INVARIANT PRESERVATION IN MACHINE LEARNED PDE SOLVERS VIA ERROR CORRECTION

Nick McGreivy
Department of Astrophysical Sciences
Princeton University
mcgreivy@princeton.edu

Ammar Hakim
Princeton Plasma Physics Laboratory
Princeton, NJ
ahakim@pppl.gov

ABSTRACT

Machine learned partial differential equation (PDE) solvers trade the reliability of standard numerical methods for potential gains in accuracy and/or speed. The only way for a solver to guarantee that it outputs the exact solution is to use a convergent method in the limit that the grid spacing Δx and timestep Δt approach zero. Machine learned solvers, which learn to update the solution at large Δx and/or Δt, can never guarantee perfect accuracy. Some amount of error is inevitable, so the question becomes: how do we constrain machine learned solvers to give us the sorts of errors that we are willing to tolerate? In this abridged version of a full-length paper, we design more reliable machine learned PDE solvers by preserving discrete analogues of the continuous invariants of the underlying PDE. Examples of such invariants include conservation of mass, conservation of energy, the second law of thermodynamics, and/or non-negative density. Our key insight is simple: to preserve invariants, at each timestep apply an error-correcting algorithm to the update rule. Though this strategy is different from how standard solvers preserve invariants, it is necessary to retain the flexibility that allows machine learned solvers to be accurate at large Δx and/or Δt. This strategy can be applied to any autoregressive solver for any time-dependent PDE in arbitrary geometries with arbitrary boundary conditions. Although this strategy is very general, the specific error-correcting algorithms need to be tailored to the invariants of the underlying equations as well as to the solution representation and time-stepping scheme of the solver. The error-correcting algorithms we introduce have two key properties. First, by preserving the right invariants they guarantee numerical stability. Second, in closed or periodic systems they do so without degrading the accuracy of an already-accurate solver.

1 INTRODUCTION AND MAIN IDEA

Scientists and engineers are interested in solving partial differential equations (PDEs). Many PDEs cannot be solved analytically, and must be approximated using discrete numerical algorithms. We refer to these algorithms as ‘PDE solvers.’ The fundamental challenge for PDE solvers is to balance between two competing objectives: first, to find an accurate approximation to the solution of the equation, and second, to do so with as few computational resources as possible.

Decades of research into discrete numerical algorithms have resulted in reliable solvers for most PDEs of interest. For time-dependent PDEs, these so-called ‘standard numerical methods’ use hand-crafted rules to update the solution at each timestep. Successful hand-crafted update rules have two key properties. First, the property of *convergence*. Convergent methods converge to the exact solution in the limit that the grid spacing Δx and the timestep Δt approach zero. Second, the property of *invariant preservation*. Time-dependent PDEs often have one or more invariants. Examples of such invariants include conservation of energy, non-decreasing entropy, and/or non-negative density. Invariant preserving PDE solvers satisfy discrete analogues of these continuous invariants when Δx and Δt are positive. As a result, they are numerically stable and do not violate qualitatively important properties.

In recent years, scientists and engineers have attempted to use machine learning (ML) to design new and better PDE solvers (Tompson et al., 2017; Bar-Sinai et al., 2019; Zhuang et al., 2021; Mohan et al., 2020; Hsieh et al., 2019; Wang et al., 2019; Beck et al., 2019; Um et al., 2020). The goal of these machine learned PDE solvers is as follows. Suppose there is a PDE we would like to find an approximate solution to for many different initial or boundary conditions (ICs or BCs) or on a very large domain. We use training data to produce a learned update rule that can find an approximate solution faster than the best-performing standard solvers. This machine learned solver can then be used to amortize the initial cost of training over many different ICs or BCs or a much larger domain, by finding sufficiently accurate solutions at reduced computational cost. To do so, machine learned solvers use a radically different approach from standard PDE solvers. Instead of designing hand-crafted update rules that converge as $\Delta x \to 0$ and $\Delta t \to 0$, machine learned solvers learn an update rule from data that is accurate at some large value(s) of Δx and/or Δt (Kochkov et al., 2021; Stachenfeld et al., 2021; Li et al., 2020; Greenfeld et al., 2019; Luz et al., 2020; Dresdner et al., 2022; List et al., 2022; Um et al., 2020; Pathak et al., 2020). Ideally, this update rule would be faster than standard methods while being equally as accurate.

Because the only way for a numerical method to guarantee perfect accuracy is (except in trivial cases) to use a convergent method in the limit $\Delta x \to 0$ and $\Delta t \to 0$, there is no way to guarantee that machine learned solvers give an accurate approximation while using large Δx and/or Δt. Some amount of error is inevitable, so the question becomes: how can we constrain machine learned solvers to give us the sorts of errors that we are willing to tolerate? In other words, how can we build more reliable machine learned PDE solvers? One approach to building more reliable machine learned PDE solvers is to change the ML model and training procedure. This approach is quite natural to students of ML. Improving the model (Brandstetter et al., 2022b), increasing the dataset size (Brandstetter et al., 2022a), changing the loss function (Um et al., 2020), and adding regularization (Kaptanoglu et al., 2021; Erichson et al., 2019) are all examples of this approach. To some extent, these techniques have been successful at improving robustness and numerical stability. However, none of these ML-based techniques are capable of *guaranteeing* numerical stability. While these solvers may give reliable results for some inputs, on other inputs the solution might blow up or be nonsensical.

The purpose of this paper is to introduce a different and mutually compatible approach to building more reliable machine learned PDE solvers: *preserving discrete invariants*. This approach is quite natural to students of computational physics, as so much of the theory and development of standard numerical methods is related to ensuring those methods preserve the right invariants. This approach can be used with any solver that uses an update rule (including standard solvers) and is otherwise agnostic to the details of the solver.

Why do we want our machine learned solvers to preserve invariant quantities? A simple but incomplete explanation is that ML models that use physical knowledge as an inductive bias tend to outperform models that don't. Invariant preservation is, when done correctly, free lunch. We know that for a given PDE our solution should preserve certain invariants, so by enforcing those invariants at each timestep we improve the solution. A second reason has to do with *numerical stability*. By preserving the right combination of invariants, we can design machine learned PDE solvers which are numerically stable by construction. These solvers, like well-designed standard solvers, are guaranteed not to blow up as $t \to \infty$. A third reason has to do with *trust*. People are unlikely to use solvers they do not trust. People are more likely to trust a numerical method if it preserves the correct set of invariants.

Our key insight is simple: to preserve discrete invariants in machine learned PDE solvers, at each timestep apply an error-correcting algorithm to the machine learned update rule. Let us now sketch how this works. Suppose that I represent the continuous solution $\boldsymbol{u}(\boldsymbol{x}, t) \in \mathbb{R}^m$ to the PDE $\frac{\partial \boldsymbol{u}}{\partial t} + \mathcal{N}[\boldsymbol{u}] = 0$ with a discrete solution $\hat{\boldsymbol{u}}(\boldsymbol{x}, t) \in \mathbb{R}^m$ which is a linear sum of N basis functions $\boldsymbol{\phi}_k(\boldsymbol{x}) \in \mathbb{R}^m$ and N coefficients $\boldsymbol{c}_k(t) \in \mathbb{R}^m$, such that $\hat{u}_j(\boldsymbol{x}, t) = \sum_{k=1}^{N} c_{jk}(t)\phi_{jk}(\boldsymbol{x})$ for $j \in [1, \ldots, m]$. Suppose also that the machine learned update rule predicts that $\hat{\boldsymbol{u}}$ will change at a rate $\frac{\partial \hat{\boldsymbol{u}}}{\partial t}$ at time t. Suppose also that we would like the solution to satisfy L discrete invariants $\mathcal{I}_\ell(\hat{\boldsymbol{u}}, \frac{\partial \hat{\boldsymbol{u}}}{\partial t})$ for $\ell = [1, \ldots, L]$ satisfying either equalities ($\mathcal{I}_\ell = 0$) or inequalities ($\mathcal{I}_\ell \geq 0$). If $\frac{\partial \hat{\boldsymbol{u}}}{\partial t}$ does not already satisfy these discrete invariants, then we use an error correcting algorithm to modify $\frac{\partial \hat{\boldsymbol{u}}}{\partial t}$ to ensure that each of the invariants is satisfied. We repeat this process each timestep. This process is meant to used at inference, though it could also be used while training.

In this abridged version of our full-length paper (McGreivy & Hakim, 2023), we only consider scalar hyperbolic PDEs and only consider a single solver type in 1D. In section 2 we review the theory of invariant preservation for scalar hyperbolic PDEs. In section 3 we introduce an invariant-preserving error-correcting algorithm which can be used to preserve non-linear invariants in machine learned PDE solvers for scalar hyperbolic PDEs. Doing so improves reliability and guarantees numerical stability without degrading the accuracy of an already-accurate solver. The strategy we introduce – applying an error-correcting algorithm to a machine learned update rule – can be applied to a range of other solver types, to other invariant-preserving PDEs, as well as to PDEs with non-invariant terms.

2 Scalar Hyperbolic PDEs

Scalar hyperbolic PDEs can be written as follows. $u(\boldsymbol{x},t) \in \mathbb{R}$, $\boldsymbol{x} \in \Omega$, $\Omega \in \mathbb{R}^d$, and $\boldsymbol{f} \in C^1(\mathbb{R}^d)$.

$$\frac{\partial u}{\partial t} + \boldsymbol{\nabla} \cdot \boldsymbol{f}(u) = 0. \tag{1}$$

Continuous Invariants: Scalar hyperbolic PDEs have one linear invariant which is constant in time and three non-linear invariants which are non-increasing in time: (1) Total mass $\int_\Omega u\, d\boldsymbol{x}$, which is conserved in time. (2) The ℓ_p-norm $\int_\Omega |u|^p\, d\boldsymbol{x}$ for $p > 1$, which is non-increasing in time. (3) The ℓ_∞-norm $\max_{\boldsymbol{x}} u(\boldsymbol{x},t)$, which is non-increasing in time. (4) The total variation, which for continuous u in 1D is $\int_0^L \left|\frac{\partial u}{\partial x}\right| dx$. The total variation is non-increasing in time. This is usually called the total variation diminishing (TVD) property.

Finite volume (FV) method: A common approach for solving hyperbolic PDEs is by using a finite volume (FV) method. FV methods divide the spatial domain Ω into a number of discrete cells Ω_j, then use a scalar value to represent the solution average within each cell. For example, on the 1D domain $x \in [0, L]$ with uniform cell width, a FV method divides the domain into N cells of width $\Delta x = L/N$ where the left and right boundaries of the jth cell for $j = 1, \dots, N$ are $x_{j-1/2} = (j-1)\Delta x$ and $x_{j+1/2} = j\Delta x$ respectively. FV methods use a scalar value $u_j(t)$ to represent the solution average within each cell where $u_j(t) := \int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\, dx$. The 1D FV update equations for u_j are simply discrete versions of eq. (1): $\partial u_j/\partial t + (f_{j+\frac{1}{2}} - f_{j-\frac{1}{2}})/\Delta x = 0$.

Discrete Invariants: FV schemes conserve a discrete analogue of the continuous linear invariant $\int_\Omega u\, d\boldsymbol{x}$ by construction. In 1D, we can see this with a short proof: $d/dt \sum_{j=1}^N u_j \Delta x = \Delta x \sum_{j=1}^N \partial u_j/\partial t = -\sum_{j=1}^N (f_{j+1/2} - f_{j-1/2}) = f_{1/2} - f_{N+1/2}$. The rate of change of the discrete mass is equal to the flux of u through the boundaries; in a periodic system this equals 0.

Although FV schemes preserve a discrete analogue of conservation of mass by construction, they do not automatically preserve discrete analogues of any of the non-linear invariants of the continuous PDE. Instead, FV methods preserve non-linear invariants through careful choice of flux. The only known way of inheriting discrete analogues of all three non-linear invariants of eq. (1) (non-increasing ℓ_p-norm, non-increasing ℓ_∞-norm, and TVD) is to use a consistent monotone flux function while satisfying a CFL condition (Mishra et al., 2019). Unfortunately, Godunov's famous theorem from 1959 implies that monotone schemes can be at most first-order accurate (Godunov & Bohachevsky, 1959). This means that while monotone schemes preserve all the invariants of the underlying PDE, they are usually not very accurate. For scalar hyperbolic PDEs, it turns out that it is possible to design accurate and stable numerical solvers by preserving just *one* of the three non-linear invariants of eq. (1) (Durran, 1999). One such scheme is the TVD-preserving MUSCL scheme, introduced in a seminal paper by van Leer (1979).

3 Invariant-Preserving Machine Learned PDE Solvers

We introduce and demonstrate an error-correcting algorithm for scalar hyperbolic PDEs which can be used to design invariant-preserving machine learned PDE solvers. Unlike standard solvers, which put local constraints on the flux, this algorithm puts global constraints on the flux. It preserves discrete analogues of two continuous invariants: mass conservation and non-increasing ℓ_2-norm.

We consider a machine learned PDE solver which uses the 1D FV update equation with periodic boundary conditions. ML can be used to predict the flux at cell boundaries $f_{j+1/2}$. Doing so pre-

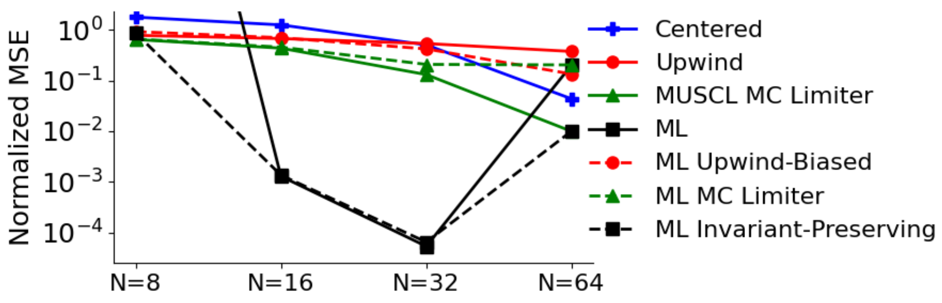

Figure 1: While there are various ways of designing invariant preserving machine learned PDE solvers, only the error-correction strategy we propose (black dotted line) preserves invariants without degrading the accuracy of an already-accurate machine learned solver (black line, $N = 16$ and $N = 32$ where N is the number of spatial grid cells). Here we compare seven different methods for solving the 1D advection equation. Three are standard numerical methods (centered flux, upwind flux, and MUSCL flux) while four are machine learned solvers. We train all the ML models with the same setup (see appendix A).

serves a discrete analogue of conservation of mass ${}^{d}/_{dt} \sum_{j=1}^{N} u_j \Delta x = 0$. A discrete analogue of the non-increasing ℓ_2-norm invariant is

$$\frac{d}{dt}\sum_{j=1}^{N}\frac{\Delta x_j}{2}u_j^2 = \sum_{j=1}^{N} u_j \frac{\partial u_j}{\partial t}\Delta x_j = -\sum_{j=1}^{N} u_j(f_{j+\frac{1}{2}} - f_{j-\frac{1}{2}}) = \sum_{j=1}^{N} f_{j+\frac{1}{2}}(u_{j+1} - u_j) \leq 0. \quad (2)$$

Let us now define ${}^{d\ell_2^{\text{old}}}/_{dt} := \sum_{j=1}^{N} f_{j+\frac{1}{2}}(u_{j+1} - u_j)$ as the original rate of change of the discrete ℓ_2-norm, and ${}^{d\ell_2^{\text{new}}}/_{dt}$ as the desired rate of change of the discrete ℓ_2-norm. To ensure non-increasing ℓ_2-norm, we want ${}^{d\ell_2^{\text{new}}}/_{dt} \leq 0$. We also define $\boldsymbol{u}_j := \{u_j\}_{j=1}^{N}$ as a vector representation of the discrete solution. We can change the time-derivative of the discrete ℓ_2-norm from ${}^{d\ell_2^{\text{old}}}/_{dt}$ to ${}^{d\ell_2^{\text{new}}}/_{dt}$ by making the following transformation to $f_{j+1/2}$:

$$f_{j+\frac{1}{2}} \Rightarrow f_{j+\frac{1}{2}} + \frac{({}^{d\ell_2^{\text{new}}}/_{dt} - {}^{d\ell_2^{\text{old}}}/_{dt})G_{j+1/2}(\boldsymbol{u}_j)}{\sum_{k=1}^{N} G_{k+1/2}(\boldsymbol{u}_k)(u_{k+1} - u_k)} \quad (3)$$

for any scalar ${}^{d\ell_2^{\text{new}}}/_{dt}$ and any non-constant, finite function $G_{j+1/2}(\boldsymbol{u}_j)$ for which $\sum_{k=1}^{N} G_{k+1/2}(\boldsymbol{u}_k)(u_{k+1} - u_k) \neq 0$. As the reader can verify by plugging eq. (3) into eq. (2), eq. (3) modifies $f_{j+1/2}$ in a way that adds a constant $({}^{d\ell_2^{\text{new}}}/_{dt} - {}^{d\ell_2^{\text{old}}}/_{dt})$ to eq. (2) via cancellation of the denominator. Note that $G_{j+1/2}(\boldsymbol{u}_j)$ is a hyperparameter that determines how each $f_{j+1/2}$ is modified and ${}^{d\ell_2^{\text{new}}}/_{dt}$ is a user-defined quantity which sets the rate of change of the discrete ℓ_2-norm. Setting $G_{j+1/2}(\boldsymbol{u}_j) = (u_{j+1} - u_j)$ corresponds to the addition of a spatially constant diffusion coefficient everywhere in space. To design an invariant-preserving machine learned PDE solver, at each timestep if ${}^{d\ell_2^{\text{old}}}/_{dt} > 0$, modify $f_{j+1/2}$ so that ${}^{d\ell_2^{\text{new}}}/_{dt} = 0$.

We train a machine learned solver to output $f_{j+1/2}$ to solve the 1D advection equation ${}^{\partial u}/_{\partial t} + c\,{}^{\partial u}/_{\partial x} = 0$ and integrate in time using a standard Runge-Kutta integration scheme. See appendix A for details. In fig. 1, we compare the accuracy of a naive machine learned solver (black solid line) and an error-corrected invariant-preserving machine learned solver (black dotted line) with standard solvers and standard approaches to preserving invariants applied to machine learned solvers. There are three key takeaways from fig. 1. First, standard approaches to preserving invariants (red and green dotted lines) degrade the accuracy of machine learned solvers too much to be accurate at large Δx. Second, the invariant-preserving algorithm we introduce does not degrade the accuracy of an already-accurate machine learned solver ($N = 16$ and $N = 32$, black lines). Third, by preserving invariants we improve the reliability of machine learned solvers ($N = 8$ and $N = 64$, black lines).

Different PDEs have different invariants. While this algorithm is designed to work for any scalar hyperbolic PDE, invariant-preserving algorithms for other PDEs must be tailored to the specific PDE.

REFERENCES

Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner. Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31):15344–15349, 2019. doi: 10.1073/pnas.1814058116. URL https://www.pnas.org/doi/abs/10.1073/pnas.1814058116. 2

Andrea Beck, David Flad, and Claus-Dieter Munz. Deep neural networks for data-driven les closure models. *Journal of Computational Physics*, 398:108910, 2019. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2019.108910. URL https://www.sciencedirect.com/science/article/pii/S0021999119306151. 2

Johannes Brandstetter, Max Welling, and Daniel E Worrall. Lie point symmetry data augmentation for neural pde solvers. *arXiv preprint arXiv:2202.07643*, 2022a. 2

Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural pde solvers, 2022b. URL https://arxiv.org/abs/2202.03376. 2

Gideon Dresdner, Dmitrii Kochkov, Peter Norgaard, Leonardo Zepeda-Núñez, Jamie A Smith, Michael P Brenner, and Stephan Hoyer. Learning to correct spectral methods for simulating turbulent flows. *arXiv preprint arXiv:2207.00556*, 2022. 2

Dale R. Durran. *Numerical methods for wave equations in geophysical fluid dynamics*. Texts in applied mathematics. Springer, New York, 1999. ISBN 0387983767. 3

N. Benjamin Erichson, Michael Muehlebach, and Michael W. Mahoney. Physics-informed autoencoders for lyapunov-stable fluid flow prediction, 2019. URL https://arxiv.org/abs/1905.10866. 2

Sergei Godunov and I Bohachevsky. Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics. *Matematičeskij sbornik*, 47(3):271–306, 1959. 3

Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Review*, 43(1):89–112, 2001. doi: 10.1137/S003614450036757X. URL https://doi.org/10.1137/S003614450036757X. 7

Daniel Greenfeld, Meirav Galun, Ronen Basri, Irad Yavneh, and Ron Kimmel. Learning to optimize multigrid PDE solvers. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2415–2423. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/greenfeld19a.html. 2

Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning neural pde solvers with convergence guarantees, 2019. URL https://arxiv.org/abs/1906.01200. 2

Alan A. Kaptanoglu, Jared L. Callaham, Aleksandr Aravkin, Christopher J. Hansen, and Steven L. Brunton. Promoting global stability in data-driven models of quadratic nonlinear dynamics. *Physical Review Fluids*, 6(9), sep 2021. doi: 10.1103/physrevfluids.6.094401. URL https://doi.org/10.1103%2Fphysrevfluids.6.094401. 2

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 7

Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. Machine learning accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21):e2101784118, 2021. doi: 10.1073/pnas.2101784118. URL https://www.pnas.org/doi/abs/10.1073/pnas.2101784118. 2

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations, 2020. URL https://arxiv.org/abs/2010.08895. 2

Björn List, Li-Wei Chen, and Nils Thuerey. Learned turbulence modelling with differentiable fluid solvers, 2022. URL `https://arxiv.org/abs/2202.06988`. 2

Ilay Luz, Meirav Galun, Haggai Maron, Ronen Basri, and Irad Yavneh. Learning algebraic multigrid using graph neural networks, 2020. URL `https://arxiv.org/abs/2003.05744`. 2

Nick McGreivy and Ammar Hakim. Invariant preservation in machine learned pde solvers via error correction, 2023. URL `https://arxiv.org/abs/2303.16110`. 3

Siddhartha Mishra, U Fjordholm, and R Abgrall. Numerical methods for conservation laws and related equations. *Lecture notes for Numerical Methods for Partial Differential Equations, ETH*, 57:58, 2019. 3

Arvind T. Mohan, Nicholas Lubbers, Daniel Livescu, and Michael Chertkov. Embedding hard physical constraints in neural network coarse-graining of 3d turbulence, 2020. URL `https://arxiv.org/abs/2002.00021`. 2

Jaideep Pathak, Mustafa Mustafa, Karthik Kashinath, Emmanuel Motheau, Thorsten Kurth, and Marcus Day. Using machine learning to augment coarse-grid computational fluid dynamics simulations, 2020. URL `https://arxiv.org/abs/2010.00072`. 2

Kimberly Stachenfeld, Drummond B. Fielding, Dmitrii Kochkov, Miles Cranmer, Tobias Pfaff, Jonathan Godwin, Can Cui, Shirley Ho, Peter Battaglia, and Alvaro Sanchez-Gonzalez. Learned coarse models for efficient turbulence simulation, 2021. URL `https://arxiv.org/abs/2112.15275`. 2

Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3424–3433. PMLR, 06–11 Aug 2017. URL `https://proceedings.mlr.press/v70/tompson17a.html`. 2

Kiwon Um, Robert Brand, Yun Fei, Philipp Holl, and Nils Thuerey. Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers, 2020. URL `https://arxiv.org/abs/2007.00016`. 2

Bram van Leer. Towards the ultimate conservative difference scheme. v. a second-order sequel to godunov's method. *Journal of Computational Physics*, 32(1):101–136, 1979. ISSN 0021-9991. doi: https://doi.org/10.1016/0021-9991(79)90145-1. URL `https://www.sciencedirect.com/science/article/pii/0021999179901451`. 3

Rui Wang, Karthik Kashinath, Mustafa Mustafa, Adrian Albert, and Rose Yu. Towards physics-informed deep learning for turbulent flow prediction, 2019. URL `https://arxiv.org/abs/1911.08655`. 2

Jiawei Zhuang, Dmitrii Kochkov, Yohai Bar-Sinai, Michael P. Brenner, and Stephan Hoyer. Learned discretizations for passive scalar advection in a two-dimensional turbulent flow. *Phys. Rev. Fluids*, 6:064605, Jun 2021. doi: 10.1103/PhysRevFluids.6.064605. URL `https://link.aps.org/doi/10.1103/PhysRevFluids.6.064605`. 2

A Details of Machine Learned Solver

We solve the 1D advection equation. We set $c = 1$ and use periodic boundary conditions. We use the continuous-time 1D FV update function and compare seven different choices for the flux at cell boundaries $f_{j+1/2}$. Because they are flux-predicting methods, all seven solvers guarantee that the discrete mass is conserved. Three are standard numerical methods:

1. The centered flux $f_{j+1/2} = \frac{u_j + u_{j+1}}{2}$. This flux conserves the discrete ℓ_2-norm.
2. The upwind flux $f_{j+1/2} = u_j$. This flux is TVD and decays the discrete ℓ_2-norm.
3. The MUSCL flux with a Monotonized Central (MC) limiter. This flux is TVD.

All three standard numerical methods (solvers 1, 2, 3) preserve one or more of the non-linear invariants and are thus numerically stable. The MUSCL scheme (solver 3) is the most accurate of the three standard numerical methods we consider. The other four methods are flux-predicting machine learned PDE solvers:

4. A machine learned solver which outputs $f_{j+1/2}$. This solver is not guaranteed to conserve any non-linear invariants.
5. An upwind-biased flux-predicting solver which outputs $\alpha_{j+1/2} \geq 0$ where $f_{j+1/2} = \alpha_{j+1/2} f^{\text{Upwind}}_{j+1/2} + (1 - \alpha_{j+1/2}) f^{\text{Centered}}_{j+1/2}$. This solver decays the ℓ_2-norm.
6. The same as solver 4, except with an MC flux limiter. This solver is TVD.
7. The same as solver 4, but using the error-correcting algorithm eq. (3) with $d\ell_2^{\text{new}}/dt \leq 0$ to ensure that the discrete ℓ_2-norm is non-increasing.

The upwind-biased solver (solver 5) and the flux-limited solver (solver 6) are examples of how standard approaches can be used to preserve invariants in machine learned PDE solvers. The error-corrected solver (solver 7) is an example of the approach proposed in this paper. In fig. 1, we plot the normalized mean squared error (MSE) for all seven solvers averaged over time from $t = 0$ to $t = 1$ averaged over 25 samples drawn from the training distribution. We compare solvers with N grid cells, where $N = 8$, 16, 32, and 64.

The machine learned solver uses the continuous-time 1D FV update equation. We use a SSPRK3 ODE integrator Gottlieb et al. (2001). We choose the timestep Δt using a CFL condition with a CFL number of 0.3. The initial condition is drawn from a sum-of-sines distribution

$$u_0(x) = \sum_{i=1}^{N^{\text{modes}}} A_i \sin\left(2\pi k_i x + \phi_i\right)$$

where $N^{\text{modes}} \sim \{1, 2, 3, 4, 5, 6\}$ and $k_i \sim \{1, 2, 3, 4\}$ are uniform draws from a set while $A_i \sim [-1.0, 1.0]$ and $\phi_i \sim [0, 2\pi]$ are draws from uniform distributions. The loss function L is given by computing the mean squared error (MSE) between the predicted time-derivative and the so-called 'exact' time-derivative

$$L = \frac{1}{N_x} \sum_{j=1}^{N_x} \left(\frac{du_j(t)}{dt} - \frac{du_j^{\text{exact}}(t)}{dt} \right)^2 .$$

Both the 'exact' solution $u_j^{\text{exact}}(t)$ and the 'exact' time-derivative du_j^{exact}/dt are coarse-grained versions of a high-resolution simulation u^{exact}, i.e., $u_j^{\text{exact}}(t) = \int_{x_{j-1/2}}^{x_{j+1/2}} u^{\text{exact}}(x,t)dx$. For each sample from the distribution of initial conditions, we take 50 snapshots evenly spaced in time from $t \in [0, 1]$. We draw 100 samples, for a total of 5000 snapshots in our training dataset. For each snapshot we store the exact trajectory u_j^{exact} and the exact time-derivative du_j^{exact}/dt. Our machine learning models are periodic convolutional neural networks (CNNs) which are given the downsampled exact trajectories $u_j^{\text{exact}}(t)$ as inputs. Solvers 4, 6, and 7 output the flux $f_{j+1/2}$, while solver 5 outputs $\alpha_{j+1/2}$. These outputs are then used to compute the predicted time-derivative $du_j(t)/dt$. All models train with a batch size of 32 and use the ADAM optimizer Kingma & Ba (2014) for 100 epochs over the training dataset with a learning rate of 1×10^{-3}, followed by another 100 epochs with a learning rate of 1×10^{-4}. Solver 7 uses $G_{j+1/2}(\boldsymbol{u}_j) = u_{j+1} - u_j$.