# Neural Ordinary Differential Equations and universal systems

**Argimiro Arratia**
argimiro@cs.upc.edu
http://www.cs.upc.edu/~argimiro/

CS, Univ. Politécnica de Catalunya (Barcelona Tech)
EcoDep Seminary, CY University, Paris
October 19, 2022

# Neural Networks (feed-forward)

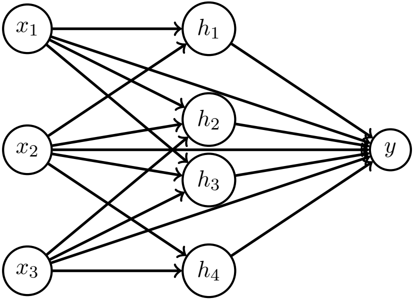



Figure: A 3-4-1 feed forward neural network with one hidden layer

$x_1$, $x_2$, $x_3$ input nodes; $y$ output node;
$h_1, \ldots, h_4$ hidden nodes (neurons) in hidden layer;
$h_j$ goes active and transmit a signal to $y$ if $z_j = \sum_{i \to j} \omega_{ij} x_i > \alpha_j$.
The signal is produced by activation function

**Single hidden layer case**

We have $d$ inputs $\boldsymbol{x} = (x_1, \ldots, x_d)$, one (or many outputs), and one hidden layer with $H_1$ units. Set $H_0 = d$. For a single output:

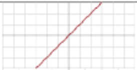
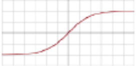
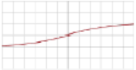
$$\boldsymbol{F}(\boldsymbol{x}) = \psi\left(\sum_{i=1}^{H_1} w_i^2 \varphi\left(\sum_{j=1}^{H_0} w_{ij}^1 x_j + b_i^1\right) + b^2\right) \tag{1}$$

Put

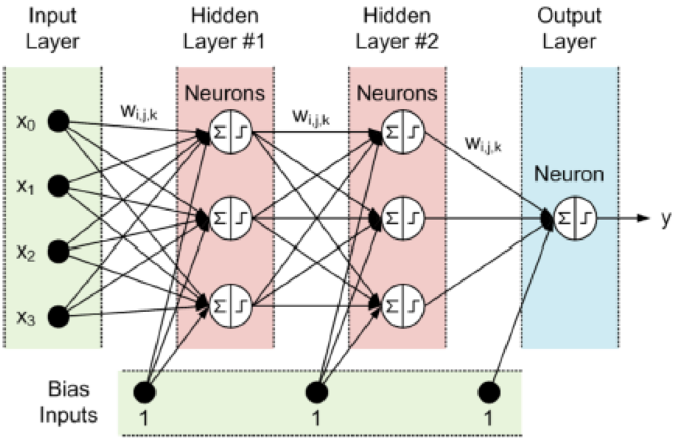
$$h_i^1 = \sum_{j=1}^{H_0} w_{ij}^1 x_j + b_i^1, \quad i = 1, 2, \ldots, H_1 \tag{2}$$

$\varphi(\cdot)$ and $\psi(\cdot)$ are nonlinear activation function. (e.g. $ReLU(x) = \max(0, x)$ )

# Nnet: Activation functions

| Name | Visualization | $f(x) =$ | Notes |
|---|---|---|---|
| Linear (= Identity) |  | $x$ | Not useful for hidden layers |
| Heaviside Step |  | $\begin{cases} 0 & \text{if} \quad x < 0 \\ 1 & \text{if} \quad x \geq 0 \end{cases}$ | Not differentiable |
| Rectified Linear (ReLU) |  | $\begin{cases} 0 & \text{if} \quad x < 0 \\ x & \text{if} \quad x \geq 0 \end{cases}$ | Surprisingly useful in practice |
| Tanh |  | $\frac{2}{1+e^{-2x}} - 1$ | A soft step function; ranges from -1 to 1 |
| Logistic ('sigmoid') |  | $\frac{1}{1+e^{-x}}$ | Another soft step function; ranges from 0 to 1 |

Deep Neural Networks (aka. multilayer neural networks)



In vectorial notation (2) is expressed as

$$\mathbf{h}^{(1)}(\boldsymbol{x}) = W^{(1)}\boldsymbol{x} + \mathbf{b}^{(1)} \in \mathbb{R}^{H_1}$$

And the output of 1-layer Nnet (Eq. (1)) as

$$\mathbf{F}(\boldsymbol{x}) = \psi(W^{(2)}\varphi(\mathbf{h}^{(1)}(\boldsymbol{x})) + \mathbf{b}^{(2)})$$

where $\varphi(\mathbf{z}) = (\varphi(z_1), \ldots, \varphi(z_{H_1}))$ and $\psi$ are activation functions ($\psi$ could be $\varphi$ or identity or other) and $\mathbf{z} \in \mathbb{R}^{H_1}$

# Deep Neural Networks

If there are $D > 1$ layers, each labelled by $\mu = 1, \ldots, D$, and with $H_\mu$ neurons in each, the recursion can be written as

$$\begin{aligned} \mathbf{h}^{(0)}(\boldsymbol{x}) &= \boldsymbol{x} \\ \mathbf{h}^{(\mu)}(\boldsymbol{x}) &= W^{(\mu)}\varphi(\mathbf{h}^{(\mu-1)}(\boldsymbol{x})) + \mathbf{b}^{(\mu)}. \end{aligned}$$

And the final output is the vector

$$\mathbf{F}(\boldsymbol{x}) = \psi(\mathbf{h}^{(D)}(\mathbf{h}^{(D-1)}(\ldots \mathbf{h}^{(2)}(\mathbf{h}^{(1)}(\boldsymbol{x})))))$$

**Forward evaluation (training)**

consists of choosing weights and biases such that the output approaches the actual values associated to input

# Nnet Backward propagation

Let training data $\{(\boldsymbol{x}^{[i]}, \boldsymbol{y}^{[i]}) : i = 1, \ldots, N\}$ of $N$ inputs $\boldsymbol{x}^{[i]} \in \mathbb{R}^{H_0}$ and corresponding $N$ outputs $\boldsymbol{y}^{[i]} \in \mathbb{R}^{H_D}$.

The parameters (e.g. weights and biases) are chosen so that some error measure is minimized (e.g. mean square error $MSE$).

In general we have cost function $\mathcal{C}$ on parameters $\theta$ and measure of error

$$Cost(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{C}(\boldsymbol{y}^{[i]} - \boldsymbol{F}(\boldsymbol{x}^{[i]}))$$

(e.g. in the case of quadratic cost, the objective to be minimized is

$$Cost(\theta) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{2}||\boldsymbol{y}^{[i]} - \boldsymbol{F}(\boldsymbol{x}^{[i]})||_2^2$$

# Optimization through Gradient Descent

Expand the cost objective using Taylor series ($\theta \in \mathbb{R}^s$):

$$\begin{aligned} Cost(\theta + \Delta\theta) &\approx Cost(\theta) + \sum_{i=1}^{s} \frac{\partial Cost(\theta)}{\partial \theta_i} \Delta\theta_i \\ &= Cost(\theta) + \nabla Cost(\theta)^\top \Delta\theta \end{aligned}$$

where $\nabla Cost(\theta)$ is the gradient vector and need to choose $\Delta\theta$ so that $\nabla Cost(\theta)^\top \Delta\theta$ is most negative at each iteration. This is achieved by updating with small step size $\eta$:

$$\theta \to \theta - \eta \nabla Cost(\theta)$$

layer through layer (gradient descent)

# Summary: Neural Network paradigm

- Forward evaluation (training)

$$\mathbf{F}(\boldsymbol{x}) = \psi(\mathbf{h}^{(D)}(\mathbf{h}^{(D-1)}(\ldots \mathbf{h}^{(2)}(\mathbf{h}^{(1)}(\boldsymbol{x})))))$$

- Measure of quality of approximation (Cost function)

$$Cost(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{C}(\boldsymbol{y}^{[i]} - \boldsymbol{F}(\boldsymbol{x}^{[i]}))$$

- Backward propagation to improve approximation. By gradient descent update through layers

$$\theta \to \theta - \eta \nabla Cost(\theta)$$

**Remark:** The functions in $Cost$ are known and differentiable.

### The Representation Theorem (Hornik et al., Cybenko, 1989-91)

Feed-forward network with one hidden layer of large enough width and a "squashing" activation function can approximate any integrable function to any accuracy.[a]

[a] Hornik, Stinchcombe, White (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* 2, 359-366

**Remark** (Bruno Després) Let $f \in C^1(\mathbb{R})$

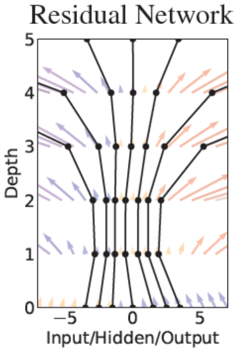
$$
\begin{aligned}
f(x) &= \int_{-\infty}^{x} f'(y)dy = \int_R H(x-y) f'(y)dy \\
&\approx \sum_{j=-J}^{J} \phi\left(\frac{x}{\epsilon} - \frac{j\Delta x}{\epsilon}\right) f'(j\Delta x)\Delta x = \sum_{j=-J}^{J} \omega_j \phi(a_j x + b_j)
\end{aligned}
$$

where $H(x)$ is Heaviside and $\phi$ a sigmoid to approximate $H$. Notice that $a, b$ tend to infinity with precision.

# From ResNet to NODE

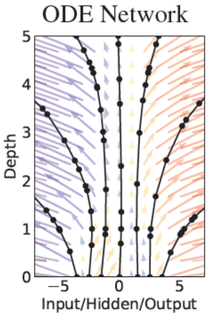
- Residual Network

$$h_{t+1} = h_t + f(h_t, \theta)$$




- Neural ODE[a]

$$\frac{h_{t+1} - h_t}{\Delta t} = \frac{f(h_t, \theta, t)}{\Delta t} \rightarrow \frac{dz}{dt} = f(z, \theta, t)$$




[a] Chen et al (2018) Neural ODE. In: Advances in Neural Information Processing Systems, 31

# Model as an IVP

- The model has become an Initial Value Problem.
  Let $z_0 := z(t_0) = \boldsymbol{x}$. Forward evaluation is

$$\boldsymbol{F}(z_0) = z(t_N) = z_0 + \int_{t_0}^{t_N} \frac{dz}{dt}\, dt = z_0 + \int_{t_0}^{t_N} f(z, \theta, t)\, dt$$

# Forward pass computes integration with ODE solver

For instance use Euler method to convert integral into many steps of addition

$$z(t+\epsilon) = z(t) + \epsilon \cdot f(z(t), \theta)$$

with $\epsilon < 1$.
Such ODE solvers are often numerically unstable (e.g. underflow error due to small step size, etc).
So, some other more sophisticated (black-box) ODE solvers are used.
**Remark.** $f(z(t), \theta)$, call it *the ODE function*, implicitly given from data, approximates $\dfrac{dz}{dt}$

# Optimization

- We can optimize: $\theta$, $t_0$, $t_N$ and $z_0$.
- Cost function

$$\text{Cost}\left(z(t_N)\right) = \text{Cost}\left(z(t_0) + \int_{t_0}^{t_N} f(z(t), \theta, t)\, dt\right)$$
$$= \text{Cost}\left(ODESolver(z(t_0), f, \theta, t_0, t_N)\right)$$

- L1, L2, ...
- We need to calculate the following gradients

$$\frac{dCost}{dz(t_0)}, \frac{dCost}{d\theta}, \frac{dCost}{dt_0}, \frac{dCost}{dt_N}$$

# Adjoint sensitivity method I

As en example $\nabla_\theta Cost$. We want to find

$$\min_\theta Cost(z(t_N)) \quad \text{s.t.} \quad \frac{dz}{dt} = f(z,\theta,t)$$

Construct Lagrangian

$$\mathcal{L} = Cost(z(t_N)) - \int_{t_0}^{t_N} \lambda(t) \left( \frac{dz}{dt} - f(z,\theta,t) \right) dt$$

integration by parts and chain rule differentiation gives

$$\frac{dCost(z_{t_N})}{d\theta} = \int_{t_N}^{t_0} -a(t) \frac{\partial f}{\partial \theta} dt$$

with $a(t)$ the adjoint state, which is solution of IVP

$$a(t_N) = \frac{dCost(z_{t_N})}{dt_N}, \quad \frac{da}{dt} = -a(t)\frac{\partial f}{\partial z}$$

Further algebraic manipulation yields gradient of cost w.r.to $\theta$ is solution at time $t_0$ of IVP

$$a_\theta(t_N) = 0, \quad \frac{da_\theta}{dt} = -a(t)\frac{\partial f}{\partial \theta}$$

# Adjoint sensitivity method II

Similar calculations yield that the gradients of Cost w.r.to $z_{t_0}$, $t_0$ and $\theta$, all result from evaluating IVPs on corresponding adjoint states at time $t_0$.
Define augmented state $s(t) := [a(t), a_\theta(t), a_t(t)]$ as concatenation of adjoints for $z$, $\theta$ and $t$

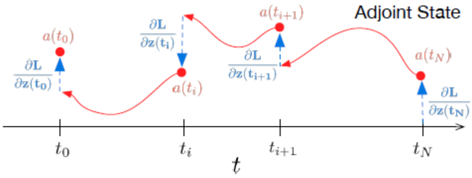
# Adjoint sensitivity method III

- Adjoint state at $t_0$

$$s(t_0) := \left[\frac{d\text{Cost}(z(t_N))}{dz(t_0)}, \frac{d\text{Cost}(z(t_N))}{d\theta}, -\frac{d\text{Cost}(z(t_N))}{dt_0}\right]$$

- Solving backwards Initial Value Problem

$$\begin{cases} s(t_N) = \left[\frac{d\text{Cost}(z_{t_N})}{dz_{t_N}}, \ \mathbf{0}, \ -a(t_N)f(z_{t_N}, \theta, t_N)\right] \\ \frac{ds(t)}{dt} = -a(t)\frac{\partial f}{\partial[z, \theta, t]} \end{cases}$$



# Neural ODE paradigm

## A Neural network with an ODE inside

- Forward evaluation: an Initial Value Problem

$$\boldsymbol{F}(z_0) = z(t_N) = z_0 + \int_{t_0}^{t_N} f(z, \theta, t)\, dt = ODESolver(z(t_0), f, \theta, t_0, t_N)$$

- Training (optimization): adjoint sensitivity method

**Remark.** The space complexity of Adjoint method is $O(1)$, whereas using backpropagation to train NODEs has space complexity proportional to number of ODEsolver steps. Their time complexities are similar.
Hence, we can train NODEs efficiently.

# Advantages and drawbacks

Advantages

- Memory savings
- Adaptive computation
- Speed and precision trade-off
- Continuous-time time series models

Drawbacks

- Can only learn homeomorphisms
- Deterministic dynamics
- Speed

# Traditional approach to Neural ODEs

- Traditional approach used by most authors employs Neural Networks to learn the ODE function $f(z, \theta, t)$

$$\frac{d\mathbf{y}}{dt} = NeuralNetwork(\mathbf{y}).$$

  - × Circling back to using NNs.
  - × turns the model inside-out: an ODE with a Neural Network inside!
  - ✓ Able to generate *universal* flows. And (in principle) has lots of potential in describing complex dynamical systems (more later).

# Our approach to Neural ODEs
# (Joint work with Carlos Ortiz, Marcel Romaní, 2022)

- Our proposed System of $n$ ODEs is given by

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} -x \\ \vdots \\ -x \end{bmatrix} + \mathbf{z}(x) \;\; \text{with} \;\; \mathbf{y}(0) = \begin{bmatrix} x \\ \vdots \\ x \end{bmatrix}$$

- It generates a (trivial) flow

$$\varphi(x,t) = (1-t)\begin{bmatrix} x \\ \vdots \\ x \end{bmatrix} + t\,\mathbf{z}(x),$$

where $\varphi(x,1) = \mathbf{z}(x)$ is the solution at $x$ of the IVP

$$\frac{d\mathbf{z}}{dt} = \mathbf{L}\left(\mathbf{z}, \theta\right) \;\; \text{with} \; \mathbf{z}(0) = \mathbf{z}_0$$

# Proposed families of SODEs

There is evidence that these families of SODEs are universal

- Lotka-Volterra systems

$$\frac{dz_i}{dt} = \lambda_i z_i + z_i \sum_{j=1}^{n} A_{ij} z_j, \quad \lambda_i, A_{ij} \in \mathbb{R},\ 1 \leq i, j \leq n$$

- Riccati systems

$$\frac{dz_i}{dt} = A_i + \sum_{j=1}^{n} B_{ij} z_j + \sum_{j,k=1}^{n} C_{ijk} z_j z_k, \quad A_i, B_{ij}, C_{ijk} \in \mathbb{R}, 1 \leq i, j, k \leq n$$

- S-systems

$$\frac{dz_i}{dt} = \alpha_i \prod_{j=1}^{n} z_j^{g_{ij}} - \beta_i \prod_{j=1}^{n} z_j^{h_{ij}}, \quad g_{ij}, h_{ij} \in \mathbb{R}, \alpha_i, \beta_i \in \mathbb{R}^+, 1 \leq i, j \leq n$$

## Setup of the experiments

Goal: approximating $g : \mathbb{R} \to \mathbb{R}$

- SODEs: Lotka-Volterra, Riccati, S-systems
- $n = 2, 5, 10$
- Domain: $[0, 3] \in \mathbb{R}$
- Functions: $Constant$, $x$, $x^2$, $\sin(3x)$, $\exp(x/2)$, $3\log(x+1)$, $3/(x+1)$

# Results I

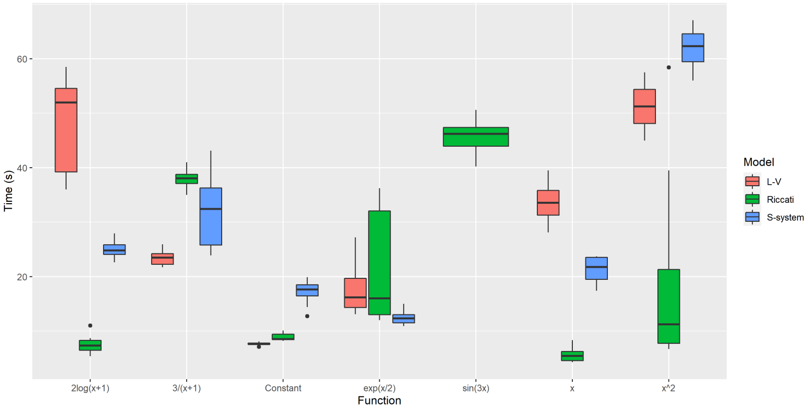



Figure: Comparison of the computation time to approximate different functions until $\varepsilon_r < 0.01$ grouped by model, $n = 2$.

# Results II

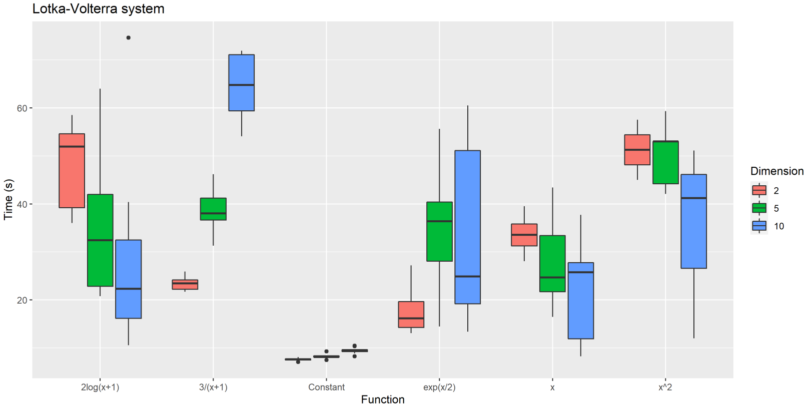



Figure: Computation time to approximate functions until $\varepsilon_r < 0.01$ using a Lotka-Volterra system with $n = 2, 5$ and $10$.

# Results III

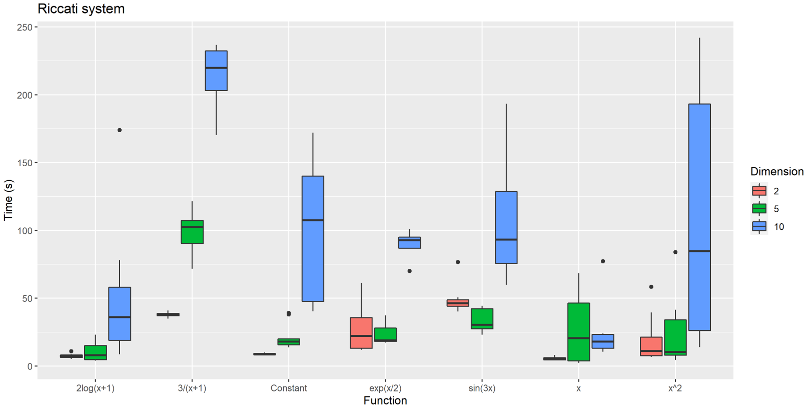



Figure: Computation time to approximate functions until $\varepsilon_r < 0.01$ using a Riccati system with $n = 2, 5$ and $10$.

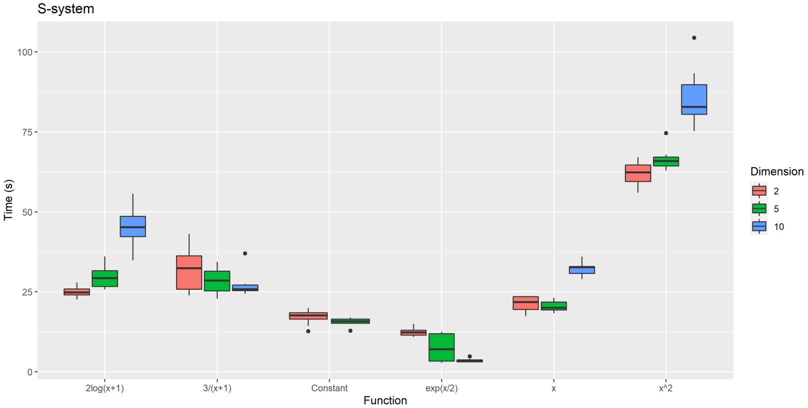
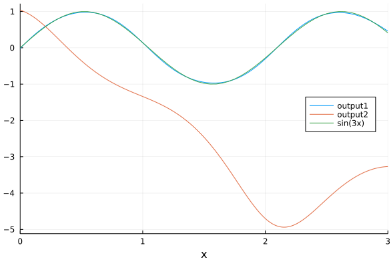
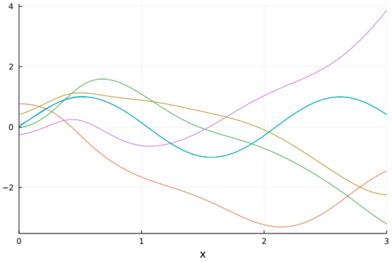



Figure: Computation time to approximate functions until $\varepsilon_r < 0.01$ using an S-system with $n = 2, 5$ and $10$.

# Function plots I



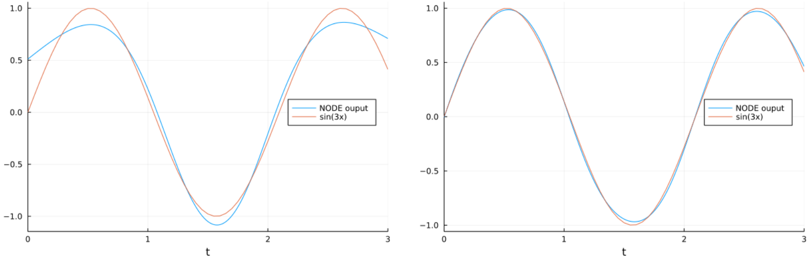

(a) $n = 2$




(b) $n = 5$

Figure: Full output of Neural ODEs approximating $\sin(3x)$.

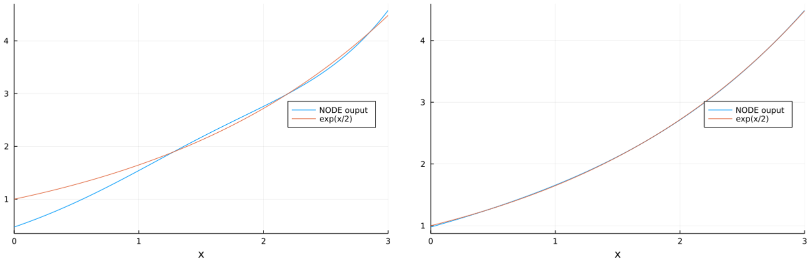
# Function plots II




(a) $\varepsilon_r = 0.01$ (b) $\varepsilon_r = 0.0005$

Figure: Approximation of the function $f(x) = \sin 3x$.

# Function plots III

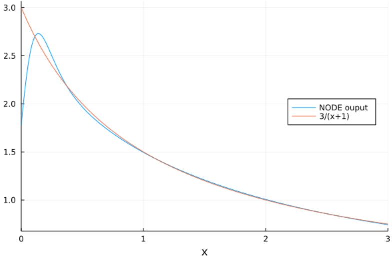



(a) $\varepsilon_r = 0.01$

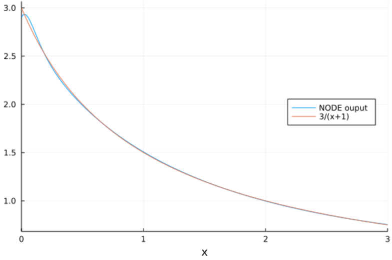
(b) $\varepsilon_r = 0.00001$

Figure: Approximation of the function $f(x) = \exp x/2$.

# Function plots IV




(a) $\varepsilon_r = 0.01$




(b) $\varepsilon_r = 0.0001$

Figure: Approximation of the function $f(x) = 3/(x+1)$.

# Conclusions

- Approximating capabilities of the families of SODE
- Input is very restricted in our framework
- Stiffness of equations lead to instabilities
- Further research should aim at benchmark problems

# Use cases of NODE (possibly relevant to EcoDep)

(Disclaimer: all these employ the twisted model $\frac{d\mathbf{y}}{dt} = NNet(\mathbf{y})$.)

- A tutorial: Forecasting the weather with neural ODEs, by Sebastian Callh https://sebastiancallh.github.io/post/neural-ode-weather-forecast/
- Some research papers:
  - Hwang et al (2021). Climate Modeling with Neural Diffusion Equations - arXiv
  - Bonnaffe et al (2020) Neural ordinary differential equations for ecological and evolutionary time series analysis. *Methods in Ecology and Evolution*
  - Raj Dandekar, Chris Rackauckas and George Barbastathis (2020). A Machine Learning-Aided Global Diagnostic and Comparative Tool to Assess Effect of Quarantine Control in COVID-19 Spread. *Patterns, v1 (9)*

The work by Dandekar et al, is more in line of *augmented dynamical systems with Neural Networks*: they define a epidemic model SIR with extra compartment to account for Quarantine individuals. This Q compartment is modeled by a Neural network

# Tools

Julia: DiffEqFlux.jl, DifferentialEquations.jl , . . . ,
all available in repository SciML (SciML Open Source Scientific Machine Learning ) https://github.com/SciML
Other: SciMLConference 2022: https://scimlcon.org/2022/talks/

# Neural Ordinary Differential Equations and universal systems

**Argimiro Arratia**
argimiro@cs.upc.edu
http://www.cs.upc.edu/~argimiro/

CS, Univ. Politécnica de Catalunya (Barcelona Tech)
EcoDep Seminary, CY University, Paris
October 19, 2022