ECE 228 Spring 2026 Lecture 14: Neural Operators Part 2

Luke Bhan

May 14, 2026

Recap of last time

Introduced this notion of an operator which represented the solutions to many of our problems (ODEs, PDEs)

What is an operator?

Operators map spaces of functions to spaces of functions.

Give me an example of operators we did NOT talk about last class!

- Identity or zero operator
- Projection operator. (Project one function space onto another)
- Fourier operator (Performs Fourier transform).
- Laplace operator (Performs Laplace transform).

Key point: Many of the problems you already know can be reformulated as operators!

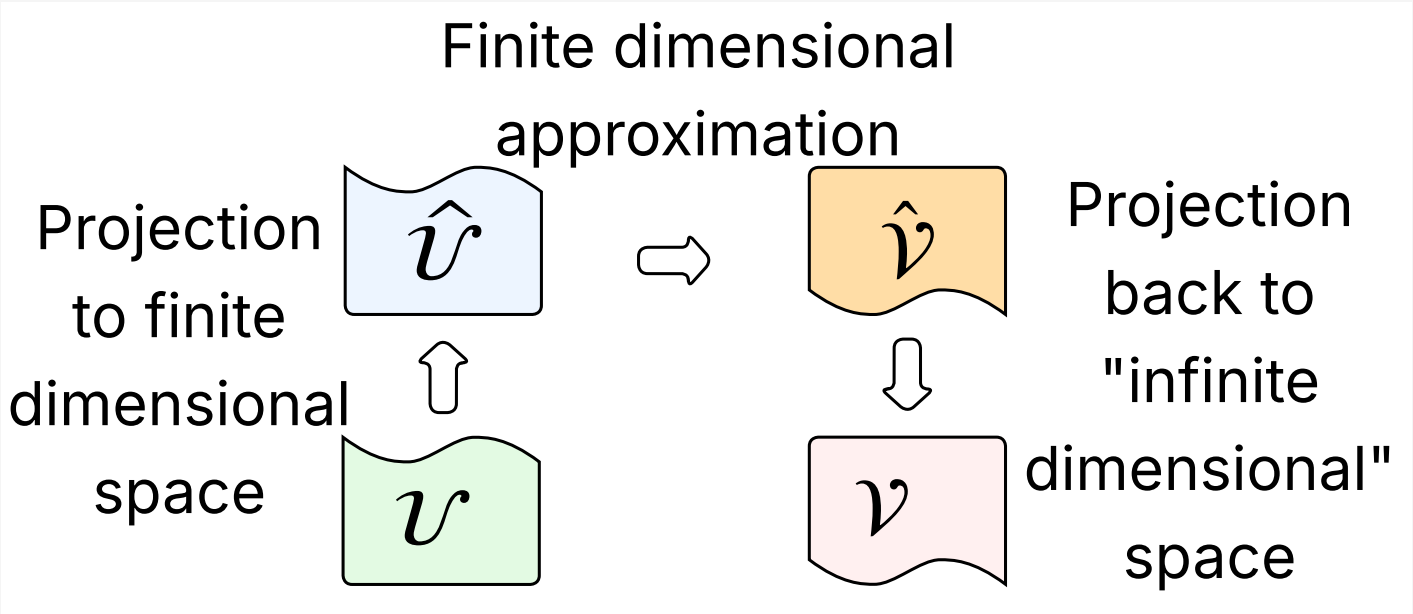
Generic neural operator structure

We discussed how to approximate operators with a "neural operator". It had three components to the structure: Can you remind us what they were?

1. Encoder: Need a way to represent the function $f(x)$
2. Finite dimensional approximation - has the **non-local** integral term

$$\mathcal{L}(f)(x) = \sigma\left(Wf(x) + b + \int_{y\in X} K_l(x,y)f(y)dy\right).$$

3. Decoder: Way to project our output to any desired point.



DeepONet Recap

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:

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

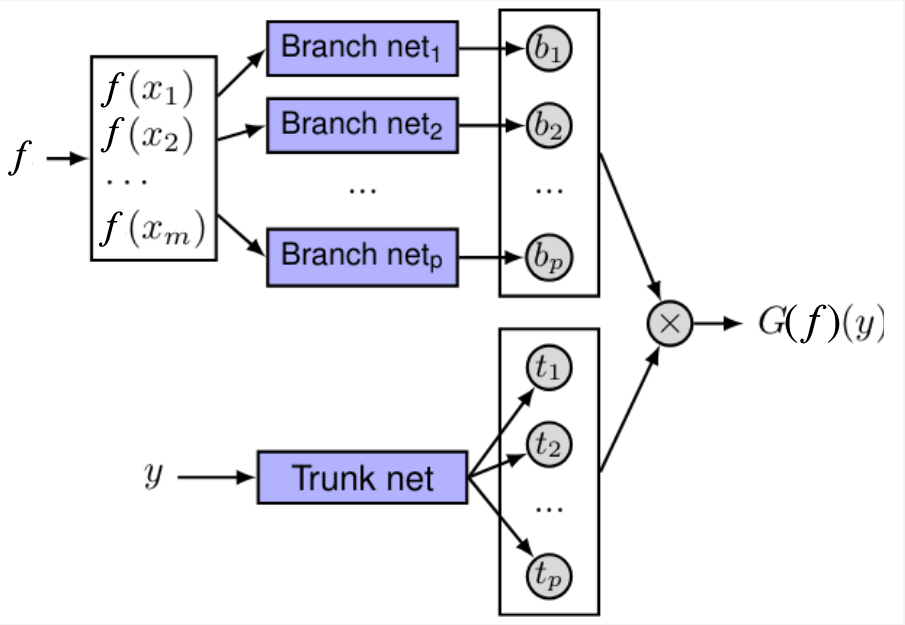
Branch and Trunk Networks

Branch net (b_i):

- Reads input function as samples.
- Outputs coefficients $b_1, ..., b_p$.
- *"Given input f, what is the right mixture?"*

Trunk net (τ_i):

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



Recall DeepONet is one kernel formulation

Our neural operator had the form

$$\mathcal{L}(f)(x) = \sigma\left(Wf(x) + b + \int_{y\in X} K_l(x,y)f(y)dy\right).$$

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:

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

FNO as a new kernel

DeepONet came from picking a *separable* kernel. FNO comes from a different choice.

Key step — choose a translation-invariant kernel:

$$\kappa(y, s) = \kappa(y - s)$$

Plug it in — the integral becomes a *convolution*:

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

By the **convolution theorem**, convolution in physical space becomes pointwise multiplication in Fourier space:

$$(\mathcal{K}_l f)(y) = \mathcal{F}^{-1}\Big(\underbrace{\mathcal{F}(\kappa)}_{=: R} \cdot \mathcal{F}(f) \Big)(y)$$

Rather than learn κ in physical space, just learn its Fourier coefficients R directly — one complex weight per kept mode. That's the **Fourier Neural Operator (FNO)**.

Aside: the convolution theorem

Convolution of two functions:

$$(g * f)(y) = \int_X g(y - s)\, f(s)\, ds\,.$$

Convolution theorem: the Fourier transform of a convolution is the pointwise product of the Fourier transforms:

$$\mathcal{F}(g * f)(\omega) = \mathcal{F}(g)(\omega) \cdot \mathcal{F}(f)(\omega)\,.$$

Equivalently, applying $\mathcal{F}^{-1}$ to both sides:

$$(g * f)(y) = \mathcal{F}^{-1}\Big(\mathcal{F}(g) \cdot \mathcal{F}(f)\Big)(y)\,.$$

A hard operation (integration against a kernel) becomes an easy one (multiplication) — if we're willing to pay the cost of two FFTs.

Let's revisit our favorite example: 1D transport PDE

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

The operator we want to learn is:

$$f \mapsto u(x,T)\,,$$

where T is some fixed value (say $T = 0.5$). We constructed a dataset already by numerically solving the PDE until $T = 0.5$ for a series of initial conditions $f_1, f_2, ..., f_{\text{num\_data}}$ with a discretization step size of $\delta x = 0.01$ and $\delta t = 0.0001$.

Our dataset for the 1D transport PDE looks like:

Thus, our dataset for our Neural Operator looks like:

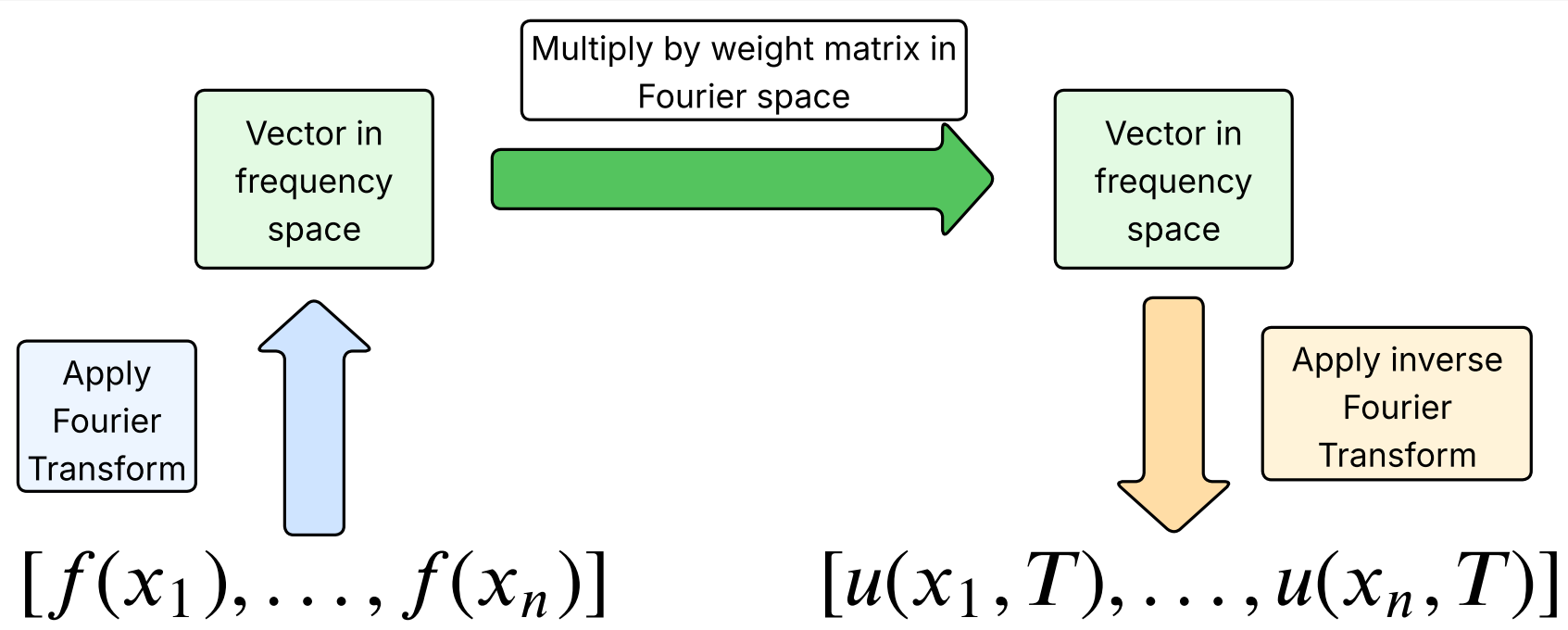
$$\begin{bmatrix} f_1(x_1) & \cdots & f_1(x_{101}) \\ \vdots & \ddots & \vdots \\ f_N(x_1) & \cdots & f_N(x_{101}) \end{bmatrix}, \quad \begin{bmatrix} u_1(x_1, T) & \cdots & u_1(x_{101}, T) \\ \vdots & \ddots & \vdots \\ u_N(x_1, T) & \cdots & u_N(x_{101}, T) \end{bmatrix}$$

- Left side is our "encoded" input.
- Right side is our supervised output.

Learn a neural operator solution that is **discretization invariant**. That is, we give it $[f(x_1), \ldots, f(x_{101})]$ and it gives us $[u(y_1, T), ..., u(y_n, T)]$ where $y_1, ..., y_n \in [0, 1]$.

Fourier Neural Operator

Key idea: Apply a Fourier transform to our input data, learn in Fourier space, and then our inverse Fourier transform



Why bother with the Fourier transform?

Recall the FNO key idea: *transform to Fourier space, learn there, transform back*.

Why is Fourier space a good place to learn?

- **Compression.** Smooth functions are well-approximated by just a few dominant frequencies (*modes*).
- **Convolutions become multiplications.** The kernel integral $\int K(x-y)f(y)\,dy$ becomes a pointwise product in Fourier space.
- **Resolution flexibility.** Once we have the frequencies, we can reconstruct f on *any* grid we want — this is what enables super-resolution.

The next few slides build up the machinery so we can cash in on these.

The continuous Fourier transform

Any (reasonable) function $f(x)$ can be written as a continuous superposition of complex exponentials.
Forward transform (signal $\to$ frequencies):

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x)\, e^{-i\omega x}\, dx\,.$$

Inverse transform (frequencies $\to$ signal):

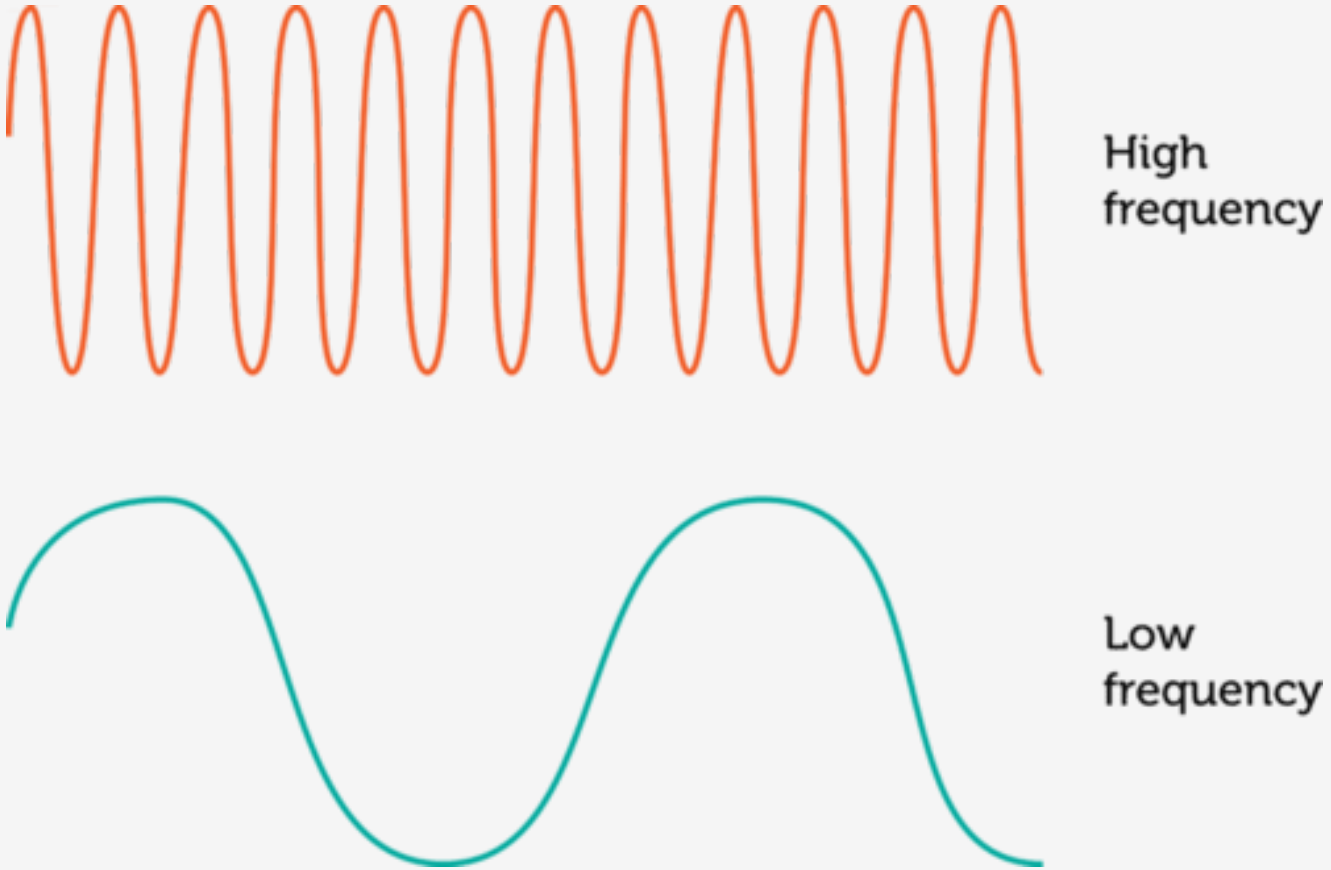
$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\omega)\, e^{i\omega x}\, d\omega\,.$$

By Euler's formula, $e^{i\omega x} = \cos(\omega x) + i\sin(\omega x)$.

Takeaway: f is a weighted sum of sin and cos; $\hat{f}(\omega)$ is the weight on frequency ω.

What does “frequency” mean?

The variable ω controls how fast the sin and cos waves oscillate.



- Small ω: slow oscillation — captures broad trends.
- Large ω: fast oscillation — captures sharp features.

From continuous to discrete: the DFT

In practice we don't have f as a continuous function — we have n samples:

$$[f(x_0), f(x_1), \ldots, f(x_{n-1})]\,.$$

Two changes turn the continuous transform into something we can compute:

- Replace the integral $\int dx$ with a sum $\sum_j$.
- Sample ω at n discrete values, indexed by an integer k (the *modes*).

The result is the **Discrete Fourier Transform (DFT)**:

$$\hat{f}(k) = \sum_{j=0}^{n-1} f(x_j)\, e^{-i\,2\pi \frac{k}{n} j}\,, \qquad k = 0, 1, \ldots, n-1\,.$$

n samples in, n complex coefficients out — no information lost.

Assumptions baked into the DFT

What does the DFT assume about f?

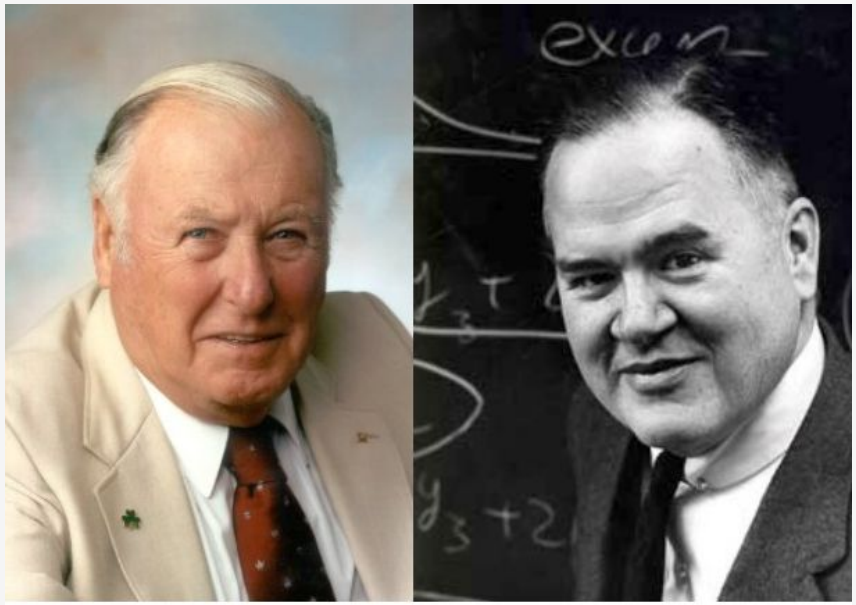
- **Periodicity:** the samples $f(x_0), \ldots, f(x_{n-1})$ are one full period of a function with period n. (Fix with padding of zeros.)
- **Uniform spacing:** $x_1 - x_0 = x_2 - x_1 = \cdots = x_{n-1} - x_{n-2}$.
- **Why the 2π in the exponent?**

Since sin and cos complete one revolution over 2π, the factor $2\pi/n$ makes mode $k = 1$ complete *exactly one* oscillation across the n samples, $k = 2$ complete two, and so on.

Fast Fourier Transform (FFT)

Computing the DFT directly is expensive — what's the cost?

Naive DFT: for each of n modes, sum over n samples $\Rightarrow$ $O(n^2)$.
FFT (Cooley–Tukey, 1965): recursively reuse partial sums
$\Rightarrow$ $O(n \log n)$.



James William Cooley (1926-)

John Wilder Tukey (1915-2000)

Why we care: the FT happens inside *every* Fourier layer of FNO.

Recap: computing the Fourier transform in FNO

Input: $[f(x_0), f(x_1), \ldots, f(x_{n-1})]$ — a function sampled on a uniform grid.
Step 1. Apply the FFT to obtain n Fourier coefficients (the modes):

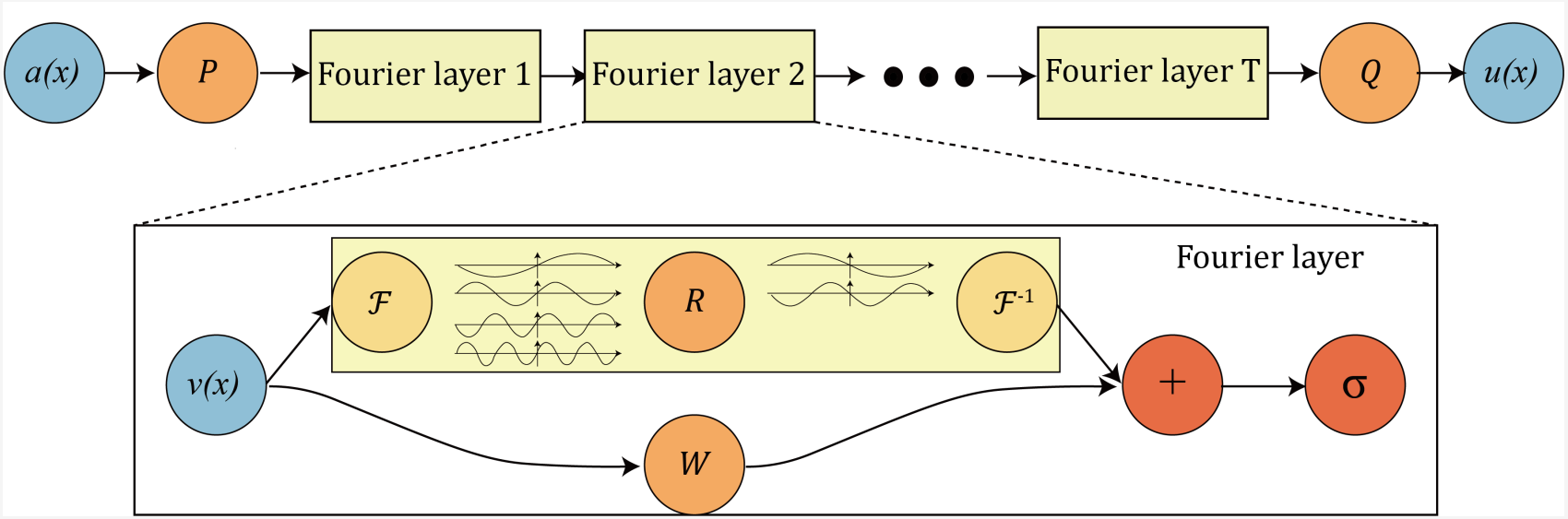
$$\hat{f}(k) = \sum_{j=0}^{n-1} f(x_j)\, e^{-i\, 2\pi \frac{k}{n} j}\,, \quad k = 0, \ldots, n-1\,.$$

Step 2. Multiply (learnably) in Fourier space — this is the Fourier layer (next slide).
Step 3. Apply the inverse FFT to return to physical space — *at any chosen resolution*, since the IFFT just evaluates the recovered basis at new points.

Is there a limit to k? Yes! The NyQuist frequency says the max information I can encode is $k = n/2$ where n is my grid resolution.

Full FNO Design

The architecture is given by



where each Fourier layer computes:

$$\text{Fourier\_Layer}(f(x)) = \sigma(Wf(x) + b + \mathcal{F}^{-1}(R\mathcal{F}(f(x)))),$$

where $\mathcal{F}, \mathcal{F}^{-1}$ are Fourier transforms and R is a weight matrix (learnable).

What does "lift to d channels" actually mean?

The lift, the Fourier layers, and the projection all act **pointwise across grid points**: at each x_i they apply the same operation with shared weights.
Lift: at each x_i, a shared MLP $\mathbb{R} \to \mathbb{R}^d$ maps the scalar $f(x_i)$ to a d-dim feature vector v_i.

$$\underbrace{[f(x_1), \ldots, f(x_n)]}_{\text{shape } (n,)} \longmapsto \underbrace{[v_1, \ldots, v_n]}_{\text{shape } (n,d)}$$

Project: mirror image — at each grid point a shared MLP $\mathbb{R}^d \to \mathbb{R}$ collapses v_i back to a scalar.
Implementation note: most FNO implementations also concatenate the coordinate x_i as an extra input channel to the lift. This lets the network depend on absolute position — necessary whenever the operator isn't translation invariant (e.g., non-periodic boundary conditions, spatially-varying coefficients).

Why can n be any resolution? Super-resolution

Train at $n_{\text{train}} = 101$, then run inference at $n_{\text{out}} = 401$ *with the same weights*:

1. **Lift:** call the trained MLP $\mathbb{R} \to \mathbb{R}^d$ at each of the 401 input points. Output shape $(401, d)$.
2. **Fourier layer:**
   - FFT of size 401 $\to$ spectrum of shape $(401, d)$.
   - Multiply the lowest $k = 16$ modes by trained R; zero out the rest.
   - IFFT of size 401 $\to$ back to physical space.
3. **Project:** call the trained MLP $\mathbb{R}^d \to \mathbb{R}$ at each of the 401 points. Output shape $(401,)$.

Why this works: the MLPs are pointwise (shared weights at every grid point), and R is indexed by mode (not by grid point). Nothing learnable has a dimension tied to n.

Example: 1D Linear Transport (our running example)

$$\begin{aligned} \partial_t u(x,t) - \partial_x u(x,t) &= 0\,, \\ u(x,0) &= u_0(x)\,, \qquad x \in (-2\pi,\, 2\pi)\,, \\ u_0(x) &= e^{-(x-k_1)^2/(4\nu)} - e^{-(x+k_2)^2/(4\nu)}\,. \end{aligned}$$

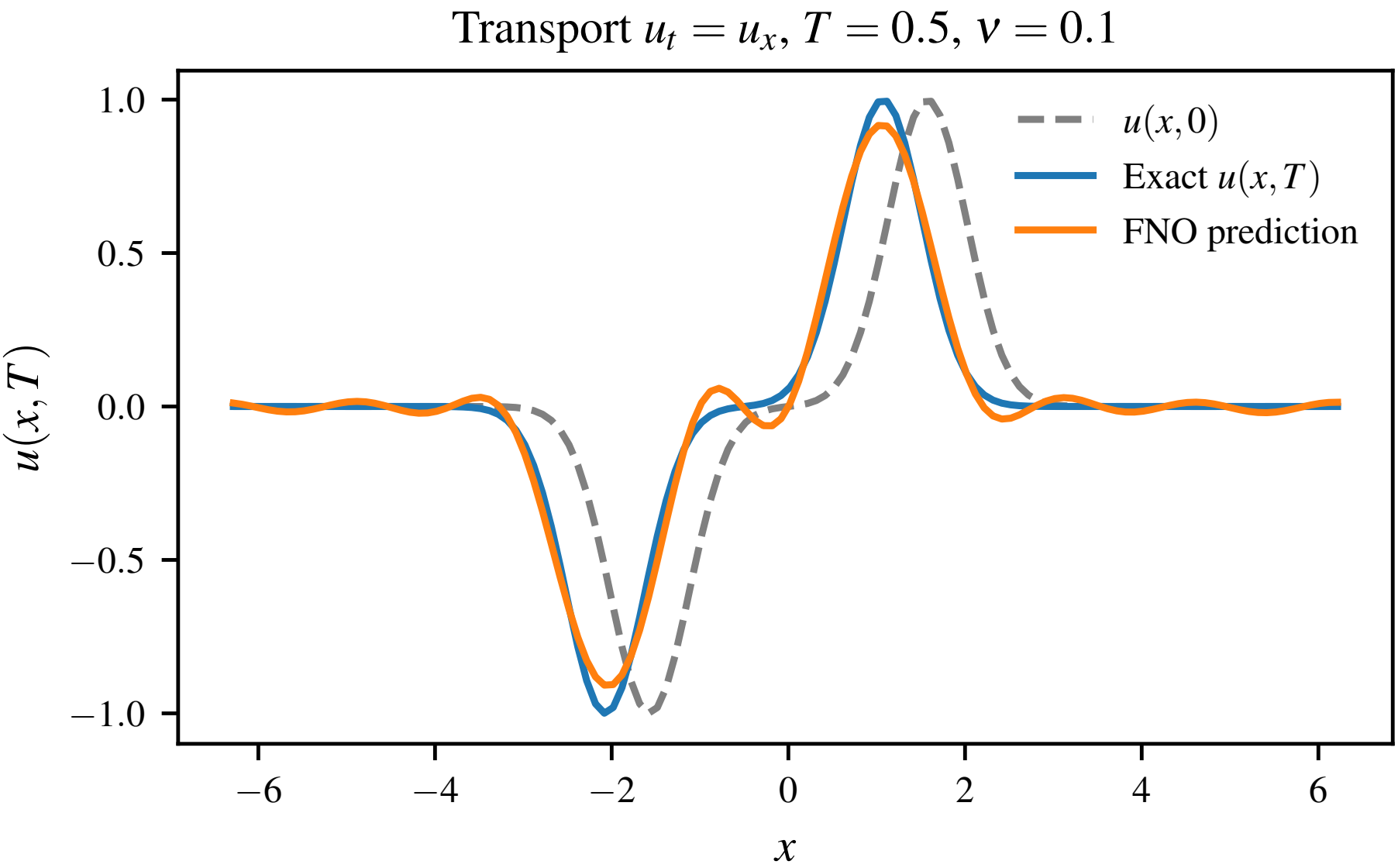
What each piece is:

- $u(x,t)$: the scalar state we want to learn. Exact solution is a pure left-shift, $u(x,t) = u_0(x+t)$, so the operator is linear.
- $u_0(x)$: a difference of two Gaussians, parameterized by centers $k_1, k_2 \in [0, \pi/2]$ and width $\nu \in [0.1, 1]$. Sampling these gives 1000 training instances.

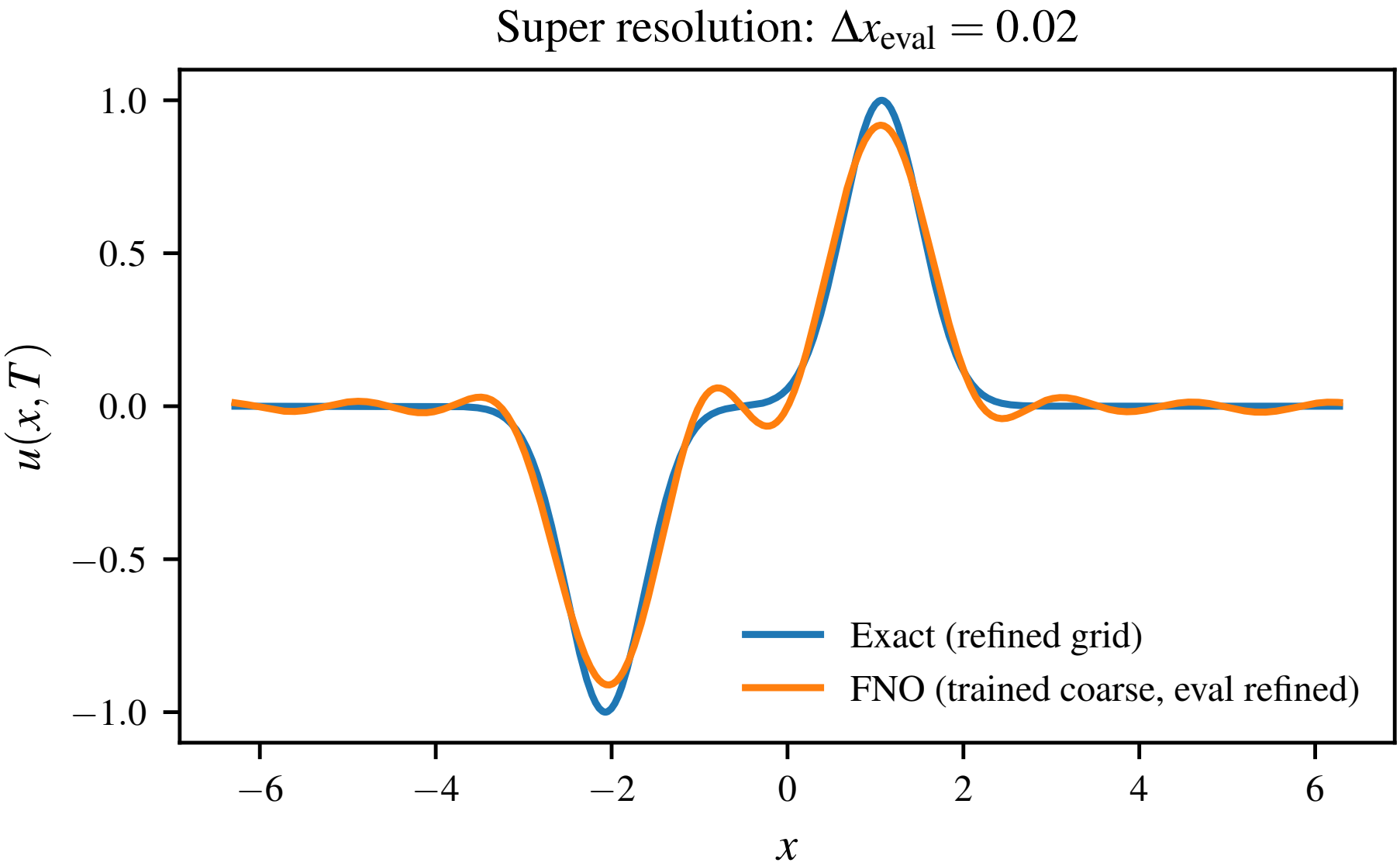
Operator to learn: given the initial condition, predict the state at a fixed final time $T(:= 0.5)$:

$$u_0 \quad \longmapsto \quad u\big|_{t=T}\,.$$

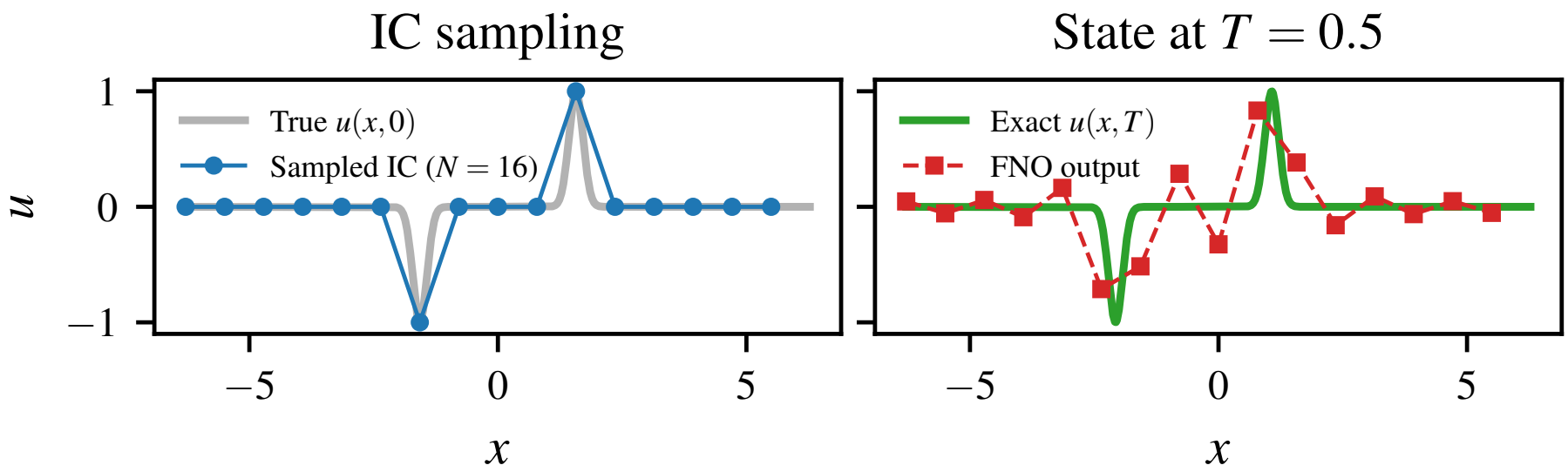
Example FNO result



Super resolution. Trained at $n = 126$, tested at $n = 629$ on $[-2\pi, 2\pi]$



Super-resolution fails when $n = 16$ as input



Example: 2D Navier–Stokes (the original FNO benchmark)

$$\begin{aligned} \partial_t u(x,t) + v(x,t)\cdot\nabla u(x,t) &= \nu\,\Delta u(x,t) + g(x)\,, \\ \nabla\cdot v(x,t) &= 0\,, \qquad u = \nabla\times v\,, \\ u(x,0) &= u_0(x)\,, \qquad x\in(0,1)^2\,. \end{aligned}$$

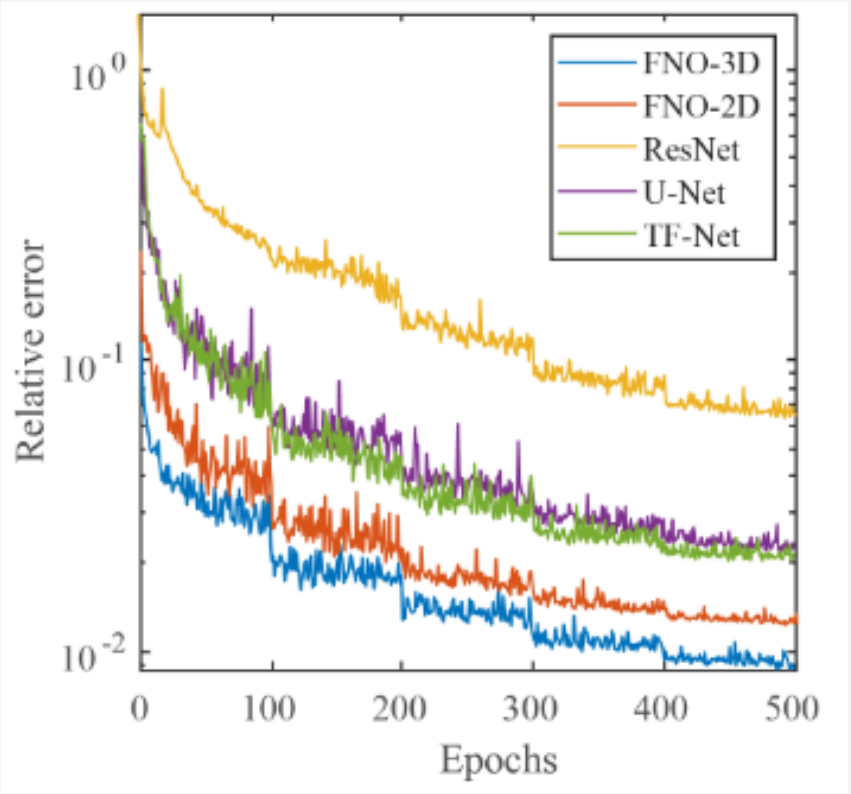
What each piece is:

- $u(x,t)$: the *vorticity* — the scalar state we want to learn.
- $v(x,t)$: the velocity field; determined by u via incompressibility ($u = \nabla\times v$, $\nabla\cdot v = 0$). Not an independent unknown.
- $\nu > 0$: kinematic viscosity. Small $\nu \Rightarrow$ sharp, turbulent features; large $\nu \Rightarrow$ smooth flow.
- $g(x)$: a fixed external forcing — a known function of space that drives the flow.

Operator to learn: given the first $T_0 (:= 10)$ seconds of vorticity, predict the rest of the trajectory up to T:

$$u\big|_{[0,T_0]} \quad\longmapsto\quad u\big|_{[T_0,T]}\,.$$

Comparison with other learning methods



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**	0.0834	**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

Disadvantages

What are the downsides of the FNO architecture?

- Requires uniform meshes to compute the Fourier transform.
- Truncation of the modes in frequency space can make it more challenging to capture sharp transitions in the model output (Model is biased to very smooth outputs, PDE solutions in many cases are not smooth. This is known as **spectral bias**.)
- FFT's can still be expensive, especially when the batch size is large.
- Comparison with DeepONet - cannot evaluate at any target point. (See this paper for more details on a "fair" comparison)
- How to handle boundary conditions in FNO especially when doing time-based super-resolution?
- Nyquist Limits

Application in climate modeling: FourCastNet(FNO)

- "FourCastNet predicts, with unparalleled accuracy at forecast lead times of up to one week, challenging variables such as surface winds and precipitation. No deep learning (DL) model thus far has attempted to forecast surface winds on global scales."

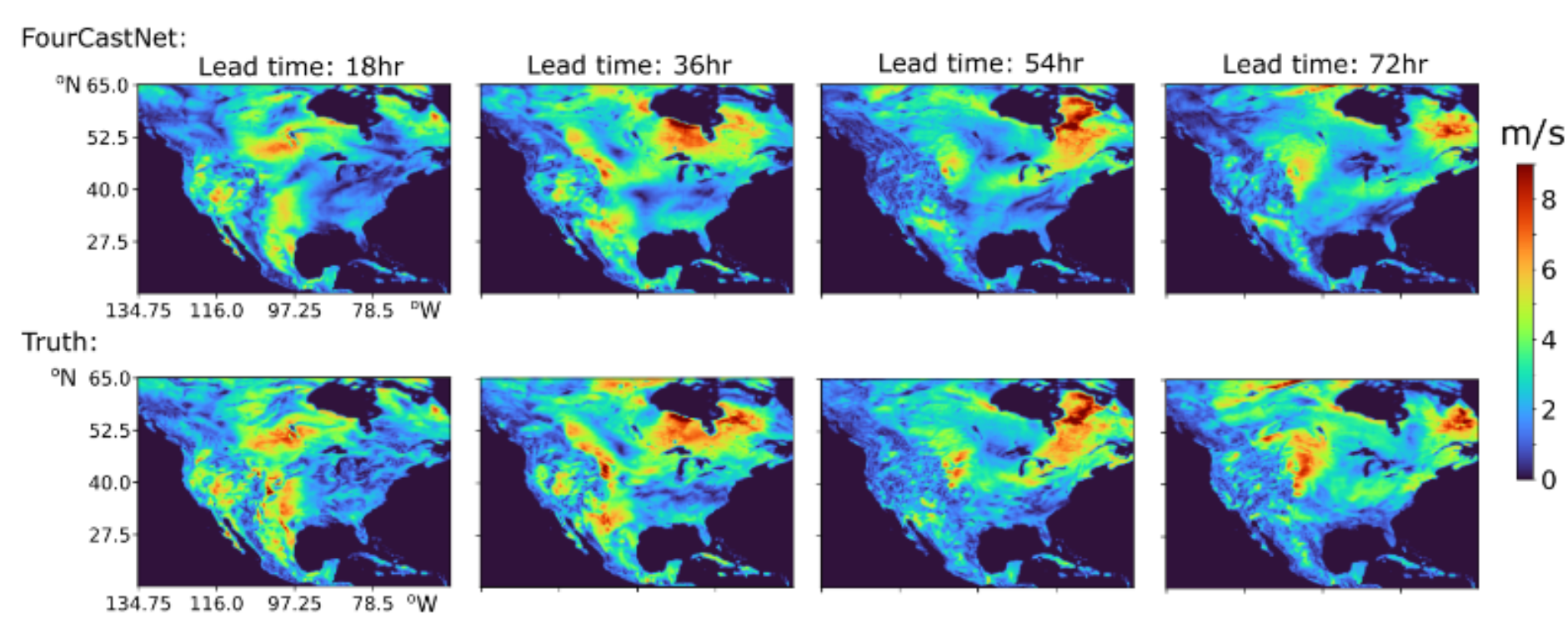


Figure: Overland wind speed (useful for turbulence)

Application in Earthquake Epicenter Detection (DeepONet)

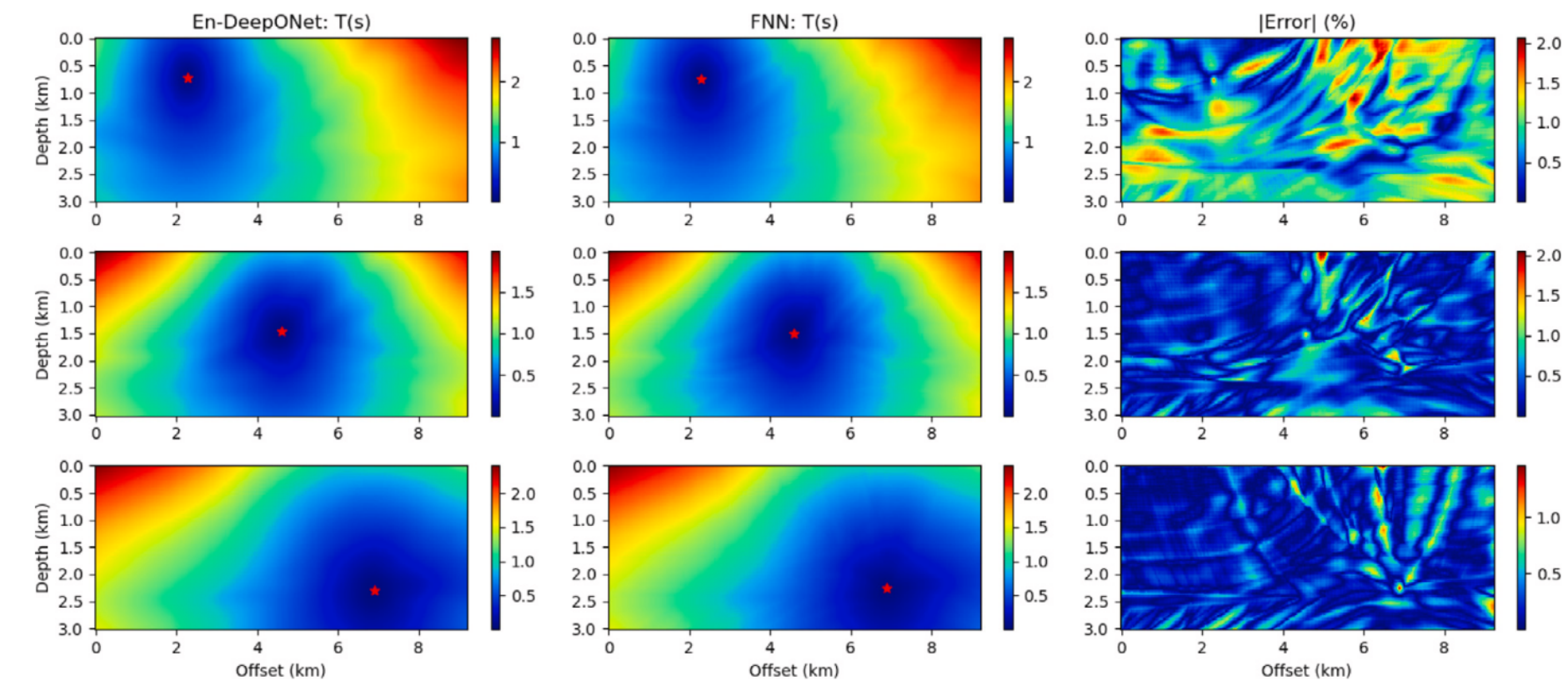


Figure: Estimation of earthquake travel times. Input is a sample of the earthquake magnitude at a few sensor locations. (Left is DeepONet and middle is classical PDE solver). See paper for more details.

Application in Motion planning in Robotics (FNO)

Given a binary occupancy grid and a goal location, compute the cost to get from any point to the goal location (See paper).

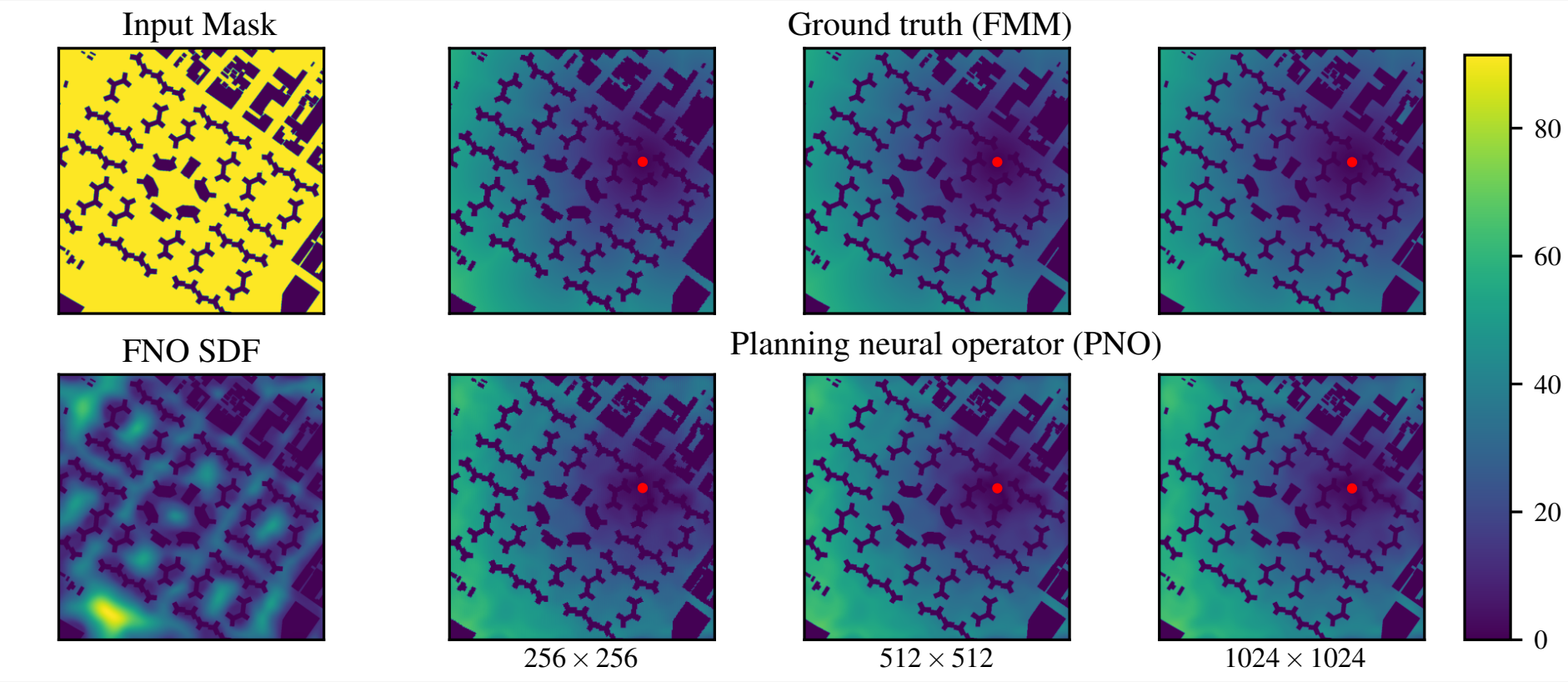
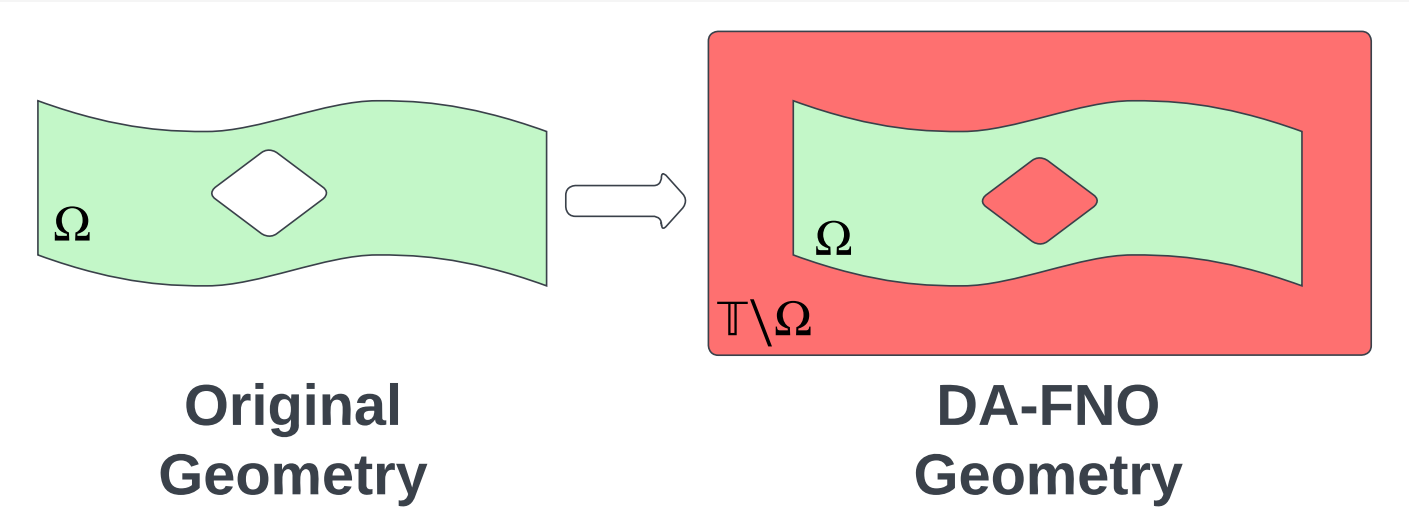


Figure: Dataset is trained using 64×64 resolution and here we see super-resolution on a map of navigating New York city.

What if my geometry is not a square (and so I can't apply FFT)?

Two approaches (there are many more):

1. Create an encoder-decoder structure that maps your geometry to a square (See paper). Downside is how to preserve the super-resolution property across the encoder-decoder?
2. Domain agnostic Fourier Neural Operators. Circumscribe geometry in a square:



Then, in the kernel function modify $K_l(x, y)$ to be zero if x or y is in the square, but not in the original geometry.

What other types of kernels can I do

- Instead of the Fourier transform, we can use other types of transforms (wavelet transform or Laplace transform)
- Separable or (Sec 4.2 of this paper, low rank) neural operator. Treat the kernel as the product of two functions

$$K_l(x, y) = \phi(x)\psi(y)$$

where $\phi(\cdot)$ and $\psi(\cdot)$ are classic MLPs.

- Graph Neural Operator: It can be very expensive to compute $K_l(x, y)$ at all points x, y. So, approximate it:

$$\int_{y \in X} K_l(x, y) f(y) dy \approx \frac{1}{S} \sum_{j=1}^{S} k(x, y_s) f(y_s)$$

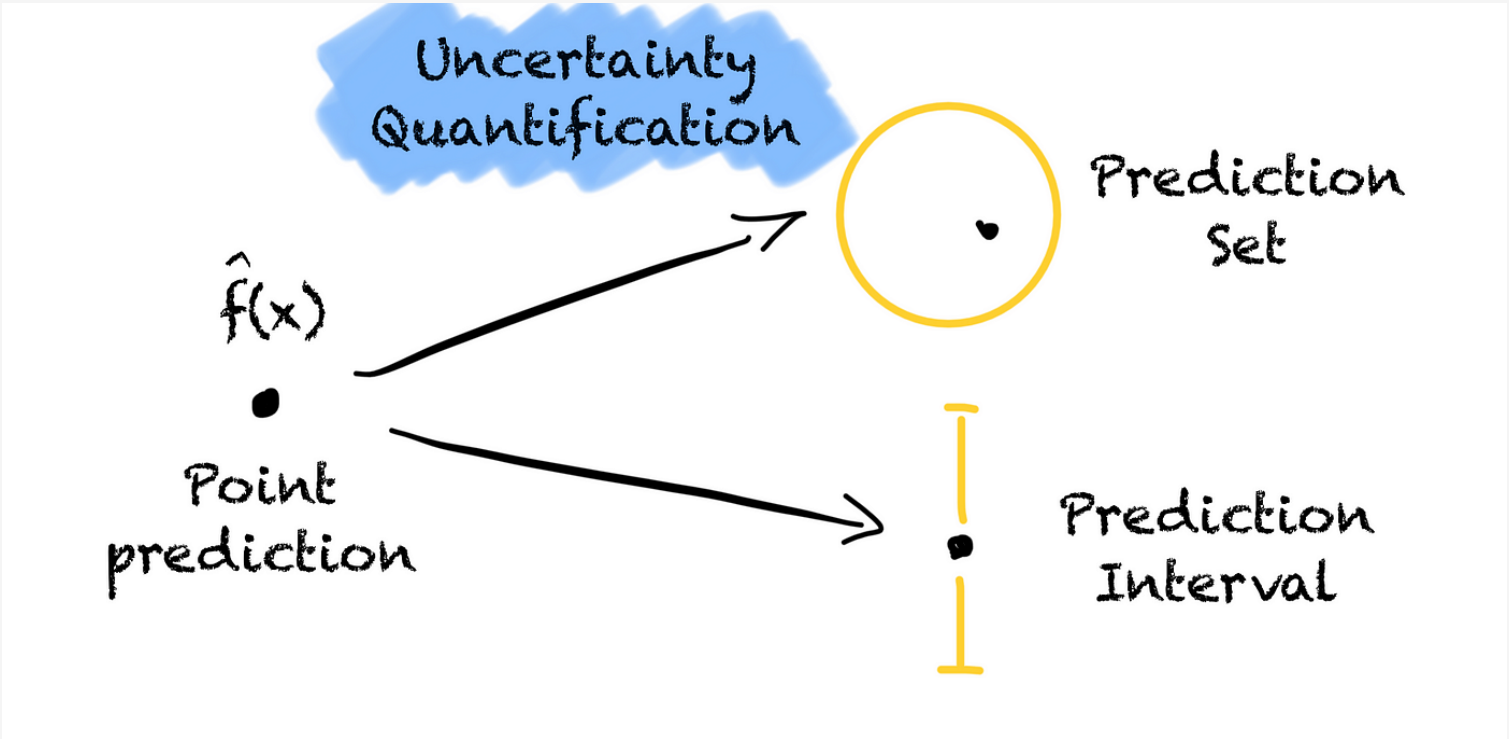
where we chose S to be some subset of all our points in X. This is called a Nystrom approximation. This is called graph neural operator as $K_l(x, y_s)$ is akin to an adjacency matrix.

- Transformers are *one instantiation* of neural operators (See Sec 5.2 of this work).

Uncertainty quantification

What is uncertainty quantification?

Answer: Uncertainty quantification tells us how confident a model is in its prediction.



Types of UQ

What are some types of UQ that you already know for ML?

- **Softmax weighting gives some sense of UQ**: If a model gives 0.9 for a class versus 0.55, in both cases we choose that class but one is "more confident"
- **Ensemble methods:** Train 5 NNs and compare how much they vary on any one test case (tight spread=more confident ideally)
- **Gaussian Processes (GP):** Essentially they are a tool that return a mean and standard deviation of a estimated prediction (That is constrained to follow a multivariate Gaussian) instead of just the prediction itself.
- **Conformal prediction:** Gives prediction intervals such that the true value is within the interval with probability $1 - \delta$.
- **Bayesian Neural Networks:** Weights of NN aren't fixed, but are sampled from a probability distribution at each forward inference call.

UQ for neural operators

We talked about how its important to do uncertainty quantification for neural operators. Here's whats been done:

- Conformal prediction for neural operators. Idea: Train a FNO. Then, train a second FNO to give you an uncertainty ball around that point.
- Ensembled neural operators. Train a bunch of neural operators and take an "average" of the outputs in a smart way.
- Bayesian/GP Neural Operators. Instead of learning a static output, they augment the NO architecture such that it outputs a mean and standard deviation that follow's a Gaussian prior at each point.

There are many more ideas here to be explored and Yuexin will explore UQ in more detail in the future weeks!

Theory on neural operators

- Universal approximation theorem of neural operators (See paper)
- Discretization error estimates. That is if I have more $x_1, ..., x_n$, how does my error theoretical change?
- Sample complexity estimates If I have more f_i, u_i pairs, how does my error change?
- Parameter complexity estimates If I want to achieve an accuracy of ϵ, I need $C \exp(c\epsilon^{-\gamma})$ (C, c, γ problem dependent) parameters for my NN.

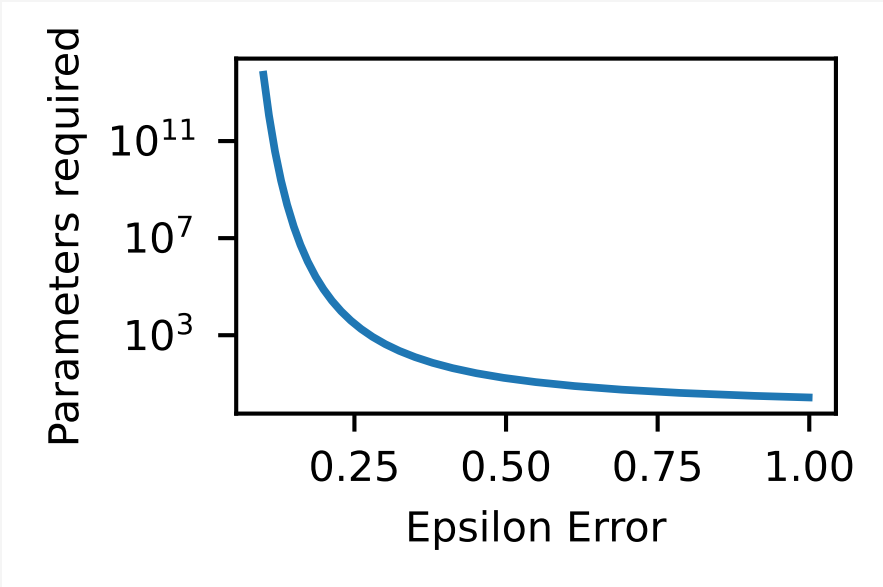


Figure: Scaling of parameters with respect to ϵ, $(C, c = 1, \gamma = 1.5)$

Open problems left in the field

- **Applications**. Many applications left for YOU to apply these.
- Complicated geometries and boundary conditions.
- Learn PDEs with discontinuities. (Very hard learning problem for FNO which is biased towards smooth functions)
- Encoding of the function f can be bad. Symbolic-based encoding (See paper)
- Integrate physics into our design (See you next lecture on PINNs)