Received: 1 December 2022 | Revised: 20 April 2023 | Accepted: 24 May 2023

DOI: 10.1002/nme.7306

SHORT COMMUNICATION

WILEY

Neural network representation of time integrators

Rainald Löhner[1] | **Harbir Antil**[2]

[1]Center for Computational Fluid Dynamics and Department of Physics, George Mason University, Fairfax, Virginia, USA

[2]Center for Mathematics and Artificial Intelligence and Department of Mathematical Sciences, George Mason University, Fairfax, Virginia, USA

Correspondence
Rainald Löhner, Center for Computational Fluid Dynamics and Department of Physics, George Mason University, Fairfax, VA, USA.
Email: rlohner@gmu.edu

Funding information
Air Force Office of Scientific Research, Grant/Award Number: FA9550-22-1-0248; National Science Foundation, Grant/Award Number: DMS-2110263

Abstract

Deep neural network (DNN) architectures are constructed that are the exact equivalent of explicit Runge–Kutta schemes for numerical time integration. The network weights and biases are given, that is, no training is needed. In this way, the only task left for physics-based integrators is the DNN approximation of the right-hand side. This allows to clearly delineate the approximation estimates for right-hand side errors and time integration errors. The architecture required for the integration of a simple mass-damper-stiffness case is included as an example.

KEYWORDS

deep neural networks, Runge–Kutta, numerical integration

1 | INTRODUCTION

Considerable effort is currently being devoted to neural nets, and in particular so-called deep neural nets (DNNs). DNNs have been shown to be very good for sorting problems (e.g., image recognition) or games (e.g., chess). Their use as ordinary or partial differential equation (PDE) solvers is the subject of much speculation, with many variants such as Residual DNNs,[1] Physically Inspired NNs (PINNs),[2] Numerically Inspired NNs (NINNs), Nudging NNs (NUNNs),[3] Fractional DNNs[4,5] and others being explored at present. The current situation is somewhat reminiscent of previous attempts to use general, easy-to-use tools from other fields to solve ordinary or partial differential equations. Examples include the "discoveries" that one could use MS Excel to solve ODEs,[6,7] Simulink for some simple PDEs,[8,9] cellular automata for ODEs and PDEs,[10,11] and ResNets for ODEs.[12]

When solving time-dependent ODEs or PDEs, the resulting system is given by:

$$u_{,t} = r(u, t),$$

where u is the (scalar or vector) unknown, r the (possibly nonlinear) right-hand-side and t time. The system can be integrated via explicit Runge–Kutta (RK) schemes which are of the form:

$$\begin{aligned} u^{n+1} &= u^n + \Delta t\ b_i\ r^i, \\ r^i &= r(t^n + c_i \Delta t,\ u^n + \Delta t\ a_{ij}\ r^j), \quad i = 1, s, \quad j = 1, s-1. \end{aligned}$$

Any particular RK method is defined by the number of stages s and the coefficients $a_{ij}, 1 \le j < i \le s$, $b_i, i = 1, s$ and $c_i, i = 2, s$.

Given that attempts are being made to replace time integrators by DNNs, one might ask: how should the architecture of the DNNs be in order to obtain the optimal time integration properties of RK schemes? This would clarify:

- The minimum number of layers required for DNNs;
- The minimum width required for DNNs;
- The weights and biases required;
- The overall efficiency of DNNs versus other alternatives; and
- Approximation properties of DNNs.

The remainder of the paper is organized as follows: Section 2 describes the standard neural network architectures. Section 3 establishes that standard polynomials can be represented by the activation functions. Section 4 which illustrates how DNNs can be built that result in standard time integrators. The architecture required for the integration of a simple mass-damper-stiffness case is included as an example in Section 5.

2 | NEURAL NET ARCHITECTURES

A general DNN configuration consists of L number of hidden layers along with one input and one output layer. Each hidden layer, the input layer and the output layer have K, N and J number of neurons, respectively. The input-to-output sequence of such a DNN may be written as follows.

$$\text{Input:} \quad G_k^1 = g\left(\sum_{n=1}^{N} w_{kn}^1 M_n + [bias]_k^1\right), \quad k = 1, K^1, \tag{1a}$$

$$\text{Hidden:} \quad G_k^l = g\left(\sum_{m=1}^{K^{l-1}} w_{km}^l G_m^{l-1} + [bias]_k^l\right), \quad k = 1, K^l, \quad l = 2, L, \tag{1b}$$

$$\text{Output:} \quad BC_j = \phi\left(\sum_{m=1}^{K^L} w_{jm}^L G_m^L\right), \quad j = 1, J, \tag{1c}$$

where M is the input vector, $\phi(x), g(x)$ activation functions, w the weights and $bias$ the biases. Typical activation functions for $\phi(x), g(x)$ include:

- Heaviside: $HS(x) = 1$ for $x \ge 0$, $\phi(x) = 0$ for $x < 0$,
- Logistic: $LG(x) = 1/(1 + \exp(-x))$,
- ReLU: $ReLU(x) = \max\{0, x\}$,
- HypTan: $HTAN(x) = \tanh(x)$.

Functions that do not have an "activation behavior" but that have proven useful include:

- Constant: $CO(x) = 1$,
- Linear: $LI(x) = x$.

3 | POLYNOMIAL FUNCTIONS IN 1-D

Let us now consider how to represent local polynomial functions via DNNs. An important question pertains to the activation functions used. In typical DNNs, these are "switched on" when the input value crosses a threshold.

3.1 | Constant function

Let us try to approximate the constant function $y(x) = 1$ via DNNs. The simplest way to accomplish this via "true activation functions" with just one neuron would be via:

$$DNN_c: \quad y(x) = HS(x - x_\infty),$$

where x_∞ is a very large value. An alternative is to use two neurons as follows:

$$DNN_{c2}: \quad y(x) = HS(x + \epsilon) + HS(-x),$$

where ϵ would be of the order of machine roundoff. Note that the desire to "activate" (which is seen as a requirement of general DNNs) in this case has a negative effect, prompting the need for either very large or small numbers—something that may lead to slow convergence of "learning" or numerical instabilities. A far better alternative would have been the use of the constant activation function $CO(x)$.

3.2 | Linear function

Let us try to approximate the linear function $y(x) = x$ via DNNs and "true activation functions". The obvious candidate would be $ReLU(x)$. But as it has to work for all values of x one can either use $ReLU(x) + HS(x)$

$$DNN_l: \quad y(x) = ReLU(x - x_\infty) + x_\infty \cdot HS(x - x_\infty),$$

or:

$$DNN_{l2}: \quad y(x) = ReLU(x + \epsilon) - ReLU(-x).$$

As before, the desire to "activate" has a negative effect, prompting the need for either very large or small numbers. A far better alternative would have been the use of the linear activation function $LI(x)$.

In DNNs, one usually refrains from using higher order functions, trying to leverage the generality of lower order or differentiable activation functions.

3.3 | Higher order polynomial functions in 1-D

Consider now the polynomial

$$y(x) = a_j x^j, \quad j \geq 2.$$

The aim is to construct a DNN that would mirror this polynomial using the usual activation functions. Given that DNNs only act in an additive manner, this is not possible. The usual recourse is to approximate it by a series of linear functions.[13] Another option is to transform to logarithmic variables, add, and then transform back—but this would imply a major change in network architecture and functions. Consider

$$f(\mathbf{x}) = \sum_{j=1}^{d} a_j x_j,$$

where a_j are free coefficients and x_j the spatial coordinates in each dimension j. Note that only additions and "weights" (a_j) are required, so the usual ReLU and HS functions should be able to reproduce this function. But how many neurons are required? Borrowing from simplex (linear) finite element shape functions, one would have to build a linear function for each face of the ball of elements (patch) surrounding a point. This implies a considerable number of neurons for higher dimensional spaces. We refer to.[14-16]

4 | EXPLICIT TIMESTEPPING FOR ODES

Consider the typical scalar ODE of the form:

$$u_{,t} = r(u,t).$$

Explicit time integration schemes take the right-hand-side r at a known time t (or at several known times), and predict the unknown u at some time in the future based on it. The simplest such scheme is the forward Euler scheme, given by:

$$u^{n+1} = u^n + \Delta t\ r(t^n, u^n).$$

Given that the function $r(u,t)$ is arbitrary, we will assume that a DNN has been constructed for it. We will denote this approximation of $r(u,t)$ as DNN_r. Note that in the scalar case this DNN has two inputs (u,t) and one output $(r(u,t))$. In order to obtain a complete DNN for the forward Euler scheme, we need to enlarge DNN_r by the "pass-through" value of u. As was shown above, this can be accomplished with one layer of 2 ReLU functions, or via one identity function. We will denote this DNN as DNN_I in the sequel. The final DNN, shown in Figure 1 can then be denoted as:

$$u^{n+1} = DNN_I(u^n) + \Delta t DNN_r(t^n, u^n).$$

In this and the subsequent figures we have highlighted the "important" or "essential" DNN_r for $r(u,t)$. The generalization to higher order schemes is given by the family of explicit **Runge–Kutta** (RK) methods, which may be expressed as:

$$\begin{aligned} u^{n+1} &= u^n + \Delta t\ b_i\ r^i, \\ r^i &= r(t^n + c_i\Delta t,\ u^n + \Delta t\ a_{ij}\ r^j), \quad i = 1,s, \quad j = 1,s-1. \end{aligned}$$

Any particular RK method is defined by the number of stages s and the coefficients $a_{ij}, 1 \le j < i \le s$, $b_i, i = 1,s$ and $c_i, i = 2,s$. These coefficients are usually arranged in a table known as a Butcher tableau (see Butcher)[17]:

	r^1	r^2	…	r^{s-1}	r^s
0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
⋮	⋮	⋮	⋱		
c_s	a_{s1}	a_{s2}	…	$a_{s,s-1}$	
	b_1	b_2	…	b_{s-1}	b_s

The two-step (second order) RK scheme is given by:

	r^1	r^2
0		
1/2	1/2	
	0	1

.

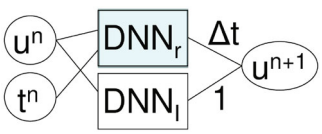

FIGURE 1 DNN for RK1 (Forward Euler) scheme.

For clarity, let us write the scheme out explicitly:

- Step 1: $u^{n+1/2} = u^n + \frac{\Delta t}{2} r(u^n, t^n)$,
- Step 2: $u^{n+1} = u^n + \Delta t\ r(u^{n+1/2}, t^{n+1/2})$.

The DNN architecture required for this time integration scheme is shown in Figure 2.

The classic fourth order RK scheme is given by:

	r^1	r^2	r^3	r^4
0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

For clarity, let us write the scheme out explicitly:

- Step 1: $u^{n+1/4} = u^n + \frac{\Delta t}{2} r(u^n, t^n)$,
- Step 2: $u^{n+1/3} = u^n + \frac{\Delta t}{2} r(u^{n+1/4}, t^{n+1/2})$,
- Step 3: $u^{n+1/2} = u^n + \Delta t r(u^{n+1/3}, t^{n+1/2})$,
- Step 4: $u^{n+1} = u^n +$
  $\frac{\Delta t}{6}\left[r(u^n, t^n) + 2r(u^{n+1/4}, t^{n+1/2}) + 2r(u^{n+1/3}, t^{n+1/2}) + r(u^{n+1/2}, t^{n+1})\right]$.

The DNN architecture required for this time integration scheme is shown in Figure 3.

Observe that schemes of this kind require the storage of **several** copies of the unknown/right hand side, as the final result requires $r^i, i = 1, s$. Furthermore, as each right-hand side possibly requires the information of all previous right-hand sides of the timestep, the resulting neural net architecture deepens.

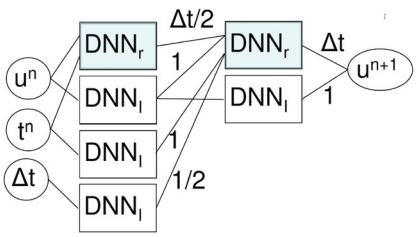

FIGURE 2 DNN for RK2 scheme.

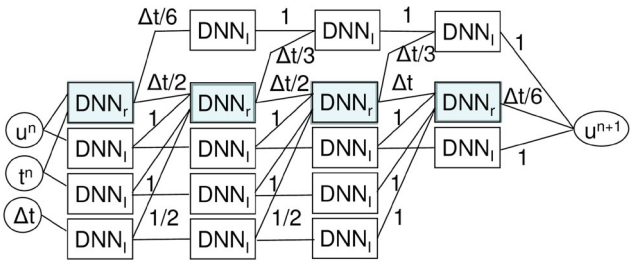

FIGURE 3 DNN for RK4 scheme.

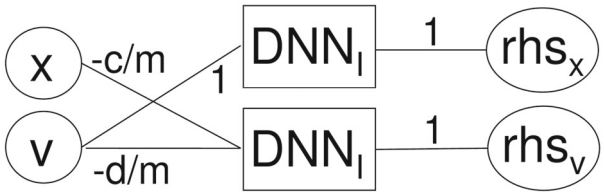

FIGURE 4 DNN for mass-damper-stiffness system.

5 | EXAMPLE: MASS-DAMPER-STIFFNESS SYSTEM

Consider the simple mass-damper-stiffness system common to structural mechanics, given by the scalar ODE:

$$m \cdot x_{,tt} + d \cdot x_{,t} + c \cdot x = 0,$$

where m, d, c, x denote the mass, damping, stiffness and displacement respectively. The ODE may be re-written as a first order ODE via:

$$x_{,t} = v \quad , \quad v_{,t} = -\frac{d}{m} v - \frac{c}{m} x.$$

The resulting DNN_r is shown in Figure 4.

6 | CONCLUSIONS AND OUTLOOK

Deep neural network (DNN) architectures were constructed that are the exact equivalent of explicit Runge–Kutta schemes for numerical time integration. The network weights and biases are given, that is, no training is needed. In this way, the only task left for physics-based integrators is the DNN approximation of the right-hand side. This allows to clearly delineate the approximation estimates for right-hand side errors and time integration errors.

As the explicit Runge–Kutta schemes require the information of all previous right-hand sides of the timestep, the resulting neural net architecture depth is proportional to the number of stages—and hence to the integration order of the scheme.

As the DNN for the approximation of the right-hand side may already be "deep", that is, with several hidden layers, the final DNN for high-order ODE integration many be considerable.

ACKNOWLEDGMENTS

This work is partially supported by NSF grant DMS-2110263 and the Air Force Office of Scientific Research under Award No: FA9550-22-1-0248.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Rainald Löhner https://orcid.org/0000-0003-0083-5131

REFERENCES

1. Haber E, Ruthotto L. Stable architectures for deep neural networks. *Inverse Probl*. 2018;34(1):014004. doi:10.1088/1361-6420/aa9a90
2. Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J Comput Phys*. 2019;378:686-707.
3. Antil H, Löhner R, Price R. *NINNs: Nudging Induced Neural Networks. Tech. Rep*. Cornell University; 2022. arXiv:2203.07947[cs,math].
4. Antil H, Khatri R, Löhner R, Verma D. Fractional deep neural network via constrained optimization. *Mach Learn Sci Technol*. 2020;2(1):015003. doi:10.1088/2632-2153/aba8e7

5. Antil H, Elman HC, Onwunta A, Verma D. Novel deep neural networks for solving Bayesian statistical inverse problems. arXiv preprint, arXiv:2102.03974. 2021.
6. Leon BS, Ulfig R, Blanchard J. Neural network theory. *Comput Appl Eng Educ*. 1996;4(2):117-125.
7. Huseynov S. *Methodology of Laboratory Workshops on Computer Modeling with Programming in Microsoft Excel Visual Basic for Applications*. IEEE; 2013:1-5.
8. Gran RJ. Creating simulations. *Numerical Computing with Simulink*. Vol I. Society for Industrial and Applied Mathematics (SIAM); 2007.
9. Shampine LF, Reichelt MW, Kierzenka JA. Solving index-1 DAEs in MATLAB and Simulink. *SIAM Rev*. 1999;41(3):538-552.
10. Wolfram S, Gad-el-Hak M. A new kind of science. *Appl Mech Rev*. 2003;56(2):B18-B19.
11. Wolfram S. Cellular automata as models of complexity. *Nature*. 1984;311(5985):419-424.
12. Chen RTQ, Rubanova Y, Bettencourt J, Duvenaud DK. Neural ordinary differential equations. In: Bengio S, Wallach H, Larochelle H, Grauman K, Cesa-Bianchi N, Garnett R, eds. *Advances in Neural Information Processing Systems*. Vol 31. Curran Associates, Inc.; 2018. https://proceedings.neurips.cc/paper_files/paper/2018/file/69386f6bb1dfed68692a24c8686939b9-Paper.pdf
13. Yarotsky D. Error bounds for approximations with deep ReLU networks. *Neural Netw*. 2017;94:103-114.
14. He J, Li L, Xu J, Zheng C. ReLU deep neural networks and linear finite elements. arXiv preprint, arXiv:1807.03973. 2018.
15. Petersen PC. *Neural Network Theory*. *University of Vienna*; 2020.
16. DeVore R, Hanin B, Petrova G. Neural network approximation. *Acta Numer*. 2021;30:327-444.
17. Butcher JC. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons; 2003.

How to cite this article: Löhner R, Antil H. Neural network representation of time integrators. *Int J Numer Methods Eng*. 2023;1-7. doi: 10.1002/nme.7306