A mathematical introduction to Neural Networks and Neural Ordinary Differential Equations

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

CS, Universitat Politécnica de Catalunya

Recap: Neural Ordinary Differential Equations (NODE)

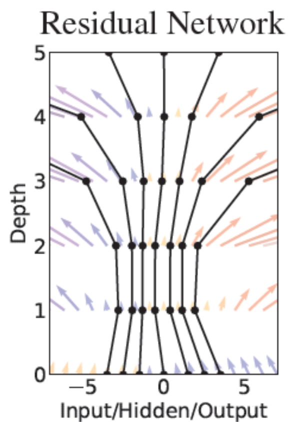
From ResNet to NODE

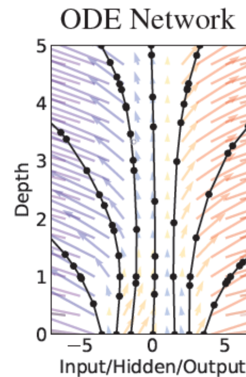
- Residual Network

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

- Neural ODE[a]

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





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

Basic ideas behind the NODE data learning approach I

1. We have a set of N data points

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

Given a new data point x^* want to make a prediction of it's value y^*, i.e. we seek a functional approximation to the relationship

$$x \to y$$

2. instead of modelling this relationship directly, we model the derivative:

$$\frac{dy}{dx} = f(x, y)$$

f is unknown
(What do we gain with this? reduction of parameters at least)

Basic ideas behind the NODE data learning approach II

3. We parametrise this approximation by a neural network with hidden states $h(t)$, depending continuously on layer depth t, with $h(t_0) = x$ and $h(T) = y$. The function approximation problem is now

$$\frac{dh(t)}{dt} = f(t, h(t), \theta)$$

(We arrived here by analogy with ResNet)

4. This is an ODE describing continuous hidden state dynamics. We can solve the data modelling problem by solving this ODE. Given an input x to the NODE, the value of hidden state at time (or depth) t is obtained by solving the integral

$$\varphi_t(x) := h(t) = x + \int_{t_0}^{t} f(s, h(s), \theta) ds \qquad \text{with} \quad h(t_0) = x$$

φ_t is called the $flow$ of the ODE at time t. If we solve the integral up to time $t = T$ we get the map $\varphi_T : x \to h(T) = y$.

Basic ideas behind the NODE data learning approach III

5. IMPORTANT Fact: According to Picard's Theorem a unique solution for the above IVP will exist if $f(s, h(s), \theta)$ is $Lipschitz$ continuous.
(Argue that this is the case for most Nnet. So it is safe to take f as Nnet.)

6. The analytical solution of this integral is not available to us. Instead we can use a numerical method (e.g. Euler method) to solve the integral at the required evaluation points:

$$\hat{y} = h(t_1) = ODEsolve(h(t_0), t_0, t_1, \theta, f)$$

Basic ideas behind the NODE data learning approach IV

7. The free parameters of this problem are t_0, t_1, θ. We optimise our choice of these free parameters w.r.to some loss function $\mathcal{L}$ by backpropagating (reverse-mode differentiation) through the ODE solver using the method of adjoints:

$$\mathcal{L}(t_0, t_1, \theta) = \mathcal{L}(ODEsolve(h(t_0), t_0, t_1, \theta, f))$$

To optimise the loss, we require gradients with respect to the free parameters.
(follow details of adjoint method in my previous slides or in M. Surtsukov github)

Further resources for NODE

- Mikhail Surtsukov github with brief intro and code in Pytorch. https://github.com/msurtsukov/neural-ode/blob/master/Neural%20ODEs.ipynb
- Emilien Dupont github on Augmented NODE: https://github.com/EmilienDupont/augmented-neural-odes
- Adria Lisa Bou, Introduction to neural ordinary differential equations TFG https://upcommons.upc.edu/handle/2117/387774
-

Neural Ordinary Differential Equations and universal systems

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

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

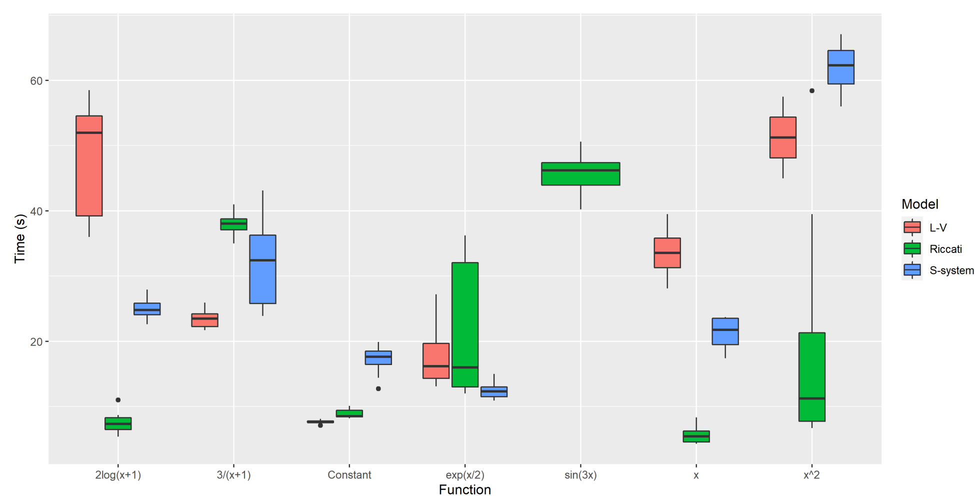


Figure: Comparison of the computation time to approximate different functions until $\varepsilon_r < 0.01$ grouped by model, $n = 2$.

Results II

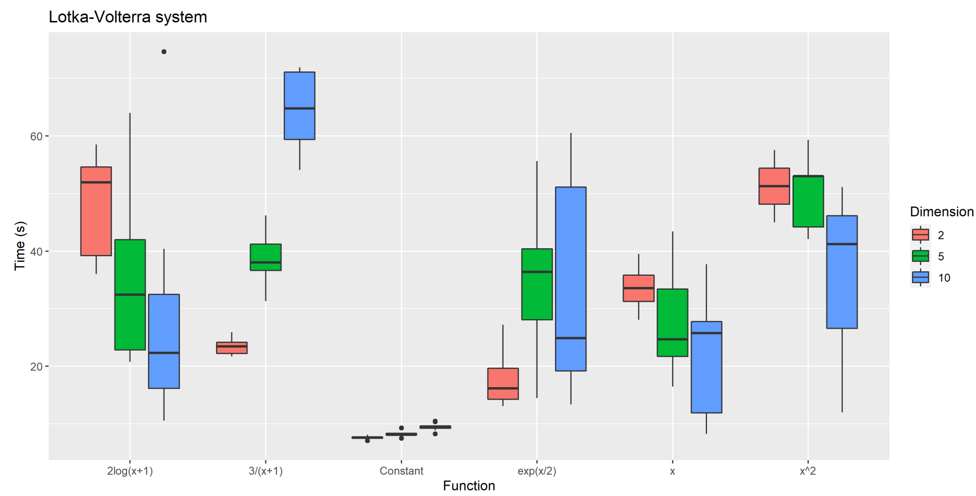


Figure: Computation time to approximate functions until $\varepsilon_r < 0.01$ using a Lotka-Volterra system with $n = 2, 5$ and 10.

Results III

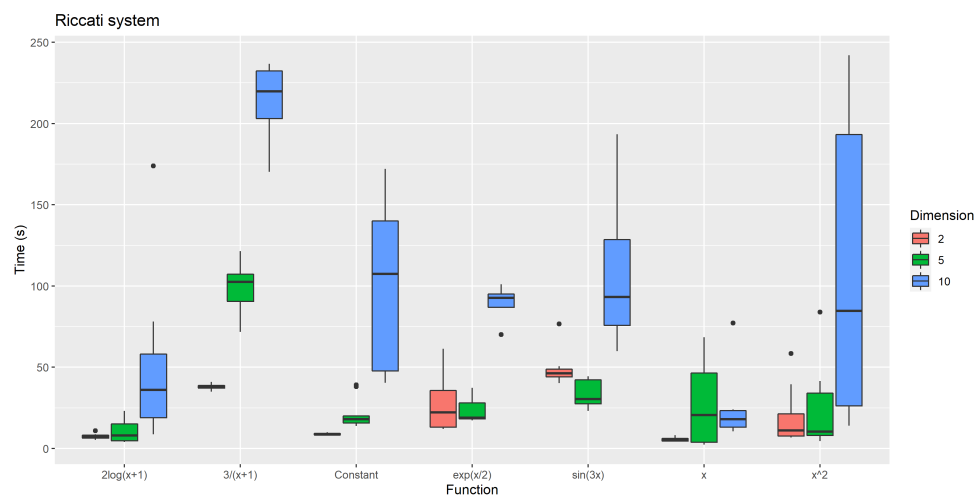


Figure: Computation time to approximate functions until $\varepsilon_r < 0.01$ using a Riccati system with $n = 2, 5$ and 10.

Results IV

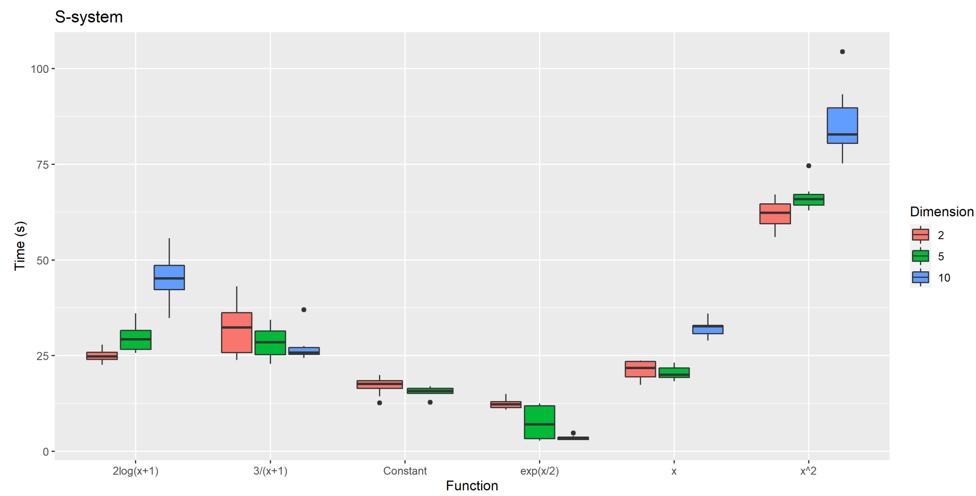
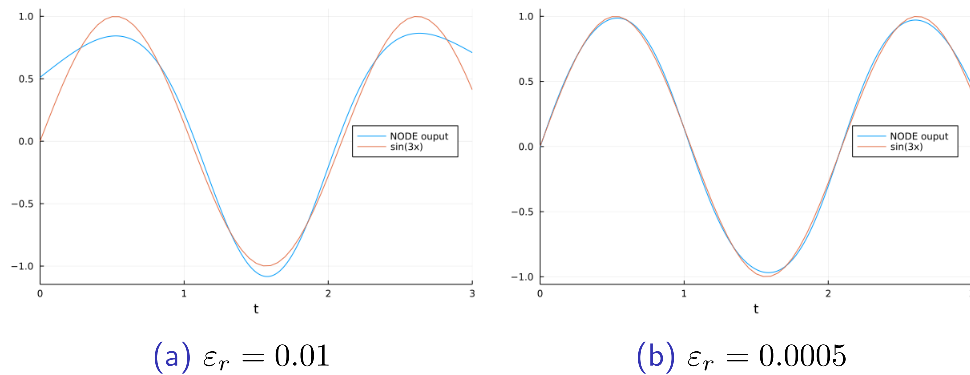


Figure: Computation time to approximate functions until $\varepsilon_r < 0.01$ using an S-system with $n = 2, 5$ and 10.

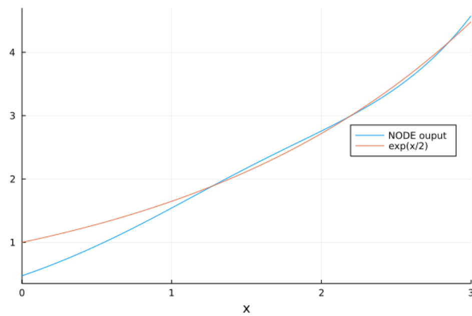
Function plots II



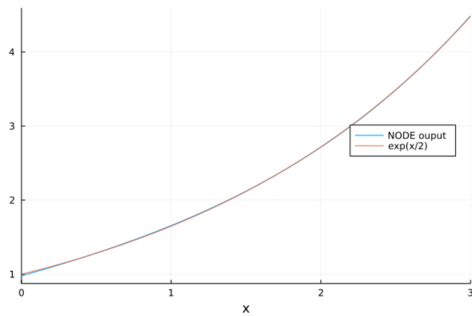
(a) $\varepsilon_r = 0.01$ (b) $\varepsilon_r = 0.0005$

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

Function plots III



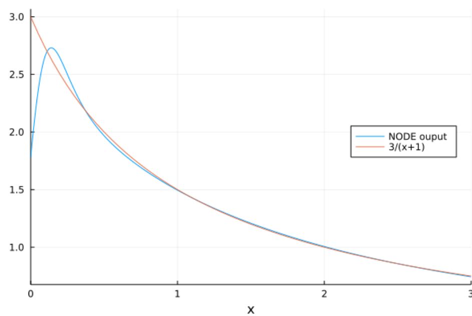
(a) $\varepsilon_r = 0.01$



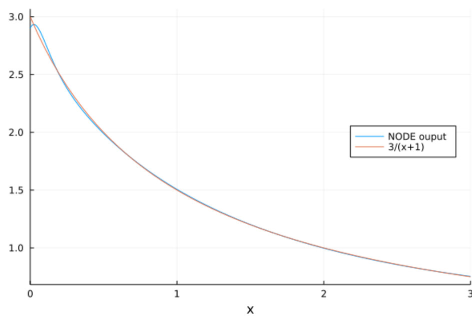
(b) $\varepsilon_r = 0.00001$

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

Function plots IV



(a) $\varepsilon_r = 0.01$



(b) $\varepsilon_r = 0.0001$

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

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

Applications: Weather Forecast, ...

Use cases of NODE

(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 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 a NNet

Tools

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

A brief Intro to Pytorch

Pytorch

PyTorch was developed by Meta AI (Facebook) 2016.
My recommendation of programming environment: Google Colab, and as Jupyter notebooks, for experimentation and research.
Official Guide: https://pytorch.org/tutorials/beginner/basics/intro.html

Pytorch basic constructs

The basic data units are tensors. (has not much to do with tensors in mathematics just that it is a kind of object from linear algebra)
A torch.Tensor is a multi-dimensional matrix containing elements of a single data type (similar to Numpy's ndarray) that can run on GPUs or other hardware accelerators. In fact, tensors and NumPy arrays can often share the same underlying memory, eliminating the need to copy data. Tensors are also optimized for automatic differentiation.
Torch defines 10 tensor types with CPU and GPU variants. see
https://pytorch.org/docs/stable/tensors.html
The other high level construct of Pytorch are Deep neural networks built on a tape-based automatic differentiation system

PyTorch vs. Tensorflow (keras)

PyTorch has a low-level API that requires you to write more code and handle more details than Keras. PyTorch also has a weaker support for distributed and parallel computing, which can affect your scalability and efficiency

Building a deep learning model in PyTorch I

Basic steps

- Import the necessary libraries: **torch** for creating and working with neural networks; **torch.nn** for defining network components.
- Design the Model Class: Define a Python class that inherits from torch.nn.Module. This class will represent the neural network architecture. The class contains layers (like building blocks) that process input data to produce output predictions.
- Initialize the Model: In the class's constructor (`_init_ method`), define the layers and other components of the neural network. These components can include linear layers (fully connected layers), convolutional layers, activation functions, and more.
- Define the Forward Pass: Inside the model class, create a `forward method`. This method describes how data flows through the layers of the network. Define the sequence of operations (like linear transformations, activation functions) that turn input data into output predictions.

Building a deep learning model in PyTorch II

Basic steps

- Instantiate the Model: Create an instance of the model class. This instance will be the neural network.
- Choose a Loss Function: Select a loss function that measures the difference between the model's predictions and the actual target values.
- Choose an Optimizer: Choose an optimization algorithm that adjusts the model's parameters (weights and biases) to minimize the loss function. Popular optimizers include Adam, SGD, and RMSProp.
- Training Loop: Iterate over your dataset in batches. For each batch: Pass the batch through the model to get predictions. Calculate the loss by comparing predictions with actual targets using the chosen loss function. Backpropagate the loss to compute gradients (slopes) of model parameters with respect to the loss. Use the optimizer to update model parameters, nudging them in a direction that reduces the loss.

Building a deep learning model in PyTorch III

Basic steps

- Repeat Training: Repeat the training loop for a specified number of epochs (complete passes through the dataset) or until the model's performance improves to an acceptable level.
- Validation and Testing: After training, evaluate the model's performance on a separate validation dataset to ensure it's not overfitting. You can also test the model on completely new data to assess its real-world performance.
- Inference: Once trained, you can use the trained model to make predictions on new, unseen data.

Neural Network class in Pytorch

A 1-hidden layer NNet with ReLU (or sigmoid) activation

```
class NeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(NeuralNetwork, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        #x = torch.sigmoid(self.layer1(x))
        x = torch.relu(self.layer1(x))
        x = self.layer2(x)
        return x
```