http://www.aimspress.com/journal/mine

Mathematics in Engineering, 1(1): 118–146
DOI:10.3934/Mine.2018.1.118
Received: 25 July 2018
Accepted: 03 September 2018
Published: 10 October 2018

Research article

A machine learning framework for data driven acceleration of computations of differential equations

Siddhartha Mishra[*]

Seminar for Applied Mathematics (SAM), D-Math, ETH Zürich, Rämistrasse 101, Zürich-8092, Switzerland

[*] **Correspondence:** Email: smishra@sam.math.ethz.ch.

Abstract: We propose a machine learning framework to accelerate numerical computations of time-dependent ODEs and PDEs. Our method is based on recasting (generalizations of) existing numerical methods as artificial neural networks, with a set of trainable parameters. These parameters are determined in an offline training process by (approximately) minimizing suitable (possibly non-convex) loss functions by (stochastic) gradient descent methods. The proposed algorithm is designed to be always consistent with the underlying differential equation. Numerical experiments involving both linear and non-linear ODE and PDE model problems demonstrate a significant gain in computational efficiency over standard numerical methods.

Keywords: machine learning; deep learning; differential equations; non-convex optimization; time-dependent problems

1. Introduction

Differential equations, both ordinary and partial, are ubiquitous in science and engineering. It is not possible to obtain explicit solution formulas for differential equations, except in the simplest cases. Hence, numerical approximations of differential equations constitutes a key tool in their study. A wide variety of numerical methods have been developed to approximate differential equations robustly and efficiently. For the initial value problem for ordinary differential equations, popular methods include Runge-Kutta and multi-step methods, [12, 16] and references therein. Widely used numerical methods for approximating PDEs include finite difference [16], finite volume [10], finite element [5] and spectral [27] methods.

The exponential increase in computational power in the last decades provides the opportunity for solving very challenging, large scale computational problems for differential equations, such as uncertainty quantification (UQ) [3, 9], (Bayesian) inverse problems [24] and (real time) optimal control, design and constrained optimization [4, 28]. One requires very large number of fast approximations of ODE and PDEs to solve such problems, for instance when evaluating Monte Carlo samples in an UQ or Bayesian inverse problem framework. Currently available numerical methods, particularly for nonlinear PDEs, tend to be too slow to allow such realistic computations.

Machine learning, in the form of artificial neural networks (ANNs), has become extremely popular in computer science in recent years. This term is applied to a plethora of methods that aim to approximate functions with layers of units (neurons), connected by linear operations between units and nonlinear activations within units, [11] and references therein. *Deep learning*, i.e an artificial neural network with a large number of intermediate (hidden) layers has proven extremely successful at diverse tasks, for instance in image processing, signal processing and natural language processing [15]. A key element in deep learning is the *training* of tuning parameters in the underlying neural network by (approximately) minimizing suitable *loss functions*. The resulting (non-convex) optimization problem, on a very high dimensional parameter space, can be efficiently solved with variants of the stochastic gradient descent method [14, 22].

Machine learning methods, particularly deep learning, are being increasingly used in the context of numerical computation of differential equations. As this is a rapidly evolving field, we will only attempt a skeletal literature survey here. One class of methods attempt to replace numerical schemes for differential equations by deep networks, see [2, 7, 20, 18] and references therein. These methods have been successfully used in different contexts, for instance in approximating very high dimensional problems arising in mathematical finance [2], by exploiting integral representation formulas for the underlying solutions. However, it is as yet unclear if an end to end deep neural network can learn the physics of the underlying PDE in the absence of such formulas. This could constitute a stumbling block in approximating solutions of complicated nonlinear PDEs with deep learning.

Another school of thought aims to augment existing numerical methods by embedding deep learning modules within them. As examples, one can think of solving the pressure Poisson equation within an incompressible flow solver by a convolutional neural network as in [26] or learning troubled cell indicators in a RKDG code by a deep network as in [21].

In this paper, we propose a variant of the machine learning framework for approximating (time-dependent) differential equations. Our starting point is the observation that evaluating approximate solutions of ODEs and PDEs on coarse space-time grids is very cheap computationally. However, the accuracy of such coarse grid representations is rather poor. Consequently, we will use a machine learning framework to train explicit or implicit parameters in (generalizations of) standard numerical methods in order to minimize a loss (error) function that measures the difference of the (trained) solution on coarse grids with projections of fine grid solutions. The resulting scheme will hopefully be significantly more accurate than the underlying standard method on the coarse grid, while being as computationally cheap. We motivate our general strategy with the following simple example.

1.1. A motivating example

We consider the following autonomous ODE,

$$\begin{aligned} u'(t) &= F(u(t)), \quad t \in (0,T), \\ u(0) &= u_0. \end{aligned} \tag{1.1}$$

Here, $u : [0, T] \to \mathbb{R}^d$ is the vector of unknowns and $F \in \mathrm{Lip}_{\mathrm{loc}}(\mathbb{R}^d)$ is the right hand side.

A very popular class of numerical methods to solve (1.1), particularly for *stiff* right hand sides, are the so-called implicit multi-step methods or backward difference formulas (BDFs) [12]. Assuming a uniform time step Δt, denoting the time level by $t_n = n\Delta t$ and the approximate solution by $U_n \approx u(t_n)$, a general form of a three-point BDF is given by,

$$(1 + g_{n+2})U_{n+2} - (1 + 2g_{n+2})U_{n+1} + g_{n+2}U_n = \Delta t F(U_{n+2}), \tag{1.2}$$

for $n \geqslant 0$ and $g_{n+2} \in \mathbb{R}$ for each n. We need initial values U_0, U_1 to march in time in (1.2). It is straightforward to check using Taylor expansions that for any $g_{n+2} \in \mathbb{R}$, the method (1.2) is consistent with (1.1) and at least first-order accurate. By setting $g_{n+2} = 0$, $\forall n$, we recover the standard backward Euler method while $g_{n+2} = \frac{1}{2}$ for all n yields the second-order accurate BDF2 method. It is customary to assume that a second-order accurate method is preferable. Hence, one always sets $g_{n+2} \equiv 0.5$.

However, this point of view does not take the data of the specific problem that we are solving into account, namely the non-linearity F, the dimension d, the initial data u_0 and the grid size Δt. It is not a priori obvious if the choice of $g_{n+2} = 0.5$ will provide the best solution (the least error) for a given data set. In fact, this is not true in general. As an example, we consider the simplest linear scalar ODE by setting $d = 1$ and $F(u) = -cu$, for some $c \in \mathbb{R}_+$, in (1.1). In this case, the explicit solution is given by

$$u(t) = u_0 e^{-ct}. \tag{1.3}$$

Now setting $U_0 = u_0$, $U_1 = e^{-c\Delta t}u_0$, the solution computed at the second time level $2\Delta t$, by the generalized form of the three point BDF scheme (1.2) is

$$U_2 = \frac{1 + 2g_2}{1 + g_2 + c\Delta t}U_1 - \frac{g_2}{1 + g_2 + c\Delta t}U_0. \tag{1.4}$$

Hence, the local error at the second time level is

$$E_2(g_2) = |U_2 - e^{-2c\Delta t}u_0|^2, \tag{1.5}$$

It is straightforward to observe that the minimizer,

$$g_2^* = \arg\min_{g_2 \in \mathbb{R}} E_2(g_2),$$

depends explicitly on the parameters Δt and c and is not universally $g^* = 0.5$. In Figure 1, we plot the function $E_2(g_2)$ for $\Delta t = 0.5$ and two different values of c, namely $c = 1$ and $c = 5$, respectively. We see from this figure that the loss (error) function is convex in the parameter g_2 and the error is vastly reduced in a minimization process, i.e, by a factor of 39.97 for $c = 1$ and a factor of 677.95 for $c = 5$, respectively. Hence, by focusing on a specific data set, we can potentially obtain speed ups of two to three orders of magnitude for this simple ODE.

1.2. Aims and scope of this paper

Our objective is to generalize the strategy presented in the above motivating example. We will recast (generalizations of) standard numerical methods for time-dependent ODEs and PDEs as *multi-layer* artificial neural networks with a set of trainable parameters. The resulting network will always be

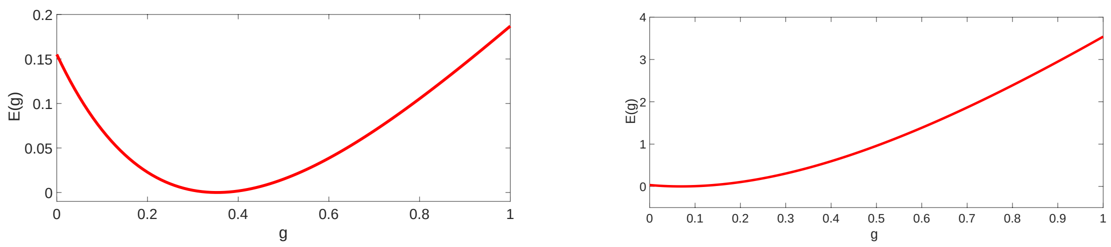

Figure 1. Error $E_2(g_2)$ (1.5) (Y-axis) vs. the scheme parameter g_2 (X-axis) in the generalized BDF scheme (1.4) for $\Delta t = 0.5$ and two different values of c. Left $c = 1$, the minimum in the error is achieved at $g_2 = 0.35$ and yields a factor of 39.97 times reduction in the error over the standard three-point BDF2 scheme i.e (1.2) with $g = 0.5$. Right: $c = 5$: The minimum is at $g_2 = 0.07$ and the error is reduced by a factor of 677.95 over the standard BDF2 scheme.

designed to be consistent with the underlying differential equation by constraining the parameter set. During an offline training phase, we will train these parameters by (approximately) minimizing a loss function, over the parameter states, by a suitable (stochastic) gradient descent method. The efficiency of the resulting trained scheme is verified on a *test set*. Thus, our methods will be (rather restricted) types of (deep) neural networks for approximating time-dependent differential equations.

We organize the rest of the paper as follows: in section 2, we present our abstract machine learning framework. In section 3, we apply the proposed algorithm to ordinary differential equations. The machine learning algorithm is applied to the heat equation, linear transport equation, scalar conservation laws and the Euler equations of gas dynamics in sections 4, 5, 6 and 7. We summarize the contents of the paper and provide a perspective in section 8.

2. The abstract machine learning framework

For definiteness, we consider a one-dimensional nonlinear time-dependent PDE of the form,

$$\begin{aligned} u_t &= L(u, u_x, u_{xx}), \quad (x,t) \in [X_l, X_r] \times [0,T], \\ u(0,x) &= u_0(x,\omega), \\ L_b u(X_l,t) &= u_l(t,\gamma_l), \quad L_b u(X_r,t) = u_r(t,\gamma_r). \end{aligned} \tag{2.1}$$

Here, $u : [X_l, X_r] \times [0,T] \to \mathbb{R}^m$ is the vector of unknowns. L is a possibly non-linear differential operator involving both the first and second spatial derivatives of u (interpreted as vectors) that will be specified in subsequent examples but is kept deliberately ambiguous here for the sake of generality. L_b refers to a *boundary* operator and the initial and boundary data depend on parameters $\omega, \gamma_{l,r} \in \mathbb{R}^D$ for possibly $D >> 1$.

For simplicity, we discretize $[X_l, X_r]$ on a uniform grid with mesh size Δx and denote the discrete points as $x_j = X_l + jDx$, for $0 \leqslant j \leqslant J+1$. Similarly, we choose a uniform time step Δt and denote the n-th time level as $t^n = n\Delta t$, with $T = N\Delta t$, and denote the approximate solution (by a finite difference scheme as) as $U_j^n \approx u(x_j, t^n)$. Moreover, one can readily consider a finite volume or finite element discretization by letting U_j^n approximate cell averages or nodal point values, respectively. We denote the vector, $U^n = \{U_j^n\}_{1\leqslant j \leqslant N}$.

On this grid, we discretize the abstract PDE (2.1) with the following numerical method,

$$U^{n+1} = L^n(\Delta t, \Delta x, U^n, A^n U^n, F(U_n)), R(U^n)). \tag{2.2}$$

Several remarks are in order about the form of the abstract scheme (2.2). First, A^n is a linear operator that subsumes potentially multiple applications of matrices. Second, the function F denotes a composite of nonlinearities that occur within the non-linear operator L in (2.1). Furthermore, we write (2.2) as a one-step explicit scheme. However, we implicitly assume that even if the underlying time discretization was (semi-)implicit, the resulting system of nonlinear equations is solved by an iterative procedure, such as a Newton method, and the results of the Newton steps are subsumed in the general form of the right hand side term L^n in (2.2). In particular, such iterations might involve multiple applications of the non-linearity F in (2.2) and could involve the function R in (2.2) that might include (multiple compositions of) the standard ReLU function,

$$\sigma(w) = \max(w, 0). \tag{2.3}$$

The whole method can be represented as a *neural network* as shown in Figure 2, but with non-linearities F, based on the underlying PDE instead of scalar activation functions as in a traditional deep network [11]. However, given the universal approximation property of standard artificial neural networks [1, 13], one can approximate the underlying nonlinearities F in (2.2) by artificial neural networks. In particular, networks with a few hidden layers can be trained to approximate smooth nonlinear functions [30]. Hence, one can think of the module F in (2.4), Figure 2 as an *additional neural network*, realizing the whole scheme (2.2) as a neural network in the sense of [11]. However, for the sake of consistency and computational efficiency, we perform a direct evaluation of the nonlinearity F in (2.2) in this paper. A very concrete realization of the neural network representation for numerical schemes is provided in section 6 for a Rusanov type scheme approximating scalar conservation laws, see Figure 7.

We constrain the numerical method (or alternatively the artificial neural network) (2.2) to be consistent with the PDE (2.1) by imposing constraints on the linear operator A^n (and the structure of the neural network approximating F). Further stability conditions on the scheme can also be imposed. Consequently, we can rewrite the numerical method (2.2) in the following parametric form,

$$U^{n+1} = L^n(\Delta t, \Delta x, U^n, \theta^n), \tag{2.4}$$

in terms of a parameter vector $\theta^n \in \mathbb{R}^d$. We impose the constraints on the scheme such that by choosing $\theta^n = \overline{\theta}^n$, leads to the recovery of standard numerical methods, for instance the choice of $g = 0.5$ in the scheme (1.2) yields the standard second-order three-point BDF2 scheme for discretizing the ODE (1.1).

For generating the training set, we select a certain subset of the parameter space $\{\omega_i, \gamma_{1,i}, \gamma_{2,i}\}_{1 \leqslant i \leqslant M}$ where $M >> 1$ is the size of the training set. Let $\Delta t_f << \Delta t$ and $\Delta x_f << \Delta x$ be the time step and mesh size for a (very) fine grid (uniform) discretization of $[0, T] \times [X_l, X_r]$. For training data $\{u_0(\omega_i), u_l(\gamma_{1,i}), u_r(\gamma_{2,i})\}_i$, we approximate (2.1) with the scheme (2.4), with a particular choice of $\theta^n = \overline{\theta}^n$, on the fine grid. The resulting solution, denoted as $U^{n,i}_{\mathrm{ref}}$, is obtained by projecting the fine grid solution to the coarse grid, either with cell averaging or point wise sampling. This training data is generated offline and will be rather computationally expensive as a fine grid solution needs to be computed.

Next we set up a *training loss function* by defining the error,

$$E(\theta) := \frac{1}{p} \sum_i \sum_{n=1}^{N} \|U^{n,i}(\theta^n) - U^{n,i}_{\mathrm{ref}}\|^p_{L^p}. \tag{2.5}$$

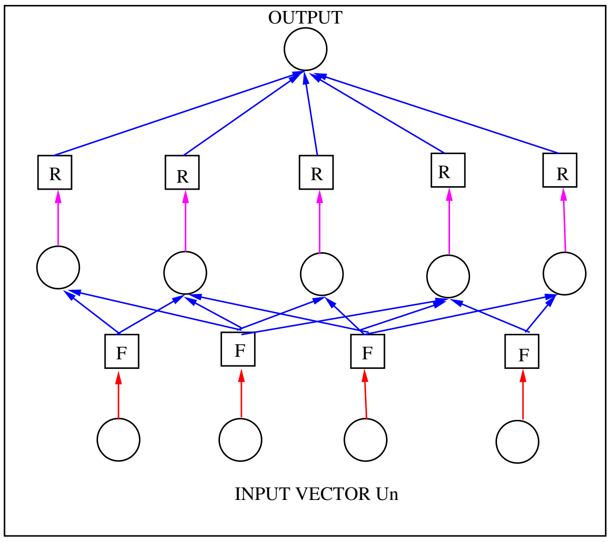

Figure 2. A simplified representation of a single time step of the abstract numerical scheme (2.2) as a *multi-layer neural network*. We show a part (4 components) of the solution vector U^n in (2.2) as the input. The blue arrows represent linear mappings (A^n) between units. The magenta arrows represent nonlinear (scalar) activations, for instance by the standard ReLU function (2.3) of units and the red arrows represent evaluation of nonlinearities F, that are inherent to the underlying differential equation (2.1). The hidden layers might correspond to steps of a Newton method. The output is a single component of the vector U^{n+1} in (2.2). Note that the function F can be replaced by an artificial neural network module to realize the whole scheme (2.2) as an artificial neural network in the sense of [11].

Here, $1 \leqslant p < \infty$ and we usually consider $p = 1$ or $p = 2$ and denote $\theta = \{\theta^n\}_{1 \leqslant n \leqslant N}$ as the combined vector of trainable parameters.

The objective of the training process is to minimize the loss function (2.5) i.e, find a minimizer θ^*:

$$\theta^* = \arg \min_{\theta \in \mathbb{R}^{Nd}} E(\theta). \tag{2.6}$$

The resulting *trained* time-marching scheme is

$$U^{n+1} = L^n(\Delta t, \Delta x, U^n, \theta^{n,*}). \tag{2.7}$$

We summarize the resulting algorithm for our machine learning framework below:

Algorithm 2.1. *Given a specific model i.e, underlying ODE or PDE, for instance* (2.1) *on a specific space-time domain, we*

Step 1**:** *Choose a consistent (and stable) numerical method (alternatively neural network) for approximating the underlying differential equation, for instance the general form* (2.2) *or* (2.4)*. This numerical method will approximate solutions of* (2.1) *on a coarse grid.*

Step 2**:** *Generate the* training set *by choosing a specific (finite but possibly large) parametric set of initial (and boundary conditions) and approximating the underlying PDE with a standard numerical method, for instance* (2.4) *but with the parameter set* $\overline{\theta}$*, on a (very) fine grid. Then, project the fine grid solutions on the coarse grid to create the training set.*

Step 3**:** *Set up the loss function* (2.5) *and use a gradient descent method to (approximately) find a local minimum of this possibly non-convex function over the parameter space. The gradient descent* *method can be initialized with the parameter values $\overline{\theta}$, corresponding to a standard numerical method.*

Step 4: *The minimizers θ^* serve as the parameters in the trained scheme* (2.7). *The trained scheme is run on a* test set *in the online phase, to ascertain gains in computational efficiency.*

We remark that the algorithm 2.1 is guaranteed to reduce the error, on the underlying coarse grid, over a standard scheme (i.e (2.4) with parameter set $\overline{\theta}$), on the training set. Moreover, the trained scheme (2.7) will always be a consistent (and stable) discretization of the underlying model. It is difficult to obtain theoretical guarantees on the amplitude of the gain in computational efficiency for our machine learning framework, on the *test set*. This gain depends on the underlying model, numerical method, grid size, choice of training set and on the efficacy of the gradient descent in finding a minimum. We will test this machine learning algorithm on a variety of problems in the following sections in order to empirically demonstrate its efficiency.

3. ODEs

In this section, we will test algorithm 2.1 on the following two ordinary differential equations,

3.1. A linear ODE

We start with the following second-order linear ODE modeling oscillators,

$$u''(t) + c^2 u(t) = 0, \quad u(0) = u_0. \tag{3.1}$$

We can readily write (3.1) as a first-order system by introducing the auxiliary variable $v = u'$, resulting in

$$u' = -cv, \quad v' = cu. \tag{3.2}$$

It is easy to see that (3.1) (equivalently (3.2)) has an explicit solution given by,

$$u(t) = u_0 \cos(ct), \quad v(t) = u_0 \sin(ct). \tag{3.3}$$

For the sake of simplicity, we choose the generalized three-point backward difference formula (1.2), with $U = [u, v]$ and $F(U) = [-cv, cu]$, as the underlying numerical scheme i.e step 1 of Algorithm 2.1. We choose a uniform grid in time with time step Δt and initialize the scheme (1.2) with $U_0 = [u_0, u_0]$ and $U_1 = [u_0 \cos(c\Delta t), u_0 \sin(c\Delta t)]$.

Our objective is to approximate the solution of (3.2) at the next two time levels, i.e determine U_2 and U_3 by (1.2) with $\Delta t = \frac{1}{3}$. As the constant c determines the frequency of oscillations in (3.1), a grid with a time step of $\Delta t = \frac{1}{3}$ is extremely coarse for large values of c, as several oscillations occur within a single time step. The task at hand is to determine whether the machine learning algorithm 2.1 can (significantly) improve the accuracy of the scheme (1.2) on such a coarse grid.

For step 2 of Algorithm 2.1, we generate the training set by randomly selecting I data points on the interval $[0, 1]$ and denoting them as $\{u_0^i\}$ with $1 \leqslant i \leqslant I = 10$. Corresponding to these data points, we use the exact solutions (3.3) at times $t_2 = \frac{2}{3}$ and $t_3 = 1$ to generate the reference training data $U_{\text{ref}}^{n,i}$ for all i and $n = 2, 3$.

In step 3 of Algorithm 2.1, we set up the l^2 error,

$$E_2(g_2, g_3) := \frac{1}{2}\sum_{i=1}^{I}\sum_{n=2,3} |U^{n,i}(g_n) - U^{l,i}_{\mathrm{ref}}|^2, \tag{3.4}$$

and minimize the loss function (3.4) over two parameters $g_{2,3} \in \mathbb{R} \times \mathbb{R}$. Given this simple two-dimensional (in parameters) problem, the minimization is performed by a standard steepest gradient descent initialized at the point $(g_2, g_3) = (0.5, 0.5)$, corresponding to the *second-order* BDF2 scheme.

Table 1. The performance of the trained three-point BDF scheme on the linear ODE (3.1) for three different values of the constant c. The gain in the fourth column is the ratio of the (mean) error (3.4) with the standard BDF2 method and the (mean) error with the trained scheme (1.2) with parameters $g^*_{2,3}$, on the test set.

c	g^*_2	g^*_3	Gain
1	0.1	1.4	4.13
10	−0.64	−2.04	2.11
100	11	0.02	13.03

The gradient descent algorithm converges quite quickly (atmost 9 steps) to a local minimum of the non-convex loss function. The minimizers $g^*_{2,3}$ are shown in Table 1 for three different values of $c = 1, 10$ and 100, and indicate a significant difference between the optimized values and the initial value of $(0.5, 0.5)$ (corresponding to the standard second-order BDF2 scheme).

We test the trained scheme i.e, (1.2) with parameters $g^*_{2,3}$ on a *test set*, chosen by randomly selecting 50 points in the interval $[-5, 5]$ as the initial data u_0 in (3.1). The mean gains in error i.e the ratio of the error with the standard BDF2 scheme and the error with the trained (data learned) scheme, with respect to the underlying exact solution, are presented in Table 1. The gain in efficiency with the trained scheme is considerable for all the three values of c, rising to at least an order of magnitude for $c = 100$. In this case, the value of u computed with the trained scheme is remarkably close to the exact solution at times $T = 2/3$ and $T = 1$. Moreover, the trained scheme even seems to outperform an explicit second-order Runge-Kutta method (see [16]) with a fine grid of $\Delta t = 0.001$, as observed in a plot of the approximate solutions on a particular realization of the test set in Figure 3.

3.2. *A non-linear ODE.*

We consider a simple but non-linear population (saturation) model for the time evolution of the population density u, described by the ODE,

$$u' = cu(1-u), \quad u(0) = u_0 \geqslant 0. \tag{3.5}$$

It is straightforward to check that the exact solution of (3.5) is given by,

$$u(t) = \frac{u_0}{u_0 + (1-u_0)e^{-ct}}. \tag{3.6}$$

Hence, any non-negative initial condition converges to a saturation value (stable equilibrium) at $u = 1$, at a time scale dictated by the constant c. We are interested in computing the solution in the time interval $[0, 1]$.

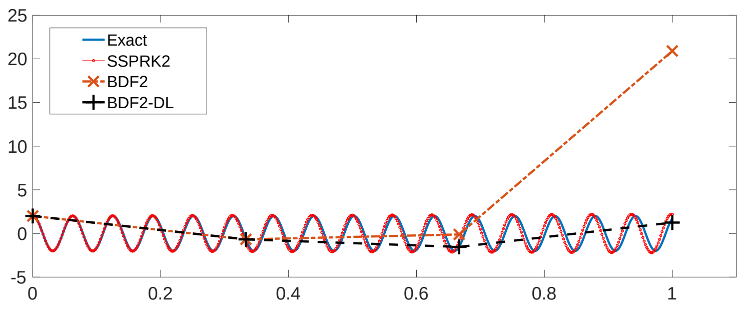

Figure 3. Solutions u of the linear ODE (3.1) (Y-axis) in time (X-axis), with $c = 100$ in time period $[0, 1]$ with an initial value from the *test set*. We compare the exact solution with a solution computed with a second-order explicit Runge-Kutta method (SSP-RK2) with 1000 time steps, the standard BDF2 method with $\Delta t = \frac{1}{3}$ and the trained scheme (labelled as BDF2-DL (data learned)), also with $\Delta t = \frac{1}{3}$. The trained scheme clearly outperforms the standard BDF2 method and more surprisingly, even the second-order accurate fine grid RK2 method.

For step 1 of Algorithm 2.1, we again choose the generalized three-point BDF scheme (1.2) and a *coarse* grid with time step $\Delta t = 0.5$. To generate the training set, we choose initial data $\{u_0^i\}_{1\leqslant i\leqslant I}$ with $I = 10$, uniformly over the interval $[0, 2]$ and set U_1^i as the exact solution (3.6) at time $t = \Delta t$. On this training set, we compute the approximate solution U_2 of (3.5) by the generalized three point BDF method (with a single trainable parameter g_2) at time $T = 2\Delta t = 1$. The exact solution $U_{\mathrm{ref}}^{2,i}$ is calculated by (3.6) at time $T = 1$ with the initial data u_0^i to define the *loss function*,

$$E_1(g_2) := \sum_{i=1}^{I} |U^{2,i}(g_2) - U_{\mathrm{ref}}^{2,i}|_1. \tag{3.7}$$

Compared to the previous example of a linear ODE, we choose the l^1 norm as the loss function in this nonlinear example.

Table 2. The performance of the trained three-point BDF scheme (1.2) on the non-linear ODE (3.5) on three different values of the constant c. The gain in the third column is the ratio of the (mean) error (3.7) with the standard BDF2 method and the (mean) error with the trained scheme (1.2) with parameter g_2^* on the test set.

c	g_2^*	Gain
0.2	0.41	1.5
1	0.16	3.02
5	0.03	10.54

The loss function for two different values of $c = 1$ and $c = 5$ is shown in Figure 4. In all cases that we tested, the loss function is convex and is readily minimized by a straightforward steepest descent algorithm. The resulting optimal parameter g_2^* for three different values of c is shown in Table 2. As seen in table 2 and Figure 4, the optimal value g^* is very different from $g = 0.5$.

The *test set* is constructed by randomly choosing 50 points from the interval $[0, 5]$ as the initial data and approximating (3.5) with the trained scheme, i.e (1.2) with parameter g_2^*. The corresponding error with respect to the exact solution is calculated and the *gain*, defined as before, is shown in Table 2 . We observe a consistent gain in computational efficiency with the trained scheme that is approximately one order of magnitude for $c = 5$. Thus, the machine learning algorithm 2.1 performs well in this nonlinear example and the gains in efficiency are similar to the linear problem even though a different loss function was used.

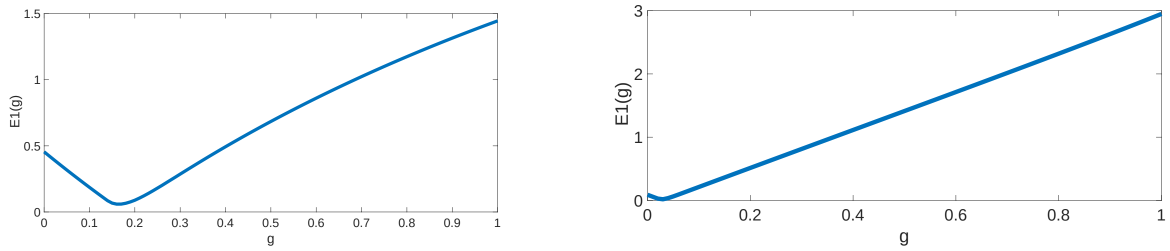

Figure 4. Error $E_1(g)$ (3.7) (Y-axis) vs. the scheme parameter g (X-axis) in the generalized BDF scheme (1.2) for $\Delta t = 0.5$ and two different values of c. Let $c = 1$, the minimum in the error is achieved at $g = 0.16$ Right: $c = 5$: The minimum is at $g = 0.03$.

4. Heat equation

As the first example for PDEs, we consider the heat equation in one space dimension,

$$\begin{aligned} u_t &= c u_{xx}, \quad (x,t) \in (0,1) \times (0,T), \\ u(x,0) &= u_0(x), \quad x \in (0,1), \\ u(0,t) &= u(1,t) = 0, \quad t \in (0,T). \end{aligned} \tag{4.1}$$

Here, u is the temperature and $0 < c \in \mathbb{R}$ is a diffusion coefficient.

4.1. Numerical scheme

We discretize the interval $[0,1]$ uniformly with a grid size Δx and label the resulting points as $x_j = j\Delta x$ for $0 \leqslant j \leqslant J+1$, with $\Delta x = \frac{1}{J+1}$. The time interval $[0,T]$ is discretized uniformly with a time step Δt and the time points are labeled as $t^n = n\Delta t$, with $0 \leqslant n \leqslant N$ and $\Delta t = T/N$. We approximate the heat equation (4.1) by evolving $U_j^n \approx u(x_j, t^n)$, with the following generalized (or weighted) five-point finite difference scheme,

$$\begin{aligned} \frac{U_j^{n+1} - U_j^n}{\Delta t} &= \frac{c(1-g^n)}{\Delta x^2}\left(b_{-2}^n U_{j-2}^{n+1} + b_{-1}^n U_{j-1}^{n+1} + b_0^n U_j^{n+1} + b_1^n U_{j+1}^{n+1} + b_2^n U_{j+2}^{n+1}\right) \\ &+ \frac{c g^n}{\Delta x^2}\left(b_{-2}^n U_{j-2}^n + b_{-1}^n U_{j-1}^n + b_0^n U_j^n + b_1^n U_{j+1}^n + b_2^n U_{j+2}^n\right), \quad \forall n, 2 \leqslant j \leqslant J-1. \end{aligned} \tag{4.2}$$

The update formulas for the points U_1^n and U_J^n is computed by setting the Dirichlet boundary conditions $U_{-1,0,J+1,J+2}^n \equiv 0$, for all n.

By using Taylor expansions, one can readily prove the following lemma,

Lemma 4.1. *For all n and any $g^n \in \mathbb{R}$, the finite difference scheme* (4.2) *is a consistent and first-order accurate discretization of the one-dimensional heat equation* (4.1) *if and only if the coefficients b_k^n, for*

$-2 \leqslant k \leqslant 2$, *satisfy the following algebraic conditions,*

$$\begin{aligned} b_2^n + b_1^n + b_0^n + b_{-1}^n + b_{-2}^n &= 0, \\ 2b_2^n + b_1^n - b_{-1}^n - 2b_{-2}^n &= 0, \\ 2b_2^n + \frac{b_1^n}{2} + \frac{b_{-1}^n}{2} + 2b_{-2}^n &= 0. \end{aligned} \tag{4.3}$$

Here, consistency and accuracy are defined in terms of the local truncation error [16]. As (4.3) has three equations containing five unknowns, we can eliminate three of them in terms of $b_{-2,-1}^n$ to obtain,

$$b_0^n = 1 - 3b_{-1}^n - 6b^n{-2}, \quad b_1^n = 3b_{-1}^n + 8b_{-2}^n - 2, \quad b_2^n = 1 - b_{-1}^n - 3b_{-2}^n. \tag{4.4}$$

Hence, per time level, the scheme (4.2) contains three undetermined parameters g^n, b_{-1}^n and b_{-2}^n. These parameters will be determined by the training process, i.e, Step 2 of Algorithm 2.1.

Remark 4.2. *Lemma 4.1 provides sufficient conditions for consistency of the finite difference scheme* (4.2). *Moreover, this scheme is* conservative *i.e,* $\sum_j U_j^{n+1} = \sum_j U_j^n$. *We can also obtain stability, for instance energy (L^2) stability or discrete maximum principles. These require additional constraints on the parameters and may constrain the training process further. We do not consider this aspect in the following.*

Although the form (4.2) of a two time-level, five point finite difference scheme is non-standard, it embeds several well-known finite difference approximations, namely

Scheme S1: Backward Euler in time and second-order accurate in space by setting $g^n = 0, b_{-2}^n = 0, b_{-1}^n = 1$ for all n.

Scheme S2: Crank-Nicolson in time and second-order accurate in space by setting $g^n = 0.5, b_{-2}^n = 0, b_{-1}^n = 1$ for all n.

Scheme S3: Backward Euler in time and fourth-order accurate in space by setting $g^n = 0, b_{-2}^n = -\frac{1}{12}, b_{-1}^n = \frac{4}{3}$ for all n.

Scheme S4: Crank-Nicolson in time and fourth-order accurate in space by setting $g^n = 0.5, b_{-2}^n = -\frac{1}{12}, b_{-1}^n = \frac{4}{3}$ for all n.

One can also set $g^n = 1$ to recover an explicit forward Euler time discretization. However, we focus on implicit time stepping methods in order to avoid the constraint of the severe CFL restriction for explicit time discretizations of the heat equation.

4.2. Training and Results on the test set

We approximate the solution u of the heat equation with scheme (4.2) at time $T = 0.05$, for three different values of the diffusion coefficient c, namely $c = 0.1, 1, 10$ ranging from slow to fast diffusion. All the experiments will be performed on a spatial grid with mesh size $\Delta x = \frac{1}{10}$ i.e, with 10 mesh points. Moreover, the time grid will based on a *single* very large time step of $\Delta t = 0.05$.

We focus on varying the initial datum u_0 in (4.1) to generate the *training set*. However, in contrast to ODEs, the initial datum u_0 for a PDE lies in an infinite dimensional function space, for instance $u_0 \in L^2((0, 1))$. Given the challenge of approximating the resulting data to solution operator, in infinite dimensions, we focus on particular classes of initial datum, defined in terms of (finite dimensional) parameters. Motivated by applications in uncertainty quantification [3] and reduced order modeling [29], we concentrate on the following specific parametric random initial data,

4.2.1. Smooth data

We consider the following L-term *Karhunen-Loeve expansion*,

$$u_0(x,\omega) = \sum_{l=1}^{L} \lambda_l Y_l(\omega) \sin(l\pi x), \tag{4.5}$$

with $L = 3$, $\lambda_l = \frac{1}{2^{l-1}}$ and the random numbers $Y_l(\omega)$ chosen from a uniform distribution on $[0,1]$. The training set is chosen by selecting (at random) I draws of the random variables $Y_l^i(\omega)$ with $1 \leqslant i \leqslant I = 20$. We compute a reference solution, for the resulting initial data u_0^i, with an explicit forward Euler time stepping and standard second-order spatial finite difference discretization [16] on a very fine grid of 1000 mesh points and a time step, chosen to satisfy the standard CFL requirement for the heat equation. This fine grid solution is projected onto the underlying coarse grid by sampling this solution at points x_j and at final time $T = \Delta t = 0.05$, $1 \leqslant j \leqslant J$. We denote this reference solution as $U_{j,\mathrm{ref}}^{n,i}$.

The *loss function* is defined as the L^2 error,

$$E_2\left(g^1, b_{-2}^1, b_{-1}^1\right) := \frac{\Delta x}{2} \sum_{i=1}^{I} \sum_{j=1}^{J} |U_j^{1,i} - U_{j,\mathrm{ref}}^{1,i}|^2. \tag{4.6}$$

The loss function (4.6) is minimized using a simplified version of the stochastic gradient algorithm [22] with a batch size of 4, initialized with the starting values of $g^1 = 0.5$, $b_{-2}^1 = 0$ and $b_{-1}^1 = 1$, corresponding to the overall second-order Crank-Nicolson type scheme S2. We denote the (approximate) minimizers as $\{g^{1,*}, b_{-2}^{1,*}, b_{-1}^{1,*}\}$ and the trained (*data learned*) scheme is the finite difference (4.2) with these parameters.

The training (for three different cases of the diffusion coefficient c considered here) resulted in (local) minimizers shown in Table 3. We observe that in most cases, the trained scheme is very different from any standard scheme. A *test set* is generated by choosing 100 random values of Y_l, $l = 1, 2, 3$ in (4.5). Care is taken to exclude repetition of values from the training set and the corresponding reference solution is computed, analogously to the training set.

Summarizing these results, we first observe that there is no clear winner among the standard schemes on the test set . For slow to moderate values of the diffusion coefficient, the scheme S2 i.e, Crank-Nicolson in time and second-order in space scores over the other three schemes whereas for a large value of the diffusion coefficient, the scheme S1 and S3 clearly perform the best. On the other hand, there is a large gain with the trained scheme compared to all the standard schemes. The gain of approximately 4 is most modest for the $c = 1$ value of the diffusion coefficient. On the other hand, there is clearly a gain of a factor of at least 10 or 20 for the extreme values of the diffusion coefficient. This gain in accuracy comes at no additional online cost and justifies the efficacy of the proposed machine learning algorithm. This significant gain in performance with the trained scheme, for one particular instance of the test set, is also displayed in Figure 5.

4.2.2. Rough data

Next, we consider the following discontinuous random initial data,

$$u_0(x,\omega) = \begin{cases} 1 + \varepsilon Y_1(\omega), & \text{if } \frac{1}{3} + \varepsilon Y_2(\omega) < x < \frac{2}{3} + \varepsilon Y_3(\omega), \\ 0, & \text{otherwise,} \end{cases} \tag{4.7}$$

Table 3. The performance of the trained finite difference scheme (4.2) on the heat equation (4.1) for three different values of the constant c and for the smooth initial data (4.5). The gain-m is the ratio of the (mean) error with the standard S-m scheme and the (mean) error with the trained scheme i.e, (4.2) with parameters $g^{1,*}, b^{1,*}_{-2,-1}$, on the test set.

c	$g^{1,*}$	$b^{1,*}_{-2}$	$b^{1,*}_{-1}$	Gain-1	Gain-2	Gain-3	Gain-4
0.1	0.64	0	1	77.31	19.58	49.6	38.97
1	0.3	0	1.12	3.97	3.21	3.59	3.98
10	0.04	1.2	2.43	11.91	46.97	11.49	48.17

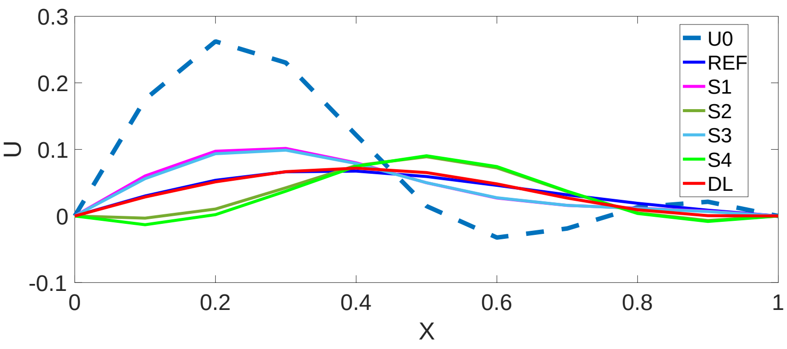

Figure 5. Solutions of the heat equation (4.1) with $c = 1$ at time $T = 0.05$ with an initial value from the *test set* for smooth data i.e (4.5) with a particular realization of $Y_{1,2,3}(\omega)$. We show the initial data (U0), reference solution (REF), solutions approximated with schemes S1 (second-order in space+Backward Euler), S2 (second-order in space+Crank-Nicolson), S3 (fourth-order in space + backward Euler), S4 (fourth-order in space + Crank Nicolson) with the solutions by computed *Data learned* (DL) trained scheme, (4.2) with parameters given in table 3, on a grid with one time step and 10 mesh points. Observe the considerable gain in accuracy with the DL scheme over all the other schemes.

Here, $\varepsilon = 0.2$ and $Y_{1,2,3}$ are chosen randomly from a uniform distribution on $[-1, 1]$. In other words, the initial data (4.7) represents a step function with two discontinuities where the amplitude of the jump at the discontinuity and the location of both jumps are random. The underlying coarse grid is the same as for the smooth case. The training and test sets are generated in a manner, identical to the smooth case and the loss function (4.6) is minimized similarly.

The training (for three different cases of the diffusion coefficient c considered here) resulted in (local) minimizers shown in Table 4. We report that the training process converged very slowly for the $c = 10$ value in the case of this rough initial data. This is reflected in the values of 20 for both spatial weights as we terminated the iterations in the gradient descent method at this stage. One can provide a heuristic explanation for the values of the minimizers in this case. Recall that $c = 10$ implies a very large amount of diffusion in the solution, such that the solution is almost zero at $T = 0.05$. The value of $g^1 = 0$ corresponds to the most diffusive backward Euler method and similarly very high values for $b^1_{-2,-1}$ also imply a large amount of diffusion and drive the approximate solution (computed by (4.2)) to zero.

The gains with the trained scheme, over the four standard schemes, are shown in Table 4 and indicate a very large gain over the best performing of the standard schemes, amounting to a factor of approximately 50 for the case of $c = 10$.

Table 4. The performance of the trained finite difference scheme (4.2) on the heat equation (4.1) for three different values of the constant c and for the rough initial data (4.7). The gain-m is the ratio of the (mean) error with the standard S-m scheme and the (mean) error with the trained scheme i.e, (4.2) with parameters $g^{1,*}, b^{1,*}_{-2,-1}$, on the test set.

c	$g^{1,*}$	$b^{1,*}_{-2}$	$b^{1,*}_{-1}$	Gain-1	Gain-2	Gain-3	Gain-4
0.1	0.24	−0.26	2.15	12.7	3.16	10.56	3.78
1	0.14	−0.38	2.53	2.09	6.03	1.98	6.38
10	0	20	20	46.02	228.93	44.72	231.32

5. Linear advection equation

The linear advection equation is considered as a prototype for the design and analysis of efficient numerical methods for hyperbolic equations. In one space dimension, it is given by

$$\begin{aligned} u_t + cu_x &= 0, \quad (x,t) \in (0,1)\times(0,T), \\ u(x,0) &= u_0(x), \quad x \in (0,1). \end{aligned} \tag{5.1}$$

For definiteness, we assume that $0 \leqslant c \in \mathbb{R}$ and supplement (5.1) with periodic boundary conditions.

5.1. Numerical scheme.

We discretize the computational domain $[0,1]\times[0,T]$ as in section 4.1 and use the following three-point finite difference scheme to approximate the linear advection equation (5.1) by evolving $U_j^n \approx u(x_j, t^n)$ with

$$\begin{aligned} \frac{U_j^{n+1} - U_j^n}{\Delta t} &= \frac{c(1-g^n)}{\Delta x}\left(b^n_{-1}U^{n+1}_{j-1} + b^n_0 U^{n+1}_j + b^n_1 U^{n+1}_{j+1}\right) \\ &+ \frac{cg^n}{\Delta x}\left(b^n_{-1}U^n_{j-1} + b^n_0 U^n_j + b^n_1 U^n_{j+1}\right), \quad \forall n, 1 \leqslant j \leqslant J. \end{aligned} \tag{5.2}$$

The update formulas for the points U_1^n and U_J^n are computed by using the periodic boundary conditions. By using Taylor expansions, one can readily prove the following lemma,

Lemma 5.1. *For all n and any $g^n \in \mathbb{R}$, the finite difference scheme* (5.2) *is a consistent and first-order accurate discretization of the linear advection equation* (5.1) *if and only if the coefficients b_k^n, for $-1 \leqslant k \leqslant 1$, satisfy the following algebraic conditions,*

$$b_1^n + b_0^n + b_{-1}^n = 0, \quad b_1^n - b_{-1}^n = 1. \tag{5.3}$$

We can eliminate two parameters in the system (5.3) in terms of the undetermined parameter b^n_{-1} to obtain,

$$b_0^n = -1 - 2b_{-1}, \quad b_1^n = 1 + b^n_{-1} \tag{5.4}$$

Hence, per time level, the scheme (5.2) contains two undetermined parameters g^n and b^n_{-1}. The scheme (5.2) with constraints (5.3) is consistent as well as *conservative* i.e, $\sum_j U_j^{n+1} = \sum_j U_j^n$.

Additional constraints on the parameters are needed to impose stability conditions such as discrete L^2 (energy) stability or a discrete maximum principle.

The generalized form (5.2) embeds several well-known finite difference approximations namely,

Scheme S1: Backward Euler in time and upwind in space by setting $g^n = 0, b^n_{-1} = 0$

Scheme S2: Crank-Nicolson in time and upwind in space by setting $g^n = 0.5, b^n_{-1} = 0$

Scheme S3: Backward Euler in time and central in space by setting $g^n = 0, b^n_{-1} = 0.5$.

Scheme S4: Crank-Nicolson in time and central accurate in space by setting $g^n = 0.5, b^n_{-1} = 0.5$.

As for the heat equation, we consider only implicit (in time) schemes as they do not require a restriction on the time step Δt. We approximate the solution u of the linear advection equation with scheme (5.2) at time $T = 0.5$ and consider two different values of the wave speed c, namely $c = 0.5$ and $c = 2$ All the experiments will be performed on a very coarse spatial grid with mesh size $\Delta x = \frac{1}{10}$ i.e, 10 mesh points, and a single, very large, time step of $\Delta t = 0.5$.

Table 5. The performance of the trained finite difference scheme (5.2) on the linear advection equation (5.1) for two different values of the wave speed c and for initial data (4.5). The gain is the ratio of the (mean) error with the best performing of the (S1,S2,S3,S4) schemes and the (mean) error with the trained scheme i.e, (5.2) with parameters $g^{1,*}, b^{1,*}_{-1}$, on the test set.

c	$g^{1,*}$	$b^{1,*}_{-1}$	Gain
0.5	0.2	−2	3.72
2.0	−20	0	96.51

5.2. *Training and results on the test set*

To generate the training and test sets, we use an identical set up as described for the heat equation in section 4.2.1. In particular, both the training and test sets are generated from the three term Karhunen-Loeve expansion (4.5). We compute a reference solution on a fine mesh of 1000 points, with an explicit forward Euler time stepping and standard upwind finite difference discretization [16]. The time step is chosen to satisfy the standard CFL requirement for the advection equation. This fine grid solution is projected onto the underlying coarse grid by sampling this solution at points x_j and at final time $T = \Delta t = 0.5$, $1 \leqslant j \leqslant J$, to generate the reference solutions. The loss function (4.6) is minimized with a simplified version of the stochastic gradient algorithm [22] with a batch size of 4. We initialize the gradient descent method with the starting values of $g^1 = 0.5$, $b^1_{-1} = 0$, corresponding to the Crank-Nicolson in time, upwind in space, scheme S2, and denote the (approximate) minimizers as $\{g^{1,*}, b^{1,*}_{-1}\}$. The minimizers, for different values of c, are shown in table 5. We observe that for $c = 0.5$, the computed minimizers deviate greatly from any standard scheme. However, this contrast is much more pronounced in the $c = 2$ case as there was very slow convergence of the stochastic gradient method and it was terminated at $g^{1,*} = -20$, indicating that there is a path along which the loss function (very slowly) approaches a value of zero.

In order to compare with standard schemes, we define a Gain as the ratio of the (mean) error on the test set with the best performing of the four schemes $S1, S2, S3, S4$ (the one with the least mean error) and the trained scheme. For $c = 0.5$, the scheme S2 is the best performing scheme and for $c = 2$, the scheme $S1$ is the best performing scheme. The computed gain is shown in Table 5. For further comparison, we plot a single randomly chosen realization of the test data for both $c = 0.5$ and $c = 2$ in Figure 6.

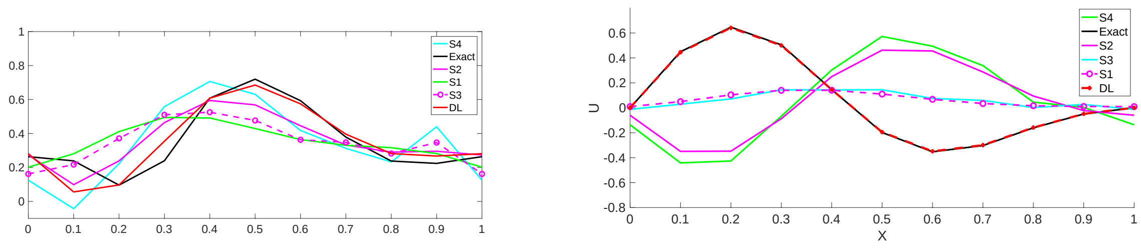

Figure 6. Comparison of five different schemes for the linear advection equation (5.1), namely Backward Euler in time and upwind in space (S1), Crank-Nicolson in time and upwind in space (S2), Backward Euler in time and central in space (S3), Crank-Nicolson in time and central in space (S4) and the trained scheme (DL) i.e, (5.2) with weights given in table 5 on grid of 10 mesh points and with one time step. Comparison with the exact solution computed with the upwind scheme with forward Euler time stepping on a fine grid of 500 points, and on a randomly chosen initial data from (4.5), for two different wave speeds. Left $c = 0.5$, Right $c = 2$.

As shown in Table 5, for the case of $c = 0.5$, the trained scheme provides a gain of 3.72 over the best performing of the standard schemes (the scheme S2). The gains with respect to the backward Euler time stepping schemes are larger. This is also shown in Figure 6 (left), where we observe that the trained scheme is significantly more accurate than standard schemes.

However, the gains with the trained scheme are enormous in the case of $c = 2$, amounting to a gain of almost two orders of magnitude vis a vis the best performing of the standard schemes, see Table 5 and Figure 6 (right). A heuristic explanation for this observation goes as follows: recall that the exact solution coincides with the initial data in this case. Thus in the limit of $g^n \to -\infty$, we can see from (5.2) that $U^{n+1} \approx U^n$ and we are very close to the initial data. It appears that the machine learning algorithm *learns* this fact, when shown training data, and provides this remarkable gain in accuracy in this special case.

6. Burgers' equation

The Burgers' equation given by

$$u_t + \left(\frac{u^2}{2}\right)_x = 0, \quad (x,t) \in (0,1) \times (0,T), \tag{6.1}$$

is a prototypical example for nonlinear hyperbolic conservation laws,

$$\begin{aligned} u_t + (f(u))_x &= 0, \\ u(x,0) &= u_0(x), \end{aligned} \tag{6.2}$$

These equations arise in a wide variety of applications and examples include the Euler equations of gas dynamics, the shallow water equations of oceanography and the MHD equations of plasma physics [6]. It is well-known that solutions of (6.2) develop finite time singularities in the form of *shock waves*, when even the initial data is smooth. Thus, solutions of (6.2) are sought in the sense of distributions and additional *entropy* conditions are imposed in order to recover uniqueness [6].

6.1. Numerical scheme

There is a large body of literature on numerical methods for hyperbolic conservation laws and popular numerical methods include the conservative finite difference schemes and discontinuous Galerkin finite element methods [10]. However, for the sake of simplicity, we consider the simplest *first order finite volume scheme* in this section.

We discretize the interval $[0,1]$ uniformly with a grid size Δx and label the resulting points as $x_j = j\Delta x$ for $0 \leqslant j \leqslant J+1$, with $\Delta x = \frac{1}{J+1}$. Thus, the interval is partitioned into cells (control volumes),

$$C_j := (x_{j-1/2}, x_{j+1/2}), \quad x_{j+1/2} = \frac{x_j + x_{j+1}}{2}, \ \forall j.$$

The time interval $[0,T]$ is discretized uniformly with a time step Δt and the time levels are denoted as $t^n = n\Delta t$. We approximate the cell averages of the solution of (6.2),

$$U_j^n \approx \int_{C_j} u(x,t^n)dx,$$

by writing the update formula,

$$\begin{aligned} U_j^{n+1} &= U_j^n - \frac{\Delta t}{\Delta x}\left(F_{j+1/2}^n - F_{j-1/2}^n\right), \\ U_j^n &= \int_{C_j} u_0(x)dx \end{aligned} \tag{6.3}$$

Here, $F_{j+1/2}^n = F(U_j^n, U_{j+1}^n)$ is a numerical flux, consistent with the flux function f in (6.2). A popular choice is the so-called Local Lax-Friedrichs or *Rusanov* flux given by,

$$F(U_j^n, U_{j+1}^n) = \frac{1}{2}\left(f(U_j^n) + f(U_{j+1}^n)\right) - \frac{1}{2}\max(|f'(U_j^n)|, |f'(U_{j+1}^n)|)(U_{j+1}^n - U_j^n). \tag{6.4}$$

Thus, the numerical diffusion is weighted by a local *wave speed*. It is well-known that the resulting scheme (6.3) with flux (6.4) is *conservative, consistent* and *monotone* [10]. Consequently, the solutions computed by the scheme converge to an entropy solution of (6.2).

We cast the finite volume scheme (6.3) in our machine learning framework by generalizing the flux (6.4) to

$$F(U_j^n, U_{j+1}^n) = \frac{1}{2}\left(f(U_j^n) + f(U_{j+1}^n)\right) - \left(w_{j+1/2}^n \max(|f'(U_j^n)|, |f'(U_{j+1}^n)|)(U_{j+1}^n - U_j^n)\right). \tag{6.5}$$

Here $w_{j+1/2}^n \in \mathbb{R}$, $\forall j$, are weights corresponding to the scaling of the local wave speed. The resulting scheme is given by,

$$\begin{aligned} U_j^{n+1} &= U_j^n - \frac{\Delta t}{2\Delta x}\left(f(U_{j+1}^n) - f(U_{j-1}^n)\right) \\ &\quad + \frac{\Delta t}{\Delta x}\left(w_{j+1/2}^n \max(|f'(U_j^n)|, |f'(U_{j+1}^n)|)(U_{j+1}^n - U_j^n) - w_{j-1/2}^n \max(|f'(U_j^n)|, |f'(U_{j-1}^n)|)(U_j^n - U_{j-1}^n)\right). \end{aligned} \tag{6.6}$$

The properties of this *generalized finite volume scheme* (6.6) are summarized in the lemma below,

Lemma 6.1. *The solutions U_j^n generated by the scheme* (6.6) *satisfy the following,*

i. For any $w_{j+1/2}^n \in \mathbb{R}$, for all j, n, the scheme (6.6) *is conservative and consistent. Moreover, it is (formally) first-order accurate.*

ii. Under the CFL condition,

$$\max_j |f'(U_j^n)| \frac{\Delta t}{\Delta x} \leqslant 1, \tag{6.7}$$

and under the condition $w_{j+\frac{1}{2}}^n \geqslant 0.5$, for all j, n, the scheme (6.6) *is monotone. Hence, the corresponding approximations converge to the entropy solution of the scalar conservation law* (6.2).

The proof of the above lemma is a straightforward adaptation of the results of [10].

Remark 6.2. *We remark that setting $w_{j+1/2}^n \equiv 0.5$ for all n, j, yields the standard Rusanov scheme. The weights w in* (6.6) *serve to modulate the numerical diffusion in the scheme. We can design an alternative scheme by replacing the arithmetic averages of the fluxes in* (6.5) *by an* entropy conservative flux *and replacing the conservative variable u in the numerical diffusion term of the flux* (6.5) *by the entropy variables [25, 8]. The resulting scheme can be proved to be* entropy stable *for any $w_{j+1/2}^n \geqslant 0$.*

Motivated by machine learning architectures such as convolution neural networks [11]. we make a further simplification by *pooling* values of the weights w^n in longer *windows* i.e by requiring that

$$w_{j+1/2}^n = w_{j_s+1/2}^n, \forall |j - j_s| \leqslant m. \tag{6.8}$$

Here $0 \leqslant m \leqslant J$ is the window length and $j_s \subseteq \{1, \ldots, J\}$ is a subset of grid points on which we center the pooling windows.

6.2. Generalized Rusanov scheme as a neural network.

The generalized Rusanov scheme (6.6) can be represented as an artificial neural network as shown in Figure 7, where we focus on the specific example of the Burgers' equation (6.1). Note that in Figure 7, the input neurons or units are components of the vector of unknowns $U^n = \{U_j^n\}$. The output at the end of each time step is the vector U^{n+1}. The network transforms the input into the output through layers of operations that consist of linear maps (connecting different neurons) and nonlinear operations. For this specific example, there are the following nonlinear operations,

ABS	: refers to $ABS(a) = \|a\| = \sigma(a) + \sigma(-a)$, with σ being the ReLU activation function (2.3).
MAX	: refers to $MAX(a, b) = \max(a, b) = a + \sigma(b - a)$.
SQ	; refers to $SQ(a) = 0.5a^2$.
PROD	: refers to $PROD(a, b) = ab$.

Thus, ABS and MAX are directly expressed in the terms of a *traditional* neural network, in the sense of [11], with a ReLU activation. On the other hand, SQ and PROD are bespoke nonlinear operations for this particular example. Hence, the neural network in Figure 7 is not a traditional neural network. However, it can be directly represented in terms of the so-called *sum product networks* [19]. On the other hand, following recent papers such as [30], one can approximate the square and product functions very efficiently in terms of neural networks with ReLU activations, see [23] Figure 1. In particular, one can approximate the square and product maps up to accuracy δ by a neural network of width that is at most logarithmic in δ. Therefore, the neural network underlying the scheme (6.6) is very readily approximated by a traditional ReLU based network architecture. Given several hidden layers per time step (see Figure 7) and multiple time steps, the scheme (6.6) is realized as a *deep neural network*.

It is standard in deep learning to optimize the weights (corresponding to entries in all matrices) for the whole network during the training process. It is in this step that we differ from traditional machine learning and take a more conservative approach. Given that we wish to be consistent (and formally first-order accurate) for any of the weights that might crop up in the training process, we severely constrain the set of free parameters within the network to only the weights w of the numerical viscosity coefficient in (6.6). Even these weights are pooled in windows. Thus, at most we have two free (trainable) parameters in the sub-network shown in Figure 7. Hence, the training process operates on a (considerably) constricted part of the deep neural network.

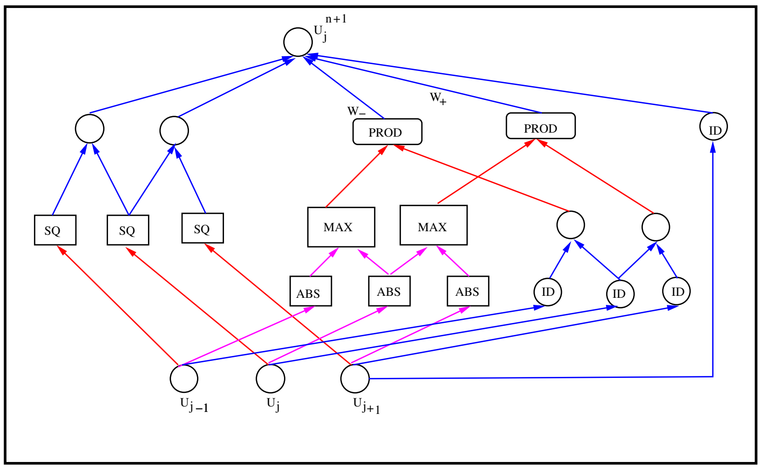

Figure 7. Representation of a single time step of the Rusanov type scheme (6.6) for approximating the Burgers' equation (6.1) as a *multi-layer neural network*. As inputs, we consider the three point stencil $\{U^n_{j-1}, U^n_j, U^n_{j+1}\}$ that are converted to the output U^{n+1}_j through a series of operations. Linear operations are shown with blue arrows. We label SQ to represent the function $f(u) = 0.5u^2$, PROD to be product of two numbers, MAX as the maximum of two numbers and ABS to represent $|u|$. Both MAX and ABS are directly represented by a combination of linear and RELU steps and are shown by magenta arrows where as SQ and PROD are shown with red arrows. The only undetermined weights $W^\pm$ are those of $w^n_{j\pm1/2}$ in (6.6). The linear operation $Au = u$ is labeled as ID. Note that SQ and PROD can be very efficiently approximated by a rather narrow sub-network based on ReLU activation units [30].

6.3. *Training and results on the test set.*

For the numerical experiments, we will only consider the Burgers' equation (6.1) with periodic boundary conditions. We fix $\Delta x = 0.1$ i.e, we discretize the interval $[0, 1]$ into 10 cells. Our final time is $T = 0.1$ the time interval is discretized into *two time steps* with time step size $\Delta t = 0.05$. On this coarse spatial grid, such a large time step is consistent with the CFL condition (6.7). Moreover, we *pool* the weights of the numerical diffusion operator by a defining a pooling window of size $m = 3$ in (6.8). Hence, at each time step we need to specify 3 weights namely $\{w^n_1, w^n_2, w^n_3\}$, correspond of the first, middle and last three of the interior interfaces. The weights on the boundary interfaces are determined from the periodic boundary conditions.

To generate the training set, we first consider the *smooth* random initial data, specified by the Karhunen-Loeve expansion (4.5). As in the previous sections, the expansion is truncated at $L = 3$ and we choose random variables $Y^i_{1,2,3}$, for $1 \leqslant i \leqslant I$ and $I = 20$, uniformly from $[0, 1]$. The corresponding initial data, labeled as u^i_0 is used to initialize the scheme (6.6) and to generate the updated solutions $U^{1,i}_j, U^{2,i}_j$ for all j. A reference solution is computed using the standard Rusanov scheme (i.e setting $w^n_{j+1/2} \equiv 0.5$) on a fine mesh of 1000 points and with the time step determined by the corresponding CFL number as in (6.7). This fine grid solution at the two time levels $\Delta t, 2\Delta t$ is projected to the underlying coarse grid by averaging over the coarse cells and is denoted as $U^{n,i}_{j,\text{ref}}$ for all n, j, i.

The training *loss function* is the L^1 error given by,

$$E_1(w) := \Delta x \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{n=1}^{2} |U^{n,i}_j - U^{n,i}_{j,\text{ref}}|. \tag{6.9}$$

Here, $w = \{w^n\}_{1,2,3}$ for $n = 1, 2$ represents the vector of 6 parameters that specifies the scheme (6.6) on this grid. We remark that the L^1 error is natural for conservation laws as it the norm under which the data to solution operator is continuous [6].

Table 6. The performance of the trained finite volume Rusanov scheme (6.6) on the Burgers' equation (6.1) for the smooth initial data (4.5). The optimized weights are labeled by time level (superscript) and spatial location (subscript). Gain is the ratio of the (mean) error with the standard Rusanove scheme and the (mean) error with the trained scheme on the test set. Speedup represents the overall gain in computational efficiency with the trained scheme over the standard Rusanov scheme.

$w^{1,*}_1$	$w^{1,*}_2$	$w^{1,*}_3$	$w^{2,*}_1$	$w^{2,*}_2$	$w^{2,*}_3$	Gain	Speedup
0.26	0.2	0	0.3	0.3	0	1.42	5.33

The loss function is minimized using a simplified version of the stochastic gradient descent algorithm with a batch size of 4. The stochastic gradient algorithm is applied sequentially i.e, first the minimizers at first time step are determined and then we determine the minimizers at the second time step. This step by step minimization may not yield the optimum in the 6 dimensional parameter space but was found to be reasonably accurate. Moreover, it is computationally cheaper and consistent with the time marching form of numerical methods for evolutionary PDEs. The algorithm was initialized by setting $w^n_{1,2,3} \equiv 0.5$ (corresponding to the standard Rusanov scheme on the coarse grid) and the approximate minimizers, labeled by the vector w^* are shown in table 6. As seen from the table, the minimizing weights are always below the value of 0.5 (being equal to zero in one case). This implies that the numerical diffusion is reduced during training as the solutions in this case are still identified as mostly smooth.

A *test set* is generated by selecting at random 100 realizations from the initial datum (4.5) and the *gain*, defined as the ratio of the mean error of the standard Rusanov scheme (on the coarse grid) to the trained scheme is calculated and shown in Table 6. This gain of 1.42 is rather modest in this case, compared to the previous examples of linear PDEs. However, when we calculate the *speed up* i.e, the ratio of the computational work (time) required to obtain a similar error as the trained scheme (on the coarse grid), but by the standard Rusanov scheme on a finer mesh. In the case of smooth data for the Burgers' equation, this speed up is given in table 6 and amounts to a factor of 5.33.

Table 7. The performance of the trained finite volume Rusanov scheme (6.6) on the Burgers' equation (6.1) for the rough initial data (4.7). The optimized weights are labeled by time level (superscript) and spatial location (subscript). Gain is the ratio of the (mean) error with the standard Rusanove scheme and the (mean) error with the trained scheme on the test set. Speedup represents the overall gain in computational efficiency with the trained scheme over the standard Rusanov scheme.

$w_1^{1,*}$	$w_2^{1,*}$	$w_3^{1,*}$	$w_1^{2,*}$	$w_2^{2,*}$	$w_3^{2,*}$	Gain	Speedup
0.24	0.24	1.25	0.24	0.26	0	2.48	9.55

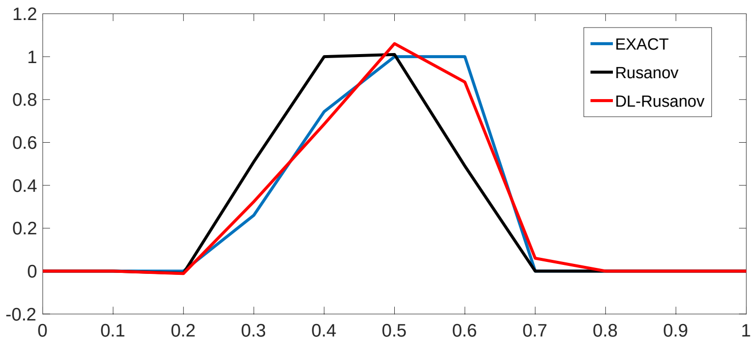

Figure 8. Solutions of the Burgers' equation (6.1) at time $T = 0.1$ with an initial value from the *test set* for rough data i.e (4.7) with a particular realization of $Y_{1,2,3}(\omega)$. We show the reference solution (EXACT), solutions approximated with the standard Rusanov scheme and the solutions by computed *Data learned* (DL) trained scheme, (6.6) with parameters given in table 6, on a grid with two time steps and 10 mesh points.

Larger gains are obtained for rough initial data given by the random initial condition (4.7). Here, the amplitude of the initial discontinuity and the locations of both jumps are uncertain. In this case, we generate the training data u_0^i, identically to the previous case. The reference solution is computed and consists of a right moving shock and a rarefaction on the left (see Figure 8), for each realization. The loss function (6.9) is minimized and the (approximate) minimizers are shown in Table 7. In this case, some optimal values of the weights are significantly higher than 0.5, indicating larger diffusion around the right moving shock whereas many weights are well below the value of 0.5, indicating a modulation of numerical diffusion around smooth regions, identified from the training set. The test set is chosen as before and gain, shown in Table 7 and amounting to a factor of 2.48, is higher than the smooth case. Moreover, the overall speed up in this case is 9.55, representing an order of magnitude computational speed up over the standard Rusanov scheme.

The solutions computed with the trained scheme and with the standard Rusanov scheme, for one particular realization of the initial data (4.7) are shown in Figure 8. We observe from the figure that the trained scheme significantly outperforms the Rusanov scheme for this realization and provides an accurate approximation, even on this very coarse grid.

7. Euler equations

The Euler equations of gas dynamics are a prototypical example of a hyperbolic system of conservation laws [6]. In one space dimension, the Euler equations, representing the conservation of mass, momentum and energy are,

$$\begin{aligned} u_t + (f(u))_x &= 0, \quad (x,t) \in (0,1) \times (0,T) \\ u &= [\rho, \rho v, E], \\ f(u) &= \left[\rho v, \rho v^2 + p, (E+p)v\right], \\ u(x,0) &= u_0(x). \end{aligned} \tag{7.1}$$

Here $u : (0,1) \times (0,T) \to \mathbb{R}^3$ is the vector of unknowns and $f : \mathbb{R}^3 \to \mathbb{R}^3$ is the flux vector. The density of the gas is denoted by ρ, the velocity by v, the pressure by p and the total energy by E. The equations are closed by specified a thermodynamic relation between the variables, such as the *ideal gas equation of state*:

$$E := \frac{p}{\gamma - 1} + \frac{1}{2}\rho v^2, \tag{7.2}$$

with gas constant γ. The system (7.1) is *hyperbolic* with the three eigenvalues,

$$\lambda_1 = u - a, \; \lambda_2 = u, \quad \lambda_3 = u + a, \tag{7.3}$$

given in terms of *sound speed*,

$$a = \sqrt{\frac{p}{\gamma \rho}}. \tag{7.4}$$

The solutions to systems of conservation laws, such as the Euler equations (7.1), develop finite time discontinuities such as shock waves and contact discontinuities, even when the initial data is smooth and as in the scalar case, there is a notion of entropy solutions for them.

7.1. Numerical scheme

We discrete the computational domain of $[0,1] \times [0,T]$ as in the case of scalar conservation laws (section 6.1), and evolve cell averages of the vector of unknowns i.e, $U_j^n = \left[\rho_j^n, (\rho v)_j^n, E_j^n\right]$ by the (generalized) finite volume scheme,

$$U_j^{n+1} = U_j^n - \frac{\Delta t}{\Delta x}\left(F_{j+1/2}^n - F_{j-1/2}^n\right), \quad U_j^n = \int_{C_j} u_0(x)dx \tag{7.5}$$

Here, $F_{j+1/2}^n = F(U_j^n, U_{j+1}^n)$ is a numerical flux vector, consistent with the flux vector f in (7.1). The Local Lax-Friedrichs or *Rusanov* flux for the Euler equations is given by,

$$F(U_j^n, U_{j+1}^n) = \frac{1}{2}\left(f(U_j^n) + f(U_{j+1}^n)\right) - \frac{1}{2}\max(|v_j^n| + a_j^n, |v_{j+1}^n| + a_{j+1}^n)(U_{j+1}^n - U_j^n), \tag{7.6}$$

with a_j^n being the sound speed corresponding to the state U_j^n, computed from (7.4). Thus, the numerical diffusion is scaled with an estimate (upper bound) of the local maximal wave speed given in (7.3). The primitive variables are calculated from the computed conservative variables at the end of each time step. The Rusanov flux is known to very diffusive for systems of conservation laws, particularly around contact discontinuities. However, we choose it here in order to be consistent with our choice in the scalar case and to investigate whether we can train a scheme to (significantly) improve on its numerical performance.

As in the scalar case, we cast the finite volume scheme (7.5) in our machine learning framework by generalizing the Rusanov flux (7.6) to

$$F(U_j^n, U_{j+1}^n) = \frac{1}{2}\left(f(U_j^n) + f(U_{j+1}^n)\right) - \left(w_{j+1/2}^n \max(|v_j^n| + a_j^n, |v_{j+1}^n| + a_{j+1}^n)(U_{j+1}^n - U_j^n)\right). \tag{7.7}$$

Here $w_{j+1/2}^n \in \mathbb{R}$, $\forall j$, are weights corresponding to the scaling of the local wave speed. The resulting scheme is given by,

$$\begin{aligned} U_j^{n+1} &= U_j^n - \frac{\Delta t}{2\Delta x}\left(f(U_{j+1}^n) - f(U_{j-1}^n)\right) \\ &+ \frac{\Delta t}{\Delta x}\left(w_{j+1/2}^n \max(|v_j^n| + a_j^n, |v_{j+1}^n| + a_{j+1}^n)(U_{j+1}^n - U_j^n)\right) \\ &- \frac{\Delta t}{\Delta x}\left(w_{j-1/2}^n \max(|v_j^n| + a_j^n, |v_{j-1}^n| + a_{j-1}^n)(U_j^n - U_{j-1}^n)\right). \end{aligned} \tag{7.8}$$

It is straightforward to verify that the scheme (7.8) is a *conservative* and *consistent* discretization of the one-dimensional Euler equations. It is also (formally) first-order accurate. As in the case of the Burgers' equation, we make a further simplification by *pooling* values of the weights w^n in longer *windows* i.e by requiring (6.8) with $0 \leqslant m \leqslant J$ as the window length and $j_s \subseteq \{1, \dots, J\}$ as a subset of grid points on which we center the pooling windows.

7.2. Training and results on the test set.

For our numerical experiments, we consider the one-dimensional Euler equations (7.1) with the ideal gas equation of state (7.2) and gas constant $\gamma = 1.4$, corresponding to a diatomic gas. We fix $\Delta x = 0.05$ i.e, we discretize the interval $[0, 1]$ into 20 cells. Our final time is $T = 0.15$ and the time interval is divided into *five time steps* with time step size $\Delta t = 0.03$. The scheme (7.8) is closed at the boundary points by imposing *transparent boundary conditions* i.e, a zeroth order extrapolation by setting $U_0^n = U_1^n$ and $U_{J+1}^n = U_J^n$.

We also *pool* the weights of the numerical diffusion by a defining a pooling window of size $m = 3$ in (6.8). Hence, at each time step we need to specify 6 weights namely $\{w_1^n, w_2^n, w_3^n, w_4^n, w_5^n, w_6^n\}$ in (7.8) by grouping every 3 cell interfaces (starting from the left)., The weights on the boundary interfaces are determined from the transparent boundary conditions. Hence, the scheme (7.8) contains 30 parameters that need to be determined in the *training* process.

To generate the training set, we consider the following *random* initial data,

$$\begin{aligned} \rho_0(x,\omega) &= \begin{cases} \rho_l + \varepsilon Y_1(\omega), & \text{if} \quad 0 < x < 0.5 + \varepsilon Y_2(\omega), \\ \rho_r + \varepsilon Y_3(\omega), & \text{if} \quad 0.5 + \varepsilon Y_2(\omega) < x < 1, \end{cases} \\ v_0(x,\omega) &= 0, \quad 0 < x < 1, \\ p_0(x,\omega) &= \begin{cases} p_l + \varepsilon Y_4(\omega), & \text{if} \quad 0 < x < 0.5 + \varepsilon Y_2(\omega), \\ p_r + \varepsilon Y_5(\omega), & \text{if} \quad 0.5 + \varepsilon Y_2(\omega) < x < 1, \end{cases} \end{aligned} \tag{7.9}$$

Here, $\rho_l = p_l = 1$, $\rho_r = p_r = 0.4$ and $\varepsilon = 0.1$. The random variables $Y_{1,2,3,4,5}(\omega)$ are drawn from a uniform distribution on the interval $[-1, 1]$. Thus, the initial data (7.9) corresponds to a stochastic version [17] of the well-known *Sod shock tube* problem for the Euler equations by considering a random interface that separate random jumps in the initial density and pressure.

We draw $I = 50$ samples in (7.9) and label the resulting initial data as u_0^i. These data initialize the scheme (7.8) to generate the updated solutions $U_j^{n,i}$ for all j and $1 \leqslant n \leqslant 5$. A reference solution is computed using the standard Rusanov scheme (i.e setting $w_{j+1/2}^n \equiv 0.5$) in (7.8) on a fine mesh of 1000 points and with the time step determined by the corresponding CFL number as in (6.7). This fine grid solution is projected on the underlying coarse grid by averaging. We denote the reference solution as $U_{j,\text{ref}}^{n,i}$ for all n, j, i.

The training *loss function* is following version of the L^1 error,

$$E_1(w) := \Delta x \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{n=1}^{5} \left(|\rho_j^{n,i} - \rho_{j,\text{ref}}^{n,i}| + |v_j^{n,i} - v_{j,\text{ref}}^{n,i}| + |p_j^{n,i} - p_{j,\text{ref}}^{n,i}| \right) \tag{7.10}$$

Here, $\rho_j^{n,i}, v_j^{n,i}, p_j^{n,i}$ are the density, velocity and pressure computed on the coarse grid by the numerical scheme (7.8) for the training initial data $u_0^{n,i}$. We choose the L^1 error in the primitive variables, rather than in the conservative variables. Our choice is motivated by the fact that in practice, one is interested in measuring the velocity and the pressure, rather than the momentum and energy.

We minimize the loss function (7.10) on the 30-dimensional parameter space by using a stochastic gradient method with batch size of 5. The stochastic gradient method is initialized by setting all the weights to $w_l^n \equiv 0.5$, corresponding to the standard Rusanov scheme. As in the case of the Burgers' equation, we will optimize the loss function sequentially in time, corresponding to each time step. The stochastic gradient method converges fairly quickly in this case to the optimized weights presented in Table 8.

Table 8. Optimized weights w_l^n for $1 \leqslant n \leqslant 5$ and $1 \leqslant l \leqslant 6$ for the trained Rusanov scheme (7.8) for the one-dimensional Euler equations (7.1) with the random initial data training set.

n	$w_1^{n,*}$	$w_2^{n,*}$	$w_3^{n,*}$	$w_4^{n,*}$	$w_5^{n,*}$	$w_6^{n,*}$
1	0.5	0.5	0.42	0.5	0.5	0.5
2	0.5	0.5	0.23	0.48	0.5	0.5
3	0.5	−0.3	0.29	0.51	0.5	0.5
4	0.5	0.17	0.19	0.54	0.5	0.5
5	0.5	0.2	0.33	0.77	0.3	0.5

The (approximate) optimized weights, shown in table 8, follow an interesting pattern. We recall that the solutions of the Euler equations (7.1) with initial data (7.9) consist of the initial discontinuity breaking down into three waves namely, a left moving rarefaction, a right moving contact and an even faster right moving shock wave, see figure 9 for a snapshot of the solution. First, we observe that only 13 of the 30 weights assume a value different from the initial value and this variation is more pronounced with time as waves develop, separate and move away from the initial discontinuity. The values assumed by the weights include locations where the optimal weights are greater than 0.5, implies that more diffusion is added but there are many locations, particularly on the left of the initial interface, corresponding to the rarefaction wave, where the weights are significantly less than the initial value of 0.5 (one is even negative), indicating that diffusion is removed near the continuous part of the solution.

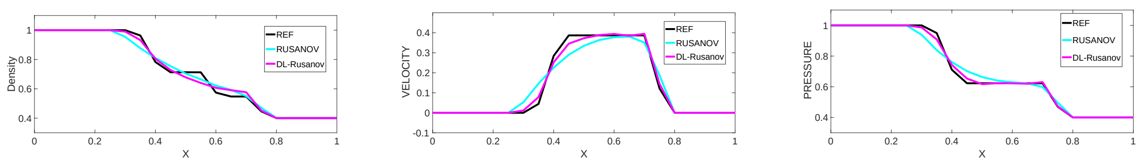

Figure 9. Numerical approximation at time $T = 0.15$ of the 1-dimensional Euler equations (7.1) for a Sod shock tube initial data i.e, (7.9) with $\varepsilon = 0$. We compare a reference solution (computed on a fine mesh but initial data on a coarse mesh), the trained scheme (7.8), on a grid of 20 mesh points and 5 time steps and with optimal weights given in table 8 (DL-Rusanov) and a standard Rusanov scheme on the same coarse grid. Left: Density, Middle: Velocity and Right: Pressure. Note the much improved approximation of the shock wave and the rarefaction wave and further diffusion of the contact discontinuity with the trained scheme.

We generate a *test set* by drawing 1000 samples from (7.9). The resulting gain, defined as the ratio of the (mean) error of the standard Rusanov scheme on the test set, to the (mean) error of the trained scheme, is computed and is determined to be a factor of 2.17. Although this seems rather modest given the much more significant gains for the heat and the linear transport equation, it is comparable to the gain for the Burgers' equation reported in Table 7. However, the key quantity to demonstrate the efficiency of the trained scheme is the *speed up* i.e the ratio of the (mean) computational time for the standard Rusanov scheme (on a finer grid) to achieve the same error as the trained scheme on the underlying coarse grid. Given that the observed (mean) order of convergence of the standard Rusanov scheme on this problem is found to be 0.57 (even if the Rusanov scheme is (formally) first-order accurate), we obtain that the trained scheme on a grid of 20 mesh points (and five time steps) is comparable in error to a standard Rusanov scheme on a grid of 80 mesh points (and time steps determined from the CFL number). Consequently, the speedup is a factor of 16.

Further insight into the performance of the trained scheme is provided in Figure 9, where we plot the computed density, velocity and pressure with the trained scheme (7.8) and compare it with the standard Rusanov scheme and a reference solution. As seen from the Figure 9 (left), the trained scheme provides a considerably more accurate approximation of the rarefaction wave and the (fast) shock, even on this very coarse grid. On the other hand, it dissipates the contact even further. This counter-intuitive behavior can be explained on the basis of the loss function (7.10). Note that the pressure and the velocity are constant across the contact. Thus, the contribution of the contact in the loss function (7.10) can be rather small. Hence, during the training process, the scheme "decides" not to approximate the contact better but to focus on approximating the shock and the rarefaction more accurately. This strategy clearly bears fruit as the velocity (Figure 9 (middle)) and pressure (Figure 9 (right)) are approximated very well, albeit with small oscillations, leading to a larger overall reduction in the loss function (7.10). This non-intuitive behavior is in stark contrast to traditional approaches that focus on approximating the contact discontinuity more accurately.

8. Discussion

Numerical methods for efficient approximation of (time-dependent) ordinary and partial differential equations are well established. However, emerging applications such as uncertainty quantification (UQ), (Bayesian) inverse problems and (real time) optimal control and design require fast (computationally cheap) yet accurate numerical methods. Existing numerical schemes, particularly for nonlinear PDEs, fail to provide reasonable accuracy at very low computational cost.

In this paper, we have proposed a *machine learning* framework, summarized in Algorithm 2.1, for designing such cheap yet accurate methods. The basis of our algorithm is the observation that the computational cost of existing numerical methods on (very) coarse (space-time) grids is rather low. However, these methods are too inaccurate on such grids to be of practical use. We aim to increase the accuracy (reduce the numerical error) of these methods on coarse grids. To this end, we recast of generalizations of standard numerical methods in terms of artificial neural networks i.e layers of units coupled with linear operators and possibly nonlinear activations, but with a set of undetermined parameters. These parameters are *trained* to minimize a *loss function* on a carefully chosen *training set* in an offline training phase. The training is performed with a *stochastic* gradient descent method initialized with parameters corresponding to a standard numerical method. Experience indicates that the training process was fast and converged to a local minimum (with a significant decrease in the loss function) in a few iterations.

The key properties of our proposed algorithm are

- The resulting method is always consistent with underlying ODE or PDE by design. Additional constraints can be imposed on the trainable parameters to ensure stability.
- The method is guaranteed to be more accurate than a standard numerical method on the same grid as the gradient descent method is initialized with parameters that correspond to a standard method.
- The method is very simple to implement with minor changes in existing numerical ODE and PDE solvers.

Although no theoretical guarantees have been established on whether the proposed algorithm significantly outperforms standard methods on the *test set*, we have presented extensive numerical experiments to ascertain this enhancement in performance. Our numerical experiments include a linear and a non-linear ODE, the linear heat and transport equations, scalar conservation laws (Burgers' equation) and the Euler equations of gas dynamics. We considered underlying numerical schemes that include implicit multi-step methods for ODEs, implicit finite difference schemes for linear PDEs and explicit finite volume schemes for nonlinear PDEs. Loss functions, measuring error in either L^1 or L^2 norms were minimized with stochastic gradient methods. The numerical experiments demonstrated a significant gain in performance (computational speed up) over the underlying standard numerical method. The gains ranged from an order of magnitude for nonlinear problems to two (or three) orders of magnitude for linear problems. In all cases considered here, the machine learning algorithm 2.1 provided a numerical method with reasonable accuracy on a very coarse space-time grid. Hence, it could serve as a basis for the solution of complex problems in UQ, inverse problems or (real time) optimal control.

It is instructive to compare our approach to possible *deep learning* of the solutions of differential equations. It is essential to recall that one can cast standard numerical methods for time-dependent differential equations in the form of a deep neural network, see section 2 (Figure 2) and section 6 (Figure 7) for concrete examples. At the very least, non-linearities that occur in standard numerical methods, can be approximated with standard deep neural networks based on ReLU activations. Thus, our approach does consist of approximating differential equations with a form of deep networks. Hence, standard deep learning methodology such as back propagation, stochastic gradients, pooling etc and software frameworks like *TENSORFLOW*, can be readily used.

However, as explained in section 6.2, there is a significant difference in our algorithm with customary deep learning. In machine learning, one usually trains all the parameters in the network. Doing so in our context, see Figure 7, may lead to a lack of consistency with the underlying differential equation. In order to retain consistency (and possibly stability), one needs to constrain the set of parameters in order to recover these properties for every value of the trainable parameters. We do so with our (natural) generalization of numerical schemes. Thus, one can consider algorithm 2.1 as a deep learning algorithm, with a very particular architecture, and with a (very) restricted set of trainable parameters. We retain consistency and at the same time, notice significant gains in computational efficiency. It could be that a free training of all the parameters in the deep network, underlying our generalized numerical method, will automatically identify regions of the parameter space (particularly if additional penalization terms are added to the loss function) to retain consistency and stability. This approach needs to be explored in the future.

It should be emphasized that the (approximate) optimal values of parameters calculated in all our examples, depend strongly on the training set, the underlying coarse grid and parameters of the problem such as the diffusion coefficient for the heat equation (4.1) or the wave speed in the linear advection equation (5.1). One can also train the algorithm to take a possible stochastic model for some of these parameters into account. Moreover for sake of simplicity of exposition, we have mostly considered model problems with very simple underlying numerical methods in this paper. The number of undetermined parameters was in the range of 2–3 for ODEs and linear PDEs to at most 30 for nonlinear PDEs. State of the art deep learning architectures handle hundreds of thousands to millions of trainable parameters and we anticipate a much larger gain in efficiency when we dramatically increase the width and depth of our schemes, represented as networks. Applications of the proposed algorithm 2.1 to realistic multi-dimensional problems in uncertainty quantification and inverse problems is the subject of ongoing work.

Acknowledgments

The author thanks Kjetil O. Lye (SAM, ETH Zürich) for interesting discussions on this topic. The research of SM was partially funded by ERC CoG 770880 COMANFLO.

Conflict of interest

All authors declare no conflicts of interest in this paper.

References

1. Barron AR (1993) Universal approximation bounds for superpositions of a sigmoidal function. *IEEE T Inform Theory* 39: 930–945.
2. Beck C, Weinan E and Jentzen A (2017) Machine learning approximation algorithms for high dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. Technical Report 2017-49, Seminar for Applied Mathematics, ETH Zürich.
3. Bijl H, Lucor D, Mishra S, et al. (2014) Uncertainty quantification in computational fluid dynamics. Lecture notes in computational science and engineering 92, Springer.
4. Borzi A and Schulz V (2012) Computational optimization of systems governed by partial differential equations, *SIAM*.
5. Brenner SC and Scott LR (2008) The mathematical theory of finite element methods. Texts in applied mathematics 15, Springer.
6. Dafermos CM (2005) Hyperbolic Conservation Laws in Continuum Physics (2nd Ed.), Springer Verlag.
7. Weinan E and Yu B (2018) The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems. *Commun Math Stat* 6: 1–12.
8. Fjordholm US, Mishra S and Tadmor E (2012) Arbitrarily high-order order accurate essentially non-oscillatory entropy stable schemes for systems of conservation laws. *SIAM J Numer Anal* 50: 544–573.
9. Ghanem R, Higdon D and Owhadi H (2016) Handbook of uncertainty quantification, Springer.
10. Godlewski E and Raviart PA (1991) Hyperbolic Systems of Conservation Laws. Mathematiques et Applications, Ellipses Publ., Paris.
11. Goodfellow I, Bengio Y and Courville A (2016) Deep learning. MIT Press. Available from: `http://www.deeplearningbook.org`.
12. Hairer E and Wanner G (1991) Solving ordinary differential equations. Springer Series in computational mathematics, 14, Springer.
13. Hornik K, Stinchcombe M, and White H (1989) Multilayer feedforward networks are universal approximators. *Neural networks* 2: 359–366.
14. Kingma DP and Ba JL (2015) Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, 1–13.
15. LeCun Y, Bengio Y and Hinton G (2015) Deep learning. *Nature* 521: 436–444.
16. LeVeque RJ (2007) Finite difference methods for ordinary and partial differential equations, steady state and time dependent problems, *SIAM*.
17. Mishra S, Schwab C and Šukys J (2012) Multi-level Monte Carlo finite volume methods for nonlinear systems of conservation laws in multi-dimensions. *J Comput Phys* 231: 3365–3388.
18. Miyanawala TP and Jaiman RK (2017) An efficient deep learning technique for the Navier-Stokes equations: application to unsteady wake flow dynamics. Preprint, arXiv :1710.09099v2.
19. Poon H and Domingos P (2011) Sum-product Networks: A new deep architecture. *International conference on computer vision (ICCV)*, 689–690.
20. Raissi M and Karniadakis GE (2018) Hidden physics models: machine learning of nonlinear partial differential equations. *J Comput Phys* 357: 125–141.
21. Ray D and Hesthaven JS (2018) An artificial neural network as a troubled cell indicator. *J Comput Phys* to appear.
22. Ruder S (2017) An overview of gradient descent optimization algorithms. Preprint, arXiv.1609.04747v2.

23. Schwab C and Zech J (2017) Deep learning in high dimension. Technical Report 2017-57, Seminar for Applied Mathematics, ETH Zürich.

24. Stuart AM (2010) Inverse problems: a Bayesian perspective. *Acta Numerica* 19: 451–559.

25. Tadmor E (2003) Entropy stability theory for difference approximations of nonlinear conservation laws and related time-dependent problems. *Acta Numerica*, 451–512.

26. Tompson J, Schlachter K, Sprechmann P, et al. (2017) Accelarating Eulerian fluid simulation with convolutional networks. Preprint, arXiv:1607.03597v6.

27. Trefethen LN (2000) Spectral methods in MATLAB, SIAM.

28. Troltzsch F (2010) Optimal control of partial differential equations. AMS.

29. Quateroni A, Manzoni A and Negri F (2015) Reduced basis methods for partial differential equations: an introduction, Springer Verlag.

30. Yarotsky D (2017) Error bounds for approximations with deep ReLU networks. *Neural Networks* 94: 103–114