Deep Operator Learning

Welcome to our Presentation

← Slides can be found here

Deep Operator Learning

Jan Sprengel, Robin Janssen

Seminar Scientific Machine Learning

DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators

Lu Lu[1], Pengzhan Jin[2], and George Em Karniadakis[1]

[1]Division of Applied Mathematics, Brown University, Providence, RI 02912, USA
[2]LSEC, ICMSEC, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China

Abstract

While it is widely known that neural networks are universal approximators of continuous functions, a less known and perhaps more powerful result is that a neural network with a single hidden layer can approximate accurately any nonlinear continuous operator [5]. This universal approximation theorem is suggestive of the potential application of neural networks in learning nonlinear operators from data. However, the theorem guarantees only a small approximation error for a sufficient large network, and does not consider the important optimization and generalization errors. To realize this theorem in practice, we propose deep operator networks (DeepONets) to learn operators accurately and efficiently from a relatively small dataset. A DeepONet consists of two sub-networks, one for encoding the input function at a fixed number of sensors x_i, $i = 1, \dots, m$ (branch net), and another for encoding the locations for the output functions (trunk net). We perform systematic simulations for identifying two types of operators, i.e., dynamic systems and partial differential equations, and demonstrate that DeepONet significantly reduces the generalization error compared to the fully-connected networks. We also derive theoretically the dependence of the approximation error in terms of the number of sensors (where the input function is defined) as well as the input function type, and we verify the theorem with computational results. More importantly, we observe high-order error convergence in our computational tests, namely polynomial rates (from half order to fourth order) and even exponential convergence with respect to the training dataset size.

1 Introduction

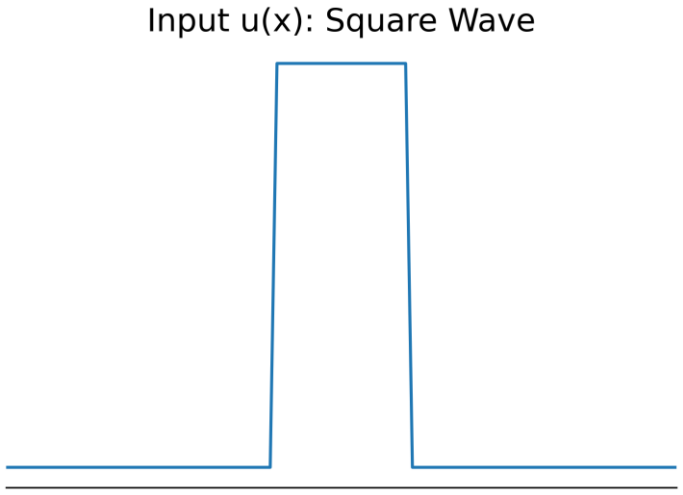
The universal approximation theorem states that neural networks can be used to approximate any continuous function to arbitrary accuracy if no constraint is placed on the width and depth of the hidden layers [7, 11]. However, another approximation result, which is yet more surprising and has not been appreciated so far, states that a neural network with a single hidden layer can approximate accurately any nonlinear continuous *functional* (a mapping from a space of functions into the real numbers) [3, 18, 25] or (nonlinear) operator (a mapping from a space of functions into another space of functions) [5, 4].

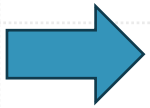
Before reviewing the approximation theorem for operators, we introduce some notation, which will be used through this paper. Let G be an operator taking an input function u, and then $G(u)$ is the corresponding output function. For any point y in the domain of $G(u)$, the output $G(u)(y)$ is a real number. Hence, the network takes inputs composed of two parts: u and y, and outputs $G(u)(y)$ (Fig. 1A). Although our goal is to learn operators, which take a function as the input, we have to represent the input functions discretely, so that network approximations can be applied. A straightforward and simple way, in practice, is to employ the

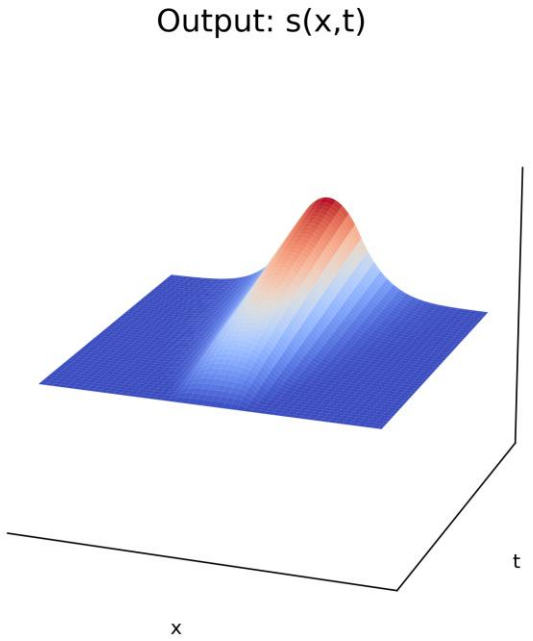
Motivation

1D Heat diffusion with a source term:

$$\partial_t s(x,t) = D * \partial_x^2 s(x,t) + u(x)$$







Contents

Operators

- Function F $\mathrm{F}: X \to Y, x \mapsto F(x)$
- Operator G $G: X \to Y, u \mapsto G(u)$

  X and Y are function spaces, e.g. $X = C(I), X = C^k(I)$ or $X = L^p(I)$
- Functional S $S: X \to Y, S \mapsto S(u)$

  But: $Y = \mathbb{R}$, not a function space

Operators

Operator Properties

- Linearity + Boundedness $G(u_1 + u_2) = G(u_1) + G(u_2)$ and $G(\alpha u_1) = \alpha G(u_1), \alpha \in \mathbb{R}$
- Continuity $u_n \to \mathrm{u} \Rightarrow G(u_n) \to G(u)$

$$\frac{d}{dx} \qquad \partial_x \qquad \nabla \qquad \Delta \qquad \square$$

$$f * g \qquad \hat{f} \qquad Pf \qquad \int dx$$

Universal Approximation Theorem

UAT – Functions

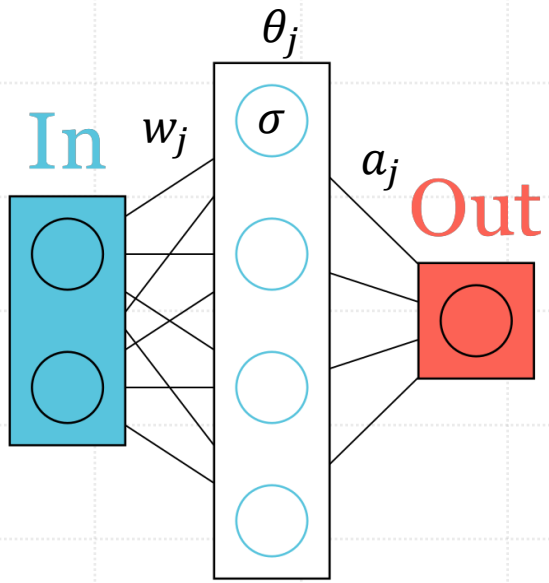
G. Cybenko, *Approximation by Superpositions of Sigmoidal Function*, 1989

Let σ be any continuous sigmoidal function. Then finite sums of the form

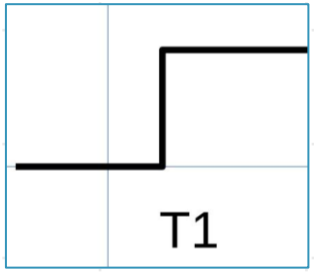
$$S(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + \theta_j)$$

are **dense** in $C(I_n)$.

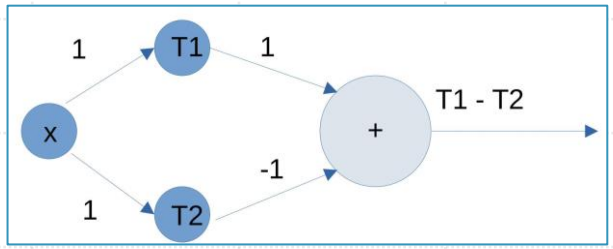
In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum $S(x)$ for which $\|S(x) - f(x)\| < \epsilon$ for all $x \in I_n$.



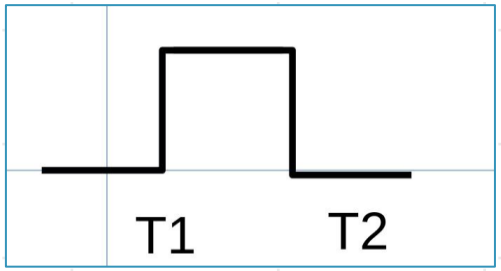
Universal Approximation Theorem



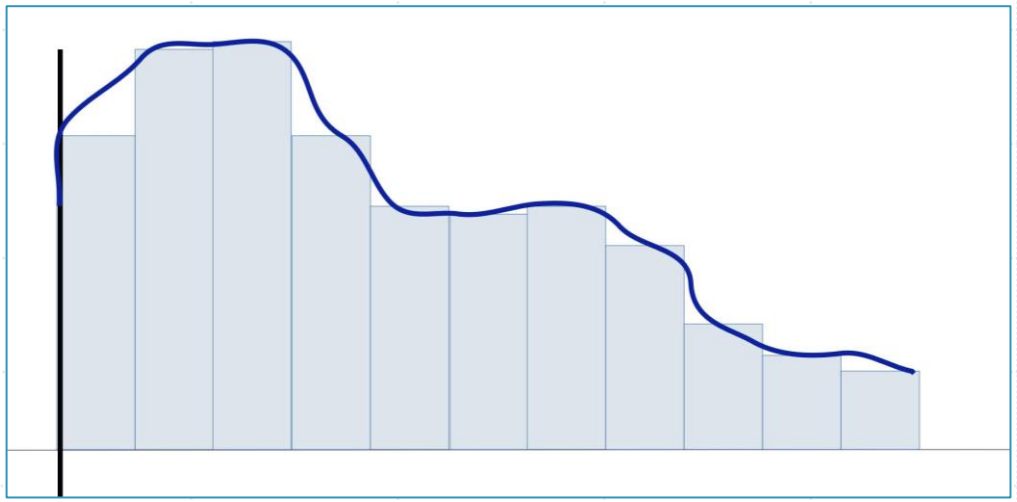
$\sigma(x) = \Theta(w^T x)$



"Superposition"



Approximation



From [2]

Universal Approximation Theorem

UAT – Operators

Chen & Chen, Universal Approximation to Nonlinear Operators..., 1995

Let $\mathrm{G}: \mathrm{V} \to \mathrm{W}, \mathrm{u} \mapsto \mathrm{G}(\mathrm{u})$ be a non-linear continuous Operator.
$G(u)(y)$ denotes the value of $G(u)$ at point y.

There is a function $f_{c,\xi,\theta,\zeta}(\boldsymbol{x}, y)$ such that for every $\epsilon > 0$

$$\left|G(u)(y) - f_{c,\xi,\theta,\zeta}(\boldsymbol{x}, y)\right| < \epsilon$$

for all $u \in V, y \in Y$ $(G(u): Y \to Z)$.

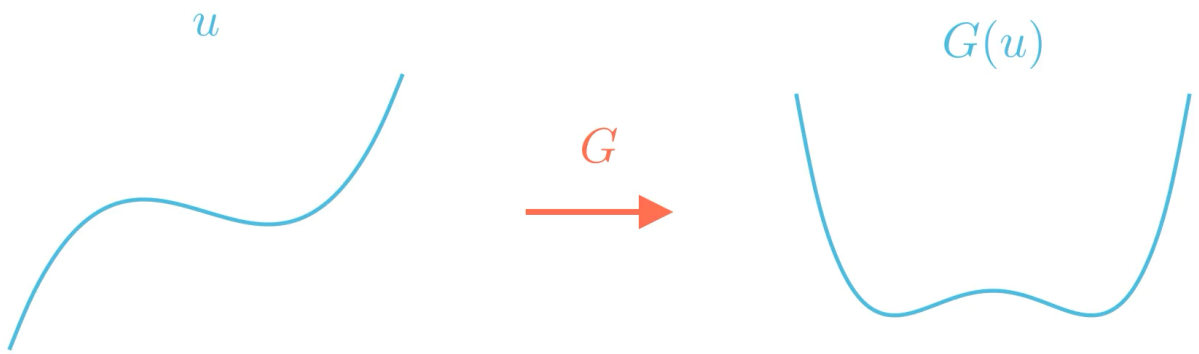
$$f_{c,\xi,\theta,\zeta}(\boldsymbol{x}, y) = \sum_{k=1}^{p} \sigma(w_k \cdot y + \zeta_k) \sum_{i=1}^{n} c_i^k \cdot \sigma\left(\sum_{j=1}^{m} \xi_{ij}^k \cdot u(x_j) + \theta_i^k\right)$$

Inputs and Outputs

Problem: Can not put a function into a NN…

Solution: **"Sensors"**
$\rightarrow$ Use function values $u(x_i)$ at locations x_i

Inputs and Outputs

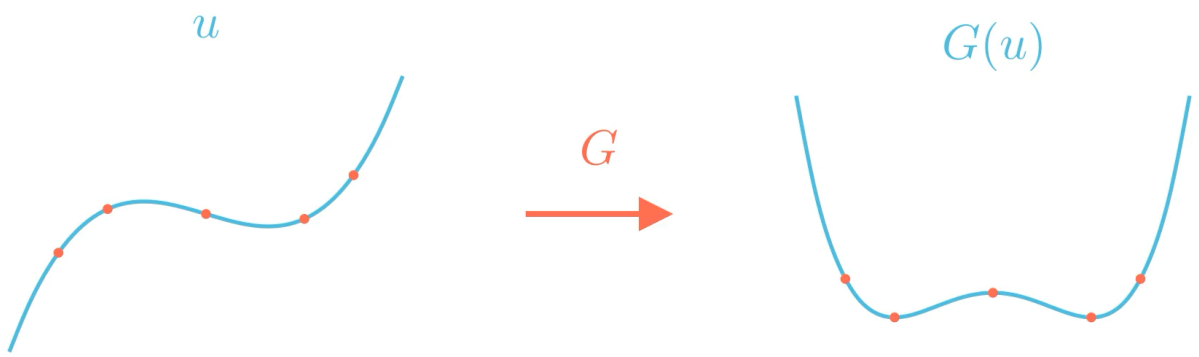


Problem: Can not put a function into an NN…

Solution: **"Sensors"**
→ Use function values $u(x_i)$ at locations x_i

More flexibility: Output $G(u)(y)$ at chosen point y

Inputs and Outputs

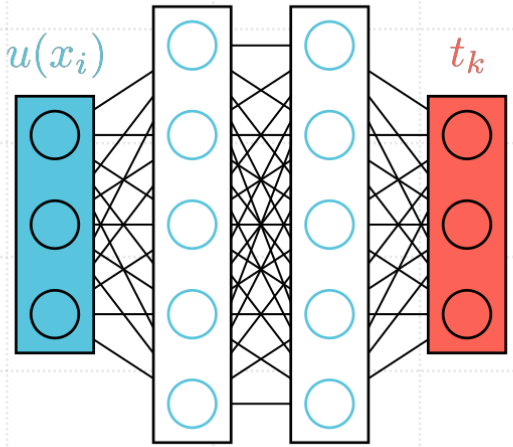


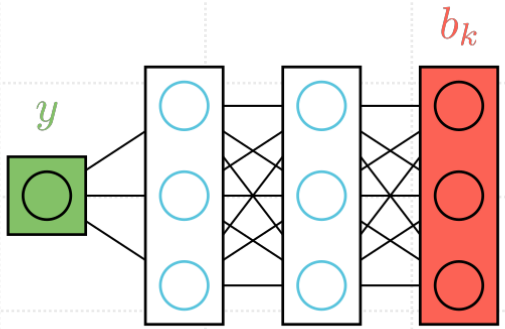
Problem: Can not put a function into an NN…

Solution: **"Sensors"**
→ Use function values $u(x_i)$ at locations x_i

More flexibility: "Arbitrary" output location y

Network Architecture





Idea: Use inductive bias

→ 2 Separate Neural Networks

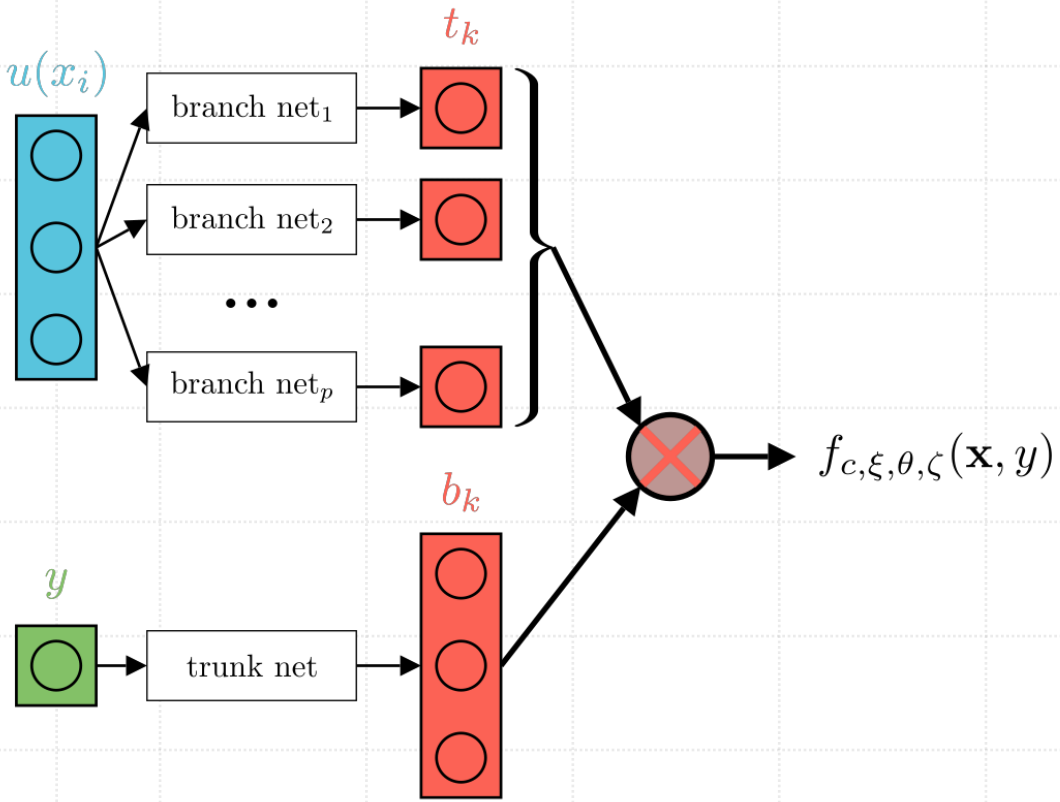
1. NN: **“Branch Net“**

$u(x)$ → encode input space

2. NN: **“Trunk Net“**

y → encode domain of output function

Connecting Architecture and Theorem



Theory – Approximation Theorem:

1. Branch Net:

$$t_k = \sum_{i=1}^{n} c_i^k \cdot \sigma\left(\sum_{j=1}^{m} \xi_{ij}^k \cdot u(x_j) + \theta_i^k\right)$$

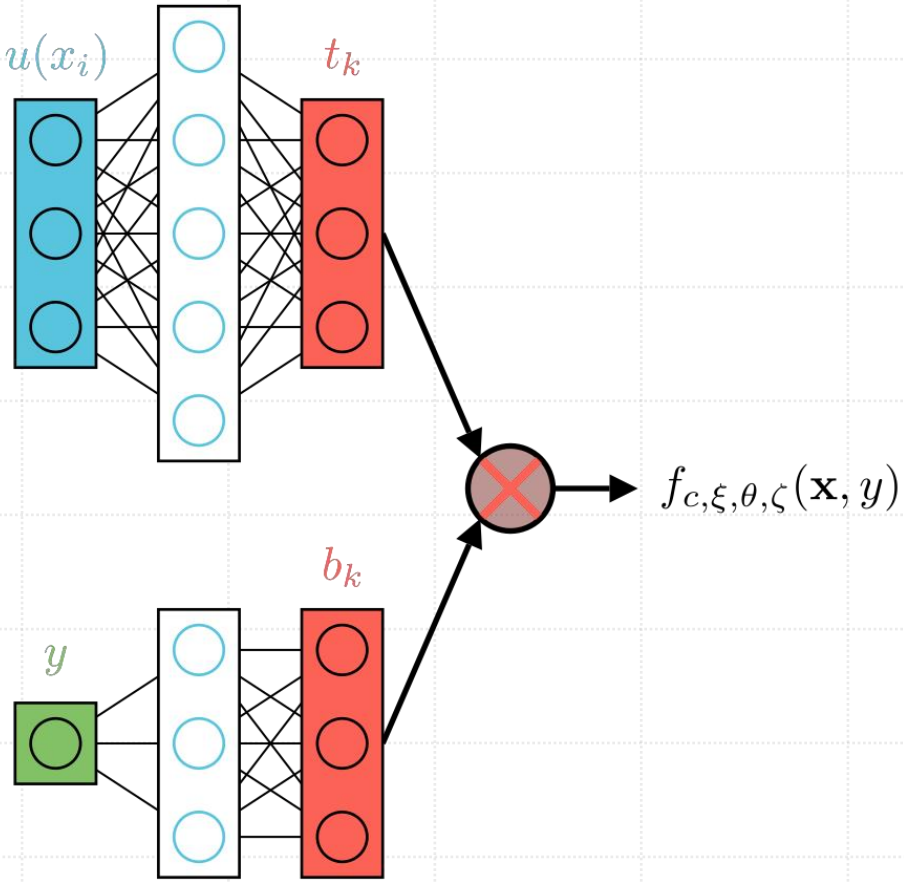
2. Trunk Net:

$$b_k = \sigma(w_k \cdot y + \zeta_k)$$

3. Output: $G(u)$

$$f_{c,\xi,\theta,\zeta}(x, y) = \sum_{k=1}^{p} t_k b_k$$

Unstacked Network



Problem: latent dimension **p >> 1**

In Practice → Merge Branch NNs

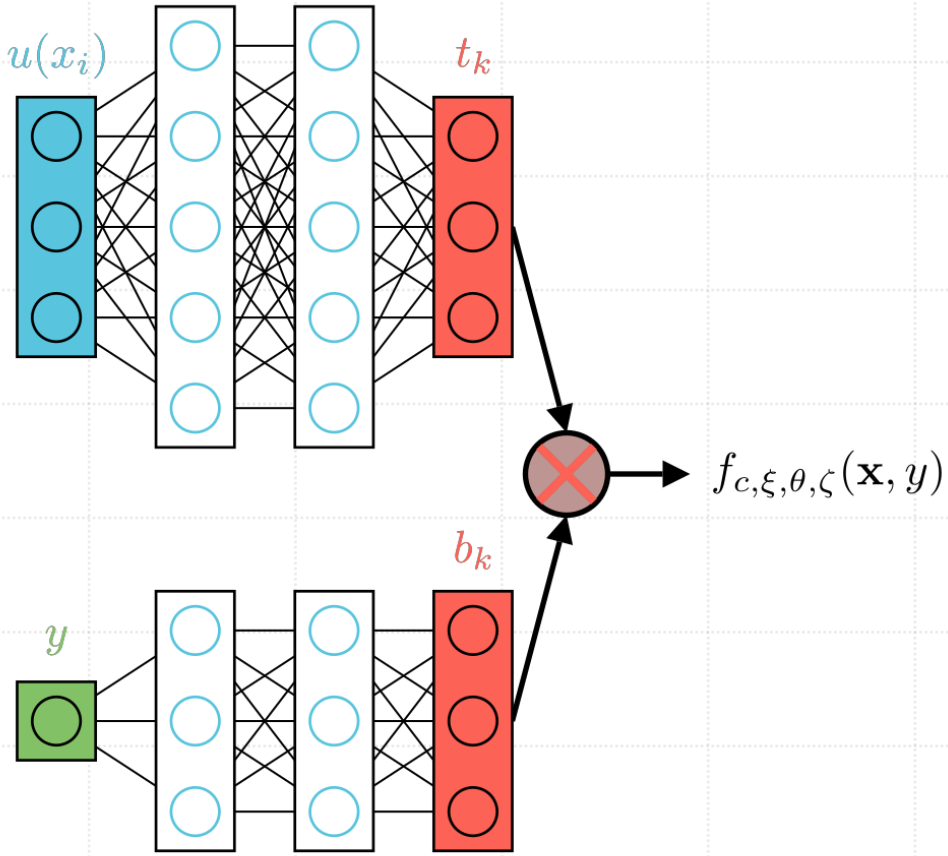
- Higher efficiency
- Generalization

- Specific Dimensions determined per use-case

  Depth: ~ 2-4 Layers,

  Width: ~ $10^2 - 10^3$ Neurons

Network Architecture – Conclusion



In Practice:

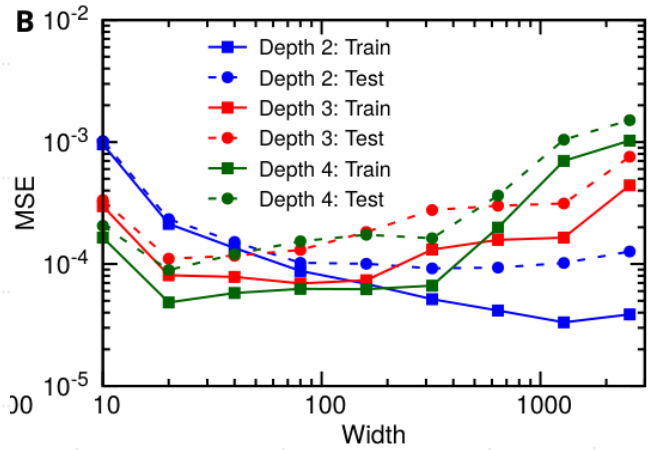
- Unstacked Architecture
- Deep Network / MLP

Hyperparameters:

- For Simple ODE:

  Depth: 2-3 Layers,

  Width: ~ 100 Neurons



Training

- Training Dataset:

$$[\boldsymbol{u}, \boldsymbol{y}, G(\boldsymbol{u})(\boldsymbol{y})] = \left[\left[\begin{matrix}\vdots \\ \boldsymbol{u}^i(\boldsymbol{x}_1), \boldsymbol{u}^i(\boldsymbol{x}_2), \dots, \boldsymbol{u}^i(\boldsymbol{x}_m) \\ \vdots\end{matrix}\right], \left[\begin{matrix}\vdots \\ \boldsymbol{y}^{(i)} \\ \vdots\end{matrix}\right], \left[\begin{matrix}\vdots \\ G(\boldsymbol{u}^{(i)})(\boldsymbol{y}^{(i)}) \\ \vdots\end{matrix}\right]\right]_{n \times p}$$

m over the function values; 1 over Loc.; 1 over Target Output

function values → Branch

Loc. → Trunk

[3]

- Can evaluate $G(u)(y)$ at different Locations y (for same u)
  - Nr. reuses of $\boldsymbol{u}^{(i)}$ → Hyperparameter $\boldsymbol{p}$

Training

Using Simulated Data:

1. Generate Input Functions $\boldsymbol{u(x)}$
   - e.g. Gaussian Random Fields
2. Numerically solve Problem: $\boldsymbol{G(u)(y)}$
3. Train NN on Data

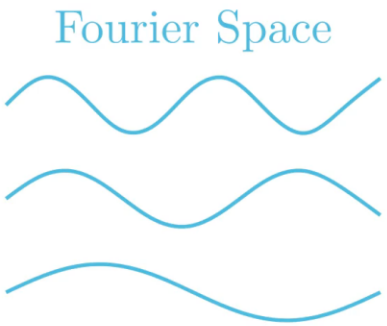
→ DeepONet has learned Operator

Real Data:

1. Data preparation
2. Train NN on Data

→ DeepONet has learned Operator

Intermission: Gaussian Random Fields



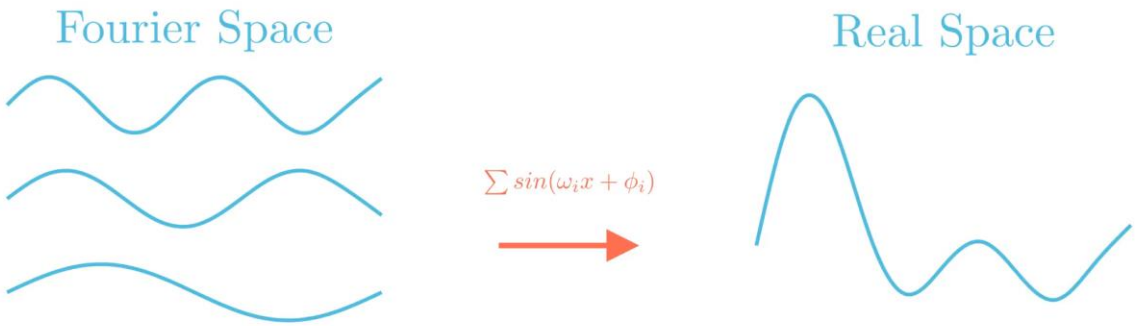
$$u\colon\ \mathbb{R}^p \to \mathbb{R}^d$$

$$[\boldsymbol{u}(\boldsymbol{x}_1), \boldsymbol{u}(\boldsymbol{x}_2), \dots, \boldsymbol{u}(\boldsymbol{x}_m)]$$

1. Fourier Space

2. Real Space

Intermission: Gaussian Random Fields



$$u\colon\ \mathbb{R}^p \to \mathbb{R}^d$$

$$[\boldsymbol{u}(\boldsymbol{x}_1), \boldsymbol{u}(\boldsymbol{x}_2), \ldots, \boldsymbol{u}(\boldsymbol{x}_m)]$$

1. Fourier Space

   randomly sample phase φ_i

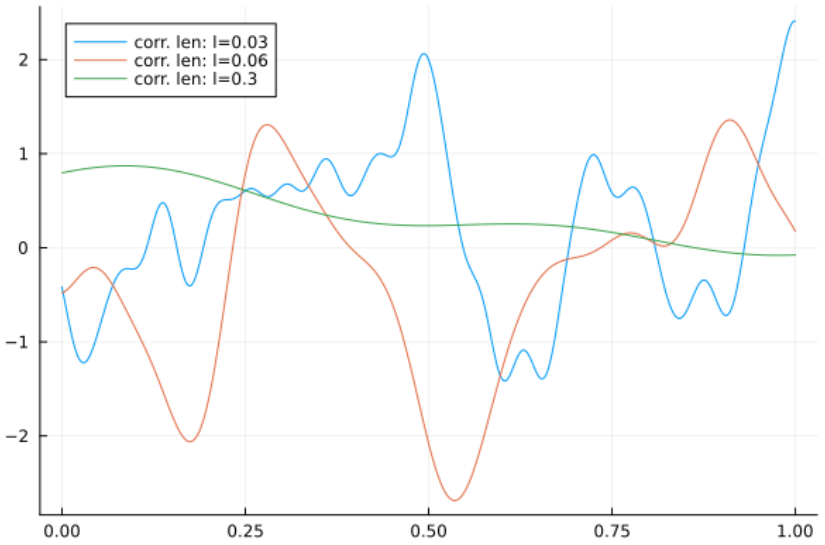
2. Real Space

   generate random function

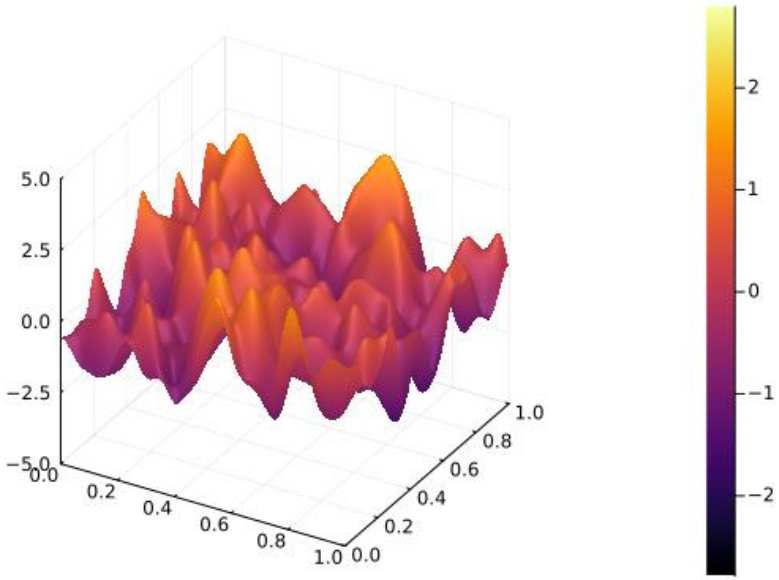
   Wavelength ➜ Pattern size

Intermission: Gaussian Random Fields

1D



2D



Application: Two Examples

Simple ODE

$$\frac{d}{dx}s(x) = u(x)$$

$$x \in [0,1],\ s(0) = c$$

$$G : u(x) \to s(y) = s(0) + \int_0^y u(\tau)d\tau$$

Diffusion-Reaction System

$$\partial_t s = D * \partial_x^2 s + ks^2 + u(x)$$

$$x \in [0,1],\ t \in [0,1]$$

$$G : u(x) \to s(x,t)$$

Explicit

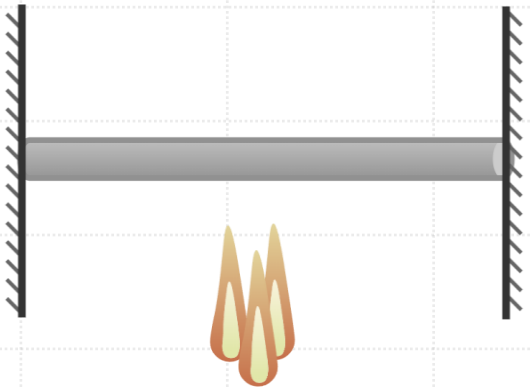
Implicit

Code Example

- DeepONet applied to 1D Heat Diffusion System:

$$\partial_t s = D\partial_x^2 s + u(x), \qquad (x,t) \in [0,1] \times [0,1]$$

1) 2)



1. Diffusion Term: Heat spreads out over t
2. Source Term: our input function $\boldsymbol{u(x)}$: $\mathbb{R} \to \mathbb{R}$

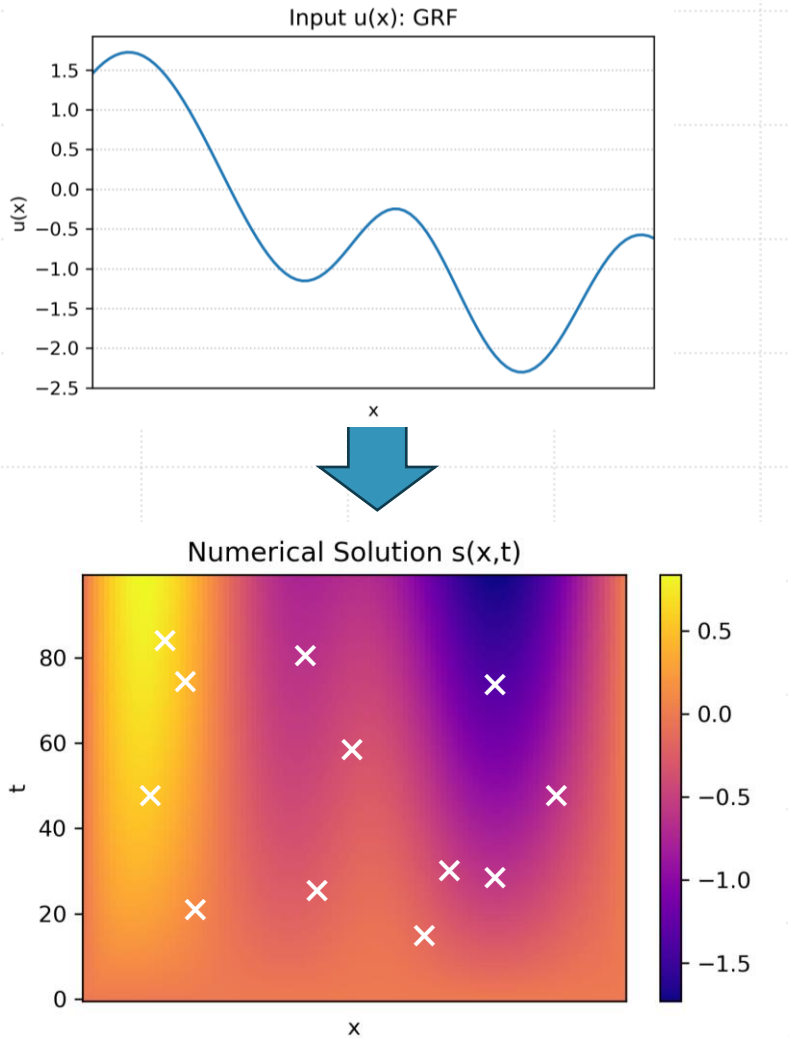
Predict: Temperature Dynamics: $\boldsymbol{s(x,t)}$: $\mathbb{R}^2 \to \mathbb{R}$

Code Example - Process

Using DeepXDE: [5]

1. Work on 100×100 Grid (x, t)
   - 100 space pts. = sensors
2. Get $u(x)$ from GRF
   - Length Scale: $l = 0.15$
3. Solve: $\partial_t s = D\partial_x^2 s + u(x)$
4. Sample $s(x, t)$ at 100 locations $y = (x, t)$

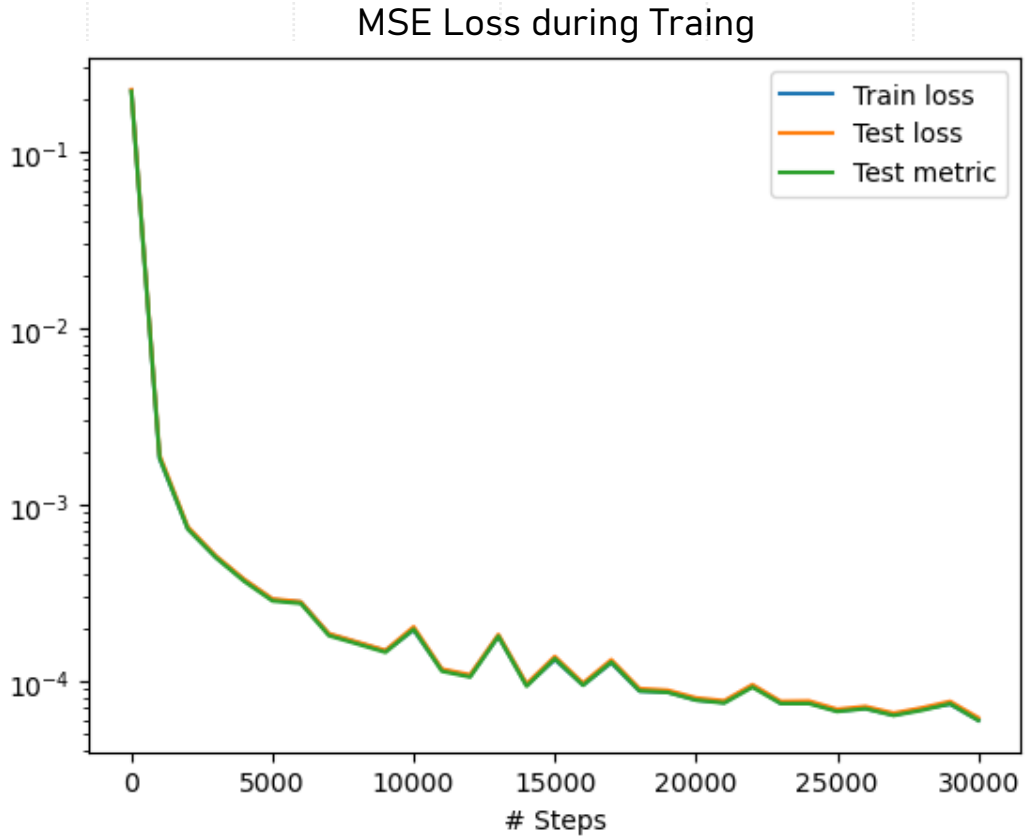
→ Train Model on Datapoints



Code Example – Training

- 2 hidden Layers × 40 Neurons
- 30k Train Samples
- 30k Test Samples

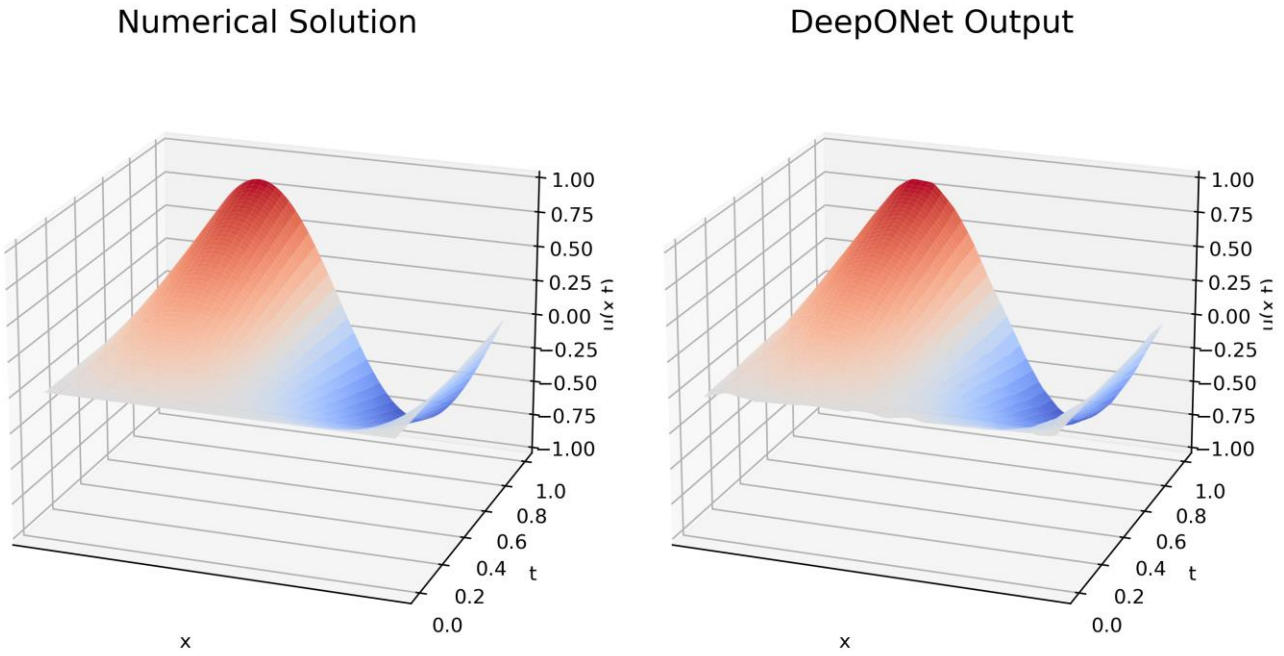
→ ~6h Training



Code Example – Results

source term: $u(x) = \sin(2\pi\, x)$

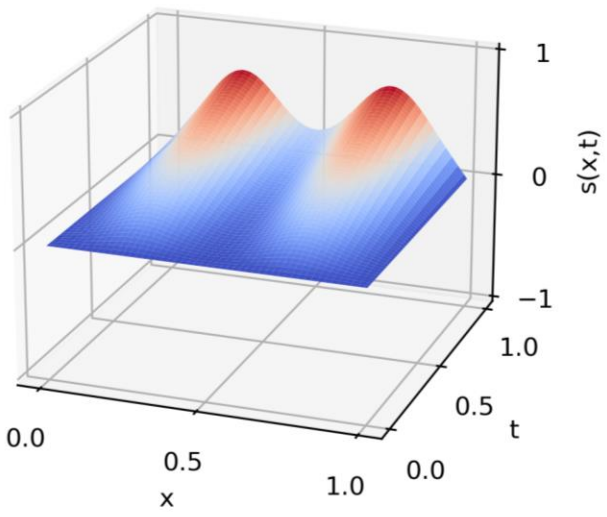
Numerics	DeepONet
-	Error: ~10^{-3}
One full simulation	NN inferences
Runtime: $9.18ms$	Time: $2.86ms$



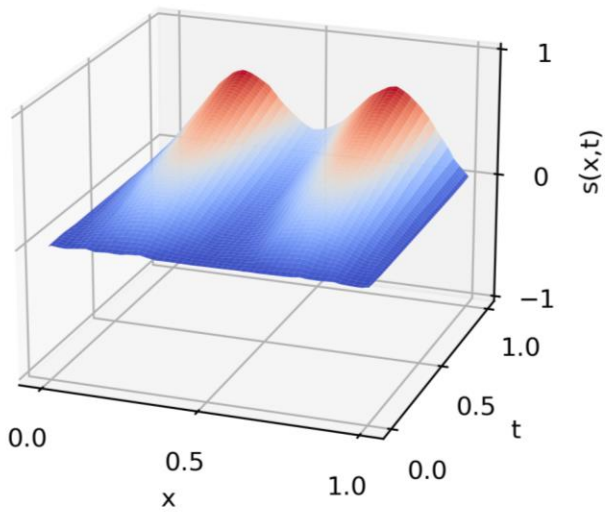
Code Example – Results

source term: $u(x)$: Double gaussian

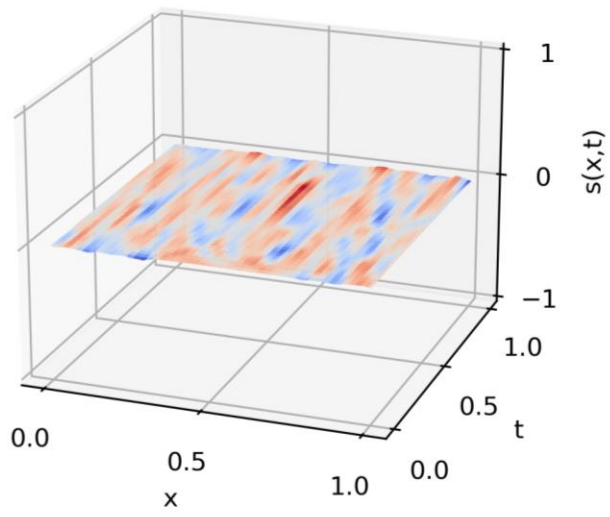
Numerical Solution



DeepONet Output

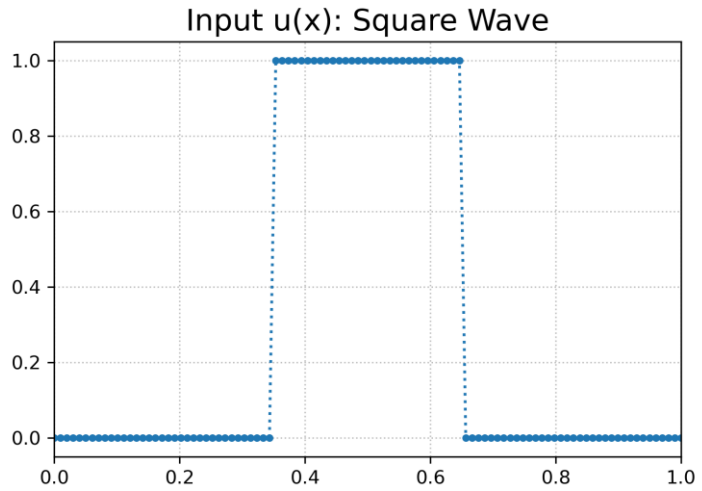


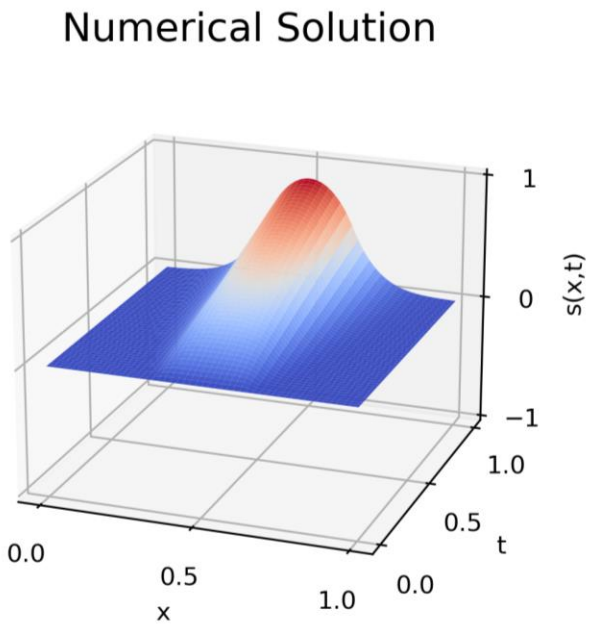
Pointwise Error

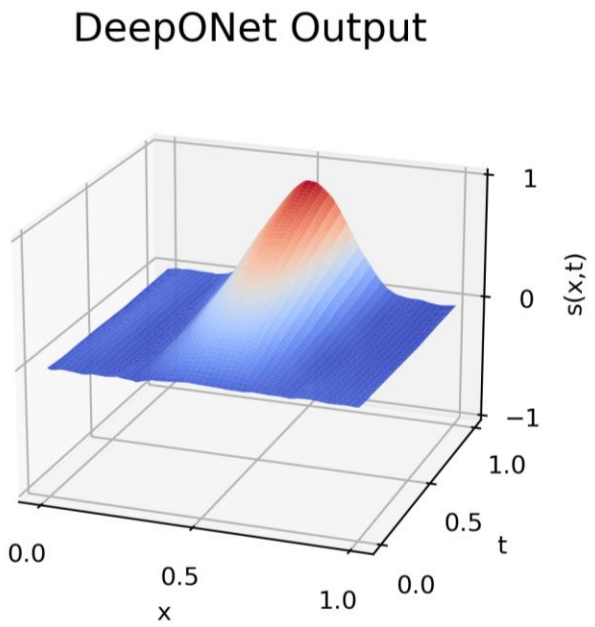


Code Example – Results

source term: $u(x)$: Square Wave







When can I use DeepONet?

Whenever there is a continuous operator!

Poisson:	$\nabla^2 V(x) = \rho(x)$	$G : \rho(x) \to V(x)$
Diffusion-Reaction:	$D * \nabla^2 s - \partial_t s = k s^2 + u(x)$	$G : u(x) \to s(x,t)$
QM Time Evolution:	$\psi(t) = e^{-iHt/\hbar}\psi_0$	$G : \psi_0 \to \psi(t)$

...ok, but when *should* I use DeepONet?

Advantages and Disadvantages

+ Learns the “solution space”
+ Fast and very efficient
+ High flexibility
+ Generalizes well
+ “Data-only” mode

– No continuous output
– “Curse of dimensionality”
– Requires training and data generation
– Dependency on sensor locations and input data

When should I use DeepONet?

Two Scenarios:

1. Measurement data with unknown relation
   (that is assumed to be representable by a continuous operator)
2. Known relation, obtaining outputs for many different “variants”/initial conditions **fast**

Use DeepONet to replace repeated (complex) numerical simulations

Summary

Architecture	Data Relation	Input	Output	Interpretability
PINN	PDE	Measurement data, BC/IC, PDE	“Hidden state” $u(t,x)$	Medium: ”Black box”, but good measure of accuracy
Symbolic Regression	Function	Measurement data: $\{x_i, y_i = f(x_i)\}$	Mathematical Relation $y = f(x)$	High (But no “feedback”)
NeuralODE	ODE	Initial State $x(0)$	Derivative $\dot{x}(t)$	Low: ”Black box”
DeepONet	(Continuous) Operator	Function values + output position: $\{u(x_j), y\}$	Function value $G(u)(y)$	Low: ”Black box”

References

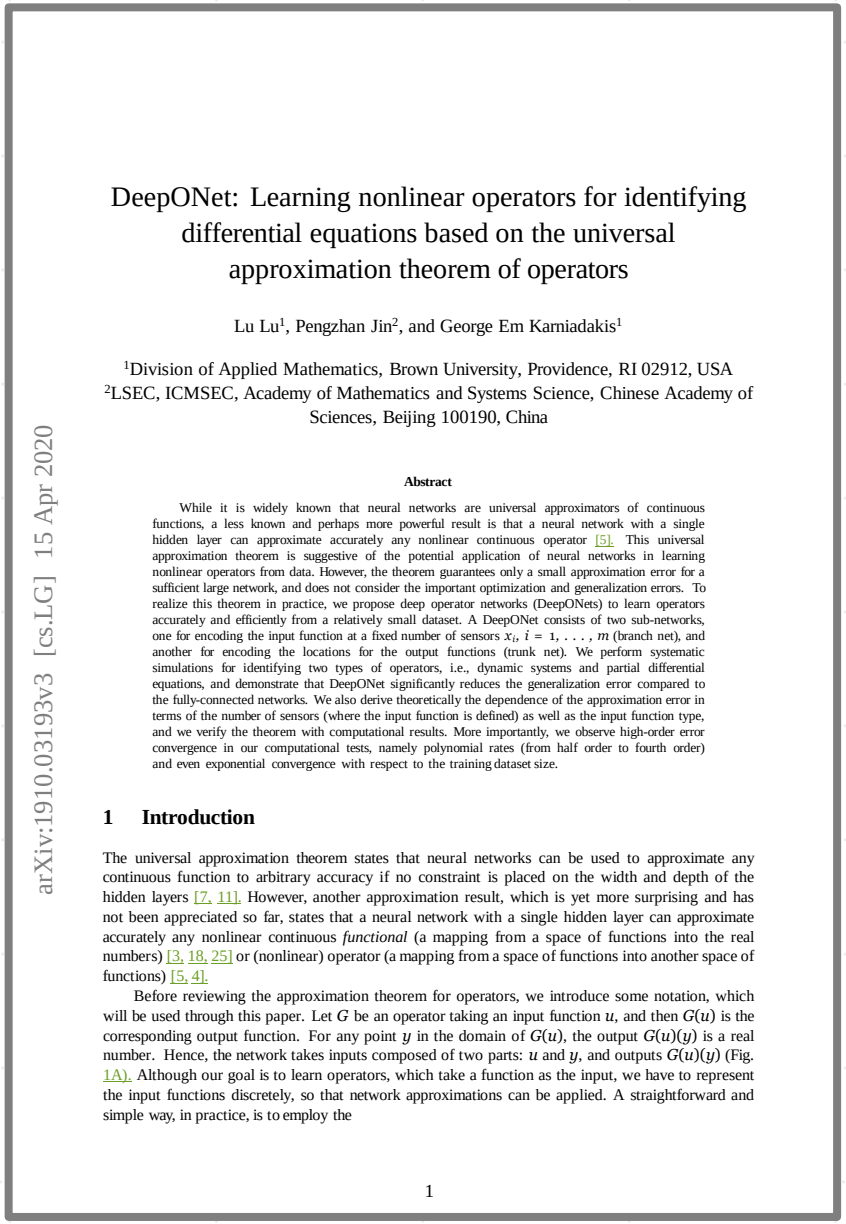
[1] Lu, Lu, et al. "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators." *Nature machine intelligence* 3.3 (2021): 218-229.

[2] https://www.analyticsvidhya.com/blog/2021/06/beginners-guide-to-universal-approximation-theorem/
(Last accessed 15.07.2023)

[3] Chen, Tianping, and Hong 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 (1995): 911-917

[4] Goswami, Somdatta, et al. "A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials." *Computer Methods in Applied Mechanics and Engineering* 391 (2022): 114587.

[5] Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. DeepXDE: A deep learning library for solving differential equations [Computer software]. https://github.com/lululxvi/deepxde



DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators

Lu Lu[1], Pengzhan Jin[2], and George Em Karniadakis[1]

[1]Division of Applied Mathematics, Brown University, Providence, RI 02912, USA
[2]LSEC, ICMSEC, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China

Abstract

While it is widely known that neural networks are universal approximators of continuous functions, a less known and perhaps more powerful result is that a neural network with a single hidden layer can approximate accurately any nonlinear continuous operator [5]. This universal approximation theorem is suggestive of the potential application of neural networks in learning nonlinear operators from data. However, the theorem guarantees only a small approximation error for a sufficient large network, and does not consider the important optimization and generalization errors. To realize this theorem in practice, we propose deep operator networks (DeepONets) to learn operators accurately and efficiently from a relatively small dataset. A DeepONet consists of two sub-networks, one for encoding the input function at a fixed number of sensors x_i, $i = 1, \ldots, m$ (branch net), and another for encoding the locations for the output functions (trunk net). We perform systematic simulations for identifying two types of operators, i.e., dynamic systems and partial differential equations, and demonstrate that DeepONet significantly reduces the generalization error compared to the fully-connected networks. We also derive theoretically the dependence of the approximation error in terms of the number of sensors (where the input function is defined) as well as the input function type, and we verify the theorem with computational results. More importantly, we observe high-order error convergence in our computational tests, namely polynomial rates (from half order to fourth order) and even exponential convergence with respect to the training dataset size.

1 Introduction

The universal approximation theorem states that neural networks can be used to approximate any continuous function to arbitrary accuracy if no constraint is placed on the width and depth of the hidden layers [7, 11]. However, another approximation result, which is yet more surprising and has not been appreciated so far, states that a neural network with a single hidden layer can approximate accurately any nonlinear continuous *functional* (a mapping from a space of functions into the real numbers) [3, 18, 25] or (nonlinear) operator (a mapping from a space of functions into another space of functions) [5, 4].

Before reviewing the approximation theorem for operators, we introduce some notation, which will be used through this paper. Let G be an operator taking an input function u, and then $G(u)$ is the corresponding output function. For any point y in the domain of $G(u)$, the output $G(u)(y)$ is a real number. Hence, the network takes inputs composed of two parts: u and y, and outputs $G(u)(y)$ (Fig. 1A). Although our goal is to learn operators, which take a function as the input, we have to represent the input functions discretely, so that network approximations can be applied. A straightforward and simple way, in practice, is to employ the

1

Backup Slides

Operators: Linearity and Continuity

Linear Operator: $G(u_1 + u_2) = G(u_1) + G(u_2)$ and $G(\alpha u_1) = \alpha G(u_1)$

Continuous Operator: $u_n \to \mathrm{u} \Rightarrow G(u_n) \to G(u)$

For linear operators continuity is equivalent to boundedness,
i.e. $\exists M: \|T(x)\| \leq M\|x\| \; \forall x \in X$

Linear operators: $\frac{\mathrm{d}}{\mathrm{dx}}, \partial, \nabla, \Delta, \Box$, Fourier Transform, Projection (-> Spin), Convolution, Time Translation

Nonlinear operators: $G(u) = u^2$

UAT for Operators

Conditions:

- σ continuous, non-polynomial
- Banach Space X
- $K_1 \subset X, K_2 \subset \mathbb{R}^d, V \subset C(K_1)$ (all compact)
- $G: V \to C(K_2)$ non-lin. cont. operator
- Network output $f_{c,\xi,\theta,\zeta}(\boldsymbol{x})(y)$

Statement: There are

- $n, m, p \in \mathbb{N}$
  $\mathrm{i} = (1, \dots, \mathrm{n}), \mathrm{j} = (1, \dots, \mathrm{m}), \mathrm{k} = (1, \dots, \mathrm{p})$
- $c_i^k, \xi_{ij}^k, \theta_i^k\ \zeta_k \in \mathbb{R},\ \ w_k \in \mathbb{R}^d,\ \ x_j \in K_1$

such that for any $\epsilon > 0$ the equation

$$|G(u)(y) - f(\boldsymbol{x})(y)| < \boldsymbol{\epsilon}$$

holds true $\forall\, u \in V, y \in K_2$

$$f_{c,\xi,\theta,\zeta}(\boldsymbol{x}, y) = \sum_{k=1}^{p} \sigma(w_k \cdot y + \zeta_k) \sum_{i=1}^{n} c_i^k \cdot \sigma\left(\sum_{j=1}^{m} \xi_{ij}^k \cdot u(x_j) + \theta_i^k\right)$$

Training – Overview

Hyperparameters – Architecture

- Network Type
  - FCNN (vs. CNN etc.)
- NN Architecture
  - Stacked – Unstacked
  - Depth & Width (branch + trunk)
  - Activation Function
- No. Sensors **m**
- Encoding dimension **p**

Hyperparameters – Training

- Function Space
  - GRF or Chebishev
  - Correlation length
- **P:** Reuse of input fcts. $u(x)$
- Loss function & Optimizer
  - PINN-style loss
- Iterations

Function, Functional, Operator

- Function
  $f: X \to Y, x \mapsto f(x)$
- Functional
  $S: X \to Y, x \mapsto S(x)$
- Operator
  $G: X \to Y, x \mapsto G(x)$

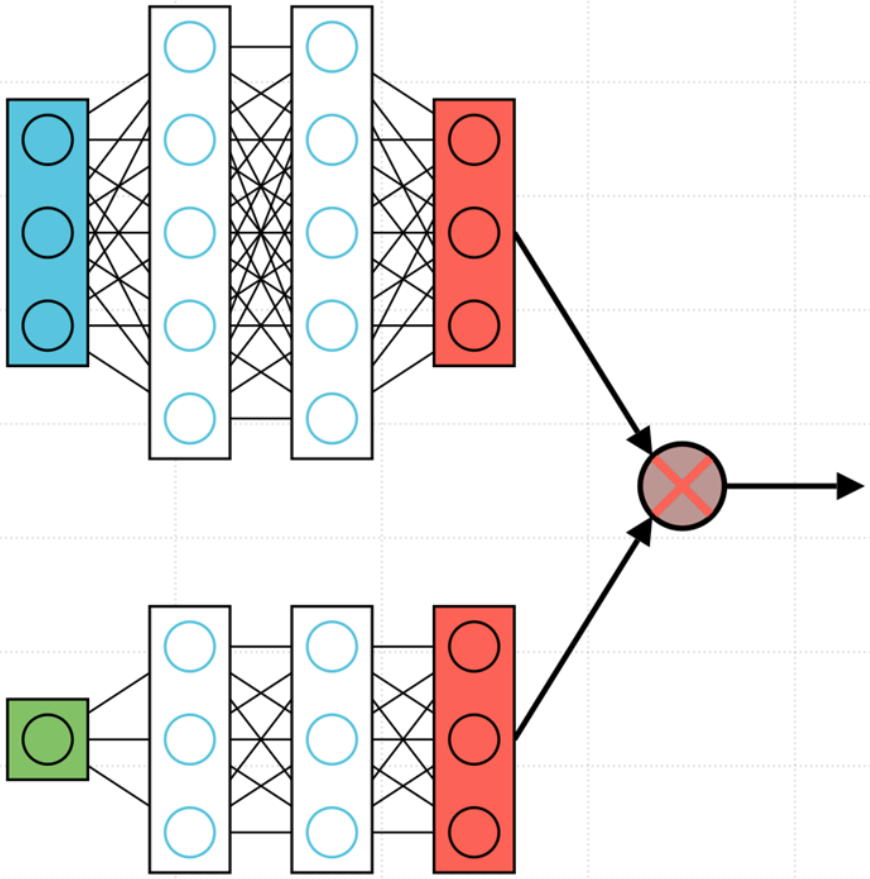
$f(x) = x^2$
$X = \mathbb{R}, Y = \mathbb{R}$

$S[\boldsymbol{q}(t)] = \int_{t_1}^{t_2} dt\, L(\boldsymbol{q}, \dot{\boldsymbol{q}}, t)$
$X = \mathrm{A}, \mathrm{Y} = \mathbb{R}$

$G(f) = \nabla f$
$X = B, Y = C$

- **Needs revision, an operator is also a function**
  Also: What is a linear and nonlinear operator? Examples for different types of operators are important, because this is directly related to use-cases of the DeepONet

DeepONet | Network Architecture



- We need a slide that links this architecture to the specific form of $f_{c,\xi,\theta,\zeta}(\boldsymbol{x}, y)$.
- Mention grid search over hyperparameters – depth 2, width 2560. However they later use width 40 for an example problem –> Networks are not very complex

DeepONet | Training

- At first, explain the basics of the training procedure – restate what the inputs and outputs of the network are.
- Mention that the sensor locations must be the same for all functions we are training on -> The chosen sensors have a strong impact on the performance and “region” of valid outputs
- Also explain that since there are many (in the paper around 100) sensor inputs, but only one input y and ground truth point G(u)(y), the sensor inputs can be ”recycled”
  -> In the paper they use the same function for 100 training iterations with different values of y
- What is the loss? How many iterations? In the paper they use 50000 (100 functions with 100 iterations each, five epochs(?))

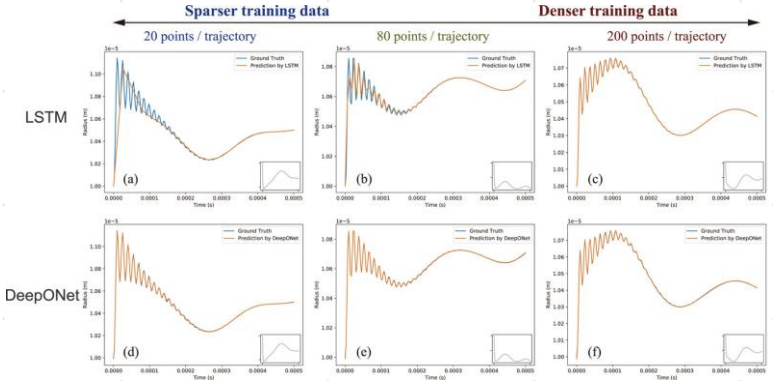
DeepONet | Training

- Now we need to talk about what the input functions look like
- Briefly explain GRF and Chebyshev Polynomials (mention cool blog article on GRF by STRUCTURES group in HD)
- Maybe make some plots to show some samples of these function spaces?

Code Example

- Show Notebook here?

Context and Outlook



In what settings is it advantageous? Which advantages does it have compared to numerical solvers or other NN approaches?

- Advantage to solvers: Learns the "solution space", thus we can easily change initial conditions and also the actual underlying function
- Advantage: Great generalisation properties -> see sparse data performance in chemical application, Fig. 8
- Advantage: Very fast inference time, computationally advantageous compared to numerical sim. -> see inference time in chemical application, Fig. 17
- Advantage: Can work with data, no need to know the actual underlying equations!
- Obvious disadvantage: No continuous output.
- Disadvantage: When the output function is higher-dim., the grid of output points scales with N^d.

Context | Use-Cases

When can I use DeepONet?

Whenever there is a continuous operator!

$$i\hbar\partial_t \Psi(\mathbf{r},t) = -\frac{\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},t) + V(\mathbf{r})\Psi(\mathbf{r},t)$$

$$\frac{\partial^2 h_{ij}}{\partial t^2} - c^2\nabla^2 h_{ij} = -16\pi G T_{ij}$$

~~$$\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{v} + \mathbf{f}$$~~

What is the nu?
Cross out Navier Stokes, it can not be represented by continuous operator!

Outlook

- We need to address the following points:
  - How fast/slow is the inference (and the training)? Can it be used only for settings where we are interested in a few points of the solution, or can we actually get so many output points that we can interpolate smooth solutions on a large domain?
- Mention Combinations with other methods?
  - Physics informed DeepONet…