

Principled Approaches for Extending Neural Architectures to Function Spaces for Operator Learning

Julius Berner^{1*}, Miguel Liu-Schiaffini^{2*}, Jean Kossaifi¹, Valentin Duruisseaux², Boris Bonev¹, Kamyar Azizzadenesheli¹, Anima Anandkumar²

¹NVIDIA ²Caltech

Abstract

A wide range of scientific problems, such as those described by continuous-time dynamical systems and partial differential equations (PDEs), are naturally formulated on function spaces. While function spaces are typically infinite-dimensional, deep learning has predominantly advanced through applications in computer vision and natural language processing that focus on mappings between finite-dimensional spaces. Such fundamental disparities in the nature of the data have limited neural networks from achieving a comparable level of success in scientific applications as seen in other fields. *Neural operators* are a principled way to generalize neural networks to mappings between function spaces, offering a pathway to replicate deep learning’s transformative impact on scientific problems. For instance, neural operators can learn solution operators for entire classes of PDEs, e.g., physical systems with different boundary conditions, coefficient functions, and geometries. A key factor in deep learning’s success has been the careful engineering of neural architectures through extensive empirical testing. Translating these neural architectures into neural operators allows operator learning to enjoy these same empirical optimizations. However, prior neural operator architectures have often been introduced as standalone models, not directly derived as extensions of existing neural network architectures. In this paper, we identify and distill the key principles for constructing practical implementations of mappings between infinite-dimensional function spaces. Using these principles, we propose a recipe for converting several popular neural architectures into neural operators with minimal modifications. This paper aims to guide practitioners through this process and details the steps to make neural operators work in practice¹.

1 Introduction to Neural Operators

In many scientific and engineering applications, data is derived from an underlying function, of which we have access to discretizations at different resolutions. We want to develop deep learning models that **respect the functional nature of the data** in the sense that they are **agnostic to the discretization and resolution**.

In the past decade, deep learning has led to unprecedented advances in domains like computer vision, speech, and natural language processing. Following this success, deep learning is poised to have an arguably even greater transformational impact on the natural sciences. While underlying data in all these areas comes in vastly different structures, the standard models powering deep learning, neural networks, consume data in a unified way: processed versions in the form of finite-dimensional *vectors*. For instance, these can be pixel representations of images and videos, word embeddings of language, or measurements of physical systems.

Yet, many physical phenomena of interest are inherently captured by *continuum* descriptions. Physical quantities describing, e.g., fluid and thermodynamics, continuum mechanics, or electromagnetism, depend on continuous variables such as spatial and temporal coordinates. Mathematically, such quantities are given as *functions* depending on continuous variables, and their behavior is often governed by differential equations modeling the relationship between physical quantities and their derivatives.

*Equal contribution

¹Our code is available at github.com/neuraloperator/NNS-to-NOs.

For instance, we can describe the temporal evolution of a given physical system, such as a heat distribution in a certain domain. This naturally leads to the task of predicting the physical system after a given time (i.e., the solution *function*) based on an initial condition of the system (i.e., an initial *function*). Common tasks in computer vision are inherently defined on functions as well since the underlying natural scenes or objects depend on spatiotemporal variables. Images or videos merely capture a discretized version with a certain resolution; e.g., an image is a discretization of a two-dimensional function.

While classical neural networks can process discretizations of continuous quantities (e.g., on a mesh or grid), they either rely on a fixed discretization or do not generalize to unseen resolutions. However, we would like the action of the neural network to be invariant² to the specific choice of discretization and its resolution. Furthermore, in numerous applications, it is essential for the output to be a function that can be queried at arbitrary coordinates, allowing for further operations (e.g., differentiation and integration). To this end, neural networks have recently been generalized to so-called *neural operators*. Neural operators are naturally formulated to operate on *functions* rather than *vectors*. In particular, *by construction*, they output functions that can be queried at arbitrary coordinates and that are consistent across different discretizations of the input function; more precisely the outputs only differ by a discretization error that vanishes as the discretization is refined. This property of being agnostic to the underlying discretization enables learning of function-to-function mappings and multi-scale phenomena for the wide range of tasks where the underlying data is captured by continuum descriptions. This has led to significantly improved performance and generalization, as well as powerful capabilities, such as zero-shot super-resolution, in a series of practical applications. In turn, these advances have spurred a large interest in developing different neural operator architectures and training paradigms. The increased popularity of neural operators has also led to the development of dedicated Python libraries, such as the NEURALOPERATOR library (Kossaifi et al., 2024), as well as large-scale benchmarks of these methods (Ohana et al., 2024).

Using neural networks to represent continuous functions, known as *implicit neural representations* or *neural fields*, has a long history (Lagaris et al., 1998). Special cases include *physics-informed neural networks* (PINNs) (Raissi et al., 2019), where the neural network approximates the solution function of a partial differential equation (PDE) by optimizing its parameters to reduce deviations from the governing equation. Shortly after the introduction of neural operators, 3D scenes in graphics and computer vision have been similarly represented using *neural radiance fields* (NeRF) (Mildenhall et al., 2020; Sitzmann et al., 2020; Jeong et al., 2022)³. In both of these cases, the neural network represents a *single* function, i.e., a single PDE solution or 3D scene. In contrast, neural operators can be viewed as generalizations of neural fields because they output a function for any given input function; instead of representing a single function, they can learn operators that act on functions. In other words, neural operators can be viewed as *conditional neural fields* (Azizzadenesheli et al., 2024), conditioned on different input functions. Alternatively, one can condition the implicit neural representations on the parameters of a (given or learned) finite-dimensional parametrization of the space of input functions (Chen et al., 2022; Serrano et al., 2023). While this can be viewed as a special case of an encoder-decoder neural operator (see Section 3.6), neural operators are more general and can handle input and output functions based on point evaluations without the need for parametrizations.

Contributions In this work, we identify and distill the core principles for constructing neural operators as practical and principled instantiations of parametrized mappings between function spaces. Using these principles, we then propose a recipe for converting popular neural architectures into neural operators with minimal modifications. This paper aims to guide practitioners through this process and details the steps to design and apply neural operators successfully in practice.

1. We **identify and motivate the main design principles of neural operators** and how they are necessary for well-posed operator learning (Section 2). In particular, neural operators are mappings

²The term “invariance” is a useful analogy; however, strictly speaking, neural operators still incur a (controllable) discretization error. Thus, we often say that a neural operator is “agnostic” to the discretization and produces “consistent” outputs for different resolutions. We introduce the appropriate mathematical concept of “discretization convergence” in Section 2.

³It is worth noting that the recent architectures (Shukla et al., 2021; Müller et al., 2022) share close similarities to early neural operator architectures such as (multipole) graph neural operators (Li et al., 2020c).

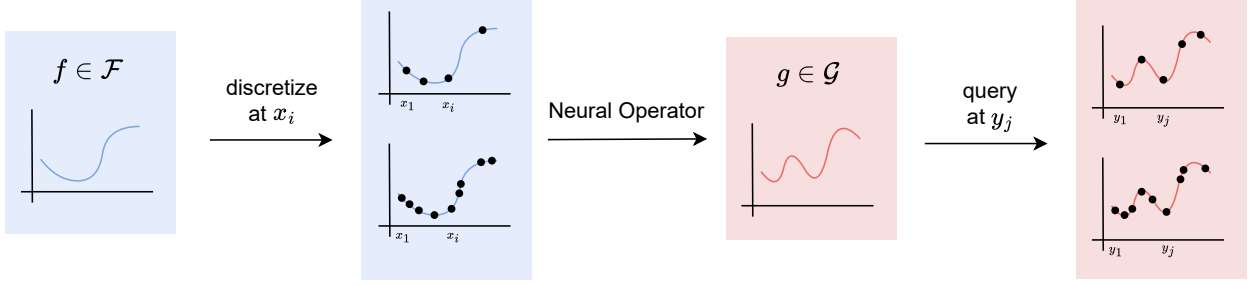


Figure 1: **Illustration of a neural operator.** The input is a function $f \in \mathcal{F}$ that can be given at different discretizations $(x_i)_{i=1}^n$. The output is a function $g \in \mathcal{G}$ that can be queried at different points $(y_j)_{j=1}^m$.

between function spaces parametrized by a finite number of parameters. They should be discretization-agnostic and be able to approximate any sufficiently regular operator with arbitrarily low error.

- Using the design principles of neural operators, **we establish a recipe for naturally extending successful architectures and layers to function spaces**, which we develop throughout Section 3 and summarize in Section 3.8. We also demonstrate throughout Section 3 that many popular neural operator architectures can be constructed using this procedure. In doing so, we also categorize existing neural operator architectures and layers. More specifically, we show that

- **Multilayer perceptrons** (Section 3.1) can be converted to integral transforms by parametrizing their weights and biases with learnable functions.
- **Convolutional neural networks** (Section 3.3) can be converted to convolutional integral operators by parametrizing the kernel as a learnable function.
- **Graph neural networks** (Section 3.4) become discretization-agnostic graph neural operators when quadrature weights are used for aggregation and neighborhoods are defined using subsets of the domain.
- **Self-attention in transformers** (Section 3.5) can be turned into resolution-agnostic self-attention using a similar strategy as for graph neural networks.
- **Encoder-decoder neural network architectures** (Section 3.6) can be generalized to function spaces by using parametric function classes for both the encoder and decoder.

We also show how other key building blocks of neural operator architectures (e.g., pointwise operators, positional encodings, and normalizations) can be similarly constructed from their finite-dimensional counterparts.

- We outline important **practical considerations for training neural operators** in Section 4. While many best practices from deep learning also directly apply to neural operators, we discuss in more detail the use of data and physics losses on function spaces (Section 4.1), and data augmentation, optimization, and auto-regressive modeling in the context of operator learning (Section 4.2).
- We survey the diverse landscape of neural operator architectures that have been designed and used successfully across a wide range of **scientific and engineering applications** (Section 5.1).
- Lastly, we **perform empirical experiments** (Section 5.2) to illustrate the importance of certain design choices for neural operators, such as fixing the receptive field with respect to the underlying domain and using quadrature weights for aggregations. We also analyze the effects of multi-resolution training and different interpolation strategies.

The more technical mathematical details are purposefully deferred to the appendix, to enhance the clarity of the presentation and prioritize an intuitive understanding. We refer the reader to Kovachki et al. (2023); Boullé & Townsend (2023); Kovachki et al. (2024b); Subedi & Tewari (2025) for a more theoretical treatment.

Table 1: Notation and comparison of neural networks to neural operators.

	Neural Network	Neural Operator	Input	Output
Space	Euclidean spaces (e.g. \mathbb{R}^d)	Function spaces	\mathcal{F}	\mathcal{G}
Elements	Vectors	Functions	$f \in \mathcal{F}$	$g \in \mathcal{G}$
Domain	Integer indices	Real-valued coordinates	$x \in D_f$	$y \in D_g$

2 From Discrete to Continuous: Learning on Function Spaces

Classical neural networks aim to approximate functions

$$x \mapsto f(x) \quad (1)$$

between (finite-dimensional) Euclidean spaces, such as \mathbb{R}^d . More precisely, they take a *vector* x as input and output another *vector* $f(x)$. In contrast, we want to construct mappings, so-called *operators*,

$$f \mapsto g \quad (2)$$

between (infinite-dimensional) function spaces⁴, such as continuous functions. Operators take a *function* f as input and output another *function* g ; see Table 1 for a comparison and Appendix A.1 for details on our notation.

The transition from discrete to continuous problems requires a change in how we represent and manipulate data. Naturally, while we are interested in learning mappings between functions, we typically do not have access to the actual functions numerically. Instead, neural operators are designed to work on a *discretization* $((x_i, f(x_i)))_{i=1}^n$ of a function f , assuming that we observe the function on a set of points x_i (i.e., a point cloud or grid; see Appendix A.2 for alternative representations). Moreover, the output of the neural operator, which is another function g , can be *queried* at arbitrary coordinates y in the domain of g ; see Figure 1 for an illustration.

2.1 Defining Principles of Neural Operators

Neural operators represent a principled extension of neural networks to function spaces that are parametrized by a **finite number of learnable parameters**. They **universally approximate function-to-function mappings** and provide **consistent predictions across resolutions**.

To learn a mapping between functions in practice, we would want to satisfy the following properties (see Appendix A.3 for details):

- **Discretization-agnostic:** the learned mapping should be applicable to functions given at any discretization and produce *consistent* outputs across resolutions.
- **Fixed number of parameters:** the number of learnable parameters needs to be *fixed* and independent of the discretization.
- **Universal approximation:** the family of mappings should be able to (provably) approximate, with arbitrarily low error, any sufficiently regular operator.

Neural Operators were designed from first principles to satisfy these properties and offer a natural framework for learning mappings between function spaces. In practice, neural operators still operate on discretized functions, but also respect the underlying functions they model and do not overfit to a fixed set of observation points. In particular, their outputs are functions that can be evaluated on arbitrary discretization, and the outputs of a neural operator for different discretizations only differ by a discretization error that can be made

⁴We use \mathcal{F} and \mathcal{G} to denote the function spaces for the input and output functions, which are typically suitable subspaces of continuous functions.

arbitrarily⁵ small by refining the discretization—we say that the neural operator is *discretization convergent*; see Appendix A.4.

2.2 Features of Neural Operators

Neural operators can be applied to data at different discretizations and **generalize across resolutions** by design. This is more **data-efficient**, **prevents overfitting** on the training resolution, and allows to **predict at arbitrary resolutions**.

Neural operators provide a principled framework to learn mappings on functional data. In the following we mention some of the unique features and advantages which are unlocked by neural operators and their ability to train and test on data with varying discretizations:

- **Well-posedness:** For many tasks, the ground truth mapping is naturally an operator, i.e., a mapping between *functions*. If we train a neural network that is not discretization-agnostic, we can overfit to the training discretization(s) (see Section 3.4). Operator learning allows us to approximate the ground truth operator and establish rigorous theoretical guarantees (see Appendix B). In addition, having output functions that can be queried arbitrarily is useful for downstream applications that require derived quantities such as derivatives or integrals.
- **Data efficiency:** Standard numerical solvers often create their meshes adaptively, which naturally leads to datasets with different discretizations (Li et al., 2024b; 2023; Rahman et al., 2024). Due to high computational costs, only a few high-resolution solutions are available in many applications, while a larger number of approximate solutions can be computed on discretizations with lower resolutions. Operator learning enables learning on different discretizations and resolutions simultaneously, ensuring sufficient data for training while maintaining the ability to resolve the underlying physics from the data at higher resolution. In practice, neural operators can work directly with functions discretized on point clouds with an arbitrary number of points and do not need an explicit mesh or partition of the domain into connected cells or elements.
- **Curriculum learning and faster training:** Neural operators can be trained via curriculum learning, where simpler, low-resolution samples are first used for training, gradually progressing to more challenging, high-resolution samples. Such an adaptively refinement of the discretization during training can not only improve performance but also accelerate convergence and reduce the computational cost compared to training only on high-resolution data (George et al., 2022; Lanthaler et al., 2024).
- **Flexible inference:** A trained neural operator can be queried at arbitrary resolution. For instance, we can obtain consistent predictions for discretizations that have not been in the training data, both at lower as well as higher resolutions, i.e., zero-shot super-resolution (Kovachki et al., 2023).

Motivated by the properties and capabilities of neural operators, the next natural question is how to construct and implement them in practice. This will be covered in the next section, in particular outlining which adaptations existing neural network layers require.

3 From Networks to Operators: Building Blocks of Neural Operators

Following a common paradigm in deep learning, an effective approach to designing neural operators is to construct simple and modular layers that can be composed to form expressive neural operator models. Each layer is itself an operator such that the intermediate outputs after each layer are *functions* (as opposed to *vectors* in standard neural networks). In particular, the discretized version of the neural operator can have a different discretization in each layer. In this section, we present how the most popular neural network layers can be converted to neural operator layers, ensuring they stay consistent across resolutions; see Figure 2 for a summary.

⁵In practice, we want to design neural operators where the discretization error is sufficiently small for the resolutions considered in applications.

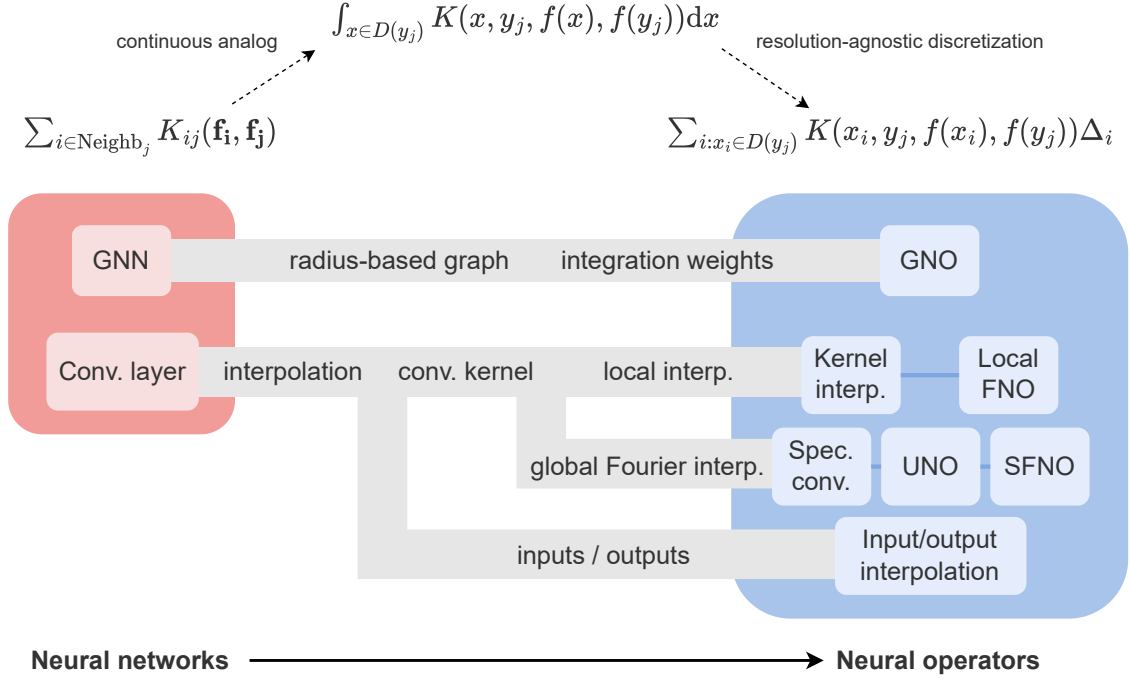


Figure 2: **Pipeline of converting neural networks to neural operators.** Graph neural network (GNN) and convolutional layers can be converted into well-posed neural operator layers through a sequence of simple modifications. GNO refers to the graph neural operator (Li et al., 2020b), “Spec. conv.” refers to a spectral convolution as used in Fourier neural operators (FNOs) (Li et al., 2020a), and LOCAL FNO refers to a FNO supplemented with local integral kernels (Liu-Schiaffini et al., 2024). We denote by UNO the U-shaped neural operator (Rahman et al., 2022b) and by SFNO the spherical Fourier neural operator (Bonev et al., 2023). The overall strategy to convert neural networks into neural operators is outlined in Section 3.8.

3.1 From Fully-Connected Layers to Integral Transforms

Integral transforms can be obtained by **parametrizing the weights and biases** of a fully-connected neural network layer **with learnable functions**. Instead of directly learning the weights for a fixed set of indices, we learn a function that maps input and query points to the value of the weights **weighted by the density of the input points**:

$$\mathbf{g}_j = \sum_i \mathbf{K}_{ji} \mathbf{f}_i + \mathbf{b}_j \quad \text{becomes} \quad g(y_j) = \sum_i K(x_i, y_j) f(x_i) \Delta_i + b(y_j).$$

We start by looking at the most basic building block of multi-layer perceptions (MLPs) and neural networks: the fully-connected layer. MLPs can simply be described by a learnable affine-linear map (we discuss the non-linearity in Section 3.2), and their parameters are typically learned end-to-end via backpropagation. However, the affine-linear map assumes a fixed, finite-dimensional vector as input. In this section, we present a natural extension to affine-linear operators on infinite-dimensional function spaces.

Fully-connected layer Assume we are given n values $\mathbf{f}_i = f(x_i)$ of a function f at points $(x_i)_{i=1}^n$ and a fully-connected layer with weight matrix $\mathbf{K} \in \mathbb{R}^{m \times n}$ and bias vector $\mathbf{b} \in \mathbb{R}^m$. The j -th coordinate of the output of the fully-connected layer can then be written as

$$\mathbf{g}_j := (\mathbf{K} \mathbf{f} + \mathbf{b})_j = \sum_{i=1}^n \mathbf{K}_{ji} \mathbf{f}_i + \mathbf{b}_j. \quad (3)$$

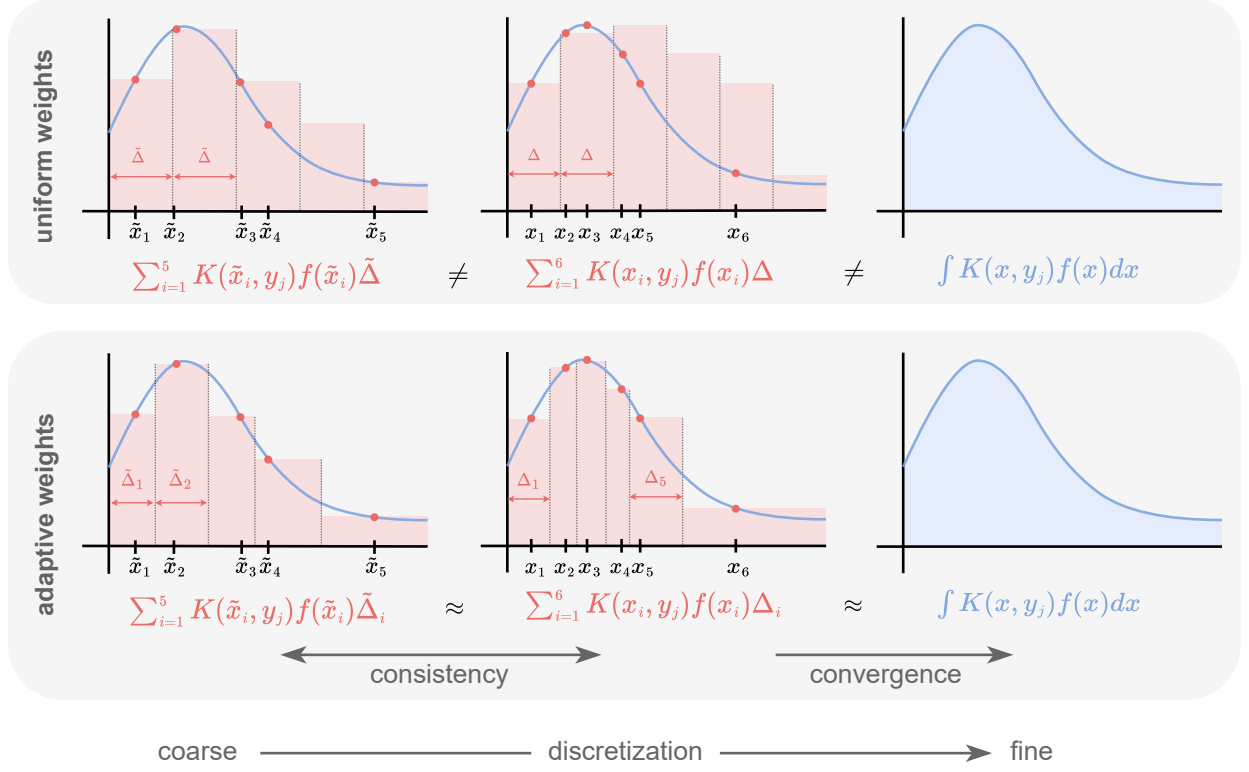


Figure 3: **Visualizing the need for quadrature weights.** Aggregating function values at irregularly-spaced points without proper quadrature weights (e.g., taking the mean, i.e., $\Delta = 1/n$, as in the top figure) adds more weight to densely sampled areas. When increasing the resolution, the output depends on the chosen refinement of the discretization and does not have a unique limit (**top**). The use of quadrature weights leads to consistent approximations for different discretizations that converge to a unique integral as the discretization is refined (**bottom**).

The weights are learned for *fixed input and output dimensions* n and m , which does not allow to work with different discretizations.

Integral transform To make this operation agnostic of the discretization, we assume that \mathbf{K}_{ji} and \mathbf{b}_j are evaluations of *functions* K and b at input points x_i and query points y_j . The result is a mapping from the input *function* f to an output *function* g given by

$$g(y_j) := \sum_{i=1}^n K(x_i, y_j) f(x_i) \Delta_i + b(y_j) \approx \int_{D_f} K(x, y_j) f(x) dx + b(y_j), \quad (4)$$

where we formally defined⁶ $\mathbf{K}_{ji} = K(x_i, y_j) \Delta_i$ and $\mathbf{b}_j = b(y_j)$ in (3). Specifically, note that we introduced so-called *quadrature weights* Δ_i , which ensure that the sum approximates the integral representation and that the output is agnostic to the resolution. For instance, we can view the sum as a *Riemann sum* for the integral, which converges as we refine the discretization. For one-dimensional grids, we can take $\Delta_i = \frac{x_{i+1} - x_{i-1}}{2}$ (with suitable definitions at the boundary); see Figure 3. For more general point clouds, we can partition the domain D_f and take the volume of the subregion containing x_i ; see Appendix A.5 and Algorithm 1. The *kernel* and *bias functions* K and b are typically parametrized by neural networks and learned end-to-end using backpropagation.

⁶Depending on the codomains of f and g , the output of K can be a scalar or a matrix. In the latter case, diagonal matrices or low-rank representations are often chosen for efficiency Kovachki et al. (2023).

Algorithm 1: Quadrature weights from a Delaunay triangulation in $d = 2$ (see Section 3.1 and Appendix A.5)

```
def compute_weights(x: Tensor[n_in, 2]):
    tri = Delaunay(x) # Compute Delaunay triangulation, e.g., from scipy.spatial
    delta = zeros(len(x)) # Initialize the quadrature weights
    for simplex in tri.simplices:
        points = x[simplex] # Vertices of the simplex (i.e., triangle in d=2)
        volume = 0.5 * abs(det(column_stack((points[1]-points[0], points[2]-points[0])))))
        for idx in simplex:
            delta[idx] += volume / 3
    return delta
```

Unlike the standard definition in (3), the expression in (4) can now be evaluated for an arbitrary number of input and output points n and m . This naturally extends the definition of a fully-connected layer, an affine-linear mapping between vectors \mathbf{f} and \mathbf{g} , to an affine-linear operator between functions f and g . Crucially, as we increase the resolution (i.e., refine the discretization), the model converges to the unique *integral operator* in (4), an instantiation of a *graph neural operator* (GNO) (Li et al., 2020b); see also Section 3.4 and Algorithm 3.

3.2 From Elementwise Functions to Pointwise Operators

Operations that are applied pointwise to the discretized function are **naturally agnostic to the resolution**. This includes activation functions as well as liftings and projections. However, they can be **queried only at the input points**.

We did not cover activation functions of the fully-connected layers in Section 3.1. However, we will see that this concept transfers directly to neural operators as part of the broader class of pointwise operators.

Elementwise function Several common layers in deep learning apply the same operation to each element of the input vector, such as activation functions, (1×1) -convolutions, or certain normalization layers. They can directly be used in neural operators if these elements correspond to the pointwise evaluations of the input function $\mathbf{f} = (f(x_i))_{i=1}^n$. In other words, there is a (potentially learnable) function K such that the layer can be written as

$$\mathbf{g}_i = K(\mathbf{f}_i). \quad (5)$$

Pointwise operator We can then define a pointwise operator, also called *Nemytskii operator* (Renardy & Rogers, 2006), mapping a function f to a function g , by⁷

$$g(x_i) = K(f(x_i)). \quad (6)$$

Such mappings are agnostic to the resolution as they can be applied to any discretization of the input function f . However, the output function g can only be queried at the same points x_i where the input function f is given, i.e., the discretizations of the functions f and g are necessarily the same. To query g at arbitrary points, one can compose the pointwise operator with another layer (e.g., in the simplest case, an interpolation layer; see also Section 3.7).

Pointwise operators allow transferring all activation functions from neural networks to neural operators. Moreover, they can be used for skip-connections and so-called lifting and projection layers. Specifically, given an input function $f: D_f \rightarrow \mathbb{R}^c$ with codomain \mathbb{R}^c , we can choose $K: \mathbb{R}^c \rightarrow \mathbb{R}^C$ (e.g., as a neural network) to map to a function $g: D_g \rightarrow \mathbb{R}^C$ with codomain \mathbb{R}^C , i.e., where the function is “lifted” ($C > c$) or “projected” ($C < c$); see Algorithm 2. Inspired by image processing, one also refers to c as the number of *channels* since a discretization \mathbf{f} on n points has shape⁸ $n \times c$. If the number of channels is adaptive (e.g., for multiphysics simulations with different physical variables), one can use attention mechanisms for the function K (Rahman et al., 2024).

⁷We note that K can also depend on both $f(x_i)$ and x_i . This is included in the definition by assuming that the coordinates are included in the output of the function, which is typically done in the form of a positional encoding; see Section 3.7.

⁸Depending on the order of the dimensions, such layers can also be efficiently realized using (1×1) -convolutions.

3.3 From Convolutional Neural Networks to Spectral Convolutional Operators

For discrete local convolutions, the receptive field is tied to the resolution and becomes increasingly local as the resolution increases. By **parametrizing the kernel as a learnable function**, convolutional operators can be obtained as special cases of integral operators **with a fixed receptive field**:

$$\mathbf{g}_j = \sum_{i=j-k}^{j+k} \mathbf{K}_{j-i} \mathbf{f}_i \quad \text{becomes} \quad g(y_j) = \Delta \sum_{i: |x_i - y_j| \leq r} K(y_j - x_i) f(x_i).$$

For **global convolutions**, this can be efficiently realized **using spectral convolutions**, leading to Fourier neural operators.

A core building block of modern neural network architectures is the convolutional layer. It can be viewed as a special case of a linear layer as in (3) and, similarly, can be extended to a special case of the integral operator in (4). Note that we omit the bias term for simplicity since it can be handled as in the previous subsection.

Convolutional layer We will see that the receptive field of convolutional layers changes when directly applying their discrete formulation to different resolutions. To demonstrate this, recall that the output of a one-dimensional convolutional⁹ layer with kernel $\mathbf{K} = (\mathbf{K}_i)_{i=-k}^k \in \mathbb{R}^{2k+1}$ of size $2k+1$ can be written as

$$\mathbf{g}_j = (\mathbf{K} * \mathbf{f})_j = \sum_{i=j-k}^{j+k} \mathbf{K}_{j-i} \mathbf{f}_i, \quad (7)$$

where $\mathbf{f}_i = f(x_i)$ are n evaluations of the input function f on an *equidistant* grid $x_i := i\Delta$ of width Δ (with suitable padding). In principle, such a convolution can be applied to discretizations of f on grids of any width Δ . However, this operation becomes increasingly local (in other words, the receptive field of the convolution shrinks) when decreasing the width Δ or, equivalently, increasing the number of discretization points. Specifically, notice that the output \mathbf{g}_j in (7) depends on f only at an interval of size $2k\Delta$ including the points x_{j-k} to x_{j+k} . When increasing the resolution, i.e., in the limit $\Delta \rightarrow 0$, the convolution thus converges to the pointwise operation¹⁰ $\mathbf{g}_j = f(x_j) \sum_i \mathbf{K}_i$ (by continuity of f); see Figure 4. Equivalently, this can be understood as zero-padding the kernel to the number of discretization points.

Convolutional operator To make the receptive field independent of the underlying resolutions, we can leverage the integral operator from Section 3.1 and again view \mathbf{K}_{j-i} as the evaluation of a (univariate) function K at the point $y_j - x_i$. Given that K has support in $[-r, r]$, we can then define the corresponding convolutional neural operator

$$g(y_j) = \Delta \sum_{i: |x_i - y_j| \leq r} K(y_j - x_i) f(x_i) \approx \int_{y_j-r}^{y_j+r} K(y_j - x) f(x) dx = (K * f)(y_j), \quad (8)$$

mapping the input function f to an output function g . Here, we formally defined $\mathbf{K}_{j-i} = K(y_j - x_i)\Delta$ with $k = \frac{r}{\Delta}$ in (7), where Δ are the quadrature weights for the equidistant¹¹ grid. In particular, the receptive field in (8) is now always of size $2r$, independent of the discretization of f . Moreover, the output function g can be queried at arbitrary points y_i (not necessarily grid points) since the function K in (8) can be evaluated at any point in its domain.

We can again parametrize K by neural networks or other parametric families of functions¹². One possibility is to interpolate the discrete kernel \mathbf{K} and scale by the quadrature weights, which also works for a pre-

⁹We treat the one-dimensional case only for simplicity, and the arguments naturally extend to higher dimensions. Moreover, we note that most modern deep learning frameworks such as PyTorch and JAX perform a *cross-correlation*, which differs from the convolution by a reflection of the kernel, i.e., a sign flip. This does not impact our presentation since the kernel is learned.

¹⁰Strictly speaking, this would converge to a pointwise operator as described in Section 3.2, however, with a very inefficient parametrization.

¹¹Using non-uniform quadrature weights $(\Delta_i)_{i=1}^n$ as in Section 3.1, it is straightforward to generalize the convolutional operator to general point clouds $(x_i)_{i=1}^n$.

¹²Similar approaches can be found in the context of *hypernetworks*, *scale-equivariant*, and *discrete-continuous (group) convolutions*; see Liu-Schiaffini et al. (2024) for an overview.

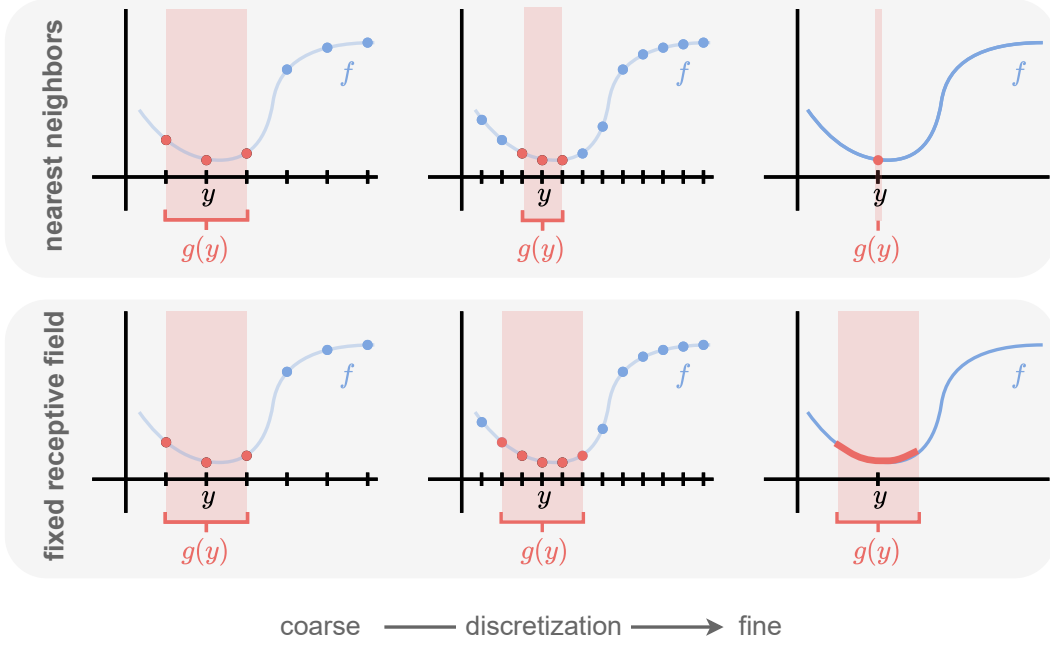


Figure 4: **Illustration of collapsing receptive fields with a nearest neighbors strategy.** The figure shows the values of the input function f (blue) which influence the output function g at a point y when using a nearest neighbors strategy (e.g. as in convolutional and graph *neural networks*) (**top**) and with a fixed receptive field (e.g. as in convolutional and graph *neural operators*) (**bottom**). If the neighborhood is selected using a nearest neighbors strategy, the receptive field (red) collapses when the discretization is refined (from **left** to **right**).

trained layer (see Section 5.2). We note that, due to the fixed receptive field, the computational cost of the convolutional operator in (8) scales as $\mathcal{O}(rn^2)$ (where $\Delta \sim \frac{1}{n}$) instead of $\mathcal{O}(kn)$ for the convolutional layer in (7).

If f and K are periodic, one possibility to obtain (quasi-)linear cost in n is to leverage the *convolution theorem* and rewrite the convolution in (8) as a multiplication in Fourier space, i.e.,

$$K * f = \mathcal{F}^{-1}(\mathcal{F}(K)\mathcal{F}(f)), \quad (9)$$

where \mathcal{F} denotes the mapping from a periodic function to its Fourier series. Instead of parametrizing the function K , we can now parametrize a *fixed* number of modes, i.e., Fourier coefficients $\mathcal{F}(K)$ (implying that K has full support). This is often referred to as a *spectral convolution* and represents the main building block of the *Fourier Neural Operator* (FNO) (Li et al., 2020a); see Algorithm 2 and Appendix A.6 for details.

In practice, \mathcal{F} and its inverse can be approximated¹³ with the *Discrete Fourier Transform* and accelerated using the *Fast Fourier Transform* (for sufficiently large grid resolutions and number of modes (Lingsch et al., 2023)) leading to an efficient implementation of (8). This can also be viewed as a Fourier interpolation of the kernel K in (7) to other resolutions. While such global convolutions are typically not used for tasks in computer vision, they can be important in scientific applications.

Multiple techniques, such as incremental learning or tensorization of $\mathcal{F}(K)$, have been developed to reduce the computational costs (George et al., 2022; Kossaifi et al., 2023). Moreover, Rahman et al. (2022b) proposed *U-shaped neural operators* (UNO) which reduce and increase the resolution of the discretization (and the number of modes) in deeper layers of the model (analogous to U-NETs (Ronneberger et al., 2015)).

¹³We refer to Bartolucci et al. (2023); Lanthaler et al. (2024) for an analysis of the discretization error (also referred to as *aliasing* error) depending on the grid resolution and spectrum of the input function.

Algorithm 2: Pointwise operator from Section 3.2, and spectral convolution from Section 3.3 in $d = 1$.

```
net = MLP(dim_in=codim_in, dim_out=codim_out) # MLP parametrizing the pointwise layer

def pointwise_layer(f: Tensor[bs, n_in, codim_in]) -> Tensor[bs, n_in, codim_out]:
    return net(f)

# Parameters for spectral convolution
params = Parameter(randn(modes, codim_in, codim_out, dtype=cfloat) * init_std)

def spectral_conv(f: Tensor[bs, n_in, codim_in], n_out=None) -> Tensor[bs, n_out, codim_out]:
    f_ft = rfft(f, dim=1)
    g_ft = einsum("bmi,mio->bmo", f_ft[:, :modes, :], params) # Multiply modes with parameters
    return irfft(g_ft, n=n_out or n_in, dim=1)
```

An alternative approach to interpolating the kernel, would be to interpolate the input function f to match the training resolution (Raonic et al., 2024). However, this has the downside of discarding information beyond the training resolution. Yet another approach is to rescale and center the kernel of discrete convolutions in order to converge to differential operators (Liu-Schiaffini et al., 2024).

3.4 From Graph Neural Networks to Graph Neural Operators

While graph neural networks can be natively applied to different discretizations of the input function, they do not necessarily generalize to unseen discretizations. To make them agnostic to the resolution, we need to **use quadrature weights for the aggregation** and **define neighborhoods using subsets of the domain**, independently of the discretization:

$$g_j = \sum_{i \in \text{Neighb}_j} K_{ij}(\mathbf{f}_i, \mathbf{f}_j) \quad \text{becomes} \quad g(y_j) = \sum_{i: x_i \in D(y_j)} K(x_i, y_j, f(x_i), f(y_j)) \Delta_i.$$

In this section, we look at discretizations of functions on more general point clouds and typical layers of graph neural networks. In their standard form, these layers are not agnostic to the resolution and require the use of quadrature weights and fixed receptive fields, as seen in previous sections. This will lead us to *Graph Neural Operators* (GNOs) (Li et al., 2020b), a general form of (nonlinear) integral operators. Similar to how graph neural networks generalize feed-forward and convolutional neural networks, GNOs can be seen as a unifying framework for the neural operator layers discussed in Sections 3.1 and 3.3.

Graph neural networks Let us interpret our discretization $(x_i)_{i=1}^n$ as a graph with a given connectivity and denote by Neighb_j the indices of points connected to x_j . The output of a prototypical *message-passing* layer would then be defined as¹⁴

$$g_j = \sum_{i \in \text{Neighb}_j} K_{ij}(\mathbf{f}_i, \mathbf{f}_j), \quad (10)$$

where K is a neural network and the sum could also be replaced by other permutation-invariant aggregation functions, such as a (weighted) mean or maximum. We note that this definition also includes *graph attention* and *graph convolution* layers (Bronstein et al., 2021). In particular, for nearest neighbor graphs on regular grids and $K_{ij}(\mathbf{f}_i, \mathbf{f}_j) = \mathbf{K}_{j-i} \mathbf{f}_i$, we recover the convolutional layers discussed in Section 3.3.

Graph neural operators As was done in the previous sections, we generalize (10) to the neural operator setting by:

- using the corresponding points x_i (and hence explicit positional information) instead of indices i ,
- aggregating the contributions in each neighborhood using quadrature weights Δ_i , and

¹⁴We omit node-wise operations since they can be treated as described in Section 3.2. Moreover, we implicitly allow for edge features by letting K depend on i and j and we assume that j is contained in its own neighborhood Neighb_j .

Algorithm 3: Linear integral operator from Section 3.4 with radius graph and diagonal kernel.

```
net = MLP(dim_in=2*dim, dim_out=codim) # MLP parametrizing the kernel

def integral_transform(
    f: Tensor[bs, n_in, codim], x: Tensor[bs, n_in, dim], delta: Tensor[bs, n_in],
    y: Tensor[bs, n_out, dim], radius=None
) -> Tensor[bs, n_out, codim]:
    # Kernel evaluation (if radius is not None, one should use sparse ops in CSR format)
    shape = [bs, n_in, n_out, dim]
    kernel_inp = [x.unsqueeze(2).expand(shape), y.unsqueeze(1).expand(shape)]
    kernel = net(cat(kernel_inp, dim=-1))

    # Weighted aggregation using the quadrature weights
    kernel *= delta.view(bs, n_in, 1, 1)
    kernel *= f.unsqueeze(2)
    if radius is not None:
        kernel[cdist(x, y) > radius, :] = 0
    return kernel.sum(dim=1)
```

- making the receptive field independent of the discretization by defining the neighborhoods based on suitable subsets $D(y_j)$ of the underlying domain (e.g., a common choice are *radius graphs* $D(y_j) = B_r(y_j)$, where points in a ball of certain radius r around y_j belong to the neighborhood).

In summary, this leads to the general (nonlinear) *graph neural operator* (GNO) given by

$$g(y_j) = \sum_{i: x_i \in D(y_j)} K(x_i, y_j, f(x), f(y_j)) \Delta_i \approx \int_{x \in D(y_j)} K(x, y_j, f(x), f(y_j)) dx, \quad (11)$$

where we formally defined $\text{Neigh}_{y_j} := \{i: x_i \in D(y_j)\}$ and $K_{ij} = K(x_i, y_j, \cdot, \cdot) \Delta_i$ in (10). It is easy to see that GNOs can represent both local and global integral operators depending on the choice of $D(y_j)$. In particular, the definition in (11) subsumes¹⁵ and generalizes the integral operators defined in (4) and (8). When using a local integral operator, Lin et al. (2025) propose to multiply the kernel with a differentiable weight function supported on $D(y_j)$ to retain the differentiability of g w.r.t. the query points y_j (which is needed for physics-based losses; see Section 4.1).

While we can use arbitrarily discretized functions f as inputs to the GNO, the general formulation in (11) only allows querying on the same points y_j . To this end, one often uses kernels of the form $K(x, y_j, f(x))$ (or variants as in (4) and (8) leading to linear operators) that do not depend on $f(y_j)$ and allow for queries at arbitrary points (Li et al., 2020b; 2024b); see Algorithm 3. While the GNO can deal with arbitrary geometries and discretizations, the evaluation of the kernel and aggregation in each neighborhood can be slow and memory-intensive. For certain discretizations and domains, considering convolutional kernels as in Section 3.3 allows to reduce the computational cost and establish equivariant¹⁶ layers.

3.5 From Transformers to Transformer Neural Operators

Attention layers in transformers can be made discretization-agnostic **using quadrature weights for the normalization and aggregation.**

While transformers can be converted into neural operators using a similar strategy as the one for GNNs, we highlight specific considerations in this section. In particular, we will focus on the attention mechanism and refer to Section 3.2 and Section 3.7 for tokenwise operations and positional embeddings.

¹⁵The operator in (4) is a GNO with $K(x, y_j, f(x), f(y_j)) = K(x, y_j)f(x)$ and $D(y_j) = D_f$, while the operator in (8) is a GNO where $K(x, y_j, f(x), f(y_j)) = K(y_j - x)f(x)$ with $D(y_j) = B_r(y_j)$.

¹⁶The approach in Section 3.3, yields translation-equivariant layers for regular grids on the torus. We refer to Helwig et al. (2023); Xu et al. (2024); Cheng & Peng (2024); Liu-Schiaffini et al. (2024) for neural operators that are equivariant to other groups.

Transformer A typical self-attention mechanism can be written as

$$g_j = \sum_{i=1}^n \frac{K(\mathbf{f}_i, \mathbf{f}_j)}{\sum_{\ell=1}^n K(\mathbf{f}_\ell, \mathbf{f}_j)} v(\mathbf{f}_i), \quad (12)$$

where K is an attention mechanism, e.g., *soft(arg)max with dot-product attention* $K(\mathbf{f}_i, \mathbf{f}_j) = \exp(\tau \langle k(\mathbf{f}_i), q(\mathbf{f}_j) \rangle)$ with temperature τ and learnable affine-linear mappings k , q , and v (see Appendix A.7).

Transformer neural operator The self-attention layer in (12) can be viewed as a variant of a graph neural network layer as in (10) with fully-connected graph and an additional normalization (i.e., the denominator in (12)). Due to the fully-connected graph, the receptive field is always the full domain, independent of the discretization. However, we still need to use quadrature weights for the aggregation and normalization to guarantee a unique limit when the discretization is refined. Doing so directly leads to the self-attention layer of *transformer neural operators* given by

$$g(y_j) = \sum_{i=1}^n \frac{K(f(x_i), f(y_j))}{\sum_{\ell=1}^n K(f(x_\ell), f(y_j))} v(f(x_i)) \Delta_i \approx \int_{D_f} \frac{K(f(x), f(y_j))}{\int_{D_f} K(f(z), f(y_j)) dz} v(f(x)) dx, \quad (13)$$

which maps input functions f to output functions g and converges to an integral operator as the discretization is refined (Kovachki et al., 2023). We note that for equal quadrature weights $\Delta_i = \Delta$, e.g., for regular grids, the weights cancel, and we recover the standard attention mechanism in (12). Since the layer only depends on the values of f , we need to assume that they incorporate a positional embedding (e.g., by concatenating the coordinates x_i to the output $f(x_i)$; see Section 3.7). Moreover, as explained in Section 3.4, the input function f can be discretized arbitrarily but g can only be queried at the same discretization. To circumvent this limitation, one can use *cross-attention* (see Appendix A.7).

Note that for self-attention layers, treating the points $f(x_i)$ as tokens leads to a quadratic cost in the size of the discretization, which might be prohibitive for fine resolutions. Alternatives include other variants of attention mechanism (without softmax and with linear computational cost) (Cao, 2021; Li et al., 2022b; Hao et al., 2023a; Li et al., 2024a), or tokenization along other dimensions (for instance, the channel dimension, i.e., dimension of the codomain of f (Rahman et al., 2024); see also Section 3.2).

To reduce computational costs, vision transformers (e.g. ViTs) use convolutional layers for the tokenization, i.e., they treat (non-overlapping) patches of the input as tokens (Choromanski et al., 2020; Dosovitskiy et al., 2020). However, these layers are dependent on the input discretization and need to be adapted as described in Section 3.3. Similarly, Calvello et al. (2024) define tokens by restricting the function to subsets of the domain (i.e., patches) and then use neural operators for tokenwise operations and inner products on function spaces for the attention mechanism. However, this requires sufficient resolution on each patch and can potentially create discontinuities at the boundaries of the patches.

3.6 From Finite-Dimensional to Infinite-Dimensional Encoders & Decoders

We can construct neural operator layers based on encoders and decoders mapping between **finite-dimensional latent spaces** and function spaces. These are often based on parametric function classes where the encoder **infers the parameters** given the input function, a learned mapping **transforms the parameters in latent space**, and the decoder **outputs the corresponding function** specified by the transformed parameters.

A common technique in deep learning is to use encoders and decoders, i.e.,

$$g = (\text{Decoder} \circ K \circ \text{Encoder})(f), \quad (14)$$

such that we can operate with a learnable function K , typically a neural network, in a latent space. Such an idea can also be used to construct neural operator layers. However, for neural operators, we need to make sure that the encoder is agnostic to the discretization of the input function f and that the output of the decoder can be queried at arbitrary resolution. In particular, we want to design encoders and decoders that map between function spaces and a finite-dimensional latent representations. Different from neural networks,

such encoder-decoder constructions are more often used for single layers of a neural operator and typically combined with pointwise skip-connections (see Section 3.2) to counteract the information bottleneck given by the finite-dimensional intermediate representation.

In the following, we present different practical versions of such encoders and decoders¹⁷.

Encoder Let us first describe different encoders, mapping discretizations of f to a k -dimensional latent representation $\mathbf{v} = \text{Encoder}(f)$, where k does *not* depend on the number of discretization points.

1. *Evaluation functionals*: In general, we can use k functionals mapping from \mathcal{F} to \mathbb{R} . An option used in DEEPONETS (Lu et al., 2019) are evaluation functionals, i.e., $\mathbf{v}_j = f(\xi_j)$, for fixed, so-called *sensor points* ξ_j . In this case, K is referred to as the *branch net*. However, such an encoder ignores the values of f at other points than ξ_j and requires the sensor points to be a subset of the discretization points.
2. *Inner products*: More generally, one can use approximations to inner products¹⁸ in function spaces, e.g.,

$$\mathbf{v}_j = \sum_{i=1}^n [b_j(x_i)]^* f(x_i) \Delta_i \approx \int_{D_f} [b_j(x)]^* f(x) dx = \langle b_j, f \rangle_{L^2} \quad (15)$$

with (potentially learnable) functions b_j as proposed in DEEPONET-OPERATORS (Kovachki et al., 2023); see also Algorithm 4.

3. *Projections*: We can project f to the closest function in a family¹⁹ $f_{\mathbf{v}}$ parametrized by the vector \mathbf{v} , which is then used as the latent representation. For a general function class, e.g., neural networks²⁰, there is no closed-form expression for the optimal parameters \mathbf{v} , requiring an inner optimization loop (Serrano et al., 2023). To this end, one often considers linear combinations $f_{\mathbf{v}} = \sum_{j=1}^k \mathbf{v}_j b_j$ given by a *dictionary* of functions b_j , e.g., subsets of classical approximation systems, such as polynomial bases, splines, Fourier bases, or wavelets (Li et al., 2020a; Gupta et al., 2021; Tripura & Chakraborty, 2022; Bartolucci et al., 2023). Finding the coefficients \mathbf{v} then leads to a linear least squares problem and, for orthogonal systems, \mathbf{v} can be inferred via inner products as in (15). For the Fourier basis, this corresponds to the discrete Fourier transform as used in the FNO.

Decoder Let us now present different decoders, mapping from a finite-dimensional vector $\mathbf{w} = K(\mathbf{v})$ to a function $g = \text{Decoder}(\mathbf{w})$ that can be queried at different points y .

1. *Linear combinations*: The finite-dimensional vector \mathbf{w} can be thought of as describing the coefficients of linear combinations²¹ $g_{\mathbf{w}} := \sum_{j=1}^k \mathbf{w}_j b_j$, where, as before, b_j are taken from a dictionary of functions. The dictionary can also be adapted to a (training) dataset of functions. For instance, one can use the first k basis functions of an (empirical) PCA decomposition (Bhattacharya et al., 2021; Lanthaler, 2023). The functions b_j can also be parametrized by neural networks (the so-called *trunk net*) and learned end-to-end as is done in DEEPONETS (Lu et al., 2019).
2. *Neural networks*: More generally, we can replace linear combinations of (learnable) dictionaries with a fully learnable neural network, where the input is given by the query point y and the vector \mathbf{w} , as proposed in the *Nonlinear Manifold Decoder* (NOMAD) (Seidman et al., 2022). Instead of using \mathbf{w} as an input, we can also view it as (latent representation of the) parameters of a neural network (Serrano et al., 2023). In this case, the resulting encoder-decoder layer can be viewed as a meta-network outputting a *neural field* conditioned on the input function f .

¹⁷For simplicity, we assume that $K: \mathbb{R}^k \rightarrow \mathbb{R}^k$ has the same input and output dimensions. We also note that the component functions of the input and output functions f and g are often considered separately for the encoder and decoder, but K is applied to the combined latent representations.

¹⁸If \mathcal{F} is a Hilbert space, the *Riesz representation theorem* shows that every continuous linear functional can be represented as an inner product. Note that for the L^2 -inner product in (15), we denote by $[b_j(x_i)]^*$ the conjugate transpose of $b_j(x_i)$.

¹⁹To guarantee discretization convergence (see Appendix A.4), we require suitable assumptions on the function class and the optimization problem.

²⁰If \mathbf{v} represents the weights of a neural network, $f_{\mathbf{v}}$ is often referred to as a *neural field* or *implicit neural representation*; see also Appendix A.2. For the function K , one can then use architectures that are specialized for operating on neural network parameters; see, e.g., Zhou et al. (2023); Lim et al. (2024).

²¹Mathematically speaking, such representations yield *linear* approximation methods (DeVore, 1998).

Algorithm 4: Encoder-decoder layer from Section 3.6 with inner product encoder and NOMAD decoder.

```

encoder = MLP(dim_in=dim, dim_out=latent_dim*codim_in)
net = MLP(dim_in=latent_dim, dim_out=latent_dim)
decoder = MLP(dim_in=latent_dim+dim, dim_out=codim_out)

def encoder_decoder(
    f: Tensor[bs, n_in, codim_in], x: Tensor[bs, n_in, dim],
    delta: Tensor[bs, n_in], y: Tensor[bs, n_out, dim]
) -> Tensor[bs, n_out, codim_out]:
    enc = encoder(x).view(bs, n_in, codim_in, latent_dim) * f.unsqueeze(-1)
    inner_products = (enc.sum(dim=2) * delta.unsqueeze(-1)).sum(dim=1)
    latent = net(inner_products)
    return decoder(cat([latent.unsqueeze(1).expand(bs, n_out, latent_dim), y], dim=-1))

```

In Appendix A.8, we show how certain integral operators, including spectral convolutions, can be viewed as encoder-decoder operators. In particular, for spectral convolutions as employed in FNOs, we use a truncated Fourier basis for the inner products in the encoder and the linear combinations in the decoder. Similarly we can recover the integral operators used in the *Laplace operator* (Cao et al., 2023) or the *(Multi-)Wavelet neural operator* (Gupta et al., 2021; Tripura & Chakraborty, 2022) when replacing the Fourier basis by the (truncated) eigenbasis of the Laplacian and the Wavelet basis, respectively.

Finally, we note that encoder-decoder operators are universal for continuous operators, i.e., for any accuracy, there exist an encoder, a mapping K , and a decoder approximating the operator uniformly on compact sets up to the given accuracy (Kovachki et al., 2023) when composed as in (14). However, in practice, we do not know the required size of the latent dimensions which governs the predefined parametric family of functions that the decoder can output. Thus, as mentioned before, one typically combines encoder-decoder layers with skip-connections to retain expressivity.

3.7 Auxiliary Layers

In this section, we outline some building blocks for neural operators that improve and stabilize performance or enable change of discretizations.

Positional encoding We can encode the “position”, i.e., the coordinates x , and add or concatenate it to the function values $f(x)$. The latter approach is often used as a first layer in neural operators to improve performance; see Algorithm 5. For the encoding of the position, often (non-learnable) sinusoidal embeddings as in NERFs or transformers are used (Mildenhall et al., 2020; Vaswani et al., 2017).

In transformers, positional encodings are important since attention layers are permutation equivariant. However, such encodings are often based on an absolute *ordering* of the tokens. Since this does not generalize to our setting where tokens correspond to (arbitrary) discretizations of a function f , we concatenate or add embeddings of the absolute (or relative; see Li et al. (2022b); Su et al. (2024)) *coordinates*.

Normalization As for neural networks, normalization of the data as well as intermediate representations is important for training. There are two important points to consider for the normalization of the data. First, we typically do not know the range of the data (e.g., in PDEs) and thus use *standardization*, e.g., centering the data to mean zero and rescaling to unit variance. Further, for computing the mean and variance, it is important to take into account the functional nature of the data, e.g., we compute

$$\mu = \frac{\sum_{i=1}^n f(x_i) \Delta_i}{\sum_{i=1}^n \Delta_i} \approx \frac{1}{|D_f|} \int_{D_f} f(x) dx, \quad (16)$$

which takes into account quadrature weights Δ_i and is thus agnostic to the resolution. The standard deviation is calculated analogously. We note that for equal weights, these statistics correspond to the standard mean and variance of the discretized vector \mathbf{f} . For preprocessing the input (and analogously the output) functions, the statistics are typically computed for each component function and averaged over the training dataset (such that μ and σ^2 have as many components as the dimension of the codomain of f). Adapting the

definition in (16), it is also straightforward to extend normalization layers, such as batch (Ioffe, 2015), layer (Lei Ba et al., 2016), or instance normalization (Ulyanov, 2016), to neural operators.

Interpolation Interpolation methods, such as spline, kernel, or Fourier interpolations, allow transferring from a discrete to a continuous representation. As mentioned in Section 3.2, they can be used after pointwise operators to allow querying at coordinates different from the input discretization.

Interpolation methods can also be used to make standard neural network layers discretization-agnostic. For instance, one can interpolate the input discretization to a latent, fixed discretization, apply a neural network, and then interpolate its output to allow querying at arbitrary points. This defines neural operator layers as in Section 3.6, where the interpolation can be viewed as (non-trainable) encoder and decoder.

In the same spirit, interpolation methods can also be applied to the input and output of a whole network to define a neural operator (Raonic et al., 2024). This method is capable of converting any pre-trained neural network into a neural operator. However, this approach suffers from some drawbacks: by fixing a fixed input resolution, the network cannot learn fine-scale details from high-resolution inputs since downsampling the input function leads to a loss of information (e.g., high frequencies for Fourier interpolation). Moreover, the output function space is restricted by the choice of interpolation method, which may dampen the model’s expressivity.

3.8 Summary

While there is no very precise and universal automated process to generalize arbitrary neural network layers and architectures to function spaces, there is a consistent underlying strategy that motivates and informs all the examples we have showcased. The first step is to investigate the neural network architecture to understand what would be an analogous transformation in terms of the continuous coordinates of the domain and the functions of interest (i.e., which continuous operator does the neural network roughly discretize). Building on this, the next step is to design strategies to discretize that continuous transformation in a way that respects the core principles presented in Section 2.

In practice, there are more precise mechanisms following that overall strategy that can be shared by families of neural network layers and architectures. As an example, many neural network architectures can be framed as variants of graph neural networks

$$\mathbf{g}_j = \sum_{i \in \text{Neigh}_j} K_{ij}(\mathbf{f}_i, \mathbf{f}_j). \quad (17)$$

The common strategy we followed for these neural network architectures is to use the explicit positional information (i.e., the coordinates x on the domain of f) and replace the message-passing summation over neighbors by an integral over a suitable neighborhood $D(y_j)$ in the underlying domain:

$$g(y_j) = \int_{x \in D(y_j)} K(x, y_j, f(x), f(y_j)) dx. \quad (18)$$

This integral operator is analogous to the graph neural network, in terms of the continuous coordinates x and the functions of interest K and f . In practice, the integral has to be discretized, which can be achieved by aggregating the contributions in each neighborhood using quadrature weights Δ_i (see Appendix A.5), thereby leading to the graph neural operator,

$$g(y_j) = \sum_{i: x_i \in D(y_j)} K(x_i, y_j, f(x_i), f(y_j)) \Delta_i, \quad (19)$$

which respects the core principles of Section 2 and enjoys the desirable properties of neural operators. We have also seen how the exact same strategy allowed to extend multilayer perceptrons to integral operators in Section 3.1, convolutional neural networks to convolutional operators in Section 3.3, graph neural networks to graph neural operators in Section 3.4, and transformer neural networks to transformer neural operators in Section 3.5. In Figure 2 we summarize how interpolating discrete convolutional kernels with various methods leads to different ways of constructing $K(x_i, y_j, \cdot, \cdot)$ from K_{ij} . For instance, (spherical) FNOs can be derived using global Fourier interpolation.

Algorithm 5: Training a neural operator (see Section 4)

```

neuralop = NeuralOperator(codim_in=codim_in+dim, codim_out=codim_out)
optim = Adam(lr=lr, neuralop.parameters())

def train_iteration(
    f: Tensor[bs, n_in, codim_in], x: Tensor[bs, n_in, dim], delta_x: Tensor[bs, n_in],
    g: Tensor[bs, n_out, codim_out], y: Tensor[bs, n_out, dim], delta_y: Tensor[bs, n_out]
):
    # Positional encoding and forward pass
    x = 2 ** arange(0, num_frequencies) * pi * x.unsqueeze(-1)
    x = stack([x.sin(), x.cos()], dim=-1)
    f = cat([f, x.view(bs, n_in, -1)], dim=-1)
    g_pred = neuralop(f, x, delta_x, y)

    # Loss with quadrature weights and backward pass
    loss = (square(g_pred - g).sum(dim=-1) * delta_y).sum()
    loss.backward()
    optim.step()

```

Other types of layers and operations require different types of modifications so they can be used in a principled manner on function spaces. For instance, encoders and decoders can rely on finite-dimensional latent representations of functions, as long as the encoder is agnostic to the discretization of the input function and the output of the decoder can be queried at arbitrary resolution (see Section 3.6). As another example, the key statistics used in normalization layers must account for the functional nature of the data, and this extends to inner products and norms which have to be defined on function spaces instead of vector spaces.

Finally, also note that certain operations are naturally agnostic to the resolution and directly transfer to neural operators, such as operations that are applied pointwise to the discretized function (e.g., activation functions, as well as liftings and projections), as discussed in Section 3.2.

4 Training neural operators

While the majority of best practices from deep learning and neural network training also directly apply to neural operators (see Algorithm 5), we present additional considerations that must be taken into account when training neural operators in this section.

4.1 Losses

For simplicity, we focus on the loss for a single input function and denote by g be the corresponding output of the neural operator provided at discretization points $(y_j)_{i=1}^m$. In practice, we use mini-batches (see Section 4.2) and average the losses of multiple input functions.

Data losses If we have ground truth data g^* , we can compute *data losses*, but need to make sure that we define the losses on *function spaces*, i.e., they should be agnostic to the resolution. For instance, using a squared L^2 -loss, we obtain

$$\text{Loss}(g, g^*) = \sum_{j=1}^m |g(y_j) - g^*(y_j)|^2 \Delta_j \approx \int_{D_g} |g(y) - g^*(y)|^2 dy, \quad (20)$$

where we use quadrature weights Δ_j for the points y_j to approximate the integral. We observe that for equal quadrature weights $\Delta_i = \Delta$ (e.g., regular grids), the loss in (20) is proportional to the usual mean squared error (MSE).

Analogously, one can also use other norms on function spaces, e.g., *Sobolev norms*, such as the squared H^1 -loss given by

$$\text{Loss}(g, g^*) = \sum_{j=1}^m \left(|g(y_j) - g^*(y_j)|^2 + |\nabla g(y_j) - \nabla g^*(y_j)|^2 \right) \Delta_j. \quad (21)$$

By providing supervision on the derivatives of the output functions, Sobolev losses guide the neural operator to accurately capture high-frequency features and smoothness properties of the ground truth function (Li et al., 2022c; Kossaifi et al., 2023). Computing Sobolev norms requires the ability to access or approximate the ground truth derivatives. For the output function g of the neural operator, derivatives (w.r.t. to its inputs y_j , *not* to the parameters of the neural operator) can be computed using, e.g., *finite-differences*, *Fourier differentiation*, or *automatic differentiation* (Wang et al., 2021; Maust et al., 2022; Lin et al., 2025). Note that when the ground truth derivatives cannot be accurately approximated (e.g., when using finite differences at low resolutions), Sobolev losses may be less effective.

Physics-informed losses If we know that the ground truth operator satisfies a certain (partial) differential equation, (spatiotemporal) derivatives of g can also be used to compute physics-informed losses, similar to *physics-informed neural networks* (PINNs) (Raissi et al., 2019; Sirignano & Spiliopoulos, 2018). This can act as a regularizer, reducing the need for extensive training data and enabling the model to be trained with low-resolution or even no data. In the later case, the PDE loss (together with appropriate boundary and initial conditions) should uniquely describe the solution g . Instead of relying on a finite number of ground truth input-output pairs, the supervisory signal coming from the PDE loss can be viewed as a “reward” that can be evaluated at arbitrarily chosen input functions f and corresponding output functions of the neural operator g (discretized at any resolution). While standard PINNs can only solve a single PDE²², physics-informed neural operators can learn the full operator from, e.g., coefficient functions or initial/boundary conditions²³ to the solution function (Wang et al., 2021; Li et al., 2024c). Still, most techniques developed for PINNs (see, e.g., Wang et al. (2023); Hao et al. (2023b) for summaries) can directly be extended to physics-informed neural operators.

Balancing multi-objective losses Scientific machine learning applications often involve predicting multiple variables and minimizing multiple loss terms (e.g., data and physics-informed losses), which are combined into a single objective function. However, appropriate weighting of the different terms is crucial for good performance and well-posedness. Since manual tuning can be a time-consuming and computationally intensive process, structured approaches have been developed that automatically balance losses based on their difficulties, gradient magnitudes, or causal dependencies²⁴; see, for instance, Kendall et al. (2018); Lam et al. (2023); Wang et al. (2023).

4.2 Training Strategies

Data augmentation For many scientific problems, it is difficult to develop physically-principled data augmentation strategies. We refer to Brandstetter et al. (2022) for a discussion on so-called *Lie symmetries* in the context of PDEs. We can also view different discretizations of input and output functions as a form of data augmentation. This can, for instance, be achieved by subsampling a given, fine discretization. While such *multi-resolution* training can also help neural networks, such as CNNs and GNNs, to work across resolutions, it does not provide any guarantees on their generalization beyond resolutions seen during training. Neural operators, on the other hand, do *not* require multi-resolution training to provide consistent predictions across (unseen) resolutions. However, it can still be employed to accelerate training (e.g., by starting with low-resolution data (Lanthaler et al., 2024)) and mitigate discretization errors (using a few samples with higher resolutions). We also note that a certain minimal resolution is required for at least part of the training data to resolve the physics and learn the ground truth operator. Finally, to handle the variable number of discretization points across samples, padding or sparse formats are required for efficient batching, analogous to common practices for GNNs (Fey & Lenssen, 2019).

²²Or, more general, families of PDEs parametrized by a *finite-dimensional* parameter.

²³In our notation, the function f collectively describes the considered family of PDEs, e.g., boundary conditions, coefficient functions, and/or geometries (e.g., given in terms of signed distance functions). Since these functions can be defined on different domains, one often uses suitable encodings or extensions to larger domains.

²⁴For instance, physics-informed losses have non-unique and often trivial solutions without appropriate boundary conditions (or additional data), motivating to put more weight on the latter in the beginning of the training.

Optimization As for neural networks, the optimization of neural operators is done using mini-batch stochastic gradient descent with gradients obtained via automatic differentiation, borrowing the same optimizers and learning rate schedulers. In particular, initialization is also crucial to enhance the convergence of stochastic gradient descent (Glorot & Bengio, 2010; He et al., 2015; Zhao et al., 2021; Zhang et al., 2019). For both initialization as well as optimization, one also needs to properly deal with complex values arising either in the spectral domain (e.g., for spectral convolutions) or due to complex-valued data (e.g., in quantum applications). This includes complex-valued initializations and arithmetics (in particular, for computing the momentum of optimizers). Finally, since achieving high accuracies is particularly important for scientific applications, we want to highlight research on natural gradients, preconditioning, and quasi-second order optimizers for improved convergence (Müller & Zeinhofer, 2024; Goldshlager et al., 2024; Anil et al., 2020).

Auto-regressive training In spatiotemporal forecasting problems, neural operators are often used in an autoregressive fashion, i.e., they are trained to predict a certain time into the future and then iteratively applied to their own output to predict longer time horizons during inference, often referred to as *rollout*. However, if the model is only trained on ground truth input data, this can lead to a distribution shift during inference causing instabilities for long autoregressive rollouts (Bonev et al., 2023). To this end, one can add artificial noise during training (see, e.g., Hao et al. (2024)), encourage dissipativity (Li et al., 2022c), or use auto-regressive rollouts also during training. In the latter case, one can avoid excessive memory overheads when backpropagating through the rollout by gradient checkpointing or backpropagating only the last few steps (Lam et al., 2023; Brandstetter et al., 2023).

5 Neural Operators in Practice

5.1 Applications of Neural Operators

In Section 3, we demonstrated how layers from popular neural networks can be modified into neural operator layers that remain consistent across resolutions. The properties of neural operators are particularly well-suited for science and engineering applications (Azizzadenesheli et al., 2024), and this section highlights a variety of domains where neural operators have already been successfully applied.

- **Fluid and solid mechanics:** Neural operators have fostered significant advancements in computational mechanics, including modeling porous media, fluid mechanics, and solid mechanics (You et al., 2022; Choubineh et al., 2023). They offer substantial speedups over traditional numerical solvers while achieving competitive accuracies and expanding their features (Kurth et al., 2023; Li et al., 2022a). For instance, FNOs constitute the first machine learning-based method to successfully model turbulent flows with zero-shot super-resolution capabilities (Li et al., 2020a). Li et al. (2022c) show that Sobolev losses and dissipativity-inducing regularization terms are effective in stabilizing long autoregressive rollouts for highly turbulent flows. Moreover, neural operators have also been used in large eddy simulations of three-dimensional turbulence (Jiang et al., 2023) and to learn the stress-strain fields in digital composites (Rashid et al., 2022). Finally, Yao et al. (2025) use neural operators in combination with diffusion models on function spaces to learn distributions over solutions when given sparse or noisy observations.
- **Nuclear fusion and plasma physics:** Neural operators have been used to accelerate magnetohydrodynamic (MHD) simulations for plasma evolution both from state and camera data (Gopakumar et al., 2024; Pamela et al., 2025). Instabilities arising in long-term rollouts using neural operators for plasma evolution have been studied in Carey et al. (2025), together with the potential of learning across different MHD simulation codes, data fidelities, and subsets of state variables. Furthermore, neural operators have been used for labeling the confinement states of tokamak discharges (Poels et al., 2025).
- **Geoscience and environmental engineering:** In the geosciences, FNOs and UNOs have been used for seismic wave propagation and inversion (Yang et al., 2021; Sun et al., 2022). Extensions of generative models to function spaces have been employed to model earth surface movements in response to volcanic eruptions or earthquakes, or subsidence due to excessive groundwater extraction (Rahman et al., 2022a; Seidman et al., 2023). Neural operators have also been used to model multiphase flow in porous media, which is critical for applications such as contaminant transport, carbon capture and storage, hydrogen storage, and nuclear waste storage (Wen et al., 2022; 2023; Chandra et al., 2025).

- **Weather and climate forecasting:** Versions of FNOs can match the accuracy of physics-based numerical weather prediction systems while being orders-of-magnitude faster (Kurth et al., 2023; Mahesh et al., 2024). To facilitate stable simulations of atmospheric dynamics on the earth, Bonev et al. (2023) introduced the *spherical Fourier neural operator* (SFNO) to extend FNOs to spherical geometries. The super-resolution capabilities of FNOs have also been leveraged for downscaling of climate data, i.e., predicting climate variables at high resolutions from low-resolution simulations (Yang et al., 2023). Additionally, neural operators have been utilized for tipping point forecasting, with potential applications to climate tipping points (Liu-Schiaffini et al., 2023).
- **Medicine and healthcare:** Neural operators have been used in multiple settings to improve medical imaging, such as ultrasound computer tomography (Dai et al., 2023; Zeng et al., 2023; Wang et al., 2025). As an example, they have been used on radio-frequency data from lung ultrasounds to accurately reconstruct lung aeration maps (Wang et al., 2025), which can be used for diagnosing and monitoring acute and chronic lung diseases. Jatyani et al. (2024) use FNOs supplemented with local integral and differential kernels (Liu-Schiaffini et al., 2024) for MRI reconstructions. Neural operators have also been used to improve the design of medical devices, such as catheters with reduced risk of catheter-associated urinary tract infection (Zhou et al., 2024). Finally, GNOs (Li et al., 2020b;c) have been used for spatial transcriptomics data classification (Ahmed & Yasin, 2023).
- **Computer Vision:** Neural operators have also been applied to several computer vision tasks. For instance, Guibas et al. (2021) use FNOs as token mixers in vision transformer architectures, and Zheng et al. (2023) use neural operators to accelerate diffusion model sampling. Other works have also explored using FNOs for image classification (Kabri et al., 2023) and segmentation (Wong et al., 2023).

5.2 Empirical Evaluations

In this section, we empirically evaluate different architectures and quantify the difference between using neural networks and neural operators. Specifically, we show that our proposed recipes for converting popular neural network architectures into well-defined operators lead to improved performance and generalization across resolutions.

For these experiments, we focus on the well-studied Navier-Stokes (NS) equations for incompressible fluids, which model the evolution of a fluid acted upon by certain forces. We consider the task of mapping from the force function to the vorticity of the fluid at a later time when initialized at zero vorticity; see Appendix C.1 for details. We compare the performance of different approaches in Figure 5 and provide details in the following paragraphs.

Receptive field When changing the resolution of the input function, neighboring points considered by the kernels of CNNs move closer or farther away from each other; see Section 3.3. We observe empirically, see “U-NET” in Figure 5, that such models do not generalize to resolutions different from the training resolution. We observe similar effects when naively applying a ViT (with convolutional tokenization within each patch); see ViT in Figure 5. While the ViT slightly outperforms the U-NET, the performance degrades substantially at resolutions different from the training resolution. The underlying reason is that the receptive field of each layer depends on the resolution. We compared several methods to fix the receptive field:

- For instance, one can *linearly interpolate the CNN kernel* (representing the kernel function K in Section 3.3), improving generalization; see “U-NET kernel interp.” in Figure 5.
- While interpolating the CNN kernel improves generalization, it still suffers from large discretization errors due to the low resolution of the kernel. However, using a *Fourier interpolation via a global spectral convolution* as in the FNO exhibits strong performance and generalization across all considered resolutions.
- A global receptive field is also used in “OFORMER” (Li et al., 2022b), which relies on a *linear version of self-attention* similar to the one discussed in Section 3.5. While it suffers from a large computational cost at higher resolutions, it provides good generalization capabilities.

Multi-resolution training Training U-NETs on multiple resolutions, see “U-NET (mixed res.)” and “U-NET kernel interp. (mixed res.)” in Figure 5, leads to good test-time performance at these different training resolutions but still poor generalization to unseen resolutions. In fact, such augmentation merely *encourage*

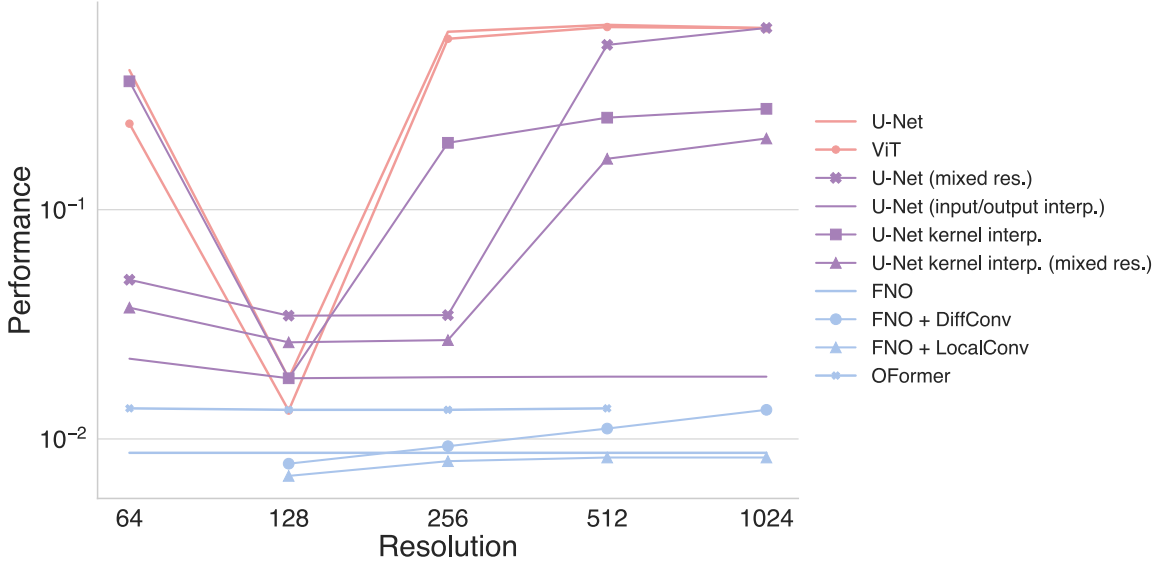


Figure 5: Relative L^2 -errors (on unseen test data) when training different methods on the Navier-Stokes equations (see Appendix C for details). Although FNO and OFORMER are only trained on resolution 128, they achieve approximately the same error for higher and lower resolutions. On the other hand, the U-NET only performs well on the training resolution 128. If we train on mixed resolutions $\{64, 128, 256\}$ (using each of the 10,000 data points three times with different resolutions in every epoch), it performs well for these resolutions—however, performance still degrades for higher resolutions. Interpolating the convolutional kernels of the U-NET (in both single and mixed-resolution training) also improves generalization across resolutions compared to the baseline U-NET. ViT is included as another common baseline in computer vision, and we observe similar results to those of the U-NET. We optimized the hyperparameters for each method and trained until convergence (see Appendix C). Due to memory constraints, the OFORMER results could not be computed for the highest resolution.

a model to be agnostic to the discretization observed in the *training* dataset, however do not provide a good inductive bias or theoretical guarantees for unseen resolutions. Since the training set can only cover a finite set of resolutions in practice, we do not consider neural networks trained with such data augmentation to be a robust and principled method of learning maps between function spaces.

Interpolation As outlined in Section 3.7, one can also use interpolation, i.e., up- and downsampling, of the input and output and then apply a neural network on the intermediate fixed resolution. While this defines a neural operator (with a fixed encoder-decoder as in Section 3.6) and generalizes across resolutions, see “U-NET (input/output interp.)” in Figure 5, its performance is limited since it discards any high-resolution information in the input. We note that this loss of high-resolution information could potentially be prevented or mitigated using skip-connections from the high-resolution input and combining it with other neural operator layers.

Composite architectures A key principle in architecture design is the idea that different resolution-agnostic layers can be combined to construct architectures that are also resolution-agnostic. In our experiments, we examine two such composite architectures, which we refer to as FNO + LOCALCONV and FNO + DIFFCONV, both of which are introduced in Liu-Schiaffini et al. (2024). LOCALCONV parametrizes the convolutional kernel K in (8) as a sum of a finite number of basis functions. DIFFCONV uses a traditional discrete convolutional kernel (e.g., a 3×3 kernel) but constrains the parameters to be mean-zero and rescales them according to the resolution. As such, although the receptive field shrinks, the action of the layer provably converges to a directional derivative as the resolution is increased. In FNO + LOCALCONV and FNO + DIFFCONV, the LOCALCONV and DIFFCONV layers are respectively placed in parallel with the spectral convolution within each layer of the model.

Composite architectures (such as FNO + LOCALCONV and FNO + DIFFCONV) inherit the properties of their various components. As such, our results show benefits of combining local (from the LOCALCONV and DIFFCONV) and global receptive fields (from the FNO) only at certain resolutions. While all these models are neural operators, different architectures can exhibit varying degrees of discretization error, depending on the training resolution and the data itself. Similarly, every neural operator architecture has a minimal resolution at which this error is controllable and such properties are also inherited by the overall composite architecture.

In summary, our experiments empirically verify three main principles developed in this paper:

- To provide the correct inductive bias for cross-resolution generalization, the receptive field of a neural operator must be fixed with respect to the underlying domain of the input function.
- Multi-resolution training can improve the performance of neural networks on the training resolutions but not for generalization to unseen resolutions.
- Interpolating the input and output of a neural network to an intermediate fixed resolution defines a naive neural operator but discards high-resolution information in the input.

6 Summary and Outlook

Successful neural network models, such as CNNs, GNNs, and transformers, have been developed for emulating mappings between finite-dimensional Euclidean spaces. However, for many applications, data arises naturally as functions. Training neural networks on such data requires to discretize the data at a fixed resolution, constraining their performance, efficiency, and generalization capabilities. In particular, predictions of neural networks made at different resolutions are in general inconsistent with each other.

In this work, we showed how simple modifications can make these popular neural network architectures agnostic to the discretization. This led us to the concept of neural operators as a principled framework for learning mappings between (infinite-dimensional) function spaces. We defined the general design principles and demonstrated how discretization-agnostic versions of popular architectures give rise to general building blocks for neural operators. Finally, we mentioned training recipes and successful applications of neural operators across various applications.

Since the field of neural operators is still in an early stage, there are a series of open research directions. While the discretization error of neural operators vanishes as the resolution is increased, it is important to be able to quantify and control this error in practical scenarios. The benefits of being discretization-agnostic also sometimes come at the cost of reduced performance at a *fixed* resolution compared to specialized neural network architectures. However, this is not a fundamental shortcoming of neural operators but rather requires further development of architectures with the right inductive biases. As starting points, one can leverage connections of neural operators to successful techniques in computer vision, such as implicit neural representations, or incorporate ideas from classical numerical methods.

Our work provides general guidelines that help practitioners understand the design principles and develop specialized neural operators for their applications. While neural operators have already seen great success in a variety of applications, we believe that they have the potential to accelerate progress and foster breakthroughs across a wide range of additional fields.

Acknowledgements

We thank Robert Joseph George for his contributions to the initial drafts. We thank Zijie Li and Md Ashiqur Rahman for fruitful discussions. M. Liu-Schiaffini is supported in part by the Mellon Mays Undergraduate Fellowship. A. Anandkumar is supported in part by Bren endowed chair, ONR (MURI grant N00014-18-12624), and by the AI2050 senior fellow program at Schmidt Sciences.

References

- Junaid Ahmed and Alhassan S. Yasin. Graph neural operators for classification of spatial transcriptomics data. *arXiv preprint arXiv:2302.00658*, 2023.
- Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.
- Kamyar Azizzadenesheli, Nikola Kovachki, Zongyi Li, Miguel Liu-Schiaffini, Jean Kossaifi, and Anima Anandkumar. Neural operators for accelerating scientific simulations and design. *Nature Reviews Physics*, pp. 1–9, 2024.
- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- Francesca Bartolucci, Emmanuel de Bézenac, Bogdan Raonić, Roberto Molinaro, Siddhartha Mishra, and Rima Alaifari. Are neural operators really neural operators? frame theory meets operator learning. *arXiv preprint arXiv:2305.19913*, 2023.
- Kaushik Bhattacharya, Bamdad Hosseini, Nikola B Kovachki, and Andrew M Stuart. Model reduction and neural networks for parametric PDEs. *The SMAI journal of computational mathematics*, 7:121–157, 2021.
- Boris Bonev, Thorsten Kurth, Christian Hundt, Jaideep Pathak, Maximilian Baust, Karthik Kashinath, and Anima Anandkumar. Spherical Fourier neural operators: learning stable dynamics on the sphere. *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 202:2806–2823, 2023.
- Nicolas Boullé and Alex Townsend. A mathematical guide to operator learning. *arXiv preprint arXiv:2312.14688*, 2023.
- Johannes Brandstetter, Max Welling, and Daniel E Worrall. Lie point symmetry data augmentation for neural PDE solvers. In *International Conference on Machine Learning*, pp. 2241–2256. PMLR, 2022.
- Johannes Brandstetter, Daniel Worrall, and Max Welling. Message passing neural PDE solvers. *arXiv preprint arXiv:2202.03376*, 2023.
- Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.
- Edoardo Calvello, Nikola B Kovachki, Matthew E Levine, and Andrew M Stuart. Continuum attention for neural operators. *arXiv preprint arXiv:2406.06486*, 2024.
- Qianying Cao, Somdatta Goswami, and George Em Karniadakis. LNO: Laplace neural operator for solving differential equations. *arXiv preprint arXiv:2303.10528*, 2023.
- Shuhao Cao. Choose a transformer: Fourier or Galerkin. *Advances in neural information processing systems*, 34:24924–24940, 2021.
- N. Carey, L. Zanisi, S. Pamela, V. Gopakumar, J. Omotani, J. Buchanan, J. Brandstetter, F. Paischer, G. Galletti, and P. Setinek. Data efficiency and long-term prediction capabilities for neural operator surrogate models of edge plasma simulations. *arXiv preprint arXiv:2502.17386*, 2025.
- Anirban Chandra, Marius Koch, Suraj Pawar, Aniruddha Panda, Kamyar Azizzadenesheli, Jeroen Snippe, Faruk O. Alpak, Farah Hariri, Clement Etienam, Pandu Devarakota, Anima Anandkumar, and Detlef Hohl. Fourier neural operator based surrogates for CO₂ storage in realistic geologies. *arXiv preprint arXiv:2503.11031*, 2025.
- Peter Yichen Chen, Jinxu Xiang, Dong Heon Cho, Yue Chang, GA Pershing, Henrique Teles Maia, Maurizio Chieramonte, Kevin Carlberg, and Eitan Grinspun. CROM: Continuous reduced-order modeling of PDEs using implicit neural representations. *arXiv preprint arXiv:2206.02607*, 2022.

- Chaoran Cheng and Jian Peng. Equivariant neural operator learning with graphon convolution. *Advances in Neural Information Processing Systems*, 36, 2024.
- Krzysztof Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.
- A Choubineh, J Chen, DA Wood, F Coenen, and F Ma. Fourier neural operator for fluid flow in small-shape 2D simulated porous media dataset. *Algorithms*, 16(1):24, 2023.
- Haocheng Dai, Michael Penwarden, Robert M. Kirby, and Sarang Joshi. Neural operator learning for ultrasound tomography inversion. *arXiv preprint arXiv:2304.03297*, 2023.
- Tim De Ryck and Siddhartha Mishra. Generic bounds on the approximation error for physics-informed (and) operator learning. *Advances in Neural Information Processing Systems*, 35:10945–10958, 2022.
- Ronald A DeVore. Nonlinear approximation. *Acta numerica*, 7:51–150, 1998.
- Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- Robert Joseph George, Jiawei Zhao, Jean Kossaifi, Zongyi Li, and Anima Anandkumar. Incremental spatial and spectral learning of neural operators for solving large-scale PDEs. *arXiv preprint arXiv:2211.15188*, 2022.
- Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.
- Gil Goldshlager, Nilin Abrahamsen, and Lin Lin. A Kacmarz-inspired approach to accelerate the optimization of neural network wavefunctions. *arXiv preprint arXiv:2401.10190*, 2024.
- Vignesh Gopakumar, Stanislas Pamela, Lorenzo Zanisi, Zongyi Li, Ander Gray, Daniel Brennand, Nitesh Bhatia, Gregory Stathopoulos, Matt Kusner, Marc Peter Deisenroth, Anima Anandkumar, the JOREK Team, and MAST Team. Plasma surrogate modelling using fourier neural operators. *Nuclear Fusion*, 64(5):056025, apr 2024.
- Philipp Grohs, Samuel Lanthaler, and Margaret Trautner. Theory-to-practice gap for neural networks and neural operators. *arXiv preprint arXiv:2503.18219*, 2025.
- John Guibas, Morteza Mardani, Zongyi Li, Andrew Tao, Anima Anandkumar, and Bryan Catanzaro. Adaptive Fourier neural operators: Efficient token mixers for transformers. *arXiv preprint arXiv:2111.13587*, 2021.
- Gaurav Gupta, Xiongye Xiao, and Paul Bogdan. Multiwavelet-based operator learning for differential equations. *Advances in neural information processing systems*, 34:24048–24062, 2021.
- Jayesh K Gupta and Johannes Brandstetter. Towards multi-spatiotemporal-scale generalized PDE modeling. *arXiv preprint arXiv:2209.15616*, 2022.
- Zhongkai Hao, Zhengyi Wang, Hang Su, Chengyang Ying, Yinpeng Dong, Songming Liu, Ze Cheng, Jian Song, and Jun Zhu. GNOT: A general neural operator transformer for operator learning. In *International Conference on Machine Learning*, pp. 12556–12569. PMLR, 2023a.
- Zhongkai Hao, Jiachen Yao, Chang Su, Hang Su, Ziao Wang, Fanzhi Lu, Zeyu Xia, Yichi Zhang, Songming Liu, Lu Lu, et al. Pinnacle: A comprehensive benchmark of physics-informed neural networks for solving PDEs. *arXiv preprint arXiv:2306.08827*, 2023b.

- Zhongkai Hao, Chang Su, Songming Liu, Julius Berner, Chengyang Ying, Hang Su, Anima Anandkumar, Jian Song, and Jun Zhu. DPOT: Auto-regressive denoising operator transformer for large-scale PDE pre-training. In *Forty-first International Conference on Machine Learning*, 2024.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.
- Jacob Helwig, Xuan Zhang, Cong Fu, Jerry Kurtin, Stephan Wojtowytsch, and Shuiwang Ji. Group equivariant Fourier neural operators for partial differential equations. *arXiv preprint arXiv:2306.05697*, 2023.
- Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- Armeet Singh Jatyani, Jiayun Wang, Aditi Chandrashekar, Zihui Wu, Miguel Liu-Schiaffini, Bahareh Tolooshams, and Anima Anandkumar. A unified model for compressed sensing mri across undersampling patterns. *arXiv preprint arXiv:2410.16290*, 2024.
- Yoonwoo Jeong, Seungjoo Shin, Junha Lee, Chris Choy, Anima Anandkumar, Minsu Cho, and Jaesik Park. Perfception: Perception using radiance fields. In *Advances in Neural Information Processing Systems*, volume 35, pp. 26105–26121, 2022.
- P. Jiang, Z. Yang, J. Wang, C. Huang, P. Xue, T.C. Chakraborty, et al. Efficient super-resolution of near-surface climate modeling using the Fourier neural operator. *Journal of Advances in Modeling Earth Systems*, 15, 2023.
- Samira Kabri, Tim Roith, Daniel Tenbrinck, and Martin Burger. Resolution-invariant image classification based on Fourier neural operators. In *International Conference on Scale Space and Variational Methods in Computer Vision*, pp. 236–249. Springer, 2023.
- Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7482–7491, 2018.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Jean Kossaifi, Nikola Kovachki, Kamyar Azizzadenesheli, and Anima Anandkumar. Multi-grid tensorized Fourier neural operator for high-resolution PDEs. *arXiv preprint arXiv:2310.00120*, 2023.
- Jean Kossaifi, Nikola Kovachki, Zongyi Li, David Pitt, Miguel Liu-Schiaffini, Valentin Duruisseaux, Robert Joseph George, Boris Bonev, Kamyar Azizzadenesheli, Julius Berner, and Anima Anandkumar. A library for learning neural operators. *arXiv preprint arXiv:2412.10354*, 2024.
- Nikola Kovachki, Samuel Lanthaler, and Siddhartha Mishra. On universal approximation and error bounds for Fourier neural operators. *Journal of Machine Learning Research*, 22(290):1–76, 2021.
- Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Learning maps between function spaces with applications to PDEs. *Journal of Machine Learning Research*, 24(89):1–97, 2023.
- Nikola B Kovachki, Samuel Lanthaler, and Hrushikesh Mhaskar. Data complexity estimates for operator learning. *arXiv preprint arXiv:2405.15992*, 2024a.
- Nikola B Kovachki, Samuel Lanthaler, and Andrew M Stuart. Operator learning: Algorithms and analysis. *arXiv preprint arXiv:2402.15715*, 2024b.
- Thorsten Kurth, Shashank Subramanian, Peter Harrington, Jaideep Pathak, Morteza Mardani, David Hall, Andrea Miele, Karthik Kashinath, and Anima Anandkumar. FourCastNet: Accelerating global high-resolution weather forecasting using adaptive Fourier neural operators. *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2023.

- Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.
- Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Ferran Alet, Suman Ravuri, Timo Ewalds, Zach Eaton-Rosen, Weihua Hu, et al. Learning skillful medium-range global weather forecasting. *Science*, 382(6677):1416–1421, 2023.
- Samuel Lanthaler. Operator learning with PCA-Net: upper and lower complexity bounds. *arXiv preprint arXiv:2303.16317*, 2023.
- Samuel Lanthaler, Siddhartha Mishra, and George E Karniadakis. Error estimates for DeepONets: A deep learning framework in infinite dimensions. *Transactions of Mathematics and Its Applications*, 6(1):tnac001, 2022.
- Samuel Lanthaler, Zongyi Li, and Andrew M Stuart. The nonlocal neural operator: Universal approximation. *arXiv preprint arXiv:2304.13221*, 2023.
- Samuel Lanthaler, Andrew M Stuart, and Margaret Trautner. Discretization error of Fourier neural operators. *arXiv preprint arXiv:2405.02221*, 2024.
- Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *ArXiv e-prints*, pp. arXiv–1607, 2016.
- Zhijie Li, Wenhui Peng, Zelong Yuan, and Jianchun Wang. Fourier neural operator approach to large eddy simulation of three-dimensional turbulence. *Theoretical and Applied Mechanics Letters*, 12(6):100389, 2022a.
- Zijie Li, Kazem Meidani, and Amir Barati Farimani. Transformer for partial differential equations’ operator learning. *arXiv preprint arXiv:2205.13671*, 2022b.
- Zijie Li, Dule Shu, and Amir Barati Farimani. Scalable transformer for PDE surrogate modeling. *Advances in Neural Information Processing Systems*, 36, 2024a.
- Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020a.
- Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*, 2020b.
- Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Andrew Stuart, Kaushik Bhattacharya, and Anima Anandkumar. Multipole graph neural operator for parametric partial differential equations. *Advances in Neural Information Processing Systems*, 33:6755–6766, 2020c.
- Zongyi Li, Miguel Liu-Schiaffini, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Learning chaotic dynamics in dissipative systems. *Advances in Neural Information Processing Systems*, 35:16768–16781, 2022c.
- Zongyi Li, Daniel Zhengyu Huang, Burigede Liu, and Anima Anandkumar. Fourier neural operator with learned deformations for PDEs on general geometries. *Journal of Machine Learning Research*, 24(388):1–26, 2023.
- Zongyi Li, Nikola Kovachki, Chris Choy, Boyi Li, Jean Kossaifi, Shourya Otta, Mohammad Amin Nabian, Maximilian Stadler, Christian Hundt, Kamyar Azizzadenesheli, et al. Geometry-informed neural operator for large-scale 3d PDEs. *Advances in Neural Information Processing Systems*, 36, 2024b.
- Zongyi Li, Hongkai Zheng, Nikola Kovachki, David Jin, Haoxuan Chen, Burigede Liu, Kamyar Azizzadenesheli, and Anima Anandkumar. Physics-informed neural operator for learning partial differential equations. *ACM/JMS Journal of Data Science*, 1(3):1–27, 2024c.

- Derek Lim, Haggai Maron, Marc T Law, Jonathan Lorraine, and James Lucas. Graph metanetworks for processing diverse neural architectures. In *International Conference on Learning Representations*, 2024.
- Ryan Y. Lin, Julius Berner, Valentin Duruisseaux, David Pitt, Daniel Leibovici, Jean Kossaifi, Kamyar Azizzadenesheli, and Anima Anandkumar. Enabling automatic differentiation with mollified graph neural operators. *arXiv preprint arXiv:2504.08277*, 2025.
- Levi Lingsch, Mike Y Michelis, Emmanuel de Bezenac, Sirani M Perera, Robert K Katzschmann, and Siddhartha Mishra. Beyond regular grids: Fourier-based neural operators on arbitrary domains. *arXiv preprint arXiv:2305.19663*, 2023.
- Miguel Liu-Schiaffini, Clare E Singer, Nikola Kovachki, Tapio Schneider, Kamyar Azizzadenesheli, and Anima Anandkumar. Tipping point forecasting in non-stationary dynamics on function spaces. *arXiv preprint arXiv:2308.08794*, 2023.
- Miguel Liu-Schiaffini, Julius Berner, Boris Bonev, Thorsten Kurth, Kamyar Azizzadenesheli, and Anima Anandkumar. Neural operators with localized integral and differential kernels. *arXiv preprint arXiv:2402.16845*, 2024.
- Lu Lu, Pengzhan Jin, and George Em Karniadakis. DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.
- Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.
- Ankur Mahesh, William Collins, Boris Bonev, Noah Brenowitz, Yair Cohen, Joshua Elms, Peter Harrington, Karthik Kashinath, Thorsten Kurth, Joshua North, et al. Huge ensembles part i: Design of ensemble weather forecasts using spherical Fourier neural operators. *arXiv preprint arXiv:2408.03100*, 2024.
- Haydn Maust, Zongyi Li, Yixuan Wang, Daniel Leibovici, Oscar Bruno, Thomas Hou, and Anima Anandkumar. Fourier continuation for exact derivative computation in physics-informed neural operators. *arXiv preprint arXiv:2211.15960*, 2022.
- Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision*, pp. 405–421, 2020.
- Johannes Müller and Marius Zeinhofer. Position: Optimization in SciML should employ the function space geometry. In *Forty-first International Conference on Machine Learning*, 2024.
- Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics*, 41(4):1–15, 2022.
- Ruben Ohana, Michael McCabe, Lucas Meyer, Rudy Morel, Fruzsina Agocs, Miguel Beneitez, Marsha Berger, Blakesly Burkhart, Stuart Dalziel, Drummond Fielding, et al. The well: a large-scale collection of diverse physics simulations for machine learning. *Advances in Neural Information Processing Systems*, 37:44989–45037, 2024.
- S.J.P. Pamela, N. Carey, J. Brandstetter, R. Akers, L. Zanisi, J. Buchanan, V. Gopakumar, M. Hoelzl, G. Huijsmans, K. Pentland, T. James, and G. Antonucci. Neural-parareal: Self-improving acceleration of fusion MHD simulations using time-parallelisation and neural operators. *Computer Physics Communications*, 307:109391, 2025.
- Yoei Poels, Cristina Venturini, Alessandro Pau, Olivier Sauter, Vlado Menkovski, the TCV team, and the WPTE team. Robust confinement state classification with uncertainty quantification through ensembled data-driven methods. *arXiv preprint arXiv:2502.17397*, 2025.
- Md Ashiqur Rahman, Manuel A Florez, Anima Anandkumar, Zachary E Ross, and Kamyar Azizzadenesheli. Generative adversarial neural operators. *arXiv preprint arXiv:2205.03017*, 2022a.

- Md Ashiqur Rahman, Zachary E Ross, and Kamyar Azizzadenesheli. U-NO: U-shaped neural operators. *arXiv preprint arXiv:2204.11127*, 2022b.
- Md Ashiqur Rahman, Robert Joseph George, Mogab Elleithy, Daniel Leibovici, Zongyi Li, Boris Bonev, Colin White, Julius Berner, Raymond A Yeh, Jean Kossaifi, et al. Pretraining codomain attention neural operators for solving multiphysics PDEs. *arXiv preprint arXiv:2403.12553*, 2024.
- Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.
- Bogdan Raonic, Roberto Molinaro, Tim De Ryck, Tobias Rohner, Francesca Bartolucci, Rima Alaifari, Siddhartha Mishra, and Emmanuel de Bézenac. Convolutional neural operators for robust and accurate learning of PDEs. In *Advances in Neural Information Processing Systems*, volume 36, 2024.
- Meer Mehran Rashid, Tanu Pittie, Souvik Chakraborty, and N.M. Anoop Krishnan. Learning the stress-strain fields in digital composites using Fourier neural operator. *iScience*, 25(11):105452, 2022.
- Michael Renardy and Robert C Rogers. *An introduction to partial differential equations*, volume 13. Springer Science & Business Media, 2006.
- Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.
- Jacob Seidman, Georgios Kissas, Paris Perdikaris, and George J Pappas. NOMAD: Nonlinear manifold decoders for operator learning. *Advances in Neural Information Processing Systems*, 35:5601–5613, 2022.
- Jacob H Seidman, Georgios Kissas, George J Pappas, and Paris Perdikaris. Variational autoencoding neural operators. *arXiv preprint arXiv:2302.10351*, 2023.
- Louis Serrano, Lise Le Boudec, Armand Kassai Koupai, Thomas X Wang, Yuan Yin, Jean-Noël Vittaut, and Patrick Gallinari. Operator learning with neural fields: Tackling PDEs on general geometries. *arXiv preprint arXiv:2306.07266*, 2023.
- Khemraj Shukla, Ameya D Jagtap, and George Em Karniadakis. Parallel physics-informed neural networks via domain decomposition. *Journal of Computational Physics*, 447:110683, 2021.
- Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.
- Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in neural information processing systems*, 33: 7462–7473, 2020.
- Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.
- Unique Subedi and Ambuj Tewari. Operator learning: A statistical perspective. *arXiv preprint arXiv:2504.03503*, 2025.
- Hongyu Sun, Yan Yang, Kamyar Azizzadenesheli, Robert W Clayton, and Zachary E Ross. Accelerating time-reversal imaging with neural operators for real-time earthquake locations. *arXiv preprint arXiv:2210.06636*, 2022.
- Tapas Tripura and Souvik Chakraborty. Wavelet neural operator: a neural operator for parametric partial differential equations. *arXiv preprint arXiv:2205.02191*, 2022.
- D Ulyanov. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
- Jiayun Wang, Oleksii Ostras, Masashi Sode, Bahareh Tolooshams, Zongyi Li, Kamyar Azizzadenesheli, Gianmarco Pinton, and Anima Anandkumar. Ultrasound lung aeration map via physics-aware neural operators. *arXiv preprint arXiv:2501.01157*, 2025.
- Sifan Wang, Hanwen Wang, and Paris Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. *Science advances*, 7(40), 2021.
- Sifan Wang, Shyam Sankaran, Hanwen Wang, and Paris Perdikaris. An expert’s guide to training physics-informed neural networks. *arXiv preprint arXiv:2308.08468*, 2023.
- Gege Wen, Zongyi Li, Kamyar Azizzadenesheli, Anima Anandkumar, and Sally M. Benson. U-FNO: an enhanced Fourier neural operator-based deep-learning model for multiphase flow. *Advances in Water Resources*, 163:104180, 2022.
- Gege Wen, Zongyi Li, Qirui Long, Kamyar Azizzadenesheli, Anima Anandkumar, and Sally M. Benson. Real-time high-resolution CO₂ geological storage prediction using nested fourier neural operators. *Energy Environ. Sci.*, 16:1732–1741, 2023.
- Ken CL Wong, Hongzhi Wang, and Tanveer Syeda-Mahmood. FNOseg3D: Resolution-robust 3D image segmentation with Fourier neural operator. In *2023 IEEE 20th International Symposium on Biomedical Imaging (ISBI)*, pp. 1–5. IEEE, 2023.
- Minkai Xu, Jiaqi Han, Aaron Lou, Jean Kossaifi, Arvind Ramanathan, Kamyar Azizzadenesheli, Jure Leskovec, Stefano Ermon, and Anima Anandkumar. Equivariant graph neural operator for modeling 3d dynamics. *arXiv preprint arXiv:2401.11037*, 2024.
- Qidong Yang, Paula Harder, Venkatesh Ramesh, Alex Hernandez-Garcia, Daniela Szwarcman, Prasanna Sattigeri, Campbell D Watson, and David Rolnick. Fourier Neural Operators for Arbitrary Resolution Climate Data Downscaling. In *ICLR 2023 Workshop on Tackling Climate Change with Machine Learning*, 2023.
- Yan Yang, Angela F Gao, Jorge C Castellanos, Zachary E Ross, Kamyar Azizzadenesheli, and Robert W Clayton. Seismic wave propagation and inversion with neural operators. *The Seismic Record*, 1(3):126–134, 2021.
- Jiachen Yao, Abbas Mammadov, Julius Berner, Gavin Kerrigan, Jong Chul Ye, Kamyar Azizzadenesheli, and Anima Anandkumar. Guided diffusion sampling on function spaces with applications to PDEs. *arXiv preprint arXiv:2505.17004*, 2025.
- Huaiqian You, Quinn Zhang, Colton J Ross, Chung-Hao Lee, and Yue Yu. Learning deep implicit Fourier neural operators (IFNOs) with applications to heterogeneous material modeling. *Computer Methods in Applied Mechanics and Engineering*, 398:115296, 2022.
- Zhijun Zeng, Yihang Zheng, Youjia Zheng, Yubing Li, Zuoqiang Shi, and He Sun. Neural born series operator for biomedical ultrasound computed tomography. *arXiv preprint arXiv:2312.15575*, 2023.
- Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.
- Jiawei Zhao, Florian Schäfer, and Anima Anandkumar. ZerO initialization: Initializing neural networks with only zeros and ones. *arXiv preprint arXiv:2110.12661*, 2021.

- Hongkai Zheng, Weili Nie, Arash Vahdat, Kamyar Azizzadenesheli, and Anima Anandkumar. Fast sampling of diffusion models via operator learning. In *International Conference on Machine Learning*, pp. 42390–42402. PMLR, 2023.
- Allan Zhou, Kaien Yang, Kaylee Burns, Adriano Cardace, Yiding Jiang, Samuel Sokota, J Zico Kolter, and Chelsea Finn. Permutation equivariant neural functionals. *Advances in neural information processing systems*, 36:24966–24992, 2023.
- Tingtao Zhou, Xuan Wan, Daniel Zhengyu Huang, Zongyi Li, Zhiwei Peng, Anima Anandkumar, John F. Brady, Paul W. Sternberg, and Chiara Daraio. AI-aided geometric design of anti-infection catheters. *Science Advances*, 10(1), 2024.

A Details

A.1 Notation

In this section, we provide additional details regarding the notation used in this paper, summarized in Table 1.

Continuous functions We are interested in learning mappings from a function space \mathcal{F} (*input functions*) to another function space \mathcal{G} (*output functions*). For input functions $f \in \mathcal{F}$, we denote by D_f their domain and by $f(x)$ their evaluation at coordinates $x \in D_f$. Similarly, for output functions $g \in \mathcal{G}$, we denote by D_g their domain and by $g(y)$ their evaluation at coordinates $y \in D_g$. For functions $g = (g_k)_{k=1}^c$ with *codomain* \mathbb{R}^c , i.e., $g: D_g \rightarrow \mathbb{R}^c$, we call the functions $g_k: D_g \rightarrow \mathbb{R}$ the *component functions* of g . While the input and output function spaces can change throughout the layers of a neural operator, we always use the same notation for ease and clarity of presentation. Moreover, we will typically denote learnable functions of layers by K (since they correspond to *kernels* of integral operators for many of our considered layers). In practice, these are typically parametrized by neural networks.

Discretized functions In practice, we will work on discretized functions, and assume that we have access to the function values $f(x_i)$ on a certain point cloud $(x_i)_{i=1}^n$ (see Appendix A.2 for other representations). Given that the function f has codomain \mathbb{R}^c , i.e., $f: D_f \rightarrow \mathbb{R}^c$, we can represent the function values as a tensor \mathbf{f} of shape $n \times c$, where, however, the size n depends on the chosen discretization. Note that we follow the convention of denoting finite-dimensional tensors (including vectors and matrices) by bold variables to distinguish them from functions. Analogously, we assume that the output function $g: D_g \rightarrow \mathbb{R}^C$ is represented by its evaluations $g(y_j)$ on a point cloud of query points $(y_j)_{j=1}^m$ of interest, leading to a tensor \mathbf{g} of shape $m \times C$ (where m depends on the number of query points). Finally, for a point x_i in a point cloud, we denote by Δ_i its *quadrature weight*, specifying its individual weight when discretizing an integral as a weighted sum over the point cloud (see Appendix A.5).

If the point clouds correspond to regular grids, we could interpret x_i and y_j as locations of pixel or voxel centers and \mathbf{f}_i and \mathbf{g}_j as the corresponding values. Borrowing terminology from computer vision, this interpretation also motivates to refer to the dimensions c and C of the codomains as the number of input and output *channels*. In the case of regular grids, the quadrature weights are equal, $\Delta_i = \Delta$, and proportional to the reciprocal number of points $\frac{1}{n}$.

A.2 Function Representations

To perform numerical operations on functions using a computer, suitable discrete representations are required. We focus on the common case where a function f is represented by its point evaluations $f(x_i)$ on a point cloud $x_i \in D_f$. If available, one can additionally consider derivative information, i.e., use

$$(x_i, f(x_i), \nabla f(x_i), \dots). \quad (22)$$

In some cases, we might have access to the function f directly, i.e., f is known analytically. This includes cases where f can be specified by coefficients, e.g., w.r.t. a suitable basis or via the parameters of a neural network²⁵. This can be viewed as a special case of our setting since we can compute evaluations of f on *arbitrary* points x . On the other hand, we discuss in Section 3.6 how to directly operate on coefficients describing a function f (which can be viewed as the output of a given *encoder*).

A.3 Neural Operators

This section provides further details on the definition and properties of neural operators. Mathematically speaking, a *neural operator* is a family of mappings

$$\text{NO}_\theta: \mathcal{F} \rightarrow \mathcal{G} \quad (23)$$

parametrized by a *finite-dimensional* parameter $\theta \in \mathbb{R}^p$. In the above, \mathcal{F} and \mathcal{G} are suitable function spaces; as an example, we can consider subspaces of continuous functions $C(D_{\mathcal{F}}, \mathbb{R}^c)$ and $C(D_{\mathcal{G}}, \mathbb{R}^C)$, where $D_{\mathcal{F}} \subseteq \mathbb{R}^d$ and $D_{\mathcal{G}} \subseteq \mathbb{R}^D$ are suitable domains. The families NO_θ , $\theta \in \mathbb{R}^p$, are typically designed to be *universal operator approximators*. Specifically, they should be able to approximate any sufficiently regular

²⁵Such neural networks are typically referred to as *neural fields* or *implicit neural representation*.

operator mapping from \mathcal{F} to \mathcal{G} arbitrarily well given a sufficiently large number of parameters p . We refer to Lanthaler et al. (2022); Kovachki et al. (2023); Lanthaler et al. (2023); Lanthaler (2023); Raonic et al. (2024) for corresponding results for some of the neural operators discussed in Section 3.

In practice, the input functions f are typically represented in the form of a discretization $f|_X = ((x_i, f(x_i)))_{i=1}^n$ on given points $X = (x_i)_{i=1}^n$; see Appendix A.2. Since we consider mappings that operate on discretizations with an arbitrary number of points n , we introduce the notation

$$(D_{\mathcal{F}} \times \mathbb{R}^c)^* := \bigcup_{n=1}^{\infty} (D_{\mathcal{F}} \times \mathbb{R}^c)^n \quad (24)$$

to denote discretizations of arbitrary size. Our goal is thus to construct *empirical versions*

$$\widehat{\text{NO}}_{\theta}: (D_{\mathcal{F}} \times \mathbb{R}^c)^* \rightarrow \mathcal{G} \quad (25)$$

of the operator NO_{θ} that are *discretization-agnostic*, i.e., that satisfy

$$\widehat{\text{NO}}_{\theta}(f|_X) \approx \text{NO}_{\theta}(f) \quad (26)$$

for every $f \in \mathcal{F}$ and discretization²⁶ X of the input domain. More precisely, we want to ensure that the error in (26) can be made arbitrarily small by considering a sufficiently fine discretization X , which can be mathematically formalized using the concept of *discretization convergence*; see Appendix A.4 and Kovachki et al. (2023) for details. In particular, the outputs of the empirical version of the neural operator at different discretizations only differ by a *discretization error* that can be made arbitrarily small by increasing the resolution. The following diagram visualizes this concept for the optimal case where we have equality in (26) (see Bartolucci et al. (2023) for corresponding conditions):

$$\begin{array}{ccc} \mathcal{F} \ni f & \xrightarrow{\text{NO}_{\theta}} & g \in \mathcal{G} \\ \downarrow \text{discretize at } X & \nearrow \widehat{\text{NO}}_{\theta} & \downarrow \text{discretize at } Y \\ (D_{\mathcal{F}} \times \mathbb{R}^c)^* \ni f|_X & \xrightarrow{\quad\quad\quad} & g|_Y \in (D_{\mathcal{G}} \times \mathbb{R}^c)^* \end{array}$$

The dashed line visualizes that, strictly speaking, some neural operator architectures (see, e.g., Sections 3.2 and 3.4) do not output a function but only its evaluation. Specifically, for those architectures, the output of $\widehat{\text{NO}}_{\theta}(f|_X)$ can only be queried on a specific point cloud $Y = (y_i)_{i=1}^m \subset D_{\mathcal{G}}$, and not the full domain $D_{\mathcal{G}}$ since this would require knowledge of f outside of the discretization points X . However, for ease of presentation, we assume that the output is suitably interpolated to $D_{\mathcal{G}}$ (see Section 3.7). Moreover, we emphasize that the same parameters θ can be employed across different discretizations of the input and output functions.

In Appendix B, we demonstrate how such assumptions quantify the “super-resolution” capabilities of a neural operator, i.e., how sufficiently expressive architectures, trained on a sufficiently fine resolution, are provably approximating the underlying ground truth operator.

A.4 Discretization Convergence

In this section, we formalize the approximation in (26) using the concept of discretization convergence. To this end, we need to assume that the discretization is refined in a “well-behaved” way. In particular, we consider sequences of nested sets

$$X_1 \subseteq X_2 \subseteq \dots \subseteq D_{\mathcal{F}} \quad (27)$$

with $|X_n| = n$. We call such a sequence a *discrete refinement* if for any radius $\varepsilon > 0$ there exists an index n , such that

$$D_{\mathcal{F}} \subseteq X_n + B_{\varepsilon}(0), \quad (28)$$

²⁶For some instantiations of neural operators, there can be a lower bound on the resolutions that they can operator. It also depends on the training discretizations and the properties of the ground truth operator whether we can accurately resolve the physics when querying at higher resolutions than the ones seen during training; see Appendix B.

i.e., the domain $D_{\mathcal{F}}$ can be covered using balls of radius ε centered at the n points in X_n (Kovachki et al., 2023). We now define the discretization convergence of an empirical version $\widehat{\text{NO}}_{\theta}$ of an operator NO_{θ} as in (23) and (25) by the following property: for any discrete refinement $(X_n)_{n \in \mathbb{N}}$ and any compact set $K \subseteq \mathcal{F}$ it holds that

$$\sup_{f \in K} \|\widehat{\text{NO}}_{\theta}(f|_{X_n}) - \text{NO}_{\theta}(f)\|_{\mathcal{G}} \rightarrow 0, \quad n \rightarrow \infty, \quad (29)$$

where $\|\cdot\|_{\mathcal{G}}$ is a given norm on \mathcal{G} ; see also Kovachki et al. (2023).

In practice, it is important to also consider the convergence speed in (29) since we want the discretization error to be as small as possible for practically relevant numbers of points n and not only in the limit. Moreover, while some layers, e.g., convolutional layers or graph neural network layers with nearest neighbor graphs, are technically discretization convergent, they converge to a typically undesirable pointwise operator; see Sections 3.3 and 3.4. For the neural operator layers considered in (4), (8), (11), and (13), the discretization convergence depends on the convergence of the utilized numerical integration scheme; see Appendix A.5.

A.5 Numerical Integration

This section provides further details on numerical integration schemes, which can be used to devise empirical versions of integral operator layers that can be applied to discretizations of functions at points $(x_i)_{i=1}^n$. Numerical integration schemes typically take the form

$$\int_D K(x) dx \approx \sum_{i=1}^n K(x_i) \Delta_i \quad (30)$$

for quadrature weights Δ_i and discretization points x_i . Different numerical integration schemes (also known as *quadrature* schemes) differ in their choices of weights Δ_i and quadrature points x_i , leading to different errors in the approximation (30) for functions K with certain regularity. In particular, this error dictates the discretization error of neural operators defined in terms of integral operators as in (4), (8), (11), and (13).

For Riemann sums, we can partition the domain D into n subdomains $(D_i)_{i=1}^n$ with $x_i \in D_i$, and choose $\Delta_i = |D_i|$ as the measure of D_i . Under mild assumptions on the partition, we obtain convergence to the integral in (30) for continuous functions K when $n \rightarrow \infty$ and $\max_{i=1}^n \Delta_i \rightarrow 0$.

For regular grids, the measure $|D_i|$ corresponds to the product of the grid-spacings in each dimension and is proportional to $\frac{1}{n}$. For general point clouds, one can, for instance, leverage a Delaunay triangulation to obtain a partition; see Algorithm 1. One can also derive methods based on analytically integrating simpler interpolating functions, such as polynomials, e.g., *Newton-Cotes formulas* for regular grids and *Gauss quadrature* for specifically chosen points.

We also want to mention *Monte-Carlo integration* for randomly distributed points x_i . To this end, we interpret the integral as an expectation and use a Monte-Carlo estimate, i.e.,

$$\int_D K(x) dx = \int_D \frac{K(x)}{p(x)} p(x) dx = \mathbb{E}_{x \sim p} \left[\frac{K(x)}{p(x)} \right] \approx \frac{1}{n} \sum_{i=1}^n \frac{K(x_i)}{p(x_i)} = \sum_{i=1}^n \underbrace{(np(x_i))^{-1}}_{:=\Delta_i} K(x_i). \quad (31)$$

where the points x_i are assumed to be i.i.d. samples drawn according to a density p .

We note that, e.g., for integral operators, K is learnable, and we can absorb parts of the quadrature weights Δ_i that only depend on the point x_i (as opposed to the full set of points or their size n) into the kernel. For instance, this includes constant factors. Moreover, for Monte-Carlo integration as in (31), we can also reparametrize K as $\tilde{K} = Kp$ such that the quadrature weights are $\frac{1}{n}$ for the integrand \tilde{K} .

A.6 Fourier Neural Operator

This section provides further background information on Fourier neural operators (FNOs) (Li et al., 2020a). In their typical form, they first apply a pointwise lifting layer (see Section 3.2) to increase the dimension of the codomain (i.e., number of channels) of the input function and, thus, the expressivity²⁷ of the overall

²⁷Lanthaler et al. (2023) show that only a single Fourier mode, i.e., just a global average, is necessary to achieve universality if the dimension of the codomain is sufficiently large. However, such an approximation is likely inefficient in terms of the required number of parameters.

architecture. This is followed by a number of FNO blocks, each consisting of a spectral convolution (see Section 3.3) and a pointwise operator (such as an activation function or an MLP; see Section 3.2) combined with skip-connections. The Fourier coefficients are computed separately for each component function, i.e., in practice, the discrete Fourier transform is batched across channels. However, for each frequency, the Fourier coefficients of different component functions are linearly combined, i.e., the channels are mixed in the Fourier space. Since the output of a spectral layer is necessarily bandlimited as the kernel only contains a fixed number of modes, we use a pointwise layer as a skip-connection to retain high frequencies. The last layer of the FNO is given by a pointwise projection layer (see Section 3.2) to account for the problem-specific dimension of the codomain of the output function.

As described in Section 3.6, one could replace the Fourier basis in the FNO with other parametrized function classes. However, the former choice has several advantages. First, for regular grids, the *Fast Fourier Transform* (FFT) offers a highly-efficient implementation with reduced computational complexity²⁸. For general grids, the discrete Fourier transforms can be used as an (approximate) mapping between a discretized function and its Fourier coefficients. Moreover, using a truncated Fourier basis is motivated by the fact that for many PDEs, such as the Navier-Stokes equations, it is known that the spectrum satisfies a certain decay, i.e., the magnitude of the Fourier coefficients decays at a certain rate as the frequency increases. An efficient strategy is thus to progressively increase the number of Fourier modes k together with the input resolution during training (George et al., 2022). This curriculum learning approach allows the FNO to initially focus on learning low-frequency components of the solution (avoiding wasted computation and overfitting) and progressively adapt to higher-frequency components as the data resolution increases.

One issue with methods relying on the Fourier basis (as the spectral convolutions in the FNO), is that they make assumptions about the periodicity of the input functions. While non-periodic output functions can still be recovered through skip-connections, non-periodicity of the input function will hinder training by introducing Gibbs phenomena. This can be mitigated by periodically extending the input function beyond its original domain, e.g., based on Fourier continuation methods (Maust et al., 2022). Even simple constant padding is often sufficient since the initial lifting layer can mitigate the discontinuities introduced by the padding. Since the input functions are now defined on an extended domain, it is important that the amount of padding changes according to the resolution of the discretization.

A.7 Attention Mechanism

Loosely inspired by human cognition, attention mechanisms assign varying levels of importance to different parts of the input, which are often referred to as *tokens*. In our setting, the tokens are given by point evaluations $\mathbf{f}_i = f(x_i)$ or, after positional encoding, $(x_i, f(x_i))$. Various attention mechanisms have been developed (Bahdanau et al., 2014; Luong et al., 2015; Veličković et al., 2017; Vaswani et al., 2017). For instance, the popular *soft(arg)max with dot-product attention* is given by

$$\mathbf{g}_j = \sum_{i=1}^n \frac{\exp(\tau \langle k(\mathbf{f}_i), q(\mathbf{f}_j) \rangle)}{\sum_{\ell=1}^n \exp(\tau \langle k(\mathbf{f}_\ell), q(\mathbf{f}_j) \rangle)} v(\mathbf{f}_i), \quad (32)$$

where v , k , and q are learnable *query*, *key*, and *value* functions. The inner product in (32) is rescaled by a so-called temperature τ , often chosen as $\tau = (d_{\text{att}})^{-1/2}$, where d_{att} is the dimension of the codomain of k and q . Moreover, multiple attention mechanisms, so-called *heads*, can be combined (Vaswani et al., 2017).

We speak of *self-attention* when evaluating the query and key functions at the same tokens (as in (32)). In the more general case of *cross-attention*, we evaluate the query function q in (32) on different tokens. In our neural operator layer in (13), this can be another function \tilde{f} , potentially discretized on a different point cloud $(\tilde{x}_i)_{i=1}^{\tilde{n}}$. Since the point cloud \tilde{x}_i defines the coordinates on which we can query the output function g of the attention layer, we can also use cross-attention to evaluate the layer on arbitrary coordinates by picking \tilde{f} to be a prescribed function, e.g., the identity or a positional embedding.

²⁸Note that in practice, the discrete Fourier transform can be more efficient than the FFT for sufficiently small numbers of Fourier modes (Lingsch et al., 2023).

A.8 Special Cases of Encoders & Decoders

In this section, we show that certain integral operators (see Section 3.4), including spectral convolutions (see Section 3.3), can be viewed as special cases of encoder-decoder layers introduced in Section 3.6. Specifically, we consider encoder-decoder layer of the form

$$\mathbf{v}_j = \int_{D_f} [b_j(x)]^* f(x) dx, \quad \mathbf{w}_j = \mathbf{K}_j \mathbf{v}_j, \quad \text{and} \quad g(y) = \sum_{j=1}^k \beta_j(y) \mathbf{w}_j, \quad (33)$$

with input function $f: D_f \rightarrow \mathbb{R}^c$ and output function $g: D_g \rightarrow \mathbb{R}^C$. This can be interpreted as an encoder based on inner products with functions b_j , linear maps \mathbf{K}_j in the latent space, and a decoder based on linear combinations with functions β_j . Thus, the overall layer can be expressed as

$$g(y) = \int_{D_f} \underbrace{\sum_{j=1}^k \beta_j(y) \mathbf{K}_j [b_j(x)]^*}_{:=K(x,y)} f(x) dx, \quad (34)$$

which corresponds to a version of the integral operator in (4) and (11). We also recover the following special cases:

- We obtain a layer of the *low-rank neural operator* (Kovachki et al., 2023; Lanthaler et al., 2023) when setting $\mathbf{K}_j = 1$ and learning $b_j: D_f \rightarrow \mathbb{R}^c$ and $\beta_j: D_g \rightarrow \mathbb{R}^C$.
- The spectral convolution in Section 3.3 is given by setting $b_{j+1}(x) = \beta_{j+1}(x) = e^{2\pi i j x}$, i.e., using the first Fourier basis functions, and learning $\mathbf{K}_j \in \mathbb{C}^{C \times c}$, where we assume that the input function is defined on the torus.

B Error Analysis for Zero-Shot Predictions on Different Discretizations

In this section we provide a high-level error analysis to outline the interplay between different errors, i.e., discretization, optimization, and approximation errors. For a detailed analysis and corresponding bounds, we refer, e.g., to Kovachki et al. (2021); Lanthaler et al. (2022); De Ryck & Mishra (2022); Lanthaler et al. (2023); Kovachki et al. (2023) for approximation and generalization results, to Kovachki et al. (2024a); Grohs et al. (2025) for results on sampling complexities, and to Lanthaler et al. (2024) for results on the discretization error.

Here, we consider the use of neural operators to predict on discretizations different than the training discretization, without retraining. In particular, this includes zero-shot super-resolution where the query discretization has a higher resolution than the training discretization. To this effect, let

- $\text{NO}^*: \mathcal{F} \rightarrow \mathcal{G}$ be the ground truth operator between Banach spaces \mathcal{G} and \mathcal{F} ,
- $\text{NO}_\theta: \mathcal{F} \rightarrow \mathcal{G}$ be a neural operator with parameter set $\theta \in \Theta$,
- $\widehat{\text{NO}}_\theta$ denote an empirical version of the neural operator NO_θ as in (25),
- X and \tilde{X} be fixed discretizations of the domain (the training and query discretizations, respectively).

Suppose we trained the neural operator on functions in \mathcal{F} discretized on X , and obtained learned parameters $\hat{\theta} \in \Theta$. We are interested in bounding the error of the neural operator on a function $f \in \mathcal{F}$ discretized on the (unseen) query discretization \tilde{X} , i.e.,

$$\varepsilon(f, \tilde{X}) := \|\text{NO}^*(f) - \widehat{\text{NO}}_{\hat{\theta}}(f|_{\tilde{X}})\|_{\mathcal{G}}. \quad (35)$$

Using the triangle inequality, we obtain

$$\varepsilon(f, \tilde{X}) \leq \underbrace{\|\text{NO}^*(f) - \widehat{\text{NO}}_{\hat{\theta}}(f|_X)\|_{\mathcal{G}}}_{=\varepsilon(f, X)} + \underbrace{\|\widehat{\text{NO}}_{\hat{\theta}}(f|_X) - \text{NO}_{\hat{\theta}}(f)\|_{\mathcal{G}}}_{\leq \varepsilon_{\text{discr}}(f, X)} + \underbrace{\|\text{NO}_{\hat{\theta}}(f) - \widehat{\text{NO}}_{\hat{\theta}}(f|_{\tilde{X}})\|_{\mathcal{G}}}_{\leq \varepsilon_{\text{discr}}(f, \tilde{X})}, \quad (36)$$

where $\varepsilon(f, X)$ is the error of the neural operator on the discretization X defined as in (35), $\varepsilon_{\text{discr}}(f, X)$ is a bound on the *discretization error* of NO on X , i.e.,

$$\sup_{\theta \in \Theta} \|\text{NO}_\theta(f) - \widehat{\text{NO}}_\theta(f|_X)\|_{\mathcal{G}} \leq \varepsilon_{\text{discr}}(f, X), \quad (37)$$

and $\varepsilon_{\text{discr}}(f, \tilde{X})$ is defined analogously. Under suitable conditions for discretization convergence (see Appendix A.4), the discretization errors $\varepsilon_{\text{discr}}$ go to zero as the resolutions of the discretizations increase. In this case, for high resolutions, the inequality in (36) shows that the error for the discretization \tilde{X} is essentially bounded by the error for the discretization X .

Next, we want to bound the error on the training discretization $\varepsilon(f, X)$ in (36). To this end, let $\theta^* \in \Theta$ be a parameter that achieves minimal *approximation error* to the ground truth operator, i.e.,

$$\sup_{f \in \mathcal{F}} \|\text{NO}^*(f) - \text{NO}_{\theta^*}(f)\|_{\mathcal{G}} = \inf_{\theta \in \Theta} \sup_{f \in \mathcal{F}} \|\text{NO}^*(f) - \text{NO}_\theta(f)\|_{\mathcal{G}} = \varepsilon_{\text{approx}}. \quad (38)$$

Moreover, suppose the *optimization error*²⁹ between the trained and optimal neural operator can be bounded by some ε_{opt} , i.e.

$$\sup_{f \in \mathcal{F}} \|\text{NO}_{\theta^*}(f) - \text{NO}_{\hat{\theta}}(f)\|_{\mathcal{G}} \leq \varepsilon_{\text{opt}}. \quad (39)$$

Then we can use the triangle inequality to bound the model error on the training discretization as

$$\varepsilon(f, X) \leq \underbrace{\|\text{NO}^*(f) - \text{NO}_{\theta^*}(f)\|_{\mathcal{G}}}_{\leq \varepsilon_{\text{approx}}} + \underbrace{\|\text{NO}_{\theta^*}(f) - \text{NO}_{\hat{\theta}}(f)\|_{\mathcal{G}}}_{\leq \varepsilon_{\text{opt}}} + \underbrace{\|\text{NO}_{\hat{\theta}}(f) - \widehat{\text{NO}}_{\hat{\theta}}(f|_X)\|_{\mathcal{G}}}_{\leq \varepsilon_{\text{discr}}}. \quad (40)$$

In particular, substituting this inequality in (36), implies that

$$\varepsilon(f, \tilde{X}) \leq \varepsilon_{\text{approx}} + \varepsilon_{\text{opt}} + 2\varepsilon_{\text{discr}}(f, X) + \varepsilon_{\text{discr}}(f, \tilde{X}). \quad (41)$$

This shows that we can bound the error $\varepsilon(f, \tilde{X})$ for a function discretized on \tilde{X} , by the approximation error in (38), the optimization error in (39), and the discretization errors for X and \tilde{X} as in (37). Due to the resolution convergence of the neural operator, the discretization errors vanish as the resolution increases. The prediction error on discretizations with higher resolution than the training data can thus be made arbitrarily small by having a *sufficiently expressive* neural operator that is trained on a *sufficiently high resolution* to *near-optimality*. In particular, this provides a theoretical explanation for the super-resolution capabilities of neural operators that approximate the underlying ground truth operator.

C Experimental Details

In this section, we describe the dataset as well as implementation and hyperparameter details for our experiments. As mentioned in Section 4.2, the general training procedure for neural operators is similar to that of neural networks (see Algorithm 5).

C.1 Navier-Stokes Equations

Since the primary applications of neural operators have been in learning the solution operators for PDEs, the experiments in this paper focus on one well-studied PDE: the Navier-Stokes (NS) equations for incompressible fluids. As in Kossai et al. (2023), we consider the vorticity-stream $(\omega - \psi)$ formulation of the 2-dimensional NS equations, given by

$$\begin{aligned} \partial_t \omega + \nabla^\perp \psi \cdot \nabla \omega &= \frac{1}{\text{Re}} \Delta \omega + f, & \text{on } \mathbb{T}^2 \times (0, T] \\ -\Delta \psi &= \omega, \quad \int_{\mathbb{T}^2} \psi = 0 & \text{on } \mathbb{T}^2 \times (0, T]. \end{aligned} \quad (42)$$

In the above, ω and ψ denote the vorticity and stream functions defined on the torus \mathbb{T}^2 and the time interval $[0, T]$, f is a forcing function distributed according to a Gaussian measure, and $\text{Re} > 0$ is the Reynolds

²⁹For simplicity, we do not explicitly consider the *generalization error* arising from training only on a finite number of samples.

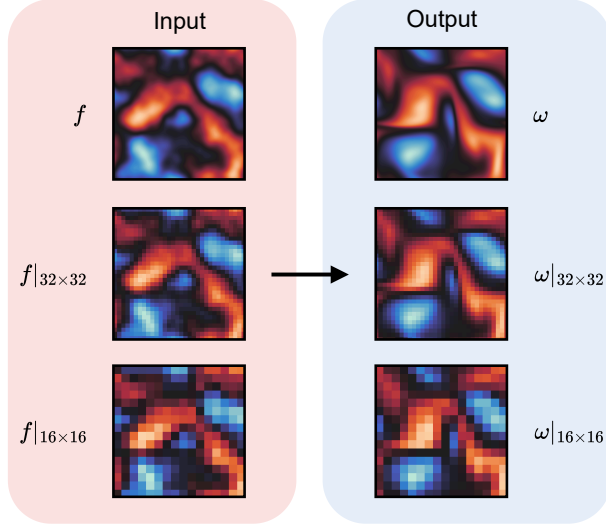


Figure 6: Sample input and output functions and two discretizations for learning the mapping from forcing function to vorticity governed by the Navier-Stokes equations in Appendix C.1.

number, which can be used to characterize the laminar or turbulent nature of the flow. As in Kossaifi et al. (2023), we fix the initial vorticity to be $\omega(\cdot, 0) = 0$ and seek to learn the nonlinear operator $f \mapsto \omega(\cdot, T)$ from the forcing function to the vorticity at some fixed time $T > 0$; see also Figure 6. Following Kossaifi et al. (2023), we use 10,000 samples for training and 2,000 samples for testing.

C.2 Implementation Details

In order to maintain the fairness of comparisons, we attempted to maintain an equal number of parameters (around 63M) across all models throughout our experiments (if allowed by GPU memory constraints). Furthermore, all models were trained until convergence using the Adam optimizer (Kingma & Ba, 2014) with a weight decay of 10^{-4} , except the OFORMER which used a weight decay of 10^{-6} . Training samples are given on a uniform spatial discretization of resolution 128×128 . Where GPU memory allowed, we evaluated our models at several data resolutions, as shown in Figure 5. All models are trained on absolute L^2 -error and evaluated using relative L^2 -error. When available, we use the implementations found in the NEURALOPERATOR Python library³⁰ (Kossaifi et al., 2024). All models were trained on a single NVIDIA RTX 4090 GPU, with the exception of the OFORMER, which was trained on a single NVIDIA H100 GPU due to higher memory requirements.

Note that for some models, multiple resolutions were used during training. In these cases, we use each of the 10,000 samples at each resolution in each epoch. The specific training resolutions used for these particular models are described below, along with the implementation details for all models used in our experiments.

U-Net Our U-NET model is adopted from the PDEARENA (Gupta & Brandstetter, 2022) baseline³¹. In our experiments, we set the number of hidden channels to be 19, the kernel size to be 9, and use layer normalization. We use channel multipliers of 1, 2, 2, 3 for each resolution; see Gupta & Brandstetter (2022) and the official repository for more details. We halve the learning rate every 50 epochs, starting at an initial learning rate of $5 \cdot 10^{-4}$.

U-Net with mixed resolution training In these experiments, we use the same U-NET architecture as above; however, we augment the training data by including the samples at three different resolutions. In addition to resolution 128×128 , we supplement the training set with the same data at resolutions 64×64 and 256×256 . We halve the learning rate every 20 epochs, starting at an initial learning rate of $5 \cdot 10^{-4}$.

³⁰The NEURALOPERATOR library is publicly available at github.com/neuraloperator/neuraloperator. Our code is available at github.com/neuraloperator/NNS-to-NOs.

³¹The PDEARENA code is publicly available at github.com/pdearena/pdearena.

U-Net with input & output interpolation U-NET with input and output interpolation fixes the U-NET resolution at 128×128 and interpolates the input and output to the desired resolution, as described in Section 3.7. For these experiments, we use the same U-NET model as described above. We use bilinear interpolation in our experiments.

FNO In our experiments, we set the number of Fourier modes to 64, the number of hidden channels to 62, and we use 4 Fourier layers consisting of spectral convolutions, pointwise MLPs, and soft-gating skip-connections as proposed by Kossaifi et al. (2023). We halve the learning rate every 33 epochs, starting at an initial learning rate of $5 \cdot 10^{-4}$.

Local neural operator In our experiments, we follow Liu-Schiaffini et al. (2024) in supplementing each spectral convolution with either local integral or differential operators. For both experiments, we use 40 Fourier modes and 99 channels, such that the total number of parameters is similar to the FNO experiments. Decreasing the number of Fourier modes encourages the FNO to learn global features, and increasing the number of channels increases the expressivity of the local operators for capturing high-frequency features. Apart from these changes, we follow the same settings as for the FNO architecture described above. For FNO + DIFFCONV, we only supplement the last layer with a differential layer, whereas for FNO + LOCALCONV, we supplement all four layers with a local integral layer. For FNO + LOCALCONV, we use the same basis as in Liu-Schiaffini et al. (2024) with a radius cutoff of 0.03125.

Vision transformer For the vision transformer (ViT) experiments, we adapt the model from Dosovitskiy et al. (2020), using the implementation in the HUGGINGFACE TRANSFORMERS library³². We use a patch size of 16×16 , a hidden size of 768, 12 hidden layers, 12 attention heads, an intermediate size of 1900, and GeLU activations. We halve the learning rate every 50 epochs, starting at an initial learning rate of $5 \cdot 10^{-4}$.

OFormer In our experiments, we adapt the OFORMER architecture from Li et al. (2022b). However, we introduce two modifications that helped for high-resolution uniform grids:

1. Since the input and output grids are the same, we removed the decoder, resulting in an encoder-only architecture.
2. To better capture low frequencies within each encoder layer, we summed the output of the attention block and the residual connection with the output of a spectral convolution (Li et al., 2020a) applied on the input, normalized by $\sqrt{3}$ for stability.

These two modifications significantly improved the results over the baseline OFORMER on this task. We use 220 hidden channels and 16 attention heads in the encoder. We used 4 encoder layers and 16 modes in the spectral convolutions. We halved the learning rate every 33 epochs, with an initial learning rate of 10^{-4} .

³²The TRANSFORMERS library is publicly available at github.com/huggingface/transformers.