

## Learned discretizations for passive scalar advection in a two-dimensional turbulent flow

Jiawei Zhuang,<sup>1,2</sup> Dmitrii Kochkov,<sup>2</sup> Yohai Bar-Sinai ,<sup>1,3,\*</sup>  
Michael P. Brenner ,<sup>1,2</sup> and Stephan Hoyer <sup>2</sup>

<sup>1</sup>*School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts 02138, USA*

<sup>2</sup>*Google Research, 1600 Amphitheatre Pkwy., Mountain View, California 94043, USA*

<sup>3</sup>*Google Research, Tel-Aviv 67891, Israel*



(Received 12 April 2020; accepted 19 April 2021; published 14 June 2021)

The computational cost of fluid simulations increases rapidly with grid resolution. This has given a hard limit on the ability of simulations to accurately resolve small-scale features of complex flows. Here we use a machine learning approach to learn a numerical discretization that retains high accuracy even when the solution is under-resolved with classical methods. We apply this approach to passive scalar advection in a two-dimensional turbulent flow. The method maintains the same accuracy as traditional high-order flux-limited advection solvers, while using  $4\times$  lower grid resolution in each dimension. The machine learning component is tightly integrated with traditional finite-volume schemes and can be trained via an end-to-end differentiable programming framework. The solver can achieve near-peak hardware utilization on CPUs and accelerators via convolutional filters.

DOI: [10.1103/PhysRevFluids.6.064605](https://doi.org/10.1103/PhysRevFluids.6.064605)

### I. INTRODUCTION

A key problem in the numerical simulation of complex phenomena is the need to accurately resolve spatiotemporal features over a wide range of length scales. For example, the computational requirement for simulating a high Reynolds number fluid flow scales like  $Re^3$ , implying that a tenfold increase in Reynolds number requires a 1000-fold increase in computing power. Over the past decades, the extra computing power made available through Moore's law has been used to increase grid resolution dramatically, leading to breakthroughs in turbulence modeling [1], weather prediction [2], and climate projection [3]. Nonetheless, there is still a formidable gap towards resolving the finest spatial scales of interest [4], especially with the recent slowdown of Moore's law [5,6]. Machine learning has given a potential way out of this conundrum, by training low-resolution models to learn the rules from their high-resolution counterparts [7–10]. The learned models aim to produce high-fidelity simulations using much less computational resources. Incorporating machine learning into numerical models also facilitates the adoption of emerging hardware, considering that the fastest growth in computing power now relies on domain-specific architectures such as graphical processing units (GPUs) [11] and tensor processing units (TPUs) [12,13] that are optimized for machine learning tasks.

---

\*Present address: The School of Physics and Astronomy, Tel Aviv University, Tel Aviv 69978, Israel.

Recently we introduced *data-driven discretizations* [14] to learn numerical methods that achieve the same accuracy as traditional finite difference methods but with much coarser grid resolution. These methods are equation specific, and require training a coarse resolution solver with high-resolution ground truth simulations. Since the dynamics of a partial differential equation is entirely local, the high-resolution simulations can be carried out on a small domain. We demonstrated the method with a set of canonical one-dimensional equations, demonstrating a 4–8× upscaling of effective resolution [14]. Here we extend this methodology to two-dimensional advection of passive scalars in a turbulent flow, a canonical problem in physics [15], and a classic challenge in atmospheric modeling [16]. We show that machine-learned advection solver can use a grid with 4× coarser resolution than classic high-order solvers while still maintaining the same accuracy. Code and tutorials for this work are available in [17].

## II. DATA-DRIVEN SOLUTION TO ADVECTION EQUATION

### A. Advection equation

We consider the advection of a scalar concentration field  $C(\vec{x}, t)$  under a specified velocity field  $\vec{u}(\vec{x}, t)$ :

$$\frac{\partial C}{\partial t} + \nabla \cdot (\vec{u}C) = 0. \quad (1)$$

If the prescribed velocity field is divergence-free

$$\nabla \cdot \vec{u} = 0, \quad (2)$$

then, Eq. (1) reduces [18] to

$$\frac{\partial C}{\partial t} + \vec{u} \cdot \nabla C = 0. \quad (3)$$

A classical Eulerian scheme uses discretizations of the spatial derivative  $\frac{\partial C}{\partial x}$ , often in a form of

$$\frac{\partial C}{\partial x} \Big|_{x=x_i} = \sum_{j=-k}^k \alpha_j C_{i+j}, \quad (4)$$

where  $\{x_1, \dots, x_N\}$  is the spatial grid points,  $C_j$  is the concentration at point  $x_j$ , and  $\{\alpha_{-k}, \dots, \alpha_k\}$  are predefined finite-difference coefficients. For example, a first-order forward difference  $\frac{C_{i+1}-C_i}{\Delta x}$  (where  $C_{i+1}$  is in the upwinding direction) leads to the upwind scheme. Sophisticated high-order methods with flux limiters will choose different coefficients depending on local fields [19]. Extension to two dimensions can be done by either operator splitting (solve for each dimension separately) [20] or a true two-dimensional discretization [21].

Although high-order Eulerian schemes are highly accurate under idealized flows [22], their accuracy breaks down to first order under turbulent or strongly sheared flows, resulting in significant numerical diffusion [16]. Adaptive mesh refinement can reduce such numerical diffusion [23], but increases software complexity. Lagrangian methods avoid numerical diffusion [24], but have inhomogeneous spatial coverage and also difficulties in dealing with nonlinear chemical reaction [25]. Semi-Lagrangian approaches involve remapping from a distorted Lagrangian mesh to a regular Eulerian mesh [26], and such remapping step exhibits similar numerical diffusion as Eulerian methods. Flow-map approaches [27] can achieve Lagrangian-like accuracy on a Eulerian mesh, but need to solve for the advection trajectory over multiple steps and requires a special treatment to incorporate additional terms (e.g., chemical reaction) between advection steps. Different from existing methods, here we aim to develop an ultra-accurate advection solver under the requirements of (1) a strictly Eulerian framework on a fixed grid, (2) explicit time-stepping, and (3) only relying on the current state to predict the next time step.

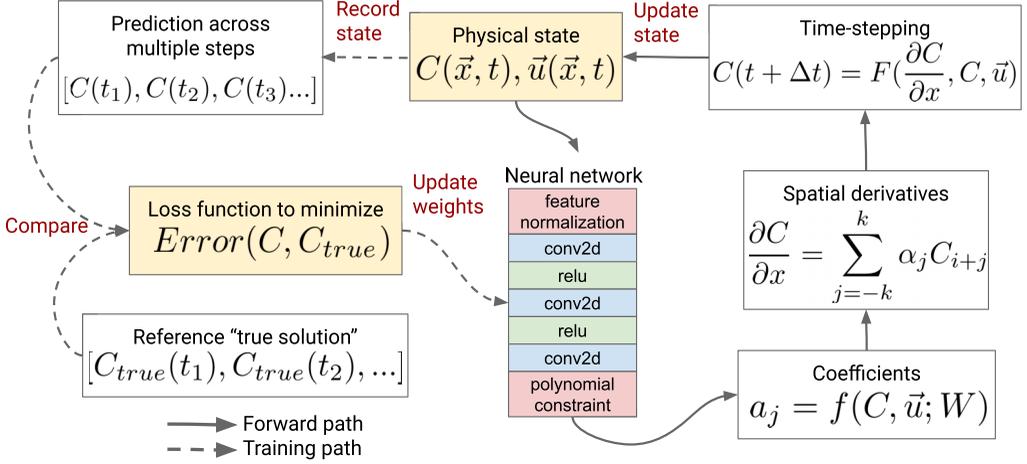


FIG. 1. End-to-end learning framework with differential programming. During training, the model is optimized to predict future concentrations across multiple time steps, based on a precomputed dataset of snapshots from high-resolution simulations. During inference, the optimized model is repeatedly applied to predict time evolution. The neural network component contains a stack of 2D convolutional layers with ReLU activation functions (degraded to 1D convolution for 1D problems). Physical constraints are imposed before and after the convolutional layers (Sec. IID). In the “Time-stepping” block,  $H$  is the advection operator that computes the concentration update based on the machine learning estimate of spatial derivatives.

## B. Learning optimal coefficients

Instead of using predefined rules to compute finite-difference coefficients [Eq. (4)], our data-driven discretizations [14] predict the local-field-dependent coefficients  $\vec{\alpha} = \{\alpha_{-k}, \dots, \alpha_k\}$  via a convolutional neural network:

$$\vec{\alpha} = f(C, \vec{u}; W). \quad (5)$$

The coefficients  $\vec{\alpha}|_{x=x_j}$  depend on the local environment around  $x_j$ , with the inputs to the neural network being the neighboring fields  $\{C_j, C_{j\pm 1}, \dots\}$  and  $\{\vec{u}_j, \vec{u}_{j\pm 1}, \dots\}$ . For simplicity of presentation, here we use 1D indices  $\{j, j \pm 1, \dots\}$  to denote spatially adjacent points. For two-dimensional (2D) advection problems, this computation involves 2D convolution across both  $x$  and  $y$  dimensions. We learn the neural network weights  $W$  by minimizing the difference between the machine learning prediction and the true solution.

Figure 1 shows the forward solver workflow and training framework. During the forward solve, we replace the computation of finite-difference coefficients with a convolution neural network, while still using classic approaches for the rest of the steps (computing the advection flux and doing the time-stepping). During training, we accumulate the forward solver prediction results over ten time steps and then compare to the reference solution over this time period, by computing the mean absolute error (MAE) over the entire spatial domain between the two time series:

$$MAE = \frac{1}{NM} \sum_{i=1}^N \sum_{j=1}^M |C_j^{\text{predict}}(t_i) - C_j^{\text{true}}(t_i)|. \quad (6)$$

The MAE is used as the loss function for neural network training [28]. We find that using this multistep loss function (as opposed to a single time step) stabilizes the forward integration, similar to the findings by [29]. In our experiments, we found using MAE resulted in slightly more accurate predictions than using mean square error, but the difference was not large.

The training of a neural network inside a classic numerical solver is made possible by writing the entire program in a differentiable programming framework [30], which allows efficient gradient-based optimization of arbitrary parameters in the code using automatic differentiation (AD) [31]. AD tools have a long history, dating back to FORTRAN 77 [32]. Recent developments of AD frameworks, such as TENSORFLOW [33], PYTORCH [34], JAX [35], FLUX.JL [36], and SWIFT [37], are even easier to program and support hardware accelerators such as GPUs and TPUs. Those developments make it easier to incorporate machine learning into scientific computing code (e.g., [38]). We implemented our advection solver in TENSORFLOW EAGER [39].

### C. Baseline solver and reference solution

As a baseline method, we use the second-order Van Leer advection scheme with a monotonic flux limiter [40]. To obtain the reference “true” solution, we run the baseline advection solver at sufficiently high resolution to ensure the solution has converged. We then down-sample the high-resolution results using conservative averaging, to produce the training and test datasets for our machine learning-based model on a coarse grid.

We remark that although higher-order schemes with more advanced limiters would be more accurate, any flux-limited high-order schemes break to first order under turbulent flows in order to ensure monotonicity [16]. Starting from second order, increasing the spatial resolution is generally more effective than further improving the solver order or the limiter [41,42].

### D. Physical constraints

There is growing emphasis on embedding physical constraints into the design of machine learning methods. This is typically done either by adding “soft” constraints as terms the loss function [43,44], or “hard” constraints in the model architecture [14,45–49]. Since here we only replace a small component in the numerical solver with machine learning, we can impose arbitrary physical constraints before and after the neural network components. Using hard constraints allows the machine learning algorithm to focus on approximation problems, by imposing physical consistency requirements by construction. In particular, we require the following:

(1) *Finite-volume representation for mass conservation.* We compute the flux across grid cell boundaries, and then apply the flux to update the concentration fields  $C_i$ . This ensures that mass is exactly conserved. The machine learning estimate of spatial derivatives  $\frac{\partial C}{\partial x}$  is used for obtaining the optimal interpolation values  $C_{i+(1/2)}$  at cell boundaries, which is then used for calculating the flux via  $u_{i+(1/2)}C_{i+(1/2)}$ .

(2) *Polynomial accuracy constraints.* Following [14], we can force the machine learning-predicted coefficients to satisfy an  $m$ th-order polynomial constraint, so that the approximation error decays as  $O(\Delta x^m)$ . This ensures that if the learned discretization is fit to solutions that are smooth on the scale of the mesh, we will recover classical finite-difference methods. In our experiments, we find that a first-order constraint gives the best result on coarse grids. This preserves a balance between accuracy constraints and model flexibility that may be particularly valuable in nonmonotonic regions, where higher order advection schemes often revert to first-order upwinding [19]. First-order accuracy requires  $\sum_{j=-k}^k \alpha_j = 0$ , and can be enforced by applying an affine transformation to the original neural network output (our implementation), or by having the neural network only output  $\{\alpha_{-k}, \dots, \alpha_{k-1}\}$  and solving for the last  $\alpha_k$ . We choose the constant vector in the affine transformation to match a centered, first-order scheme (equal weight on the two nearest grid cells). Accordingly, our randomly initialized neural net at the start of training produces interpolation coefficients that are very close to a centered, first-order scheme.

(3) *Input feature normalization.* Before feeding the current concentration field  $C$  to the neural network, we normalize it globally to  $[0,1]$ . This ensures that the overall magnitude of the concentration does not affect the prediction of finite-difference coefficients, and thus our solver satisfies the “semilinear” requirement for advection schemes that  $H(aC + b) = aH(C) + b$  where  $H$  is the

advection operator and  $\{a, b\}$  are constants (Eqs. 2.12 and 2.13 of [20]). Without such normalization, we find that the trained model diverges quickly during the forward integration.

### E. Other choices of learned terms

Our training framework can be easily adapted to learn other parameters besides the finite-difference coefficients. In this section, we describe other approaches that we experimented with but did not choose.

Numerical methods introduce artificial numerical dissipation, so it is natural to consider adding explicit corrections to diffusion. One of the earliest flux-correct transport algorithms [50] includes an antidiffusion coefficient of  $1/8$  as a correction term, though the choice of  $1/8$  was subjective and it was later acknowledged that such correction should better be velocity- and wave-number dependent [51]. We considered learning diffusive correction directly, in the form

$$\frac{\partial C}{\partial t} + \nabla \cdot (\bar{u}C) + \left( D_{xx} \frac{\partial^2 C}{\partial x^2} + D_{xy} \frac{\partial^2 C}{\partial x \partial y} + D_{yy} \frac{\partial^2 C}{\partial y^2} \right) = 0, \quad (7)$$

where the (anti-)diffusion coefficients  $\bar{D} = \{D_{xx}, D_{xy}, D_{yy}\}$  are computed by a convolutional neural network  $\bar{D} = f(C, \bar{u}; W)$ , while the advection-diffusion equation itself is still solved by a traditional high-order finite volume method. The idea resembles learning the Reynolds stress tensor [10] in a Reynolds-averaged Navier-Stokes simulation. As in Sec. II B, here the neural network is trained by minimizing the difference between the model prediction and the reference solution. In practice, we found that this learned diffusion model achieves about  $3\times$  upscaling compared to the second-order baseline solver, but performs slightly worse than our original approach of learning finite-difference coefficients (Sec. II B) that can achieve  $4\times$  upscaling.

We also experimented with other learned terms, including (1) a pure machine learning approach, by having the neural network directly predict the concentration at the next step  $C(t + \Delta t)$  based on the current state  $C(t)$  and  $\bar{u}(t)$ ; and (2) having the neural network directly predict the spatial derivative  $\frac{\partial C}{\partial x}$  instead of the finite-difference coefficients  $\bar{a}$  that need to be further multiplied with the concentration field  $C$  to obtain the spatial derivative. We found those methods to be unstable due to the lack of physical constraints (Sec. II D).

## III. NUMERICAL RESULTS

We apply the data-driven discretization to one- or two-dimensional advection. Two-dimensional advection is highly relevant for atmospheric modeling, as the vertical dimension can be decoupled from the horizontal dimensions and solved independently [20].

The performance of our learned advection solver (the “neural network model” hereafter) depends on the hyperparameters of the convolutional neural network component. For simplicity, this section only presents the results with the default hyperparameter configuration. For 1D problems, we use 4 convolutional layers and 32 filters in each layer; For 2D problems, we use 10 convolutional layers and 128 filters in each layer. All cases use a three-point finite-difference stencil [ $k = 1$  in Eq. (4)]. The impact of hyperparameters on model accuracy and computational speed is further examined in Sec. IV. We use the Adam optimizer [52] with default parameters for neural network training. Our simple convolutional neural network architecture already achieves a high accuracy, without additional operations such as residual connections and batch normalization.

### A. 1D advection under constant velocity

We first show that our neural network model can achieve a near-perfect result for a canonical test problem: 1D advection constant velocity [40]. We consider a periodic 1D grid of 32 grid points. The concentration field is shifted by a constant distance per time step, determined by the Courant-Friedrichs-Lewy (CFL) number  $\frac{u\Delta t}{\Delta x}$ . We set  $\text{CFL} = 0.5$  ( $\Delta x = 1$ ,  $\Delta t = 0.5$ ,  $u = 1$ ), so that the

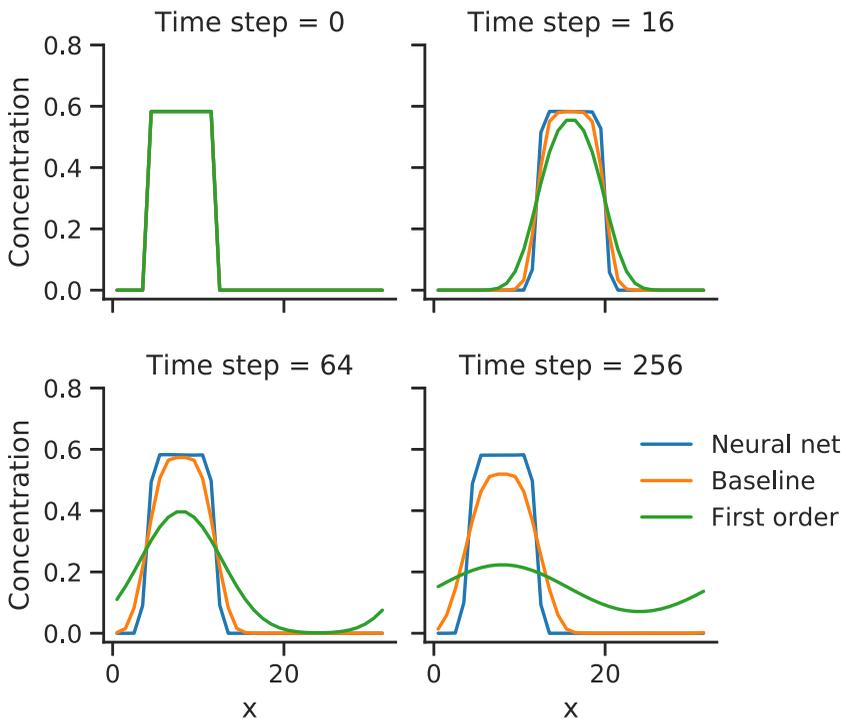


FIG. 2. One test sample for 1D advection under constant velocity. The concentration field is advected by a half grid box every time step, and returns to the original position after every 64 time steps because the domain is periodic. Our neural network model is able to maintain the initial shape indefinitely, while traditional solvers accumulate numerical diffusion over time.

concentration field is shifted by a half grid box every time step, and returns to the original position after every 64 time steps.

To generate training data, we initialize 30 square waves with heights randomly sampled from  $[0.1, 0.9]$  and widths from two to eight grid points. Test data are randomly sampled from the same range of width and height. The reference “true” solution is generated by the baseline solver at  $8\times$  resolution (256 grid points) and down-sampled to the original coarse grid.

Figure 2 shows one test sample during the forward integration. The first-order upwind scheme exhibits large numerical diffusion, due to its second-order spatial discretization error [53]. The second-order Van Leer scheme (our baseline) is more accurate but still accumulates diffusion over time. In contrast, our neural network model closely tracks the reference “true solution” obtained by the  $8\times$  resolution baseline. When a slight numerical diffusion occurs at one step, the next step applies a slight antidiffusion to correct it. Intuitively, the solver learns that the optimal solution in one-dimensional advection is to maintain the initial shape.

Figure 3 shows the mean absolute error over time, averaged over all test samples. The error indicates the deviation from the reference solution obtained by the baseline solver at 256 grid points. The neural network model achieves a factor of 8 less error than the baseline second-order Van Leer scheme.

We further investigate this intriguing behavior of our neural network model using out-of-sample test data. As shown in Fig. 4, when the model (trained on square waves) is applied to Gaussian initial conditions, it gradually turns Gaussian waves into squares, which are the only shape in the training data. Then, the model can maintain the squares indefinitely. Such phenomenon of “turning other shapes to squares” also exists in manually designed schemes that are overly optimized for

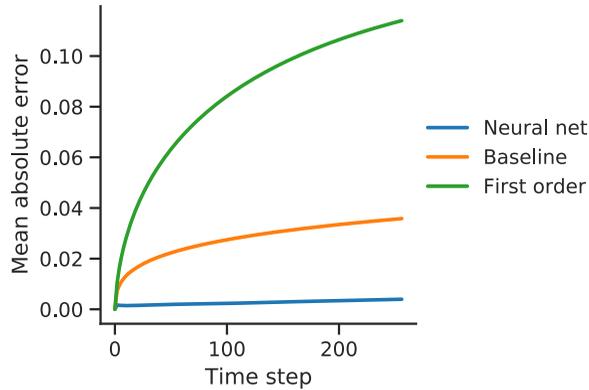


FIG. 3. Error for 1D advection on test data. Here we only plot even time steps (0, 2, 4, ...) for a smooth curve, because the error oscillates between odd and even time steps (a result of  $CFL = 0.5$ ).

square waves [51]. The overfitting problem here can be easily fixed by adding Gaussian shapes into training data; after that the neural network model can track both Gaussian and square shapes perfectly. Given that the input features for convolutional neural networks are localized in space, covering representative input patterns only requires limited amounts of training data.

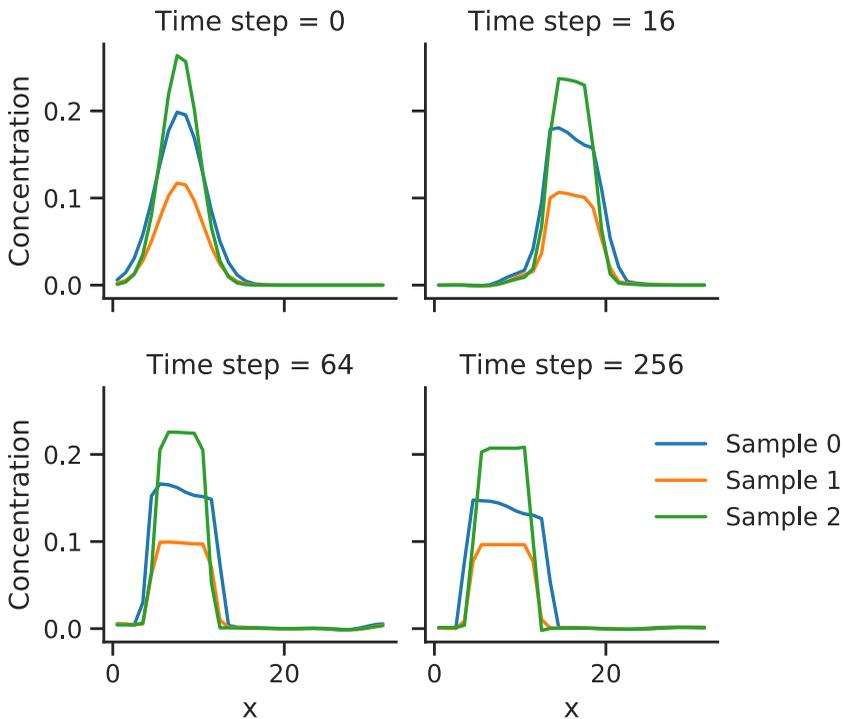


FIG. 4. Neural network prediction on out-of-sample data. The neural network model is only trained on square waves, but applied to Gaussian initial conditions. The model gradually turns Gaussian waves into squares, and then maintains the squares indefinitely.

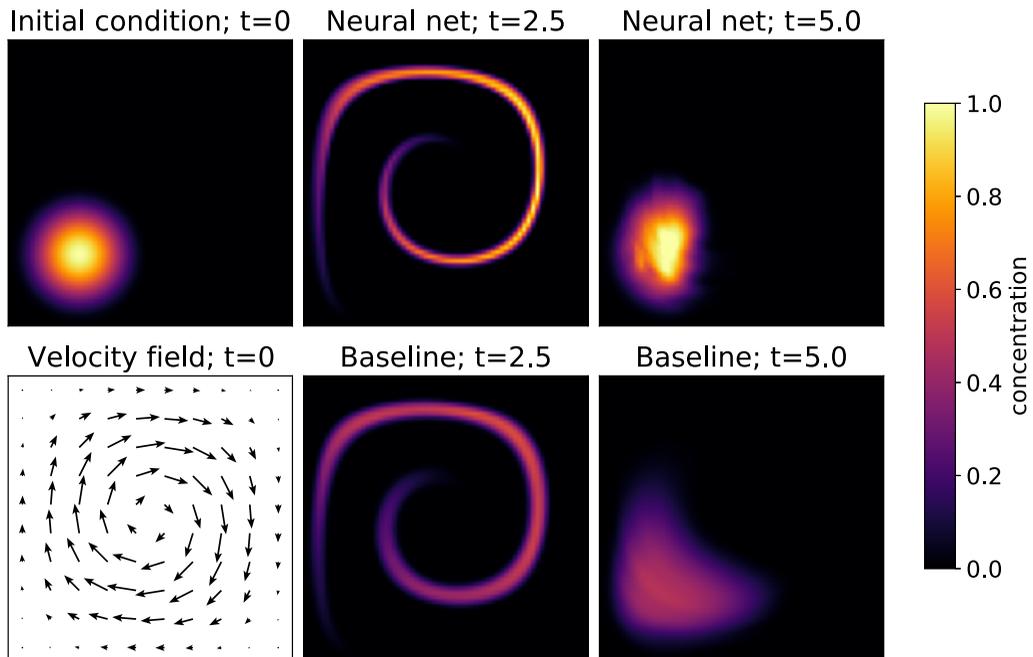


FIG. 5. Result on 2D deformational flow. The flow reverses at  $t = 2.5$  and returns to the initial condition at  $t = 5$ . The neural network model is able to maintain a sharp gradient, while the baseline model incurs large numerical diffusion. The spatial domain is  $[0, 1] \times [0, 1]$  (not plotted on axis).

### B. 2D deformational flow

We next demonstrate that our neural network model can also achieve a near-perfect result for a 2D deformational flow test, originally proposed by [18] and later extended to spherical coordinates as a standard test for atmospheric advection schemes [54,55]. The spatial domain is a square  $[0, 1] \times [0, 1]$ , and the velocity field is a periodic swirling flow:

$$\begin{aligned} u(x, y, t) &= \sin^2(\pi x) \sin(2\pi y) \cos(\pi t/T), \\ v(x, y, t) &= \sin^2(\pi y) \sin(2\pi x) \cos(\pi t/T), \end{aligned} \quad (8)$$

where the period  $T = 5$  in our setup. The direction of this flow reverses at  $t = (n - \frac{1}{2})T$  for any positive integer  $n$ . The exact solution at  $t = nT$  is identical to the initial condition.

The initial concentration field is a blob centered at  $[1/4, 1/4]$ :

$$\begin{aligned} C(x, y) &= \frac{1}{2}[1 + \cos(\pi r)], \\ r(x, y) &= \min\left[1, 4\sqrt{(x - 1/4)^2 + (y - 1/4)^2}\right]. \end{aligned} \quad (9)$$

The model is not directly trained on this deformational flow, but instead on an ensemble of periodic, divergence-free, random velocity fields, implemented as superpositions of sinusoidal waves as described by [56]. The trained model is able to generalize across different flows as long as the training data contain representative local patterns.

Figure 5 shows the advection under deformational flow for the baseline and the neural network models, both on  $64 \times 64$  grid points. The time step is chosen so that the maximum CFL number is 0.5. The neural network model is able to maintain a sharp concentration gradient, while the baseline Van Leer scheme incurs large numerical diffusion when the initial blob is stretched to a thin filament [16].

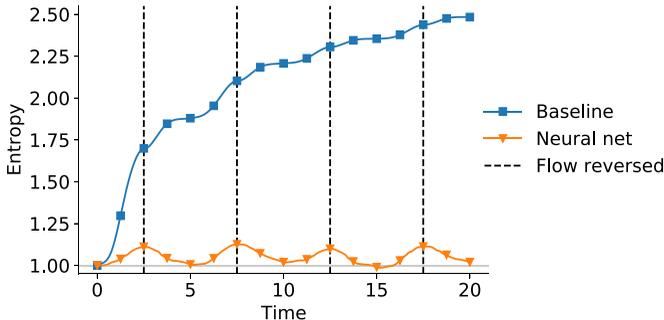


FIG. 6. Entropy for advection under 2D deformational flow. Entropy is conserved under pure advection and increases under diffusion. Traditional monotonic solvers are only allowed to increase entropy, while our neural network model is allowed to decrease entropy and thus minimizes diffusion error over a long time.

To quantify the numerical diffusion, we use the entropy  $S$  as a metric [57]:

$$S = -\beta \sum_i C_i \log(C_i), \quad (10)$$

where the concentration  $C$  is scaled such that the initial conditions fall in the range  $[0, 1]$ , and  $\beta$  is a normalization factor so that the initial entropy is 1. Entropy is conserved under pure advection and increases under diffusion. To avoid an undefined answer if any  $C_i < 0$ , we first set negative values of the concentration to zero, and evaluate  $C = 0$  via the limit  $x \log x = 0$  as  $x \rightarrow 0$ .

Figure 6 shows the entropy over time. Any monotonic advection solver can only increase entropy; any entropy decrease indicates nonphysical antidiffusion, which often occurs due to numerical instability. Strikingly, the neural network model *can* decrease entropy, while still remaining numerically stable. Although such behavior seems to be nonphysical, it is indeed the best possible solution on such a coarse grid. On a grid that perfectly resolves the concentration field, the entropy remains constant under the deformational flow. Yet on a coarse grid view, the computed entropy increases when the initial blob turns into filament due to conservative averaging, and then decreases when the filament reverts back into a blob. Our neural network model can disobey the commonly used constraint of nondecreasing entropy, and thus more closely matches the exact solution, when compared to traditional monotonic solvers.

### C. 2D turbulent flow

As the final test, we use the velocity fields from freely evolving, decaying 2D turbulence simulations in PYQG [58]. The spatial domain is  $[0, 2\pi] \times [0, 2\pi]$  with periodic boundary condition. We use a  $256 \times 256$  grid for generating the reference solution using the baseline solver, and a  $32 \times 32$  coarse grid for model evaluation. As in previous cases, here the advection time step is chosen so that the maximum CFL number is 0.5.

The training and test velocities are generated from different random seeds. We start with the McWilliams-84 random initial condition [59] and let the turbulence decay with time. We discard the initial 4 s of the simulation so that the velocity field can be resolved on the coarse grid. For the initial concentration field, we use an ensemble of ten blobs with width 0.5 at random locations. Note that the spatial scale of the concentration field under turbulent advection can become much smaller than the scale of the velocity field [15,60], making it challenging for traditional advection solvers to resolve the concentration gradient. We use 20 random initial conditions for training data and 20 for test data. The actual sample size for the training dataset is much larger, as each initial condition is integrated into a time series of 1024 steps on the fine grid or 128 steps on the coarse grid, which is further broken into many ten-step time series for calculating the multistep loss function.

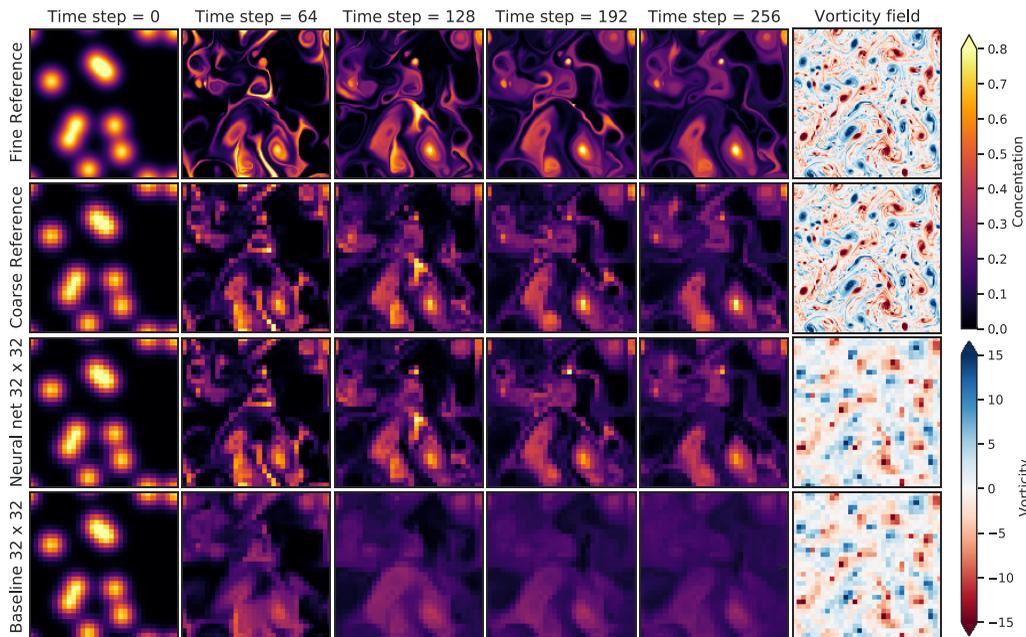


FIG. 7. One test sample under 2D turbulent flow. The initial blobs (first column) are stretched into thin filaments under the turbulent flow (last column), illustrated by the vorticity  $\omega = \partial_x u_y - \partial_y u_x$  (rightmost column). The baseline solver (second-order Van Leer scheme) can resolve such filaments on the fine-resolution grid, but incurs large numerical diffusion on the coarse grid. The neural network model can preserve the sharp features on the coarse grid. The spatial domain is  $[0, 2\pi] \times [0, 2\pi]$  (not plotted on axis).

Figure 7 shows one test sample under the 2D turbulent flow, for both the initial condition (the left column of Fig. 7) and the integration results after 256 time steps (the middle and right columns of Fig. 7). Note that this is twice the maximum number of time steps used for model training. The initial blobs are stretched into thin filaments under the turbulent flow. The baseline solver (second-order Van Leer scheme) can resolve such filaments on the fine-resolution grid, but incurs large numerical diffusion on the coarse grid and loses the sharp concentration gradient. However, when the fine-grid solution is directly resampled to the coarse grid, most sharp features can actually be preserved. Thus, the inability to resolve sharp gradients is not due to the coarse grid itself, but instead due to the numerical error in the baseline advection solver. Our neural network model, trained to track the optimal reference solution on the coarse grid, is able to preserve sharp features during the forward integration. The model performs well on all test samples, with more shown in the Appendix.

Figure 8 shows a variety of error metrics for advection under turbulent flow, averaged over all test samples. The error is computed as the deviation from the reference solution obtained by the baseline solver at the  $256 \times 256$  grid. We also compare the baseline solver at intermediate grid resolutions ( $64 \times 64$  and  $128 \times 128$ ). All solutions are resampled to the  $32 \times 32$  coarse grid for error calculation. We use two measurements of accuracy, mean absolute error (our training loss) and the coefficient of determination  $R^2$ , which means the goodness of fit for linear regression models for the reference solution. Based on these metrics, our neural network model achieves roughly the same accuracy as the baseline method at  $4 \times$  resolution ( $128 \times 128$ ). We also evaluate the entropy for all solutions based on Eq. (10), which suggests that from a statistical perspective our neural net model almost perfectly matches the reference simulation on which it was trained.

Figure 9 illustrates the limitations of stability and generalization for our neural net model by integrating for far longer than the 128 time steps used for training data. After 1000 time integration

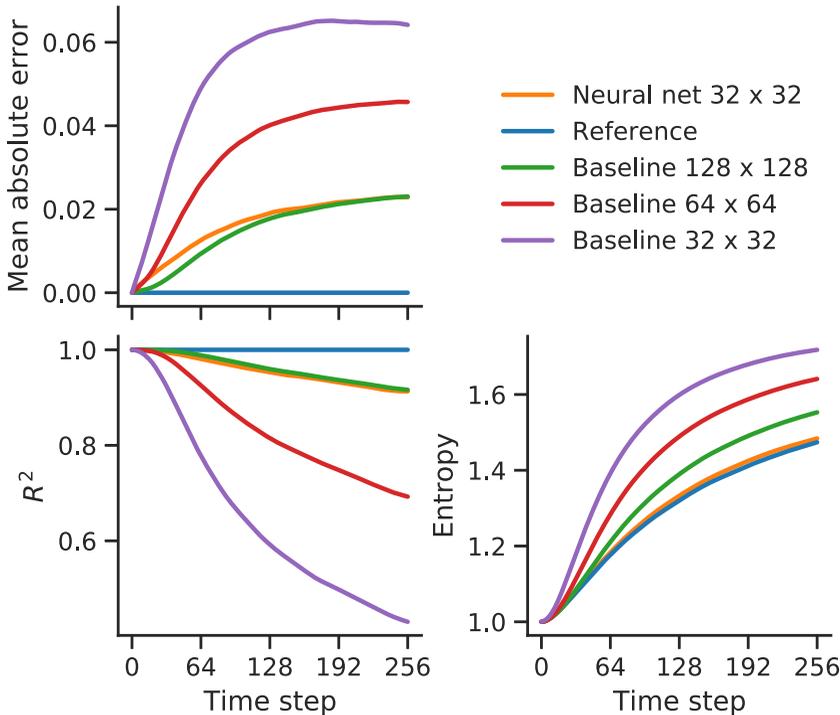


FIG. 8. Error for 2D turbulent advection on test data. The neural network model achieves the same accuracy as the second-order Van Leer baseline scheme at  $4\times$  resolution, and entropy similar to the baseline at  $8\times$  resolution.

steps, our neural net model shows obvious numerical artifacts (checkerboard patterns) and very poor accuracy for about 10% of random seeds. Unlike the baseline models, our neural net model does not guarantee properties such as monotonicity, and when presented with examples outside of its training data it occasionally extrapolates poorly. Figuring out how to guarantee stability for neural net models, either by training on more comprehensive datasets or by imposing architecture constraints, is an important topic for future work.

Finally, to get a glimpse into the inner workings of the trained model, Fig. 10 examines predicted interpolation coefficients for the  $x$  component of the velocity field. We see that similar to handcrafted methods, the learned interpolation depends on both velocity and concentration. While some of the symmetries have been clearly learned from the data, we believe that incorporating such priors could improve the results further.

#### IV. COMPUTATIONAL PERFORMANCE AND ACCURACY WITH DIFFERENT HYPERPARAMETERS

There is a trade-off between accuracy and speed for our neural network model, as using a larger convolutional neural network increases both the accuracy and the run time. We performed a grid search on model hyperparameters, for the number of layers ranging from [4, 6, 8, 10], the number of convolutional filters ranging from [16, 32, 64, 128], and the finite-difference stencil size ranging from [3, 5], with each case replicated three times with different random seeds. The model accuracy is evaluated on the 2D turbulence case in Sec. III C, and the run time is measured on a single Nvidia V100 GPU.

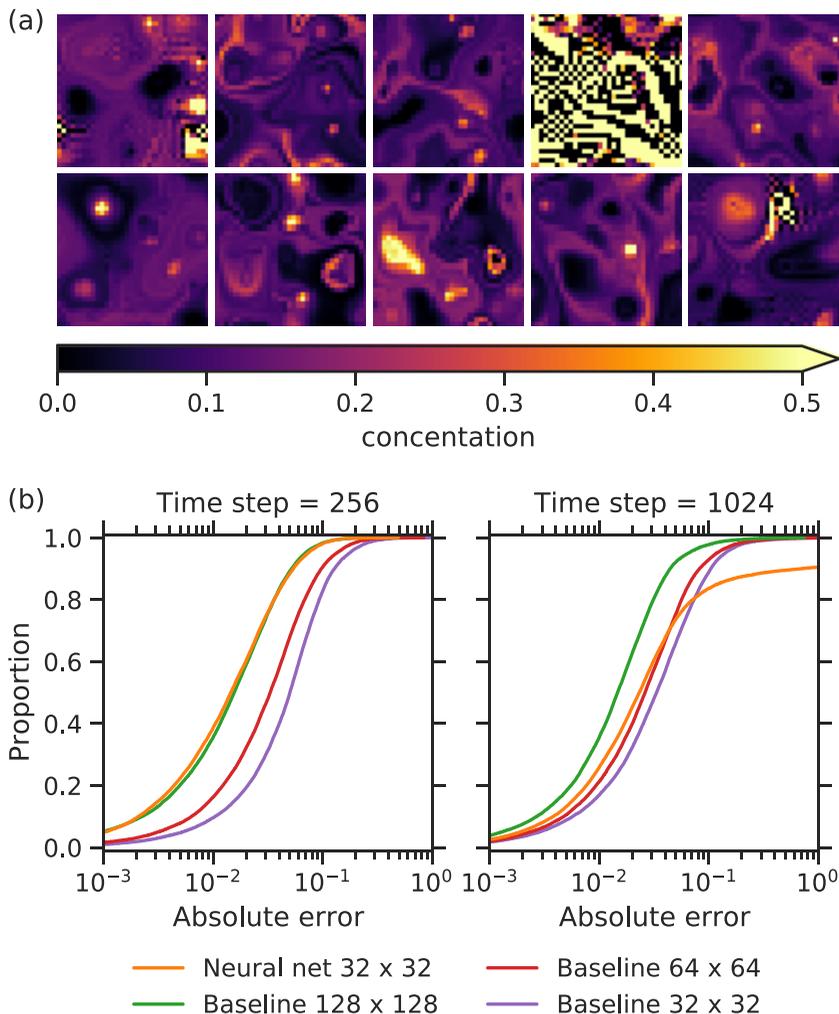


FIG. 9. Limitations of long-time stability under 2D turbulent flow. (a) Ten randomly chosen examples of concentration fields from the neural net model after 1024 time steps. One field (first row, fourth column) is entirely covered by “checkerboard” artifacts, and two others (top left and bottom right) show checkerboard artifacts in limited regions. (b) Empirical distribution function for absolute error across all models after 256 and 1024 time steps. The neural net performance suffers significantly, with about 10% of solutions having an absolute error greater than 1.

Figure 11 shows the model accuracy and speed using different hyperparameters. The performance of the baseline solver at intermediate grid resolutions ( $64 \times 64$  and  $128 \times 128$ ) is overlaid for comparison. A large neural network (8–10 layers and 128 filters) achieves comparable accuracy and speed as the baseline solver at  $4\times$  resolution, while a small neural network (4 layers and 32 filters) performs similarly to the baseline solver at  $2\times$  resolution. Figure 12 shows that using 64 filters and 10 layers achieves a good balance between accuracy and speed, in which case the model achieves similar accuracy as the  $4\times$  resolution baseline while being 80% faster.

The model speed largely depends on the code implementation and software configuration. Our current implementation of the neural network model has a lot of room for performance optimization. For example, our code still requires unnecessary, large memory allocation, and does not use the reduced-precision tensor cores in the V100 GPU. With more performance tuning as well as

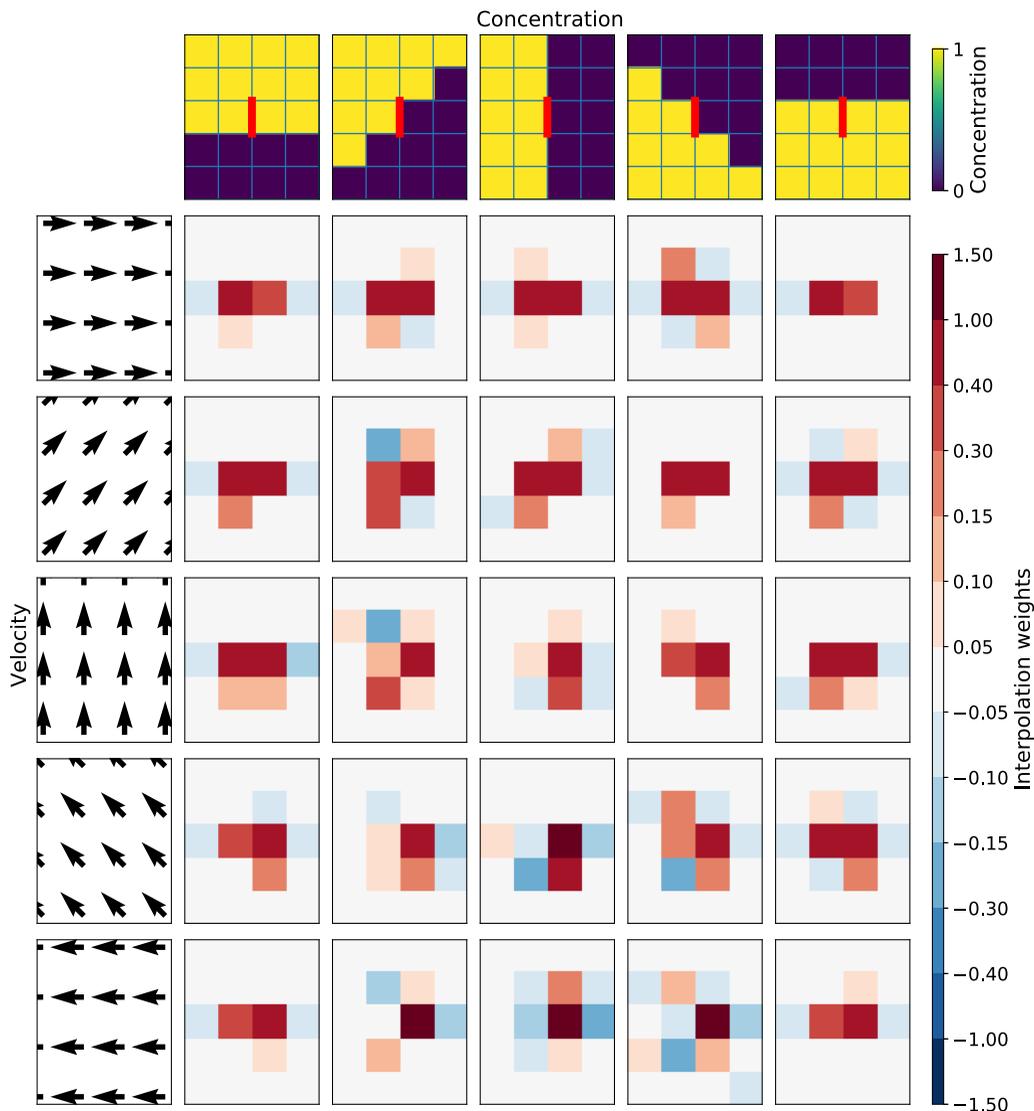


FIG. 10. Interpolation coefficients for 2D advection. Illustration of how prediction of the interpolation coefficients changes for different combinations of concentration (top row) and velocity field (left column) inputs. Concentration values represented by color; velocity field has unit magnitude and changes direction as shown in the plot. The target location for the interpolation is marked by a red bar on the concentration plots. The model predominantly interpolates along the velocity field and concentration edges, rediscovering the upwindinglike methods at the corner cases of the facet. While the model learned some general symmetries, we expect even better results for models that incorporate symmetries by construction.

techniques like model compression and quantization [61], the neural network model may significantly outperform the baseline in terms of computational performance.

Incorporating neural networks into numerical methods also allows better utilization of current and emerging hardware. It is reported that “current (Earth system) models running on next generation architectures use less than 2% of peak performance” [62]. This is because classic numerical methods (e.g., finite difference, finite volume) are limited by memory bandwidth rather than

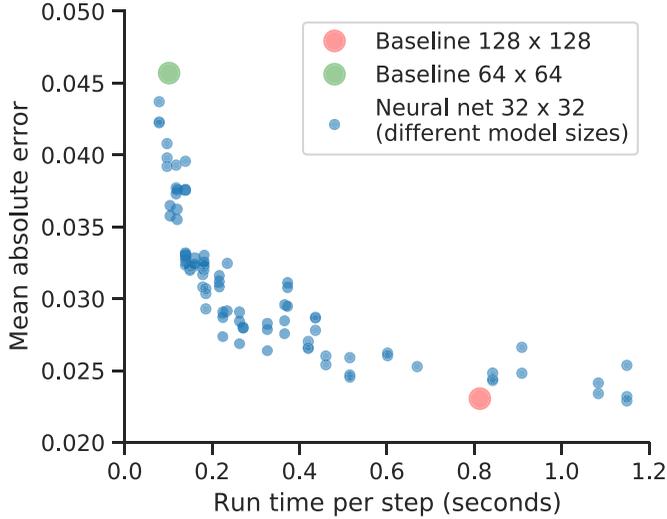


FIG. 11. Accuracy-speed trade-off for neural network model. Each data point is a neural network model with different hyperparameters (detailed in Fig. 12). The performance of the baseline solver at intermediate grid resolutions ( $64 \times 64$  and  $128 \times 128$ ) is overlaid for comparison. The model accuracy is evaluated on the 2D turbulence case after 256 time steps (Sec. III C), and the run time is measured on a single Nvidia V100 GPU. The  $x$  axis shows the wall-clock time per advection time step on the coarse grid, which requires two or four time steps for the  $64 \times 64$  or  $128 \times 128$  baseline due to the CFL condition.

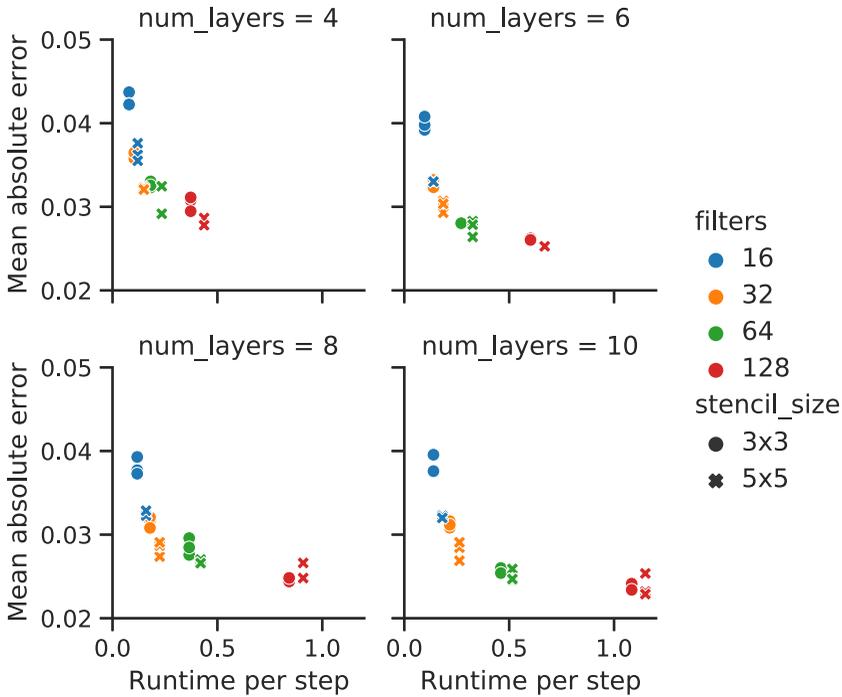


FIG. 12. Hyperparameter effect on neural network model performance. Same as Fig. 11, but here the data points are grouped by different numbers of convolutional layers and the number of filters, with symbol shape denoting the size of the stencil. Duplicate points correspond to identical models trained with different random seeds.

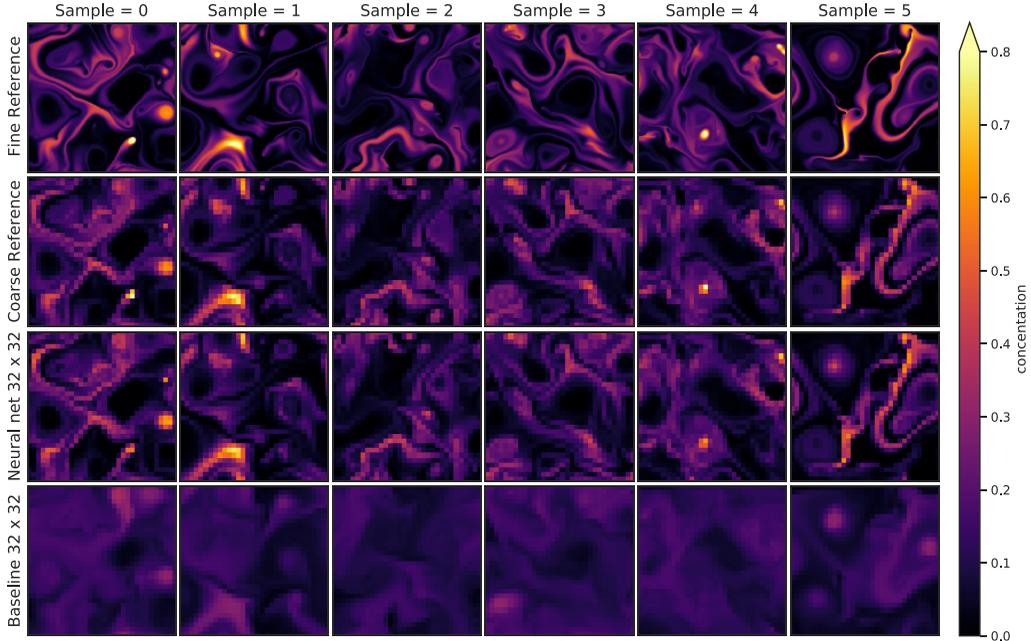


FIG. 13. Sample results for advection under 2D turbulent flow. Concentration fields after 256 time steps are illustrated for six different randomly initialized concentration and velocity fields. The models are the same as those illustrated in Fig. 7.

processor speed [63,64]. In contrast, neural networks mostly consist of dense matrix multiplications with a high compute-to-memory ratio, and therefore can achieve near-peak performance on both CPUs and hardware accelerators (see the Roofline charts [65] in [66]). We measure the machine utilization using Perf on CPU and NVProf on GPU, and find that the neural network model achieves 80% of peak FLOPs (floating point operations per second), while the baseline solver only uses 1 ~ 2% of peak FLOPs. Clever use of neural network emulations inside existing models may provide a way to squeeze out “free compute cycles” that are currently not utilized.

## V. CONCLUSION

We developed a data-driven discretization for solving passive scalar advection in one or two dimensions. Instead of using predefined coefficients to compute the spatial derivatives in the partial differential equation, we used a convolutional neural network to learn the optimal finite-difference coefficients, so that the solution best matches the “true” result obtained by high-resolution reference simulations.

Our neural network-based model is able to obtain near-perfect results for idealized 1D and 2D test problems, while a traditional high-order solver incurs significant diffusion error. Under a 2D turbulent flow, the neural network model running on a coarse grid can achieve the same accuracy as a traditional high-order solver running on a  $4\times$  higher resolution grid. The model learns local features of the flow, which enables it to generalize far beyond the velocity fields where they are trained. The learned discretizations act *locally*, and so the training data needs to cover the range of local velocity gradients that exist in the flow. This means that by training the the model on periodic, divergence-free random velocity fields we were able to create a model that accurately reproduced a standard test problem [18] very far from this training data.

It is worth noting that the results in this paper accelerate Eq. (1), which neglects diffusion. Dissipation is an intrinsic feature of multidimensional flows. Nonetheless, equations of this form

are routinely studied by fields such as atmospheric chemistry, where the diffusivity is much smaller than the numerical diffusion even on very fine meshes. Indeed, the test problem we used in Fig. 5 is a standard test problem originally proposed by Leveque [18] in the atmospheric chemistry community [54,55], where practitioners routinely study a pure advection equation on the sphere. In general if a passive scalar has a diffusivity  $D$  and a velocity field induces a stretching rate  $\gamma$ , there is a length scale  $\sqrt{D/\gamma}$  where diffusion balances stretching. The pure advection problem sends  $D \rightarrow 0$  so that this scale vanishes. When this occurs, this dissipative scale is set by numerical diffusion. The important point from the standpoint of this paper is that *nonetheless*, the algorithms we present are able to accurately reproduce resolved dynamics on coarser grids. As long as the underlying numerical method that the model is trained to reproduce is convergent in the sense of classical numerical analysis, then the resulting solutions will be accurate.

The neural network model exhibits several interesting behaviors that may help explain its unusual accuracy. Our learned models have been specifically optimized for modeling a specific class of flows used as training data, which limits their range of validity. For example, in 1D the model converts unseen shapes into known shapes, and on 2D turbulent flows the model occasionally fails entirely when asked to make predictions for much longer times than were used in training. An important challenge for future work is to identify techniques that can ensure learned discretizations are robust even to such out-of-distribution inputs. Alternatively, it may be able to identify training datasets that cover the full range of phenomena of interest, e.g., in the context of weather or pollution forecasts where the same equations are solved day after day.

At the same accuracy, the speed of our neural network model is comparable to the baseline high-order solver (that runs at  $4\times$  higher resolution). There is a lot of room for further optimizing the neural network performance in our code implementation. Notably, the neural network model can achieve a much higher machine utilization than traditional finite-difference methods, and will better utilize emerging hardware accelerators.

An open question is how to apply our method in existing computational fluid dynamics (CFD) or climate and weather models, which tend to be implemented in large codebases of C++ or Fortran. Although past work has successfully replaced one component in a model with neural networks [67], our approach works best in an end-to-end differentiable program. Recent efforts in implementing models in JULIA [68] and JAX [38] should ease the integration of scientific computing and machine learning.

## ACKNOWLEDGMENTS

We thank Anton Gerashenko, Jamie Smith, Peyman Milanfar, Pascal Getreuer, Ignacio Garcia Dorado, and Rif Saurous for important conversations. Y.B.-S. acknowledges support from the James S. McDonnell postdoctoral fellowship for the study of complex systems. M.P.B. acknowledges support from NSF Grant No. DMS-1715477 as well as the Simons Foundation.

## APPENDIX: SAMPLE RESULTS FOR 2D TURBULENT ADVECTION

Figure 13 shows more test samples for the 2D turbulent advection problem in Sec. III C. In all test samples, the neural network model is able to maintain a sharp gradient that closely matches the reference true resolution, while the baseline model incurs significant numerical diffusion error.

- 
- [1] J. Jiménez, Computers and turbulence, *Eur. J. Mech. B. - Fluids* **79**, 1 (2020).
  - [2] P. Bauer, A. Thorpe, and G. Brunet, The quiet revolution of numerical weather prediction, *Nature (London)* **525**, 47 (2015).
  - [3] T. Schneider, J. Teixeira, C. S. Bretherton, F. Brient, K. G. Pressel, C. Schär, and A. P. Siebesma, Climate goals and computing the future of clouds, *Nat. Clim. Change* **7**, 3 (2017).

- [4] P. Neumann, P. Düben, P. Adamidis, P. Bauer, M. Brück, L. Kornbluh, D. Klocke, B. Stevens, N. Wedi, and J. Biercamp, Assessing the scales in numerical weather and climate predictions: Will exascale be the rescue? *Philos. Trans. R. Soc., A* **377**, 20180148 (2019).
- [5] T. N. Theis and H.-S. P. Wong, The end of Moore’s law: A new beginning for information technology, *Comput. Sci. Eng.* **19**, 41 (2017).
- [6] J. Shalf, The future of computing beyond Moore’s law, *Philos. Trans. R. Soc., A* **378**, 20190061 (2020).
- [7] T. Schneider, S. Lan, A. Stuart, and J. Teixeira, Earth system modeling 2.0: A blueprint for models that learn from observations and targeted high-resolution simulations, *Geophys. Res. Lett.* **44**, 12,396 (2017).
- [8] M. Reichstein, G. Camps-Valls, B. Stevens, M. Jung, J. Denzler, N. Carvalhais, and Prabhat, Deep learning and process understanding for data-driven earth system science, *Nature (London)* **566**, 195 (2019).
- [9] J. N. Kutz, Deep learning in fluid dynamics, *J. Fluid Mech.* **814**, 1 (2017).
- [10] K. Duraisamy, G. Iaccarino, and H. Xiao, Turbulence modeling in the age of data, *Annu. Rev. Fluid Mech.* **51**, 357 (2019).
- [11] S. Hatfield, M. Chantry, P. Düben, and T. Palmer, Accelerating high-resolution weather models with deep-learning hardware, in *Proceedings of the Platform for Advanced Scientific Computing Conference* (Association for Computing Machinery, New York, 2019), pp. 1–11.
- [12] J. L. Hennessy and D. A. Patterson, A new golden age for computer architecture, *Commun. ACM* **62**, 48 (2019).
- [13] J. Dean, The deep learning revolution and its implications for computer architecture and chip design, [arXiv:1911.05289](https://arxiv.org/abs/1911.05289).
- [14] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner, Learning data-driven discretizations for partial differential equations, *Proc. Natl. Acad. Sci. USA* **116**, 15344 (2019).
- [15] B. I. Shraiman and E. D. Siggia, Scalar turbulence, *Nature (London)* **405**, 639 (2000).
- [16] Y. Rastigejev, R. Park, M. P. Brenner, and D. J. Jacob, Resolving intercontinental pollution plumes in global models of atmospheric transport, *J. Geophys. Res.: Atmos.* **115**, D02302 (2010).
- [17] <https://github.com/google-research/data-driven-advection>.
- [18] R. J. LeVeque, High-resolution conservative algorithms for advection in incompressible flow, *SIAM J. Numer. Anal.* **33**, 627 (1996).
- [19] P. K. Sweby, High resolution schemes using flux limiters for hyperbolic conservation laws, *SIAM J. Numer. Anal.* **21**, 995 (1984).
- [20] S.-J. Lin and R. B. Rood, Multidimensional flux-form semi-Lagrangian transport schemes, *Mon. Weather Rev.* **124**, 2046 (1996).
- [21] P. A. Ullrich, C. Jablonowski, and B. Van Leer, High-order finite-volume methods for the shallow-water equations on the sphere, *J. Comput. Phys.* **229**, 6104 (2010).
- [22] P. Colella and M. D. Sekora, A limiter for ppm that preserves accuracy at smooth extrema, *J. Comput. Phys.* **227**, 7069 (2008).
- [23] A. Semakin and Y. Rastigejev, Numerical simulation of global-scale atmospheric chemical transport with high-order wavelet-based adaptive mesh refinement algorithm, *Mon. Weather Rev.* **144**, 1469 (2016).
- [24] A. Stohl, C. Forster, A. Frank, P. Seibert, and G. Wotawa, Technical note: The Lagrangian particle dispersion model FLEXPART version 6.2, *Atmos. Chem. Phys.* **5**, 2461 (2005).
- [25] S. D. Eastham and D. J. Jacob, Limits on the ability of global Eulerian models to resolve intercontinental transport of chemical plumes, *Atmos. Chem. Phys.* **17**, 2543 (2017).
- [26] P. H. Lauritzen, R. D. Nair, and P. A. Ullrich, A conservative semi-Lagrangian multi-tracer transport scheme (CSLAM) on the cubed-sphere grid, *J. Comput. Phys.* **229**, 1401 (2010).
- [27] C. S. Kulkarni and P. F. Lermusiaux, Advection without compounding errors through flow map composition, *J. Comput. Phys.* **398**, 108859 (2019).
- [28] Using this MAE loss indicates that we require pointwise agreement between the machine learning prediction and the reference true solution, on every time step. This criteria is relevant for atmospheric transport modeling where the goal is to simulate the instantaneous, deterministic concentration field [16]. For turbulence modeling, matching the high-order statistics might be more desirable than requiring pointwise agreement of instantaneous fields. This would require changes in the loss function.

- [29] N. D. Brenowitz and C. S. Bretherton, Prognostic validation of a neural network unified physics parameterization, *Geophys. Res. Lett.* **45**, 6289 (2018).
- [30] M. Innes, A. Edelman, K. Fischer, C. Rackauckus, E. Saba, V. B. Shah, and W. Tebbutt, Zygote: A differentiable programming system to bridge machine learning and scientific computing, [arXiv:1907.07587](https://arxiv.org/abs/1907.07587).
- [31] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, Automatic differentiation in machine learning: A survey, *J. Mach. Learn. Res.* **18**, 5595 (2018).
- [32] C. Bischof, P. Khademi, A. Mauer, and A. Carle, Adifor 2.0: Automatic differentiation of Fortran 77 programs, *IEEE Comput. Sci. Eng.* **3**, 18 (1996).
- [33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard *et al.*, Tensorflow: A system for large-scale machine learning, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (USENIX Association, Savannah, GA, 2016), pp. 265–283.
- [34] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, Automatic differentiation in PyTorch, *NIPS 2017 Workshop on Autodiff* (2017), <https://openreview.net/forum?id=BJJsrnfCZ>.
- [35] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, JAX: Composable transformations of Python+NumPy programs, <http://github.com/google/jax>.
- [36] M. Innes, Flux: Elegant machine learning with Julia, *J. Open Source Software* **3**, 602 (2018).
- [37] Swift differentiable programming manifesto, 2018, <https://github.com/apple/swift>.
- [38] S. S. Schoenholz and E. D. Cubuk, JAX, M.D.: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python, [arXiv:1912.04232](https://arxiv.org/abs/1912.04232).
- [39] A. Agrawal, A. N. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga *et al.*, TensorFlow eager: A multi-stage, python-embedded DSL for machine learning, in *Proceedings of the 2nd SysML Conference, Palo Alto, CA, USA* (2019), <https://mlsys.org/Conferences/2019/doc/2019/88.pdf>.
- [40] S.-J. Lin, W. C. Chao, Y. Sud, and G. Walker, A class of the van Leer-type transport schemes and its application to the moisture transport in a general circulation model, *Mon. Weather Rev.* **122**, 1575 (1994).
- [41] H. Smaoui and B. Radi, Comparative study of different advective schemes: Application to the MECCA model, *Environ. Fluid Mech.* **1**, 361 (2001).
- [42] H. Hassanzadeh, J. Abedi, and M. Pooladi-Darvish, A comparative study of flux-limiting methods for numerical simulation of gas–solid reactions with Arrhenius type reaction kinetics, *Comput. Chem. Eng.* **33**, 133 (2009).
- [43] M. Raissi and G. E. Karniadakis, Hidden physics models: Machine learning of nonlinear partial differential equations, *J. Comput. Phys.* **357**, 125 (2018).
- [44] M. Raissi, P. Perdikaris, and G. E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *J. Comput. Phys.* **378**, 686 (2019).
- [45] E. de Bézenac, A. Pajot, and P. Gallinari, Deep learning for physical processes: Incorporating prior scientific knowledge, *J. Stat. Mech.* (2019) 124009.
- [46] K. Um, R. Brand, Y. Fei, P. Holl, R. Brand, and N. Thuerey, Solver-in-the-Loop: Learning from differentiable physics to interact with iterative PDE-Solvers, [arXiv:2007.00016](https://arxiv.org/abs/2007.00016).
- [47] J. Pathak, M. Mustafa, K. Kashinath, E. Motheau, T. Kurth, and M. Day, Using machine learning to augment coarse-grid computational fluid dynamics simulations, [arXiv:2010.00072](https://arxiv.org/abs/2010.00072).
- [48] H. Frezat, G. Balarac, J. Le Sommer, R. Fablet, R. Lguensat, Physical invariance in neural networks for subgrid-scale scalar flux modeling, *Phys. Rev. Fluids* **6**, 024607 (2021).
- [49] J. Ling, A. Kurzawski, and J. Templeton, Reynolds averaged turbulence modelling using deep neural networks with embedded invariance, *J. Fluid Mech.* **807**, 155 (2016).
- [50] J. P. Boris and D. L. Book, Flux-corrected transport. I. SHASTA, a fluid transport algorithm that works, *J. Comput. Phys.* **11**, 38 (1973).
- [51] D. L. Book, The conception, gestation, birth, and infancy of FCT, *Flux-Corrected Transport* (Springer, New York, 2012), pp. 1–21.
- [52] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).

- [53] M. T. Odman, A quantitative analysis of numerical diffusion introduced by advection algorithms in air quality models, *Atmos. Environ.* **31**, 1933 (1997).
- [54] R. D. Nair and P. H. Lauritzen, A class of deformational flow test cases for linear transport problems on the sphere, *J. Comput. Phys.* **229**, 8868 (2010).
- [55] P. H. Lauritzen, P. A. Ullrich, C. Jablonowski, P. A. Bosler, D. Calhoun, A. J. Conley, T. Enomoto, L. Dong, S. Dubey, O. Guba *et al.*, A standard test case suite for two-dimensional linear transport on the sphere: Results from a collection of state-of-the-art schemes, *Geosci. Model Dev.* **7**, 105 (2014).
- [56] T. Saad and J. C. Sutherland, Comment on “Diffusion by a random velocity field” [Phys. Fluids 13, 22 (1970)], *Phys. Fluids* **28**, 119101 (2016).
- [57] J. Zhuang, D. J. Jacob, and S. D. Eastham, The importance of vertical resolution in the free troposphere for modeling intercontinental plumes, *Atmos. Chem. Phys.* **18**, 6039 (2018).
- [58] <https://github.com/pyqg/pyqg>.
- [59] J. C. McWilliams, The emergence of isolated coherent vortices in turbulent flow, *J. Fluid Mech.* **146**, 21 (1984).
- [60] J. Methven and B. Hoskins, The advection of high-resolution tracers by low-resolution winds, *J. Atmos. Sci.* **56**, 3262 (1999).
- [61] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, A survey of model compression and acceleration for deep neural networks, [arXiv:1710.09282](https://arxiv.org/abs/1710.09282).
- [62] J. Carman, T. Clune, F. Giraldo, M. Govett, B. Gross, A. Kamrath, T. Lee, D. McCarren, J. Michalakes, S. Sandgathe, and T. Whitcomb, Position paper on high performance computing needs in Earth system prediction, *National Earth System Prediction Capability* (2017), <https://doi.org/10.7289/V5862DH3>.
- [63] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, and C. Schär, Reflecting on the goal and baseline for exascale computing: A roadmap based on weather and climate simulations, *Comput. Sci. Eng.* **21**, 30 (2018).
- [64] Indeed, not all traditional solvers have ultralow machine utilization. For example, the discontinuous Galerkin method can have a much higher machine utilization of 20% [69]. Although CFD models can exploit high-order methods, climate models tend to use low-order schemes [63].
- [65] S. Williams, A. Waterman, and D. Patterson, Roofline: An insightful visual performance model for multicore architectures, *Commun. ACM* **52**, 65 (2009).
- [66] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, In-datacenter performance analysis of a tensor processing unit, in *Proceedings of the 44th Annual International Symposium on Computer Architecture* (IEEE, New York, 2017), pp. 1–12.
- [67] S. Rasp, M. S. Pritchard, and P. Gentine, Deep learning to represent subgrid processes in climate models, *Proc. Natl. Acad. Sci. USA* **115**, 9684 (2018).
- [68] Clima: Climate machine, <https://github.com/climate-machine/CLIMA>.
- [69] A. Breuer, Y. Cui, and A. Heinecke, Petaflop seismic simulations in the public cloud, in *International Conference on High Performance Computing* (Springer, New York, 2019), pp. 167–185.