

ECE 228 Spring 2026 Lecture 13: Neural Operators Part 1

Luke Bhan

May 12, 2026

Neural ODEs introduced us to this idea of learning continuous representations. How can we take this further?

Recall that Neural ODEs learn the flow field f of a dynamical system

$$\frac{dz(t)}{dt} = f(z(t), t; \theta), \quad z(0) = \bar{z}_0.$$

This is the first type of continuous learning representation we have discussed. However, there's more to be explored here. NeuralODEs require us to build a set of points $t \in [0, T]$ to learn the solution of which we use the final output time T .

Motivating question: How can we design neural architectures that we can query at any time (or for PDEs, any space) location?

First a little history: How do we simulate PDEs (See homework 3)

- ① **Start with a PDE:**

$$\frac{\partial u}{\partial t} = \mathcal{F}(u, \nabla u, \nabla^2 u, \dots)$$

This represents the evolution of a physical quantity u .

- ② **Discretize Space and Time:** Replace continuous space/time with a grid:

$$x_0, x_1, \dots, x_N \quad \text{and} \quad t_0, t_1, \dots, t_M$$

- ③ **Approximate Derivatives:** Use finite differences, finite volumes, or finite elements to approximate spatial and temporal derivatives.
- ④ **Step Forward in Time:** Update the solution using a time-stepping method (e.g., Euler, Runge-Kutta).
- ⑤ **Repeat:** Continue iterating until the final time is reached.

History: Key limitation of numerically simulating PDEs

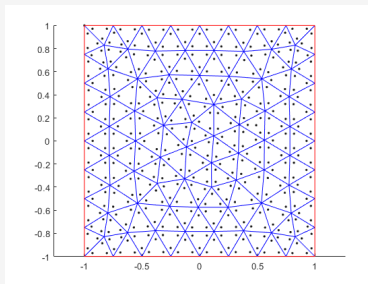


Figure: Example of a triangle mesh for a PDE.

- **Fixed Resolution:** Simulations are restricted to predefined spatial and temporal grid points. Fine grids increase accuracy but also computational cost and can lead to millions of simulation points
- **Poor Generalization:** Traditional solvers compute solutions only at fixed parameters. New parameters require *re-running the simulation from scratch*.

How do we *classically* solve PDEs?

Take the simplest PDE (1D transport):

$$\begin{aligned}\frac{\partial u(x, t)}{\partial t} &= \frac{\partial u(x, t)}{\partial x}, & x \in (0, 1), t \in \mathbb{R}^+, \\ u(x, 0) &= f(x), \\ u(1, t) &= 0.\end{aligned}$$

This PDE just shifts the initial condition with space and time.

Example of a transport PDE

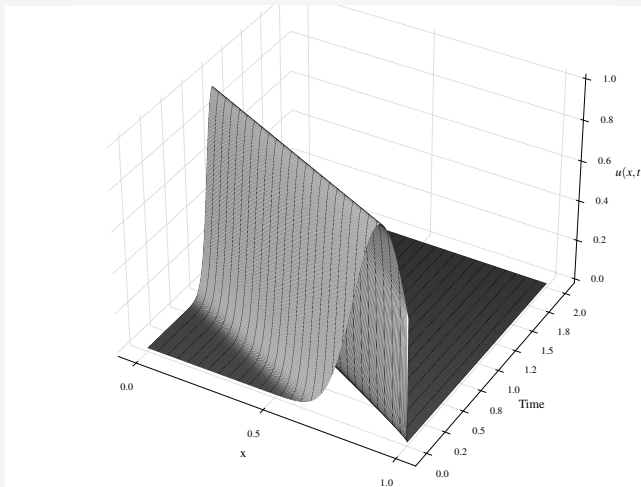


Figure: Transport PDE with Gaussian initial condition centered at $x = 0.9$.

How do we *classically* solve PDEs?

Can apply Euler's method:

$$\frac{\partial u(x, t)}{\partial t} \approx \frac{u(x, t + \delta t) - u(x, t)}{\delta t} + \mathcal{O}(\delta t),$$
$$\frac{\partial u(x, t)}{\partial x} \approx \frac{u(x + \delta x, t) - u(x, t)}{\delta x} + \mathcal{O}(\delta x),$$

Combining yields the **finite difference** scheme

$$u(x, t + \delta t) = u(x, t) + \frac{\delta t}{\delta x} (u(x + \delta x, t) - u(x, t))$$

where we impose the conditions

$$u(x, 0) = f(x),$$

$$u(t, 0) = 0.$$

Then, to simulate until say $T = 5$, we just recursively run the above calculation until our desired timestep is reached.

PDE solvers get very expensive very quickly

Unlike Euler's method for ODEs, PDE solvers become computationally costly extremely quickly.

- This is due to the fact we have both a spatial and temporal component, and thus we need to solve this PDE across both space and time.
- Furthermore, As with ODE's we require the step sizes δx and δt to be small. A reasonable choice is $\delta x = 0.01$ and $\delta t = 0.0005$ (for most numerical schemes $\delta t \ll \delta x$ as discussed in your homework).
- Then, to solve to $T = 5$, we require $(1/\delta x + 1) \times (5/\delta t + 1) = 1,010,101$ points of evaluation.

Solving computationally becomes much worse as dimension scales

Consider the Navier Stokes PDE in 2D.

- For 1D, to solve to $T = 5$, we require $(1/\delta x + 1) \times (5/\delta t + 1) = 1,010,101$ points of evaluation.
- For 2D, to solve to $T = 5$, we require $(1/\delta y + 1) \times (1/\delta x + 1) \times (5/\delta t + 1) = 102,020,201$ points.
- For 3D, for $T = 5$, we require ten billion points which is not possible on everyday machines.



Figure: Sierra supercomputer in Livermore California.

How can we design a machine learning model that can learn PDE solutions and be queried at any grid location?

Answer: Design a machine learning model using *continuous representations*.

To do so, we need to introduce the mathematical notion of an **operator**. One reasonable viewpoint is to look at an operator as an abstraction of functions. That is, functions map vectors to vectors:

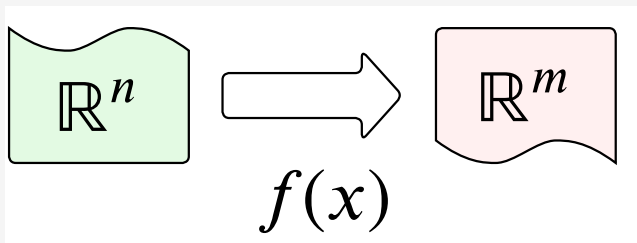


Figure: The function f take an input vector $x \in \mathbb{R}^n$ and maps to an output vector in \mathbb{R}^m .

What is an operator?

Now, imagine we considered a set of functions, for example all polynomials of degree 2 or all sin functions of the form $f(x) = A\sin(Bx + C)$ where A, B, C can vary. Then, like a function that maps vectors to vectors, an operator maps *functions to functions*:

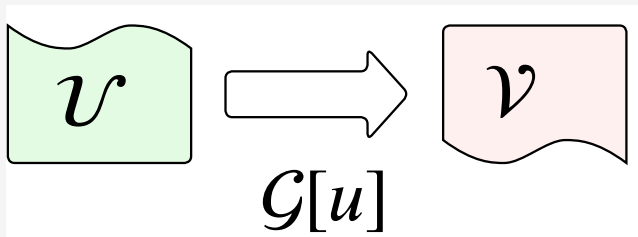


Figure: Let \mathcal{U} be a set of functions and \mathcal{V} be another set of functions. Then the operator \mathcal{G} takes a function in $u \in \mathcal{U}$ and maps it to a function $v \in \mathcal{V}$.

Examples of operators

You already know many types of operators. Perhaps, the most famous are:

- The integral operator. Let $f \in \mathcal{F}$ and $g \in \mathcal{G}$, then the integral acts as

$$\mathcal{G}[f](y) = \int_0^y f(x)dx .$$

- Analogously the derivative of a function is an operator.
- Another operator is the shift operator. Given a parameter c , it acts as:

$$g(x) = \mathcal{G}[f](x) = f(x + c)$$

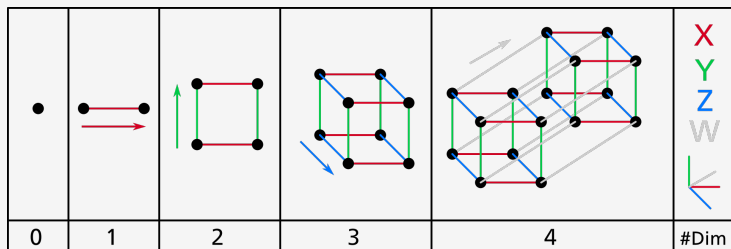
- What other operators can you think of? (Think of recent neural network architectures)

An important property of vector spaces

In vector spaces, we had this important structure of dimension which essentially told us:

- We need exactly d linearly independent vectors to create any vector in the space (recall this is a basis)

This essentially means that our vector field has d degrees of information or one can think of this as d values need to specify the object.



Operators typically make function spaces, which in many cases have ∞ dimension

Consider the set of all continuous functions \mathcal{C} on $[0, 1]$.

Using the fact that dimension says we need d number of linearly independent objects to build any object in the space, what dimension do you think \mathcal{C} is?

Answer: \mathcal{C} is infinite dimensional! This will be an important property when we consider operator mappings, as we *cannot* write down an infinite dimensional object in the form of a finite number of basis vectors! This will become important in our neural network design.

Approximation of operators

Neural networks approximate functions. Neural operators approximate operators.

What does it mean to approximate an operator?

Main questions:

- How do we input a function f which is inherently continuous and (sometimes) a member of an infinite dimensional set into a computer?
- Say we can do this, what does it mean to get an output function from our neural network?
- Why would anyone want to do this?

Let's answer the third question before moving on. Do you see any advantages?

Why do we care about approximating operators?

The solution to dynamical systems - both ODEs and PDEs is inherently an *operator*. For example consider ODEs

$$\frac{dz(t)}{dt} = f(z(t), t), \quad z(0) = \bar{z}_0.$$

Then, one can write the solution as

$$z(t) = \int_0^t f(z(s), s) ds + \bar{z}_0.$$

The solution to an ODE is given by an **operator**!

Thus, imagine we want to learn $z(t)$ for any input \bar{z}_0 . Then, if we can learn this implicit integral operator, we have a solution to the ODE for all times t with all initial conditions \bar{z}_0 removing the expensive numerical schemes (Euler's, RK4, etc.)

Perhaps you think ODEs are too simple

Consider the holy grail of PDE's: Navier-Stokes

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f},$$
$$\nabla \cdot \mathbf{u} = 0,$$

where:

- \mathbf{u} is the velocity field
- p is the pressure field
- ρ is the fluid density (constant for incompressible flow)
- μ is the dynamic viscosity
- \mathbf{f} represents body forces (e.g., gravity)

First, if you can just prove that a solution exists congrats you're a millionaire.^a

^ahttps://en.wikipedia.org/wiki/Millennium_Prize_Problems

Operator formulation of Navier-Stokes

Given an initial velocity field at time 0, $\mathbf{u}(\mathbf{x}, 0) = \mathbf{g}(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^n$. Then we can write the solution operator as

$$\mathbf{u}(\mathbf{x}, t) = \mathcal{G}[\rho, \rho, \mu, \mathbf{f}, \mathbf{g}](\mathbf{x}, t)$$

which takes an input of $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times C^0(\mathbb{R}^n \times \mathbb{R}^0; \mathbb{R}^n) \times C^0(\mathbb{R}^n; \mathbb{R}^n)$ to an output of some function \mathbf{u} (of which we don't know if it's smooth. See Millennium problems).

- Notice, for PDEs, the operator learned is a mapping of function spaces.
- As with the ODE problem, if we can learn an approximation of this operator, then we obtain the Navier-Stokes equations at any desired \mathbf{x}, t . The grid independence or otherwise known as **discretization invariance** will enable us to perform calculations that are otherwise impossible with traditional solvers.

What is the operator input *in practice*?

The abstract formulation

$$\mathbf{u}(\mathbf{x}, t) = \mathcal{G}[p, \rho, \mu, \mathbf{f}, \mathbf{g}](\mathbf{x}, t)$$

takes *five* different functional/scalar inputs. In practice, we fix most and learn the operator over **one** varying functional input.

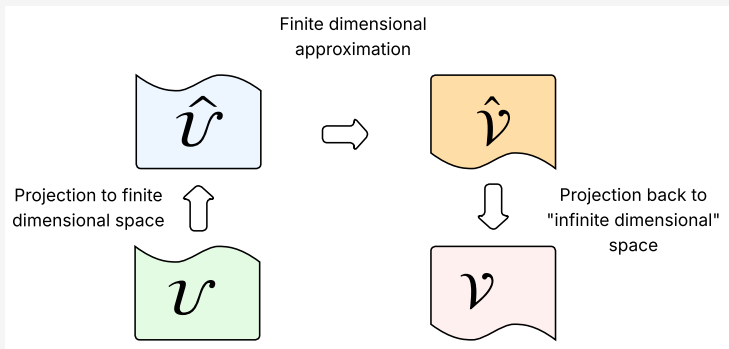
- **Time-evolution problems:** Fix p, ρ, μ, \mathbf{f} . Learn $\mathbf{g}(\mathbf{x}) \mapsto \mathbf{u}(\mathbf{x}, t)$.
(*Most common setup.*)
- **Parametric PDEs:** Fix the initial condition. Learn $\mu(\mathbf{x}) \mapsto \mathbf{u}(\mathbf{x}, t)$
— predict solutions for varying viscosity fields.
- **Source-term / inverse problems:** Learn $\mathbf{f}(\mathbf{x}, t) \mapsto \mathbf{u}(\mathbf{x}, t)$ to recover hidden forcing.
- **Geometry generalization:** Learn the operator over varying domain shapes (e.g., airfoil design).

The neural operator \mathcal{G}_θ becomes a fast *surrogate* for “solve the PDE for this input.”

So why learn an approximation of an operator

It gives us a framework to develop machine learning solutions to ODE and PDEs that can be queried at any desired x, t location!

So what does this look like in practice? **Answer:** There are a variety of approaches, but they all follow the same structure.



Let's walk through an example: Transport PDE

Consider the 1D transport PDE we talked about earlier:

$$\begin{aligned}\frac{\partial u(x, t)}{\partial t} &= \frac{\partial u(x, t)}{\partial x}, & x \in (0, 1], t \in \mathbb{R}^+, \\ u(x, 0) &= f(x), \\ u(0, t) &= 0.\end{aligned}$$

Given, $f(x)$, we want to know the solution of the PDE from time $T = 3$ to $T = 5$. That is, we want to learn the operator mapping

$$f(x) \mapsto u(x, t), \quad t \in [3, 5], x \in [0, 1].$$

Step 1: How do we encode $f(x)$ (project $f(x)$ to finite dimensional space)?

Encoding: Simplest approach.

Consider points $x_1, x_2, \dots, x_n \in [0, 1] = X$. Simplest case is a uniform spacing $x_1 = 0, x_2 = 0.1, x_3 = 0.2, \dots, x_n = 1$. One possible encoding of $f(x)$ is to evaluate the function at these points:

$$[f(x_1), f(x_2), \dots, f(x_n)].$$

More commonly, one extends this by a classical feedforward neural network with $[f(x_1), \dots, f(x_n)]$ as its input (sometimes, this projects to a *higher dimensional space* - you can think of the NN as performing interpolation between the x_k points).

Key: This introduces error in our approximation as we can't pass the true functional form, but this is a representation which can be passed into a computer as a vector.

How to do a finite dimensional approximation of an operator

Let a typical MLP for a neural network be given by

$$f(x) = \sigma(Wx + b).$$

Then, a neural operator approximation is given by

$$\mathcal{L}(f)(x) := \sigma(Wf(x) + b) + (\mathcal{K}_I f)(x)$$

where

$$(\mathcal{K}_I f)(x) = \int_{y \in X} K_I(x, y) f(y) dy.$$

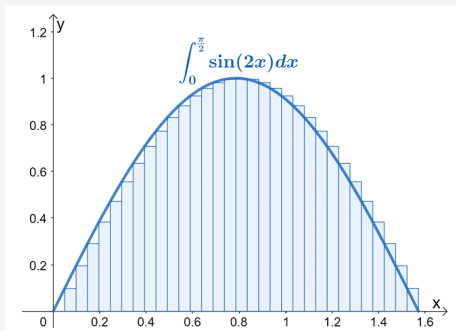
So what is new: Essentially, we added this **non-local** integral term which now depends on every point in the space and multiplies by the weight matrix K_I .

Why do we need a nonlocal term?

Simple example. Consider the integral operator

$$\mathcal{G}[f](y) = \int_0^y f(x) dx.$$

Assume we have no analytical form of this integral. Then, evaluating $\mathcal{G}[f](y)$ depends on all the points between $[0, y]$ - not just the point y .



Hence operators are nonlocal and to approximate them, at minimum we need this extra term.

The kernel structure is also not new...

Write the neural operator layer with the kernel integral spelled out:

$$\mathcal{L}(f)(x) = \sigma\left(\underbrace{Wf(x) + b}_{\text{local (pointwise, MLP-like)}}\right) + \underbrace{\int_X K_I(x, y) f(y) dy}_{\text{non-local: mixes info across all of } X}$$

The choice of K_I recovers architectures you already know:

CNN — kernel depends only on the *offset* $x - y$:

$$K_I(x, y) = k_I(x - y) \implies \int k_I(x - y) f(y) dy = (k_I * f)(x)$$

Self-attention — kernel is computed *from the input* f :

$$K_I(x, y) \propto \exp(\langle W_Q f(x), W_K f(y) \rangle)$$

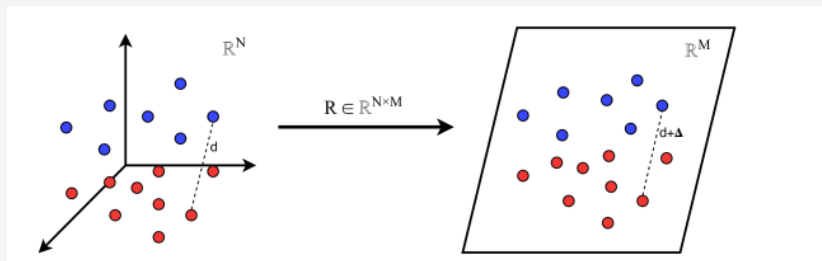
Mixing weights adapt to the data \implies “which y should x attend to?”

Takeaway: CNNs and Transformers are special cases of the operator layer — they just pick different families of kernels K_I .

Decoder structure

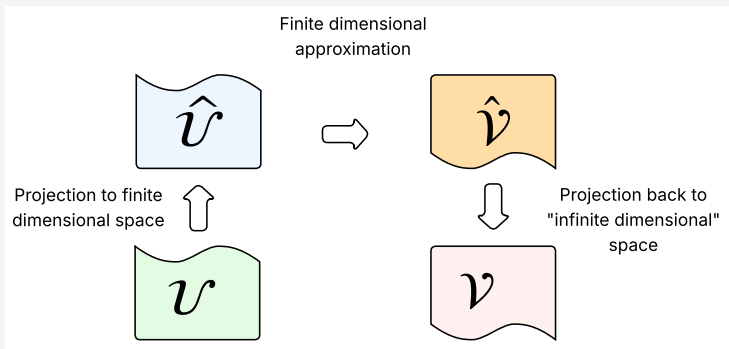
Take the output of the hidden layers and *project* back to the desired evaluation points.

How this projection is done can vary, and there are a variety of approaches (Example of one approach later today).



Theory recap: Neural Operators

- ① **Encoder:** Obtain some point-wise representation of our functional input.
- ② **Hidden layers:** Perform *Non-local* finite dimensional approximation with our kernel matrix
- ③ **Decoder:** Project back to desired grid locations (architecture dependent)



Practice: How do we actually train a neural operator?

We need pairs of (input function, solution function). Where do they come from?

- ① **Generate training data with a classical solver.** For $i = 1, \dots, N$:
 - Sample an input function $f^{(i)}(x)$
 - Run finite differences / FEM / spectral solver to get $u^{(i)}(x, t)$.
- ② **Supervised loss** at query points $\{(x_j, t_j)\}$:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \sum_j \left\| \mathcal{G}_\theta[f^{(i)}](x_j, t_j) - u^{(i)}(x_j, t_j) \right\|^2$$

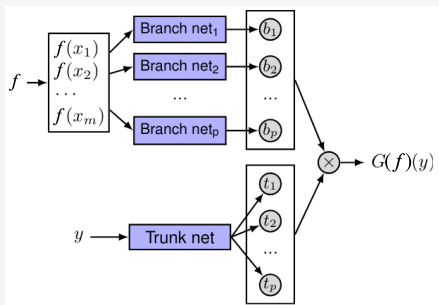
- ③ **Backprop** through \mathcal{G}_θ as usual.

Wait — if we need the solver to train, what did we gain?

Amortization. Pay the solver cost *once* during training (N runs). Inference is then milliseconds per new input — often 10^3 – $10^5 \times$ speedup.

Architecture Example: Deep Operator Networks (DeepONet)

- 1 Encoder: Standard Point-wise function evaluations
 $[f(x_1), \dots, f(x_n)]$
- 2 Finite dimensional approximation: Let us consider p different neural networks (MLP) that all take in $[f(x_1), \dots, f(x_n)]$ as their input and output β_1, \dots, β_p .
- 3 Decoder: To construct our target function v , we use a second NN that takes in the evaluation point y which is where to evaluate the target function and perform a linear combination.



DeepONet intuition: learn a basis for the output

Main idea: Use the finite-dimensional approximation to learn a *basis* for the target function, which can then be queried at any point.

Recall: any function admits a basis decomposition

$$u(y) \approx \sum_{i=1}^p c_i \phi_i(y).$$

For a Fourier series: $\phi_i(y) = \sin(i\pi y)$ and c_i are the Fourier coefficients.

DeepONet's twist: *both* pieces are learned from data, and the coefficients depend on the input function:

$$\mathcal{G}(f)(y) \approx \sum_{i=1}^p \underbrace{b_i([f(x_1), \dots, f(x_n)])}_{\text{input-dependent coefficients}} \times \underbrace{\tau_i(y)}_{\text{learned basis}}$$

DeepONet intuition: the two networks have distinct jobs

Branch net (b_i):

- Reads input function as samples.
- Outputs coefficients b_1, \dots, b_p .
- “Given input f , what is the right mixture?”

Trunk net (τ_i):

- Reads a query location y .
- Outputs $\tau_1(y), \dots, \tau_p(y)$.
- “What do the basis functions look like at y ?”

Super-resolution consequence: The trunk can be queried at *any* y without re-running the branch. Train on a coarse grid, evaluate on a fine grid — for free.

Connection to the theoretical definition of Neural Operators

Claim: DeepONet is the operator layer with a specific, simple kernel choice.

Start from the abstract operator layer and strip it down:

$$\mathcal{L}(f)(y) = \sigma(Wf(y) + b) + (\mathcal{K}_I f)(y) \xrightarrow{W=0, b=0, \sigma=\text{id}} \mathcal{L}(f)(y) = (\mathcal{K}_I f)(y)$$

The kernel term is an integral over the input domain X :

$$(\mathcal{K}_I f)(y) = \int_X \kappa(y, s) f(s) ds$$

Key step — choose a separable (low-rank) kernel:

$$\kappa(y, s) = \sum_{i=1}^p \tau_i(y) \psi_i(s)$$

DeepONet and Theoretical Neural Operator Connection

Key step — choose a separable (low-rank) kernel:

$$\kappa(y, s) = \sum_{i=1}^p \tau_i(y) \psi_i(s)$$

Plug it in and the integral splits into a sum:

$$(\mathcal{K}_I f)(y) = \int_X \left(\sum_{i=1}^p \tau_i(y) \psi_i(s) \right) f(s) ds = \sum_{i=1}^p \tau_i(y) \underbrace{\int_X \psi_i(s) f(s) ds}_{=: b_i(f)}$$

Each $b_i(f)$ is a scalar functional of f — exactly what the **branch net** learns from samples $[f(x_1), \dots, f(x_n)]$. Each $\tau_i(y)$ is what the **trunk net** outputs at query point y .

¹See Section 3.6 of Kovachki et al., *Nonlocal Neural Operator* for the full derivation.

Key features of DeepONet

- Can query the NN at any desired location y *without retraining* (Super-resolution). Branch and trunk NNs are independently computable.
- Easily extended where the two networks are of any form.
- Theoretical universal approximation guarantees.
- "Notion of interpretability" for the basis vectors.

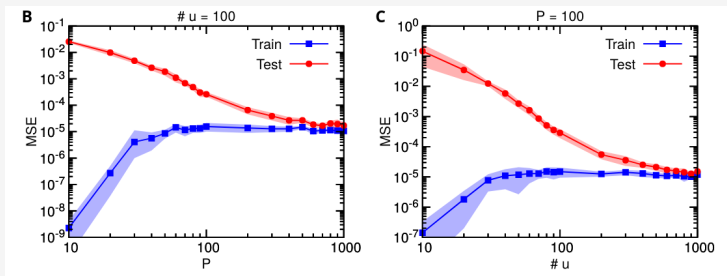


Figure: (Left) scaling with respect to number of evaluation points x_1, \dots, x_P . (Right) Scaling with respect to number of training data. Both for a reaction diffusion PDE.

What can you think of?

- The linear combination structure is less expressive. Do these types of dot products have issues in other architectures?
- Can struggle with generalization and be very sample inefficient due to its very generic architecture.
- Resolution mismatch - if your training data is given at uniform grid, will struggle to evaluate on non-uniform grids.

There have been a variety of extensions enhancing DeepONet.

- NOMAD - Generalizes the dot-product in DeepONet to be the input to a third NN (Paper link).
- Use DeepONet with pretrained encoder-decoder structure of functional evaluations for better encoding/decoding. (Paper link)
- Multiscale DeepONet that solves the non-uniform grid challenge (Paper link).
- Integration with RNNs for temporal dependence (Paper link)
- Uncertainty quantification ingrained DeepONets (Paper link)

Next time...

- Introduce other types of neural operator architectures (Fourier Neural Operator)

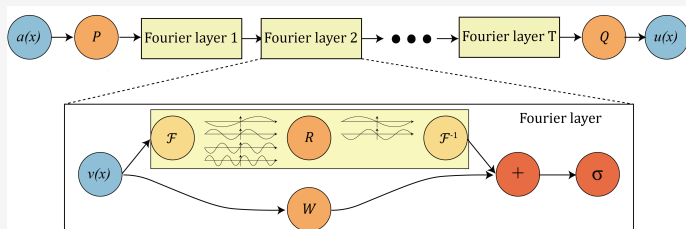


Figure: Fourier Neural Operator Architecture

- Discuss whether super-resolution really works?
- If time permits, a few examples of how these can be applied in climate modeling, robotics, and earthquakes (See your homework).