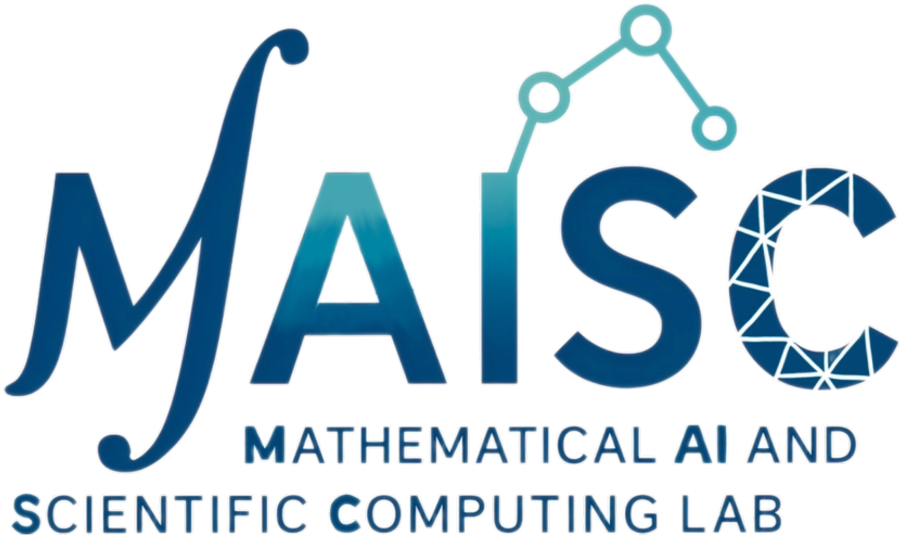
Scientific ML for PDEs: PINNs & Neural Operators



Jaeyong Lee

Department of AI

Chung-Ang University

CAC 2026, June 22-26, 2026

Contents

Part0. Introduction to Scientific AI

Jaeyong Lee

- Assistant Professor, Department of AI, Chung-Ang University

(Personal Homepage: https://www.jaeyong-lee.com/) (Mar. 2024 - Present)

Jaeyong Lee

- Assistant Professor, Department of AI, Chung-Ang University

(Personal Homepage: https://www.jaeyong-lee.com/) (Mar. 2024 - Present)

- **AI Research Fellow, Center for AI and Natural Sciences (CAINS), Korea Institute for Advanced Study (KIAS)**, Republic of Korea (Mar. 2022 - Feb. 2024)

- Ph.D. in Mathematics, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea (Mar. 2017 - Feb. 2022)

Advisor: Prof. Hyung Ju Hwang

- B.S. in Mathematics, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea (Mar. 2011 - Feb. 2017)

Jaeyong Lee

- Assistant Professor, Department of AI, Chung-Ang University

(Personal Homepage: https://www.jaeyong-lee.com/)

- Supervisor of **Mathematical AI and Scientific Computing (MAISC) Lab**

(Lab Homepage: https://sites.google.com/view/maisc-lab/)

MAISC

MATHEMATICAL AI AND SCIENTIFIC COMPUTING LAB

Jaeyong Lee

- Assistant Professor, Department of AI, Chung-Ang University

(Personal Homepage: https://www.jaeyong-lee.com/)

- Supervisor of **Mathematical AI and Scientific Computing (MAISC) Lab**

(Lab Homepage: https://sites.google.com/view/maisc-lab/)

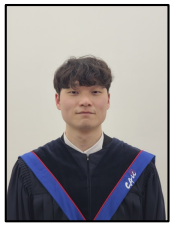
MAISC
MATHEMATICAL AI AND SCIENTIFIC COMPUTING LAB

Family

Marc Haltmayer
Ph.D. Student

Junyeong Jang
M.S.-Ph.D. Integrated Student

Yu Seung Lee
Ph.D. Student



Hyeonjun Choi
M.S.-Ph.D. Integrated Student

Seomkyun Song
M.S. Student

Jinkyu Jeong
Ph.D. Student

Jaemin Seo
M.S.-Ph.D. Integrated Student

Jaeyong Lee

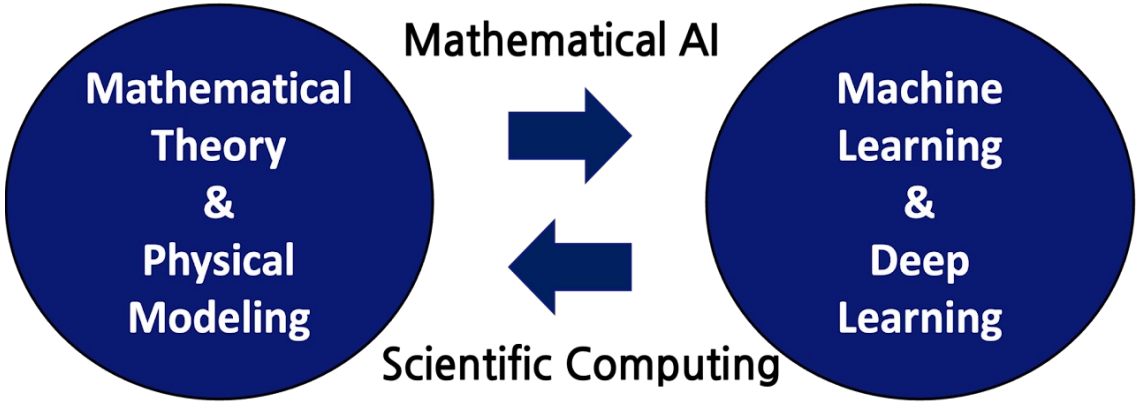
- Assistant Professor, Department of AI, Chung-Ang University

(Personal Homepage: https://www.jaeyong-lee.com/)

- Supervisor of **Mathematical AI and Scientific Computing (MAISC) Lab**

(Lab Homepage: https://sites.google.com/view/maisc-lab/)

MAISC
MATHEMATICAL AI AND SCIENTIFIC COMPUTING LAB



[Research Topics]

- AI for Science (AI4Science)
- Scientific Machine Learning (SciML)
- Neural Ordinary Differential Equations (Neural ODE)
- Physics-Informed Machine Learning
- Neural PDE solver
- Data-Driven Physics Modeling

Jaeyong Lee

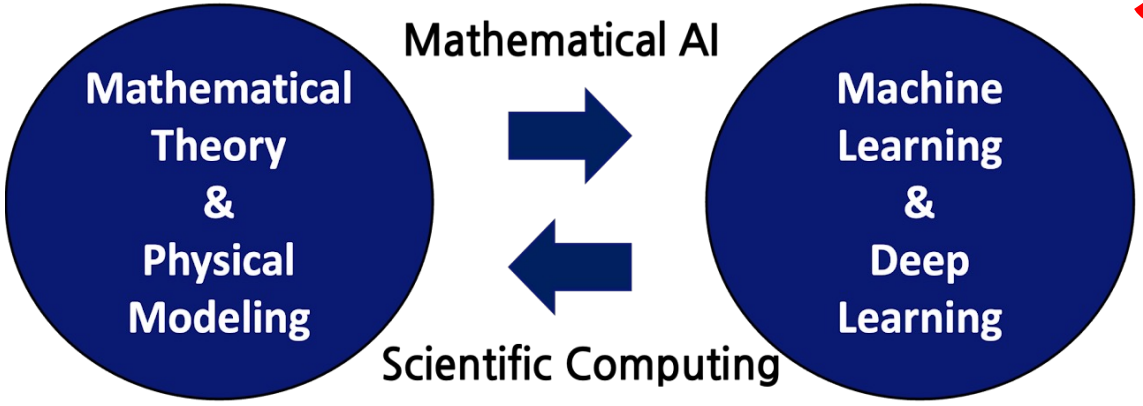
- Assistant Professor, Department of AI, Chung-Ang University

(Personal Homepage: https://www.jaeyong-lee.com/)

- Supervisor of **Mathematical AI and Scientific Computing (MAISC) Lab**

(Lab Homepage: https://sites.google.com/view/maisc-lab/)

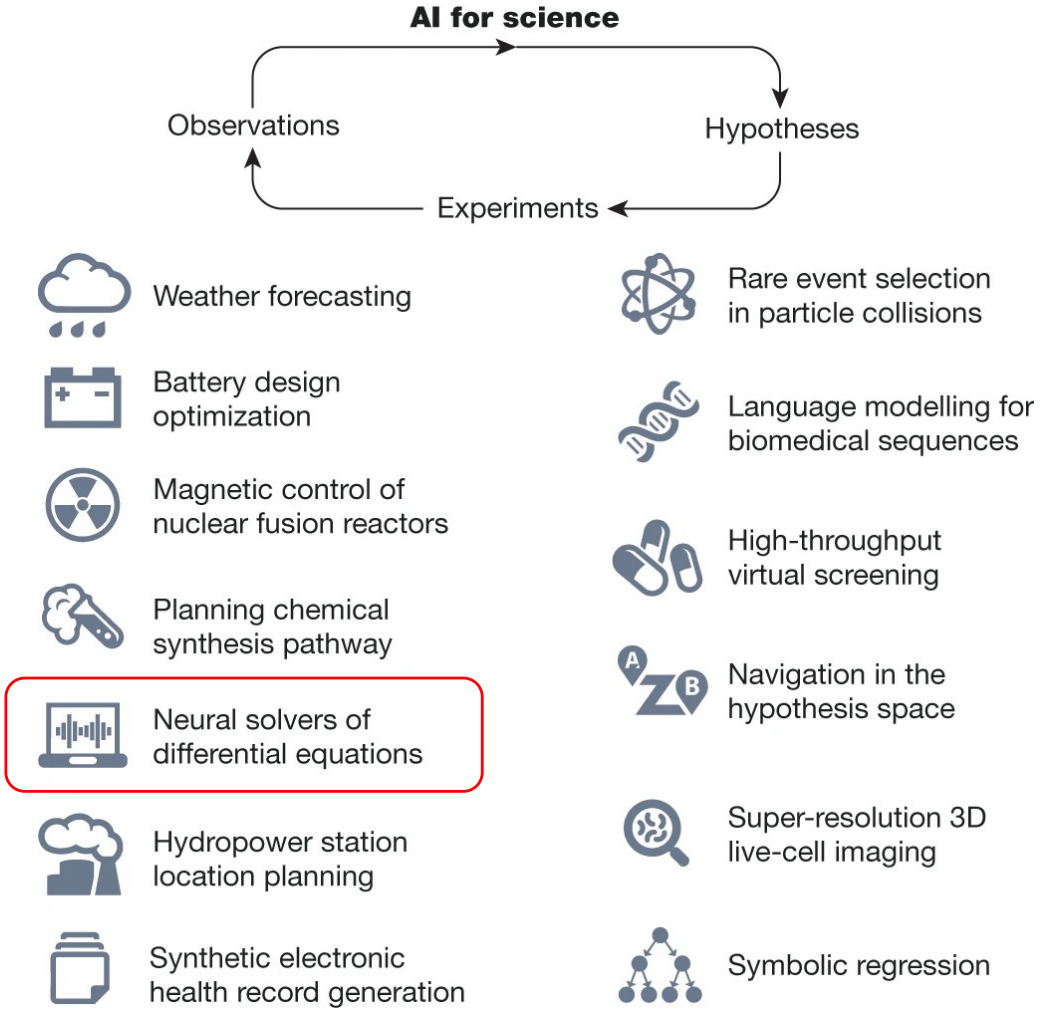
MAISC
MATHEMATICAL AI AND SCIENTIFIC COMPUTING LAB



[Research Topics]

- **AI for Science (AI4Science)**
- **Scientific Machine Learning (SciML)**
- Neural Ordinary Differential Equations (Neural ODE)
- Physics-Informed Machine Learning
- Neural PDE solver
- Data-Driven Physics Modeling

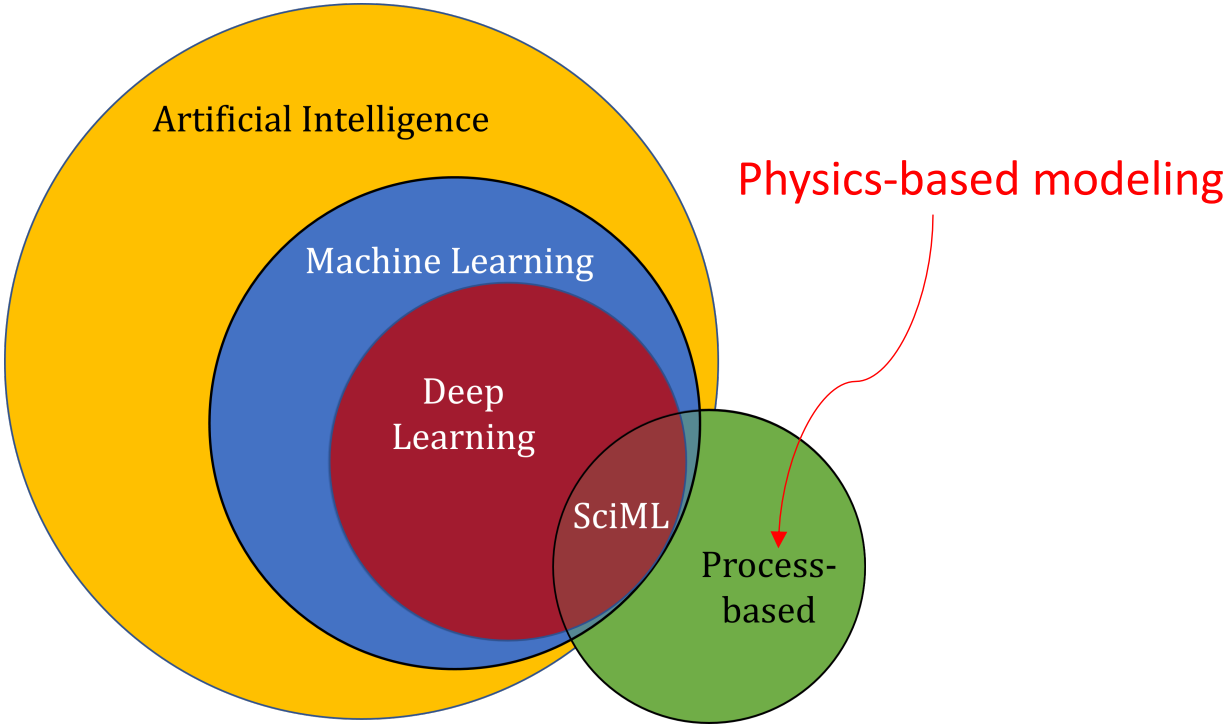
AI for Science (AI4Science)



- Accelerate scientific observation and discovery
- Interpret large datatsets
- Gain insights that might not have been possible using traditional scientific methods alone.

https://ai4sciencecommunity.github.io/

Scientific Machine Learning (SciML)



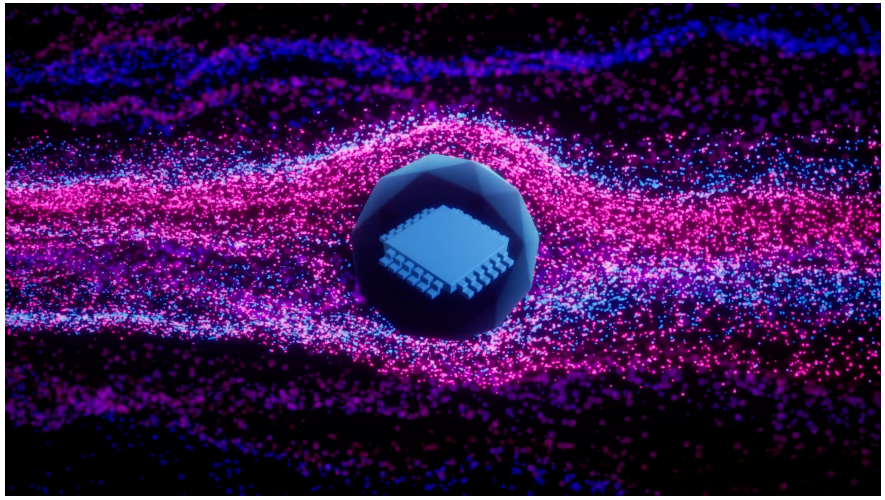
SciML = Machine learning + Science(Physics-based modeling)

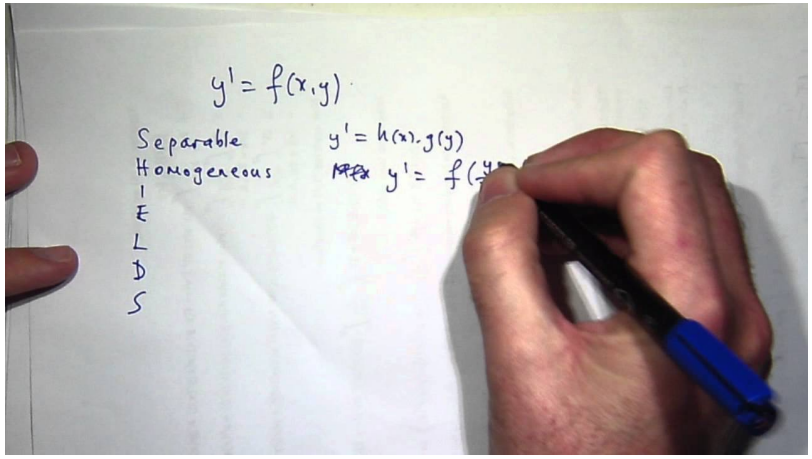
- Physics-based models contain mathematical formulas that are based on scientific knowledge and physical laws.
- To build these models, observational studies and experiments were carried out.

https://sciml.wur.nl/reviews/sciml/sciml.html

Differential equations

- **In mathematics, a differential equation is an equation that relates one or more functions and their derivatives.**
  - The functions generally represent **physical quantities.**
  - The derivatives represent **their rates of change.**
  - The differential equation defines a **relationship between the above two.**

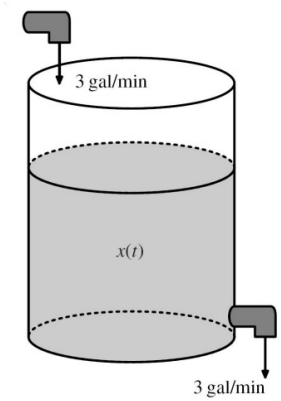




https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/
https://www.youtube.com/watch?v=jb9U-ZOlvDs

Ordinary Differential equations (ODEs) and Partial Differential equations (PDEs)

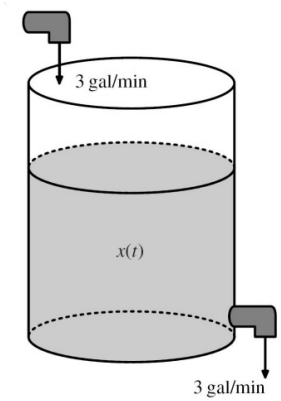
ODE example)



Let $x(t)$ denote the amount of salt at time t.

Ordinary Differential equations (ODEs) and Partial Differential equations (PDEs)

ODE example)



Let $x(t)$ denote the amount of salt at time t.

Balance Law: (rate in)-(rate out)

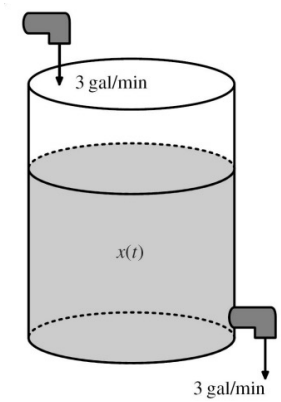
$$\frac{dx(t)}{dt} = 3 - 3x(t).$$

Initial condition (IC) :

$$x(t=0) = 100.$$

Ordinary Differential equations (ODEs) and Partial Differential equations (PDEs)

ODE example)



Let $x(t)$ denote the amount of salt at time t.

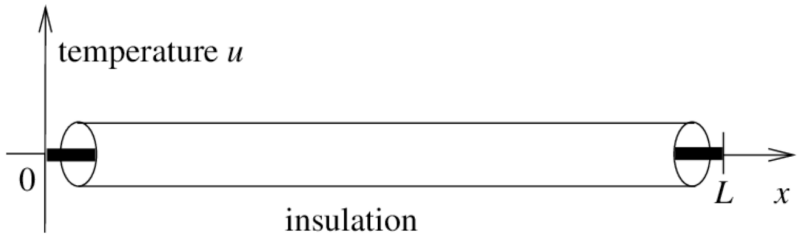
Balance Law: (rate in)-(rate out)

$$\frac{dx(t)}{dt} = 3 - 3x(t).$$

Initial condition (IC) :

$$x(t=0) = 100.$$

PDE example)



Let $u(t,x)$ denote the temperature at point x at time t.

Governing equation (heat equation) :

$$\frac{\partial u(t,x)}{\partial t} = k\frac{\partial^2 u(t,x)}{\partial x^2}.$$

Initial condition (IC) :

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

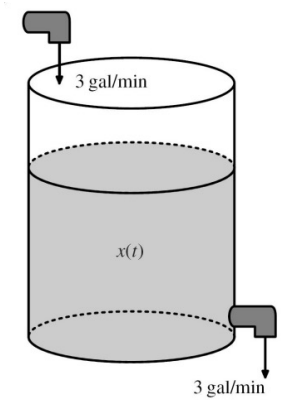
Boundary condition (BC) :

$$u(t,x=0) = h(t), \qquad u(t,x=L) = g(t).$$

https://math.libretexts.org/Bookshelves/Differential_Equations/Book%3A_Differential_Equations_for_Engineers_(Lebl)/4%3A_Fourier_series_and_PDEs/4.06%3A_PDEs_separation_of_variables_and_the_heat_equation

It is hard to solve the equation exactly!!

ODE example)



Let $x(t)$ denote the amount of salt at time t.

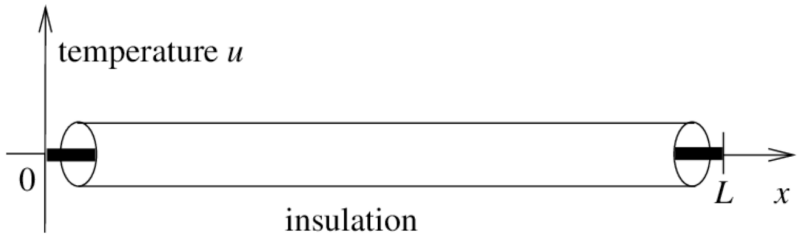
Balance Law: (rate in)-(rate out)

$$\frac{dx(t)}{dt} = 3 - 3x(t).$$

Initial condition (IC) :

$$x(t=0) = 100.$$

PDE example)



Let $u(t,x)$ denote the temperature at point x at time t.

Governing equation (heat equation) :

$$\frac{\partial u(t,x)}{\partial t} = k\frac{\partial^2 u(t,x)}{\partial x^2}.$$

Initial condition (IC) :

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

Boundary condition (BC) :

$$u(t,x=0) = h(t), \qquad u(t,x=L) = g(t).$$

It is hard to solve the equation exactly!!

Let $x(t)$ denote the amount of salt at time t.

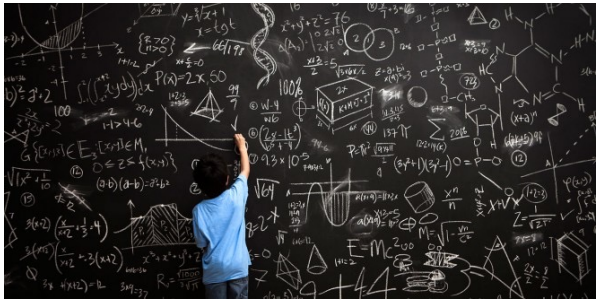
Balance Law: (rate in)-(rate out)

$$\frac{dx(t)}{dt} = 3 - 3x(t).$$

Initial condition (IC) :

$$x(t=0) = 100.$$

Numerical analysis



Let $u(t,x)$ denote the temperature at point x at time t.

Governing equation (heat equation) :

$$\frac{\partial u(t,x)}{\partial t} = k\frac{\partial^2 u(t,x)}{\partial x^2}.$$

Initial condition (IC) :

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

Boundary condition (BC) :

$$u(t,x=0) = h(t), \qquad u(t,x=L) = g(t).$$

Traditional numerical methods

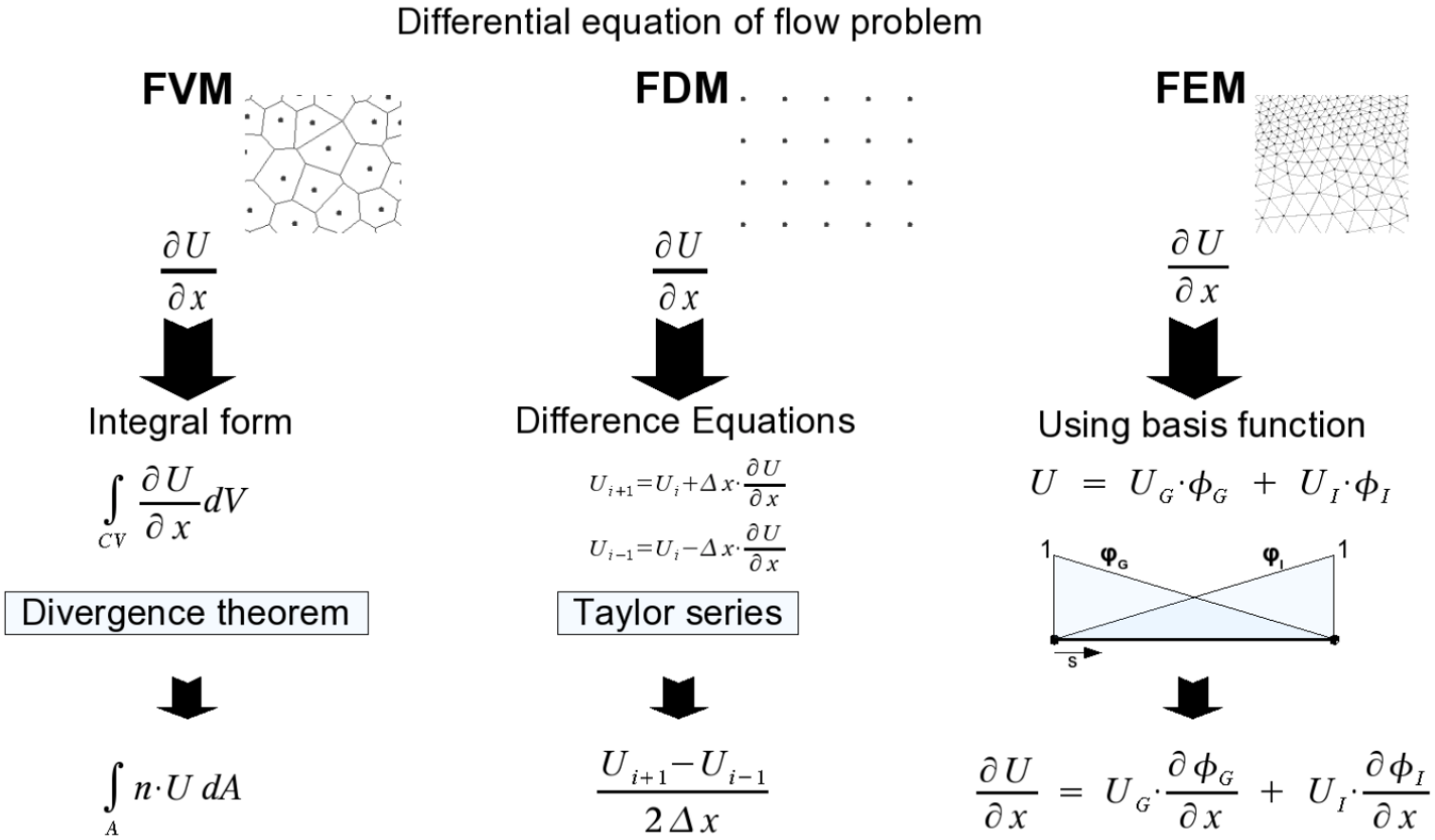
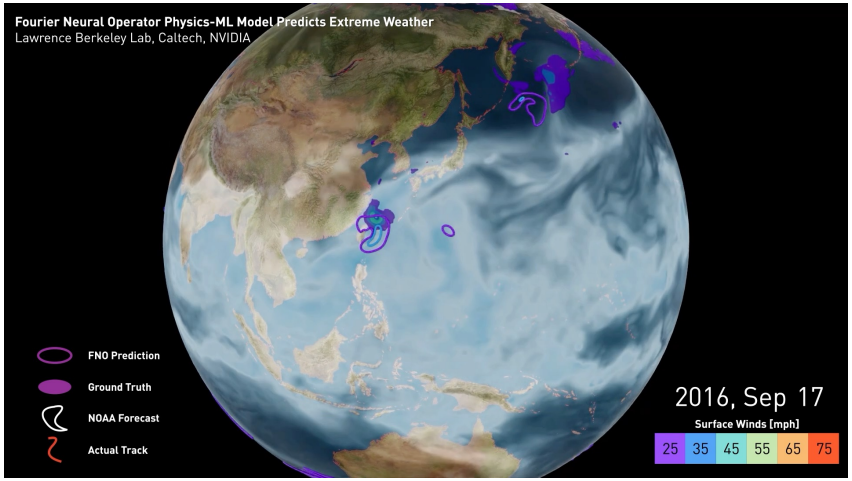
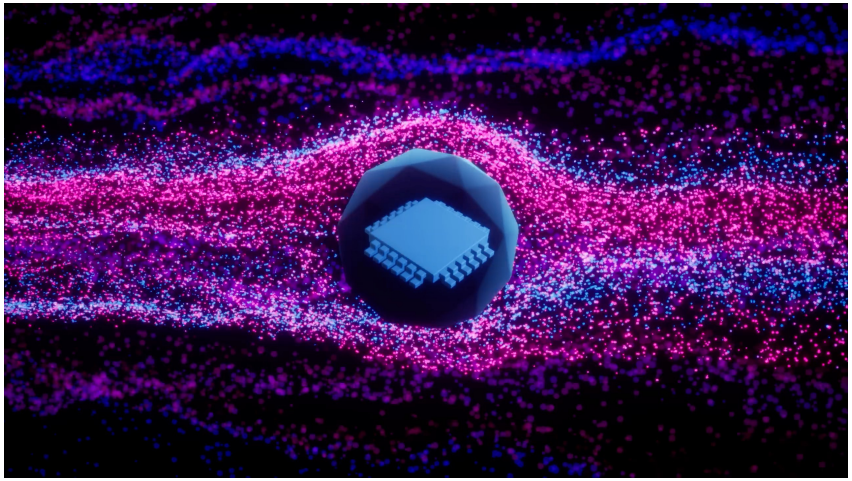


Figure 1: The most commonly used numerical methods and their essential differences

https://www.researchgate.net/publication/266465949_GENERALIZED_STABILIZATION_TECHNIQUES_IN_COMPUTATIONAL_FLUID_DYNAMICS

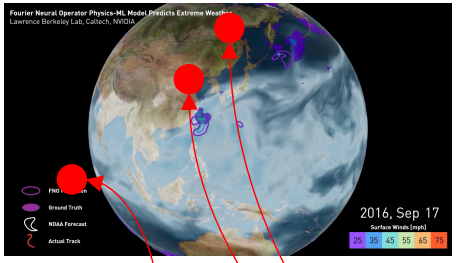
Physics-Informed Machine Learning



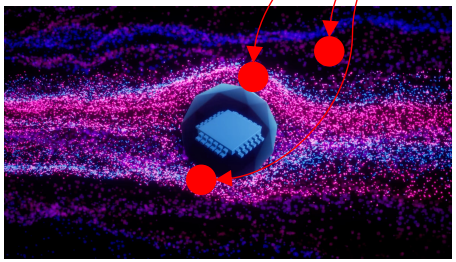


https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/

Physics-Informed Machine Learning

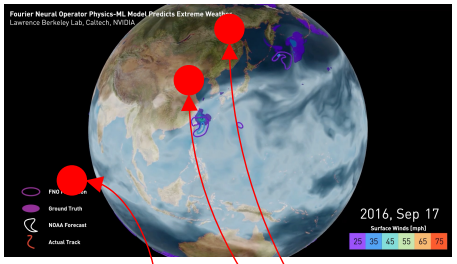


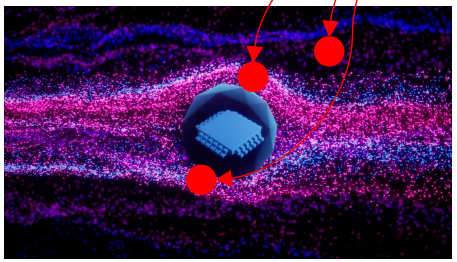
Sensors



https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/

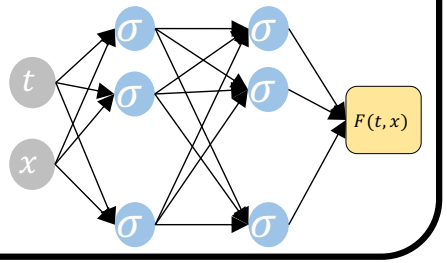
Physics-Informed Machine Learning





Sensors → **Data science**

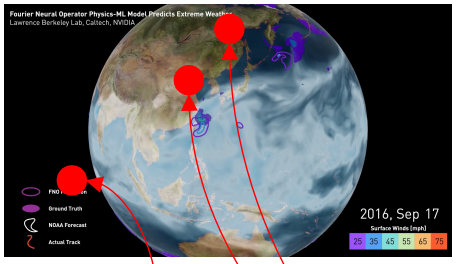
Machine learning, AI

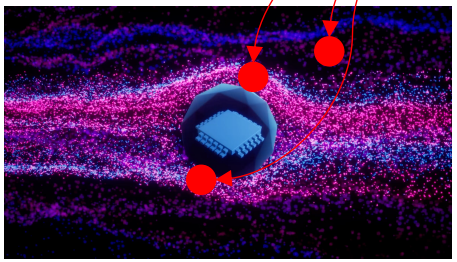


Understand and Predict

https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/

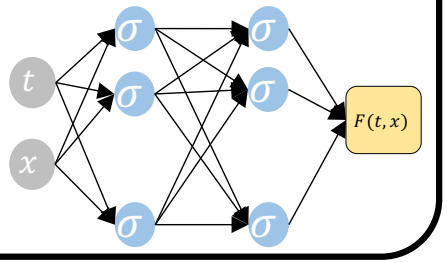
Physics-Informed Machine Learning





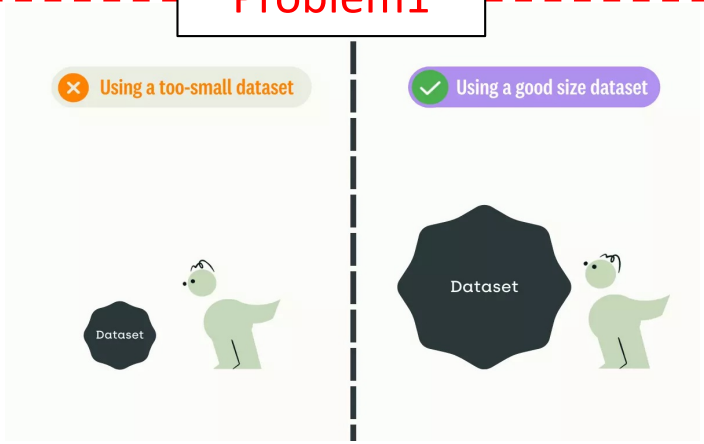
Sensors → **Data science**

Machine learning, AI



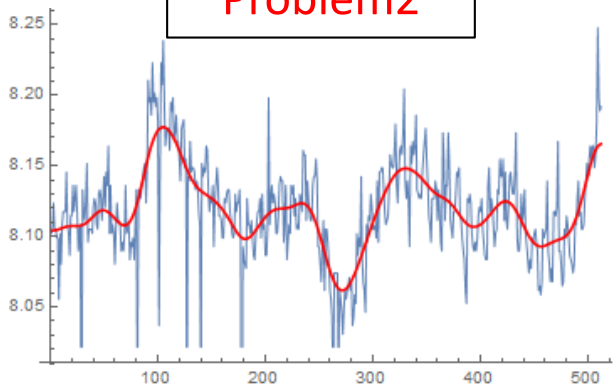
Understand and Predict

Problem1



Insufficient amount of data

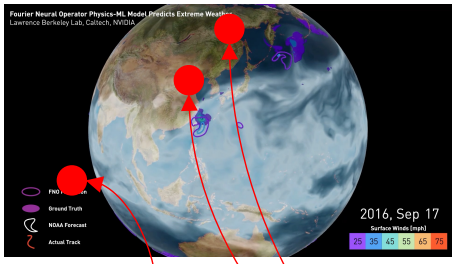
Problem2

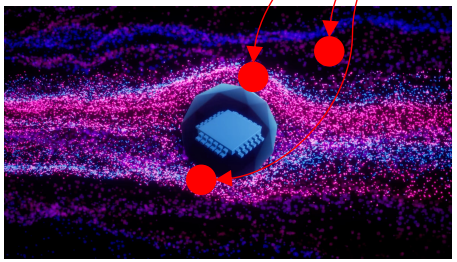


Noisy data

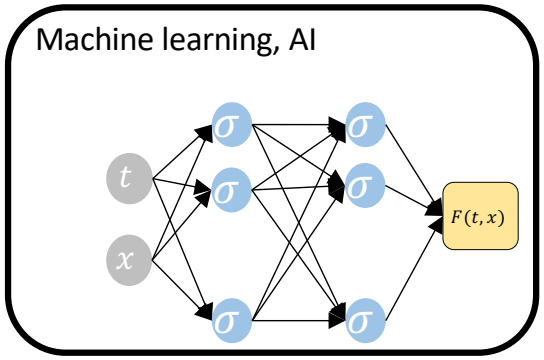
https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/

Physics-Informed Machine Learning





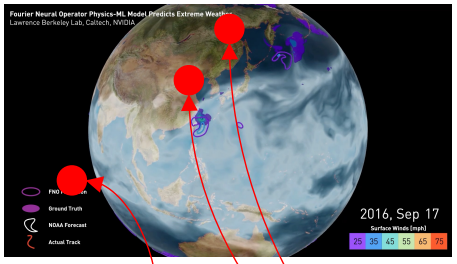
Sensors → **Data science**

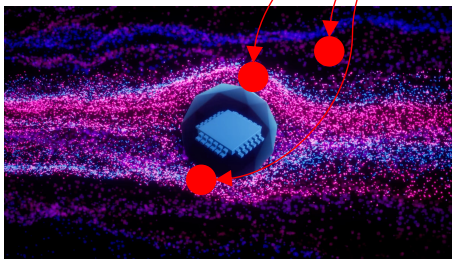


Understand and Predict

https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/

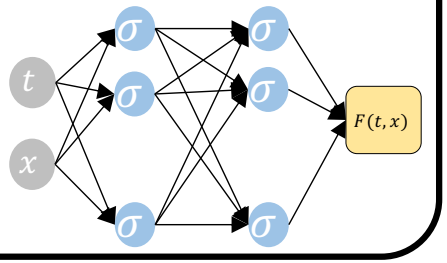
Physics-Informed Machine Learning



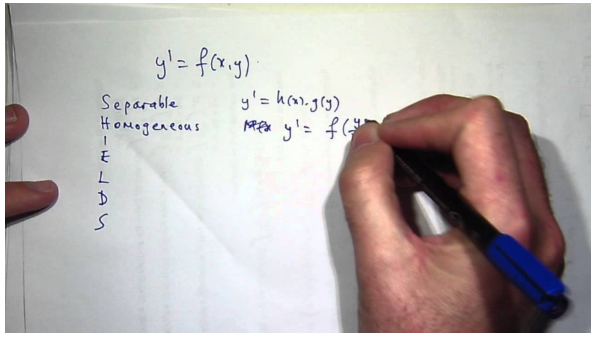


Sensors → **Data science**

Machine learning, AI



Understand and Predict



Physical and mathematical meaning

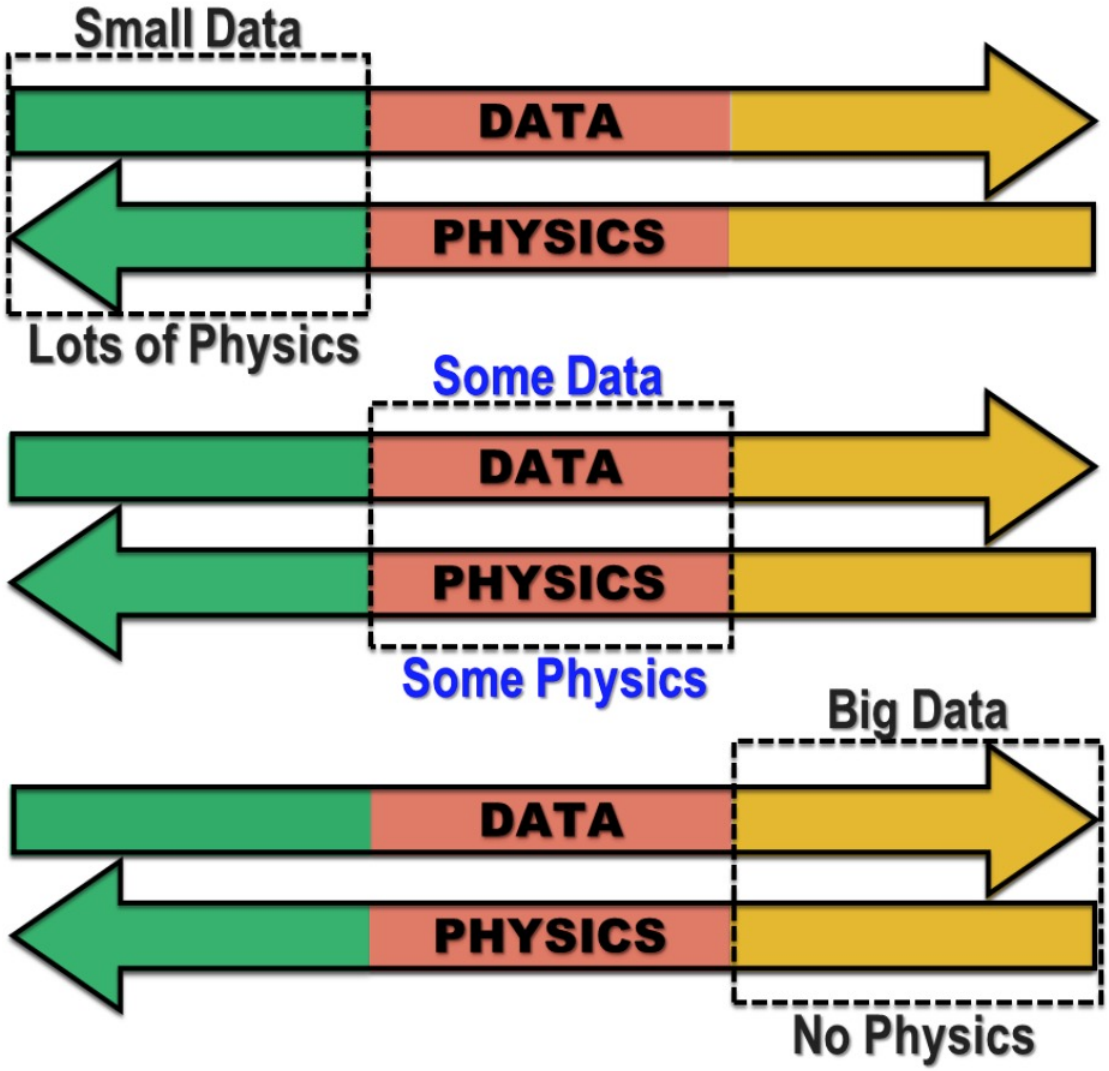
$u(t,x)$: Temperature at location x at time t.

→ Heat equation $\frac{\partial u(t,x)}{\partial t} = \frac{\partial^2 u(t,x)}{\partial x^2}$

Partial **D**ifferential **E**quation (**PDE**)

https://www.quantamagazine.org/latest-neural-nets-solve-worlds-hardest-equations-faster-than-ever-before-20210419/

Physics-Informed Machine Learning



Yang, Liu, Dongkun Zhang, and George Em Karniadakis. "Physics-informed generative adversarial networks for stochastic differential equations." *SIAM Journal on Scientific Computing* 42.1 (2020): A292-A317.

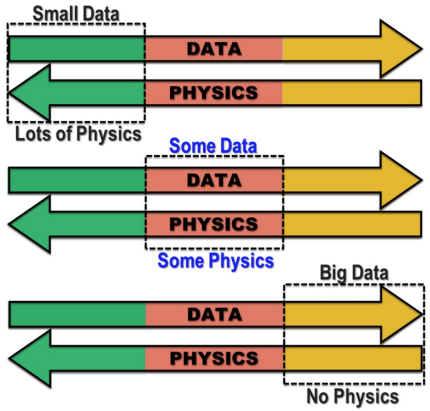
Physics-Informed Machine Learning

Physics

Computational Fluid Dynamics (CFD)

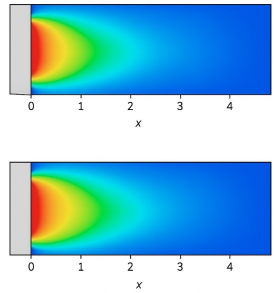
(e.g., Navier-Stokes equation)

$$\rho\left[\frac{\partial V}{\partial t}+(V\cdot\nabla)V\right]=-\nabla p+\rho\vec{g}+\mu\nabla^2\vec{V}$$



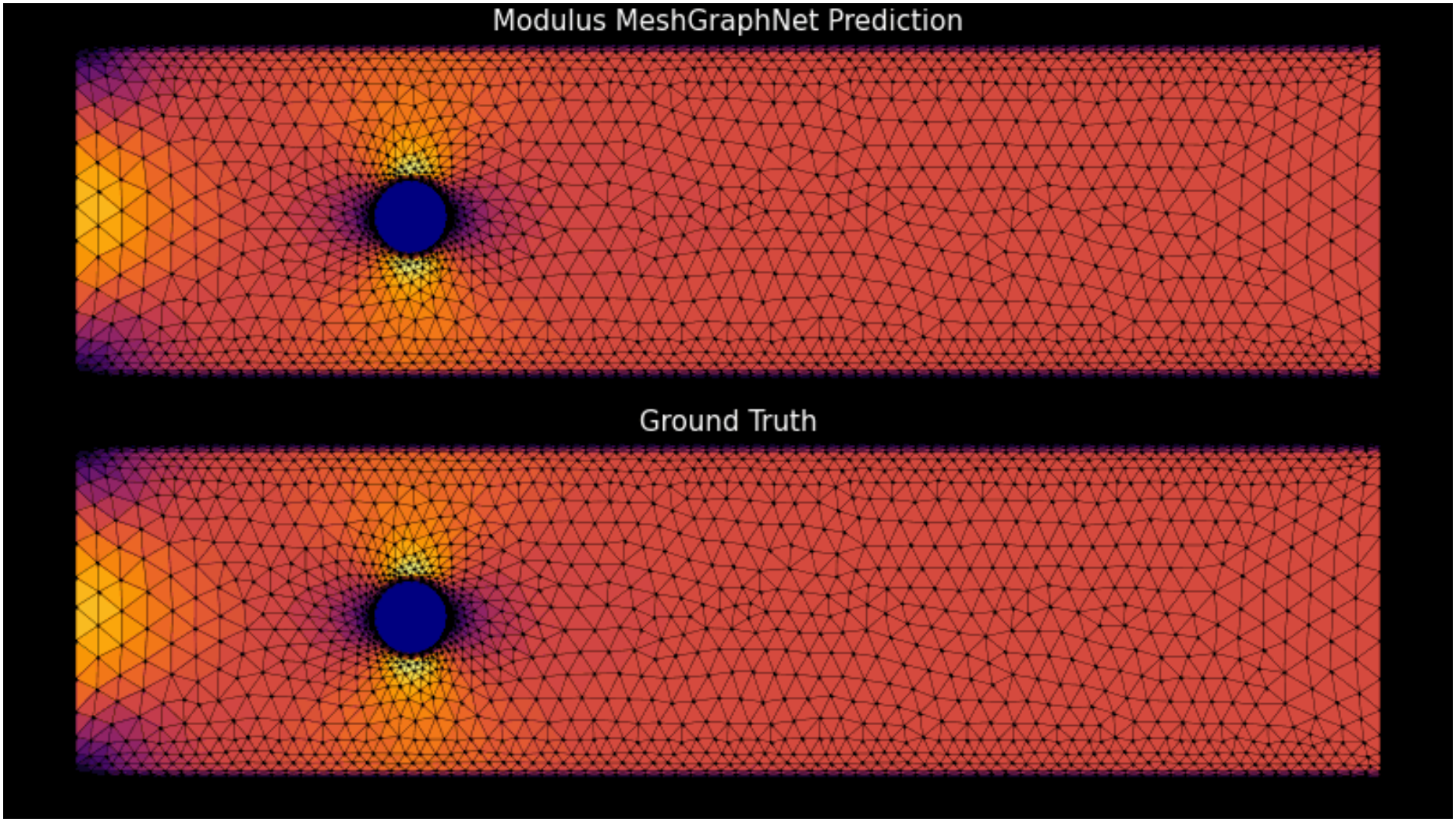
Data

(Experimental results)



Yang, Liu, Dongkun Zhang, and George Em Karniadakis. "Physics-informed generative adversarial networks for stochastic differential equations." *SIAM Journal on Scientific Computing* 42.1 (2020): A292-A317.

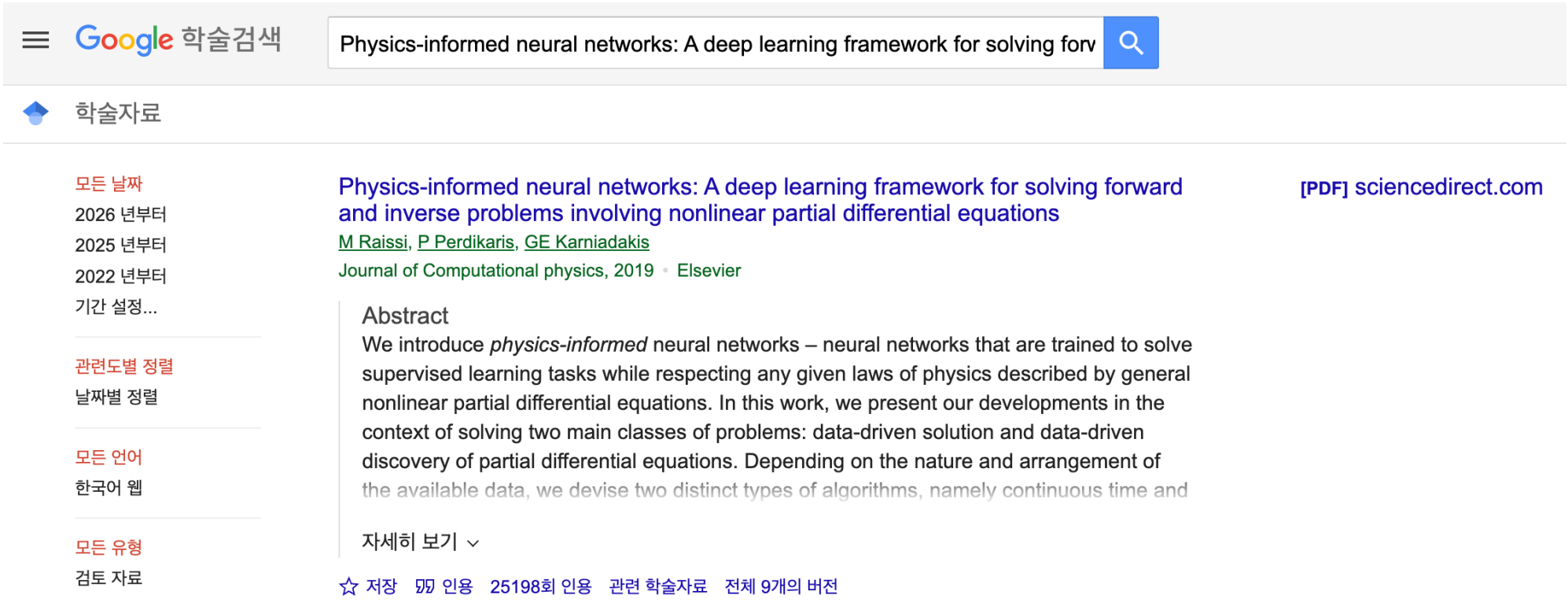
Physics-Informed Machine Learning



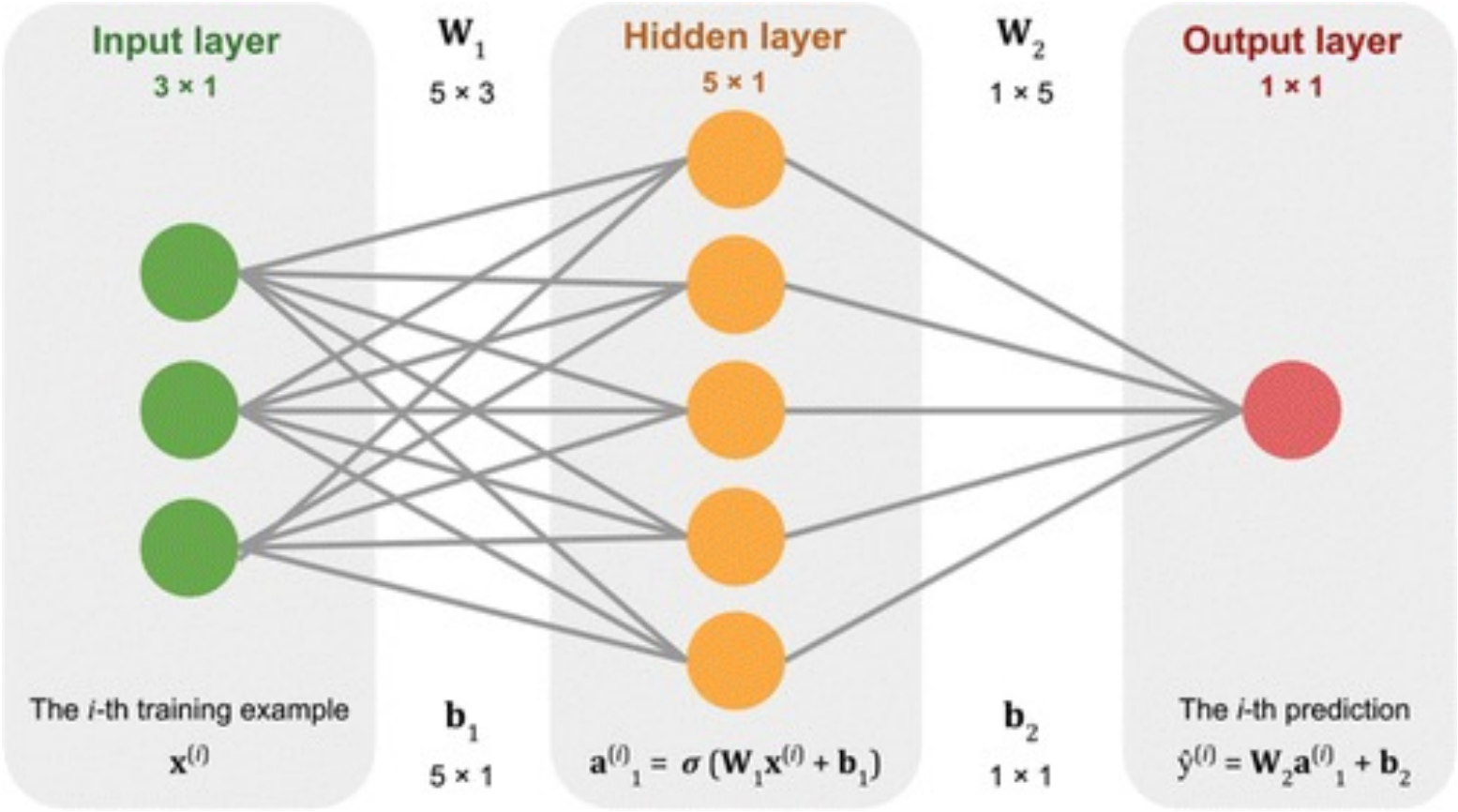
Yang, Liu, Dongkun Zhang, and George Em Karniadakis. "Physics-informed generative adversarial networks for stochastic differential equations." *SIAM Journal on Scientific Computing* 42.1 (2020): A292-A317.

Part1. Physics-Informed Neural Network

What is a PDE solver using deep learning?



Deep neural network



Goal: Find neural network parameters $\theta = \{W_i, b_i\}$

https://wiki.seg.org/wiki/Neural_networks

Deep learning approach to PDEs

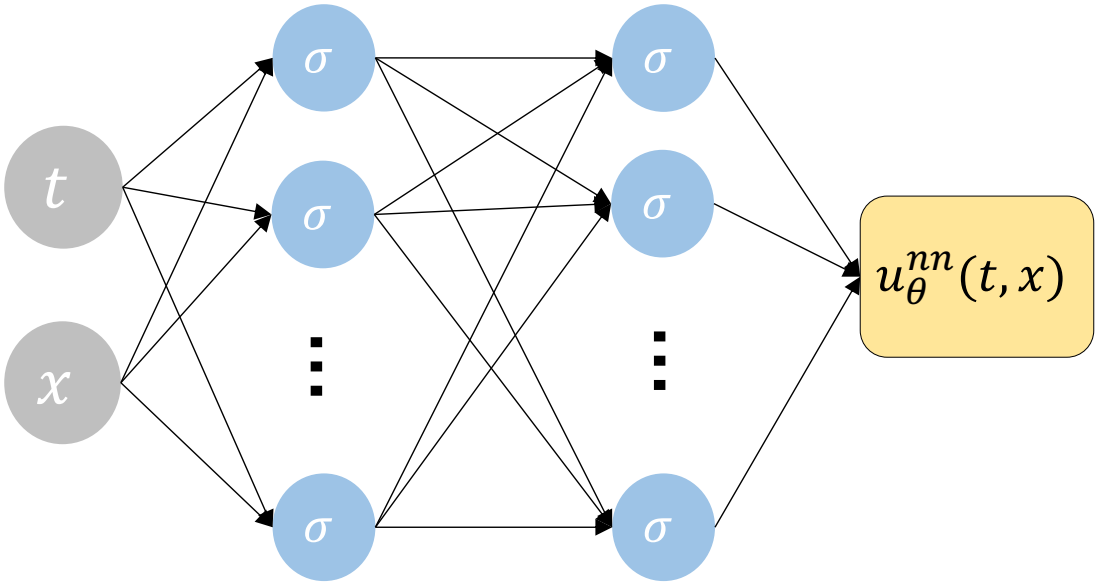
- **Using neural networks directly to parametrize the solution to PDEs.**
- **Solve one instance of PDE at a time.**
- **Models**
  - Physics Informed Neural Network (PINN)
  - Deep Ritz Method (DRM)

Find $\boldsymbol{u(t,x)}$ satisfying

$$\mathcal{L}_{PDE} = f(u, u_t, u_x, u_{xx}, \dots) = 0$$

$$\mathcal{L}_{IC} = u(0,x) - g(x) = 0$$

$$\mathcal{L}_{BC} = u|_{\partial\Omega} - h(t,x)|_{\partial\Omega} = 0$$



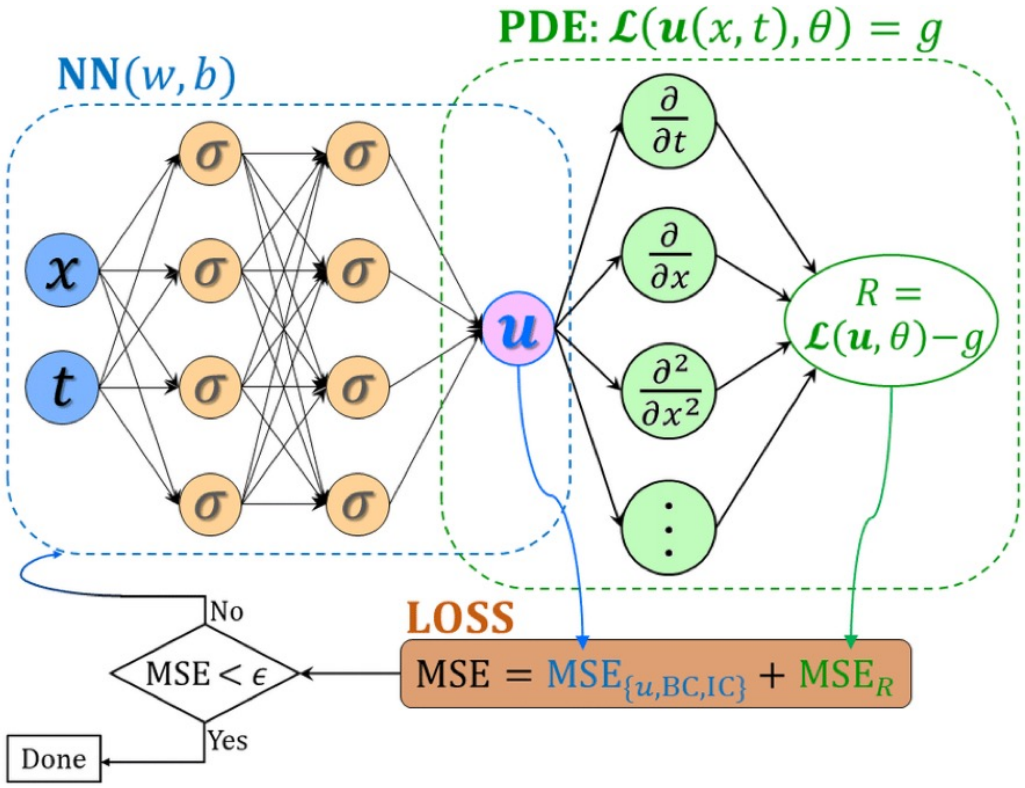
Physics-informed neural network (PINN)

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computation al physics*, *378*, 686-707.

Automatic differentiation and the back-propagation algorithm

"Here we revisit them using modern computational tools, and apply them to more challenging dynamic problems described by time-dependent nonlinear partial differential equations."

[Schematic of a PINN]



Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, *63*(1), 208-228

Physics-informed neural network (PINN) - Forward problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + uu_x - (0.01/\pi)u_{xx} = 0$$
$$\text{IC}: u(0, x) = -\sin(\pi x)$$
$$\text{BC}: u(t, -1) = u(t, 1) = 0$$

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

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

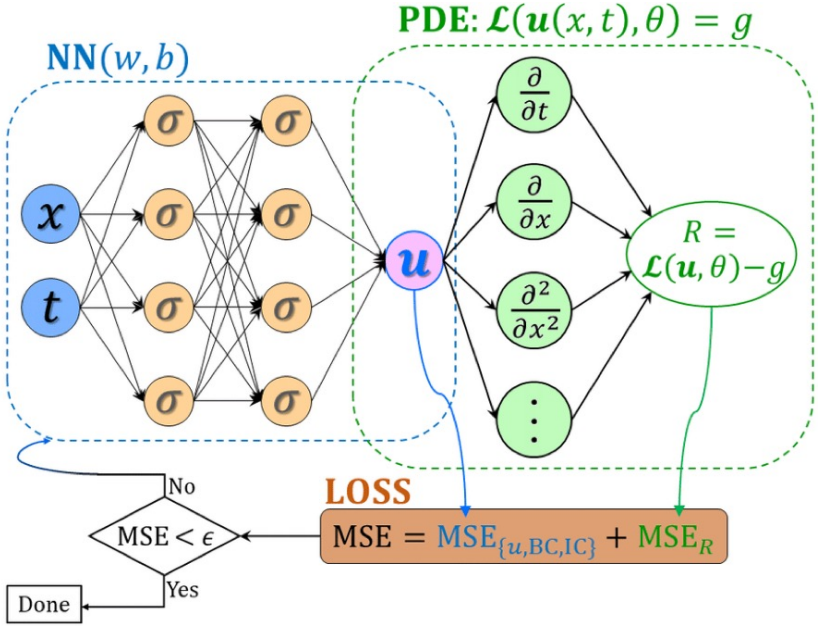
Physics-informed neural network (PINN) - Forward problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + uu_x - (0.01/\pi)u_{xx} = 0$$
$$\text{IC}: u(0,x) = -\sin(\pi x)$$
$$\text{BC}: u(t,-1) = u(t,1) = 0$$

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

Data:
$(t_i, x_i) \in [0,1]\times[-1,1]$
$(t_j, x_j) \in \{0\}\times[-1,1]$
$(t_k, x_k) \in [0,1]\times\{-1,1\}$



Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Forward problem

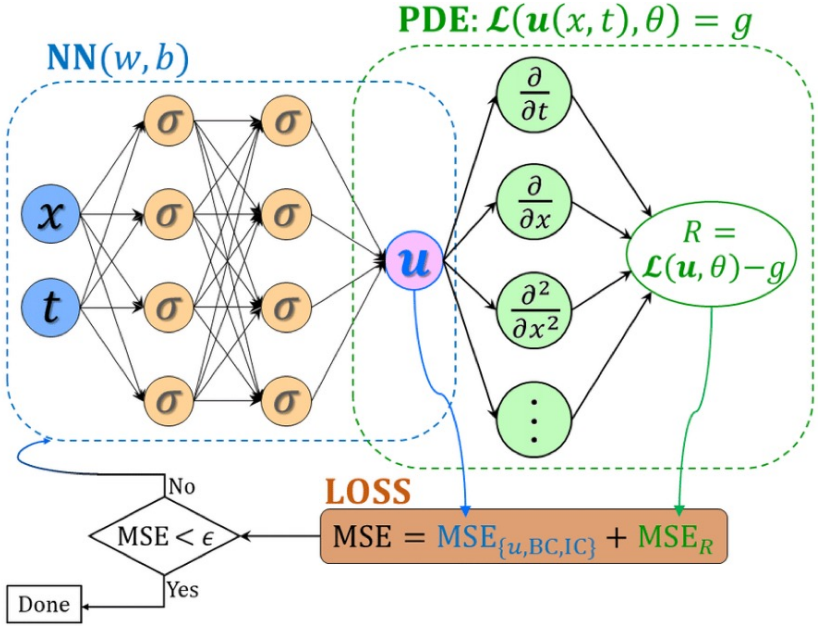
(e.g.) Burgers' equation

$$\text{Eq}: u_t + uu_x - (0.01/\pi)u_{xx} = 0$$
$$\text{IC}: u(0,x) = -\sin(\pi x)$$
$$\text{BC}: u(t,-1) = u(t,1) = 0$$
$$x \in [-1,1], t \in [0,1]$$

Loss function

$$\boldsymbol{L(\theta)} = \frac{1}{N_{Eq}}\sum_{i=1}^{N_{Eq}} \left|u_t(t^i,x^i) + u(t^i,x^i)u_x(t^i,x^i) - (0.01/\pi)u_{xx}(t^i,x^i)\right|^2 + \frac{1}{N_{IC}}\sum_{j=1}^{N_{IC}} \left|u(t^j,x^j) + \sin(\pi x^j)\right|^2 + \frac{1}{N_{BC}}\sum_{k=1}^{N_{BC}} \left|u(t^k,x^k)\right|^2$$

Data:
$(t_i, x_i) \in [0,1]\times[-1,1]$
$(t_j, x_j) \in \{0\}\times[-1,1]$
$(t_k, x_k) \in [0,1]\times\{-1,1\}$



Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Forward problem

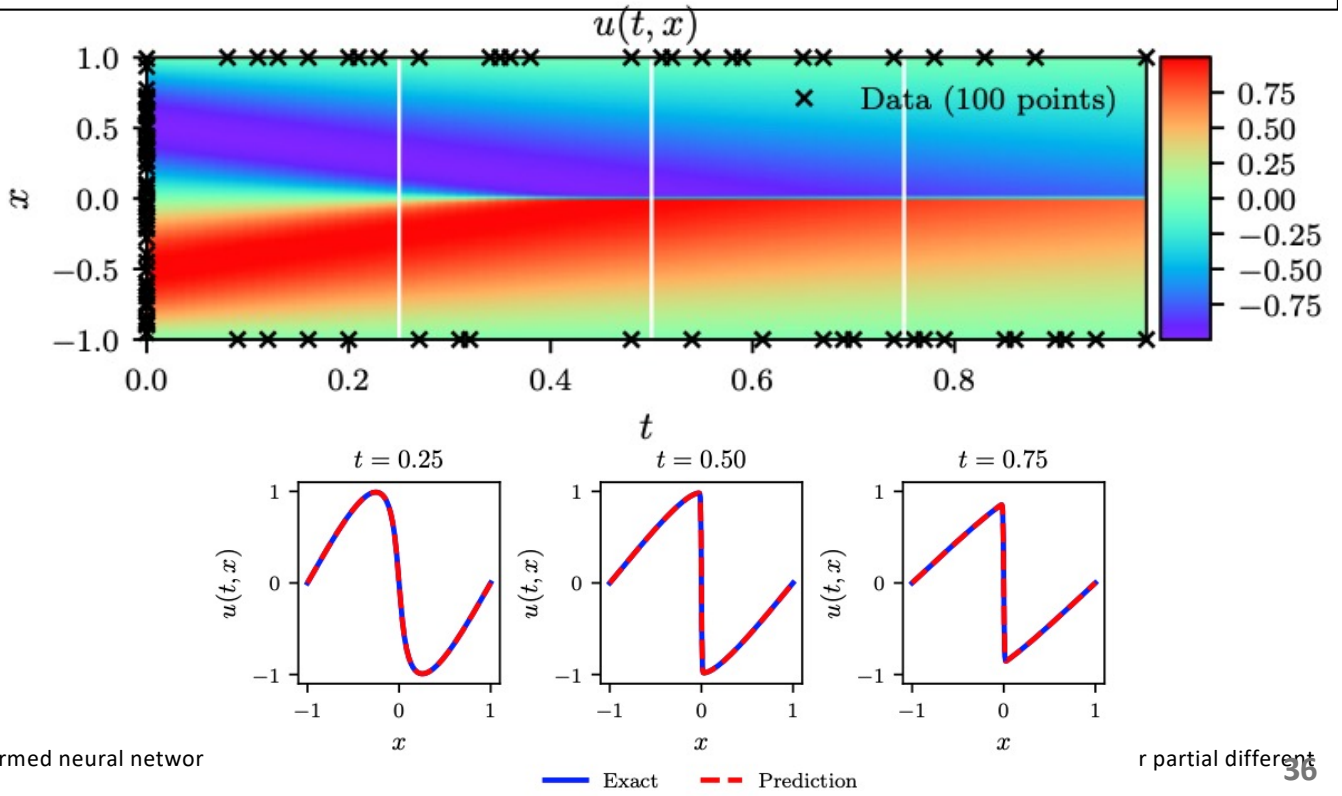
(e.g.) Burgers' equation

$$\text{Eq}: u_t + uu_x - (0.01/\pi)u_{xx} = 0$$
$$\text{IC}: u(0,x) = -\sin(\pi x)$$
$$\text{BC}: u(t,-1) = u(t,1) = 0$$
$$x \in [-1,1], t \in [0,1]$$

Loss function

$$\boldsymbol{L(\theta)} = \frac{1}{N_{Eq}}\sum_{i=1}^{N_{Eq}} \left|u_t(t^i,x^i) + u(t^i,x^i)u_x(t^i,x^i) - (0.01/\pi)u_{xx}(t^i,x^i)\right|^2 + \frac{1}{N_{IC}}\sum_{j=1}^{N_{IC}} \left|u(t^j,x^j) + \sin(\pi x^j)\right|^2 + \frac{1}{N_{BC}}\sum_{k=1}^{N_{BC}} \left|u(t^k,x^k)\right|^2$$

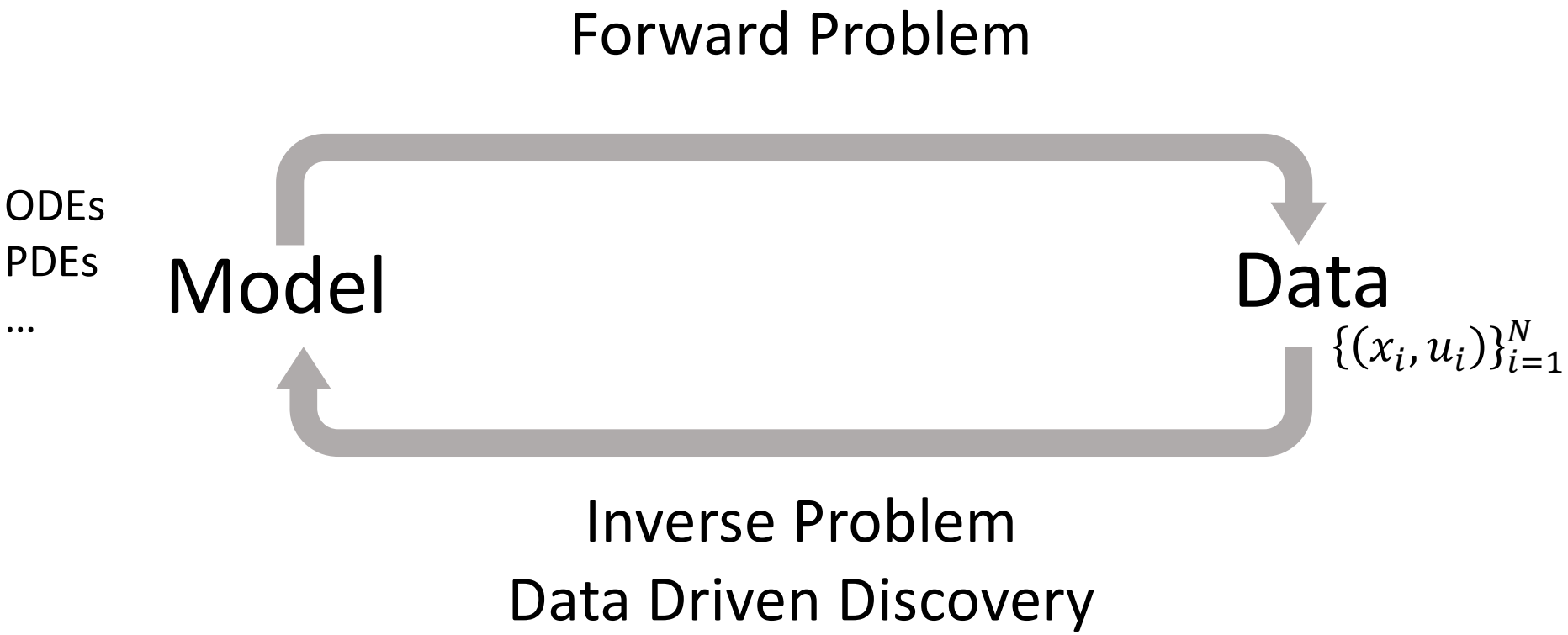
Data:
$(t_i, x_i) \in [0,1]\times[-1,1]$
$(t_j, x_j) \in \{0\}\times[-1,1]$
$(t_k, x_k) \in [0,1]\times\{-1,1\}$



Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networ
ial equations. *Journal of Computational physics*, *378*, 686-707.

r partial different

Physics-informed neural network (PINN) - Inverse problem???



Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

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

Forward problem

Eq : $u_t + uu_x - (0.01/\pi)u_{xx} = 0$
IC : $u(0, x) = -\sin(\pi x)$
BC : $u(t, -1) = u(t, 1) = 0$

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$
$$\text{IC}: u(0, x) = -\sin(\pi x)$$
$$\text{BC}: u(t, -1) = u(t, 1) = 0$$

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

Forward problem

$$\text{Eq}: u_t + u u_x - (0.01/\pi) u_{xx} = 0$$
$$\text{IC}: u(0, x) = -\sin(\pi x)$$
$$\text{BC}: u(t, -1) = u(t, 1) = 0$$

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$
$$\text{IC}: u(0, x) = -\sin(\pi x)$$
$$\text{BC}: u(t, -1) = u(t, 1) = 0$$

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

Data-driven discovery of PDE (Solving inverse problem for unknown parameter λ_1 and λ_2)

- We want to identify the unknown parameters by introducing additional trainable parameters.
- Assume a small amount of additional data $\{t_l, x_l, u_l\}_{l=1}^{N_{data}}$ is given.
- Update not only nn parameter θ but also the PDE parameters λ_1 and λ_2.

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$
$$\text{IC}: u(0, x) = -\sin(\pi x)$$
$$\text{BC}: u(t, -1) = u(t, 1) = 0$$

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

Data:

$(t_i, x_i) \in [0,1] \times [-1,1]$

$(t_j, x_j) \in \{0\} \times [-1,1]$

$(t_k, x_k) \in [0,1] \times \{-1,1\}$

$(t_l, x_l, u_l) \in [0,1] \times [-1,1] \times \mathbb{R}$

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$
$$\text{IC}: u(0,x) = -\sin(\pi x)$$
$$\text{BC}: u(t,-1) = u(t,1) = 0$$

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

Loss function

$$\boldsymbol{L(\theta, \lambda_1, \lambda_2)}$$
$$= \frac{1}{N_{Eq}} \sum_{i=1}^{N_{Eq}} \left| u_t(t^i, x^i) + \lambda_1 u(t^i, x^i) u_x(t^i, x^i) - \lambda_2 u_{xx}(t^i, x^i) \right|^2 + \frac{1}{N_{IC}} \sum_{j=1}^{N_{IC}} \left| u(t^j, x^j) + \sin(\pi x^j) \right|^2 + \frac{1}{N_{BC}} \sum_{k=1}^{N_{BC}} \left| u(t^k, x^k) \right|^2$$
$$+ \frac{1}{N_{data}} \sum_{l=1}^{N_{data}} \left| u(t^l, x^l) - u_l \right|^2$$

Data:

$(t_i, x_i) \in [0,1] \times [-1,1]$

$(t_j, x_j) \in \{0\} \times [-1,1]$

$(t_k, x_k) \in [0,1] \times \{-1,1\}$

$(t_l, x_l, u_l) \in [0,1] \times [-1,1] \times \mathbb{R}$

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$
$$\text{IC}: u(0,x) = -\sin(\pi x)$$
$$\text{BC}: u(t,-1) = u(t,1) = 0$$

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

Loss function

$$\boldsymbol{L(\theta, \lambda_1, \lambda_2)}$$
$$= \frac{1}{N_{Eq}} \sum_{i=1}^{N_{Eq}} \left| u_t(t^i, x^i) + \lambda_1 u(t^i, x^i) u_x(t^i, x^i) - \lambda_2 u_{xx}(t^i, x^i) \right|^2 + \frac{1}{N_{IC}} \sum_{j=1}^{N_{IC}} \left| u(t^j, x^j) + \sin(\pi x^j) \right|^2 + \frac{1}{N_{BC}} \sum_{k=1}^{N_{BC}} \left| u(t^k, x^k) \right|^2$$

$$+ \frac{1}{N_{data}} \sum_{l=1}^{N_{data}} \left| u(t^l, x^l) - u_l \right|^2$$

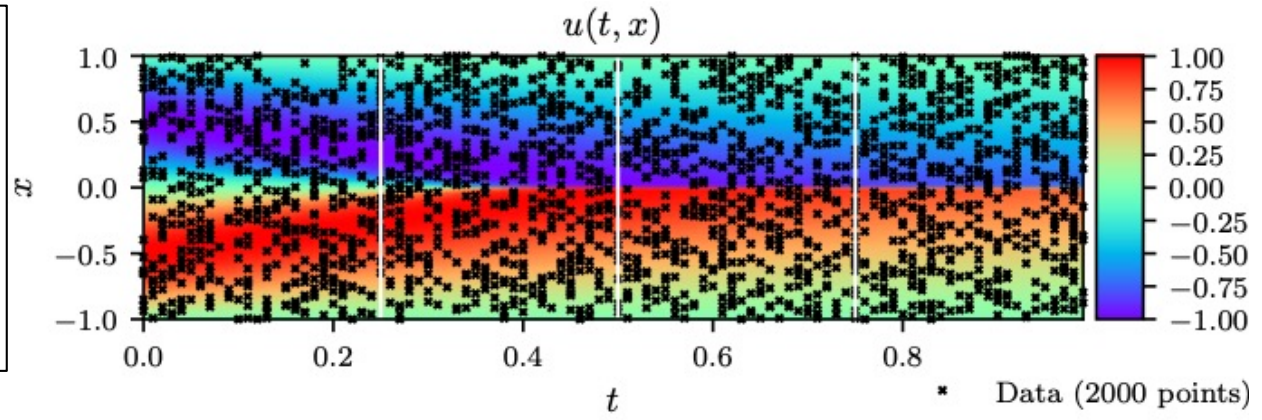
Data:

$(t_i, x_i) \in [0,1] \times [-1,1]$

$(t_j, x_j) \in \{0\} \times [-1,1]$

$(t_k, x_k) \in [0,1] \times \{-1,1\}$

$(t_l, x_l, u_l) \in [0,1] \times [-1,1] \times \mathbb{R}$



Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Physics-informed neural network (PINN) - Inverse problem

(e.g.) Burgers' equation

$$\text{Eq}: u_t + \lambda_1 u u_x - \lambda_2 u_{xx} = 0$$
$$\text{IC}: u(0,x) = -\sin(\pi x)$$
$$\text{BC}: u(t,-1) = u(t,1) = 0$$
$$x \in [-1,1], t \in [0,1]$$

Loss function

$$\boldsymbol{L(\theta, \lambda_1, \lambda_2)}$$
$$= \frac{1}{N_{Eq}} \sum_{i=1}^{N_{Eq}} \left| u_t(t^i, x^i) + \lambda_1 u(t^i, x^i) u_x(t^i, x^i) - \lambda_2 u_{xx}(t^i, x^i) \right|^2 + \frac{1}{N_{IC}} \sum_{j=1}^{N_{IC}} \left| u(t^j, x^j) + \sin(\pi x^j) \right|^2 + \frac{1}{N_{BC}} \sum_{k=1}^{N_{BC}} \left| u(t^k, x^k) \right|^2$$
$$+ \frac{1}{N_{data}} \sum_{l=1}^{N_{data}} \left| u(t^l, x^l) - u_l \right|^2$$

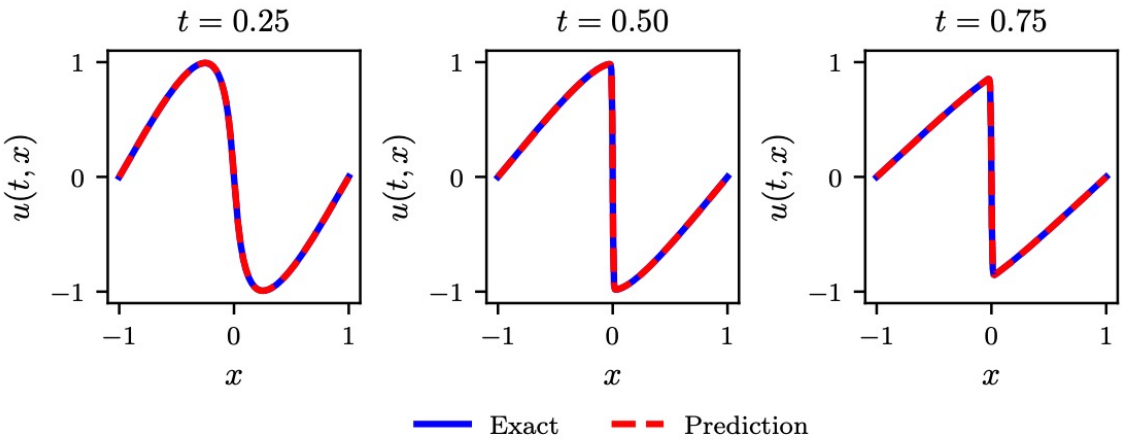
Data:

$(t_i, x_i) \in [0,1] \times [-1,1]$

$(t_j, x_j) \in \{0\} \times [-1,1]$

$(t_k, x_k) \in [0,1] \times \{-1,1\}$

$(t_l, x_l, u_l) \in [0,1] \times [-1,1] \times \mathbb{R}$



Correct PDE	$u_t + uu_x - 0.0031831u_{xx} = 0$
Identified PDE (clean data)	$u_t + 0.99915uu_x - 0.0031794u_{xx} = 0$
Identified PDE (1% noise)	$u_t + 1.00042uu_x - 0.0032098u_{xx} = 0$

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686-707.

Benefits and drawbacks of PINN

Dockhorn, T. (2019). A discussion on solving partial differential equations using neural networks. *arXiv preprint arXiv:1904.07200*
.

Table 6: Benefits and drawbacks of solving partial differential equations with neural networks.

Benefits	**Drawbacks**
Expressive power: Already small neural networks are able to approximate the solution of partial differential equations well.	*Convergence:* Most probably due to optimization issues, we could not empirically show that errors decrease (with a certain convergence rate) with increasing neural network size; showing theoretical convergence seems to be even more difficult.
Ease of implementation: Our algorithm is straightforward and can be easily implemented using backpropagation.	*Run time:* At least when the BFGS optimizer is used, our simulations seem to be considerably slower than classical numerical methods. Using the proposed algorithm in combination with a first order optimization method, e.g. Adam [16], is subject to future work.
Arbitrary domains (in higher dimensions): The algorithm is based on drawing random points from a domain, which can be readily extended to arbitrary domains; no triangulation of the domain is needed.	*Scalability:* At the moment, it is unclear how well the algorithm will scale to more difficult problems, e.g., three-dimensional turbulent flows.
Freedom of approximation spaces: For some classical methods and systems of PDEs, we have certain restrictions on the function spaces of the solutions, e.g., the inf–sup condition [15] needs to be satisfied for the Navier–Stokes equations when using the finite element method.	*Randomness:* Different random initializations lead to different results of the algorithm whereas (most) classical numerical methods are deterministic. Further investigation is necessary to better exploit this nondeterminism and to eventually turn it into an advantage.
Sensor data: We can easily incorporate (noisy) information of sensors by adding a term to the loss function.	

Practical Coding!

Let's implement

Physics-Informed Neural Network!

Supervised Learning - Regression

(Parametric) Regression task consists of the following procedures :

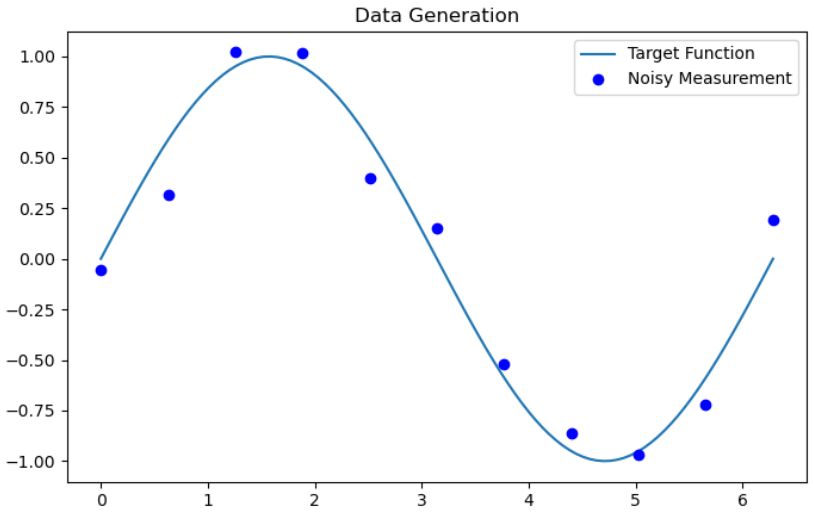
0. Given dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$

1. Choose a hypothesis class $H = \{f \mid f_w : X \to Y, \dots\}$ parameterized by $w \in R^k$, e.g., $f(x; w) = w^T x$

2. Choose a proper loss functional $L : H \to R$, e.g., $L(f_w) = \sum_{i=1}^{N} (f(x_i; w) - y_i)^2$

3. Minimize the loss function with respect to the parameter w

$$y_i = \sin x_i + \epsilon, \qquad \epsilon \sim N(0, 0.1^2)$$



Supervised Learning - Regression

(Parametric) Regression task consists of the following procedures :

0. Given dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$

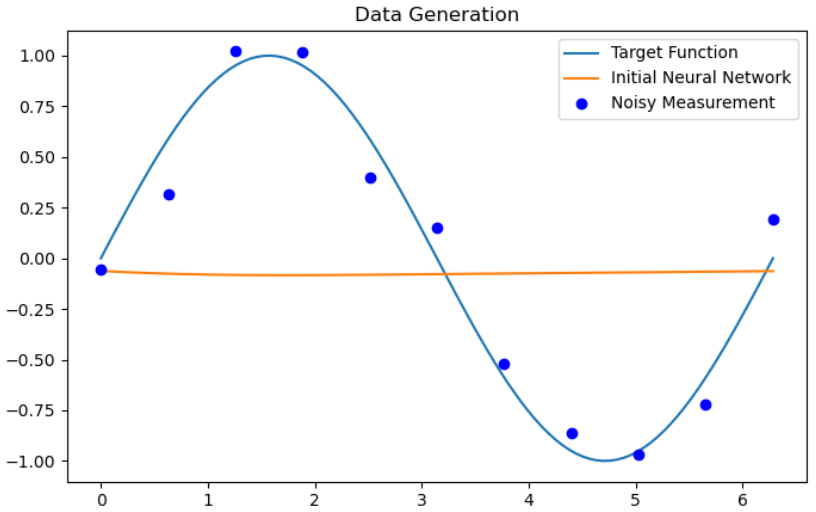
1. Choose a hypothesis class $H = \{f \mid f_w : X \to Y, \dots\}$ parameterized by $w \in R^k$, e.g., $f(x; w) = w^T x$

2. Choose a proper loss functional $L : H \to R$, e.g., $L(f_w) = \sum_{i=1}^{N} (f(x_i; w) - y_i)^2$

3. Minimize the loss function with respect to the parameter w

Initialize a neural network $f(x; w)$,

$$w \sim UNIF(-\frac{1}{n}, \frac{1}{n})$$



Supervised Learning - Regression

(Parametric) Regression task consists of the following procedures :

0. Given dataset $D = \{(x_i, y_i)\}_{i=1}^N$

1. Choose a hypothesis class $H = \{f \mid f_w : X \to Y, \dots\}$ parameterized by $w \in R^k$, e.g., $f(x; w) = w^T x$

2. Choose a proper loss functional $L : H \to R$, e.g., $L(f_w) = \sum_{i=1}^N (f(x_i; w) - y_i)^2$

3. Minimize the loss function with respect to the parameter w

$$y_i = f(x_i; w) + \epsilon, \qquad \epsilon \sim N(0, \sigma^2), iid$$

$$\Rightarrow y_i = f(x_i; w) + \epsilon \sim N(f(x_i; w), \sigma^2)$$

$$\Rightarrow P(Y|w) = \Pi_{i=1}^N N(y_i | f(x_i; w), \sigma^2)$$

$$\Rightarrow \log P(Y|w) = \sum_{i=1}^N \log N(y_i | f(x_i; w), \sigma^2)$$

$$\Rightarrow \log P(Y|w) = C - \frac{1}{2\sigma^2} \sum_{i=1}^N \left(y_i - f(x_i; w)\right)^2$$

$$w^{MLE} = \text{argmax}_w P(Y|w)$$

$$\Rightarrow w^{MLE} = \text{argmin}_w \sum_{i=1}^N \left(y_i - f(x_i; w)\right)^2$$

Supervised Learning - Regression

(Parametric) Regression task consists of the following procedures :

0. Given dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$

1. Choose a hypothesis class $H = \{f \mid f_w : X \to Y, \dots\}$ parameterized by $w \in R^k$, e.g., $f(x; w) = w^T x$

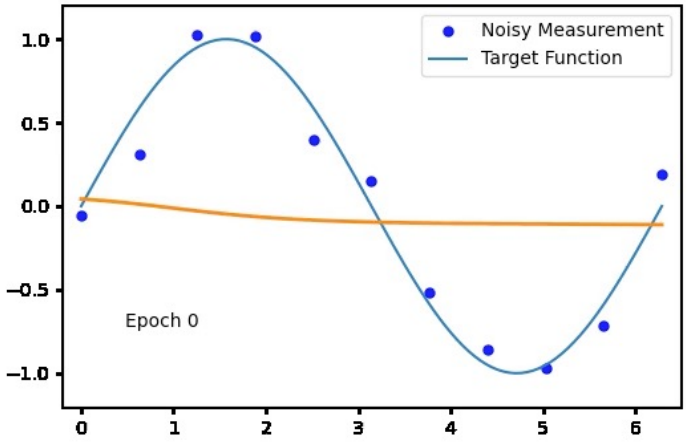
2. Choose a proper loss functional $L : H \to R$, e.g., $L(f_w) = \sum_{i=1}^{N} (f(x_i; w) - y_i)^2$

3. Minimize the loss function with respect to the parameter w

Gradient-based optimization algorithms

Gradient descent : The simplest one

$$w^{t+1} = w^t - \eta * \nabla_w \sum_{i=1}^{N} \left(y_i - f(x_i; w^t)\right)^2$$



Basic Settings

Initial Boundary Value Problem (PDE)

We will solve the Burgers equation :

$$\partial_t u + u\partial_x u = \nu \partial_{xx} u, \text{ for } (t,x) \in [0,1] \times [-1,1], \tag{4}$$

with the Dirichlet boundary conditions

$$u(t,-1) = u(t,1) = 0, \text{ for } t \in [0,1], \tag{5}$$

and an initial condition

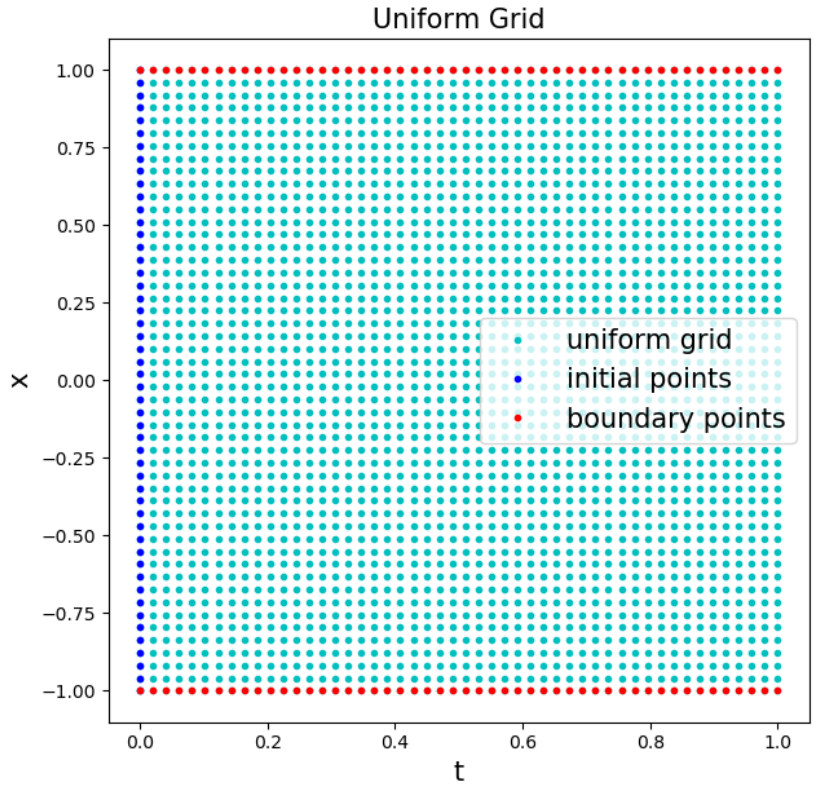
$$u(0,x) = -\sin(\pi x). \tag{6}$$

Data Generation

In [37]:

```
Nt = 50                                                    # Number of samples
Nx = 50                                                    # Number of samples
t = torch.linspace(0, 1, Nt)                               # Input data for t (N x 1)
x = torch.linspace(-1, 1, Nx)                              # Input data for x (N x 1)

tx = torch.meshgrid(t,x)
tx_grid = torch.cat([tx[0].reshape(-1,1), tx[1].reshape(-1,1)],dim=1)

x_init = tx_grid[tx_grid[:,0]==0]
u_init = -torch.sin(np.pi*x_init[:,1]).view(-1,1)
x_bdry = tx_grid[torch.logical_or((tx_grid[:,1]==1), (tx_grid[:,1]==-1))]
u_bdry = torch.zeros_like(x_bdry[:,0].view(-1,1))
```

Data Generation



Neural Network

In [46]:

```
# Build a neural network

class model(nn.Module) :
    def __init__(self, hidden_dims) :                    # Hidden_dims : [h1, h2, h3, ..., hn]
        super(model, self).__init__()

        self.layers = []
        for i in range(len(hidden_dims)-1) :
            self.layers.append(nn.Linear(hidden_dims[i], hidden_dims[i+1])) # hidden layers
        self.layers = nn.ModuleList(self.layers)

        for layer in self.layers :                       # Weight initialization
            nn.init.xavier_uniform_(layer.weight)        # Also known as Glorot initialization

        self.act = nn.Tanh()                             # Nonlinear activation function

    def forward(self, x) :
        for layer in self.layers[:-1] :
            x = self.act(layer(x))
        x = self.layers[-1](x)
        return x
```

Loss Function, Optimizer

In [47]:

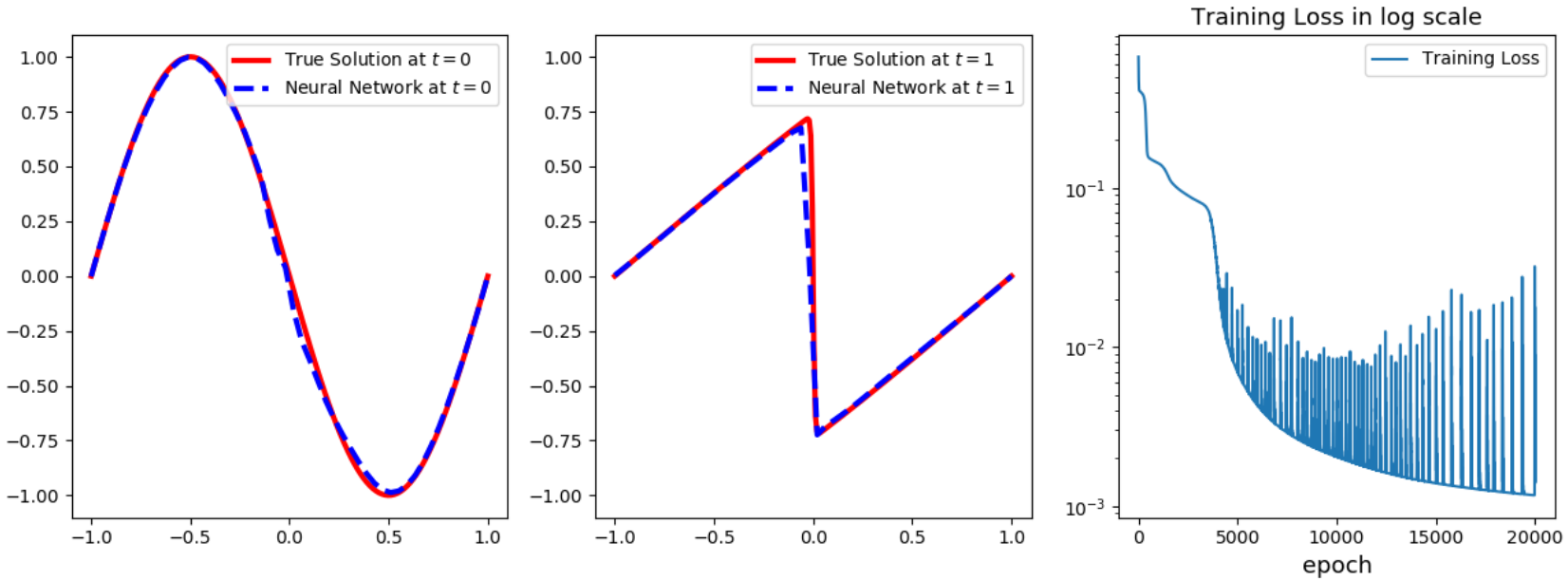
```
# Prepare for training

network = model(hidden_dims=[2,64,64,64,64,1]).to(device)       # Pass the network to GPU
tx_grid = tx_grid.to(device).requires_grad_(True)               # Pass data to GPU
x_init = x_init.to(device)                                      # Pass data to GPU
u_init = u_init.to(device)                                      # Pass data to GPU
x_bdry = x_bdry.to(device)                                      # Pass data to GPU
u_bdry = u_bdry.to(device)                                      # Pass data to GPU

loss_f = nn.MSELoss()                                           # Mean Square Error loss function
optimizer = optim.Adam(network.parameters(), lr=1e-4)           # Adam optimizer
EPOCHS = 20000                                                  # Number of Training Iterations
```

Training

In [49]:

```
# Train
loss_list = []
network.train()

for i in range(1, EPOCHS+1) :
    optimizer.zero_grad()
    output = network(tx_grid)
    output_init = network(x_init)
    output_bdry = network(x_bdry)

    grad_output = derivative(output, tx_grid)  # Wnabla_{t,x} u
    output_t = grad_output[:,0].view(-1,1)
    output_x = grad_output[:,1].view(-1,1)
    output_xx = derivative(grad_output[:,1], tx_grid)[:,1].view(-1,1)

    loss_init = loss_f(output_init, u_init)
    loss_bdry = loss_f(output_bdry, u_bdry)
    loss_ge = loss_f(output_t + output*output_x - (0.01/np.pi)*output_xx, torch.zeros_like(output))

    loss = loss_ge + loss_init + loss_bdry
    loss.backward()
    optimizer.step()

    loss_list.append(loss.item())
    if not i % 100 :
        print('EPOCH : %6d/%6d | Loss_ge : %8.7f | Loss_init : %8.7f | Loss_bdry : %8.7f' W
              %(i, EPOCHS, loss_ge.item(), loss_init.item(), loss_bdry.item()))
        #clear_output(wait=True)
print('Training Finished.')
```

Results


True Solution at $t = 0$
Neural Network at $t = 0$
True Solution at $t = 1$
Neural Network at $t = 1$
Training Loss in log scale
Training Loss
epoch

Basic Settings

Inverse problem for the Lorenz system

We will solve the Lorentz system :

$$\frac{dx}{dt} = \sigma(y - x), \quad \frac{dy}{dt} = x(\rho - z) - y, \quad \frac{dz}{dt} = xy - \beta z, \quad \text{for } t \in [0, 3] \tag{7}$$

with initial conditions

$$x(0) = -8, \quad y(0) = 7, \quad z(0) = 27. \tag{8}$$

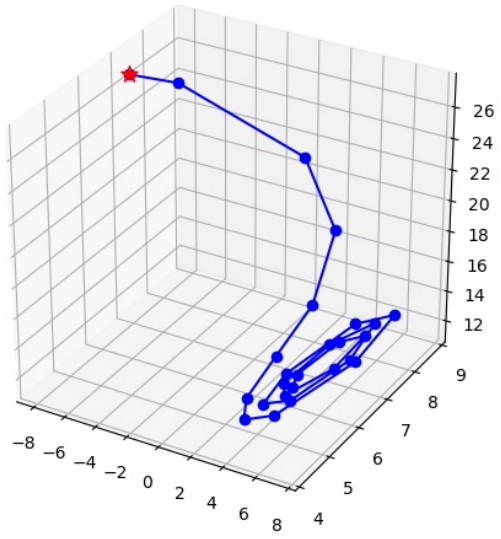
The parameters $\sigma = 10, \rho = 15, \beta = \frac{8}{3}$ will be identified from observations of the system.

Data Generation

In [19]:

```
N = 100                                          # Number of samples
t = torch.linspace(0, 3, N).view(-1,1)           # Input data (N x 1)
t_init = torch.zeros([1,1])                      # Input data the initial conditions (1 x 1)
y_init = torch.FloatTensor([[-8, 7, 27]])        # Target data the initial conditions (1 x 2)
```

Observations :



Neural Network

In [23]:

```
# Build a neural network

class model(nn.Module) :
    def __init__(self, hidden_dims) :                    # Hidden_dims : [h1, h2, h3, ..., hn]
        super(model, self).__init__()

        self.layers = []
        for i in range(len(hidden_dims)-1) :
            self.layers.append(nn.Linear(hidden_dims[i], hidden_dims[i+1])) # hidden layers
        self.layers = nn.ModuleList(self.layers)

        for layer in self.layers :                       # Weight initialization
            nn.init.xavier_uniform_(layer.weight)        # Also known as Glorot initialization

        self.act = nn.Tanh()                             # Nonlinear activation function

    def forward(self, x) :
        for layer in self.layers[:-1] :
            x = self.act(layer(x))
        x = self.layers[-1](x)
        return x
```

Loss Function, Optimizer

In [24]:

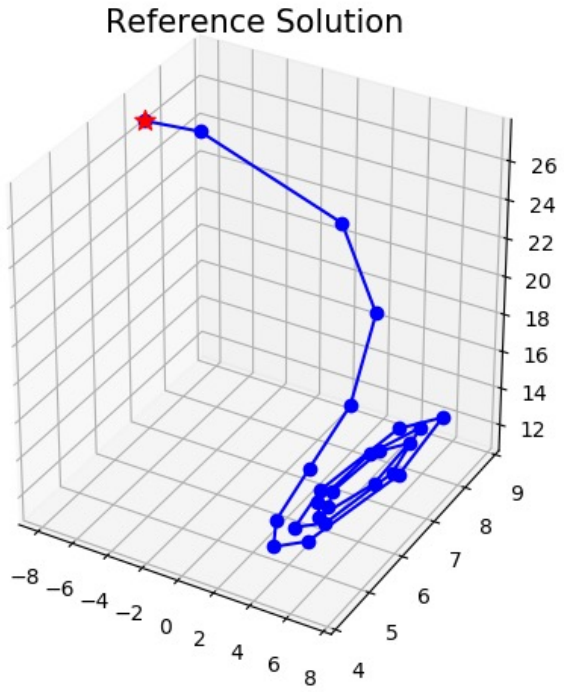
```
# Prepare for training

network = model(hidden_dims=[1,64,64,64,3]).to(device) # Pass the network to GPU
t = t.to(device).requires_grad_(True)                  # Pass data to GPU
t_init = t_init.to(device)                             # Pass data to GPU
y_init = y_init.to(device)                             # Pass data to GPU
t_obs = t_obs.to(device)                               # Pass data to GPU
y_obs = y_obs.to(device)                               # Pass data to GPU

sigma = torch.ones(1).to(device).requires_grad_(True)          # Parameter approximator
rho = torch.ones(1).to(device).requires_grad_(True)            # Parameter approximator
beta = torch.ones(1).to(device).requires_grad_(True)           # Parameter approximator

loss_f = nn.MSELoss()                                          # Mean Square Error loss function
optimizer = optim.Adam([{'params': network.parameters()},₩
                        {'params': sigma, 'lr':1e-1}, {'params': rho, 'lr':1e-1},₩
                        {'params': beta, 'lr':1e-1}], lr=1e-3)          # Adam optimizer
EPOCHS = 100000                                                 # Number of Training Iterations
```

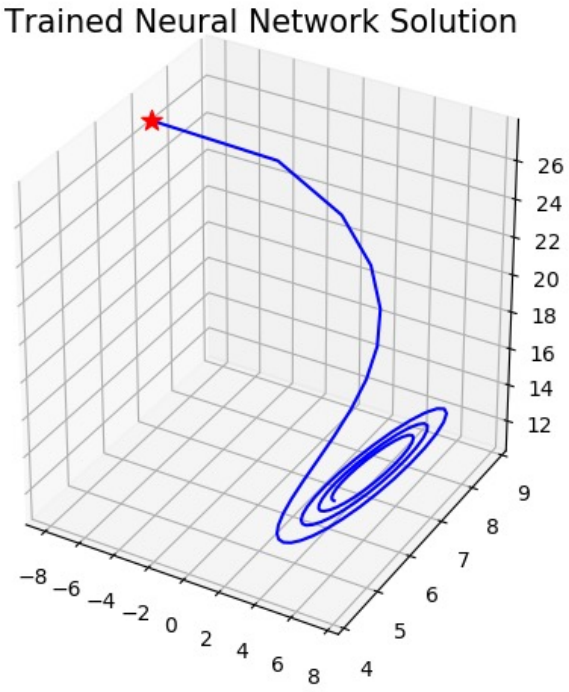
Training

In [26]:

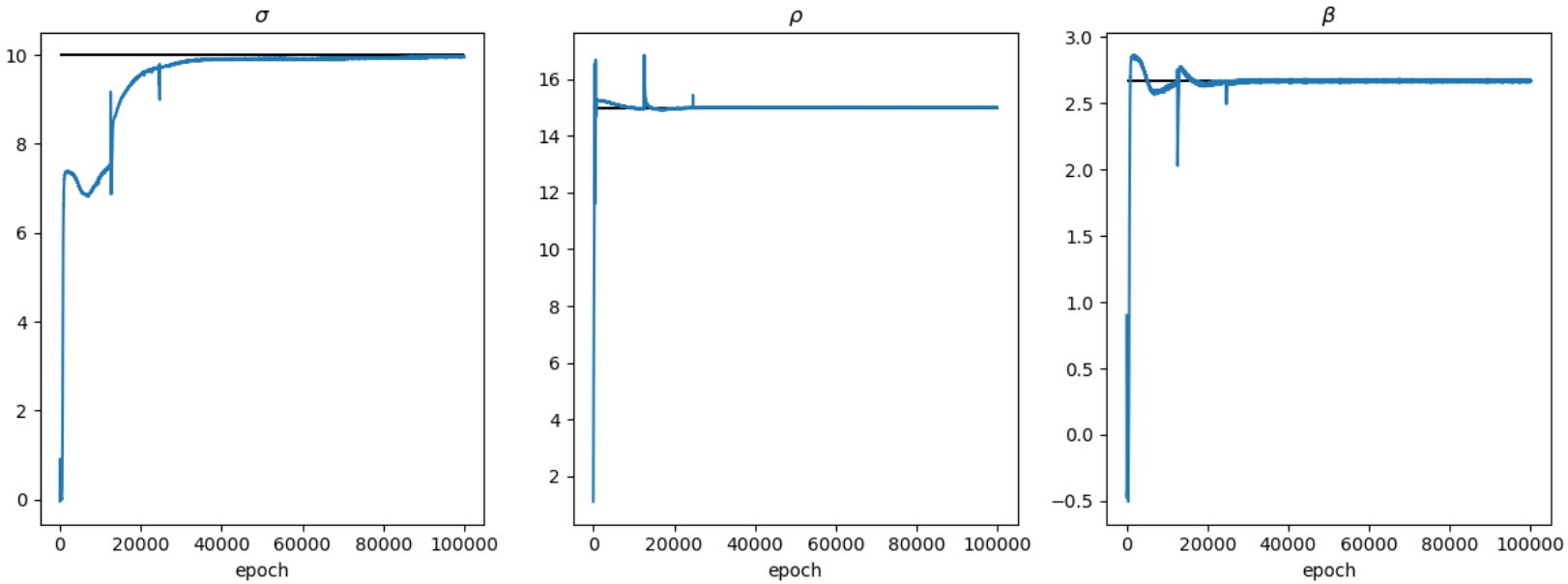
```
# Train
loss_list = []
sigma_list, rho_list, beta_list = [], [], []
network.train()

for i in range(1, EPOCHS+1) :
    optimizer.zero_grad()
    output = network(t)
    output_init = network(t_init)
    output_obs = network(t_obs)

    dy1 = derivative(output[:,0], t).view(-1)
    dy2 = derivative(output[:,1], t).view(-1)
    dy3 = derivative(output[:,2], t).view(-1)

    loss_init = loss_f(output_init, y_init)
    loss_obs = loss_f(output_obs, y_obs)
    loss_ge = loss_f(dy1 - sigma*(output[:,1]-output[:,0]), torch.zeros_like(dy1)) +₩
              loss_f(dy2 - output[:,0]*(rho-output[:,2]) + output[:,1], torch.zeros_like(dy2))+₩
              loss_f(dy3 - output[:,0]*output[:,1] + beta*output[:,2], torch.zeros_like(dy3))

    loss = loss_ge + loss_init + loss_obs
    loss.backward()
    optimizer.step()
```

Results


Reference Solution
26
24
22
20
18
16
14
12
−8 −6 −4 −2 0 2 4 6 8
4 5 6 7 8 9


Trained Neural Network Solution
26
24
22
20
18
16
14
12
−8 −6 −4 −2 0 2 4 6 8
4 5 6 7 8 9

Results


σ
10
8
6
4
2
0
0
20000
40000
60000
80000
100000
epoch
ρ
16
14
12
10
8
6
4
2
0
20000
40000
60000
80000
100000
epoch
β
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
20000
40000
60000
80000
100000
epoch

Part2. Neural Operator

Revisit PINN

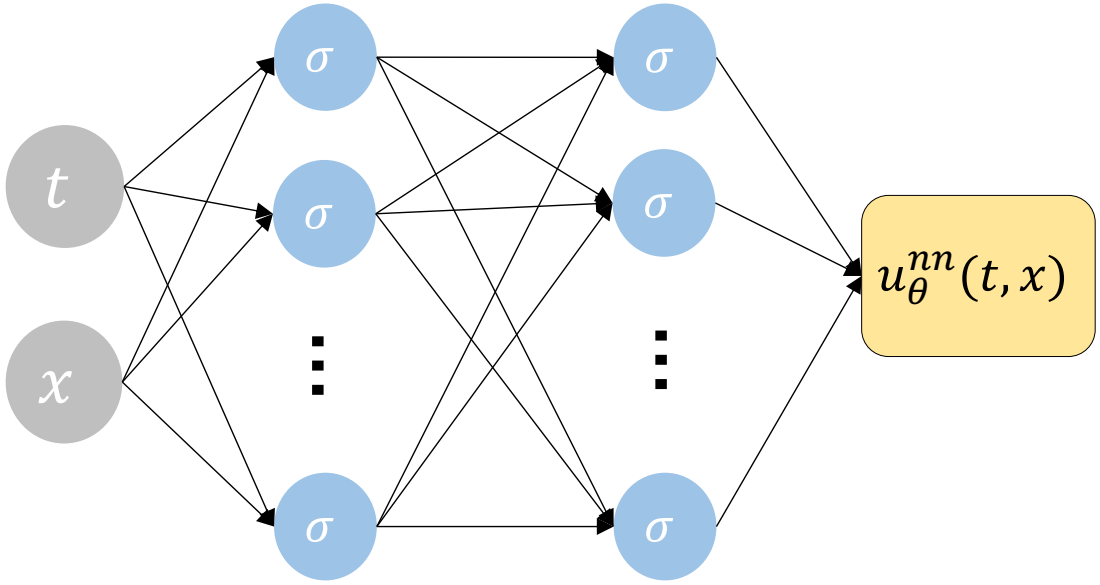
- Solving **only one instance of PDE** at a time? ? ?

Find $\boldsymbol{u(t,x)}$ satisfying

$$\mathcal{L}_{PDE} = f(u, u_t, u_x, u_{xx}, \dots) = 0$$

$$\mathcal{L}_{IC} = u(0,x) - g(x) = 0$$

$$\mathcal{L}_{BC} = u|_{\partial\Omega} - h(t,x)|_{\partial\Omega} = 0$$



Neural Operator

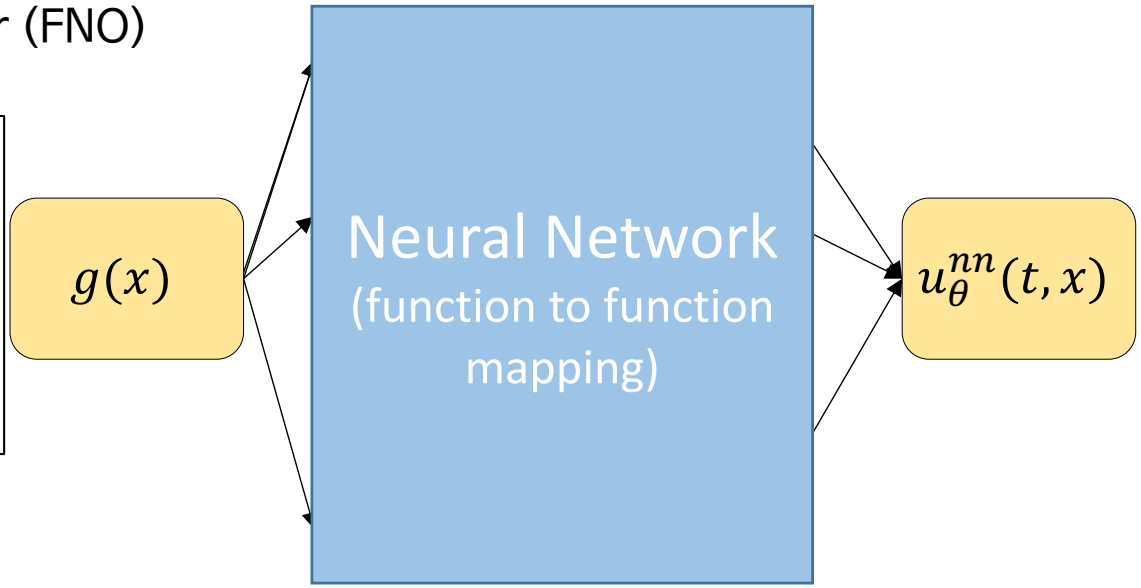
- **New Approach: Neural operator**
  - Learning a mapping from the parameters (e.g. external force, initial, and boundary conditions) of the PDEs to the corresponding solution.
  - Learning a family of PDEs from data.
  - Models
    - Deep Operator Network (DeepONet)
    - Fourier Neural Operator (FNO)

Find a map $\mathcal{G}: g(x) \mapsto u(t,x)$ satisfying

$$\mathcal{L}_{PDE} = f(u, u_t, u_x, u_{xx}, \dots) = 0$$

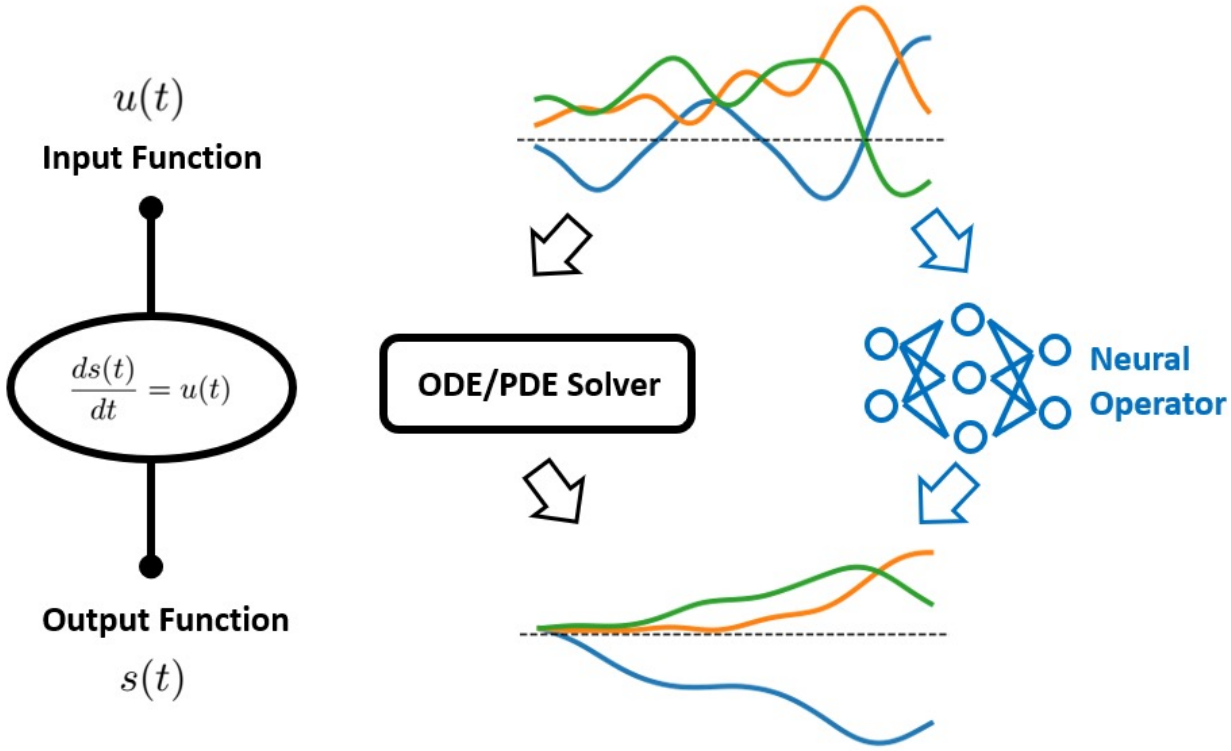
$$\mathcal{L}_{IC} = u(0,x) - g(x) = 0$$

$$\mathcal{L}_{BC} = u|_{\partial\Omega} - h(t,x)|_{\partial\Omega}$$



Neural Operator

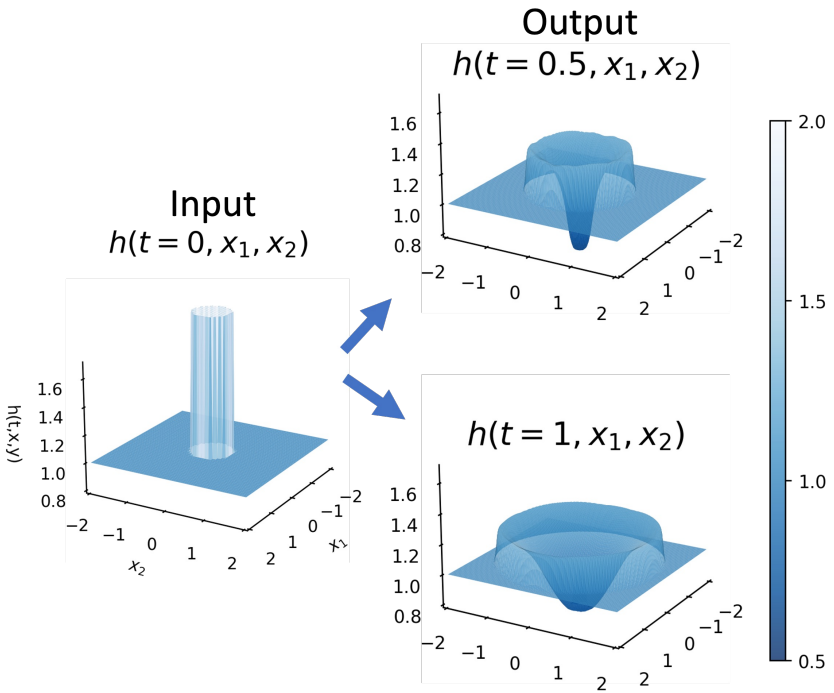
- **Operator learning** for mapping between infinite-dimensional function spaces has been recently studied.
- One of the common problems is learn a **mapping from the parameters** (e.g. external force, initial, and boundary conditions) of the PDEs **to the corresponding solution.**



https://towardsdatascience.com/operator-learning-via-physics-informed-deeponet-lets-implement-it-from-scratch-6659f3179887

Neural Operator

- **Operator learning** for mapping between infinite-dimensional function spaces has been recently studied.
- One of the common problems is learn a **mapping from the parameters** (e.g. external force, initial, and boundary conditions) of the PDEs **to the corresponding solution.**
- For example, in a dam break scenario, it is important to predict the fluid flow over time according to a random initial height of the fluid.

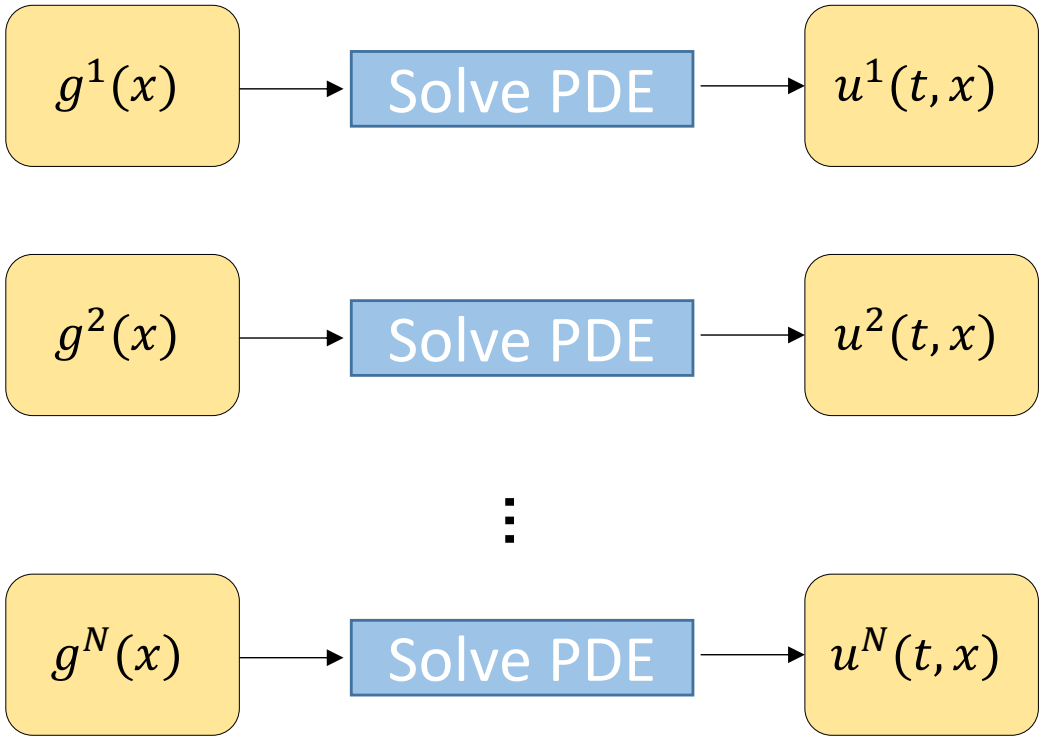


Neural Operator

- Traditional numerical methods had to **repeat** similar calculations for each input function to derive its output function.

Find a map $\mathcal{G}: g(x) \mapsto u(t,x)$ satisfying

$$\mathcal{L}_{PDE} = f(u, u_t, u_x, u_{xx}, \dots) = 0$$
$$\mathcal{L}_{IC} = u(0,x) - g(x) = 0$$
$$\mathcal{L}_{BC} = u|_{\partial\Omega} - h(t,x)|_{\partial\Omega}$$

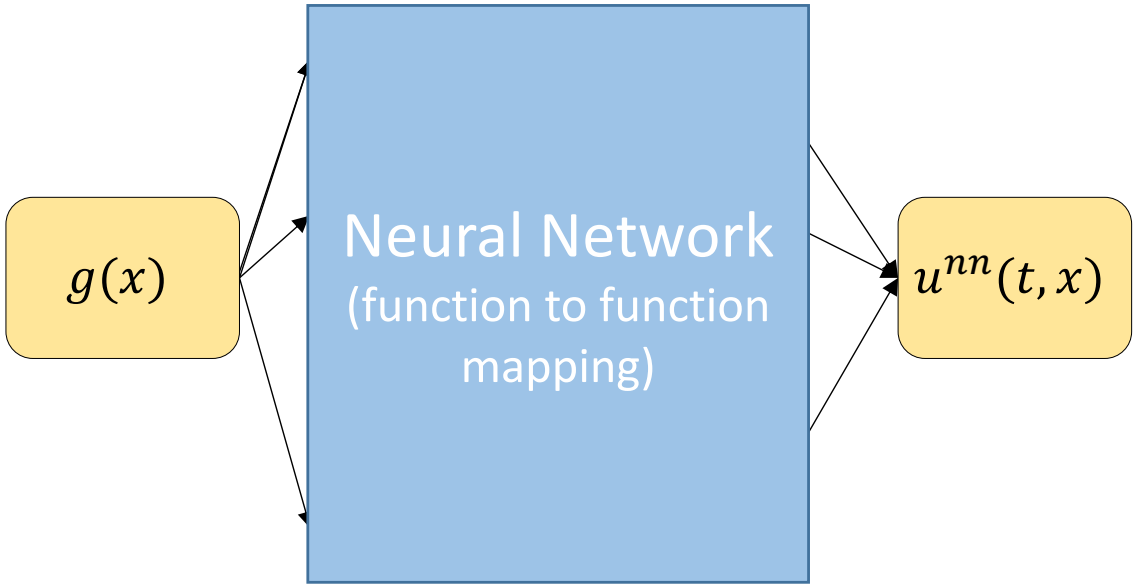


Neural Operator

- Traditional numerical methods had to **repeat** similar calculations for each input function to derive its output function.
- **Deep learning-based studies** have been emerged to learn the operator from input function to the output function.

Find a map $\mathcal{G}: g(x) \mapsto u(t,x)$ satisfying

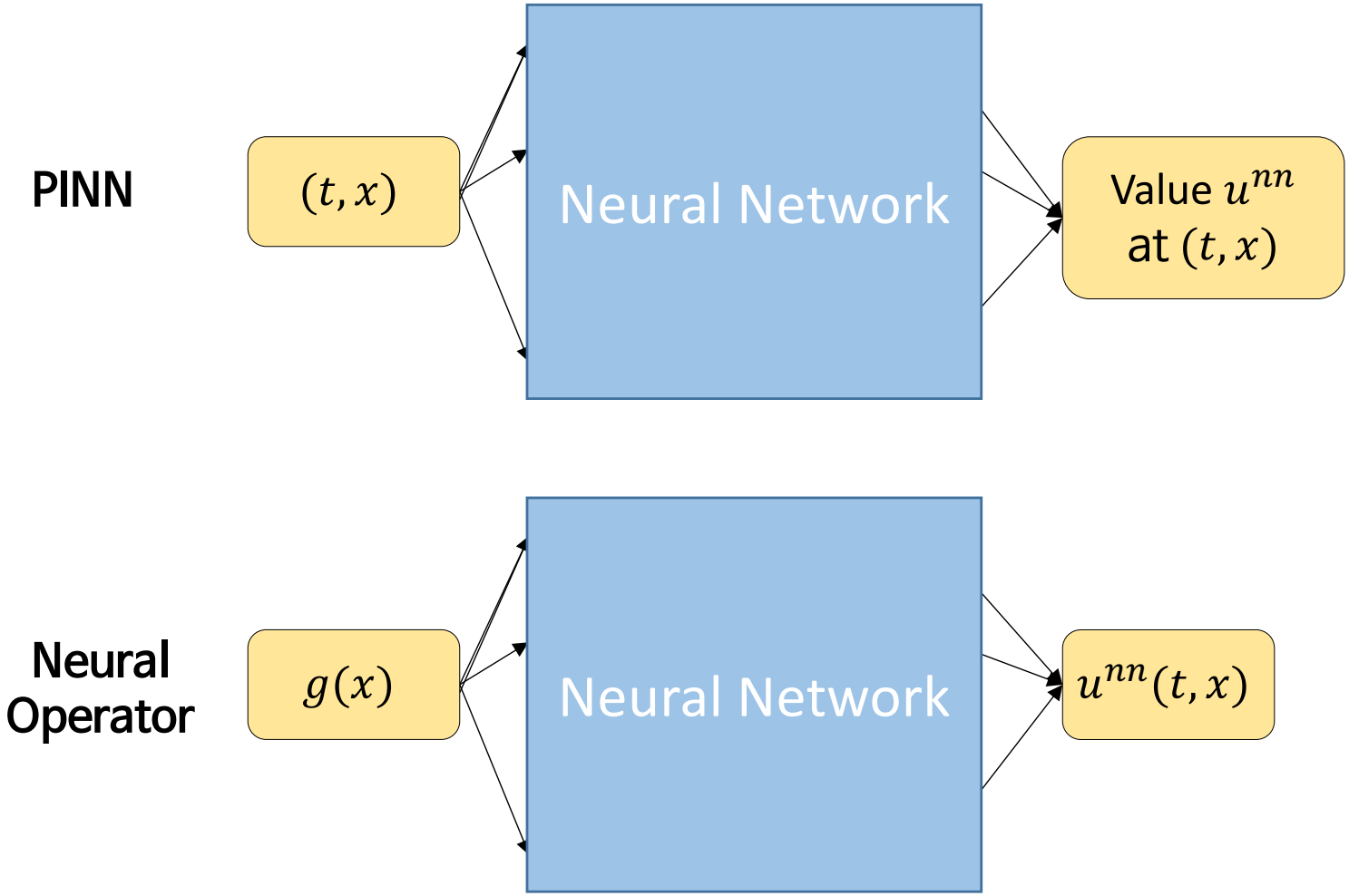
$$\mathcal{L}_{PDE} = f(u, u_t, u_x, u_{xx}, \dots) = 0$$
$$\mathcal{L}_{IC} = u(0,x) - g(x) = 0$$
$$\mathcal{L}_{BC} = u|_{\partial\Omega} - h(t,x)|_{\partial\Omega}$$



  - We need a dataset $\{g^i(x), u^i(t,x)\}_{i=1}^N$.
  - Predict $u^{new}(t,x)$ for a new $g^{new}(x)$.
- Models
  - Deep Operator Networks (DeepONet)
  - Fourier Neural Operator (FNO)

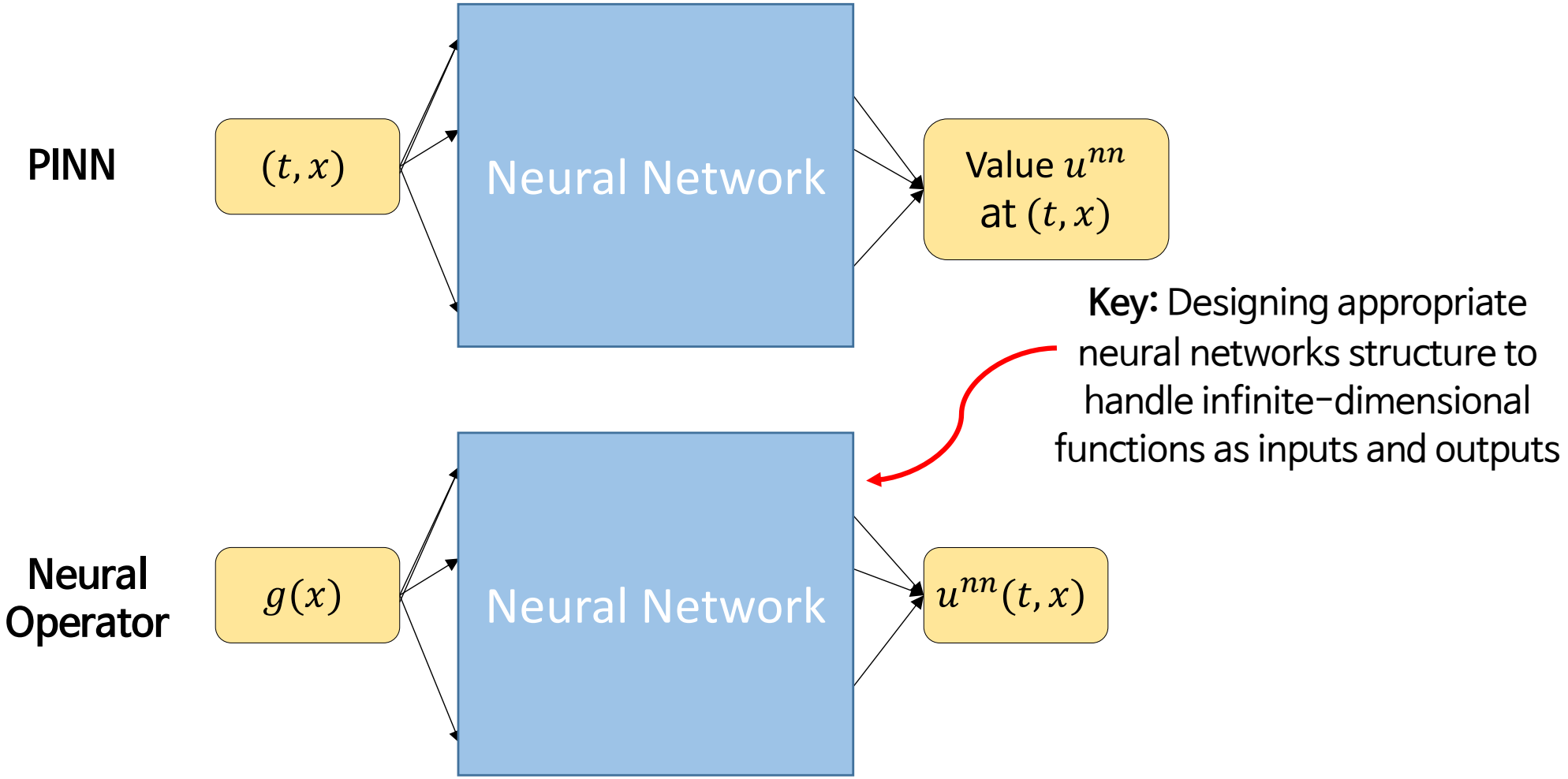
Comparison between PINN and Neural Operator

- Compared to Physics-Informed Neural Network (PINN)



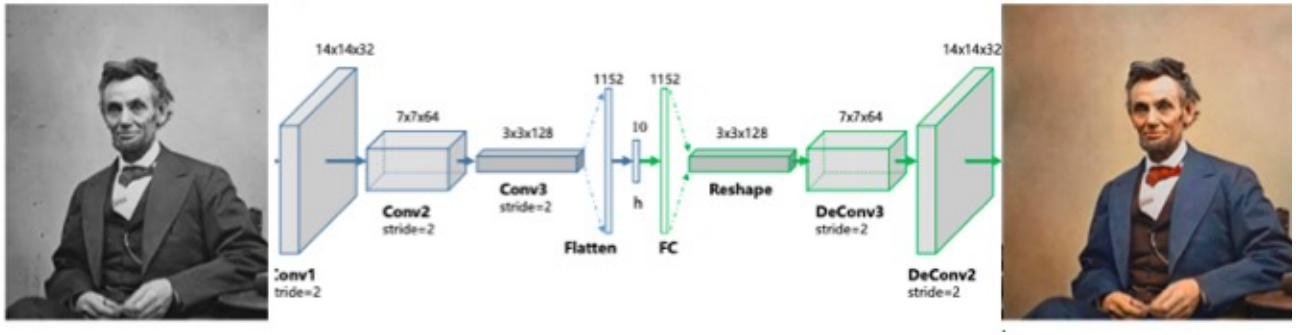
Comparison between PINN and Neural Operator

- Compared to Physics-Informed Neural Network (PINN)



Natural approach : image-to-image regression

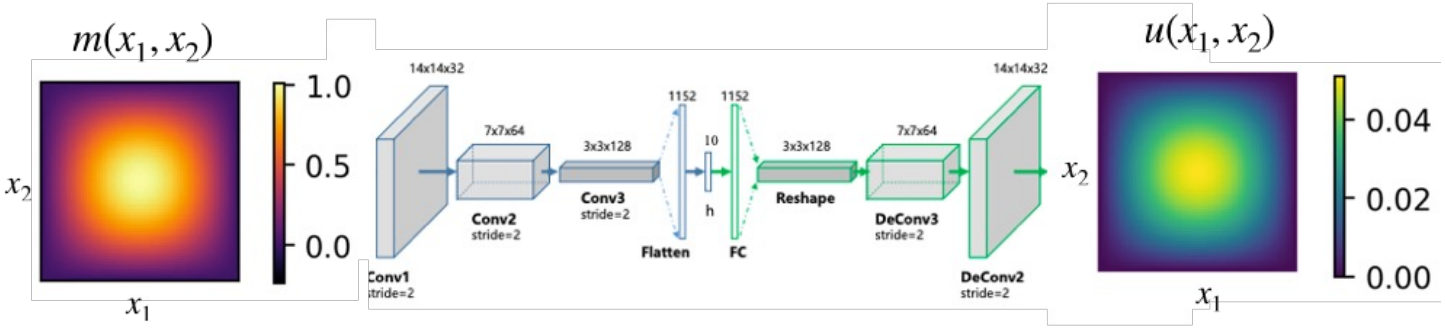
- **Convolutional Neural Network encoder-decoder (Image-to-image regression)**
  - Image segmentation, Image Noise Reduction, Image coloring



https://towardsdatascience.com/convolutional-autoencoders-for-image-noise-reduction-32fce9fc1763

Natural approach : image-to-image regression

- **Operator learning (function-to-function regression)**



Ex) Poisson equation

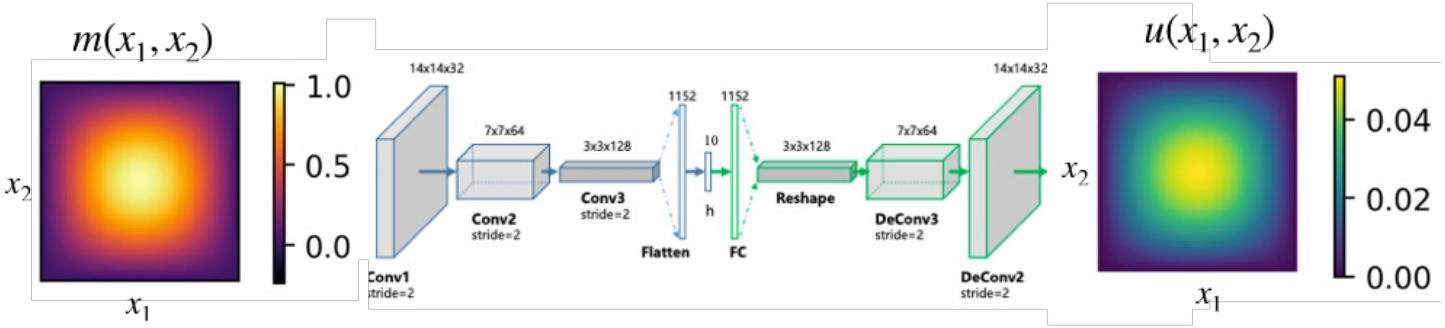
$$-\Delta u - m = 0 \text{ in } \Omega \subset \mathbb{R}^2$$
$$u = 0 \text{ in } \partial\Omega$$

Goal) Find a solution $u(x_1, x_2)$ for different source term $m(x_1, x_2)$.
(Learning operator from the source term to the solution)

$$m(x_1, x_2) \mapsto u(x_1, x_2)$$

Natural approach : image-to-image regression

- **Operator learning (function-to-function regression)**



Ex) Poisson equation

$$-\Delta u - m = 0 \text{ in } \Omega \subset \mathbb{R}^2$$
$$u = 0 \text{ in } \partial\Omega$$

Goal) Find a solution $u(x_1, x_2)$ for different source term $m(x_1, x_2)$.
(Learning operator from the source term to the solution)

$$m(x_1, x_2) \mapsto u(x_1, x_2)$$

- We can consider the functions m and u as 2d images $m(x_1, x_2)$ and $u(x_1, x_2)$.
- Limitation
  - Only when input function values are known in a uniform grid.
  - Only when all input function values are known in all grids.

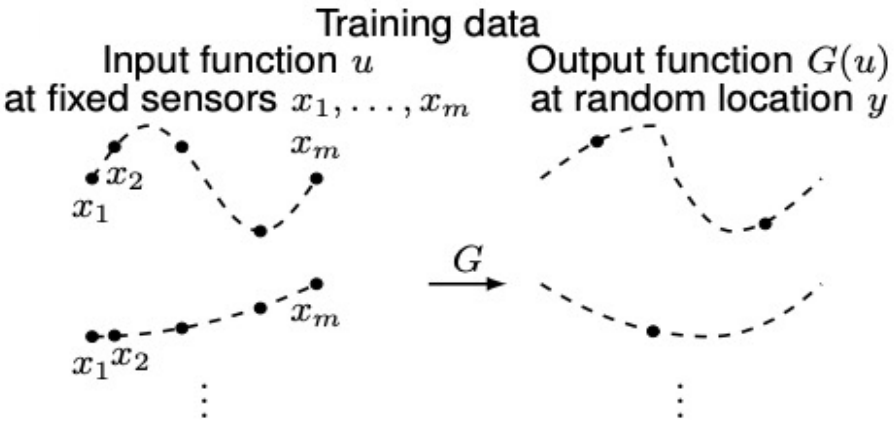
Part2-1. Deep Operator Network

Deep Operator Networks (DeepONets)

Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature machine intelligence*, *3*(3), 218-229.

Learning an operator $G: u(x) \mapsto G(u)(y)$ using the network

Data

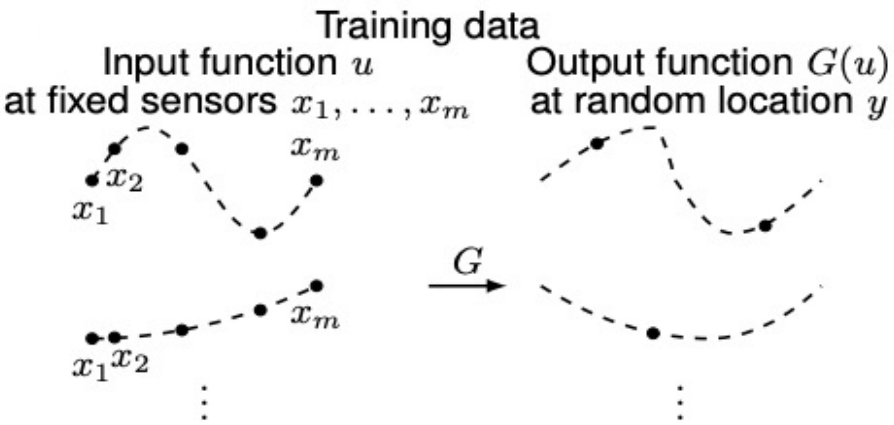


Deep Operator Networks (DeepONets)

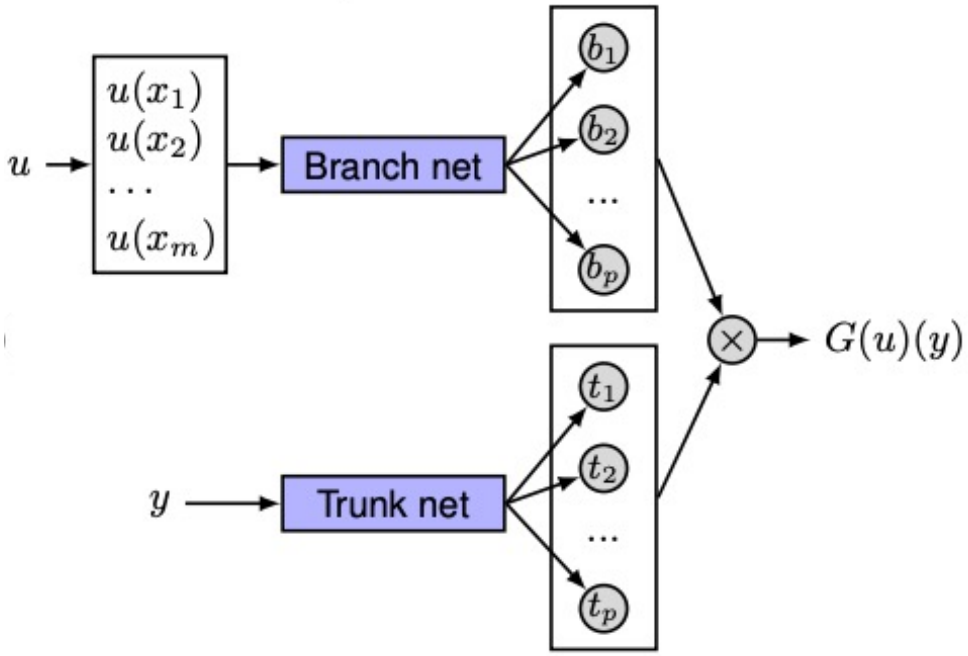
Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature machine intelligence*, *3*(3), 218-229.

Learning an operator $G: u(x) \mapsto G(u)(y)$ using the network

Data



Model



Deep Operator Networks (DeepONets)

Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature machine intelligence*, *3*(3), 218-229.

Learning an operator $G: u(x) \mapsto G(u)(y)$ using the network

[Chen and Chen, 1995]

Theorem 1 (Universal Approximation Theorem for Operator). *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*

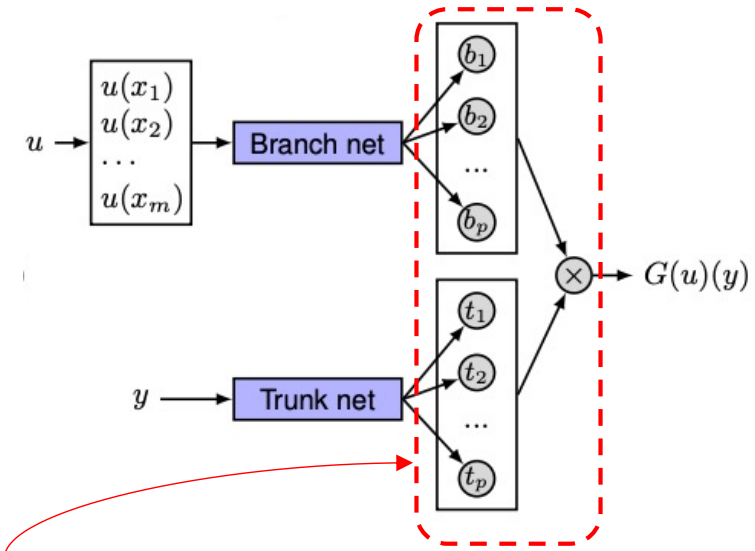
$$\left| G(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{branch} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{trunk} \right| < \epsilon \tag{1}$$

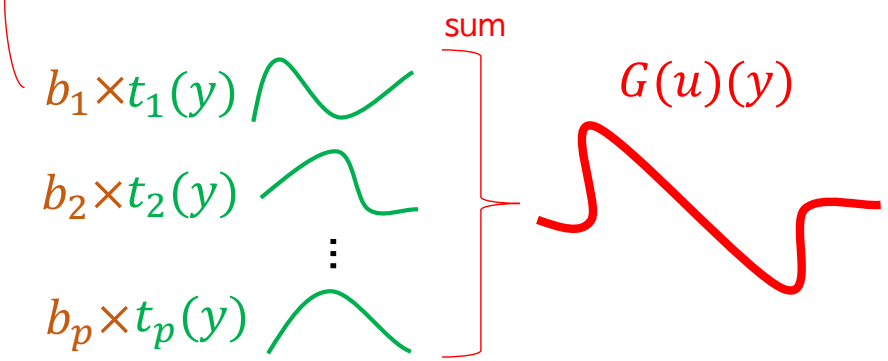
holds for all $u \in V$ and $y \in K_2$.

Understanding of DeepONet: linear approximation

Learning an operator $G: u(x) \mapsto G(u)(y)$ using the network

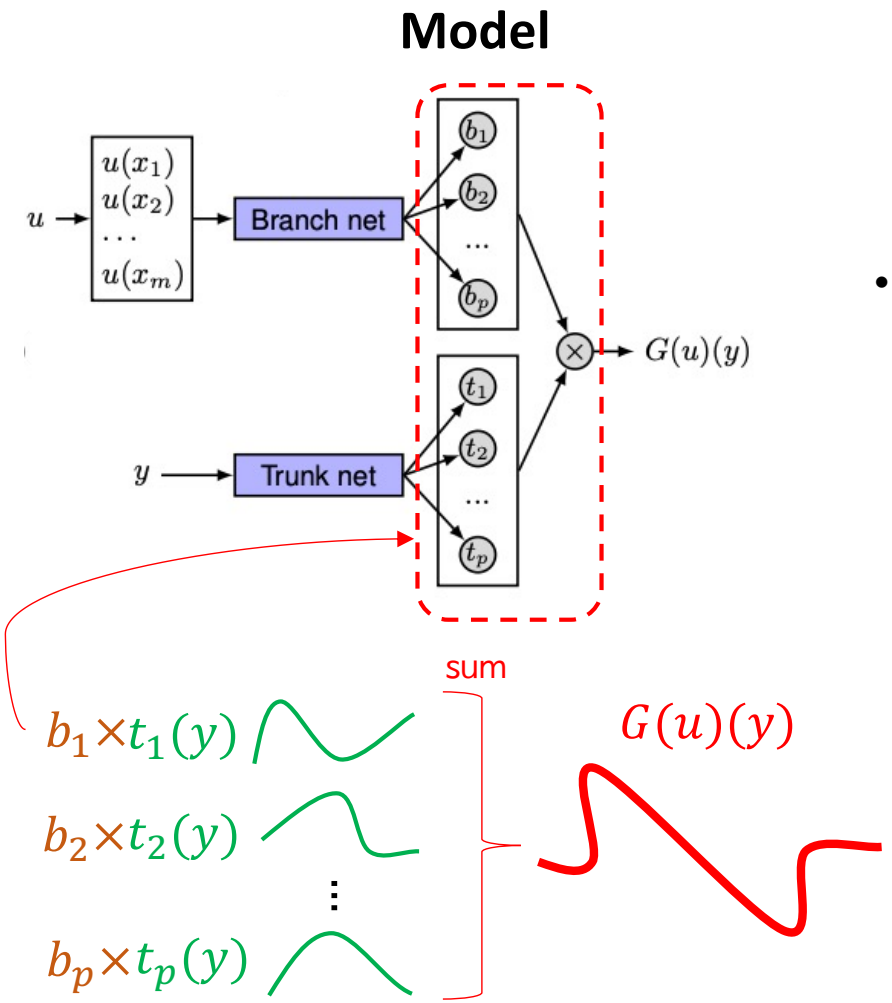
Model





Understanding of DeepONet: linear approximation

Learning an operator $G: u(x) \mapsto G(u)(y)$ using the network



- It approximates the target function $G(u)(y)$ as

$$G(u)(y) \approx b_1 t_1 + b_2 t_2 + \cdots + b_p t_p$$

  - Here, $t_1(y), t_2(y), \ldots, t_p(y)$ are the **p-basis functions** for the target function space domain $\mathcal{Y}$ ($y \in \mathcal{Y}$).
  - $b_1, b_2, \ldots, b_p$ are the **p-coefficient values** for the target function $G(u)$ (depends on the input function u)

Understanding of DeepONet: linear approximation

https://arxiv.org/abs/1910.03193

[Submitted on 8 Oct 2019 (v1), last revised 15 Apr 2020 (this version, v3)]

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

Lu Lu, Pengzhan Jin, George Em Karniadakis

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. 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, $i = 1, \cdots, m$ (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.

Practical Coding!

Let's implement

Deep Operator Network(DeepONet)!

DeepONet: Practical Example

We will learn an antiderivative operator $\mathcal{G} : u(x) \mapsto s(x)$ such that

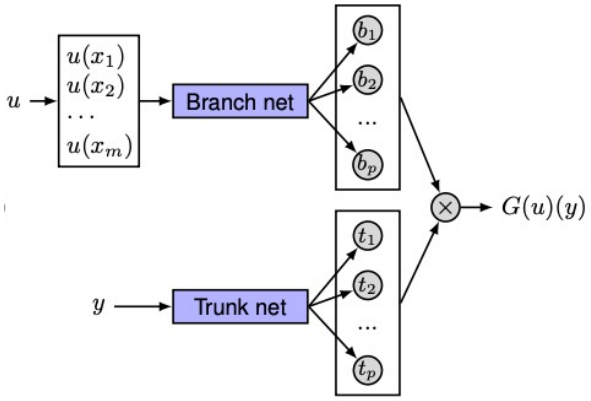
$$\frac{ds(x)}{dx} = g(s(x), u(x), x), \quad \text{for } x \in [-1, 1],$$

with $g(s(x), u(x), x) = u(x)$ and $s(0) = 0$ and

$$u(x) \in V_{\text{poly}} = \left\{ \sum_{i=0}^{N-1} a_i T_i(x) : a_i \sim Unif[-M, M] \right\}$$

where $T_i(x)$ are Chebyshev Polynimials of the first kind.

Then, the data point is triplet $(u(x), x, G(u)(x) = s(x))$



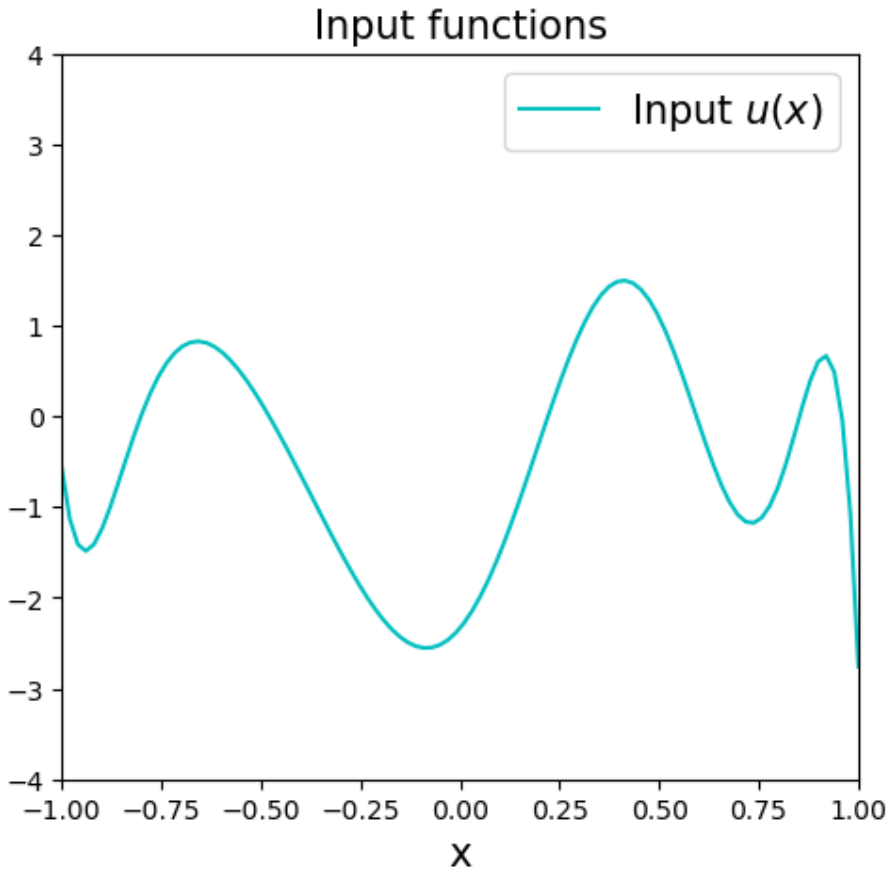
DeepONet: Practical Example

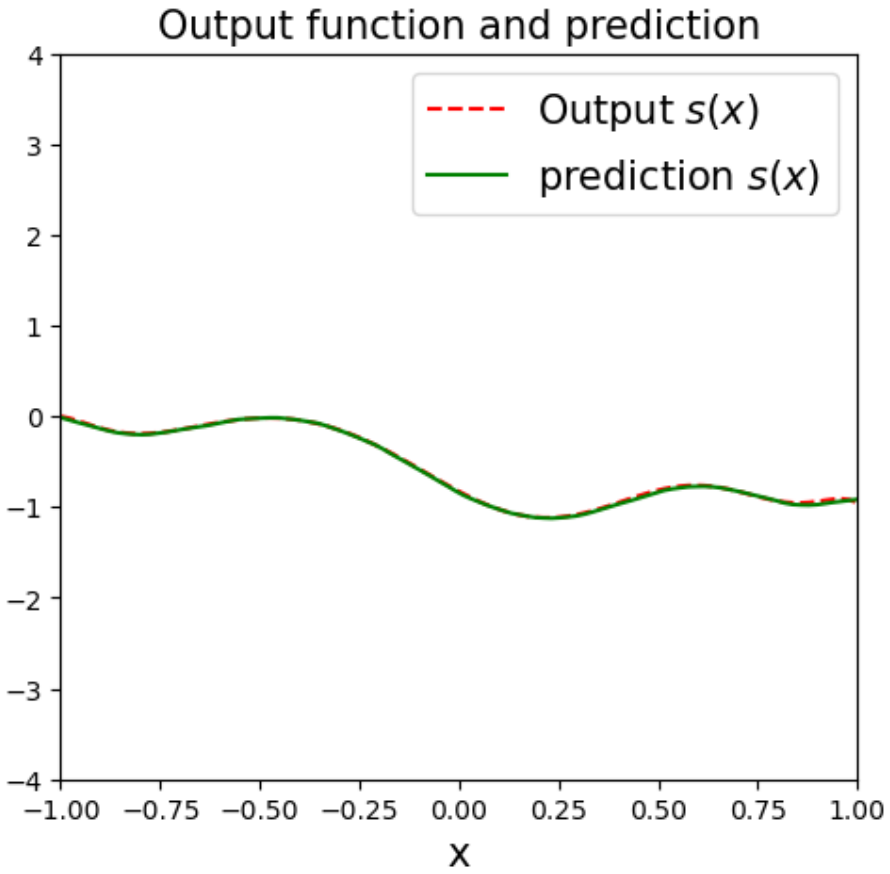
4. DeepONet

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output

import torch
import torch.nn as nn
import torch.optim as optim

import os
import pickle
import torch.utils.data as data_0
from scipy.integrate import odeint
from tqdm.notebook import tqdm
```

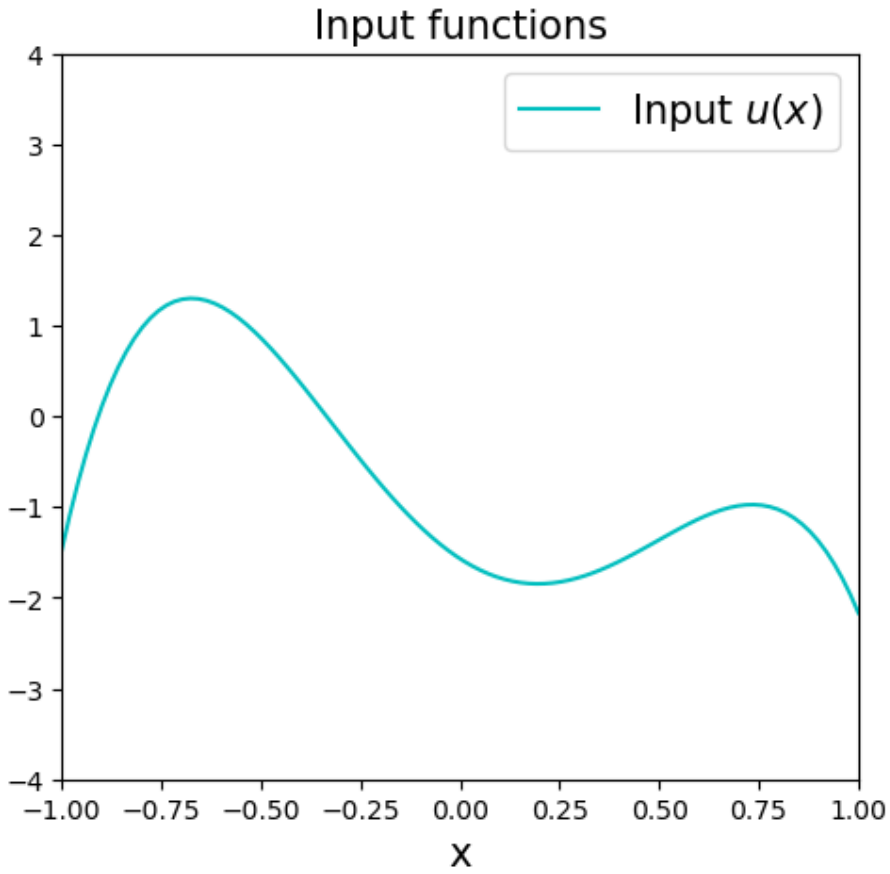
In [2]:
```
device = torch.device('cuda:2')
```

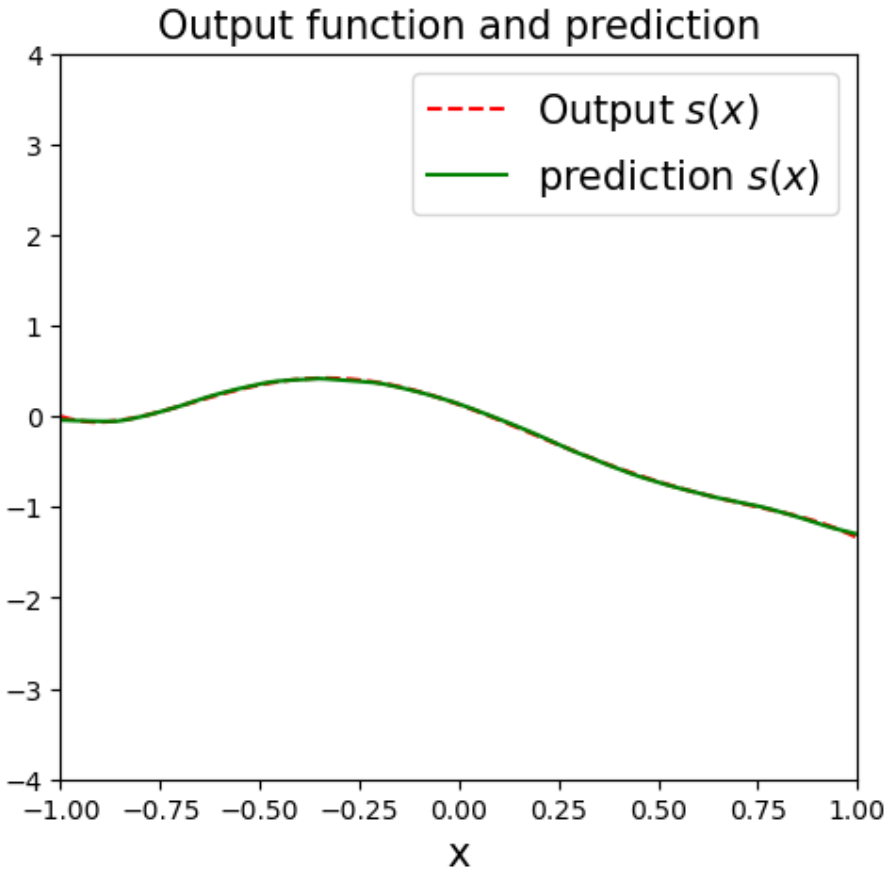
DeepONet: Practical Example





DeepONet: Practical Example





Quiz)

We will learn an antiderivative operator $\mathcal{G} : u(x) \mapsto s(x)$ such that

$$\frac{ds(x)}{dx} = g(s(x), u(x), x), \quad \text{for } x \in [-1, 1],$$

with $g(s(x), u(x), x) = u(x)$ and $s(0) = 0$ and

$$u(x) \in V_{\text{poly}} = \left\{ \sum_{i=0}^{N-1} a_i T_i(x) : a_i \sim Unif[-M, M] \right\}$$

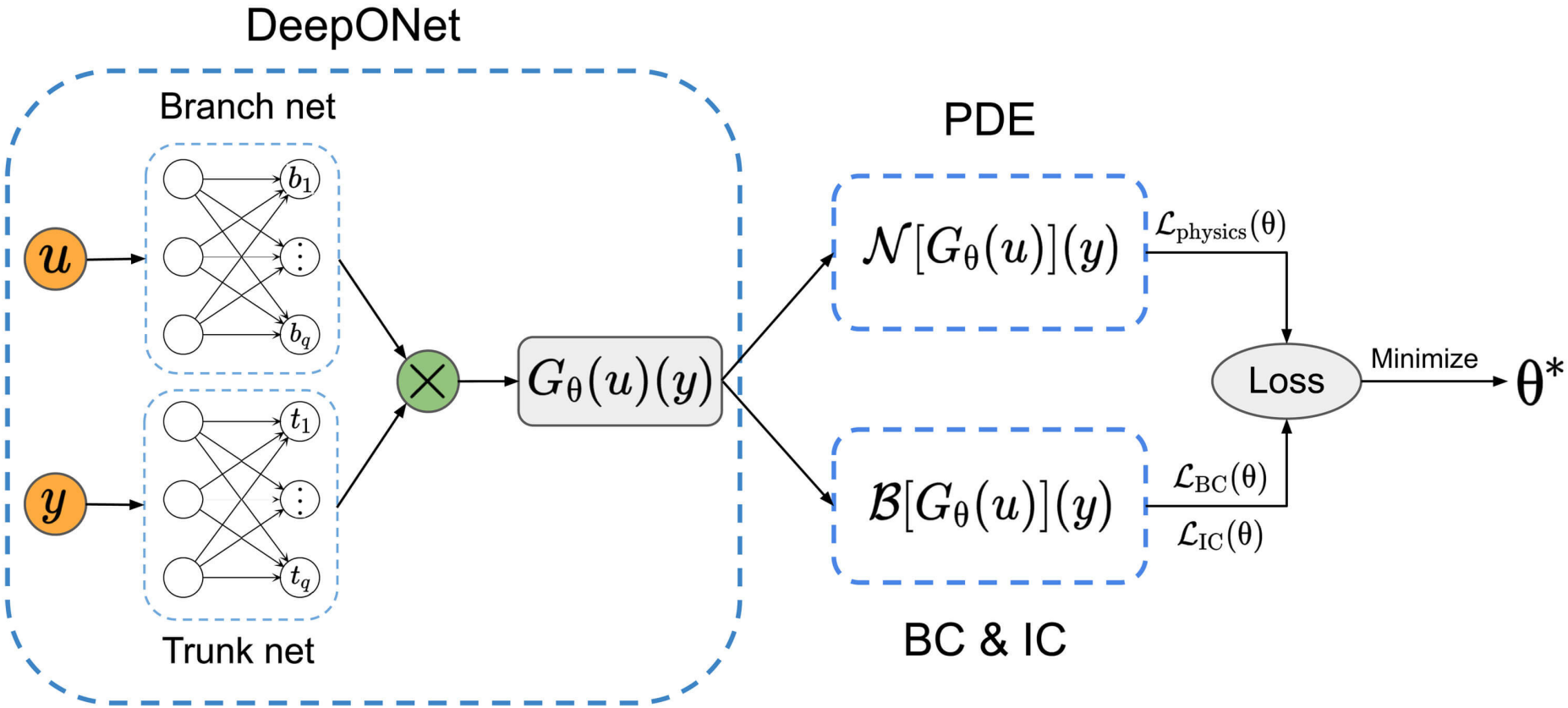
where $T_i(x)$ are Chebyshev Polynimials of the first kind.

Then, the data point is triplet $(u(x), x, G(u)(x) = s(x))$

Can we give a physics-informed loss function in DeepONet similar to PINN?

Physics-Informed DeepONet

Wang, Sifan, Hanwen Wang, and Paris Perdikaris. "**Learning the solution operator of parametric partial differential equations with physics-informed DeepONets**." *Science advances* 7.40 (2021): eabi8605.



Part2-2. Fourier Neural Operator

Neural Operator

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between functi on spaces with applications to pdes. Journal of Machine Learning Research, 24(89), 1-97.

Journal of Machine Learning Research 23 (2022) 1-97 Submitted 12/21; Revised 10/22; Published 12/22

Neural Operator: Learning Maps Between Function Spaces With Applications to PDEs

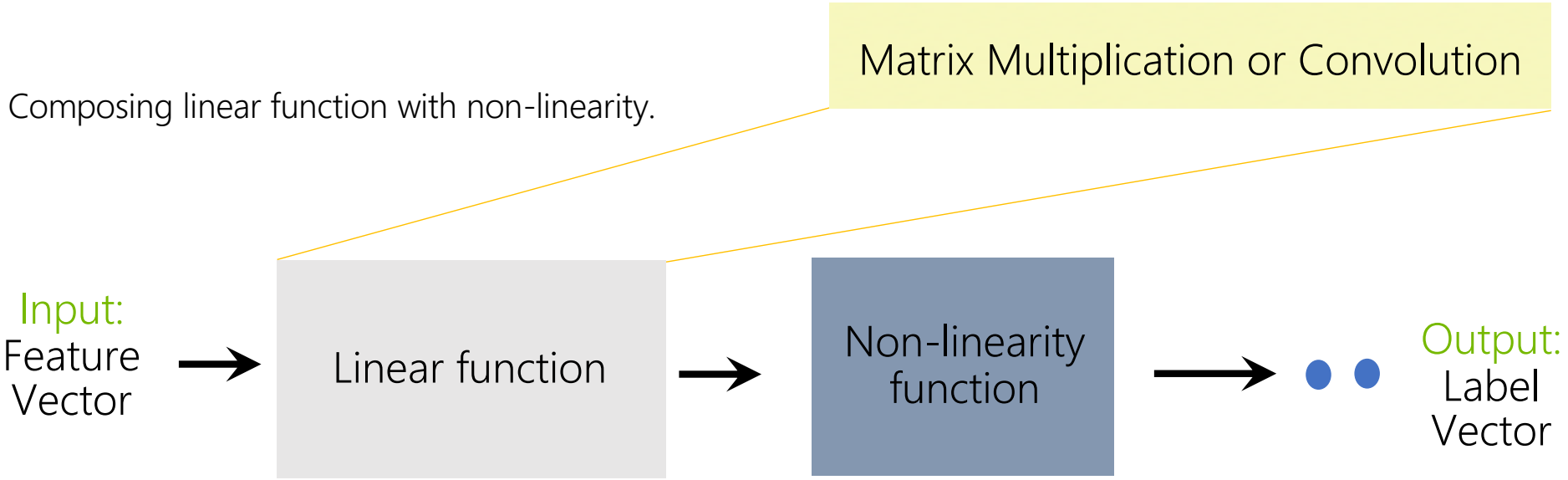
Nikola Kovachki*† NKOVACHKI@NVIDIA.COM *Nvidia*
Zongyi Li* ZONGYILI@CALTECH.EDU *Caltech*
Burigede Liu BL377@CAM.AC.UK *Cambridge University*
Kamyar Azizzadenesheli KAMYARA@NVIDIA.COM *Nvidia*
Kaushik Bhattacharya BHATTA@CALTECH.EDU *Caltech*
Andrew Stuart ASTUART@CALTECH.EDU *Caltech*
Anima Anandkumar ANIMA@CALTECH.EDU *Caltech*

Editor: Lorenzo Rosasco

Neural Operator

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between functi on spaces with applications to pdes. Journal of Machine Learning Research, 24(89), 1-97.

Neural networks -> Learning in fixed dimensions

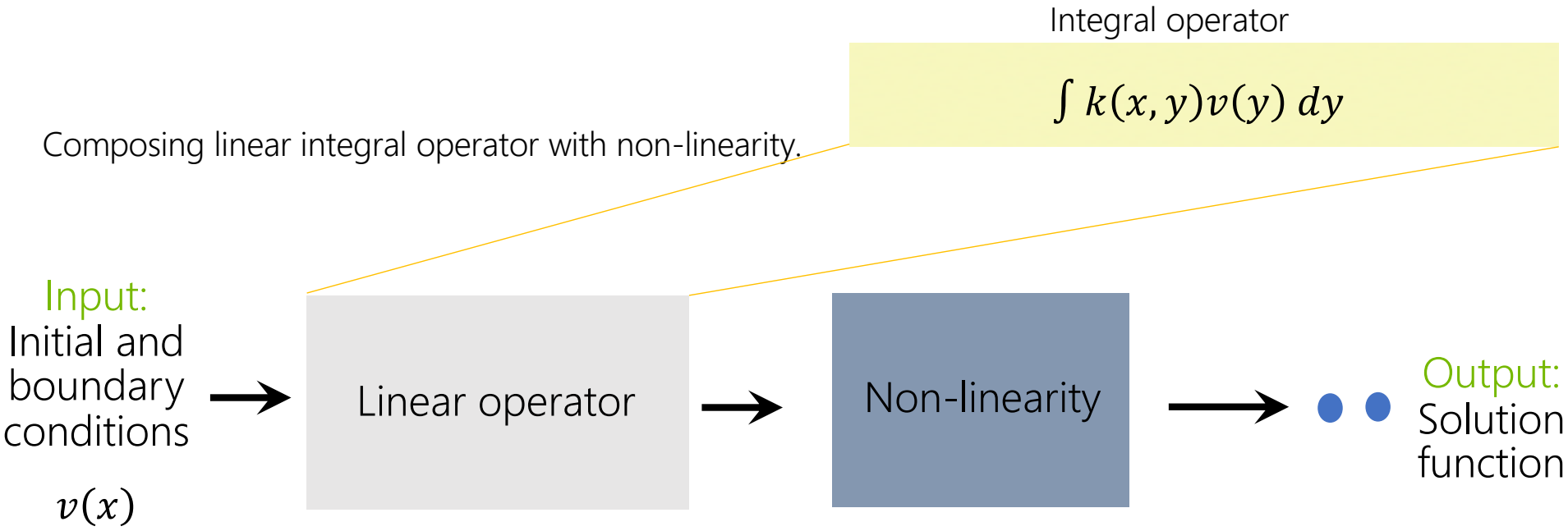


Neural Network can approximate any continuous function.

Neural Operator

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between functi on spaces with applications to pdes. Journal of Machine Learning Research, 24(89), 1-97.

Neural operator -> Learning in infinite dimensions



Neural Operator can approximate any continuous operator

Neural Operator

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between functi on spaces with applications to pdes. Journal of Machine Learning Research, 24(89), 1-97.

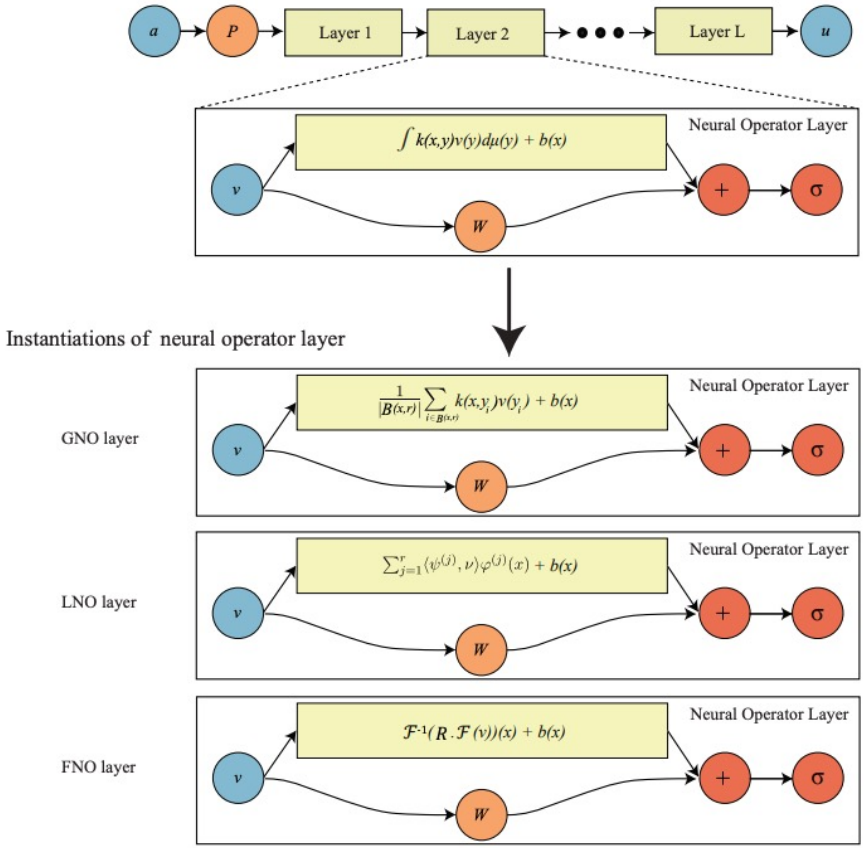


Figure 2: Neural operator architecture schematic

The input function a is passed to a pointwise lifting operator P that is followed by T layers of integral operators and pointwise non-linearity operations σ. In the end, the pointwise projection operator Q outputs the function u. Three instantiation of neural operator layers, GNO, LNO, and FNO are provided.

Neural Operator

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between functi on spaces with applications to pdes. Journal of Machine Learning Research, 24(89), 1-97.

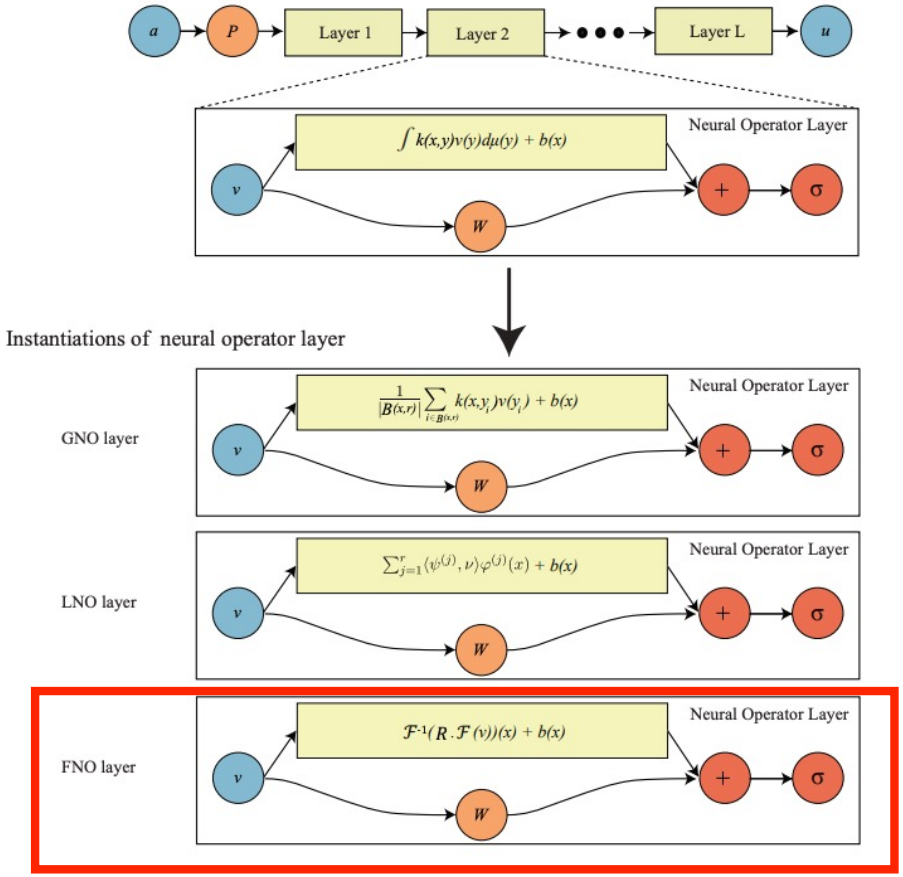


Figure 2: Neural operator architecture schematic

The input function a is passed to a pointwise lifting operator P that is followed by T layers of integral operators and pointwise non-linearity operations σ. In the end, the pointwise projection operator Q outputs the function u. Three instantiation of neural operator layers, GNO, LNO, and FNO are provided.

Neural Operator

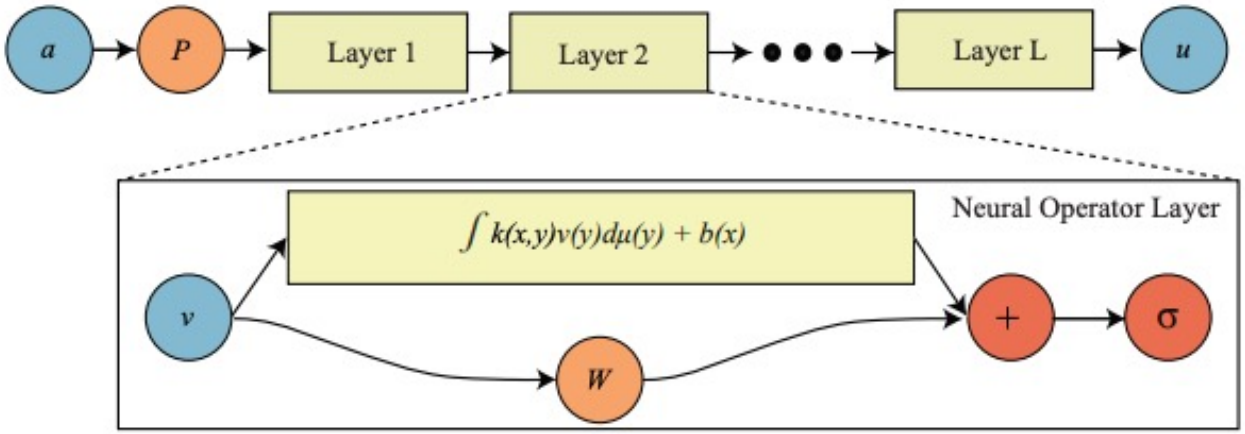
Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2020). Fourier neural operator for parametric partial di fferential equations. *arXiv preprint arXiv:2010.08895*.

Learning an operator $\mathcal{G}: a(x) \mapsto \mathcal{G}(a)(x) = u(x)$ using the network

- **Neural Operator**
  - Lifting : $v_0(x) = P(a(x))$
  - Iterative Kernel Integration

$v_{t+1}(x) = \sigma\left(W_\theta v_t(x) + \boldsymbol{K_\phi v_t(x)}\right)$, where $\boldsymbol{K_\phi v_t(x)} = \int_{\boldsymbol{D}} \boldsymbol{\kappa_\phi}\left(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{a(x)}, \boldsymbol{a(y)}\right)\boldsymbol{v_t(y)dy}$

  - Projection : $\mathrm{u}(x) = Q(v_T(x))$



Fourier Neural Operator

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2020). Fourier neural operator for parametric partial di fferential equations. *arXiv preprint arXiv:2010.08895*.

Learning an operator $\mathcal{G}: a(x) \mapsto \mathcal{G}(a)(x) = u(x)$ using the network

- **Neural Operator**
  - Lifting : $v_0(x) = P(a(x))$
  - Iterative Kernel Integration

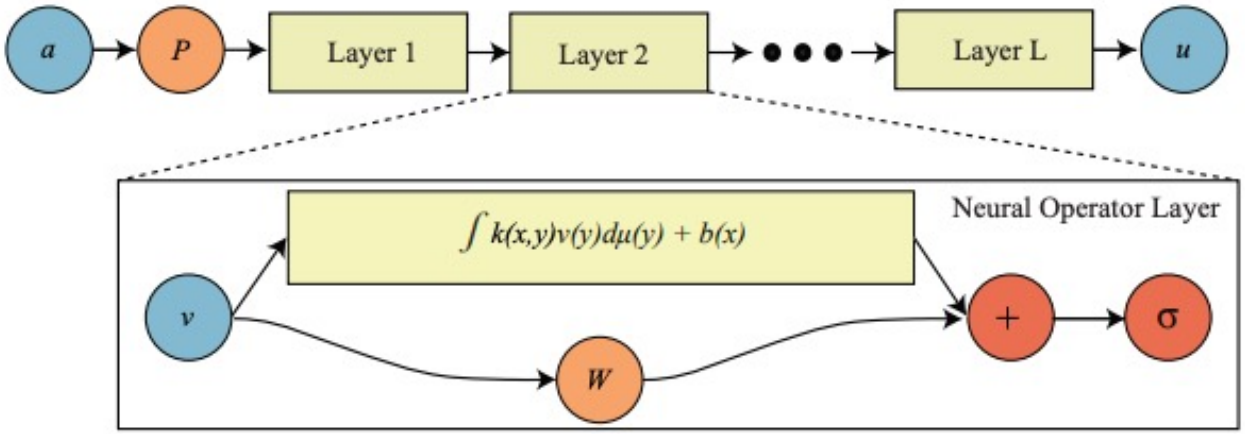
$\boldsymbol{\kappa_\phi \approx \kappa_\phi(x-y)}$

$= \boldsymbol{\kappa_\phi(x) * v_t(x)}$

$= \boldsymbol{\mathcal{F}^{-1}\left[\mathcal{F}[\kappa_\phi]\mathcal{F}[v_t]\right]}$

$v_{t+1}(x) = \sigma\left(W_\theta v_t(x) + \boldsymbol{K_\phi v_t(x)}\right)$, where $\boldsymbol{K_\phi v_t(x) = \int_D \kappa_\phi(x, y, a(x), a(y)) v_t(y) dy}$

  - Projection : $\mathrm{u}(x) = Q(v_T(x))$



Fourier Neural Operator

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2020). Fourier neural operator for parametric partial di fferential equations. *arXiv preprint arXiv:2010.08895*.

Learning an operator $\mathcal{G}: a(x) \mapsto \mathcal{G}(a)(x) = u(x)$ using the network

- **Neural Operator**
  - Lifting : $v_0(x) = P(a(x))$
  - Iterative Kernel Integration

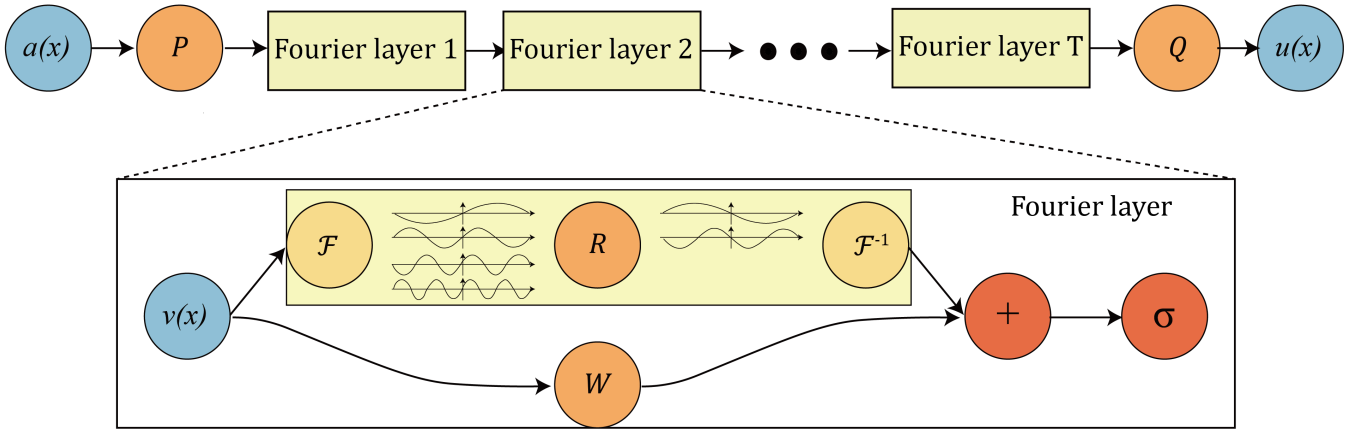
$\boldsymbol{\kappa_\phi \approx \kappa_\phi(x-y)}$

$\boldsymbol{= \kappa_\phi(x) * v_t(x)}$

$\boldsymbol{= \mathcal{F}^{-1}\left[\mathcal{F}[\kappa_\phi]\mathcal{F}[v_t]\right]}$

$v_{t+1}(x) = \sigma\left(W_\theta v_t(x) + \boldsymbol{K_\phi v_t(x)}\right)$, where $\boldsymbol{K_\phi v_t(x) = \int_D \kappa_\phi\left(x, y, a(x), a(y)\right) v_t(y) dy}$

  - Projection : $u(x) = Q(v_T(x))$



- **Fourier Neural Operator**
  - $K_\phi v_t(x) = \mathcal{F}^{-1}\left(R_\phi \cdot (\mathcal{F} v_t)\right)(x)$

Fourier Neural Operator

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2020). Fourier neural operator for parametric partial di fferential equations. *arXiv preprint arXiv:2010.08895*.

- **Neural Operator: Iterative Kernel Integration**

$$v_{t+1}(x) = \sigma\left(W_\theta v_t(x) + \boldsymbol{K_\phi v_t(x)}\right)$$

where $\boldsymbol{K_\phi v_t(x)} = \int_{\boldsymbol{D}} \boldsymbol{\kappa_\phi(x,y) v_t(y) dy} \approx \boldsymbol{\mathcal{F}^{-1}}\left[\boldsymbol{\mathcal{F}[\kappa_\phi]}\mathcal{F}[\boldsymbol{v_t}]\right]$.

$= \boldsymbol{R_\phi}$

Fourier Neural Operator

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2020). Fourier neural operator for parametric partial di fferential equations. *arXiv preprint arXiv:2010.08895*.

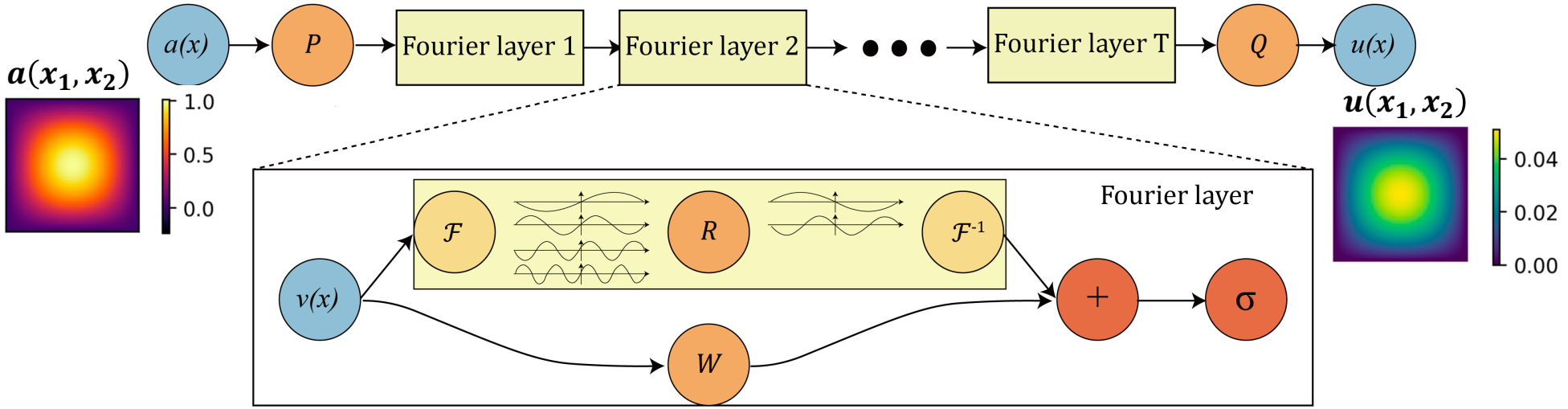
- **Neural Operator: Iterative Kernel Integration**

$$v_{t+1}(x) = \sigma\left(W_\theta v_t(x) + \boldsymbol{K_\phi v_t(x)}\right)$$

where $\boldsymbol{K_\phi v_t(x)} = \int_D \boldsymbol{\kappa_\phi(x,y) v_t(y) dy} \approx \boldsymbol{\mathcal{F}^{-1}\left[\mathcal{F}[\kappa_\phi]\mathcal{F}[v_t]\right]}$.

$\boldsymbol{= R_\phi}$

Learning an operator $\mathcal{G}: a(x) \mapsto \mathcal{G}(a)(x) = u(x)$ using the network

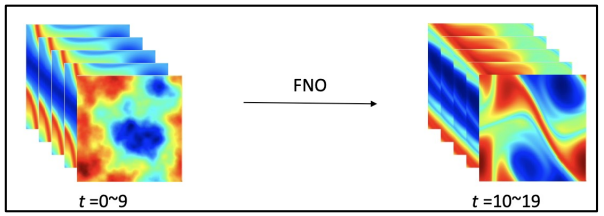


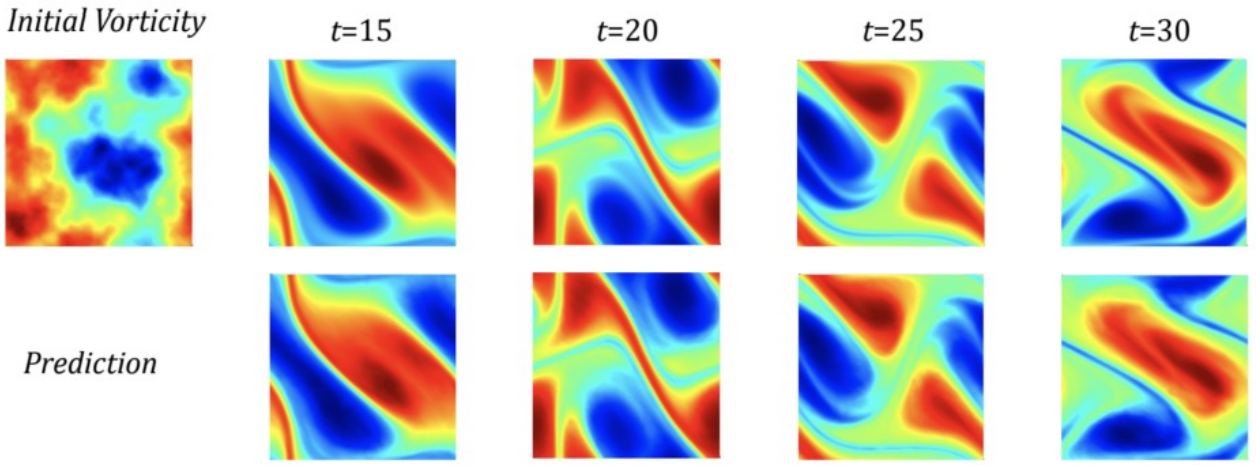
Fourier Neural Operator

(ex) 2-D Navier-Stokes equation (viscous, incompressible fluid in vorticity form)

$$\begin{cases} \partial_t w(x,t) + u(x,t) \cdot \nabla w(x,t) = \nu \Delta w(x,t) + f(x), & x \in (0,1)^2 \\ \nabla \cdot u(x,t) = 0, & t \in (0,T] \\ w(x,0) = w_0(x), & \\ w(x,t) = \nabla \times u(x,t), & \end{cases}$$

Learning an operator $\mathcal{G}$: $w|_{(0,1)^2 \times [0,9]} \mapsto w|_{(0,1)^2 \times [10,19]}$





Zero-shot super-resolution: Navier-Stokes Equation with viscosity $\nu = 1e{-}4$; Ground truth on top and prediction on bottom; trained on $64 \times 64 \times 20$ dataset; evaluated on $256 \times 256 \times 80$ (see Section 5.4).

Fourier Neural Operator

(ex) 2-D Navier-Stokes equation (viscous, incompressible fluid in vorticity form)

$$\begin{cases} \partial_t w(x,t) + u(x,t)\cdot\nabla w(x,t) = \nu\Delta w(x,t) + f(x), & x \in (0,1)^2 \\ \nabla\cdot u(x,t) = 0, & t \in (0,T] \\ w(x,0) = w_0(x), & \\ w(x,t) = \nabla\times u(x,t), & \end{cases}$$

Learning an operator $\mathcal{G}$: $w|_{(0,1)^2\times[0,9]} \mapsto w|_{(0,1)^2\times[10,19]}$

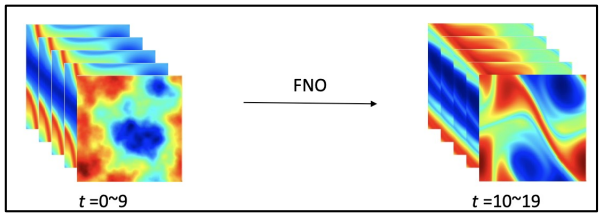


Table 1: Benchmarks on Navier Stokes (fixing resolution 64×64 for both training and testing)

Config	Parameters	Time per epoch	$\nu = 1\text{e}{-3}$ $T = 50$ $N = 1000$	$\nu = 1\text{e}{-4}$ $T = 30$ $N = 1000$	$\nu = 1\text{e}{-4}$ $T = 30$ $N = 10000$	$\nu = 1\text{e}{-5}$ $T = 20$ $N = 1000$
FNO-3D	6,558,537	38.99s	**0.0086**	0.1918	**0.0820**	0.1893
FNO-2D	414,517	127.80s	0.0128	**0.1559**	0.0834	**0.1556**
U-Net	24,950,491	48.67s	0.0245	0.2051	0.1190	0.1982
TF-Net	7,451,724	47.21s	0.0225	0.2253	0.1168	0.2268
ResNet	266,641	78.47s	0.0701	0.2871	0.2311	0.2753

Practical Coding!

Let's implement

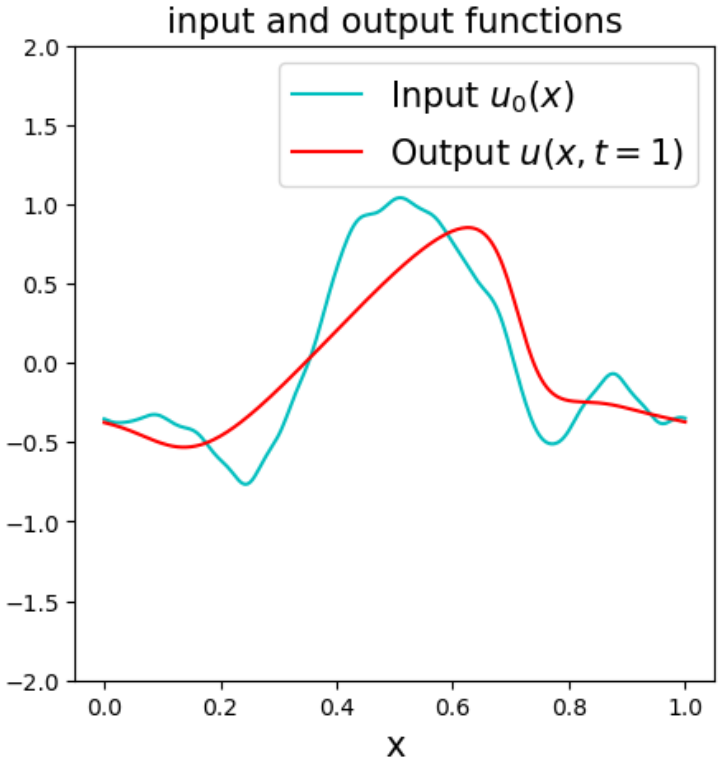
Fourier Neural Operator (FNO)!

Fourier Neural Operator - Burgers' eq example

(ex) 1-D Burgers' equation

$$\begin{cases} \partial_t u(x,t) + \partial_x(u^2(x,t)/2) = \nu \partial_{xx} u(x,t), & x \in (0,1) \\ u(x,0) = u_0(x), & t \in (0,1] \end{cases}$$

Goal: Learning an operator $\mathcal{G}: \mathrm{u}_0 \mapsto u(\cdot, 1)$



Fourier Neural Operator - Burgers' eq example

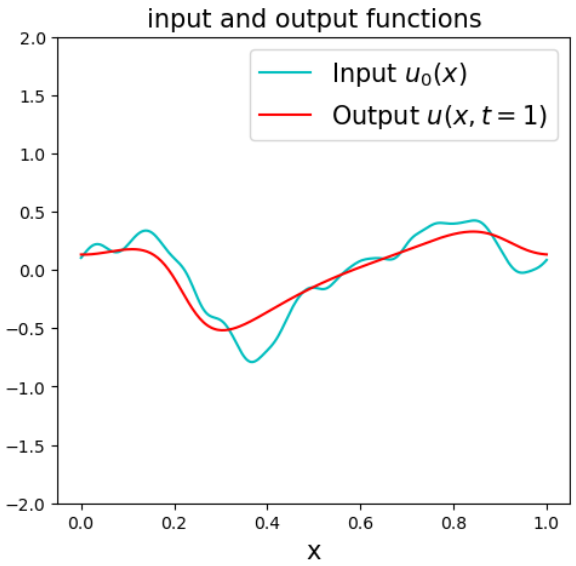
```
################################################################
# fourier layer
################################################################
class SpectralConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, modes1):
        super(SpectralConv1d, self).__init__()

        """
        1D Fourier layer. It does FFT, linear transform, and Inverse FFT.
        """

        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
```

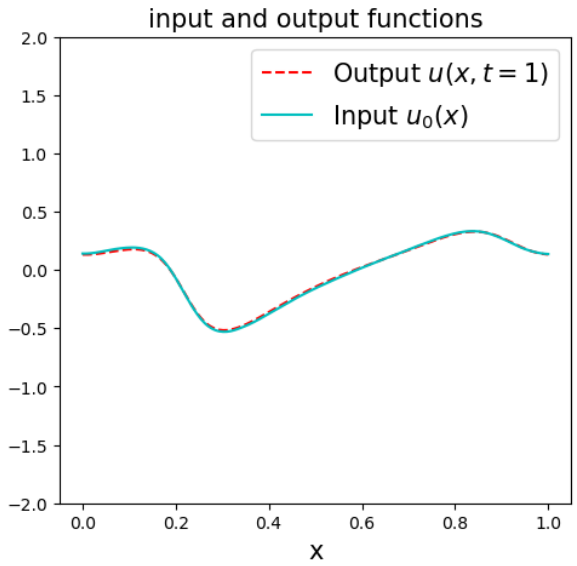
Fourier Neural Operator - Burgers' eq example

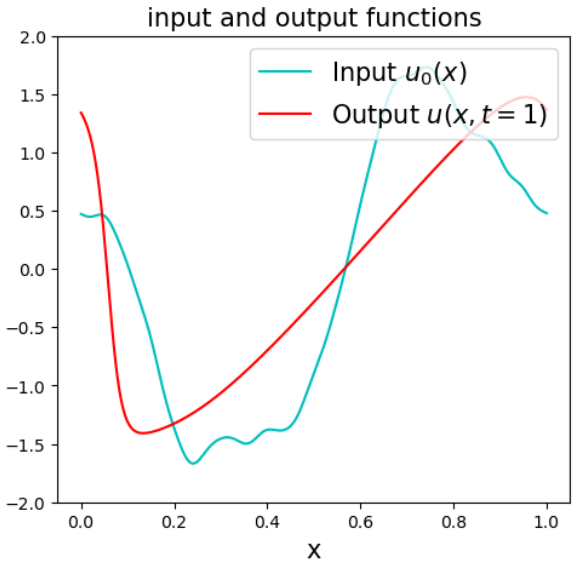
```python
class FNO1d(nn.Module):
    def __init__(self, modes, width):
        super(FNO1d, self).__init__()

        """
        The overall network. It contains 4 layers of the Fourier layer.
        1. Lift the input to the desire channel dimension by self.fc0 .
        2. 4 layers of the integral operators u' = (W + K)(u).
            W defined by self.w; K defined by self.conv .
        3. Project from the channel space to the output space by self.fc1 and self.fc2 .

        input: the solution of the initial condition and location (a(x), x)
        input shape: (batchsize, x=s, c=2)
        output: the solution of a later timestep
        output shape: (batchsize, x=s, c=1)
        """

        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)

    def forward(self, x):
```

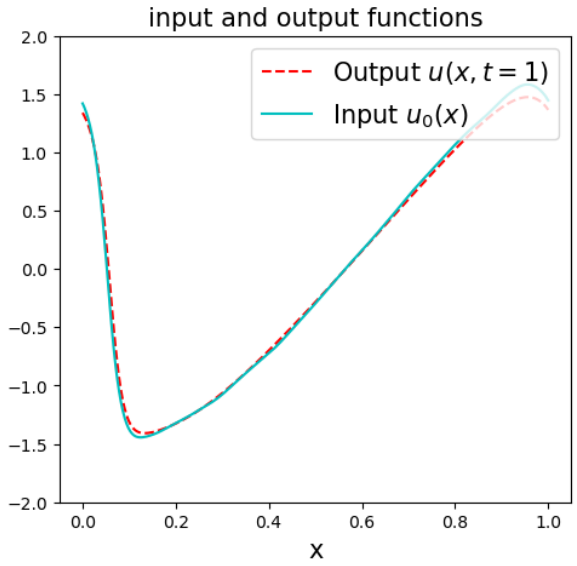
```python
    def forward(self, x):
        grid = self.get_grid(x.shape, x.device)
        x = torch.cat((x, grid), dim=-1)
        x = self.fc0(x)
        x = x.permute(0, 2, 1)
        # x = F.pad(x, [0,self.padding]) # pad the domain if input is non-periodic

        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[..., :-self.padding] # pad the domain if input is non-periodic
        x = x.permute(0, 2, 1)
        x = self.fc1(x)
        x = F.gelu(x)
        x = self.fc2(x)
        return x

    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)
```

Fourier Neural Operator - Burgers' eq example

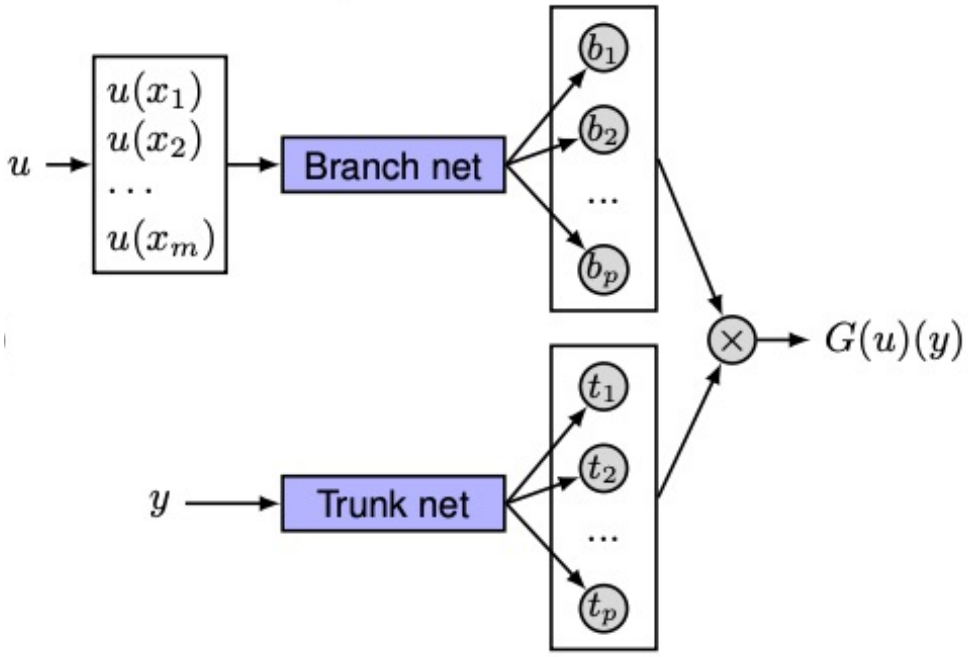


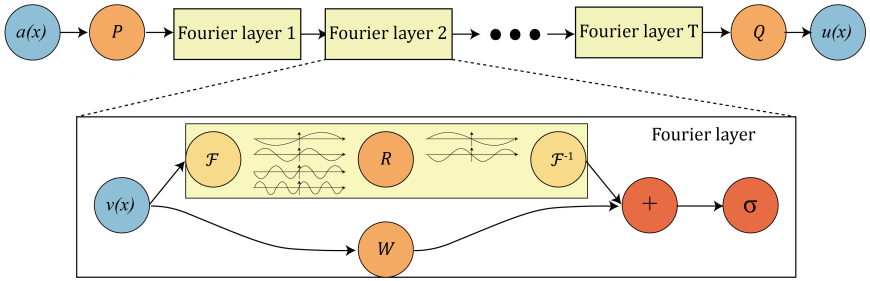






DeepONet vs Fourier Neural Operator





Lu, L., Jin, P., Pang, G., Zhang, Z., & Karniadakis, G. E. (2021). Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature machine intelligence*, *3*(3), 218-229.

Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2020). Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*.

DeepONet vs Fourier Neural Operator

Kovachki, N., Li, Z., Liu, B., Azizzadenesheli, K., Bhattacharya, K., Stuart, A., & Anandkumar, A. (2023). Neural operator: Learning maps between functi on spaces with applications to pdes. Journal of Machine Learning Research, 24(89), 1-97.

Property \ Model	NNs	DeepONets	Interpolation	Neural Operators
Discretization Invariance	✗	✗	✓	✓
Is the output a function?	✗	✓	✓	✓
Can query the output at any point?	✗	✓	✓	✓
Can take the input at any point?	✗	✗	✓	✓
Universal Approximation	✗	✓	✗	✓

Table 1: Comparison of deep learning models. The first row indicates whether the model is discretization invariant. The second and third rows indicate whether the output and input are a functions. The fourth row indicates whether the model class is a universal approximator of operators. Neural Operators are discretization invariant deep learning methods that output functions and can approximate any operator.

DeepONet vs Fourier Neural Operator

Lu, L., Meng, X., Cai, S., Mao, Z., Goswami, S., Zhang, Z., & Karniadakis, G. E. (2022). A comprehensive and fair comparison of two neural operators (wi th practical extensions) based on fair data. Computer Methods in Applied Mechanics and Engineering, 393, 114778.

Table 2
Comparison between vanilla DeepONet and vanilla FNO.

	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
Discontinuous functions	Good	Questionable

Part3. Recent works for SciML

Selected Research Works

[1] Lee, J. Y., SungWoong, C. H. O., & Hwang, H. J. (2023). HyperDeepONet: learning operator with complex target function space using the limited resources via hypernetwork, **ICLR 2023.**

[2] Lee, J.Y., Ko, S., Hong, Y. (2025). Finite element operator network for solving elliptic-type parametric pdes. **SIAM Journal on Scientific Computing**.

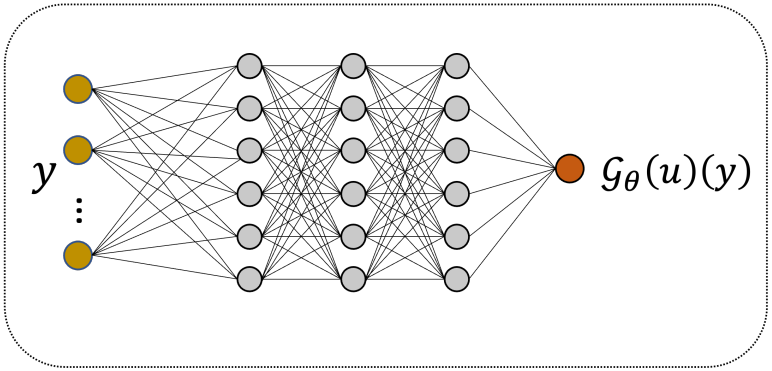
[3] Song, S. K., Shin, B. G., & Lee, J. Y. (2026). Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation. **ICML 2026.**

[4] Seo, J., Lee, S., & Lee, J. Y. (2026). Unbiased and Second-Order-Free Training for High-Dimensional PDEs. **ICML 2026.**

[ICLR 2023] HyperDeepONet

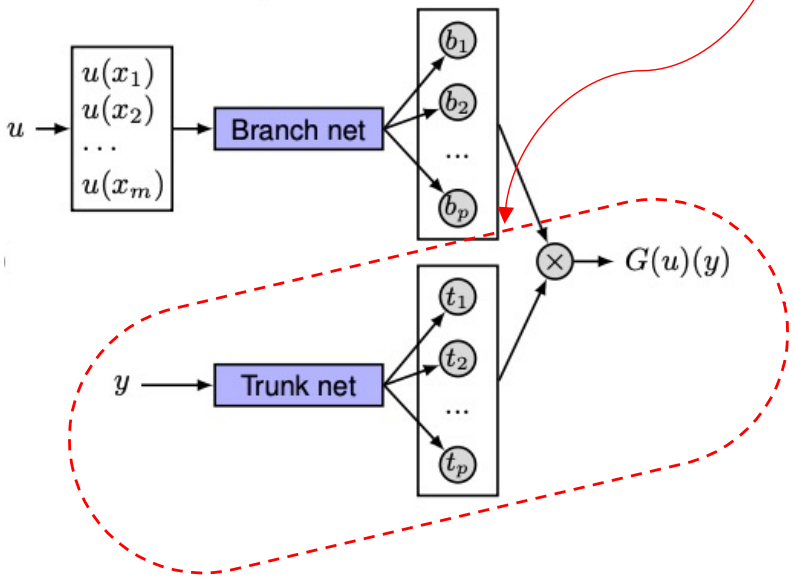
Learning an operator $G: u(x) \mapsto G(u)(y)$ using the network

How to put information of $u \in \mathcal{U}$



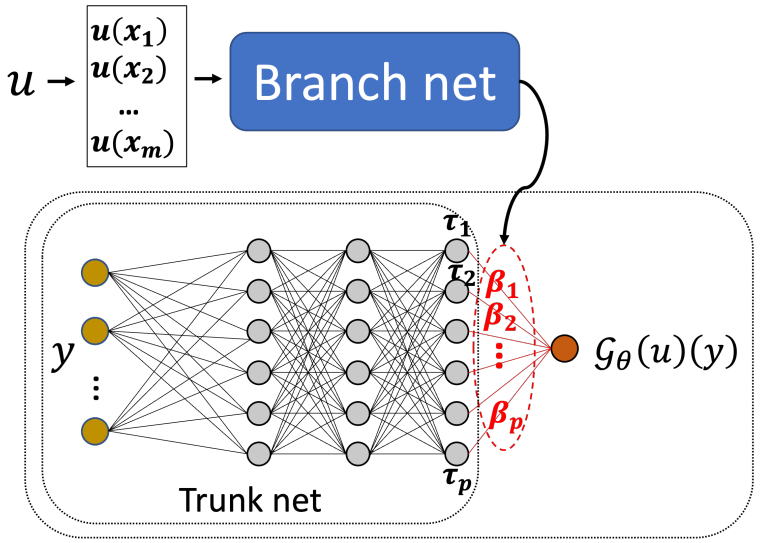
Target network

DeepONet



=

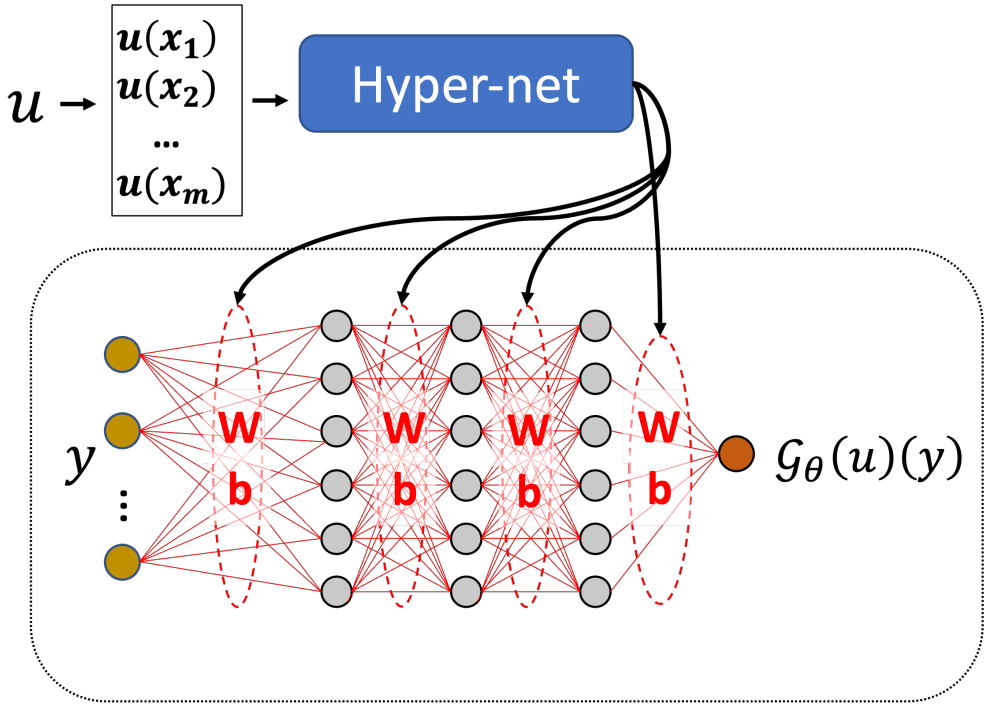
New perspective of DeepONet



Target network

[ICLR 2023] HyperDeepONet

- We propose a general model **HyperDeepONet**, which contains all the previo us models, as a special case of the HyperDeepONet.
- The **hypernetwork** generates all parameters of the target network.



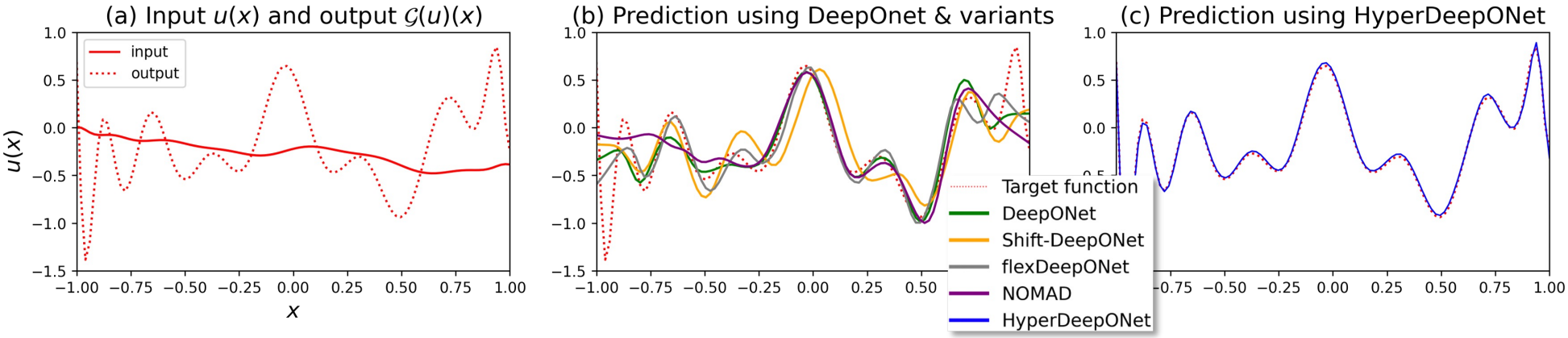
Target network

HyperDeepONet (Our model)

[ICLR 2023] HyperDeepONet

Toy example

- $G: u(x) \mapsto \frac{d}{dx}u(x)$ ($u \sim$ **Chebyshev degree 20**)
- **We choose the differentiation operator to focus on operator learning when the operator's output function space is complicated.**



[SIAM SISC 2025] Finite Element Operator Network(FEONet)

SIAM J. SCI. COMPUT.
Vol. 47, No. 2, pp. C501–C528

FINITE ELEMENT OPERATOR NETWORK FOR SOLVING ELLIPTIC-TYPE PARAMETRIC PDEs*

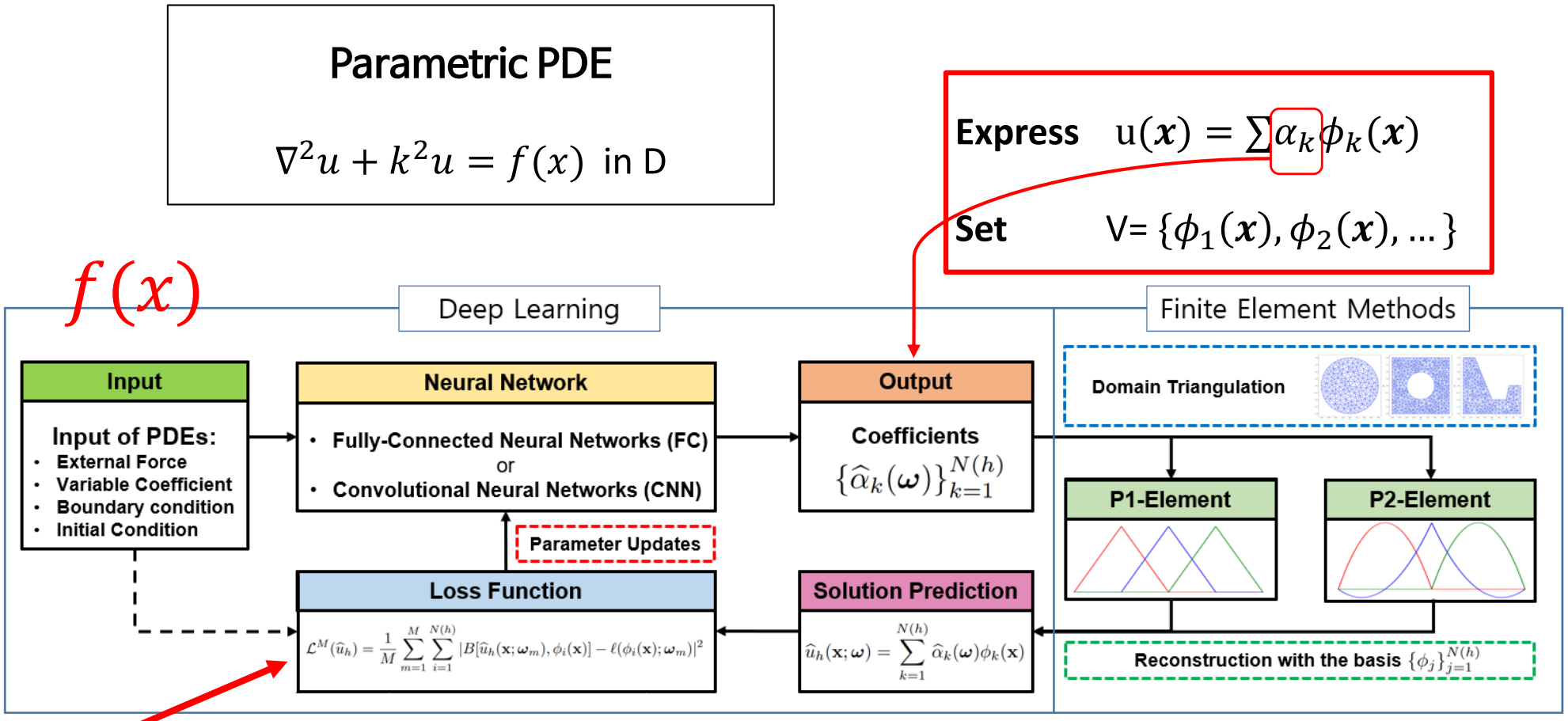
JAE YONG LEE†, SEUNGCHAN KO‡, AND YOUNGJOON HONG§

Abstract. Partial differential equations (PDEs) underlie our understanding and prediction of natural phenomena across numerous fields, including physics, engineering, and finance. However, solving parametric PDEs is a complex task that necessitates efficient numerical methods. In this paper, we propose a novel approach for solving parametric PDEs using a Finite Element Operator Network (FEONet). Our proposed method leverages the power of deep learning in conjunction with traditional numerical methods, specifically the finite element method, to solve parametric PDEs in the absence of any paired input-output training data. We performed various experiments on several benchmark problems and confirmed that our approach has demonstrated excellent performance across various settings and environments, proving its versatility in terms of accuracy, generalization, and computational flexibility. While our method is not meshless, the FEONet framework shows potential for application in various fields where PDEs play a crucial role in modeling complex domains with diverse boundary conditions and singular behavior. Furthermore, we provide theoretical convergence analysis to support our approach, utilizing finite element approximation in numerical analysis.

Key words. scientific machine learning, finite element methods, physics-informed operator learning, parametric PDEs, boundary layer

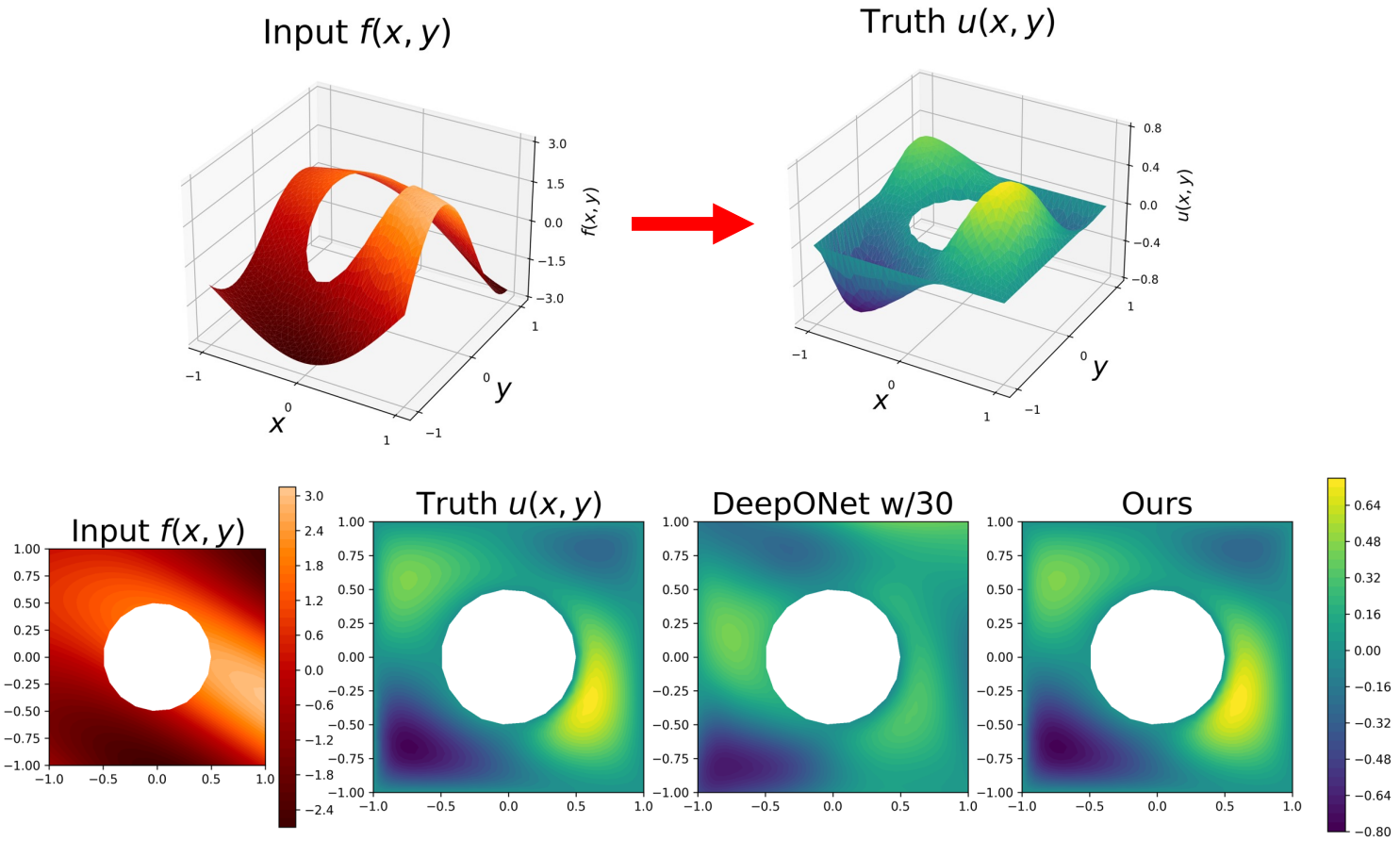
[SIAM SISC 2025] Finite Element Operator Network(FEONet)

- Schematic diagram of our model



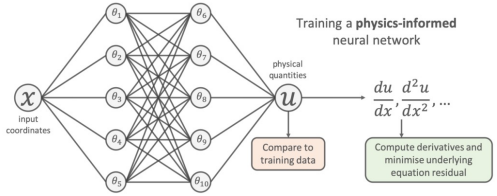
[SIAM SISC 2025] Finite Element Operator Network(FEONet)

- Results



[ICML 2026] Unbiased and Second-Order-Free Training for High-Dimensional PDEs

Question.



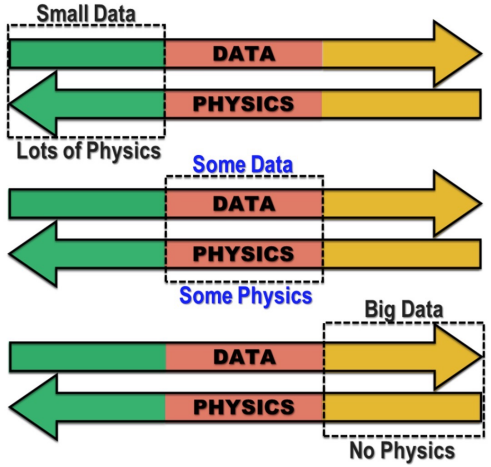
Q. Can we efficiently solve high-dimensional PDEs?

Physics

Computational Fluid Dynamics (CFD)

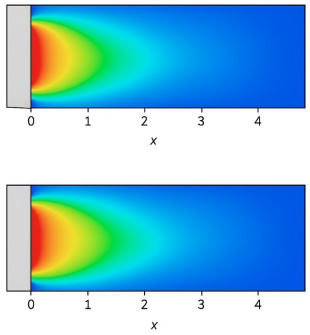
(e.g., Navier-Stokes equation)

$$\rho\left[\frac{\partial V}{\partial t} + (V \cdot \nabla)V\right] = -\nabla p + \rho\vec{g} + \mu\nabla^2\vec{V}$$



Data

(Experimental results)



[ICML 2026] Unbiased and Second-Order-Free Training for High-Dimensional PDEs

https://surinlee.github.io/Un-EM-BSDE/

[ICML 2026] Unbiased and Second-Order-Free Training for High-Dimensional PDEs

- Existing approaches inherently suffer from discretization-induced bias for finite time step sizes Δt.
- Moreover, recent approaches that address this issue often require second-order computations, resulting in significantly increased computational cost.
- To address these issues, we propose the Unbiased EM-BSDE method (Un-EM-BSDE), which is both unbiased and second-order-free.

Table 1. Summary of bias properties, computational requirements, and training time.

Method	Unbiased	2nd-order-free	Time
EM-BSDE (Raissi, 2024)	✗	✓	1x
Shotgun (Xu & Zhang, 2025)	✗	✓	0.75x
Multi-Shot EM-BSDE	✗	✓	1.74x
Heun-BSDE (Park & Tu, 2025)	✓	✗	42.91x
FS-PINNs (Park & Tu, 2025)	✓	✗	32.07x
Un-EM-BSDE (ours)	✓	✓	**1.79x**

[ICML 2026] Unbiased and Second-Order-Free Training for High-Dimensional PDEs

Table 2. Mean RL2 errors ($\times 10^{-2}$) with standard deviations ($\times 10^{-2}$) over three random seeds. The best and second-best results are highlighted in bold with underline and bold, respectively. For AC, the error is computed only at $t = 0$.

Cases	Biased methods			Unbiased methods		
	EM-BSDE	Shotgun	Multi-Shot EM-BSDE	Heun-BSDE	FS-PINNs	**Un-EM-BSDE (Ours)**
Soft constraint						
HJB	0.4055±0.0107	1.1409±0.1588	0.1617±0.0144	0.1424±0.0102	**0.0867**±0.0018	**0.1348**±0.0153
BSB	0.3483±0.0008	39.9934±23.6762	0.1046±0.0024	0.1030±0.0036	**0.0478**±0.0015	**0.0814**±0.0026
AC	0.0462±0.0154	0.0951±0.1270	**0.0206**±0.0010	0.0774±0.0027	0.0325±0.0018	**0.0147**±0.0044
BZ	5.3633±0.0338	86.5295±115.6677	**5.3379**±0.0304	6.1697±0.0845	174.4345±120.3304	**5.1778**±0.0533
PIDE	1.1094±0.0753	1.2072±0.0344	1.0771±0.0314	**0.9199**±0.0448	**0.4973**±0.0097	1.0878±0.0481
Hard constraint						
HJB	0.3433±0.0051	0.2036±0.0110	**0.1172**±0.0086	0.2655±0.0876	0.1876±0.0487	**0.1444**±0.0275
BSB	0.3456±0.0002	0.1629±0.0016	0.0739±0.0002	0.0201±0.0016	**0.0048**±0.0009	**0.0120**±0.0004
AC	0.0400±0.0142	0.0136±0.0016	**0.0113**±0.0029	0.0912±0.0004	0.0301±0.0004	**0.0034**±0.0024
BZ	1.1912±0.0083	0.2383±0.0403	1.1952±0.0070	1.1885±0.0027	**0.2129**±0.0063	**0.2279**±0.0010
PIDE	0.0374±0.0155	0.4057±0.0205	0.0245±0.0087	0.1874±0.0162	**0.0137**±0.0022	**0.0226**±0.0028

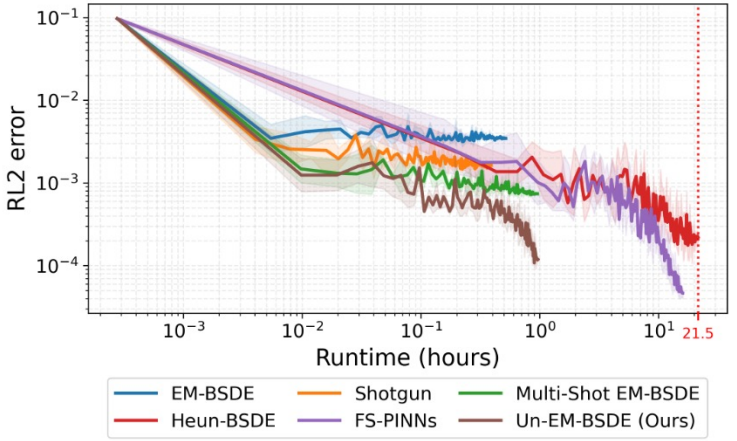


Figure 3. Runtime–accuracy on the BSB benchmark: RL2 recorded over runtime. Both axes are plotted on a log scale.

[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

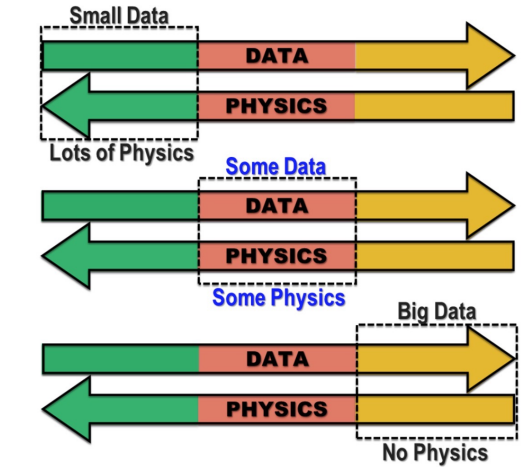
Question.

Physics

Computational Fluid Dynamics (CFD)

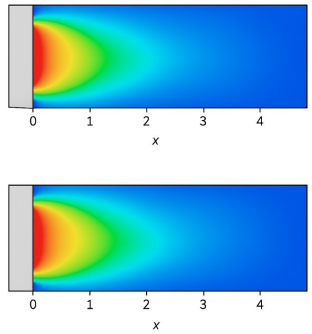
(e.g., Navier-Stokes equation)

$$\rho\left[\frac{\partial V}{\partial t}+(V\cdot\nabla)V\right]=-\nabla p+\rho\vec{g}+\mu\nabla^2\vec{V}$$

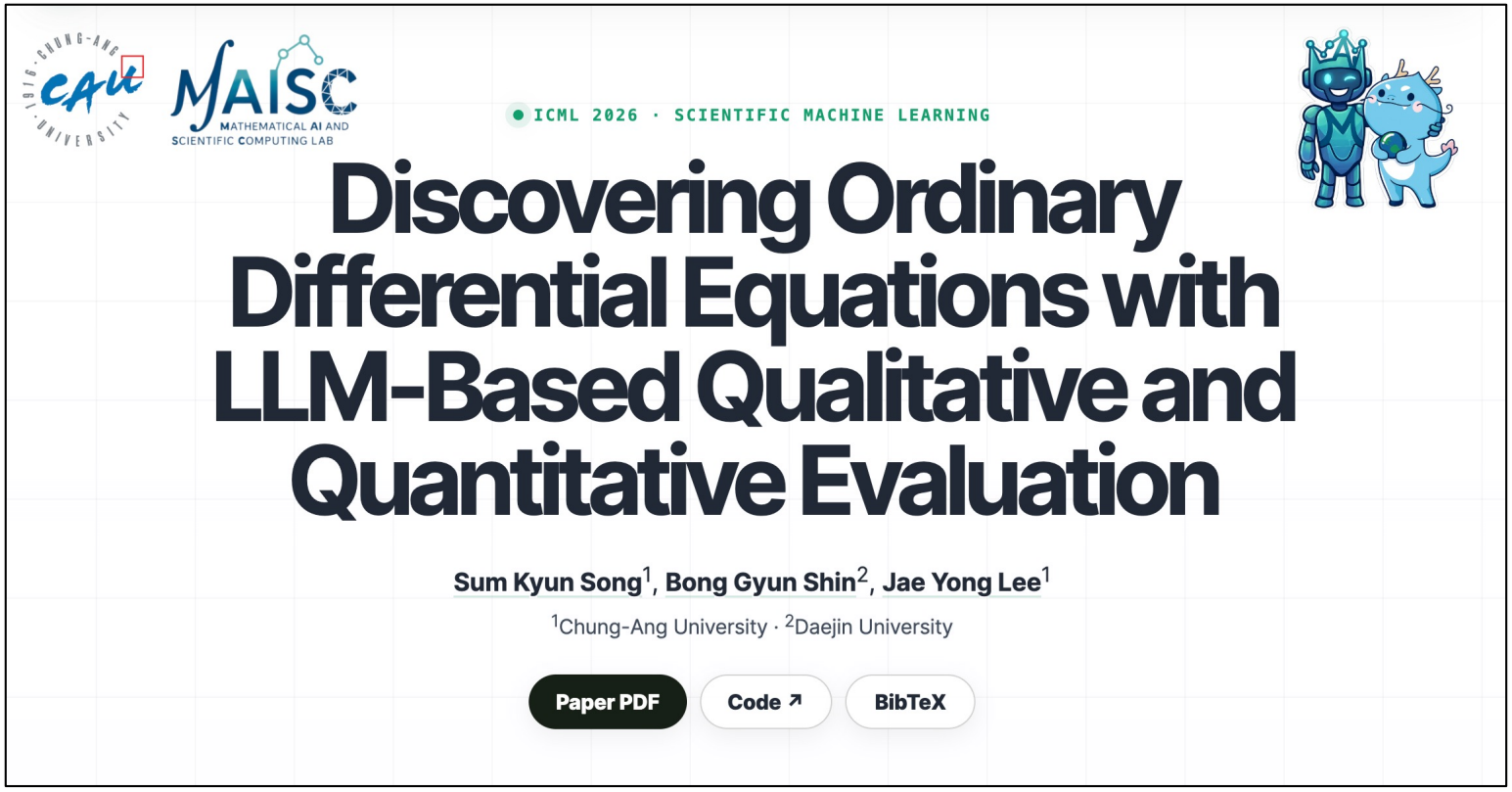


Data

(Experimental results)



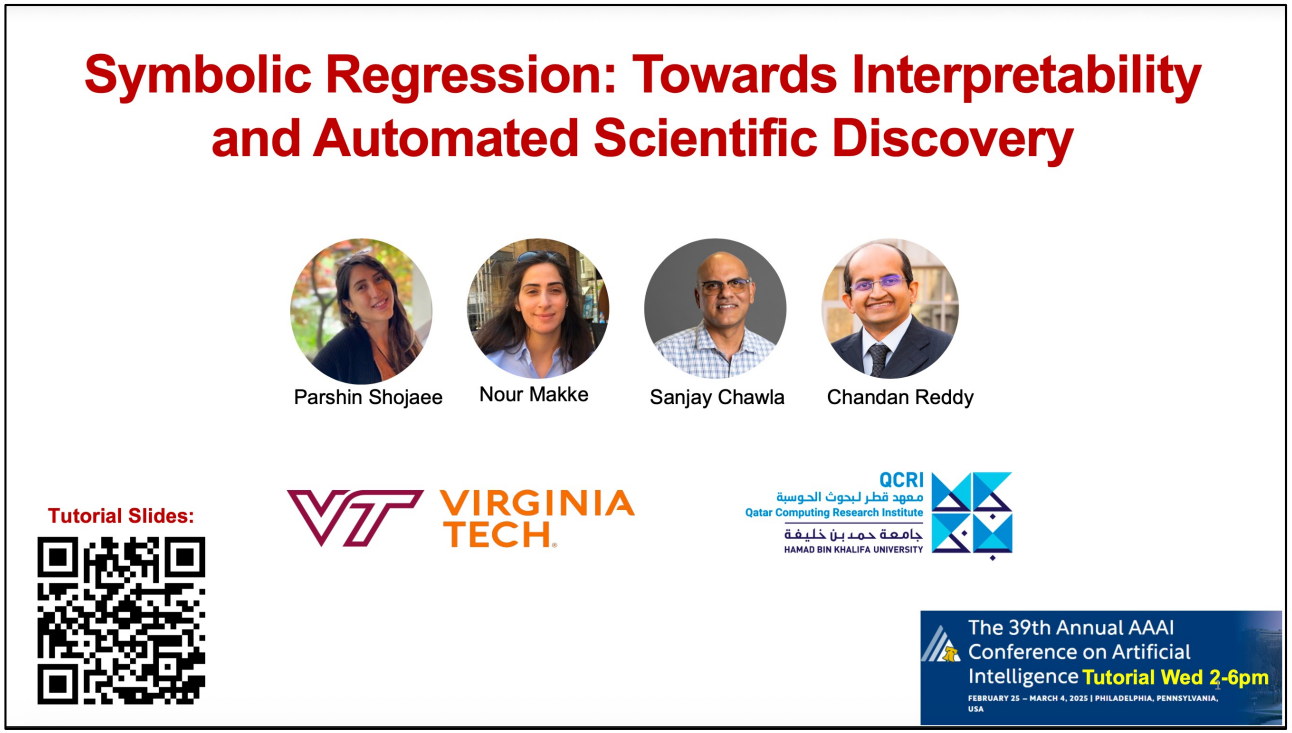
Q. Can LLMs help discover governing PDEs by incorporating domain-specific context and physical principles?



https://bon99yun.github.io/DoLQ/

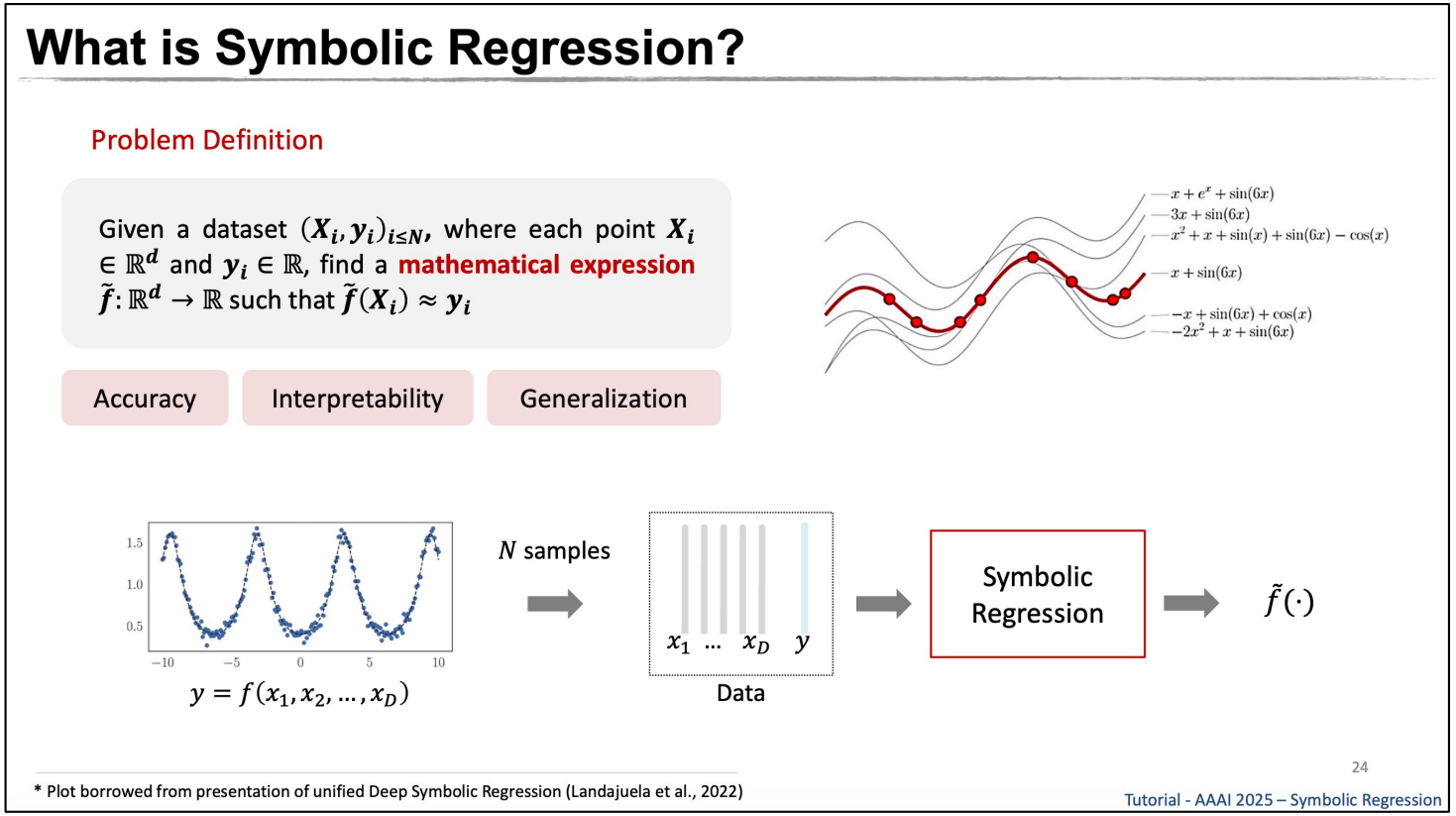
[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

- **Motivation:** https://symbolicregression2025.github.io/



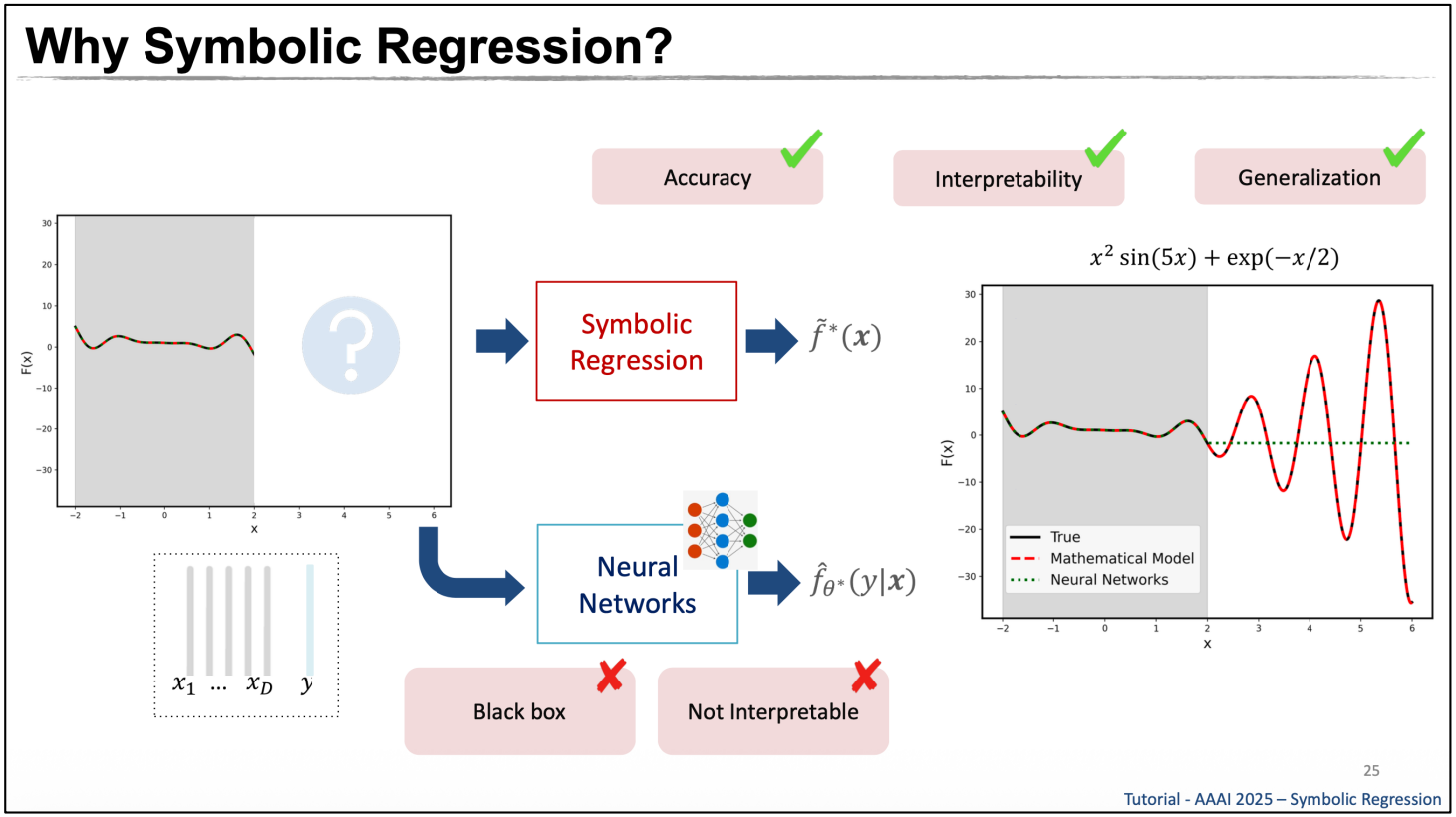
[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

- **Motivation:** https://symbolicregression2025.github.io/



[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

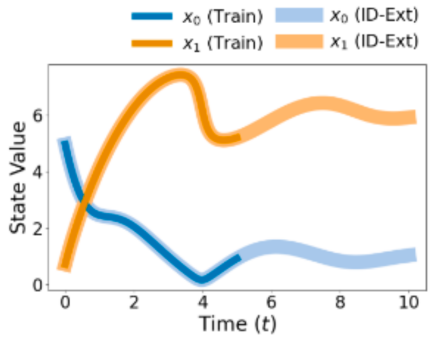
- **Motivation:** https://symbolicregression2025.github.io/



[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

- **Problem setting:** Discovering governing ODEs from trajectory data and semantic descriptions

Problem Definition



+ Description of data

How can we find the ODE?

$$\dot{x}_0 = -0.2x_0^2 - \sin(x_1)$$

$$\dot{x}_1 = x_0 - \frac{\cos(x_1)}{x_0}$$

[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

- **Motivation:** Toy example!

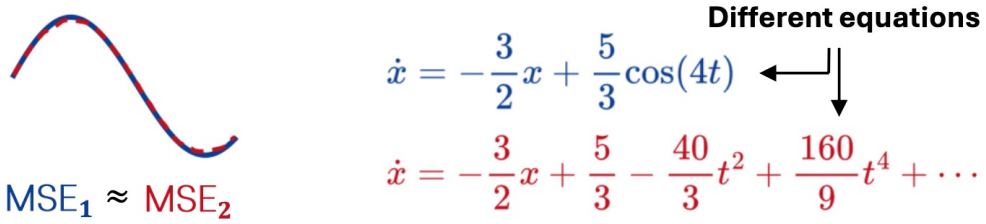


Figure 1. Similar MSE does not imply correct discovery: even with a low MSE, the identified equation may differ from the true one, suggesting that quantitative metrics alone are insufficient and that qualitative evaluation is therefore necessary.

[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

- We propose an LLM-driven iterative framework for ODE discovery that combines symbolic generation, numerical optimization, and qualitative scientific reasoning.

DoLQ: **D**iscovers **O**rdinary differential equations with **L**LM-based **Q**ualitative and **Q**uantitative evaluation

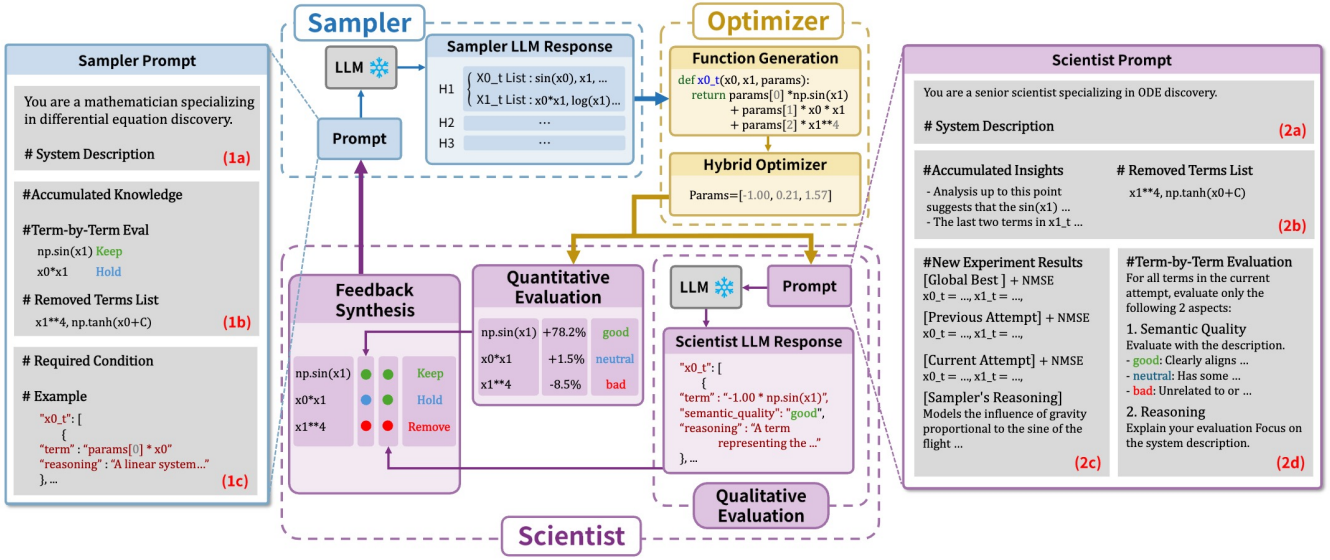


Figure 2. Overview of the DoLQ framework for LLM-based ODE discovery. The framework operates through an iterative loop among three components: (1) the Sampler Agent proposes candidate terms with physical justifications based on the system description and Scientist Agent's feedback; (2) the Parameter Optimizer makes functions and fits their parameters; and (3) the Scientist Agent evaluates each term through qualitative semantic assessment and quantitative iterative comparison, determining actions that guide subsequent iterations.

[ICML 2026] Discovering Ordinary Differential Equations with LLM-Based Qualitative and Quantitative Evaluation

- **Result:**

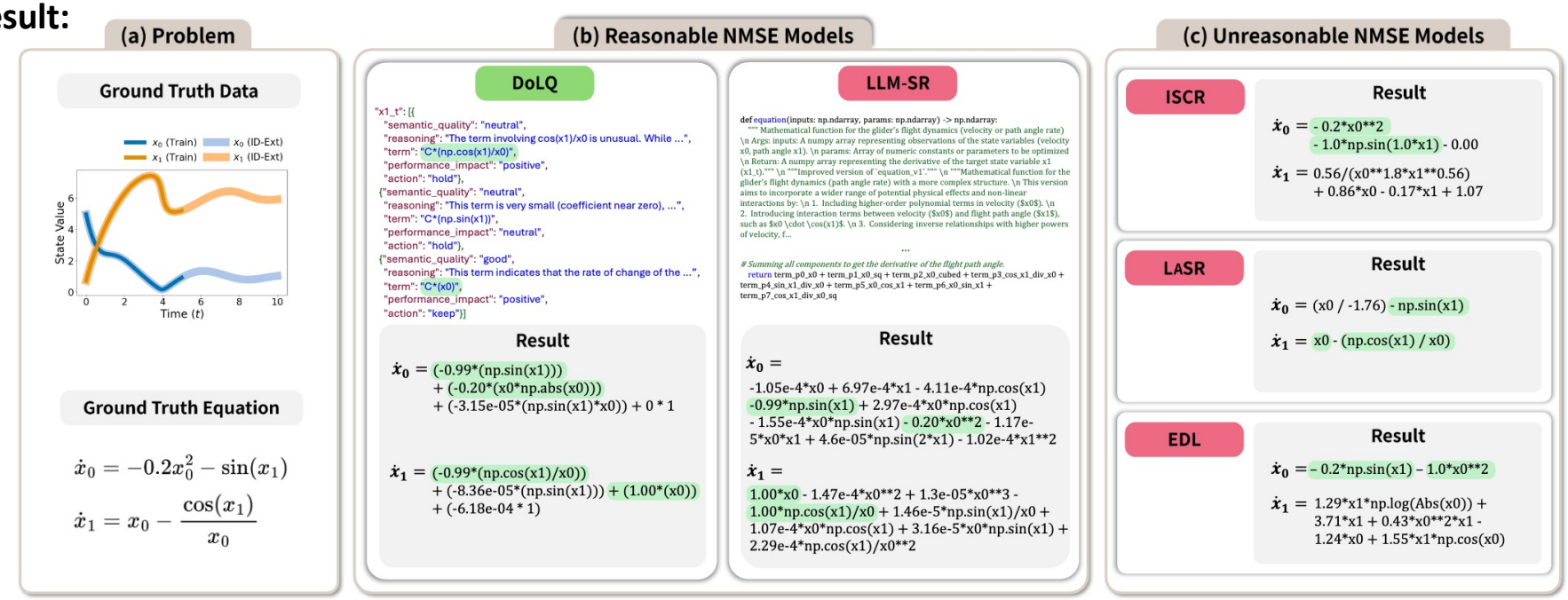


Figure 3. Equation comparison on the 2D dimensionless Glider system. Ground truth terms are highlighted in green. Group (b) shows models with reasonable NMSE: DoLQ achieves integral NMSE $< 10^{-3}$ across all dimensions, and LLM-SR exhibits the lowest NMSE among the remaining methods. Group (c) shows models with higher NMSE. Each model's discovered equation terms and their corresponding optimized coefficients are displayed.

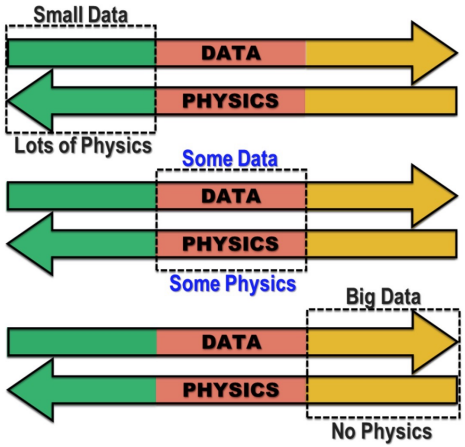
Conclusion

Physics

Computational Fluid Dynamics (CFD)

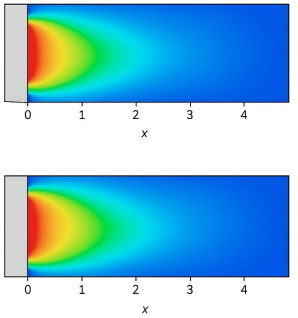
(e.g., Navier-Stokes equation)

$$\rho\left[\frac{\partial V}{\partial t} + (V \cdot \nabla)V\right] = -\nabla p + \rho\vec{g} + \mu\nabla^2\vec{V}$$



Data

(Experimental results)



By bridging the gap between physical principles and data, Scientific ML revolutionizes efficiency and generalization in physics simulations

Thank you for listening!

Personal Homepage: https://www.jaeyong-lee.com/

Lab Homepage: https://sites.google.com/view/maisc-lab/

E-mail: jaeyong@cau.ac.kr