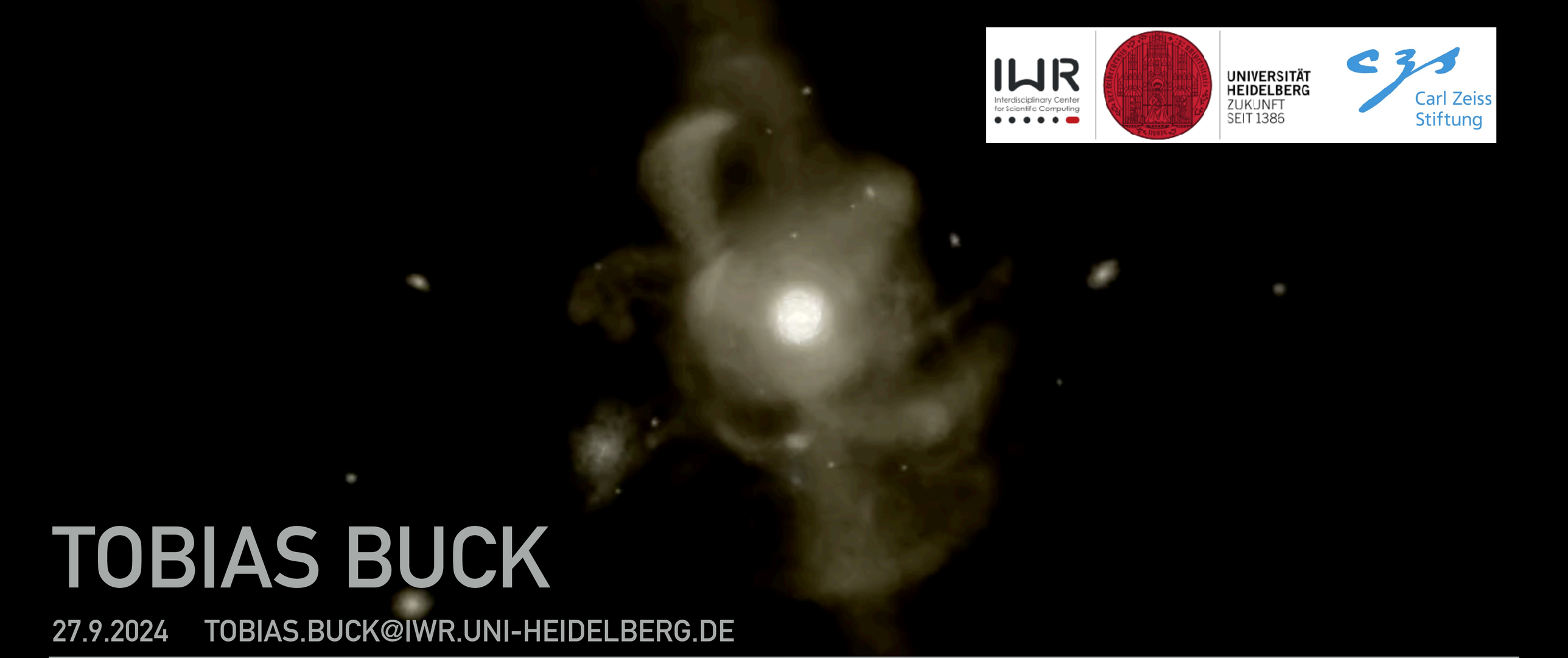


TOBIAS BUCK

27.9.2024 TOBIAS.BUCK@IWR.UNI-HEIDELBERG.DE

SCIENTIFIC ML

OUTLINE OF TODAY

1. Scientific ML
   A. Neural ODEs
   B. Symbolic Regression
   C. Operator Learning
   D. Physics Informed Neural Networks
   E. Hamiltonian Neural Networks

WHERE TO FIND THE LECTURE MATERIAL

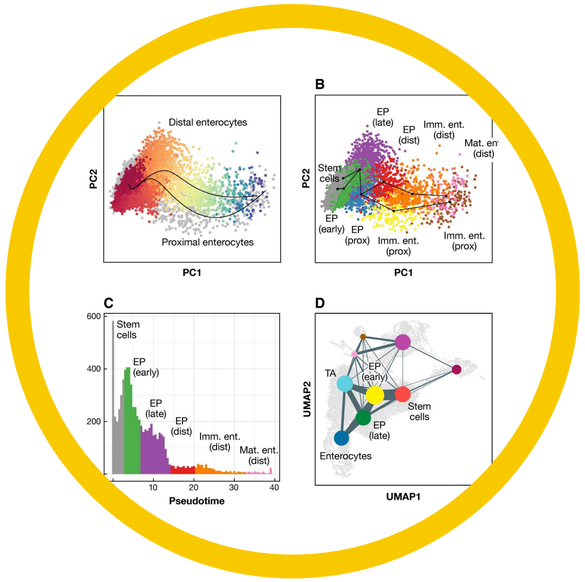
https://github.com/TobiBu/graddays

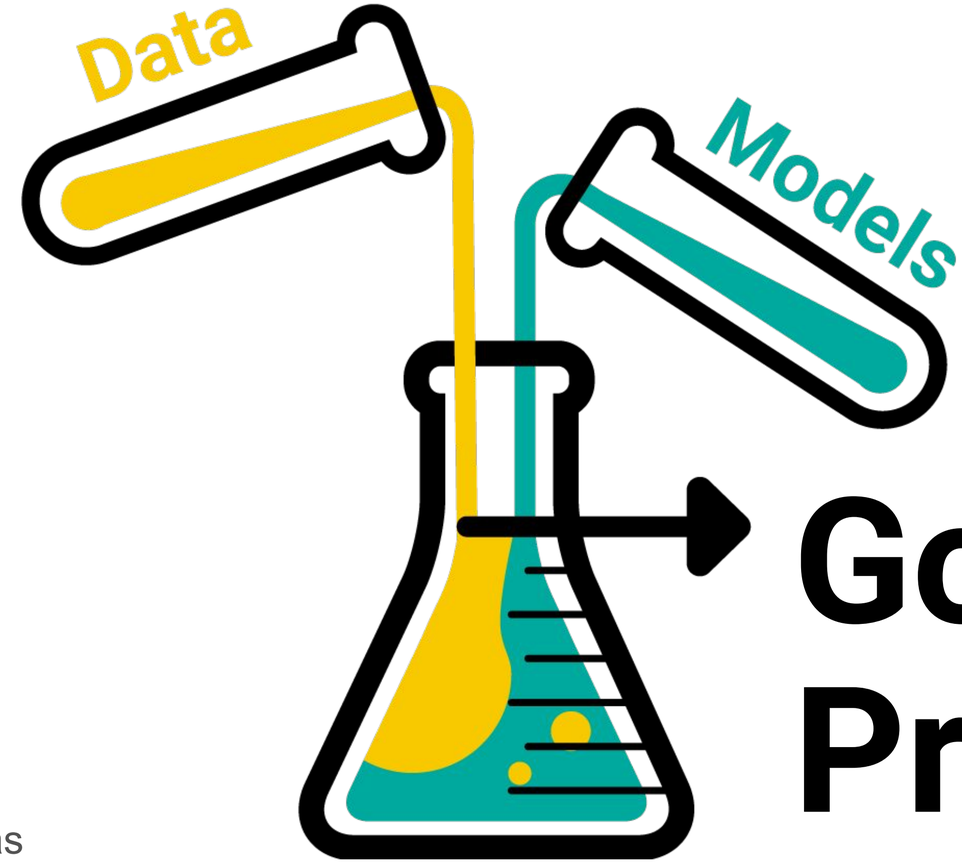
SCIENTIFIC MACHINE LEARNING

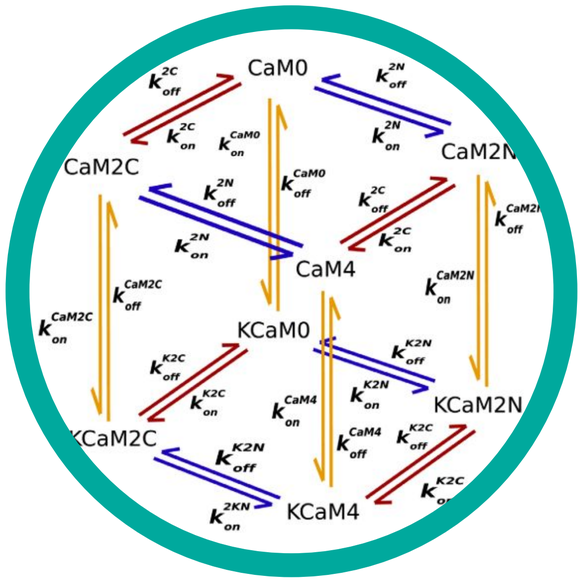
»Scientific machine learning is the burgeoning field combining techniques of machine learning into traditional scientific computing and mechanistic modeling.« [e. g. Zubov et al, 2021]

SCIENTIFIC MACHINE LEARNING IS MODEL-BASED DATA-EFFICIENT ML

How do we simultaneously use both sources of knowledge?

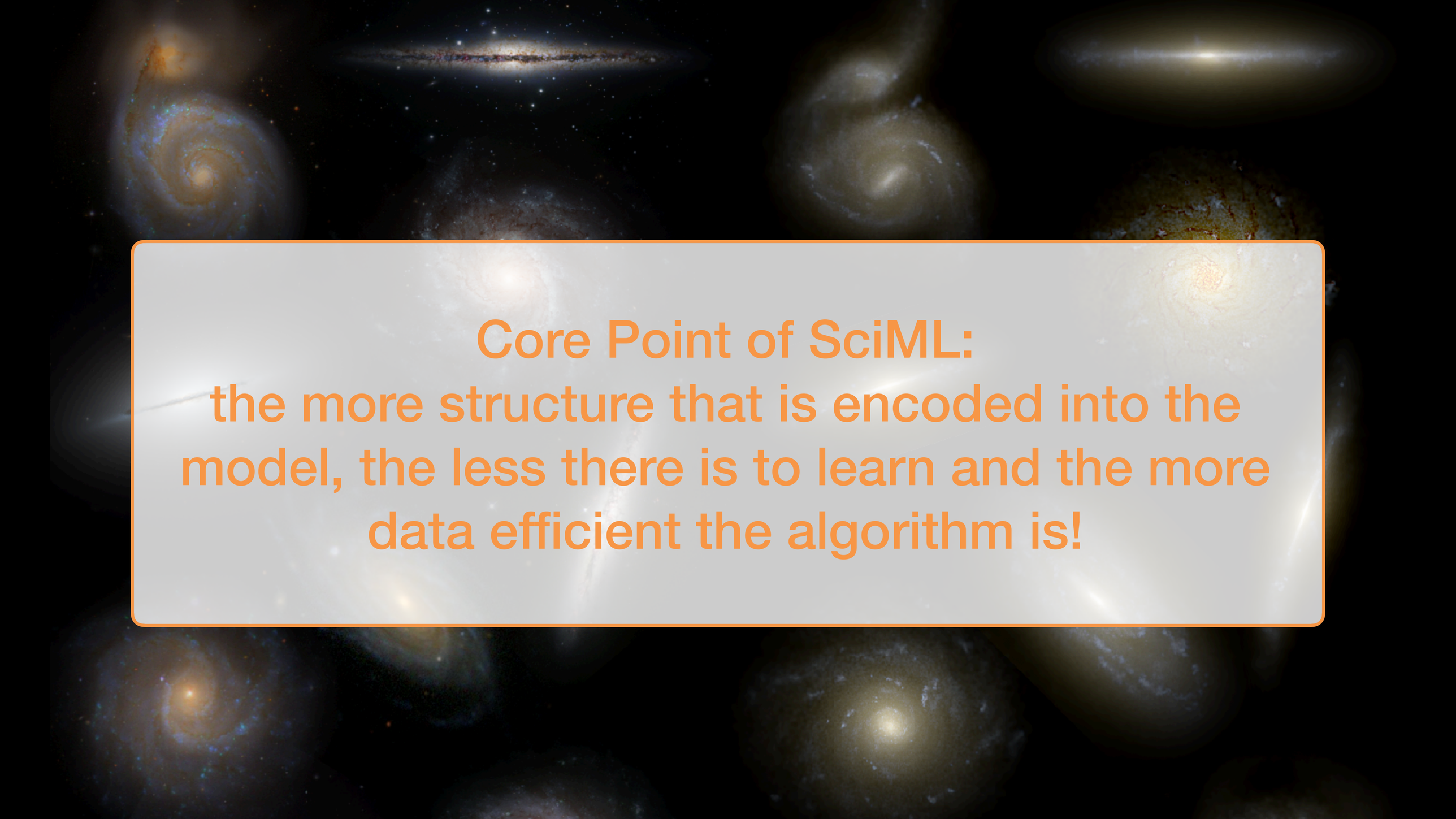






Good Predictions

slide inspired by Chris Rackauckas

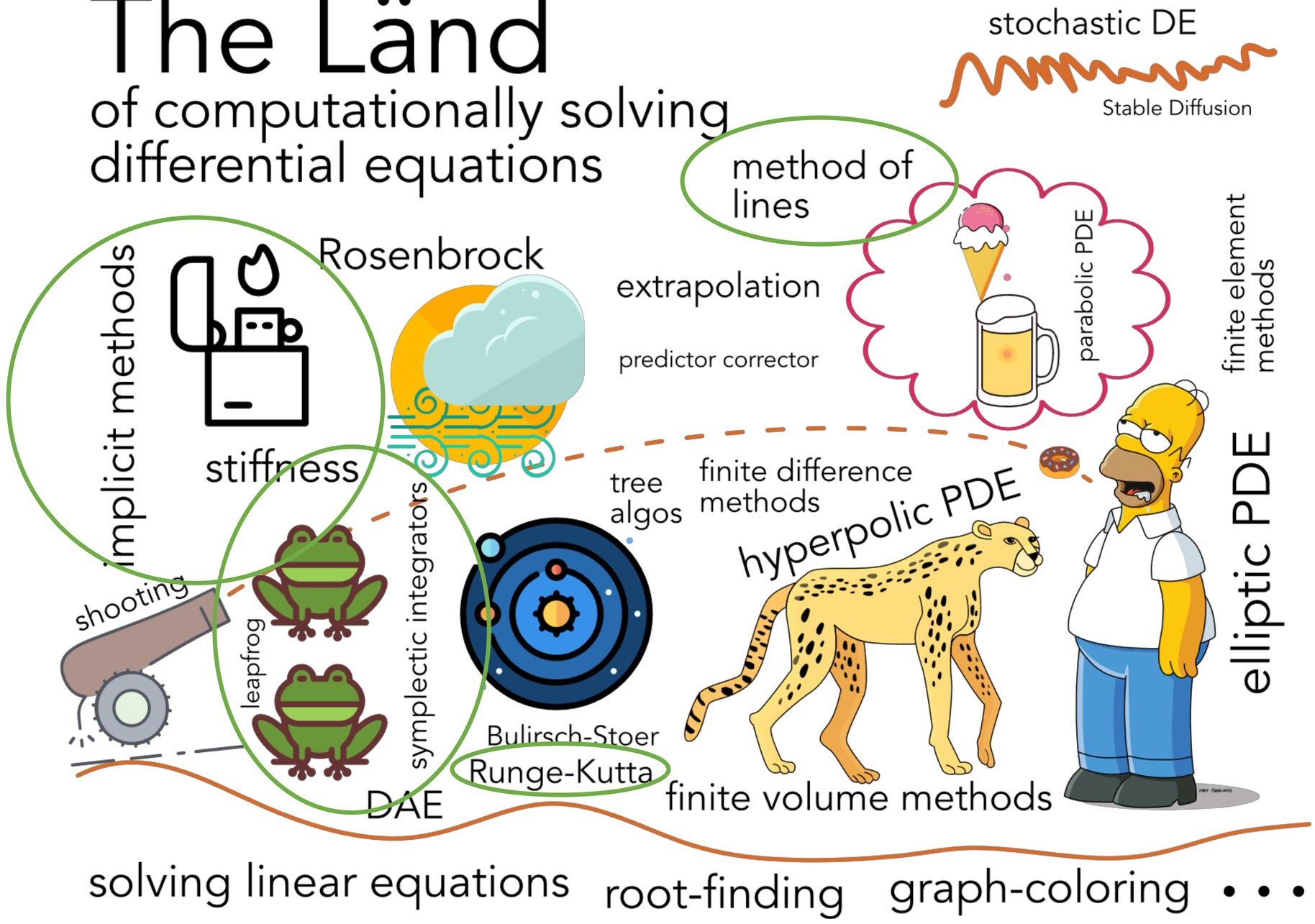

Core Point of SciML:
the more structure that is encoded into the model, the less there is to learn and the more data efficient the algorithm is!

Traditional methods for solving DEs

It's big.

The Länd

of computationally solving differential equations



Slide taken from Leonard Storcks

THE SOLUTION OF ODES

Explicit Euler Method:

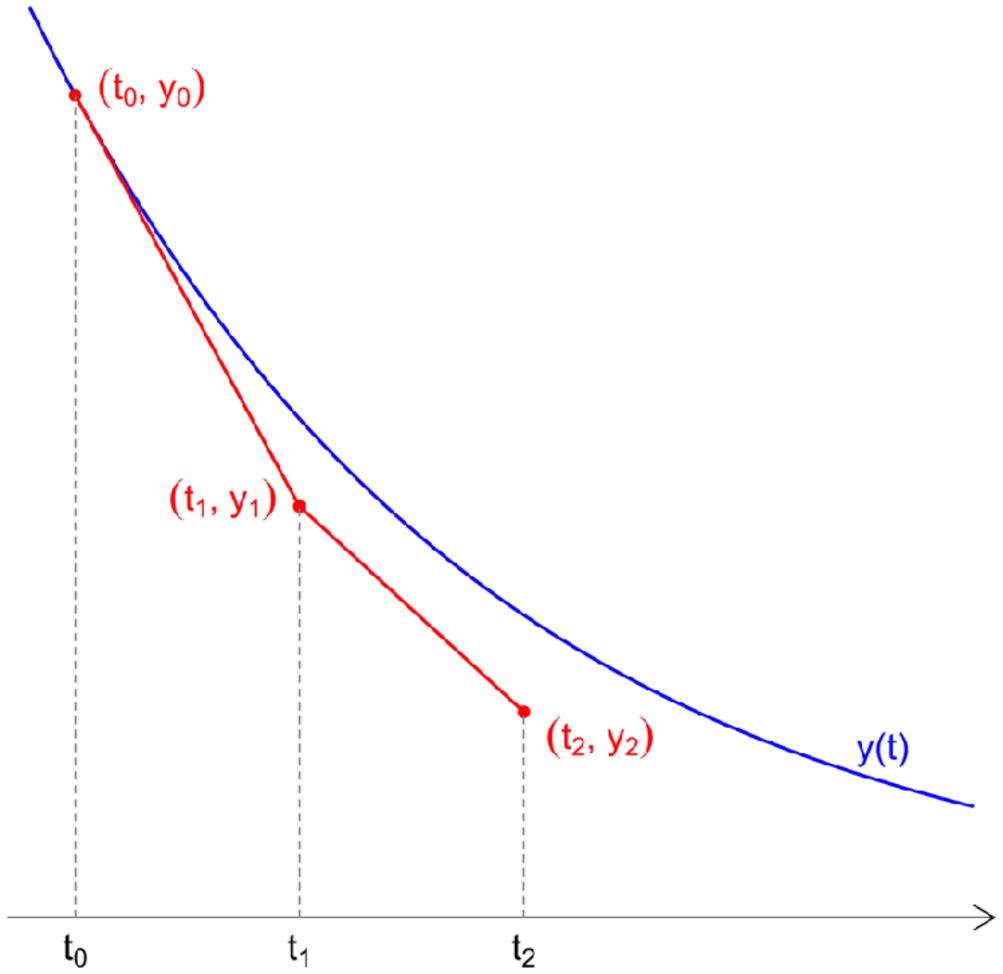
$$y^{n+1} = y^n + \frac{\partial y}{\partial t}\Delta t$$

Order of the solver:

Taylor expansion

$$y^{n+1} = y^n + \frac{\partial y^n}{\partial t}\Delta t + \mathcal{O}(\Delta t^2)$$

→ first order accurate

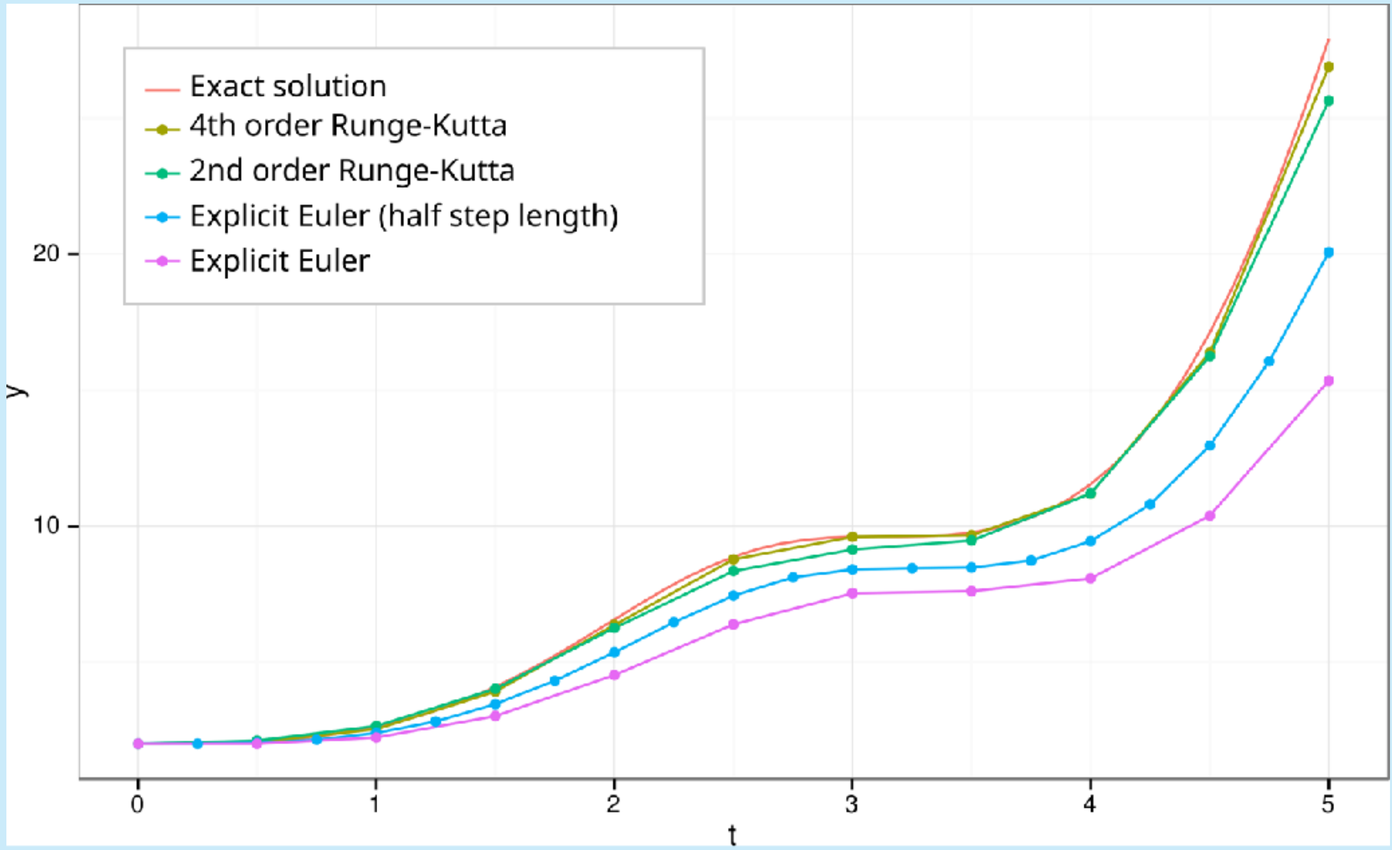


HIGHER ORDER SOLVERS

Runge-Kutta-Method:

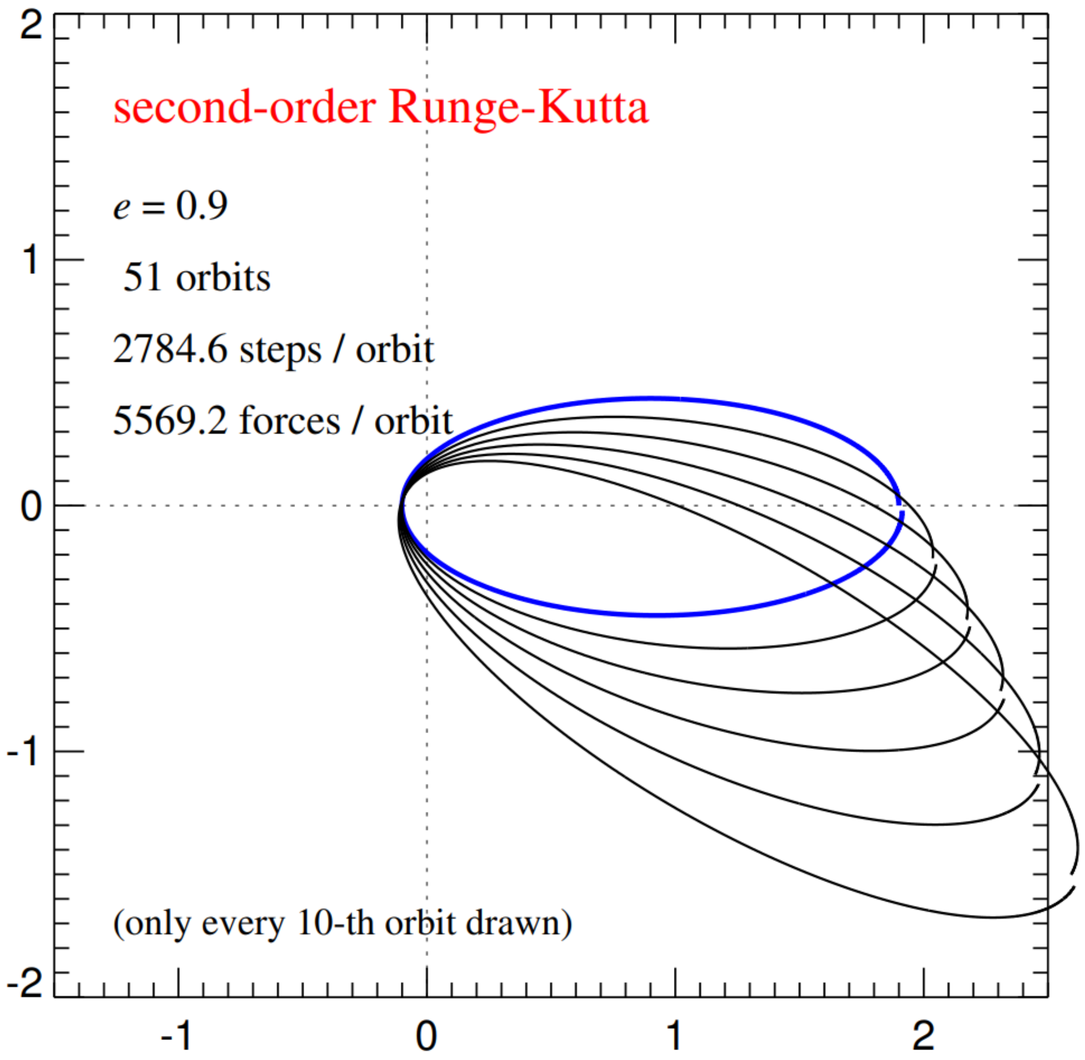
weighted combination of simple derivatives

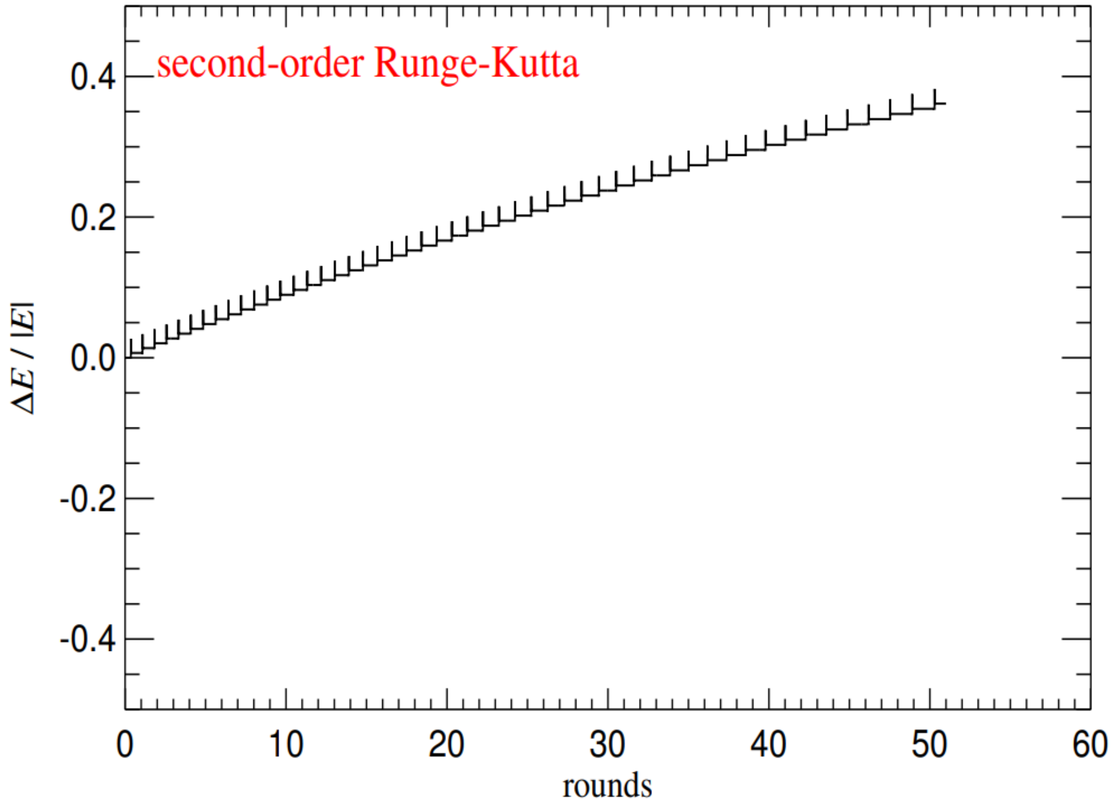
$$y^{n+1} = y^n + h \sum_{i=1}^{s} b_i k_i$$



$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n + c_2 h, y_n + (a_{21} k_1) h), \\
k_3 &= f(t_n + c_3 h, y_n + (a_{31} k_1 + a_{32} k_2) h), \\
&\vdots \\
k_s &= f(t_n + c_s h, y_n + (a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1}) h).
\end{aligned}$$

SOLUTION OF THE TWO-BODY PROBLEM



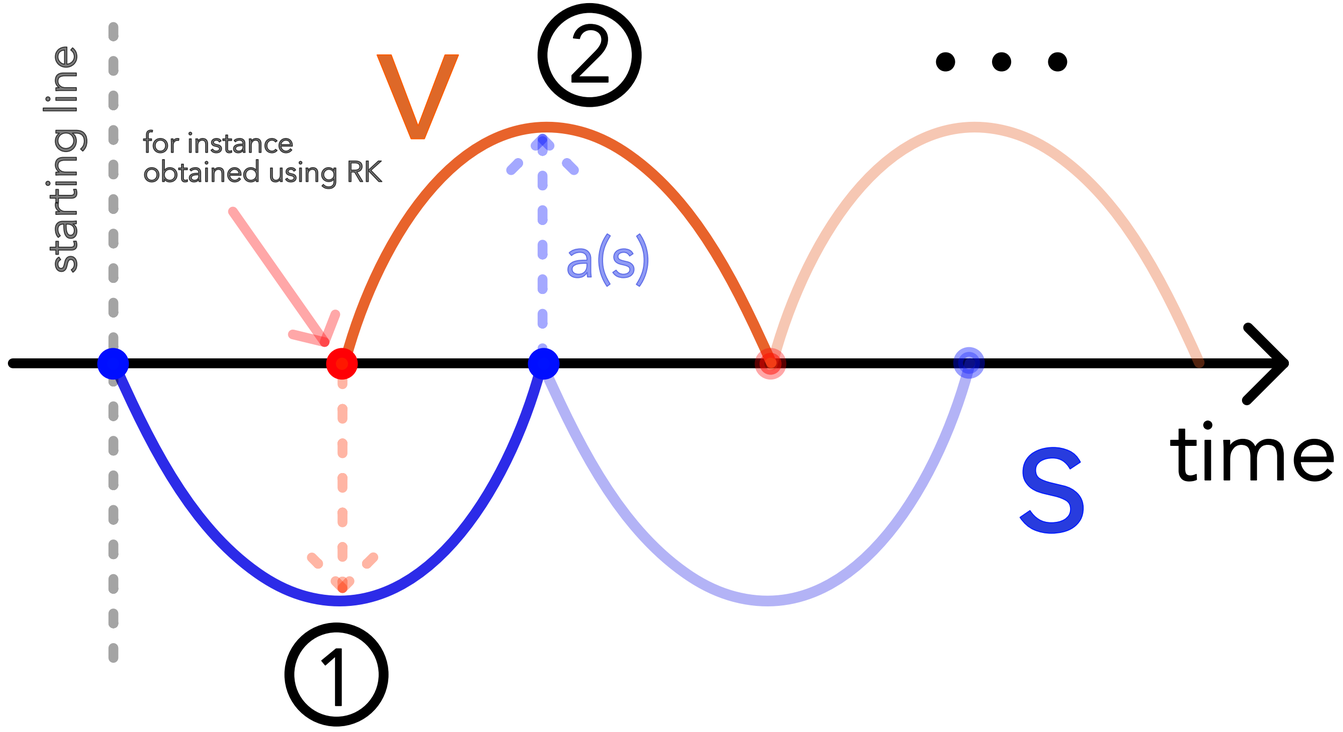


Energy error in each step!

[from the Fundamentals of Simulation Methods lecture notes by Springel et al., version 2022/2023]

SOLUTION OF THE TWO-BODY PROBLEM

Leapfrog



Symplectic integrator with good energy conservation properties; time reversibility; exact angular momentum conservation

1. update location:

$$\underline{s}\left(t+\frac{1}{2}\Delta t\right)=\underline{s}\left(t-\frac{1}{2}\Delta t\right)+\underline{v}(t)\Delta t+\mathcal{O}(\Delta t^3)$$

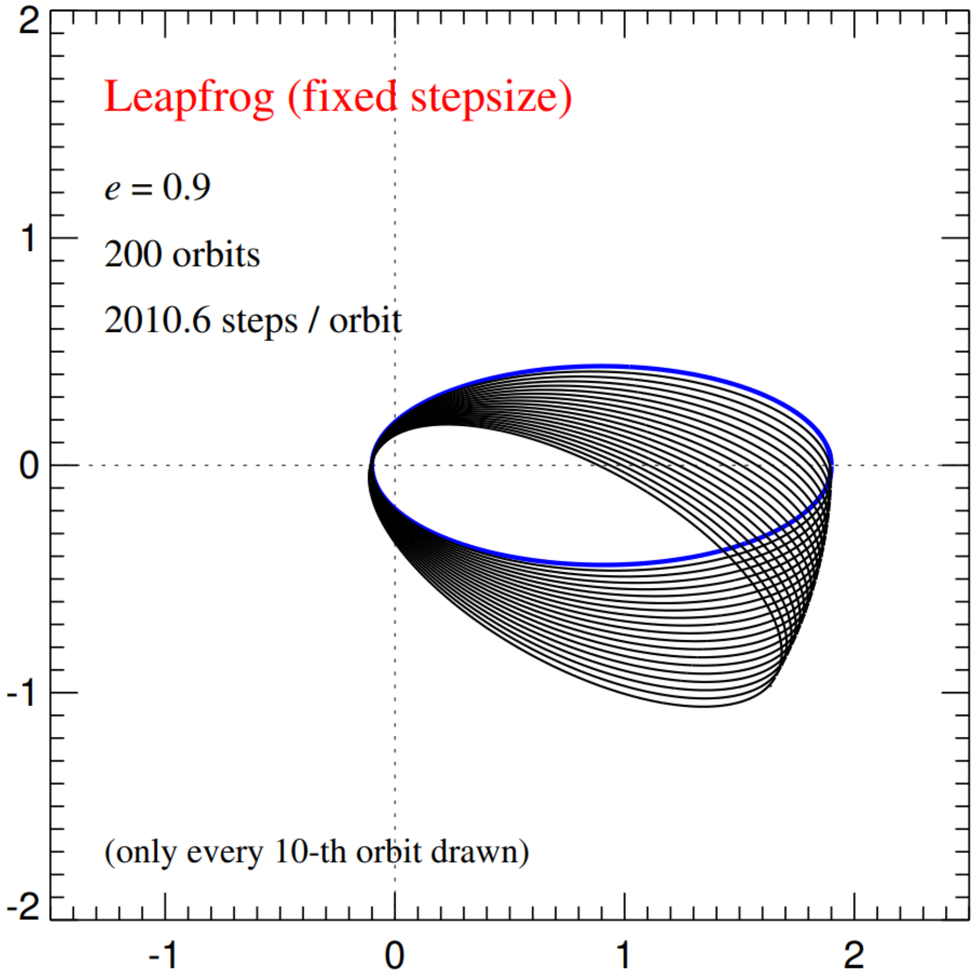
2. update velocity:

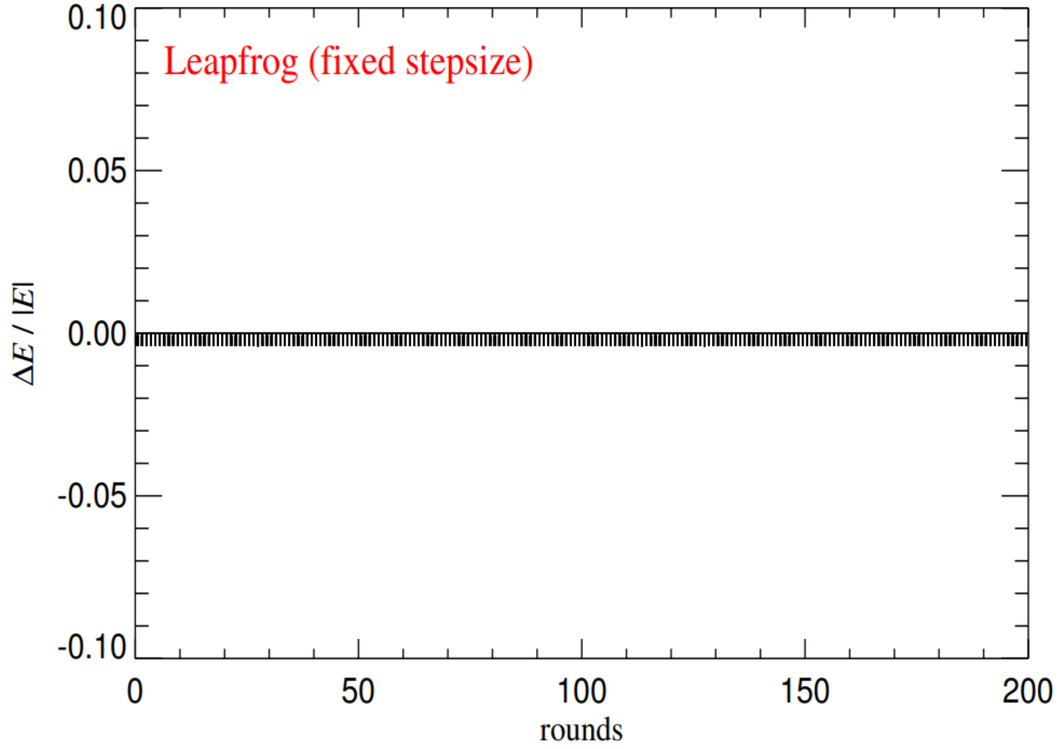
$$\underline{v}(t+\Delta t)=\underline{v}(t)+\underline{a}\left(t+\frac{1}{2}\Delta t\right)\Delta t+\mathcal{O}(\Delta t^3)$$

SOLUTION OF THE TWO-BODY PROBLEM

Exact solution to $H_{leap} = H + H_{err}$,

$$H_{err} \propto \frac{\Delta t^2}{12}\left\{\{H_{kin}, H_{pot}\}, H_{kin} + \frac{1}{2}H_{pot}\right\} + \mathcal{O}(\Delta t^3)$$





[from the "Fundamentals of Simulation Methods" lecture notes by Springel et al., version 2022/2023]

BREAK DOWN OF CLASSICAL METHODS

- Strong turbulence:
  - turbulent flow simulations are limited by Reynolds numbers. Unreasonable small grid required
  - →handle small scales by ML

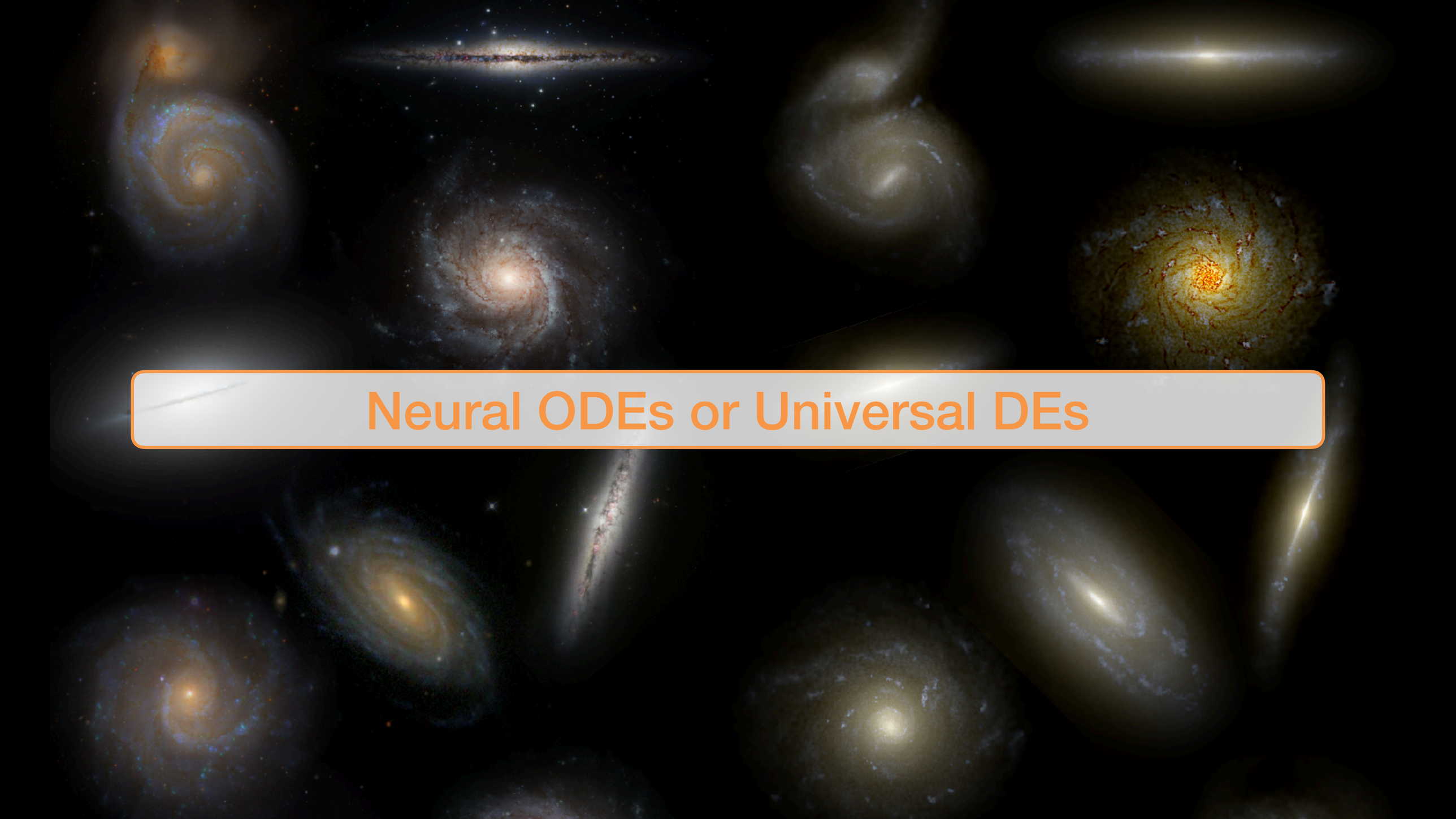
- Problem of different scales
  - Different scales can be very problematic (although there are adaptive methods, mesh free methods, tree algorithms etc.)

- curse of dimensionality for high-dimensional PDEs
  - (e. g. Schrödinger in Physics, Black-Scholes in finance): The computational cost for solving them goes up exponentially with the dimensionality. [Han et al, 2018]

- Need for inverse modelling

- how to blend DEs with vast data sets?

- …


Neural ODEs or Universal DEs

SCIENTIFIC ML — LEARNING THE SOLUTION OF ODES AND PDES

- XDEs are good for:
  - population models
  - motion of the planets
  - structural integrity of a bridge
  - fluid dynamics
  - …

ODEs are kind of easy — only derivatives with respect to one variable

PDEs are more complicated — derivatives with respect to many variables and differential equations are local while solutions exhibit non-local properties

- Traditional solution: discretisation (in time and space) and iterative solution

NEURAL ODES, OPERATOR LEARNING AND PINNS

Neural ODE: $\frac{df}{dt} = h_\theta(x_0, t, p)$

(neural net = right hand side of diff eq.

solution: integrate entire neural net.)

Neural Operator: $G_\theta : X \to Y \quad u \mapsto G_\theta(u)$ with X, Y function spaces (infinite dimensional)

(neural net approximates the operator
i.e. the map between function space)

PINN: $f(x,t) = h_\theta(x,t,p)$ with $\frac{df}{dt} = \frac{dh_\theta}{dt}, \quad \frac{df}{dx} = \frac{dh_\theta}{dx}$ need to fulfil the diff eq.

(solution is given by neural net,
autodiff and diff eq. are used in loss)

An introduction to neural ODEs in scientific machine learning

Patrick Kidger

Cradle.bio

Slides taken from Patrick Kidger

01

From traditional parameterised modelling to neural ODEs

Slides taken from Patrick Kidger

ODEs are really really (really) good models.

🐰(0) = 2 $\quad d🐰/_{dt} =$ 🐰 − 🐰🦊

🦊(0) = 1 $\quad d🦊/_{dt} =$ − 🦊 + 🐰🦊

Good for:

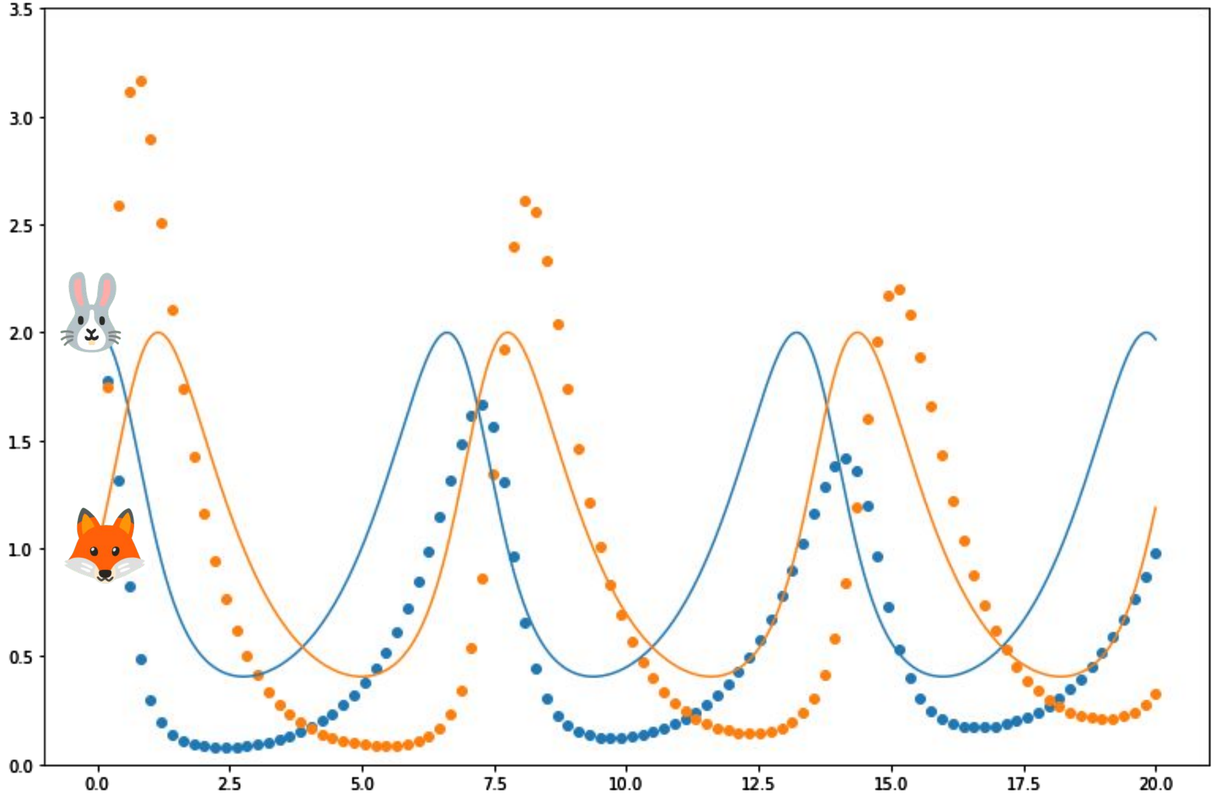
- Population modelling;
- Motion of the planets;
- Structural integrity of a bridge;
- Fluid dynamics;
- …

Simple idea: just write down how a value:

- Starts
- And how it changes over time.

Slides taken from Patrick Kidger

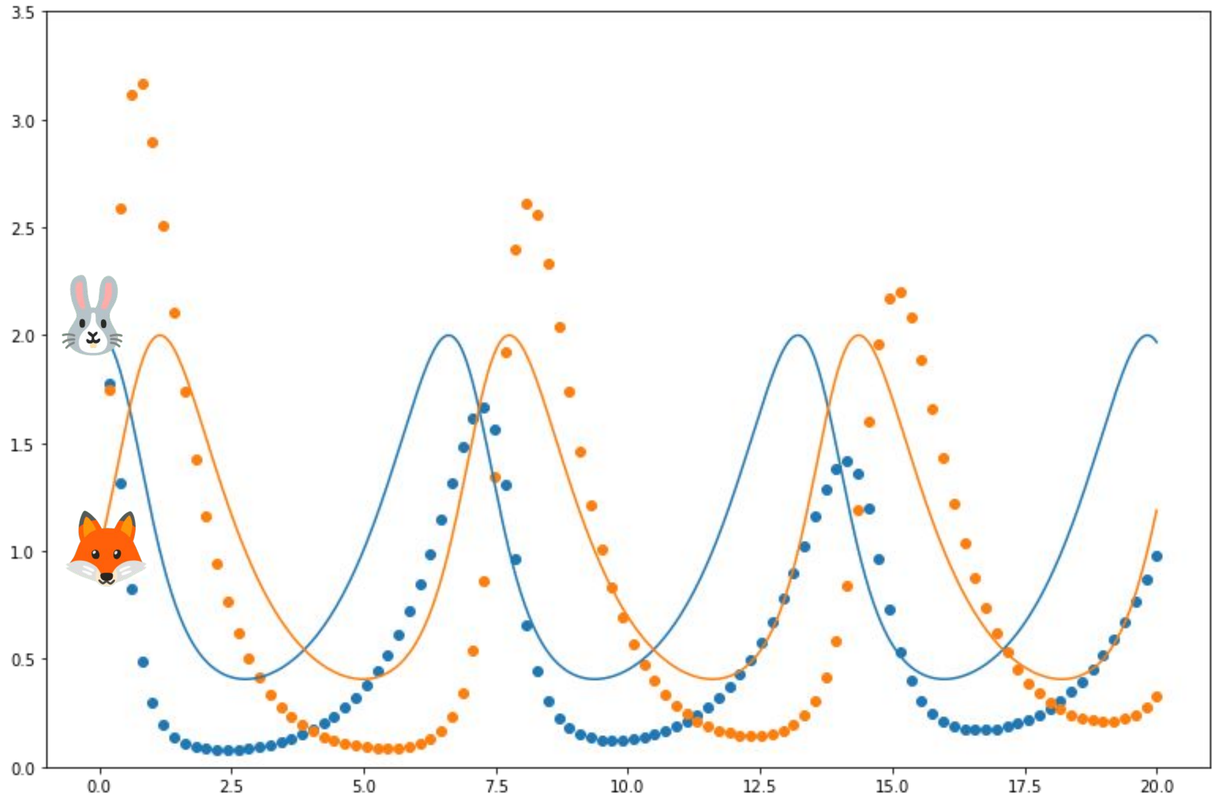
Lotka—Volterra (Predator–Prey)



Parameterised ODEs

🐰(0) = 2 $\quad \frac{d🐰}{dt} = 🐰 - 🐰🦊$

🦊(0) = 1 $\quad \frac{d🦊}{dt} = -🦊 + 🐰🦊$



Slides taken from Patrick Kidger

Parameterised ODEs

$🐰(0) = 2 \qquad d🐰/dt = \alpha🐰 - \beta🐰🦊$

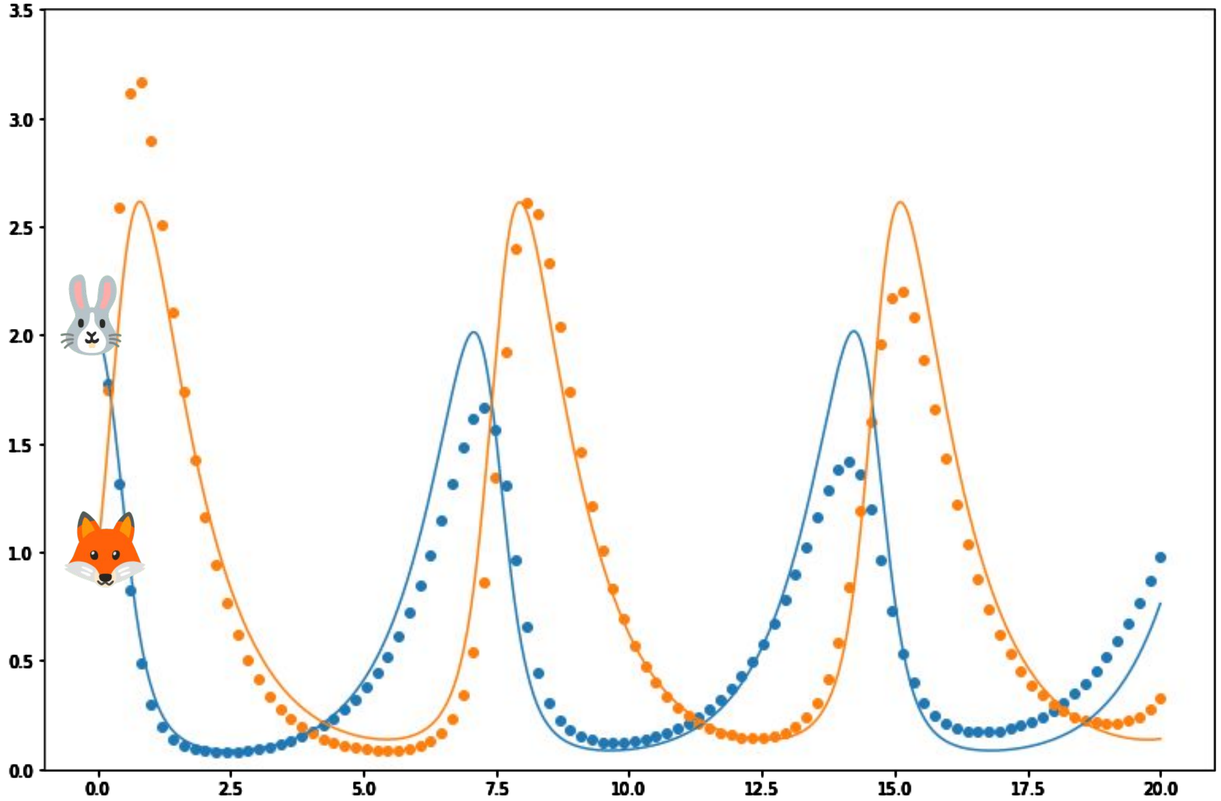
$🦊(0) = 1 \qquad d🦊/dt = -\gamma🦊 + \delta🐰🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs

$🐰(0) = 2 \qquad d🐰/dt = \alpha 🐰 - \beta 🐰 🦊 + NN_\theta(🐰, 🦊)$

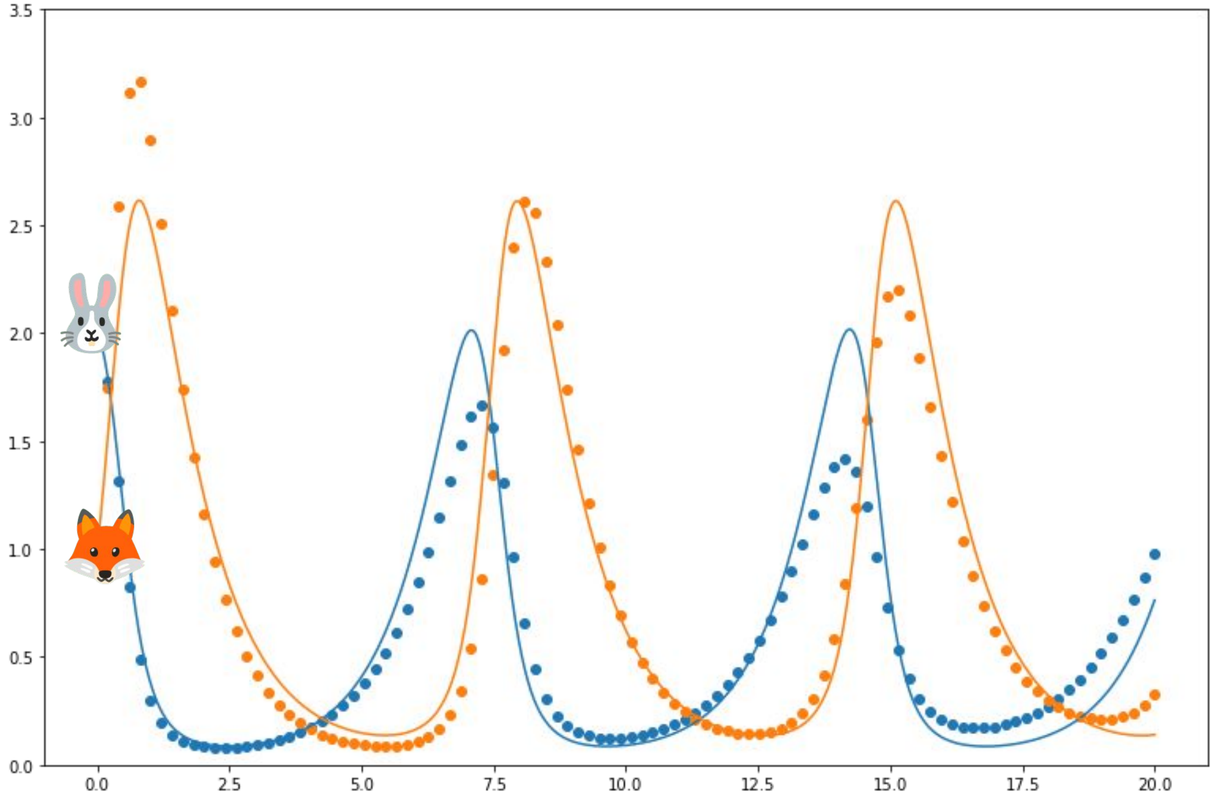
$🦊(0) = 1 \qquad d🦊/dt = -\gamma 🦊 + \delta 🐰 🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs

$🐰(0) = 2$ $\quad d🐰/_{dt} = \alpha 🐰 - \beta 🐰 🦊 + NN_\theta(🐰, 🦊)$

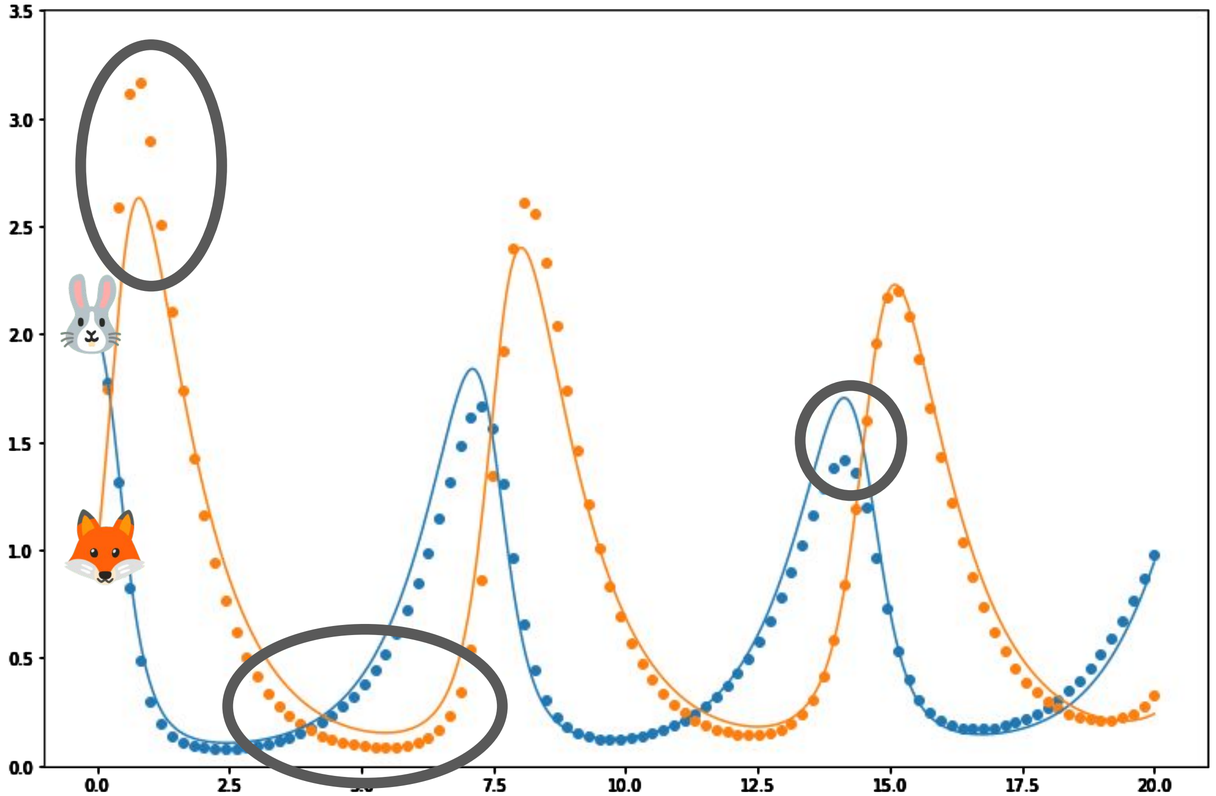
$🦊(0) = 1$ $\quad d🦊/_{dt} = -\gamma 🦊 + \delta 🐰 🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs: version 2

$🐰(0) = 2 \qquad d🐰/dt = \alpha 🐰 - \beta 🐰🦊$

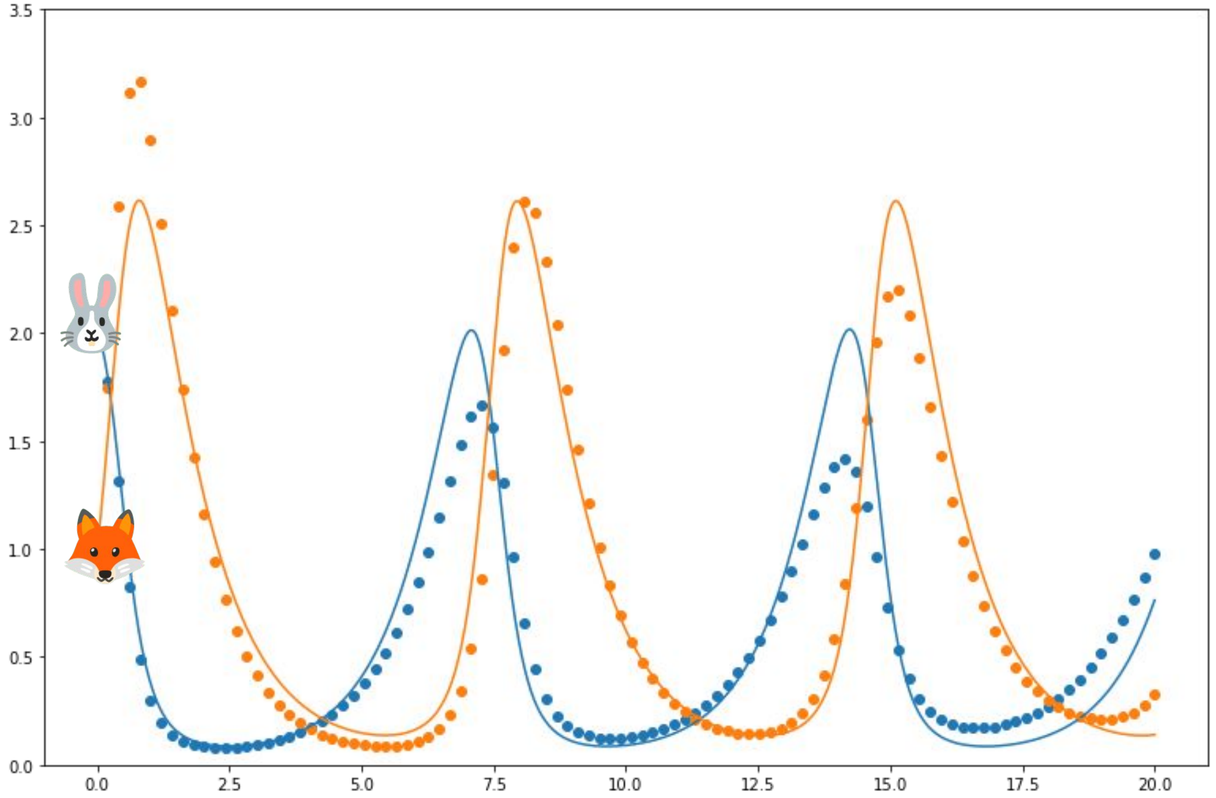
$🦊(0) = 1 \qquad d🦊/dt = -\gamma 🦊 + \delta 🐰🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs: version 2

$$🐰(0) = 2 \qquad d🐰/_{dt} = \alpha 🐰 - \beta 🐰 🦊$$

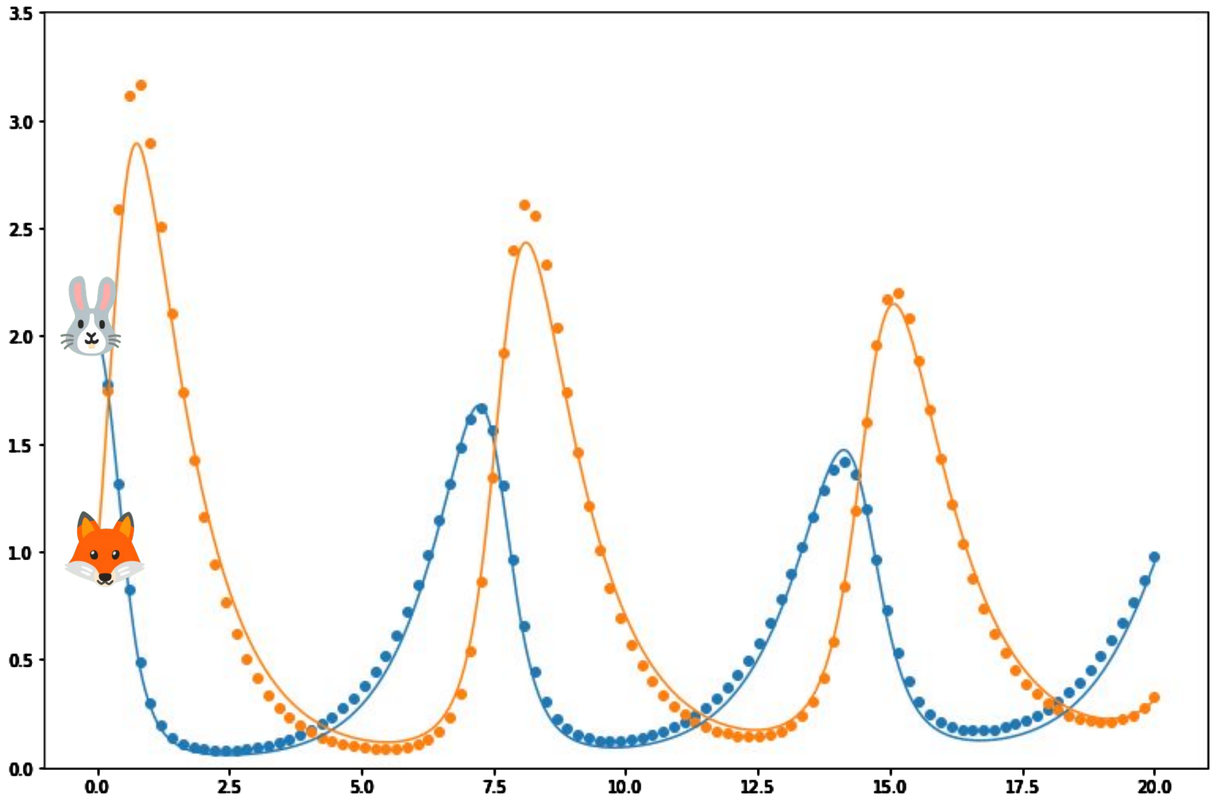
$$🦊(0) = 1 \qquad d🦊/_{dt} = -\gamma 🦊 + \delta 🐰 (🦊 + NN_\theta(🐰, 🦊))$$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Summary of part 1:

- Neural ODEs are a great model for unknown physics / unknown biology /…
    - When you don’t fully understand the physics.
    - When the physics is too expensive to fully simulate. (E.g. in weather.)
    - When a traditional parameterised model is difficult to optimise. (E.g. in biology.)
    - *Differentiable simulation, surrogate models*.

- Keep the physical model if you think it’s good.
    - Put the NN on the bit you think isn’t well-modelled.
    - Here the NN is tiny: 4 neurons wide, 1 layer deep.
    - Physical parameters trained first; them kept fixed whilst training NN.

- Neural ODE trained in the same way as the mechanistic ODE.
    - In both cases: a parameterised vector field trained with backprop+SGD.
    - Easy to implement: software for NN, ODEs, SGD, …

- “Neural ODE” ≈ “hybrid ODE” ≈ “universal ODE” ≈ “parameterised ODE”

Slides taken from Patrick Kidger

Symbolic regression

Let's try to interpret NN_θ.

$$\mathrm{d}🐰/\mathrm{dt} = 1.1🐰 - 1.3🐰🦊$$
$$\mathrm{d}🦊/\mathrm{dt} = -0.98🦊 + 1.6🐰(🦊 + \mathrm{NN}_\theta(🐰, 🦊))$$

Generate a dataset of ~10k samples:

$$\{(🐰_1, 🦊_1, y_1), \ldots, (🐰_n, 🦊_n, y_n)\} \text{ with } y_i = \mathrm{NN}_\theta(🐰_i, 🦊_i).$$

(Sample $🐰_i$ and $🦊_i$ in your favourite way, e.g. points from typical ODE trajectories.)

Symbolically regress y_i against $(🐰_i, 🦊_i)$:

$$\{\text{PySR, Lasso, ...}\} \Rightarrow y = 🦊^{0.9} - 🦊 + 0.5🦊^2$$

Retrain all parameters differentiably:

$$\mathrm{d}🐰/\mathrm{dt} = 0.9🐰 - 1.1🐰🦊$$
$$\mathrm{d}🦊/\mathrm{dt} = -1.2🦊 + 2.1🐰🦊^{0.95}$$

vs SINDy?

SINDy performs Lasso of d(🐰,🦊)/dt against (🐰, 🦊). Thus it requires an estimate of the derivative. That's not needed here!

Also, we're not constrained to just Lasso: other symbolic regression (e.g. PySR i.e. regularised evolution) can be used.

Slides taken from Patrick Kidger

Conditions + Learnt Initialisations

Recall our general ODE formulation.

$$(🐰, 🦊)(0) = g \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊)$$

Both f and g may depend on additional data, e.g., the latitude L that the 🐰 and 🦊 live at.

$$(🐰, 🦊)(0) = g(L) \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊, L)$$

Recall that an ODE is determined by:

- An initial condition;
- A vector field.

So far we've tried learning the vector field.

$$(🐰, 🦊)(0) = g_\theta(L) \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊, L)$$

We can also learn the initial condition.

Slides taken from Patrick Kidger

Generalised observables

We might observe things that aren't part of the evolving state of our ODE, but which still somehow give us information.

For example, we might observe the amount of rabbit... output 💩.

Now fit this to the data as well.

Even if we don't care about 💩, this can help us fit the 🐰, 🦊 parts better.

Important extreme case: "hidden state".

Slides taken from Patrick Kidger

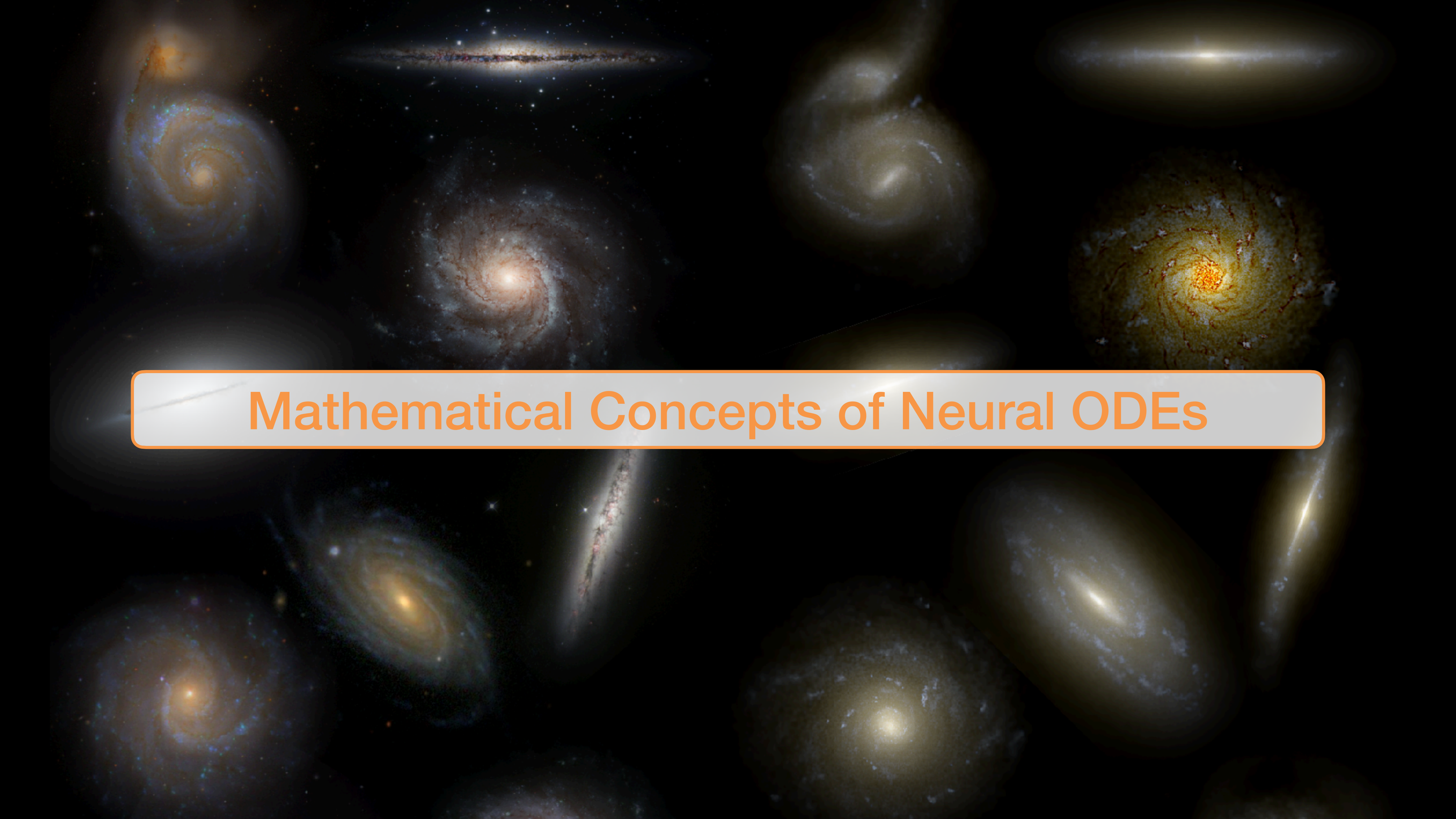
$$(🐰, 🦊)(0) = g_\theta(L) \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊, L)$$

$$💩 = h_\theta(🐰, L)$$

$$\text{loss} = \|🐰 - 🐰_{\text{data}}\|^2 + \|🦊 - 🦊_{\text{data}}\|^2 + \|💩 - 💩_{\text{data}}\|^2$$

$$y(0) = g_\theta(c) \qquad dy/dt = f_\theta(y, c) \qquad (🐰, 🦊, 💩) = h_\theta(y, c)$$

This is actually the the most typical formulation of a neural ODE!


Mathematical Concepts of Neural ODEs

NEURAL ODES

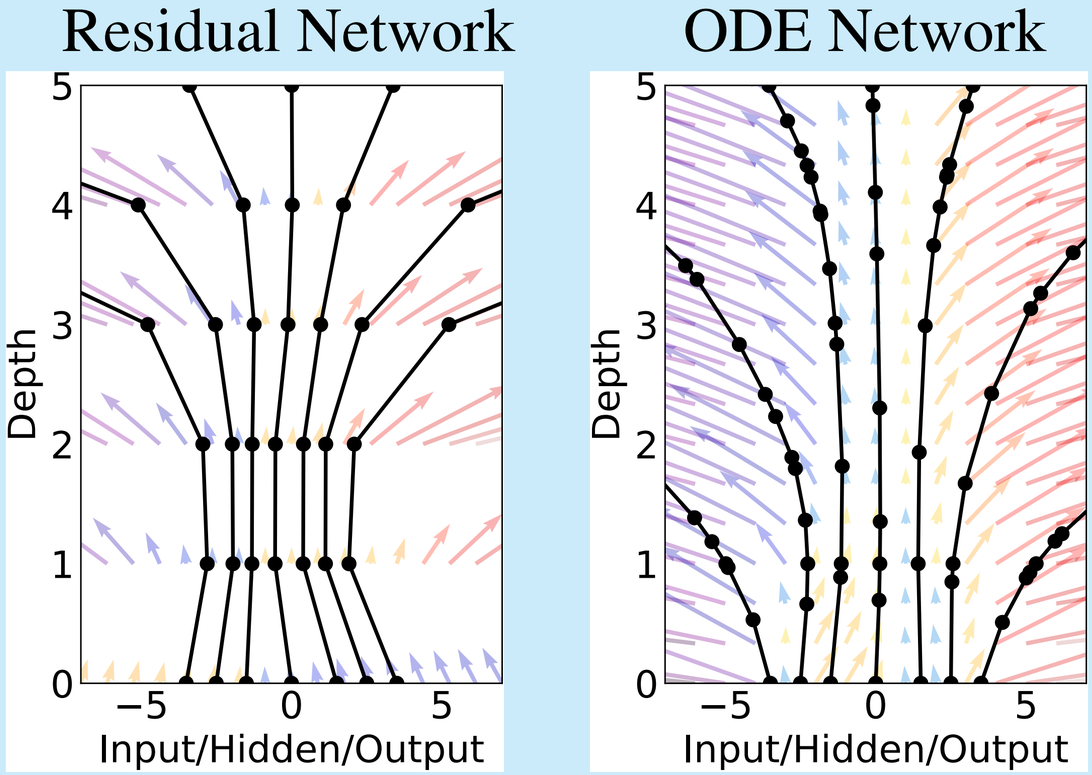
- Traditional solution: discretisation (in time and space) and iterative solution

Euler discretization

$$h_{t+1} = h_t + f_t(h_t, \theta_t)$$

$$h(t + \Delta t) = h(t) + \Delta t \cdot f(t, h(t), \theta)$$

$$\frac{h(t + \Delta t) - h(t)}{\Delta t} = f(t, h(t), \theta)$$



Chen+2019

NEURAL ODES

- Traditional solution: discretisation (in time and space) and iterative solution

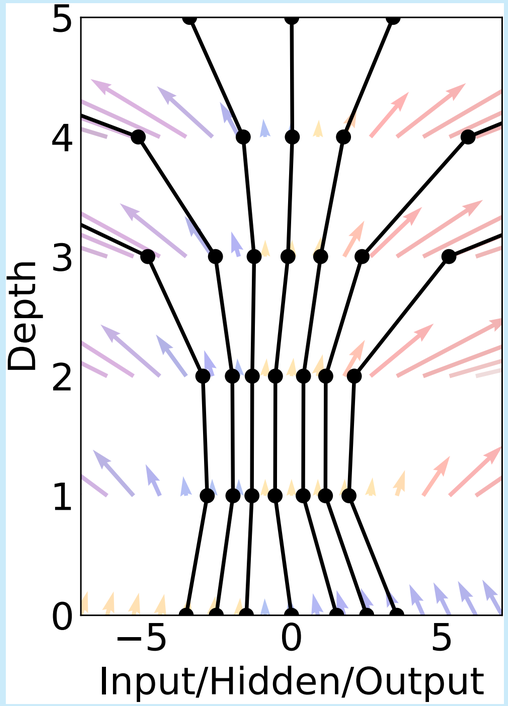
Euler discretization

$$h_{t+1} = h_t + f_t(h_t, \theta_t)$$

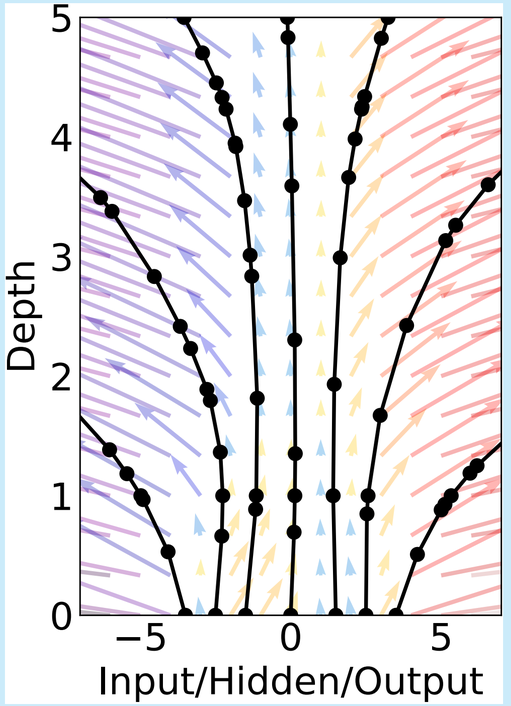
$$h(t + \Delta t) = h(t) + \Delta t \cdot f(t, h(t), \theta)$$

$$\frac{h(t + \Delta t) - h(t)}{\Delta t} = f(t, h(t), \theta)$$

Residual Network



ODE Network



Chen+2019

NEURAL ODES

What do we need to train a NeuralODE? → Reverse-mode automatic differentiation of ODE solutions

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right)$$

Problem is to find $\frac{\mathrm{dL}(\mathrm{z}(\mathrm{t}_1))}{\mathrm{d}\theta}$ efficiently!

Don’t worry about the details… diffrax (Patrick Kidger), torchdiffeq (Ricky Chen), DiffEqFlux.jl got you convert!

NEURAL ODES TRAINING

In general two options for NODE training:

1. directly use autodiff for $\frac{dL(z(t_1))}{d\theta}$ and back propagate through the solver.
aka „discretize then optimise“

advantage: faster, more accurate gradient; disadvantage: more memory consumption

2. solve $\frac{dL(z(t_1))}{d\theta}$ analytically.

Doing so will result in a backwards-in-time ODE (adjoint equation) that must be solved numerically
aka „optimise then discretise“

advantage: less memory consumption; disadvantage: more computationally expensive, only approximate gradients

ADJOINT EQUATION: PROBLEM STATEMENT

$$\text{Find} \underset{\theta}{\operatorname{argmin}} \; L(z(t_1)) \tag{PM}$$

$$\text{subject to}$$

$$F(\dot{z}(t), z(t), \theta, t) = \dot{z}(t) - f(z(t), \theta, t) = 0 \tag{1}$$

$$z(t_0) = z_{t_0} \tag{2}$$

$$t_0 < t_1$$

- f is our neural network with parameters θ
- $z(t_0)$ is our input, $z(t_1)$ is the output, $z(t)$ is the state reached from $z(t_0)$ at time $t \in [t_0, t_1]$
- $\dot{z}(t) = \frac{\mathrm{d}z(t)}{\mathrm{d}t}$ is the time derivative of our state $z(t)$.
- L is our loss and its a function of the output $z(t_1)$.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION: PROBLEM STATEMENT

$$\text{Find} \underset{\theta}{\operatorname{argmin}} \; L(z(t_1)) \tag{PM}$$

$$\text{subject to}$$

$$F(\dot{z}(t), z(t), \theta, t) = \dot{z}(t) - f(z(t), \theta, t) = 0 \tag{1}$$

$$z(t_0) = z_{t_0} \tag{2}$$

$$t_0 < t_1$$

- f is our neural network with parameters θ
- $z(t_0)$ is our input, $z(t_1)$ is the output, $z(t)$ is the state reached from $z(t_0)$ at time $t \in [t_0, t_1]$
- $\dot{z}(t) = \frac{\mathrm{d}z(t)}{\mathrm{d}t}$ is the time derivative of our state $z(t)$.
- L is our loss and its a function of the output $z(t_1)$.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let’s rewrite our initial objective with a Lagrange multiplier $\lambda(t)$

$$\psi = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t) F(\dot{z(t)}, z(t), \theta, t) dt$$

since

$$F(\dot{z(t)}, z(t), \theta, t) = \dot{z(t)} - f(z(t), \theta, t) = 0$$

the integral vanishes and

$$\frac{\mathrm{d}\psi}{\mathrm{d}\theta} = \frac{\mathrm{d}L(z(t_1))}{\mathrm{d}\theta}$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let's rewrite our initial objective with a Lagrange multiplier $\lambda(t)$

$$\psi = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t) F(\dot{z(t)}, z(t), \theta, t) dt$$

since

$$F(\dot{z(t)}, z(t), \theta, t) = \dot{z(t)} - f(z(t), \theta, t) = 0$$

the integral vanishes and

$$\frac{\mathrm{d}\psi}{\mathrm{d}\theta} = \frac{\mathrm{d}L(z(t_1))}{\mathrm{d}\theta}$$

Let's choose $\lambda(t)$ to eliminate hard to compute derivatives, such as the Jacobian $\frac{\mathrm{dz}(t_1)}{\mathrm{d}\theta}$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let the math magic happen…
(using integration by parts, chain rule and some recursive plugging in of equations…)
if you want the details, see the blog post.

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1}\left(\dot{\lambda(t)} + \lambda(t)\frac{\partial f}{\partial z}\right)\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}dt + \int_{t_0}^{t_1}\lambda(t)\frac{\partial f}{\partial \theta}dt$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Derivative	Description	Ease
$\frac{\partial L}{\partial z(t_1)}$	Gradient of loss with respect to output.	Easy
$\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta}$	Jacobian of output with respect to params.	Not easy
$\dot{\lambda(t)}$	Derivative of lambda, a vector, with respect to time.	Easy
$\lambda(t)\frac{\partial f}{\partial z}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial z}$.	Easy
$\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}$	Jacobian of arbitrary layer with respect to params.	Not easy
$\lambda(t)\frac{\partial f}{\partial \theta}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial \theta}$.	Easy

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Derivative	Description	Ease
$\frac{\partial L}{\partial z(t_1)}$	Gradient of loss with respect to output.	Easy
$\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta}$	Jacobian of output with respect to params.	Not easy
$\dot{\lambda(t)}$	Derivative of lambda, a vector, with respect to time.	Easy
$\lambda(t)\frac{\partial f}{\partial z}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial z}$.	Easy
$\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}$	Jacobian of arbitrary layer with respect to params.	Not easy
$\lambda(t)\frac{\partial f}{\partial \theta}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial \theta}$.	Easy

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let the math magic happen…
(using integration by parts, chain rule and some recursive plugging in of equations…)
if you want the details, see the blog post.

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1}\left(\dot{\lambda(t)} + \lambda(t)\frac{\partial f}{\partial z}\right)\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}dt + \int_{t_0}^{t_1}\lambda(t)\frac{\partial f}{\partial \theta}dt$$

Can we get rid of $\frac{\mathrm{dz(t)}}{\mathrm{d}\theta}$ by a smart choice of $\lambda(t)$?

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let the math magic happen…
(using integration by parts, chain rule and some recursive plugging in of equations…)
if you want the details, see the blog post.

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

Let's define the backward ODE as:

$$\dot{\lambda}(t) = -\lambda(t)\frac{\partial f}{\partial z} \quad \text{s.t.} \quad \lambda(t_1) = \frac{\partial L}{\partial z(t_1)}$$

giving

$$\lambda(t_0) = \lambda(t_1) - \int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial z} dt$$

$\lambda(t)$ is called the adjoint.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

Which simplifies our objective a lot!

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

Which simplifies our objective a lot!

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

(note, we switch the integration limits)

With this equation we trade memory for computation! Memory consumption is O(1) with respect to the number of layers.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

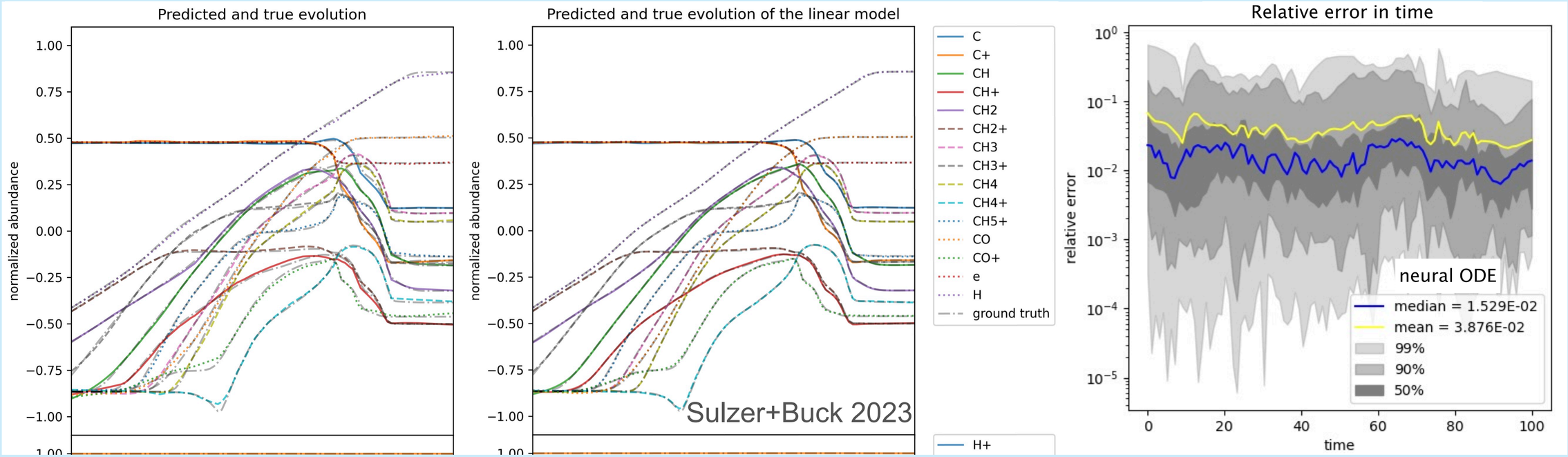
TRAINING NODE WITH ADJOINT METHOD STEP BY STEP RECIPE

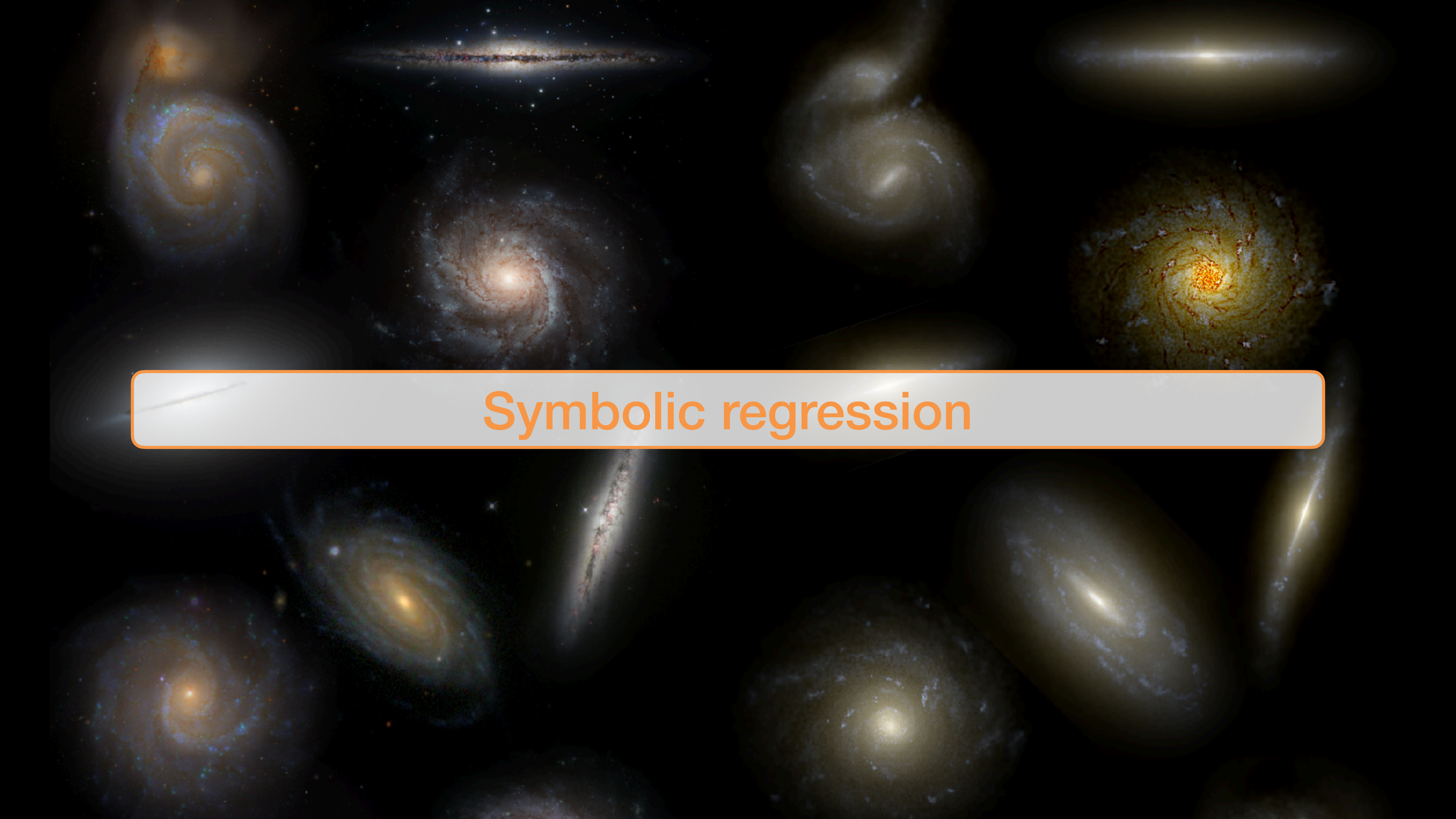
1. Forward pass: Solve the ODE (1)-(2) from time t_0 to t_1 and get the output $z(t_1)$.
2. Loss calculation: Calculate $L(z(t_1))$.
3. Backward pass: Solve ODEs (11) and (12) from reverse time t_1 to t_0 to get the gradient of the loss $\frac{\mathrm{d}L(z(t_1))}{\mathrm{d}\theta}$.
4. Use the gradient to update the network parameters θ.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

NEURAL ODES IN ASTROPHYSICS

- Neural Astrophysical Wind Models (Nguyen 2023)
- Neural ODEs as a discovery tool to characterize the structure of the hot galactic wind of M82 (Nguyen+2023)
- Speeding up astrochemical reaction networks with autoencoders and neural ODEs (Sulzer+Buck 2023)



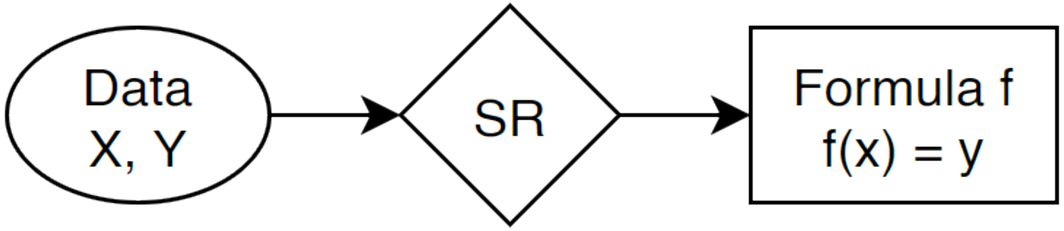

Symbolic regression

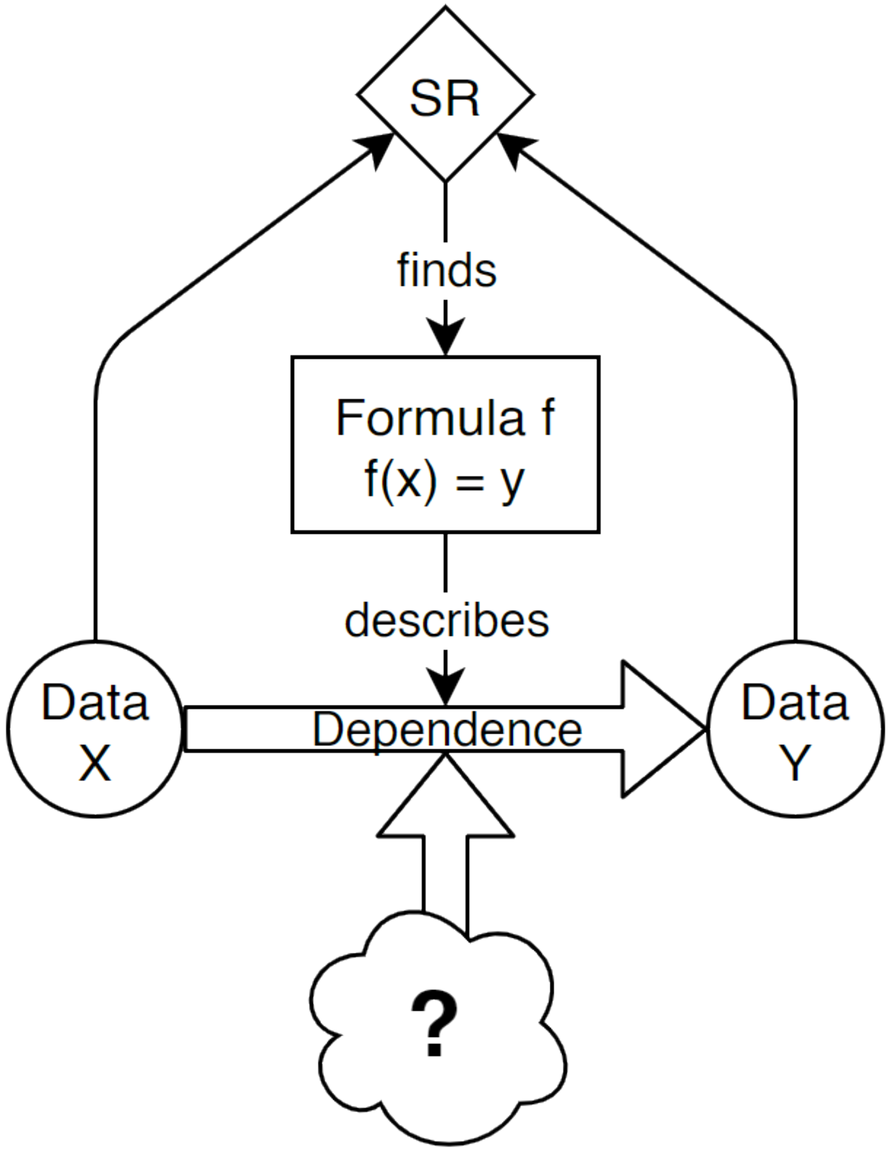
MOTIVATION

- Given: Data $\{(\vec{x}_i, y_i) | i \in \{1, ..., N\}\}$
  $\overbrace{x_{i,1}, ..., x_{i,d}}$
- Wanted: Formula f: $f(x) = y$

MOTIVATION

- Given: Data $\{(\vec{x}_i, y_i) | i \in \{1, ..., N\}\}$ $\overbrace{x_{i,1}, ..., x_{i,d}}$
- Wanted: Formula f: $f(x) = y$





MOTIVATION

How does y depend on x?

x	y
0	0
1	1
2	4
3	9
4	16
5	25

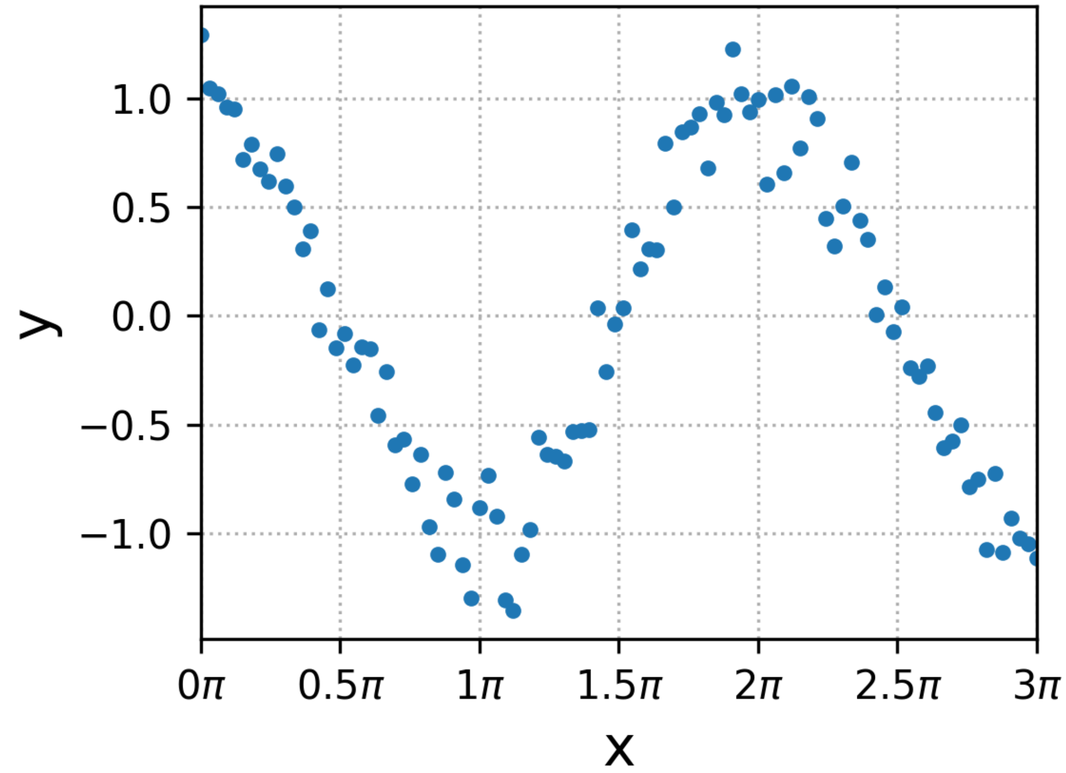
x_1	x_2	y
0	0	0
3	6	9
4.5	5.5	10
0	123	123
99	1	100
5	25	30

MOTIVATION

How does y depend on x?

x	y
0	0
1	1
2	4
3	9
4	16
5	25

x_1	x_2	y
0	0	0
3	6	9
4.5	5.5	10
0	123	123
99	1	100
5	25	30



FINDING FORMULAS

What is a formula?

- functions
- arguments
  - functions
  - input x_i
  - constants

$$f((x_1, x_2)) = \sin(a * x_1) + \exp\left(\frac{x_2}{b}\right)$$

$$+(\sin(*(a, x_1)), \exp(/(x_2, b)))$$

$$F_1 = \{+, -, *, /, \sin, \cos\} \rightarrow \tan(.) = /(\sin(.), \cos(.))$$
$$F_2 = \{+, -, *, /, \cos, \tan\} \rightarrow \sin(.) = *(\tan(.), \cos(.))$$

IDENTIFYING THE „BEST“ FORMULA: ACCURACY VS. COMPLEXITY

Which „metric“ to use?

Want a number for quantification

1. Accuracy: $|y - \hat{y}| = |y - f(x)| \rightarrow$ Loss
2. Length/complexity of the formula
   - Taylor series of $f(x)$: $\sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n$
   - Overfitting + Noise:
     True solution: $y = \sin(x)$
     Lowest loss: $f(x) = 1.005 * \sin(0.008 + 0.998 * x)$

   $\rightarrow$ favor shorter formulas, punish longer formulas

$$L_{\text{total}} = c_1 \underbrace{\mathcal{L}(f, \text{data})}_{\text{Accuracy}} + c_2 \underbrace{\mathcal{L}(f)}_{\text{Complexity}}$$

SYMBOLIC REGRESSION AS A CLASSIFICATION PROBLEM

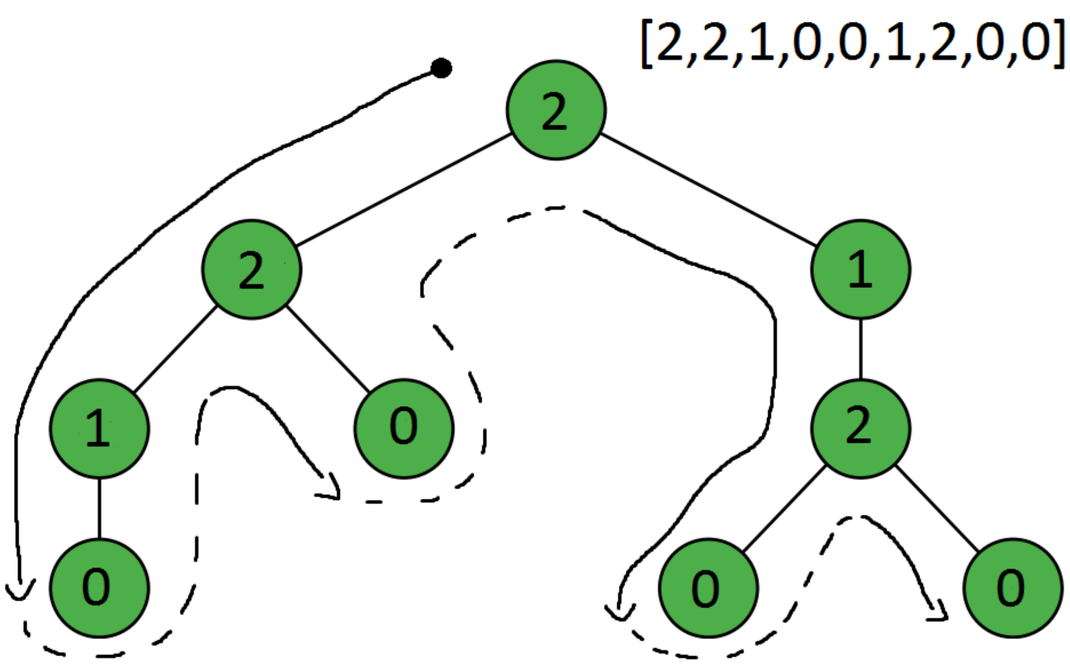
NP-hard

supervised learning task

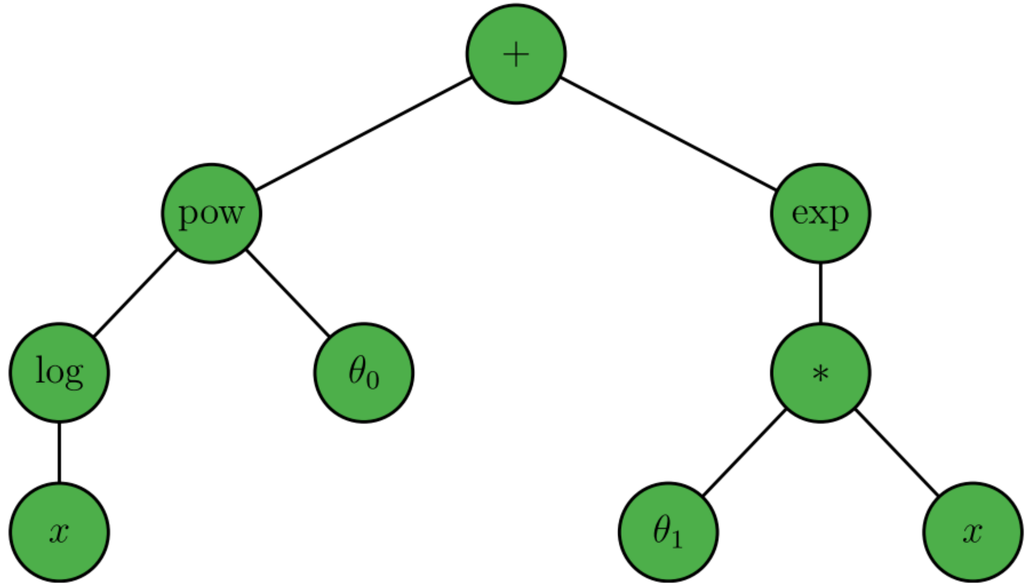
Multi-object optimisation (→ Pareto-optimal solution)

EXHAUSTIVE SEARCH

- Brute force
- Operators
  - functions & arguments
  - Types: nullary = 0, unary = 1, binary = 2
- Tree as operator list
- Complexity k = length of operator list



$(\log(x))^{\theta_0} + \exp(\theta_1 x) = [+, \text{pow}, \log, x, \theta_0, \exp, *, \theta_1, x]$



GENETIC PROGRAMMING

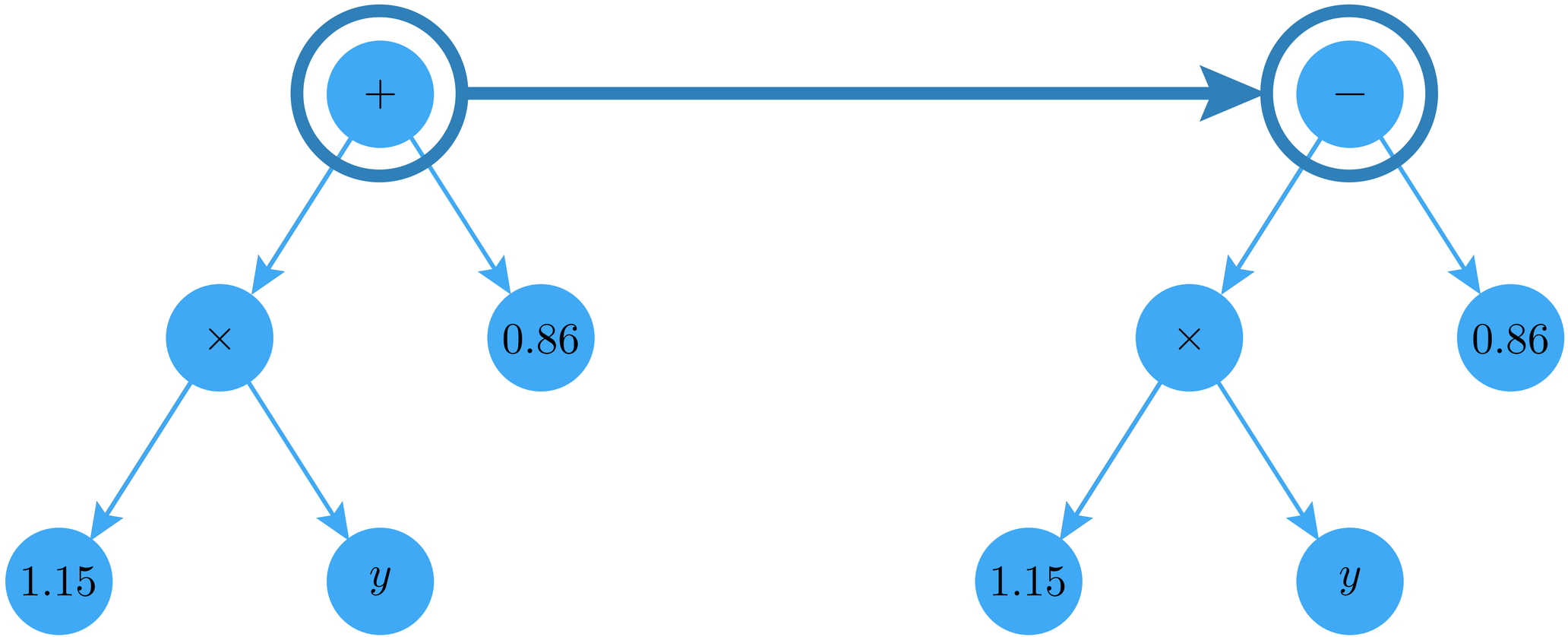
Basic algorithm:

- initialize a population of “individuals”
- repeat the evolution process:
  1. randomly choose a subset → competitors in “tournament”
  2. evaluated fitness of competitors
  3. select a winner
  4. replace the weakest/oldest individual with mutation of winner
- return fittest individual

→ splitting allows for parallelization

→ “Tournament selection”

GENETIC PROGRAMMING: MUTATION

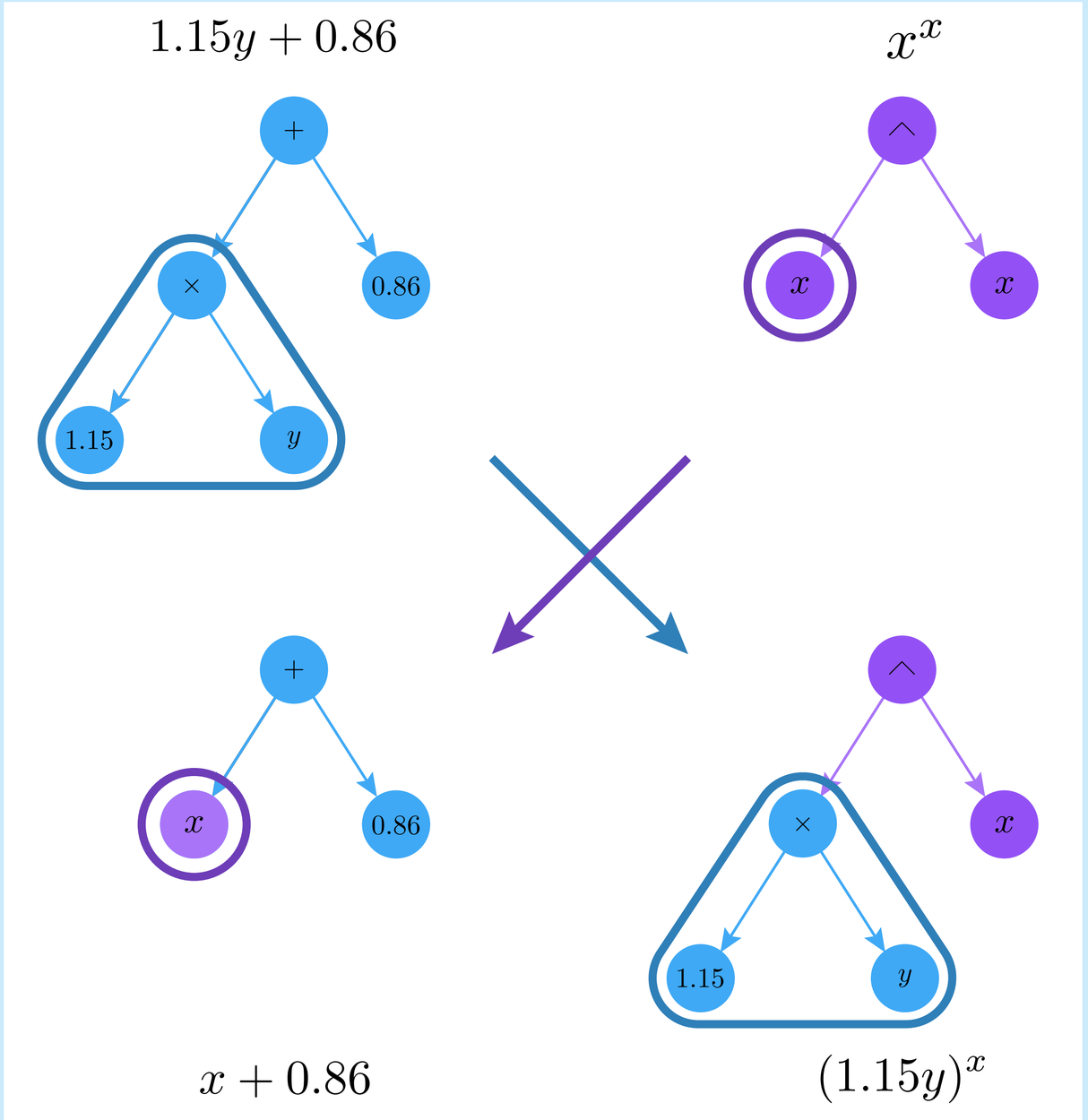


$1.15y + 0.86$

$1.15y - 0.86$

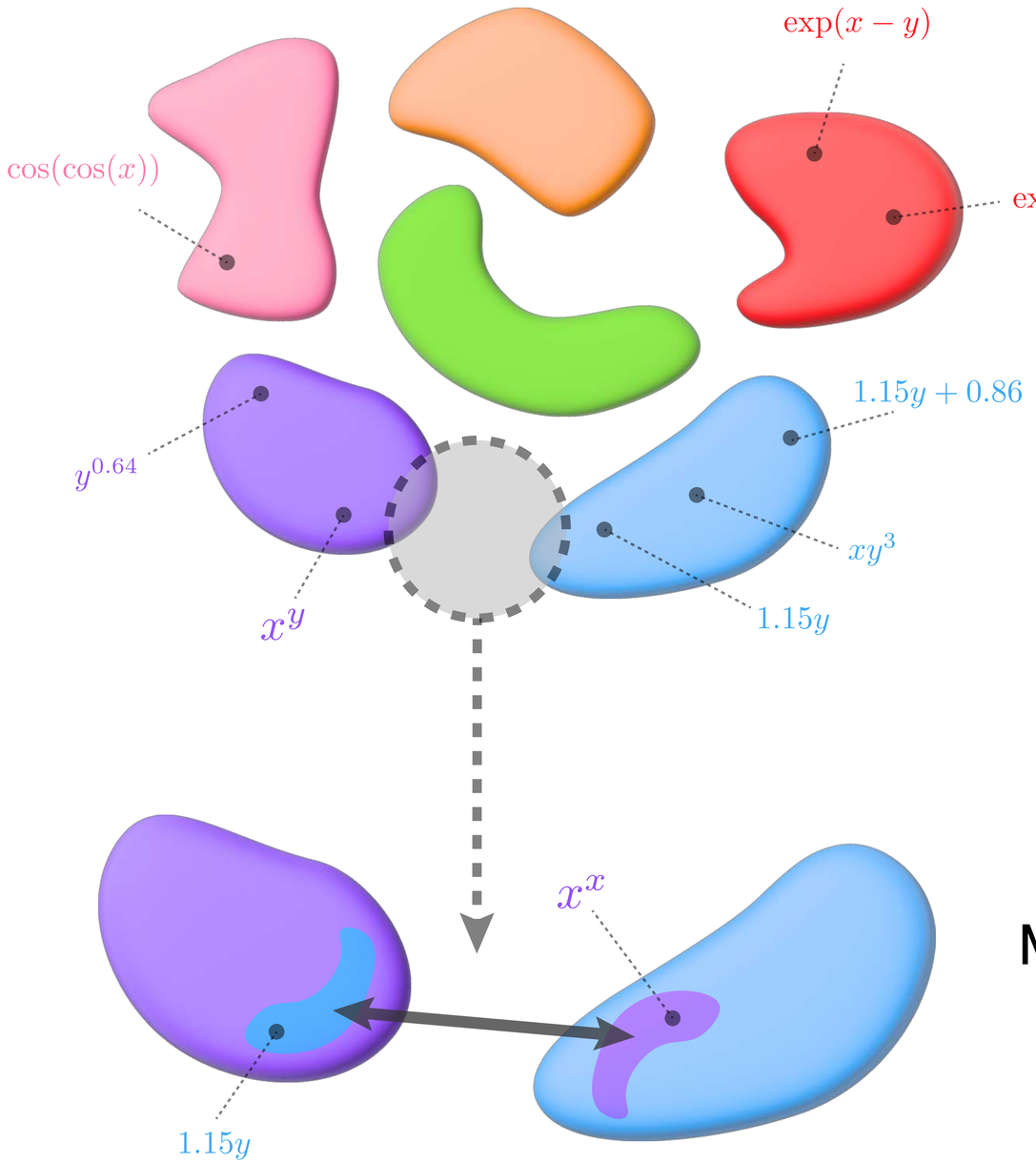
Cranmer+2023

GENETIC PROGRAMMING: CROSSOVER



Cranmer+2023

GENETIC PROGRAMMING: MIGRATION



Independent “islands" of expressions, each undergoing evolution

Migration between islands

Cranmer+2023

GENETIC PROGRAMMING: PYSR

- performs SR using genetic algorithm
- open source, backend in `julia`
- 2nd place in SRBench 2022
- evolve-simplify-optimize-loop
- custom operators (broken power laws, dilogarithm)
- returns best functions of several complexities

Cranmer+2023

PROS AND CONS OF GENETIC PROGRAMMING

+ exploration of huge search space
+ speedup by parallel processing
+ solution diversity
− problem should be low-dimensional
− can only optimize over discrete spaces
− exploration is probabilistic
− premature convergence

AI-FEYNMAN — SOLVE MYSTERIES

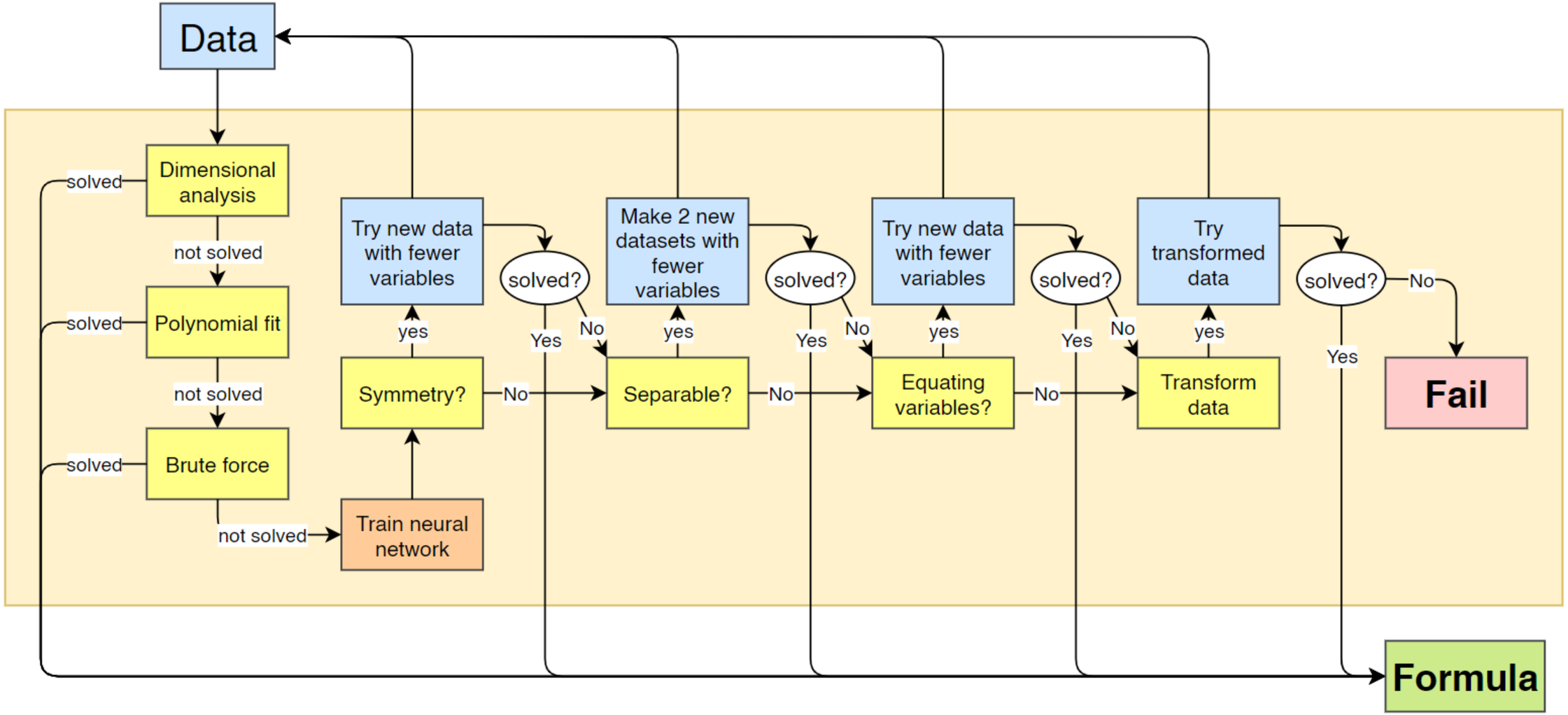
`https://github.com/SJ001/AI-Feynman`

- [2] one of the most cited paper about SR
- Use properties that often show up in physical/scientific equations
  - Units
  - Low order polynomial elements
  - Symmetry
  - Separability...
- Neural network for data interpolation

Udrescu&Tegmark 2020

AI-FEYNMAN — SOLVE MYSTERIES

Algorithm



Udrescu&Tegmark 2020

SR TRANSFORMER

Obtain function from DL model?

- idea: predict formulas from samples with a deep learning model
- 2022: “End-to-end symbolic regression with transformers” [6]

End-to-end symbolic regression with transformers

Pierre-Alexandre Kamienny*†[1,2], Stéphane d’Ascoli††[1,3], Guillaume Lample[1] and François Charton[1]

[1]Meta AI, Paris
[2]ISIR MLIA, Sorbonne Université, Paris
[3]Department of Physics, Ecole Normale Supérieure, Paris

Abstract

Symbolic regression, the task of predicting the mathematical expression of a function from the observation of its values, is a difficult task which usually involves a two-step procedure: predicting the "skeleton" of the expression up to the choice of numerical constants, then fitting the constants by optimizing a non-convex loss function. The dominant approach is genetic programming, which evolves candidates by iterating this subroutine a large number of times. Neural networks have recently been tasked to predict the correct skeleton in a single try, but remain much less powerful.

In this paper, we challenge this two-step procedure, and task a Transformer to directly predict the full mathematical expression, constants included. One can subsequently refine the predicted constants by feeding them to the non-convex optimizer as an informed initialization. We present ablations to show that this end-to-end approach yields better results, sometimes even without the refinement step. We evaluate our model on problems from the SRBench benchmark and show that our model approaches the performance of state-of-the-art genetic programming with several orders of magnitude faster inference.

Introduction

Inferring mathematical laws from experimental data is a central problem in natural science; having observed a variable y at n points $\{x_i\}_{i\in\mathbb{N}_n}$, it implies finding a function f such that $y_i \approx f(x_i)$ for all $i \in \mathbb{N}_n$. Two types of approaches exist to solve this problem. In *parametric statistics* (PS), the function f is defined by a small number of parameters that can directly be estimated from the data. On the other hand, *machine learning* (ML) techniques such as decision trees and neural networks select f from large families of non-linear functions by minimizing a loss over the data. The latter relax the assumptions about the underlying law, but their solutions are more difficult to interpret, and tend to overfit small experimental data sets, yielding poor extrapolation performance.

Symbolic regression (SR) stands as a middle ground between PS and ML approaches: f is selected from a large family of functions, but is required to be defined by an interpretable analytical expression. It has already proved extremely useful in a variety of tasks such as inferring physical laws [1, 2].

SR is usually performed in two steps. First, predicting a “skeleton”, a parametric function using a pre-defined list of operators – typically, the basic operations $(+, \times, \div)$ and functions $(\mathrm{sqrt}, \exp, \sin,$ etc.). It determines the general shape of the law up to a choice of constants, e.g. $f(x) = \cos(ax + b)$. Then, the constants in the skeleton (a, b) are estimated using optimization techniques, typically the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS).

*pakamienny@fb.com
†stephane.dascoli@gmail.com
†Equal contribution.

SR TRANSFORMER: PROS AND CONS

+ fast inference
+ prediction of simple equations

  $\rightarrow$ training data only contains simple equations
- selection of operators is fixed
- still worse than GP

  $\rightarrow$ could initialize evolutionary method with transformer results

WHERE TO FIND THE LECTURE MATERIAL

https://github.com/TobiBu/graddays


Operator Learning

UNIVERSAL APPROXIMATION THEOREM FOR NONLINEAR OPERATORS

UAT – Operators

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

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

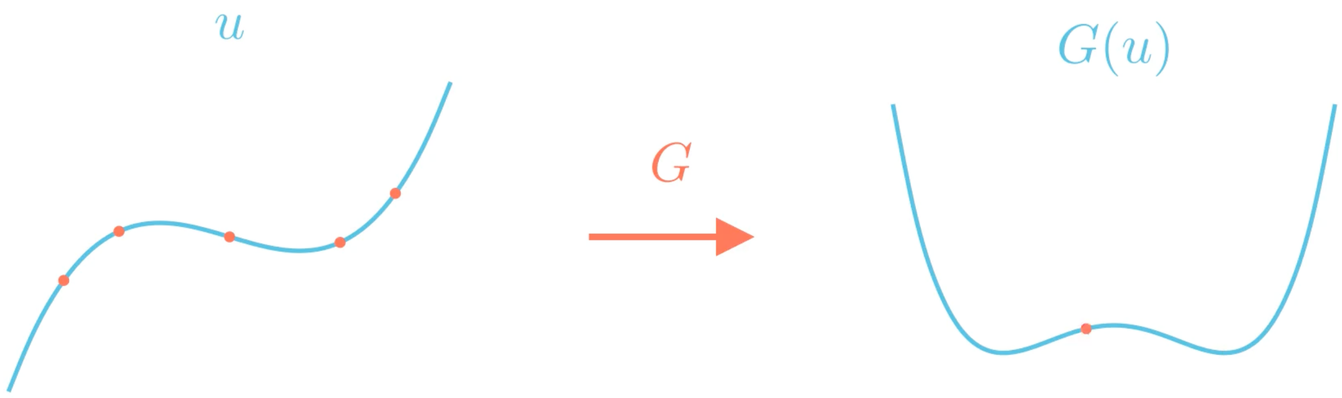
There is a function $f_{c,\xi,\theta,\zeta}(\boldsymbol{x}, y)$ such that for every $\epsilon > 0$

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

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

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

OPERATOR LEARNING: INPUT AND OUTPUT

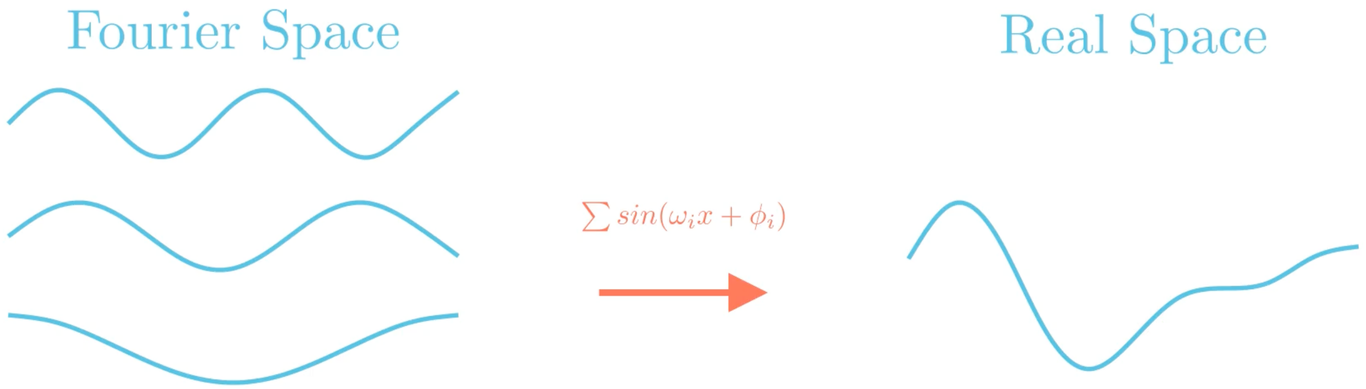


Problem: Can not put a function into an NN…

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

More flexibility: "Arbitrary" output location y

INTERMISSION: GAUSSIAN RANDOM FIELDS



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

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

1. Fourier Space

   randomly sample phase φ_i

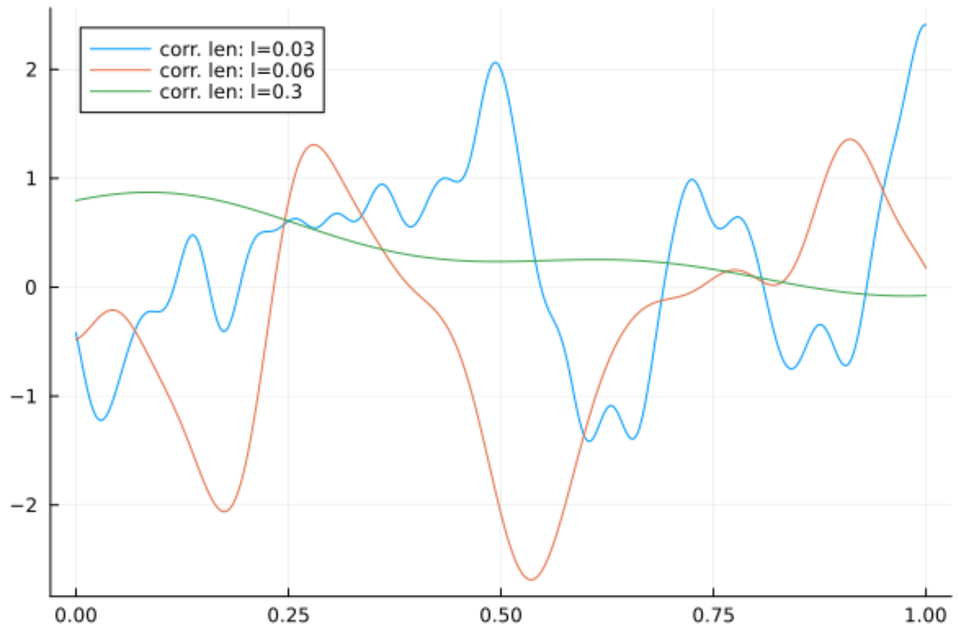
2. Real Space

   generate random function

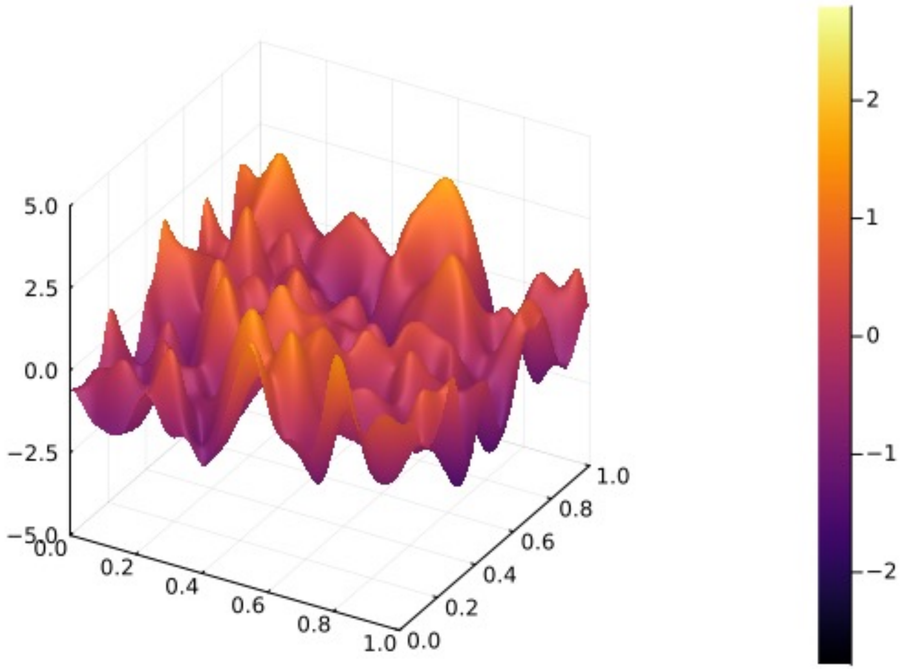
   Wavelength → Pattern size

INTERMISSION: GAUSSIAN RANDOM FIELDS

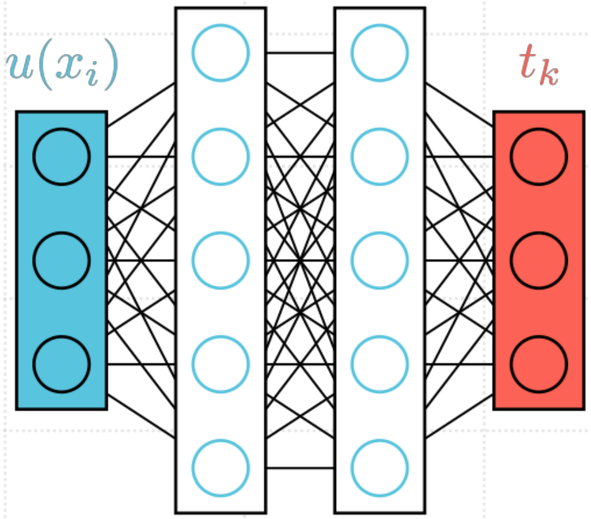
1D

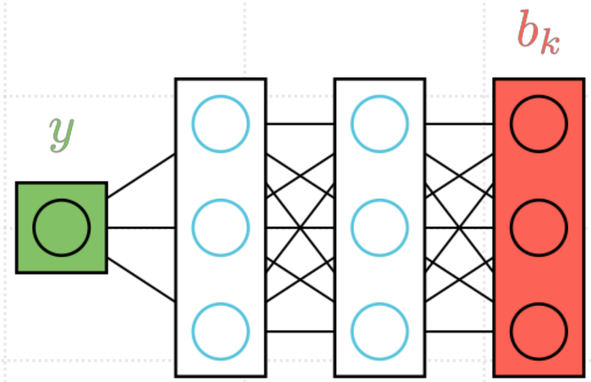


2D



DEEP OPERATOR ARCHITECTURE





Idea: Use inductive bias

→ 2 Separate Neural Networks

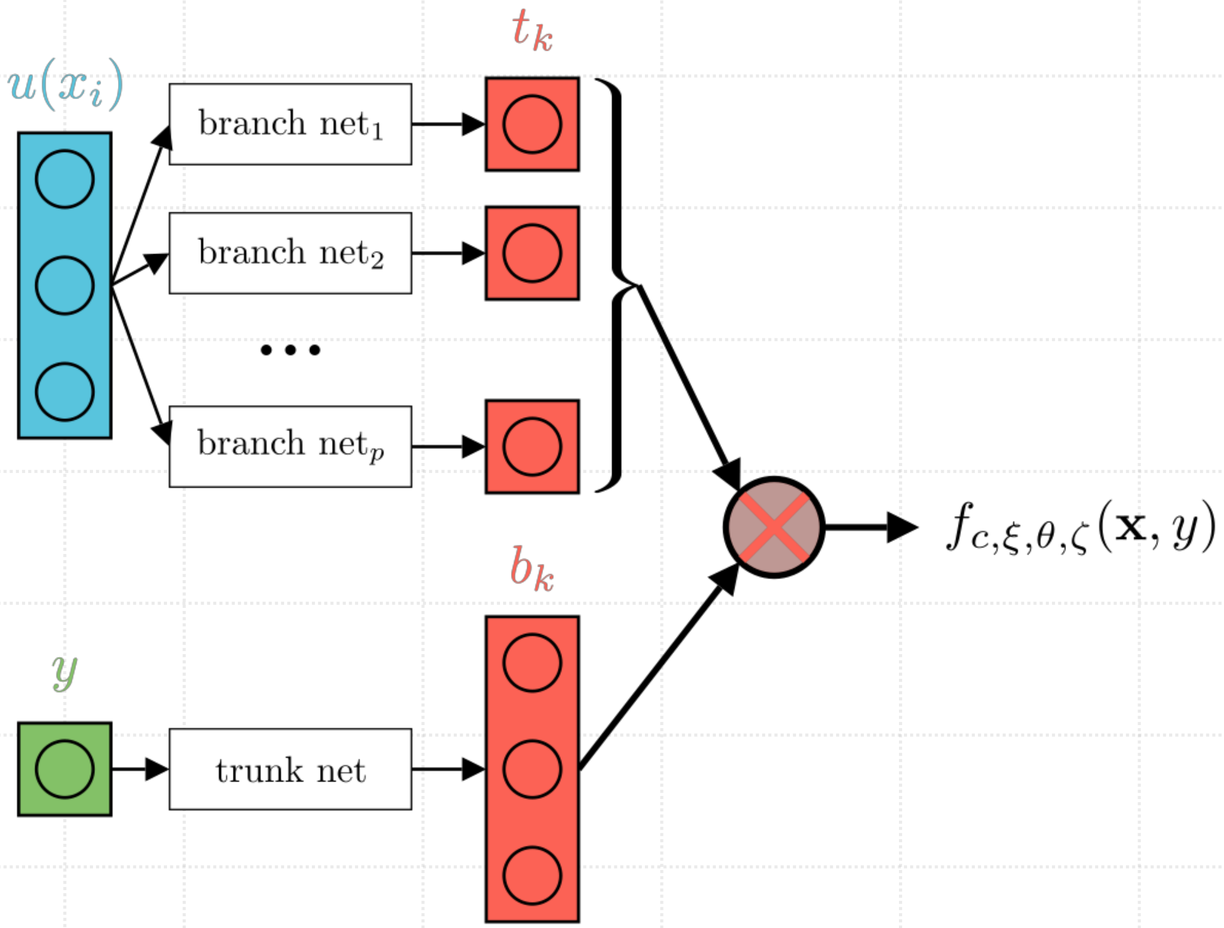
1. NN: **“Branch Net“**

$u(x)$ → encode input space

2. NN: **“Trunk Net“**

y → encode domain of output function

DEEP OPERATOR ARCHITECTURE



Theory – Approximation Theorem:

1. Branch Net:

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

2. Trunk Net:

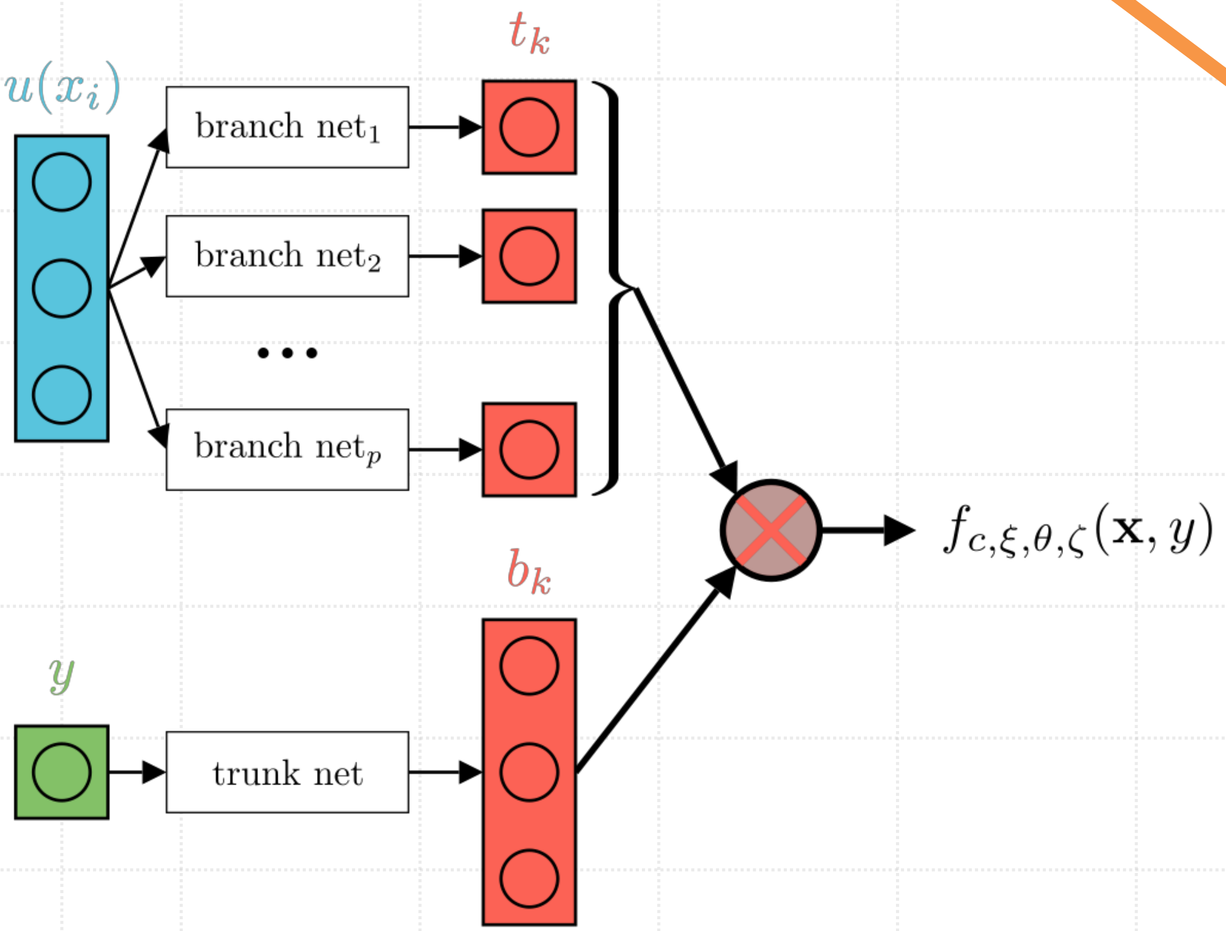
$$b_k = \sigma(w_k \cdot y + \zeta_k)$$

3. Output: $G(u)$

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

DEEP OPERATOR ARCHITECTURE

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



Theory – Approximation Theorem:

1. Branch Net:

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

2. Trunk Net:

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

3. Output: $G(u)$

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

DEEP OPERATOR TRAINING

- Training Dataset:

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

m (function values) → Branch; 1 (Loc.) → Trunk; 1 (Target Output)

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

DEEP OPERATOR TRAINING

Using Simulated Data:

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

➜ DeepONet has learned Operator

Real Data:

1. Data preparation
2. Train NN on Data

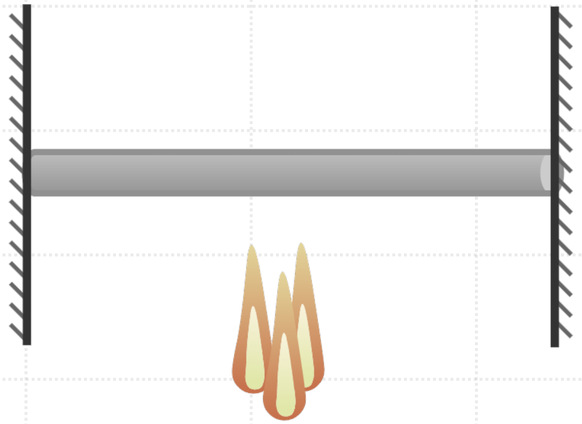
➜ DeepONet has learned Operator

DEEP OPERATOR LEARNING: APPLICATION EXAMPLE**

- DeepONet applied to 1D Heat Diffusion System:

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

1) 2)



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

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

** You find this example as a tutorial on colab!

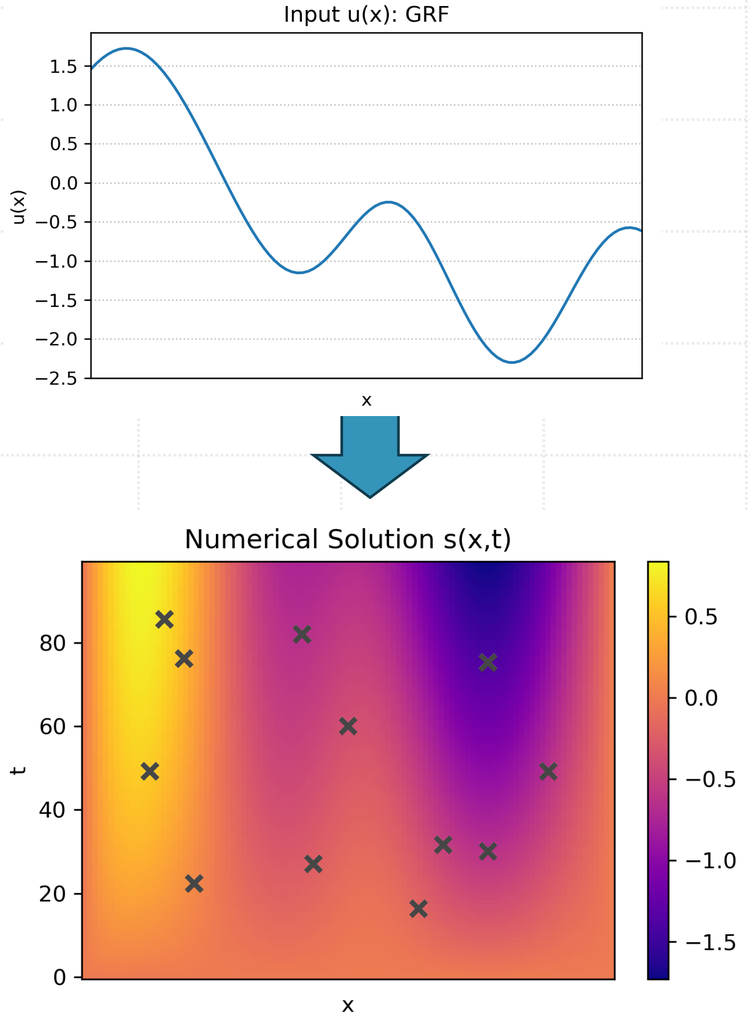
DEEP OPERATOR LEARNING: APPLICATION EXAMPLE

Code Example – Process

Using DeepXDE: [5]

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

→ Train Model on Datapoints

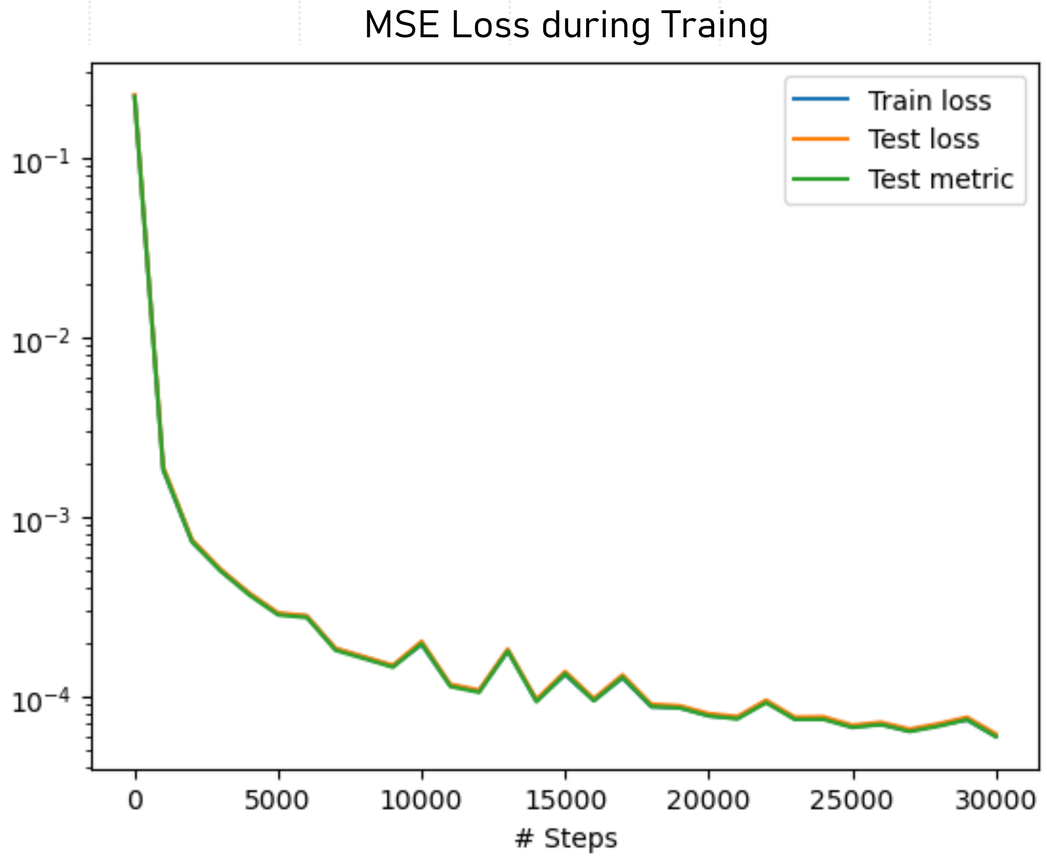


DEEP OPERATOR LEARNING: APPLICATION EXAMPLE

Code Example – Training

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

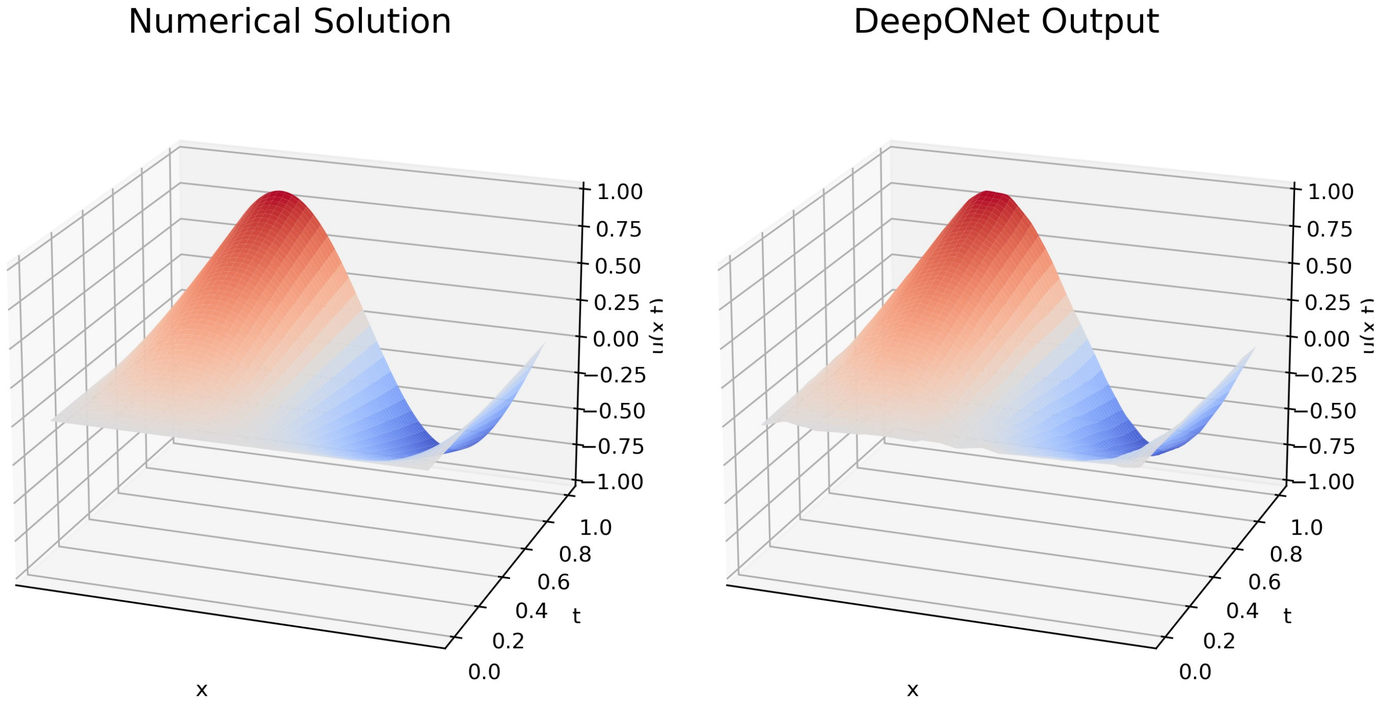
→ ~6h Training



DEEP OPERATOR LEARNING: APPLICATION EXAMPLE I

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

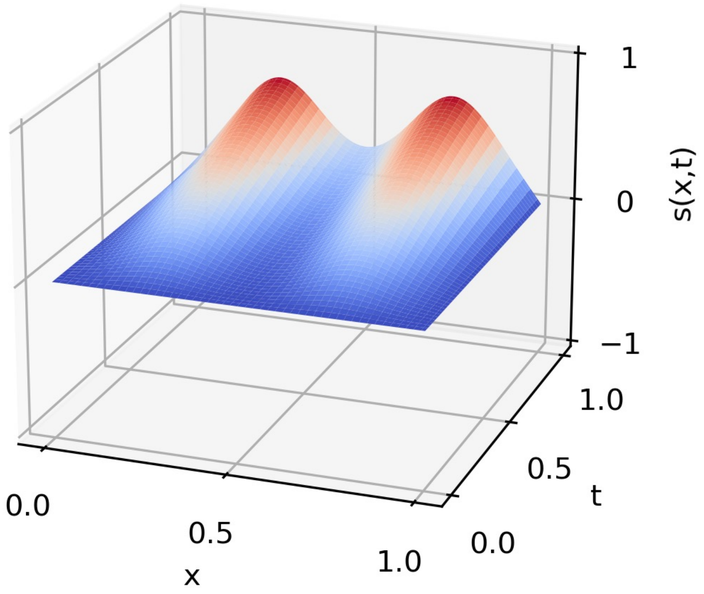
Numerics	DeepONet
-	Error: ~10^{-3}
One full simulation	NN inferences
Runtime: $9.18ms$	Time: $2.86ms$



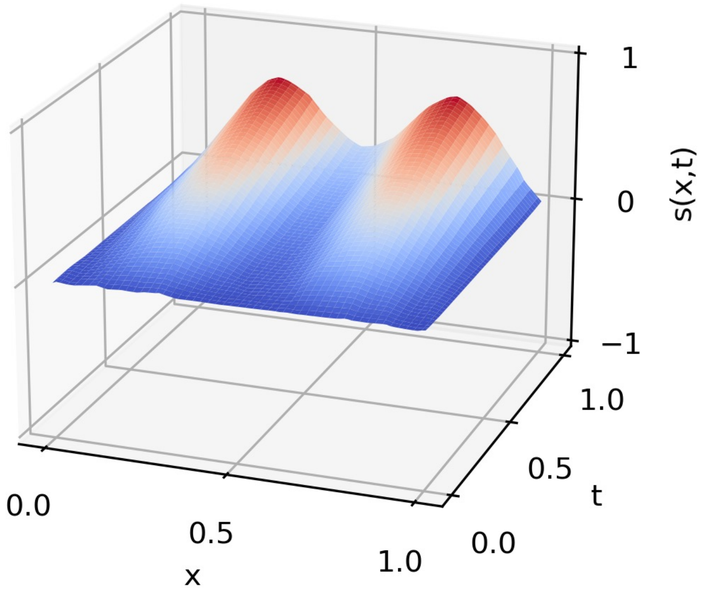
DEEP OPERATOR LEARNING: APPLICATION EXAMPLE II

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

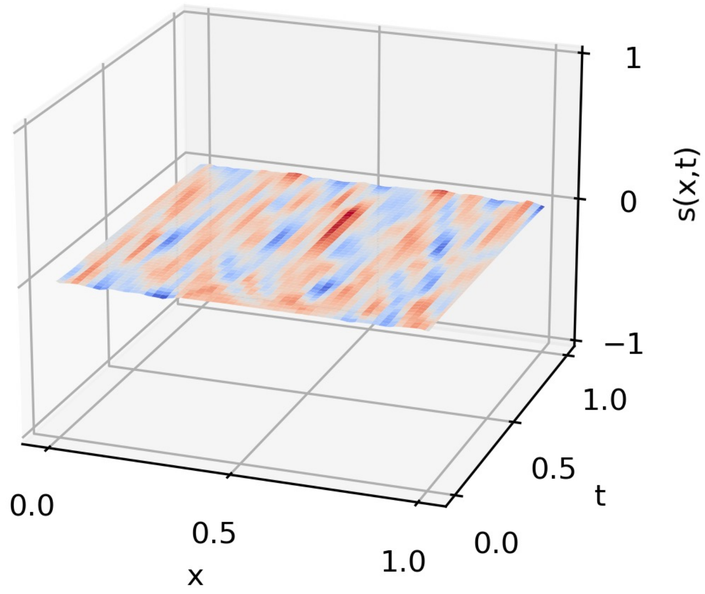
Numerical Solution



DeepONet Output

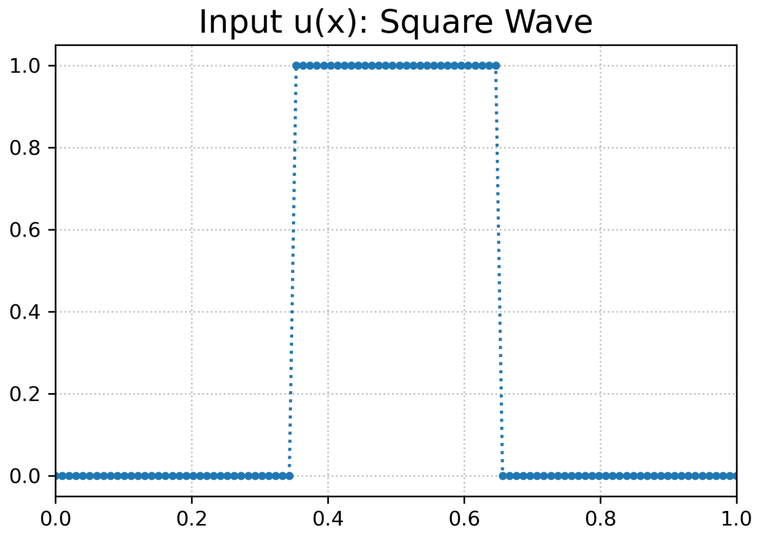


Pointwise Error

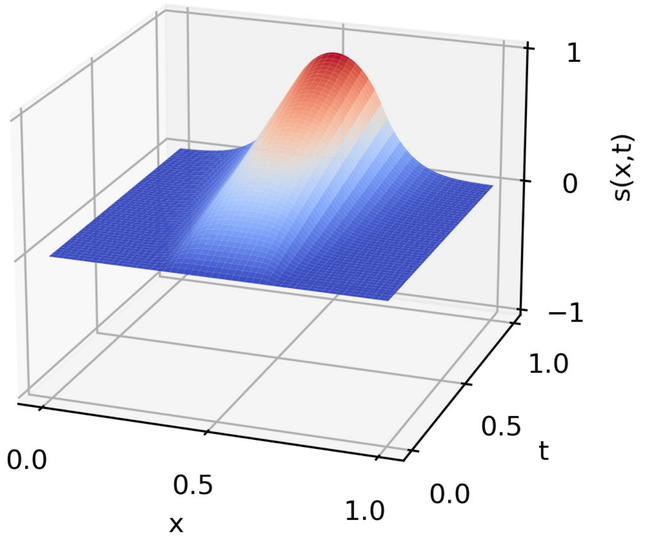


DEEP OPERATOR LEARNING: APPLICATION EXAMPLE III

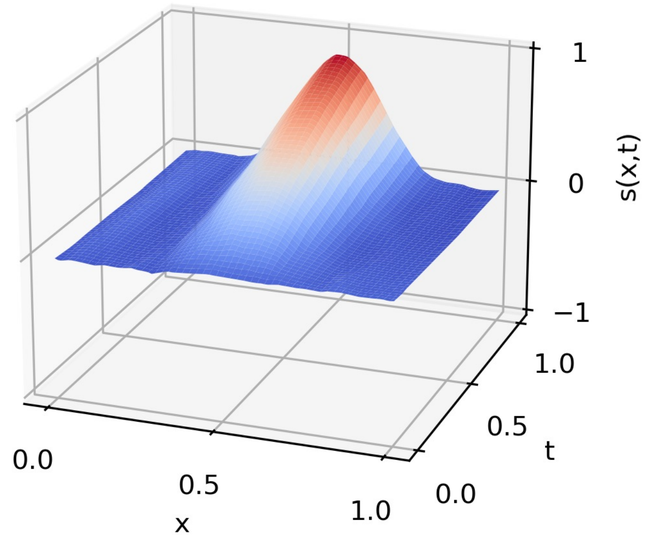
source term: $u(x)$: Square Wave



Numerical Solution



DeepONet Output



DISCUSSION: WHEN TO USE DEEPONET?

Whenever there is a continuous operator!

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

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

ADVANTAGES AND DISADVANTAGES OF DEEPONET

Pros	Cons
Learns the “solution space”	No continuous output
Fast and very efficient	“Curse of dimensionality”
High flexibility	Requires training and data generation
Generalizes well	Dependency on sensor locations and input data
“Data-only” mode	

USE CASES FOR DEEPONETS

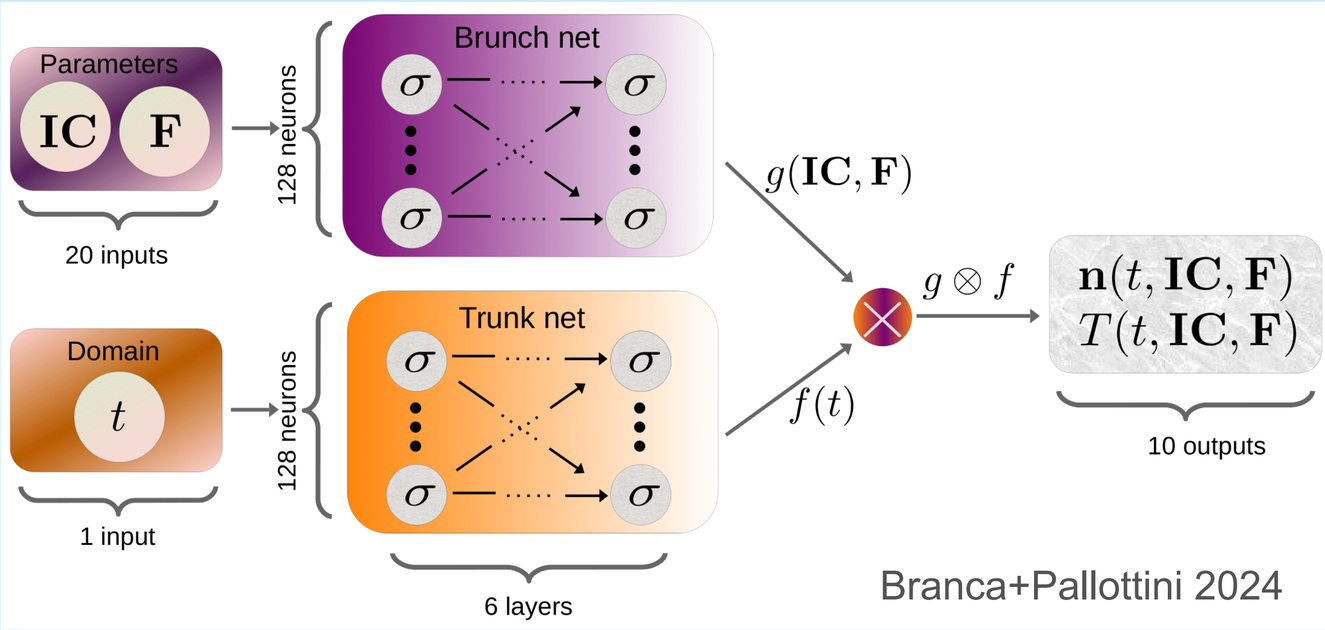
Two Scenarios:

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

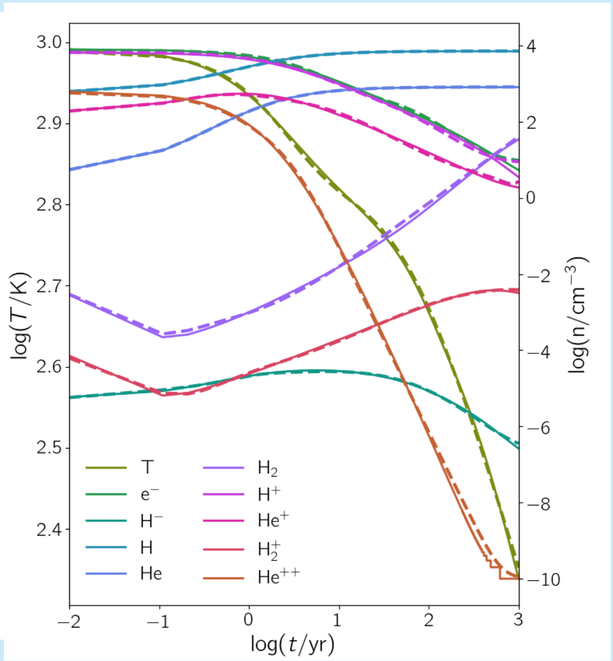
Use DeepONet to replace repeated (complex) numerical simulations

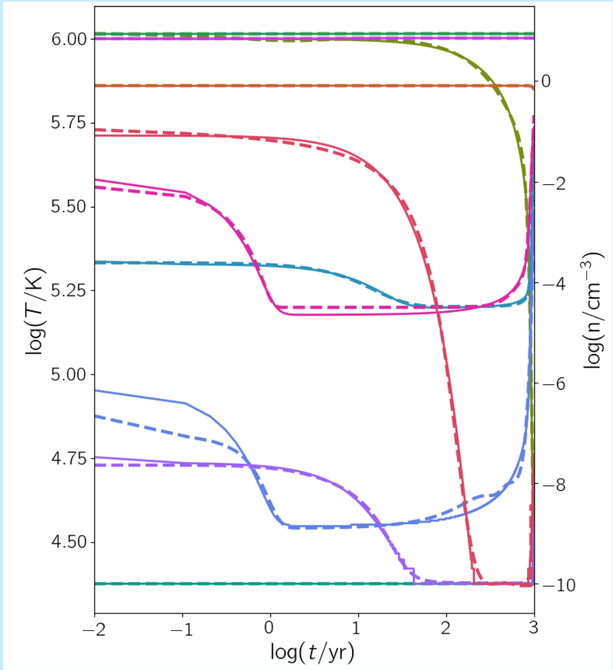
DEEP OPERATORS IN ASTROPHYSICS

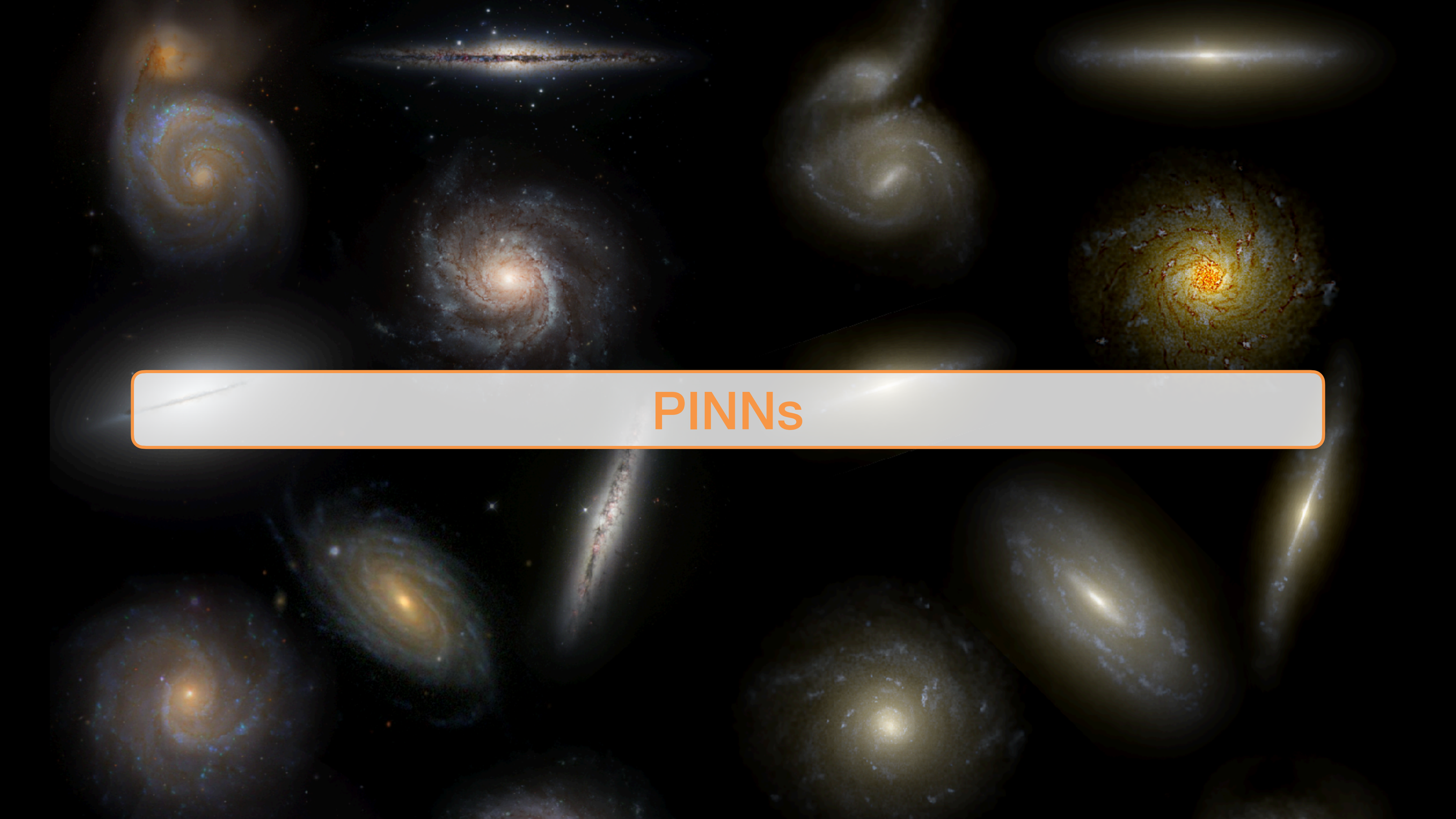
- PPDONet: Deep Operator Networks for Fast Prediction of Steady-State Solutions in Disk-Planet Systems (Mao+2023)
- Emulating the interstellar medium chemistry with neural operators (Branca+Pallottini 2024)
- CODES Benchmark for neuralODEs and Operator Learning for astrochemistry (Janssen,Sulzer+Buck 2024)



Branca+Pallottini 2024






PINNs

PINNS VS. NEURAL ODES

PINNs = Physics Informed Neural Networks

Neural ODE

$$\frac{\mathrm{d}x}{\mathrm{dt}} = f_\theta(t, x(t))$$

neural net r.h.s. of DE

PINN

$$x(t) = f_\theta(t, x_0)$$

neural net approx.
solution of DE

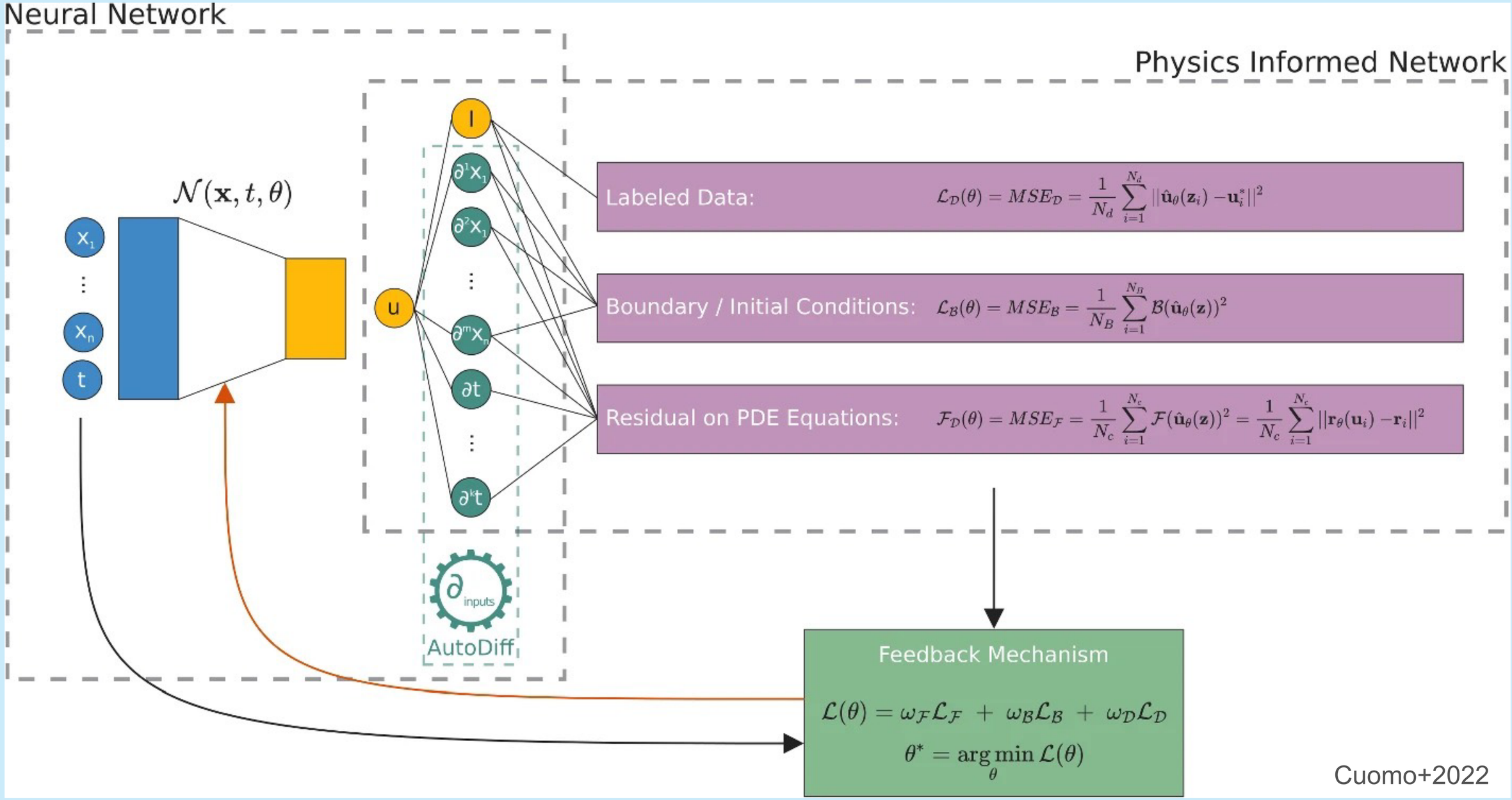
Hamiltonian Neural Net

$$\frac{\mathrm{d}q}{\mathrm{d}t} = \frac{\partial H_\theta(p, q)}{\partial p}$$

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -\frac{\partial H_\theta(p, q)}{\partial q}$$

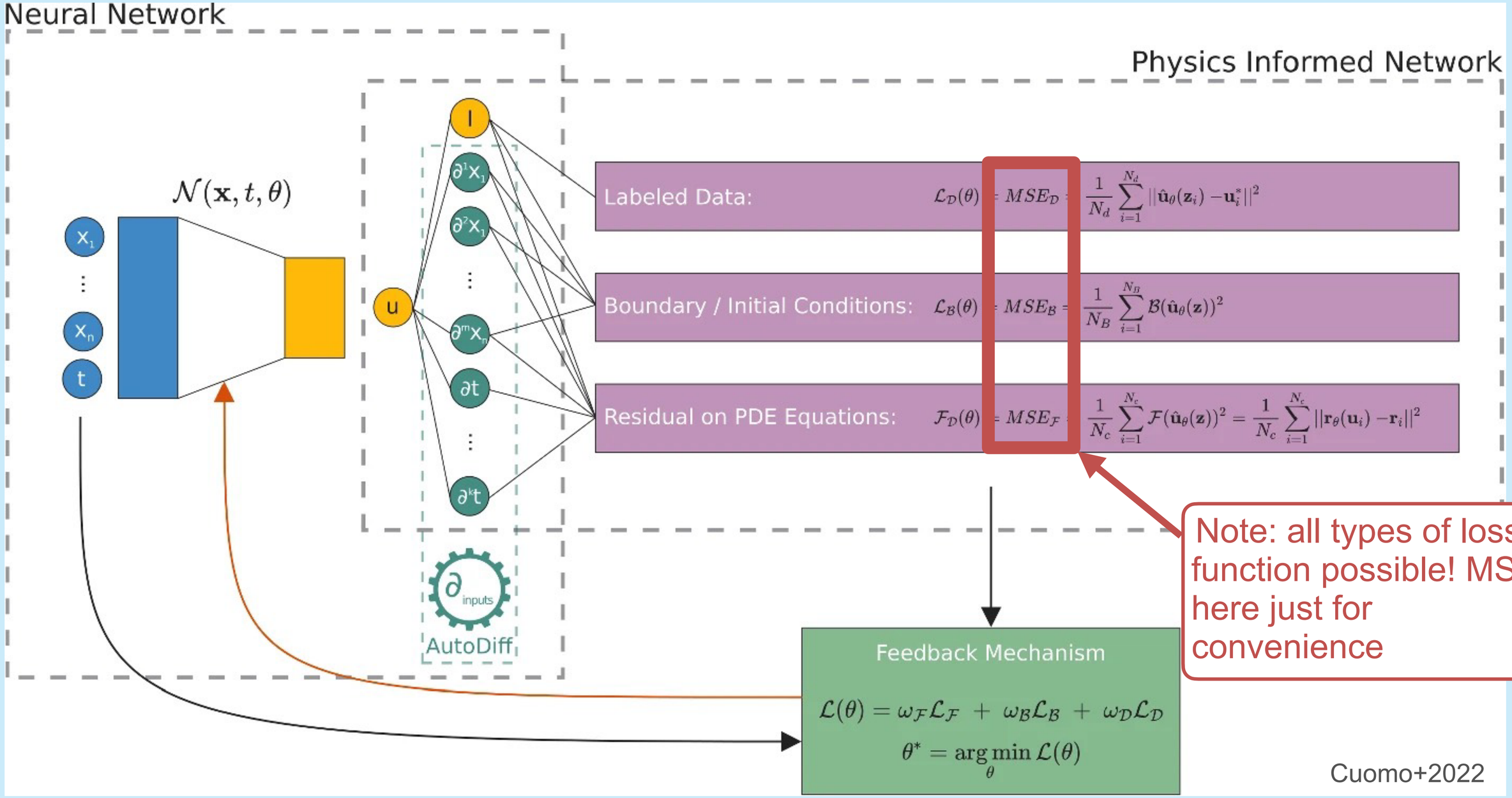
neural net represents a
Hamiltonian

PINNS VS. NEURAL ODES



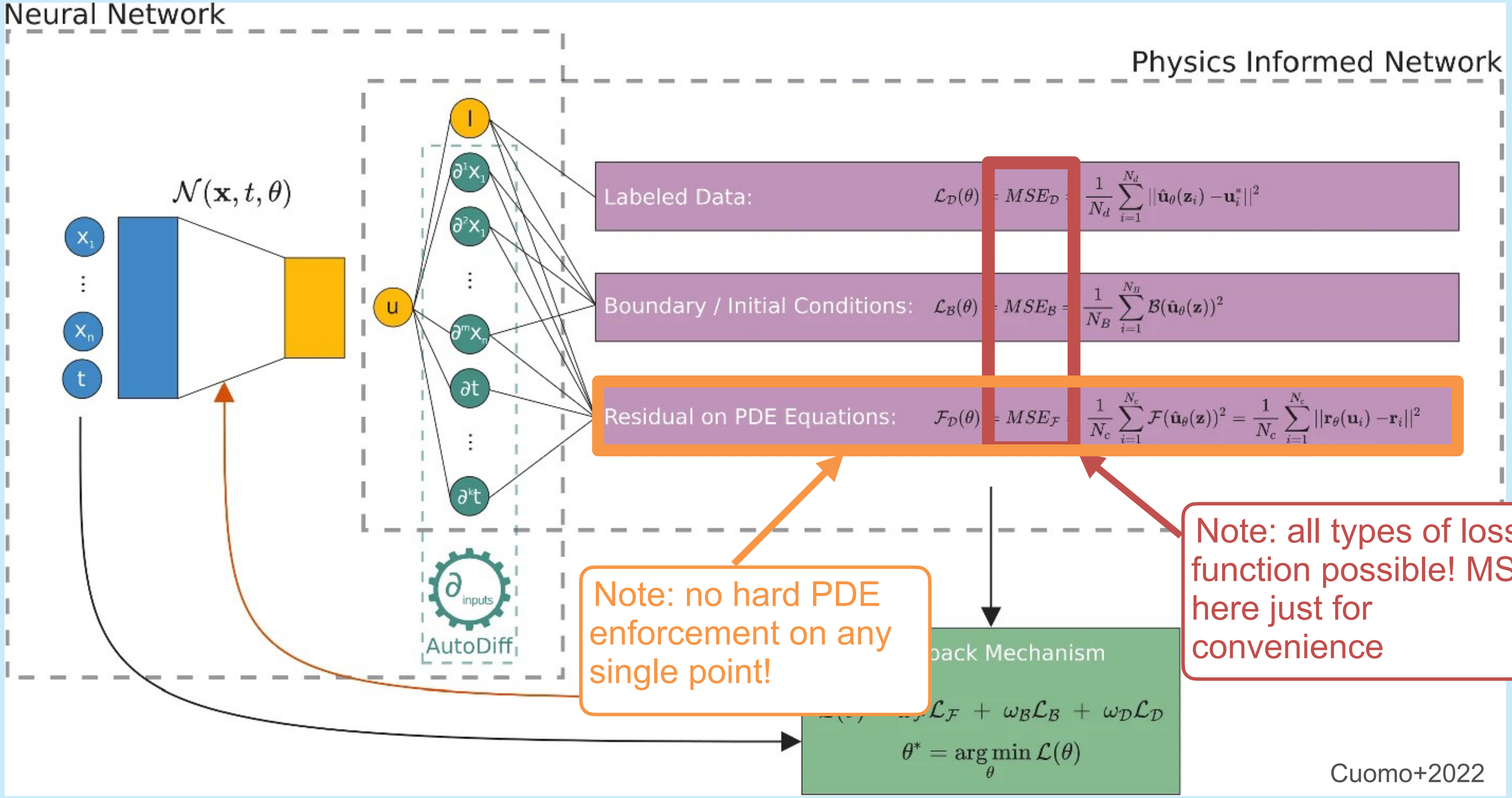
Cuomo+2022

PINNS VS. NEURAL ODES



Cuomo+2022

PINNS VS. NEURAL ODES



Cuomo+2022

PHYSICS INFORMED NEURAL NETS

Loss function for PINNs

Differential Equation: $\mathcal{F}[u(x,y)] = f(x,y)$

Dataset: $(x_i, y_i, u_i);\ i = 1, \dots, N_{data}$

Collocation points: $(x_j, y_j); j = 1, \dots, N_C$

Initial Condition: (x_o, y_0, u_0)

$$L_{DiffEq} = \frac{1}{N_c}\sum_{j=1}^{N_c}\left(\mathcal{F}[u(x_j, y_j)] - f(x_j, y_j)\right)^2$$

$$L_{total} = \omega_{data} \cdot L_{data} + \omega_{DiffEq} \cdot L_{DiffEq} + \omega_{IC} \cdot L_{IC}$$

$$L_{data} = \frac{1}{N_{data}}\sum_{i=1}^{N_{data}}(u(x_i, y_i) - u_i)^2$$

$$L_{IC} = (u(x_0, y_0) - u_0)^2$$

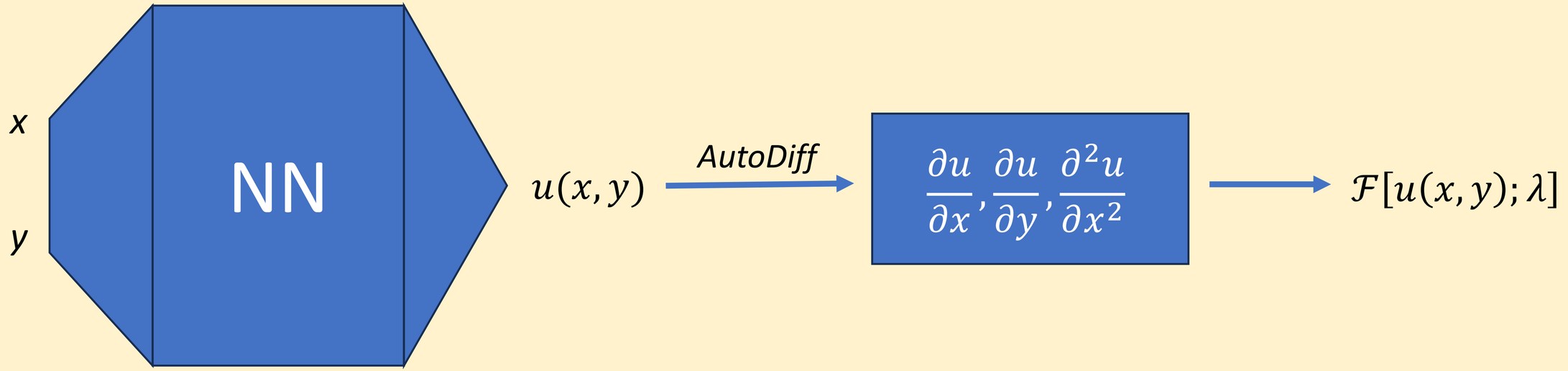
PINNS: EXAMPLE

- Example: $\mathcal{F}[u(x,y);\lambda] = \frac{\partial u}{\partial x} + \lambda_1 \frac{\partial u}{\partial y} + \lambda_2 \frac{\partial^2 u}{\partial x^2}$ and $f(x,y) = x^2 + y$
- In order to calculate $\mathcal{F}$, we need the derivatives: $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial^2 u}{\partial x^2}$

➔ Automatic differentiation

$$\mathcal{F}[u(x,y);\lambda] = f(x,y)$$

$$\lambda := \{\lambda_1, \lambda_2\}$$



PINNS: EXAMPLE

- Example: $\mathcal{F}[u(x,y);\lambda] = \frac{\partial u}{\partial x} + \lambda_1 \frac{\partial u}{\partial y} + \lambda_2 \frac{\partial^2 u}{\partial x^2}$ and $f(x,y) = x^2 + y$
- In order to calculate $\mathcal{F}$, we need the derivatives: $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial^2 u}{\partial x^2}$

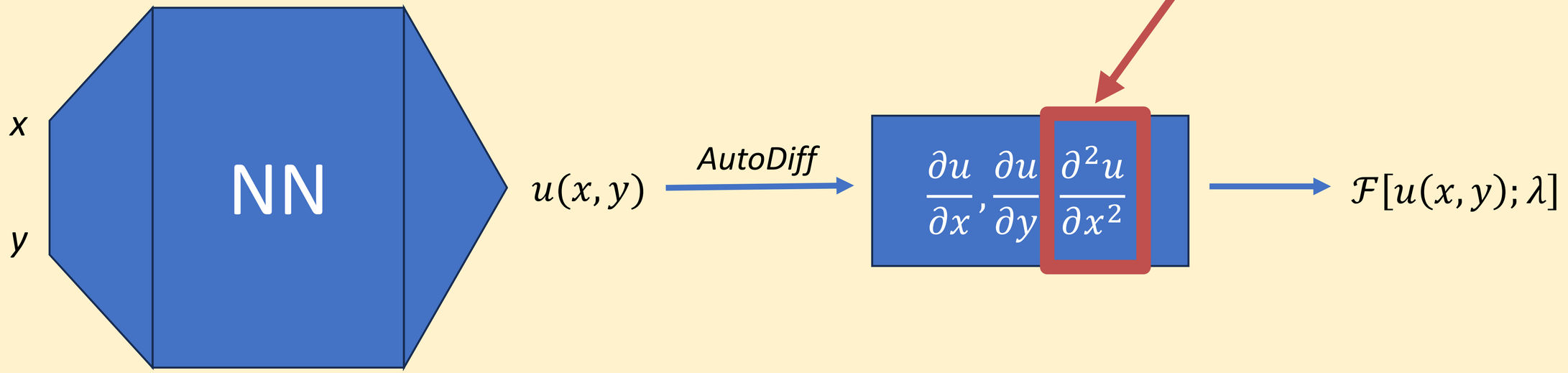
➔ Automatic differentiation

$$\mathcal{F}[u(x,y);\lambda] = f(x,y)$$

$$\lambda := \{\lambda_1, \lambda_2\}$$

Careful with RELU!

Vanishing Hessian



NEURAL ODES VS. PINNS

NDEs are a *modelling approach*:

- Learn θ in the model $^{dy}/_{dt}(t) = f_\theta(t, y(t))$, so that you match your data.
- Loss function: $\min_\theta \| y(t) - data(t) \|^2$

PINNs are a *numerical method:*

- Learn φ in $y(t)=y_\varphi(t)$, so that it matches the diffeq $^{dy}/_{dt}(t) = f(t, y(t))$.
- Loss function: $\min_\varphi \| ^{dy}/_{dt}(t) - f(t, y(t)) \|^2$

These terms have precise meaning, but the terms often muddled up, as sometimes both techniques are done at the same time!

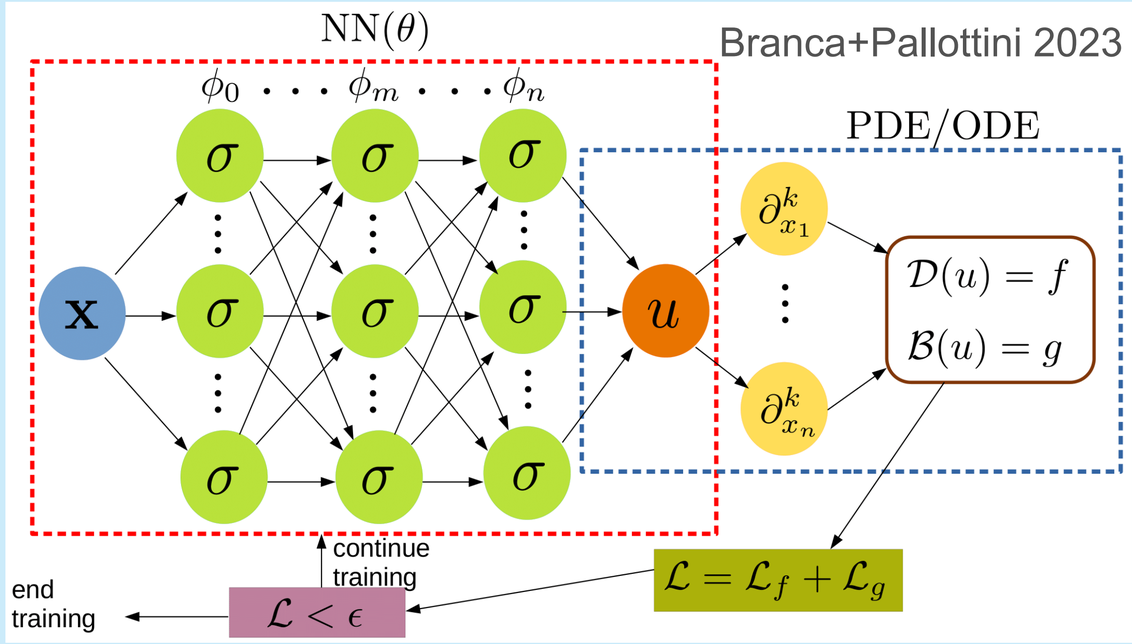
- Loss function: $\min_{\theta,\varphi}(\| y(t) - data(t) \|^2 + \| ^{dy}/_{dt}(t) - f(t, y(t)) \|^2)$

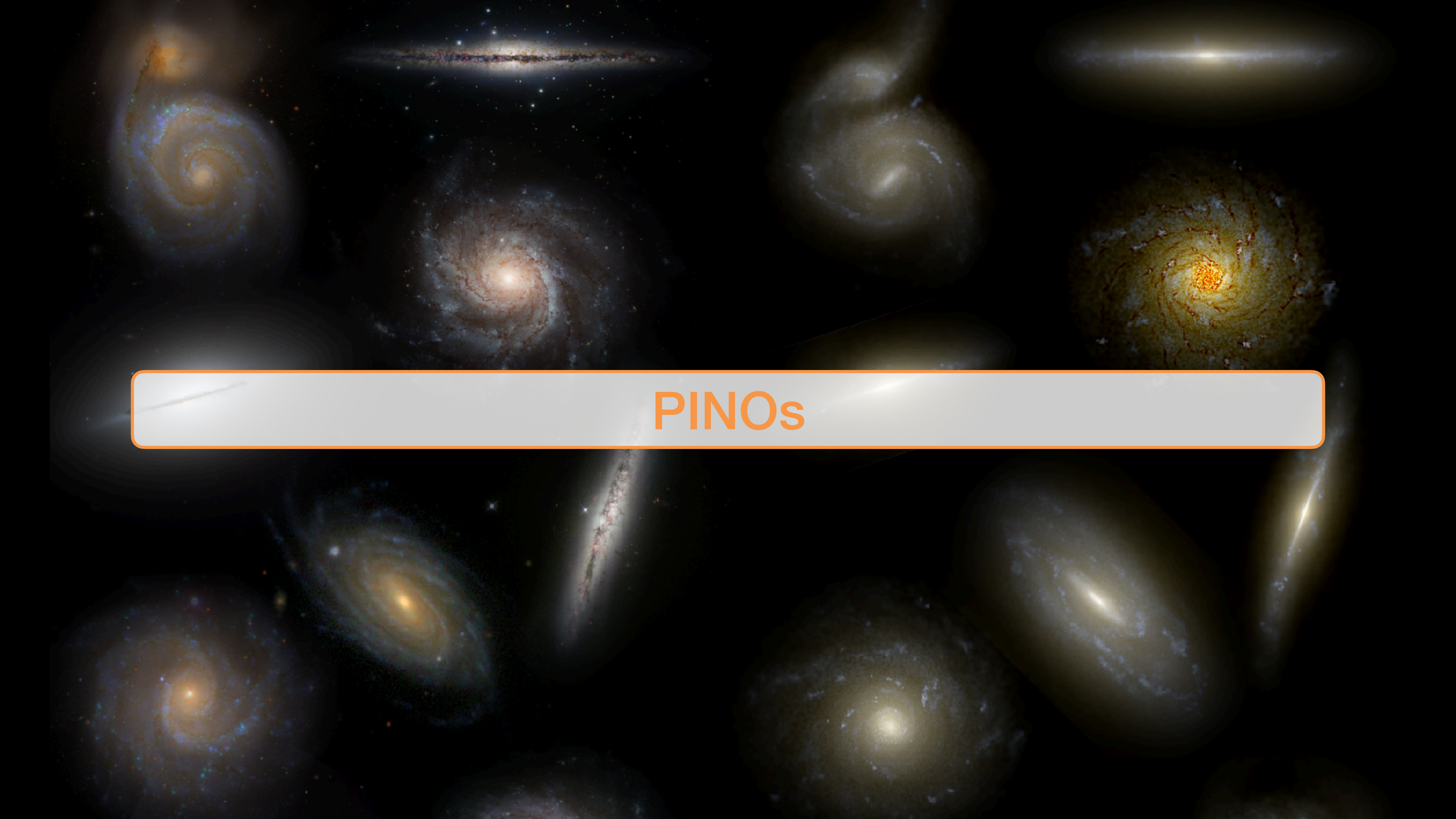
PINNs are useful when traditional numerical methods fail (high-dimensional PDEs; nonlocal integral operators.). But mostly… just use traditional numerical methods.

Slides taken from Patrick Kidger

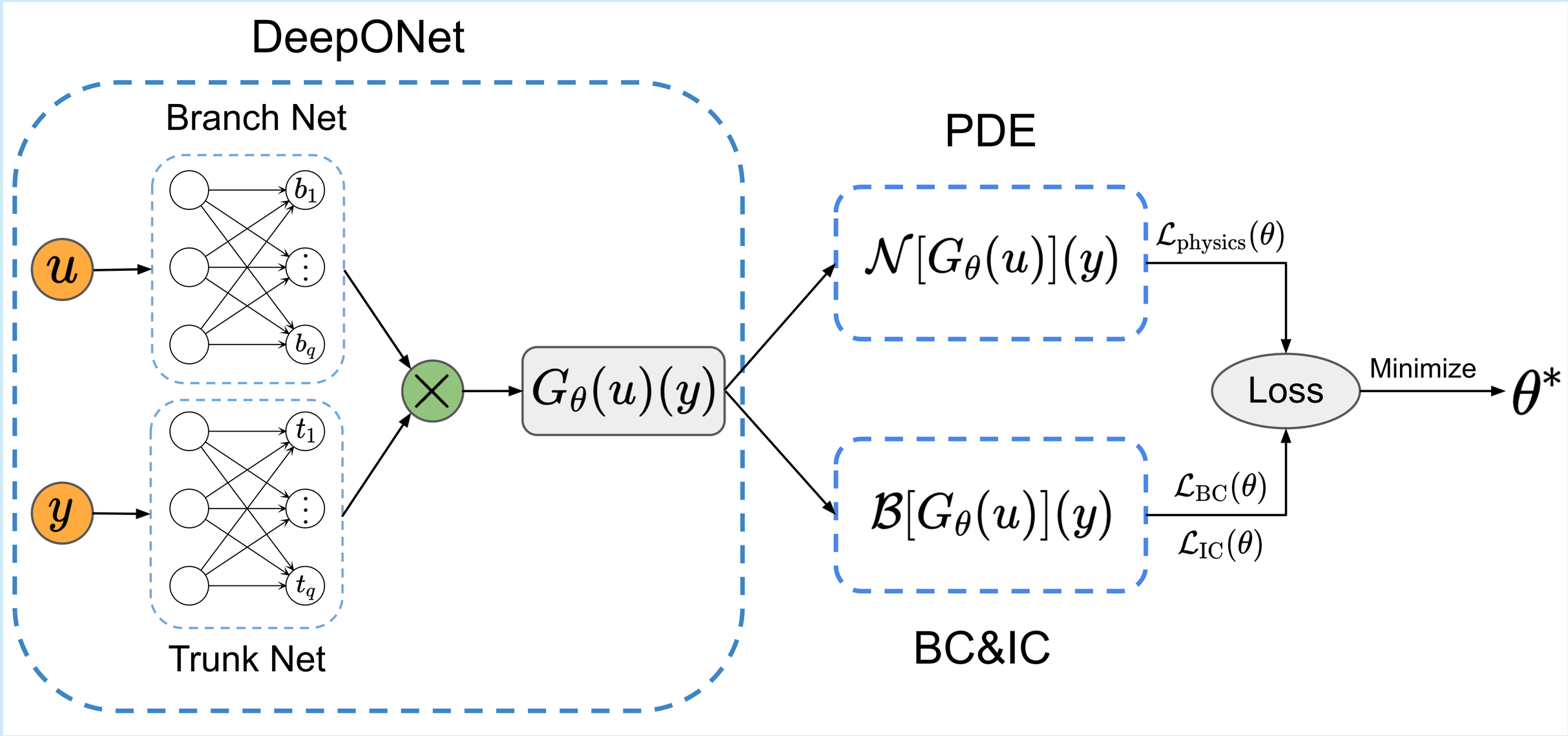
PINNS IN ASTROPHYSICS

- Physics-informed neural networks for modeling astrophysical shocks (Moschou+2023)
- Probing the solar coronal magnetic field with physics-informed neural networks (Jarolim+2022)
- Physics-informed neural networks in the recreation of hydrodynamic simulations from dark matter (Dai+2023)
- Physics informed neural networks for simulating radiative transfer (Mishra+Molinaro 2021)
- Neural networks: solving the chemistry of the interstellar medium (Branca+Pallottini 2023)




PINOs

PHYSICS INFORMED DEEP OPERATOR



Wang+2021

PHYSICS INFORMED DEEP OPERATOR
EXAMPLE: ANTI-DERIVATIVE OPERATOR

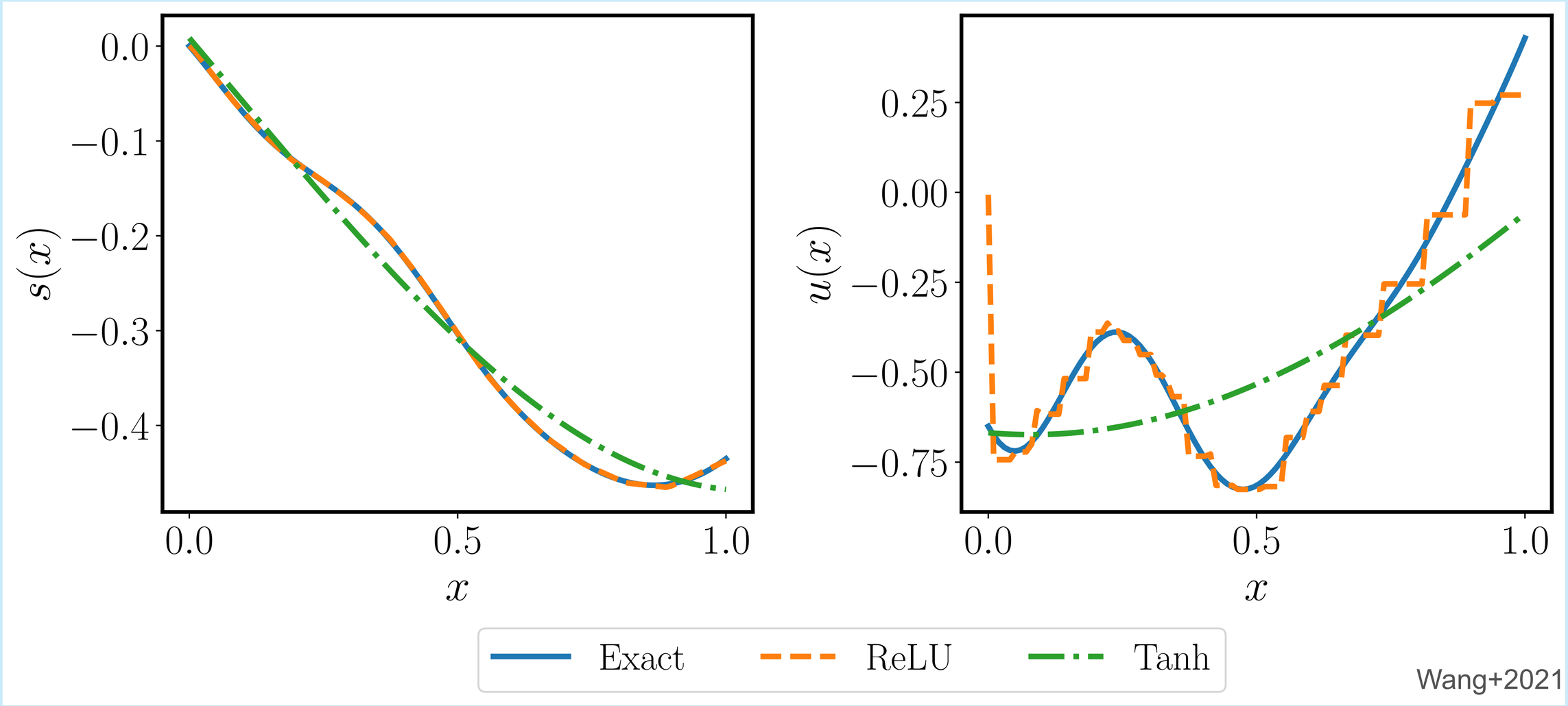
Consider this initial value problem with $s(0) = 0$

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

Goal is to learn to the anti-derivative operator

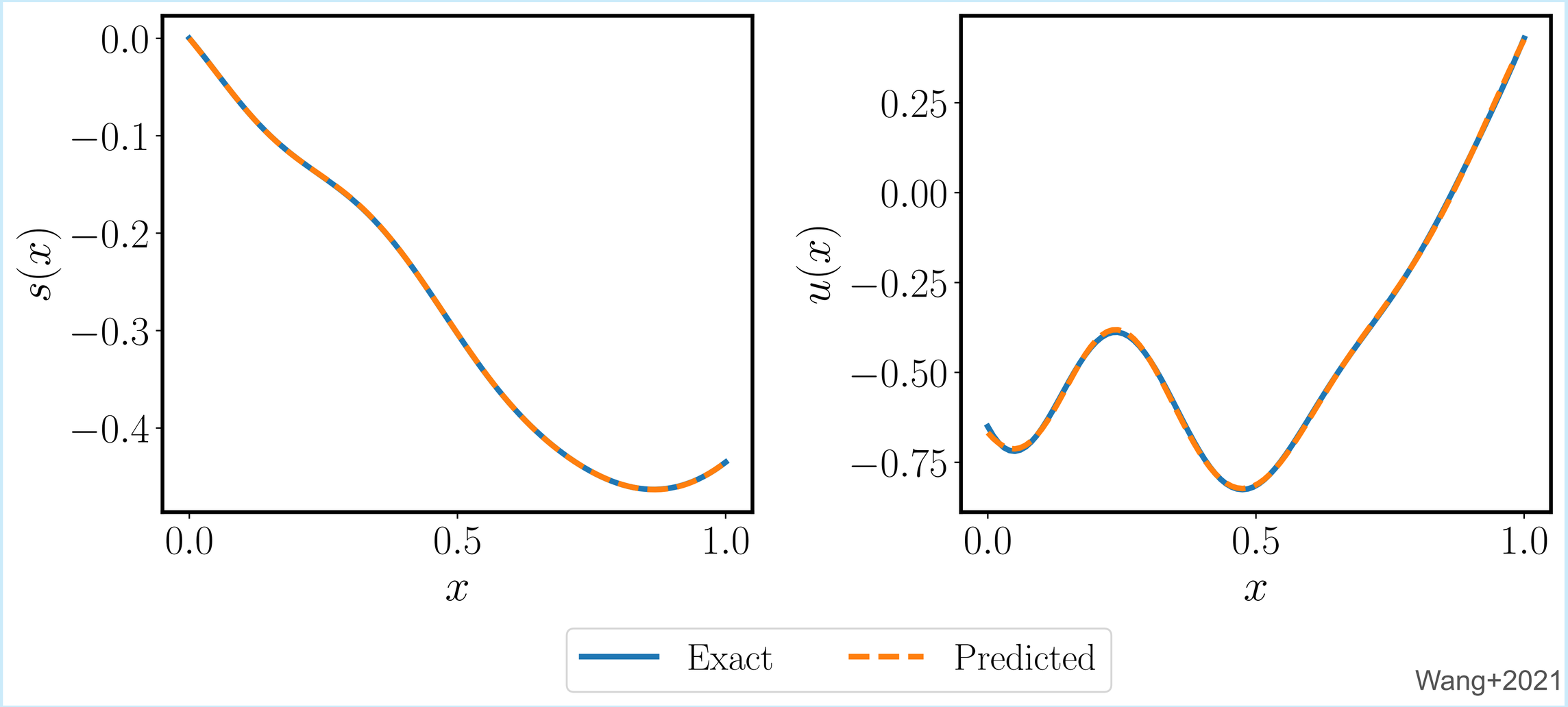
$$G : u(x) \longrightarrow s(x) = s(0) + \int_0^x u(t)dt, \quad x \in [0, 1].$$

EXAMPLE: ANTI-DERIVATIVE OPERATOR STANDARD DEEPONET



Wang+2021

EXAMPLE: ANTI-DERIVATIVE OPERATOR PINO



Wang+2021

EXAMPLE: DIFFUSION REACTION SYSTEM

$$\frac{\partial s}{\partial t} = D\frac{\partial^2 s}{\partial x^2} + ks^2 + u(x), \quad (x,t) \in (0,1] \times (0,1],$$

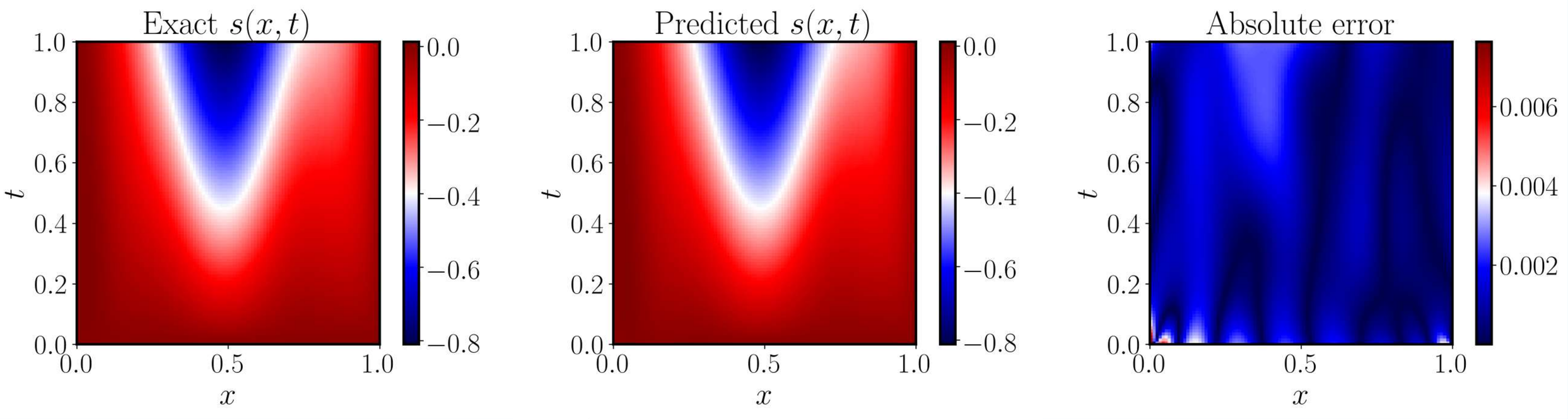
With the following PDE residual

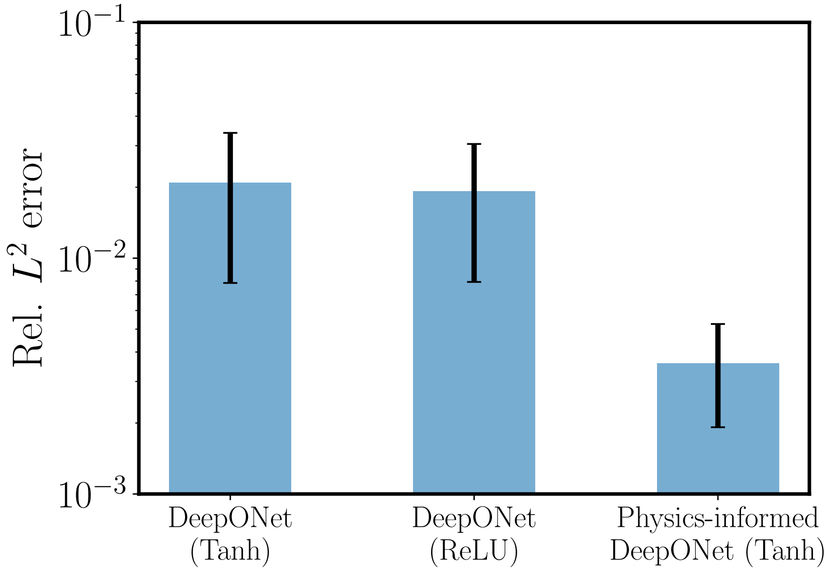
$$R_{\boldsymbol{\theta}}^{(i)}(x,t) = \frac{dG_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dt} - D\frac{d^2 G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)}{dx^2} - k[G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x,t)]^2,$$

And total loss function

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= \mathcal{L}_{\text{operator}}(\boldsymbol{\theta}) + \mathcal{L}_{\text{physics}}(\boldsymbol{\theta}) \\ &= \frac{1}{NP}\sum_{i=1}^{N}\sum_{j=1}^{P}\left|G_{\boldsymbol{\theta}}(\boldsymbol{u}^{(i)})(x_{u,j}^{(i)}, t_{u,j}^{(i)})\right|^2 + \frac{1}{NQ}\sum_{i=1}^{N}\sum_{j=1}^{Q}\left|R_{\boldsymbol{\theta}}^{(i)}(x_{r,j}^{(i)}, t_{r,j}^{(i)}) - u^{(i)}(x_{r,j}^{(i)})\right|^2.\end{aligned}$$

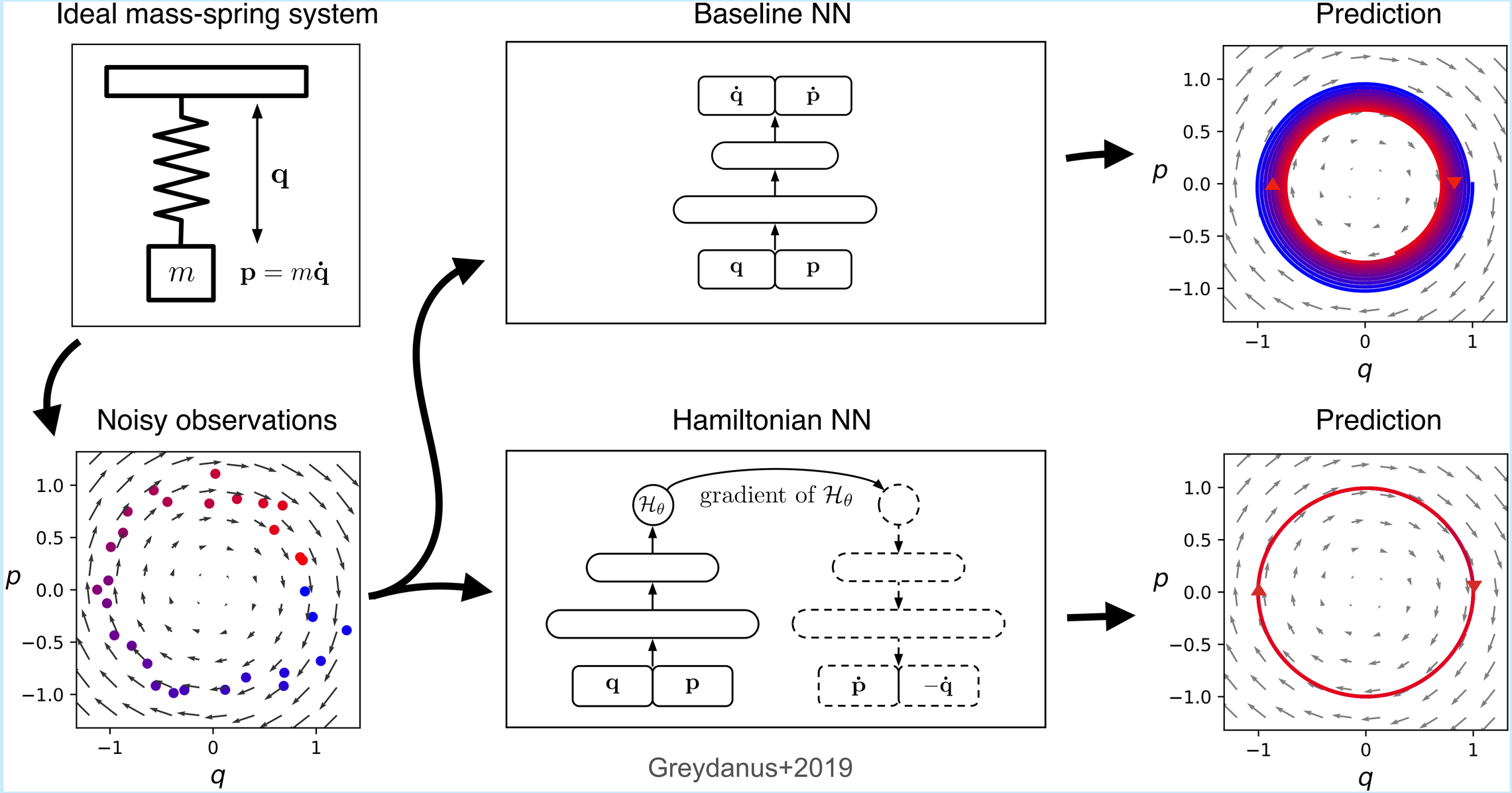
EXAMPLE: DIFFUSION REACTION SYSTEM





Wang+2021

HAMILTONIAN NEURAL NETWORKS



SUMMARY & CONCLUSION

Main take away:

scientific motivated inductive bias helps to be more robust, more data efficient and better interpretable

My personal message:

Write better code!
Share more data!
Build more open-source software!

This will accelerate research cycles and lets you engage with peers early on!

WHERE TO FIND THE LECTURE MATERIAL

https://github.com/TobiBu/graddays


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS


Differentiable Simulators

DIFFERENTIABLE SIMULATORS