Lecture 3a: Neural Operator methods for Differential Equations

Chris Budd and Aengus Roberts[1]

[1]University of Bath

CWI, October 2025

Some papers/books to look at

- Courant and Hilbert *Methods of mathematical physics Volumes 1,2*
- Stuart et. al. *Fourier Neural Operator for PDEs*
- Kovachi et. al. *Operator learning: algorithms and analysis*
- Boullé and Townsend *Learning elliptic PDEs*
- Halko et. al. *Finding structure with randomness*

Motivation: Solution Operators

Have studied using PINNS and DRMs to solve PDE problems of the form

$$u_t = F(\mathbf{x}, u, \nabla u, \nabla^2 u) \quad \text{with BC,}$$

$$u(0, x) = u_0(x)$$

At time T we have the solution $u_T(x) \equiv u(x, T)$.

Solution $u(x, t)$ for all x and t is obtained by minimising a function directly associated with the PDE eg. residual.

Neural Operator methods take a different approach

- Consider u_T as a function F of u_0. $u_T = F(u_0)$
- F is an operator mapping one infinite dimensional function space to another $F : A \to B$. eg. $A, B = H^1(\Omega)$
- Train a Neural Operator NN to approximate this operator note infinite dimensions
- Train it by generating a (large) set of solution pairs (u_0^i, u_T^i)

Can generate solution pairs using a (conventional) numerical method eg. Finite Element, Pseudo-Spectral, Symplectic.

eg. ERA5 data for 24 hour weather forecasts.

Example 1: A finite dimensional problem [Halko et. al.]

- Have an $n \times n$ matrix B
- Have a random set of N vectors $\mathbf{x}_i \in R^n$
- Compute the N (noisy) matrix vector products

$$B\,\mathbf{x}_i = \mathbf{y}_i \in R^n$$

Challenge: Construct the matrix B from the set of N solution pairs $(\mathbf{x}_i, \mathbf{y}_i)$

Methodology:

- Let $X = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N]$, $Y = [\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N]$
- Find $B = Y\,X^+$ using the SVD/Moore-Penrose pseudo-inverse.

Example 2: A linear ODE system

Consider the linear ODE

$$\frac{d\mathbf{u}}{dt} = A\,\mathbf{u}, \quad \mathbf{u}(0) = \mathbf{u}_0, \quad \mathbf{u} \in R^n.$$

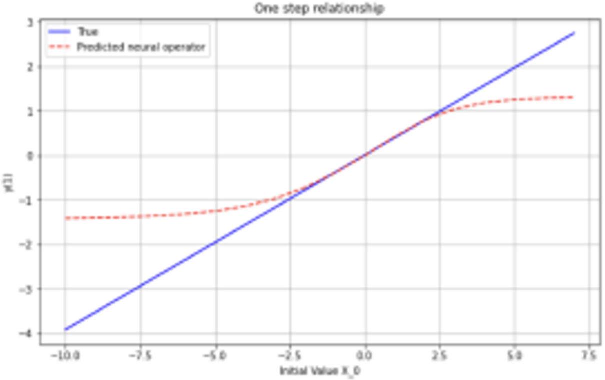
Solution

$$\mathbf{u}(T) = e^{A\;T}\mathbf{u}_0 \equiv B\;\mathbf{u}_0$$

$$B \equiv e^{AT} = I + AT + \frac{A^2T^2}{2!} + \frac{A^3T^3}{3!} + \ldots$$

Properties of the solution operator

- Operator B is linear
- Operator is continuous over any subset of R^n
- Can easily learn the matrix B from data pairs if we assume that the operator is linear in advance!
- If we learn B from a subset of the data pairs then we can extrapolate this to ALL data pairs
- This is NOT true if don't make the linearity assumption. Many NN methods will locally approximate the operator to be linear, but will not give this as a global approximation.


One step relationship
True
Predicted neural operator
y(1)
Initial Value X_0
3
2
1
0
−1
−2
−3
−4
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5

Latent space description of the operator

Let A have eigenvectors ϕ_i so that

$$A\,\phi_i = \lambda_i \phi_i$$

Set $\mathbf{u} = \sum a_i(t)\,\phi_i$ then

$$\frac{d\mathbf{u}}{dt} = \sum \frac{da_i}{dt}\phi_i = A\mathbf{u} = \sum \lambda_i a_i \phi_i$$

so that

$$\frac{da_i}{dt} = \lambda_i a_i \implies a_i = a_i(0)e^{\lambda_i t}$$

Assume A is symmetric. Then can set

$$\phi_i^T \phi_j = \delta_{ij}$$

Hence

$$\mathbf{u}(T) = \sum \phi_i^T \mathbf{u}_0 e^{\lambda_i T} \phi_i.$$

Takes the form of

- **Encoder**: $\phi_i^T \mathbf{u}_0$.
- **Latent space evolution**: $e^{\lambda_i T}$
- **Decoder** Multiply by ϕ_i

This structure is used in the design of the **Deep-O-Net** Neural Operator

Example 3: Parabolic PDEs

Consider the parabolic PDE [picture]

$$u_t = u_{xx} + f(x), \quad x \in [0, 2\pi], \quad u(0, x) = u_0(x), \quad periodicBC$$

We can express $u(x, t)$ in terms of convolutional integral operators:

$$u(x, t) = G * u_0 + H * f \equiv \int_0^{2\pi} G(x-y, t) u_0(y) \, dy + \int_0^{2\pi} H(x-y, t) f(y) \, dy$$

These operators act on the infinite dimensional space $L^2[0, 2\pi]$.

Can find $G(z, t)$ and $H(z, t)$ **explicitly** using a Fourier series.

Fourier Series

As u and f are 2π periodic we can set:

$$u(x,t) = \sum_j c_j(t) e^{ijx}, \quad f(x) = \sum_j f_j e^{ijx},$$

Substituting into the PDE we have

$$\frac{du_j}{dt} = -j^2 u_j(0) + f_j,$$

with
Hence

$$c_j(T) = e^{-j^2 T}\left(c_j(0) - \frac{f_j}{j^2}\right) + \frac{f_j}{j^2}, \quad c_0(T) = c_0(0) + f_0 T$$

with

$$c_j(0) = \frac{1}{2\pi}\int_0^{2\pi} e^{-ijy} u_0(y)\, dy, \quad f_j = \frac{1}{2\pi}\int_0^{2\pi} e^{-ijy} f(y)\, dy.$$

Hence

$$u(x, T) = \int_0^{2\pi} \frac{1}{2\pi} \sum_j e^{ij(x-y)} e^{-j^2 T} u_0(y)\, dy$$

$$+ \int_0^{2\pi} \frac{1}{2\pi} \sum_j e^{ij(x-y)} j^{-2} f(y)\, dy + \dots$$

So we can see that this has the correct integral form with

$$G(z, T) = \sum_j \frac{1}{2\pi} e^{-j^2 T} e^{ijz}, \quad H(z, t) = \sum_j \frac{1}{2\pi} j^{-2} e^{ijz} + \dots$$

Trivially $G(z, T)$ has Fourier Coefficents

$$G_j = \frac{1}{2\pi} e^{-j^2 T}.$$

Learning G and H

- Suppose for a fixed $f(x)$ we have lots of solution pairs $(u_0^k(x), u_T^k(x))$ ($k = 1..N$ random set)
- Use FFT to find the Fourier coefficients $u_0^k \to u_0^{k,j}, u_T^k \to u_T^{k,j}, f \to f_j$
- For each j find the FCs of G and H by solving the minimisation problem

$$(G_j, H_j) = \mathrm{argmin}_k \| G_j u_0^{k,j} + H_j f_j - u_T^{k,j} \|$$

This methodology motivates the construction of the Fourier Neural Operator (FNO)

The FNO: In general

The FNO architecture is based on the process of solving the linear heat equation, but also works for **nonlinear problems.** The FNO constructs a 'Neural Map' Ψ parametrised by θ as follows:

$$\Psi(a,\theta)_{FNO} \equiv Q \circ \mathcal{L}_L \circ \ldots \circ \mathcal{L}_2 \circ \mathcal{L}_1 \circ P(a).$$

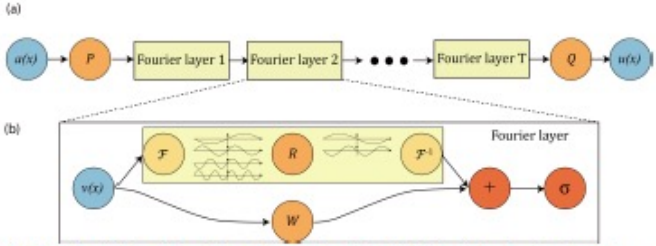
$$\mathcal{L}_n(v)(x,\theta) = \sigma(W_n v(x) + b_n + K(v))$$

Here W is a pointwise linear local map. $K(v)$ is a global convolutional integral operator, kernal $G_n(\theta)$. Evaluate Kv using an FFT via

$$FFT(Kv) = FFT(G_n)\; FFT(v).$$

FFT restricted to M modes. Nonlinearity and higher order modes introduced via the activation function σ.

Figure from FNO paper



(a) The full architecture of neural operator: start from input a. 1. Lift to a higher dimension channel space by a neural network P. 2. Apply four layers of integral operators and activation functions. 3. Project back to the target dimension by a neural network Q. Output u. **(b) Fourier layers:** Start from input v. On top: apply the Fourier transform $\mathcal{F}$; a linear transform R on the lower Fourier modes and filters out the higher modes; then apply the inverse Fourier transform $\mathcal{F}^{-1}$. On the bottom: apply a local linear transform W.

Figure 2: **top:** The architecture of the neural operators; **bottom:** Fourier layer.

FNO in detail

- Input $a_j(x) \in \mathcal{A}$ output $u_j(x) = N(a_j) \in \mathcal{U}$ are functions on $x \in D \subset R^d$
- Assume have access to pointwise observations of a only at points in $x_i \in D_j \subset D$. Output u does not depend on D_j: super-resolution
- Lift a to a higher dimensional representation $v_0(x) = P(a(x))$ by a shallow NN.
- Calculate a series of updates $v_n \to v_{n+1}$ via the local W_n and global (integral) K_n operators:

$$v_{n+1}(x) = \sigma\left(W_n v_n(x) + (K_n(\theta)v_n)(x)\right).$$

- For example $\sigma = ReLU$: This introduces nonlinearity into the map in a slightly uncontrolled way
- Project $v_L \to u(x) = Q(v_L)$
- Learn P, Q, W_n, K_n from the data pairs

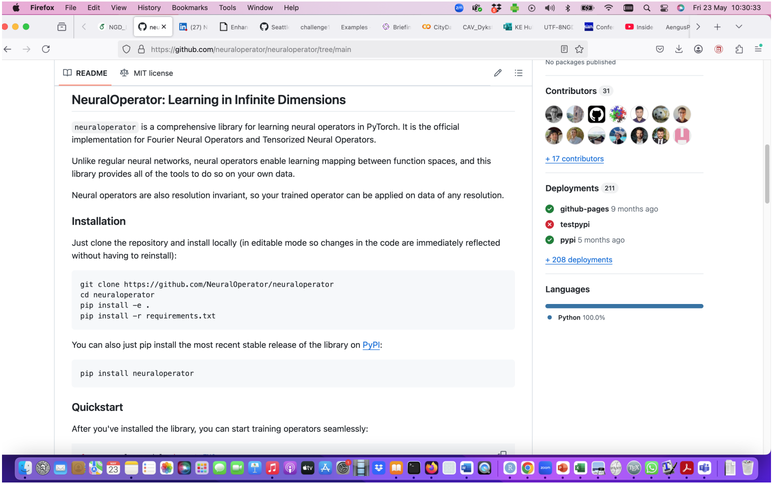
Training

- Assume input $a \in \mathcal{A}$
- In the original FNO paper take a_j as an i.i.d sequence from $\mathcal{A}$.
- Construct pairs $(a_j, N(a_j))$ using an accurate solver eg. pseudo-spectral method
- In FNO paper take $N = 1000$ training and 200 training instances. Adam optimiser to find parameters θ via:

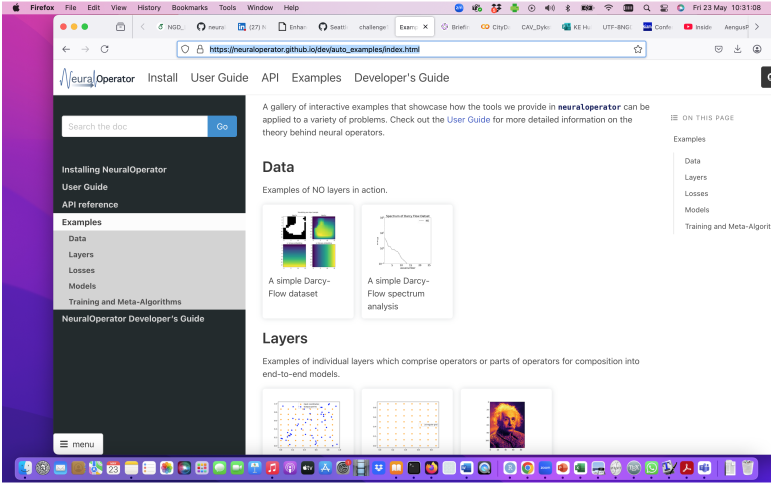
$$min_\theta E_{a \sim \mu}[\|\psi(a, \theta) - N(a)\|]$$

- Can significantly improve training by a more careful selection of input and output pairs [Liu, B, et. al.]

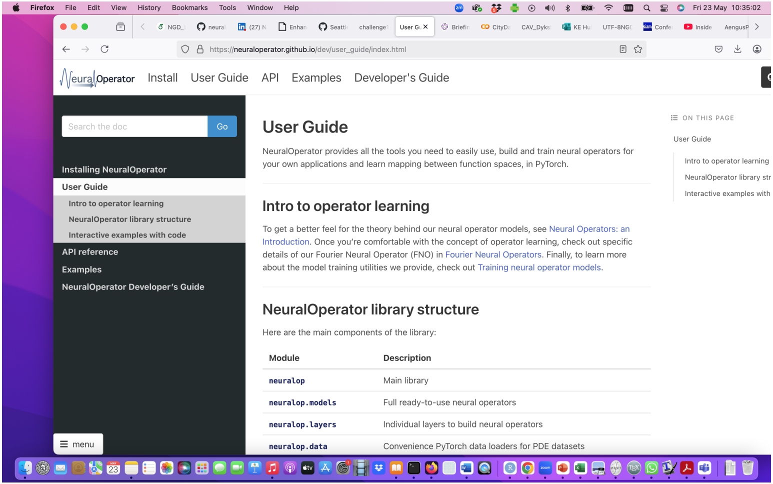
FNO online 1


Firefox File Edit View History Bookmarks Tools Window Help
Fri 23 May 10:30:33
https://github.com/neuraloperator/neuraloperator/tree/main
README
MIT license
NeuralOperator: Learning in Infinite Dimensions
neuraloperator is a comprehensive library for learning neural operators in PyTorch. It is the official implementation for Fourier Neural Operators and Tensorized Neural Operators.
Unlike regular neural networks, neural operators enable learning mapping between function spaces, and this library provides all of the tools to do so on your own data.
Neural operators are also resolution invariant, so your trained operator can be applied on data of any resolution.
Installation
Just clone the repository and install locally (in editable mode so changes in the code are immediately reflected without having to reinstall):
git clone https://github.com/NeuralOperator/neuraloperator
cd neuraloperator
pip install -e .
pip install -r requirements.txt
You can also just pip install the most recent stable release of the library on PyPI:
pip install neuraloperator
Quickstart
After you've installed the library, you can start training operators seamlessly:
No packages published
Contributors 31
+ 17 contributors
Deployments 211
github-pages 9 months ago
testpypi
pypi 5 months ago
+ 208 deployments
Languages
Python 100.0%

FNO online 2


Firefox File Edit View History Bookmarks Tools Window Help
Fri 23 May 10:31:08
https://neuraloperator.github.io/dev/auto_examples/index.html
NeuralOperator
Install
User Guide
API
Examples
Developer's Guide
Search the doc
Go
Installing NeuralOperator
User Guide
API reference
Examples
Data
Layers
Losses
Models
Training and Meta-Algorithms
NeuralOperator Developer's Guide
menu
A gallery of interactive examples that showcase how the tools we provide in neuraloperator can be applied to a variety of problems. Check out the User Guide for more detailed information on the theory behind neural operators.
Data
Examples of NO layers in action.
A simple Darcy-Flow dataset
A simple Darcy-Flow spectrum analysis
Layers
Examples of individual layers which comprise operators or parts of operators for composition into end-to-end models.
ON THIS PAGE
Examples
Data
Layers
Losses
Models
Training and Meta-Algorit

FNO online 3

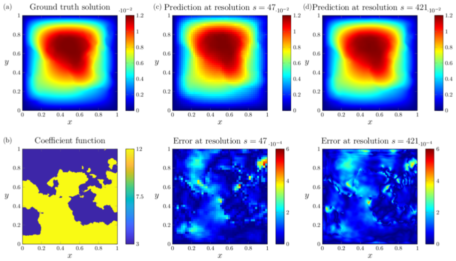

https://neuraloperator.github.io/dev/user_guide/index.html
NeuralOperator
Install
User Guide
API
Examples
Developer's Guide
Search the doc
Go
Installing NeuralOperator
User Guide
Intro to operator learning
NeuralOperator library structure
Interactive examples with code
API reference
Examples
NeuralOperator Developer's Guide
menu
User Guide
NeuralOperator provides all the tools you need to easily use, build and train neural operators for your own applications and learn mapping between function spaces, in PyTorch.
Intro to operator learning
To get a better feel for the theory behind our neural operator models, see Neural Operators: an Introduction. Once you're comfortable with the concept of operator learning, check out specific details of our Fourier Neural Operator (FNO) in Fourier Neural Operators. Finally, to learn more about the model training utilities we provide, check out Training neural operator models.
NeuralOperator library structure
Here are the main components of the library:
Module
Description
neuralop
Main library
neuralop.models
Full ready-to-use neural operators
neuralop.layers
Individual layers to build neural operators
neuralop.data
Convenience PyTorch data loaders for PDE datasets
ON THIS PAGE
User Guide
Intro to operator learning
NeuralOperator library str
Interactive examples with
Fri 23 May 10:35:02

Example 4: The Darcy Problem

The Darcy problem relates a permeability $a(x)$ to a velocity field $u(x)$

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

This induces a (nonlinear) map $N : a \to u, \quad N : L^2(\Omega) \to H^1_0(\Omega)$



We can approximate this map using the Finite Element Method

$$u(x) \approx U(x) = \sum U_i \, \phi_i(x).$$

$$-\nabla \cdot (a(x)\nabla u) = f \implies \int a(x)\nabla u(x) \cdot \nabla \phi_i(x) \, dx = \int f(x)\phi_i(x) \, dx \equiv f_i$$

Giving the linear system

$$A\mathbf{U} = \mathbf{f}, \quad \mathbf{U}_i = U_i, \quad \mathbf{f}_i = f_i, \quad A_{ij} = \int a(x) \, \nabla \phi_i \cdot \nabla \phi_j \, dx.$$

Hence we can approximate the nonlinear map via:

$$\mathbf{U} = A^{-1} \, \mathbf{f}$$

And ...

We can LEARN this map by

- Doing lots of finite element calculations to find solution pairs $(a(x), u(x))$
- Learn the operator between these pairs using an FNO

Methods such as FNO work as much as possible in the infinite-dimensional function space.

They Construct and train Neural Operators which are approximations to the true operator (or its inverse) which are independent of the resolution of the underlying function/image

Convergence [Kovachi et. al.]

- FNO and DeepONet can approximate a wide variety of operators
- Assume that input space $\mathcal{U}$ is a separable Banach space and the map N is compact
- Prove convergence on any finite dimensional set using the universal approximation theorem
- Take an appropriate limit (approximation theory of Banach spaces which applies to the sets over which PDEs are typically formulated)

Nonlinear problems and a warning

Consider now the nonlinear parabolic PDE

$$u_t = u_{xx} + f(x, u), \quad u(0) = u(1) \quad u(0, x) = u_0(x)$$

This does not always induce a continuous map from $u(0, x) \to u(1, x)$.

- If $f(x, u)$ is Globally Lipshitz in x and u then all is OK
- If not then we may have problems
- See Case Study!

Example

Let

$$f(x, u) = u^2, \quad u_0(x) = \gamma > 0$$

Then

$$u(1, x) = \frac{\gamma}{1-\gamma}.$$

Map is only continuous on the interval $\gamma \in [0, 1)$

If we train only on data with $\gamma < 1$ we will get a false result if we try to extend to $\gamma > 1$.

Areas for improvement and research on FNO

- Observe poor conservation laws at the moment
- Generating a good training set is crucial and can be slow. How to make it good and fast?
- FNO struggles away from the training set. This is OK for MCMC emulators for UQ.
- BUT Need to broaden its scope and extend the theorems on its convergence
- NONE of this theory applies, for example, to the nonlinear heat equation

$$u_t = u_{xx} + u^2.$$