Neural Operator: Learning Maps Between Function Spaces

Nikola Kovachki* NKOVACHKI@CALTECH.EDU *Caltech*

Zongyi Li* ZONGYILI@CALTECH.EDU *Caltech*

Burigede Liu BL377@CAM.AC.UK *Cambridge University*

Kamyar Azizzadenesheli KAMYAR@PURDUE.EDU *Purdue University*

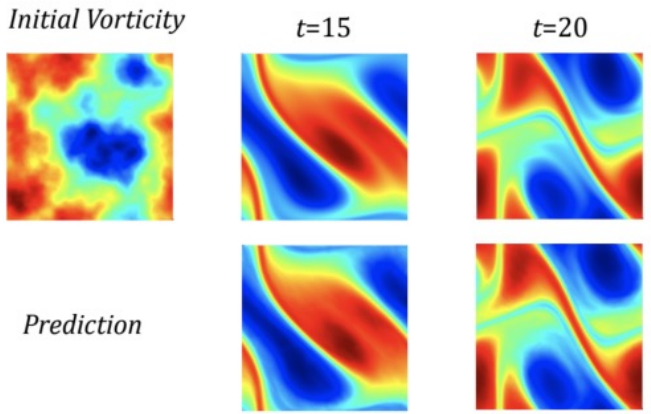
Kaushik Bhattacharya BHATTA@CALTECH.EDU *Caltech*

Andrew Stuart ASTUART@CALTECH.EDU *Caltech*

Anima Anandkumar ANIMA@CALTECH.EDU *Caltech*

Neural Operator

- Learn the maps between ***Functions to Functions***



Overview

- Crash introduction
- Background
- Settings
- Models
- Methods
  - Graph Neural Operator (GNO)
  - Low-rank Neural Operator (LNO)
  - Multipole Graph Neural Operator (MGNO)
  - Fourier Neural Operator
- Experiments

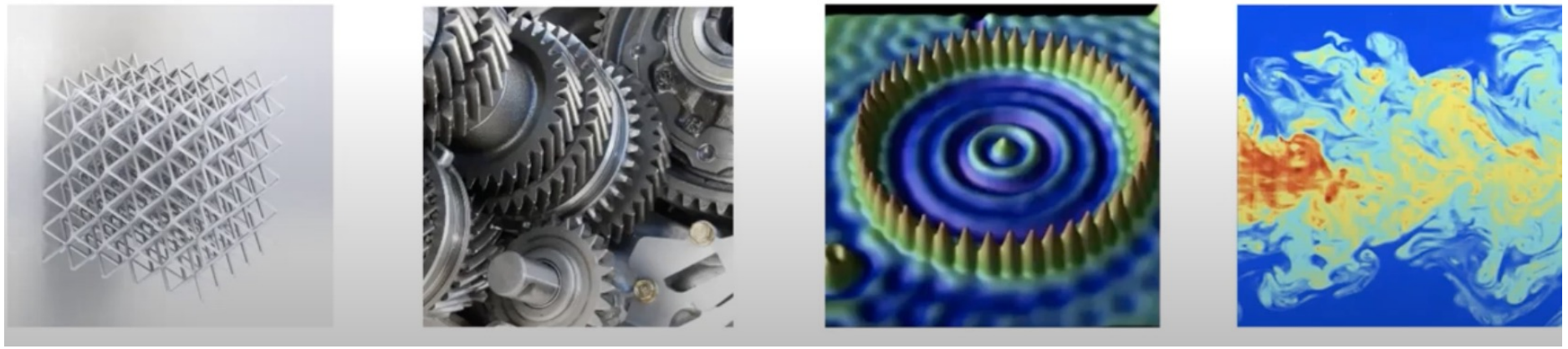
Crash Introduction

Crash Introduction

Crash Introduction

PDE as Foundation for Scientific Modeling

- **PDEs**: “Differential equations… represent the most powerful tool humanity has ever created for making sense of the matrial world.” (Strogatz 2009).
- Challenges:
  - Identify the governing model for complex systems
  - Efficiently solving large-scale non-linear systems of equations



PDE as Foundation for Scientific Modeling

- System identification:
  - Requires extensive prior knowledge in the corresponding field
  - Ex: modeling the deformation and failure of solid structure requires detailed knowledge of the relationship between stress and strain in the constituent material
- Solving complicated PDE
  - Ex: those arising from turbulence and plasticity are computational demanding and intractable
  - Numerical solvers vs. data driven solvers

Learning PDE Solution Operator

- In PDE applications:
  - Governing equations are by definition local
  - However, the solution operator exhibits non-local properties
  - Such non-local effects can be described by integral operators explicitly in the spatial domain

Learning PDE Solution Operator

Parametric PDE settings

Second order elliptic equation:

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in D$$
$$u(x) = 0, \quad x \in \partial D$$

Input: a(x)

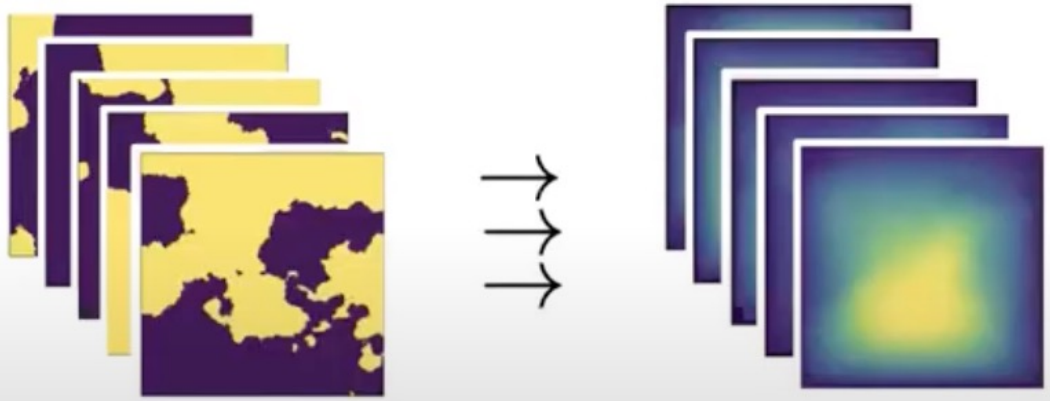
Output: u(x)

$$\mathcal{F} : \mathcal{A} \times \Theta \to \mathcal{U}$$

Solve vs. Learn

Solving PDEs is slow.
Learn the mapping from data (coefficients & solutions pairs).

- Fix an equation
- Multiple training instances
- Learn the mapping



Input: coefficients

Output: solutions

Slow to train. Fast to evaluate. $\mathcal{F} : \mathcal{A} \times \Theta \to \mathcal{U}$

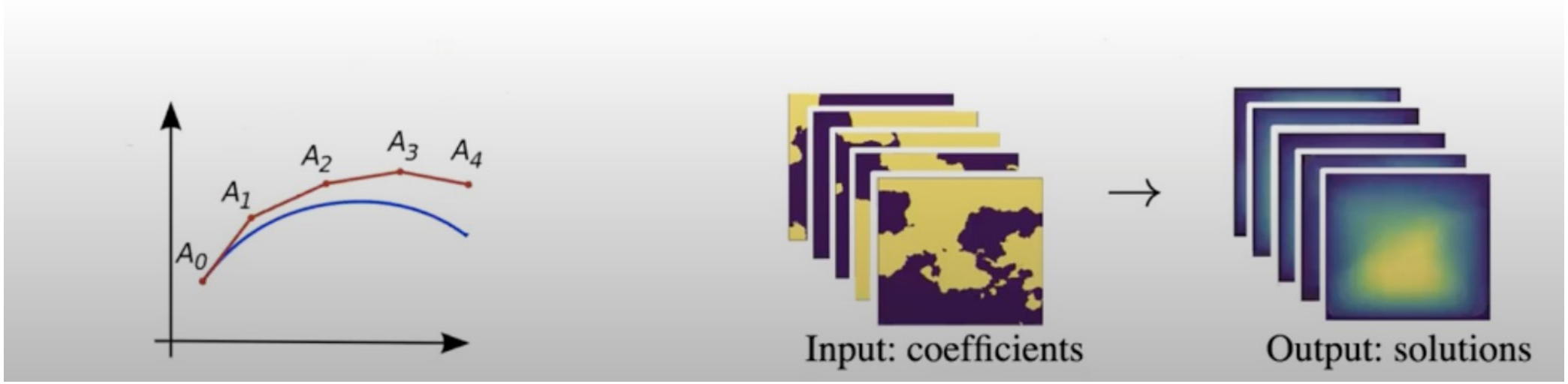
Solve vs. Learn

Conventional methods:

- Solve one instance
- Require the explicit form
- trade-off on resolution
- Slow on fine grids; fast on coarse grids

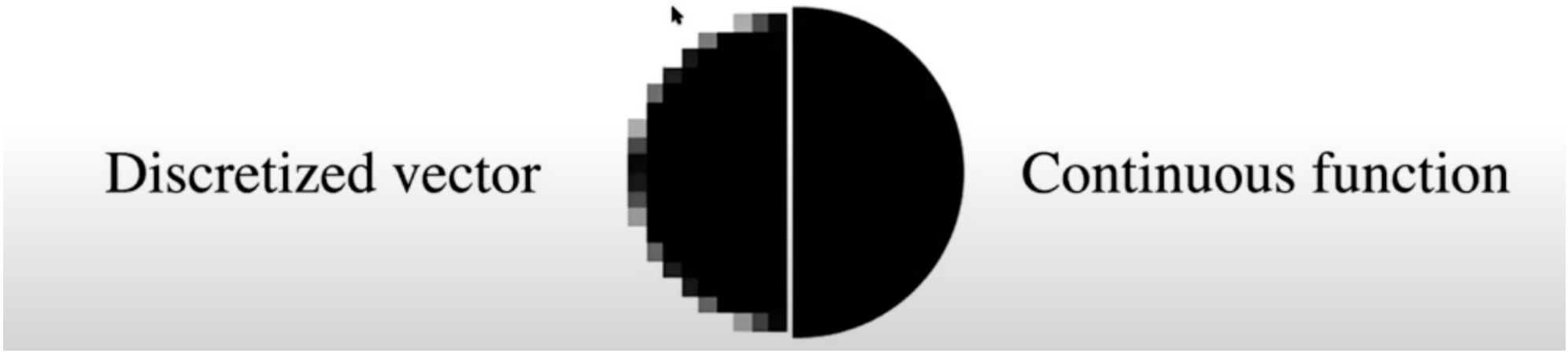
Data-driven methods:

- Learn a family of PDE
- Black-box, data-driven
- Resolution-invariant, mesh-invariant
- Slow to train; fast to evaluate



Operator Learning

- Not vector-to-vector mapping.
- But function-to-function mapping.



Key idea: represent function & operator in mesh-invariant way

Highlights of Neural Operators

- Map between infinite dimensional function spaces (vs. learning mappings between finite dimensional Euclidean spaces or finite sets.)
- Complex nonlinear operators by composing linear integral operators and non-linear functions
- Universal approximator
- Resolution-invariant (constant error for any discretization)
- Several order of magnitude faster than conventional PDE solvers

Problems Settings

Let $\mathcal{A}$ and $\mathcal{U}$ be Banach spaces of functions defined on bounded domains $D \subset \mathbb{R}^d$, $D' \subset \mathbb{R}^{d'}$ respectively and $\mathcal{G}^\dagger : \mathcal{A} \to \mathcal{U}$ be a (typically) non-linear map. Suppose we have observations $\{a_j, u_j\}_{j=1}^N$ where $a_j \sim \mu$ are i.i.d. samples drawn from some probability measure μ supported on $\mathcal{A}$ and $u_j = \mathcal{G}^\dagger(a_j)$ is possibly corrupted with noise. We aim to build an approximation of $\mathcal{G}^\dagger$ by constructing a parametric map

$$\mathcal{G}_\theta : \mathcal{A} \to \mathcal{U}, \quad \theta \in \mathbb{R}^p \tag{2}$$

In particular, assuming $\mathcal{G}^\dagger$ is μ-measurable, we will aim to control the $L^2_\mu(\mathcal{A};\mathcal{U})$ Bochner norm of the approximation

$$\|\mathcal{G}^\dagger - \mathcal{G}_\theta\|^2_{L^2_\mu(\mathcal{A};\mathcal{U})} = \mathbb{E}_{a\sim\mu}\|\mathcal{G}^\dagger(a) - \mathcal{G}_\theta(a)\|^2_{\mathcal{U}} = \int_{\mathcal{A}} \|\mathcal{G}^\dagger(a) - \mathcal{G}_\theta(a)\|^2_{\mathcal{U}}\, d\mu(a). \tag{3}$$

This is a natural framework for learning in infinite-dimensions as one could seek to solve the associated empirical-risk minimization problem

$$\min_{\theta\in\mathbb{R}^p} \mathbb{E}_{a\sim\mu}\|\mathcal{G}^\dagger(a) - \mathcal{G}_\theta(a)\|^2_{\mathcal{U}} \approx \min_{\theta\in\mathbb{R}^p} \frac{1}{N}\sum_{j=1}^N \|u_j - \mathcal{G}_\theta(a_j)\|^2_{\mathcal{U}} \tag{4}$$

Intuition: Green's Function For Linear PDE

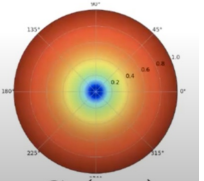
$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in D$$
$$u(x) = 0, \quad x \in \partial D$$

Inverse of differential operator can be written in form of kernel

$$u(x) = \int_D G_a(x,y) f(y)\, dy.$$

Where G is the green function

$$u(x) = \int_D G_a(x,y)[f(y) + (\Gamma_a u)(y)]\, dy.$$



$G_a(x,y)$

The elliptic operator $\mathcal{L}_a \cdot = -\text{div}(a\nabla\cdot)$ from equation (3) is an example. Under fairly general conditions on $\mathcal{L}_a$ [Evans, 2010], we may define the Green's function $G : D \times D \to \mathbb{R}$ as the unique solution to the problem

$$\mathcal{L}_a G(x,\cdot) = \delta_x$$

where δ_x is the delta measure on $\mathbb{R}^d$ centered at x. Note that G will depend on the parameter a thus we will henceforth denote it as G_a. The solution to (4) can then be represented as

$$u(x) = \int_D G_a(x,y) f(y)\, dy. \tag{5}$$

This is easily seen via the formal computation

$$\begin{aligned}(\mathcal{L}_a u)(x) &= \int_D (\mathcal{L}_a G(x,\cdot))(y) f(y)\, dy \\ &= \int_D \delta_x(y) f(y)\, dy \\ &= f(x)\end{aligned}$$

Iterative Solver for General PDE: Stack Layers

Add iterations for $l = 1,\ldots,L$

Let $v_0=f$ (input) and $v_L=u$ (output)
Add the bias term W and activation function σ

$$v_{l+1}(x) = \sigma\left(W v_l(x) + \int_D \kappa\big(x, y, a(x), a(y)\big) v_l(y) \mathrm{d}y\right)$$

Neural Operators

$$u = Q\left(K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0\right) P\, v$$

In this section, we outline the neural operator framework. We assume that the input functions $a \in \mathcal{A}$ are $\mathbb{R}^{d_a}$-valued and defined on the bounded domain $D \subset \mathbb{R}^d$ while the output functions $u \in \mathcal{U}$ are $\mathbb{R}^{d_u}$-valued and defined on the bounded domain $D' \subset \mathbb{R}^{d'}$. The proposed architecture $\mathcal{G}_\theta : \mathcal{A} \to \mathcal{U}$ has the following overall structure:

1. **Lifting**: Using a pointwise function $\mathbb{R}^{d_a} \to \mathbb{R}^{d_{v_0}}$, map the input $\{a : D \to \mathbb{R}^{d_a}\} \mapsto \{v_0 : D \to \mathbb{R}^{d_{v_0}}\}$ to its first hidden representation. Usually, we choose $d_{v_0} > d_a$ and hence this is a lifting operation performed by a fully local operator.

2. **Iterative Kernel Integration**: For $t = 0, \ldots, T-1$, map each hidden representation to the next $\{v_t : D_t \to \mathbb{R}^{d_{v_t}}\} \mapsto \{v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}\}$ via the action of the sum of a local linear operator, a non-local integral kernel operator, and a bias function, composing the sum with a fixed, pointwise nonlinearity. Here we set $D_0 = D$ and $D_T = D'$ and impose that $D_t \subset \mathbb{R}^{d_t}$ is a bounded domain.

3. **Projection**: Using a pointwise function $\mathbb{R}^{d_{v_T}} \to \mathbb{R}^{d_u}$, map the last hidden representation $\{v_T : D' \to \mathbb{R}^{d_{v_T}}\} \mapsto \{u : D' \to \mathbb{R}^{d_u}\}$ to the output function. Analogously to the first step, we usually pick $d_{v_T} > d_u$ and hence this is a projection step performed by a fully local operator.

Neural Operators

The outlined structure mimics that of a finite dimensional neural network where hidden representations are successively mapped to produce the final output. In particular, we have

$$\mathcal{G}_\theta := \mathcal{Q} \circ \sigma_T(W_{T-1} + \mathcal{K}_{T-1} + b_{T-1}) \circ \cdots \circ \sigma_1(W_0 + \mathcal{K}_0 + b_0) \circ \mathcal{P} \tag{5}$$

where $\mathcal{P} : \mathbb{R}^{d_a} \to \mathbb{R}^{d_{v_0}}$, $\mathcal{Q} : \mathbb{R}^{d_{v_T}} \to \mathbb{R}^{d_u}$ are the local lifting and projection mappings respectively, $W_t \in \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}}$ are local linear operators (matrices), $\mathcal{K}_t : \{v_t : D_t \to \mathbb{R}^{d_{v_t}}\} \to \{v_{t+1} : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}\}$ are integral kernel operators, $b_t : D_{t+1} \to \mathbb{R}^{d_{v_{t+1}}}$ are bias functions, and σ_t are fixed activation functions acting locally as maps $\mathbb{R}^{v_{t+1}} \to \mathbb{R}^{v_{t+1}}$ in each layer. The output dimensions $d_{v_0}, \ldots, d_{v_T}$ as well as the input dimensions $d_1, \ldots, d_{T-1}$ and domains of definition $D_1, \ldots, D_{T-1}$ are hyperparameters of the architecture.

Local vs Global Maps

By local maps, we mean that the action is pointwise, in particular, for the lifting and projection maps, we have $(\mathcal{P}(a))(x) = \mathcal{P}(a(x))$ for any $x \in D$ and $(\mathcal{Q}(v_T))(x) = \mathcal{Q}(v_T(x))$ for any $x \in D'$ and similarly, for the activation, $(\sigma(v_{t+1}))(x) = \sigma(v_{t+1}(x))$ for any $x \in D_{t+1}$. The maps, $\mathcal{P}$, $\mathcal{Q}$, and σ_t can thus be thought of as defining Nemitskiy operators (Dudley and Norvaisa, 2011, Chapters 6,7) when each of their components are assumed to be Borel measurable.

$$u = Q\left(K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0\right) P\, v$$

Integral Kernel Operators

$\kappa^{(t)} \in C(D_{t+1} \times D_t; \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}})$ and let ν_t be a Borel measure on D_t

$$(\mathcal{K}_t(v_t))(x) = \int_{D_t} \kappa^{(t)}(x, y) v_t(y) \, \mathrm{d}\nu_t(y) \qquad \forall x \in D_{t+1}.$$

For the second, let $\kappa^{(t)} \in C(D_{t+1} \times D_t \times \mathbb{R}^{d_a} \times \mathbb{R}^{d_a}; \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}})$. Then we define $\mathcal{K}_t$ by

$$(\mathcal{K}_t(v_t))(x) = \int_{D_t} \kappa^{(t)}(x, y, a(\Pi^D_{t+1}(x)), a(\Pi^D_t(y))) v_t(y) \, \mathrm{d}\nu_t(y) \qquad \forall x \in D_{t+1}.$$

, let $\kappa^{(t)} \in C(D_{t+1} \times D_t \times \mathbb{R}^{d_{v_t}} \times \mathbb{R}^{d_{v_t}}; \mathbb{R}^{d_{v_{t+1}} \times d_{v_t}})$. Then we define $\mathcal{K}_t$ by

$$(\mathcal{K}_t(v_t))(x) = \int_{D_t} \kappa^{(t)}(x, y, v_t(\Pi_t(x)), v_t(y)) v_t(y) \, \mathrm{d}\nu_t(y) \qquad \forall x \in D_{t+1}.$$

Example of Single Hidden Layer Construction

$$v_{t+1}(x) = \sigma_{t+1}\left(W_t v_t(\Pi_t(x)) + \int_{D_t} \kappa^{(t)}(x,y) v_t(y)\, \mathrm{d}\nu_t(y) + b_t(x)\right) \qquad \forall x \in D_{t+1} \qquad (9)$$

where $\Pi_t : D_{t+1} \to D_t$ are fixed mappings. We remark that, since we often consider functions on the same domain, we usually take Π_t to be the identity.

$$(\mathcal{G}_\theta(a))(x) = \mathcal{Q}\left(\int_D \kappa^{(1)}(x,y)\sigma\left(W_0\mathcal{P}(a(y)) + \int_D \kappa^{(0)}(y,z)\mathcal{P}(a(z))\,\mathrm{d}z + b_0(y)\right)\,\mathrm{d}y\right) \qquad (10)$$

In this example, $\mathcal{P} \in C(\mathbb{R}^{d_a}; \mathbb{R}^{d_{v_0}})$, $\kappa^{(0)} \in C(D \times D; \mathbb{R}^{d_{v_1} \times d_{v_0}})$, $b_0 \in C(D; \mathbb{R}^{d_{v_1}})$

$W_0 \in \mathbb{R}^{d_{v_1} \times d_{v_0}}$, $\kappa^{(0)} \in C(D' \times D; \mathbb{R}^{d_{v_2} \times d_{v_1}})$, and $\mathcal{Q} \in C(\mathbb{R}^{d_{v_2}}; \mathbb{R}^{d_u})$

Miscellaneous

- DeepONets are Neural Operators
- Transformers are Neural Operators
- Universal Approximator
  - Approximate errors is zero when the number of parameters to infinity
  - The errors is not related to the degree of discretization

Parameterization and Computation

To simplify notation, we will only consider a single layer of (5) i.e. (9) and choose the input and output domains to be the same. Furthermore, we will drop the layer index t and write the single layer update as

$$u(x) = \sigma\left(Wv(x) + \int_D \kappa(x,y)v(y)\,\mathrm{d}\nu(y) + b(x)\right) \qquad \forall x \in D \tag{15}$$

where $D \subset \mathbb{R}^d$ is a bounded domain, $v : D \to \mathbb{R}^n$ is the input function and $u : D \to \mathbb{R}^m$ is the output function. When the domain domains D of v and u are different, we will usually extend them to be on a larger domain. We will consider σ to be fixed, and, for the time being, take $\mathrm{d}\nu(y) = \mathrm{d}y$ to be the Lebesgue measure on $\mathbb{R}^d$. Equation (15) then leaves three objects which can be parameterized: W, κ, and b. Since W is linear and acts only locally on v, we will always parametrize it by the values of its associated $m \times n$ matrix; hence $W \in \mathbb{R}^{m\times n}$ yielding mn parameters. We have found empirically that letting $b : D \to \mathbb{R}^m$ be a constant function over any domain D works at least as well as allowing it to be an arbitrary neural network. Perusal of

Parameterization and Computation

choosing the kernel function $\kappa : D \times D \to \mathbb{R}^{m \times n}$

$$u(x) = \int_D \kappa(x, y) v(y) \, \mathrm{d}\nu(y) \qquad \forall x \in D. \tag{16}$$

To demonstrate the computational challenges associated with (16), let $\{x_1, \ldots, x_J\} \subset D$ be a uniformly-sampled J-point discretization of D. Recall that we assumed $\mathrm{d}\nu(y) = \mathrm{d}y$ and, for simplicity, suppose that $\nu(D) = 1$, then the Monte Carlo approximation of (16) is

$$u(x_j) = \frac{1}{J} \sum_{l=1}^{J} \kappa(x_j, x_l) v(x_l), \qquad j = 1, \ldots, J.$$

Therefore to compute u on the entire grid requires $\mathcal{O}(J^2)$ matrix-vector multiplications. Each of these matrix-vector multiplications requires $\mathcal{O}(mn)$ operations;

Kernel Matrix. It will often times be useful to consider the kernel matrix associated to κ for the discrete points $\{x_1, \ldots, x_J\} \subset D$. We define the kernel matrix $K \in \mathbb{R}^{mJ \times nJ}$ to be the $J \times J$ block matrix with each block given by the value of the kernel i.e.

$$K_{jl} = \kappa(x_j, x_l) \in \mathbb{R}^{m \times n}, \qquad j, l = 1, \ldots, J$$

where we use (j, l) to index an individual block rather than a matrix element.

Graph Neural Operator (GNO)

Nyström Approximation. A simple yet effective method to alleviate the cost of computing (16) is employing a Nyström approximation. This amounts to sampling uniformly at random the points over which we compute the output function u. In particular, let $x_{k_1}, \dots, x_{k_{J'}} \subset \{x_1, \dots, x_J\}$ be $J' \ll J$ randomly selected points and, assuming $\nu(D) = 1$, approximate (16) by

$$u(x_{k_j}) \approx \frac{1}{J'} \sum_{l=1}^{J'} \kappa(x_{k_j}, x_{k_l}) v(x_{k_l}), \qquad j = 1, \dots, J'.$$

We can view this as a low-rank approximation to the kernel matrix K, in particular,

$$K \approx K_{JJ'} K_{J'J'} K_{J'J} \tag{17}$$

where $K_{J'J'}$ is a $J' \times J'$ block matrix and $K_{JJ'}$, $K_{J'J}$ are interpolation matrices, for example, linearly extending the function to the whole domain from the random nodal points. The complexity of this computation is $\mathcal{O}(J'^2)$ hence it remains quadratic but only in the number of subsampled points J' which we assume is much less than the number of points J in the original discretization.

Graph Neural Operator (GNO)

Truncation. Another simple method to alleviate the cost of computing (16) is to truncate the integral to a sub-domain of D which depends on the point of evaluation $x \in D$. Let $s : D \to \mathcal{B}(D)$ be a mapping of the points of D to the Lebesgue measurable subsets of D denoted $\mathcal{B}(D)$. Define $\mathrm{d}\nu(x, y) = \mathbb{1}_{s(x)}\mathrm{d}y$ then (16) becomes

$$u(x) = \int_{s(x)} \kappa(x, y)v(y)\,\mathrm{d}y \qquad \forall x \in D. \tag{18}$$

If the size of each set $s(x)$ is smaller than D then the cost of computing (18) is $\mathcal{O}(c_s J^2)$ where $c_s < 1$ is a constant depending on s. While the cost remains quadratic in J, the constant c_s can have a significant effect in practical computations, as we demonstrate in Section 7. For simplicity and ease of implementation, we only consider $s(x) = B(x, r) \cap D$ where $B(x, r) = \{y \in \mathbb{R}^d : \|y - x\|_{\mathbb{R}^d} < r\}$ for some fixed $r > 0$. With this choice of s and assuming that $D = [0, 1]^d$, we can explicitly calculate that $c_s \approx r^d$.

Graph Neural Operator (GNO)

Graph Neural Networks

To do the integration, we again need some discretization.
Assuming a uniform distribution of y, the integral $\int_{B(x,r)} \kappa_\phi(x, y, a(x), a(y))v_t(y)\,\mathrm{d}y$ can be approximated by a sum:

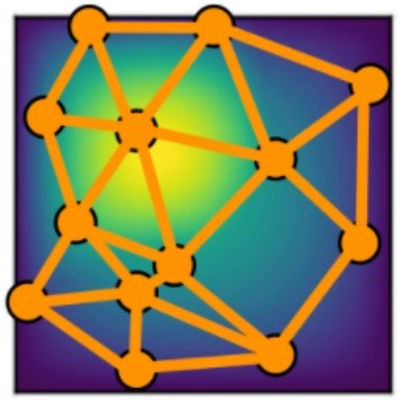
$$\frac{1}{|N|}\sum_{y\in N(x)} \kappa(x, y, a(x), a(y))v_t(y)$$

> *Observation: the kernel integral is equivalent to the message passing on graphs*

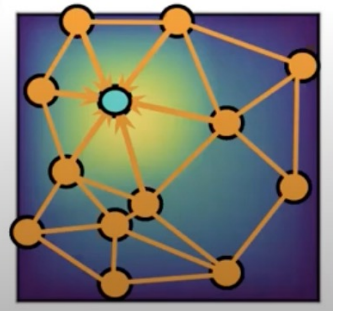
Given node features $v_t(x) \in \mathbb{R}^n$, edge features $e(x, y) \in \mathbb{R}^{n_e}$, and a graph G, the message passing neural network with averaging aggregation is

$$v_{t+1}(x) = \sigma\Big(Wv_t(x) + \frac{1}{|N(x)|}\sum_{y\in N(x)} \kappa_\phi\big(e(x, y)\big)v_t(y)\Big)$$

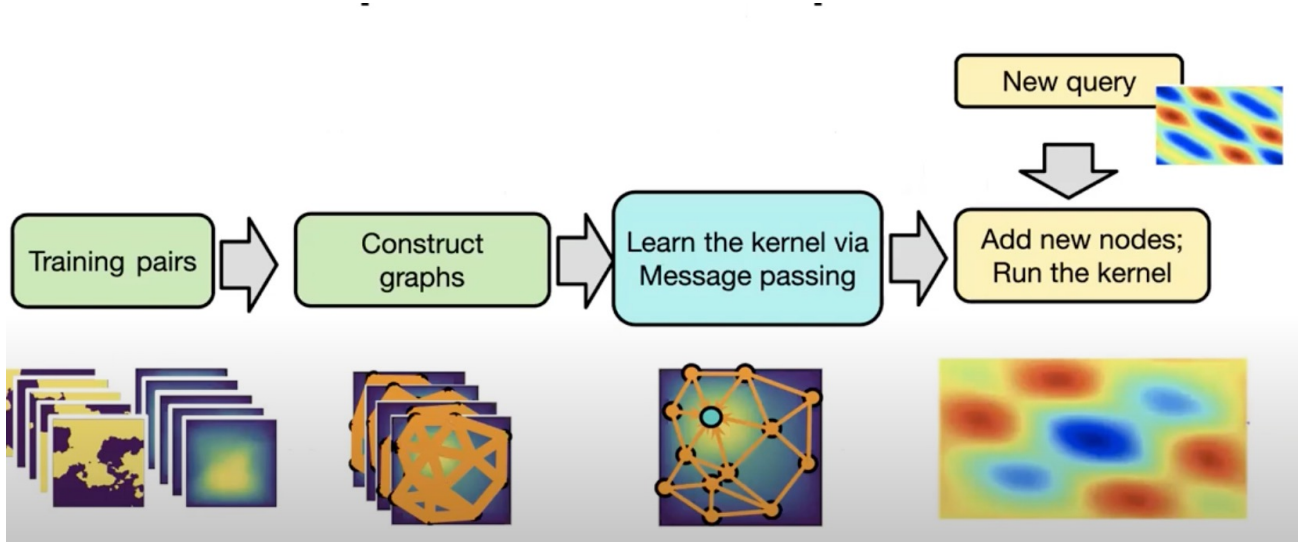
where $W \in \mathbb{R}^{n\times n}$, $N(x)$ is the neighborhood of x according to the graph, $\kappa_\phi\big(e(x, y)\big)$ is a neural network taking as input edge features and as output a matrix in $\mathbb{R}^{n\times n}$. Relating to our kernel formulation, $e(x, y) = (x, y, a(x), a(y))$.



- Adjacency matrix = kernel matrix.
- Kernel integration = message passing



Graph Neural Operator (GNO)



Training:

- for each training pair (a, u), sample several random graphs.
- Learn a universal kernel.

Testing:

- To evaluate at a specific location, simply add a node at this location.
- No interpolation needed.

Low-rank Neural Operator (LNO)

We start by assuming $\kappa : D \times D \to \mathbb{R}$ is scalar valued and later generalize to the vector valued setting. We express the kernel as

$$\kappa(x, y) = \sum_{j=1}^{r} \varphi^{(j)}(x)\psi^{(j)}(y) \qquad \forall x, y \in D$$

for some functions $\varphi^{(1)}, \psi^{(1)}, \ldots, \varphi^{(r)}, \psi^{(r)} : D \to \mathbb{R}$ that are normally given as the components of two neural networks $\varphi, \psi : D \to \mathbb{R}^r$ or a single neural network $\Xi : D \to \mathbb{R}^{2r}$ which couples all functions through its parameters. With this definition, and supposing that $n = m = 1$, we have that (16) becomes

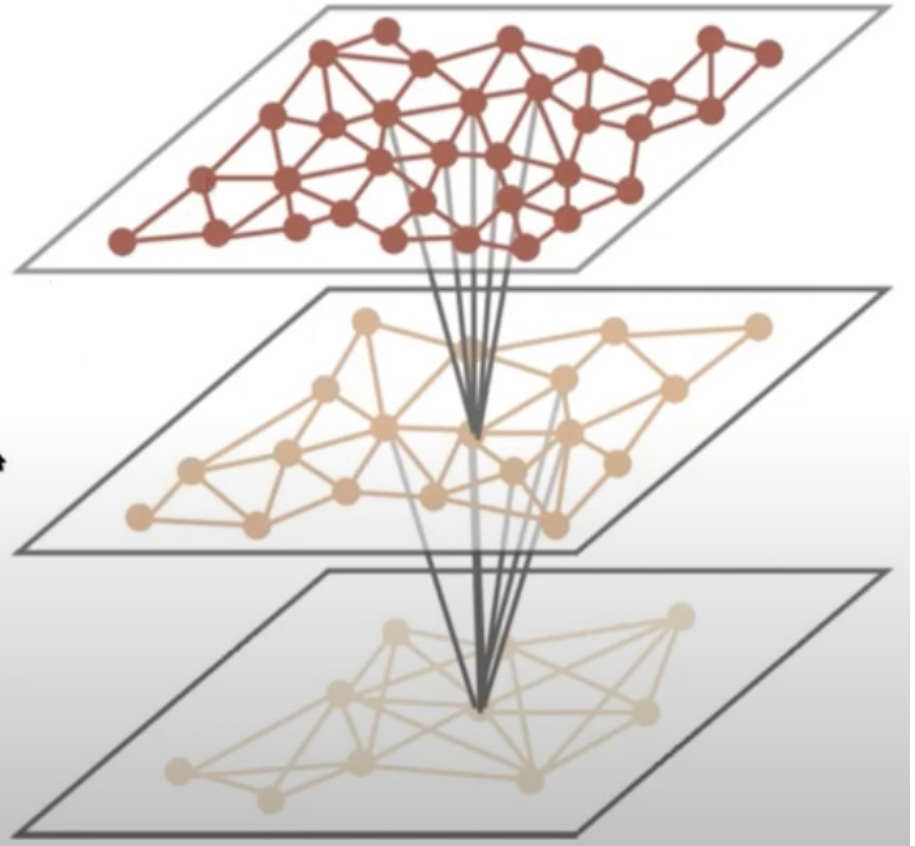
$$\begin{aligned} u(x) &= \int_D \sum_{j=1}^{r} \varphi^{(j)}(x)\psi^{(j)}(y)v(y)\,\mathrm{d}y \\ &= \sum_{j=1}^{r} \int_D \psi^{(j)}(y)v(y)\,\mathrm{d}y\,\varphi^{(j)}(x) \\ &= \sum_{j=1}^{r} \langle \psi^{(j)}, v \rangle \varphi^{(j)}(x) \end{aligned}$$

where $\langle \cdot, \cdot \rangle$ denotes the $L^2(D; \mathbb{R})$ inner product. Notice that the inner products can be evaluated independently of the evaluation point $x \in D$ hence the computational complexity of this method is $\mathcal{O}(rJ)$ which is linear in the discretization.

Multipole Graph Neural Operator (MGNO)

- Construct multi-level of graphs
- Long-term interaction captured by coarser-level graphs

⇨ Multipole method



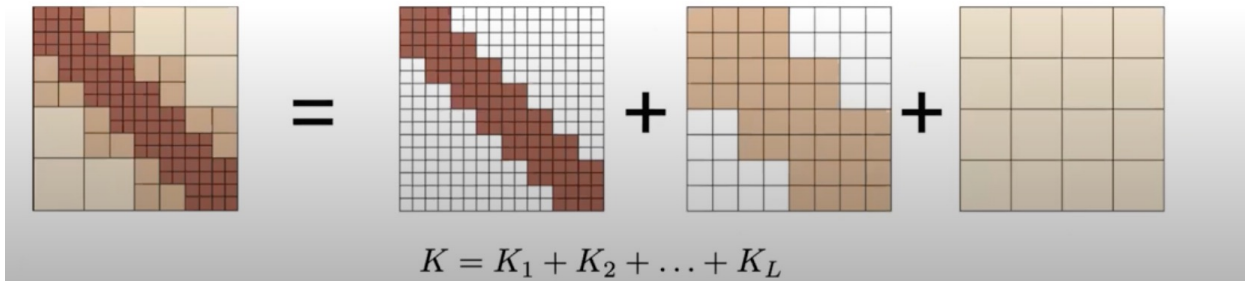
Multipole Graph Neural Operator (MGNO)

The key to the fast multipole method's linear complexity lies in the subdivision of the kernel matrix according to the range of interaction, as shown in Figure 3:

$$K = K_1 + K_2 + \ldots + K_L, \tag{21}$$

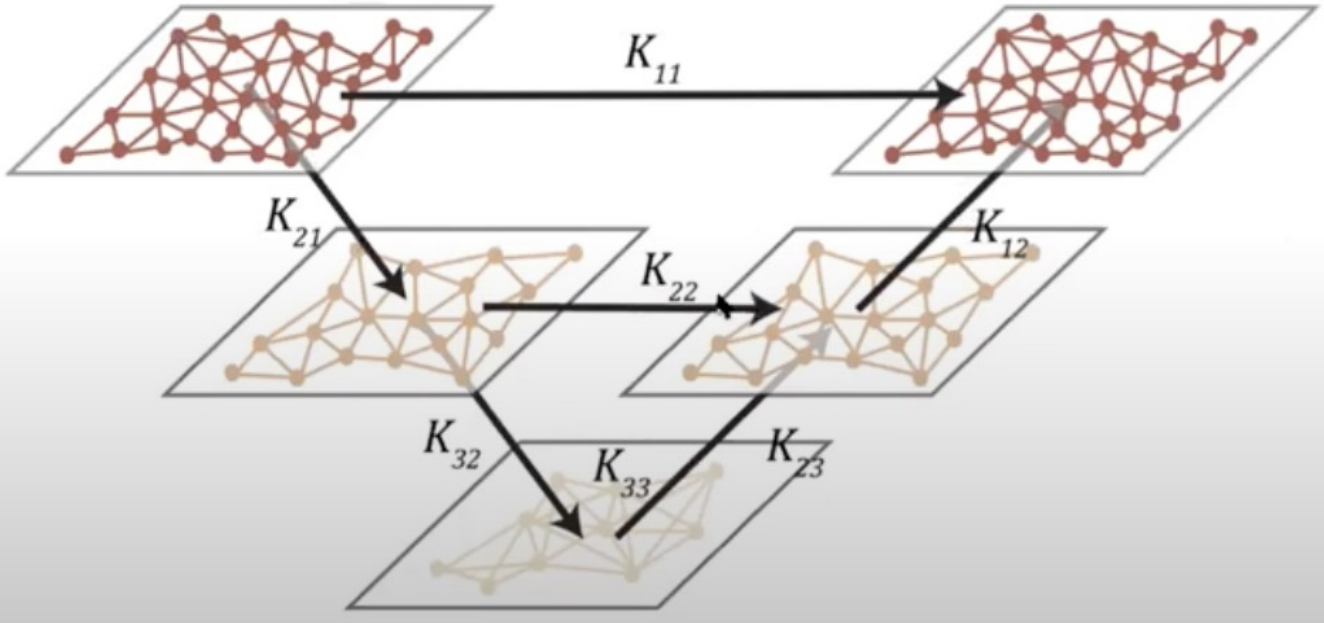
where K_ℓ with $\ell = 1$ corresponds to the shortest-range interaction, and $\ell = L$ corresponds to the longest-range interaction; more generally index ℓ is ordered by the range of interaction. While the uniform grids depicted in Figure 3 produce an orthogonal decomposition of the kernel, the decomposition may be generalized to arbitrary discretizations by allowing slight overlap of the ranges.

- Insight: the long-range interaction is usually smooth
- Decompose the interaction into different ranges
  - Short-range matrix is sparse
  - Long-range matrix is low-rank



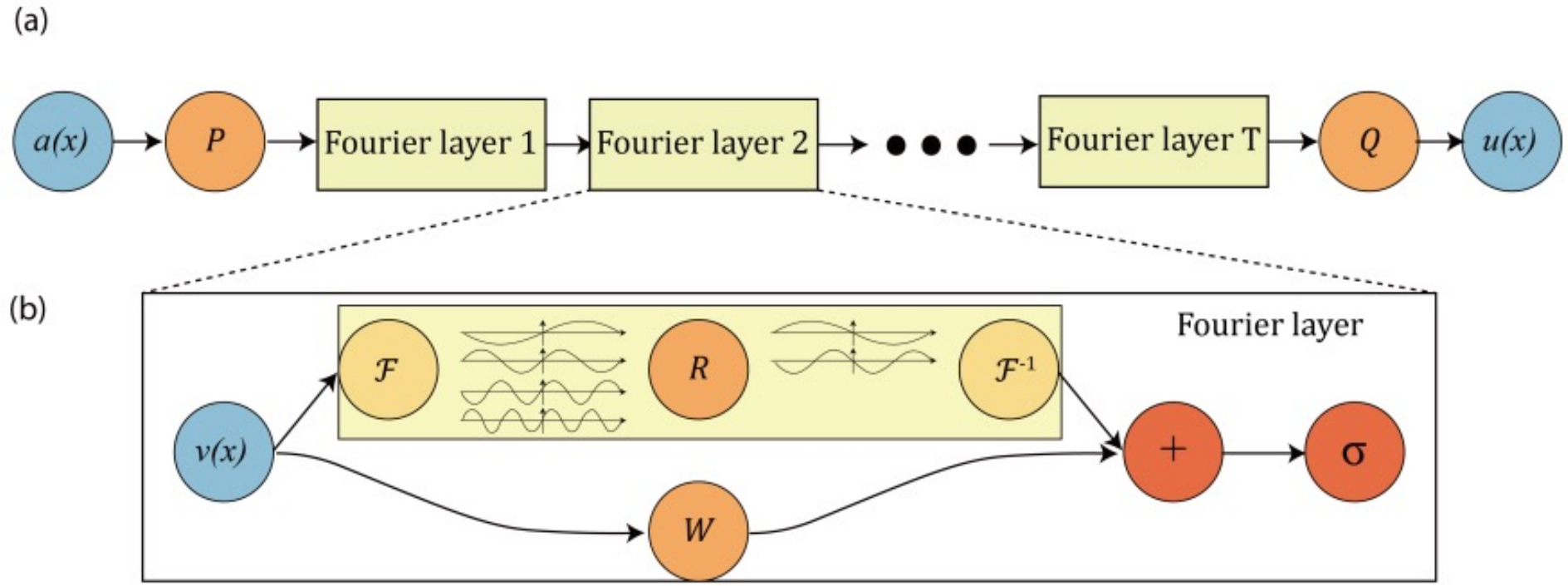
Multipole Graph Neural Operator (MGNO)

- Recursive low-rank structure = multi-resolution decomposition
- Equivalent to message passing via V-cycle algorithm



$$K \approx K_{1,1} + K_{1,2}K_{2,2}K_{2,1} + K_{1,2}K_{2,3}K_{3,3}K_{3,2}K_{2,1} + \cdots$$

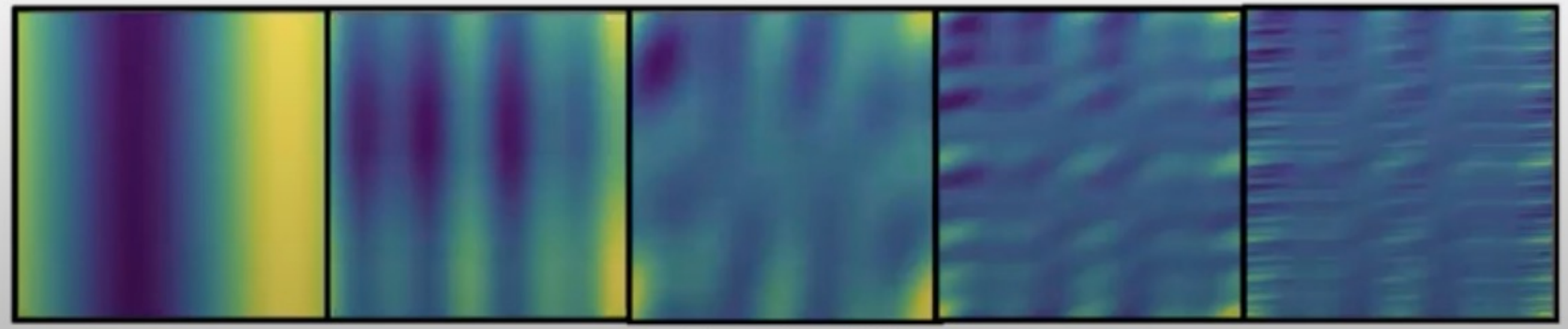
Fourier Neural Operator (FNO)



Fourier Neural Operator (FNO)

Fourier representation is more efficient than CNN.

Filters in CNN



Fourier Filters

A Light Recap of Fourier Theory

- Orthogonal set of functions

$$f, g \in \mathbb{R} \to \mathbb{R}$$

$$\langle f, g \rangle = \int_a^b f(x) g(x) dx$$

$$\langle f, g \rangle = 0$$

$$\{f_0, f_1, \ldots, f_n\}$$

$$m \neq n$$

$$\langle f_m, f_n \rangle = 0$$

$$\left\{ \cos 0, \cos(\omega_0 x), \sin(\omega_0 x), \cos(2\omega_0 x), \sin(2\omega_0 x), \ldots, \cos(n\omega_0 x), \sin(n\omega_0 x) \right\}$$

$$m, n, \ \langle \sin(m\omega_0 x), \cos(n\omega_0 x) \rangle = 0$$

$$\int_{-\pi}^{+\pi} \cos mx \cdot \sin nx \, dx$$

$$= \int_{-\pi}^{+\pi} \frac{\sin(m+n)x - \sin(m-n)x}{2} dx$$

$$= 0$$

$$m \neq n, \ \langle \sin(m\omega_0 x), \sin(n\omega_0 x) \rangle = 0$$

$$\int_{-\pi}^{+\pi} \sin mx \cdot \sin nx \, dx$$

$$= \int_{-\pi}^{+\pi} \frac{\cos(m-n)x - \cos(m+n)x}{2} dx$$

$$= 0$$

$$\langle \cos(m\omega_0 x), \cos(n\omega_0 x) \rangle = 0$$

$$\int_{-\pi}^{+\pi} \cos mx \cdot \cos nx \, dx$$

$$= \int_{-\pi}^{+\pi} \frac{\cos(m+n)x + \cos(m-n)x}{2} dx$$

$$= 0$$

A Light Recap of Fourier Theory

- Fourier Series: Any periodic function can be expanded as a series of cosine functions

$$f(x) = f(x+T) \qquad f_T(x) = c_0 + \sum_{n=1}^{\infty} c_n \cos(n\omega_0 x + \varphi_n) = c_0 + \sum_{n=1}^{\infty} c_n \sin(n\frac{2\pi}{T}x + \theta_n)$$

$$f_T(x) = c_0 + \sum_{n=1}^{\infty} c_n \cos(n\omega_0 x + \varphi_n)$$

$$= c_0 + \sum_{n=1}^{\infty} c_n \cos(n\omega_0 x)\cos(\varphi_n) - c_n \sin(n\omega_0 x)\sin(\varphi_n)$$

$$a_0 = c_0$$
$$a_n = c_n \cos(\varphi_n)$$
$$b_n = -c_n \sin(\varphi_n)$$

$$f_T(x) = a_0 + \sum_{n=1}^{+\infty} a_n \cos(n\omega_0 x) + \sum_{n=1}^{+\infty} b_n \sin(n\omega_0 x)$$

$$a_0 = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f_T(x)dx$$

$$a_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} \cos(n\omega_0 x)\cdot f_T(x)dx$$

$$b_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} \sin(n\omega_0 x)\cdot f_T(x)dx$$

$$c_n = \sqrt{a_n^2 + b_n^2}$$
$$c_0 = a_0$$
$$\varphi_n = \arctan(-\frac{b_n}{a_n})$$

A Light Recap of Fourier Theory

- Complex Domain

$$e^{i\theta} = \cos\theta + i\sin\theta \qquad \begin{cases} \cos\theta = \frac{e^{i\theta}+e^{-i\theta}}{2} \\ \sin\theta = \frac{-i(e^{i\theta}-e^{-i\theta})}{2} \end{cases}$$

$$f_T(x) = \sum_{n=-\infty}^{+\infty} d_n e^{in\omega_0 x} \qquad d_n = \frac{1}{T}\int_0^T e^{-in\omega_0 x} \cdot f_T(x)\, dx$$

A Light Recap of Fourier Theory

- General function (not periodic)

$$T \to +\infty$$

$$\omega_0 = \frac{2\pi}{T} \longrightarrow d\omega$$

$$\sum_{-\infty}^{+\infty} n\omega_0 \longrightarrow \int_{-\infty}^{+\infty} d\omega_0$$

$$\begin{aligned} f_{T\to\infty}(x) &= \sum_{n=-\infty}^{+\infty} \left\{ \frac{1}{T} \int_0^T e^{-in\omega_0 x} \cdot f_T(x)\, dx \right\} \cdot e^{in\omega_0 x} \\ &= \sum_{n=-\infty}^{+\infty} \left\{ \frac{\omega_0}{2\pi} \int_0^T e^{-in\omega_0 x} \cdot f(x)\, dx \right\} \cdot e^{in\omega_0 x}. \\ &= \sum_{n=-\infty}^{+\infty} \left\{ \omega \int_0^T e^{-in2\pi\omega x} \cdot f(x)\, dx \right\} \cdot e^{in2\pi\omega x}. \end{aligned}$$

$$F(\omega) = \int_{-\infty}^{+\infty} e^{-i\cdot 2\pi\omega x} \cdot f(x)\, dx$$

$$f(x) = \frac{1}{2\pi} \int_{n=-\infty}^{+\infty} F(\omega) \cdot e^{i\cdot 2\pi\omega x}\, d\omega$$

Fourier Neural Operator (FNO)

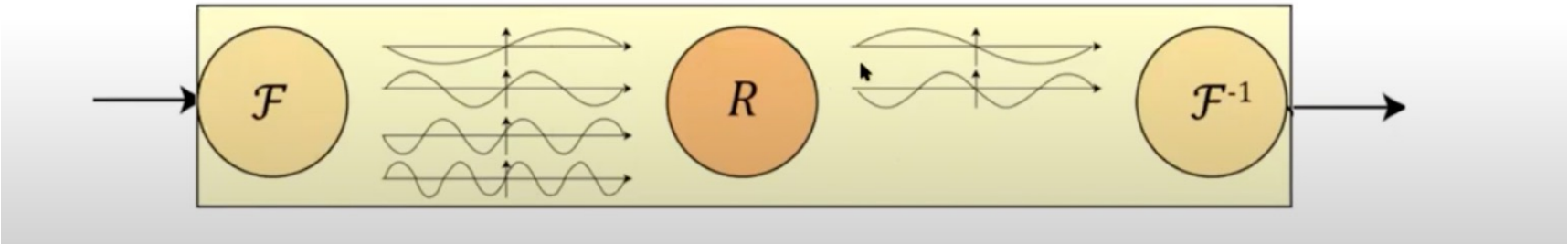
Use convolution as the integral operator
and implement with Fourier transform

$$\big(\mathcal{K}(a;\phi)v_t\big)(x) := \int_D \kappa\big(x, y, a(x), a(y); \phi\big) v_t(y)\mathrm{d}y,$$

$$\big(\mathcal{K}(\phi)v_t\big)(x) = \mathcal{F}^{-1}\Big(R_\phi \cdot (\mathcal{F}v_t)\Big)(x)$$

1. Fourier transform
2. Linear transform
3. Inverse Fourier transform

$$\big(\mathcal{K}(\phi)v_t\big)(x) = \mathcal{F}^{-1}\Big(R_\phi \cdot (\mathcal{F}v_t)\Big)(x)$$



Fourier Neural Operator (FNO)

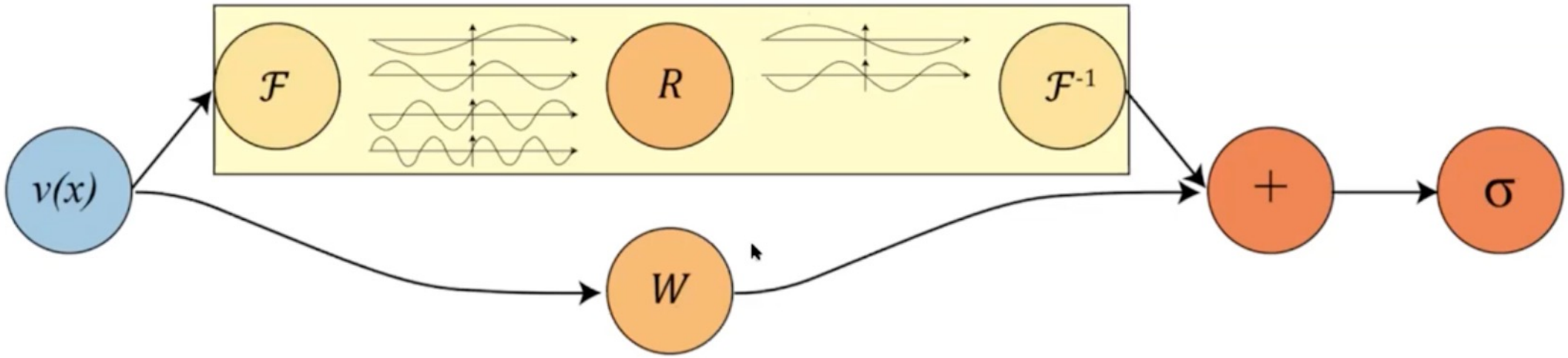
```
def forward(self, x):
    batchsize = x.shape[0]
    #Compute Fourier coeffcients up to factor of e^(- something constant)
    x_ft = torch.rfft(x, 2, normalized=True, onesided=True)

    # Multiply relevant Fourier modes
    out_ft = torch.zeros(batchsize, self.in_channels,  x.size(-2), x.size(-1)//2 + 1, 2, device=x.device)
    out_ft[:, :, :self.modes1, :self.modes2] = \
        compl_mul2d(x_ft[:, :, :self.modes1, :self.modes2], self.weights1)
    out_ft[:, :, -self.modes1:, :self.modes2] = \
        compl_mul2d(x_ft[:, :, -self.modes1:, :self.modes2], self.weights2)

    #Return to physical space
    x = torch.irfft(out_ft, 2, normalized=True, onesided=True, signal_sizes=( x.size(-2), x.size(-1)))
    return x
```

Fourier Neural Operator (FNO)

The linear transform W outside keep the track of the location information (x) and non-periodic boundary



$$v_{t+1}(x) = \sigma\left(W v_t(x) + \int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y)\, \nu_x(dy)\right)$$

Summary

- **GNO**: Subsample J' points from the J-point discretization and compute the truncated integral

$$u(x) = \int_{B(x,r)} \kappa(x,y)v(y)\,\mathrm{d}y \tag{30}$$

  at a $\mathcal{O}(JJ')$ complexity.

- **LNO**: Decompose the kernel function tensor product form and compute

$$u(x) = \sum_{j=1}^{r} \langle \psi^{(j)}, v \rangle \varphi^{(j)}(x) \tag{31}$$

  at a $\mathcal{O}(J)$ complexity.

- **MGNO**: Compute a multi-scale decomposition of the kernel

$$\begin{aligned} K &= K_{1,1} + K_{1,2}K_{2,2}K_{2,1} + K_{1,2}K_{2,3}K_{3,3}K_{3,2}K_{2,1} + \cdots \\ u(x) &= (Kv)(x) \end{aligned} \tag{32}$$

  at a $\mathcal{O}(J)$ complexity.

- **FNO**: Parameterize the kernel in the Fourier domain and compute the using the FFT

$$u(x) = \mathcal{F}^{-1}\big(R_\phi \cdot \mathcal{F}(v)\big)(x) \tag{33}$$

  at a $\mathcal{O}(J \log J)$ complexity.

Experiment

To that end, let $(\mathcal{A}, \mathcal{U}, \mathcal{F})$ be a triplet of Banach spaces. The first two problem classes considered are derived from the following general class of PDEs:

$$\mathsf{L}_a u = f \tag{34}$$

where, for every $a \in \mathcal{A}$, $\mathsf{L}_a : \mathcal{U} \to \mathcal{F}$ is a, possibly nonlinear, partial differential operator, and $u \in \mathcal{U}$ corresponds to the solution of the PDE (34) when $f \in \mathcal{F}$ and appropriate boundary conditions are imposed. The second class will be evolution equations with initial condition $a \in \mathcal{A}$ and solution $u(t) \in \mathcal{U}$ at every time $t > 0$. We seek to learn the map from a to $u := u(\tau)$ for some fixed time $\tau > 0$; we will also study maps on paths (time-dependent solutions).

Our goal will be to learn the mappings

$$\mathcal{G}^\dagger : a \mapsto u \qquad \text{or} \qquad \mathcal{G}^\dagger : f \mapsto u;$$

we will study both cases, depending on the test problem considered. We will define a probability measure μ on $\mathcal{A}$ or $\mathcal{F}$ which will serve to define a model for likely input data. Furthermore, measure μ will define a topology on the space of mappings in which $\mathcal{G}^\dagger$ lives, using the Bochner norm (3). We will assume that each of the spaces $(\mathcal{A}, \mathcal{U}, \mathcal{F})$ are Banach spaces of functions defined on a bounded domain $D \subset \mathbb{R}^d$. All reported errors will be Monte-Carlo estimates of the relative error

$$\mathbb{E}_{a\sim\mu} \frac{\|\mathcal{G}^\dagger(a) - \mathcal{G}_\theta(a)\|_{L^2(D)}}{\|\mathcal{G}^\dagger(a)\|_{L^2(D)}}$$

Experiment

Setup of the Four Methods: We construct the neural operator by stacking four integral operator layers as specified in (5) with the ReLU activation. No batch normalization is needed. Unless otherwise specified, we use $N = 1000$ training instances and 200 testing instances. We use the Adam optimizer to train for 500 epochs with an initial learning rate of 0.001 that is halved every 100 epochs. We set the channel dimensions $d_{v_0} = \cdots = d_{v_3} = 64$ for all one-dimensional problems and $d_{v_0} = \cdots = d_{v_3} = 32$ for all two-dimensional problems. The kernel networks $\kappa^{(0)}, \ldots, \kappa^{(3)}$ are standard feed-forward neural networks with three layers and widths of 256 units. We use the following abbreviations to denote the methods introduced in Section 5.

- **GNO:** The method introduced in subsection 5.1, truncating the integral to a ball with radius $r = 0.25$ and using the Nyström approximation with $J' = 300$ sub-sampled nodes.
- **LNO:** The low-rank method introduced in subsection 5.2 with rank $r = 4$.
- **MGNO:** The multipole method introduced in subsection 5.3. On the Darcy flow problem, we use the random construction with three graph levels, each sampling $J_1 = 400, J_2 = 100, J_3 = 25$ nodes nodes respectively. On the Burgers' equation problem, we use the orthogonal construction without sampling.
- **FNO:** The Fourier method introduced in subsection 5.4. We set $k_{\max,j} = 16$ for all one-dimensional problems and $k_{\max,j} = 12$ for all two-dimensional problems.

Experiment

- Sanity Check of 1D Poisson Equation

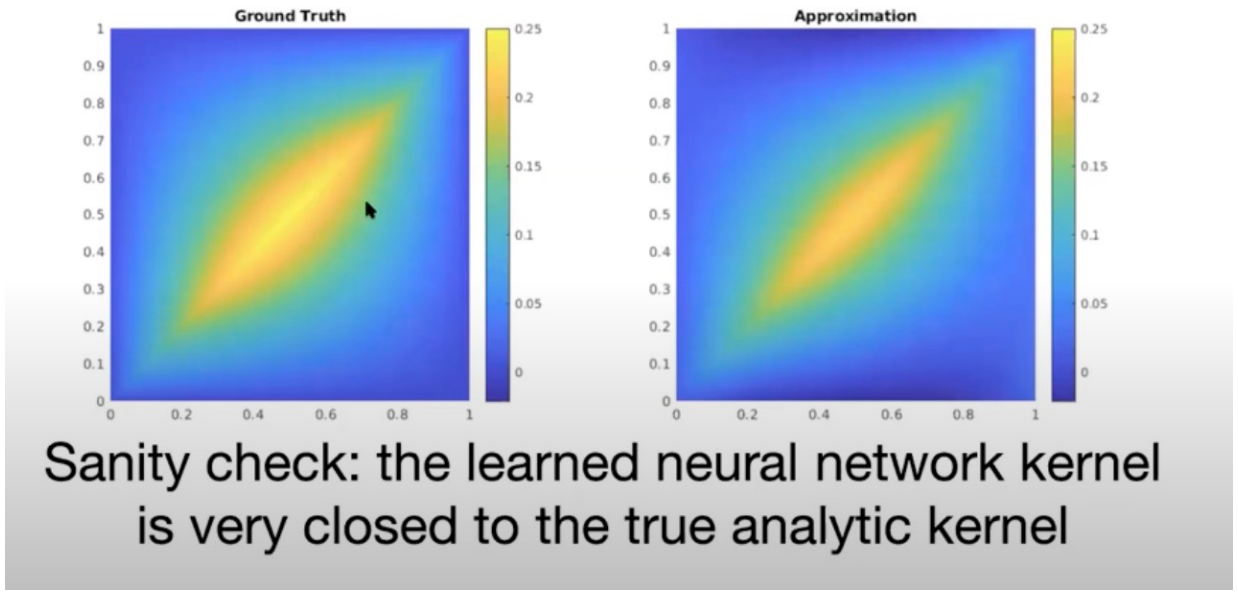
$$-\frac{d^2}{dx^2}u(x) = f(x), \qquad x \in (0,1)$$
$$u(0) = u(1) = 0$$

In this setting, $\mathcal{G}^\dagger$ has a closed-form solution given as

$$\mathcal{G}^\dagger(f) = \int_0^1 G(\cdot, y) f(y)\, \mathrm{d}y$$

where

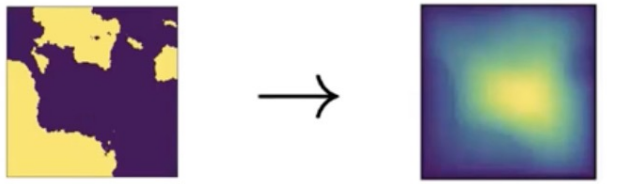
$$G(x,y) = \frac{1}{2}(x + y - |y - x|) - xy, \qquad \forall (x,y) \in [0,1]^2$$



Experiment

Example 2: 2d Darcy Flow

$$-\nabla \cdot (a(x)\nabla u(x)) = f(x) \qquad x \in (0,1)^2$$
$$u(x) = 0 \qquad x \in \partial(0,1)^2$$

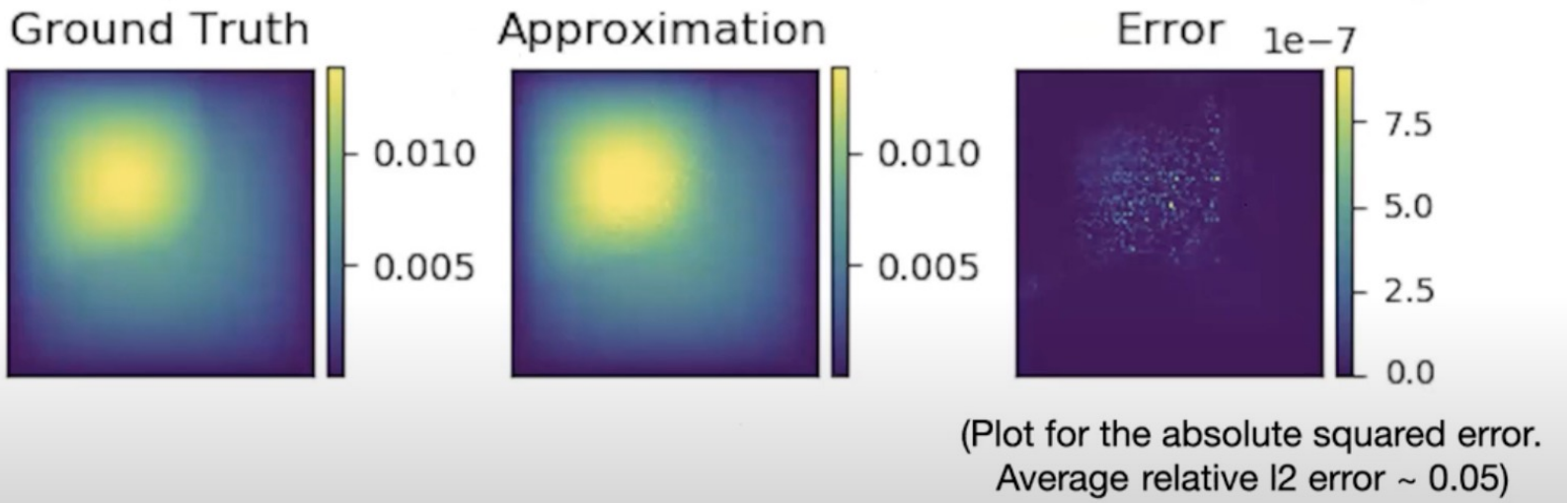


Input: coefficient → Output: solution

$$a \sim \mu \text{ where } \mu = \psi_{\#}\mathcal{N}(0, (-\Delta + 9I)^{-2})$$

Experiment

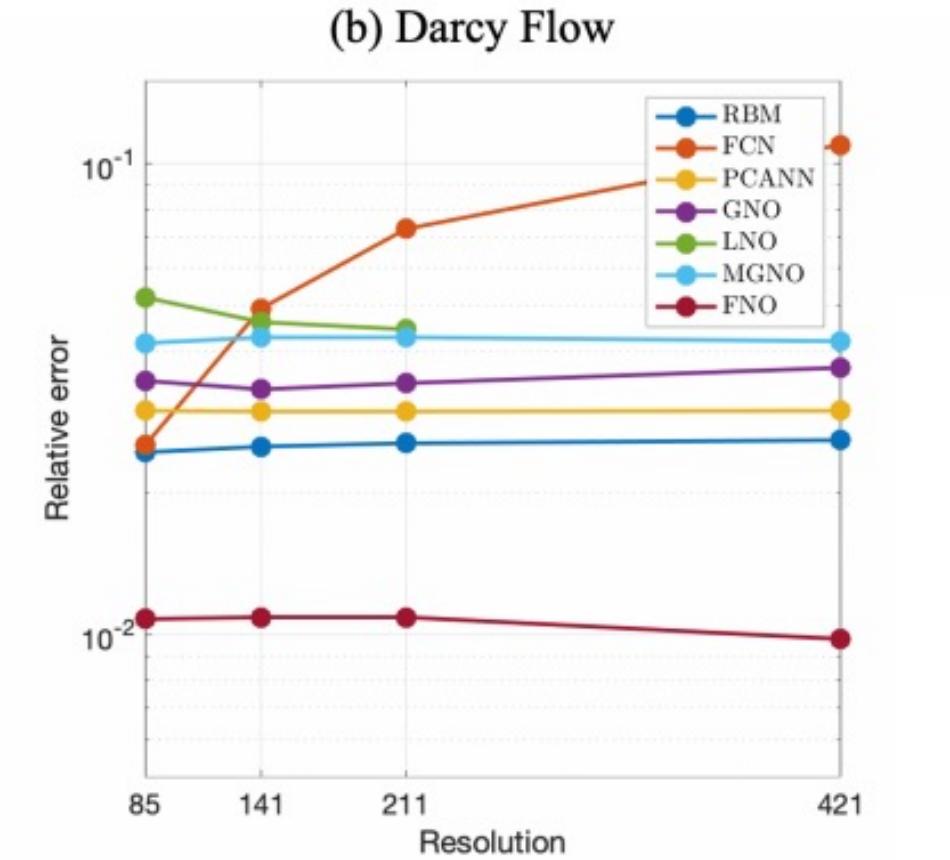
Train on 16*16, test on 241*241



(Plot for the absolute squared error.
Average relative l2 error ~ 0.05)

Graph kernel network does super-resolution

Experiment



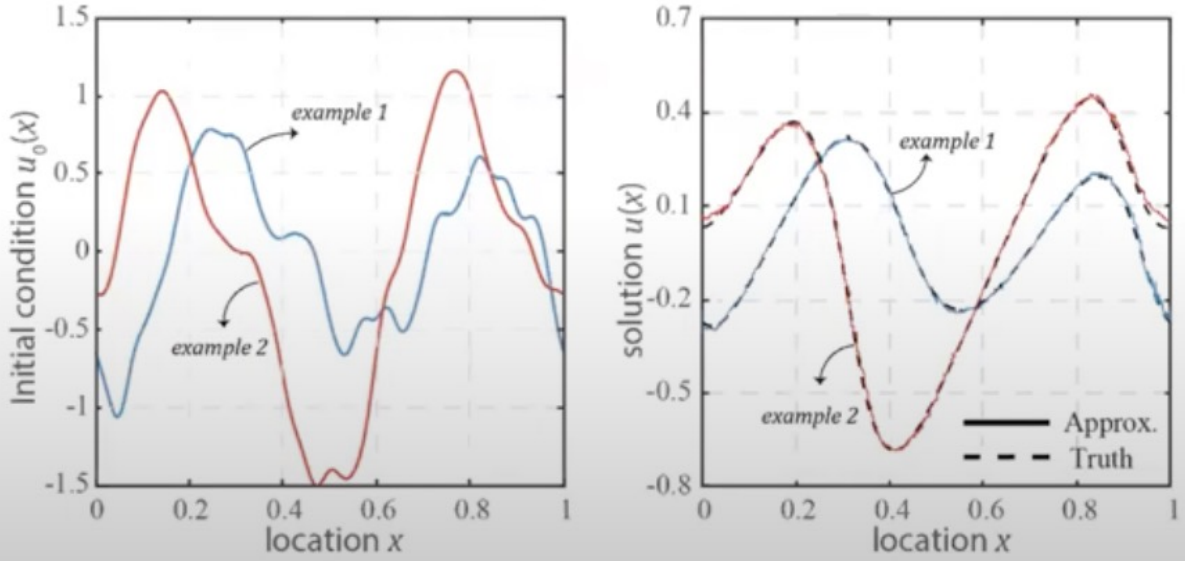
Benchmarks for time-independent problems (Burgers and Darcy):

- NN: a simple point-wise feedforward neural network.
- RBM: the classical Reduced Basis Method (using a POD basis).
- FCN: a the-state-of-the-art neural network architecture based on Fully Convolution Networks.
- PCANN: an operator method using PCA as an autoencoder on both the input and output data and interpolating the latent spaces with a neural network.
- GNO: the original graph neural operator.
- MGNO: the multipole graph neural operator.
- LNO: a neural operator method based on the low-rank decomposition of the kernel.
- FNO: the newly purposed Fourier neural operator.

Networks	$s = 85$	$s = 141$	$s = 211$	$s = 421$
NN	0.1716	0.1716	0.1716	0.1716
FCN	0.0253	0.0493	0.0727	0.1097
PCANN	0.0299	0.0298	0.0298	0.0299
RBM	0.0244	0.0251	0.0255	0.0259
DeepONet	0.0476	0.0479	0.0462	0.0487
GNO	0.0346	0.0332	0.0342	0.0369
LNO	0.0520	0.0461	0.0445	–
MGNO	0.0416	0.0428	0.0428	0.0420
FNO	**0.0108**	**0.0109**	**0.0109**	**0.0098**

Experiment

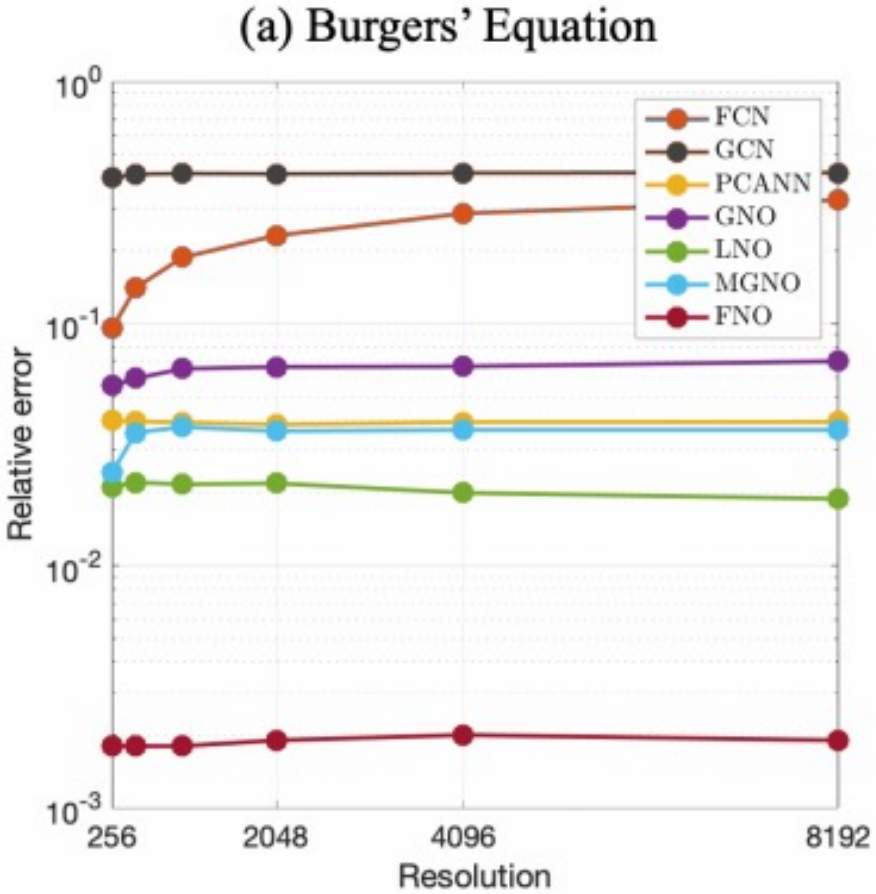
$$\partial_t u(x,t) + \partial_x(u^2(x,t)/2) = \nu \partial_{xx} u(x,t), \qquad x \in (0,1), t \in (0,1]$$
$$u(x,0) = u_0(x), \qquad x \in (0,1)$$



$$u_0 \sim \mu \text{ where } \mu = \mathcal{N}(0, 625(-\Delta + 25I)^{-2})$$

Experiment

Networks	$s = 256$	$s = 512$	$s = 1024$	$s = 2048$	$s = 4096$	$s = 8192$
NN	0.4714	0.4561	0.4803	0.4645	0.4779	0.4452
GCN	0.3999	0.4138	0.4176	0.4157	0.4191	0.4198
FCN	0.0958	0.1407	0.1877	0.2313	0.2855	0.3238
PCANN	0.0398	0.0395	0.0391	0.0383	0.0392	0.0393
DeepONet	0.0569	0.0617	0.0685	0.0702	0.0833	0.0857
GNO	0.0555	0.0594	0.0651	0.0663	0.0666	0.0699
LNO	0.0212	0.0221	0.0217	0.0219	0.0200	0.0189
MGNO	0.0243	0.0355	0.0374	0.0360	0.0364	0.0364
FNO	**0.0018**	**0.0018**	**0.0018**	**0.0019**	**0.0020**	**0.0019**



Experiment

3. Navier-Stokes Equation

We consider the 2-d Navier-Stokes equation for a viscous, incompressible fluid in vorticity form on the unit torus:

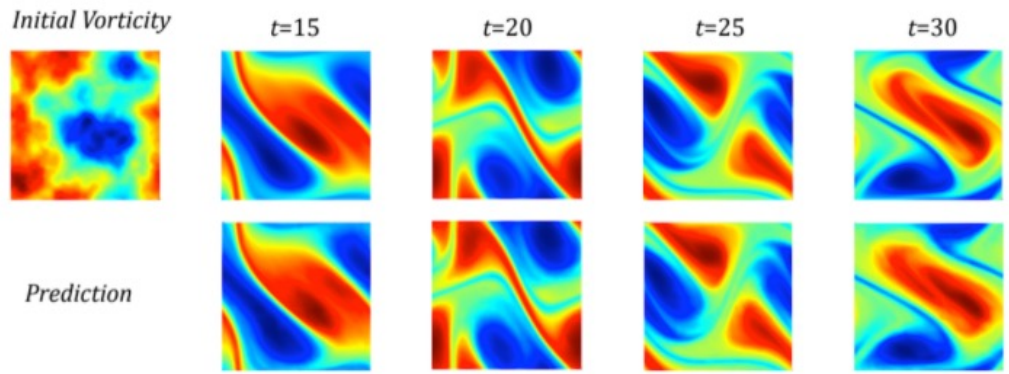
$$\partial_t w(x,t) + u(x,t)\cdot\nabla w(x,t) = \nu\Delta w(x,t) + f(x), \qquad x\in(0,1)^2, t\in(0,T]$$

$$\nabla\cdot u(x,t) = 0, \qquad x\in(0,1)^2, t\in[0,T]$$

$$w(x,0) = w_0(x), \qquad x\in(0,1)^2$$

where u is the velocity field, $w=\nabla\times u$ is the vorticity, w_0 is the initial vorticity, ν is the viscosity coefficient, and f is the forcing function. We are interested in learning the operator mapping the vorticity up to time 10 to the vorticity up to some later time $T>10$, defined by $w|_{(0,1)^2\times[0,10]}\mapsto w|_{(0,1)^2\times(10,T]}$. We experiment with the viscosities $\nu=1e{-}3, 1e{-}4, 1e{-}5$, decreasing the final time T as the dynamic becomes chaotic.

Configs	Parameters	Time per epoch	$\nu=1e{-}3$	$\nu=1e{-}4$	$\nu=1e{-}5$
FNO-3D	6,558,537	38.99s	**0.0086**	**0.0820**	0.1893
FNO-2D	414,517	127.80s	0.0128	0.0973	**0.1556**
U-Net	24,950,491	48.67s	0.0245	0.1190	0.1982
TF-Net	7,451,724	47.21s	0.0225	0.1168	0.2268
ResNet	266,641	78.47s	0.0701	0.2311	0.2753



Benchmarks for time-dependent problems (Navier-Stokes):

- ResNet: 18 layers of 2-d convolution with residual connections.
- U-Net: A popular choice for image-to-image regression tasks consisting of four blocks with 2-d convolutions and deconvolutions.
- TF-Net: A network designed for learning turbulent flows based on a combination of spatial and temporal convolutions.
- FNO-2d: 2-d Fourier neural operator with an RNN structure in time.
- FNO-3d: 3-d Fourier neural operator that directly convolves in space-time.

Exploration

- Continuous Neural Operator
- Active learning for Neural Operator