Neural ODE and Backpropagating Through RK4 Solver

Tyrone DeSilva*
Department of Applied Mathematics
University of Washington
tdslv@uw.edu

Abstract

Neural Ordinary Differential Equations(NODE) are a class of neural network architectures that are continuous in depth. Motivated to understand how NODE works and how neural networks can be applied to learning to interpolate time-series data, we first implement backpropagation through the RK4 solver itself to learn the dynamics of a Lotka-Volterra(LKV) model. Next we train a NODE using the adjoint sensitivity method to backpropagate through the ODE solver. We also seek to summarize and solidify our understanding of the backpropagation algorithm for the adjoint method.

1 Introduction

1.1 Background

Neural Ordinary Differential Equations(NODE) can be thought of as a type of continuous depth neural network. By using a neural network to define a differential equation, we can train the parameters and learn the dynamics over the 'depth' direction. Let $h(t)$ be the hidden state of the NODE, t represents the depth, and θ the parameters of the neural network.

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

Although f is a function of t, from here on out we'll usually assume the neural network f is time-invariant. In the original NODE paper [1], ResNet is referenced as an inspiration for developing NODE. Specifically ResNet resembles the Euler discretization of the above differential equation.

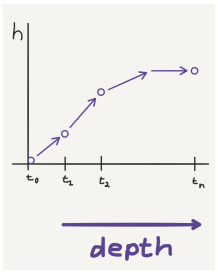
$$z(t + \Delta t) = z(t) + \Delta t f(z(t), t, \theta)$$

So, even before NODE there is precedent of training a neural network whose task is to learn the differences at each time-step, or each layer. Of course, ResNet has discrete layers. NODE is continuous; we can compute the hidden state for any t between the initial t_0 and t_1 of our NODE layer.

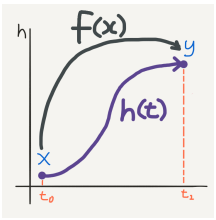
It may be useful to visualize the two different approaches. Figure 1a would correspond to either solving the NODE with a Euler discretization with fixed time steps or using ResNet. Figure 1b corresponds to the continuous depth approach. In reality, there are discrete time steps that the solver calculates in the forward pass to get the numerical solution in Figure 1b. The time steps are not fixed, and the hidden state can be evaluated for any depth, not just $t_1, t_2, \ldots t_N$.

*Webpage, GitHub

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.



(a) Euler Discretization of the hidden state h



(b) Continuous in Depth Hidden State

Figure 1: Comparison between discrete and continuous depth [2]

1.2 Advantages and Uses

NODE does add some complication to training the neural network, since we now need to figure out how to solve the forward and backward pass of the ODE. However there are several advantages to using NODE over traditional discrete layers.

- Memory Efficiency
- Adaptive Computation
- Continuous Time Series Models

Memory efficiency is due to the NODE not needing to save intermediate states to calculate the gradient. Adaptive computation relies on being able to use an adaptive ODE solver, which might result in taking fewer iterations while keeping the error low. Finally, NODE handles continuous time series well because we can use the continuous depth to represent continuous time data. For instance, instead of using an RNN where each discrete RNN block represents a point in time, we can use the depth parameter t to also represent time. The network could be trained given an initial value at t_0 and predict values at $t_1, t_2, \ldots t_N$. Of course the ODE solver can be evaluated at additional points to interpolate or extrapolate the time series.

1.3 Methods for Backpropagation

We've already mentioned that ResNet is similar to an Euler discretization. One 'ODE solver' we could use is the first-order Euler method. In this case we can backpropagate directly through the solver. However this has a fixed timestep and we would need to save the intermediate hidden state. In this paper, we will go over backpropagation through the RK4 solver.

The alternative proposed in [1] is the adjoint sensitivity method. This method allows us to treat the ODE solver as a black box, meaning we can use an adaptive solver. We also do not need to save the intermediate states unless needed to calculate the loss.

2 Data Generation

The main task we will apply NODE to is interpolating the Lotka-Volterra(LKV) predator prey model. LKV is a coupled system of two differential equations. x_1 represents prey population, x_2 represents the predator population.

$$\frac{dx_1}{dt} = \alpha x_1 - \beta x_1 x_2,$$
$$\frac{dx_2}{dt} = \delta x_1 x_2 - \gamma x_2$$

The four parameters α, β, γ, and δ determine how quickly the prey grows, the rate predation decreases the prey population, the rate predation grows the predator population, and the rate of dying off in the predator population. The generated data was for an LKV model with initial conditions

$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.8 \\ 0.4 \end{pmatrix}$ and parameters $\alpha = 1.1$, $\beta = 0.4$, $\delta = 0.1$, $\gamma = 0.4$. The data were sampled at 500 points uniformly randomly selected in the time interval $[0, 100]$.

3 Backpropagating through RK4 Solver

To help get an understanding of the improvement that the adjoint sensitivity has over existing methods, we first tried implementing backpropagation through the classic RK4 solver. This also helps demonstrate how useful neural networks can be used function approximators for differential equations.

3.1 RK4 Method

The RK4 method is a classical method for numerically solving ODE problems that is a little more complex than the Euler method mentioned previously.

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 &= f(t_n, y_n, \theta) \\
k_2 &= f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}, \theta) \\
k_3 &= f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}, \theta) \\
k_4 &= f(t_n + h, y_n + hk_3, \theta) \\
\frac{dy_n}{d\theta} &= \frac{\partial y_n}{\partial \theta} + \frac{\partial y_n}{\partial y_{n-1}}\frac{dy_{n-1}}{d\theta} \\
\frac{d\mathcal{L}(y_1, y_2, ..., y_N)}{d\theta} &= \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial y_i}\frac{dy_i}{d\theta}
\end{aligned}
$$

By just applying the chain rule, and treating y_n as function of y_{n-1} and θ, we can calculate the gradient of the loss $\mathcal{L}$ with respect to θ.

3.2 Training

To train our neural network, we took each consecutive pair of points from the generated LKV training data, along with $T = t_2 - t_1$. The neural network is trained to predict $z(t_1)$ given $z(t_0)$. Since our ODE is time invariant, we actually just integrate forward from $t = 0$ to $t = T$. In order to make batch training easier, the number of iterations between 0 and T is a constant 100 steps. The actual step size changes for each sample, since T is random.

We trained one model using the actual LKV equations to try to rediscover the parameters $\alpha, \beta, \delta, \gamma$. The network is trained using Mean Square Error(MSE) loss for 800 epochs using gradient descent with 1E-4 learning rate. The estimated parameters were initialized with the distribution $\mathcal{U}(0, 1)$.

The other model we trained uses a fully connected neural network that maps from $\mathbb{R}^2$ to $\mathbb{R}^2$. The hidden layers are dimension 32, 64, and 16 with a ReLU activation. This network was trained using ADAM optimization with a learning rate of 1E-4.

3.3 Results

The LKV equation model effectively learned the original model parameters.

	α	β	δ	γ
Actual	1.1	0.4	0.1	0.4
Estimated	1.096	0.399	0.101	0.419

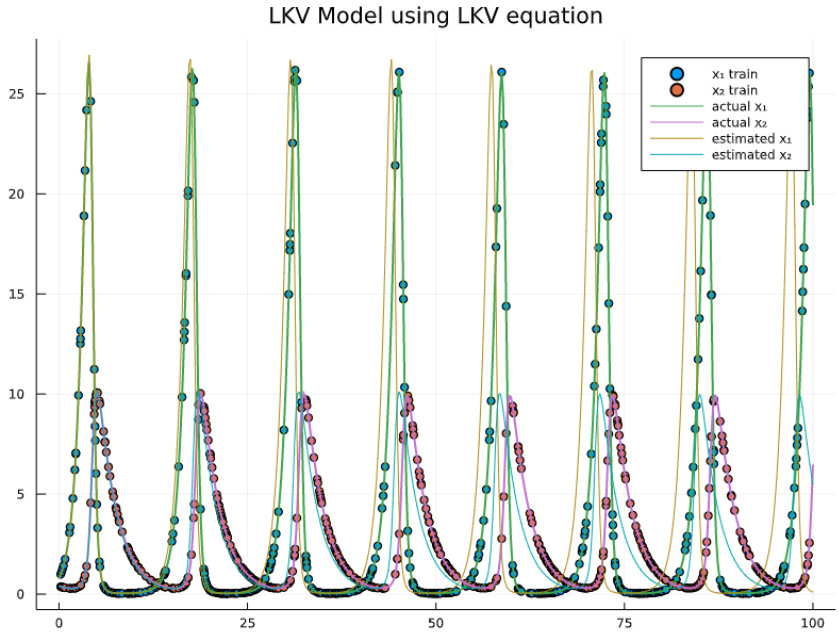


Figure 2: LKV Equation Model Fit. The plot displays both the true dynamics, sampled points, and the learned dynamics.

Plotting the true dynamics of the LKV system vs the learned dynamics in Figure 2, we see that they are pretty similar. Over time they diverge, since the estimated parameters are slightly different and the errors will accumulate.

For the neural network model, we don't have any interpretable parameters like we do using the LKV equation. However one advantage is we did not need to assume the data was generated by an LKV model. It's possible this model would be able to fit other dynamics as well without any prior knowledge about what generated our data. The fit this time is much closer in Figure 3, but there was some inconsistency between each training run. The LKV equation model was much more consistent overall. Both methods seem to be effective for interpolation.

4 Backpropagating Through Black Box Solver

In this section we will provide an overview of our attempt to fit the LKV data using the adjoint sensitivity method, as well as some background on the method.

4.1 Adjoint Sensitivity Method

The main idea with the adjoint sensitivity method is that we can backpropagate by solving an ODE backwards in time. Define the adjoint state as

$$a(t) = \frac{d\mathcal{L}}{dz(t)}$$

Using the instantaneous equivalent of the chain rule, it can be shown that the dynamics of $a(t)$ can be described the differential equation

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

It can also be shown that

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt$$

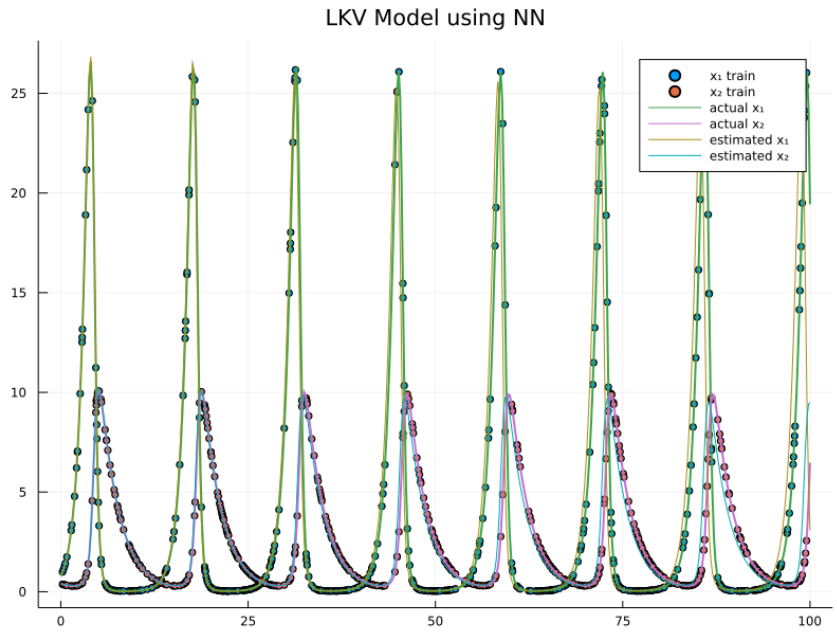


Figure 3: LKV Neural Network Fit. The plot displays both the true dynamics, sampled points, and the learned dynamics.

The proof of these statements is seen in Appendix B of [1]. In our case we treat the initial times t_0 and t_1 as fixed, but the adjoint method can calculate the gradient of the loss with respect to t_0 and t_1 as well.

The ODE we will solve is an augmented system for hidden state $z(t)$, the adjoint state, and the parameter adjoint state $a_\theta(t) = \frac{d\mathcal{L}}{d\theta(t)}$, where $\frac{d\theta(t)}{dt} = 0$, due to our time-invariant assumption. If we did want to solve the gradient of $\mathcal{L}$ with respect to t_0, t_1 we would add an additional term to our augmented state.

The initial conditions for the augmented ODE $\begin{pmatrix} z(t) \\ \frac{d\mathcal{L}}{dz(t)} \\ \frac{d\mathcal{L}}{d\theta(t)} \end{pmatrix}$ are $\begin{pmatrix} z(t_1) \\ \frac{d\mathcal{L}}{dz(t_1)} \\ \mathbf{0} \end{pmatrix}$. Solving backwards in time to t_0 will give us $\frac{d\mathcal{L}}{dz(t_0)}$ and $\frac{d\mathcal{L}}{d\theta(t_0)} = \frac{d\mathcal{L}}{d\theta}$.

4.2 Training

It was originally our goal to implement the adjoint backpropagation algorithm from scratch, but time was insufficient. Instead we utilized a Julia library, DiffEqFlux.jl [3], for adjoint backpropagation. The training method is different from previously. This time we sampled the same LKV data randomly, but this time on a shorter interval of just 5 seconds. We also sampled just 20 points in this time span. Each training epoch consisted of the entire sequence, whereas for the previous RK4 method we used just two points at a time. MSE loss was calculated across all the sample points. The actual network was simply a fully connected network with a single 64 dimension hidden layer using sigmoid activation. Training was done for 1,000 epochs using Adam optimizer with 0.1 learning rate. Our method for training this NODE was largely inspired by a tutorial blog post [4], which contains more details about how to apply this method to a different dataset.

4.3 Results

The reason we chose to train our data on a more limited dataset was because training on the same sized dataset as before resulted in a model that was severely under-fitting. In Figure 4 we only plotted

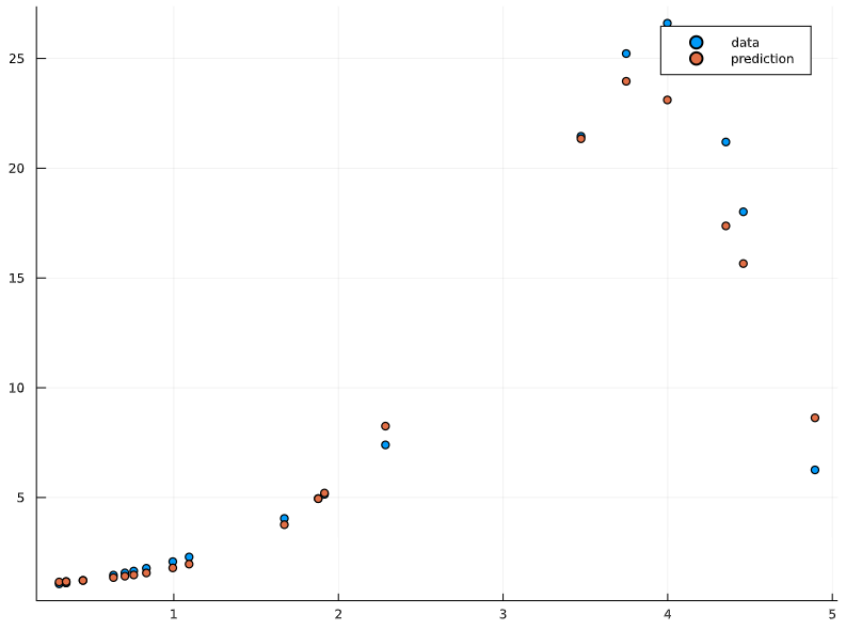


Figure 4: NODE predictions vs ground truth at sample times.

predictions at sample times, but this method is just as capable of interpolating as the previous RK4 backpropagation model.

5 Code

The code used to generate the results is located at the GitHub repository tqhdesilva/AMATH563-final-project in files named **train_lkv_node.jl** and **train_lkv_rk4.jl**. The project is a work in progress and contains code for experiments related to NODE that weren't described in this paper.

6 Conclusion

The main purpose of implementing RK4 backpropagation and attempting to compare it to NODE was mainly to better understand the advantages and inner workings of using the adjoint sensitivity method. We were also able to demonstrate that neural networks are very flexible when being used as function approximators, such as when approximating a differential equation.

Originally, the plans for this project were much more ambitious. Before writing the proposal, we actually had planned to try using Neural Stochastic Differential Equations(SDE) [5], since that tied in nicely to the previous course AMATH562 in the series. Then the goals were to implement the adjoint sensitivity method ourselves and try applying it to video interpolation [6]. After iteratively dialing back our expectations while trying to get a NODE model to train, we eventually settled on implementing backpropagation through RK4 and fitting a NODE on a downsized dataset. Many details, such as how to actually implement backpropagation when the loss depends on multiple hidden states, were not obvious until actually attempting to implement the method. In that sense, we were successful in learning more deeply about NODE. Other issues, such as needing to understand how Flux.jl and DifferentialEquations.jl work, also led to learning more about the Julia ecosystem around combining machine learning and differential equations.

References

[1] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[2] Jonty Sinai. Understanding neural odes, Jan 2019.

[3] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Ramadhan. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[4] Christopher Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. Diffeqflux.jl - A julia library for neural differential equations. *CoRR*, abs/1902.02376, 2019.

[5] Xuechen Li, Ting-Kam Leonard Wong, Ricky T. Q. Chen, and David Duvenaud. Scalable gradients for stochastic differential equations, 2020.

[6] Sunghyun Park, Kangyeol Kim, Junsoo Lee, Jaegul Choo, Joonseok Lee, Sookyung Kim, and Edward Choi. Vid-ode: Continuous-time video generation with neural ordinary differential equation, 2021.