Project Paper: Neural ODEs

Emily Ribando-Gros

1. Introduction

First introduced in [1], neural ordinary differential equations (ODEs) offer a new deep neural network paradigm where the output of a neural network is computed through a differential equation solver. This method leverages well known tools from numerical analysis and differential equations to learn powerful continuous models. Neural ODEs have a much lower memory cost then their previous counterpart, Residual Networks or ResNets, due to continuous-depth.

Neural ODEs can be sampled at irregular evaluation intervals so to more closely model real-world physical systems. Finally, neural ODEs offer direct control over the accuracy to speed trade-off through ODE solvers. Within this paper we present neural ODEs along with some simple implementations of the supervised learning of dynamical systems.

2. Background

2.1. Related Work

We start with some sequenced data to model $z(t)$ parameterized by t, often thought of and referred to here as time and $z(t)$ is known to follow some process

$$\frac{dz}{dt} = f(z(t), t) \tag{1}$$

otherwise known as an ordinary differential equation. In order to solve such an ODE, an initial condition $z(t_0)$ and final t_1 must be given. Then the solution to the ODE is given through integration,

$$z(t) = \int_{t_0}^{t_1} f(z(t), t)dt.$$

Such processes naturally appear in biology, chemistry, and physics with many numerical methods developed to solve them.

Previously, such models were learned via residual units where,

$$z(t_n) = t_{n-1} + f(z(t_{n-1}), t_{n-1}).$$

Chaining together these residual units resulted in the popular ResNet architecture [2]. However, to achieve a desirable accuracy, deeper networks need to be trained requiring a large number of parameters and thus high memory costs. Furthermore, ResNets are only able to give solutions at evaluation time steps resulting is piecewise linear approximations of continuous representations as seen in Figure 1. As shown by Figure 2, ResNets also offer poor extrapolation outside of sampled or training data.

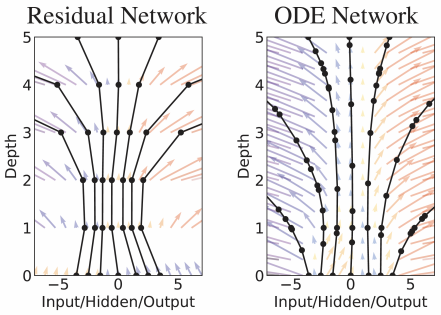
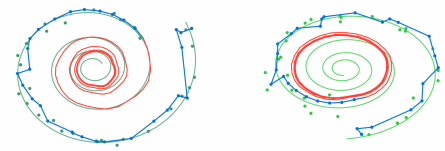


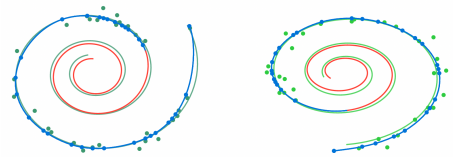
Figure 1. Comparison of ResNet to Neural ODEs for vector field model. [1]

2.2. Neural ODEs

As an answer to the challenges involved in ResNets, neural ODEs were developed adding parameters θ to the usual dynamics from equation 1



(a) Recurrent Neural Network



(b) Latent Neural Ordinary Differential Equation

Figure 2. Comparison of ResNet to Neural ODEs for a dynamical system. [1]

and letting f be our neural network. The architecture of f can take many different forms, although the activation functions used must be differentiable. Then the problem statement can be reformulated as the following optimization:

Find $\arg\min_\theta L\left(z\left(t_1\right)\right)$ subject to

$$F(z(t'), z(t), \theta, t) = z'(t) - f(z(t), \theta, t) = 0,$$
$$z\left(t_0\right) = z_{t_0},$$
$$t_0 < t_1.$$

where L is the loss function.

2.3. Adjoint Sensitivity Method

Yet, one technical issue remains, how to efficiently calculate gradients for backpropagation or reverse-mode differentiation through ODE solvers. The authors of [1] adopted the adjoint sensitivity method from optimization theory which calculates an augmented adjoint state backwards through time. The adjoint state is $a(t) = \frac{\partial L}{\partial z(t)}$ with dynamics given by another ODE,

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z(t)}$$

with initial value $\frac{\partial L}{\partial z(t_1)}$.

If the time interval is subdivided (not necessarily equally subdivided), then a is updated backwards in time in the direction of the partial derivative of the loss with respect to each observation, see Figure 3.

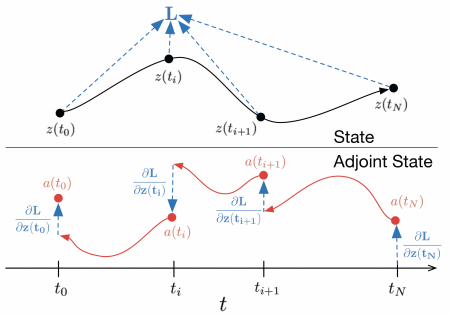


Figure 3. Representation of the loss update and adjoint state update.

Then the steps of the neural ode model are as follows,

- Step 1: Forward Process – Solve ODE $z'(t) - f(z(t), \theta, t) = 0$
- Step 2: Calculate Loss $L(z)$
- Step 3: Backward(by reverse time mode ODE solvers)

$$\begin{aligned}
\frac{\partial L}{\partial z\left(t_0\right)} &= \int_{t_1}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial z} dt \\
\frac{\partial L}{\partial \theta} &= \int_{t_1}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt \\
\frac{\partial L}{\partial t_0} &= \int_{t_1}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial t} dt \\
\frac{\partial L}{\partial t_1} &= -a(t) \frac{\partial f(z(t), t, \theta)}{\partial t}
\end{aligned}$$

  where $a(t) = -\frac{\partial L}{\partial z(t)}$

- Step 4: Update parameters θ

2.4. Dynamics Application and Implementation Details

Dynamical systems, functions which describe positions which depend on time, are naturally described with ordinary differential equations. In the simple

2D case, positions or states $z(t) \in \mathbb{R}^2$ are described by dynamics

$$\frac{dz}{dt} = f(z(t), t, \theta).$$

Parameters θ are defined by the network layers, with two architectures explored,
(1) Input $\rightarrow$ Linear $\rightarrow$ tanh $\rightarrow$ Linear $\rightarrow$ Output
(2) Input $\rightarrow$ Linear $\rightarrow$ Linear $\rightarrow$ Output.

For our MATLAB implementation, weights are initialized using a normalized Xavier initialization scheme with bias terms initialized to 0 vectors. Mini batching is implemented to aid in the calculation though not necessarily needed as stated in [1]. The loss computed between exact and predicted dynamics can be chosen as the L2 norm. MATLAB's default ode solver, ode45 was used, as one of the state of the art Runge-Kutta methods [3]. Results for one dynamics model is given in Figure 4.

References

[1] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018.

[2] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[3] Lawrence F. Shampine and Mark W. Reichelt. "The MATLAB ODE Suite". In: *SIAM Journal on Scientific Computing* 18.1 (1997), pp. 1–22. DOI: 10.1137/S1064827594276424. eprint: https://doi.org/10.1137/S1064827594276424. URL: https://doi.org/10.1137/S1064827594276424.

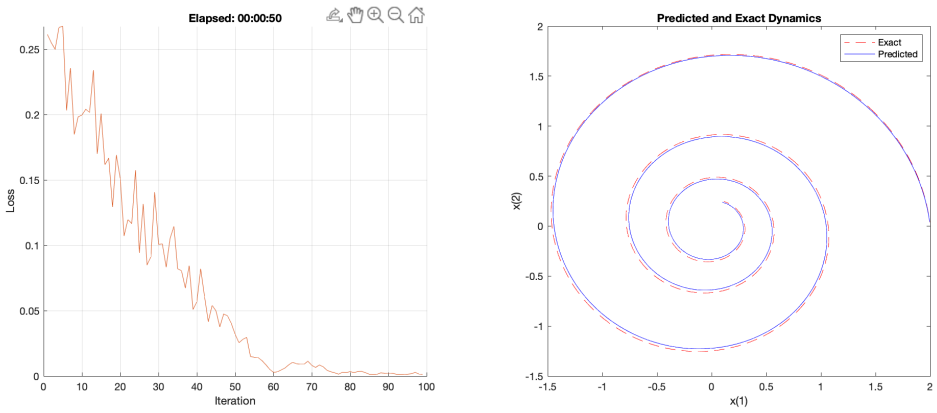


Figure 4. Loss (left) and phase space (right) of a learned dynamical system $z' = Az$, for $A = \begin{bmatrix} -0.1 & -1 \\ 1 & -0.1 \end{bmatrix}$ and $z(t_0) = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$.