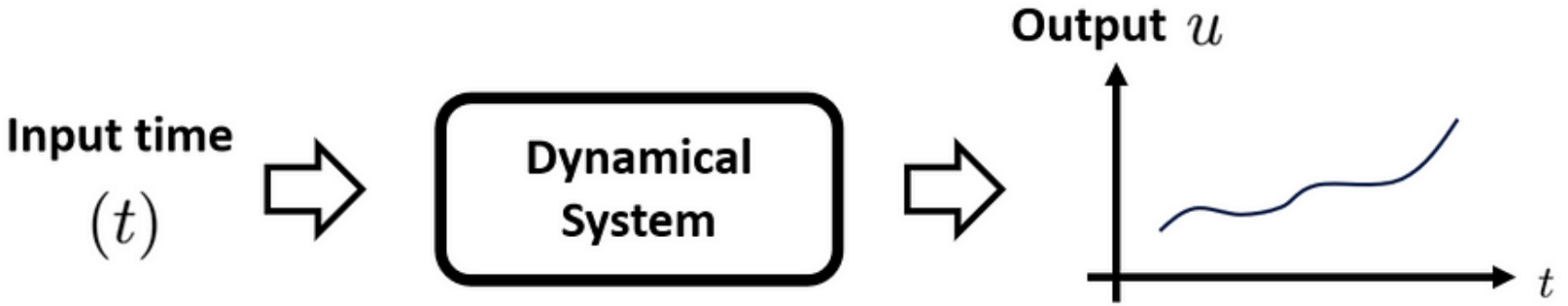
THE UNIVERSITY of TENNESSEE KNOXVILLE

Lecture 16: NeuralODEs, DeepONets, and Neural Operators

Sergei V. Kalinin

Neural Differential Equations



Partially known differential equations

$$\begin{cases} \dfrac{du}{dt} = a \cdot u + b \\ \dfrac{du}{dt} = f(t, u) \end{cases}$$

Unknown parameters

Unknown functions

https://medium.com/towards-data-science/modeling-dynamical-systems-with-neural-ode-a-hands-on-guide-71c4cfdb84dc

Neural Differential Equations

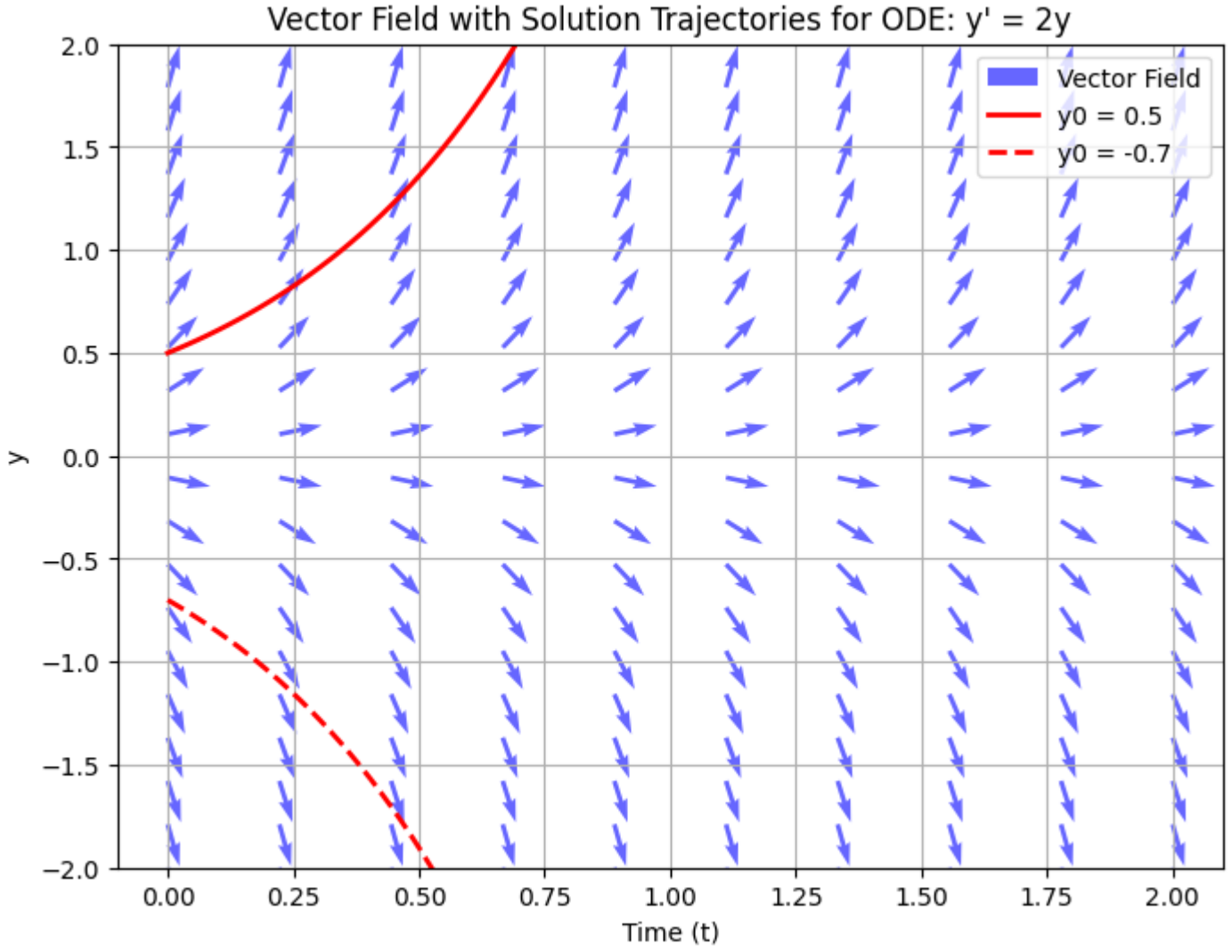
$$\frac{du}{dt} = f(u,t) \quad \Rightarrow \quad \frac{du}{dt} \approx f_{NN}(u,t)$$

Key idea: Instead of discretizing, use a continuous-depth neural model.

Why is this useful?
- Handles irregular time-series data.
- Provides adaptive computation (solves for arbitrary time points).
- Enables physics-informed learning by integrating known dynamics.

Solving Neural Differential Equations

Forward Pass: Solve ODE using a numerical integrator (RK, Euler, etc.).



Use the forward difference to approximate dx/dt

$$\frac{dx}{dt} \approx \frac{x(t+h) - x(t)}{h}$$

gives the **Euler method** of solving the equation for x(t)

$$x(t+h) = x(t) + h\, f[x(t), t]$$

ResNet and Euler Methods

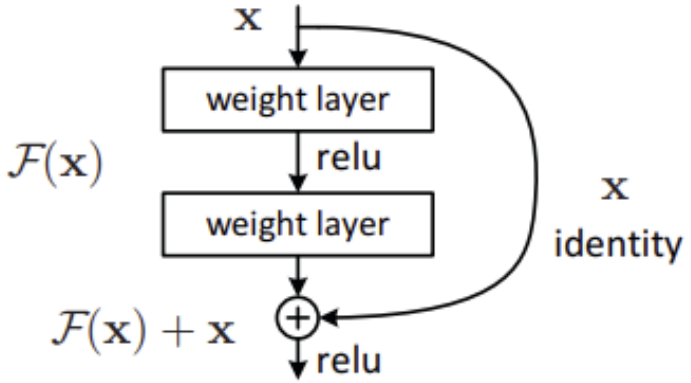


Figure 2. Residual learning: a building block.

$$\mathbf{h}_{t+1} = \mathbf{h}_t + \mathbf{F}(\mathbf{h}_t; \theta_t)$$

Computer Science > Computer Vision and Pattern Recognition

[Submitted on 10 Dec 2015]

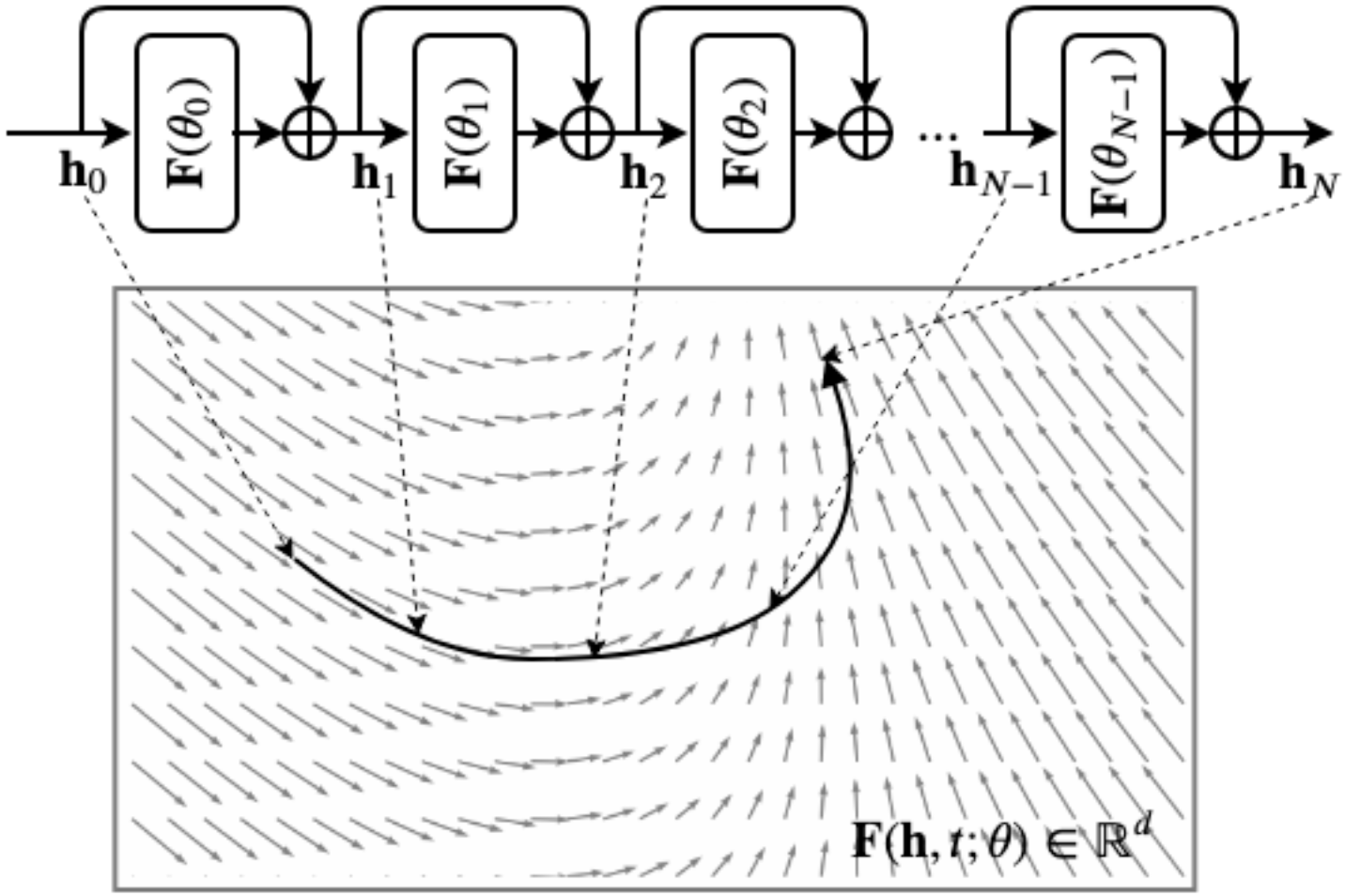
Deep Residual Learning for Image Recognition

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers---8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

NeuralODE: Forward Pass



$$\mathbf{h}_N = \text{ODESolve}(start\_state = \mathbf{h}_0, dynamics = \mathbf{F}, t\_start = 0, t\_end = N; \theta)$$

https://ayandas.me/blogs/2020-03-20-neural-ode.html

NeuralODE: Backward Pass

To avoid storing all intermediate states, the adjoint method treats the gradient computation as another ODE and solves it backward in time.

The gradient at the end of the ODE layer is

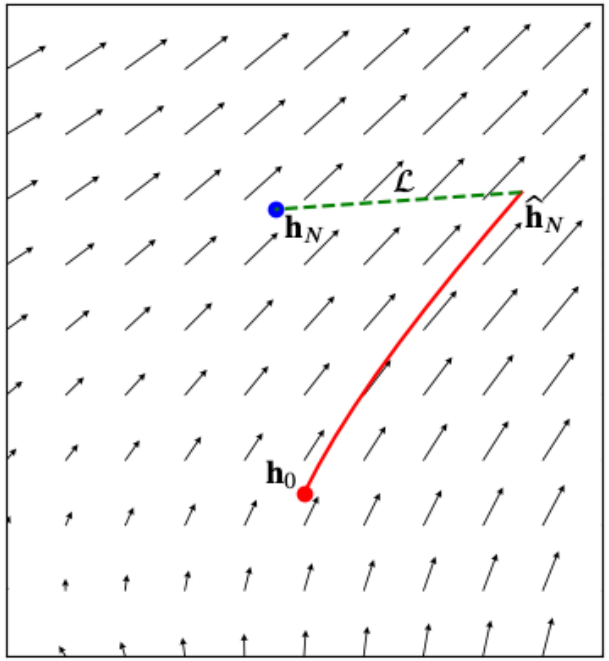
$$\frac{d\tilde{\mathcal{L}}}{d\mathbf{h}_N}$$

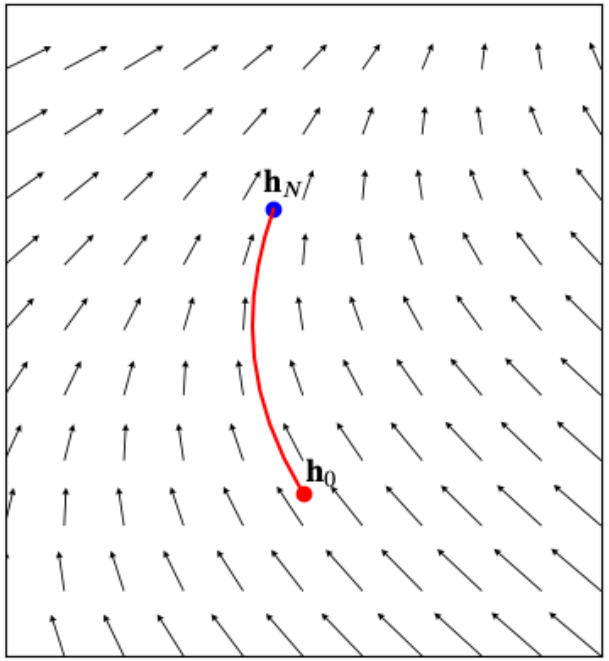
where L is a scalar loss.

All we have to do is use this incoming gradient to compute

$$\frac{d\mathcal{L}}{d\theta}$$

and perform an SGD (or any variant) step.





NeuralODE: Backward Pass

Instead of backpropagation, define the adjoint variable $a(t)$ as:

$$\mathbf{a}_t \triangleq \frac{d\mathcal{L}}{d\mathbf{h}_t}$$

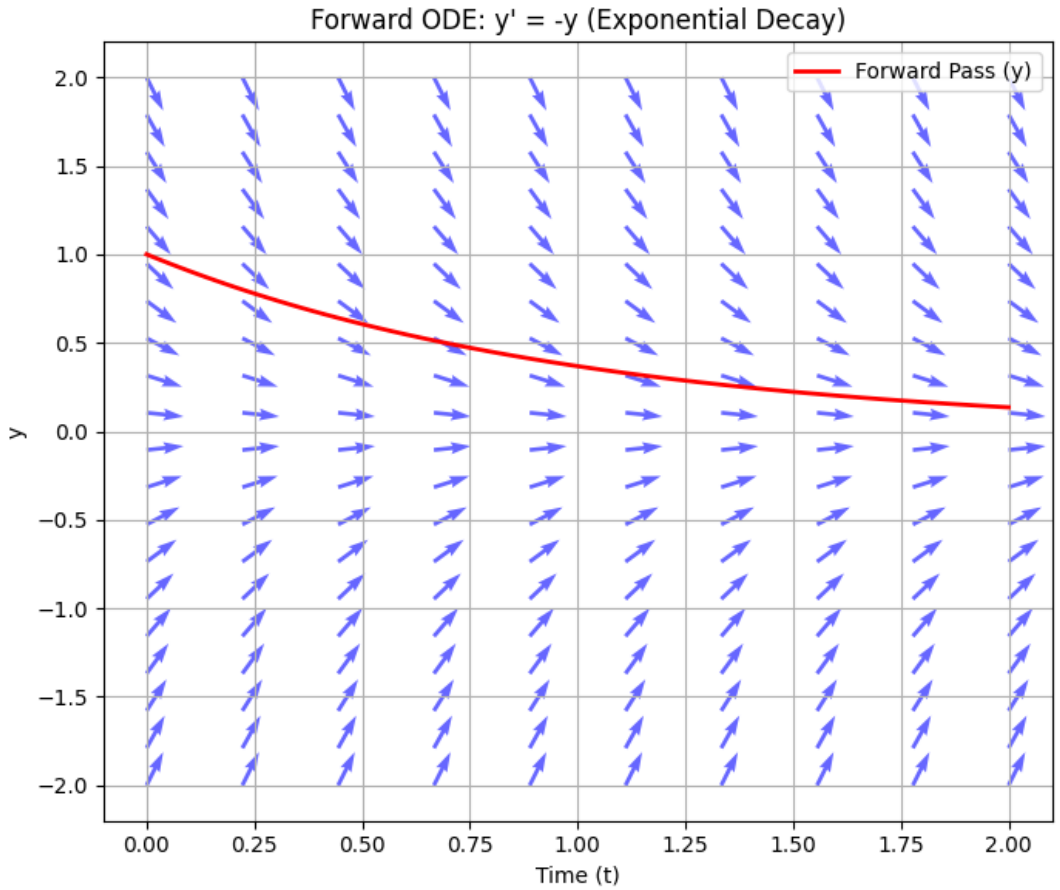
Using the chain rule and integration by parts, we get another ODE for $a(t)$:

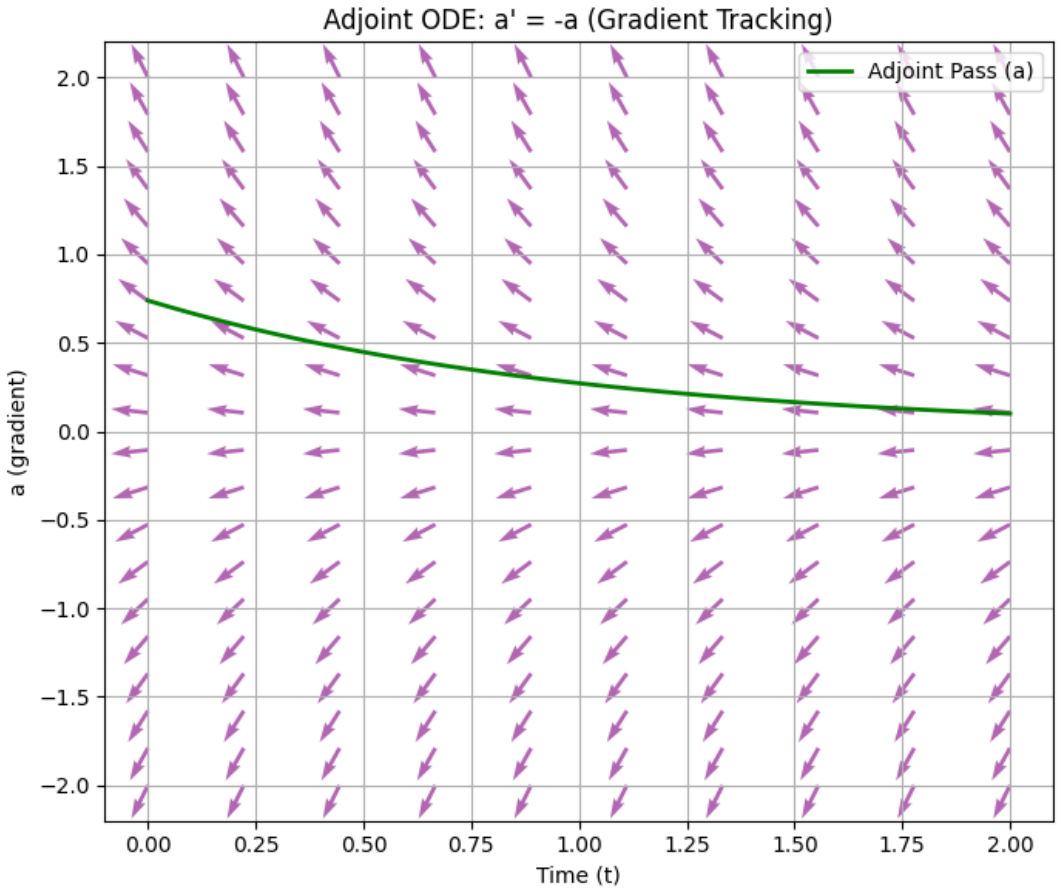
$$\mathbf{F}_a(\mathbf{a}_t, \mathbf{h}_t, t, \theta) \triangleq \frac{d\mathbf{a}_t}{dt} = -\mathbf{a}_t \frac{\partial \mathbf{F}}{\partial \mathbf{h}_t}$$

Thus, we solve this adjoint ODE backward in time from t=T to t=0, instead of storing intermediate states.

$$\mathbf{a}_{N-1}, \cdots, \mathbf{a}_0 = \mathrm{ODESolve}(\mathbf{a}_N, \mathbf{F}_a, N, 0; \theta)$$

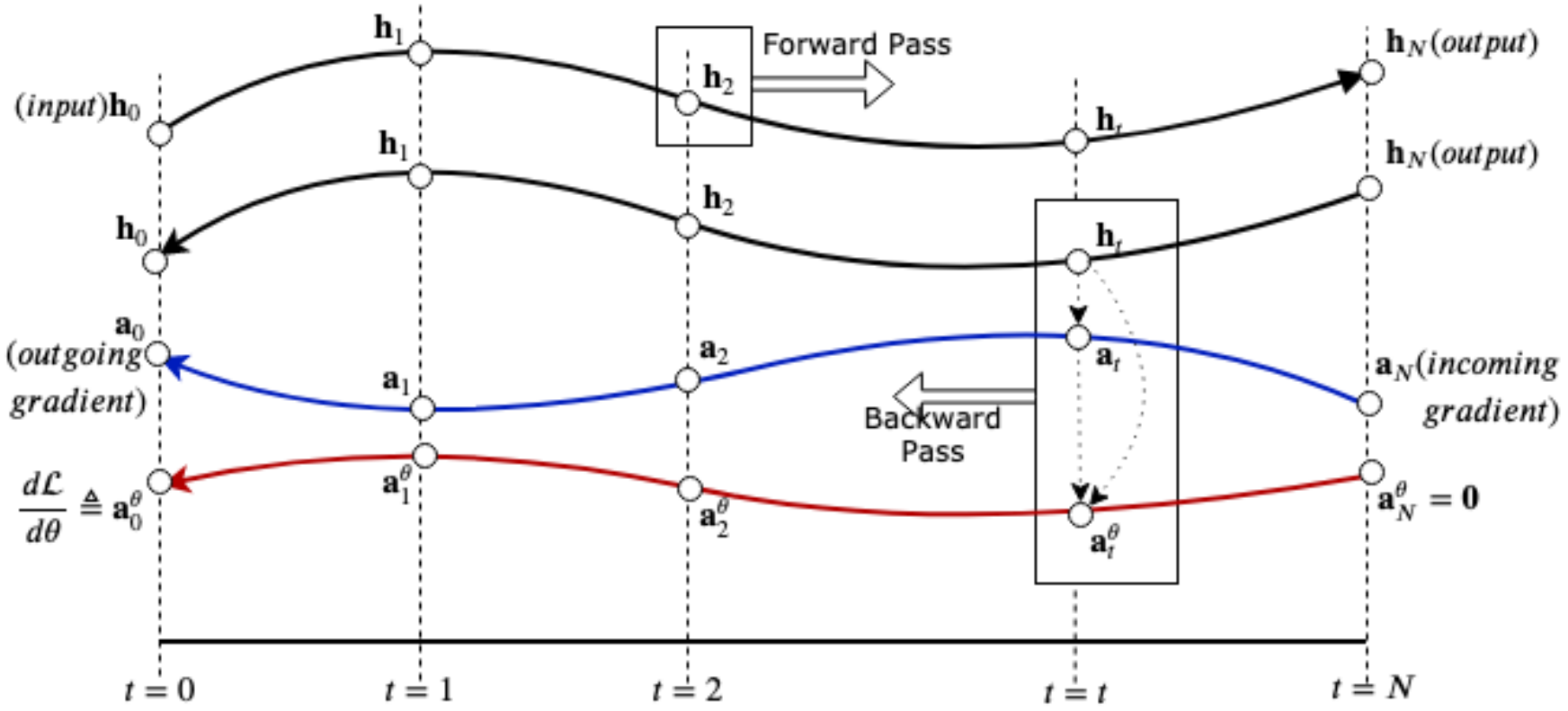
- No need to store intermediate ODE states.
- Works well with adaptive ODE solvers, unlike backpropagation.
- Can be applied to continuous-depth models (e.g., Normalizing Flows, time-series modeling).


Forward ODE: y' = -y (Exponential Decay)
Forward Pass (y)
y
Time (t)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Adjoint ODE: a' = -a (Gradient Tracking)
Adjoint Pass (a)
a (gradient)
Time (t)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

Solving Neural Differential Equations

$$[\mathbf{a}_{N-1}; \mathbf{h}_{N-1}; \_], \cdots, [\mathbf{a}_0; \mathbf{h}_0; \mathbf{a}_0^\theta] = \text{ODESolve}([\mathbf{a}_N; \mathbf{h}_N; \mathbf{0}], [\mathbf{F}_a; \mathbf{F}; \mathbf{F}_\theta], N, 0; \theta)$$



https://ayandas.me/blogs/2020-03-20-neural-ode.html

Learning from Data

$$dx(t)/dt = f_\theta(x(t), t)$$

where f_θ is a neural network with trainable parameters θ.
The model learns f_θ such that it **matches observed data points** in time-dependent systems.

1. **Collecting data:** Given time-series data $(x_0, x_1, ..., x_T)$; we assume they follow some unknown continuous dynamics.
2. **Forward solving the ODE:**
   - The model starts at **x0** and numerically integrates the learned function f_θ.
   - This produces an estimated trajectory $\hat{x}(t)$
3. **Computing the loss:**
   - Compare the predicted trajectory $\hat{x}(t)$ with actual observations $x(t)$
   - A loss function like Mean Squared Error (MSE) is used: $L = \sum_i ||\hat{x}(t_i) - x(t_i)||^2$
4. **Backpropagating through time:**
   - Instead of backpropagating through an entire computational graph, Neural ODEs use the **adjoint method** to efficiently compute gradients and update parameters θ.

✅ **Handles Noisy & Missing Data** → It can learn smooth representations despite measurement gaps.
✅ **Adaptive Time Steps** → Neural ODEs can predict at any time point, even between observed data points.
✅ **Generalization** → Once trained, they can **predict system behavior beyond observed time points**.

Learning from Data

1️⃣ **Physics-Informed Loss Functions** – Penalize deviations from known equations in the loss function.
2️⃣ **Hybrid Models (Neural ODE + Known Equations)** – Use partial equations and let the Neural ODE learn the missing part.
3️⃣ **Encoding Symmetries and Conservation Laws** – Enforce known physical laws (e.g., energy conservation, Hamiltonian dynamics).
4️⃣ **Data-Driven Discovery of Unknown Parameters** – Use known equations but let the model learn uncertain parameters.
5️⃣ **Regularization for Smoothness and Stability** – Add constraints to avoid physically unrealistic behavior.
6️⃣ **Using Known Boundary and Initial Conditions** – Constrain Neural ODE predictions based on observed conditions.
7️⃣ **Learning in Latent Space** – Encode domain knowledge in a latent representation before solving the Neural ODE.

Population Dynamics

$$\frac{dp(t)}{dt} = rp(t)(1 - \frac{p(t)}{k}) - H(p(t))$$

$p(t)$ is the **population size** at time t,
r is the intrinsic **growth rate** of the population,
k is the **carrying capacity** of the environment.

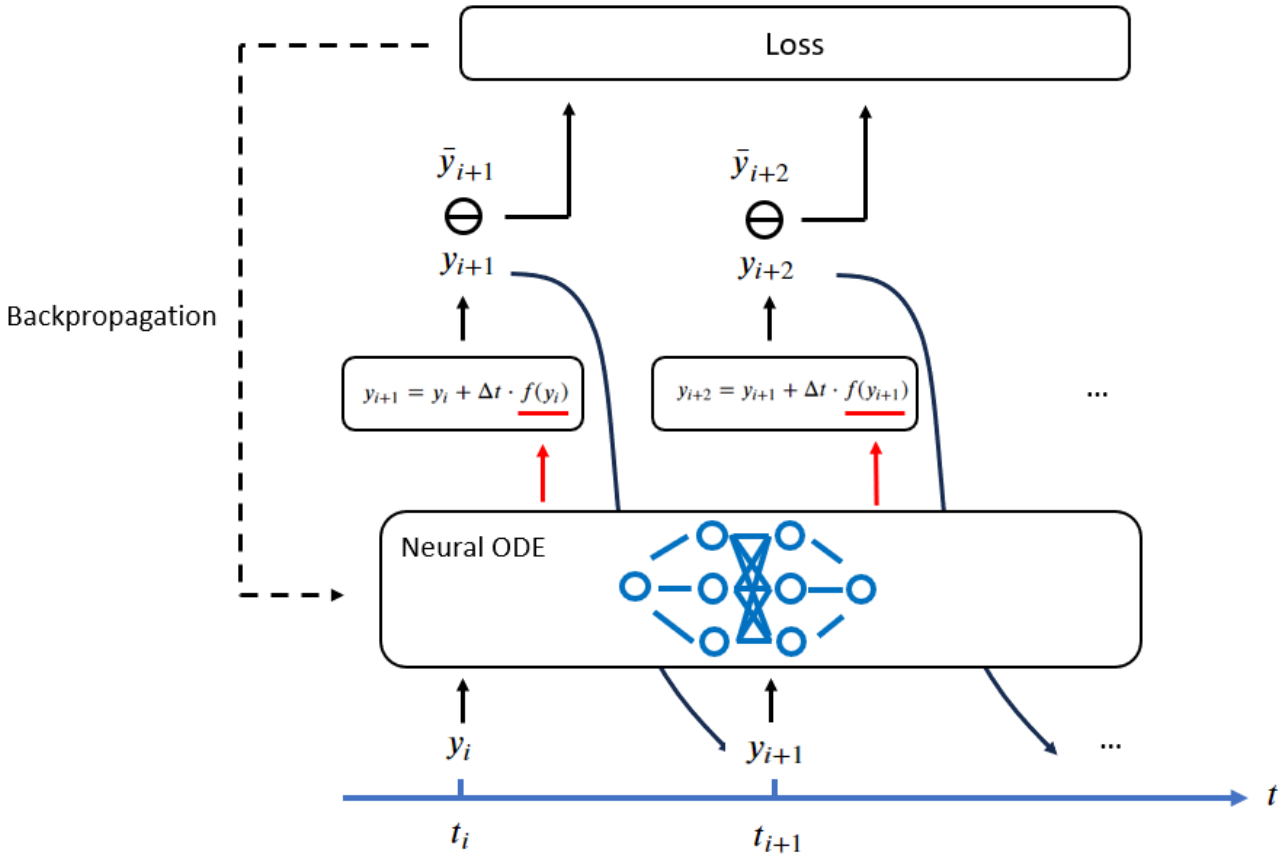
$$H(p(t)) = min(h \cdot p, H_{max})$$

$H(\cdot)$ is the harvesting function that reflects the removal of the animal population due to harvesting (or culling). Here h denotes the harvesting rate and Hmax represents the maximum harvesting capacity.

https://medium.com/towards-data-science/modeling-dynamical-systems-with-neural-ode-a-hands-on-guide-71c4cfdb84dc

Neural Differential Equations

$$\frac{dp}{dt} = f_{unknown}(p) - min(h \cdot p, H_{max})$$



https://medium.com/towards-data-science/modeling-dynamical-systems-with-neural-ode-a-hands-on-guide-71c4cfdb84dc

Population Numerical vs. Neural ODEs

Feature	Numerical ODE	Neural ODE
Solving Known ODEs	✅ Well-suited for explicit equations	✅ Can approximate solutions but requires training
Handling Noisy Data	❌ Struggles with noise and missing data	✅ Learns from noisy data and handles gaps well
Data-Driven Discovery	❌ Requires explicit formulation	✅ Learns hidden dynamics directly from data
Adaptive Computation	✅ Adaptive solvers exist but can be computationally expensive	✅ Naturally adaptive and efficient with adjoint methods
Model Predictive Control (MPC)	✅ Used in control but limited to predefined models	✅ Integrates well into MPC workflows
High-Dimensional Systems	❌ Struggles with large-scale systems	✅ Scales efficiently to high-dimensional problems
Generalization Beyond Training	❌ Cannot extrapolate well beyond observed scenarios	✅ Generalizes better if trained properly
Computational Efficiency	✅ Fast for small problems but costly for complex ones	❌ Training is expensive but efficient at inference

Applications of NeuralODE

- Scientific Computing & Physics-Based Modeling → Solving differential equations, fluid dynamics, reaction kinetics.
- Time-Series Analysis → Financial forecasting, healthcare, climate modeling, power grid optimization.
- Machine Learning & AI Architectures → Normalizing flows, sequence modeling, memory-efficient deep learning.
- Control & Robotics → Optimal control, reinforcement learning, motion planning for autonomous systems.
- Biomedical & Healthcare Applications → Neural activity modeling, drug response prediction, growth modeling.
- Anomaly Detection & Industrial Systems → Fault detection in engineering, finance, and cybersecurity.

Green's Functions

A **Green's function** is a fundamental solution used to solve **linear differential equations** with boundary conditions. It represents the **response of a system to a point source**, similar to how the **impulse response** works in signal processing.

Mathematically, for a **linear differential operator** L, the Green's function $G(x,x')$ satisfies:

$$LG(x,x')=\delta(x-x'),$$

where $\delta(x-x')$ is the **Dirac delta function**, representing a **unit impulse** at x'.

$G(x,x')$ is known, it can be used to **solve inhomogeneous differential equations** $Lu(x)=f(x)$ by integrating:

$$u(x)=\int G(x,x')f(x')dx'u(x)$$

Thus, Green's functions allow us to express the solution as a **superposition of responses to point sources**.

Green's Function in Electrostatics

Green's Function:

$$L \circ u(x) = f(x)$$

$$L \circ G(x,s) = \delta(x-s)$$

$$u(x) = \int G(x,s) f(s) ds$$

An Example:

$$\nabla^2 \varphi(\vec{r}) = -\frac{\rho(\vec{r})}{\varepsilon_0}$$

$$G(\vec{r},\vec{s}) = \frac{1}{4\pi\varepsilon_0} \frac{1}{|\vec{r}-\vec{s}|}$$

$$\varphi(\vec{r}) = \int G(\vec{r},\vec{s}) \rho(\vec{s}) d^3 s$$

https://galileo-unbound.blog/tag/greens-function/

Green's Functions depend on D and Domains

Table 10.1 Fundamental Green's Functions[a]

	Laplace ∇^2	Helmholtz[b] $\nabla^2 + k^2$	Modified Helmholtz[c] $\nabla^2 - k^2$
1-D	$\frac{1}{2}\lvert x_1 - x_2\rvert$	$-\frac{i}{2k}\exp(ik\lvert x_1 - x_2\rvert)$	$-\frac{1}{2k}\exp(-k\lvert x_1 - x_2\rvert)$
2-D	$\frac{1}{2\pi}\ln\lvert\boldsymbol{\rho}_1 - \boldsymbol{\rho}_2\rvert$	$-\frac{i}{4}H_0^{(1)}(k\lvert\boldsymbol{\rho}_1 - \boldsymbol{\rho}_2\rvert)$	$-\frac{1}{2\pi}K_0(k\lvert\boldsymbol{\rho}_1 - \boldsymbol{\rho}_2\rvert)$
3-D	$-\frac{1}{4\pi}\frac{1}{\lvert\mathbf{r}_1 - \mathbf{r}_2\rvert}$	$-\frac{\exp(ik\lvert\mathbf{r}_1 - \mathbf{r}_2\rvert)}{4\pi\lvert\mathbf{r}_1 - \mathbf{r}_2\rvert}$	$-\frac{\exp(-k\lvert\mathbf{r}_1 - \mathbf{r}_2\rvert)}{4\pi\lvert\mathbf{r}_1 - \mathbf{r}_2\rvert}$

General Features of Green's Functions

1. Linearity – The differential equation must be linear since Green's functions rely on the superposition principle.

2. Specified Differential Operator – The equation must be in the form $Lu(x)=f(x)$ where LL is a known linear operator.

3. Specified Domain – The Green's function depends on the shape and size of the domain (e.g., infinite space, bounded region, irregular geometries).

4. Specified Boundary Conditions – The solution is uniquely determined only when boundary conditions (Dirichlet, Neumann, or mixed) are clearly defined.

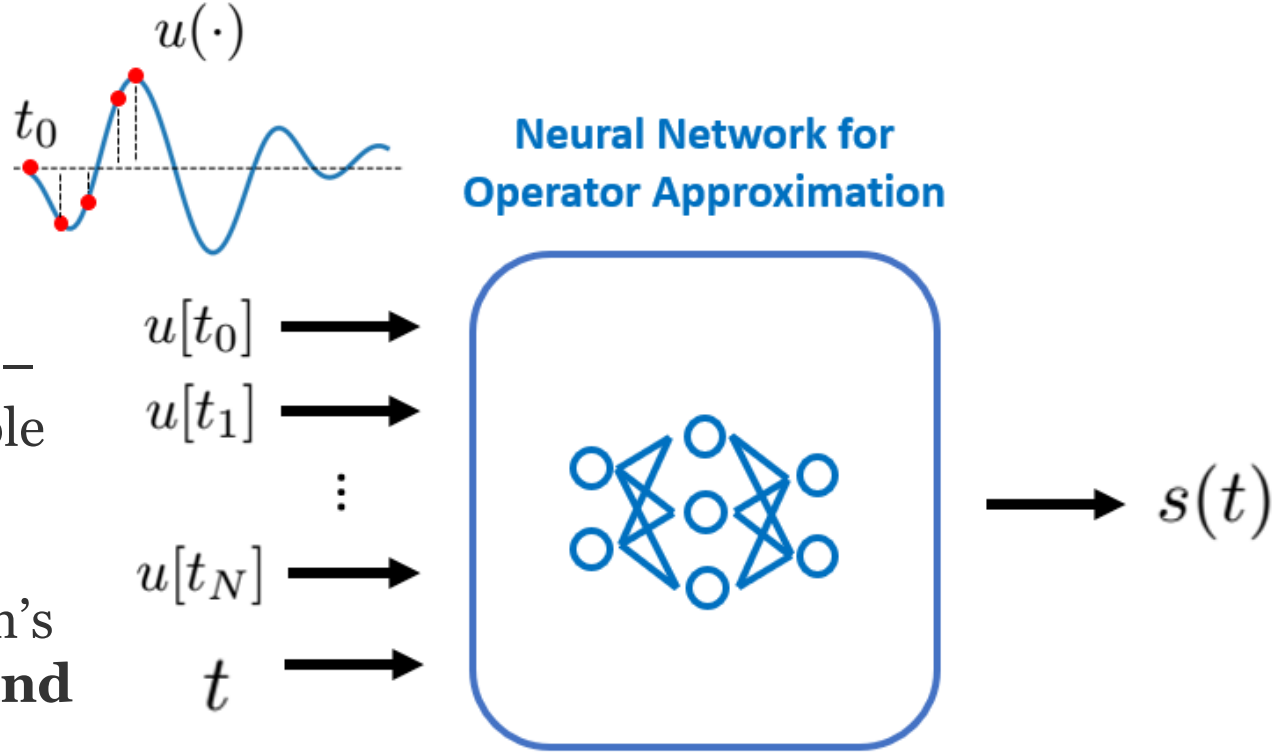
5. Existence of Fundamental Solution – The Green's function must satisfy $LG(x,x')=\delta(x-x')$ and should be well-defined in the given domain.

6. Integral Representation of the Solution – The final solution must be expressible as an integral of the Green's function weighted by the source term

7. Solvability – In simple cases, Green's functions are found analytically; for complex domains, numerical techniques (e.g., finite elements, boundary elements) may be required.

Neural Operators

- **Nonlinear PDEs** – Green's functions require linearity
- **Variable Boundary Conditions** – Green's functions require a new derivation for each set of conditions
- **Irregular and Complex Geometries** – Green's functions are often limited to simple shapes
- **High-Dimensional Problems** – Green's functions are difficult to compute for **3D and high-dimensional PDEs**
- **Hybrid Data-Driven and Physics-Based Learning** – Green's functions need explicit physics



https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

Universal Approximation Theorem

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 $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, $i = 1, \dots, n$, $k = 1, \dots, p$, $j = 1, \dots, m$, 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 $y \in K_2$.

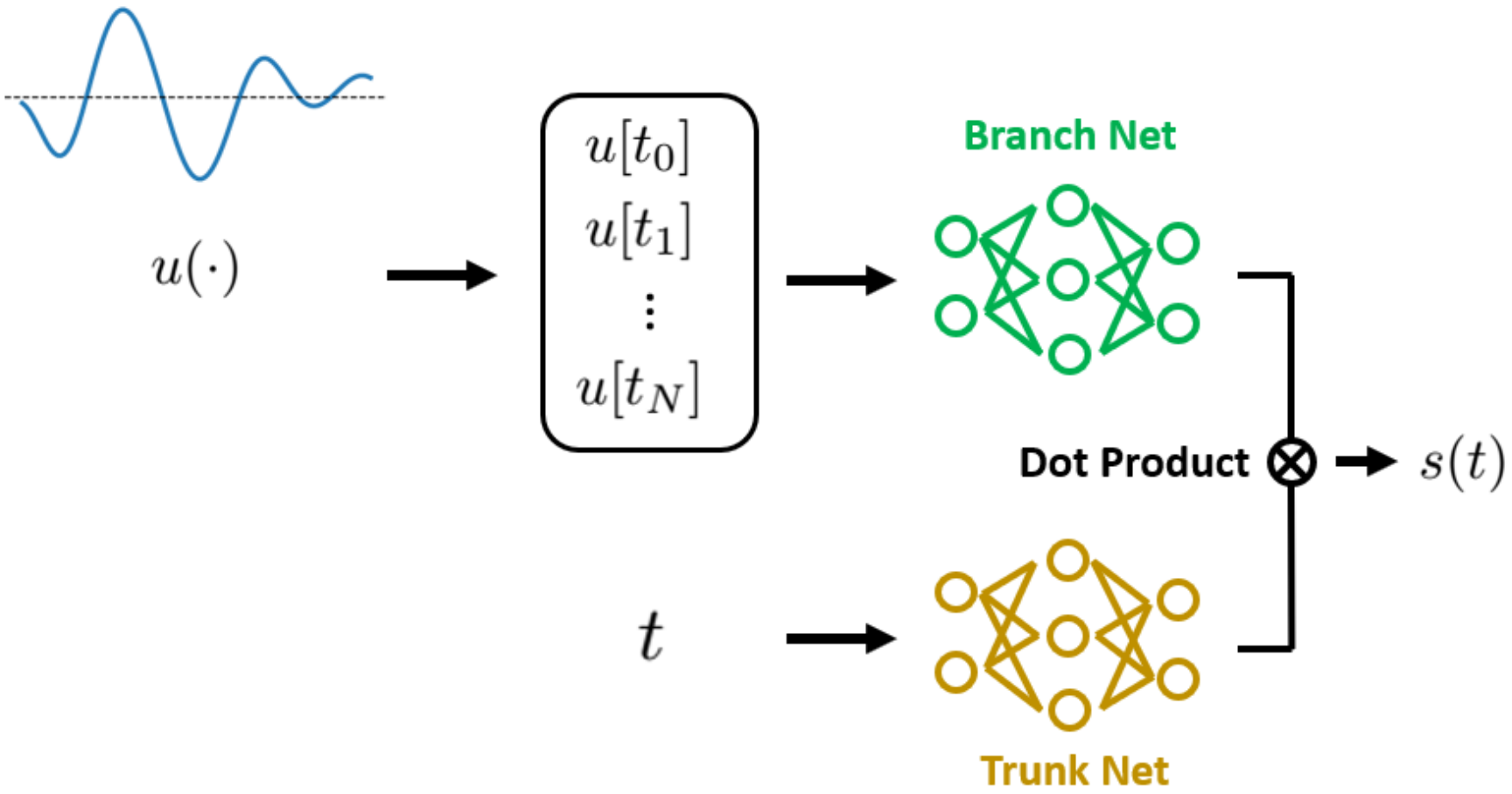
Computer Science > Machine Learning

[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

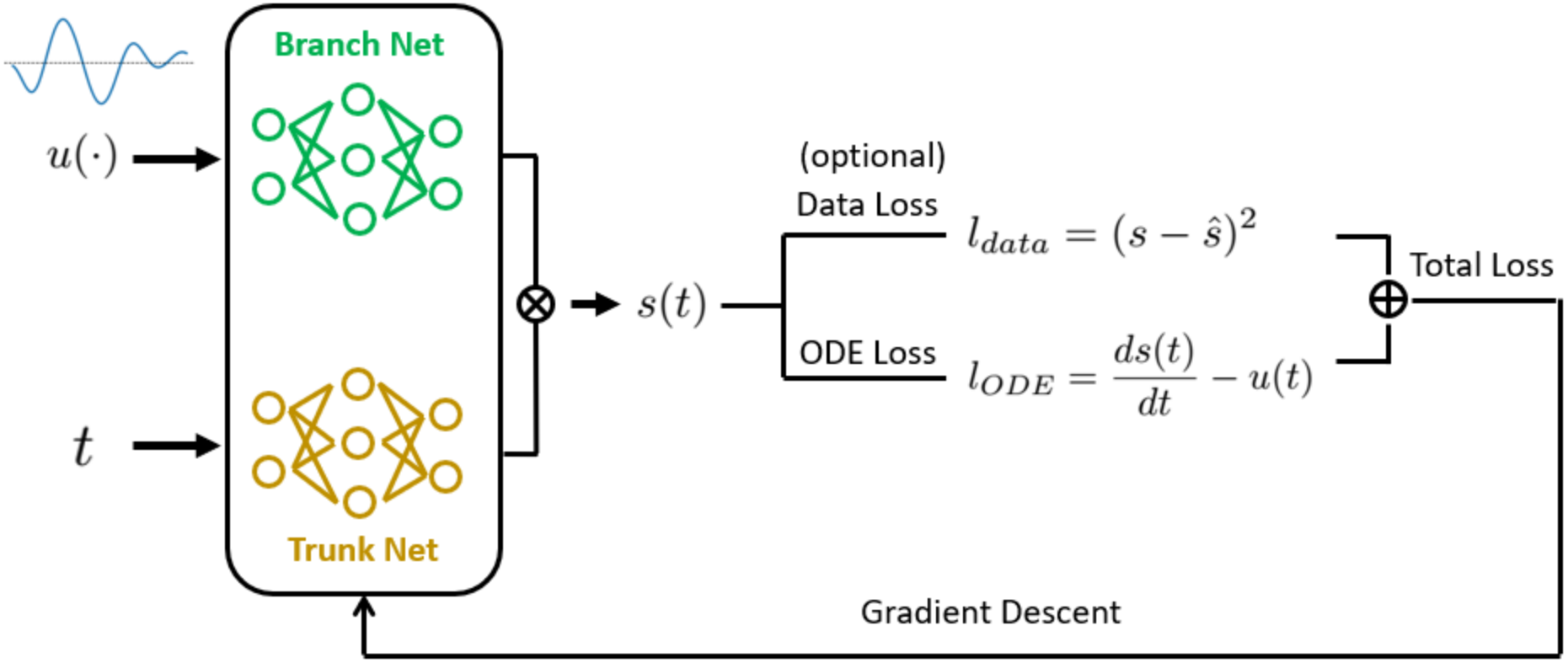
Lu Lu, Pengzhan Jin, George Em Karniadakis

DeepONet



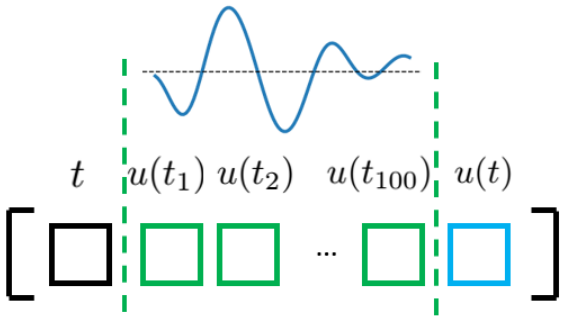
https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

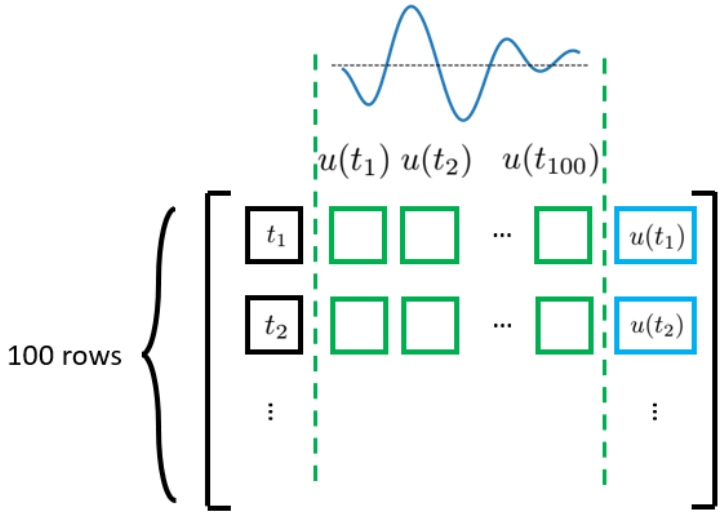
Physics-Informed DeepONet

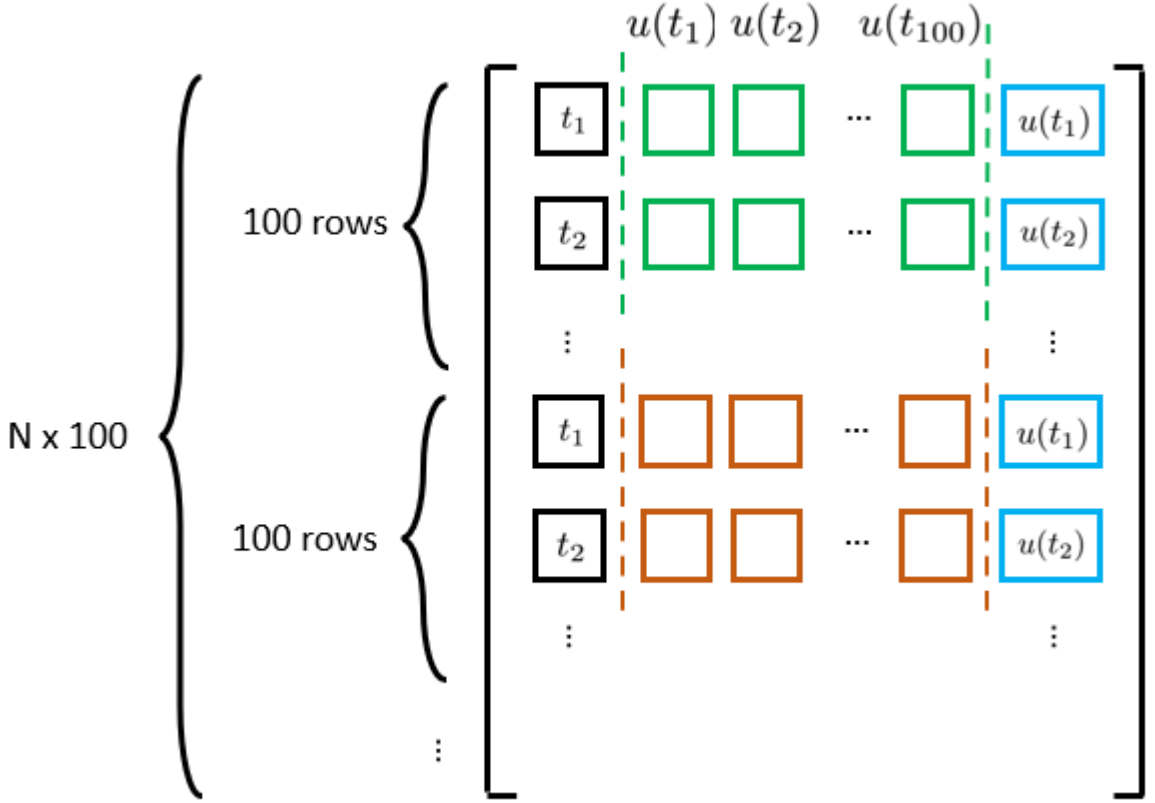


https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

DeepONet Training

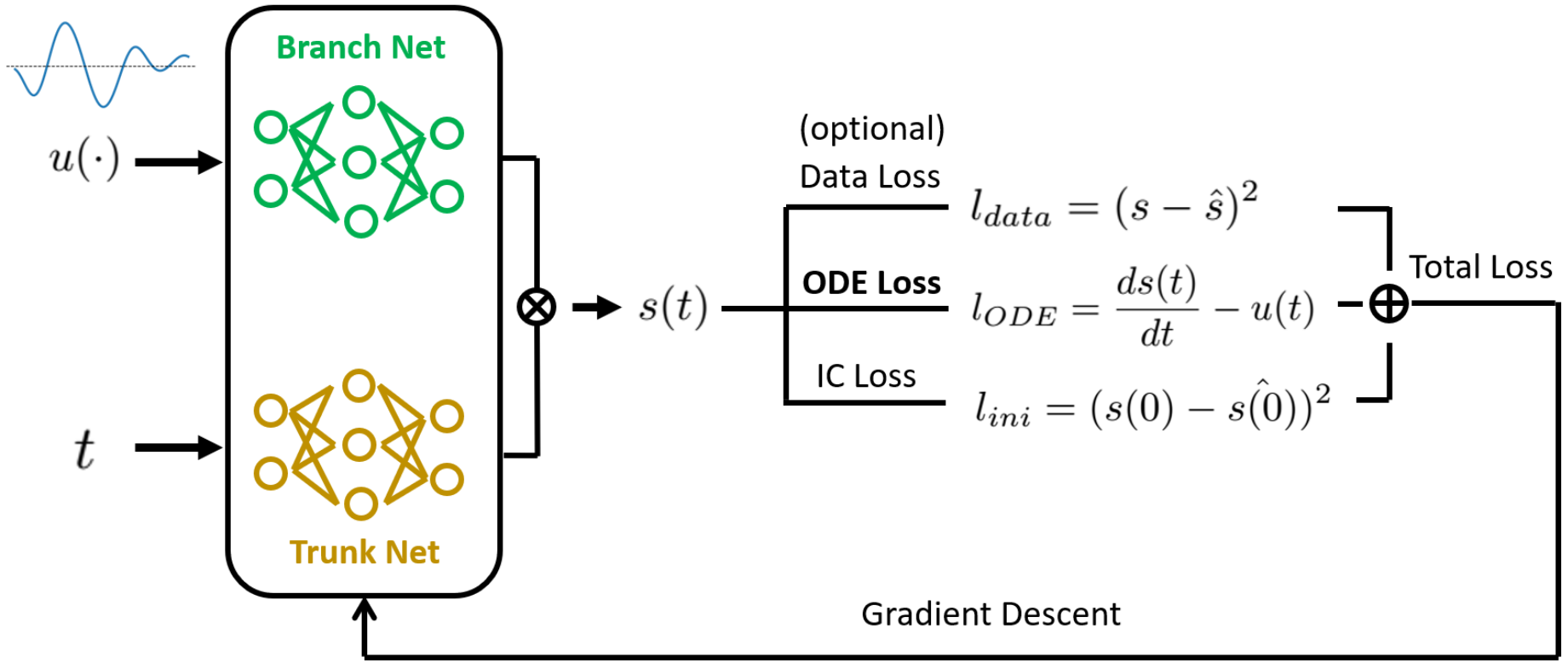






https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

Physics-Informed DeepONet



https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

Physics-Informed DeepONet Applications

Parameter discovery

$$\frac{ds(t)}{dt} = a \cdot u(t) + b,\ t \in [0, 1]$$

s(0) = 0, a and b are **unknowns**. Our objective here is to estimate the values of *a* and *b*, given the observed u(·) and s(·) profiles.

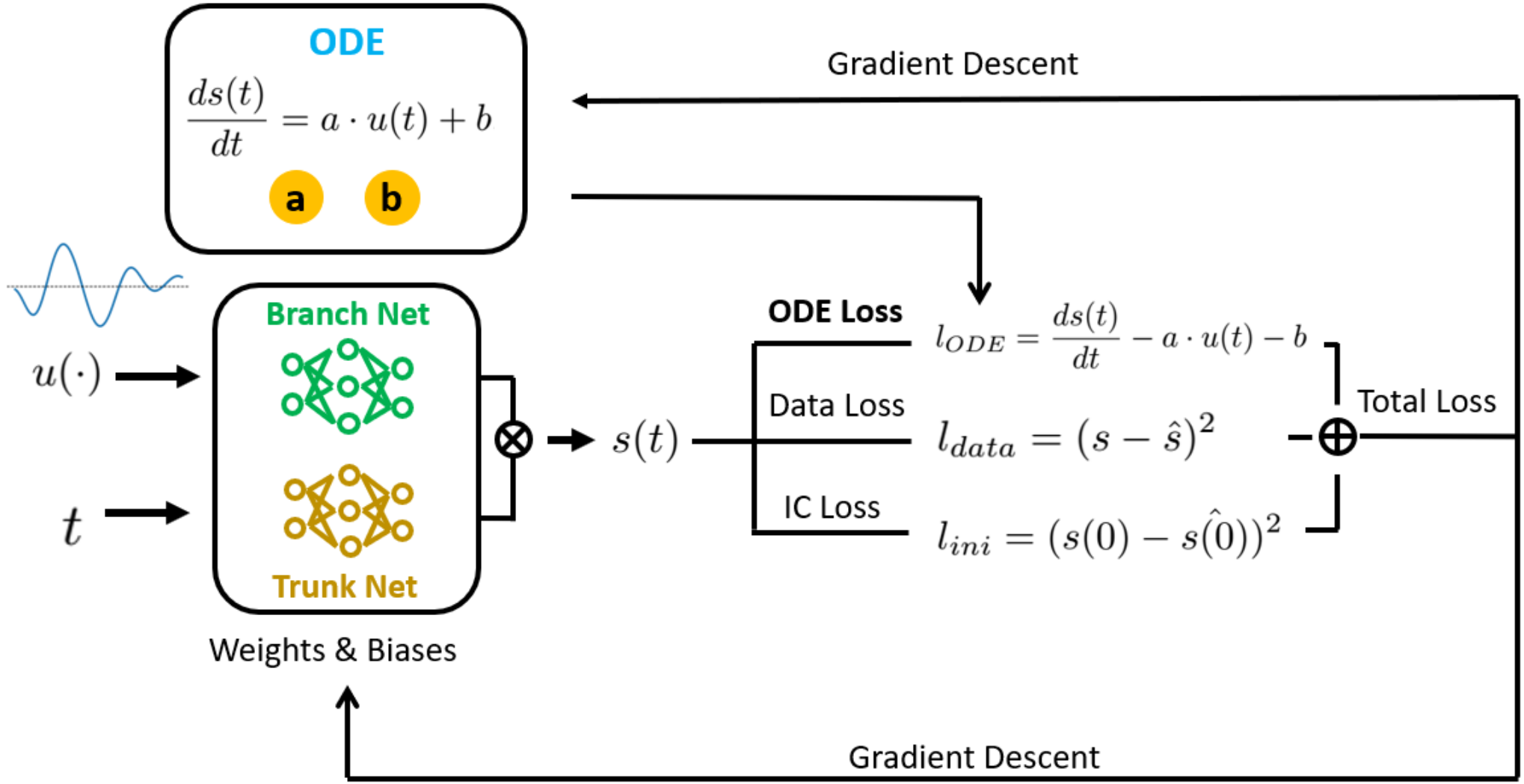
Function discovery

$$\frac{ds(t)}{dt} = 0.5 \cdot u(t) + 0.5,\ t \in [0, 1]$$

initial condition s(0) = 0, a=0.5, b=0.5. We aim to learn u(t)

https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

Physics-Informed DeepONet Applications



https://towardsdatascience.com/solving-inverse-problems-with-physics-informed-deeponet-a-practical-guide-with-code-implementation-27795eb4f502/

Green's Functions vs. PI-DeepONets

• **Analytical Exact Solutions** – Green's functions provide closed-form solutions, while DeepONets are approximations.
• **Infinite Domains** – Green's functions can naturally handle infinite space problems, while DeepONets require **training on finite domains**.
• **Singular Solutions** – Green's functions handle **point singularities** well, while DeepONets may struggle with sharp features.
• **Universal Applicability** – Green's functions work for any system that satisfies the linear PDE, whereas DeepONets must be trained for each problem.
• **Lack of Data Dependency** – Green's functions can be derived directly from physics, while DeepONets require **training data or simulations**.

- **Linearity -** Green's functions require linearity, while DeepONets can approximate solutions for nonlinear equations.
- **Variable Boundary Conditions** – Green's functions require a new derivation for each set of conditions, while DeepONets can generalize across different boundaries.
- **Irregular and Complex Geometries** – Green's functions are often limited to simple shapes, whereas DeepONets can learn solutions in **arbitrary domains**.
- **High-Dimensional Problems** – Green's functions are difficult to compute for **3D and high-dimensional PDEs**, while DeepONets scale more efficiently.
- **Hybrid Data-Driven and Physics-Based Learning** – Green's functions need explicit physics, but DeepONets can **combine physics with real-world data**.

Neural Operators

1️⃣ **Nonlinear PDEs –** Green's functions rely on **linearity**, and DeepONets struggle with **generalizing across nonlinear interactions**. What about **nonlinear function mappings** ?

2️⃣ **New Domains & Geometries –** Green's functions are **derived for specific shapes** (e.g., rectangles, spheres), and DeepONets are trained on **fixed-sized inputs**. What about **arbitrary domains** (without retraining)?

3️⃣ **Changing Boundary Conditions –** Green's functions require **re-derivation** for each new boundary condition, and DeepONets need **new training**. What about new **boundary variations?**

4️⃣ **High-Dimensional PDEs –** Green's functions involve **expensive integration**, and DeepONets require **training on large datasets**. How can we scale to **high-dimensional systems?**

5️⃣ **Data-Scarce Problems –** DeepONets need **large training datasets**, Can we learning continuous function mappings with fewer datapoints?

6️⃣ **Multiscale & Complex Physics –** Problems like **turbulence, climate modeling, and coupled physical systems** require models that generalize **across scales and conditions**, which Green's functions and DeepONets struggle with.

Neural Operators

arXiv > cs > arXiv:2108.08481

Computer Science > Machine Learning

[Submitted on 19 Aug 2021 (v1), last revised 2 May 2024 (this version, v6)]

Neural Operator: Learning Maps Between Function Spaces

Nikola Kovachki, Zongyi Li, Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar

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.

Neural Operators generalize across different geometries and boundary conditions without retraining - a significant advantage for applications like climate modeling and fluid dynamics

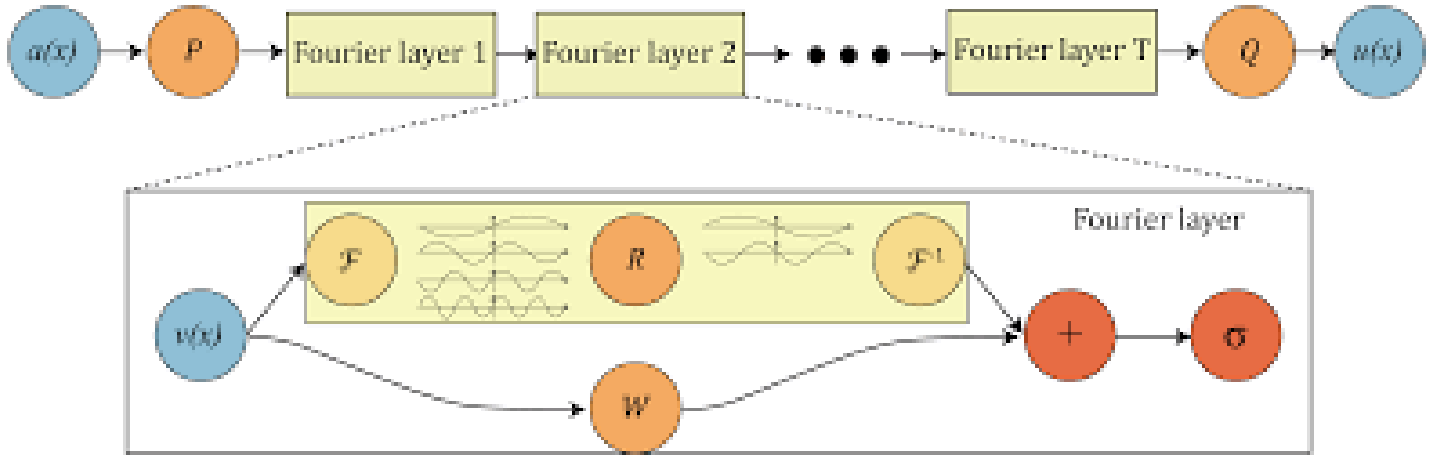
Neural Operators vs. PINNs and Solvers

Feature	Neural Operators	PINNs	Classic Solvers
What is learned?	Function-to-function mapping	Single PDE solution	Single PDE solution
Training	Data-driven	Physics-constrained	Not needed
Computational Cost	Expensive to train, fast inference	Moderate	Expensive
Generalization	Can generalize across PDE families	Solves one PDE at a time	One equation per simulation
Best for	High-dimensional PDEs, complex geometries	Data-sparse problems, physics-driven tasks	Benchmarking, high-precision cases

Neural Operators: A Peek Below the Hood

Neural Operators aim to approximate **operator mappings** between function spaces. Given an input function f(x) defined over a domain Ω, a Neural Operator learns a mapping: G:f↦u where u(x) is the solution of a differential equation governed by f(x), and G represents the **solution operator**.

Let D be a space of functions defined on a domain Ω, and let $G{:}D{\to}U$ be an operator mapping from the input function space D (e.g., forcing terms, initial conditions) to the output function space U (e.g., solution of a PDE). For a given function f, the corresponding solution is: $u{=}G(f)$. The goal of Neural Operators is to learn G from data.



Neural Operators: A Peek Below the Hood

Step 1: Lifting the Input to a High-Dimensional Space
Define a **lifting operator** PPP that embeds the input function into a higher-dimensional space:
$v_0(x)=P(f(x))$ where v_0 is an intermediate representation in an expanded feature space.

Step 2: Learning the Integral Kernel Operator
Instead of traditional convolutional layers, Neural Operators use a **global integral kernel** K_θ to model interactions between function values:

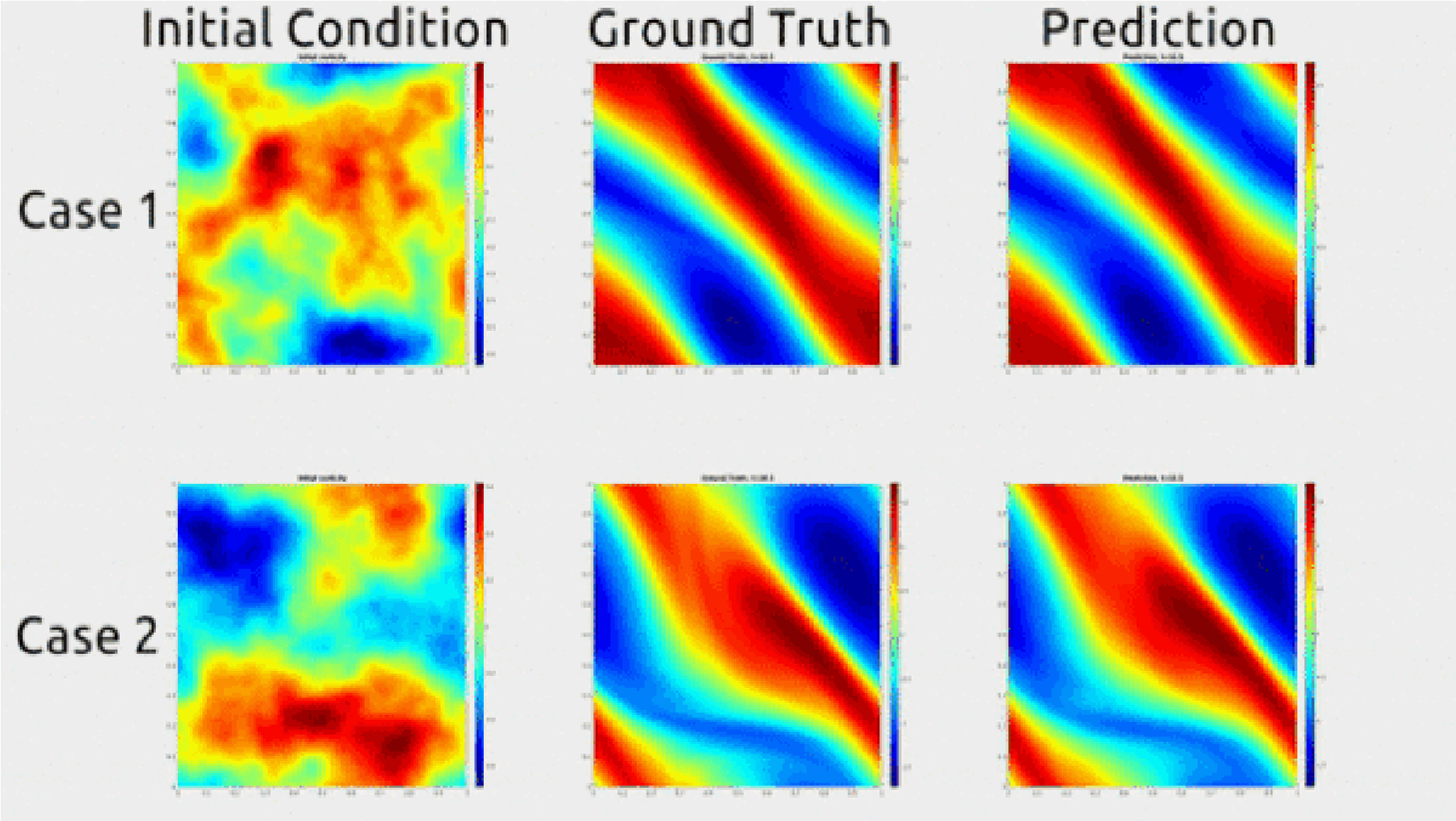
$$v_{k+1}(x)=\sigma(\int_\Omega K_\theta(x,x')v_k(x')\,dx'+Wv_k(x))$$

where:
- $K_\theta(x,x')$ is the learned **integral kernel** that encodes relationships between points x and x'.
- W is a local transformation (e.g., pointwise linear operator).
- σ is a nonlinear activation function.

This equation defines how function values interact across the entire domain, rather than just locally.

Step 3: Projection Back to Output Space
After several layers of kernel updates, the final output is obtained by applying a **projection operator** Q:
$u(x)=Q(v_k(x))$
This ensures that the output function remains in the correct solution space.

Neural Operators Examples



https://zongyi-li.github.io/neural-operator/

Feature	Green's Functions	DeepONet	Neural Operators
Problem Type	Linear differential equations	Data-driven function approximation	Learning operators for PDEs
Boundary Conditions	Required, explicitly handled	Implicitly learned from data	Can incorporate, but not always explicitly defined
Domain Shape	Typically simple (e.g., rectangular, spherical)	Can handle irregular domains if trained on them	Works on general domains, flexible input structures
Input Data	Differential equation and source term	Function values at discrete points	Function values, can handle grid-based or irregular data
Solution Representation	Integral over Green’s function	Neural network-based mapping	Maps functions to functions directly
Training Data Needed?	No, purely analytical	Yes, needs training data	Yes, needs training data for operator learning
Computational Cost	High for complex problems, needs explicit integral	Training is costly, but inference is fast	Training is costly, but efficient at inference
Adaptability to New Problems	Must derive new Green’s function for each problem	Can generalize to similar problems if trained properly	Highly generalizable to different PDEs and domains
Memory Usage	Large for storing Green’s functions in high dimensions	Moderate, depends on network size	More efficient for large-scale problems

Type of Error	Definition	Causes	How to Reduce
Approximation Error	Difference between best possible function in model class and the true function f∗f^*f∗	Limited model capacity, wrong architecture	Increase network size, change activation functions, improve architecture
Optimization Error	Difference between best possible function and what is learned during training	Poor optimizer, bad learning rate, local minima, bad initialization	Use Adam/SGD, adjust learning rate, batch norm, weight initialization
Generalization Error	Difference between training error and test error	Overfitting, small dataset, high variance	Regularization, more data, dropout, early stopping