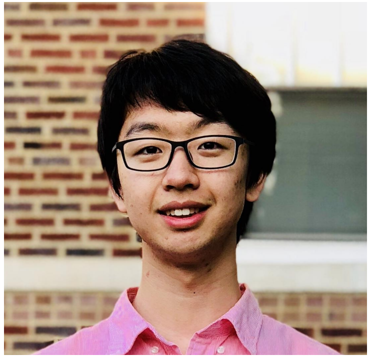
Neural Operator For Parametric PDEs

April 2021

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu,
Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar

Caltech and Purdue University



Zongyi Li

Nikola Kovachki

Burigede Liu

Kamyar Azizzadenesheli

Anima Anandkumar

Andrew Stuart

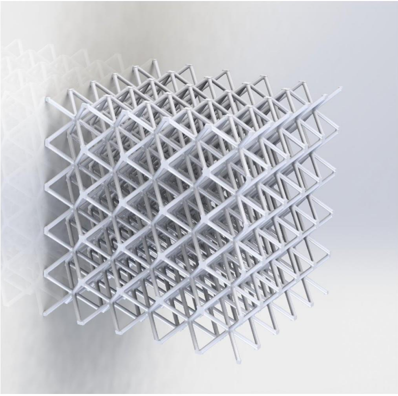
Kaushik Bhattacharya

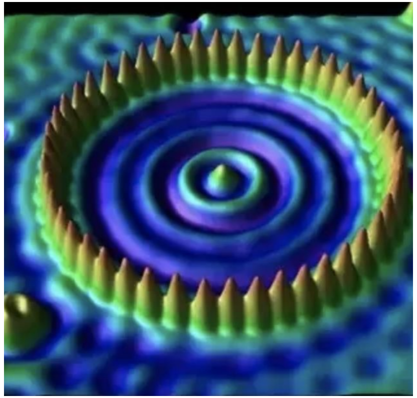
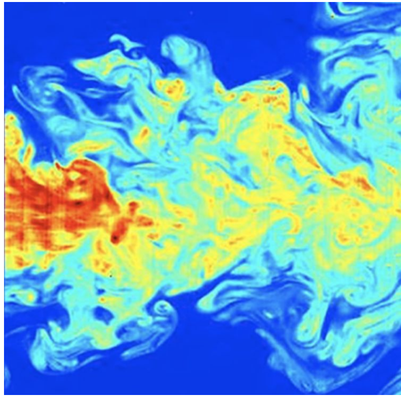
Overview

1. Introduction
    a. Neural operator vs FDM/FEM
    b. Neural operator vs CNN
2. Neural operator
    a. Intuition: Green’s function
    b. Formulation
3. Fourier neural operator
4. Experiments
5. Future work

1. Introduction

Problems in science and engineering reduce to PDEs.



Introduction

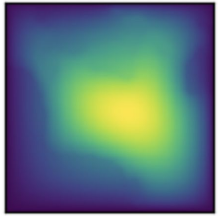
- Learning parametric PDE:

  Given the a set of coefficients/boundary conditions

  Find the solution functions

Input: coefficient

→

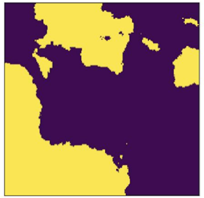


Output: solution

Problem Setting

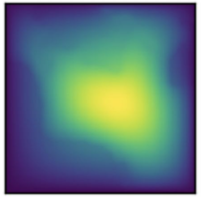
Second order elliptic equation:

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



Input: a(x)

$\rightarrow$



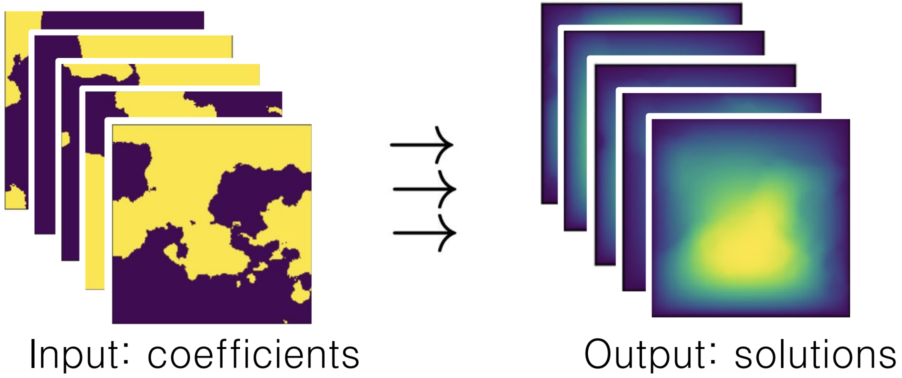
Output: u(x)

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

Operator learning

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

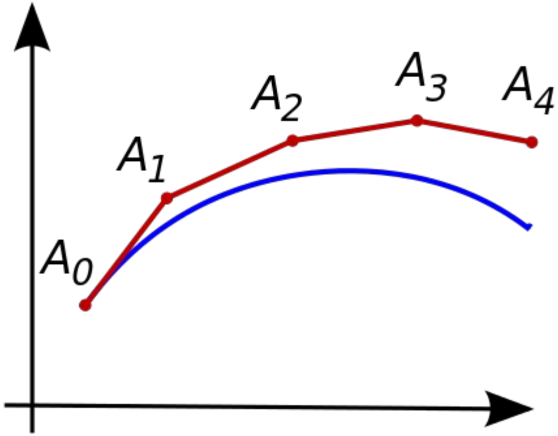
- Fix an equation
- Multiple training instances
- Learn the mapping



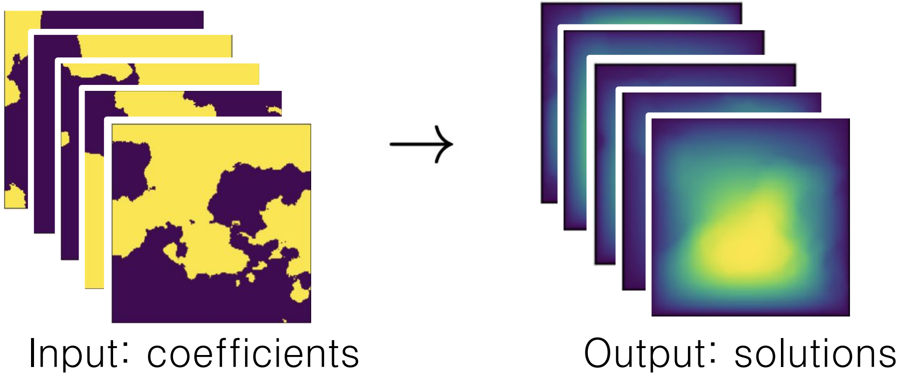
Slow to train. Fast to evaluate. $\mathcal{F} : \mathcal{A} \times \Theta \rightarrow \mathcal{U}$

Solve vs learn

Conventional methods:
Solve the equation
By approximation on a mesh



Data-driven methods:
Learn the trajectory
From a distribution



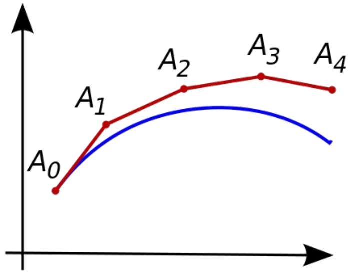
Solve vs learn

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



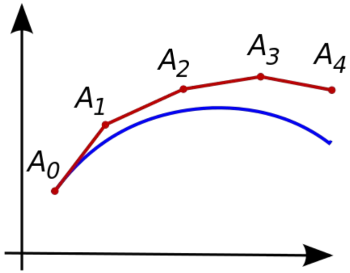
Solve vs learn

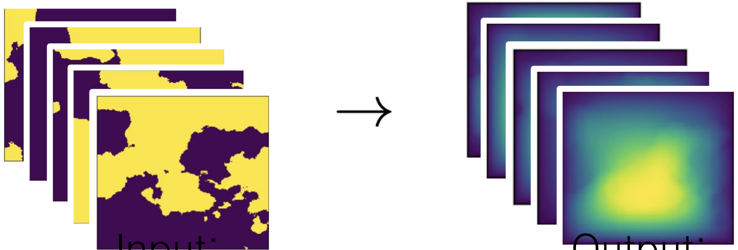
Conventional methods:

- Solve for any parameters
- Worst case guarantees
- Consistency

Data-driven methods:

- Parameters from a distribution
- Less guaranteed
- Not “consistent”





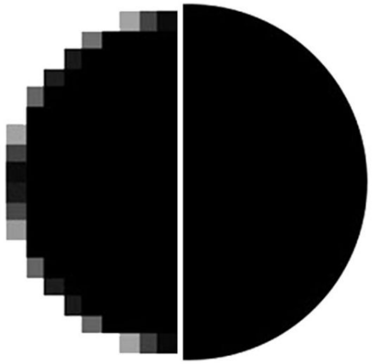
Input:
coefficients

Output:
solutions

Operator learning

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

Discretized vector

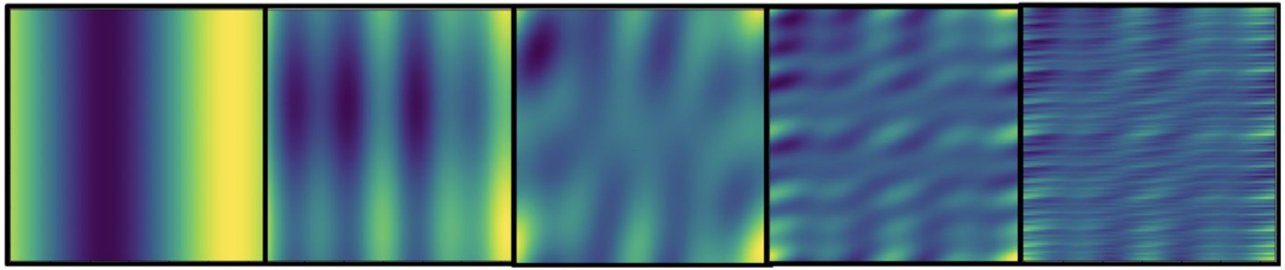


Continuous function

Operator learning

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

Filters in CNN



Fourier Filters

2. Neural operator

$$u = (K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0) v$$

Problem Setting

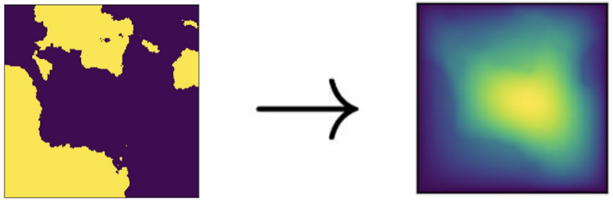
Second order elliptic equation:

$$
\begin{aligned}
-\nabla \cdot (a(x)\nabla u(x)) &= f(x), \qquad x \in D \\
u(x) &= 0, \qquad x \in \partial D
\end{aligned}
$$

Given $\{a_j, u_j\}_{j=1}^N$ pairs of functions

Want to learn the **operator**

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



Input: a(x) Output: u(x)

Intuition: kernel method

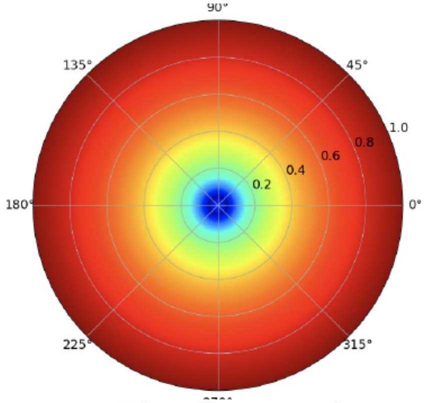
$$-\nabla \cdot (a(x)\nabla u(x)) = f(x), \quad x \in D$$
$$u(x) = 0, \quad x \in \partial D$$

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

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



$G_a(x,y)$.

Integral Operator

Idea: Approximate the kernel by a **neural network** κ_ϕ

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

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

Iterative solver: stack layers

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

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

Add iterations for t = 1,…,T, like an implicit method

$$K : v_t \mapsto v_{t+1}$$

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

Neural operator

$$u = (K_l \circ \sigma_l \circ \cdots \circ \sigma_1 \circ K_0)\, v$$

K are linear non-local integral operator
σ are non-linear local activation functions

Neural operator

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

P, Q are local network (encoder, decoder)

P lifts the input to a high dimensional channel space.
Q projects the representation back to the original space

Approximation bound

For any continuous operator defined on a compact domain, there exists a two-layers neural operators can approximate it.
Derivation following Chen & Chen and DeepONet (Lu et. al.)

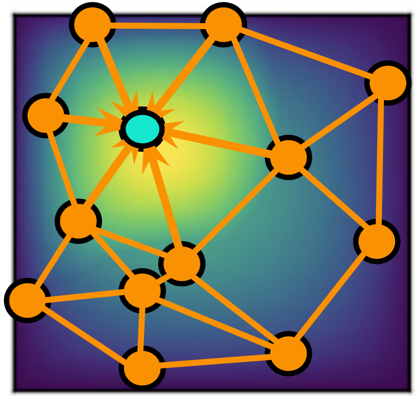
Neural operator

$$\int_D \kappa_\phi(x, y, a(x), a(y))v_t(y)\, \nu_x(dy)$$

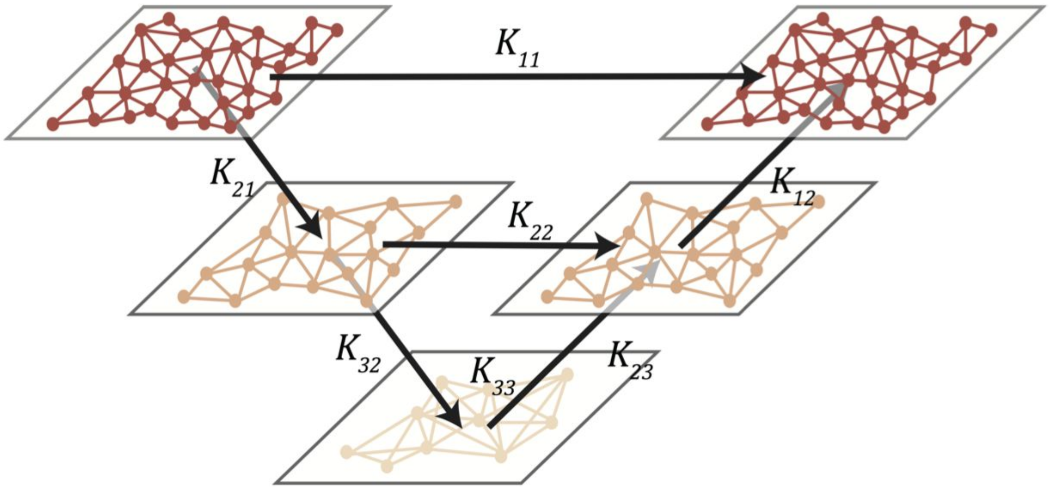
Four variations:

1. Graph neural operator
2. Multipole graph neural operator
3. Low-rank neural operator
4. Fourier neural operator

Graph-based neural operators



GKN

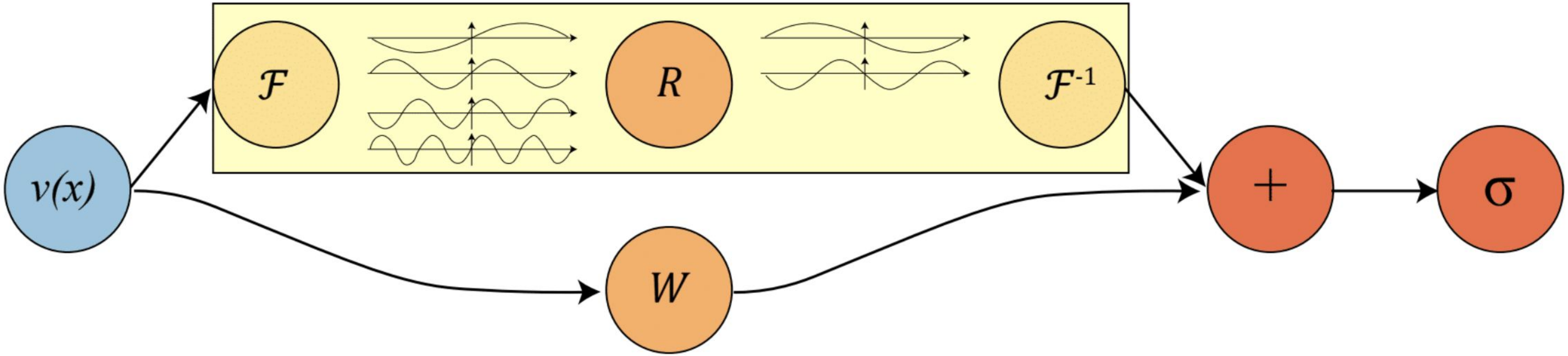


MGKN

https://arxiv.org/abs/2003.03485

https://arxiv.org/abs/2006.09535 (Neurips2020)

3. Fourier neural operator

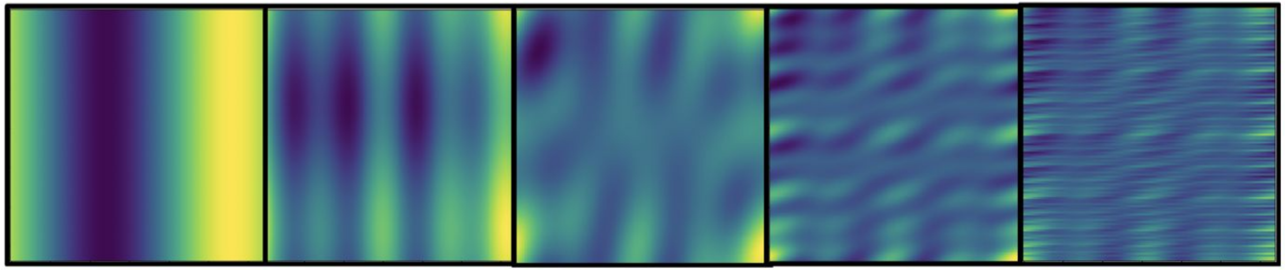


https://arxiv.org/abs/2006.09535 (ICLR2021)

Fourier filters

Fourier representation is more efficient than CNN.

Filters in CNN



Fourier Filters

Fourier layer

Use convolution as the integral operator
and implement with Fourier transform

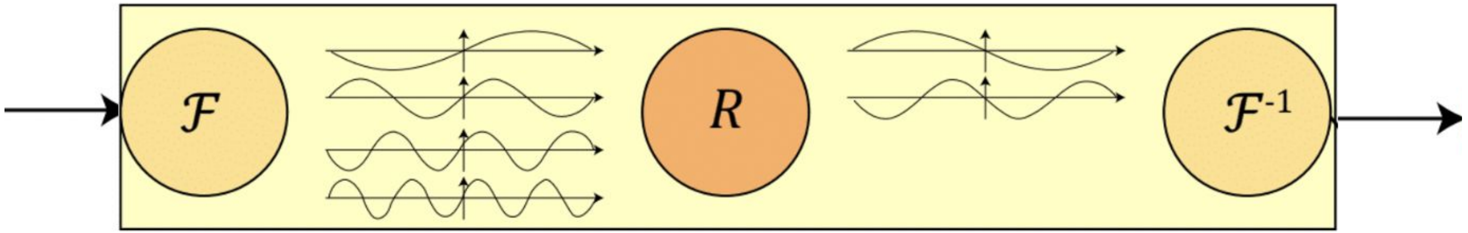
$$\big(\mathcal{K}(a;\phi)v_t\big)(x) := \int_D \kappa\big(x, y, a(x), a(y); \phi\big)v_t(y)\mathrm{d}y,$$

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

Fourier layer

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 layer

```
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
```

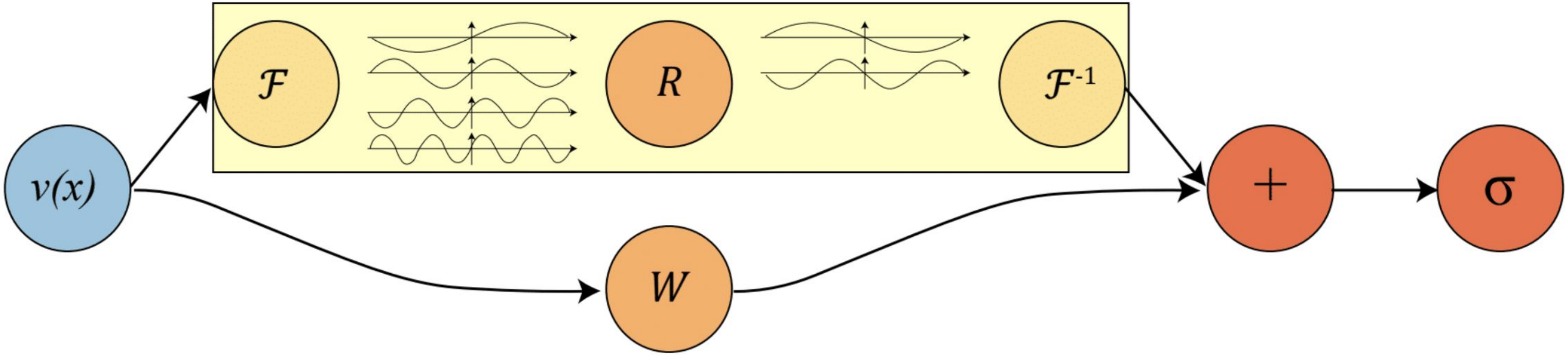
Fourier layer

Encoding & decoding
Activation function on the spatial domain

Recover high frequency modes

Fourier layer

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



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

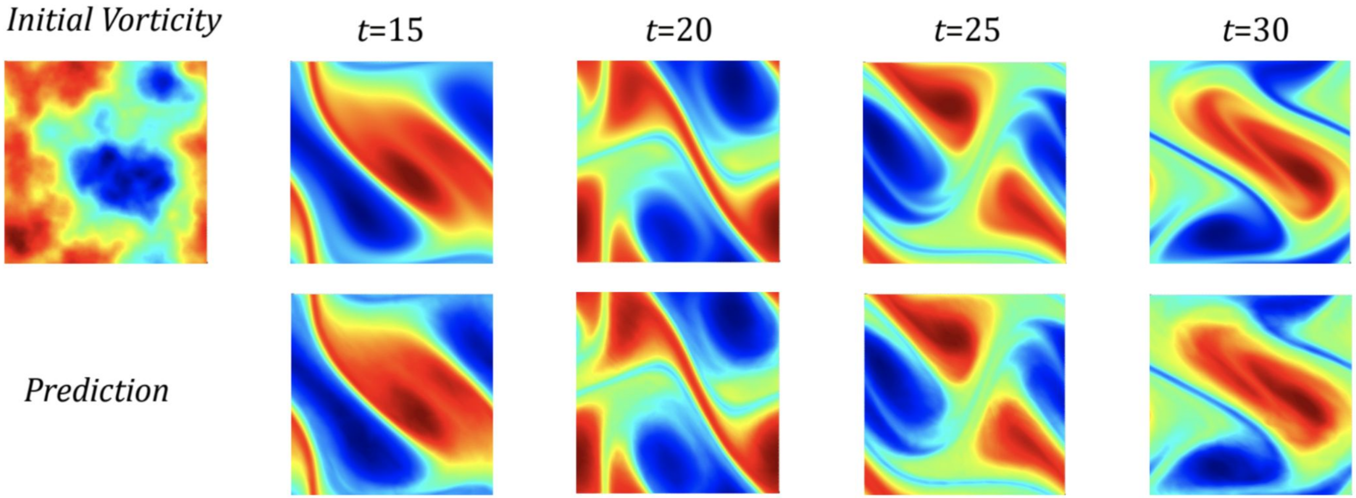
Fourier layer

Complexity:

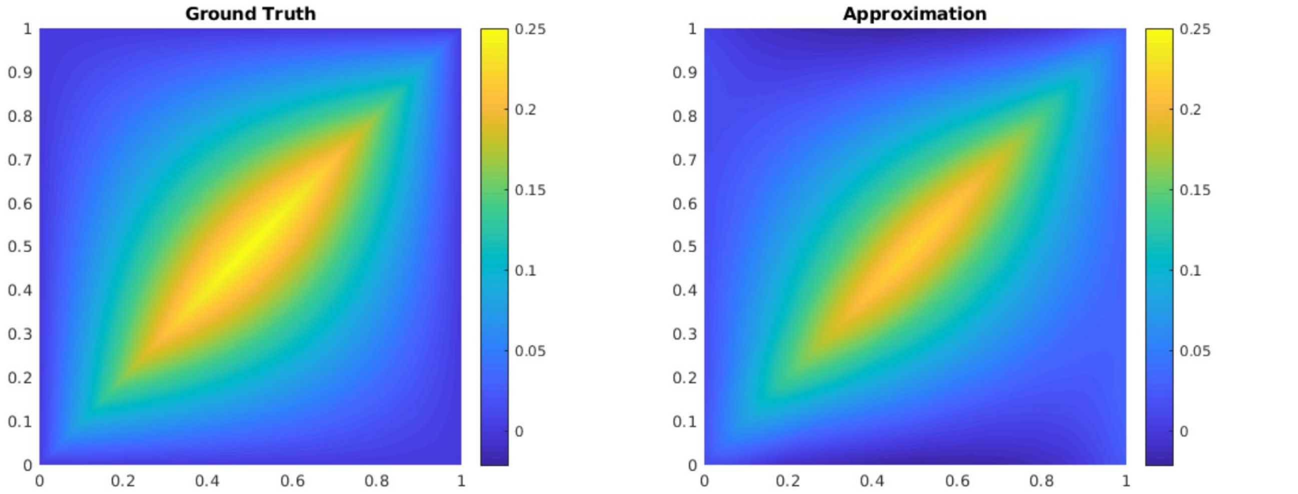
- Fourier transform O(k n)
- FFT O(nlogn)
- Linear O(n)

Resolution-invariant
Mesh-invariant

4. Experiments



Example 1: 1d-Poisson



Sanity check: the learned neural network kernel is very closed to the true analytic kernel

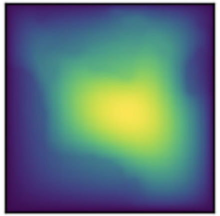
Example 2: 2d Darcy Flow

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

$$u(x) = 0 \qquad x \in \partial(0,1)^2$$

Input: coefficient

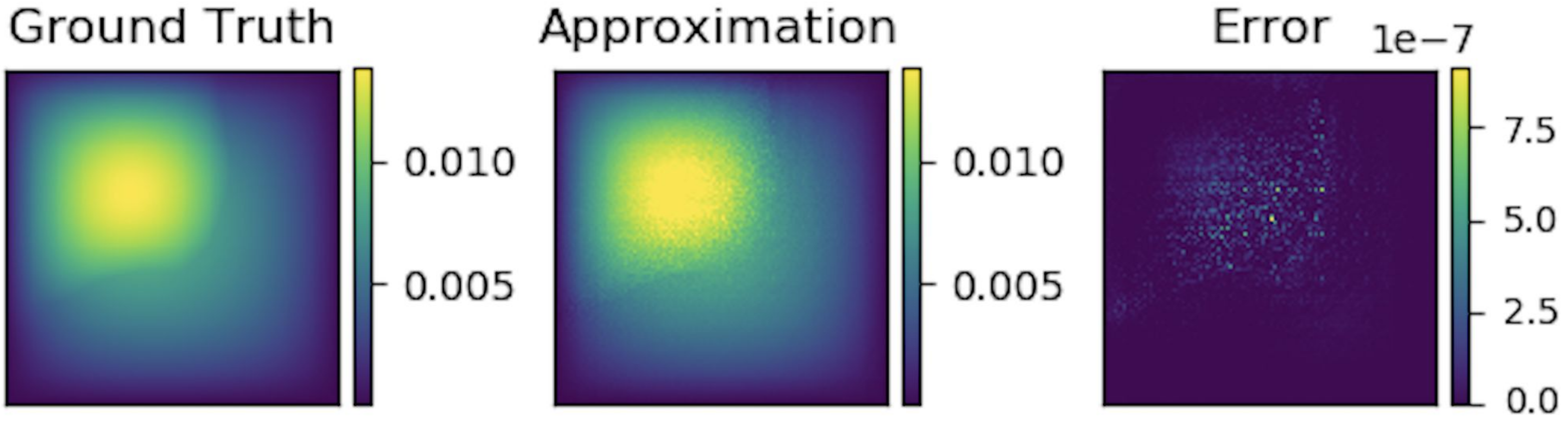
$\longrightarrow$



Output: solution

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

Train on 16*16, test on 241*241

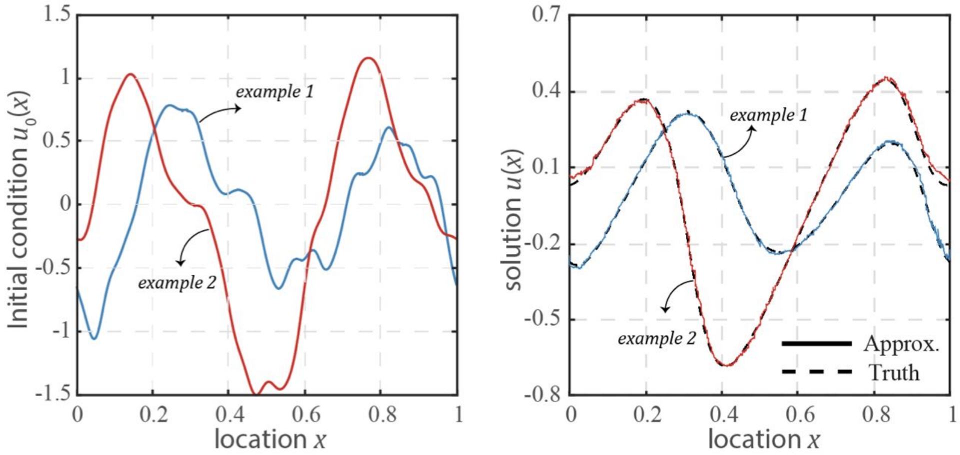


(Plot for the absolute squared error.
Average relative l2 error ~ 0.05)

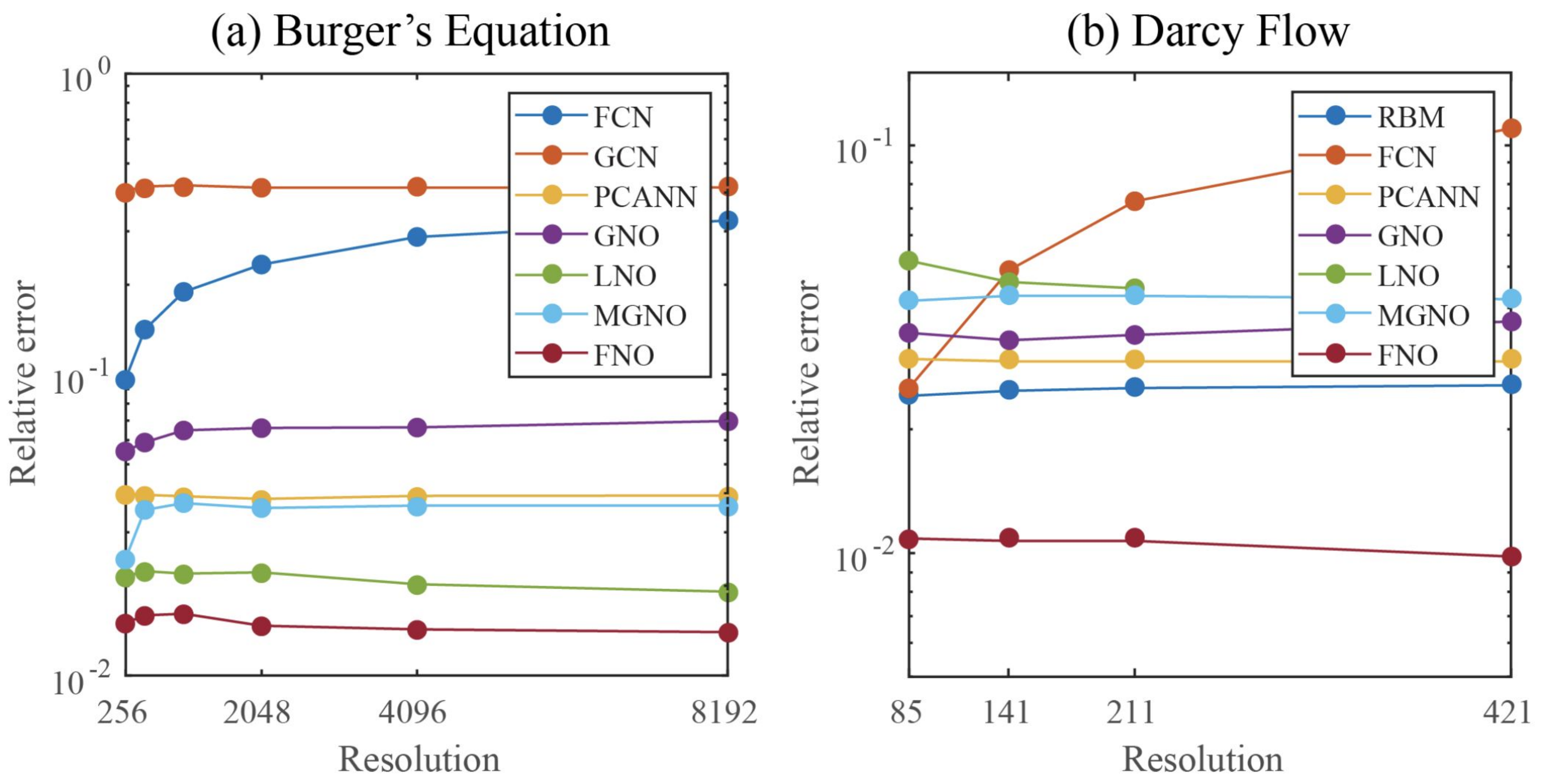
Graph kernel network does super-resolution

Example 3: 1d Burgers

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



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



(a) Burger's Equation
Relative error
Resolution
256
2048
4096
8192
10^{0}
10^{-1}
10^{-2}
FCN
GCN
PCANN
GNO
LNO
MGNO
FNO
(b) Darcy Flow
Relative error
Resolution
85
141
211
421
10^{-1}
10^{-2}
RBM
FCN
PCANN
GNO
LNO
MGNO
FNO

Example 4: Navier-Stokes

$$\partial_t w(x,t) + u(x,t) \cdot \nabla w(x,t) = \nu \Delta w(x,t) + f(x), \qquad x \in (0,1)^2, t \in (0,T]$$
$$\nabla \cdot u(x,t) = 0, \qquad x \in (0,1)^2, t \in [0,T]$$
$$w(x,0) = w_0(x), \qquad x \in (0,1)^2$$

$$f(x) = 0.1(\sin(2\pi(x_1 + x_2)) + \cos(2\pi(x_1 + x_2)))$$

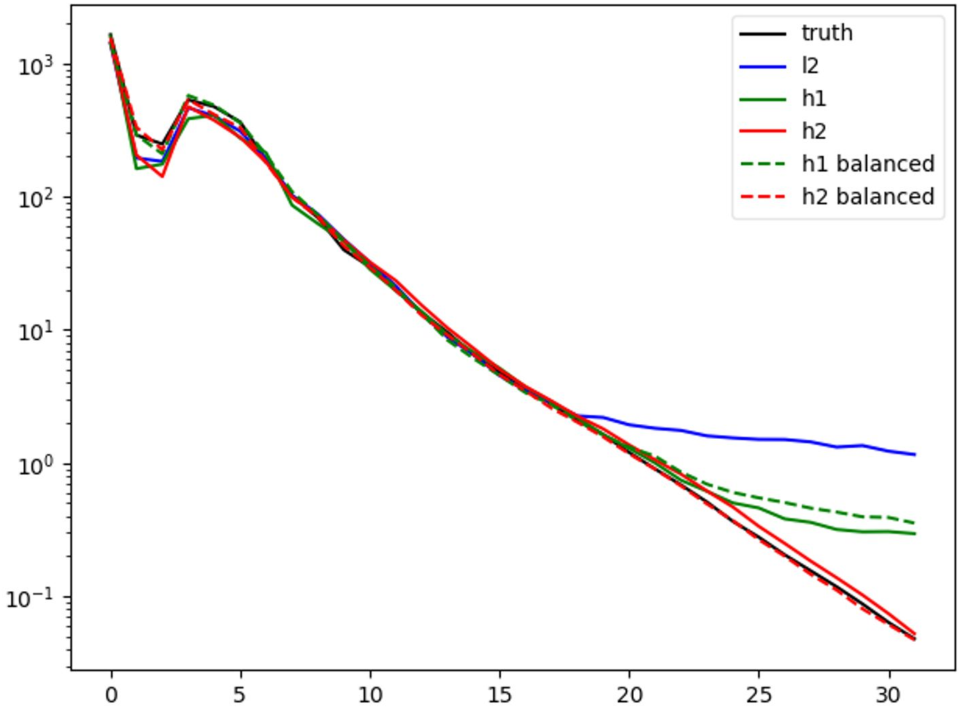
$$w_0 \sim \mu \text{ where } \mu = \mathcal{N}(0, 7^{3/2}(-\Delta + 49I)^{-2.5})$$

viscosities $\nu = 1\text{e}{-3}, 1\text{e}{-4}, 1\text{e}{-5}$

Example 4: Navier-Stokes

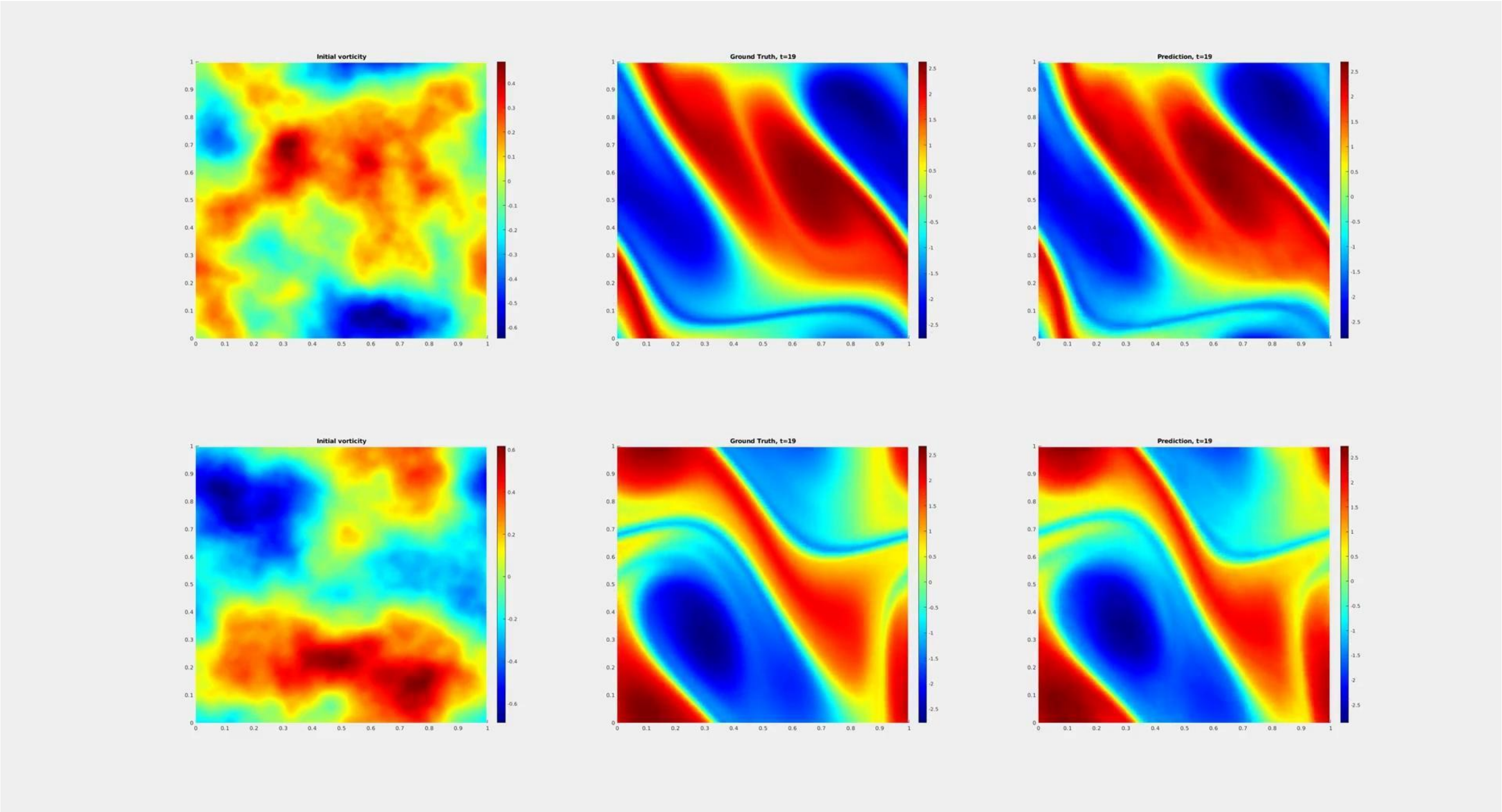
Config	**Parameters**	**Time per epoch**	$\nu = 1e{-}3$ $T = 50$ $N = 1000$	$\nu = 1e{-}4$ $T = 30$ $N = 1000$	$\nu = 1e{-}4$ $T = 30$ $N = 10000$	$\nu = 1e{-}5$ $T = 20$ $N = 1000$
FNO-3D	$6,558,537$	$38.99s$	**0.0086**	0.1918	**0.0820**	0.1893
FNO-2D	$414,517$	$127.80s$	0.0128	**0.1559**	0.0973	**0.1556**
U-Net	$24,950,491$	$48.67s$	0.0245	0.2051	0.1190	0.1982
TF-Net	$7,451,724$	$47.21s$	0.0225	0.2253	0.1168	0.2268
ResNet	$266,641$	$78.47s$	0.0701	0.2871	0.2311	0.2753

Energy spectrum

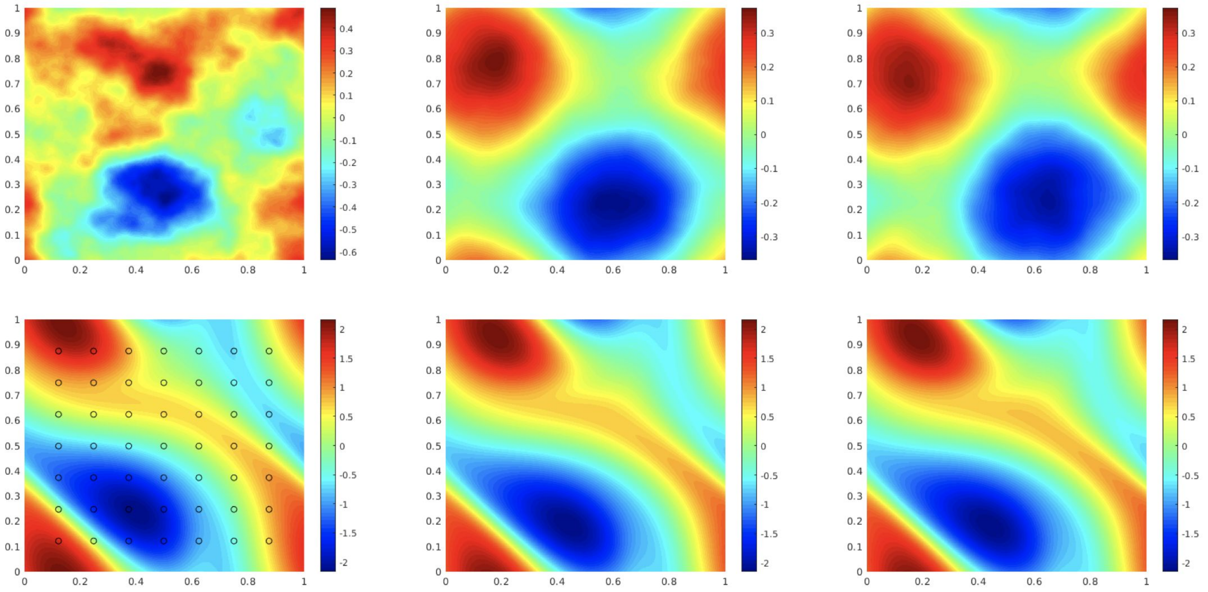


Train with derivatives (Sobolev norm) helps recover the higher frequencies.

V=1e-4, zero-shot super-resolution



Example 5: Bayesian inverse problem:



We a MCMC method, sampling initial conditions and evaluating them with the traditional solver and Fourier operator. The Fourier operator takes **0.005s** to evaluate each initial condition, while the traditional solver takes **2.2s**.

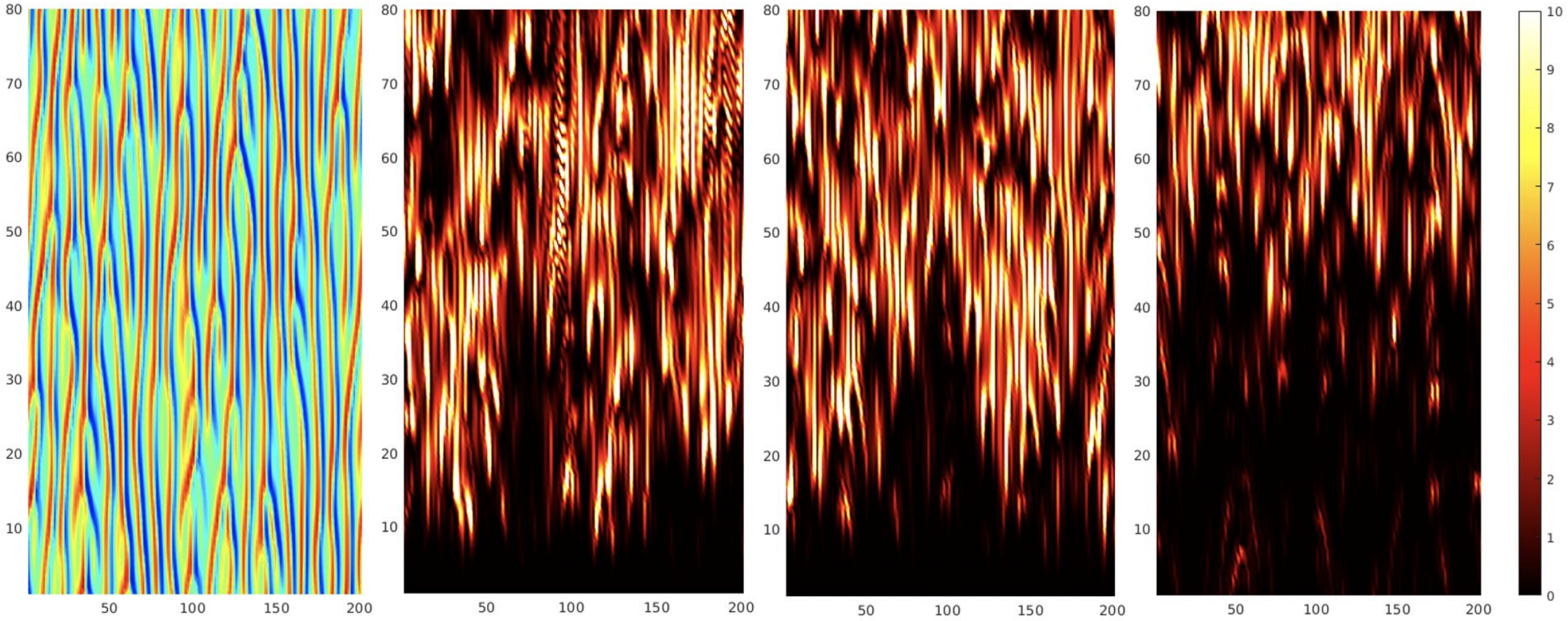
Example 6: KS equation

$$\partial_t u = -u\partial_x u - \partial_{xx} u - \partial_{xxxx} u,$$
$$u(\cdot, 0) = u_0,$$

1-d Kuramoto-Sivashinsky equation. Use neural operator to learn the update/residual. Compose the operator to reach for long time.

$$G : u(t) \mapsto u(t + \mathrm{d}t)$$
$$u(n \cdot \mathrm{d}t) \approx \underbrace{(\hat{G}_{\mathrm{d}t} \circ \cdots \circ \hat{G}_{\mathrm{d}t})}_{n \text{ times}}(u_0)$$

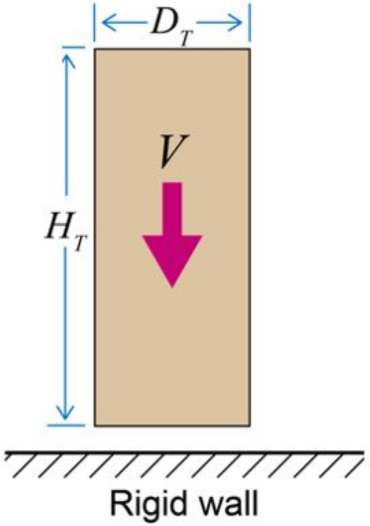
Example 6: KS equation



(a) ground truth (b) error of LSTM (c) error of GRU (d) error of FNO

Neural operator captures the invariant measures of chaotic system

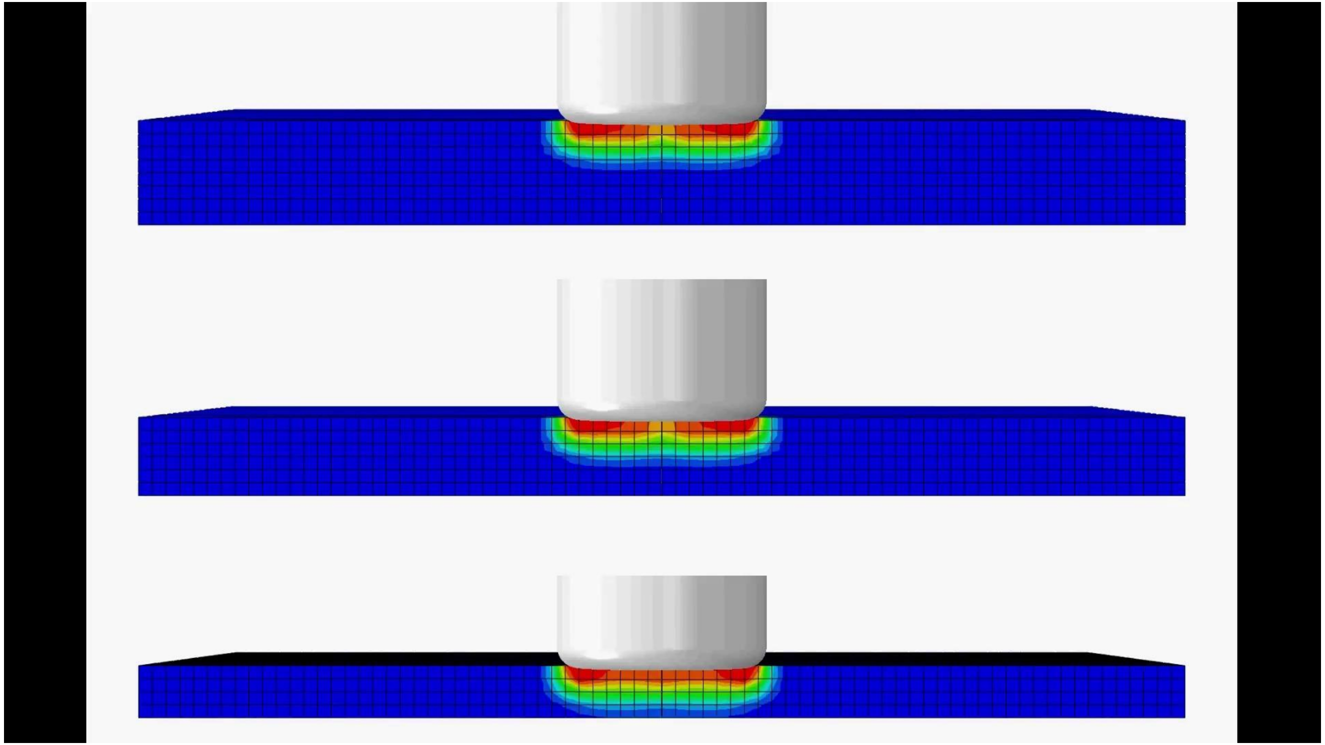
Example 7: Plasticity



$$\nabla \cdot S^\varepsilon = \rho^\varepsilon u^\varepsilon_{tt}$$
$$K^\varepsilon = 0$$
$$u^\varepsilon(x,0) = u_0(x), \quad u^\varepsilon_t(x,0) = v_0(x), \quad \xi^\varepsilon(x,0) = \xi_0$$
$$u^\varepsilon(x,t) = u^*(x,t)$$
$$S^\varepsilon(\nabla u^\varepsilon, \xi^\varepsilon, x)n(x) = s^*(x,t)$$

Multi-scale method: use neural operator to map from strain to stress on the unit cell; update macroscale with Abaqus solver.

Example 7: Plasticity



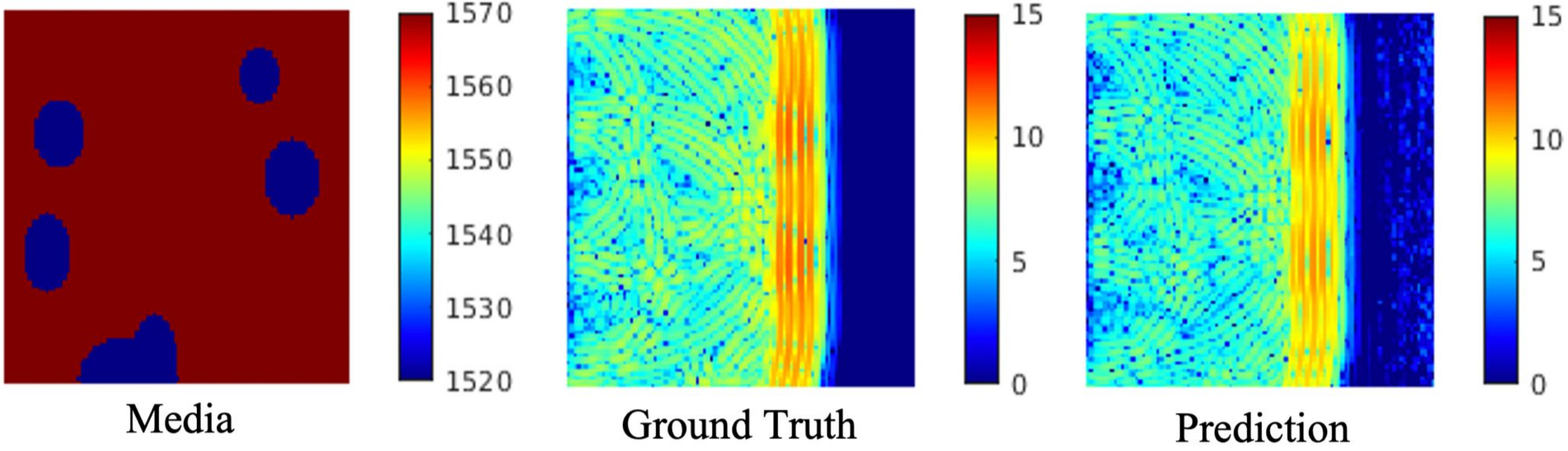
PCA-operator solves multi-scale plasticity problem (Burigede et. al.)

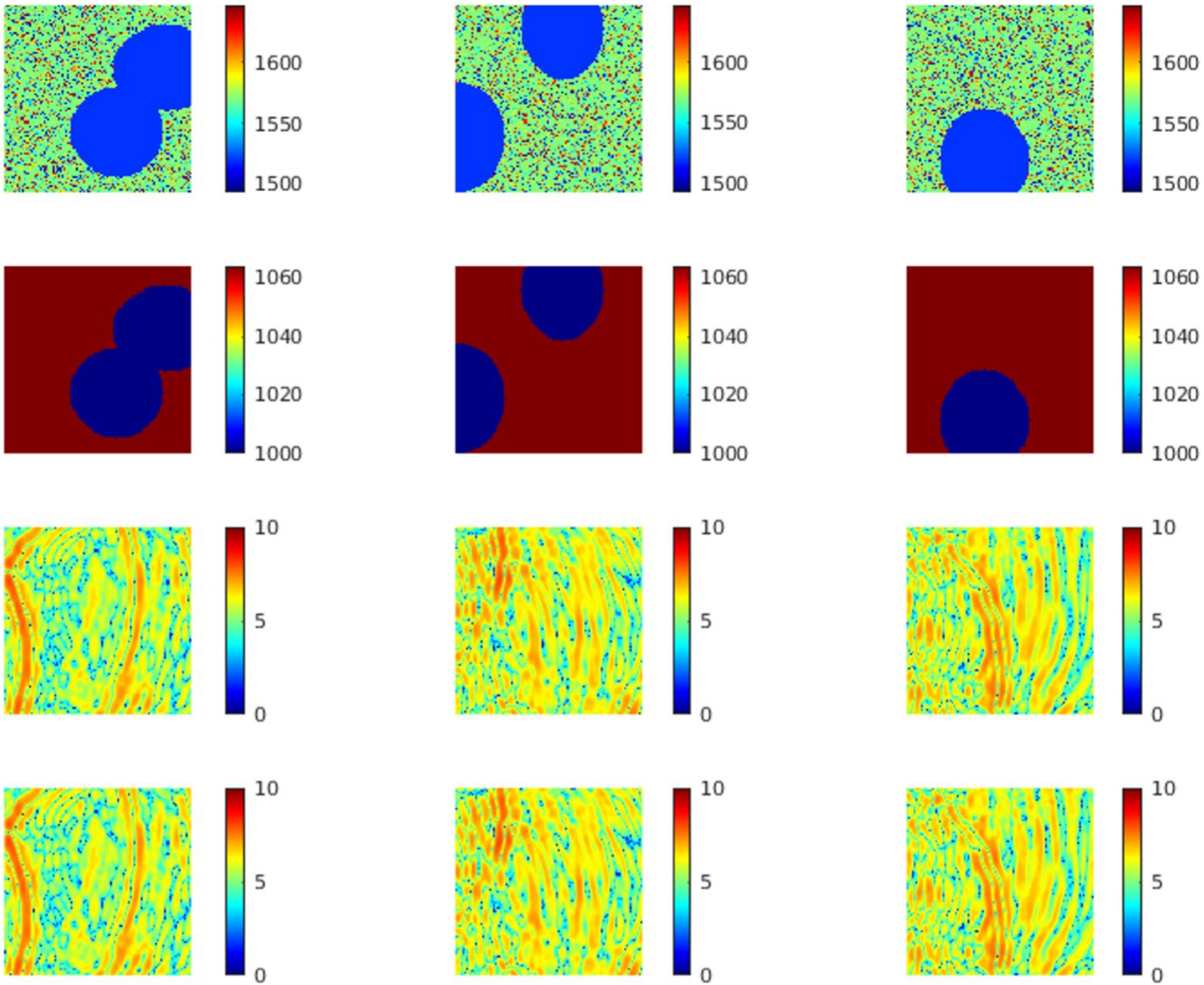
Example 7: Plasticity

Time complexity: neural operator is 10^5 faster

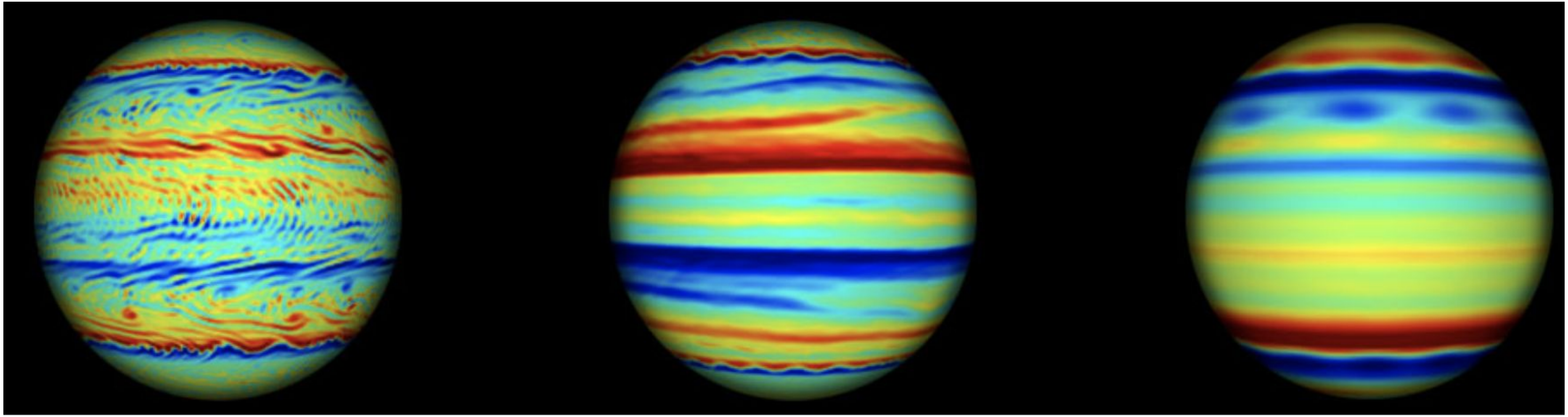
- PCA-operator:
  - 10^6 generate data
  - 10^4 training
  - 10^3 inference
- Taylor averaging
  - 10^8 (est.) to solve
- Full multi-scale simulation
  - 10^12 (est.) to solve

Example 8: Ultrasound




1600
1550
1500
1060
1040
1020
1000
10
5
0

5. Future work



Future work

1. New applications:
   Any problems that admit a fair Fourier expansion
   Replace the pseudo-spectral solvers / CNN / Unets

- Chaotic dynamics
- Geology
- Magneto Hydrodynamic (MHD)
- 3D Navier-stokes

Future work

2. Hybrid solvers

- Physics-informed/constraint setting
- Solver in the loop
- Neural ODE

Takeaway

1. Data-driven method: learn the equation
2. Operator-learning: parameterize the mesh-invariant operator
3. Fourier method: efficient for continuous inputs and outputs
4. Results: accurate than other deep learning method, faster than conventional solvers
5. Future work: combine with solvers. Scale up.

Reference

Arxiv:
https://arxiv.org/abs/2003.03485
https://arxiv.org/abs/2006.09535
https://arxiv.org/abs/2010.08895

Code:
https://github.com/zongyi-li/graph-pde
https://github.com/zongyi-li/fourier_neural_operator

Blog posts:
https://zongyi-li.github.io/blog/2020/graph-pde/
https://zongyi-li.github.io/blog/2020/fourier-pde/