Neural Operators on Finite Element Spaces

Nadav Port

Advisor: Professor Colin Cotter

September 6, 2024

Contents

1 Introduction

Machine learning, and in particular Deep Neural Networks (DNNs), have empirically proven successful in diverse applications, such as image classification, natural language processing, text-to-speech, and more. Mathematically, most of these problems can be formulated as approximating maps between finite-dimensional spaces, e.g. image classification can be thought of as a mapping between the space of possible pixel values to labels. Nevertheless, many problems require a more general, infinite-dimensional setting. These problems are concerned with operators, which are maps between infinite-dimensional spaces. A prominent example of operator problems is Partial Differential Equations (PDEs) that lack analytical solutions. To answer these problems, a machine learning method for approximating operators, termed "operator learning", has been developed for quite some time.

A few DNN architectures and theories have been proposed for operator learning, first appearing in 1995 [2]. More recent work has introduced architectures such as Multiwavelet networks [3], DeepONetworks [4], and Neural Operators [7].

This work will be based on the latter "Neural Operator" approach, first proposed as Graph and Fourier Neural Operators [5] [6], and developed further in [1] [7]. We aim to describe this construction under the finite element method, taking advantage of its extensively researched theory and software. Instead of discretizing functions in the usual sense, we represent them as coefficients in the function basis of a finite element space, thus providing an explicit formula for the calculations that compose the neural operator.

Ultimately, we have three broad goals; the first is to replicate known results of neural operators in our finite element framework. Second, we wish to discover relations between the architecture of neural operators, i.e. their hyperparameters, and how well they approximate a given problem. These would shed light on how to optimize a neural operator to a particular problem or need. Third, we want to characterize some limitations of the architecture and criticize it.

Hopefully, one could consult the limitations we present before deciding on a neural operator as an approximation method to a complicated PDE, and gain insights from the relations we unravel if they do attempt to use it.

2 Literature Review

2.1 Machine learning framework

The neural operator is first and foremost a data-driven approximation that lies in the realm of machine learning. Thus, we need to consider a supervised learning approach for operators in infinite-dimensions.

Given an operator between two function spaces $\mathcal{G} : \mathcal{A} \to \mathcal{U}$ on domain $\Omega \subset \mathbb{R}^d$, we'd like to construct a surrogate operator $\mathcal{N}$ that will "learn" to imitate the action of $\mathcal{G}$ to arbitrary precision. Let $\{(a^{(i)}, u^{(i)})\}_{i=1}^S \subset \mathcal{A} \times \mathcal{U}$ be pairs of input-outputs, i.e. $u(x) = (\mathcal{G}a)(x)$, $x \in \Omega$, where $a^{(i)} \sim \mu$ are i.i.d samples drawn from a probability measure μ on $\mathcal{A}$. The approximating operator $\mathcal{N}_\theta : \mathcal{A} \to \mathcal{U}$ (later to be introduced as the neural operator) is an operator that is dependent on parameters $\theta \in \mathbb{R}^p$ for some $p \in \mathbb{N}$.

In statistical learning terms, we consider a set of approximating functions called the hypothesis class, $\{\mathcal{N}_\theta | \theta \in \mathbb{R}^p\}$, and look for a function that minimizes a loss/objective function. Our chosen loss function is simply the error norm, so we wish to minimize w.r.t. θ,

$$\mathbb{E}_{a\sim\mu} \| (\mathcal{G}a)(x) - (\mathcal{N}_\theta a)(x) \|_{\mathcal{U}} \,, \tag{2.1}$$

which is also known as the "true risk".

In general, we don't know explicitly $(\mathcal{G}a)(x)$ for all $a \sim \mu$ and $x \in \Omega$ (otherwise the problem is redundant), we only have access to a finite number of samples $i = 1, \ldots, S$. Thus, as is customary in machine learning [8] (ch. 8), we turn to empirical-risk minimization,

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{S} \sum_{i=1}^{S} \| u^{(i)}(x) - (\mathcal{N}_\theta a^{(i)})(x) \|_{\mathcal{U}}, \tag{2.2}$$

which approximately minimizes the true risk.

In what follows, we assume that the minimizer of the empirical risk converges to the minimizer of the true risk with $S \to \infty$ [17].

2.2 Neural Operators

2.2.1 Definition

Let $\mathcal{G} : \mathcal{A} \to \mathcal{U}$ be a nonlinear operator acting between Banach function spaces of 1D functions. We denote the input and output functions as $a, u : \Omega \subset \mathbb{R} \to \mathbb{R}$.

Definition 2.1 — Neural Operator. A Neural Operator $\mathcal{N}$ is constructed as follows [1],

$$\begin{aligned} &\mathcal{N} : \mathcal{A} \to \mathcal{U} \\ &\mathcal{N} \overset{\text{def.}}{=} \mathcal{Q} \circ \mathcal{L}_L \circ \ldots \circ \mathcal{L}_1 \circ \mathcal{R} \end{aligned} \tag{2.3}$$

with $L, D \in \mathbb{N}$ and,

- **Lifting operator** - $\mathcal{R}$ is a pointwise operator that maps the input function a to a D-valued function using a fully connected layer, i.e. $\mathcal{R}a(x) = Wa(x) + \boldsymbol{b}$ with $W, \boldsymbol{b}$ learnable network parameters of dimensions $1 \times D, D$. $\boldsymbol{b}$ can be thought of as constant functions (see remark below).

- **Lowering operator** - $\mathcal{Q}$ (a.k.a "projection" operator) is a pointwise operator that maps the output from $\mathcal{L}_L$, which is a D-valued function, to the output in $\mathcal{U}$, using a fully connected DNN, i.e. $\mathcal{Q}\boldsymbol{v}(x) = W\boldsymbol{v}(x)+\boldsymbol{b}$, with $W, \boldsymbol{b}$ learnable network parameters of dimensions $1 \times D, D$.

- **Inner-layer Operator** - $\{\mathcal{L}_l\}_{l=1}^L$ are non-local, non-linear operators defined by,

$$(\mathcal{L}_l\boldsymbol{v})(x) \overset{\text{def.}}{=} \sigma\left(W_l\boldsymbol{v}(x) + \boldsymbol{b}_l + \sum_{m=1}^{M} \langle T_{l,m}\boldsymbol{v}(x), \psi_m(x)\rangle_{L^2}\psi(x)\right), \tag{2.4}$$

  where σ is a nonlinear pointwise function, $T_{l,m}, W_l$, and $\boldsymbol{b}_l$ are matrices and vector of learnable network parameters of dimensions $D \times D$, D, and $\{\psi\}_{m=1}^M$ are functions $\psi_m : \Omega \to \mathbb{R}^D$ called projection functions.

Remarks

- A more general formulation is to replace the sum of inner products in (2.4) with an integral kernel operator as in [7], and consider multidimensional domain Ω or range. We chose to use this formulation as it is better suited for integration with FEM. This formulation is called "Non Local Neural Operator" in [1] and termed "Low-rank Neural Operator" when using the more general framework.

- Generally the bias $\boldsymbol{b}_l$ are functions, but as discussed in p. 13 of [7], we can use constant functions (i.e. parameters) without disrupting the theory or practice.

- The neural operator $\mathcal{N}$ is dependent on the learnable parameters in $T_{l_m}, W_l, \boldsymbol{b}_l$, and those appearing in the implementation of the lifting and lowering operators. We concatenate all of the these parameters into one vector denoted as $\theta \in \mathbb{R}^p$, and sometimes write $\mathcal{N}_\theta$ instead of $\mathcal{N}$ to emphasize the dependence on those parameters.

- Due to time constraints, the implementation part of this work will focus solely on *Fourier neural operator*, which is a neural operator as defined above with Fourier modes for projection functions $\{\psi\}_{m=1}^M$.

When implemented on a computer, the neural operator has to be reduced from infinite-dimensions to finite-dimensions, and is usually denoted by $\hat{\mathcal{N}}$. The network parameters of $\hat{\mathcal{N}}$ are the same as for $\mathcal{N}$, as these are already finite matrices/vectors, but now the operator must take a finite-dimensional input. In the literature, the discretization is usually done by evaluating the input-output functions on an N point discretization of Ω, denoted as $P_N = \{x_1, \ldots, x_N\}$ [6] (p. 4). In Section 3, we provide an alternative discretization using FEM and basis coefficients.

For now, it means that the error we measure on the computer is between $\hat{\mathcal{N}} : \mathbb{R}^N \to \mathcal{U}$ and the original operator $\mathcal{G}$. Reformulating [7] (p. 12), the error can be separated into two,

$$\|\hat{\mathcal{N}}(a|_{P_N}) - \mathcal{G}a\|_{\mathcal{U}} \leq \underbrace{\|\hat{\mathcal{N}}(a|_{P_N}) - \mathcal{N}a\|_{\mathcal{U}}}_{\text{discretization error}} + \underbrace{\|\mathcal{N}a - \mathcal{G}a\|_{\mathcal{U}}}_{\text{approximation error}} . \tag{2.5}$$

We will expand upon these theoretical error bounds in Section 3.2.

2.2.2 Approximation theory

The theory of neural operators is still mostly unestablished, but a few results regarding approximation theory have been proven. We will state a simplified version of the result most important to this work, and reference the relevant paper for a proof. Generally, we assume the function domain Ω to be bounded with Lipschitz boundary, and we'll restrict ourselves to Sobolev Hilbert spaces $H^1(\Omega) = W^{1,2}(\Omega)$.

Theorem 2.2 — D convergence. Let $\Omega \subset \mathbb{R}$, $\mathcal{G} : H^1(\Omega) \to H^1(\Omega)$ a continuous operator acting between Sobolev spaces. Then, for any fixed $L \geq 1$, $M \geq 0$, and $\forall \epsilon > 0$ there exists a neural operator $\mathcal{N}_\theta$ for some $\theta \in \mathbb{R}^p$, such that

$$\forall a \in K : \quad \|(\mathcal{G}a)(x) - (\mathcal{N}_\theta a)(x)\|_{H^1} \leq \epsilon, \tag{2.6}$$

where $K \subset H^1$ is a compact set. Furthermore, we can choose the functions ψ_m in (2.4) to be the identity, or more sophisticated functions.

The full statement and proof are theorem 1.2 and appendix A.6 in [1].

L, M are fixed in the above theorem, $(\psi_m)_m$ can be set to unity, and we assume θ is optimized, then the only possible variability in the neural operator architecture is in the number of lifting channels D, meaning,

$$\|(\mathcal{G}a)(x) - (\mathcal{N}_{\theta(D)}a)(x)\|_{H^1} \xrightarrow{D\to\infty} 0. \tag{2.7}$$

Moreover, the theorem implies the universal-approximation property for neural operators, i.e. that there exists a neural operator $\mathcal{N}$ that approximates $\mathcal{G}$ to infinitesimal accuracy. Based on this, the approximation error in (2.7) is arbitrarily small. See [7] (ch. 8) for a thorough discussion of universal-approximation for neural operators, where the focus is on the density of neural operators in a certain wide set of operators, rather than on the architecture of the operator and D dependence.

Another important result is called discretization-invariance,

Theorem 2.3 — Discretization-Invariance of neural operators. Let $\Omega \subset \mathbb{R}$, $\mathcal{G} : H^1(\Omega) \to H^1(\Omega)$ a continuous operator acting between Sobolev spaces. Let $\mathcal{N}$ be a neural operator as defined in Definition 2.1 with continuous $\sigma(x)$. Then $\mathcal{N}$ is discretization-invariant.
That is, given a sequence of partitions $(P_N)_{N=1}^{\infty}$ of the domain Ω, there exists a corresponding

sequence of discretized neural operators $(\hat{\mathcal{N}}_N)_{N=1}^{\infty}$ such that

$$\forall a \in K: \quad \|\hat{\mathcal{N}}_N(a|_{P_N}) - \mathcal{G}(a)\|_{H^1} \xrightarrow{N\to\infty} 0.$$

where $K \subset H^1$ is a compact set.

The full statement and proof are definitions 1-4, theorem 8, and appendix E in [7].

This theorem assures us that the discretization error in (2.7) can be controlled by the number of discretization points N.

2.2.3 Drawbacks

It's important to keep in mind that the neural operator method is limited by traditional solvers in order to create data. If a solver is only accurate up to 0.01%, then even if a neural operator is trained practically zero test loss, it's still unfeasible for it to have an accuracy of more than 0.01% when compared against the analytical solution.

Additionally, although Theorem 2.2 promises that in theory the error between the surrogate and original operators converges to zero, in practice it raises questions. For example, what's the decline rate of the error? For a given channel choice D, which choice of hyperparameters, $L, M, (\psi_m)_m$ provides the least error? Under limited computational resources (time and/or memory), what combination of hyperparameters is the best? If the total number of parameters p is to remain constant, should we increase D on behalf of L, or vice versa? Counting on discretization invariance, to what extent can a low resolution neural operator approximate high resolutions?

Confined to specific PDE examples, we hope to analyze numerically some of these questions in what follows, (most prominently in Section 4).

2.3 Finite element spaces

The finite element method (FEM) is a method widely used in engineering, physics, and applied mathematics for solving PDEs. It can handle complex multidimensional geometries, complex boundary conditions, make use of parallel computing, and can be applied to diverse problems. FEM is slow compared to a forward pass of a neural network, thus a surrogate neural operator model can aid by providing quick, low-resolution approximations to the problem. One application that may find it useful is weather forecasting, as it uses multiple low-resolution models to determine probabilities of weather scenarios (an attempt at weather forecasting with neural operators was made in [14]). Furthermore, many of today's real-world numerical simulations rely on FEM, making the potential impact of surrogate neural operator models even bigger.

In this section, we briefly discuss FEM basics that will be put to use later. The main concepts we aim to define are the CG1, CG3, and Hermite finite element spaces (FES). For a more rigorous discussion, see [16].

Let $\mathcal{T}$ be a subdivision of the domain Ω into a finite number of non-overlapping elements K, also called a triangulation. We use the standard definition of a finite element on K,

Definition 2.4 — Finite element. $(K, \mathcal{P}, \{\varphi_1, \dots, \varphi_k\})$ is called a finite element where,

1. K is the element domain
2. $\mathcal{P}$ is a finite dimensional space of functions on K
3. $\{\varphi_1, \dots, \varphi_k\}$ is a basis for the dual space $\mathcal{P}^*$ called nodal variables.

We also call the basis $\{\phi_i\}_{i=1}^k$ of $\mathcal{P}$ that satisfies $\varphi_i(\phi_j) = \delta_{ij}$, a local nodal basis.

With that, we can build a finite element space,

Definition 2.5 — Finite element space. Let $\mathcal{T}$ be a triangulation with triangles K_i and a corresponding finite element on each triangle. The space V of functions on $\bigcup \mathcal{T}$ is called a finite element space if every $u \in V$ satisfies $\forall K_i \in \mathcal{T}$, $u|_{K_i} \in \mathcal{P}_i$, where $\mathcal{P}_i$ is the function space of the i'th finite element.

2.3.1 CGk (Lagrange) spaces

We now introduce a specific 1D finite element that will be used to construct our desired FES.

Definition 2.6 — 1D Lagrange finite element. A 1D Lagrange element $(K, \mathcal{P}, \{\varphi_1, \dots, \varphi_{k+1}\})$ of degree k is defined by,

1. $K = [a, b]$, $a, b \in \mathbb{R}$
2. $\mathcal{P}$ is the space of k degree polynomials on K
3. The nodal variables are point-evaluations, i.e. for $v \in \mathcal{P}$, $\varphi_i(v) = v(x_i)$, where $x_i = a + \frac{i-1}{k}(b - a)$, $i = 1, ..., k + 1$.

To define the Lagrange FES, we need to distinguish two cases. In the simple one, the functions in the space are simply stitched together at the edges of the elements with no regard for continuity, thus we get a FES with discontinuous functions which is termed "discontinuous Galerkin" (DG). In the other case, we enforce continuity by properly matching nodal variables on different elements (in a process called global geometric decomposition), that way we end up with a space of continuous functions called "continuous Galerkin" (CG). A CG1 space is a continuous Galerkin space where the composing finite elements are Lagrange degree 1 (affine polynomials).

The space has global basis functions, which we call the nodal basis during this work (not to be confused with the local nodal basis above). The nodal basis functions extend the local basis functions to the whole domain Ω, but are taken to be 0 in most of the domain outside of their associated element. The amount of global basis functions, which is also termed the degrees-of-freedom (DOF) of the space and denoted here by N, depends on the number of finite elements, their type, and continuity constraints.

The above is summed up in the next definition.

Definition 2.7 — CGk finite element space. Let $a = x_1 < \cdots < x_N = b$ be N equidistant points on the interval $\Omega = [a, b]$. The CGk space V on Ω is a finite element space consisting of degree k Lagrange finite elements on the intervals $[x_{i-1}, x_i]$, with enforced C^0 continuity. We denote its nodal basis by $\{\phi_n\}_{n=1}^{kN}$.

As such, every function $u(x) \in V$ can be represented in the $(\phi_n)_n$ basis as

$$u(x) = \sum_{n=1}^{kN} u_n \phi_n(x)\,,$$

where the coefficients are point-evaluations $u_n = u(x_n)$.

2.3.2 Hermite space

As Lagrange elements have no constraints on derivatives, functions of CGk spaces are in general not differentiable (although they have weak derivatives). This is an issue when we want to solve a PDE that requires a C^1 continuity, such as the KS equation below in Section 5. For these problems we define a different finite element that involves derivatives.

Definition 2.8 — 1D Hermite finite element. A 1D Hermite element $(K, \mathcal{P}, \{\varphi_1, \ldots, \varphi_4\})$ is defined by,

1. $K = [a, b]$, $a, b \in \mathbb{R}$
2. $\mathcal{P}$ is the space of degree 3 polynomials on K, $\operatorname{span}\{1, x, x^2, x^3\}$
3. The nodal variables are point-evaluations and derivative point-evaluations, i.e. for $v \in \mathcal{P}$, $\varphi_1(v) = v(a), \varphi_2(v) = v'(a), \varphi_3(v) = v(b), \varphi_4(v) = v'(b)$.

The Hermite finite element enables us to enforce C^1 continuity. This time, we "stitch" together not only the functions on the edges of each element K_i, but also the derivative of the function. This makes the Hermite space a subspace of CG3, as both consist of degree 3 polynomials, but Hermite enforces differentiability of the functions. Note that the coefficients of the nodal basis expansion are not only point-evaluations, but also derivative point-evaluations.

Definition 2.9 — Hermite (HER) finite element space. Let $a = x_1 < \cdots < x_N = b$ be N equidistant points on the interval $\Omega = [a, b]$. The Hermite space V on Ω is a finite element space consisting of 1D Hermite finite elements on the intervals $[x_{i-1}, x_i]$, with enforced C^1 continuity. We denote its nodal basis by $\{\phi_n\}_{n=1}^{2N}$.

As such, every function $u(x) \in V$ can be represented in the $(\phi_n)_n$ basis as

$$u(x) = \sum_{n=1}^{2N} u_n \phi_n(x)\,,$$

where the coefficients are either point-evaluations $u_n = u(x_n)$ or derivative point-evaluations $u_n = u'(x_n)$, depending on n.

3 Numerical analysis

Neural operators are formulated on infinite-dimensional function spaces, such as L^2 or H^1. Unfortunately, computers are discretized machines, so implementations of operators are inevitably restricted to finite-dimensional spaces. As an example of this difference, we can consider the input to the operator. Although the input to a neural operator $\mathcal{N}$ is a function $a(x)$ in theory, in practice it is a finite array of function values $[a(x_1), \dots, a(x_N)]$. We call this step of the analysis discretization, and although it can be implemented in different ways, recent research has seen only straight forward point-evaluation of functions like the input array.

Our approach is to replace the finite-dimensional discretization step with a FEM framework. Instead of describing an input function $a(x)$ as an array of point-evaluations, we describe it as an array of *coefficients* in the nodal basis $\{\phi_n(x)\}_{n=1}^N$ of some FES V. We do not focus much on the functional analysis theory of infinite-dimensional neural operators, rather, we show that the finite-dimensional discretization part can be done with FEM, and might even prove more advantageous than traditional implementations.

3.1 Neural operators in finite element spaces

Let V be a finite element space with N DOF and nodal basis $\{\phi_n(x)\}_{n=1}^N$. Each function $u \in V$ can be represented as $u = \sum_{n=1}^N u_n\phi_n(x)$, where the coefficients $\{u_n\}_{n=1}^N$ are real numbers. We aim to describe the neural operator framework presented in Definition 2.1 using only manipulation of the coefficients u_n under the space V, rather than operators between infinite-dimensional Banach spaces. Note that the implementation is independent of the resolution N, which is to be expected from the aforementioned discretization-invariance property.

3.1.1 Lifting and lowering operators

The lifting operator $\mathcal{R}$ is first in the sequence of operators in (2.3). It's responsible for "fanning-out" $u(x)$ into D channels, it takes in a 1D function and outputs a "stack" of D functions,

$$(\mathcal{R}u)(x) = \boldsymbol{u}(x) \overset{\text{def.}}{=} \begin{bmatrix} u_1(x) & \dots & u_D(x) \end{bmatrix}^T . \tag{3.1}$$

It is implemented as a fully-connected (linear) layer that takes $(u(x), x)$ as input and returns functions $\{u_i(x)\}_{i=1}^D$ as output. Using coefficients it returns,

$$\underbrace{\begin{bmatrix} u_1 & x_1 \\ \vdots & \vdots \\ u_N & x_N \end{bmatrix}}_{\text{input}} \begin{bmatrix} a_{11} & \dots & a_{1D} \\ a_{21} & \dots & a_{2D} \end{bmatrix} + \begin{bmatrix} b_1 & \dots & b_D \\ \vdots & & \vdots \\ b_1 & \dots & b_D \end{bmatrix}, \tag{3.2}$$

where a_{ij}, b_j are learnable network parameters. The result is a $N \times D$ matrix, but we take the transpose of it to be consistent with (3.1) being a column vector of functions.

We don't need to keep the cumbersome expression in (3.2). Instead, we write it as a representation of (3.1) in the nodal basis $(\phi_n)_n$ with notation $[\cdot]_\phi$,

$$[\mathcal{R}u]_\phi = U \overset{\text{def.}}{=} \begin{bmatrix} u_{1,1} & \dots & u_{1,N} \\ \vdots & & \vdots \\ u_{D,1} & \dots & u_{D,N} \end{bmatrix}, \tag{3.3}$$

where u_{ij} is the j'th coefficient of $u_i(x)$ expansion in the nodal basis.

The lowering operator $\mathcal{Q}$ is treated similarly as a fully-connected neural network.

Remark: We follow the claim that the input $(u(x), x)$ yields better results than simply $u(x)$ [7] (p. 12, "Preprocessing" paragraph). This might not be significant to our implementation however, as we work with with periodic boundary conditions (the solution operator is translation invariant), but we aimed for consistency with current literature.

3.1.2 Inner layer operator

The central part of the neural operator is the inner-layer operator $\mathcal{L}_l$, it represents the transition between one inner layer l to the next one, $l+1$. For this we introduce a second set of functions, the projection functions, $\{\psi_m(x)\}_{m=1}^M \subset V$, with $M \leq N$. These functions act as the tools we approximate our solution with. Choosing the right projection functions for a specific problem can be a major step in optimizing a neural operator to a problem. In general, $\{\psi_m(x)\}_{m=1}^M$ can be any functions, but we implemented solely Fourier modes up to frequency M, interpolated to V. Now, we turn to formulate the inner-layer operator (2.4) with nodal basis coefficients.

Recall the definition of the inner-layer operator,

$$(\mathcal{L}_l \boldsymbol{u}_{l-1})(x) = \sigma\left(W_l \boldsymbol{u}_{l-1}(x) + \boldsymbol{b}_l + \sum_{m=1}^{M} \langle T_{l,m} \boldsymbol{u}_{l-1}(x), \psi_m(x) \rangle_{L^2} \psi_m(x) \right). \tag{3.4}$$

For $1 \leq l \leq L$, the inner-layer operator $\mathcal{L}_l$ gets the input $\boldsymbol{u}_{l-1}(x)$ and updates it. In coefficient form, $\boldsymbol{u}_{l-1}(x)$ is a $D \times N$ matrix U_{l-1} as in (3.3). Suppressing l, the first two terms become,

$$[W\boldsymbol{u} + \boldsymbol{b}]_\phi = WU + B\,, \tag{3.5}$$

where W, B are real matrices of sizes $D \times D$, $D \times N$ respectively. B is a length D vector broadcasted to a corresponding matrix (so we avoid N learnable parameters as desired).

The third (and most complicated) term in (3.4) can be expressed in the finite element space using pre-calculated inner products. When implementing, we pre-calculate the inner products between the nodal basis and the projection function, then save it as a matrix. That way, we reduce the network's computations from integrals (or FFT) to rudimentary linear algebra. To that end, we define the coefficient matrix as

$$C_{mn} \overset{\text{def.}}{=} \langle \phi_n(x), \psi_m(x) \rangle_{L^2}\,, \tag{3.6}$$

and represent the projection functions $(\psi_m(x))_m \subset V$ using coefficients,

$$\psi_m(x) = \sum_{n=1}^{N} \Psi_{mn}\phi_n(x), \quad \Psi \overset{\text{def.}}{=} [(\psi_m(x))_m]_\phi \,. \tag{3.7}$$

Both C and Ψ are matrices of dimension $M \times N$, and in general, $C_{mn} = \langle \phi_n, \psi_m \rangle_{L^2} \neq \Psi_{mn}$ (we didn't define V to be an inner-product space, let alone with L^2 inner-product).

The learnable parameters $T_{l,m}$ in (3.4) are $D \times D$ real matrices. Suppressing l again, we denote their elements by $t_{ij}^{(m)}$ and calculate for each channel (row) j,

$$(T_m\boldsymbol{u}(x))_j = \sum_{d=1}^{D} t_{jd}^{(m)} u_d(x) = \sum_{d=1}^{D} t_{jd}^{(m)} \left(\sum_{n=1}^{N} u_{dn}\phi_n(x) \right) = \sum_{n=1}^{N} \left(\sum_{d=1}^{D} t_{jd}^{(m)} u_{dn} \right) \phi_n(x) \,. \tag{3.8}$$

Plugging this in the third term of (3.4) results in a formula for the channel j function,

$$\begin{aligned} \left(\sum_{m=1}^{M} \langle T_m\boldsymbol{u}(x), \psi_m(x)\rangle \psi_m(x) \right)_j &= \sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{d=1}^{D} t_{jd}^{(m)} u_{dn} \langle \phi_n(x), \psi_m(x)\rangle \psi_m(x) \\ &= \sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{d=1}^{D} t_{jd}^{(m)} u_{dn} C_{mn} \psi_m(x) \,. \end{aligned} \tag{3.9}$$

We can replace the summations over n and d with matrix multiplication,

$$\begin{aligned} &\sum_{n=1}^{N} u_{dn} C_{mn} = (UC^T)_{dm} \\ &\sum_{d=1}^{D} t_{jd}^{(m)} (UC^T)_{dm} = (T_m U C^T)_{jm} \,. \end{aligned} \tag{3.10}$$

Then (3.9) becomes,

$$\sum_{m=1}^{M} (T_m U C^T)_{jm} \psi_m(x) \,, \tag{3.11}$$

or in coefficient form, the n'th coefficient of the j'th channel is,

$$S_{jn} \overset{\text{def.}}{=} \sum_{m=1}^{M} (T_m U C^T)_{jm} \Psi_{mn} \,. \tag{3.12}$$

Finally, plugging the $D \times N$ matrix S and (3.5) into the inner-layer operator (3.4) yields

$$[\mathcal{L}\boldsymbol{u}(x)]_\phi = \sigma\left(WU + B + S \right). \tag{3.13}$$

This expression is computationally efficient, as C^T and Ψ in (3.12) are independent of the layer l or the initial input, so they can be computed once before the training or evaluation phases. Moreover, when implementing using Fourier projection, equation (3.12) captures both the learnable parameters and Fourier series expansion in a simple Numpy `einsum()`.

Note that the number of learnable parameters in a single layer is dominated by the matrices $\{T_m\}_{m=1}^{M}$, which has an order of $\mathcal{O}(MD^2)$.

3.2 Error sources

In recent literature, two sources of error are considered — the approximation error and discretization error, as introduced in (2.5). However, we aim to be more descriptive with the sources of error, to better describe and distinguish phenomena witnessed in later sections.

Given an operator $\mathcal{G} : \mathcal{A} \to \mathcal{U}$, and a finite element space $V_N \subset \mathcal{A}$ with N DOF, we denote by $\hat{\mathcal{N}}_{\theta(S)} : \mathbb{R}^N \to \mathcal{U}$ the neural operator implemented on a computer after training. The $\hat{\cdot}$ represents finite element discretization (in the sense of Section 3.1), $\theta(S) \in \mathbb{R}^p$ represents the learnable parameters of the network post-optimization, and $S \in \mathbb{N}$ is the number of training samples. θ is dependent on the loss function that is minimized, which is in itself dependent on the amount of samples S chosen from the distribution μ on $\mathcal{A}$. Therefore θ is dependent on S, and we denote this relation by $\theta(S)$.

We isolate 4 main components of the error, ordered by abstraction,

1. **Approximation error** – error between the operator $\mathcal{G}$ and the approximating, infinite-dimensional neural operator $\mathcal{N}_{\theta^*}$. Theorem 2.2 promises that for every $\varepsilon > 0$, there exists $\mathcal{N}_{\theta^*}$ for some particular choice of $\theta^* \in \mathbb{R}^p$, such that $\|\mathcal{G}a - \mathcal{N}_{\theta^*}a\|_{\mathcal{U}} < \varepsilon$. Furthermore, we can control the error with the number of channels of the neural operator D.

2. **Discretization error** – error between the infinite-dimensional neural operator $\mathcal{N}_{\theta^*}$ and its discretization $\hat{\mathcal{N}}_{\theta^*}$ defined on the finite element space V_N. Rigorous definition requires introduction of variational forms of PDEs, and Galerkin approximations, and is not given here (see [16]). However, for the PDEs considered below, this error is controlled by the dimension of the FES N.

3. **Sampling error** – error between the minimizer of the true risk $\hat{\mathcal{N}}_{\theta^*}$ and minimizer of the empirical risk $\hat{\mathcal{N}}_{\theta^*(S)}$ that is dependent on the finite sample size S (see Section 2.1). As the empirical risk (a computable function) converges to the true risk (theoretic function) with $S \to \infty$, this error is controlled by S.

4. **Optimization error** – error between the minimizer of the empirical risk $\hat{\mathcal{N}}_{\theta^*(S)}$ and the parameters found using optimization $\hat{\mathcal{N}}_{\theta(S)}$. This error arises because the optimization algorithm isn't guaranteed to find the minimizer of the empirical risk (2.2) in finite time, due to the loss function being non-convex.

Combining these errors, we get

$$\begin{aligned} \|\mathcal{G}a - \hat{\mathcal{N}}_{\theta(S)}a\|_{\mathcal{U}} \leq &\underbrace{\|\mathcal{G}a - \mathcal{N}_{\theta^*}a\|_{\mathcal{U}}}_{\text{approximation error}} + \underbrace{\|\mathcal{N}_{\theta^*}a - \hat{\mathcal{N}}_{\theta^*}a\|_{\mathcal{U}}}_{\text{discretization error}} \\ &+ \underbrace{\|\hat{\mathcal{N}}_{\theta^*}a - \hat{\mathcal{N}}_{\theta^*(S)}a\|_{\mathcal{U}}}_{\text{sampling error}} + \underbrace{\|\hat{\mathcal{N}}_{\theta^*(S)}a - \hat{\mathcal{N}}_{\theta(S)}a\|_{\mathcal{U}}}_{\text{optimization error}} . \end{aligned} \tag{3.14}$$

3.3 Loss function

In this section, we briefly define a loss function that will be minimized in the optimization process (the “learning”), and show its interpretation as norms in CGk spaces. As PyTorch's automatic differentiation is needed for optimization, our goal is to approximate the relevant norms with the information available to PyTorch — coefficients. While more accurate approximations to the norms are possible through a Firedrake-PyTorch interface, we decided to avoid it due to time constraints.

Let $\{\boldsymbol{a}^{(i)}, \boldsymbol{u}^{(i)}\}_{i=1}^{S}$ be S pairs of initial-conditions drawn from some distribution and corresponding solutions, all represented as vectors of coefficients in the nodal basis $\{\phi_n(x)\}_{n=1}^{N}$ of a FES V. Let $\hat{\mathcal{N}}$ be a discretized neural operator, and denote by $\hat{\boldsymbol{u}}^{(i)}$ the prediction of the trained network on the input, i.e. $\hat{\boldsymbol{u}}^{(i)} = \hat{\mathcal{N}}\boldsymbol{a}^{(i)}$. Throughout the implementation, we use the relative mean loss defined by,

$$\text{RelMean} \overset{\text{def.}}{=} \frac{1}{S}\sum_{i=1}^{S} \frac{\|\boldsymbol{u}^{(i)} - \hat{\boldsymbol{u}}^{(i)}\|_{\ell^2}}{\|\boldsymbol{u}^{(i)}\|_{\ell^2}} . \tag{3.15}$$

where $\|\cdot\|_{\ell^2}$ is the Euclidean norm for N dimensional vectors. This loss approximates what we call the “RelL2” loss in both CG1 and CG3 FES (on 1D intervals). To see why is that, suppose V is a CGk space as in Definition 2.7. The nodal variables are point-evaluations at kN equidistant points, among the N cells. Denote length between two consecutive points h and two functions $u, \hat{u} \in V$. The RelMean loss between the corresponding coefficient vectors $\boldsymbol{u}, \hat{\boldsymbol{u}}$ becomes a quotient of Riemann sums,

$$\frac{\|\boldsymbol{u} - \hat{\boldsymbol{u}}\|_{\ell^2}}{\|\boldsymbol{u}\|_{\ell^2}} = \frac{\sqrt{h\sum_{n=1}^{N}(u(x_n) - \hat{u}(x_n))^2}}{\sqrt{h\sum_{n=1}^{N} u^2(x_n)}} \xrightarrow{N\to\infty} \frac{\sqrt{\int_a^b (u(x) - \hat{u}(x))^2 dx}}{\sqrt{\int_a^b u^2(x) dx}} = \frac{\|u - \hat{u}\|_{L^2}}{\|u\|_{L^2}} , \tag{3.16}$$

where we term the mean of the RHS across S samples as the “RelL2” loss.

We conclude by noting that we also experimented with ℓ^1/L^1 loss, but found it empirically inferior to the $p = 2$ case. This is possibly due to the absolute value in L^1 norms making it non-differentiable.

3.4 Numerical verification

Before analyzing how neural operators approximate PDE's, we first make sure that the math above is translated correctly to code. Hence, we examine individual parts of the code with simple tests. All of the figures, tables, code, and data in this report are original work and described in detail in Appendix A.2.

3.4.1 Fourier series

We'd like to test the derivation of the term S in (3.12), which accounts for the projection of the input function to a different function basis, e.g. Fourier series expansion. We separate the part in the code implementing an inner-layer (equivalently a neural operator with $L = 1$ inner-layers and no lifting/lowering layers) and set the activation function to identity $\sigma(x) = I(x) = x$, set the weight matrices to identity matrices of same shape, $\forall m,\ T_m = I_{D\times D}$, and ignore the linear transformation part $W = B = 0$. Combined, these reduce a neural operator to a single inner-layer operator (2.4), which is just a standard change of basis,

$$(\mathcal{N}\boldsymbol{u})(x) = (\boldsymbol{\mathcal{L}}\boldsymbol{u})(x) = \sum_{m=1}^{M} \langle \boldsymbol{u}(x), \psi_m(x)\rangle_{L^2} \psi_m(x)\,. \tag{3.17}$$

Taking $\psi_m(x)$ to be *normalized* Fourier modes, we get a standard Fourier series expansion. Implementation is described in Section A.1.1.

The verification is implemented in file `verification.py`, as a class called `DummyNeuralOperator` that inherits from the inner-layer class `NeuralOperatorLayer`. The inheritance makes sure that `DummyNeuralOperator` follows the exact same calculations as all of the other networks in this report (following Section 3.1), but with redundant weights, biases, and activation. Note that `DummyNeuralOperator` is restricted only to this subsubsection, all other parts of the report use the full definition and implementation of the neural operator, with nontrivial choices for $\sigma(x), T_m$, and W.

Results of this verification are presented in Figure 1, where we deliberately chose a function featuring Gibbs phenomena. As expected, the output functions from the `DummyNeuralOperator` are a Fourier series expansion of the input truncated at M. As M increases, the approximation becomes better.

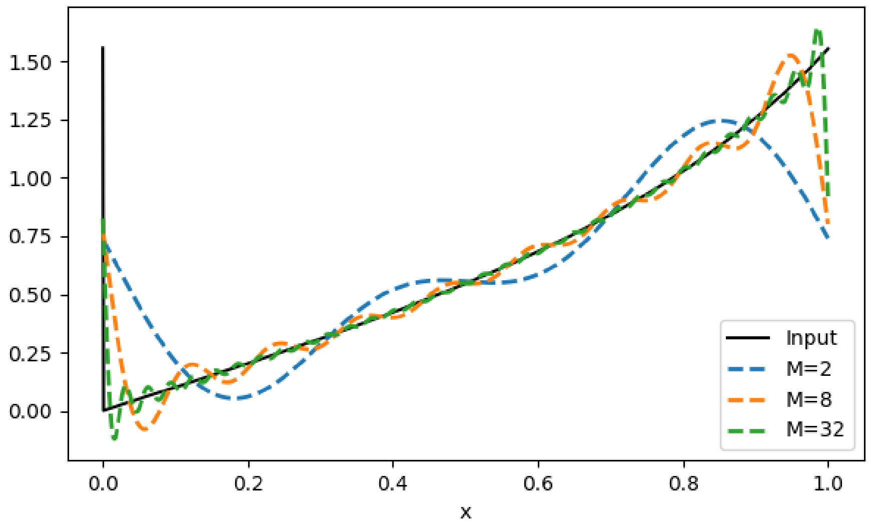


Figure 1. *Input function* $\tan(x)$ *in black and its Fourier basis approximations for different number of Fourier modes* $M = 2, 8, 32$ *(blue, orange, green resp.). Created using DummyNeuralOperator as described in equation* (3.17).

This ensures that the long calculation of (3.13) and its implementation are correct. The verification acts as a "green-light" to proceed with the analysis and from now on add nontrivial weights $W \neq 0, T_m \neq I_{D\times D}$, biases $B \neq 0$, and activation function $\sigma(x) \neq I(x)$.

3.4.2 Loss calculation

We want to affirm that our implementation of loss functions which is based on sums of coefficients is correct. Table 1 shows the difference between test loss values computed explicitly using (3.15) and the same expression calculated using Firedrake's `errornorm` and `norm` functions with the L^2 configuration. The models are exactly the same trained models used as in Section 3.4.3.

N	64	128	256	512	1024	2048	4096
Burgers	1.7×10^{-5}	9.7×10^{-6}	2.4×10^{-6}	2.2×10^{-7}	9.1×10^{-7}	8.7×10^{-8}	7.0×10^{-8}
KS	2.5×10^{-4}	3.9×10^{-5}	5.0×10^{-6}	1.1×10^{-6}	1.5×10^{-5}	3.6×10^{-6}	—

Table 1. *Difference between loss calculated with Firedrake and loss calculated using (3.15) for Burgers and KS. Computed on relevant testset predictions, with constant model architecture* $D = 64, M = 16, L = 4$.

Note that KS isn't monotonically decreasing, possibly due to the KS equation producing functions with big fluctuations and oscillations, making Riemann sums on different partitions less predictable. Nonetheless, increasing N refines the Riemann sums, and decreases the discrepancy between our loss functions and Firedrake results. Therefore the implementation of loss function is valid for further use.

3.4.3 Discretization invariance

The original neural operator architecture boasts a result termed "discretization invariance" where the RelL2 loss is similar across different mesh sizes [7] (Table 3 or Figure 8) — little variation in the test loss is shown between identical networks trained on different mesh sizes. It's claimed to reflect independence of the error to the resolution, which complies to Theorem 2.3, and that the finite-dimensional implementation of the neural operator has learned the infinite-dimensional one. We claim that this result shows that the discretization error is smaller than the other sources of error (Section 3.2) and no more. Nevertheless, it is an important property of the architecture in the literature, and we expect to reproduce said property.

Table 2 shows test loss values of networks with identical architectures (recall that the architecture is independent of N), trained on datasets with different mesh resolutions N (further described later in Section 4.1). There is no apparent trend relating the loss values to the mesh size N. Differences between the values can be attributed to the optimization procedure — the loss function is non-convex so the ADAM algorithm (which is momentum stochastic gradient descent) will probably not converge to a unique global minimum of the complex loss landscape.

N	64	128	256	512	1024	2048	4096	8192
RelL2 Loss	0.0019	0.0017	0.0013	0.0018	0.0016	0.0015	0.0014	0.0016

Table 2. *Test loss values for Burgers equation calculated as described in subsection 3.3, for increasing mesh size on identical (and N-restricted) test dataset. Constant network architecture with* $D = 64, , M = 16, L = 4$, *and 557,569 learnable parameters.*

To conclude, Table 2 shows at the very least that the discretization error is smaller than the other errors, which is in agreement with results shown in recent literature [7] [6].

4 Burgers equation

In this section, we use the methods outlined in Section 3 to construct and implement a neural operator that will learn the 1D Burgers equation.

We consider the following 1D PDE,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0. \tag{4.1}$$

on a periodic spatial interval and positive time $x \in \Omega = [0, 1], t \in \mathbb{R}^+$. In this work, whenever we mention Burgers' equation, we refer to this problem.

4.1 Data generation

We used the finite element software Firedrake [9] to sample 1,200 random functions on a discontinuous Galerkin (DG) finite element space built on a mesh with $N = 8192$ equispaced cells over Ω. Then, we used a Matern field method to smoothen out the functions and transfer them into a continuous Galerkin (CG1) space. Burgers equation with $\nu = 0.01, T = 1sec, dt = 1/10N$ was solved for each such initial function, so we ended up with a dataset of 1,200 pairs of initial-conditions and solutions. Finally, we interpolated this high resolution dataset into a few CG1 spaces on coarse meshes, with the number of cells being $N = 64, 128, 256, 512, 1024, 2048, 4096$.

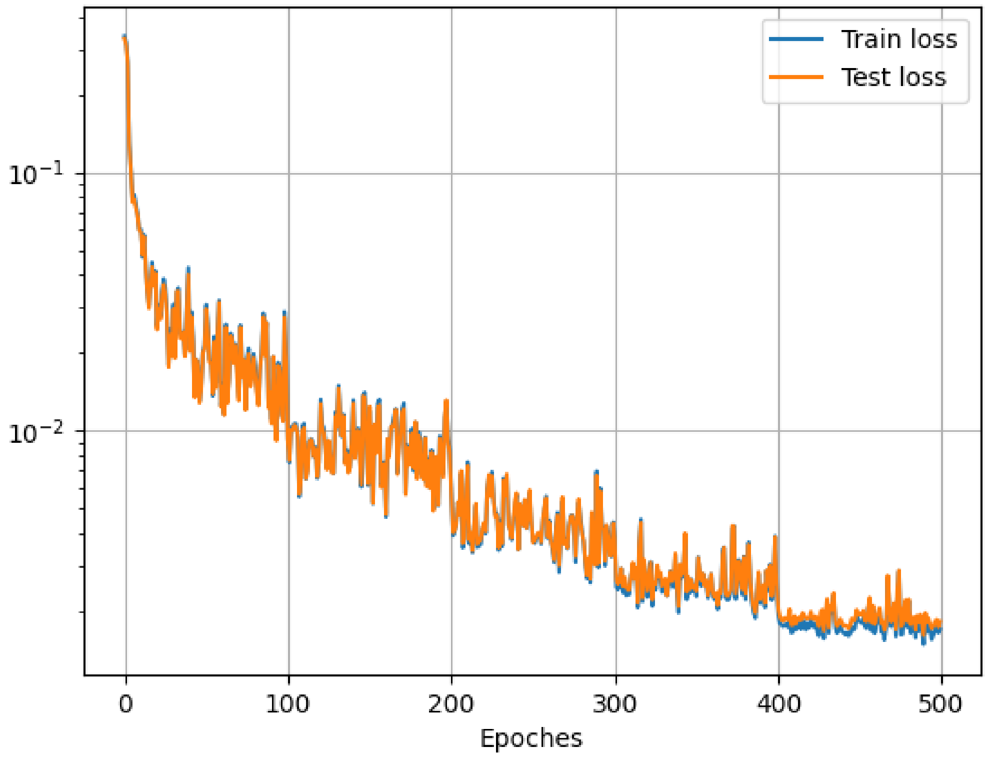
In order to train networks in PyTorch, we extracted the coefficients of both the initial-conditions and solutions in the nodal basis $(\phi_n)_{n=1}^N$ from Firedrake and saved them as PyTorch tensors (as this is a CG1 space, the coefficients are simply point-evaluations at cell edges). Arbitrarily we split the data into 1000 training samples, and 200 testing samples.

The Fourier coefficients matrix C_{mn}, and the Fourier modes functions, were computed for each desired pair M, N, as described in Section A.1.1. Full description of the code can be found in Appendix A.2.

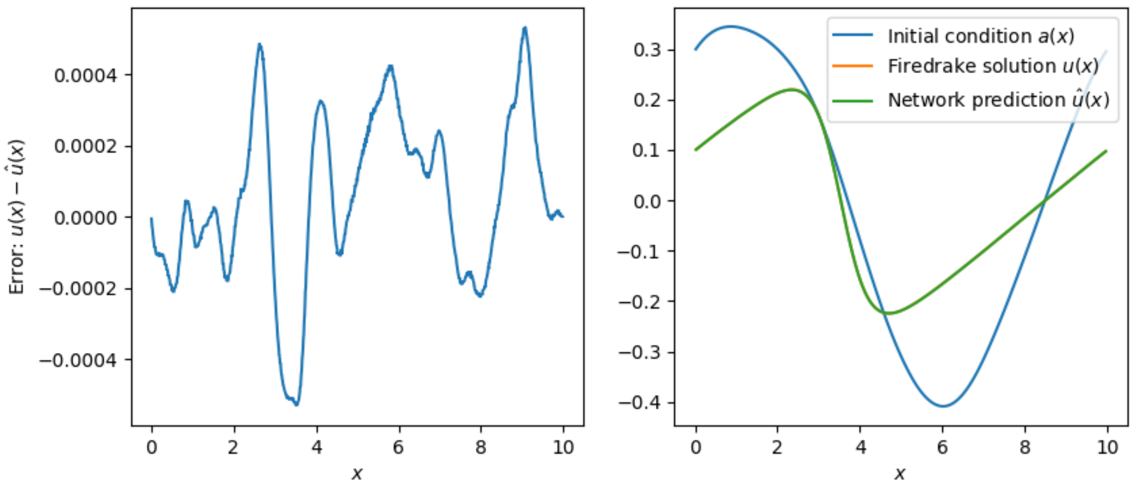
4.2 Training

Using the database described in the previous section, we trained DNNs that follow the neural operator architecture above to learn Burgers' equation. Each network's architecture is determined by M, D, L (which sets the number of learnable parameters), and it can be trained on a dataset of any mesh size N. Unless otherwise specified, the networks in this section were trained with Fourier projection basis $(\psi_m)_{m=1}^M$, for 500 epochs, with ADAM optimization, RelL2 loss (Section 3.3), and learning rate of 0.001 that is halved every 100 epochs. In subsequent figures and text, the test-loss or RelL2 loss is the Relative L2 loss as in (3.15), calculated on the relevant $S = 200$ testing samples that the network wasn't trained on.

An example of a typical network's loss-epoch curve during training, and input-output example are given in Figure 2. Note that in (b) Right the network's prediction to the input coincides with the finite element computed solution.



(a) Loss values during training



(b) Example of input-output, and prediction to the network from (a).

*Figure 2. **(a)** Train and test loss values at each epoch of a network trained on Burgers data, log scale. **(b) Right** An initial input function from the test dataset $a(x)$ (blue), its corresponding Burgers solution computed by Firedrake $u(x)$ (orange), and the prediction of the trained network from (a) $\hat{u}(x)$(green). At this zoomed-out resolution, the solution and prediction coincide. **(b) Left** Error between solution and prediction $u(x) - \hat{u}(x)$ The network has 164,353 learnable parameters, number of modes $M = 4$, number of layers $L = 4$, and number of channels $D = 64$. Dataset grid size is $N = 1024$.*

4.3 Channel analysis

Theorem 2.2 says, in essence, that the approximation error converges to zero as $D \to \infty$ even with constant projection functions. Theoretically, this is a strong result, but as mentioned in the literature review (Section 2.2.3), it is unpractical as the convergence rate is unknown and the discrete neural operator $\hat{\mathcal{N}}$ isn't guaranteed to converge to the infinite-dimensional neural operator $\mathcal{N}$ with finite datasets and SGD-type optimizations. Nevertheless, we expect to witness the effect empirically, as our numerical model is a proxy to the theoretical one.

In Figure 3, each point represents the RelL2 loss of a different network post-training. The x-axis represent how many channels D the network has, and different curves represent different number of Fourier modes M. The figure captures multiple interesting phenomena. First, for all M, the loss curves decrease with D until reaching a plateau. Second, the constant function $M = 0$ is clearly insufficient for learning the operator, as it cannot be used to reach errors of less than 1%, even with tens of thousands learnable parameters at $D = 120$. Finally, for $M = 4$ and above, there are no meaningful differences between loss curves. This might mean that for our dataset — which is determined by our specific distribution of initial conditions and specific parameters for solving Burgers' equation — only 4 Fourier modes are enough to learn the principal part of the information, and 2 Fourier modes are definitely not enough. We test this hypothesis in the next section.

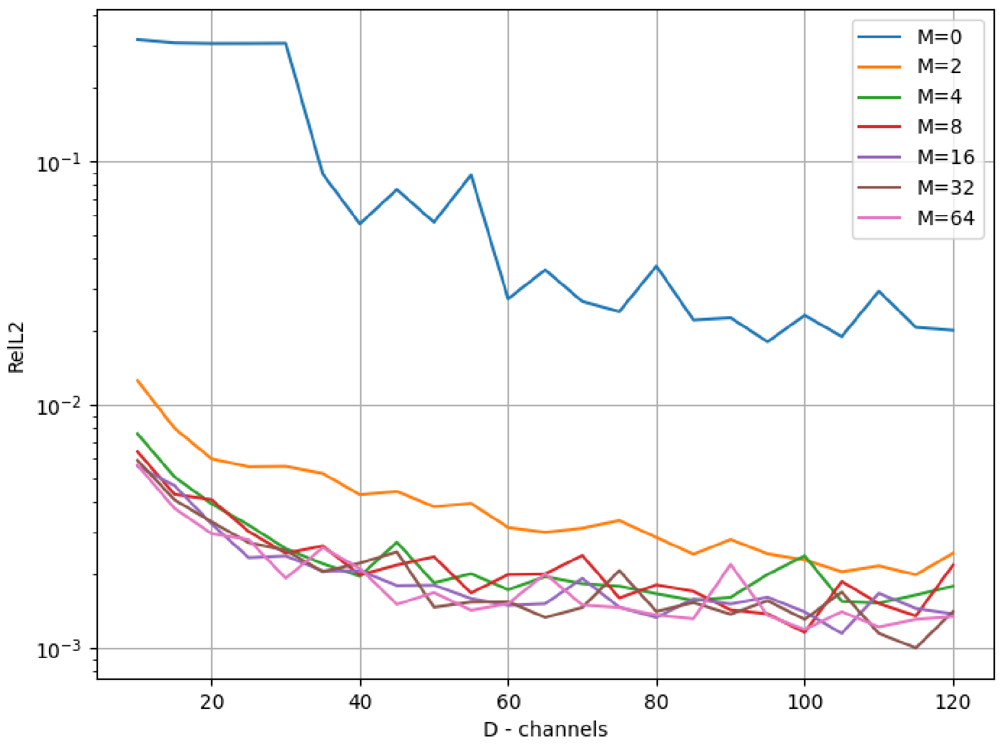


Figure 3. Every curve represents the test loss values for networks with a different number of Fourier modes M, and an increasing number of channels D. Constant resolution $N = 4096$, constant number of layers $L = 4$, trained on 1000 training samples. Log scaled.

Figure 4 shows a similar experiment — each point represents the RelL2 loss of a post-training network, but now $M = 16 =$ const and the difference between curves is the amount of training samples S the network was trained on. The motivation for this experiment is to isolate the sampling error discussed earlier (see equation (3.14)), and determine if its the cause for the plateau witnessed earlier. Evidently, increasing the amount of training samples reduces the loss values. All other variables between the networks are constant, therefore the sampling error is the predominant error in the plateau.

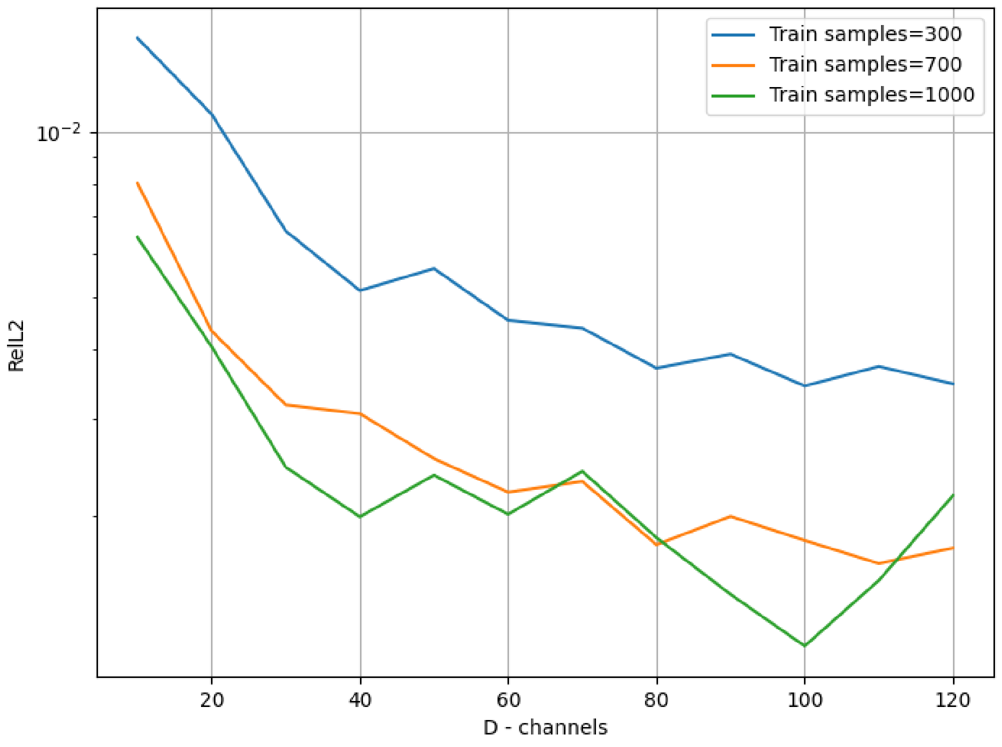


Figure 4. *Test loss values for networks with same architecture $M = 16, L = 4$, and increasing number of channels D. Each curve represents networks trained with a different amount of samples S. Constant resolution $N = 4096$. Log scaled.*

The experiments above can provide insight on the best choices of hyperparameters for particular use cases. Recall that the amount of learnable parameters in each inner-layer of the neural operators (in 1D) is dominated by $\mathcal{O}(D^2M)$. If, for example, the top priority in some application is accuracy and an error tolerance of 0.1% is desired, then based on Figure 3 a network with $M = 32, D = 115$ should be chosen, although it has $32 \times 115^2 \approx 420$k learnable parameters in each layer, which is computationally heavy. On the other hand, if the desired tolerance is 2%, then a network with $M = 4, D = 40$ would be adequate, which has merely $6 \times 40^2 \approx 6.4$k learnable parameters per layer. Similar considerations can be applied to the amount of training samples.

4.4 Fourier mode analysis

One of the interests that lead this work is the question of how to choose an optimal neural operator architecture for a given problem, i.e. minimal loss value for a minimal number of network parameters. The case of D has been discussed in the previous section, and now we turn our attention to the Fourier modes M.

4.4.1 Multilayered networks

Figure 5 exhibits the test loss of networks trained with constant number of channels and layers, $D = 64, L = 4$, constant resolution $N = 4096$ and training samples $S = 1000$, but varying number of Fourier modes indicated by the x-axis. For each M, we had to compute a corresponding coefficient matrix C_{mn} and interpolated Fourier modes Ψ. The purpose of this experiment is to study how the number of Fourier modes M affects the loss.

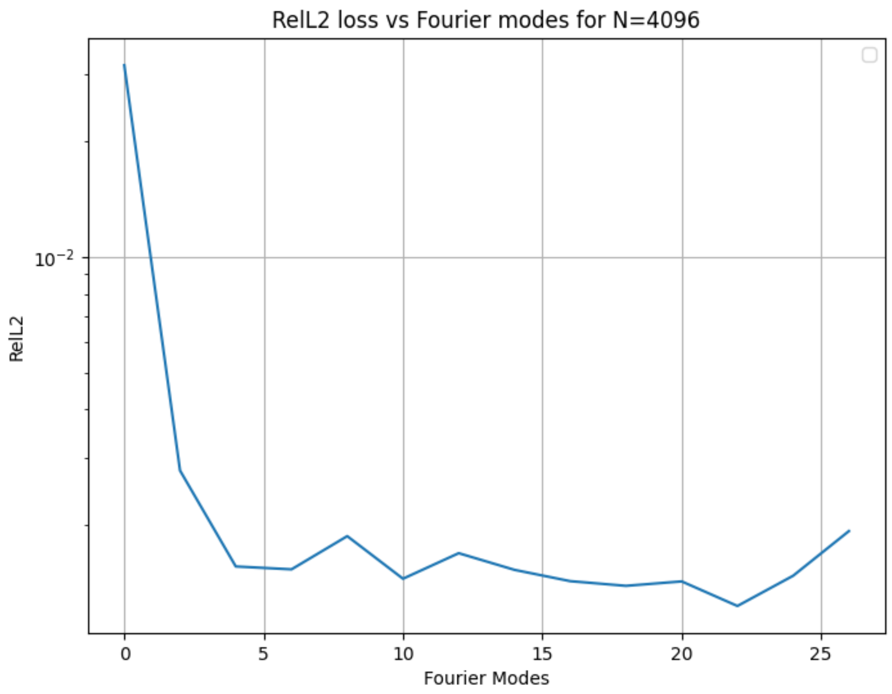


Figure 5. *Test loss values on a $N = 4096$ dataset, for networks trained with constant number of channels $D = 64$, layers $L = 4$, and varying number of Fourier modes M. Log scaled.*

Two results that are aligned with the preceding section can be seen in the figure. First, the loss values reach a plateau around $M = 4$, meaning that from this point onwards, adding more Fourier modes and increasing the network's size does not help with learning. Second, at $M = 0$, which corresponds to one constant function, the loss value is considerably high. This strengthens the ethos of the FNO framework — the network's ability to learn an operator is enhanced by incorporating information from a different function basis (specifically with some non-local operation like integration).

4.4.2 One layer network

The previous section had hinted that neural operators do not need many Fourier modes to learn the dataset — a plateau was reached for $M > 4$. However, in this section we show that this might not be true for all hyperparameter choices, and is surely not the case for $L = 1$. Furthermore, we demonstrate a relation between the effective number of Fourier modes in a network M, and the frequencies most prevalent in the dataset the network was trained on.

Figure 6 Top shows an experiment identical to the one above but instead of a multilayered network with $L = 4$, we used a single inner-layer $L = 1$. Unsurprisingly, the loss values are higher, due to the number of parameters being 4 times smaller and the absence of processing layers. The important part though, is that the plateau in this case starts later, around $M = 25$. In other words, the loss values of single layer networks have a stronger dependence on M than the multilayered ones.

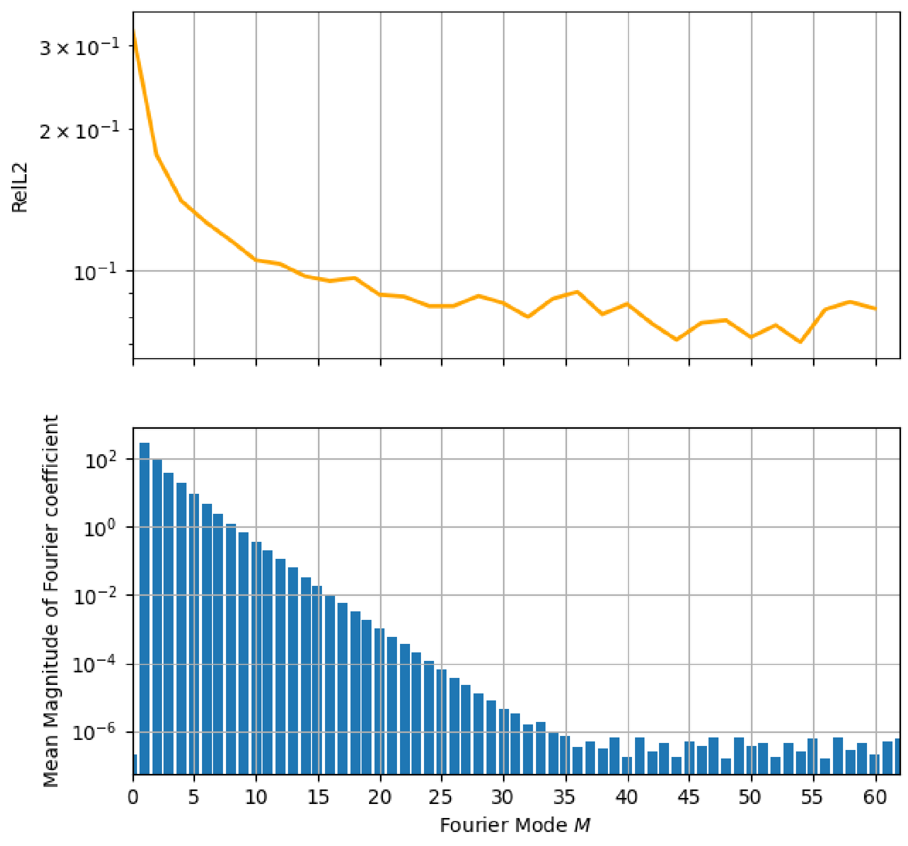


Figure 6. ***Top*** *— Test loss values for single layered networks trained with constant number of channels $D = 64$, layers $L = 1$, and varying Fourier modes M. Resolution of dataset $N = 4096$. Log scaled.*
Bottom *— Mean of Fourier coefficients' magnitude across the output functions of the $N = 4096$ dataset. Cutoff at $M = 60$ (out of $N/2$), log scaled.*

In Figure 6 Bottom, we analysed the Fourier modes of the dataset the network from Top was trained on. We took the discrete Fourier transform of every output function in the dataset and averaged the magnitude of Fourier coefficients over all functions in the dataset. If we denote by

$\{a^{(i)}(x), u^{(i)}(x)\}_{i=1}^{S}$ the input-output pairs in the dataset, and with $\{\hat{a}^{(i)}(M), \hat{u}^{(i)}(M)\}_{i=1}^{S}$ the coefficient of their Fourier series expansion (discrete in M), then each bar M in Figure 6Bottom represents

$$\frac{1}{S}\sum_{i=1}^{S} |\hat{u}^{(i)}(M)| \,. \tag{4.2}$$

The resulting graph shows us which Fourier modes are most prevalent amongst the dataset output functions. We see that the dataset is characterized by low Fourier modes, which is in agreement to the outputs being low-frequency functions (smooth and without rapid changes).

Comparison of Figure 6 Top and Bottom clarifies why the plateau in RelL2 loss occurs around $M = 25$. As can be seen, the mean Fourier magnitudes decay exponentially. So above a certain point around $M = 25$, the information carried by Fourier modes is insubstantial in the dataset, therefore the networks' ability to predict outputs doesn't improve by adding the higher, insignificant, Fourier modes.

4.4.3 Weight contribution

The correlation between the dominating frequencies in the dataset to the networks' loss plateau does not explain, however, why a multilayered network can use merely $M = 4$ Fourier modes to achieve loss values that are an order-of-magnitude better than a single layered, $M = 25$ network. One possible explanation for this phenomena is that the single layered network, which has less processing units, has to take out the most of the data at its disposal. In our example, the single layer network will give more weight to the high Fourier modes than the multilayered network, which "ignores" the insignificant high Fourier modes.

To test this hypothesis, we compared the post-training weights corresponding to different Fourier modes T_m, $m = 1, \dots, 2M+1$ of both networks (note that for each $M \geq 1$ there are both sines and cosines which overall makes $2M+1$ functions indexed by m). To measure quantitatively how big is the influence of each mode on the outcome, we summed each weight matrix in absolute value,

$$\forall m,\ T_m \,\text{Contribution} \overset{\text{def.}}{=} \sum_{i,j=1}^{D} |t_{i,j}^{(m)}|, \tag{4.3}$$

where $t_{i,j}^{(m)}$ are the learnable parameters in the matrix T_m.

The idea is, if the prediction of a network heavily relies on some Fourier mode m, than the Contribution of T_m would be large. Instead, if a network dismisses a certain Fourier mode, we expect the Contribution to be near zero. In multilayered networks we are only interested in the weights of the first layer $l = 1$, as this layer receives the data right after a preprocessing phase which is identical to the single-layer network.

Figure 7 compares the T_m Contribution for all Fourier modes m, between a single layered network and a multilayered one. Both networks show a similar contribution of the low Fourier modes $m = 0, \dots, 10$, which agrees with them being dominant in the dataset as shown above. However, the single layered network strongly utilizes higher frequency modes $m = 15, \dots, 52$ as opposed to the multilayered one.

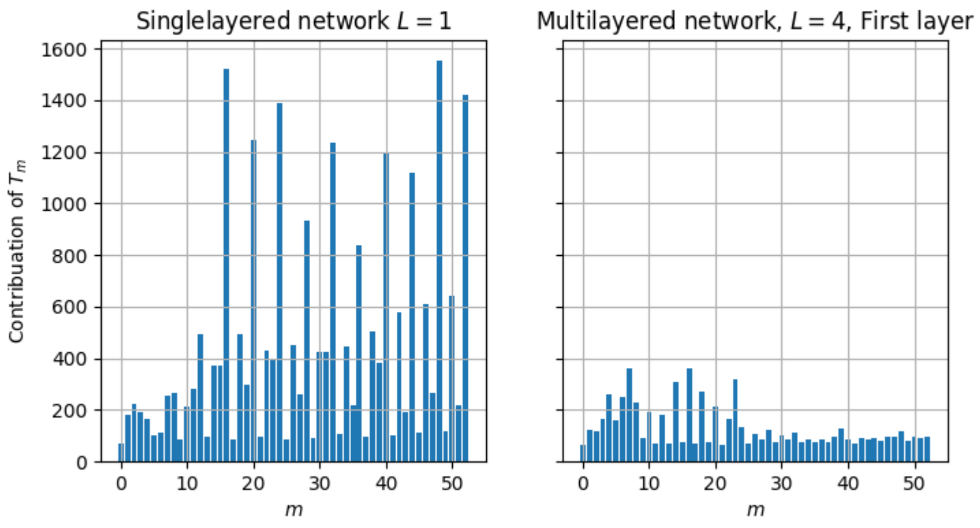


Figure 7. Comparing the T_m contribution as defined in (4.3) of Fourier modes between a single layer network (Left) and a 4-layered network (Right). Networks' architecture $D = 64, M = 26$ trained on $N = 4096$ dataset.

Inferring these results to practical matters, when one is trying to find optimal hyperparameters, compensating for a low number of Fourier modes M with more layers is a possibility. While existing research often emphasizes balancing D and M, our results indicate that the interplay between L and M should also be considered. Specifically, reducing the number of layers, which decreases the number of parameters by MD^2, might be offset by a corresponding increase in M. This highlights a new dimension of trade-offs in neural network design that could lead to more efficient architectures in practice

4.4.4 Remarks

We conclude this section with some intriguing points about how the neural operator uses the modes to learn the dataset.

First, note that in Figure 7 the weights of the 0–5 Fourier modes are low, although they dominant in the dataset (Figure 6 Bottom). A possible reason for that is that the linear part of the inner-layer operator ($WU + B$ in (3.13)) covers this part of the information, which reduces their importance in T_m.

Second, when measuring the T_m Contribution of a single layered network, some of the modes always come on top, even for networks with different M's. For example, modes $m = 16, 24, 32$ always have the highest Contribution, while modes $m = 0, 9, 6, 5$ the least. This raises a possible method of optimization (compression) for neural operator - rank the modes by their Contribution on a $L = 1$ network (e.g. Table B.1), then implement a multilayered network using only the $0 < \alpha < 1$ fraction of modes that contribute the most. This would reduce the number of learnable parameters to $\mathcal{O}(\alpha MD^2)$ per layer. Additionally, note that the highest modes in our example are always even m's, which correlate to $\cos(mx)$ functions. It emphasizes that not all Fourier modes are required for adequate learning, and that choices of appropriate projections functions can be made based on the dataset.

Finally, when looking at the values of the matrix T_m portrayed on a $D \times D$ grid, it is apparent that some channels carry more information than others, as some rows/columns show intricate patterns, while others are close to zero (see Figure B.1). An analysis of these patterns and their importance can lead to further optimization and compression of neural operators (e.g. throwing out the less useful channels), and is an interesting research direction.

4.5 Super-resolution

As the neural operator architecture and implementation is independent of the mesh size N, it enables us to train networks on low resolution grids and evaluate them on high resolution ones. This process is termed "super-resolution", and is desirable because it produces high resolution results at a low resolution computational cost. The super-resolution property of neural operators was mentioned in the literature [6] [7] (Figure 1 and Figure 9 respectively), but the connection between the error to the size of the high resolution mesh was never tested.

In Figure 8, each curve shows a neural network model that was trained on one resolution, and tested on higher resolutions. All curves start at similar loss values, this is probably due to the aforementioned discretization invariance. However, the difference between the curves is probably a result of the discretization error growing and overcoming the other errors. The plateau of the curves might be explained by the fact that the difference between Riemann sums of two resolutions decreases as their discretizations become finer (akin to Cauchy series).

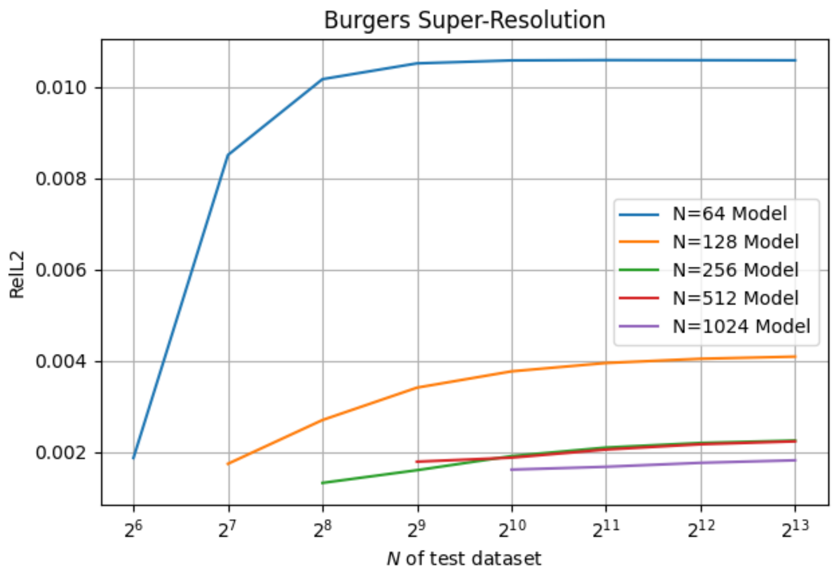


Figure 8. RelL2 test loss values for networks trained on meshes $N = 64, 128, 256, 512, 1024$, and tested on all of the datasets with higher resolutions (and on themselves). Constant network architecture $D = 64, M = 16, L = 4$.

Notice that at the $N = 2^{13} = 8192$ vertical line, the differences between the loss values decrease with the curves. Therefore, if one is looking to super-resolution a network to $N = 8192$, a cheap $N = 256$ network might serve better than the heavier $N = 1024$. This gives another optimization method for neural operator's architecture — train various low resolution models and choose a desired one based on their high resolution performance.

5 Kuramoto-Sivashinsky equation

In this section, we use the methods outlined in Section 3 to construct, implement, and analyze a neural operator that will learn the Kuramoto-Sivashinsky (KS) equation in its chaotic regime.

The Kuramoto-Sivashinsky equation can be written as follows,

$$\begin{aligned}&\frac{\partial u}{\partial t}+\frac{\partial^2 u}{\partial x^2}+u\frac{\partial u}{\partial x}+\nu\frac{\partial^4 u}{\partial x^4}=0,\\&x\in\Omega,\ t\in[0,T],\end{aligned}\tag{5.1}$$

where ν is a positive parameter that describes the viscosity in fluid dynamics applications, $\Omega = [0, 10]$ the domain with periodic boundary conditions, and the time horizon T will be varied throughout the analysis.

The KS equation captures a "fight" between its terms — the negative diffusion aggregates nearby function values, the nonlinear term creates wave steepening, and the hyperdiffusion smoothens out both of the former effects. Solutions to KS equation for different values of ν vary in their dynamics. For $\nu \geq 1$, the equation produces constant states (i.e. spatial functions without temporal evolution) because the hyperdiffusion outweighs the aggregating/steepening effects. Lower values of ν continuously shift this delicate balance, resulting in interesting dynamics. The $\nu \in [0.029756, 1)$ range exhibits many forms of nonlinear dynamics including bimodal, trimodal, and periodic attractors, and for $\nu \in [0.0252, 0.029755]$ we find chaotic behaviour [10]. As we're interested in analysing the ability of the neural operator to capture chaotic dynamics, we set $\nu = 0.029$ which is within the chaotic regime.

As the equation features a fourth-order derivative, the corresponding variational problem has a second-order derivative. Therefore, we cannot use a CG finite element space to define and solve it, as it is missing second derivatives. Instead, we used a Hermite finite element space (Definition 2.9), that ensures the existence of a weak second-order derivatives.

5.1 Data generation

Our objective is to "teach" a neural operator to predict the advancement of trajectories on the equation's chaotic attractor. Randomly sampled initial-conditions aren't guaranteed to be part of the attractor, so we can't use the method previously described for the Burger's dataset. Instead, we inserted a pure sinusoidal wave $u(x, t = 0) = \sin(2\pi x/10)$ as input and tracked an observable — specifically the L_2 norm — of the dynamics to find when the observable would stabilize, which would indicate that the solution has reached the attractor.

Figure 9 Top shows the L_2 norm of the solution over time. Initially there is an obvious growing trend, but it stabilizes after $T_{\text{transient}}$. Therefore, we sampled the spatial state in intervals of $T_{\text{transient}}$, to make sure the solution "forgot" the preceding captured sample, and saved all those as the initial-conditions. Then, we inserted each initial-condition to the equation, and ran it for T time units. The end result is a dataset of input-output pairs, where the inputs are sampled from the attractor and the outputs are T time units advancements of the inputs.

Following [10], we keep track of the error of our solution using the matter-conservation property of the KS equation. The spatial integral of a solution is a time-invariant quantity, and the spatial integral of the initial sine wave is $\int u(x, 0)dx = 0$. Therefore, for $t > 0$ any deviation from 0 in the value of the integral would represent a numerical error. Figure 9 Bottom shows that the spatial integral — the "error" — is consistently below 10^{-13} for $t > 0$.

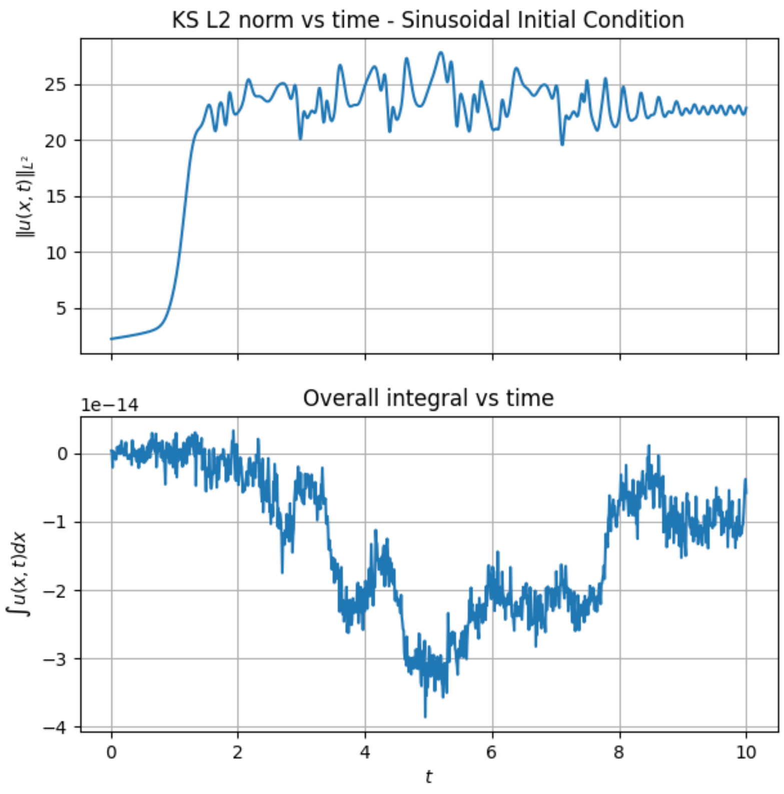
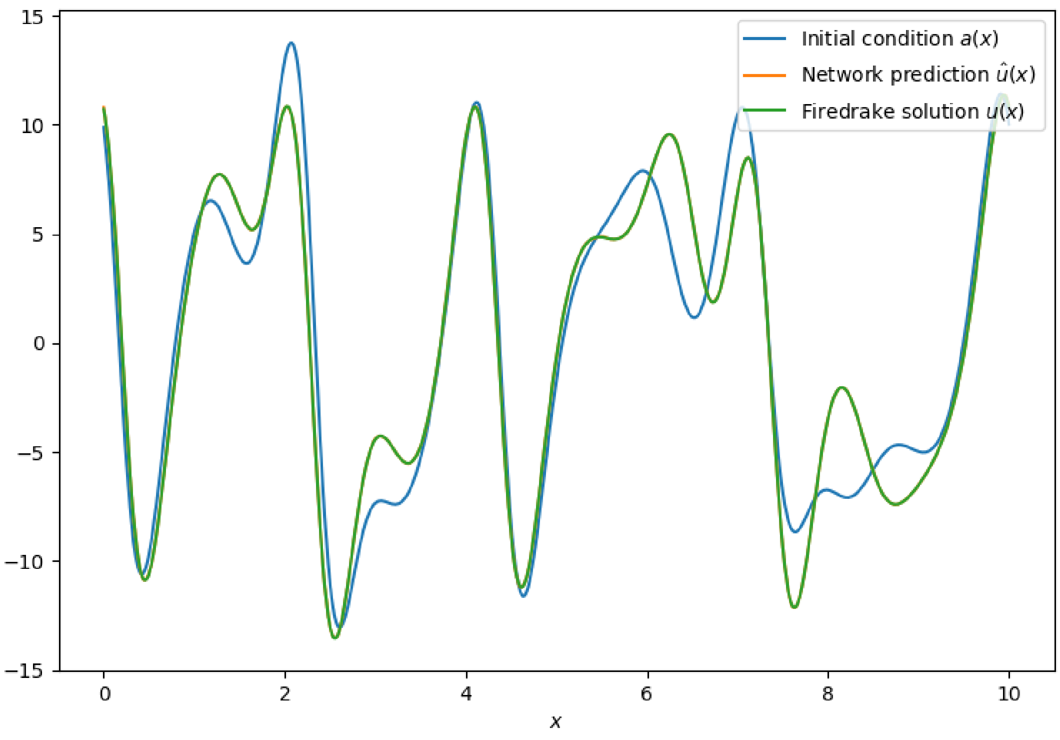


Figure 9. ***Top*** *– L2 norm of KS solution over time, showing stabilization from $T_{transient} = 2$ onwards.* ***Bottom*** *– Integral of the solution over time. The integral is conserved (conservation of matter) and $\int u(x, t = 0)dx = 0$, therefore the graph represents a numerical error.*

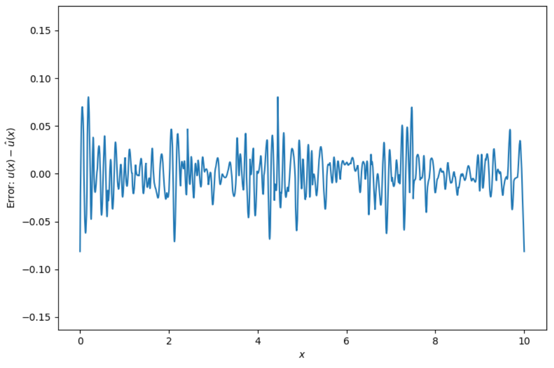
Using the procedure above, we created multiple datasets, one for each $T \in \{0.1, ..., 1.5\}$. We used Firedrake, with time unit $dt = 0.01$ and a Hermite finite element space constructed on a mesh with resolution $N = 4096$ on $\Omega = [0, 10]$. However, integration of Hermite FES with the neural operator framework we developed led to poor and inconsistent results (specifics in Appendix A.1.2). To counter this, we interpolated the $T \in \{0.1, ..., 1.5\}$ datasets from Hermite FES to a CG3 FES on the same mesh. Recall that Hermite is a subspace of CG3 (Section 2.3), then in theory it is a lossless transformation.

Finally, we interpolated the 15 datasets from a $N = 4096$ mesh to $N = 2048, 1024, 512, 256, 128, 64$ meshes, all on CG3 spaces. We also calculated the inner products C_{mn} and the projection function coefficients Ψ as described in Appendix A.1.1.

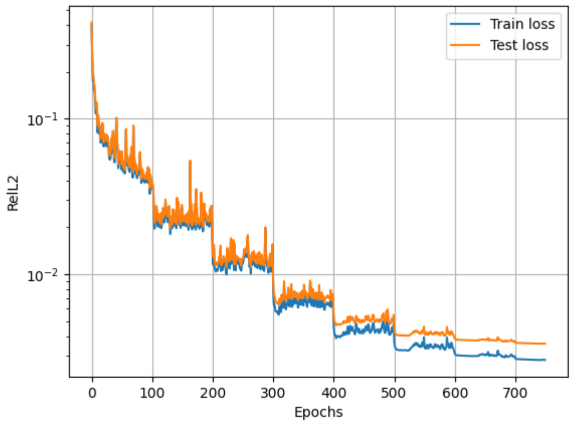
Figure 10 shows an example of a typical input-output pair from a $T = 1$ dataset, the prediction of a network, the error between output and prediction, and the loss graph of the network during training. Full description of the code can be found in Appendix A.2.



(a) Example of input-output, and prediction (coincides with output) for KS.



(b) Error between output and prediction of (a).



(c) Loss values of network while training.

Figure 10. ***(a)*** *— An initial input function from the test dataset $a(x)$ (blue), its corresponding KS solution computed by Firedrake $u(x)$ (green), and the prediction of a trained network $\hat{u}(x)$(orange). At this zoomed-out resolution, the solution and prediction coincide.* ***(b)*** *— Error between solution and prediction $u(x) - \hat{u}(x)$ from (a).* ***(c)*** *— Train and test loss values at each epoch of a network trained on KS data, log scale. The network has 557569 learnable parameters, number of modes $M = 16$, number of layers $L = 4$, and number of channels $D = 64$. Dataset grid size is $N = 128$, and time horizon $T = 1$.*

5.2 Time horizon comparison

We trained networks on the datasets described in the section above. Again, unless otherwise specified, the networks in this section were trained with Fourier projection basis $(\psi_m)_{m=1}^M$, for 750 epochs, with ADAM optimization, RelL2 loss, and learning rate of 0.001 that is halved every 100 epochs.

Figure 11 Left shows the measured RelL2 loss for networks that were trained on each of the different T datasets, for different mesh sizes. A few observations can be made. First, the networks' there is no one curve that is clearly above/below the others. This represents that the discretization error is smaller than the other errors, and is consistent with the above theory and results (i.e. discretization invariance). Second, in order to coherently see a trend, we look at the average on Figure 11 Right. The loss values increase with T, meaning that the ability of neural operators to predict solutions reduces as T increases. This is unsurprising, as when the time horizon increases, the chaotic nature of the equation makes the output "differ" more from the input, and therefore harder for the network to learn. It is interesting to note that the increase in non-monotone, and the rate isn't exponential.

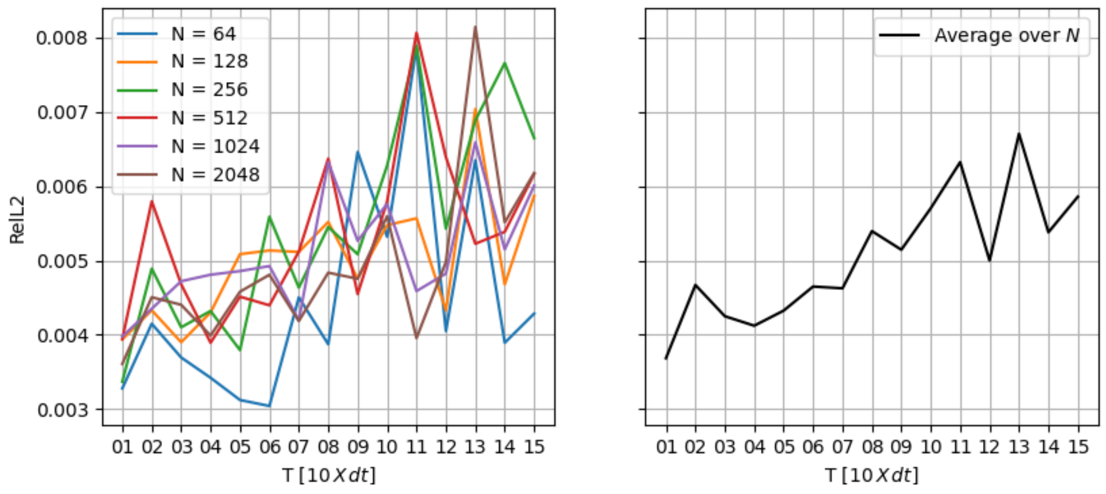


Figure 11. RelL2 test loss values for networks $\mathcal{N}_T$ trained on datasets of different time horizons $T = 1, \ldots, 15\,[10 \times dt]$ of the KS equation. ***Left*** *– each curve represents a different mesh size N.* ***Right*** *– average over N for the curves from Left. Constant network architecture $D = 64, M = 16, L = 4$.*

5.3 Super resolution

Figure 12 shows the results of a “super-resolution” test for KS equation. As done with Burgers equation in Section 4.5, we trained models on $N = 64, 128, 256, 512, 1024$, then tested them on datasets of higher resolutions (both for $T = 0.1$). Although KS and Burgers being very different — KS is chaotic and defined on Hermite FES, Burgers non-chaotic and defined on CG1 — we find that both behave similarly with the super-resolution test. Both Figure 12 and Figure 8 show that models that were trained on some resolution exhibit a big jump up in loss for slightly bigger resolutions, but eventually reach a plateau. This again can be attributed to the error being predominant among the other sourced of error. However, the difference between two consecutive resolutions become smaller as the mesh is refined, therefore we see a plateau.

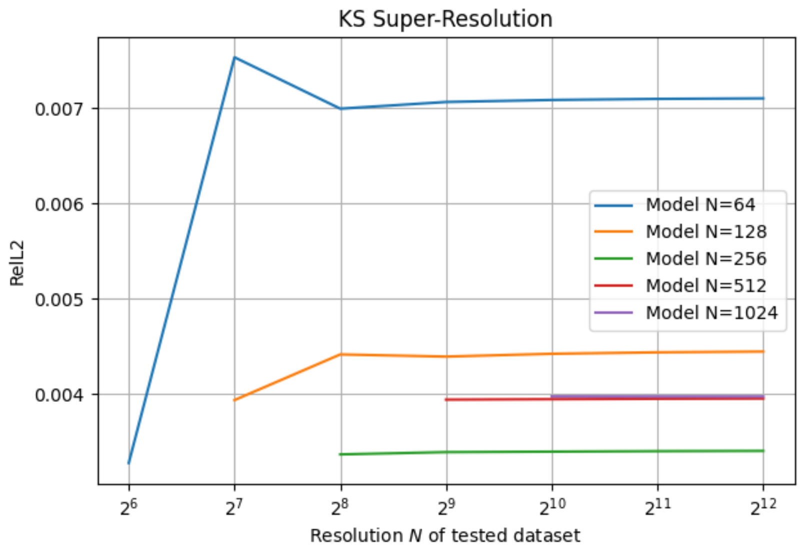


Figure 12. *RelL2 loss values for models trained on mesh $N = 64, 128, 256, 512, 1024$ and tested on datasets with higher resolutions (and on themselves). Both the models train dataset and the tested datasets are of time horizon $T = 0.1$. Network architecture $D = 64, M = 16, L = 4$.*

5.4 Composing time horizons

Assuming we start with an initial condition $u_0(x, t = 0) \in H^1$ on the attractor of the KS equation, we are interested in predicting the solution of (5.1) at time T, $u(x, T)$. We denote by $\mathcal{N}_h$ a neural operator that approximates the temporal evolution of the KS equation for h time units. Then if $T = kh$, $k \in \mathbb{N}$,

$$\mathcal{N}_h^k\left(u_0(x,0)\right) = \overbrace{\mathcal{N}_h \circ \cdots \circ \mathcal{N}_h}^{k\,\text{times}}\left(u_0(x,0)\right) \approx u(x,T)\,. \tag{5.2}$$

Figure 13 exhibits an example of an input-output pair, $a(x), u(x)$ from a $T = 2$ dataset, and a prediction from a composed network. The network was trained to predict $h = 1$ time advancement and was composed on itself two times, i.e. the prediction $\hat{u}(x) = \mathcal{N}_h \circ \mathcal{N}_h(a(x))$. As can be seen, the results aren’t good — at some points the network’s prediction (orange) strays away from the output (green).

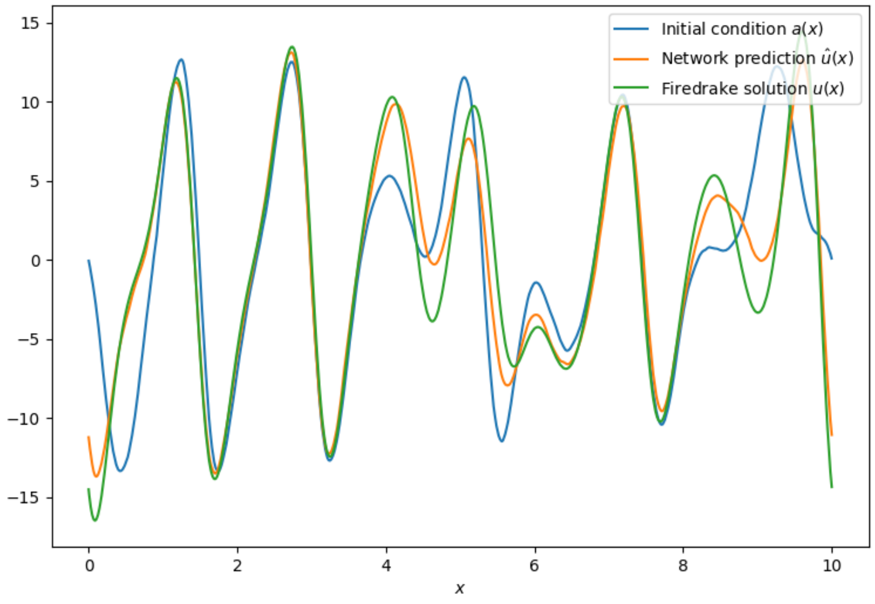


Figure 13. *Input-output pair (blue, green) from a* $T = 2$ *dataset, and prediction of a network trained on* $h = 1$ *composed on itself twice, i.e.* $\hat{u} = \mathcal{N}_h^2(a)$. *At some regions the prediction resembles the output, at others it strays away.* $N = 128$, *sample 1090, network architecture* $D = 64, M = 16, L = 4$.

In Figure 14 we tested the loss of such compositions. It shows the test loss of a network $\mathcal{N}_h$ trained on a $h = 1$ dataset and composed on itself $T = 1, \ldots, 15$ times. The loss was calculated between the composed predictions, and the outputs of the T dataset. The figure's x-axis indicates how many times the network was composed, i.e. k in equation (5.2). Evidently, the increase in loss is tremendous in the first step, jumping up by two orders of magnitude (from $T = 1$ to $T = 2$), meaning that the error from one output is carried on to the input in the next stage and exacerbated.

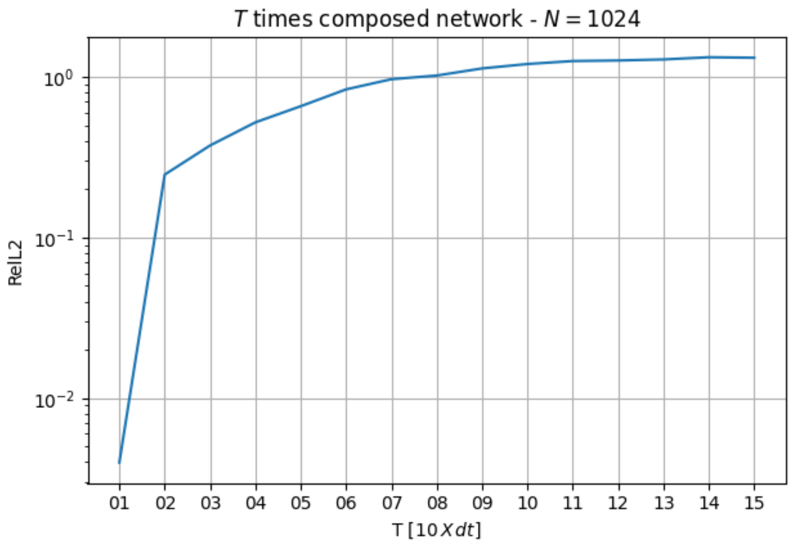


Figure 14. *RelL2 loss values for a network* $\mathcal{N}_h^T$ *trained on a* $h = T_1 = 1$ *dataset and composed on itself* T *times for* $T = 1, \ldots, 15\,[10 \times dt]$ *datasets, log scale. Mesh size* $N = 1024$, *network architecture* $D = 64, M = 16, L = 4$.

The experiment above shows that composed neural operators yield predictions that differ greatly from the long time horizon solutions we seek. We tried to isolate the discretization error, optimization error, and approximation error by repeating this experiment with different resolutions in Figure 15, networks that were trained for longer (1500 epochs) Figure 16(a), and big network architectures $D = 640$ Figure 16(b). All of these permutations gave almost identical poor outcomes, indicating that a major change in theory/implementation is needed before composition of neural operators would work well.

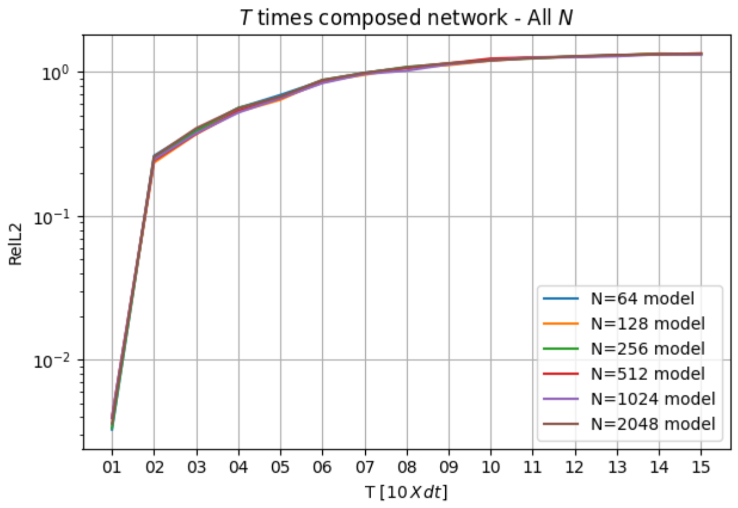
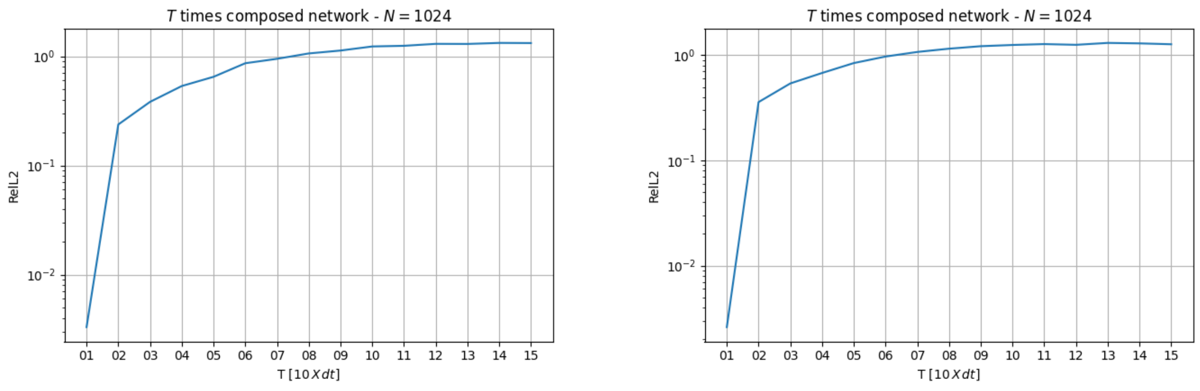


Figure 15. *RelL2 loss values for networks $\mathcal{N}_h^T$ trained on a $h = T_1 = 1$ dataset and composed on itself T times for $T = 1, \ldots, 15\,[10 \times dt]$ datasets. Each curve represents a different mesh size N, networks' architecture $D = 64, M = 16, L = 4$.*



(a) *1500 Epoch network* ***(b)*** *$D = 640$ network*

Figure 16. *RelL2 loss values for networks $\mathcal{N}_h^T$ trained on a $h = T_1 = 1$ dataset and composed on itself T times for $T = 1, \ldots, 15\,[10 \times dt]$ datasets.* ***(a)*** *— A standard network with $D = 64$ trained for 1500 epochs.* ***(b)*** *— A big network $D = 640$ trained for standard 750 epochs. Constant network architecture $M = 16, L = 4$.*

An interesting follow-up question is to what degree the predictions of composed neural operators adhere to the matter-conservation principle of the KS equation. Recall that the spatial integrals at all times $t \geq 0$ should evaluate to 0. Thus, we can define "the error" in this section to be nonzero values of the predictions' spatial integrals.

Before we turn to composed networks though, we need to examine the error of the original network with respect to the various datasets it was trained on. The errors of the outputs, which are Firedrake's solutions for various T, were cast from Float64 to Float32 so their error increased from 10^{-13} to 10^{-7}. The prediction errors for these outputs, taken from non-composed networks as in Section 5.2, were ranging in the scales 10^{-6} and 10^{-1}, and did not appear to have any dependence in T or N. Most of the predictions had errors bigger than 10^{-3} (see Figure B.3), meaning that for most inputs, the predictions are about 5 orders of magnitude away from the error of the Firedrake's outputs.

For composed networks, the result is unclear. We tested the sample with the smallest error in all our datasets, namely sample 1,036 of dataset $N = 1024, T = 1$. We tracked the error of its predictions from a network composed 10 times on itself, meaning we took the prediction as input to the network and iterated 10 times. The results in Table 3 show that by the second iteration, i.e. the first composition, the error jumps up by 4 orders of magnitude. However, further compositions did not increase the error, instead the error seems to fluctuate in the 10^{-1} regime. This effect is present with other samples, although they reach higher values of error. Therefore, although the compositions do not provide correct predictions and possess increased error, they maintain that error and it does not grow. It might indicate that the network learned some kind of matter-preservation.

k	**Error**
1	2.44×10^{-5}
2	0.43
3	0.12
4	0.63
5	0.20
6	0.03
7	0.19
8	0.07
9	0.05
10	0.15

Table 3. *Errors (integral of prediction) for sample 1,036 of dataset $N = 1024, T = 1$, for a network trained on the same dataset, composed on itself k times.*

6 Summary

Neural operators have gained significant attention in recent years due to their promise of efficiently approximating complex, infinite-dimensional operators that arise in various scientific and engineering applications. As we've shown, there are many ways to optimize a neural operator architecture to a given problem, making it a powerful contender in the league of quick, low-grade approximation models. Integration of FEM theory into the model provides even more versatility and advantages. However, while the potential of neural operators is undeniable, our analysis reveals several critical limitations, particularly in the context of chaotic dynamics, that warrant careful consideration.

One of the keys of this analysis was the successful integration of neural operators with finite element spaces, which allowed us to represent functions as coefficients in the nodal basis rather than as discretized point values (Section 3). This approach offers potential advantages in terms of computational efficiency, e.g. reuse of pre-calculated inner products instead of performing FFT in every single forward-pass through the network. Furthermore, the theory is independent of the choice of finite element space, making this framework compatible with the multitude of existing finite element variations (although in practice it'd usually require modifications). Using our framework, we managed to replicate known and expected results such as Fourier series expansion and discretization-invariance, thus verifying both our theory and code.

In Section 4, we applied our neural operator framework to the 1D Burgers equation, a well-known PDE that models various physical phenomena. Our results provide insights into the question of how to optimize a neural operator architecture for a given problem. We observed that while increasing the number of channels D or Fourier modes M generally improves accuracy, there is a point of diminishing returns. Additionally, we demonstrated a tradeoff between D and M, and a less obvious balance between L and M.

A particularly interesting finding from Sections (4.4.2-4.4.3) is that for one layered networks, the success of a network with certain M Fourier modes is correlated to the strongest frequencies in the data (specifically the outputs). This opens the possibility of matching the projection functions $(\psi_m)_m$ to the dataset for better results, and provides concrete boundaries for the choice of M. We had also shown that this effect is weaker in multilayered networks, as evidenced by the comparison between weight matrices.

In Section 5, we extended our analysis to the chaotic Kuramoto-Sivashinsky (KS) equation, which posed additional challenges due to its requirement for higher-order derivatives, Hermite FES, and CG3 FES. Despite these challenges, when the neural operator was trained and tested on the same time horizon dataset T, it approximated the solutions with reasonable loss of less than 1%. We did notice that accuracy declined for longer time horizons, which indicates that neural operators' performance is degraded when the period of chaotic evolution grows.

The limitations of neural operators became apparent when studying the composition of neural operators. Upon a single composition, the loss rose by 4 orders of magnitude, from about 0.1% to 10%, which makes it an infeasible approximation method. This behaviour persisted for different mesh resolutions, bigger networks, and longer training times. For further inspection, we examined how the neural operator adheres to the matter-preserving property of the equation, by comparing spatial integrals to the analytical value which is 0. While the ground truth solutions had integral values of 10^{-7}, non-composed networks produced values of 10^{-3}, 4 orders of magnitude away. Composed predictions fluctuated around the 10^{-1} value, which indicates that matter-conservation was learned, albeit the error increased by a few orders of magnitude. When taken together, these results suggest that composed neural operators don't approximate well the solutions, and aggressively diverge when composed multiple times. These findings raise questions on the applicability of neural operators to predict the long-term dynamics of chaotic equations. To that end, operators can be trained with composition in mind, perhaps by modifying the loss function to preserve desired measures, as demonstrated in [12].

We suggest a few things for further research. First, replicating our results using other FEM software and with other PDEs, both chaotic and predictable. Second, we suggest researching the choice of projection functions $(\psi_m)_m$. Clever choices of projection functions, such as non-analytical functions carefully suited to the data, might elevate the efficiency of a network considerably (akin to the frequency analysis of the dataset in Section 4.4.2). Thirdly, for implementations, we suggest to experiment with the tradeoffs in the architecture as described above. Keep in mind that bigger networks are not always slower, as some big networks achieve low loss values in shorter times (fewer training epochs) than small networks. Finally, further research should be put into a theory integrating chaotic equations and neural operators.

In conclusion, our analysis demonstrates that neural operators, when implemented within finite element spaces, offer a powerful and flexible tool for approximating solutions to complex PDEs. While neural operators have shown promise for some applications, their limitations emerge when applied to chaotic systems, specifically chaotic systems composed on themselves. The challenges of handling chaotic dynamics, particularly in terms of preserving governing laws and composing, suggest that neural operators, or at least our implementation, may not be the best tool for all types of problems.

References

[1] S. Lanthaler, Z. Li, and A. M. Stuart, *The Nonlocal Neural Operator: Universal Approximation (Version 1)*, arXiv, 2023, http://doi.org/10.48550/ARXIV.2304.13221.

[2] T. Chen and H. Chen, *Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems*, IEEE Trans. Neural Networks, 6 (1995), pp. 911–917, https://doi.org/10.1109/72.392253.

[3] G. Gupta, X. Xiao, and P. Bogdan, *Multiwavelet-based Operator Learning for Differential Equations (Version 2)*, arXiv, 2021, http://doi.org/10.48550/ARXIV.2109.13459.

[4] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators*, Nature Mach. Intell., 3 (2021), pp. 218–229, https://doi.org/10.1038/s42256-021-00302-5.

[5] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Neural Operator: Graph Kernel Network for Partial Differential Equations (Version 1)*, arXiv, 2020, http://doi.org/10.48550/ARXIV.2003.03485.

[6] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Fourier Neural Operator for Parametric Partial Differential Equations (Version 3)*, arXiv, 2020, http://doi.org/10.48550/ARXIV.2010.08895.

[7] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar, *Neural Operator: Learning Maps Between Function Spaces (Version 6)*, arXiv, 2021, http://doi.org/10.48550/ARXIV.2108.08481.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[9] D. Ham, P. Kelly, L. Mitchell, C. Cotter, R. Kirby, K. Sagiyama, and G. Markall, *Firedrake User Manual*, Imperial College London, 2023, https://doi.org/10.25561/104839.

[10] D. T. Papageorgiou and Y. S. Smyrlis, *The route to chaos for the Kuramoto-Sivashinsky equation*, Theor. Comput. Fluid Dyn., 3 (1991), pp. 15–42, https://doi.org/10.1007/bf00271514.

[11] Z. Li, M. Liu-Schiaffini, N. Kovachki, B. Liu, K. Azizzadenesheli, K. Bhattacharya, and A. Anandkumar, *Learning Dissipative Dynamics in Chaotic Systems (Version 2)*, arXiv, 2021, http://doi.org/10.48550/ARXIV.2106.06898.

[12] R. Jiang, P. Y. Lu, E. Orlova, and R. Willett, *Training neural operators to preserve invariant measures of chaotic attractors (Version 3)*, arXiv, 2023, http://doi.org/10.48550/ARXIV.2306.01187.

[13] Z. Li, H. Zheng, N. Kovachki, D. Jin, H. Chen, B. Liu, and A. Anandkumar, *Physics-Informed Neural Operator for Learning Partial Differential Equations (Version 4)*, arXiv, 2021, http://doi.org/10.48550/ARXIV.2111.03794.

[14] J. Pathak, S. Subramanian, P. Harrington, S. Raja, A. Chattopadhyay, M. Mardani, and A. Anandkumar, *FourCastNet: A Global Data-driven High-resolution Weather Model using Adaptive Fourier Neural Operators (Version 1)*, arXiv, 2022, http://doi.org/10.48550/ARXIV.2202.11214.

[15] D. A. Ham and C. J. Cotter, *Finite Element Method Course*, accessed Aug. 17, 2024, https://finite-element.github.io/Finiteelementcourse.pdf.

[16] S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*, Texts Appl. Math., Springer, New York, 2008, http://doi.org/10.1007/978-0-387-75934-0.

[17] V. Vapnik, *Principles of risk minimization for learning theory*, in Proc. 4th Int. Conf. Neural Information Processing Systems (NIPS'91), Morgan Kaufmann, San Francisco, CA, 1991, pp. 831–838.

A Appendices

A.1 Implementation

A.1.1 Implementation of Fourier modes

To employ the neural operator method discussed above, we choose a Fourier projection basis as follows:

$$\begin{aligned}
&\psi_0(x) = 1 \\
&\psi_{2m-1}(x) = \sin(\frac{2\pi m}{L}x) \\
&\psi_{2m}(x) = \cos(\frac{2\pi m}{L}x)
\end{aligned} \tag{A.1}$$

for $m \leq M,$ and L the length of the domain (1 for Burgers, 10 for KS). For example, setting $M = 8$ would give $2M + 1 = 17$ projection basis functions. These functions were interpolated (projected) into CG1 (Hermite) space, and we then saved their coefficients in a matrix referred to as the `functions` attribute of `ProjectionCoefficient` class, and Ψ in the analysis above.

The coefficients for Fourier series expansion, were calculated in a similar manner,

$$\begin{aligned}
C_{0n} &= \frac{1}{L}\int \phi_n(x)dx \\
C_{m,2n-1} &= \frac{2}{L}\int \phi_n(x)\sin(\frac{2\pi m}{L}x)dx \\
C_{m,2n} &= \frac{2}{L}\int \phi_n(x)\cos(\frac{2\pi m}{L}x)dx
\end{aligned} \tag{A.2}$$

for the same range of M, and $\{\phi_n\}_{n=1}^N$ nodal basis. These were calculated using Firedrake's `assemble` function and then saved. In the code the matrix is referred to as the `coeff` attribute of the `ProjectionCoefficient` class.

A.1.2 Hermite trial and error

Initially, we tried computing the KS equation on Hermite FES and transferring it into Pytorch without interpolating into CG3 space. Then, the coefficient in Pytorch would be point-evaluations and derivative point-evaluations. Then the loss defined above translates into Relative Sobolev $W^{1,2}$ norm $\|\cdot\|_{H^1}$, as,

$$\sqrt{h}\|\boldsymbol{u}\|_{\ell^2} = \sqrt{h\sum_{n=1}^{N} u^2(x_n) + h\sum_{n=1}^{N}(u'(x_n))^2} \xrightarrow{N\to\infty} \left(\int_a^b u^2(x)dx + \int_a^b (u'(x))^2dx\right)^{\frac{1}{2}} = \|u\|_{H^1}\,, \tag{A.3}$$

We trained and tested models with these coefficients and loss function, but got poor results, so we switched to CG3 space instead. There were two main problems with the implemntation of Hermite space:

1. On Hermite spaces, Firedrake stores the derivative point-evaluations on the reference element, rather than in the physical domain Ω. That means that the extracted data is a mix

of normal function point-evaluations, and scaled down derivative point-evaluations. To account for that, we tried scaling up the derivative coefficient by the relevant factor ($\frac{N}{L}$). Although this method was tested and verified, it left a nontrivial error that's dependent on the mesh size N, and was present in our results.

2. The activation function we used $\sigma(x)$ is gelu(x), which is a smooth version of relu$(x) = x \cdot 1_{\{x>0\}}$. The problem is, this function zeros out values that are too negative. With Hermite FES, some of the coefficients are derivative point-evaluations. Thus, the activation function can zero out derivatives without accordingly changing the function, or vice-versa. Other choices of activation function might cause similar problems, as they are all required to be nonlinear.

A.2 Code structure

All of the code and data used in this project were created by me and can be accessed here: `github.com/nadav7679/fem_neural_operator`. The code base for the project can get a bit involved. The data folder, including all of the testing/training samples for KS and Burgers, and thousands of post-trained models, weighs about 11Gb. However, the most important files are probably those in `classes`, and `analysis` folders, as they are the implementation to the neural operator framework and all of the graphs presented above. Below is a short description of each file, and later the hierarchy of folders.

- `classes\Dataset.py` implements a class that inherits from Pytorch's dataset class. It concatenates the coefficient inputs and the grid points input $\{x_n\}_{n=1}^{N}$ into one tensor, and saves the corresponding outputs (target).
- `classes\NetworkTrainer.py` has some basic boilerplate code to train and test a network.
- `classes\NeuralOperatorModel.py` has an abstract class that saves a neural network and its metadata, and handles various things about it like loading from file, saving to a file, training, and so on.
- `classes\NeuralOperatorNetwork.py` has my implementation of a neural operator corresponding to FEM and based on Section 3. Note that the `NeuralOperatorLayer` uses `xavier_uniform` initialization for weights, and a redundant Conv1D layer (redundant because `kernal_size=1`) for the linear part. Also note that it uses the coefficients and functions from `ProjectionCoefficient`, which essentialy would work with any functions and not jus Fourier modes.
- `classes\ProjectionCoefficient.py` calculation of Fourier coefficients and interpolated Fourier modes with Firedrake. The class also handles saving and loading its instances, which is in fact, a headache.

- `analysis\burgers\channel_analysis.py` and `analysis\burgers\channel_analysis_sample_size.py` produced Figure 3 and Figure 4.
- `analysis\burgers\mesh_analysis.py` produced part of the results in Table 2 and Table 1.
- `analysis\burgers\modes_analysis.py` produced Figure 5, and Figure 6.
- `analysis\burgers\super_resolution.py` produced, Figure 8.
- `analysis\burgers\miscellaneous.ipynb` produced Figure 2, Figure 7 Figure B.1, and was used for many, many tests.
- `analysis\KS\super_resolution.py` produced Figure 12
- `analysis\KS\mesh_analysis.py` produced the other part of Table 1.
- `analysis\KS\time_horizon.py` produced Figure 11.
- `analysis\KS\compositions.py` produced Figure 14, Figure 15, Figure 16.
- `analysis\KS\miscellaneous.ipynb` produced Figure 13, Figure B.2, Figure 9, Figure 10, Figure B.3.
- `.\verification.py` produced Figure 1.
- `data_generation\*.py` produced all of the initial, high resolution data for Burgers and KS equations (initial conditions and outputs). It was then downsampled and/or interpolated to other FES (such as Hermite to CG3) with the python files `data\*\samples\*.py`.

The hierarchy of the folders is as follows:

```
analysis/
|-- burgers/
|-- KS/
classes/
data/
|-- burgers/
|   |-- meshes/
|   |-- models/
|   |-- projection_coefficients/
|   |-- samples/
|-- KS/
|   |-- meshes/
|   |-- models/
|   |   |-- CG3/
|   |   |-- HER/
|   |-- projection_coefficients/
|   |   |-- CG3/
|   |   |-- HER/
|   |-- samples/
data_generation/
media/
```

B Supplementary material

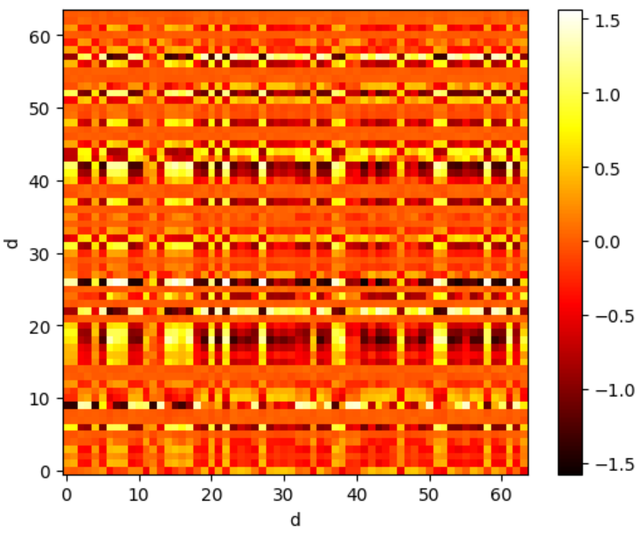


Figure B.1. Heatmap showing T_{16} for a trained $L = 1$ Burgers network.

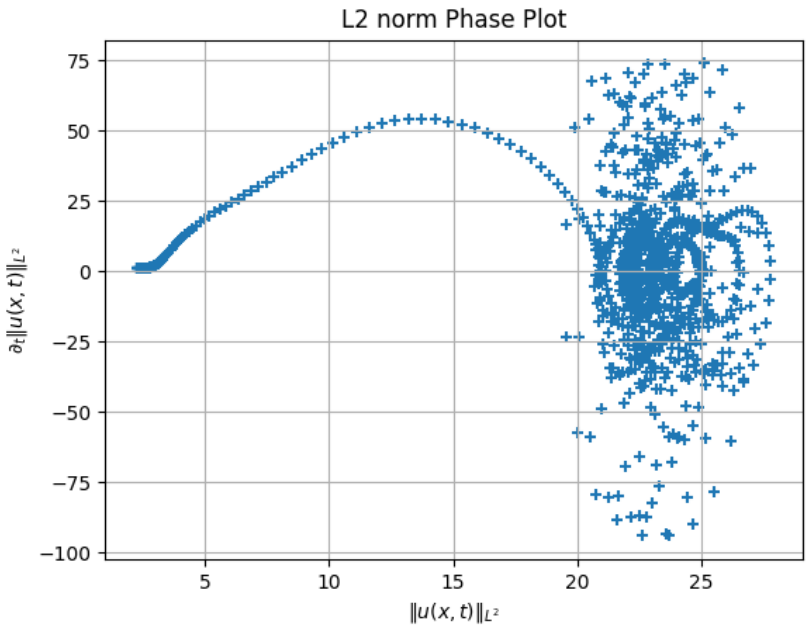


Figure B.2. Phase plot correlating to Figure 9, depicting transient stage on the left and chaotic attractor on the right.

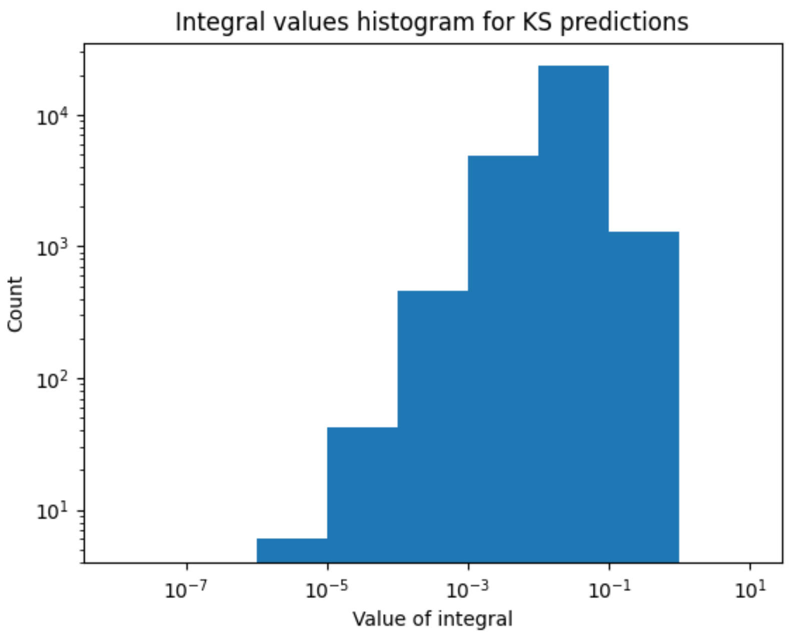


Figure B.3. *A histogram of the integrals of the predictions from KS networks. For each trained network and corresponding dataset in* $N = 64, 128, \ldots, 4096, T = 1, \ldots, 15$, *we computed the predictions of the test dataset (forward-pass of the inputs), computed the integrals using Firedrake, and made a histogram of them all.*

M	8	10	12	14	16	18	20	22	24	26
	16	16	20	24	24	28	32	32	16	48
	12	20	24	16	32	36	24	40	24	16
	14	18	22	28	28	20	16	24	36	52
	8	12	16	20	20	24	40	44	28	24
	10	14	12	26	30	32	36	16	44	20
	15	8	18	22	16	16	34	28	32	32
	11	15	14	12	26	30	38	36	20	40
	7	7	8	18	18	26	30	20	40	44
	2	10	23	14	22	34	28	30	48	28
	3	11	15	8	14	12	20	38	12	36
	4	19	11	15	12	22	26	42	46	50
	1	3	10	10	31	18	22	26	26	46
	6	2	19	23	11	14	18	34	34	42
	9	4	3	11	23	8	12	12	42	38
	13	1	7	7	8	11	14	22	38	18
	5	6	2	19	19	15	15	18	30	12
	0	13	4	27	27	19	23	14	18	26
		5	1	3	10	23	31	8	22	34
		9	17	2	3	7	11	23	31	22
		17	9	1	7	27	19	15	23	30
		0	5	4	2	31	39	39	14	31
			21	17	4	10	27	43	8	23
			13	25	1	4	7	7	39	39
			6	13	25	35	35	10	15	43
			0	6	5	2	10	11	47	7
				9	6	3	3	35	27	10
				21	1	1	2	27	19	3
				13	25	33	8	19	35	2
				6	5	25	1	2	7	8
				0	17	29	4	37	11	47
					13	21	1	3	2	27
					9	17	4	4	4	7
					0	13	1	29	43	2
						9	41	33	3	51
						0	25	1	1	35
							21	41	6	10
							13	25	43	43
							17	21	3	3
							9	13	1	1
							5	17	6	4
							0	9	41	49
								5	5	6
								6	45	45
								0	37	33
									29	41
									17	5
									9	37
									13	21
									33	13
									25	29
									21	25
									0	17
										9
										0

Table B.1. *Fourier modes for networks with different M, ordered by top to bottom by their weights. Note that 0 has always the least weight.*