Physics-informed machine learning

Emmanuel Skoufris

University of Queensland

July 5, 2024

Introduction

- To recap, last week we examined how to solve a PDE given some boundary conditions using physics loss.
- Ideally, to improve upon traditional methods and PINNs, we would like to have a solver that can account for a variety of boundary/initial conditions.
- PINNs also "struggl[e] to propagate information from the initial or boundary conditions to unseen parts of the interior or to future times."
- Still, the space of boundary/initial conditions is *infinite* dimensional, so we would need to learn a map from some infinite dimensional space to another, namely, the space of possible solutions.

Neural operators

- Consider the PDE

$$(\mathcal{L}_a u)(x) = f(x),\ x \in \mathcal{D}$$
$$u(x) = 0,\ x \in \partial\mathcal{D},$$

  where $a \in L^2(\mathcal{D}_0; \mathbb{R}^{d_a})$ and $u \in L^2(\mathcal{D}_0; \mathbb{R}^{d_u})$, and $\mathcal{D}_0$ is a bounded domain (open, connected) subset of $\mathbb{R}^n$.
- For each $x, y \in \mathcal{D}_0$, let $G_a(x, y)$ satisfy

$$(\mathcal{L}_a G_a)(x, y) = \delta_x(y) \cong \begin{cases} 0, & y \neq x \\ \infty, & y = x. \end{cases}$$

- Remark: this is known as the *Green's function.*

Neural operators

- The solution is given by $u(x) = \int_{\mathcal{D}_0} G_a(x, y) f(y) dy$, since

$$(\mathcal{L}_a u)(x) = \int_{\mathcal{D}_0} (\mathcal{L}_a G_a)(x, y) f(y) dy = \int_{\mathcal{D}_0} \delta_x(y) f(y) dy = f(x).$$

- One can similarly check that the boundary condition is satisfied.
- Inspired by this, we build a neural network to approximate the solution operator $\mathcal{G} : L^2(\mathcal{D}_0, \mathbb{R}^{d_a}) \to L^2(\mathcal{D}_0, \mathbb{R}^{d_u})$ mapping between two *infinite* dimensional vector (Hilbert) spaces.
- If such a map can be successfully learned, we won't need to fully retrain a model for each different set of parameters/parameter functions, in contrast to traditional numerical solvers.

Neural operators

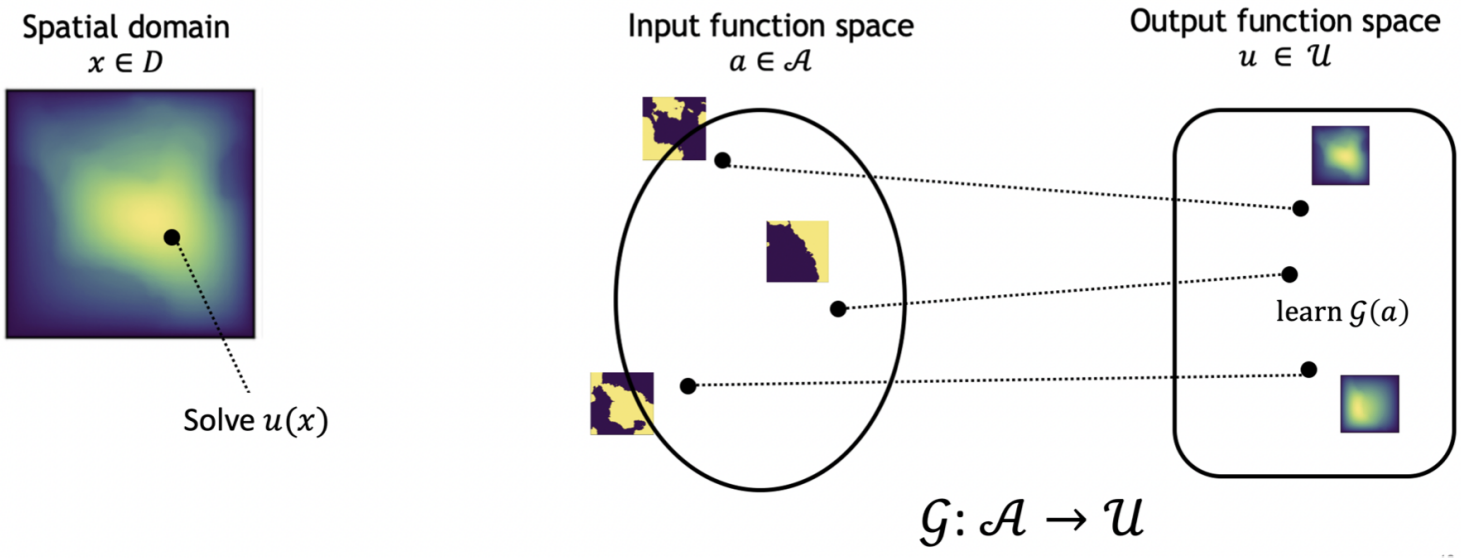


Figure: Basic idea of a neural operator
Left: numerical solvers and PINNs focus on solving one specific instance of a PDE. Right: neural operators learn the solution operator for a family of equations.

Neural operators

- We approximate $\mathcal{G}$ using a parametric function $\mathcal{G}_\theta : L^2(\mathcal{D}_0, \mathbb{R}^{d_a}) \to L^2(\mathcal{D}_0, \mathbb{R}^{d_u})$ for a vector of parameters $\theta \in \mathbb{R}^p$.
- The architecture is given by

$$\mathcal{G}_\theta = Q \circ \sigma(W_L + K_L + b_L) \circ \cdots \circ \sigma(W_1 + K_1 + b_1) \circ P,$$

where

  - $P \in \mathbb{R}^{d_0 \times d_a}$, $Q \in \mathbb{R}^{d_u \times d_L}$ are lifting and projection operators, respectively;
  - $b_\ell : \mathcal{D}_\ell \to \mathbb{R}^{d_{\ell+1}}$ for $\ell = 1, \ldots, L$ are the bias functions;
  - $W_\ell \in \mathbb{R}^{d_{\ell+1} \times d_\ell}$ for $\ell = 1, \ldots, L-1$;
  - σ is the activation function, acting pointwise on the inputs.

- But what is K_ℓ? Its definition is slightly is more complicated.

Neural operators

- Given $a \in L^2(\mathcal{D}_0, \mathbb{R}^{d_a})$, we let $v_0 = Pa$, where $v_0(x) = (Pa)(x) = Pa(x)$, and Q works similarly.
- We thereby get a sequence of functions

$$v_0 \mapsto v_1 \mapsto \cdots \mapsto v_L \mapsto Qv_L = \hat{u},$$

with $v_\ell : \mathcal{D}_\ell \to \mathbb{R}^{d_{\ell+1}}$ for $\ell = 0, 1, \ldots, L-1$, defined recursively by

$$v_\ell(x) = \sigma\left[W_\ell v_{\ell-1}(\Pi_\ell(x)) + (K_\ell v_{\ell-1})(x) + b_\ell(x)\right], \; x \in D_\ell,$$

where $\Pi_\ell : \mathcal{D}_\ell \to \mathcal{D}_{\ell-1}$ are fixed mappings.

Neural operators

- $K_\ell : L^2(\mathcal{D}_{\ell-1}; \mathbb{R}^{d_\ell}) \to L^2(D_\ell; \mathbb{R}^{d_{\ell+1}})$ is a *linear integral kernel operator*, defined by

$$(K_\ell v_\ell)(x) = \int_{\mathcal{D}_{\ell-1}} \kappa_\ell(x, y) v_{\ell-1}(y)\, dy \quad \text{for } x \in \mathcal{D}_\ell,$$

  where $\kappa_\ell : \mathcal{D}_\ell \times \mathcal{D}_{\ell-1} \to \mathbb{R}^{d_{\ell+1} \times d_\ell}$ is continuous.
- Typically, we'll let $d_\ell = d_{\ell'}$ for each $\ell, \ell' = 0, 1, \ldots, L$, although it would be interesting to experiment with mapping between different dimensions!
- In that case, and to make things simpler, we also impose the constraint that

$$\kappa_\ell(x, y) = \kappa_\ell(x - y),$$

  making K_ℓ a *convolution operator*; $K_\ell v_\ell = \kappa_\ell * v_\ell$.

Neural operators

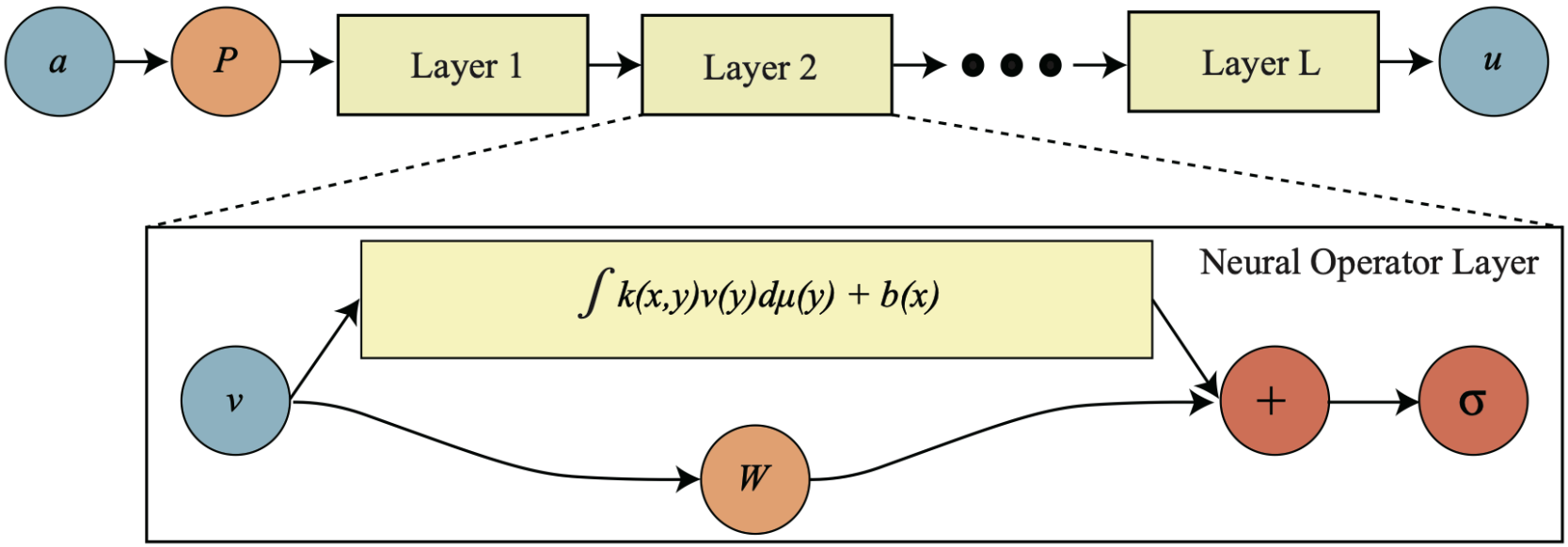


Figure: Neural operator architecture.

Fourier transform

- For a function $f : \mathbb{R}^n \to \mathbb{R}^k$, the Fourier transform is given by

$$\mathcal{F}\{f\}(\boldsymbol{\xi}) = \int_{\mathbb{R}^n} f(\mathbf{x}) e^{-2\pi i \boldsymbol{\xi} \cdot \mathbf{x}} d\mathbf{x}.$$

- The inverse Fourier transform is given by

$$\mathcal{F}^{-1}\{f\}(\boldsymbol{\xi}) = \int_{\mathbb{R}^n} f(\mathbf{x}) e^{2\pi i \boldsymbol{\xi} \cdot \mathbf{x}} d\mathbf{x}.$$

- We also have the convolution theorem:

$$\mathcal{F}\{f * g\}(\boldsymbol{\xi}) = \mathcal{F}\{f\}(\boldsymbol{\xi}) \mathcal{F}\{g\}(\boldsymbol{\xi}),$$

where $*$ and multiplication are performed component-wise.

Neural operators

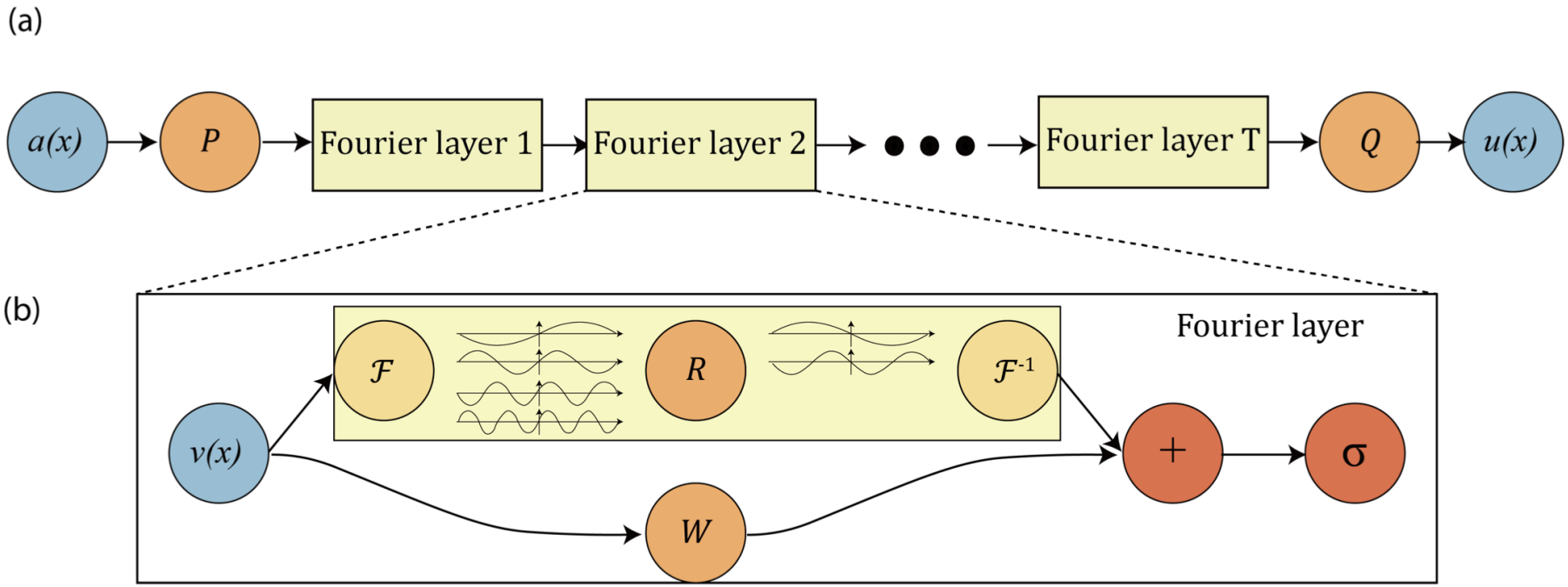
- Thus, for the Fourier neural operator, by applying the convolution theorem we let the integral operator be given by

$$(K_\ell v_\ell)(x) = \mathcal{F}^{-1}(\mathcal{F}\{\kappa_\ell\}\mathcal{F}\{v_\ell\})(x) = \mathcal{F}^{-1}(R_\phi \cdot \mathcal{F}\{v_\ell\})(x),$$

  where R_ϕ is now the object to be estimated.
- Notice that all of these operations have been defined abstractly, without regard to how to actually compute them - we will now try to describe how to work with these objects in practice.
- We'll make use of the discrete Fourier transform (DFT):

$$\mathbf{X}_{k_1,\ldots,k_d} = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} \mathbf{x}_{n_1,\ldots,n_d} \exp\left(-2\pi\left(\frac{n_1 k_1}{N_1} + \cdots + \frac{n_d k_d}{N_d}\right)\right)$$

Fourier neural operator



Fourier neural operator

- Let the domain $\mathcal{D}_0$ be discretized with n points, so that $v_\ell \in \mathbb{R}^{n \times d_{\ell+1}}$, and $\mathcal{F}\{v_\ell\} \in \mathbb{C}^{n \times d_{\ell+1}}$, where $\mathcal{F}\{v_\ell\}$ is the DFT at n points of the domain - the result is n vectors of length $d_{\ell+1}$.
- We assume that κ_ℓ is periodic, so that it admits a Fourier expansion, with integer frequencies for each component.
- Given some integer $k_{\max}$, we let

$$N = |\{k \in \mathbb{Z}^{d_\ell} : \|k\|_\infty \leq k_{\max}\}|.$$

- Usually, in applying the discrete Fourier transform, the low-frequency modes are determined by the ℓ_1-norm, but here we use the ℓ_∞-norm for computational efficiency.

Fourier neural operator

- For each mode $\mathbf{k} \in \mathcal{D}_\ell$, we have $R_\phi(k) = \mathcal{F}\{\kappa_\ell\}(k) \in \mathbb{C}^{d_{\ell+1} \times d_\ell}$, so $R_\phi \in \mathbb{C}^{n \times d_{\ell+1} \times d_\ell}$.
- We truncate the higher modes to get $R_\phi \in \mathbb{C}^{N \times d_{\ell+1} \times d_\ell}$.
- Since we convolve v_t with a function with only the lower frequencies, we truncate the higher modes to get $\mathcal{F}\{v_\ell\} \in \mathbb{C}^{N \times d_{\ell+1}}$.
- We then get that

$$(R_\phi \cdot \mathcal{F}\{v_\ell\})_{k,l} = \sum_{j=1}^{d_\ell} R_{k,l,j} (\mathcal{F}\{v_\ell\})_{k,j},$$

  where $k = 1, \ldots, N, j = 1, \ldots, d_{\ell+1}$.
- The R coefficients are the numbers we estimate during training.
- When computing the DFTs, we use the *Fast Fourier Transform* (FFT), however we must use a uniform grid to do so.

Training the FNO

- Our goal is to attain

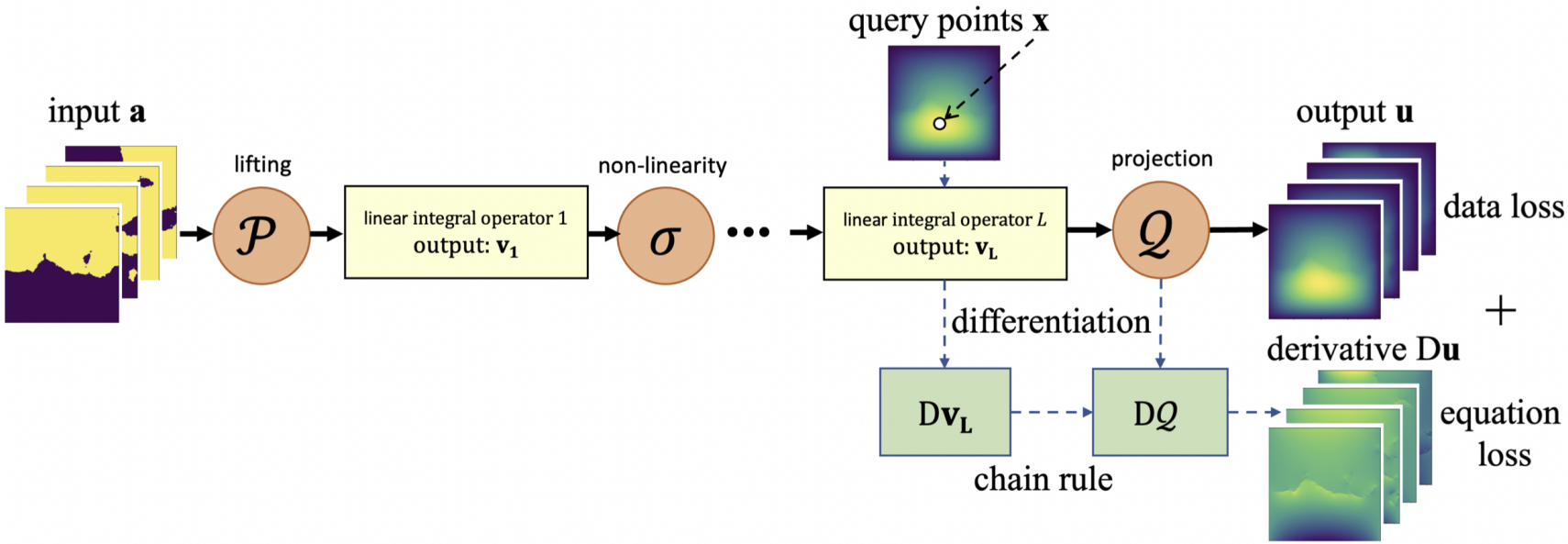
$$\min_{\theta\in\mathbb{R}^p} \mathbb{E}_{a\sim\mu} C[\mathcal{G}_\theta(a), \mathcal{G}(a)] \approx \min_{\theta\in\mathbb{R}^p} \frac{1}{N}\sum_{j=1}^{N} \|\mathcal{G}_\theta(a^j) - \mathcal{G}(a^j)\|^2_{L^2(\mathcal{D}_0;\mathbb{R}^{d_u})}$$

$$\approx \min_{\theta\in\mathbb{R}^p} \frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{N_j} \frac{1}{N_j} |\mathcal{G}_\theta(a_i^j) - \mathcal{G}(a_i^j)|^2.$$

- The functions a^j are sampled from some distribution μ.
- We obtain $\mathcal{G}(a_i^j)$ via numerical approximation usually.
- Ideally, we'd like to not have to use numerical solutions for efficiency reasons.

Physics-informed neural operator (PINO)

- We'd like to instead use the physics loss to train the network as opposed to using numerical approximations:

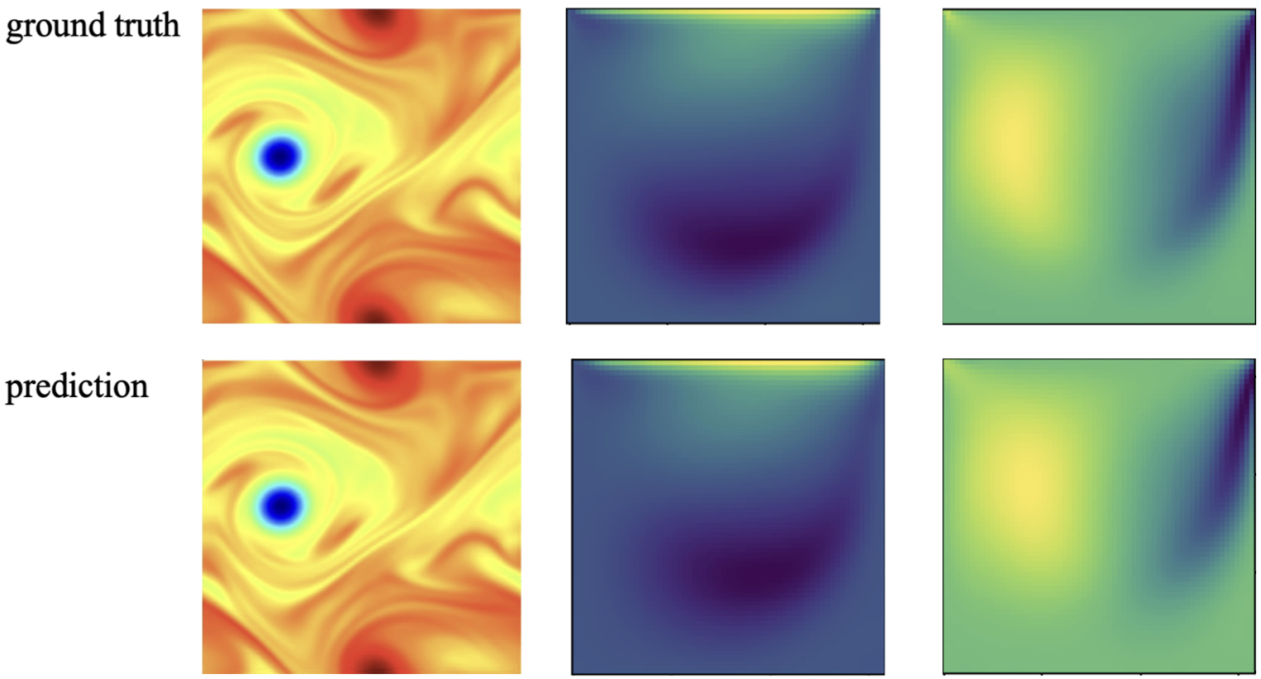


Physics-informed neural operator

- It is (apparently) a challenge to implement autograd with respect to the inputs when using the FFT.
- We can also train the PINO in two stages:
  - **Operator learning**: learn a neural operator $\mathcal{G}_\theta$ to approximate the target solution operator $\mathcal{G}$ using either/both the data loss data or/and the physics-loss.
  - **Instance-wise fine-tuning**: use $\mathcal{G}_\theta(\mathrm{a})$ as the ansatz to approximate u with the physics loss and/or an additional operator loss obtained from the operator learning phase.

Example solution

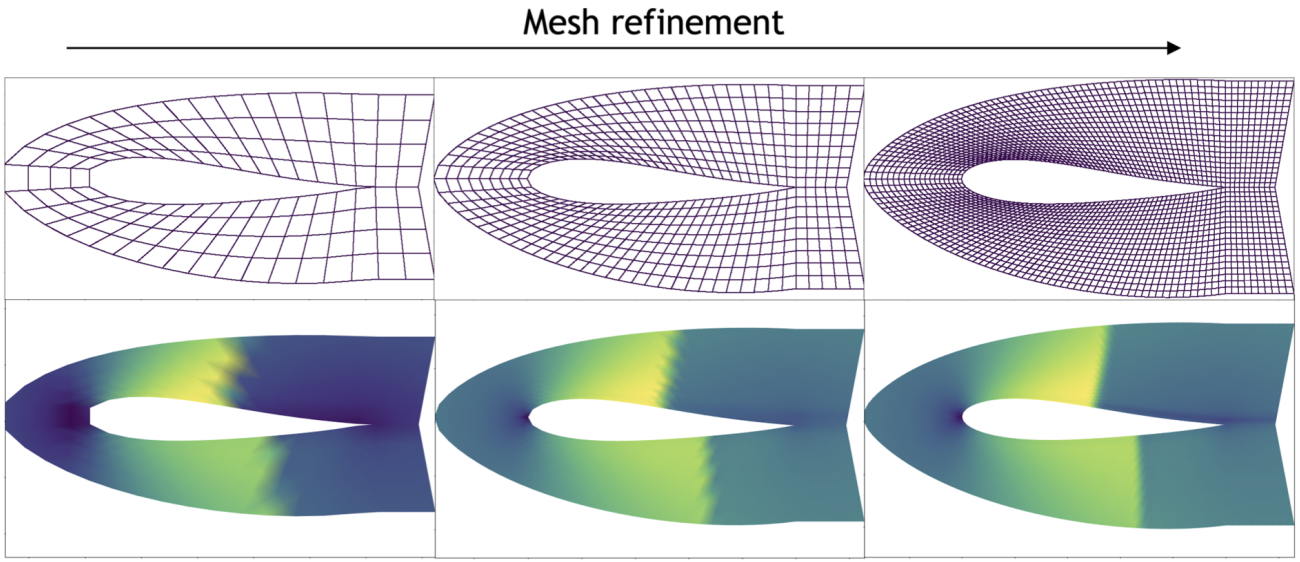
$$\partial_t w(x,t) + u(x,t) \cdot \nabla w(x,t) = \nu w(x,t) + f(x),\ x \in (0,l)^2, t \in (0,T]$$
$$\nabla u(x,t) = 0, x \in (0,l)^2, t \in [0,T]$$
$$w(x,0) = w_0(x),\ x \in (0,l)^2$$



(a) KF: vorticity (b) Cavity: velocity in x, y

Benefits

- The FNO benefits from the Fast-Fourier Transfrom (FFT) algorithm.
- These models are also *discretization invariant*:
  - The FNO can ouput a solution given any discretization of the domain.
  - Given finer and finer refinements, the predicted points will also converge in some sense to the output of some continuum operator (although, of course, convergence to the real solution is not guaranteed).
  - This is in general not true for neural networks.



The week ahead

- How are FNOs implemented in practice?
- Why are neural operators discretization invariant?
- What are the other ways to estimate the integral kernel operator?
- Are there other ways to learn maps between infinite dimensional vector spaces, without using the kernel method inspired by Green's functions whilst still guaranteeing discretization invariance?
- How does one apply the FNO to general geometries (Geo-FNO)?