NODE-FNO and C-FNO: Fourier Neural Operators with Adjoint Training and Conservation Enforcing

Kai Chang*

Abstract

Fourier neural operator (FNO) is a recently developed Deep Learning architecture that learns mappings between infinite-dimensional spaces. Mainly used for solving parametric partial differential equations (PDEs) by directly mapping a parametric function to the solution, FNOs have been shown to have competitive performances in practice. However, there are two issues with the vanilla FNOs. For one thing, the size of the kernel tensor that is used in the convolution step and learned during training grows exponentially as the dimension of the target problem gets higher. This limits the depth of FNOs and its applicability to large-scale simulations. For the other, there is barely any underlying physical information being considered in FNOs. In this work, we propose to modify FNOs in two parallel ways: modeling FNOs as Neural ODEs (NODE-FNOs) and enforcing conservation laws on FNOs (C-FNOs) to alleviate these two issues. For NODE-FNOs, we not only consider the vanilla Neural ODEs but also various more advanced variants of Neural ODEs such as Augmented Neural ODEs, Neural ODEs trained with the Adaptive Checkpoint Adjoint method, etc. We conduct extensive numerical experiments on numerous benchmarks including 1-dimensional Burgers' equation, 2-dimensional Darcy flow problem, and 2-dimensional Navier-Stokes equations. We provide comprehensive comparisons for the performances of all these variants of FNOs with well-tuned hyperparameters. We show that both NODE-FNOs and C-FNOs achieve superior performances compared to the vanilla FNOs. In particular, NODE-FNOs beat the vanilla FNOs in all the benchmarks.

1 Introduction

Deep Learning (DL) methods [15] have received great attentions from the community of scientific computing. They have shown success in fields such as inverse problems [22, 1] and computational fluid dynamics (CFD) [14, 12]. A large amount of Deep Learning's success in CFD relies on the technique of using Deep Learning methods to solve partial differential equations (PDEs) fluid dynamics are modeled by PDEs. The idea of using DL models to solve PDEs was introduced independently in [8, 21]. Later, other types of DL-based PDE

*Department of Mathematics, Emory University. Work done during the author's internship at the Oden Institute for Computational Engineering and Sciences, University of Texas at Austin in Summer 2022

solvers were introduced. Many of them are take an autoregressive approach to solve time-dependent PDEs by mapping the solution at the previous time step to that at the current time step. Examples include [2, 10, 7, 3].

Recently, a novel class of algorithms called neural operators have been developed for solving a class of partial differential equations. Neural operators, based on Deep Learning architectures, are able to approximate mappings between infinite-dimensional spaces by taking advantage of the fact that neural networks are universal function approximators [5, 9]. Variants of neural operators include Graph Neural Operators (GNOs) [18], Multipole Graph Neural Operators (MGNOs) [19], Low-rank Neural Operators (LNOs) [13], Fourier Neural Operators (FNOs) [17], FNOs on complex geometries (Geo-FNOs) [16], Physics-Informed Neural Operators (PINOs) [20], etc.

Among all these variants, FNOs are shown to have the best performance in practice. However, there are two issues related to the vanilla FNOs. For one thing, the size of the kernel tensor that is used in the convolution step and learned during training grows exponentially as the dimension of the target problem gets higher. This limits the depth of FNOs and its applicability to large-scale simulations. For the other, there is barely any underlying physical information being considered in FNOs. Although PINOs add an additional PDE-loss term to the original FNOs, the physical properties such as conservation laws associated with the equation being learned are still not yet considered in the model. We attempt to alleviate those two issues with two independent methods in this work.

Our contributions can be summarized as follows:

- We proposed and developed a memory-efficient variant of FNOs by modeling them as Neural Ordinary Differential Equations.
- We compared the performances between advanced Neural ODEs.
- We proposed and developed a conservation-enforced FNO for time-dependent PDEs.
- We provided extensive numerical experiments on existing benchmarks to show the superior performance of our methods.

The rest of the paper is organized as the following. We describe some preliminary knowledge in Section 2 and present our proposed approaches in Section 3 and 4. We give the problem statements of the benchmarks we tested on in Section 5, show the corresponding numerical results in Section 6, and draw our conclusion in Section 7.

2 Preliminaries

2.1 Fourier Neural Operators (FNOs)

For demonstrations and simplicity of notations, in this section we only consider FNOs for time-independent PDEs. They can easily be generalized to time-dependent PDEs. We will extend it for individual cases whenever necessary in later discussions.

Suppose we are given a class of PDEs

$$\mathcal{D}_a(u(x)) = 0, \quad x \in \Omega \tag{1}$$

where Ω is the domain of the solution u and $\mathcal{D}_a$ is a differential operator parameterized by a function $a \in \mathcal{A}$ where $\mathcal{A}$ is some functional space. An FNO is a neural network $\mathcal{NN}_\theta$ parameterized by θ such that $\mathcal{NN}_\theta$ maps an arbitrary $a \in \mathcal{A}$ to u where u is the solution to the PDE (1) defined by a. Let $\mathcal{G} = \{x_i\}_{i=1}^{N_d}$ be N_d points in Ω and $\boldsymbol{x} = [x_1, x_2, \ldots, x_{N_d}]^\top$. Note that here we are abusing the notation of x_i's for higher dimensional vectors as well: when Ω lives in a space with a dimension of more than 1, x_i simply denotes the vector of coordinates that characterizes the location of the points in the space. We use $a(\boldsymbol{x})$ to denote the vector of values of a evaluated on $\mathcal{G}$; that is $a(\boldsymbol{x}) = [a(x_1), a(x_2), \ldots, a(x_{N_d})]^\top$. Similarly, we use $u(\boldsymbol{x})$ to denote the vector of values of u evaluated on $\mathcal{G}$; that is $u(\boldsymbol{x}) = [u(x_1), u(x_2), \ldots, u(x_{N_d})]^\top$. In the rest of the discussion, we use $f(\boldsymbol{x})$ to denote the vector of the values of a function f applied point-wise to a vector $\boldsymbol{x}$ and $f(x)$ to denote its function representation.

As it turns out, inputting $a(\boldsymbol{x})$ per se into $\mathcal{NN}_\theta$ is not enough for FNOs. Instead, the network needs to know both the function value and the location of the points it is being evaluated on: we need to input both $a(\boldsymbol{x})$ and $\boldsymbol{x}$. Let $\boldsymbol{A} := \begin{bmatrix} a(\boldsymbol{x}) & \boldsymbol{x} \end{bmatrix}^\top \in \mathbb{R}^{N_{in} \times N_d}$ be the matrix out of concatenating $a(\boldsymbol{x})$ and $\boldsymbol{x}$. N_{in}, the number of rows of $\boldsymbol{A}$, depends on a, Ω in (1), and any other feature that is needed to describe a discretization point. Note that without loss of generality, it suffices to assume $\boldsymbol{A} = a(\boldsymbol{x}) \in \mathbb{R}^{N_d}$ and $N_{in} = 1$ because it will only be a matter of adding an extra dimension by concatenating other features. Thus, we ideally want

$$\mathcal{NN}_\theta(a(\boldsymbol{x})) = u(\boldsymbol{x}).$$

We are now ready to introduce the architectural design of FNOs. An FNO can be decomposed into three main components: an encoder, Fourier layers, and a decoder.

During the encoding phase, FNOs apply an affine transformation $\boldsymbol{P}$ to the input so that the dimension of the input is lifted up to a high-dimensional representation: that is, we apply

$$\boldsymbol{P}(a(\boldsymbol{x})) := \boldsymbol{W}_{enc}\, a(\boldsymbol{x})^\top + \boldsymbol{B}_{enc} \tag{2}$$

where $\boldsymbol{W}_{enc} \in \mathbb{R}^{N_h \times N_{in}}$ and $\boldsymbol{B}_{enc} \in \mathbb{R}^{N_h \times N_d}$ with N_h being the dimension of the representation. Note that the N_d columns of $\boldsymbol{B}_{enc}$ are the same: $\boldsymbol{B}_{enc} = \begin{bmatrix} \boldsymbol{b}_{enc} & \boldsymbol{b}_{enc} & \cdots & \boldsymbol{b}_{enc} \end{bmatrix}$ where $\boldsymbol{b}_{enc} \in \mathbb{R}^{N_h}$ is a so-called bias vector. $\boldsymbol{W}_{enc}$ and $\boldsymbol{b}_{enc}$ are part of the parameter θ that defines $\mathcal{NN}_\theta$. We let $u_0(\boldsymbol{x}) = \boldsymbol{P}(a(\boldsymbol{x}))$ and $u_0(x)$ be its function representation.

The Fourier layers are what distinguish FNOs from other types of neural operators. In its continuous formulation, a typical layer of a Neural Operator has the form of

$$u_{l+1}(x) := \sigma\left(T(u_l(x); W_l) + (\mathcal{K}(a; \phi)u_l)(x)\right),$$

for $l = 0, 1, 2, \ldots, L$ where L is the total number of layers, $T(\,\cdot\,; W_l)$ which serves as a bias term is a convolution operator with kernel W_l , $\sigma(\cdot)$ is a nonlinear actication function, and $\mathcal{K}((a; \phi)u_l)(x)$ is defined as

$$(\mathcal{K}(a; \phi)u_l)(x) := \int_{\mathcal{D}} \kappa_\phi(x, y, a(x), a(y))\, u_l(y)\, dy \tag{3}$$

with κ_ϕ being a neural network. A Fourier layer takes advantage of the fact that by imposing $\kappa_\phi(x, y, a(x), a(y)) = \kappa_\phi(x - y)$, we can apply the convolution theorem to rewrite (3) into

$$(\mathcal{K}(a; \phi)u_l)(x) = \mathcal{F}^{-1}(\mathcal{F}(\kappa_\phi) \cdot \mathcal{F}(u_l))(x) \tag{4}$$

with $\mathcal{F}$ and $\mathcal{F}^{-1}$ denoting the Fourier Transform and the inverse Fourier Transform respectively. We replace $\mathcal{F}(\kappa_\phi)$ in (4) with a linear transformatrion $R(\cdot)$. In its vector form,

$$(\mathcal{K}(a(\boldsymbol{x});\phi)u_l)(\boldsymbol{x}) = \mathcal{F}^{-1}\left(\boldsymbol{R}_l \cdot \mathcal{F}(u_l(\boldsymbol{x}))\right)$$

and hence, the vector form of a Fourier layer is

$$u_{l+1}(\boldsymbol{x}) := \sigma\left(T(u_l(\boldsymbol{x}); \boldsymbol{W}_l, \boldsymbol{b}_l) + \mathcal{F}^{-1}(\boldsymbol{R}_l \cdot \mathcal{F}(u_l(\boldsymbol{x})))\right),$$

where $T(\,\cdot\,; \boldsymbol{W}_l, \boldsymbol{b}_l)$ is a convolution operator with a kernel matrix $\boldsymbol{W}_l$ and a bias vector $\boldsymbol{b}_l$, $\boldsymbol{R}_l$ is a potentially complex tensor, and $\sigma(\cdot)$ applies point-wise to the input. It is worth noting that $\boldsymbol{W}_l$, $\boldsymbol{b}_l$, and $\boldsymbol{R}_l$ are all part of the parameter θ that defines an FNO and are learned during training.
In order to compute the Fourier Transform and its inverse, Fast Fourier Transform (FFT) and inverse FFT are used. The dimension of the FFT used will be the same as the dimension of the input $a(x)$. For time-dependent PDEs, there are two options: using FFT on both spacial and temporal dimensions or using FFT on only spacial dimensions and modeling the temporal dimension with another model, such as ResNet. When the PDE being learned is real valued, the real FFT can be used, which returns $N_d/2 + 1$ modes since the other modes are complex conjugates of these for real-valued input. In order to increase the efficiency of the Fourier layer, the output of the FFT will be limited to the first m modes. The Fourier layer will still be capable of representing higher frequencies, however, because of the activation function $\sigma()$ applied to the output of the inverse FFT.

Last but not least, we apply a decoder $\boldsymbol{Q}(\cdot)$ that is an affine transformation to $u_{L+1}(\boldsymbol{x})$ by computing

$$\boldsymbol{Q}(u_{L+1}(\boldsymbol{x})) = \left(\boldsymbol{W}_{dec}\, a(\boldsymbol{x})\right)^\top + \boldsymbol{b}_{dec}$$

where $\boldsymbol{W}_{dec} \in \mathbb{R}^{1\times N_h}$ is and $\boldsymbol{b}_{dec} \in \mathbb{R}^{N_d}$ are part of the learned parameters.

In summary, an FNO applies an encoder, L Fourier layers, and a decoder sequentially to the input. The parameter that defines the network is $\theta = \{\boldsymbol{W}_{enc}, \boldsymbol{b}_{enc}, \boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L, \boldsymbol{W}_{dec}, \boldsymbol{b}_{dec}\}$ and is learned with gradient descent algorithms in practice.

2.2 Neural Ordinary Differential Equations (NODEs)

Neural ordinary differential equations (NODEs) [4] are a class of neural networks originally developed as a continuous formulation of the Residual neural networks (ResNets). Suppose we are given training data $\{(x_i, y_i)\}_{i=1}^N$ where x_i is the input and y_i is the ground truth. We want to build a supervised learning model to make predictions for x_i by solving the optimization problem

$$\min_\psi \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\mathcal{N}_\psi(x_i), y_i)$$

where $\mathcal{L}$ is a loss function and $\mathcal{N}_\psi$ is a ResNet parameterized by ψ. The updating formula for a ResNet reads

$$z_{t+1} = z_t + f_t(z_t, \psi_t)$$

where z_t is the hidden state, f_t is some neural network layer parameterized by ψ_t, and $t \in 0, 1, \ldots, L$ is the layer index with $z_0 = x_i$ and $z_{L+1} = \mathcal{N}_\psi(x_i)$ being the input and the output respectively. This is equivalent to discretizing the ordinary differential equation (ODE)

$$\frac{dz(t)}{dt} = f(z, t, \psi_t) \quad \text{with } z(0) = x$$

with the forward Euler method

$$z_{t+1} = z_t + f(z_t, t, \psi_t)$$

by setting the step size as 1.

The advantage of NODEs mainly lies in memory efficiency by using the so-called adjoint method during training. The adjoint is defined as the gradient of $\mathcal{L}$ with respect to the hidden states. By defining the adjoint state as

$$\lambda(t) = \frac{\partial \mathcal{L}(z(T), y)}{\partial z(t)}, \quad 0 \leq t \leq T,$$

we obtain a new ODE

$$\frac{d\lambda(t)}{dt} = g(\lambda, t, \psi_t), \quad 0 \leq t \leq T,$$

where $g(\lambda, t, \psi_t)$ governs the dynamics of the adjoint ODE, and is defined by

$$g(\lambda, t, \psi_t) = -\lambda(t)^\top \frac{\partial f(z(t), t, \psi_t)}{\partial z(t)}, \quad 0 \leq t \leq T.$$

We may also define gradient of $\mathcal{L}$ with respect to the parameter ψ that defines the neural network through the adjoint, which reads

$$\frac{\partial \mathcal{L}(z(T), y)}{\partial \psi_t} = -\int_T^0 \lambda(t)^\top \frac{\partial f(z(t), t, \psi_t)}{\partial \psi_t} dt, \quad 0 \leq t \leq T. \tag{5}$$

In order to increase computational efficiency and parallelization, the adjoint ODE and the integral evaluation for the loss gradient are done simultaneously with an augmented ODE. In addition, after the computation of the forward ODE, only the terminal condition of the ODE solution is available, so the forward ODE is also augmented to the adjoint ODE to give access to the ODE solution $z(t)$ at any time t. The integral equation 5 is equivalent to solving to following ODE:

$$\frac{dh(t)}{dt} = -\lambda(t)^\top \frac{\partial f(z(t), t, \psi_t)}{\partial \psi_t}, \quad h(T) = 0$$

since

$$\frac{d\mathcal{L}(z(T), y)}{d\psi_t} = -\int_T^0 \lambda(t)^\top \frac{\partial f(z(t), t, \psi_t)}{\partial \psi_t} dt = \int_T^0 \frac{dh(t)}{dt} dt = h(0) - h(T) = h(0) \tag{6}$$

The numerical implementation of Neural ODE with the adjoint method can be divided into the following two steps:

S.1 Solve $z(T)$ from $t = 0$ to $t = T$ with numerical ODE solvers (forward-time pass). Determine $\lambda(T) = \dfrac{\partial \mathcal{L}(z(T), y)}{\partial z(T)}$.

S.2 Numerically solve the following augmented ODE from $t = T$ to $t = 0$ (reverse-time pass):

$$\frac{dz(t)}{dt} = f(z(t), t, \psi_t), \quad z(T) = z(T) \tag{7}$$

$$\frac{d\lambda(t)}{dt} = -\lambda(t)^\top \frac{\partial f(z(t), t, \psi_t)}{\partial z(t)}, \quad \lambda(T) = \frac{\partial \mathcal{L}(z(T), y)}{\partial z(T)} \tag{8}$$

$$\frac{d\mathcal{L}(z(T), y)}{d\psi_t} = h(0), \quad \frac{dh(t)}{dt} = -\lambda(t)^\top \frac{\partial f(z(t), t, \psi_t)}{\partial \psi_t}, \quad h(T) = 0 \tag{9}$$

The equations (7), (8), and (9) constitute an augmented ODE which is solved numerically during backpropagation. The memory complexity of the adjoint method is $O(1)$ which makes it preferrable during practice.

In summary, NODEs are better than its discrete counterparts for both theoretical and practical reasons. Theoretically, tools such as stability analysis and error estimate that have been developed for years in numerical analysis can be employed to analyze the performance. Practically, NODEs require much less memory cost by using the adjoint method to train the model; computational speed and accuracy can be explicitly traded off during training by adjusting the error tolerance.

2.3 Variants of Neural ODEs

In this section we review several more advanced variants of NODEs which we will need in Section 3. It has been shown that they have preferable properties either in terms of numerical accuracy or computational complexity compared to the vanilla NODEs.

Augmented Neural ODEs [6]

It was proved in [6] that NODEs have a limited expressive power in the sense that they can only learn a feature map that is a homeomorphism, or continuous bijection who also has a continuous inverse . This implies that NODEs can only continuously deform the input space.

Augmented NODEs take a step towards enhancing the expressive power of NODEs with a simple modification made on the original model. The hidden state $z(t)$ is lifted into a higher dimension by augmenting with another vector-valued function $h(t)$. The initial condition for $h(t)$ is the zero vector. Instead of modeling the problem as

$$\frac{dz(t)}{dt} = f(z, t, \psi_t) \quad with\ z(0) = x,$$

it is proposed to model it as an augmented ODE

$$\frac{d}{dt}\begin{bmatrix} z(t) \\ h(t) \end{bmatrix} = f\left(\begin{bmatrix} z(t) \\ h(t) \end{bmatrix}, t, \psi_t\right) \quad with \quad \begin{bmatrix} z(0) \\ h(0) \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix}.$$

The adjoint for the augmented NODE is

$$a(t) = \frac{\partial \mathcal{L}}{\partial \begin{bmatrix} z(t) \\ h(t) \end{bmatrix}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z(t)} & \frac{\partial \mathcal{L}}{\partial h(t)} \end{bmatrix},$$

and is the solution to the adjoint ODE:

$$\frac{da(t)}{dt} = -a(t)^\top \frac{\partial f}{\partial \begin{bmatrix} z(t) \\ h(t) \end{bmatrix}} \left(\begin{bmatrix} z(t) \\ h(t) \end{bmatrix}, t, \psi_t\right). \tag{10}$$

It has been shown that such a simple modification of lifting into a higher dimension would make the learned neural network f smoother and that the resulting network achieves better generalization and has lower computational cost.

Adaptive Checkpoint Adjoint Neural ODEs [23]

Compared with the adjoint method, ACA guarantees the accuracy of reverse-time trajectory. This in return gives more accurate gradient estimations than those in the adjoint method which is crucial during training.

The accuracy problem of the adjoint method lies in the step S.1. Note that during the numerical implementation of this step, the forward-time trajectory $z(t), 0 < t < T$ is deleted along the way to save memory. They are then reconstructed in the ODE solver in reverse time. We denote the reconstructed trajectory as $\overline{z(t)}$. The reconstruction introduces unavoidable numerical errors causing potentially large mismatch between $z(t)$ and $\overline{z(t)}$. Therefore, the gradient estimation is not accurate.

ACA alleviates this issue with a simple step of memorizing the forward trajectory. Instead of numerically reconstructing for $z(t)$, $z(t)$ is recorded at each time discretization point (layer). This can introduce heavy memory cost for large-scale applications since the memory complexity is linear in t. To mitigate this, ACA adaptively searches for the optimal discretization of the time domain $[0, T]$ and deletes useless computation graphs. The discretization points in the time domain adaptively searched by ACA are $\{t_i\}_{i=0}^{L+1}$ with $t_0 = 0$ and $t_{L+1} = T$. With ACA the adjoint ODE step S.2 is changed so that the hidden state is solved locally at each time discretization point using the checkpointed values from the forward ODE. The adjoint state and loss gradient are updated at each time discretization point based on the gradient of the locally solved hidden state. The entire ACA algorithm for backpropagation can be seen below:

Memory-Efficient Asynchronous Leapfrog Integrator [24]

While ACA accurately reconstructs the reverse time trajectory by recording the forward time trajectory, it does so, as mentioned, at the sacrifice of memory growth. This will limit the method to only relatively small scale applications.

Algorithm 1 ACA Backpropagation

$\lambda \leftarrow \dfrac{\partial \mathcal{L}}{\partial z(T)}$

$\dfrac{\partial \mathcal{L}}{\partial \psi} \leftarrow 0$

for $i \leftarrow L+1, L, ..., 1, 0$ **do**

  $\bar{z}_i = ODESolve(f_\psi, z_{i-1}, t_{i-1}, h_i = t_i - t_{i-1})$

  $\dfrac{\partial \mathcal{L}}{\partial \psi} \leftarrow \dfrac{\partial \mathcal{L}}{\partial \psi} - \lambda^\top \dfrac{\partial \bar{z}_i}{\partial \psi}$

  $\lambda \leftarrow \lambda^\top \dfrac{\partial \bar{z}_i}{\partial z_{i-1}}$

end for

return $\dfrac{\partial \mathcal{L}}{\partial \psi}, \lambda$

MALI is better than both ACA and the vanilla adjoint method in that it achieves to both accurately reconstruct the reverse-time trajectory and have the memory cost remain constant throughout the training. It achieves so by incorporating Asynchronous Leapfrog Integrator (ALF) into the model. The main feature of ALF is that it can be easily inverted in practice. For the detailed algorithmic descriptions of the one-step forward ALF and the inverted ALF, see Algorithm 2 and 3 in [24].

The gist of MALI is summarized in the following. Suppose the time domain $[0, T]$ is discretized by $\{t_i\}_{i=0}^{L+1}$ with $t_0 = 0$ and $t_{L+1} = T$. During the forward-time pass, ALF is applied to propagate forward:

$$z_{t_{i+1}} = ALF(z_{t_i}) \quad with\ z(t_0) = x$$

for $i = 1, 2, \ldots, L$. It suffices to only record the endpoint $z(T)$ as in the adjoint method. In the reverse-time pass, the forward time trajectory $z(t)$, $0 < t < T$ is accurately reconstructed by applying the inverted ALF (IALF):

$$z_{t_i} = IALF(z_{t_{i+1}})$$

for $i = L, L-1, \ldots, 1$. Hence, this allows for accurate trajectory reconstructions and thereby gradient estimations.

3 FNOs Trained as NODEs

Recall that the main component of an FNO is the Fourier layers which consists of a convolution operation and a bias term. The convolution kernel tensors $\{\boldsymbol{R}_l\}_{l=1}^{L}$ are learned during training. However, the size of $\boldsymbol{R}_l$ grows exponentially as the dimension of the problem goes higher. Using several Fourier layers in practice therefore puts a huge burden on the memory required. This limits the applicability of FNOs to large-scale simulations.

On the other hand, the memory complexity of NODEs does not grow with the depth of the network. We therefore propose to model FNOs as NODEs (NODE-FNOs) to improve

the training of FNOs without deteriorating the performance of the FNOs. In fact, experiments presented in Section 6 show that modeling FNOs as NODEs actually improve the performance by a good amount. NODE-FNO are a continuous model as apposed to discrete FNOs. The continuous model is fitting as FNOs seek to model solutions to PDEs, which are continuous. FNOs simplely learn a mapping directly from parameters to a solution, but NODE-FNOs learn a latent ODE representation of a PDE using Fourier convolutions to govern the dynamics of the ODE.

Recall that the updating formula of Fourier layers is

$$u_{l+1}(\boldsymbol{x}) = \sigma\left(T(u_l(\boldsymbol{x}); \boldsymbol{W}_l, \boldsymbol{b}_l) + \mathcal{F}^{-1}(\boldsymbol{R}_l \cdot \mathcal{F}(u_l(\boldsymbol{x})))\right),$$

for $l = 0, 1, 2, \ldots, L$ where $\boldsymbol{W}_l$, $\boldsymbol{b}_l$, and $\boldsymbol{R}_l$ are learned parameters, $T(\,\cdot\,; \boldsymbol{W}_l, \boldsymbol{b}_l)$ is a convolution operator with a kernel matrix $\boldsymbol{W}_l$ and a bias vector $\boldsymbol{b}_l$, and $\sigma(\cdot)$ applies point-wise to the input. We replace the formula with

$$u_{l+1}(\boldsymbol{x}) = u_l(\boldsymbol{x}) + \sigma\left(T(u_l(\boldsymbol{x}); \boldsymbol{W}_l, \boldsymbol{b}_l) + \mathcal{F}^{-1}(\boldsymbol{R}_l \cdot \mathcal{F}(u_l(\boldsymbol{x})))\right), \tag{11}$$

which can be viewed as the forward Euler discretization of the ODE

$$\frac{du(t, \boldsymbol{x})}{dt} = \sigma\left(T(u(t, \boldsymbol{x}); \boldsymbol{W}, \boldsymbol{b}) + \mathcal{F}^{-1}(\boldsymbol{R} \cdot \mathcal{F}(u(t, \boldsymbol{x})))\right)$$

by taking step size as 1. We can thus view the modeified Fourier layers as a NODE.

Let $\mathcal{NN}_\theta$ be our modified FNO (NODE-FNO) and $\mathcal{L}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x}))$ be a loss function where $a(\boldsymbol{x})$ is the input of $\mathcal{NN}_\theta$ and $u(\boldsymbol{x})$ is the ground truth. The forward propagation of NODE-FNO is summarized in Algorithm 2. Note that in line 4, $ODESolve$ also implicitly returns the gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ and $\frac{\partial \mathcal{L}}{\partial u(t_i, \boldsymbol{x})}$ by solving the augmented system of ODEs:

$$\frac{du(t, \boldsymbol{x})}{dt} = f(u(t, \boldsymbol{x}), t, \theta_t), \quad u(T, \boldsymbol{x}) = u(T, \boldsymbol{x})$$

$$\frac{d\lambda(t)}{dt} = -\lambda(t)^\top \frac{\partial f(u(t, \boldsymbol{x}), t, \theta_t)}{\partial u(t, \boldsymbol{x})}, \quad \lambda(T) = \frac{\partial \mathcal{L}(u(T, \boldsymbol{x}), y)}{\partial u(T, \boldsymbol{x})}$$

$$\frac{dh(t)}{dt} = -\lambda(t)^\top \frac{\partial f(u(t, \boldsymbol{x}), t, \theta_t)}{\partial \theta_t}, \quad h(T) = 0$$

where the adjoint is $\lambda(t) = \frac{\partial \mathcal{L}(u(t, \boldsymbol{x}), y)}{\partial u(t, \boldsymbol{x})}$, and the gradient of the loss with respect to the parameters is $\frac{d\mathcal{L}(u(T, \boldsymbol{x}), y)}{d\theta_t} = h(0)$. Hence, the gradients can be computed efficiently in practice.

4 Conservative FNOs

The development of FNOs, without any doubt, has been a breakthrough in the field of solving PDEs with deep learning approaches. However, as a purely data-driven approach, FNOs do not take the underlying physics into account. Although the recently developed PINOs [20] incorporate the PDE loss into their model as inspired by physics-informed neural networks [21], the physical properties such as symmetries and conservations are barely reflected in the

Algorithm 2 NODE-FNO

Input: $a(\boldsymbol{x})$, σ, T, $\mathcal{L}$
Output: $u(\boldsymbol{x})$

1: Define $f(\cdot) = \sigma\left(T(\ \cdot\ ; \boldsymbol{W}, \boldsymbol{b}) + \mathcal{F}^{-1}(\boldsymbol{R} \cdot \mathcal{F}(\cdot))\right)$, $\theta = \{\}$
2: Initialize $\boldsymbol{W}_{enc}$, $\boldsymbol{b}_{enc}$, $\boldsymbol{W}$, $\boldsymbol{R}$, $\boldsymbol{W}_{dec}$, $\boldsymbol{b}_{dec}$ randomly
3: Add $\boldsymbol{W}_{enc}$, $\boldsymbol{b}_{enc}$, $\boldsymbol{W}$, $\boldsymbol{R}$, $\boldsymbol{W}_{dec}$, $\boldsymbol{b}_{dec}$ to θ
4: Compute $u_0(\boldsymbol{x}) \leftarrow \boldsymbol{W}_{enc}\, a(\boldsymbol{x})^\top + \boldsymbol{b}_{enc}$
5: Compute $u_L(\boldsymbol{x}) \leftarrow ODESolve(f, u(0, \boldsymbol{x}) = u_0(\boldsymbol{x}), t_0 = 0, t_1 = T)$
6: Compute $u(\boldsymbol{x}) \leftarrow (\boldsymbol{W}_{dec} u_L(\boldsymbol{x}))^\top + \boldsymbol{b}_{dec}$
7: Compute $loss = \mathcal{L}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x})$

model. Those properties are especially important when modeling time-dependent PDEs, i.e. dynamical systems. In this section, we propose to explicitly incorporate certain conservation laws into the architecture and later show with numerical experiments that this can, as expected, improve the performance of the architecture.

Suppose, following the notations in Section 2.1, that the input of the network is $a(\boldsymbol{x})$ and the ground truth is $u(\boldsymbol{x})$. The commonly used loss function for Fourier Neural Operators is the discrete L^p norm of the relative difference between the predicted solution and ground truth:

$$\mathcal{L}_{FNO}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x})) = \frac{\|\mathcal{NN}_\theta(a(\boldsymbol{x})) - u(\boldsymbol{x})\|_{L^p}}{\|u(\boldsymbol{x})\|_{L^p}}.$$

To further improve the generalization of the model, we propose to constrain the model by directly incorporate the underlying conservation laws into the loss function. Note that the conservation law is problem-dependent.

For demonstration, let us consider 1-dimensional Burgers' equation

$$\begin{aligned} \partial_t u(x,t) + \partial_x\left(u^2(x,t)/2\right) &= \nu \partial_{xx} u(x,t) && x \in [0,1], t \in (0,1] \\ u(x,0) &= u_0(x) && x \in [0,1] \end{aligned}$$

with periodic boundary conditions. The derivation is essentially the same with other types of boundary conditions. The associated conservation law reads

$$\frac{d}{dt}\int_{x_0}^{x_1} u(x,t)dx = \frac{1}{2}(u(x_0,t)^2 - u(x_1,t)^2) + \nu(u_x(x_1,t) - u_x(x_0,t)) \tag{12}$$

where u_x stands for the derivative of u with respect to x.

Since we use periodic boundary conditions, $u(x_0,t) = u(x_1,t)$ and $u_x(x_0,t) = u_x(x_1,t)$. Therefore by equaiton (12), we have

$$\frac{d}{dt}\int_{x_0}^{x_1} u(x,t)dx = 0.$$

Integrating the equation above over the time domain yields

$$\int_{x_0}^{x_1} u(x,t)dx = \text{constant} \quad \forall t \in [0,\ 1] \tag{13}$$

This conservation law can be used to constrain FNOs for solving parametric 1-dimensional Burgers' equation. Since the algorithm is problem-dependent, we will demonstrate the algorithmic details further in Section 6 when presenting experiments.

5 Benchmarks

Here we give detailed statements of the problems we test for.

1-Dimensional Burgers' Equation

Consider 1-D Burger's Equation

$$\begin{aligned} \partial_t u(x,t) + \partial_x \left(u^2(x,t)/2\right) &= \nu \partial_{xx} u(x,t) && x \in [0,1], t \in (0,1] \\ u(x,0) &= u_0(x) && x \in [0,1] \end{aligned} \tag{14}$$

with periodic boundary conditions (BC) where u_0 is the initial condition (IC) and $\nu \in \mathbb{R}_+$ is the viscosity coefficient. We define $\mathcal{G}_B = \{x_i\}_{i=1}^{N_d}$ where $x_1 = 0$, $x_{N_d} = 1$, $x_i < x_{i+1}$ for $1 \le i \le N_d - 1$, and $\{[x_i,\, x_{i+1}]_{i=1}^{N_d-1}\}$ is a partition for the spatial domain [0, 1]. That is, we discretize the spatial domain with N_d points. Similarly, suppose we discretize the time domain [0, 1] with N_t points $\mathcal{T} = \{t_i\}_{i=1}^{N_t}$ where $t_1 = 0$ and $t_{N_t} = 1$. Assume, for simplicity, that we have uniform meshes: $x_{i+1} - x_i = h$ for $1 \le i \le N_d - 1$ and $t_{k+1} - t_k = h_t$ for $1 \le k \le N_t - 1$ such that h and h_t are two constants. We let $\boldsymbol{x} = [x_1, x_2, ..., x_{N_d}]^\top$, $u_0(\boldsymbol{x}) = [u_0(x_1), u_0(x_2), ..., u_0(x_{N_d})]^\top$, $u(\boldsymbol{x}, 1) = [u(x_1, 1), u(x_2, 1), ..., u(x_{N_d}, 1)]^\top$, and $u(\boldsymbol{x}, \boldsymbol{t}) = [\, u(x_1, t_1), u(x_2, t_1), ..., u(x_{N_d}, t_1),\, u(x_1, t_2), \ldots, u(x_{N_d}, t_2), \ldots, u(x_1, t_{N_t}), \ldots, u(x_{N_d}, t_{N_t})]^\top$.

We consider two parallel problems associated with the 1-dimensional Burgers' equation.

B.1 For the first, we want to construct a mapping $\mathcal{NN}_\theta$ parameterized by θ such that $\mathcal{NN}_\theta$ maps an arbitrary IC function u_0 to the solution of equation (14) defined by u_0 evaluated at time 1. In our case, we assume $u_0(\boldsymbol{x}) \sim \mathcal{P}_B$ where $\mathcal{P}_B = \mathcal{N}(\boldsymbol{0}, 625(-\Delta + 25\boldsymbol{I})^{-2})$ is the multivariate normal distribution with mean vector of zeros and covariance matrix $625(-\Delta + 25\boldsymbol{I})^{-2}$ with periodic boundary conditions. Here, Δ denotes the Laplacian and $\boldsymbol{I}$ denotes the identity matrix. We hence want

$$\mathcal{NN}_\theta\left(u_0(\boldsymbol{x})\right) = u(\boldsymbol{x}, 1)$$

for any discretization $\mathcal{G}_B = \{x_i\}_{i=1}^{N_d}$ and $u_0(\boldsymbol{x}) \sim \mathcal{P}_B$. We define the loss function as

$$\mathcal{L}(\theta, u_0(\boldsymbol{x}), u(\boldsymbol{x}, 1)) := \frac{\|\mathcal{NN}_\theta(u_0(\boldsymbol{x})) - u(\boldsymbol{x}, 1)\|_{L^p}}{\|u(\boldsymbol{x}, 1)\|_{L^p}}$$

where $\|\cdot\|_{L_p}$ denotes the discrete L^p norm. The problem we are trying to solve is

$$\min_\theta \mathbb{E}_{u_0 \sim \mathcal{P}_B}\left[\mathcal{L}(\theta, u_0(\boldsymbol{x}), u(\boldsymbol{x}, 1))\right].$$

Suppose we are given N_{train} observations $(u_0^i(\boldsymbol{x}), u_i(\boldsymbol{x}, 1))$ where $1 \leq i \leq N_{train}$. $u_i(x, 1)$ is the solution at time 1 of (14) defined by $u_0^i(x)$. The problem we want to solve thus becomes

$$\min_{\theta} \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \mathcal{L}(\theta, u_0^i(\boldsymbol{x}), u_i(\boldsymbol{x}, 1)).$$

B.2 For the second, we want to construct a mapping $\mathcal{NN}_\theta$ parameterized by θ such that $\mathcal{NN}_\theta$ maps an arbitrary IC function u_0 to the solution of equation (14) defined by u_0. Again we assume $u_0 \sim \mathcal{P}_B$ where $\mathcal{P}_B = \mathcal{N}(\boldsymbol{0}, 625(-\Delta + 25\boldsymbol{I})^{-2})$ with pediodic boundary conditions. We hence want

$$\mathcal{NN}_\theta\left(u_0(\boldsymbol{x})\right) = u(\boldsymbol{x}, \boldsymbol{t})$$

for any spatial discretization $\mathcal{G}_B$, time discretization $\mathcal{T}$, and $u_0(\boldsymbol{x}) \sim \mathcal{P}_B$. We define the loss function as

$$\mathcal{L}(\theta, u_0(\boldsymbol{x}), u(\boldsymbol{x}, \boldsymbol{t})) := \frac{\|\mathcal{NN}_\theta(u_0(\boldsymbol{x})) - u(\boldsymbol{x}, \boldsymbol{t})\|_{L^p}}{\|u(\boldsymbol{x}, \boldsymbol{t})\|_{L^p}}. \tag{15}$$

We hope to solve the problem

$$\min_{\theta} \mathbb{E}_{u_0 \sim \mathcal{P}_B}\left[\mathcal{L}(\theta, u_0(\boldsymbol{x}), u(\boldsymbol{x}, \boldsymbol{t}))\right].$$

Suppose we are given N_{train} observations $(u_0^i(\boldsymbol{x}), u_i(\boldsymbol{x}, \boldsymbol{t}))$ where $1 \leq i \leq N_{train}$. $u_i(x, t)$ is the solution of (14) defined by $u_0^i(x)$. The problem of interest thus becomes

$$\min_{\theta} \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \mathcal{L}(\theta, u_0^i(\boldsymbol{x}), u_i(\boldsymbol{x}, \boldsymbol{t})).$$

2-Dimensional Darcy Flow

Consider

$$\begin{aligned} -\nabla \cdot (a(x)\nabla u(x)) &= f(x) && x \in (0,1) \times (0,1) \\ u(x) &= 0 && x \in \partial\{(0,1) \times (0,1)\} \end{aligned} \tag{16}$$

with a Dirichlet boundary condition where a is the diffusion coefficient function. Define $\mathcal{G}_D = \{(x_i, y_j)\}_{i,j=1}^{N_d}$ where $x_1 = y_1 = 0$, $x_{N_d} = y_{N_d} = 1$, $x_i < x_{i+1}$ for $1 \leq i \leq N_d - 1$, $y_j < y_{j+1}$ for $1 \leq j \leq N_d - 1$, and $\{[x_i, x_{i+1}] \times [y_j, y_{j+1}]\}_{i,j=1}^{N_d-1}$ is a partition of the domain $[0, 1] \times [0, 1]$. That is, we discretize the domain with N_d^2 points. Assume we have uniform meshes: $x_{i+1} - x_i = y_{j+1} - y_j = h$ for $1 \leq i, j \leq N_d - 1$. To simplify notations, we denote each point (x_i, y_j) by $x_{i,j}$ from now. Let $\boldsymbol{x}_i = [x_{i,1}, x_{i,2}, ..., x_{i,N_d}]^\top$ and $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_{N_d}]^\top$, $a(\boldsymbol{x}_i) = [a(x_{i,1}), a(x_{i,2}), ..., a(x_{i,N_d})]^\top$, $a(\boldsymbol{x}) = [a(\boldsymbol{x}_1), a(\boldsymbol{x}_2), ..., a(\boldsymbol{x}_{N_d})]^\top$, and so forth for $u(\boldsymbol{x}_i)$ and $u(\boldsymbol{x})$.

We want to construct a mapping $\mathcal{NN}_\theta$ parameterized by θ such that $\mathcal{NN}_\theta$ maps an arbitrary diffusion coefficient function a to the solution u of (16) defined by a. We assume

$a(\boldsymbol{x}) \sim \mathcal{P}_D$ where $\mathcal{P}_D = \Phi\left(\mathcal{N}(\boldsymbol{0}, (-\Delta + 9\boldsymbol{I})^{-2})\right)$. $\mathcal{N}(\boldsymbol{0}, (-\Delta + 9\boldsymbol{I})^{-2})$ is the multivariate normal distribution with mean vector of zeros and covariance matrix $(-\Delta + 9\boldsymbol{I})^{-2}$. $\Phi(\cdot)$ is a point-wise threshold function which maps a positive number to 12 and a negative number to 3. Thus, we want

$$\mathcal{NN}_\theta\left(a(\boldsymbol{x})\right) = u(\boldsymbol{x})$$

for any discretization $\mathcal{G}_D = \{(x_i, y_j)\}_{i,j=1}^{N_d}$ and $a(\boldsymbol{x}) \sim \mathcal{P}_D$. We define the loss function as

$$\mathcal{L}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x})) := \frac{\|\mathcal{NN}_\theta(a(\boldsymbol{x})) - u(\boldsymbol{x})\|_{L^p}}{\|u(\boldsymbol{x})\|_{L^p}}.$$

We hope to solve the problem

$$\min_\theta \mathbb{E}_{a\sim\mathcal{P}_B}\left[\mathcal{L}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x}))\right].$$

Suppose we are given N_{train} observations $(a_i(\boldsymbol{x}), u_i(\boldsymbol{x}))$ where $1 \le i \le N_{train}$. u_i is the solution of (16) defined by a_i. The problem of interest thus becomes

$$\min_\theta \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \mathcal{L}(\theta, a_i(\boldsymbol{x}), u_i(\boldsymbol{x})).$$

2-D Navier-Stokes

Consider the vorticity formulation of the 2-dimensional Navier-Stokes equations

$$\begin{aligned} \partial_t w(x,t) + u(x,t)\cdot\nabla w(x,t) &= \nu\Delta w(x,t) + f(x) && x \in (0,1)\times(0,1), t \in (0,T] \\ \nabla\cdot u(x,t) &= 0 && x \in (0,1)\times(0,1), t \in (0,T] \\ w(x,0) &= w_0(x) && x \in (0,1)\times(0,1) \end{aligned} \tag{17}$$

where $u(x,t)$, the velocity field, is a continuous function in t, $w = \nabla \times u$ is the vorticity, w_0 is the initial vorticity, $\nu \in \mathbb{R}_+$ is the viscosity coefficient, and f is the forcing function. We define $\mathcal{G}_N = \{(x_i, y_j)\}_{i,j=1}^{N_d}$ where $x_1 = y_1 = 0$, $x_{N_d} = y_{N_d} = 1$, $x_i < x_{i+1}$ for $1 \le i \le N_d - 1$, $y_j < y_{j+1}$ for $1 \le j \le N_d - 1$, and $\{[x_i,\, x_{i+1}] \times [y_j,\, y_{j+1}]\}_{i,j=1}^{N_d-1}$ is a partition of the domain $[0,\, 1] \times [0,\, 1]$. That is, we discretize the domain with N_d^2 points. Similarly, we discretize the time domain $[0,\, 1]$ with N_t points $\mathcal{T} = \{t_k\}_{k=1}^{N_t}$. Assume we have uniform meshes: $x_{i+1} - x_i = y_{j+1} - y_j = h$ for $1 \le i, j \le N_d - 1$ and $t_{k+1} - t_k = h_t$ for $1 \le k \le N_t - 1$ such that h and h_t are two constants.

Have each discretization point (x_i, y_j) be denoted by $x_{i,j}$. Let $\boldsymbol{x}_i = [x_{i,1}, x_{i,2}, ..., x_{i,N_d}]^\top$ and $\boldsymbol{x} = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_{N_d}]^\top$, $w(\boldsymbol{x}_i, t_k) = [w(x_{i,1}, t_k), w(x_{i,2}, t_k), ..., w(x_{i,N_d}, t_k)]^\top$, $w(\boldsymbol{x}, t_k) = [w(\boldsymbol{x}_1, t_k), w(\boldsymbol{x}_2, t_k), ..., w(\boldsymbol{x}_{N_d}, t_k)]^\top$, and so forth for $w_0(\boldsymbol{x}_i)$ and $w_0(\boldsymbol{x})$. Let $w(\boldsymbol{x}, t_m : t_n)$ $(m \le n)$ denote the solution of (17) evaluated on $\mathcal{G}_N$ at time steps $t_m, t_{m+1}, \ldots, t_n$ stacked up together; that is, $w(\boldsymbol{x}, t_m : t_n) = [w(\boldsymbol{x}, t_m)^\top, w(\boldsymbol{x}, t_{m+1})^\top, \ldots, w(\boldsymbol{x}, t_n)^\top]^\top$.

Let N_0 be such an integer that $1 \le N_0 \le N_t$. We want to construct a mapping $\mathcal{NN}_\theta$ such that $\mathcal{NN}_\theta$ maps the vorticity w of (17) evaluated on $\mathcal{G}_N$ at the first N_0 time steps to w evaluated on $\mathcal{G}_N$ at the last $N_t - N_0$ time steps. We assume $w_0(\boldsymbol{x}) \sim \mathcal{P}_N$ where

$\mathcal{P}_N = \mathcal{N}(\mathbf{0}, 7^{1.5}(-\Delta + 49\boldsymbol{I})^{-2.5})$ is the multivariate normal distribution with mean vector of zeros and covariance matrix $7^{1.5}(-\Delta + 49\boldsymbol{I})^{-2.5}$. In mathematical terms, this means

$$\mathcal{NN}_\theta\left(w(\boldsymbol{x}, t_1 : t_{N_0})\right) = w(\boldsymbol{x}, t_{N_0+1} : t_{N_t})$$

for any discretization $\mathcal{G}_N$ where w is the solution of (17). Define the loss function as

$$\mathcal{L}\left(\theta, w(\boldsymbol{x}, t_1 : t_{N_0}), w(\boldsymbol{x}, t_{N_0+1} : t_{N_t})\right) := \frac{\|\mathcal{NN}_\theta\left(w(\boldsymbol{x}, t_1 : t_{N_0})\right) - w(\boldsymbol{x}, t_{N_0+1} : t_{N_t})\|_{L^p}}{\|w(\boldsymbol{x}, t_{N_0+1} : t_{N_t})\|_{L^p}}$$

where $\|\cdot\|_{L^p}$ denotes the discrete L^p norm. We hope to solve

$$\min_\theta \mathbb{E}_{w_0 \sim \mathcal{P}_N}\left[\mathcal{L}(\theta, w(\boldsymbol{x}, t_1 : t_{N_0}), w(\boldsymbol{x}, t_{N_0+1} : t_{N_t}))\right].$$

Suppose we are given N_{train} observations $(w_i(\boldsymbol{x}, t_1 : t_{N_0}), w_i(\boldsymbol{x}, t_{N_0+1} : t_{N_t}))$ where $1 \leq i \leq N_{train}$ and $\boldsymbol{x}$ corresponds to some discretization $\mathcal{G}_N$. w_i is the solution of (17) defined by $w_i(\boldsymbol{x}, t_1 = 0)$. The problem of interest thus becomes

$$\min_\theta \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \mathcal{L}\left(\theta, w_i(\boldsymbol{x}, t_1 : t_{N_0}), w_i(\boldsymbol{x}, t_{N_0+1} : t_{N_t})\right).$$

We finally make a quick note on the training and testing. All of the problems discussed above are defined upon some discretization $\mathcal{G}$ and $\mathcal{T}$ of the spatial and time domain. However, the discretizations of the training instances and that of the testing instances do not have to be the same. In fact, we train the models on finer discretizations while testing them on much coarser ones.

6 Numerical Results

6.1 NODE-FNOs

We present the performance of our proposed NODE-FNOs in this section. In particular, we consider FNOs modeled by vanilla NODEs (denoted by NODE-FNOs), augmented NODEs (denoted by ANODE-FNOs), NODEs trained with ACA (ACA-NODE-FNOs), and NODEs trained with MALI (MALI-NODE-FNOs). We also consider the ResNet-alike architecture with updating formula defined in equation (11) which we named as FNOs trained with discretized NODEs (dNODE-FNOs).

We test the models on the following problems defined in Section 5: B.1 and B.2 of 1-dimensional Burgers' equation, 2-dimensional Darcy flow equation, and 2-dimensional Navier-Stokes equations. Without further notice, we assume that the norm used in the loss function is the L^2-norm; that all our models are trained with 1000 instances while being tested with 200 instances that have not been seen by the models during training; and that the models to which the adjoint method does not apply are trained with the Adam method [11]. The results are presented in Figures 1, 2, 3, and 4.

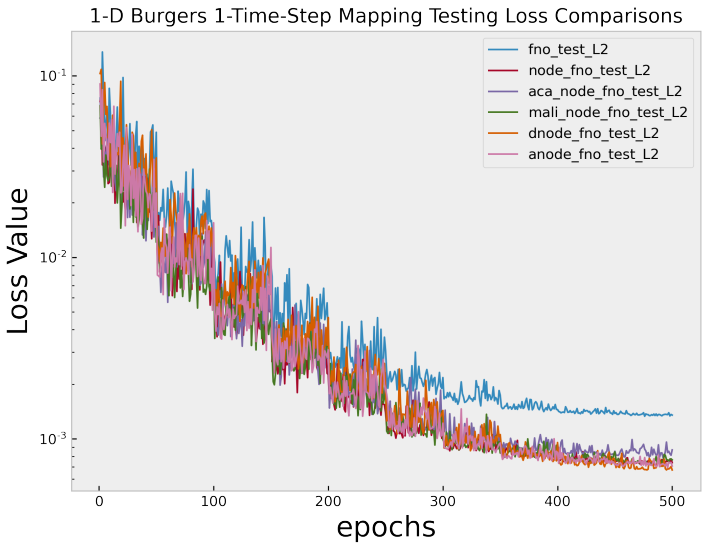

Figure 1: Comparison between the evolution of the testing losses between the vanilla FNO and variants of NODE-FNOs on problem B.1. Discretization size of the spatial domain for training instances: 512. Discretization size of the spatial domain for testing instances: 2048. Number of epochs: 500. Initial learning rate: 0.01. Learning rate decay: 0.5 per 50 epochs.

Figure 1 shows the comparison between the performance of the vanilla FNO and the variants of NODE-FNO when solving problem B.1. We see that the generalization error of the NODE-FNOs is better than that of the vanilla FNO for at least 40.8% and at most 49.4% with dNODE-FNO achieving the best. Figure 2 shows the comparisons of the performance of the models when solving problem B.2. The result is similar to the previous case: all variants of NODE-FNOs achieve lower generalization errors than the vanilla FNO. In particular, NODE-FNOs are better than the vanilla FNOs for at least 17.5% and at most 33.3%, demonstrating the superior performance of NODE-FNOs.

Figure 3 shows the comparisons of the performance of the models when solving the 2-dimensional Darcy flow problem. In the original paper [17], the authors added a normalization step before feeding the input to the network. However, they used the same normalizer for both the training set and the testing set: subtracting the same mean and divided by the same standard deviation for both the training set and teh testing set. Our test shows that this does make the vanilla FNO perform better (which we still do not understand why), but such an implementation would no longer allow for testing the model on instances with finer discretizations than those of the training instances. We thus modified it in such a way that the training set and testing set have their own normalizers. Then we train the models on 1000 instances discretized with a grid of size 85×85 and test them on 200 instances discretized with a grid of size 421×421. It is clear from the figure that that all variants of NODE-FNOs achieve lower generalization error than the vanilla FNO with an improvement range of 32.1% to 36.8%.

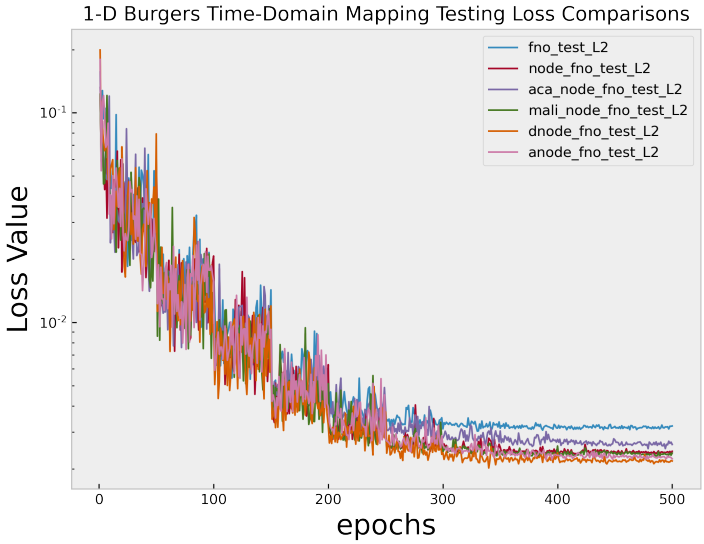

Figure 2: Comparison between the evolution of the testing losses between the vanilla FNO and variants of NODE-FNOs on problem B.2. Discretization size of the time domain and the spatial domain for training instances: 51×64. Discretization size of the time domain and the spatial domain for testing instances: 101×128. Number of epochs: 500. Initial learning rate: 0.01. Learning rate decay: 0.5 per 50 epochs.

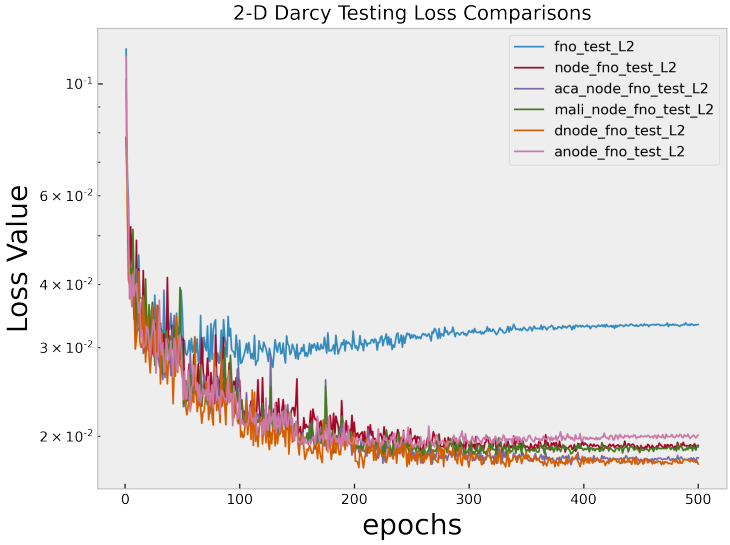

Figure 3: Comparison between the evolution of the testing losses between the vanilla FNO and variants of NODE-FNOs on the 2-D Darcy Flow problem introduced in Section 5. Discretization size of the spatial domain for training instances: 85×85. Discretization size of the spatial domain for testing instances: 421×421. Number of epochs: 500. Initial learning rate: 0.01. Learning rate decay: 0.5 per 50 epochs.

Figure 4 shows the comparisons of the performance of the models when solving for the 2-dimensional Navier-Stokes equations described in Section 5. In our case, N_t, the size of time-domain discretization is chosen as 50. We set $N_0 = 10$, the number of time steps the models can see during training. Again, to make the models generalizable, we modify the normalization step so that the training set and the testing set do not share the same normalizer. However, we found that if we normalize the data in this way, all of the models actually perform a lot worse than if there were not normalization at all. Therefore, we removed the normalization step. We found that in this case NODE-FNO performs a bit worse than the vanilla FNO, potentially because of inaccurate gradient estimations. Despite this, the other NODE-FNO variants still achieve superior performance, with the best one having a generalization error of 28.6% lower than the vanilla FNO.

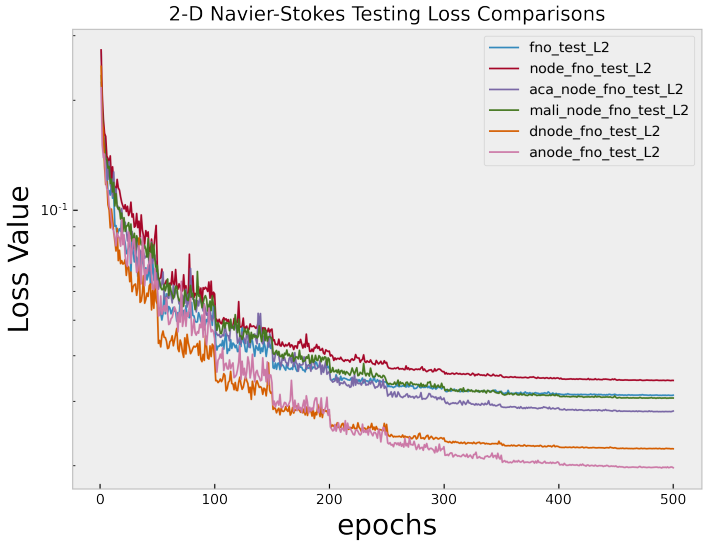

Figure 4: Comparison between the evolution of the testing losses between the vanilla FNO and variants of NODE-FNOs on the 2-dimensional Navier-Stokes equations introduced in Section 5. Discretization size of the spatial domain for training instances: 64×64. Discretization size of the spatial domain for testing instances: 64×64. Number of epochs: 500. Initial learning rate: 0.01. Learning rate decay: 0.5 per 50 epochs.

6.2 Conservative FNOs

We consider the case B.2 for 1-dimensional Burgers' equation in this section; namely, the problem of mapping an initial condition to the solution on the whole time domain. Following the notations in Section 5, let $\mathcal{G}_B = \{x_i\}_{i=1}^{N_d}$ and $\mathcal{T} = \{t_i\}_{i=1}^{N_t}$ be the discretization points on the spatial and time domain respectively. The conservation law for 1-D Burgers' equation has been developed in equation (13), namely

$$\int_0^1 u(x,t)dx = \text{constant} \quad \forall t \in [0,\, 1].$$

This implies that

$$\int_0^1 u(\boldsymbol{x}, t_i)dx = \int_0^1 u(\boldsymbol{x}, t_j)dx$$

for all $1 \leq i, j \leq N_t$ where $\boldsymbol{x}$ is defined in Section 5. This gives a natural choice of an additional loss term

$$\left(\int_0^1 [u(\boldsymbol{x}, t_i) - u(\boldsymbol{x}, t_j)]dx\right)^2.$$

In practice, however, it is not likely to compute the difference for every pair of (i, j) since N_t is typically large. Therefore, we sample N_s points $\mathcal{T}_s = \{t_{i_k}\}_{k=1}^{N_s}$ from $\mathcal{T}$ first and only compute the difference between $\left(\int_0^1 [u(\boldsymbol{x}, 1) - u(\boldsymbol{x}, t_{i_k})]dx\right)^2$ where $u(\boldsymbol{x}, 1)$ is the solution at time 1 evaluated on $\mathcal{G}_B$. We now define the unsupervised loss

$$\mathcal{L}_C(\theta, a(\boldsymbol{x})) := \frac{1}{N_s}\sum_{k=1}^{N_s}\left(\int_0^1 \left[\mathcal{NN}_\theta(a(\boldsymbol{x}))|_{t=1} - \mathcal{NN}_\theta(a(\boldsymbol{x}))|_{t=t_{i_k}})\right] dx\right)^2.$$

The loss function for C-FNOs for solving 1-dimensional Burgers' equation is then

$$\mathcal{L}_{CFNO}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x})) := \mathcal{L}_{FNO}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x})) + \mathcal{L}_C(\theta, a(\boldsymbol{x}))$$

where $\mathcal{L}_{FNO}$ has been defined in equation (15). The algorithmic details for the C-FNO for 1-D Burgers' equation is summarized in Algorithm 3.

Algorithm 3 C-FNO for 1-D Burgers' equation

Input: $\mathcal{NN}_\theta$, $a(\boldsymbol{x})$, $\mathcal{T}$, N_s
Output: $u(\boldsymbol{x}, \boldsymbol{t})$

1: Sample N_s points $\{t_{i_k}\}_{k=1}^{N_s}$ from $\mathcal{T}$
2: Compute $\mathcal{NN}_\theta(a(\boldsymbol{x}))$
3: Compute $loss = \mathcal{L}_{CFNO}(\theta, a(\boldsymbol{x}), u(\boldsymbol{x}))$
4: Backpropagation with Adam optimizer

The numercal result is presented in Figure 5. Despite the training loss of C-FNO is no better than that of the vanilla FNO, the generalization error of C-FNO achieves 22% lower. This is expected because the architecture has learned about the underlying physical conservation law during training. Therefore, the solution output solution gets closer to the true physical phenomenon for which the PDE simulates.

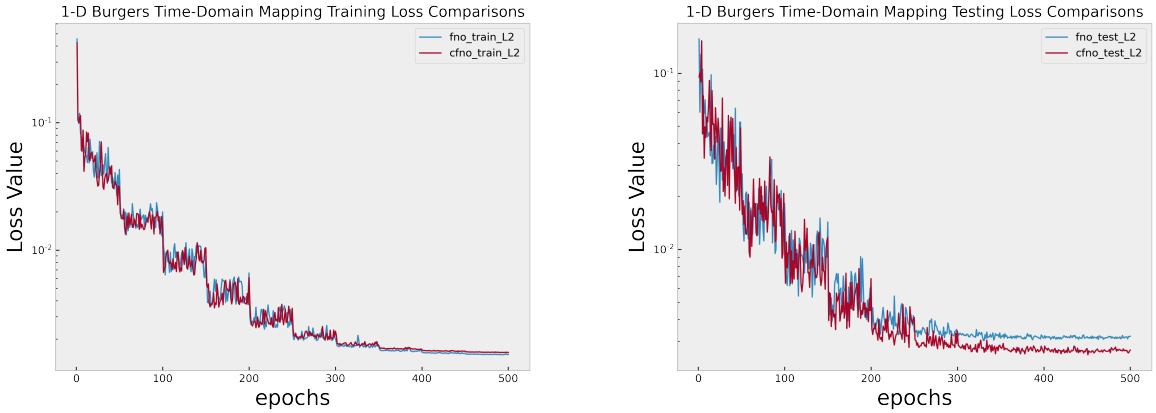

Figure 5: Comparison between the performance of the vanilla FNO and C-FNO on problem B.2. Left: evolution of training losses. Right: evolution of testing losses. Discretization size of the time domain and the spatial domain for training instances: 51×64. Discretization size of the time domain and the spatial domain for testing instances: 101×128. Number of epochs: 500. Initial learning rate: 0.01. Learning rate decay: 0.5 per 50 epochs.

7 Discussion and Conclusion

We have proposed two independent ways of improving FNOs: modeling FNOs as Neural ODEs and enforcing conservation laws on FNOs (C-FNOs) by incorporating the conservation into the loss function. For NODE-FNOs, we modelled FNOs with vanilla NODEs as well as more advanced variants such as augmented NODEs, NODEs trained with the ACA method, NODEs trained with the MALI method, etc. We tested NODE-FNOs on 1-dimensional Burgers' equation, 2-dimensional Darcy flow problem, and 2-dimensional Navier-Stokes equations. For 1-dimensional Burgers', we consider both the case of mapping initial condition to the solution at the first time step and the case of mapping initial condition to the solution on the whole time domain. We found that NODE-FNO and its variants beat the vanilla FNO in all the benchmarks by a good amount. For C-FNOs, we tested on 1-dimensional Burgers' equation with whole-time-domain mapping. We found that despite the training loss of the C-FNO did not get lower than the vanilla FNO, the generalization error was reduced by 22%. In both cases, we have seen improvement of our proposed models compared to the baseline. We hope such findings may give some insights in improving neural operator architectures in the future.

References

[1] H. K. Aggarwal, M. P. Mani, and M. Jacob, *Modl: Model-based deep learning architecture for inverse problems*, IEEE transactions on medical imaging, 38 (2018), pp. 394–405.

[2] Y. Bar-Sinai, S. Hoyer, J. Hickey, and M. P. Brenner, *Learning data-driven discretizations for partial differential equations*, Proceedings of the National Academy of Sciences, 116 (2019), pp. 15344–15349.

[3] J. Brandstetter, D. Worrall, and M. Welling, *Message passing neural pde solvers*, arXiv preprint arXiv:2202.03376, (2022).

[4] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, *Neural ordinary differential equations*, Advances in neural information processing systems, 31 (2018).

[5] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of control, signals and systems, 2 (1989), pp. 303–314.

[6] E. Dupont, A. Doucet, and Y. W. Teh, *Augmented neural odes*, Advances in Neural Information Processing Systems, 32 (2019).

[7] D. Greenfeld, M. Galun, R. Basri, I. Yavneh, and R. Kimmel, *Learning to optimize multigrid pde solvers*, in International Conference on Machine Learning, PMLR, 2019, pp. 2415–2423.

[8] J. Han, A. Jentzen, and W. E, *Solving high-dimensional partial differential equations using deep learning*, Proceedings of the National Academy of Sciences, 115 (2018), pp. 8505–8510.

[9] K. Hornik, M. Stinchcombe, and H. White, *Multilayer feedforward networks are universal approximators*, Neural networks, 2 (1989), pp. 359–366.

[10] J.-T. Hsieh, S. Zhao, S. Eismann, L. Mirabella, and S. Ermon, *Learning neural pde solvers with convergence guarantees*, arXiv preprint arXiv:1906.01200, (2019).

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[12] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, *Machine learning–accelerated computational fluid dynamics*, Proceedings of the National Academy of Sciences, 118 (2021), p. e2101784118.

[13] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Neural operator: Learning maps between function spaces*, arXiv preprint arXiv:2108.08481, (2021).

[14] J. N. Kutz, *Deep learning in fluid dynamics*, Journal of Fluid Mechanics, 814 (2017), pp. 1–4.

[15] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, nature, 521 (2015), pp. 436–444.

[16] Z. Li, D. Z. Huang, B. Liu, and A. Anandkumar, *Fourier neural operator with learned deformations for pdes on general geometries*, arXiv preprint arXiv:2207.05209, (2022).

[17] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Fourier neural operator for parametric partial differential equations*, arXiv preprint arXiv:2010.08895, (2020).

[18] ——, *Neural operator: Graph neural operator for partial differential equations*, arXiv preprint, arXiv:2003.03485, (2020).

[19] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, A. Stuart, K. Bhattacharya, and A. Anandkumar, *Multipole graph neural operator for parametric partial differential equations*, Advances in Neural Information Processing Systems, 33 (2020), pp. 6755–6766.

[20] Z. Li, H. Zheng, N. Kovachki, D. Jin, H. Chen, B. Liu, K. Azizzadenesheli, and A. Anandkumar, *Physics-informed neural operator for learning partial differential equations*, arXiv preprint arXiv:2111.03794, (2021).

[21] 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*, Journal of Computational physics, 378 (2019), pp. 686–707.

[22] J. C. Ye, Y. Han, and E. Cha, *Deep convolutional framelets: A general deep learning framework for inverse problems*, SIAM Journal on Imaging Sciences, 11 (2018), pp. 991–1048.

[23] J. Zhuang, N. Dvornek, X. Li, S. Tatikonda, X. Papademetris, and J. Duncan, *Adaptive checkpoint adjoint method for gradient estimation in neural ode*, in International Conference on Machine Learning, PMLR, 2020, pp. 11639–11649.

[24] J. Zhuang, N. C. Dvornek, S. Tatikonda, and J. S. Duncan, *Mali: A memory efficient and reverse accurate integrator for neural odes*, arXiv preprint arXiv:2102.04668, (2021).