x$. Furthermore, we consider time lines $\{t^n\}_{n=0}^{N_t}$ with $N_t\Delta t = T$. Our discrete version of (1) is based on the Rusanov scheme [12] which takes the form

$$u_j^{n+1} = u_j^n - \lambda(F_{j+1/2}^n - F_{j-1/2}^n), \qquad \lambda = \frac{\Delta t}{\Delta x},$$

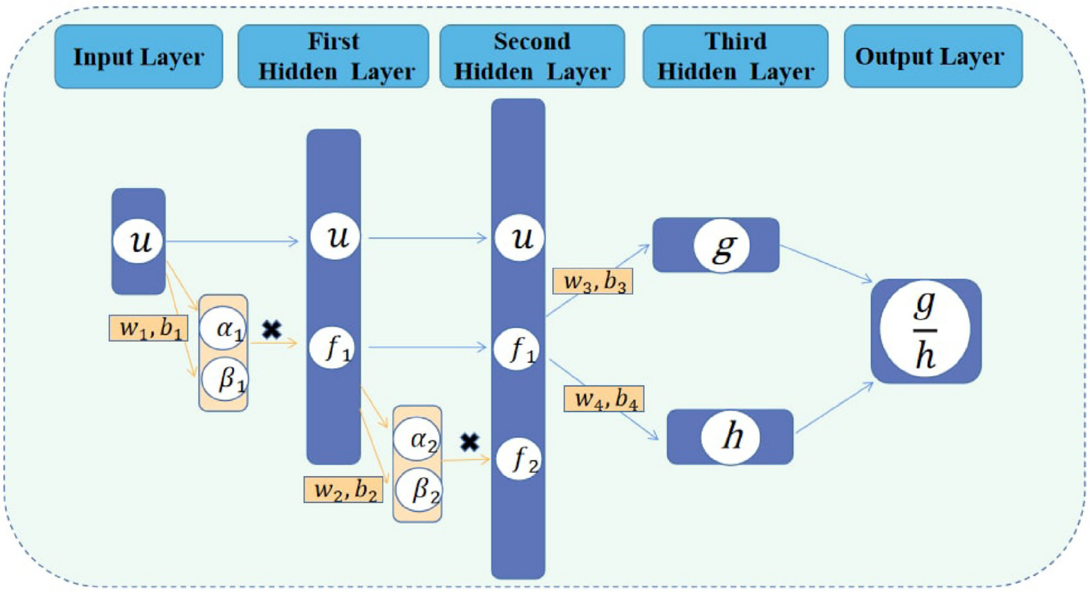


Fig. 2. The structure of S-Net with three hidden layers.

$$u_1^{n+1} = u_2^{n+1}, \qquad u_{N_x}^{n+1} = u_{N_x-1}^{n+1} \tag{6}$$

with $j = 2, \ldots, N_x - 1$ and where the Rusanov flux takes the form

$$F_{j+1/2}^n = \frac{f(u_j^n;\beta) + f(u_{j+1}^n;\beta)}{2} - \frac{M_{j+1/2}^n}{2}(u_{j+1}^n - u_j^n).$$

We adopt a local estimation by using $M_{j+1/2}^n = \max\{|f_u(u_j^n;\beta)|, |f_u(u_{j+1}^n;\beta)|\}$ instead of a global approximation through $M_{j+1/2}^n = \max_u |f_u(u;\beta)|$, as used in [11]. The local estimation is beneficial to simulation accuracy but harmful to the optimization of the model. The CFL condition determines the magnitude of Δt for a given Δx through the constraint [12]

$$\mathrm{CFL} := \frac{\Delta t}{\Delta x} M \le 1, \qquad M = \max_u |f_u(u;\beta)|.$$

We invoke the CFL condition in Algorithm 1. Algorithm 2 presents how to learn the solution $U^n = \left\{u(x_j, t^n)\right\}_{j=1}^{N_x}$ of the discrete conservation law (6). The local estimation $M_{j+1/2}^n$ is used to define the interface flux $F_{j+\frac{1}{2}}$ in Algorithm 2. Note that we for simplicity ignore the explicit dependence on β in Algorithm 1 and Algorithm 2 since they are applied for $f(u;\beta)$ with a fixed choice of β.

Algorithm 1: CFL

```
Input: $L$: length of the spatial domain; $N_x$: the number of
       spatial grid cells; $f(u)$: the nonlinear flux function;
       $T$: computational time period;
Output: $\Delta t$: local time interval
$\Delta x = L/N_x$
$M = \max_u |f'(u)|$
$dt = (\frac{3}{4}\Delta x)/(M + 0.0001)$
$n\_time = \lfloor T/dt \rfloor$
$\Delta t = T/n\_time$
```

2.1.2. S-net

Inspired by [34,35], we use the S-Net to represent the unknown function $f(u;\beta)$ because S-Net can learn the analytical expression of the flux function $f(u;\beta)$. In this work we no longer distinguish between S-Net-M and S-Net-D, but instead use S-Net-D as S-Net to learn all functions. Fig. 2 depicts the building of

Algorithm 2: DataGenerator

```
Input: $T$: computational time period; $N_x$: the number of
       spatial grid cells; $L$: length of the spatial domain;
       $u_0 = \{u_0(x_j)\}_{j=1}^{N_x}$: initial state set of dimension $N_x$;
       $f(u)$: the flux function;
Output: $U = \{u_j^n\}$: the solution based on initial state $u_0$;
$\Delta t$ = CFL($L$, $N_x$, $f(u)$, $T$)
$\Delta x = L/N_x$
$U[0] = u_0$
$\tilde{u} = u_0$
for n = 1,...,$T/\Delta t$ do
    for j = 1,...,$N_x$ - 1 do
        $F_{j+1/2} = \frac{1}{2}\left(f(\tilde{u}_j) + f(\tilde{u}_{j+1})\right) - \frac{\max\{|f'(\tilde{u}_j)|, |f'(\tilde{u}_{j+1})|\}}{2}\left(\tilde{u}_{j+1} - \tilde{u}_j\right)$
    end
    for j = 2,...,$N_x$ - 1 do
        $u_j = \tilde{u}_j - \frac{\Delta t}{\Delta x}\left(F_{j+1/2} - F_{j-1/2}\right)$
    end
    $u_1 = u_2$
    $u_{N_x} = u_{N_x-1}$
    $\tilde{u} = u$
    $U[i] = u$
end
```

S-Net in the form of $f = g/h$ with three hidden layers. As shown in Fig. 2, the identity directly maps u from input layer to the first hidden layer. The linear combination map uses parameters $\mathbf{w_1}$ and $\mathbf{b_1}$ to choose two elements from u and are denoted by α_1 and β_1.

$$(\alpha_1, \beta_1)^T = \mathbf{w_1} \cdot (u) + \mathbf{b_1}, \mathbf{w_1} \in \mathbb{R}^{2\times 1}, \mathbf{b_1} \in \mathbb{R}^{2\times 1} \tag{7}$$

These two elements of α_1 and β_1 are multiplied which give

$$f_1 = \alpha_1 \beta_1 \tag{8}$$

Apart from u gotten by the identity map, f_1 also is input to the second hidden layer

$$(\alpha_2, \beta_2)^T = \mathbf{w_2} \cdot (u, f_1)^T + \mathbf{b_2}, \mathbf{w_2} \in \mathbb{R}^{2\times 2}, \mathbf{b_2} \in \mathbb{R}^{2\times 1}. \tag{9}$$

Similarly with the first hidden layer, we get another combination f_2

$$f_2 = \alpha_2 \beta_2 \tag{10}$$

Then we obtain the numerator part g and the denominator part h of the flux function $f(u)$ based on $\mathbf{w_3}$, $\mathbf{b_3}$ and $\mathbf{w_4}$, $\mathbf{b_4}$, respectively.

$$g = \mathbf{w_3} \cdot (u, f_1, f_2)^T + \mathbf{b_3}, \mathbf{w_3} \in \mathbb{R}^{1\times 3}, \mathbf{b_3} \in \mathbb{R} \tag{11}$$

$$h = \mathbf{w_4} \cdot (u, f_1, f_2)^T + \mathbf{b_4}, \mathbf{w_4} \in \mathbb{R}^{1\times 3}, \mathbf{b_4} \in \mathbb{R} \tag{12}$$

The analytic expression of the flux function f is the combination of g and h, i.e.,

$$f = \frac{g}{h}. \tag{13}$$

The parameters involved in the network described above is denoted by $\boldsymbol{\theta}$ and the resulting function according to (13) is denoted by $f_{\boldsymbol{\theta}}(u)$. More information about S-Net can be found in [11].

2.1.3. The ConsLaw-net

The ConsLaw-Net combines ECDNS and S-Net to learn $f(u; \beta)$ with $\beta \in \{\beta_0^*, \beta_1^*, \ldots, \beta_{N_\beta}^*\}$. We consider observation data in terms of x-dependent data at fixed times $\{t_i^*\}_{i=1}^{N_{\text{obs}}}$ extracted from the solution U as follows:

$$U_{\text{sub}} = \left\{ u(x_j, t_1^*), u(x_j, t_2^*), \ldots, u(x_j, t_{N_{\text{obs}}}^*) \right\}, \quad j = 1, \ldots, N_x. \tag{14}$$

Eq. (14) is utilized to select synthetic observation data denoted by U_{sub} as well as predictions based on the learned $f_{\boldsymbol{\theta}}(u; \beta)$ written as $\hat{U}_{\text{sub}}$. We specify times for collecting the time dependent data

$$T_{\text{obs}} = \{t_i^* = i\Delta t^{\text{obs}} : i = 1, \ldots, N_{\text{obs}}\}. \tag{15}$$

Typically, we set $N_{\text{obs}} = 9$ with $\Delta t^{\text{obs}} = 0.1$. For clarity, we use $f_{\boldsymbol{\theta}}(u; \beta)$ to denote the function learned by S-Net. During S-Net training, $f_{\boldsymbol{\theta}}(u; \beta)$ is fed into Algorithm 2 to obtain the predicted solution $\hat{U}$. Similar with U_{sub}, choose predicted solutions $\hat{U}_{sub}$ according to (15). The difference between U_{sub} and $\hat{U}_{sub}$, denoted as L^{data}, is involved in the optimization of the model. In addition to L^{data}, we also introduce $L_\gamma^{denominator}$ and $L_\gamma^{numerator}$ in the loss function to limit the search space of the flux function. Detailed information on the loss functions is given in Section 2.3. Herein, we use the second-order quasi-Newton method, L-BFGS-B [36,37], to update the parameter vector $\boldsymbol{\theta}$. Finally, we get the best flux function $f_{\boldsymbol{\theta}^*}(u; \beta)$ and use it to represent the learned conservation law for the selected β. The learning process is depicted in Algorithm 3.

Overall, learning the fixed parameter problem $f(u; \beta)$ can be formalized as a nonlinear optimization problem as follows:

$$\begin{aligned} &\min_{f(u)} Loss(U_{sub}, \hat{U}_{sub}), \\ &\text{s.t.} \quad f(u) \in R(u) \end{aligned} \tag{16}$$

where $f(u)$ is the flux function to be learned and $R(u)$ is the set of rational functions. U_{sub} and $\hat{U}_{sub}$ are extracted from the solution of Eq. (1) with fixed parameter β based on the true flux function and learned flux function, respectively. *Loss* is the operator that measures the difference between U_{sub} and $\hat{U}_{sub}$ and $Loss(U_{sub}, \hat{U}_{sub})$ is the objective function.

2.1.4. Why use S-net instead of the piecewise affine functions method or the finite polynomial expansion method?

A wide range of methods exists to solve nonlinear optimization problems, including traditional approaches like gradient-based and Quasi-Newton methods, as well as contemporary methods such as machine learning techniques. James and Sepúlveda [38,39] minimized $Loss(U_{sub}, \hat{U}_{sub})$ by means of gradient methods. Since $Loss(U_{sub}, \hat{U}_{sub})$ may not be differentiable with respect to the parameters of $f(u; \beta)$ and the dependence of objective function on these parameters occurs via the PDE solution, additional assumptions have to be made, and the formal gradient is obtained by means of an adjoint equation. Already for a hyperbolic scalar conservation law in one dimension, the problem of identifying a nonlinear flux $f(u; \beta)$ is generally ill-posed in the sense that $f(u; \beta)$ is not uniquely determined by observed data. For example, the same discontinuity in a solution may arise from different flux functions. Recently, Diehl in [25] formulated a general flux identification method based on expanding the flux function in terms of piecewise affine functions with unknown weights and obtained the least squares solution to the integrated form of the conservation equation which could also contain a diffusion term.

Algorithm 3: ConsLaw-Net

```
Input: T: computational time period; N_x: the number of
       spatial grid cells; L: length of the spatial domain;
       u_0: initial state vector of dimension N_x; f(u; β):
       true flux function with fixed parameters β; θ_0:
       initial parameters of S-Net; θ: parameters of S-Net;
       f_θ(u; β): flux function generated by S-Net; T_obs:
       observation time points Eq. (15); Loss: loss
       function; epoch: the number of epoch;
       DataGenerator: Algorithm 2
Output: f_θ*(u; β): the best flux function generated by the
        S-Net based on parameter vector θ*;
$U = DataGenerator(T, N_x, L, u_0, f(u; \beta))$
$U\_sub = \{u \in U | t \in T\_obs\}$
$\boldsymbol{\theta} = \boldsymbol{\theta}_0$
for i = 1,...,epoch do
    $\hat{U} = DataGenerator(T, N_x, L, u_0, f_{\boldsymbol{\theta}}(u; \beta))$
    $\hat{U}\_sub = \{u \in \hat{U} | t \in T\_obs\}$
    $loss = Loss(U\_sub, \hat{U}\_sub)$
    Updating $\theta$ and loss by optimizer L-BFGS-B;
end
$\boldsymbol{\theta}^* = \boldsymbol{\theta}$
```

To solve the nonlinear optimization problem (16), we have opted for machine learning methods, which offer greater robustness and flexibility. In particular, we are using S-Net, as employed in [7,8,34,35], to learn the unknown function. Besides S-Net, other classical methods explicitly represent functions, such as the piecewise affine functions method described in [25] and the finite polynomial expansion method. In the piecewise affine functions method, divide the u-axis into n equidistant intervals: let u^{max} be the largest value of the data and set $u_k = ku^{max}/n$ for $k = -1, \ldots, n+1$. Assume that

$$f(u) = \sum_{k=0}^{n} f_k \psi_k(u) \tag{17}$$

where f_k is $n+1$ parameters to be determined, and the affine functions are defined by

$$\psi_k(u) = \begin{cases} \frac{u - u_{k-1}}{u_k - u_{k-1}}, & u_{k-1} < u \le u_k \\ \frac{u_{k+1} - u}{u_{k+1} - u_k}, & u_k < u \le u_{k+1} \\ 0, & \text{Otherwise} \end{cases} \tag{18}$$

where $k = 0, \ldots, n$. The polynomial expression with respect to u is given by:

$$f(u; \beta) = \frac{c_0 + c_1 u + c_2 u^2 + c_3 u^3 + \cdots + c_n u^{n_1}}{d_0 + d_1 u + d_2 u^2 + d_3 u^3 + \cdots + d_n u^{n_2}} \tag{19}$$

We acquire the coefficients $\{f_k\}_{k=0}^{n}$ of (17) and $\{c_i\}_{i=0}^{n_1}$ and $\{d_i\}_{i=0}^{n_2}$ of Eq. (19), respectively, by using the observation data set. From the various methods available, we have decided to implement the S-Net approach for the following reasons:

- Experimental performance: In Section 3.1.2 we do a comparison of S-Net, the piecewise affine function representation, and the finite polynomial expansion method. It is observed that the performance of S-Net is clearly better than these two other methods for our test cases.
- Robustness: S-Net demonstrates a high capacity to handle noisy and incomplete data [11]. In our problem, the number of local time steps (of length Δt) we need to compute numerical solutions through ECDNS is higher than the number of observation data, i.e., $\Delta t \ll \Delta t^{\text{obs}}$. Since Δt is dictated by the CFL condition for the given choice of the flux function f (which will vary during the training), we do not know that $\Delta t K_i = t_i^*$ for $i = 1, \dots, N_x$ for some integer K_i. In that case, we choose the one closest to t_i^*. Given that the observations are derived from approximate data, we have opted for S-Net due to its robustness.
- Flexibility: S-Net is flexible and powerful. Expanding functions with more variables is easy with S-Net, as it can automatically detect nonlinear relationships between input variables. The finite polynomial expansion method requires the manual listing of all possible combinations, which becomes an exhaustive task as the number of variables increases, potentially leading to an exponential growth in the number of combinations. Furthermore, incorporating nonlinearities such as sine and cosine into S-Net is straightforward, enabling it to learn combinations of variables that act on sine or cosine.

2.2. The second step: using LRNN to learn $f(u, \beta)$

From the first step, we have learned the fixed parameter problems $f_{\theta^*}(u; \beta)$, $\beta \in \{\beta_0^*, \beta_1^*, \dots, \beta_{N_\beta}^*\}$. Based on the above results, we learn $f(u, \beta)$ using LRNN in the following section.

2.2.1. Generate observation data $\mathbf{Y}$

We have learned the analytical expressions of $f_{\theta^*}(u; \beta)$ with some fixed β in the first step, denoted as

$$\mathbf{Y}' = \left\{ f_{\theta^*}(u; \beta) \middle| \; u \in [\min(u_0), \max(u_0)], \;\; \beta \in \{\beta_0^*, \beta_1^*, \dots, \beta_{N_\beta}^*\} \right\}.$$

Since the solution of (1) is TVD (total variation diminishing) [15, 40], we know that the solution $u(x, t)$ at any time $t > 0$ does not contain any new maxima or minima as compared to the initial data $u_0(x)$, i.e., $\min u_0(x) \le u(x, t) \le \max u_0(x)$. In the following we tactically assume that $0 \le u_0(x) \le 1$, therefore $0 \le u(x, t) \le 1$, $\forall t \in [0, T]$. Consider a discretization of $u \in [0, 1]$, denoted as

$$\mathscr{U} = \left\{ u \middle| \; u_i = i\Delta u, i = 0, 1, 2, \dots, N_u, \Delta u = 1/N_u \right\}. \tag{20}$$

Therefore, a discrete version of Y' is obtained by

$$\mathbf{Y} = \left\{ f_{\theta^*}(u_i; \beta) \middle| \; u_i \in \mathscr{U}, \;\; \beta \in \{\beta_0^*, \beta_1^*, \dots, \beta_{N_\beta}^*\} \right\}. \tag{21}$$

Typically, $N_u = 400$. Y will be used as the observation data to train LRNN.

2.2.2. Construct the candidate models for $f(u, \beta)$ based on Linear Regression Neural Network (LRNN)

Next, we create the candidate models of $f(u, \beta)$ that contain multiplication function form (22) and division function form (23):

$$f(u, \beta) = \sum_{i=0}^{K} \sum_{j=0}^{J_i} \left(a_{ij} \beta^i u^j \right) \tag{22}$$

$$f(u, \beta) = \frac{\sum_{i=0}^{K} \sum_{j=0}^{J_i} \left(a_{ij} \beta^i u^j \right)}{\sum_{i=0}^{P} \sum_{j=0}^{Q_i} \left(b_{ij} \beta^i u^j \right)} \tag{23}$$

where $K, J_i, P, Q_i \in \mathbb{N}$, and $\mathscr{P} = \left\{ a_{ij} | i = 0, \dots, K; j = 0 \cdots J_i \right\} \cup \left\{ b_{ij} | i = 0, \dots, P; j = 1, \dots, Q_i \right\} \subset \mathbb{R}$, β^i denotes the i-th power of β, u^j represents the j-th power of u. The largest power of u, i.e., J_i, Q_i, can in principle be any positive integer. The use of (22) and (23) implies that the flux function $f(u)$ we seek to identify is restricted to be a polynomial, rational function or one that can be well approximated within this class. We can appropriately limit the candidate models by using the learned expressions of $f(u; \beta^*)$ obtained from the first step. Generally speaking, the biggest power of u in the true function will not be greater than the largest power in the learned expressions. $\mathscr{P}$ is the vector of the parameters we want to learn in the second phase. According to the Occam's razor principle, the preferred formula is singled out by being the simplest one that still predicts well. Therefore, we test the candidate model based on the principle from simplicity to complexity. That is, from multiplication to division models, and from lower to higher powers. If the simple model can meet the error requirements in the validation set, we use this simple model instead of the complex model.

Specifically, to begin with a straightforward approach, we set $K = 1$ and $J_0 = J_1 = 1$ in (22), which yields a simple model,

$$f(u, \beta) = a_{00} + a_{01}u^1 + a_{10}\beta + a_{11}\beta u^1. \tag{24}$$

There are four unknown parameters, namely $a_{00}, a_{01}, a_{10}, a_{11}$, in (24). These parameters can be optimized based on the results obtained in the first step described in Section 2.1, denoted as $a_{00}^*, a_{01}^*, a_{10}^*, a_{11}^*$. The learning process is explained in detail in Sections 2.2.3 and 2.2.4. The resulting equation can be denoted as

$$f(u, \beta) = a_{00}^* + a_{01}^* u^1 + a_{10}^* \beta + a_{11}^* \beta u^1. \tag{25}$$

To evaluate the ability of (25) in representing the true flux function, we must validate it using a separate dataset. This validation dataset, denoted as U_{sub}^{valid}, comprises the solutions extracted from Eq. (1) for specific time points (15), generated using β that was not used in the first step. We can then solve for Eq. (1) with the learned flux function (25) using the same β as the validation dataset, resulting in the estimated solutions $\hat{U}_{sub}^{valid}$. If the errors between U_{sub}^{valid} and $\hat{U}_{sub}^{valid}$ satisfy the specific error requirement Δ, we consider the learned equations to be a suitable representation of the true flux function. In this case, we do not conduct further validation of the other candidate models. However, if the errors do not meet the requirements, we move to a more complex model. For instance, we may set $K = 1$ and $J_0 = J_1 = 2$ in (22). We gradually increase the complexity of the candidate models until we find one that satisfies the error requirements.

2.2.3. Establish the input $\mathbf{X}$

After choosing the possible expression of $f(u, \beta)$, we have to establish the features of the expression in order to learn the coefficients a_{ij}, b_{ij} in (22) or (23). For each item $a_{ij}(b_{ij})\beta^i u^j$ in (22) and (23), we can build features according to $\beta^i u^j$. Take the possible expression (26) as an example.

$$f(u, \beta) = a_{00} + a_{01}u^1 + a_{02}u^2 + \beta(a_{10} + a_{11}u^1 + a_{12}u^2). \tag{26}$$

For every $\beta \in \{\beta_0^*, \beta_1^*, \dots, \beta_{N_\beta}^*\}$ and $u_i \in \mathscr{U}$, we can create a set of features

$$\mathbf{X}_{i,\beta} = \{1, u_i^1, u_i^2, \beta, \beta u_i^1, \beta u_i^2\}.$$

Denote the features of the potential formulation of $f(u, \beta)$ as

$$\mathbf{X} = \left\{ \mathbf{X}_{i,\beta} \middle| \; u_i \in \mathscr{U}, \;\; \beta \in \{\beta_0^*, \beta_1^*, \dots, \beta_{N_\beta}^*\} \right\}. \tag{27}$$

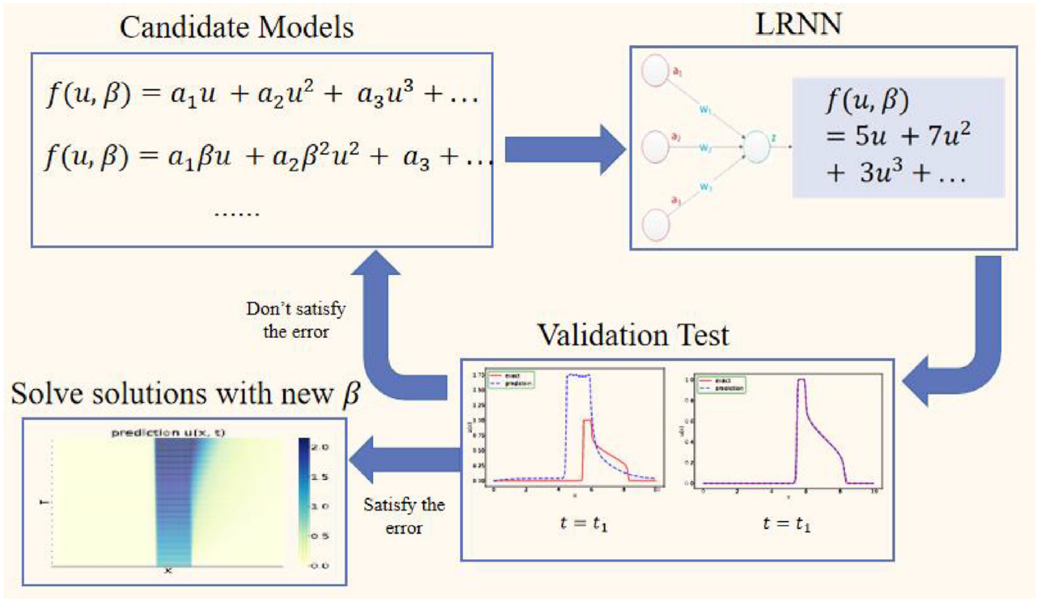

Candidate Models
$f(u,\beta) = a_1u + a_2u^2 + a_3u^3 + \dots$
$f(u,\beta) = a_1\beta u + a_2\beta^2u^2 + a_3 + \dots$
......
LRNN
$f(u,\beta) = 5u + 7u^2 + 3u^3 + \dots$
Don't satisfy the error
Validation Test
Solve solutions with new β
prediction u(x, t)
Satisfy the error
$t = t_1$
$t = t_1$

Fig. 3. Pipeline for the second step of our approach: Choose one possible expression $f(u,\beta)$ from candidate models. Build the observation data Y (21) and features of $f(u,\beta)$ (27), and get the coefficients $a_{ij}(b_{ij})$ by LRNN. Test the learned $f(u,\beta)$ on the validation data. If the error criteria is met, stop; otherwise, continue with the operation.

2.2.4. Train the model

Assume the parameter vector of LRNN is $\mathbf{W}$, so

$$\mathbf{Y} = \mathbf{W}^T\mathbf{X}. \tag{28}$$

Using the neural network, we can easily get $\mathbf{W}$ which is the vector of coefficients of the possible expression of $f(u,\beta)$.

2.2.5. Test on the validation dataset

After obtaining the trained model, it is tested on the validation set. If the error condition is satisfied, the model behavior can be predicted using the new parameter β and initial state u_0. Fig. 3 depicts the pipeline for the second step of our approach.

2.3. Loss functions

2.3.1. Loss function used in the first step

Optimizations of the ConsLaw-Net, including using a local estimate $M_{j+1/2} = \max\{|f_u(u_j^n,\beta)|, |f_u(u_{j+1}^n,\beta)|\}$ instead of a global estimate $M = \max_u |f_u(u,\beta)|$ and employing a unified S-Net to learn $f(u;\beta)$, place high demands on our loss functions. To obtain the solutions of (1), we need to estimate the quantity M, that decides the local time interval Δt used in Algorithm 2. If M is too large, it will cause Δt too small according to $\Delta t = \frac{\frac{3}{4}\Delta x}{M+0.0001}$ in Algorithm 1. In this case, the numerical scheme needs to be iterated tens of thousands of times to obtain the solutions of (1), which takes an inordinate amount of time. As a result, we estimate an upper bound on M, denoted as α. In our experiments, $\alpha = 100$. When $M < \alpha$, we compare the difference between U_{sub} and $\hat{U}_{\rm sub}$ to guide optimization, written as L^{data} and defined in (30). S-Net has two sub-neural networks, $g(u)$ and $h(u)$, to learn the numerator and the denominator, respectively.

There are two cases where the M value is too large caused by $g(u)$ and $h(u)$. One is when the denominator $h(u)$ is very close to 0, and the other is when the value of the denominator is reasonable, but the numerator $g(u)$ is vast. The above situations inevitably occur during the optimization of the model. To deal with these two cases, we introduce $L_\gamma^{denominator}$ and $L_\gamma^{numerator}$ to optimize the model. Therefore, the loss function used in Algorithm 3 can be expressed as

$$Loss = \begin{cases} L^{data} & M < \alpha \\ L_\gamma^{denominator} & M \geq \alpha,\ L_\gamma^{denominator} \neq 0 \\ L_\gamma^{numerator} & M \geq \alpha,\ L_\gamma^{denominator} = 0 \end{cases} \tag{29}$$

where L^{data}, $L_\gamma^{denominator}$, and $L_\gamma^{numerator}$ are defined as follows:

$\boldsymbol{L^{data}}$: Assume we have K_{ini} different initial states used for the training process, and solutions are described on a grid of N_x grid cells and at $N_{\rm obs}$ different times. The true observation data set denoted by $\{U_{{\rm sub},k}(x_j,t_i^*) : 1 \leq k \leq K_{ini};\ 1 \leq j \leq N_x;\ 1 \leq i \leq N_{obs}\}$ is obtained using Algorithm 2. The predicted data is created by an iterative loop in Algorithm 3, and is denoted by $\{\hat{U}_{{\rm sub},k}(x_j,t_i^*) : 1 \leq k \leq K_{\rm ini};\ 1 \leq j \leq N_x;\ 1 \leq i \leq N_{obs}\}$. So we define the data approximation term L^{data} as:

$$L^{data} = \frac{1}{K_{ini}N_xN_{obs}} \sum_{k=1}^{K_{ini}} \sum_{j=1}^{N_x} \sum_{i=1}^{N_{obs}} \left(U_{{\rm sub},k}(x_j,t_i^*) - \hat{U}_{{\rm sub},k}(x_j,t_i^*)\right)^2. \tag{30}$$

$\boldsymbol{L_\gamma^{denominator}}$: To steer the network away from small values of the denominator, we add a penalty term to our objective (29), denoted by $L_\gamma^{denominator}$. Compute the values of the subneural network $h(u)$ on $\mathscr{U}$ (20), written as

$$H = \{h(u_i) | i = 0, 1, 2, \dots, N_u\}.$$

If there are elements in H smaller than γ, it can cause M to be too large (i.e., $M \geq \alpha$). In our experiment, we set $\gamma = 0.001$ and let $L_\gamma^{denominator}$ be associated with the net effect of these small values of H

$$L_\gamma^{denominator} = \sum_0^{N_u} |h(u_i)|\, I(h(u_i) < \gamma),$$

$$I(h(u_i) < \gamma) = \begin{cases} 1, & \text{if } h(u_i) < \gamma \\ 0, & \text{otherwise} \end{cases} \tag{31}$$

Note that $0 < L_\gamma^{denominator} < (N_u + 1)\gamma$.

$\boldsymbol{L_\gamma^{numerator}}$: When the value of the denominator h is greater than γ, i.e., $h(u_i) \geq \gamma$ for all $i \in \mathscr{U}$ implying that $L_\gamma^{denominator} = 0$, and the value of the numerator g is too large, M can also be too

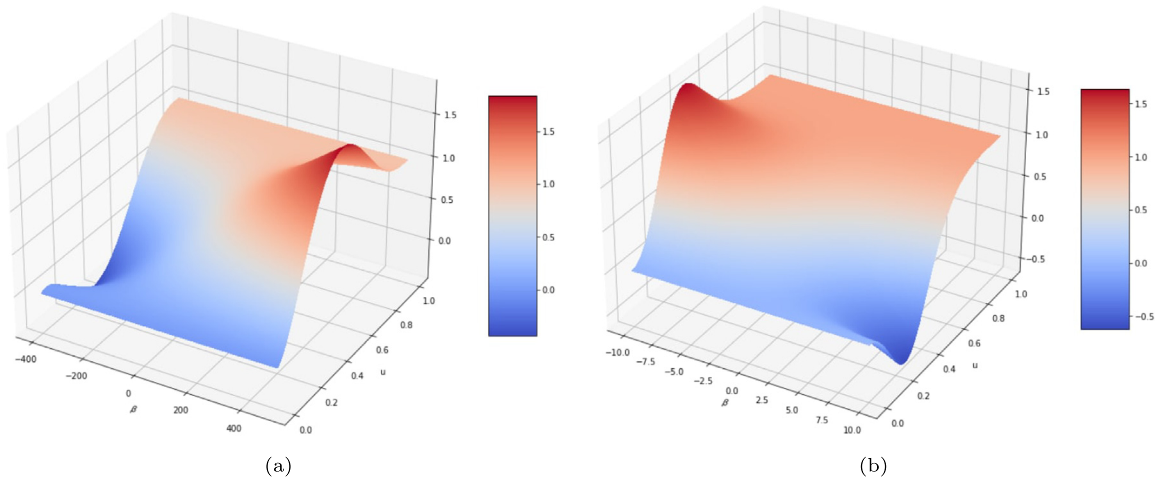


Fig. 4. (a) Flux function (34); (b) Flux function (39).

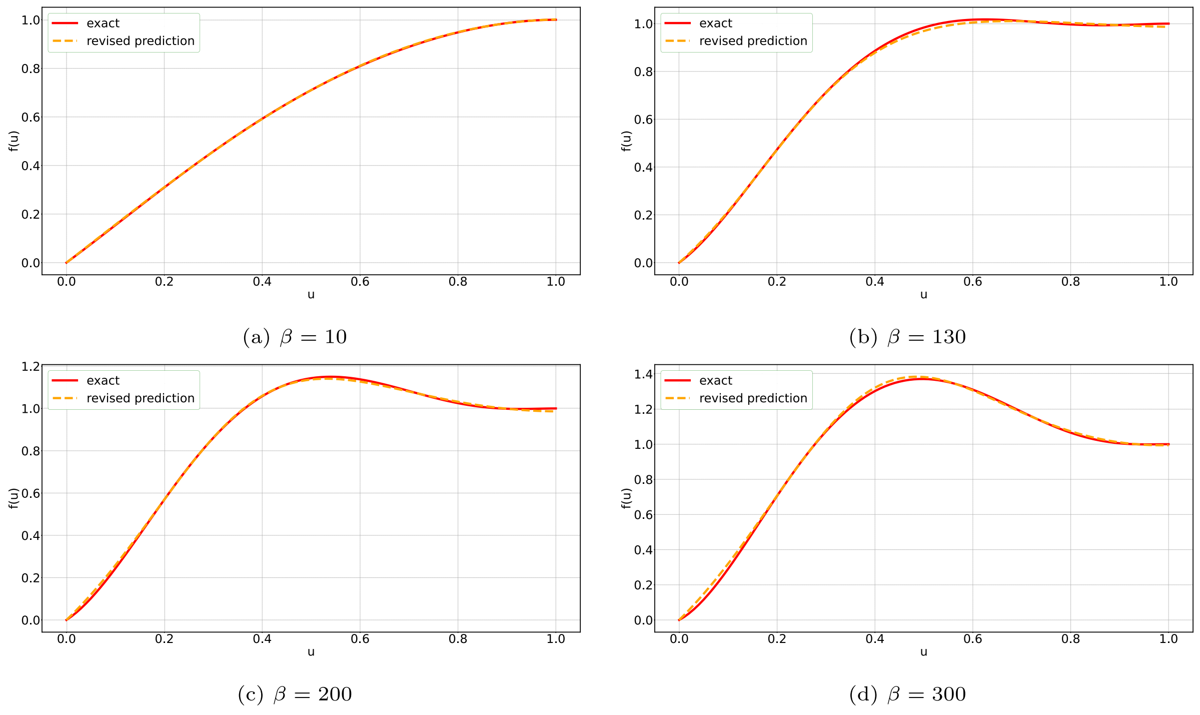


Fig. 5. The graphical results of identification of flux function $f_{\theta^*}(u;\beta)$ on $\beta = 10, 130, 200, 300$. In each subplot, the solid red line is the true function, and the orange dashed line is the learned function.

big (i.e., $M \geq \alpha$). In this situation, we compute the approximate derivative of $f(u)$ with respect to $u \in [0, 1]$. For that purpose we introduce

$$\mathscr{U}' = \mathscr{U} \cup \{1 + \Delta u\} = \{u_i | i = 0, 1, 2, \dots, N_{u+1}\}.$$

We compute the discrete derivative of $f(u)$ on $\mathscr{U}'$, denoted by

$$Df(u) = \left\{ \frac{f(u_{i+1}) - f(u_i)}{\Delta u} \middle| i = 0, 1, 2, \dots, N_u \right\}$$

where $L_\gamma^{numerator}$ is set to be

$$L_\gamma^{numerator} = \max(|Df(u)|) - \alpha. \tag{32}$$

The purpose of $L_\gamma^{denominator}$ and $L_\gamma^{numerator}$ is to steer the loss function (29) in a direction such that $Loss = L^{data}$ is the dominating term, which allows to identify the unknown flux function. As u is bounded by [0, 1], L^{data} in the case of $M < \alpha$ is much smaller than $L_\gamma^{denominator}$ or $L_\gamma^{numerator}$ which are at work in the case of $M \geq \alpha$. During the optimization process of the neural network, the parameters are updated in the direction that minimizes the loss function (29), i.e., the situation where loss is governed by $L_\gamma^{denominator}$ or $L_\gamma^{numerator}$ will be avoided. In this sense, the presence of $L_\gamma^{denominator}$ and $L_\gamma^{numerator}$ steers the loss function (29) in a direction such that $Loss = L^{data}$ is the governing term, which allows to identify the unknown flux function.

2.3.2. Loss function used in the second step

We optimize models using the Adam optimizer [41] for 5000 epochs and minimize MSE between observation data (21) and predicted values $\bar{\mathbf{Y}}_W = \left\{ \bar{f}_W(u_i, \beta) \middle| u_i \in \mathscr{U}, \beta \in \{\beta_0^*, \beta_1^*, \dots, \beta_{N_\beta}^*\} \right\}$ generated by LRNN where W represents the parameter vector, see (28). One of the primary reasons for selecting Adam is its ability to automatically adjust the learning rate, which ensures the accuracy of learning. For clarity, we denote the trained LRNN function as $\bar{f}_{W^*}(u, \beta)$. The loss function is written as

$$Loss = \frac{1}{(N_u + 1)(N_\beta + 1)} \sum_{i=0}^{N_u} \sum_{j=0}^{N_\beta} \left(f_{\theta^*}(u_i; \beta_j^*) - \bar{f}_W(u_i, \beta_j^*)\right)^2 \quad (33)$$

3. Learning a general class of nonlinear flux functions $f(u, \beta)$

In this section, we consider a class of nonlinear conservation laws that naturally arise from studying displacement of one fluid by another fluid in a vertical domain. The resulting displacement process involves a balance between buoyancy and viscous forces. Depending on the properties of the fluids used, there is room for various types of displacement behavior. This is expressed by the fact that one can derive a family of flux functions $f(u, \beta)$ which takes the following form as illustrated in [9]

$$f(u, \beta) = \frac{1}{2}u(3 - u^2) + \frac{\beta}{12}u^2\left(\frac{3}{4} - 2u + \frac{3}{2}u^2 - \frac{1}{4}u^4\right). \quad (34)$$

The parameter β represents the balance between gravity (buoyancy) and viscous forces. In this work, $\beta \in [-400, 500]$. Different values of β result in different types of flux functions. Fig. 4(a) shows the shape of $f(u, \beta)$.

In the following, we generate synthetic data with a few different values of β and a class of initial data $u_0(x)$. As mentioned in the introduction, observation data of the entropy solution leaves room for many possible flux functions to be candidates to explain the observations. However, as demonstrated in [11], the combination of using a few initial data combined with additional regularity by representing the unknown flux function through symbolic multilayer neural networks, enables identification of the relevant flux function quite effectively. In other words, as more initial data is added (with corresponding observation data), more details of the physically relevant flux function are revealed.

We consider a spatial domain $L = 10$ such that $x \in [0, 10]$ and a time interval $[0, T]$ with $T = 2$. We collect observation data in the form (14) with $N_{obs} = 9$. The aim is to identify the unknown flux function $f(u, \beta)$ for $u \in [0, 1]$.

3.1. The first step: using the improved ConsLaw-net to learn $f(u; \beta)$ where $\beta = \{10, 130, 200, 300\}$

3.1.1. The results of using S-net to represent $f(u; \beta)$

In order to learn $f(u; \beta)$ for $u \in [0, 1]$, we consider a set of initial data $\{u_0^k\}_{k=1}^K$ such that $0 \le u_0^k(x) \le 1$. We choose box-like states that give rise to Riemann problems, one at each initial discontinuity. In the following, we apply a numerical grid composed of $N_x = 400$ grid cells and choose observation data (14) with time points

$$\{t_i^*\} = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}. \quad (35)$$

S-Net is used to represent the unknown flux function $f(u; \beta)$ where $\beta \in \{10, 130, 200, 300\}$. Set three hidden layers in S-Net, and the total number of trainable parameters in every subproblem is 28. Choose the observations from the solutions U generated by Algorithm 2 based on initial states shown in Table 7 in Appendix A.1. Finally, we obtain after training the S-Net denoted by $f_{\theta^*}(u; \beta)$ by applying Algorithm 3. The resulting analytical expressions are given in Table 1. $f_{\theta^*}(u; \beta)$ differs from the true one. However, we recall that what matters is the derivative $f'_{\theta^*}(u; \beta)$. To visualize the learning effect, we plot the translated function $f_{\theta^*}(u; \beta) - f(u = 0; \beta)$ in Fig. 5. Clearly, the improved ConsLaw-Net has the ability to identify the true flux function with quite good accuracy for the flux $f_{\theta^*}(u; \beta)$ with $\beta \in \{10, 130, 200, 300\}$.

3.1.2. Comparison of S-net versus the piecewise affine functions and the finite polynomial expansion methods.

In addition to utilizing S-Net to represent the flux function, we also investigated the piecewise affine functions method (17) and (18) and the finite polynomial expansion method (19). Figs. 6(a) and 6(b) demonstrate the learned flux function $f(u; \beta)$ for $\beta = 200$ and $\beta = 300$, respectively, using the piecewise affine functions method with $n = 10$ in (17). Lower values of n gave better results with less oscillatory behavior. Figs. 6(c) and 6(d) demonstrate the learned flux function $f(u; \beta)$ for $\beta = 200$ and $\beta = 300$, respectively, using the finite polynomial expansion method (19) with $n_1 = 8$ ($\beta = 200$) and $n_1 = 7$ ($\beta = 300$). Upon comparing these figures with Figs. 5(c) and 5(d), it is evident that the flux function learned by S-Net is clearly better than the results obtained by these two methods.

3.2. The second step: using LRNN to learn the flux function (34)

In this section, we distill the analytical expression corresponding to (34) from the results of the four subproblems where $\beta = 10, 130, 200, 300$. We build candidate models in the form of (22) and (23) with different values of K, J_i, P, Q_i. For example, setting $K = 1$ and $J_i = 6$ in (22), we obtain

$$\begin{aligned} f_W(u, \beta) &= a_{00} + a_{01}u + a_{02}u^2 + a_{03}u^3 + a_{04}u^4 + a_{05}u^5 + a_{06}u^6 \\ &\quad + \beta(a_{10} + a_{11}u + a_{12}u^2 + a_{13}u^3 \\ &\quad + a_{14}u^4 + a_{15}u^5 + a_{16}u^6). \end{aligned} \quad (36)$$

The learning details using LLNN on (36) are shown in the following. Herein, $\{a_{ij} | i = 0, 1; j = 0, 1, 2, \ldots, 6\}$ is the set of parameters that we will learn in the second step. Discretize $u \in [0, 1]$ into 401 points: $\{u_i\}_{i=0}^{i=400}$, and use the following 14 features at each point $u_i \in \{u_i\}_{i=0}^{i=400}$:

$$\mathbf{X}_{i,\beta} = [1, u_i, u_i^2, u_i^3, u_i^4, u_i^5, u_i^6, \beta, \beta u_i, \beta u_i^2, \beta u_i^3, \beta u_i^4, \beta u_i^5, \beta u_i^6].$$

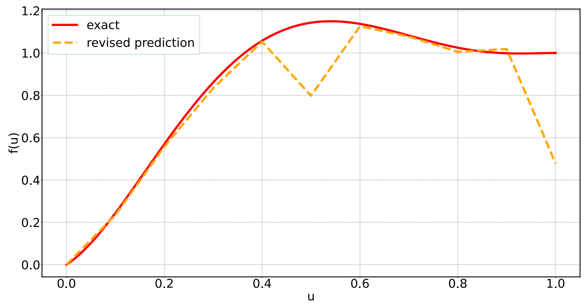
Feed features $\mathbf{X} = \{\mathbf{X}_{i,\beta} | i = 0, 1, 2 \ldots 400; \beta = 10, 130, 200, 300\}$ into LRNN, and finally we acquire the learned parameters of $\{a_{ij} | i = 0, 1; j = 0, 1, 2, \ldots, 6\}$ by using Adam optimizer to train the LRNN. The learned analytical expression of (36) is denoted by

$$\begin{aligned} f_{W^*}(u, \beta) &= 0.15651u^6 - 0.018213u^5 - 0.39308u^4 - 0.33295u^3 \\ &\quad + 0.19598u^2 + 1.3823u + 0.008492 \\ &\quad + \beta(-0.037291u^6 + 0.051649u^5 + 0.059704u^4 \\ &\quad - 0.12439u^3 + 0.048243u^2 + 0.0020499u \\ &\quad - 1.7243e-5). \end{aligned} \quad (37)$$

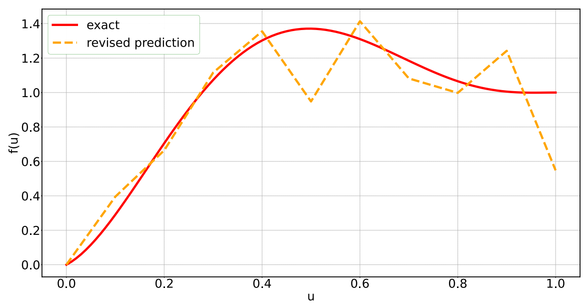
Is the identified two variable function $f_{W^*}(u, \beta)$ in (37) capable to represent (34) for different β in the whole interval $[-400, 500]$? We test on the validation set where $\beta = -50$ and $\beta = 350$ (which is well outside the β values used for training). Table 2 shows the test results of (37) where $K = 1$ and $J_i = 6$. Obviously, for both cases $\beta = -50$ and $\beta = 350$, the error is relatively small, and meets the criteria we set, $\Delta = 0.0001$. We also test other candidate models on the validation set. The results are shown in Table 3. These models may perform better in one case, but worse in another, such as $K = 1$ and $J_i = 4$. Or even worse in both cases, such as $K = 1$ and $J_i = 2$. It is obvious that when $K = 1$ and $J_i = 6$, the candidate model has the best performance. Therefore, we choose (37) as the preferred learned model and use it to test for new initial states and parameters.

Table 1
The expression results of identification of flux function $f_{\theta^*}(u;\beta)$ with specific β.

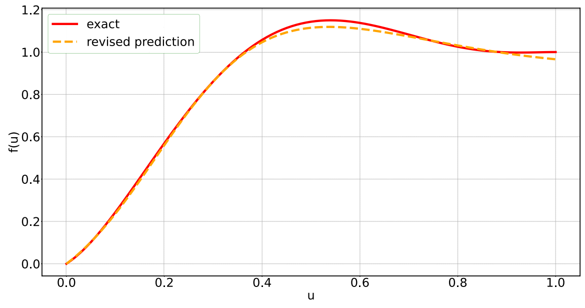
β	The identification of flux function $f_{\theta^*}(u;\beta)$
$\beta_1 = 10$	$f_{\theta^*}(u;\beta_1) = \frac{-0.0005u^8+0.0029u^7-0.0036u^6-0.0030u^5+0.3214u^4-0.9641u^3-0.0198u^2+1.7007u-1.6453}{0.0002u^8-0.0014u^7+0.0018u^6+0.0015u^5-0.0013u^4+0.0025u^3+0.0431u^2-0.0663u+1.0205}$
$\beta_2 = 130$	$f_{\theta^*}(u;\beta_2) = \frac{0.0171u^8-0.0451u^7-0.0553u^6+0.1688u^5+0.6588u^4-0.9677u^3-1.2969u^2+1.2216u-0.5410}{0.0058u^8-0.0153u^7-0.0188u^6+0.0573u^5-0.4797u^4+0.5975u^3+0.9857u^2-0.3791u+0.3038}$
$\beta_3 = 200$	$f_{\theta^*}(u;\beta_3) = \frac{-3.004e{-}6u^8+0.0001u^7-0.0005u^6-0.0144u^5-0.0181u^4-0.3600u^3-2.2609u^2+1.7718u-0.6950}{9.4576e{-}6u^8-0.0004u^7+0.0015u^6+0.0453u^5+0.1119u^4+0.0464u^3+0.7181u^2-0.4093u+0.2220}$
$\beta_4 = 300$	$f_{\theta^*}(u;\beta_4) = \frac{2.001e{-}5u^7-0.0071u^6-0.0981u^5-0.2864u^4-0.1568u^3-3.1849u^2+2.6000u-0.9848}{-1.1929e{-}5u^7+0.0042u^6+0.0585u^5+0.1709u^4-0.0887u^3+0.6433u^2-0.4293u+0.2083}$



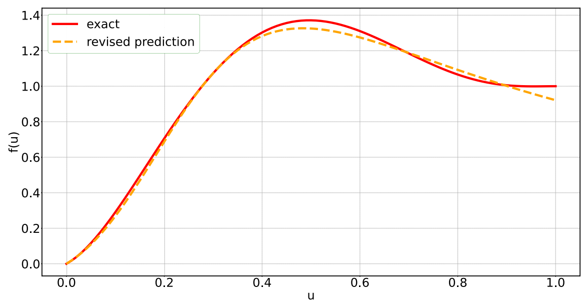
(a) $\beta = 200$ (piecewise affine)



(b) $\beta = 300$ (piecewise affine)

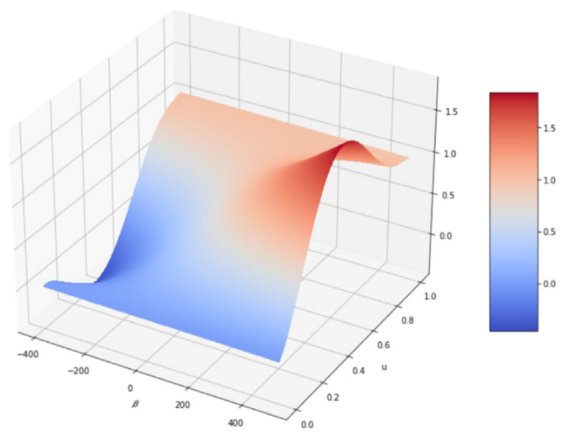


(c) $\beta = 200$ (finite polynomial)

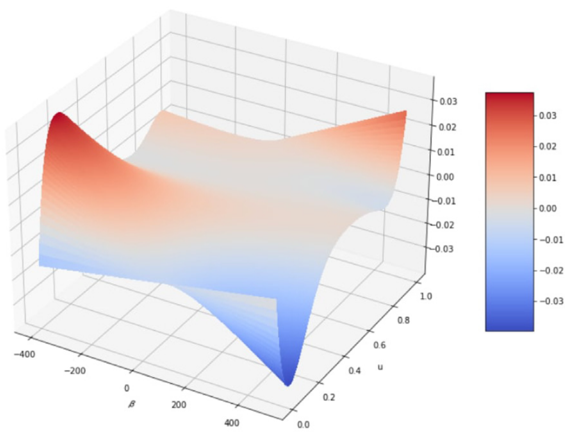


(d) $\beta = 300$ (finite polynomial)

Fig. 6. The graphical results of identification of flux function $f_{\theta^*}(u;\beta)$ using the piecewise affine functions method (17) and (18) (upper row) and the finite polynomial expansion method (19) on $\beta = 200, 300$ (lower row). In each subplot, the solid red line is the true function, and the orange dashed line is the learned function.



(a)

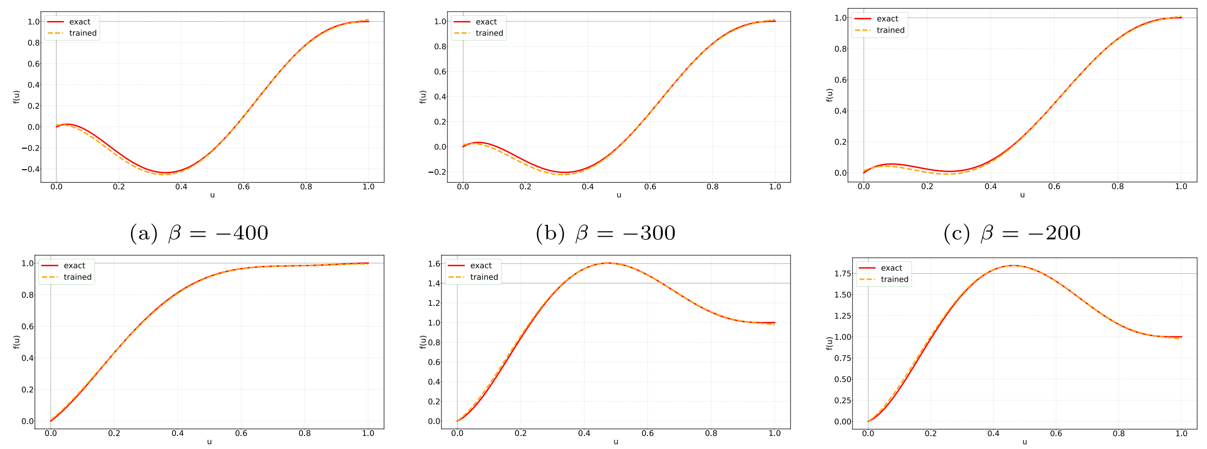


(b)

Fig. 7. (a) Learned flux function (37). (b) The difference between the true flux function (34) and learned (37).

3.3. Results and discussions

In Fig. 7(a) we first visualize the learned flux function $f_{W^*}(u,\beta)$ given by (37) whereas Fig. 7(b) visualizes the error between learned flux function and the true flux function $f(u,\beta)$ given by (34) for $\beta \in [-200, 500]$ and $u \in [0, 1]$. The absolute error of the two functions is less than 0.038 over the entire domain. Specifically, we set $\beta = -400, -300, -200, 100, 400, 500$ separately to



(a) $\beta = -400$ (b) $\beta = -300$ (c) $\beta = -200$

(d) $\beta = 100$ (e) $\beta = 400$ (f) $\beta = 500$

Fig. 8. The generalization ability of (37) tested for $\beta = -400, -300, -200, 100, 400, 500$. In each subplot, the solid red line is the true function, and the orange dashed line is the learned function.

Table 2
The results of (37) applied on the test data $\beta = -50$ and $\beta = 350$. The value of Error is the MSE between true and predicted solutions of (1) at times (35). We also plot the solution at time point $t = 0.3, 0.6, 0.9$ respectively. In each subplot, the solid red line is the true solution, and the blue dashed line is the solution generated by (37) based on initial state $u_0 = \begin{cases} 1.0, & \text{if } x \in [4.5, 6] \\ 0.0, & \text{otherwise} \end{cases}$.

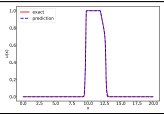
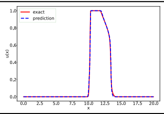
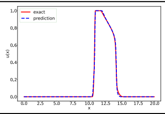
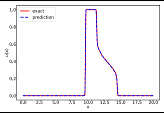
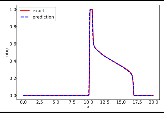
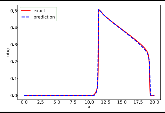
β	Error	t = 0.3	t = 0.6	t = 0.9
$\beta = -50$	7.5443e−05			
$\beta = 350$	0.0001			

Table 3
The results of different candidate models on the validation set. The value is the MSE between true and predicted solutions of (1) at times (35) for four different initial states $u^0_{\beta=-50,350} = \begin{cases} 1.0, & \text{if } x \in [4.5, 6] \\ 0.0, & \text{otherwise} \end{cases}$, $u^1_{\beta=-50,350} = \begin{cases} 0.9, & \text{if } x \in [3.5, 5] \\ 0.0, & \text{otherwise} \end{cases}$, $u^2_{\beta=-50,350} = \begin{cases} 0.8, & \text{if } x \in [5, 6.5] \\ 0.0, & \text{otherwise} \end{cases}$ and $u^3_{\beta=-50,350} = \begin{cases} 0.7, & \text{if } x \in [3, 6] \\ 0.0, & \text{otherwise} \end{cases}$, respectively.

β	$K = 1$, $J_i = 1$	$K = 1$, $J_i = 2$	$K = 1$, $J_i = 3$	$K = 1$, $J_i = 4$	$K = 1$, $J_i = 5$	$K = 1$, $J_i = 7$	$K = 1$, $J_i = 8$
$\beta = -50$	0.0013	0.0011	0.0006	0.0002	0.0002	0.0002	0.0003
$\beta = 350$	0.0515	0.0046	0.0083	0.0019	0.0028	0.0002	0.0012
β	$k = 2$, $J_i = 1$	$K = 2$, $J_i = 2$	$K = 2$, $J_i = 3$	$K = 2$, $J_i = 4$	$K = 1$, $J_i = 3$, $P = 1$, $Q_i = 3$	$K = 1$, $J_i = 4$, $P = 1$, $Q_i = 4$	$K = 1$, $J_i = 5$, $P = 1$, $Q_i = 5$
$\beta = -50$	0.0018	0.0056	0.0177	0.0095	0.0004	0.0002	0.0001
$\beta = 350$	0.0522	0.0049	0.0977	0.0221	0.0004	0.0005	0.0004

test the generalization ability of $f_{W^*}(u, \beta)$ (37). In Fig. 8, we show the true and the learned function with different values of β. It seems clear that (37) has relatively strong generalization ability.

In addition, we compare the modified graph neural networks (GNNs) used in [17] with our approach. The GNNs method offers faster runtimes than other solvers like PINNs and grid-based CNNS [31,32] and better adaptivity to the simulation domain [17–19]. One major difference between our method and the GNNs method is that our method first learns the expression of $f(u, \beta)$ and then uses the learned model to compute solutions of (1) based on new initial states and parameters. In contrast, the GNNs method directly predicts the solution of (1). This type of GNNs is called message passing neural networks (MPNNs) [42], and is given in the form

$$\frac{du(x_i, t)}{dt} = \hat{F}_\theta(x_{\mathcal{N}(i)} - x_i, u_i, u_{\mathcal{N}(i)}, \beta_i) \tag{38}$$

where x_i is the coordinate of the node i, $\mathcal{N}(i)$ is the neighborhood of node i, θ denote parameters of the MPNNs and β_i is the physical parameter in the flux function. We use Rprop optimizer [43] to minimize the loss function with learning rate set to $lr = 10^{-6}$

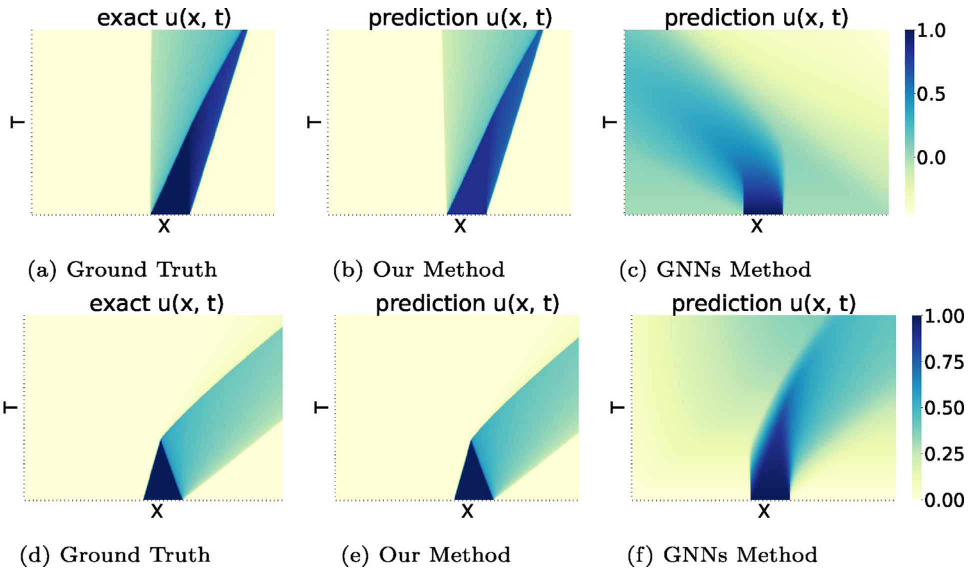


Fig. 9. The compared results of our method and the GNNs method. We compare the two methods at $\beta = -200$ (top row) and $\beta = 400$ (bottom row) based on initial state $u_0 = \begin{cases} 1.0, & \text{if } x \in [4.5, 6] \\ 0.0, & \text{otherwise} \end{cases}$. Left: True solutions of (1). Middle: Solutions generated by our method. Right: Solution generated by the GNNs method.

for 5000 iterations and batch size set to 4. To be fair, the GNNs method and our method use the same observational data. The trained model is used to predict based on new initial states and parameter β.

Fig. 9 shows the solutions of (1) with $\beta = 400$ and $\beta = -200$ using the two different methods. Our method performs better than the GNNs model. The ConsLaw-Net method is based on a discrete scheme which is entropy consistent and therefore can deal with formation of discontinuities in the solutions. The above simulation result for GNN suggests that it has not a built-in capacity to deal with discontinuous solutions.

4. Learning the flux function $f(u, \beta) = \frac{u^2\left(1-\beta(1-u)^4\right)}{u^2+0.5(1-u)^4}$

In this section, we consider a class of nonlinear conservation laws that appears in the context of two-phase flow in porous media [10]. Displacement of two fluids in an inclined reservoir gives rise to a family of flux functions in the form

$$f(u, \beta) = \frac{u^2}{u^2 + 0.5(1-u)^4}\left(1 - \beta(1-u)^4\right). \tag{39}$$

The parameter β represents the gravity effect. Herein, we study $\beta \in [-10, 10]$. Fig. 4(b) shows the shape of $f(u, \beta)$. In the following, we generate synthetic data with different values of β and a class of initial data $u_0(x)$. Consider a spatial domain $L = 10$ such that $x \in [0, 10]$ and a time interval $[0, T]$ with $T = 2$.

4.1. The first step: using the improved ConsLaw-net to learn $f(u; \beta)$ where $\beta \in \{-1, 1, 3, 5, 7\}$

Set a numerical grid composed of $N_x = 400$ grid cells, and consider observation data (14) with

$$\{t_i^*\} = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}. \tag{40}$$

S-Net with three hidden layers is used to represent the unknown flux function $f(u; \beta)$, where $\beta \in \{-1, 1, 3, 5, 7\}$. The total number of trainable parameters in every subproblem is 28. Choose the observations from the solutions U generated by Algorithm 2 based on initial states shown in Table 8 in Appendix A.2. Finally, we obtain after training the S-Net denoted by $f_{\theta^*}(u; \beta)$ by applying Algorithm 3. The resulting analytical expressions are given in Table 4. We plot the translated function $f_{\theta^*}(u; \beta) - f(u = 0; \beta)$ in Fig. 10. Clearly, the improved ConsLaw-Net has the ability to identify the true flux function in most ranges except for some lack of accuracy for $u \in \{0, 0.2\}$.

4.2. The second step: using LRNN to learn Eq. (39)

In this section, we distill the analytical expression that can approximate the true function (39) from the five subproblems where $\beta = -1, 1, 3, 5, 7$. We build candidate models in the form of (22) and (23) with different values of K, J_i, P, Q_i. For example, setting $K = 1, J_i = 5, P = 1$ and $Q_i = 5$ in (23), then we get Eq. (41) (see Box I) where $W = \left\{a_{ij}, b_{ij} | i = 0, 1; j = 0, 1, 2, \dots, 5\right\}$ is the set of parameters that we will learn in the second step. Discretize $u \in [0, 1]$ into 401 points: $\{u_i\}_{i=0}^{i=400}$, and build the following 12 features at each point $u_i \in \{u_i\}_{i=0}^{i=400}$:

$$\mathbf{X}_{i,\beta} = [1, u_i, u_i^2, u_i^3, u_i^4, u_i^5, \beta, \beta u_i, \beta u_i^2, \beta u_i^3, \beta u_i^4, \beta u_i^5].$$

Put features $\mathbf{X} = \left\{\mathbf{X}_{i,\beta} | i = 0, 1, 2 \dots 400; \beta = -1, 1, 3, 5, 7\right\}$ into LRNN, and finally we acquire the learned parameters of $\left\{a_{ij}, b_{ij} | i = 0, 1; j = 0, 1, 2, \dots, 5\right\}$ by using Adam optimizer. The learned analytical expression of $f(u, \beta)$, in terms of g and h, then becomes

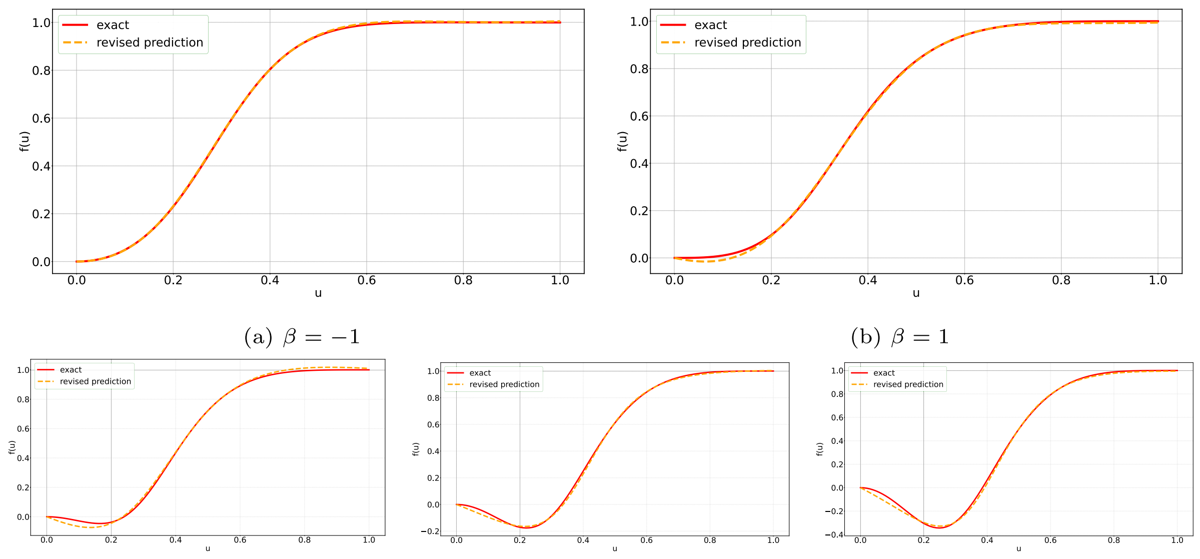
$$\begin{aligned} g_{W^*}(u, \beta) = {} & \beta(0.099u^5 + 0.078u^4 + 0.261u^3 - 0.184u^2 \\ & - 0.021u - 0.00012) \\ & + 1.920u^5 + 0.380u^4 - 0.113u^3 + 0.852u^2 \\ & - 0.060u + 0.0003 \\ h_{W^*}(u, \beta) = {} & \beta(0.126u^5 + 0.183u^4 + 0.0645u^3 - 0.186u^2 \\ & + 0.048u - 0.0016) \\ & + 1.710u^5 + 0.400u^4 + 0.509u^3 + 0.487u^2 \\ & - 0.3112u + 0.17557 \end{aligned} \tag{42}$$

Hence, the learned expression of $f(u, \beta)$ is given by

$$f_{W^*}(u, \beta) = \frac{g_{W^*}(u, \beta)}{h_{W^*}(u, \beta)}. \tag{43}$$

Table 4
The results of identification of flux function $f_{\theta^*}(u;\beta)$ with specific β.

β	The identification of flux function $f_{\theta^*}(u;\beta)$
$\beta_1=-1$	$f_{\theta^*}(u;\beta_1)=\frac{-0.0808u^8+0.2555u^7-0.0379u^6-0.4234u^5+3.2486u^4-4.6848u^3-3.2596u^2+3.6620u-1.2970}{0.3590u^8-1.1358u^7+0.1685u^6+1.8822u^5-2.9914u^4+2.7129u^3+2.6686u^2-1.7852u+0.6337}$
$\beta_2=1$	$f_{\theta^*}(u;\beta_2)=\frac{-0.9589u^8+4.7491u^7-6.1089u^6-0.2182u^5+7.73097u^4-13.0489u^3+0.7633u^2+4.3662u-2.3523}{1.4639u^8-7.2501u^7+9.3260u^6+0.3332u^5-7.0604u^4+8.1785u^3+1.9316u^2-2.5813u+1.2348}$
$\beta_3=3$	$f_{\theta^*}(u;\beta_3)=\frac{0.0002u^8+0.0091u^7+0.1226u^6+0.7346u^5+2.5724u^4+6.5190u^3+7.1617u^2-1.9897u-0.0680}{0.0003u^8+0.0121u^7+0.1630u^6+0.9770u^5+3.3709u^4+7.7091u^3+5.9935u^2-5.3643u+2.4714}$
$\beta_4=5$	$f_{\theta^*}(u;\beta_4)=\frac{1.1340u^8+5.2921u^7+8.9497u^6+6.3207u^5+2.1223u^4+1.8374u^3+1.4766u^2-0.3334u-0.5300}{1.7459u^8+8.1478u^7+13.7791u^6+9.7314u^5+3.8413u^4+4.16754u^3+1.1539u^2-2.7925u+1.4449}$
$\beta_5=7$	$f_{\theta^*}(u;\beta_5)=\frac{7.2266e-5u^8+0.0031u^7+0.0486u^6+0.3395u^5+0.8249u^4-0.1620u^3+2.3615u^2-0.6630u-0.1760}{7.7078e-5u^8+0.0033u^7+0.0518u^6+0.3621u^5+0.8955u^4+0.1576u^3+4.0315u^2-3.2001u+0.9126}$



(a) $\beta = -1$ (b) $\beta = 1$

(c) $\beta = 3$ (d) $\beta = 5$ (e) $\beta = 7$

Fig. 10. The graphical results of identification of flux function $f_{\theta^*}(u;\beta)$ for $\beta = -1, 1, 3, 5, 7$. In each subplot, the solid red line is the true function, and the orange dashed line is the learned function.

$$f_W(u,\beta) = \frac{a_{00}+a_{01}u+a_{02}u^2+a_{03}u^3+a_{04}u^4+a_{05}u^5+\beta(a_{10}+a_{11}u+a_{12}u^2+a_{13}u^3+a_{14}u^4+a_{15}u^5)}{b_{00}+b_{01}u+b_{02}u^2+b_{03}u^3+b_{04}u^4+b_{05}u^5+\beta(b_{10}+b_{11}u+b_{12}u^2+b_{13}u^3+b_{14}u^4+b_{15}u^5)} \tag{41}$$

Box I.

Next, we verify that (43) is a model that can represent well the true family of flux functions (39). We test (43) on the validation set where $\beta = -3$ and $\beta = 9$. Table 5 shows the results. Obviously, whether it is for $\beta = -3$ or for $\beta = 9$, the error is relatively small, and meets the criteria we set $\varDelta = 0.0007$. We also test other candidate models on the validation set. The results are shown in Table 6. It is obvious that when $K = 1$, $J_i = 5$, $P = 1$ and $Q_i = 5$ the candidate model has the best performance. Therefore, we choose (42) and (43) as the preferred learned model and use it to test new initial states and parameters.

4.3. Results and discussions

In Fig. 11 we first show the learned flux function (43) (panel a) and the error between true flux function (39) and the learned (43) (panel b) with $\beta \in [-10, 10]$ and $u \in [0, 1]$. The absolute error of the two functions is less than 0.118 over the entire domain. Specifically, we set $\beta = -10, -7, -5, 8, 9, 10$ separately to test the generalization ability of (43). In Fig. 12, we show the true and the learned function with different values of β. The model performs well for most parameter choice of β in $[-10, 10]$, however, conduct worse when $\beta < -8$ although still within our acceptable range $\varDelta = 0.13$.

Finally, we compare the GNNs method with our approach. We use Rprop optimizer [43] to minimize the loss function with learning rate set to $lr = 10^{-6}$ for 5000 iterations and batch size set to 5. Fig. 13 shows the solutions of (1) with $\beta = -7$ and $\beta = 9$ using the two different methods. The proposed method enables quite a nice approximation of the true predicted behavior whereas the GNNs fail to capture it, most likely, since it has not been constructed to handle discontinuous behavior.

5. Conclusion

In this paper we have developed a framework that combines the recently proposed ConsLaw-Net with LRNN to learn the functional form of the two variable function $f(u, \beta)$ involved in the

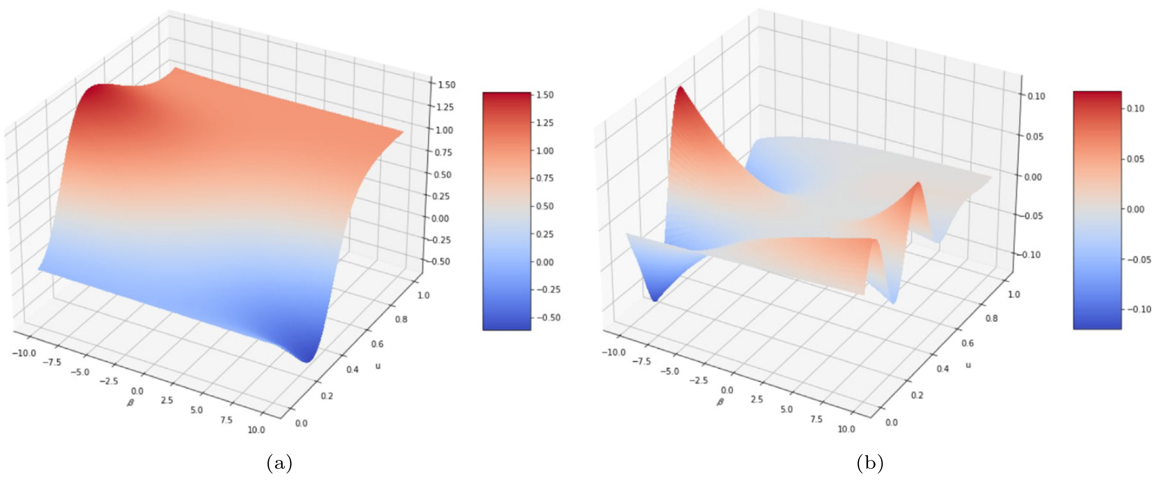


Fig. 11. (a) Flux function (43). (b) The difference between the flux function (39) and (43).

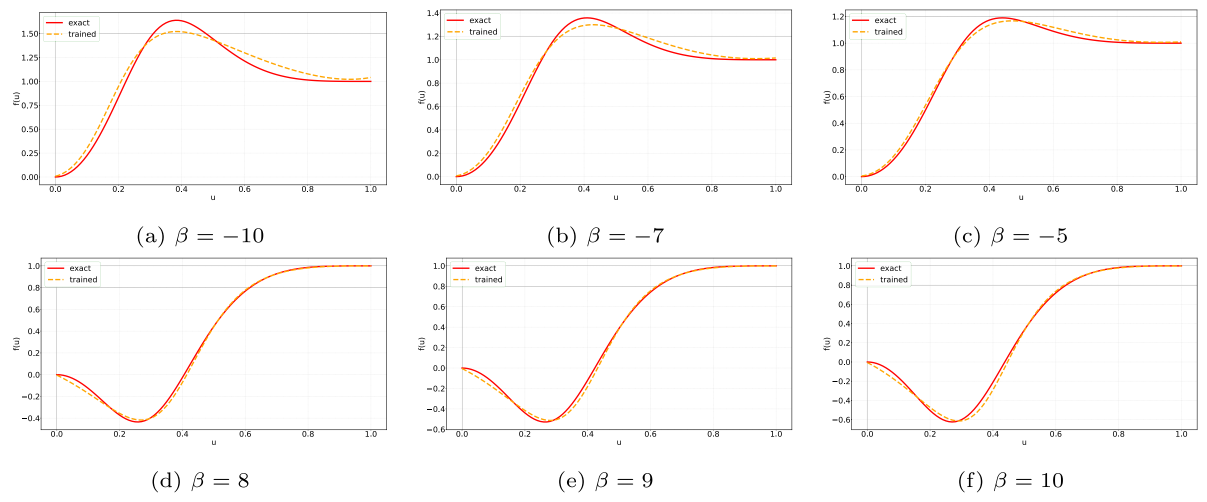


Fig. 12. The generalization ability of the model (43) for $\beta = -10, -7, -5, 8, 9, 10$. In each subplot, the red solid line is the true function, and the orange dashed line is the learned function.

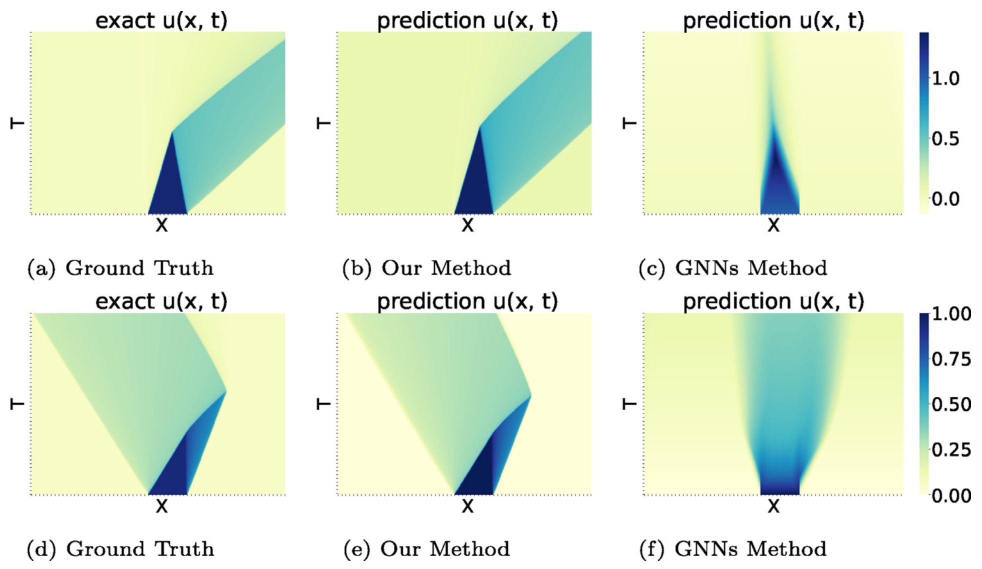


Fig. 13. The compared results of our method and the GNNs method. We compare the two methods at $\beta = -7$(top row) and $\beta = 9$(bottom row) based on initial state $u_0 = \begin{cases} 1.0, & \text{if } x \in [4.5, 6] \\ 0.0, & \text{otherwise} \end{cases}$. Left: True solutions of (1). Middle: Solutions generated by our method. Right: Solutions generated by the GNNs method.

Table 5
The results of (43) on the validation data $\beta = -3$ and $\beta = 9$. The value of Error is the MSE between true and generated solutions of (1) at times (40). We also plot the solution at time point $t = 0.3, 0.6, 0.9$ respectively. In each subplot, the solid red line is the true solution, and the blue dashed line is the solution generated by (43) based on initial state $u_0 = \begin{cases} 1.0, & \text{if } x \in [4.5, 6] \\ 0.0, & \text{otherwise} \end{cases}$.

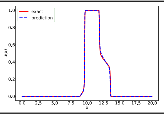
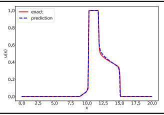
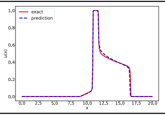
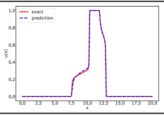
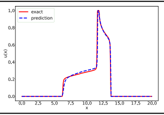
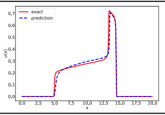
β	Error	t = 0.3	t = 0.6	t = 0.9
$\beta = -3$	0.0002			
$\beta = 9$	0.0006			

Table 6
The results of other candidate models on the validation set. The value is the MSE between true and predicted solutions of (1) at times (40) for four different initial states given by $u^0_{\beta=-3,9} = \begin{cases} 1.0, & \text{if } x \in [4.5, 6] \\ 0.0, & \text{otherwise} \end{cases}$, $u^1_{\beta=-3,9} = \begin{cases} 0.9, & \text{if } x \in [3.5, 5] \\ 0.0, & \text{otherwise} \end{cases}$, $u^2_{\beta=-3,9} = \begin{cases} 0.8, & \text{if } x \in [5, 6.5] \\ 0.0, & \text{otherwise} \end{cases}$ and $u^3_{\beta=-3,9} = \begin{cases} 0.7, & \text{if } x \in [3, 6] \\ 0.0, & \text{otherwise} \end{cases}$, respectively.

β	$K = 1$, $J_i = 6$	$K = 1$, $J_i = 7$	$K = 1$, $J_i = 8$	$K = 1$, $J_i = 3$, $P = 1$, $Q_i = 3$	$K = 1$, $J_i = 4$, $P = 1$, $Q_i = 4$
$\beta = -3$	0.0014	0.0005	0.0005	0.0210	0.0003
$\beta = 9$	0.0038	0.0036	0.0026	0.0146	0.0008

conservation law (1). More precisely, the method is composed of the following two steps: (i) learn $f(u;\beta_i)$ for a few selected β_i in an interval of interest; (ii) Combine the obtained $f(u;\beta_i)$ with LRNN to learn the full two-variable function $f(u,\beta)$. As a by product of this method we have improved the ConsLaw-Net by employing an entropy consistent numerical scheme which relies on local information (relatively the discrete spatial grid) about the unknown flux function $f(u,\beta)$. This makes the resulting ConsLaw-Net more generic and useful for challenging real-world data. The well-known challenge with identification of a nonlinear flux function from data due to appearance of discontinuities and lack of uniqueness (i.e., many different flux functions can fit with the observation data), is dealt with by relying on an entropy consistent numerical scheme in combination with a small set of more or less randomly selected initial data. We experimentally demonstrate the effectiveness of our method when applied to synthetic data generated from two different families of flux functions $f(u,\beta)$ given by (34) and (39), respectively.

Our method is sensitive to the selection of the initial states, as initial states affect the distribution of the observed data. If the data is not sufficiently evenly distributed, it may lead to failure in the first step of the learning process. The optimization of the second step is heavily dependent on the results of the first step, and a small error in the first step may amplify this error in the second step. However, in a setting where the observation data comes from measurements of experimental data our approach gives a systematic way to try to learn an underlying conservation law that can explain the observation data. Before using the learned conservation law for prediction one must be aware that enough variation in initial data has been provided.

Overall, our approach based on ConsLaw-Net and LRNN has shown a good ability learn the unknown flux function with variable parameters $f(u,\beta)$. Possible further extensions can be to explore the method in higher dimensional spaces and consider other types of observation data than we have used in this work. An interesting direction for further investigations is to see if there is room for including in the loss function some explicit characteristics of the entropy solution that will make the identification of the flux function more efficient.

CRediT authorship contribution statement

Qing Li: Methodology, Coding, Investigation, Writing, Visualization. **Jiahui Geng:** Software, Writing – review & editing. **Steinar Evje:** Coding, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix

A.1. The initial states used to train $f(u;\beta)$ in Section 3.1

See Table 7.

A.2. The initial states used to train $f(u;\beta)$ in Section 4.1

See Table 8.

Table 7
The initial states used to train $f(u;\beta)$ in Section 3.1.

$u^0_{\beta=10} = \begin{cases} 1.0, & \text{if } x \in [0,3] \\ 0, & \text{otherwise} \end{cases}$	$u^0_{\beta=130} = \begin{cases} 1.0, & \text{if } x \in [4.5,6] \\ 0.3, & \text{otherwise} \end{cases}$	$u^1_{\beta=130} = \begin{cases} 0.8, & \text{if } x \in [5,6.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^2_{\beta=130} = \begin{cases} 0.85, & \text{if } x \in [4,7] \\ 0, & \text{otherwise} \end{cases}$
$u^3_{\beta=130} = \begin{cases} 0.95, & \text{if } x \in [0,3] \\ 0, & \text{otherwise} \end{cases}$	$u^4_{\beta=130} = \begin{cases} 0.7, & \text{if } x \in [3,6] \\ 0, & \text{otherwise} \end{cases}$	$u^5_{\beta=130} = \begin{cases} 0.9, & \text{if } x \in [3.5,5] \\ 0, & \text{otherwise} \end{cases}$	$u^0_{\beta=200} = \begin{cases} 1.0, & \text{if } x \in [4.5,6] \\ 0.3, & \text{otherwise} \end{cases}$
$u^1_{\beta=200} = \begin{cases} 0.8, & \text{if } x \in [5,6.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^2_{\beta=200} = \begin{cases} 0.85, & \text{if } x \in [4,7] \\ 0, & \text{otherwise} \end{cases}$	$u^3_{\beta=200} = \begin{cases} 0.95, & \text{if } x \in [0,3] \\ 0, & \text{otherwise} \end{cases}$	$u^4_{\beta=200} = \begin{cases} 0.7, & \text{if } x \in [3,6] \\ 0, & \text{otherwise} \end{cases}$
$u^5_{\beta=200} = \begin{cases} 0.9, & \text{if } x \in [3.5,5] \\ 0, & \text{otherwise} \end{cases}$	$u^0_{\beta=300} = \begin{cases} 1.0, & \text{if } x \in [4.5,6] \\ 0.3, & \text{otherwise} \end{cases}$	$u^1_{\beta=300} = \begin{cases} 0.8, & \text{if } x \in [5,6.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^2_{\beta=300} = \begin{cases} 0.85, & \text{if } x \in [4,7] \\ 0, & \text{otherwise} \end{cases}$
$u^3_{\beta=300} = \begin{cases} 0.95, & \text{if } x \in [0,3] \\ 0, & \text{otherwise} \end{cases}$	$u^4_{\beta=300} = \begin{cases} 0.7, & \text{if } x \in [3,6] \\ 0, & \text{otherwise} \end{cases}$	$u^5_{\beta=300} = \begin{cases} 0.9, & \text{if } x \in [3.5,5] \\ 0, & \text{otherwise} \end{cases}$	$u^6_{\beta=300} = \begin{cases} 0.6, & \text{if } x \in [3.5,5] \\ 0, & \text{otherwise} \end{cases}$
$u^7_{\beta=300} = \begin{cases} 0.4, & \text{if } x \in [3.5,5] \\ 0, & \text{otherwise} \end{cases}$	$u^8_{\beta=300} = \begin{cases} 0.35, & \text{if } x \in [3.5,5] \\ 0, & \text{otherwise} \end{cases}$		

Table 8
The initial states used to train $f(u;\beta)$ in Section 4.1.

$u^0_{\beta=-1} = \begin{cases} 0.8, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^1_{\beta=-1} = \begin{cases} 1.0, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^2_{\beta=-1} = \begin{cases} 0.9, & \text{if } x \in [3,5] \\ 0, & \text{otherwise} \end{cases}$	$u^3_{\beta=-1} = \begin{cases} 0.95, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$
$u^4_{\beta=-1} = \begin{cases} 0.85, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$	$u^0_{\beta=1} = \begin{cases} 0.8, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^1_{\beta=1} = \begin{cases} 1.0, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^2_{\beta=1} = \begin{cases} 0.9, & \text{if } x \in [3,5] \\ 0, & \text{otherwise} \end{cases}$
$u^3_{\beta=1} = \begin{cases} 0.7, & \text{if } x \in [2.5,4.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^4_{\beta=1} = \begin{cases} 0.6, & \text{if } x \in [2.5,4.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^5_{\beta=1} = \begin{cases} 0.5, & \text{if } x \in [2.5,4.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^0_{\beta=3} = \begin{cases} 0.8, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$
$u^1_{\beta=3} = \begin{cases} 1.0, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^2_{\beta=3} = \begin{cases} 0.9, & \text{if } x \in [3,5] \\ 0, & \text{otherwise} \end{cases}$	$u^3_{\beta=3} = \begin{cases} 0.95, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$	$u^4_{\beta=3} = \begin{cases} 0.85, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$
$u^5_{\beta=3} = \begin{cases} 1.0, & \text{if } x \in [2.5,4.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^6_{\beta=3} = \begin{cases} 0.9, & \text{if } x \in [2.5,4.5] \\ 0.2, & \text{otherwise} \end{cases}$	$u^0_{\beta=5} = \begin{cases} 0.8, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^1_{\beta=5} = \begin{cases} 1.0, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$
$u^2_{\beta=5} = \begin{cases} 0.9, & \text{if } x \in [3,5] \\ 0, & \text{otherwise} \end{cases}$	$u^3_{\beta=5} = \begin{cases} 0.95, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$	$u^4_{\beta=5} = \begin{cases} 0.85, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$	$u^0_{\beta=7} = \begin{cases} 0.8, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$
$u^1_{\beta=7} = \begin{cases} 1.0, & \text{if } x \in [4,6] \\ 0, & \text{otherwise} \end{cases}$	$u^2_{\beta=7} = \begin{cases} 0.9, & \text{if } x \in [3,5] \\ 0, & \text{otherwise} \end{cases}$	$u^3_{\beta=7} = \begin{cases} 0.95, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$	$u^4_{\beta=7} = \begin{cases} 0.85, & \text{if } x \in [2.5,4.5] \\ 0.3, & \text{otherwise} \end{cases}$

References

[1] D. Pardo, L. Demkowicz, C. Torres-Verdin, M. Paszynski, A self-adaptive goal-oriented hp-finite element method with electromagnetic applications. Part II: Electrodynamics, Comput. Methods Appl. Mech. Engrg. 196 (37–40) (2007) 3585–3597.

[2] Thomas D Economon, Francisco Palacios, Sean R Copeland, Trent W Lukaczyk, Juan J Alonso, SU2: An open-source suite for multiphysics simulation and design, AIAA J. 54 (3) (2016) 828–846.

[3] Ravi Ramamurti, William Sandberg, Simulation of flow about flapping airfoils using finite element incompressible flow solver, AIAA J. 39 (2) (2001) 253–260.

[4] Peter Bauer, Alan Thorpe, Gilbert Brunet, The quiet revolution of numerical weather prediction, Nature 525 (7567) (2015) 47–55.

[5] Christoph Schwarzbach, Ralph-Uwe Börner, Klaus Spitzer, Three-dimensional adaptive higher order finite element simulation for geo-electromagnetics—a marine CSEM example, Geophys. J. Int. 187 (1) (2011) 63–74.

[6] Maziar Raissi, Paris Perdikaris, George 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 (2019) 686–707.

[7] Zichao Long, Yiping Lu, Xianzhong Ma, Bin Dong, Pde-net: Learning PDEs from data, in: International Conference on Machine Learning, PMLR, 2018, pp. 3208–3216.

[8] Zichao Long, Yiping Lu, Bin Dong, PDE-Net 2.0: Learning PDEs from data with a numeric-symbolic hybrid deep network, J. Comput. Phys. 399 (2019) 108925.

[9] Hans Joakim Skadsem, Steinar Kragset, A numerical study of density-unstable reverse circulation displacement for primary cementing, . Energy Resour. Technol. 144 (12) (2022) 123008.

[10] Olga Fuks, Hamdi A. Tchelepi, Limitations of physics informed machine learning for nonlinear two-phase transport in porous media, J. Mach. Learn. Model. Comput. 1 (1) (2020).

[11] Qing Li, Steinar Evje, Learning the nonlinear flux function of a hidden scalar conservation law from data, Netw. Heterog. Media 18 (2023).

[12] R.J. LeVeque, Finite volume methods for hyperbolic problems, in: Cambridge Texts in Applied Mathematics, 2007.

[13] J.W. Thomas, Numerical partial differential equations. conservation laws and elliptic equations, in: Texts in Applied Mathematics 33, 1999.

[14] J.S. Hesthaven, Numerical methods for conservation laws. From analysis to algorithms, SIAM. Comput. Sci. Eng. (2017).

[15] D. Kröener, Numerical schemes for conservation laws, in: Wiley-Teubner Series Advances in Numerical Mathematics, 1997.

[16] Siddhartha Mishra, U. Fjordholm, R. Abgrall, Numerical methods for conservation laws and related equations, in: Lecture Notes for Numerical Methods for Partial Differential Equations, ETH, vol. 57, 2019, p. 58.

[17] Valerii Iakovlev, Markus Heinonen, Harri Lähdesmäki, Learning continuous-time PDEs from sparse data with graph neural networks, in: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, 2021.

[18] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, Peter W. Battaglia, Learning mesh-based simulation with graph networks, in: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, 2021.

[19] Qingqing Zhao, David B. Lindell, Gordon Wetzstein, Learning to solve PDE-constrained inverse problems with graph networks, in: Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, Sivan Sabato (Eds.), International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA, in: Proceedings of Machine Learning Research, vol. 162, PMLR, 2022, pp. 26895–26910.

[20] Helge Holden, Fabio Simone Priuli, Nils Henrik Risebro, On an inverse problem for scalar conservation laws, Inverse Problems 30 (3) (2014) 035015.

[21] María Cristina Bustos, EM Tory, Raimund Bürger, F Concha, Sedimentation and thickening: Phenomenological foundation and mathematical theory, Vol. 8, Springer Science & Business Media, 1999.

[22] Stefan Diehl, Estimation of the batch-settling flux function for an ideal suspension from only two experiments, Chem. Eng. Sci. 62 (17) (2007) 4589–4601.

[23] Raimund Bürger, Stefan Diehl, Convexity-preserving flux identification for scalar conservation laws modelling sedimentation, Inverse Problems 29 (4) (2013) 045008.

[24] Raimund Bürger, Julio Careaga, Stefan Diehl, Flux identification of scalar conservation laws from sedimentation in a cone, IMA J. Appl. Math. 83 (3) (2018) 526–552.

[25] Stefan Diehl, Numerical identification of constitutive functions in scalar nonlinear convection–diffusion equations with application to batch sedimentation, Appl. Numer. Math. 95 (2015) 154–172.

[26] Lukas Mosser, Olivier Dubrule, Martin J. Blunt, Stochastic seismic waveform inversion using generative adversarial networks as a geological prior, Math. Geosci. 52 (1) (2020) 53–79.

[27] Qinglong He, Yanfei Wang, Reparameterized full-waveform inversion using deep neural networks, Geophysics 86 (1) (2021) V1–V13.

[28] Tiffany Fan, Kailai Xu, Jay Pathak, Eric Darve, Solving inverse problems in steady-state navier-stokes equations using deep neural networks, 2020, arXiv preprint arXiv:2008.13074.

[29] Hayden Schaeffer, Learning partial differential equations via data discovery and sparse optimization, Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci. 473 (2197) (2017) 20160446.

[30] Sung Ha Kang, Wenjing Liao, Yingjie Liu, Ident: Identifying differential equations with numerical time evolution, J. Sci. Comput. 87 (2021) 1–27.

[31] Nils Thuerey, Konstantin Weißenow, Lukas Prantl, Xiangyu Hu, Deep learning methods for Reynolds-averaged Navier–Stokes simulations of airfoil flows, AIAA J. 58 (1) (2020) 25–36.

[32] Nils Wandel, Michael Weinmann, Reinhard Klein, Learning incompressible fluid dynamics from scratch - towards fast, differentiable fluid models that generalize, in: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net, 2021.

[33] Siddhartha Mishra, A machine learning framework for data driven acceleration of computations of differential equations, Mathematics in Engineering 1 (2019) 118–146.

[34] Georg Martius, Christoph H. Lampert, Extrapolation and learning equations, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings, OpenReview.net, 2017.

[35] Subham Sahoo, Christoph Lampert, Georg Martius, Learning equations for extrapolation and control, in: International Conference on Machine Learning, PMLR, 2018, pp. 4442–4450.

[36] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, Jorge Nocedal, Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization, ACM Trans. Math. Softw. 23 (4) (1997) 550–560.

[37] Sebastian Ruder, An overview of gradient descent optimization algorithms, 2016, arXiv preprint arXiv:1609.04747.

[38] François James, M. Sepúlveda, Parameter identification for a model of chromatographic column, Inverse Problems 10 (6) (1994) 1299.

[39] François James, Mauricio Sepúlveda, Convergence results for the flux identification in a scalar conservation law, SIAM J. Control Optim. 37 (3) (1999) 869–891.

[40] Helge Holden, Nils Henrik Risebro, Front Tracking for Hyperbolic Conservation Laws, Vol. 152, Springer, 2015.

[41] Ilya Loshchilov, Frank Hutter, Decoupled weight decay regularization, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, la, USA, May 6-9, 2019, OpenReview.net, 2019.

[42] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, George E Dahl, Neural message passing for quantum chemistry, in: International Conference on Machine Learning, PMLR, 2017, pp. 1263–1272.

[43] Martin Riedmiller, Heinrich Braun, Rprop-a fast adaptive learning algorithm, in: Proc. of ISCIS VII, Universitat, Citeseer, 1992.