NVIDIA DEEP LEARNING INSTITUTE | BROWN

Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

Lecture 9: Neural Operators

Instructors: George Em Karniadakis, Khemraj Shukla, Lu Lu
Teaching Assistants: Vivek Oommen and Aniruddha Bora

CRUNCH TIME
CRUNCH Group
Brown University

NVIDIA DLI TEACHING KITS
SCIENCE AND ENGINEERING

Course Roadmap

Module-1 (Basics)

- Lecture 1: Introduction
- Lecture 2: A primer on Python, NumPy, SciPy and *jupyter* notebooks
- Lecture 3: Deep Learning Networks
- Lecture 4: A primer on TensorFlow and PyTorch
- Lecture 5: Training and Optimization
- Lecture 6: Neural Network Architectures

Module-2 (PDEs and Operators)

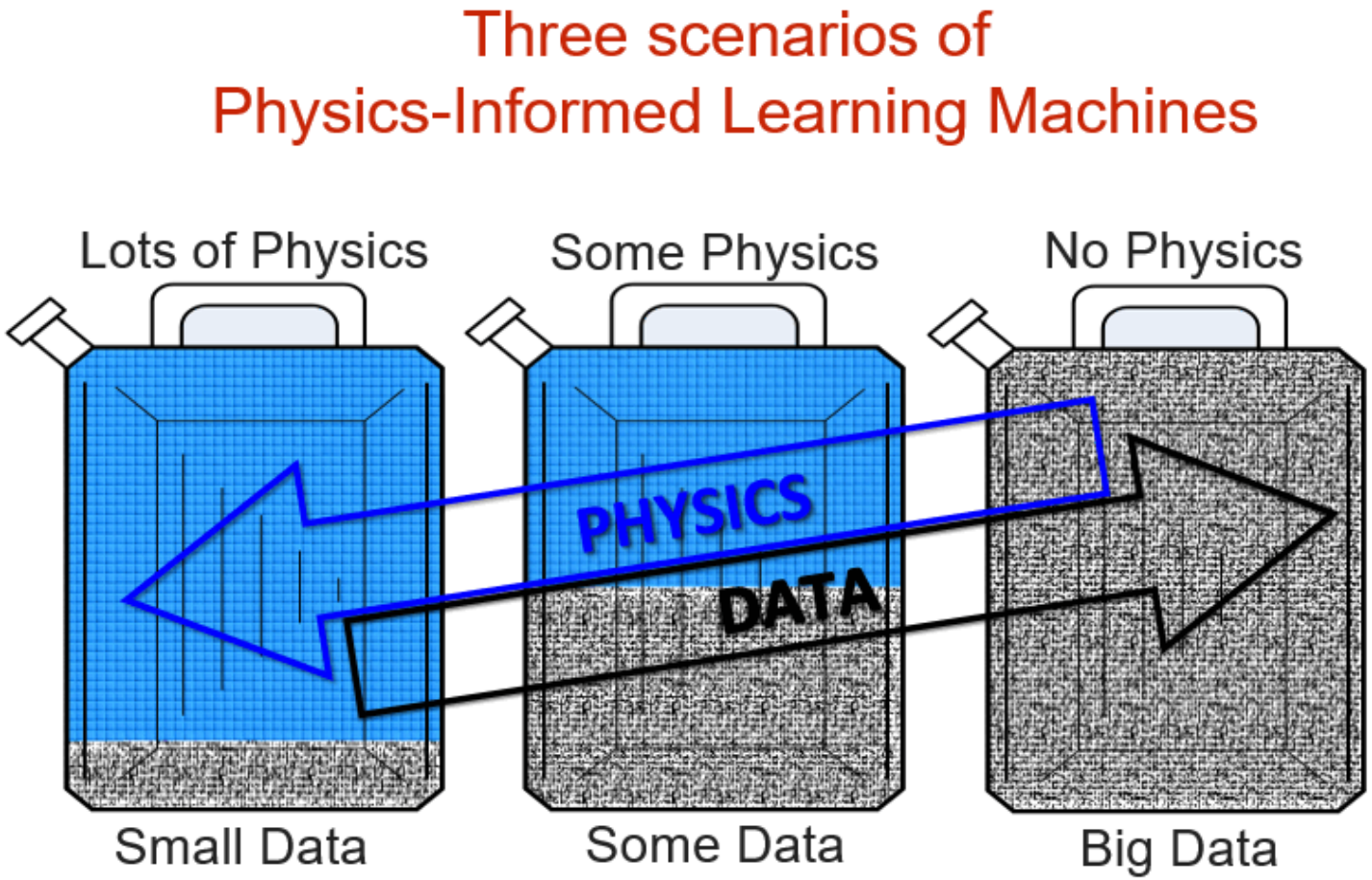
- Lecture 7: Physics-Informed Neural Networks (PINNs)
- Lecture 8: PINN Extensions
- Lecture 9: Neural Operators

Module-3 (Codes & Scalability)

- Lecture 10: Multi-GPU Scientific Machine Learning

Contents

Data + Physical Laws



Universal Approximation Theorem for Functionals: 1-dim case

Theorem

Suppose that U is a compact set in $C[a,b]$, f is a continuous functional defined on U, and $\sigma(x)$ is a bounded generalized sigmoidal function, then for any $\epsilon > 0$, there exist $m+1$ points $a = x_0 < \cdots < x_m = b$, a positive integer N and constants $W_i^1, \theta_i, W_{i,j}^2, i = 1, \cdots, N, j = 0, 1, \cdots, m$, such that

$$\left| f(u) - \sum_{i=1}^{N} W_i^1 \sigma \left(\sum_{j=0}^{m} W_{i,j}^2 u(x_j) + \theta_i \right) \right| < \epsilon$$

- T.P. Chen and H. Chen, Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Transactions on Neural Networks, 910-918, 4(6), 1993.

Universal Approximation Theorem for Functionals: n-dim case

Theorem

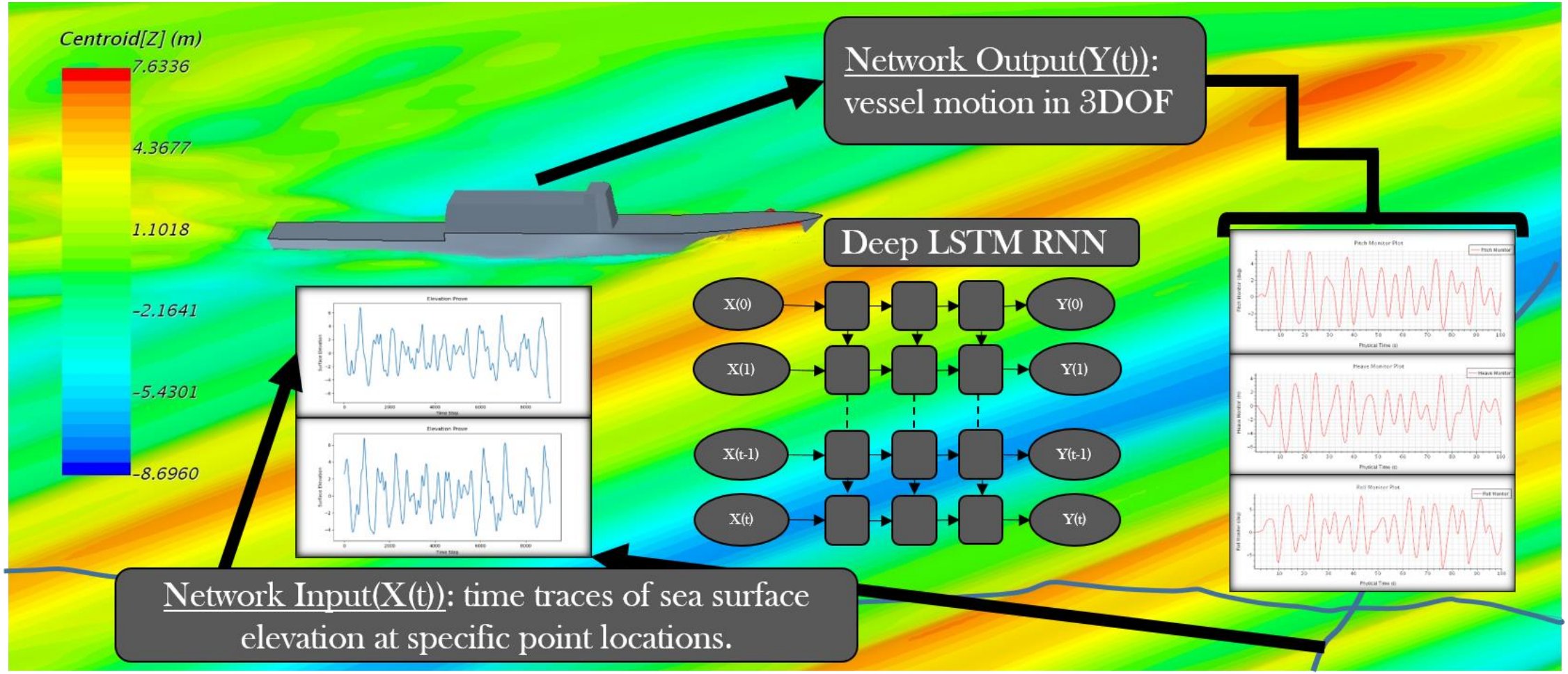
Suppose that U is a compact set in $C_V\left(\Pi_{k=1}^n [a_k, b_k]\right)$, f is a continuous functional defined on U, and $\sigma(x)$ is a bounded generalized sigmoidal function, then for any $\epsilon > 0$, there exist $(m+1)^n$ points $\left(x_1^{j_1}, \cdots, x_n^{j_n}\right), x_k^{j_k} = a_k + j_k(b-a)/m, j_k = 0, 1, \cdots, m, k = 1, \cdots, n, a$ positive integer N, and constants W_i^1, θ_i, and $q \times (m+1)^n$-vectors $\bar{W}_i^2$, such that

$$\left| f(u) - \sum_{i=1}^{N} W_i^1 \sigma\left(\bar{W}_i^2 \cdot \bar{u}_{q,n,m} + \theta_i\right) \right| < \epsilon, \quad \forall u \in U$$

where $\bar{u}_{q,n,m} = \left(u_l\left(x_1^{j_1}, \cdots, x_n^{j_n}\right)\right), l = 1, \cdots, q, j_k = 1), 1, \cdots, m, k = 1, \cdots, n$ are $q \times (m+1)^n$-vectors.

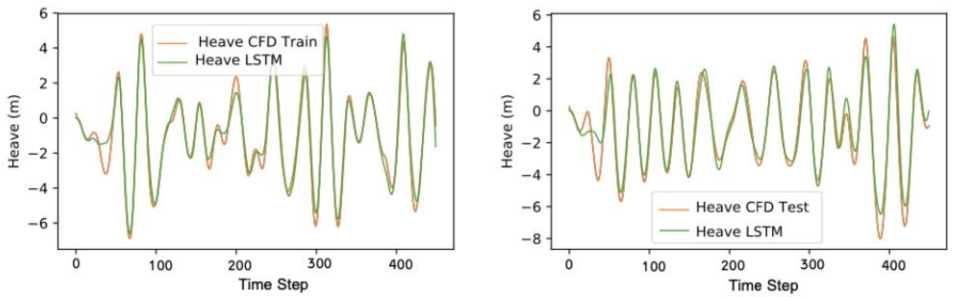
- T.P. Chen and H. Chen, Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Transactions on Neural Networks, 910-918, 4(6), 1993.

Autonomy: Predicting the motion of battleships in extreme sea states



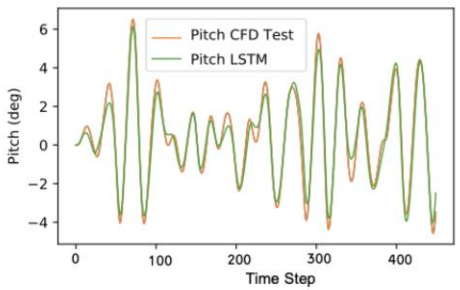
Jose del Aguila Ferrandis, Chryssostomos Chryssostomidi, Michael Triantafyllou, Karniadakis, G.E. Learning functionals via LSTM neural networks for predicting vessel dynamics in extreme sea states, Proc. R. Soc. A.47720190897, 2021.

Predicting the motion of battleships in extreme sea states

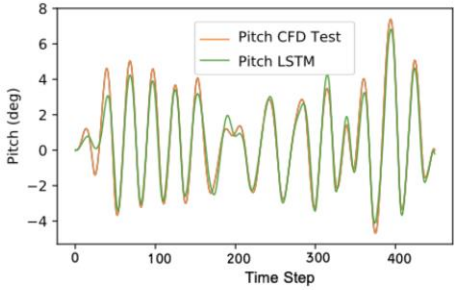


(a) RSE = 0.0380

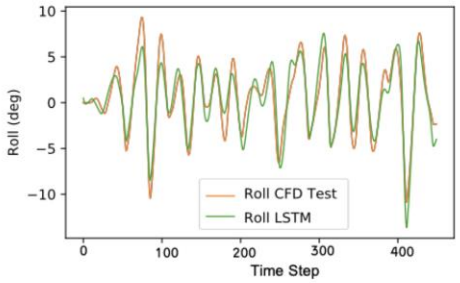
(b) RSE = 0.0773



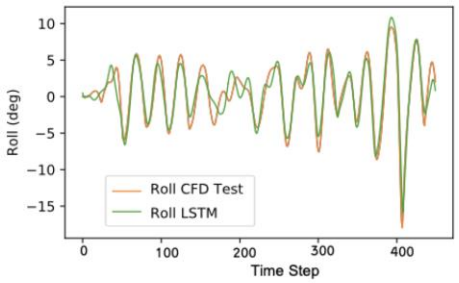
(c) RSE = 0.0444



(d) RSE = 0.0463



(e) RSE = 0.1490

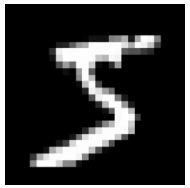


(f) RSE = 0.1215

- Heave, Pitch and Roll motions for 3-DOF motion approximation of a notional DTBM battleship sailing in WMO sea state at high Froude Number 0.4.
- The motion approximations provided in the unseen realizations of the sea state obtain a good performance both in amplitude and frequency approximation of the vessel response.
- Roll spectrum is very different to that of heave and pitch motions.
- The performance of the network approximating roll motions is somewhat worse compared to heave and pitch motions.

Functions map data, Operators map functions

- Function: $\mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$

  e.g., image classification:  $\mapsto 5$

- Operator: function (∞-dim) $\mapsto$ function (∞-dim)
  e.g., derivative (local): $x(t) \mapsto x'(t)$
  e.g., integral (global): $x(t) \mapsto \int K(s,t)x(s)ds$

  e.g., dynamic system: 

  e.g., biological system
  e.g., social system

$\Rightarrow$ Can we learn operators via neural networks?
$\Rightarrow$ How?

Universal Approximation Theorem for Operators

$$G : u \mapsto G(u), G(u) : y \in \mathbb{R}^d \to \mathbb{R}$$

Theorem (Chen & Chen, IEEE Trans. Neural Netw., 1995)

Suppose that σ is a continuous non-polynomial function, X is a Banach Space, $K_1 \subset X, K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}, w_k \in \mathbb{R}^d, x_j \in K_1, i = 1, \ldots, n, k = 1, \ldots, p, j = 1, \ldots, m$, such that

$$|G(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{\text{branch}} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{\text{trunk}}| < \epsilon$$

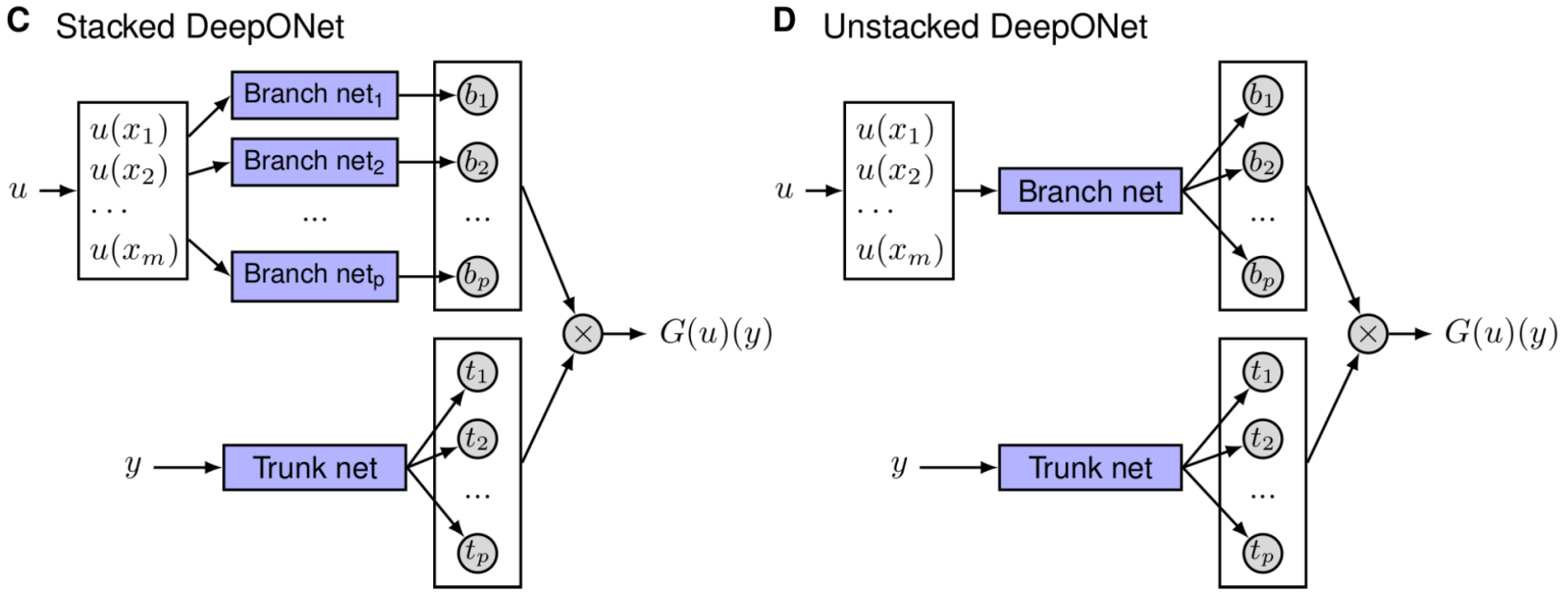
holds for all $u \in V$ and $y \in K_2$.

- T. Chen and H. Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. IEEE Transactions on Neural Networks, 6(4):911–917, 1995.

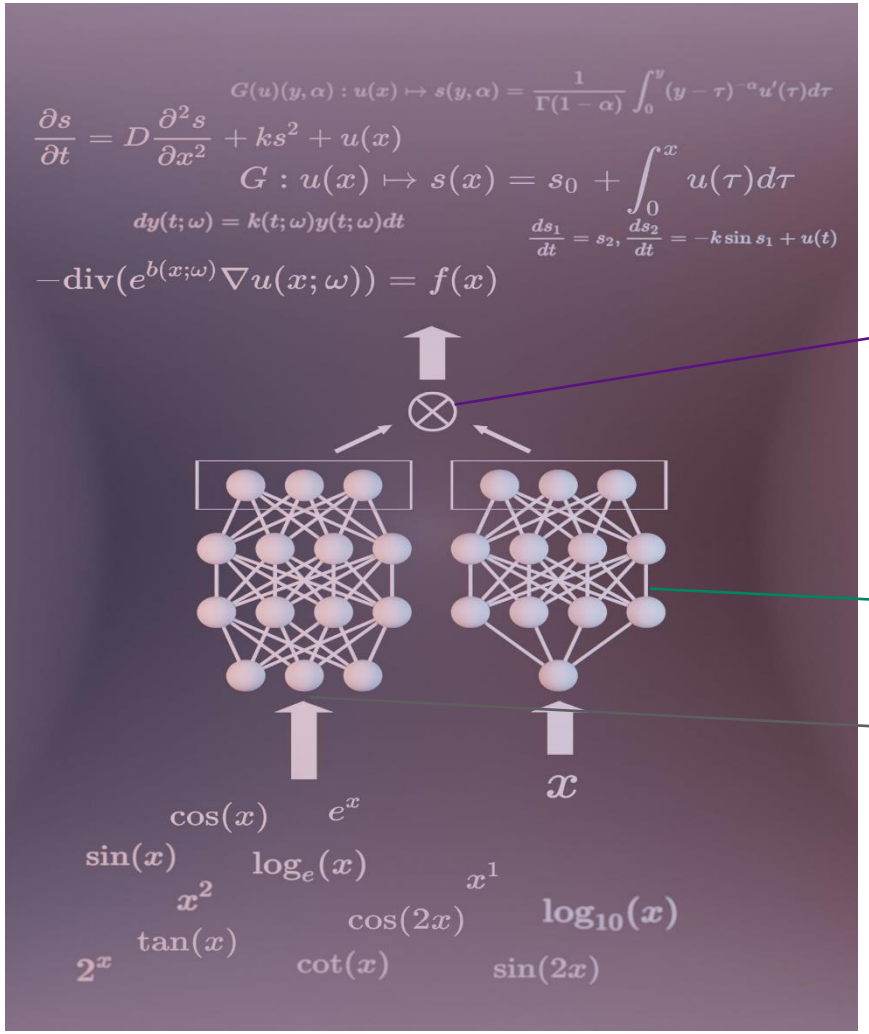
Deep Operator Network (DeepONet)

Recall the Theorem:

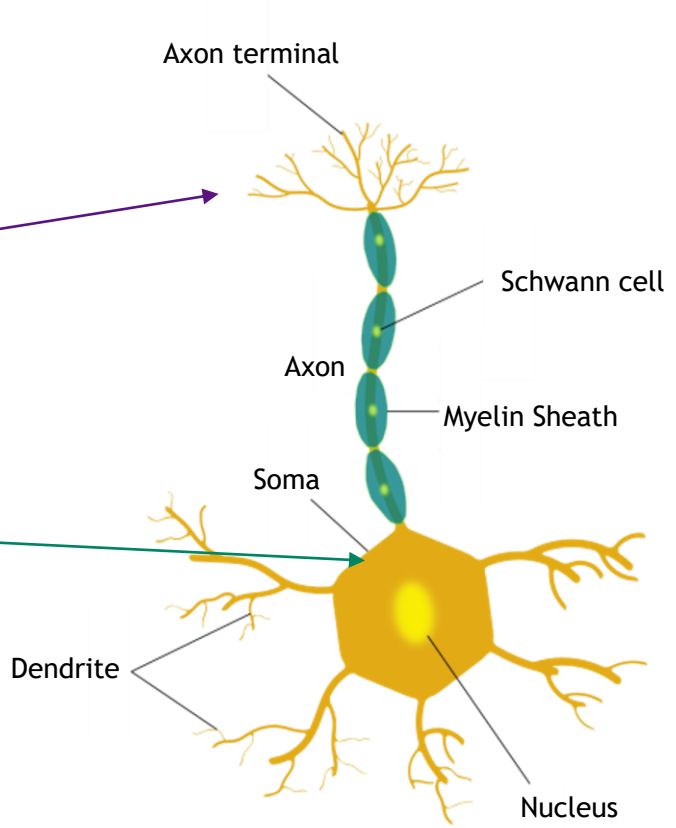
$$G(u)(y) \approx \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{branch} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{trunk}$$



DeepOnet (first published in the arXiv Oct 2019)



- Some resemblance to a human neuron

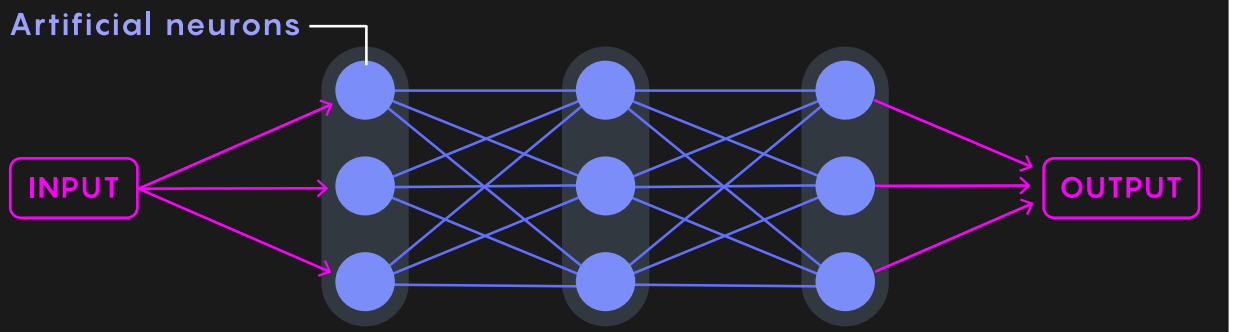


“…DeepONet is *blazingly fast* next to numerical solvers…”

Quantamagazine

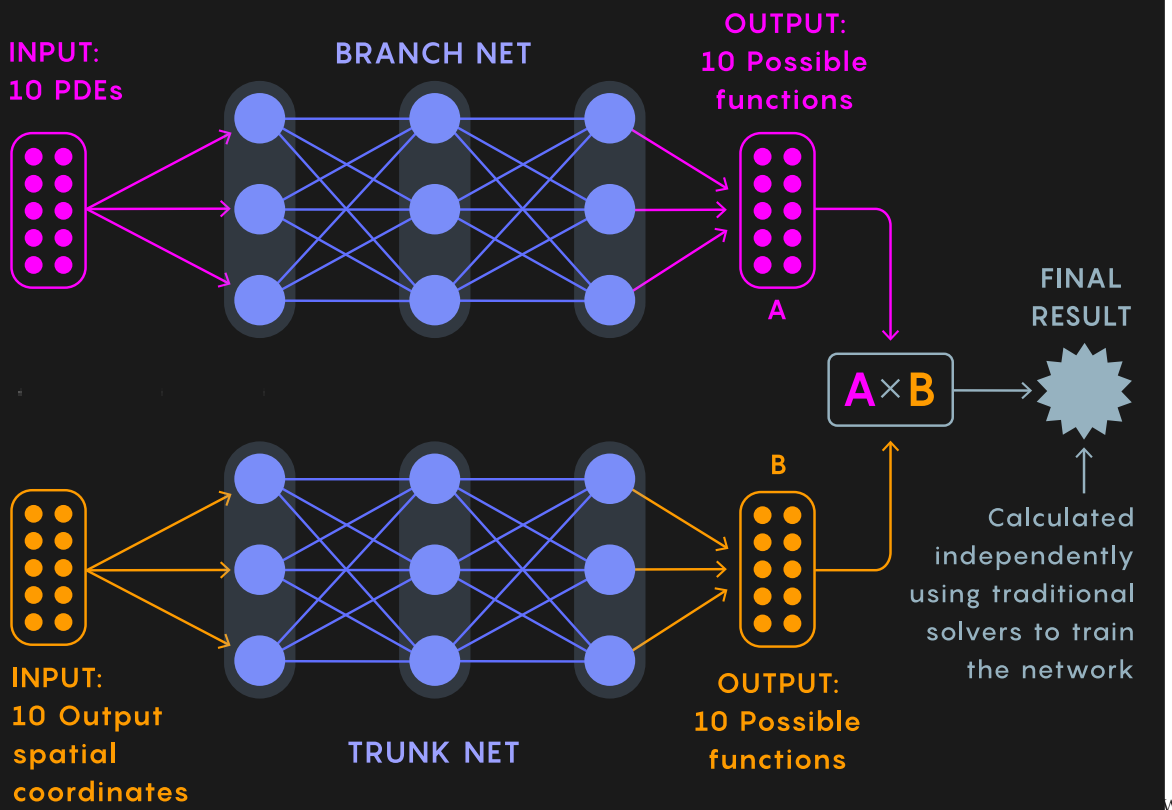
Deep Neural Network

An artificial neuron multiplies each input it receives by a weight, and adds them to get a weighted sum. The sum is passed through a nonlinear function that transforms it into an output. A neural network, which is composed of layers of artificial neurons, can be trained with known inputs and outputs by adjusting the weights of the neurons until the network produces the expected output for a given input.



DeepONet

The DeepONet architecture is split into two parallel networks: a **branch** that approximates functions related to input, and a **trunk** that does the same for the output. By combining the outputs from both branches, it can learn to approximate an operator that effectively solves partial differential equations (PDEs) much faster than traditional solvers.



Universal Approximation of Operators via Deep Neural Networks

Theorem (Lu et al, Nature Machine Intelligence 2021)

Let X be a Banach Space and $K_1 \subset X$ be compact. Let $V \subset C(K_1)$ be compact and $K_2 \subset \mathbb{R}^d$ be compact. Assume that $\mathcal{G} : V \to C(K_2)$ is a nonlinear continuous operator. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants $c_i^k \in \mathbb{R}$, neural networks $f\mathcal{N}\left(\cdot; \theta^{(k)}\right)$ and $g^{\mathcal{N}}\left(\cdot; \Theta^{(k,i)}\right), x_j \in K_1, i = 1, \ldots, n, k = 1, \ldots, p, j = 1, \ldots, m$, such that

$$|\mathcal{G}(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k g^{\mathcal{N}}\left(u_m; \Theta^{(k,i)}\right)}_{\text{branches}} \underbrace{f^{\mathcal{N}}\left(y; \theta^{(k)}\right)}_{\text{trunk}}| < \epsilon$$

holds for all $u \in V$ and $y \in K_2 \subseteq \mathbb{R}^d$, where

- $\mathrm{u}_m = [u(x_1), u(x_2), \cdots, u(x_m)]^\top \in \mathbb{R}^m$,
- and the neural networks $f^{\mathcal{N}}\left(y; \theta^{(k)}\right)$ and $g^{\mathcal{N}}\left(u_m; \Theta^{(k,i)}\right)$ satisfy the universal approximation theorem of continuous functions on compact sets.

- Lu L, Jin P, Pang G, Zhang Z, Karniadakis GE. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. Nature Machine Intelligence. 2021 Mar;3(3):218-29.

DeepONet – Problem Setup

Input function $\rightarrow v$ defined on the domain $D \subset \mathbb{R}^d$, e.g., $f(x,t)$ or $u_0(x)$,

$$v : D \ni x \mapsto v(x) \in \mathbb{R},$$

output function $\rightarrow u$ defined on the domain $D' \subset \mathbb{R}^{d'}$

$$u : D' \ni \xi \mapsto u(\xi) \in \mathbb{R}.$$

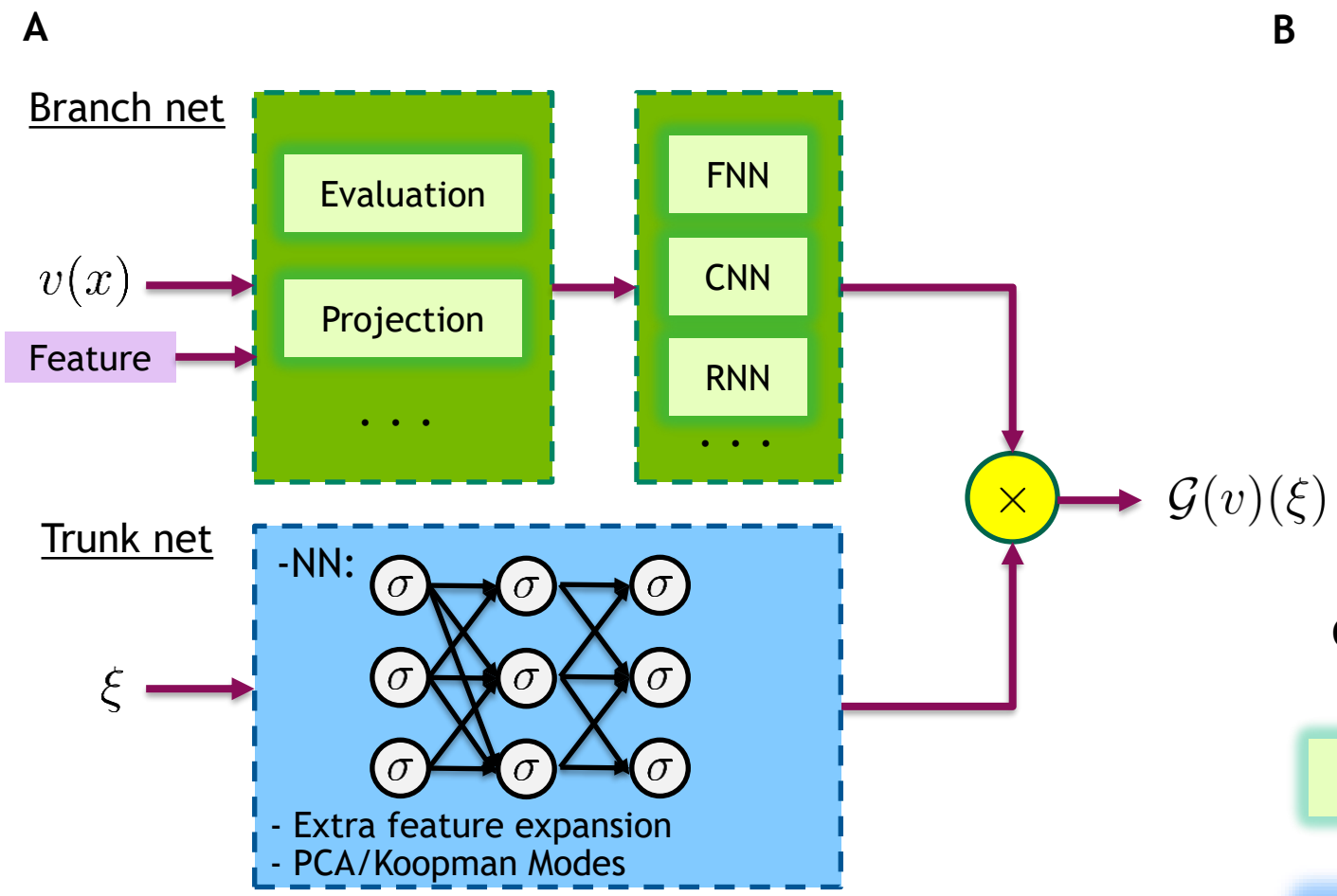
$\mathcal{V} \rightarrow$ space for v and $\mathcal{U} \rightarrow$ space for u.

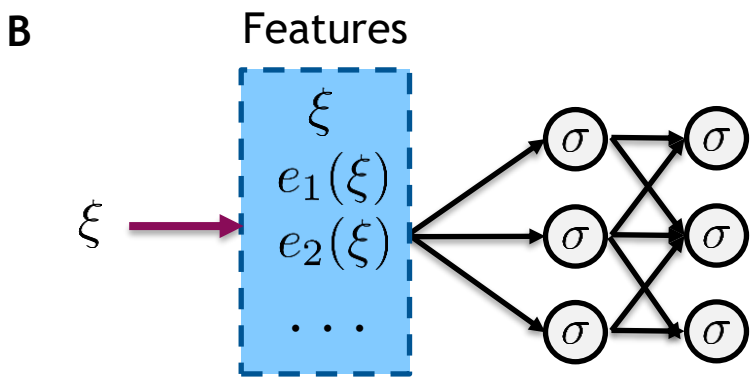
The mapping from the input function v to the output function u is denoted by an operator

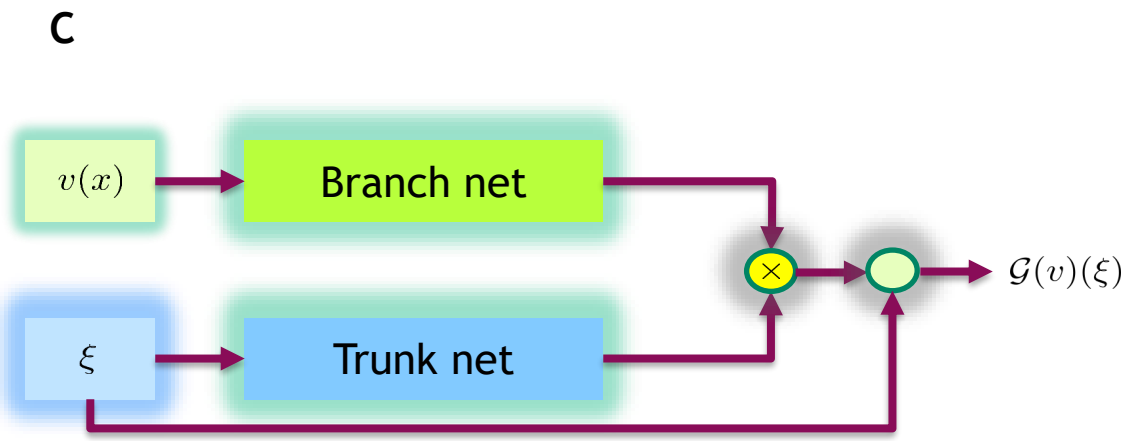
$$\boxed{\mathcal{G} : \mathcal{V} \ni v \mapsto u \in \mathcal{U}.}$$

- Lu L, Meng X, Cai S, Mao Z, Goswami S, Zhang Z, Karniadakis GE. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. arXiv preprint arXiv:2111.05512. 2021 Nov 10.

Deep Operator Network (DeepONet)

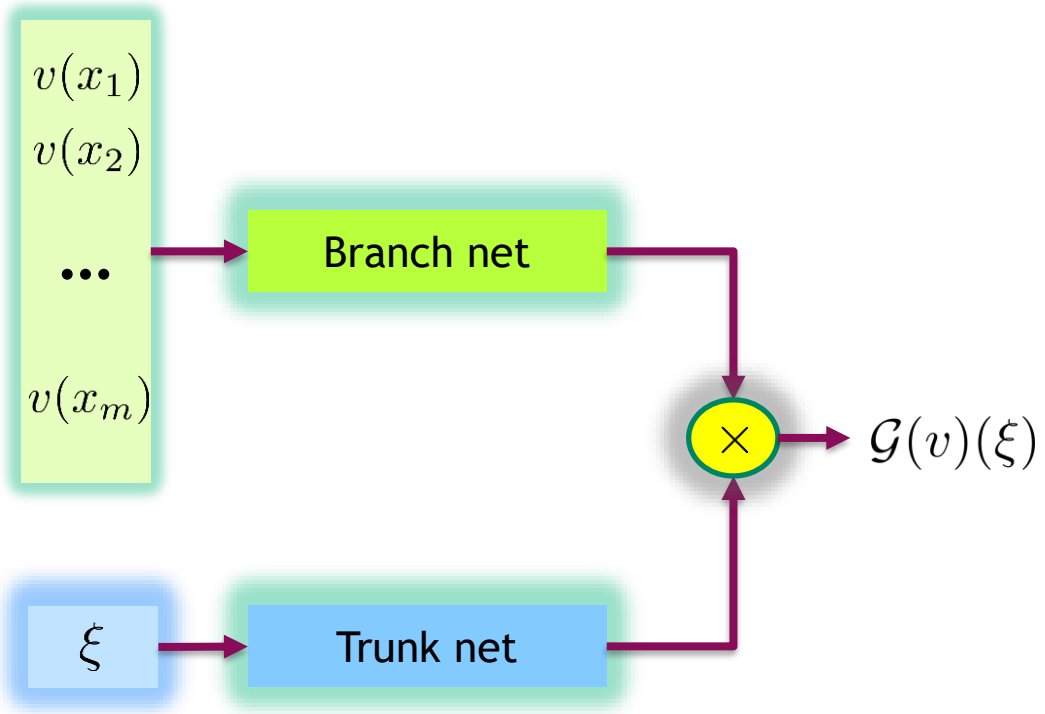




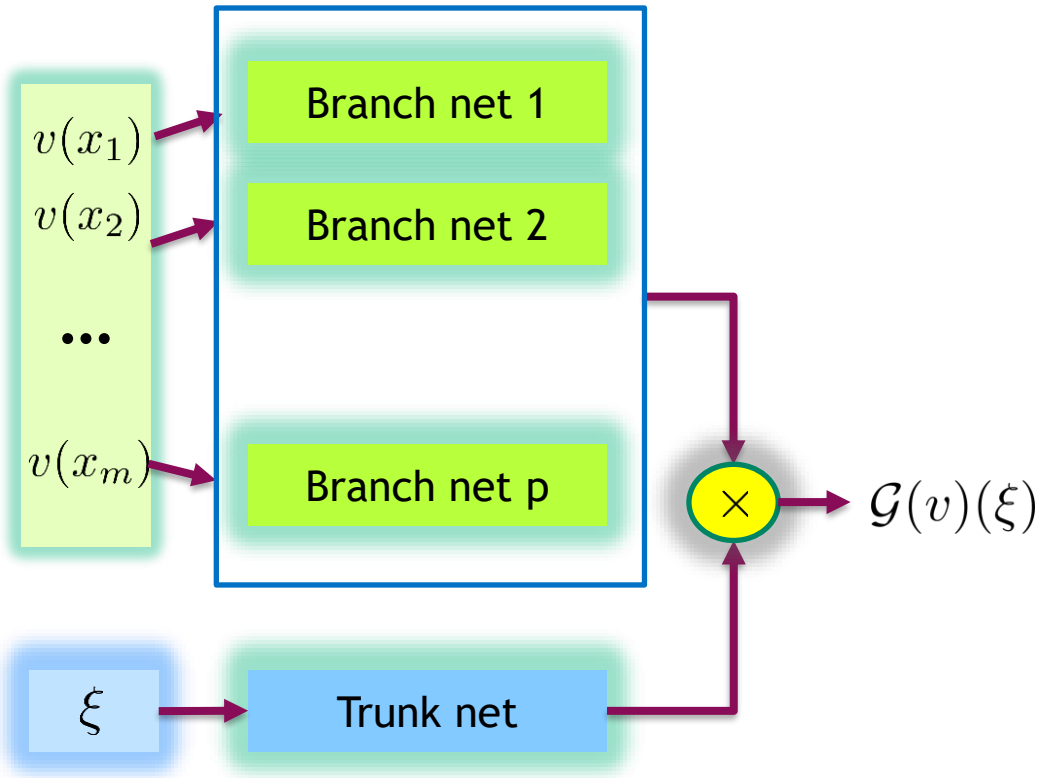


- Lu L, Meng X, Cai S, Mao Z, Goswami S, Zhang Z, Karniadakis GE. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. arXiv preprint arXiv:2111.05512. 2021 Nov 10.

Stacked/Unstacked DeepONet



Unstacked DeepONet



Stacked DeepONet

- Lu L, Meng X, Cai S, Mao Z, Goswami S, Zhang Z, Karniadakis GE. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. arXiv preprint arXiv:2111.05512. 2021 Nov 10.

DeepONet

DeepONet approximates both linear and nonlinear operators. This computational model consists of two deep neural networks:

- Branch net: encodes the discrete input function space.
  (takes the discretized function $v \in \mathrm{D}$ as input)
- Trunk net: encodes the domain of the output functions.
  (takes the coordinates $\xi \in D'$ as the input)

Output of the network :

$$\mathcal{G}(v)(\xi) = \sum_{k=1}^{p} b_k(v) t_k(\xi) + b_0,$$

where $b_0 \in \mathbb{R}$ is a bias. $\{b_1, b_2, \ldots, b_p\}$ are the p outputs of the branch net, and $\{t_1, t_2, \ldots, t_p\}$ are the p outputs of the trunk net.

(DeepONet is a high-level framework without restricting the branch net and the trunk net to any specific architecture.)

DeepONet for Integration

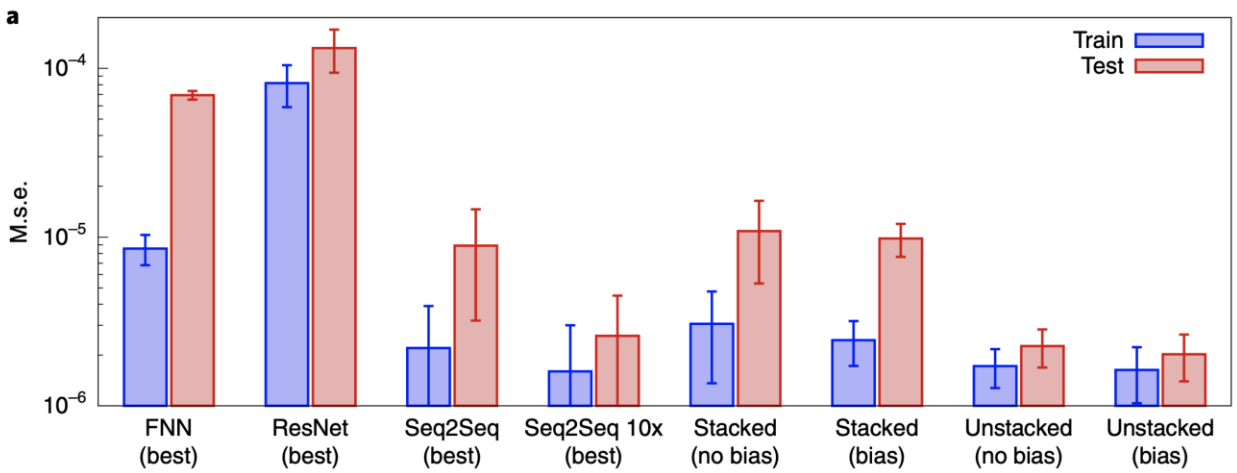
A simple ODE case

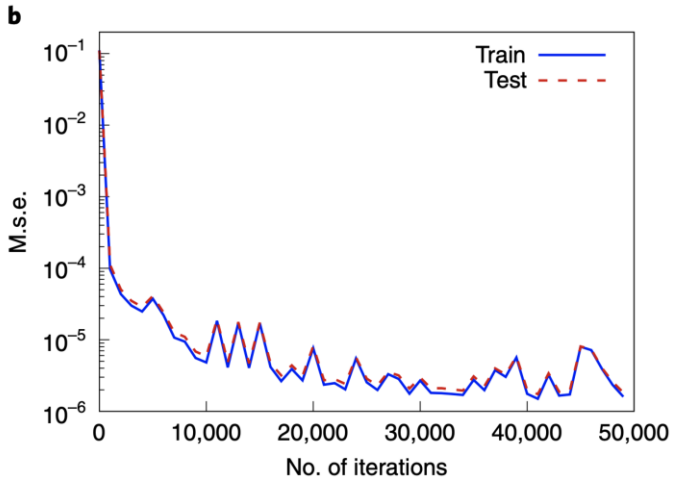
$$\frac{ds(x)}{dx} = u(x), \quad x \in [0, 1]$$

with an initial condition $s(0) = 0$.

$$G : u(x) \mapsto s(x) = \int_0^x u(\tau) d\tau$$

100 sensors, 10000×1 training points Very small generalization error!





Implementation of DeepONet for anti-derivative operator

1. Goal: Consider an ODE given by

$$\frac{\mathrm{d}u(x)}{dx} = g(u(x), v(x), x), \qquad x \in [0, 1],$$

with a boundary condition $u(0) = 0$. The goal is to predict $u(x)$ over the whole domain $[0, 1]$ for any $v(x)$

2. Here, we consider a linear problem by choosing $g(u(x), v(x), x) = v(x)$, which is equivalent to antiderivative operator and reads as

$$\frac{\mathrm{d}u(x)}{dx} = v(x), \qquad x \in [0, 1],$$

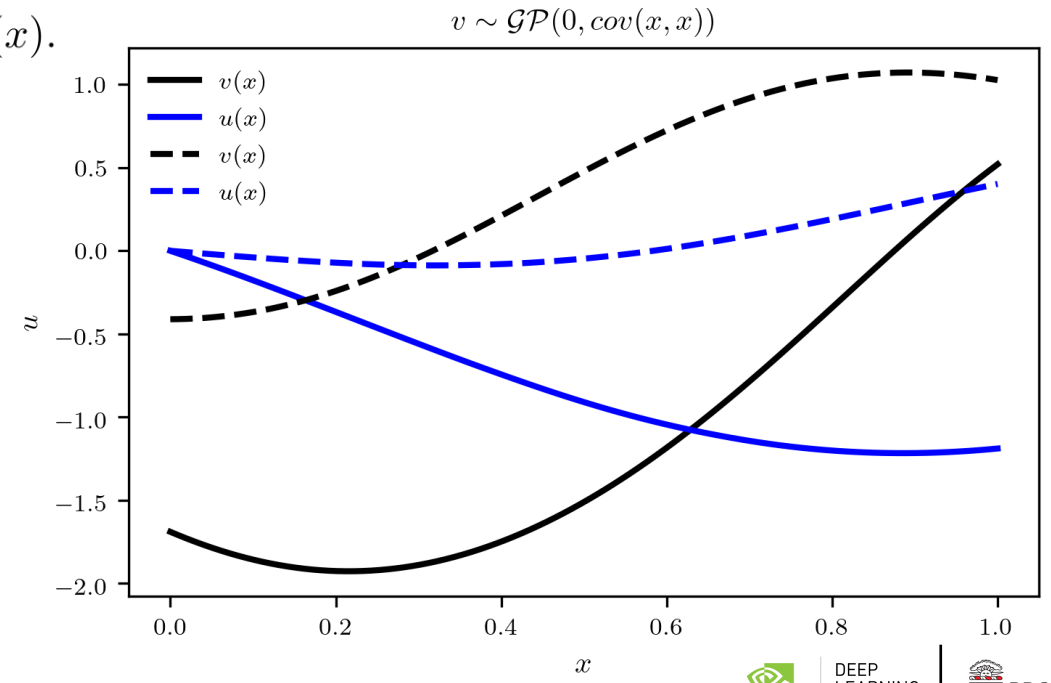
Implementation of DeepONet for anti-derivative operator

1. Data : Train Data: antiderivative_train.npz
   Test Data: antiderivative_test.npz

- Sample $v(\boldsymbol{x}) \sim \mathcal{GP}(0, \text{cov}(x, x))$: Refer Module 2: Lecture 1 on Multi-fidelity.
- For a sampled $v(x)$ compute following initial value problem using any numerical scheme for ODE.

$$\frac{du(x)}{dx} = v(x) \qquad x \in [0, 1]$$

- Right image shows examples of computed two $u(x)$ for two sampled $f(x)$.



Implementation of DeepONet for anti-derivative operator

2. Import Modules

```
import os
import time
import tensorflow as tf
import numpy as np
import scipy.io as io
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as io
np.random.seed(1234)
import sys
```

3. Class for DeepONet Architecture

```
class DNN:
    def __init__(self):
        pass
    #initialization for DNNs
    def hyper_initial(self, layers):
        L = len(layers)
        W = []
        b = []
        for l in range(1, L):
            in_dim = layers[l-1]
            out_dim = layers[l]
            std = np.sqrt(2.0/(in_dim+out_dim))
            weight = tf.Variable(tf.random.truncated_normal(shape=(in_dim, out_dim), stddev = std))
            bias = tf.Variable(tf.random.truncated_normal(shape=(1, out_dim), stddev = std))
            W.append(weight)
            b.append(bias)
        return W, b

    def fnn_B(self, X, W, b):
        A = X
        A = tf.cast(A, dtype=tf.float32)
        L = len(W)
        for i in range(L-1):
            A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))
        Y = tf.add(tf.matmul(A, W[-1]), b[-1])
        return Y

    def fnn_T(self, X, W, b):
        A = X #2.*(X - Xmin)/(Xmax - Xmin) - 1.0
        A = tf.cast(A, dtype=tf.float32)
        L = len(W)
        for i in range(L-1):
            A = tf.tanh(tf.add(tf.matmul(A, W[i]), b[i]))
        Y = tf.add(tf.matmul(A, W[-1]), b[-1])
        return Y
```

Branch Net

Trunk Net

Implementation of DeepONet for Anti-derivative operator

4. Prepare datasets for Training and Testing: Contd.

Training Data

Testing Data

Branch Dimension

Latent Dimension

```
#Load Training Data
d = np.load("antiderivative_train.npz", allow_pickle=True)
v_train, x_train, u_train = d["X"][0], d["X"][1], d["y"]


d = np.load("antiderivative_test.npz", allow_pickle=True)
v_test, x_test, u_test = d["X"][0], d["X"][1], d["y"]

#input dimension for Branch Net
u_dim = 100

#output dimension for Branch and Trunk Net
G_dim = 100

#Branch Net
layers_f = [u_dim] + [40]*2 + [G_dim]

# Trunk dim
x_dim = 1

#Trunk Net
layers_x = [x_dim] + [40]*2 + [G_dim]

model = DNN()
```

Implementation of DeepONet for Anti-derivative operator

5. Training and Testing routines

```
def train_step(model, W_branch, b_branch, W_trunk, b_trunk, v, x, u, opt):
    train_vars = W_branch + b_branch + W_trunk + b_trunk
    with tf.GradientTape() as tape:
        tape.watch([W_branch, b_branch, W_trunk, b_trunk])
        u_out_branch = model.fnn_B(v, W_branch, b_branch)
        u_out_trunk = model.fnn_T(x, W_trunk, b_trunk)
        u_pred = tf.einsum('ij,jj->ij', u_out_branch, u_out_trunk)
        loss = tf.reduce_mean(tf.square(u_pred - u))
        grads = tape.gradient(loss, train_vars)
        opt.apply_gradients(zip(grads, train_vars))
        return loss, u_pred

def test_step(model, W_branch, b_branch, W_trunk, b_trunk, v, x, u, opt):
    x_test_tf = tf.convert_to_tensor(x, dtype=tf.float32)
    v_test_tf = tf.convert_to_tensor(v, dtype=tf.float32)
    u_test_tf = tf.convert_to_tensor(u, dtype=tf.float32)
    u_out_branch = model.fnn_B(v_test_tf, W_branch, b_branch)
    u_out_trunk = model.fnn_T(x_test_tf, W_trunk, b_trunk)
    u_pred = tf.einsum('ij,jj->ij', u_out_branch, u_out_trunk)
    loss = tf.reduce_mean(tf.square(u_pred - u))
    return loss, u_pred
```

6. Initializing the model and hyperparameters

```
W_branch, b_branch = model.hyper_initial(layers_f)
W_trunk, b_trunk =   model.hyper_initial(layers_x)

n = 0
nmax = 10000
lr = 5e-4
start_time = time.perf_counter()
time_step_0 = time.perf_counter()
optimizer = tf.optimizers.Adam(learning_rate=lr)

train_err_list = []
test_err_list = []
train_loss_list = []
test_loss_list = []
```

Implementation of DeepONet for anti-derivative operator

7. Driver Routine: Training and Testing

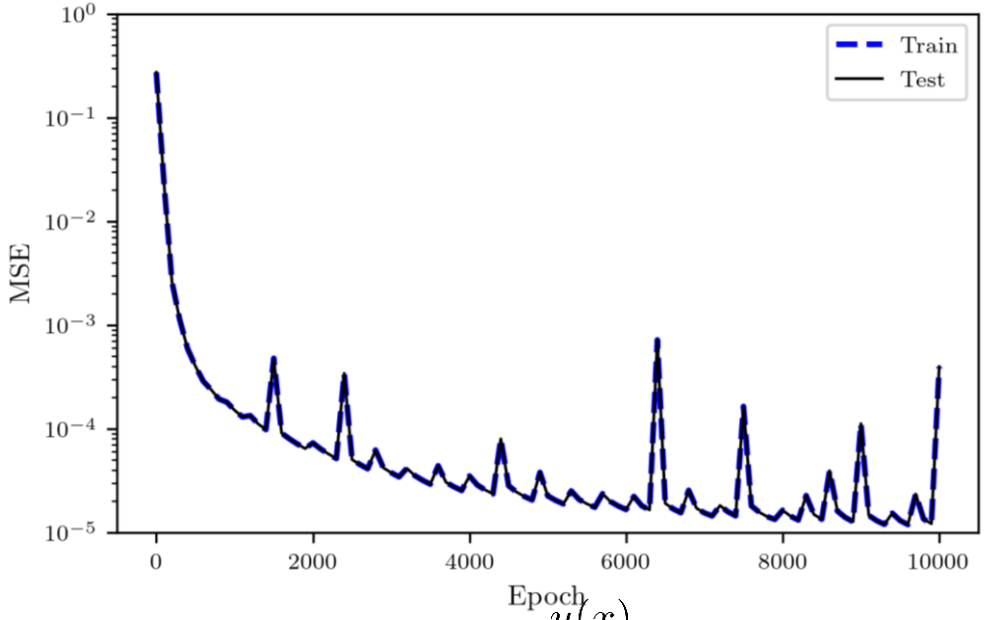
```
while n <= nmax:
    x_train_tf = tf.convert_to_tensor(x_train, dtype=tf.float32)
    v_train_tf = tf.convert_to_tensor(v_train, dtype=tf.float32)
    u_train_tf = tf.convert_to_tensor(u_train, dtype=tf.float32)
    loss_train, u_train_pred = train_step(model, W_branch, b_branch, W_trunk, \
                                          b_trunk, v_train_tf, x_train_tf, u_train_tf, \
                               optimizer)
    err_train = np.mean(np.linalg.norm(u_train - u_train_pred, 2, axis=1)/\
                np.linalg.norm(u_train , 2, axis=1))

    loss_test, u_test_pred = test_step(model, W_branch, b_branch, W_trunk, b_trunk, v_test, x_test, u_test, \
                               optimizer)
    err_test = np.mean(np.linalg.norm(u_test - u_test_pred, 2, axis=1)/\
            np.linalg.norm(u_test , 2, axis=1))

    if n % 100 == 0:
        print(f"Iteration: {n} Train_loss:{loss_train}, Test_loss: {loss_test}")
    train_err_list.append(err_train)
    test_err_list.append(err_test)
    train_loss_list.append(loss_train)
    test_loss_list.append(loss_test)
    n = n + 1
```

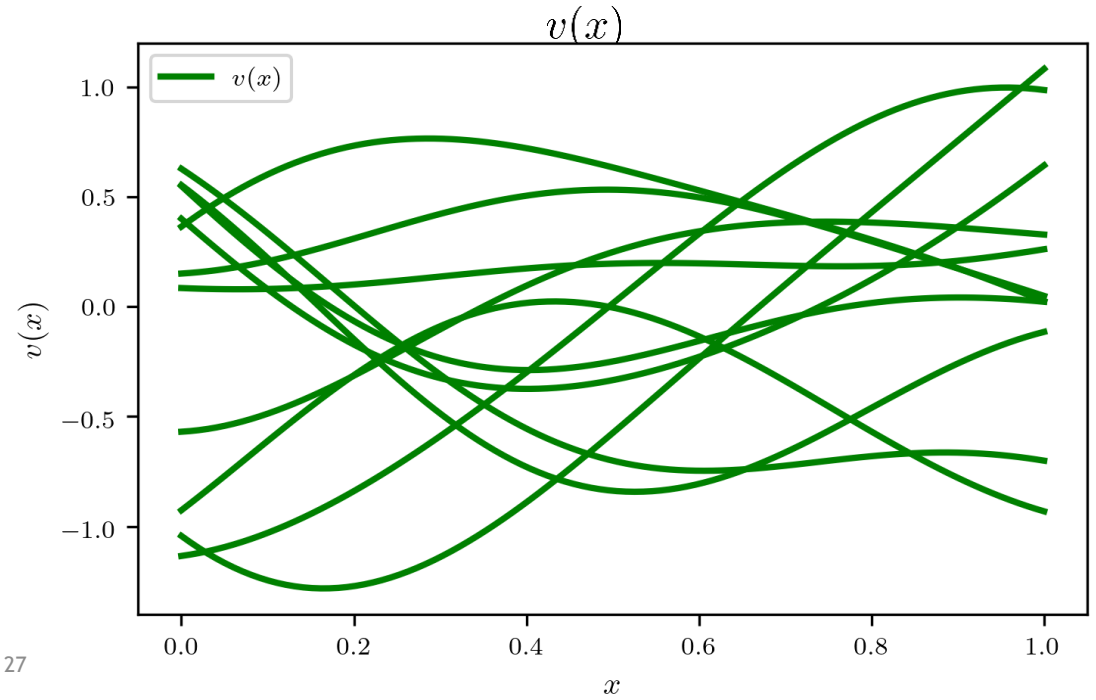
Implementation of DeepONet for Anti-derivative operator

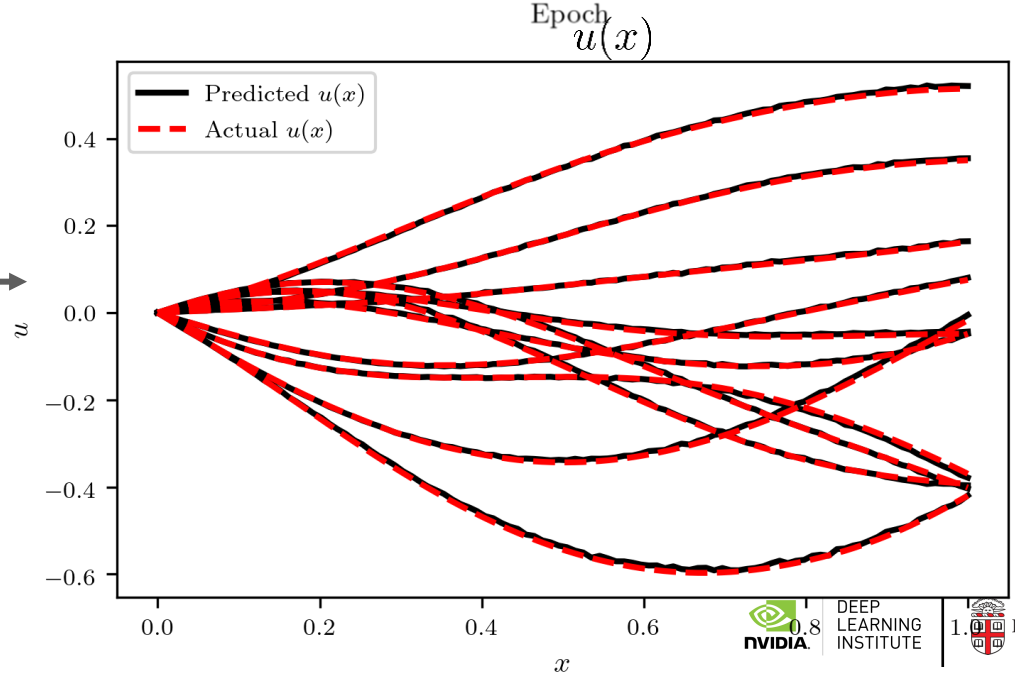
8. Results: Training Loss vs Test Error

Anti-derivative equation reads

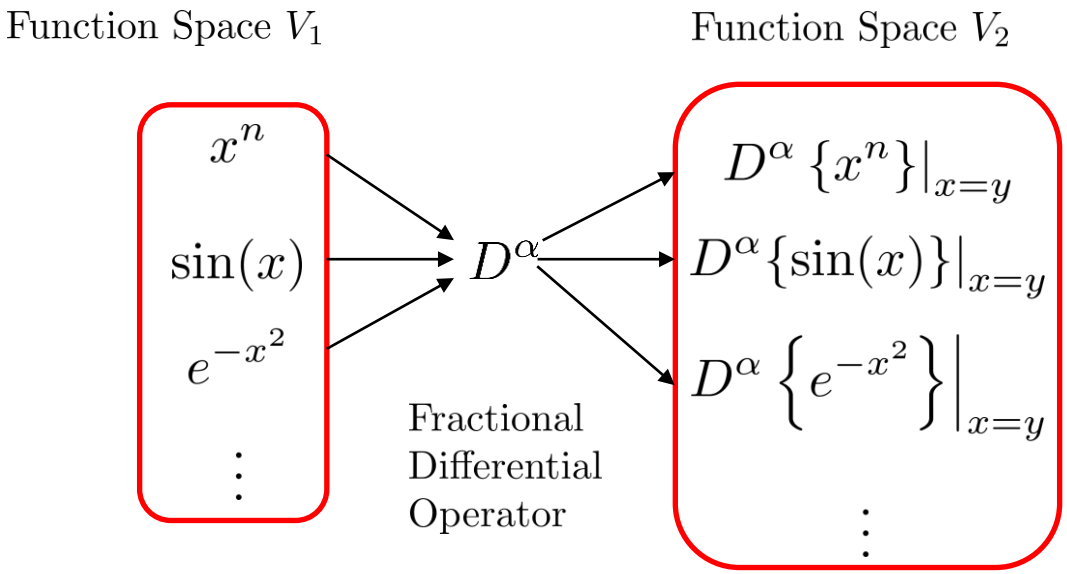
$$\frac{du(x)}{dx} = v(x); \qquad x \in [0, 1]$$







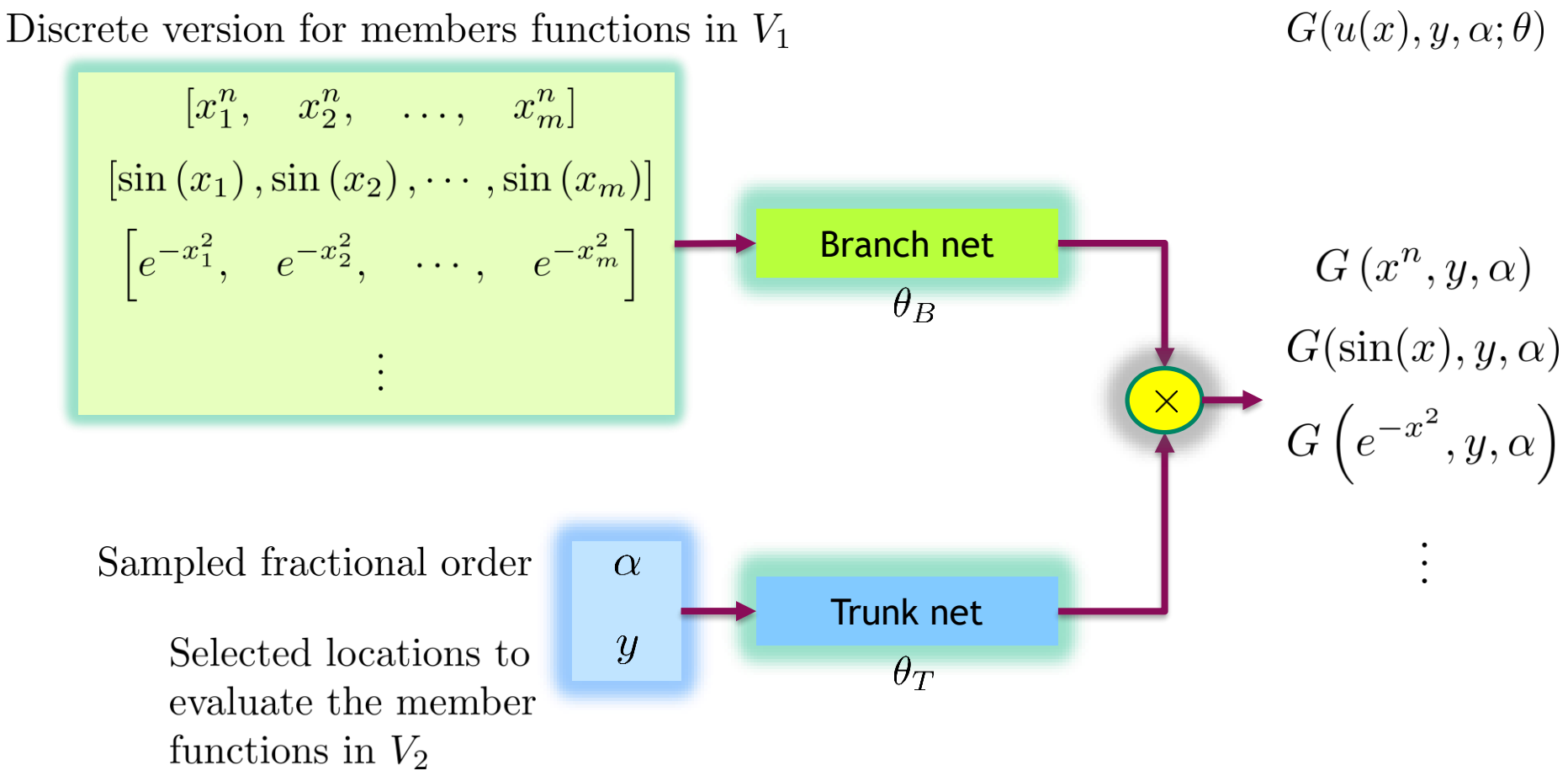
DeepONet for Fractional Differential Operators



Train a DeepONet $G()$ to approximate D^α by minimizing the following loss function

$$L\left(\boldsymbol{\theta}_B, \boldsymbol{\theta}_T\right) = \left[D^\alpha \{x^n\}|_{x=y} - G\left(x^n, y, \alpha; \boldsymbol{\theta}_B, \boldsymbol{\theta}_T\right)\right]^2$$
$$+ \left[D^\alpha\{\sin(x)\}|_{x=y} - G\left(\sin(x), y, \alpha; \boldsymbol{\theta}_B, \boldsymbol{\theta}_T\right)\right]^2$$
$$+ \left[D^\alpha \left\{e^{-x^2}\right\}\Big|_{x=y} - G\left(e^{-x^2}, y, \alpha; \boldsymbol{\theta}_B, \boldsymbol{\theta}_T\right)\right]^2 + \cdots$$

DeepONet for Fractional Differential Operators



Approximating 1D Caputo Fractional Differential Operator using DeepONet

$$G(u(x), y, \alpha) \approx \left(D^{\alpha} u\right)(y) = \frac{1}{\Gamma(1-\alpha)} \int_0^y (y-t)^{-\alpha} u'(t) dt$$

$u \in V$ (certain function space), $y \in [0,1], \alpha \in (0,1)$

Loss: $\min_\theta \sum_{ijk} \left[G\left(\boldsymbol{u}^i, y^j, \alpha^k; \boldsymbol{\theta}\right) - \left(\widetilde{D}^{\alpha^k} \boldsymbol{u}^i\right)\left(y^j\right) \right]^2$

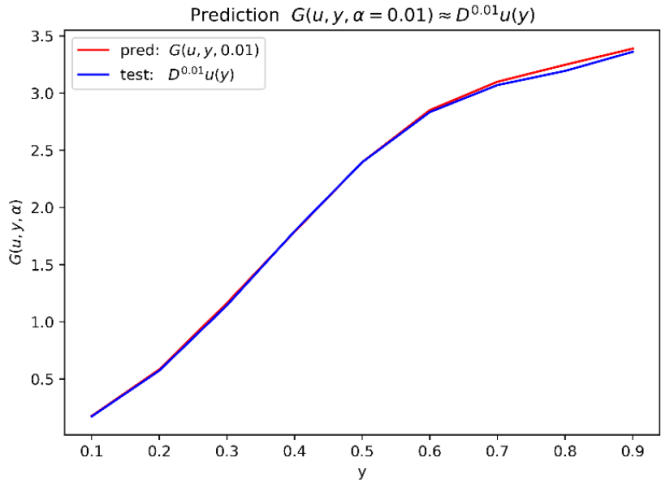
- $\boldsymbol{\theta}$ collects tunable parameters of G
- $\widetilde{D}^{\alpha}$ is the discrete form of D^{α} (E.g., L_1 scheme)

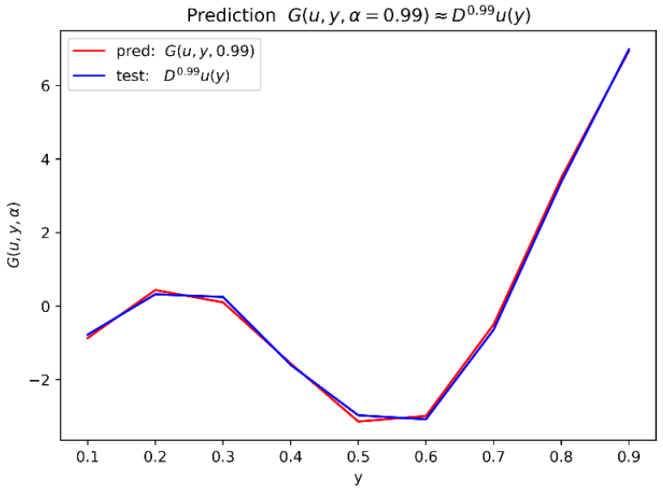
Input of $G() : \left(\boldsymbol{u}^i, y^j, \alpha^k\right) \in R^{m+2}$

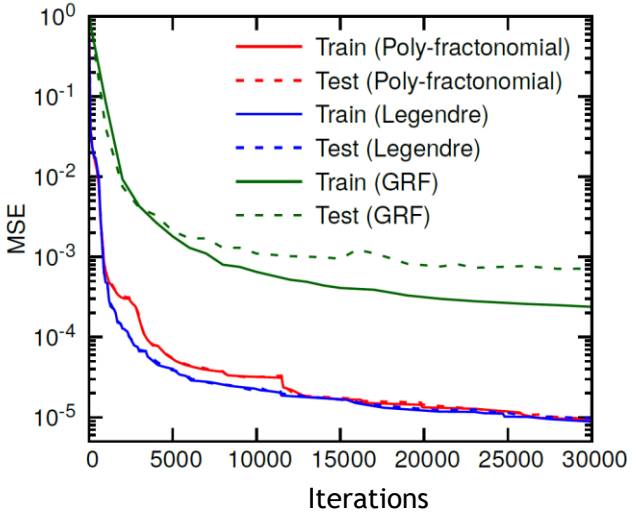
$$\begin{cases} \boldsymbol{u}^i := \left[u^i(x_1), u^i(x_2), \ldots, u^i(x_m)\right] \in R^m, \\ \qquad i = 1, 2, \ldots, N_u \\ y^j \in [0,1], \quad j = 1, 2, \ldots, N_y \\ \alpha^k \in (0,1), \quad k = 1, 2, \ldots, N_\alpha \end{cases}$$

Output of $G() : G\left(\boldsymbol{u}^i, y^j, \alpha^k\right) \in R$

Approximating 1D Caputo Fractional Differential Operator using DeepONet



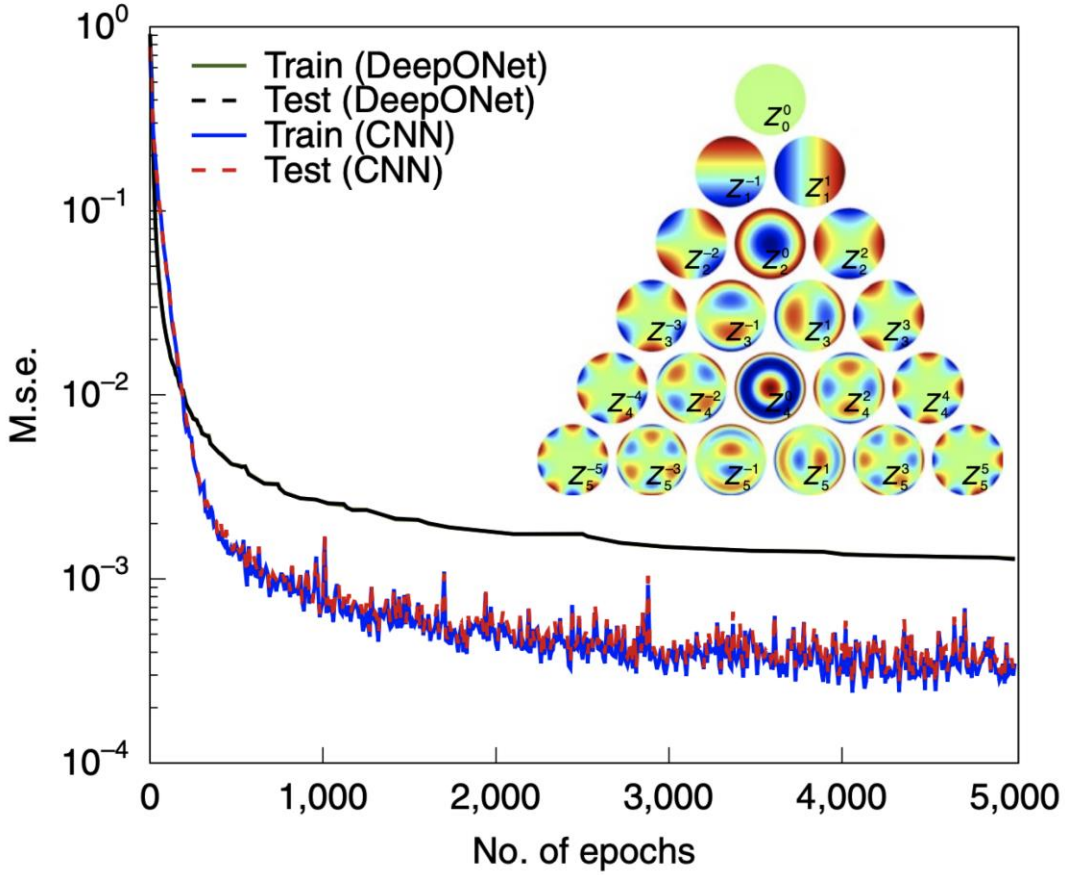




Training set: $N_u = 10000, N_y = 10, N_a = 10, m = 40$ Number of training points: 1 million $\boldsymbol{u}^i$ are realization of GPs with the SE kernel (characteristic length 0.1)

Test set: $N_u = 10000, N_y = 10, N_a = 10, m = 40$ Number of test points: 1 million $\boldsymbol{u}^i$ are realization of GPs with the SE kernel (characteristic length 0.3)
Test relative error: 2.3%

Approximating 2D Riesz Fractional Laplacian $D^{\alpha} = (-\Delta)^{\alpha/2}$ defined on a unit disk



- Using Zernike Polynomials to represent the input

The function space V_1 where u stays is now assumed to be an orthogonal polynomial space spanned by 36 Zernike polynomials:

$$V_1 = \left\{ u(r,\theta) = \sum_{n=0}^{7} \sum_{m=-n}^{n} a_{mn} Z_n^m(r,\theta), r \in [0,1], \theta \in [0,2\pi] \right\}$$

where $Z_n^m(r,\theta) \equiv 0$ for $n-m$ odd

The samples of u were generated by randomly selecting the coefficients $a_{mn} \in [-1,1]$

DeepONet for Approximating Functionals: Predicting unsteady pressure and lift/drag-force coefficients

- Simulation of NACA0012 airfoil (Nektar by Z. Wang)

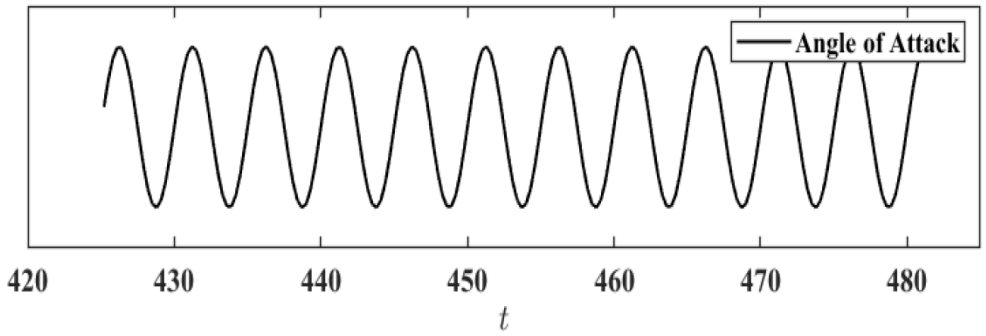
inflow

$$u = U_\infty \cos\left(\frac{\alpha_0 \pi}{180} \times \frac{\sin(2f\pi t) + 1.0}{2}\right),$$

$$v = U_\infty \sin\left(\frac{\alpha_0 \pi}{180} \times \frac{\sin(2f\pi t) + 1.0}{2}\right),$$

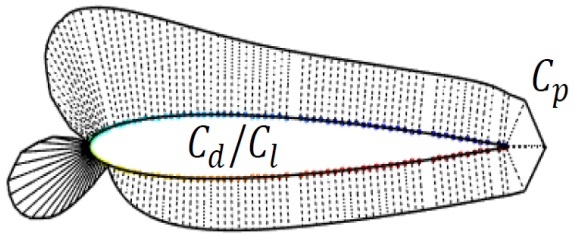
$$\alpha = 15^o \quad f = 0.2 \quad U_\infty = 1 \quad Re = 2500$$

- Generate time-dependent AOA



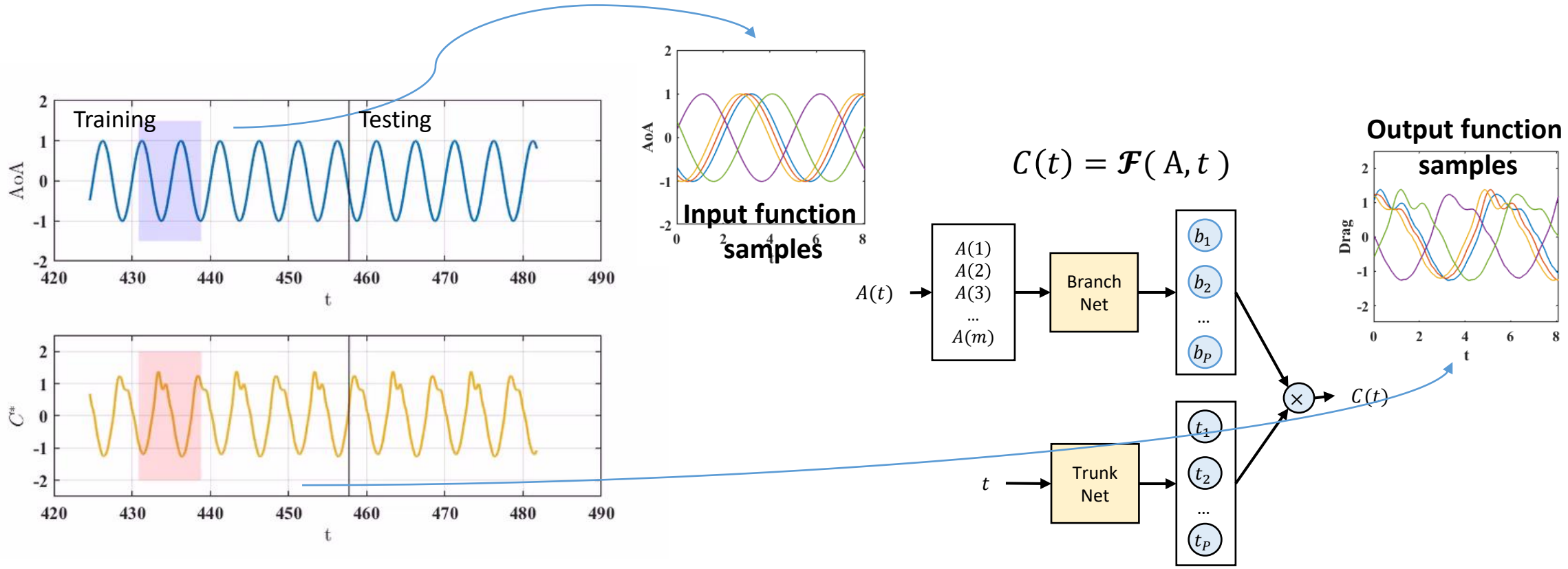
predict

- Time-dependent coefficients of drag, lift, pressure



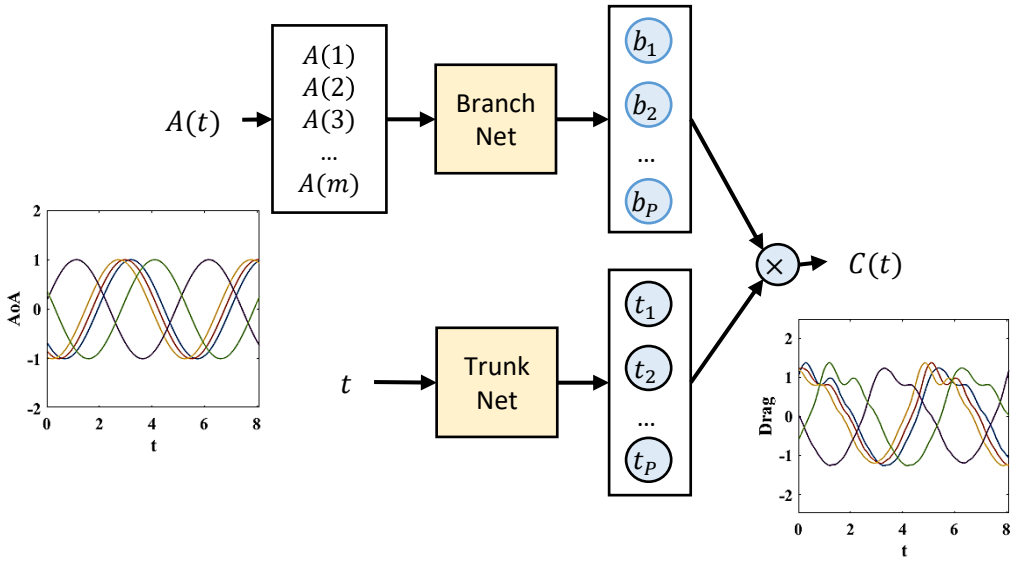
Using Standard DeepONet

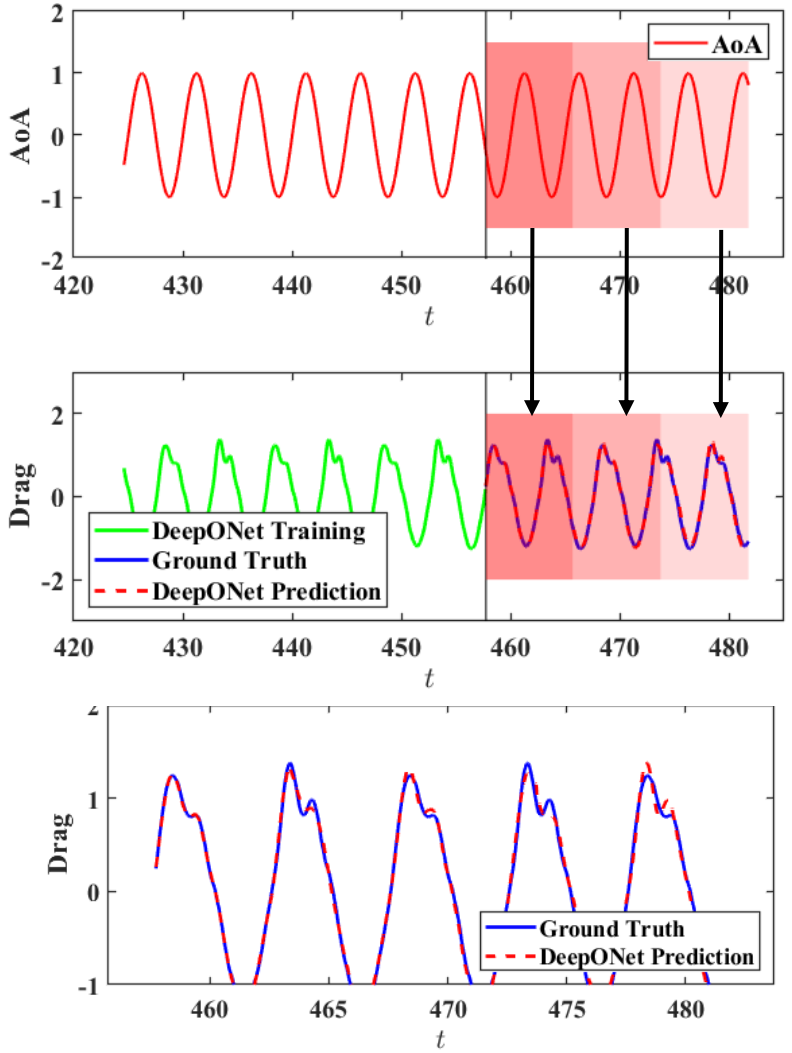
- Predicting drag/lift coefficient
- Cut the time-dependent signal into pieces, to generate a bunch of input-output pairs



Coefficient of Drag

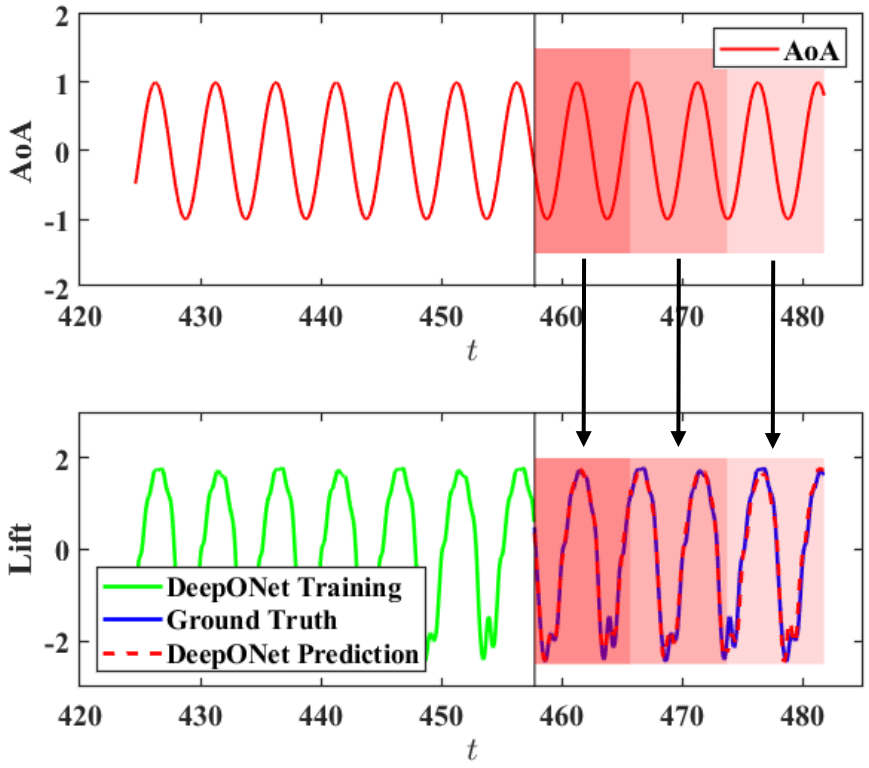
- Result of predicting drag coefficient
- Each input/output function covers Δt = 8.1

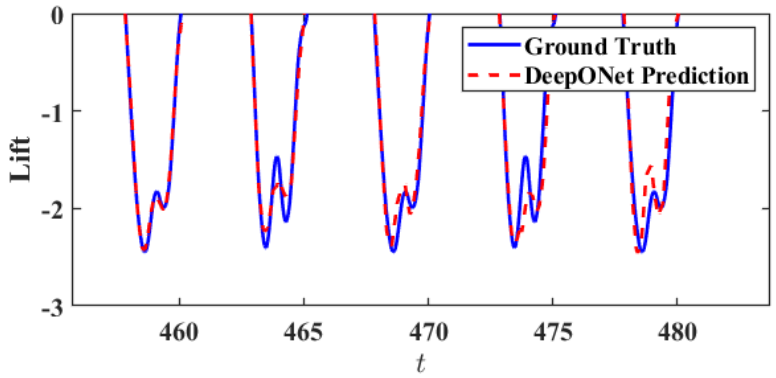




Coefficient of Lift

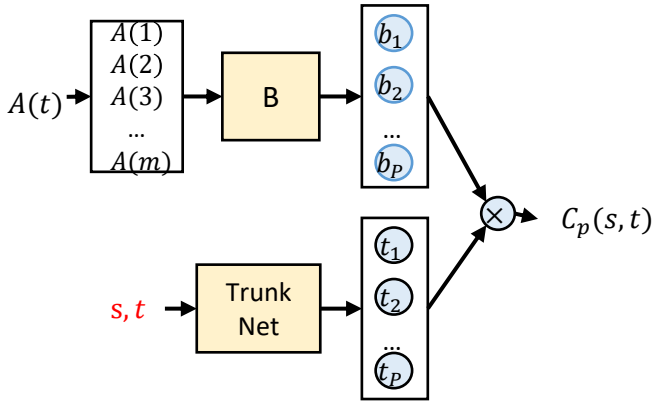
- Result of predicting lift coefficient
- Each input/output function covers $\Delta t = 8.1$



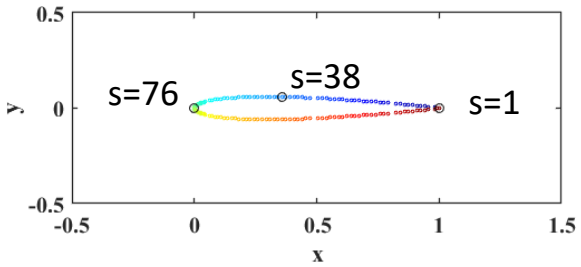


Coefficient of Pressure

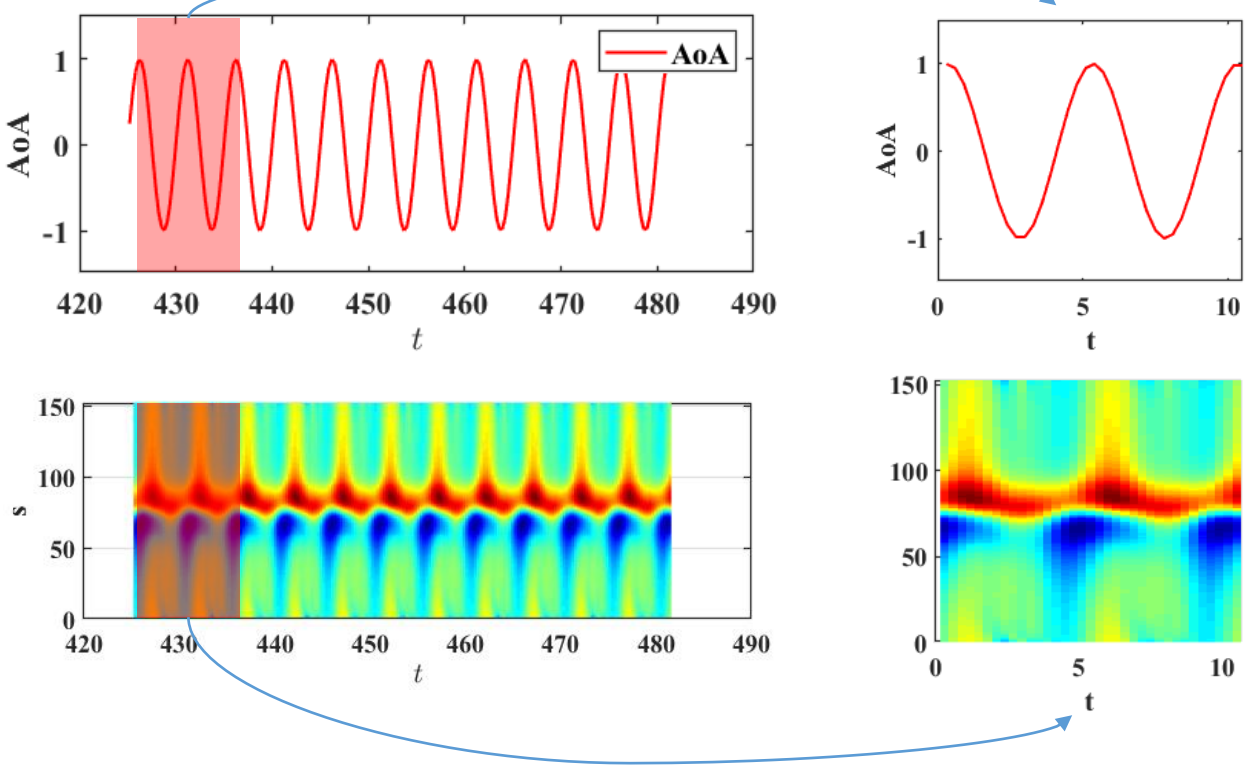
- Result of predicting pressure coefficient $C(s,t)$ - s: location index



$$C_p(s,t) = \mathcal{F}(\mathrm{A},(s,t))$$

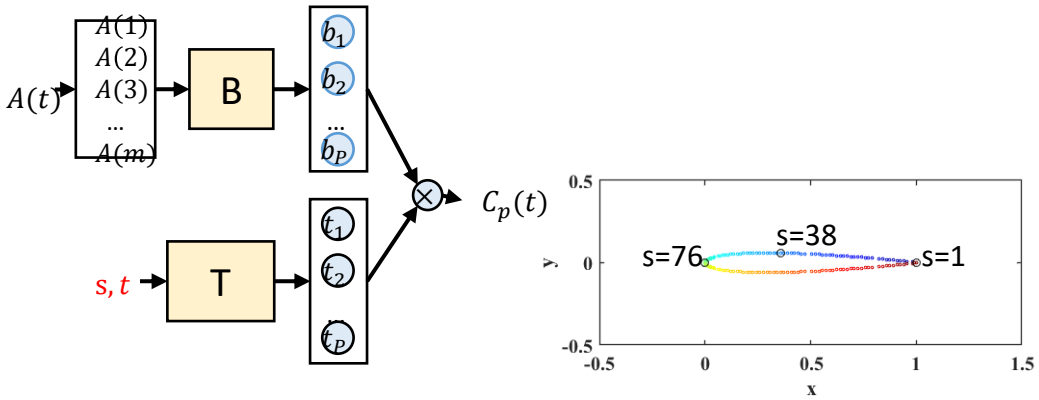


- Input-output pair

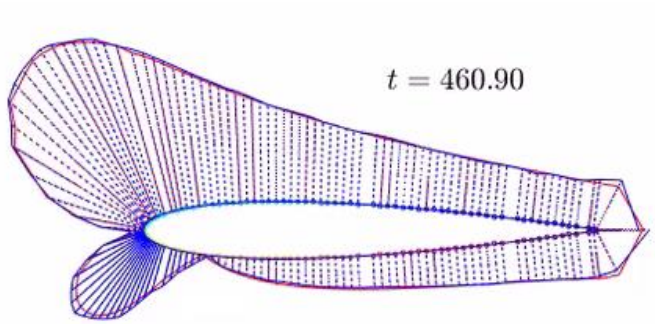


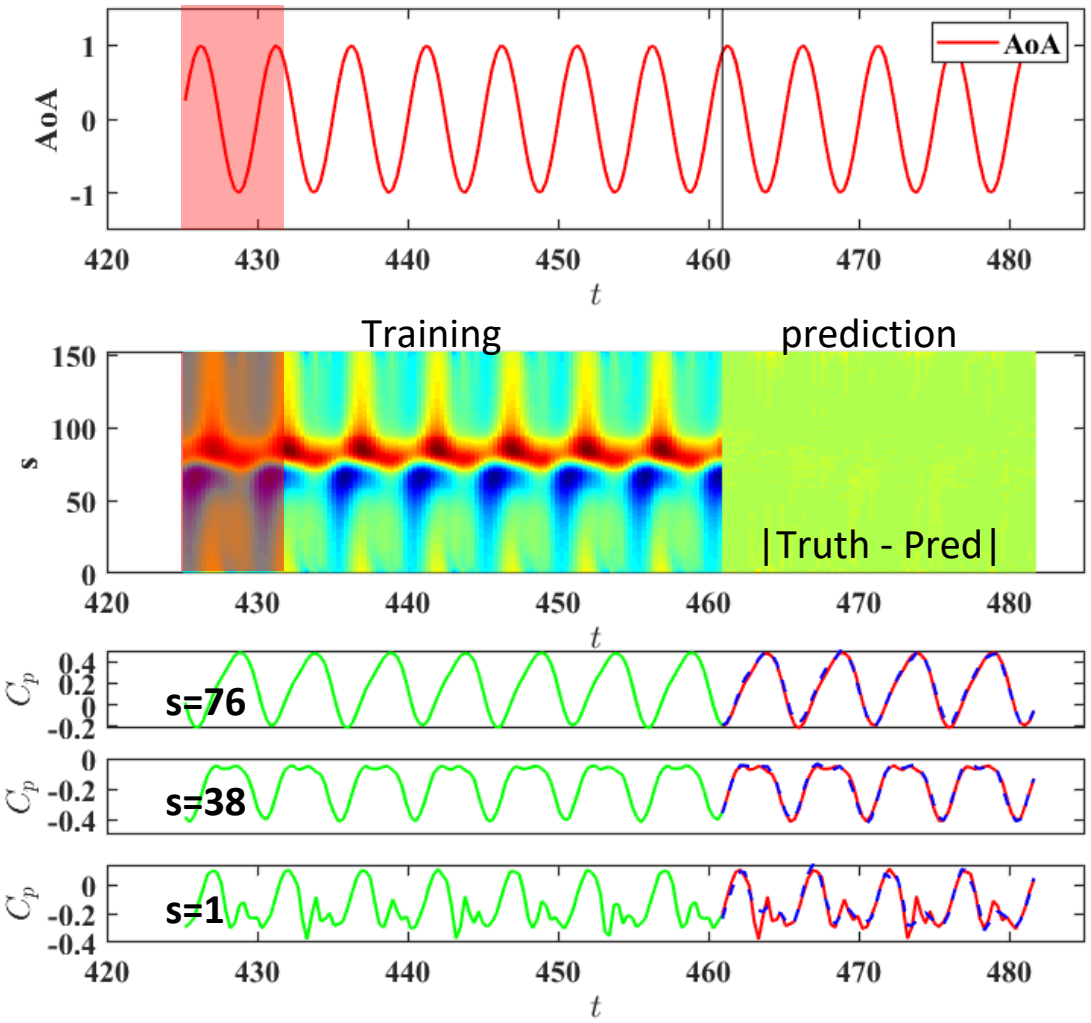
Coefficient of Pressure

- Result of predicting pressure coefficient $C(s,t)$
- Each input/output function covers Δt = 10.2



Red: truth
Blue: prediction
Solid: positive
Dash: negative



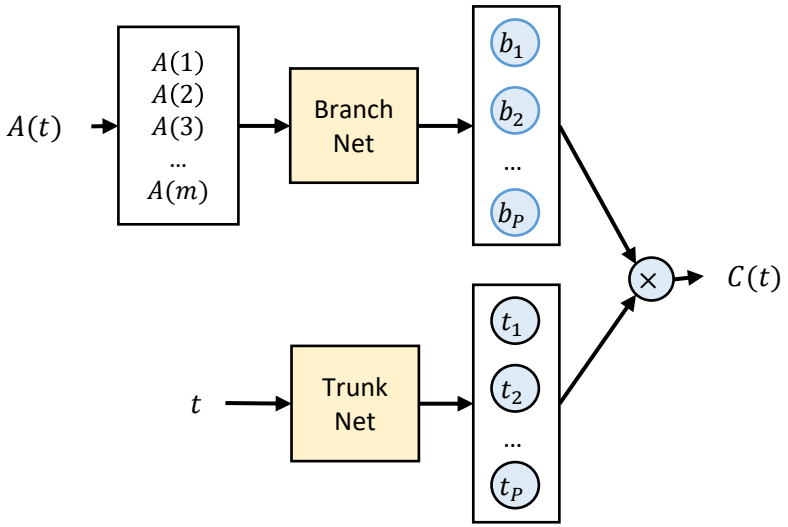


Modification of DeepONet

- **Predicting the current state $C(t)$**

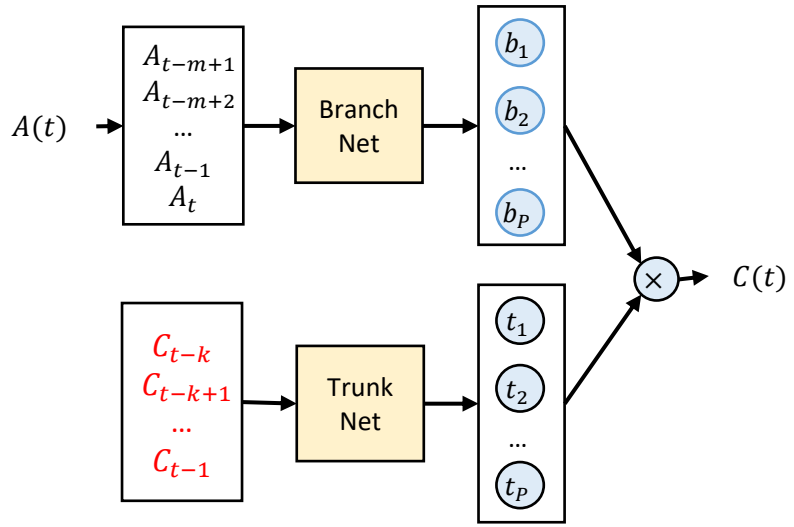
For predicting time series data
Replace the **time coordinate** with the **memory of prediction**
- Recurrent neural network (e.g., LSTM)

- Standard DeepONet $C(t) = \mathcal{F}(\mathrm{A}, t)$



1. A in a time window $(A_{t-m+1}, \dots, A_{t-1}, A_t)$
2. Time coordinate t

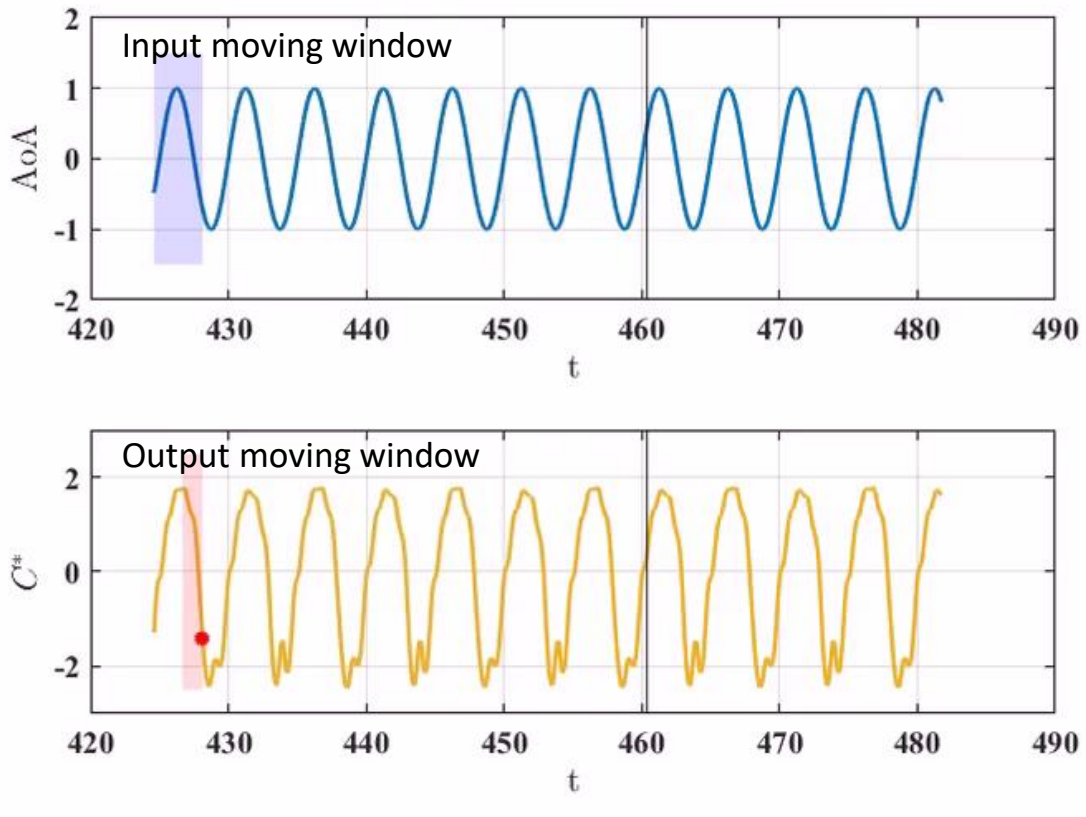
- Modified network $C(t) = \mathcal{F}(\mathrm{A}, C_{t-1}, C_{t-2}, \cdots, C_{t-k})$



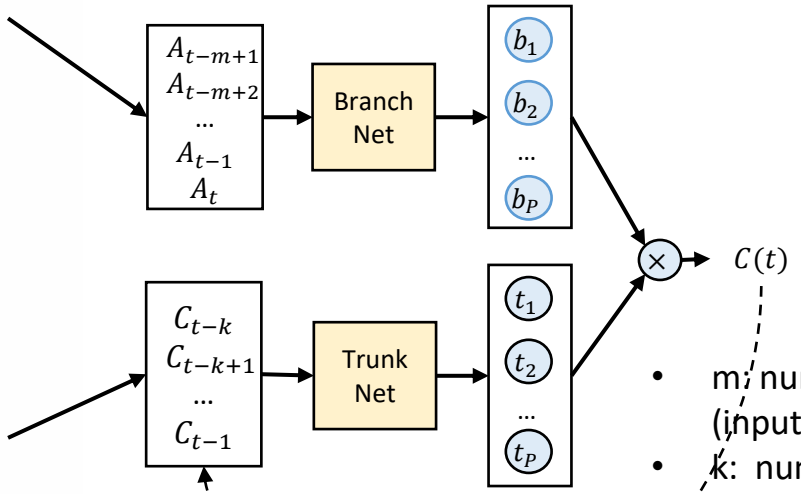
1. A in a time window $(A_{t-m+1}, \dots, A_{t-1}, A_t)$
2. History state in a small time window $C_{t-k}, \dots, C_{t-1}$

Modification of DeepONet

- **Make use of the history information**



$$C(t) = \mathcal{F}(\,\mathrm{A},\qquad (C_{t-1}, C_{t-2}, \cdots, C_{t-k})\,)$$



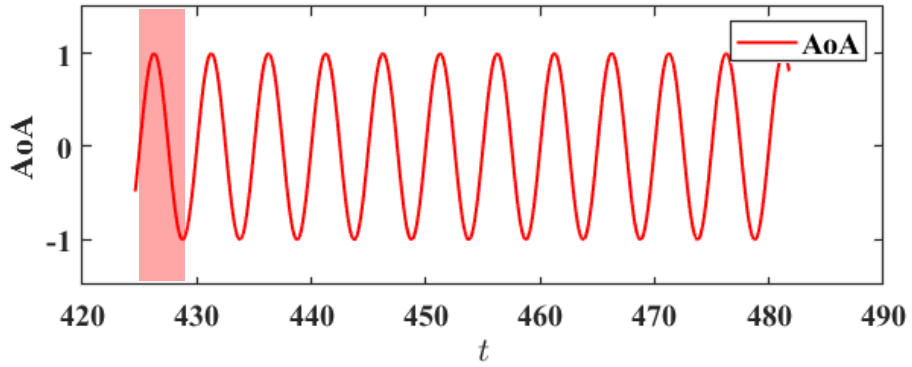
- m: number of input points (input moving window)
- k: number of the history points (output moving window)

When moving forward

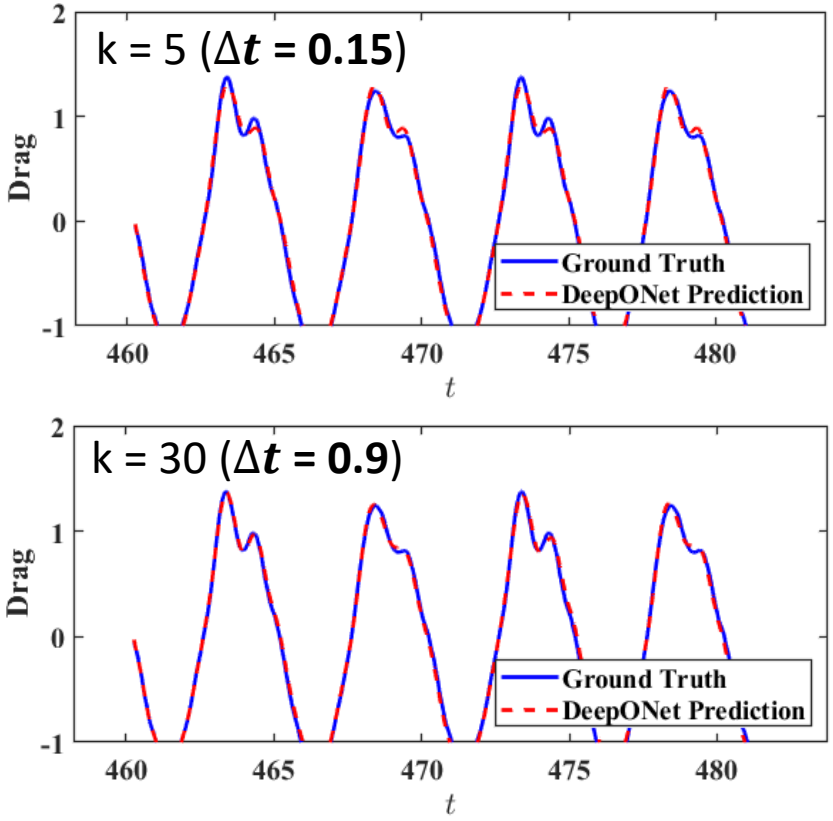
Coefficient of Drag

- Result of predicting drag coefficient

m = 120 (Δt = 3.6)



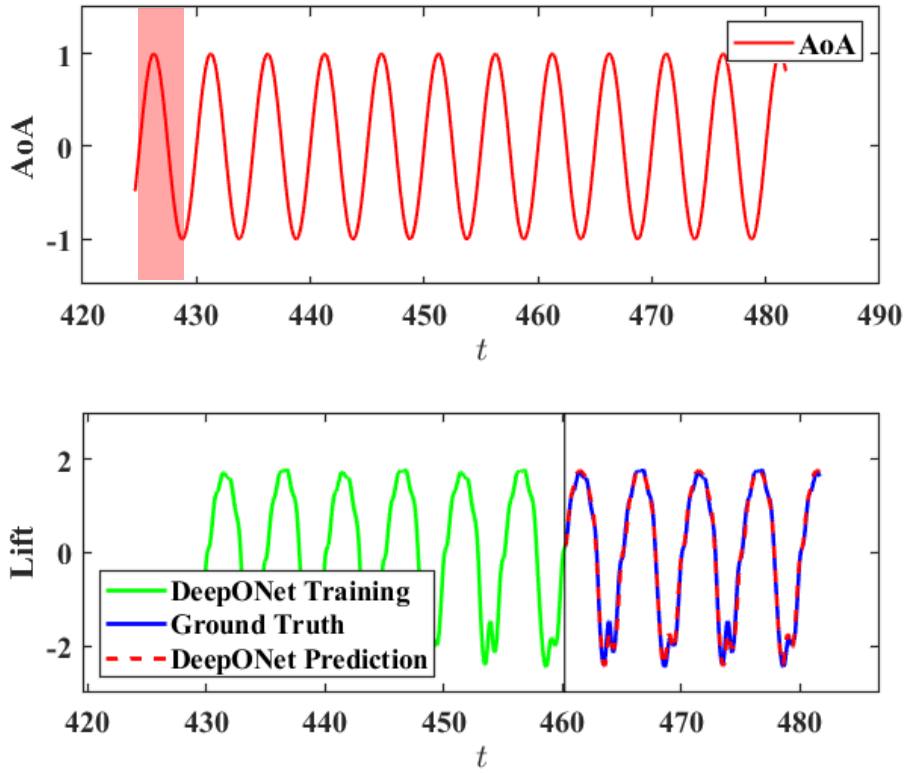
Size of the output moving window (i.e., history points)



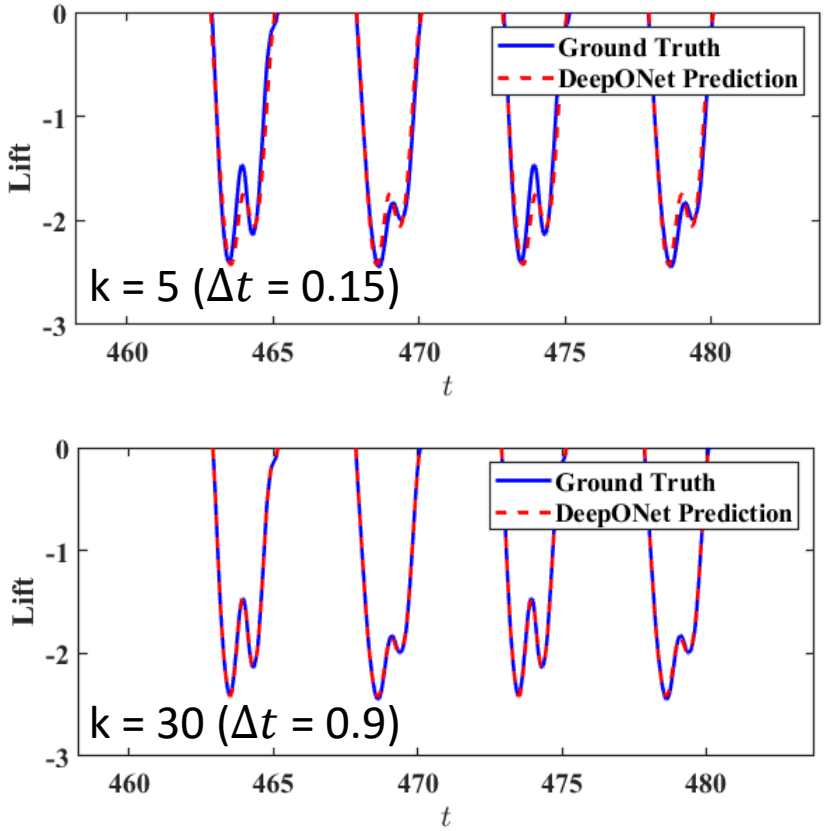
Coefficient of Lift

- Result of predicting lift coefficient

m = 120 (Δt = 3.6)



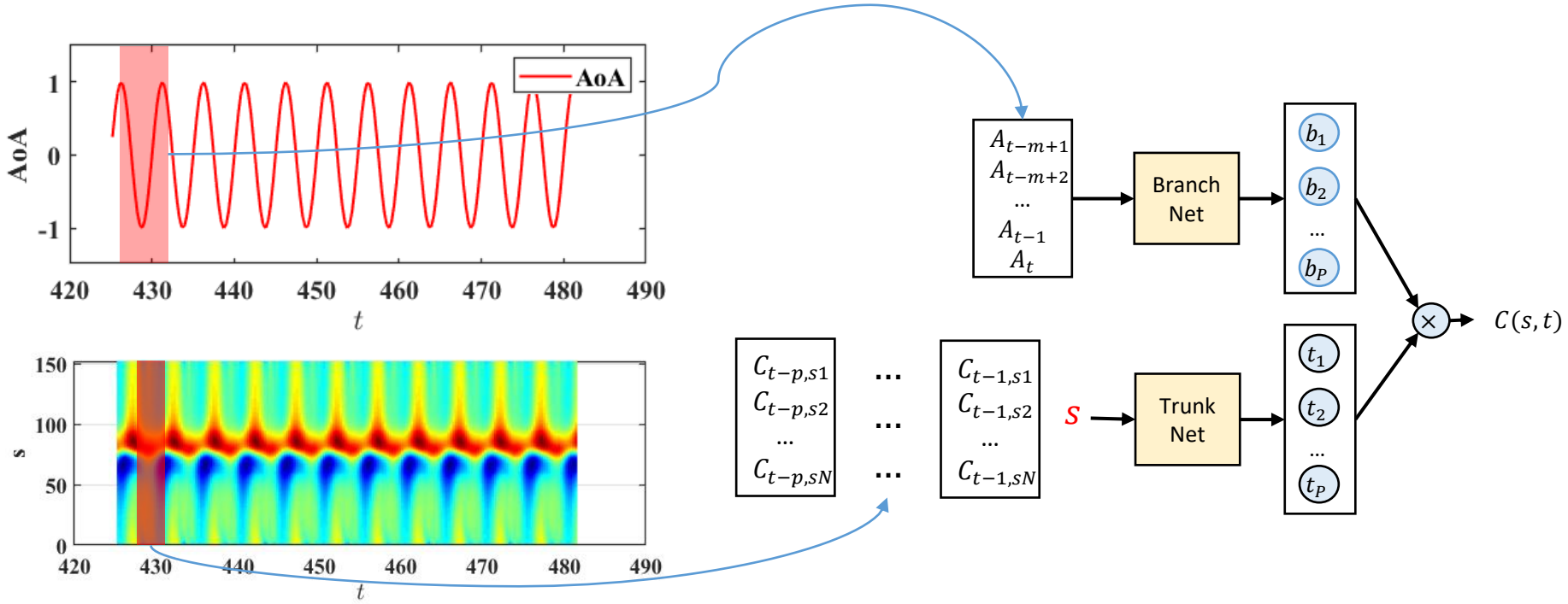
Size of the output moving window (i.e., history points)



Coefficient of Pressure

Replace the **time coordinate** with the **memory of prediction**

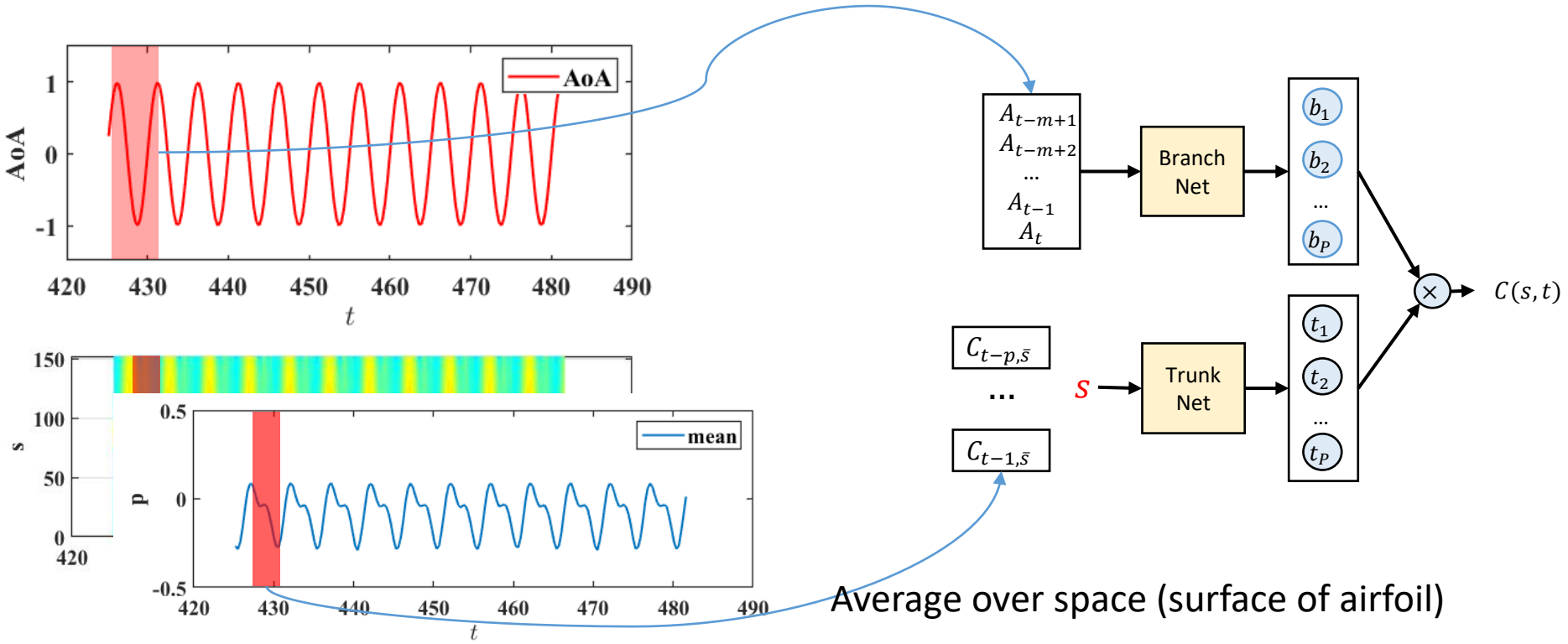
- Result of predicting pressure coefficient $C(s,t)$ - s: location index
- History information $C_{t-1}(\boldsymbol{s}), \ldots C_{t-k}(\boldsymbol{s})$ goes to Trunk net (together with the spatial coordinate)



- Too many inputs for trunk net
- The spatial coordinate contributes little
- The performance is unstable and sensitive to the prediction error

Coefficient of Pressure

- Result of predicting pressure coefficient $C(s,t)$ - s: location index
- History information $C_{t-1}(\bar{s}), \dots C_{t-k}(\bar{s})$ goes to Trunk net (together with the spatial coordinate)

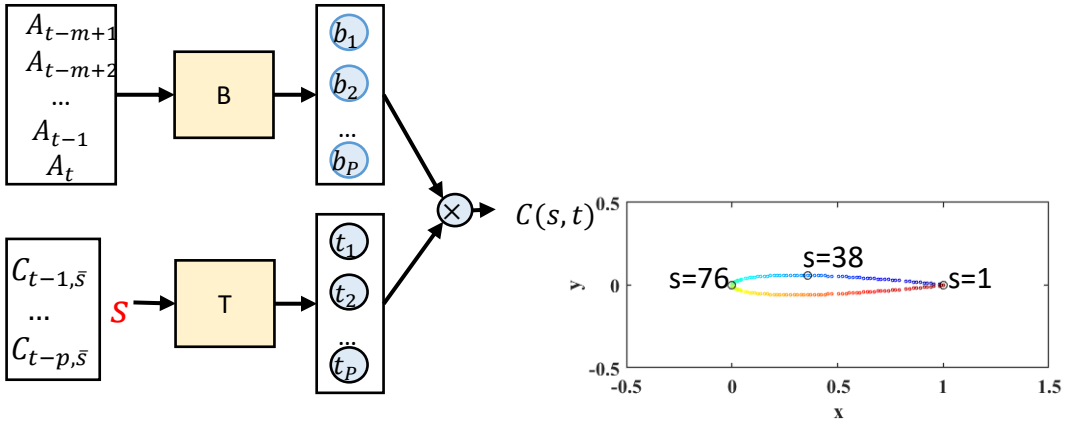


Average over space (surface of airfoil)

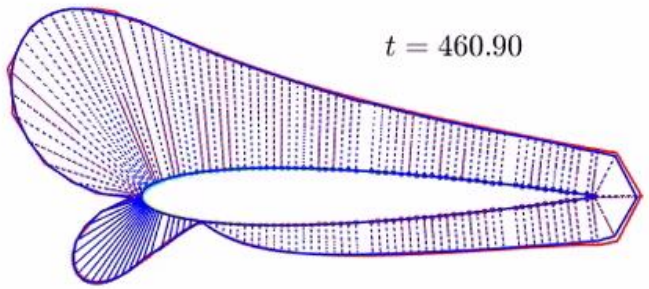
$$C(t,\bar{s}) = \text{mean}(C(t,\boldsymbol{s}))$$

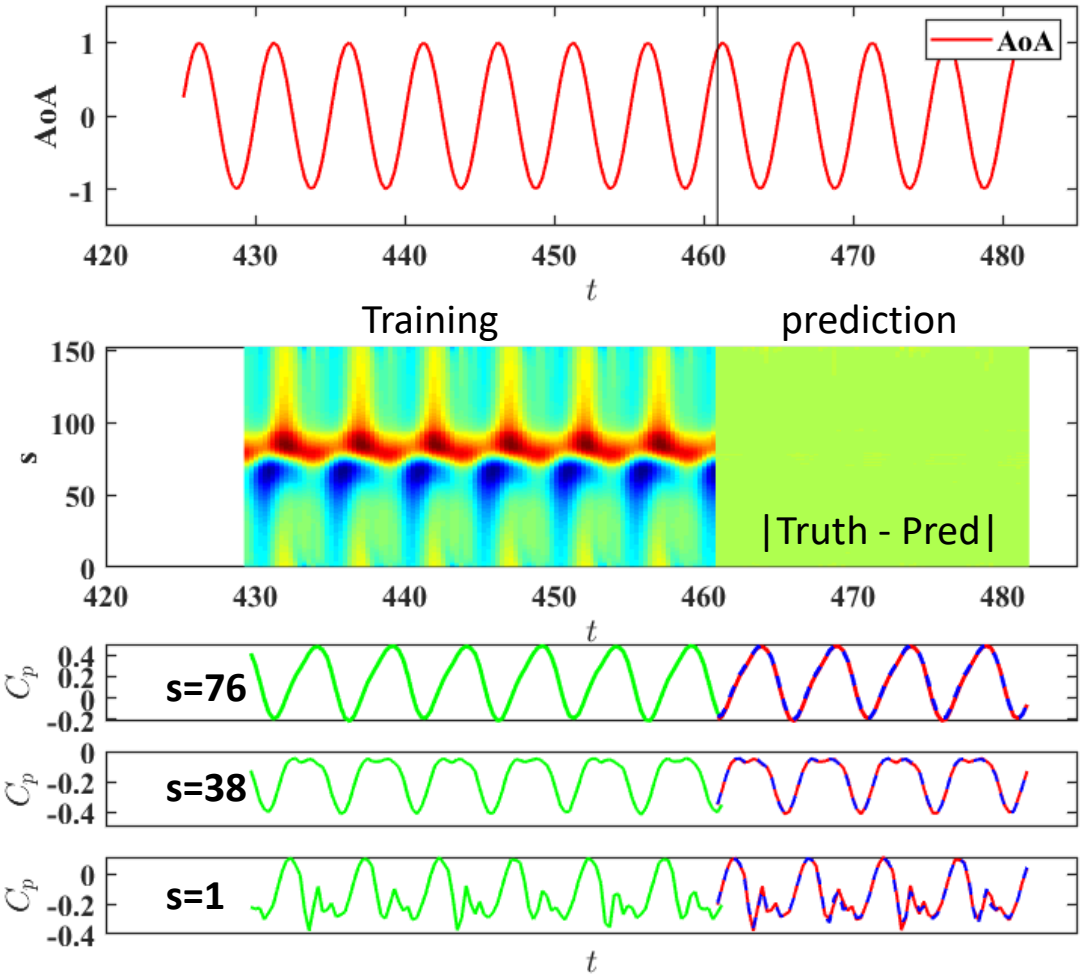
Coefficient of Pressure

- Result of predicting pressure coefficient $C(s,t)$
- k = 1 (Δt = **0.3**) / k = 5 (Δt = **1.5**) / k = 10 (Δt = **3**)



Red: truth
Blue: prediction
Solid: positive
Dash: negative

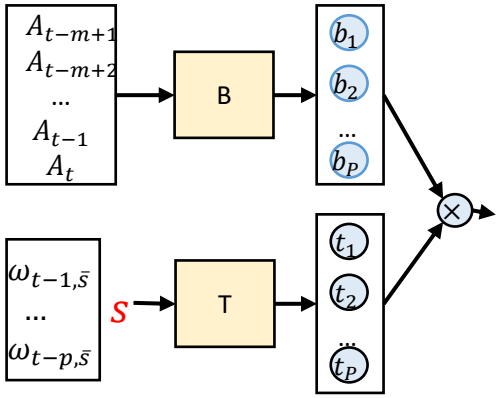


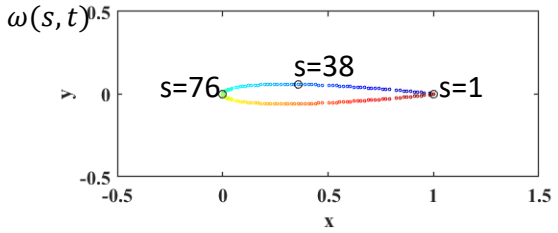


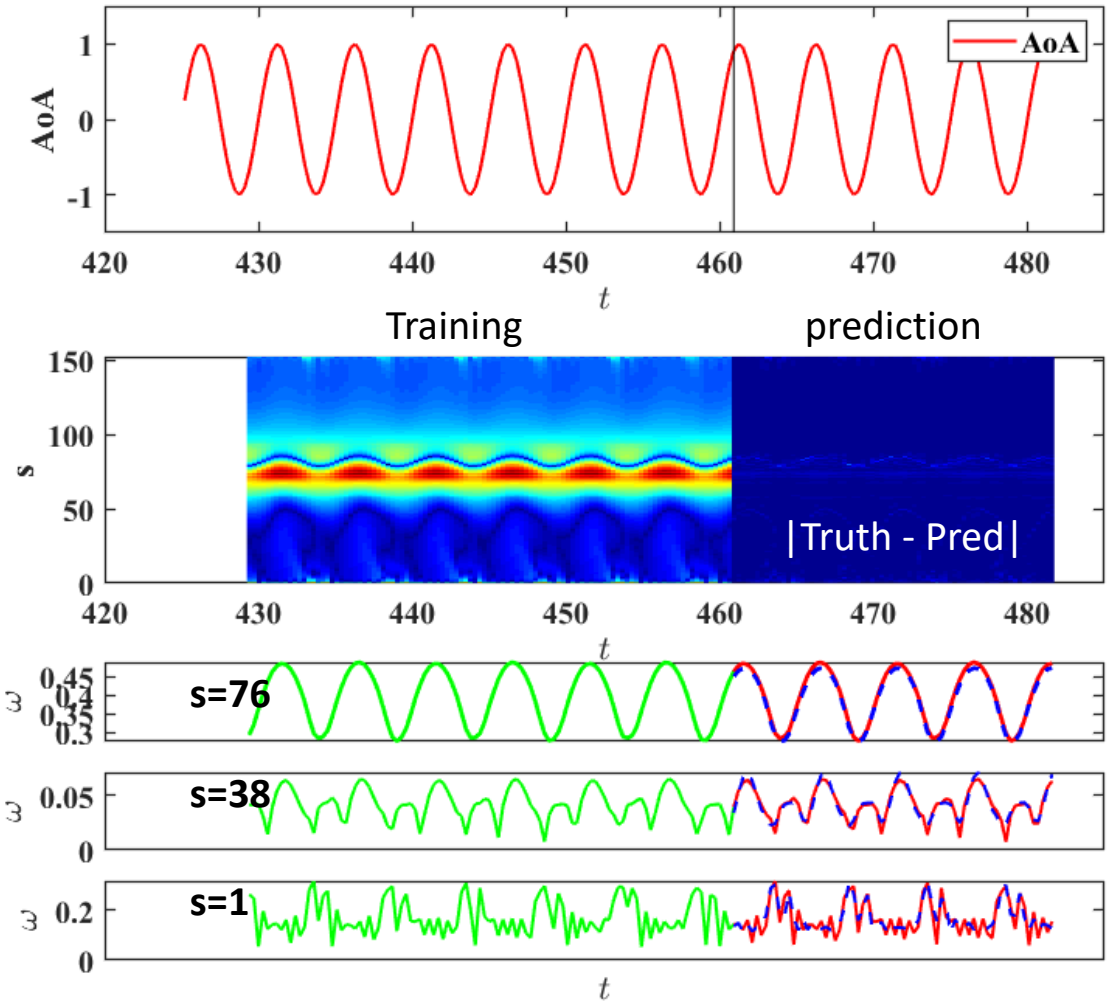
Vorticity on the Surface

Normalized $\omega^* = \sqrt{\frac{\omega}{Re}}$

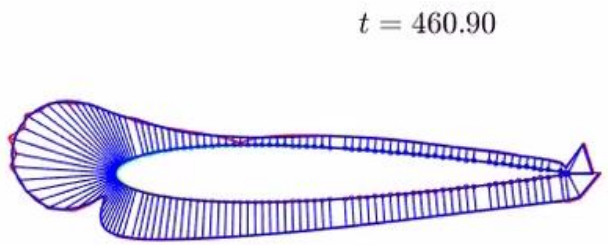
- Result of predicting vorticity $\omega^*(s,t)$
- k = 10 (Δt = 3)







Red: truth
Blue: prediction
Solid: positive
Dash: negative

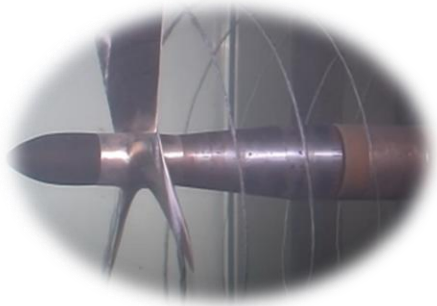


DeepONet for Bubble Dynamics

- The dynamics of bubble growth and collapse (cavitation) is important in many fields such as engineering or biomedicine
- **Rayleigh–Plesset equation** is an ordinary differential equation which governs the dynamics of a spherical bubble in an infinite body of incompressible fluid

$$\frac{P_\infty(t) - P_B(t)}{\rho_L} = R\frac{d^2R}{dt^2} + \frac{3}{2}\left(\frac{dR}{dt}\right)^2 + \frac{4\nu_L}{R}\frac{dR}{dt} + \frac{2\gamma}{\rho_L R}$$

where
$\mathbf{P_B(t)}$ is the pressure within the bubble, assumed to be uniform and $P_\infty(t)$ is the external pressure infinitely far from the bubble
$\rho_\mathbf{L}$ is the density of the surrounding liquid, assumed to be constant
$\mathbf{R(t)}$ is the radius of the bubble
$\nu_\mathbf{L}$ is the kinematic viscosity of the surrounding liquid, assumed to be constant
γ is the surface tension of the bubble-liquid interface



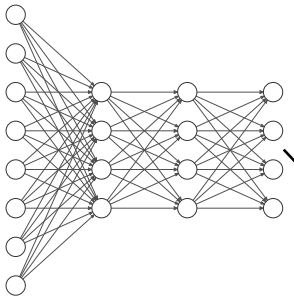
The Rayleigh-Plesset equation is derived from the Navier-Stokes equations under the assumption of spherical symmetry.

DeepONet for Bubble Dynamics

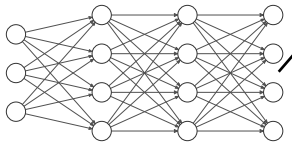
Architecture

Branch Network

$[\Delta p(t_1), \Delta p(t_2), \cdot \cdot \cdot \Delta p(t_m)]$



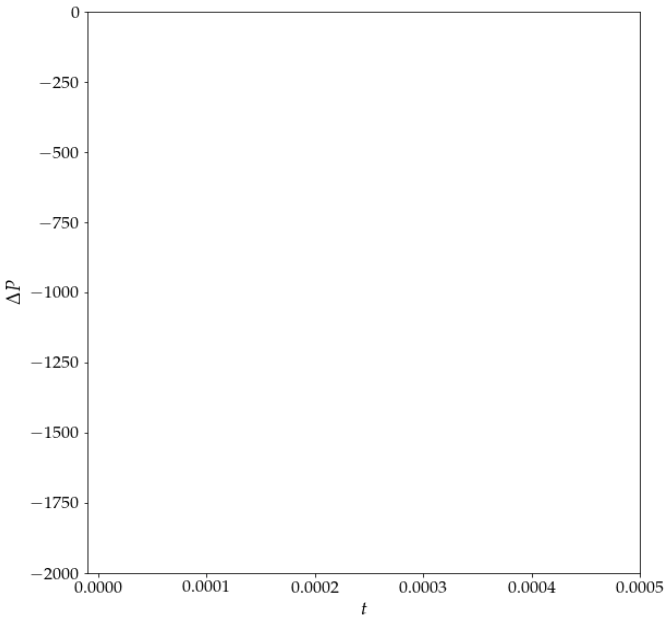
R_0, t



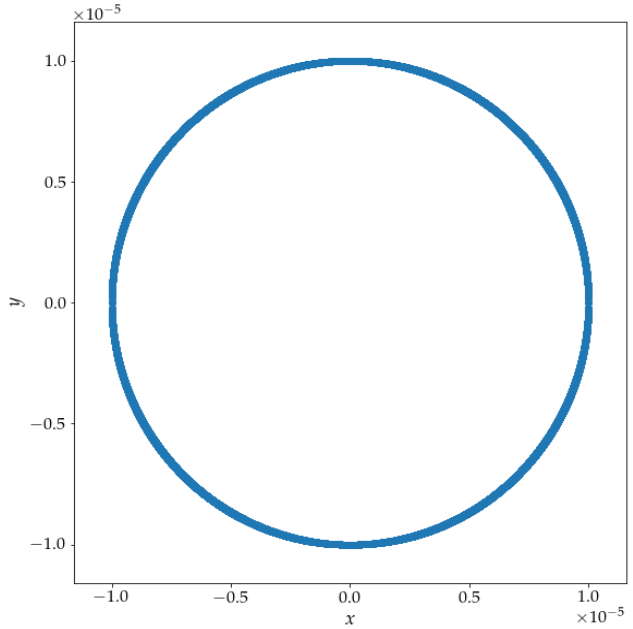
$\frac{R(t) - R_0}{R_0}$

Trunk Network

Random Pressure Field



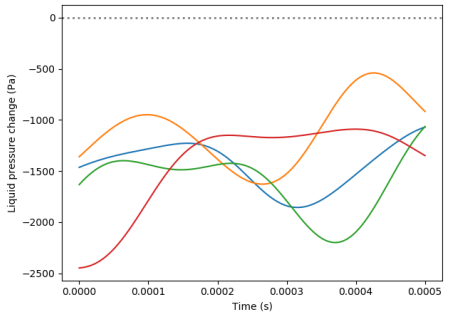
Fluctuation in Radius



DeepONet for Bubble Dynamics

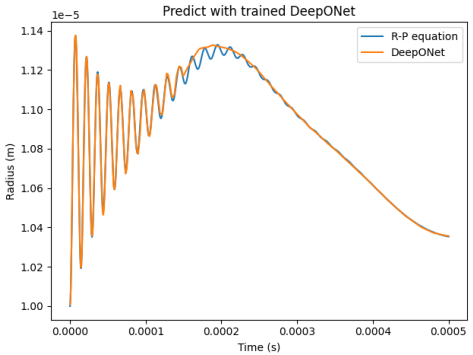
Liquid pressure (Input)

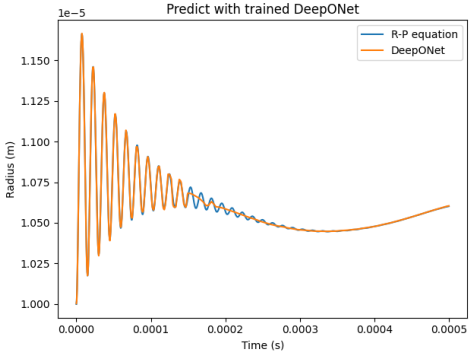
$$P_\infty(t) - P_B(t)$$

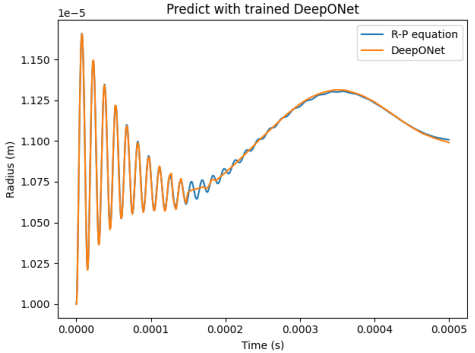


# liquid pressure 3000+1000	# liquid pressure sensor 50	# radius sensor 20	# Train data 60000	# Test data 20000
Type of DeepONet Unstacked	Branch depth 3	Branch width 300	Trunk depth 3	Trunk width 300
Activation Relu	Learning rate 0.0005	# Epoch. 150000		

Prediction VS Ground truth

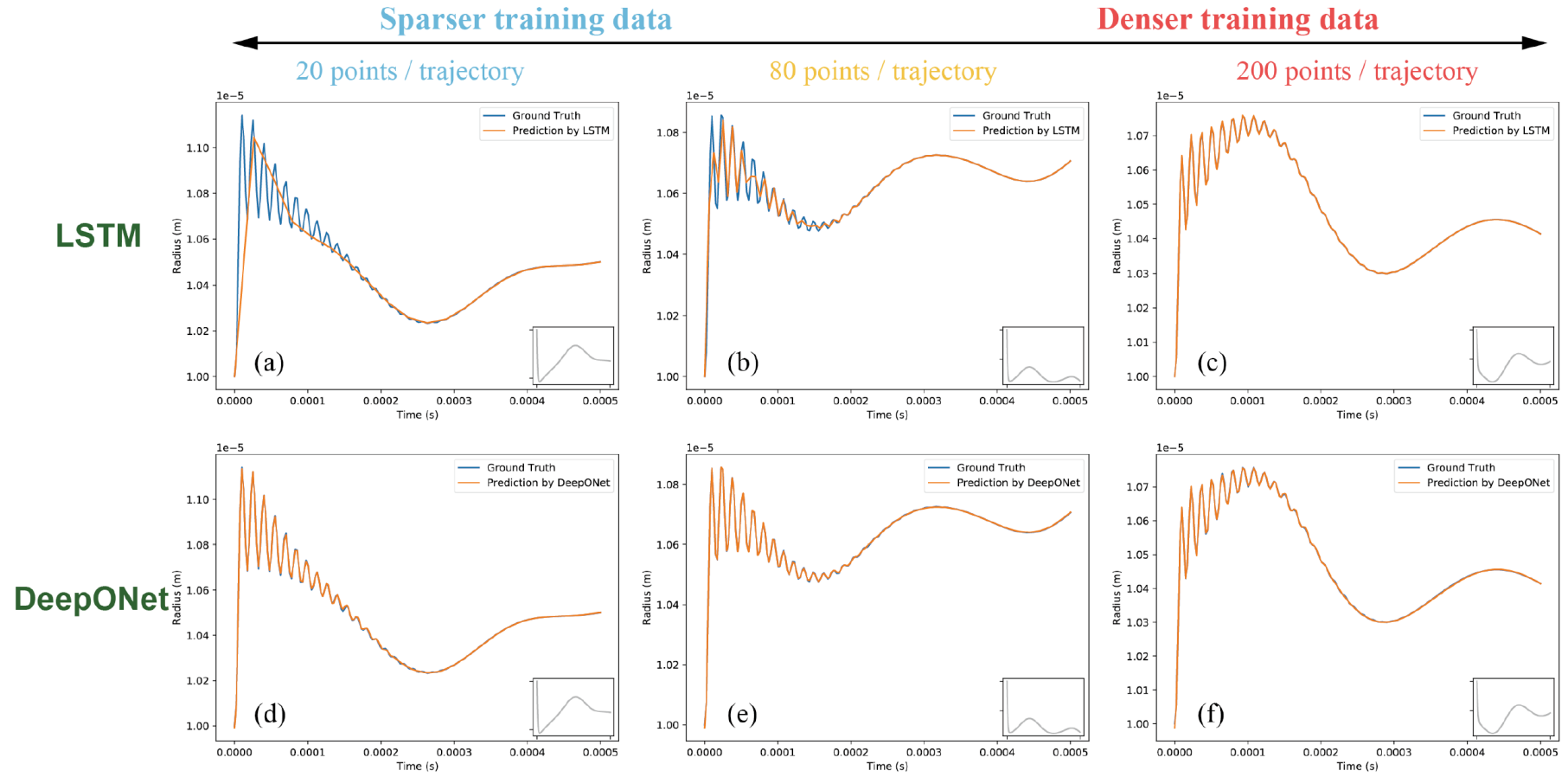


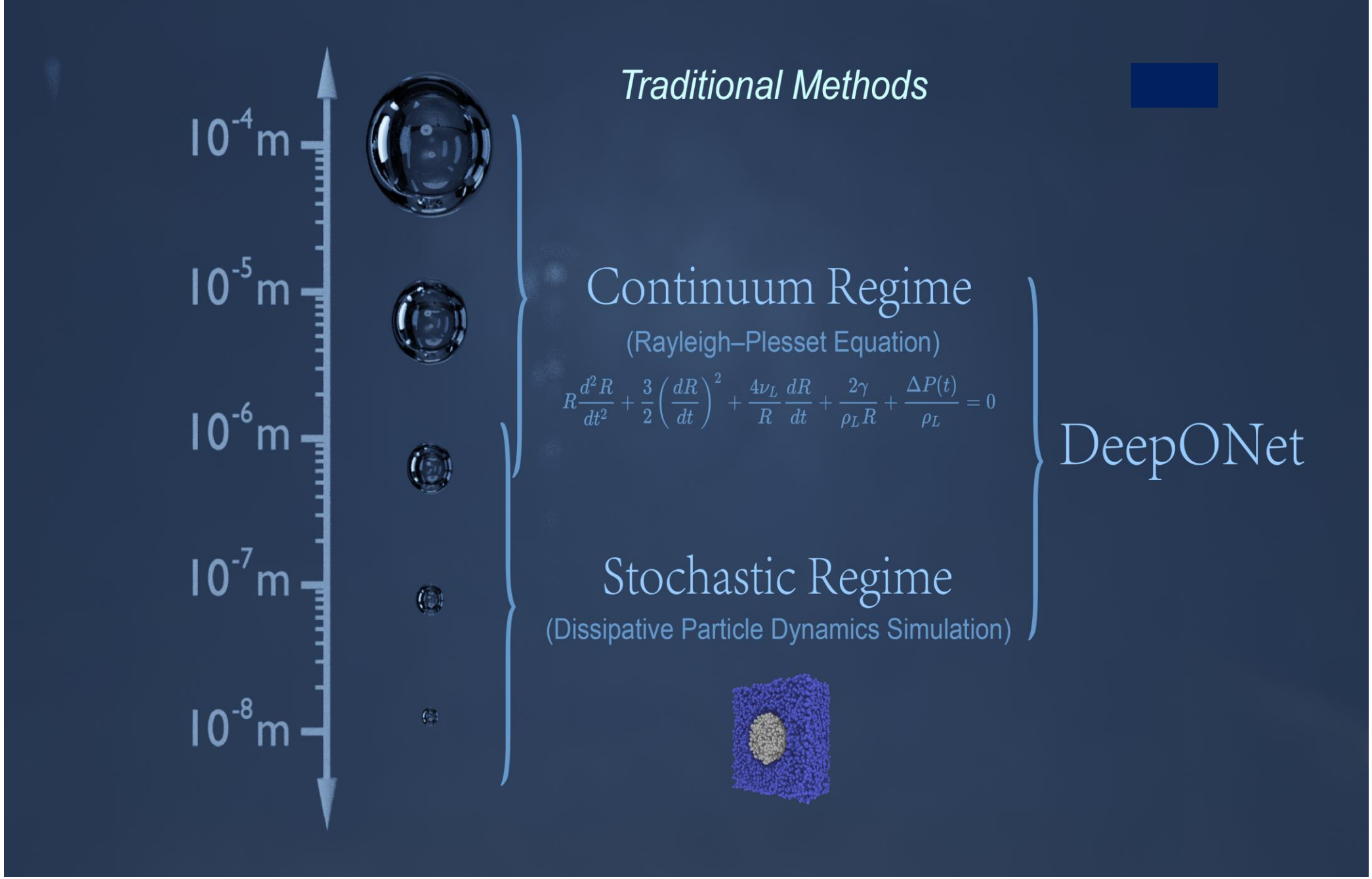




Comparison of DeepONet and LSTM

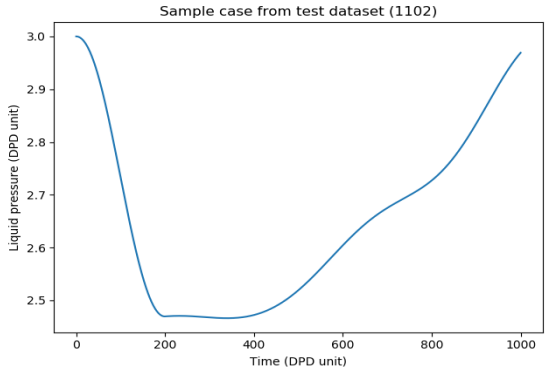
- 5000 training trajectories for both LSTM and DeepONet
- Test different number of sensors on each trajectory
- DeepONet has huge advantage when it's not possible to collect dense data or time scales are unknown

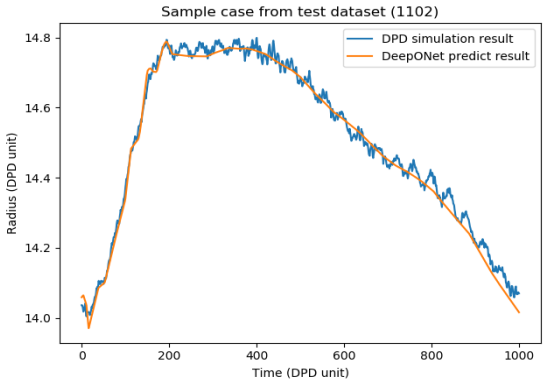


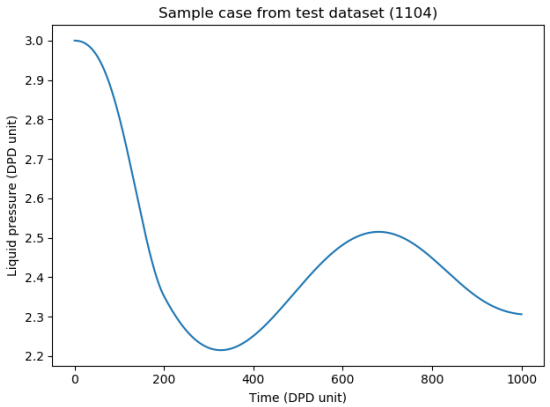

Traditional Methods
10^{-4}m
10^{-5}m
10^{-6}m
10^{-7}m
10^{-8}m
Continuum Regime
(Rayleigh–Plesset Equation)
$R\frac{d^2R}{dt^2} + \frac{3}{2}\left(\frac{dR}{dt}\right)^2 + \frac{4\nu_L}{R}\frac{dR}{dt} + \frac{2\gamma}{\rho_L R} + \frac{\Delta P(t)}{\rho_L} = 0$
Stochastic Regime
(Dissipative Particle Dynamics Simulation)
DeepONet

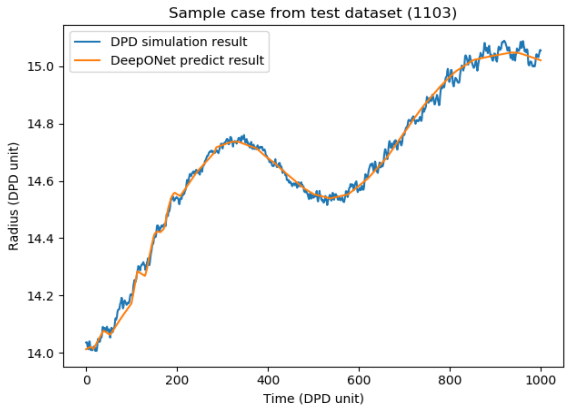
DeepONet for Bubble Dynamics

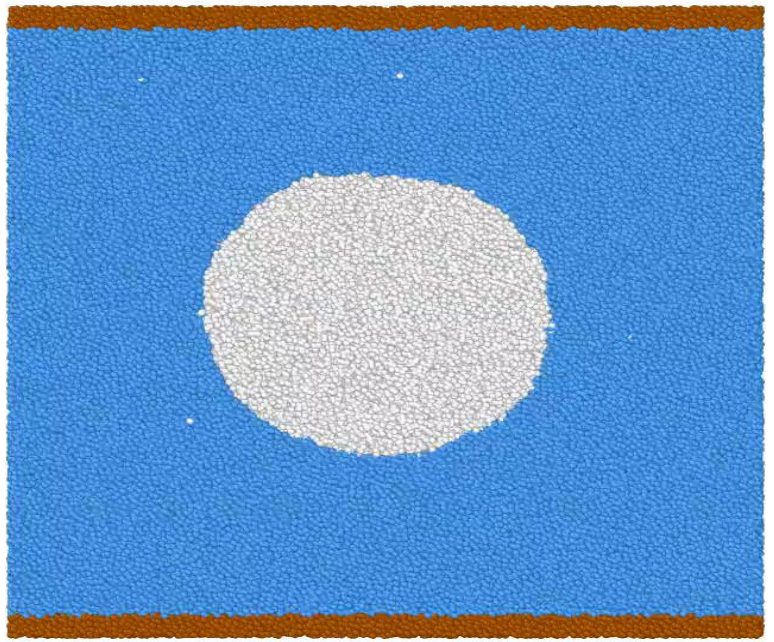
For nanobubbles, the thermal fluctuation cannot be ignored. Instead of solving R-P equation, the training data are generated by particle simulation (Dissipative Particle Dynamics)









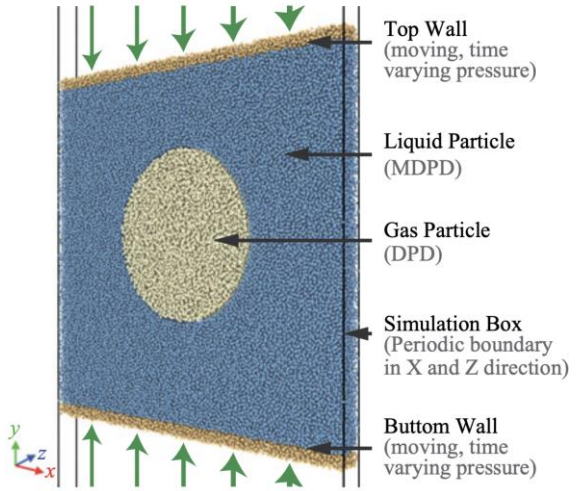


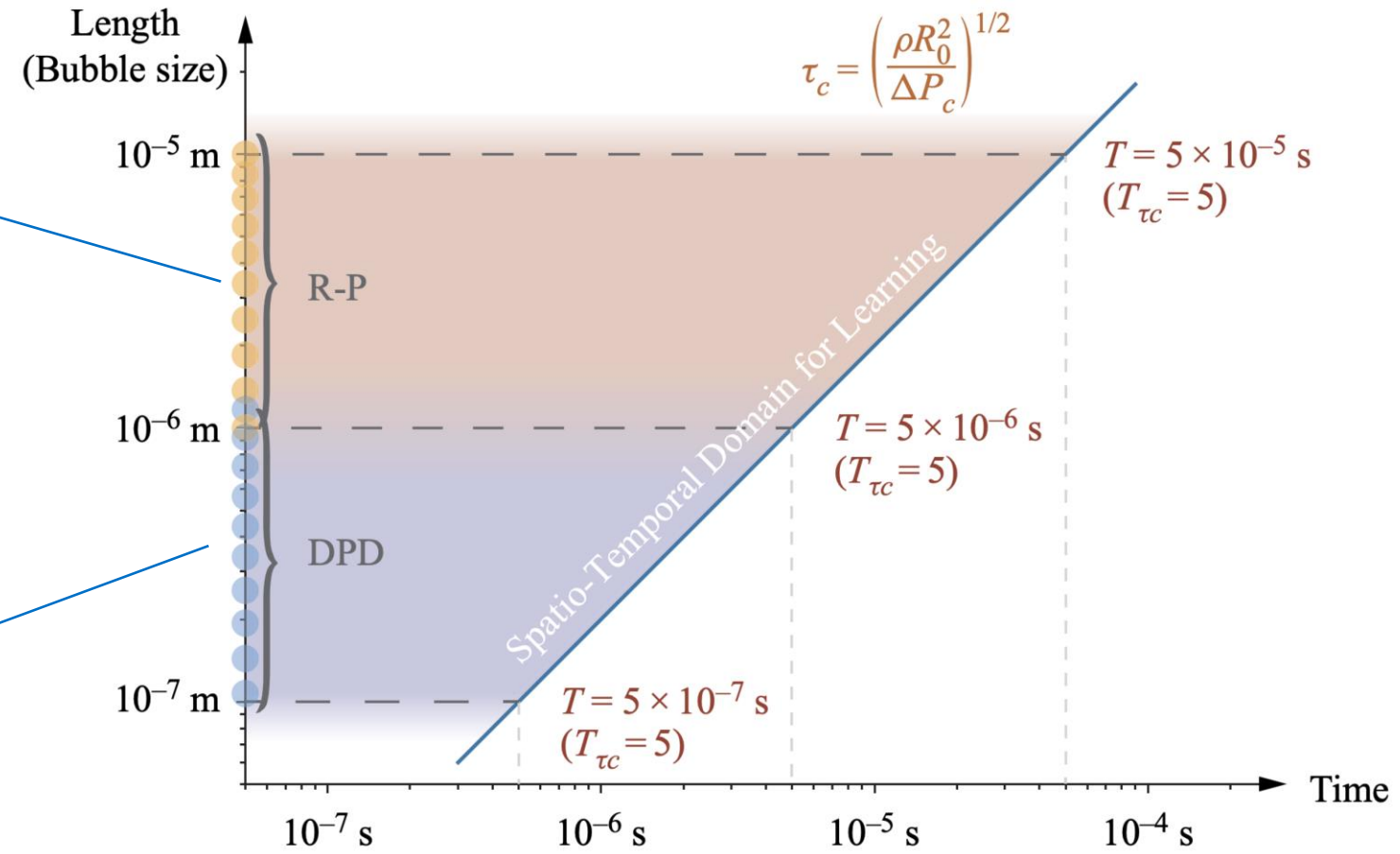
DeepONet as a Seamless Multiscale Operator

R-P Model

$$\frac{P_\infty(t) - P_B(t)}{\rho_L} = R\frac{d^2R}{dt^2} + \frac{3}{2}\left(\frac{dR}{dt}\right)^2 + \frac{4\nu_L}{R}\frac{dR}{dt} + \frac{2\gamma}{\rho_L R}$$

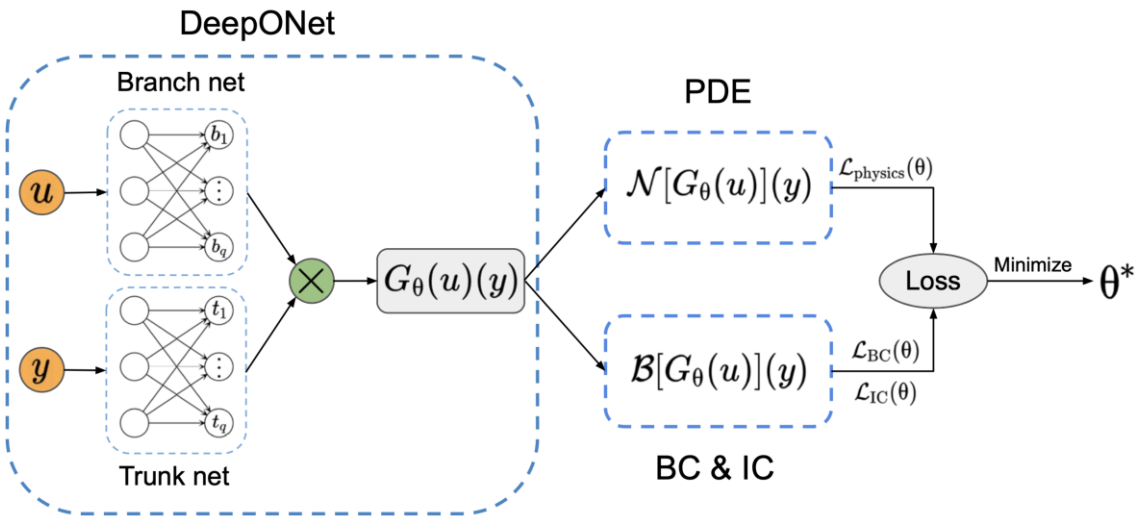
DPD Model





Lin C, Maxey M, Li Z, Karniadakis GE. A seamless multiscale operator neural network for inferring bubble dynamics. Journal of Fluid Mechanics. 2021 Dec;929.

Physics-Informed DeepONet



DeepONet output is defined as:

$$G_\theta(\mathbf{u})(\mathbf{y}) = \sum_{k=1}^{q} \underbrace{b_k\left(\mathbf{u}\left(\mathbf{x}_1\right), \mathbf{u}\left(\mathbf{x}_2\right), \ldots, \mathbf{u}\left(\mathbf{x}_m\right)\right)}_{\text{branch}} \underbrace{t_k(\mathbf{y})}_{\text{trunk}}$$

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{operator}}(\theta) + \mathcal{L}_{\text{physics}}(\theta)$$

where

$$\mathcal{L}_{\text{operator}}(\theta) = \frac{1}{NP} \sum_{i=1}^{N} \sum_{j=1}^{P} \left| G_\theta\left(\mathbf{u}^{(i)}\right)\left(\mathbf{y}_{u,j}^{(i)}\right) - G\left(\mathbf{u}^{(i)}\right)\left(\mathbf{y}_{u,j}^{(i)}\right) \right|^2$$

$$\mathcal{L}_{\text{physics}}(\theta) = \frac{1}{NQm} \sum_{i=1}^{N} \sum_{j=1}^{Q} \sum_{k=1}^{m} \left| \mathcal{N}\left(u^{(i)}\left(\mathbf{x}_k\right), G_\theta\left(\mathbf{u}^{(i)}\right)\left(\mathbf{y}_{r,j}^{(i)}\right)\right) \right|^2$$

The PDE loss is computed based on the derivatives of the DeepONet output with respect to trunk network input.

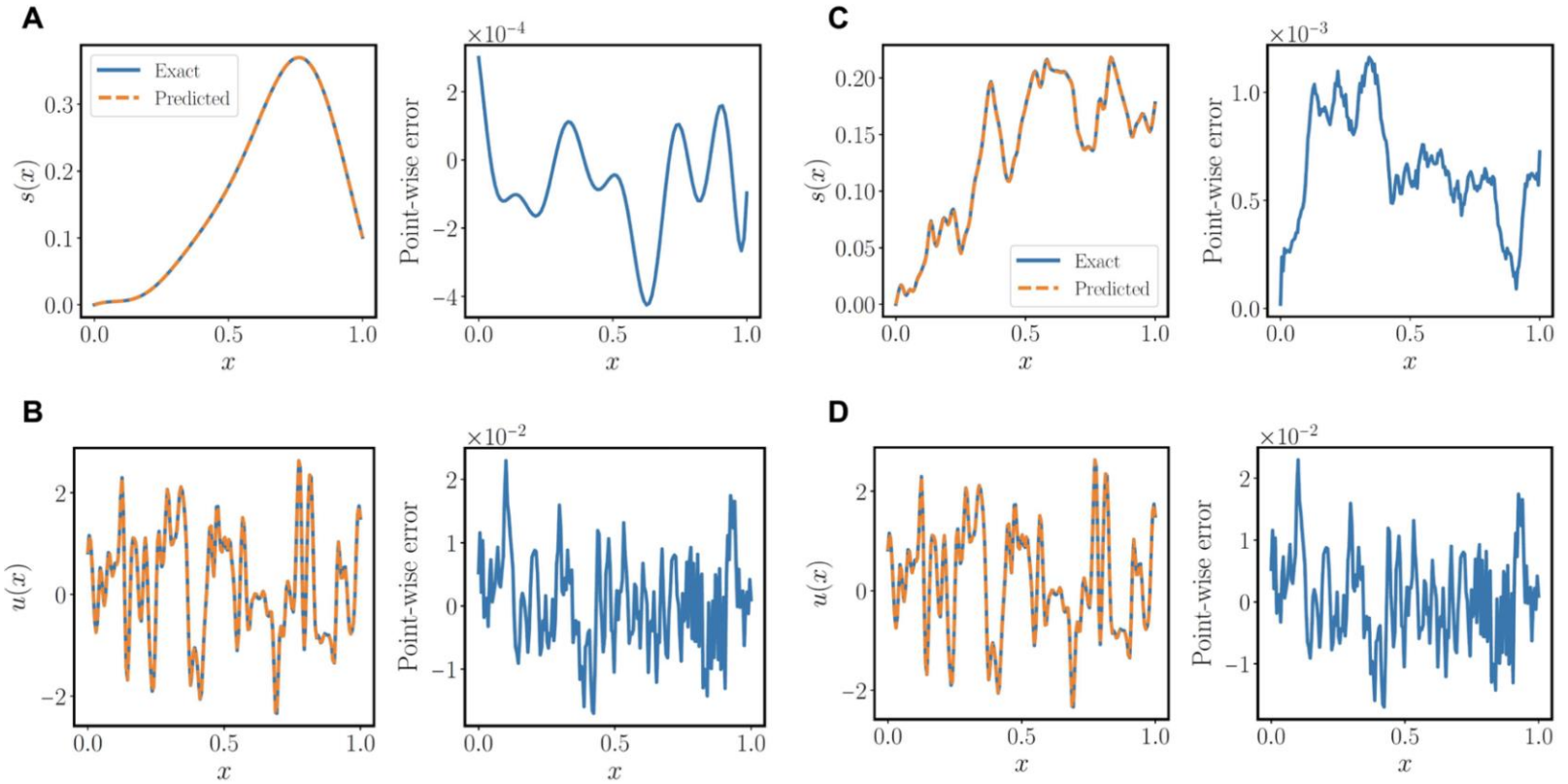
- Wang S, Wang H, Perdikaris P. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. Science advances. 2021 Sep 29;7(40):eabi8605.

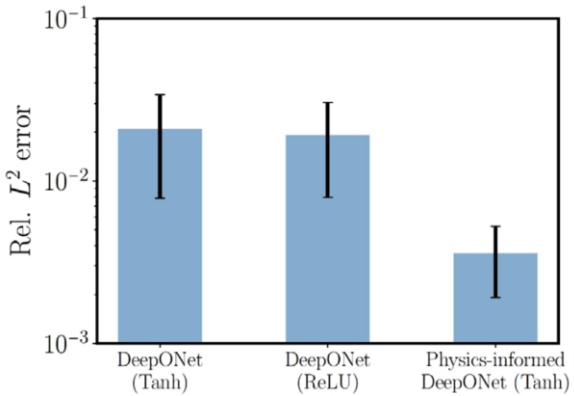
Physics-Informed DeepONet: Test Case

$$\frac{ds(x)}{dx} = u(x), x \in [0, 1]$$

$$s(0) = 0$$

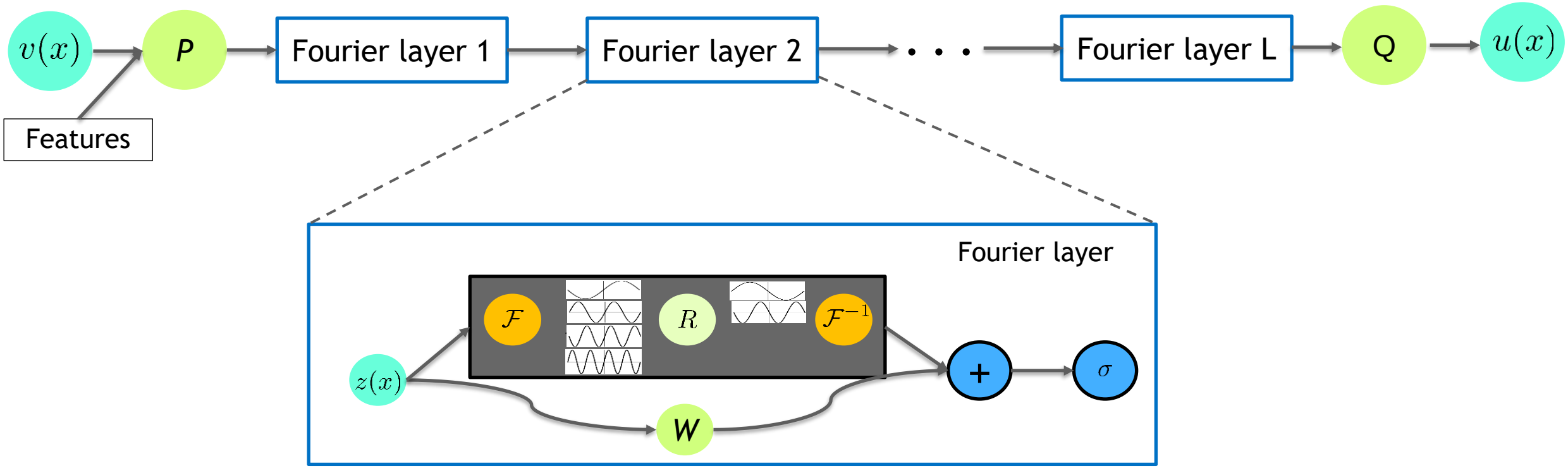
$u(x)$ is sampled from a Gaussian Random Field. DeepONet tries to approximate the anti-derivative operator.





- Wang S, Wang H, Perdikaris P. Learning the solution operator of parametric partial differential equations with physics-informed DeepONets. Science advances. 2021 Sep 29;7(40):eabi8605.

Fourier Neural Operator (FNO) for Parametric Partial Differential Equations



Idea: The main network parameters are defined and learned in the Fourier space rather than the physical space.

- Li Z, Kovachki N, Azizzadenesheli K, Liu B, Bhattacharya K, Stuart A, Anandkumar A. Fourier neural operator for parametric partial differential equations. arXiv preprint arXiv:2010.08895. 2020 Oct 18.

Structure of FNO

Step 1: Function value $v(x)$ is lifted to a higher dimensional representation $z_0(x)$ by

$$z_0(x) = P(v(x)) \in \mathbb{R}^{d_z}$$

Transformation $P : \mathbb{R} \to \mathbb{R}^{d_z}$ is a shallow fully-connected NN or simply a linear layer. d_z is like the channel size in CNN.

Step 2: L Fourier layers are applied iteratively to z_0. z_L is the output of the last Fourier layer, and the dimension of $z_L(x)$ is d_z.

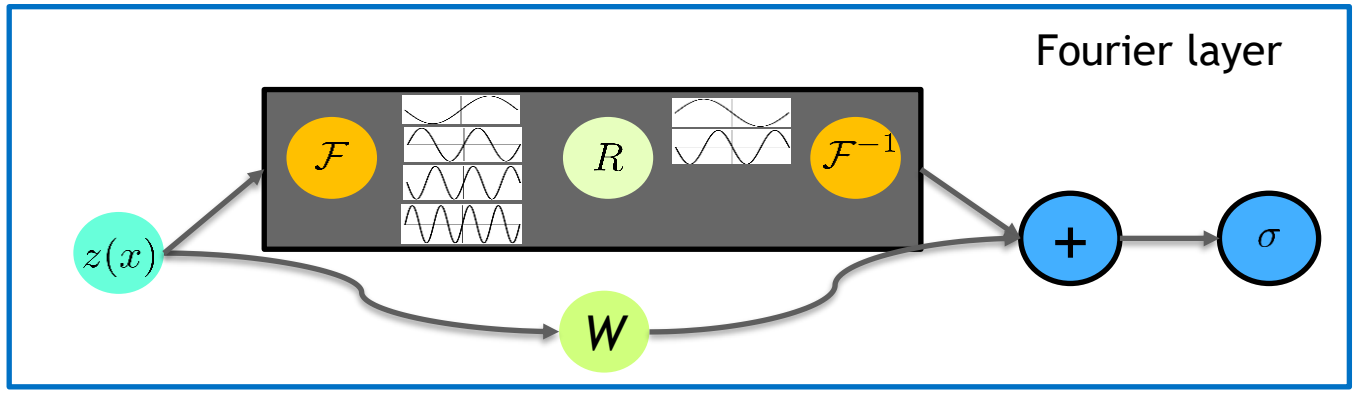
Step 3: Transformation $Q : \mathbb{R}^{d_z} \to \mathbb{R}$ is applied to project $z_L(x)$ to the output by

$$u(x) = Q(z_L(x))$$

Q is parameterized by a fully-connected NN.

- Li Z, Kovachki N, Azizzadenesheli K, Liu B, Bhattacharya K, Stuart A, Anandkumar A. Fourier neural operator for parametric partial differential equations. arXiv preprint arXiv:2010.08895. 2020 Oct 18.

Fourier Layer using Fast Fourier Transform (FFT)



For the output of the lth Fourier layer z_l with d_v channels:

Step 1: Compute the transform by FFT $\mathcal{F}$ and inverse FFT $\mathcal{F}^{-1}$:

$$\mathcal{F}^{-1}\left(R_l \cdot \mathcal{F}(z_l)\right)$$

$\mathcal{F}$ is applied to each channel of z_l separately; Truncate the higher modes of $\mathcal{F}(z_l)$, keeping only the first k Fourier modes in each channel. So $\mathcal{F}(z_l)$ has the shape $d_v \times k$.

Step 2: Apply a different (complex-number) weight matrix of shape $d_v \times d_v$ for each mode index of $\mathcal{F}(z_l)$ Have k trainable matrices, which form a weight tensor $R_l \in \mathbb{C}^{d_v \times d_v \times k}$. $R_l \cdot \mathcal{F}(z_l)$ has the same shape of $d_v \times k$ as $\mathcal{F}(z_l)$.

Step 3: Inverse FFT Need to append zeros to $R_l \cdot \mathcal{F}(z_l)$ to fill in the truncated modes.
Moreover, in each Fourier layer, a residual connection with a weight matrix $W_l \in \mathbb{R}^{d_v \times d_v}$. The output of the $(l+1)$th Fourier layer z_{l+1} is $\boldsymbol{z}_{l+1} = \sigma\left(\mathcal{F}^{-1}\left(R_l \cdot \mathcal{F}\left(\boldsymbol{z}_l\right)\right) + W_l \cdot \boldsymbol{z}_l + \boldsymbol{b}_l\right)$

Differences between DeepONet and FNO

Properties	DeepONet	FNO
Input domain D & Output domain D'	Arbitrary	Cuboid, $D = D'$
Discretization of output function u	No	Yes
Mesh	Arbitrary	Grid
Prediction location	Arbitrary	Grid points
Full field observation data	No	Yes

Implementation of Fourier Neural Operator (FNO)

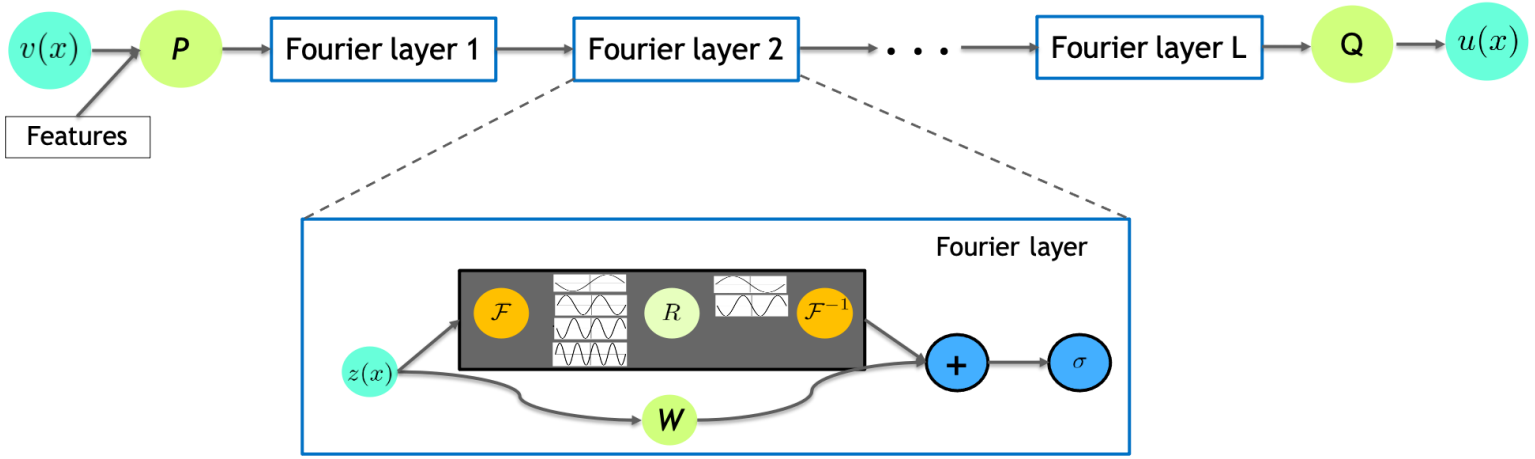
1. Consider the Burger's equation given by

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = \nu \frac{\partial^2 u}{\partial x^2}, \qquad x \in (0,1), \quad t \in (0,1]$$

   and $\nu = 0.1$ with initial condition (IC) $u(x,0) = u_0(x)$ and periodic boundary condition.

2. We aim to learn the operator mapping the initial condition to the solution at time one $u_0 \to u(\cdot, 1)$

3. The data is generated for different initial condition $u_0 \sim \mathcal{N}(0, 625(-\Delta + 25\boldsymbol{I})^{-2})$



Code Implementation for FNO:

1. Import Modules

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.parameter import Parameter
import operator
from functools import reduce
from functools import partial
from timeit import default_timer
from op_utilities import *
from torch.optim import Adam

torch.manual_seed(0)
np.random.seed(0)
```

2. Construction of Fourier Layers

```
class SpectralConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, modes1):
        super(SpectralConv1d, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes1 = modes1  #Number of Fourier modes to multiply, at most floor(N/2) + 1

        self.scale = (1 / (in_channels*out_channels))
        self.weights1 = nn.Parameter(self.scale * torch.rand(in_channels, out_channels, \
                                                             self.modes1, dtype=torch.cfloat))
    # Complex multiplication
    def compl_mul1d(self, input, weights):
        # (batch, in_channel, x ), (in_channel, out_channel, x) -> (batch, out_channel, x)
        return torch.einsum("bix,iox->box", input, weights)

    def forward(self, x):
        batchsize = x.shape[0]
        #Compute Fourier coeffcients up to factor of e^(- something constant)
        x_ft = torch.fft.rfft(x)

        # Multiply relevant Fourier modes
        out_ft = torch.zeros(batchsize, self.out_channels, x.size(-1)//2 + 1,  \
                             device=x.device, dtype=torch.cfloat)
        out_ft[:, :, :self.modes1] = self.compl_mul1d(x_ft[:, :, :self.modes1], self.weights1)

        #Return to physical space
        x = torch.fft.irfft(out_ft, n=x.size(-1))
        return x
```

Code Implementation for FNO

3. FNO NN models

```
class FNO1d(nn.Module):
    def __init__(self, modes, width):
        super(FNO1d, self).__init__()
        self.modes1 = modes
        self.width = width
        self.padding = 2 # pad the domain if input is non-periodic
        self.fc0 = nn.Linear(2, self.width) # input channel is 2: (a(x), x)

        self.conv0 = SpectralConv1d(self.width, self.width, self.modes1)
        self.conv1 = SpectralConv1d(self.width, self.width, self.modes1)
        self.conv2 = SpectralConv1d(self.width, self.width, self.modes1)
        self.conv3 = SpectralConv1d(self.width, self.width, self.modes1)
        self.w0 = nn.Conv1d(self.width, self.width, 1)
        self.w1 = nn.Conv1d(self.width, self.width, 1)
        self.w2 = nn.Conv1d(self.width, self.width, 1)
        self.w3 = nn.Conv1d(self.width, self.width, 1)

        self.fc1 = nn.Linear(self.width, 128)
        self.fc2 = nn.Linear(128, 1)
```

4. Forward Pass

```
    def forward(self, x):
        grid = self.get_grid(x.shape, x.device)
        x = torch.cat((x, grid), dim=-1)
        x = self.fc0(x)
        x = x.permute(0, 2, 1)
        x1 = self.conv0(x)
        x2 = self.w0(x)
        x = x1 + x2
        x = F.gelu(x)

        x1 = self.conv1(x)
        x2 = self.w1(x)
        x = x1 + x2
        x = F.gelu(x)

        x1 = self.conv2(x)
        x2 = self.w2(x)
        x = x1 + x2
        x = F.gelu(x)

        x1 = self.conv3(x)
        x2 = self.w3(x)
        x = x1 + x2

        x = x.permute(0, 2, 1)
        x = self.fc1(x)
        x = F.gelu(x)
        x = self.fc2(x)
        return x
```

Input data $(u(x,0),x)$, where x is to be gridded

```
def get_grid(self, shape, device):
    batchsize, size_x = shape[0], shape[1]
    gridx = torch.tensor(np.linspace(0, 1, size_x), dtype=torch.float)
    gridx = gridx.reshape(1, size_x, 1).repeat([batchsize, 1, 1])
    return gridx.to(device)
```

Code Implementation for FNO

5. Hyperparameters

```
ntrain = 1000
ntest = 100

sub = 2**3 #subsampling rate
h = 2**13 // sub #total grid size divided by the subsampling rate
s = h

batch_size = 20
learning_rate = 0.001

epochs = 500
step_size = 50
gamma = 0.5

modes = 16
width = 64
```

6. Reading sampling of the data: 10 Realizations

```
dataloader = MatReader('burgers_data_R10.mat')
x_data = dataloader.read_field('a')[:,::sub]
y_data = dataloader.read_field('u')[:,::sub]

x_train = x_data[:ntrain,:]
y_train = y_data[:ntrain,:]
x_test = x_data[-ntest:,:]
y_test = y_data[-ntest:,:]

x_train = x_train.reshape(ntrain,s,1)
x_test = x_test.reshape(ntest,s,1)

train_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(x_train, y_train),\
                                           batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(x_test, y_test),\
                                          batch_size=batch_size, shuffle=False)

# model
model = FNO1d(modes, width)
```

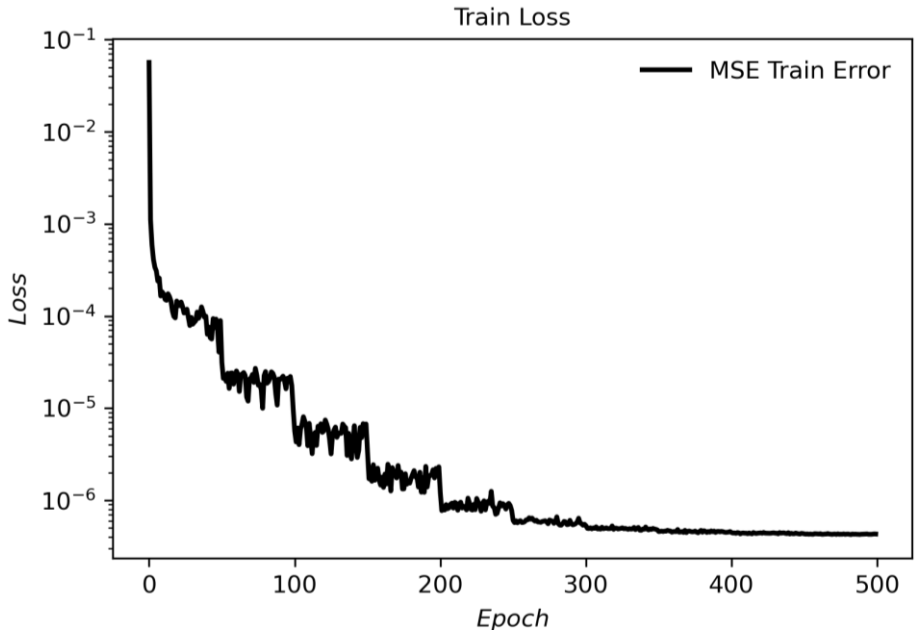
7. Training and Error Computation

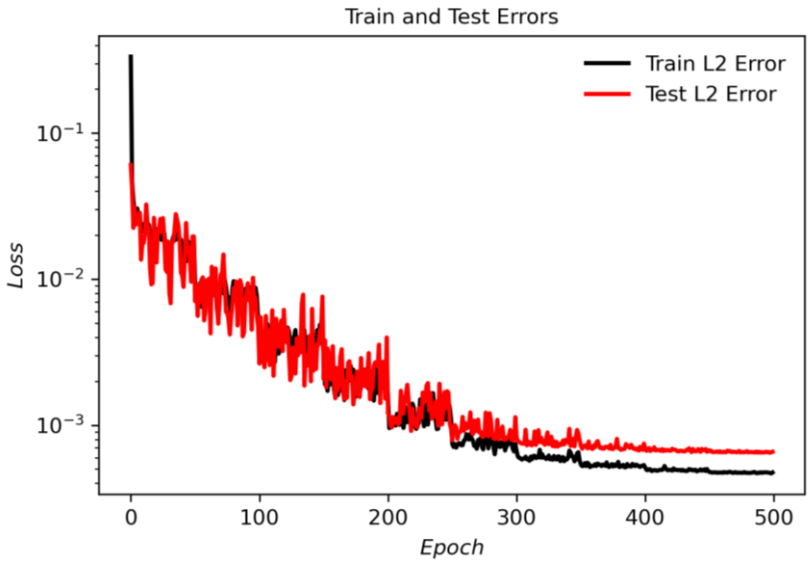
```
optimizer = Adam(model.parameters(), lr=learning_rate, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)
myloss = LpLoss(size_average=False)
for ep in range(epochs):
    model.train()
    t1 = default_timer()
    train_mse = 0
    train_l2 = 0
    for x, y in train_loader:
        x, y = x, y
        optimizer.zero_grad()
        out = model(x)

        mse = F.mse_loss(out.view(batch_size, -1), y.view(batch_size, -1), reduction='mean
        l2 = myloss(out.view(batch_size, -1), y.view(batch_size, -1))
        l2.backward() # use the l2 relative loss

        optimizer.step()
        train_mse += mse.item()
        train_l2 += l2.item()

    scheduler.step()
    model.eval()
    test_l2 = 0.0
    with torch.no_grad():
        for x, y in test_loader:
            x, y = x, y

            out = model(x)
            test_l2 += myloss(out.view(batch_size, -1), y.view(batch_size, -1)).item()

    train_mse /= len(train_loader)
    train_l2 /= ntrain
    test_l2 /= ntest
```

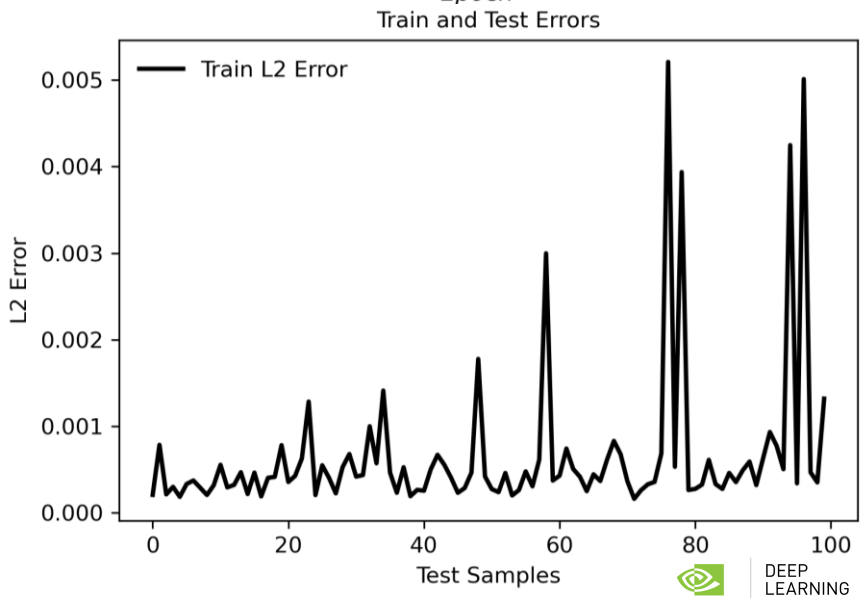
Code Implementation for FNO

8. Prediction and Results

```
pred = torch.zeros(y_test.shape)
index = 0
test_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(x_test, y_test),\
                                          batch_size=1, shuffle=False)
with torch.no_grad():
    for x, y in test_loader:
        test_l2 = 0
        x, y = x, y #x.cuda(), y.cuda()

        out = model(x).view(-1)
        pred[index] = out

        test_l2 += myloss(out.view(1, -1), y.view(1, -1)).item()
        print(index, test_l2)
        index = index + 1
```

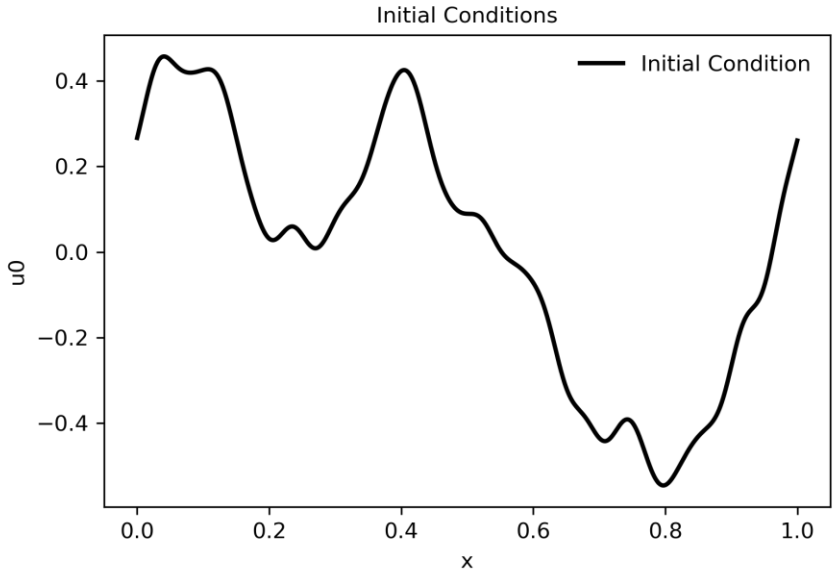


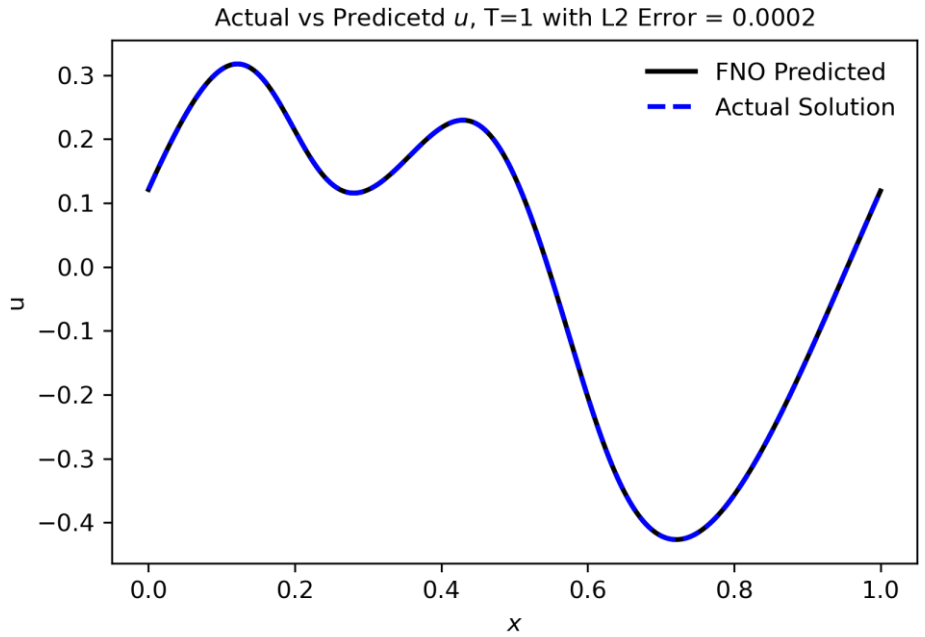




Code Implementation for FNO

8. Prediction and Results





Summary

- Neural networks are universal approximators of functions, functionals and nonlinear operators
- DeepOnet was designed based on theorem of Chen & Chen and it was extended to deep NNs
- DeepOnet has some resemblance to biological neurons
- DeepOnet converges exponentially fast with the training data but in practice it saturates for big data
- In addition to learning mathematical operators, DeepOnet can learn multiscale operators
- Physics-informed DeepOnet can enhance accuracy and generalization but hybrid training is more robust
- Fourier neural operator (FNO) is a subcase of the DeepOnet framework
- Neural operators can be used as building blocks for multiphysics problems and digital twins
- The accuracy of neural operators can be enhanced with expansion features added to the branch and trunk nets
- Rigorous theory for neural operators has already been developed

References

- Cai S, Wang Z, Lu L, Zaki TA, Karniadakis GE. DeepM&Mnet: Inferring the electroconvection multiphysics fields based on operator approximation by neural networks. Journal of Computational Physics. 2021 Jul 1;436:110296.
- Goswami S, Yin M, Yu Y, Karniadakis GE. A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials. Computer Methods in Applied Mechanics and Engineering. 2022 Mar 1;391:114587.
- Lanthaler S, Mishra S, Karniadakis GE. Error estimates for DeepOnets: A deep learning framework in infinite dimensions. arXiv preprint arXiv:2102.09618. 2021 Feb 18.
- Li Z, Kovachki N, Azizzadenesheli K, Liu B, Bhattacharya K, Stuart A, Anandkumar A. Fourier neural operator for parametric partial differential equations. arXiv preprint arXiv:2010.08895. 2020 Oct 18.
- Lu L, Jin P, Pang G, Zhang Z, Karniadakis GE. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. Nature Machine Intelligence. 2021 Mar;3(3):218-29.
- Lu L, Meng X, Cai S, Mao Z, Goswami S, Zhang Z, Karniadakis GE. A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data. arXiv preprint arXiv:2111.05512. 2021 Nov 10.
- Mao Z, Lu L, Marxen O, Zaki TA, Karniadakis GE. DeepM&Mnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators. Journal of Computational Physics. 2021 Dec 15;447:110698.

References

- del Aguila Ferrandis J, Chryssostomidis C, Triantafyllou M, Karniadakis, G.E. Learning functionals via LSTM neural networks for predicting vessel dynamics in extreme sea states, Proc. R. Soc. A.47720190897, 2021.

- Lin C, Li Z, Lu L, Cai S, Maxey M, Karniadakis GE. Operator learning for predicting multiscale bubble growth dynamics. The Journal of Chemical Physics. 2021 Mar 14;154(10):104118.

- Lin C, Maxey M, Li Z, Karniadakis GE. A seamless multiscale operator neural network for inferring bubble dynamics. Journal of Fluid Mechanics. 2021 Dec; 929.

- Marcati C, Schwab C. Exponential convergence of Deep Operator Networks for elliptic partial differential equations. arXiv preprint arXiv:2112.08125. 2021 Dec 15.

- Chen TP and Chen H, Approximations of continuous functionals by neural networks with application to dynamic systems, IEEE Transactions on Neural Networks, 910-918, 4(6), 1993.

- Chen TP and Chen H. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. IEEE Transactions on Neural Networks, 6(4):911–917, 1995.

- Wang S, Wang H, Perdikaris P. Learning the solution operator of parametric partial differential equations with physics-informed DeepOnets. arXiv preprint arXiv:2103.10974. 2021 Mar 19.

Deep Learning for Science and Engineering Teaching Kit

Thank You