Using Neural Differential Equations to Model Unknown Dynamical Systems

Team #1003
Los Alamos High School
1300 Diamond Drive
Los Alamos, NM 87544

April 8, 2020

New Mexico
Supercomputing Challenge
Final Report

Team Members: Robert Strauss - robert.strauss@pm.me
Project Mentor: Charlie Strauss

Abstract

To model a system with an ordinary differential equation, the mechanics of the system, represented by the functional form of the ODE, must be known. Theoretically, because of the universal approximation theorem, a neural net could approximate the true ODE and could learn just from data. This means a system for which we do not have an equation could be learned from measurements, and then modeled. Even if there is no ODE that could represent the system, a neural network could approximate the physics. This is called a neural differential equation, or neural ODE. The purpose of this project is to test the usefulness of neural ODEs to model systems with unknown dynamics. The Lotka-Volterra equations of predator/prey population dynamics are used as an example application. Solvers for the equations and neural ODE setups are constructed in the julia programming language. Truth data is generated by solving the equations, then neural ODEs are trained on this with automatic differentiation and back-propagation. First a neural ODE is trained to match 2-species dynamics, then 3-species dynamics with one population hidden from the net to simulate an unknown term, and finally 3-species dynamics with the initial population and its evolution fully hidden. Novel neural ODE architectures are designed for each problem, incorporating a recurrent channel and an external second neural net to provide an initial condition. In each case, neural ODEs learn to match testing data with initial conditions and time spans not used in training. The usefulness of neural ODEs for modeling unknown dynamical systems is demonstrated successfully.

Contents

1 Overview

1.1 Problem Statement

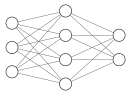
How can we model a dynamic system without a known equation describing it?

1.2 Executive Summary

Ordinary Differential Equations (ODEs) model the state of a system over time with an input/output "rule" relating the rate of change of the state to its current value.

$$\frac{\partial u}{\partial t} = f(u, t)$$

Neural Networks are input/output systems that can be trained to mimic the behavior of any function.



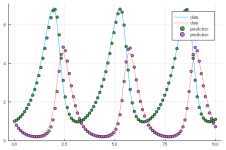
A neural net can act as the rule for a differential equation, allowing unknown behaviors to be learned. This is called a neural ODE.

$\frac{\partial u}{\partial t} =$ 

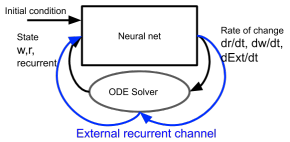
As an example problem, differential equations of predator/prey populations are tested with this neural ODE technique.

$$\frac{\partial r}{\partial t} = \alpha r - \beta r w$$
$$\frac{\partial w}{\partial t} = \delta r w - \gamma w$$

I trained a neural ODE on batches of data from solving the known equations. I then tested it on data it had never seen before, and over longer time ranges.



I designed my own neural ODE architectures for solving different problems. These are completely new inventions.



My neural ODE designs work as intended - they are able to infer the physics of an unknown system from data. Thus I have demonstrated the utility of neural ODEs.

2 Introduction

2.1 Motivations and historical context

The goals here are easy to motivate without much mathematics. Physics, for example, gives the rules of how a physical system evolves in time from an initial state. Specifically, Newton's law $F = ma$, is a concise statement of the rules written as a differential equation for the time evolution.

Often we are in the situation of having no idea what the rules/equations are! For example, the periodic behaviour of a simple pendulum can be measured, but before Newton, it would not have been known $F = ma$ governs its oscillations. In modern times, atmospheric turbulence and ocean waves are examples of systems we still don't know the exact differential equations for (only approximations).

A neural net is, in principal, able to represent any mathematical function [2], and can be trained using observed data. So the inspiration here is to simply replace the approximate or unknown terms with a neural net, then train it to act like the missing function using observed data. This is combination of neural networks and differential equations is called neural differential equations, or neural ODEs.

Training conventional neural networks, even deep and complex ones, is a sophisticated but routine task using back propagation. However back propagation through a differential equation solver, instead of just the neural net, wasn't fully practical prior to very recent innovations. Now that this can be done, neural ODE training is proving successful and effective.

While related concepts have been around since 1993 [3], the neural ODE was revived with the debut of the first truly practical software libraries implemented in Python in 2018 [1] and more fully in Julia in 2019. In fact, the Julia library is still in alpha and its authors will be publishing what may be the first comprehensive peer reviewed high-impact-journal publication in 2020. These have very recently been dubbed "Universal Differential Equations" [5].

The neural ODE is so new that many fundamental applications have yet to be explored. In particular, an unknown function in an equation could be replaced, but the relevant variables describing the system's state still have to be known. For example, the state of a simple pendulum (of known mass and length and gravity) is fully described by two variables: the angle of the pendulum and how fast the pendulum is traveling. But what if you didn't know, or could not measure, all the state variables? Suppose, unbeknownst to you, there was a second pendulum hiding inside the end of the first pendulum. In that case you could observe only the erratic wobbles in the outer pendulum's swing but not measure the state of the inner pendulum. Without the missing state variables of the internal pendulum there's no possibility of writing down the complete set of differential equations. So it's no longer a matter of simply filling in an unknown function with a neural net.

Can we use a neural ODE to learn not just the unknown functions in a differential equation but also to fill in the missing state variables and their relevant equations? This work originates a solution to that problem.

I reveal my solution in the context of a well-known toy problem (predator-prey population dynamics) by extending it into a direction that has not been previously considered.

2.2 Some Background on ODEs and Neural Networks

The rate of change of a variable is called its differential (with respect to time) An ordinary differential equation (ODE) relates *the rate of change of a variable to its present value.*

For example,

1. The rate of change of a bank account value is the interest rate times the current bank value.

$$\frac{\partial Money}{\partial t} = (Interest) * (Money)$$

2. The rate of change of the number of people infected with a virus is the number of people infected times the rate of transmission, minus the number of infected people times the rate of resolution.

$$\frac{\partial Sick}{\partial t} = TransmissionRate * Sick - ResolutionRate * Sick$$

3. Newton's Law $F = ma$ says the rate of change of velocity (i.e. acceleration)

is the force divided by the mass.

$$\begin{aligned} a &= \frac{F}{m} \\ \frac{\partial Velocity}{\partial t} &= \frac{F}{m} \\ \frac{\partial^2 Position}{\partial t^2} &= \frac{F}{m} \end{aligned} \tag{1}$$

Let us consider the left hand side of an ODE to always be the differential of a variable, and the right hand side to be some function of that variable. Any set of ODEs operating on multiple variables can be rewritten as a single ODE operating on a vector containing the variables. Let us call the collection of the variables describing a system the state of the system. Typically, to evolve a system's behavior through time this equation is integrated by iterating in short time steps: the rate of change of the state of the system is calculated by evaluating the equation, and then state is updated by the amount it changed in the short interval of time.

For our purposes, a neural network can be thought of as a black box, which will take in a vector input, and provide a vector output. By the universal approximation theorem [2], a neural network can be trained to return a specific output vector from a given input vector such that it can approximate any mathematical function.

2.3 Problem Statement

How can we model a dynamic system without a known equation describing it? Being able to do this could allow us to model almost any system we do not currently understand. And what if there are unknown variables at play?

2.4 Proposed Solution

Neural ODEs should be able to infer the rule of a system from data, allowing it to then be modeled. The second question, of modeling systems with possible unknown variables at play, has not previously been attempted with a neural ODE. We hope to do just this with a neural ODE by adding a "recurrent ODE channel".

2.5 Application: the Lotka-Volterra Equations

We choose the example problem of predator-prey population modeling as an application, which are described by the Lotka-Volterra equations (2). We will refer to the species in this model as rabbits, which are prey only, and wolves, which

eat rabbits. Bears, which eat both wolves and rabbits, can also be added in a three-species model (3). We generate truth data from solving the Lotka-Volterra equations rather than collecting real population data in the field, which would be significantly more difficult and restrictive.

The Lotka-Volterra equations are:

$$\begin{aligned} \frac{\partial r}{\partial t} &= \alpha r - \beta r w \\ \frac{\partial w}{\partial t} &= \delta r w - \gamma w \end{aligned} \tag{2}$$

Where w is the wolf population, r is the rabbit population, and α, β, γ, and δ are constants.

Modified for three species, they become:

$$\begin{aligned} \frac{\partial r}{\partial t} &= \alpha r - \beta r w - \epsilon r b \\ \frac{\partial w}{\partial t} &= \delta r w - \gamma w - \zeta w b \\ \frac{\partial b}{\partial t} &= \eta r b + \theta b w - \iota b \end{aligned} \tag{3}$$

Where b is the bear population and all Greek letters are constants.

To train the neural ODE we require a set of data representing measurements of the actual dynamical system state over time. To make this easy, we will generate the simulated measurements by solving the Lotka-Volterra equations with various parameter values. The data set will be samples of the continuous solution at specified time points. We will sample this coarsely for the data supplied to the neural ODE which we may pretend are measurements from the field, but for purposes of displaying it we can sample the smooth wave forms as finely as desired.

To avoid a potential confusion for the reader, we belabor the following point. This "true" ODE will only be used as a convenient and self consistent way to generate the data. No part of this "true" equation will be used in the neural ODE. None of the parameters of the true equation will be supplied to the neural ODE. And unless stated otherwise we even withhold the initial condition of the measurement simulation from the neural ODE. Thus once we create the data the reader could safely forget they ever saw the true ODE. It is not used in any way in the neural ODE and the only function term in the neural ODE is the neural net itself. The only purpose of the "true" ODE is to generate data more easily than acquiring it from the field.

3 Experimental Overview

To make this easier to follow by showing the intermediate steps, the goal is divided into three stages of problems that build on each other.

1. Can a neural net predict how quickly wolves will eat rabbits?
   (a) First, to test if a neural ODE really can learn to model a system without knowing an equation (just from data), we train a neural ODE to model a two species predator-prey population system without giving it any knowledge of the equations.
2. What if there are uncounted bears in the forest?
   (a) Next, to test how the neural ODE handles an unmeasured variable, we add a third species to the system but hide the data for it, except for the initial population, from the neural ODE.
3. What if we had no idea there were bears in the first place?
   (a) Finally, to test how a neural ODE handles a hidden variable problem, we fully hide the third species from the neural ODE and force it to model the other two species. It doesn't have any hint there even is a third species.

In each case, a "truth" model is created for solving the Lotka-Volterra equations, then an empty neural net is created and repeatedly integrated over a short time span and adjusted according to a loss function for training, then tested on a longer time span.

This was all completed by writing code in the Julia programming language version 1.0. An ODE function for the Lotka-Volterra equations was written. Neural networks were created with the `Flux` package. Differential equations were solved with the `DifferentialEquations` package, and back-propagation through a neural ODE used the `DiffEqFlux` package.

3.1 Experiment 1: Just Wolves and Rabbits

We aim to test the ability of a neural ODE to learn to model a system governed by unknown equations[1] just from data by training it on solutions from the 2-species Lotka-Volterra equations. Loss is calculated by predicting the time series by integrating the neural ODE and computing the squared difference to the true population "measurements" (samples from integrating the Lotka-Volterra equations). The neural ODE is trained with this definition of loss (and back-propagates all the way through many iterations of the ODE solver) on batches of many different initial conditions and time spans. To test what it learned, the

[1] the equations or functional form are never revealed to the neural ODE

neural ODE and Lotka-Volterra equations are solved with an initial condition not seen in the training set and over a longer period of time than in training. (This experiment was also done with the 3-species model but the results are unsurprising and omitted for brevity.)

3.2 Experiment 2: Hidden Bears

We aim to test the ability of a neural ODE to handle an unmeasured state variable by hiding one of the populations (bears) from it. The "measurements" or truth data are computed by integrating the true 3-species Lotka-Volterra equations. The neural ODE also is a 3-species version (it has 3 state-variable inputs and 3 differential outputs). However, the loss function remains the same 2-species loss function as in experiment 1. Only the squared difference sum for first two populations (rabbits and wolves) is calculated for loss. The predicted bear population is ignored in the loss function, because in this case we are pretending there are no measurements for the bear population (other than the initial condition, which is changed in experiment 3). The neural ODE is trained with this definition of loss on batches of many different initial conditions and time spans. To test what it learned, the neural ODE and Lotka-Volterra equations are solved with an initial condition not seen in the training set and over a longer period of time than in training.

The crucial innovation here is the "recurrent ODE channel" we create. This allows the neural ODE to keep track of unknown variables, which is the key to it succeeding in this experiment and the next. This is discussed in greater detail in the discussion section.

However, the recurrent channel was initiated with the initial bear population in this experiment, so bears are not entirely hidden. In the next experiment, even this information will be removed. (This experiment is a stepping stone to the next.)

3.3 Experiment 3: What Bears?

We aim to test the ability of a neural ODE to compensate for a completely hidden variable without a clue the variable even exists by completely hiding the population of the bears through time. This is almost the same as the previous test, except for one key difference. Previously, the recurrent channel of the neural ODE was initiated with the starting bear population. In this case, we instead implemented a separate neural network outside of the neural ODE which receives a short-time section of the wolf and rabbit populations and outputs a single scalar to initiate the recurrent channel with. The "true" initial value of the bear population is not involved in the system in any way.

Note that now both of the external neural net and the neural net inside the recurrent ODE must be trained simultaneously. We are not separately training

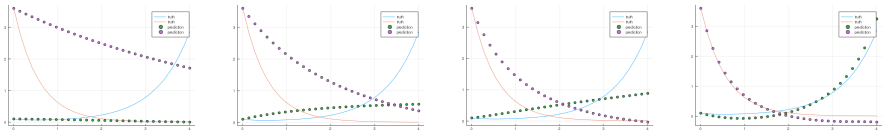


Figure 2: **Training (Wolves and Rabbits)** *All*: x-axis is time, y-axis is population. Dots represent the prediction from integrating the neural ODE, lines represent the true solution to the Lotka-Volterra equations. There are four plots, each made after different amounts of training. Note that going left to right (increasing amounts of training) the predictions begin to match the true solution.

the external net to predict some known bear population; we would not have that initial bear information since we are pretending we don't know there are bears at all. This means we also back-propagate all the way through all iterations of the solver and neural ODE just to get to training this network.

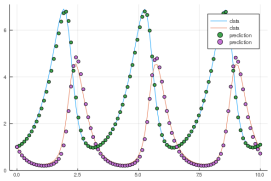
4 Results

In each of the test cases, after training, the neural ODE successfully matches testing data quite well. More importantly, despite minor numerical imperfections, it captured the characteristic behaviours of the wave forms over many cycles of the oscillations, extrapolating well beyond the short-time intervals of the training data. Stochastic minimization on batches was used, so different re-runs of the experiment produced slightly different results. As expected, with more epochs of training nearly all of solutions converged to the target wolf and rabbit test data and grew increasingly accurate. This is shown in figure **??**. (Occasional non-convergence is simply an expected pathology of stochastic minimization, not an alternative solution.)

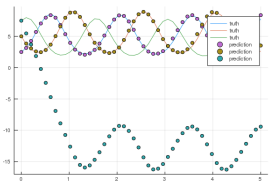
It should be noted that convergence to the data does not mean convergence to the same parameter values inside the neural net. The "surprising" outcome, addressed in the discussion below, is that no experimental run matched the actual hidden bear data. This is actually quite explainable and logical. Additionally the "bear" channel (the predicted bear population) produced a different output every run, so the internal parameters of the model were different. Since the neural net parameterization changed each re-run so did the ODE they represent, but all of these ODEs match the rabbit and wolf data, just not the "bear" population.

4.1 Training

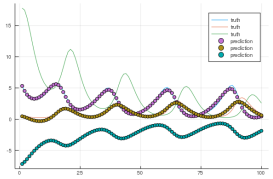
Through training, the prediction from integrating the neural ODE goes from inaccurate and meaningless to matching the true solution of the Lotka-Volterra equations. The gradual progression of training to match the Lotka-Volterra



(a) **Testing (Wolves and Rabbits)** The prediction from integrating the neural ODE matches the true solution of the Lotka-Volterra equations, is over a time span of multiple periods, and recovers from perturbations.



(b) **Testing (Hidden Bears)** Two of the three dotted lines match with the lines, while the third dotted line has a different behavior.



(c) **Testing (What Bears?)** The plots have nonlinear behaviors, with variable periods and amplitude. Two of the three lines match, while the third dotted line is far from the corresponding line.

Figure 3: *All*: X-axis is time, y-axis is population, lines represent the solution of the Lotka-Volterra equations, dots represent the prediction from integrating the neural ODE.
Middle and right: Blue is rabbits, orange is wolves, green is bears, purple dots are predicted rabbits, yellow dots are predicted wolves, blue dots are predicted bears/recurrent variable.
Left: Blue is rabbits, orange is wolves, green dots are predicted rabbits, purple dots are predicted wolves.

equations is shown in figure 2. The neural net is trained by back-propagating loss found by subtracting the solution of the neural ODE and the Lotka-Volterra equations. The two are solved from each initial condition in a batch and on the same time span. To test if the neural network actually learned something rather than memorizing, the model is applied to an initial condition it hasn't trained on and over a longer time span, so it most extrapolate what it knows. The plots in figure 3 are the results from these tests in each experiment.

4.2 Experiment 1: Just Wolves and Rabbits

Figure 3a shows the prediction from integrating the neural ODE matches up with the true solution to the Lotka-Volterra equations almost perfectly. The neural net was successfully trained through back-propagation through the ODE solver and successfully learned the Lotka-Volterra equations. The neural ODE was trained on solutions with much shorter time intervals than that shown in figure **??**. The initial condition used in this case is also unique, it did not appear in the training set. This means the neural ODE truly learned something, as it was able to match truth data it had never seen before, and extrapolate past the time interval it had learned in.

4.3 Experiment 2: Hidden Bears

Figure 3b shows the prediction from integrating the neural ODE matches up with the true solution to the Lotka-Volterra equations in only two of the three species. The neural ODE still figured out how to model the two species without knowing the population of the third. The third dotted line shown is actually data from the recurrent channel. Interestingly, this does not match with the bear population. Again, the neural ODE was trained on many different initial conditions and smaller time spans than what it was tested on, yet it still matched the two species we wanted it to on testing data. In the specific case shown, the data all looks very periodic. However, the neural ODE also learned the complicated non-periodic behaviors only describable with a third variable.

4.4 Experiment 3: What Bears?

Figure 3c shows the prediction from integrating the neural ODE matches up with the true solution to the Lotka-Volterra equations in two of the three species, and has a different behavior in the third. The two species the neural ODE trained for match almost perfectly. The third dotted line is again data from the recurrent channel, this time starting at a value chosen by the external neural network. Interestingly, this does not match with the bear population. The neural ODE also learned the complicated non-periodic behaviors only describable with a third variable.

5 Discussion and Conclusions

In all of the plots shown in the results, we are comparing the dotted lines representing the solution to the neural ODE, a sort of prediction, to the truth data, the solution to the Lotka-Volterra equations. Training was conducted on batches of solutions from many different initial conditions and of much shorter time ranges than in training. The fact that the solution to the neural ODE matches even over much longer time intervals than it trained for, and on an initial condition not seen in training, proves the neural ODE is generalizing and learning, not memorizing training data. So neural ODEs can safely be applied to new scenarios after training on a finite set of measurements.

The third population in the solution to the neural ODE, the bears, does not match the truth data. At first this seems puzzling because we theorized the neural ODE would learn to predict the bear population in this channel, but at closer inspection this makes sense. The third channel is not trained to the bear population, so the neural ODE may learn to put any information in this channel it finds useful. In fact, the only way the third channel could match the bear population is out of sheer luck. There's no way of it knowing the bear population, so it is unlikely it will happen to predict it. This third channel we are calling bears is no longer associated with the bear population at all. It's simply an abstract third channel that provides recurrent information.

The enabling innovation of this work is the "recurrent ODE channel". Crucially, this is distinct from a "recurrent channel" in the conventional neural network sense of the term. This is implemented simply as another output of the neural ODE which we keep track of, and return back to the neural ODE as an input later. However, unlike what is normally meant by a recurrent channel, this "recurrent channel" actually passes through the ODE solver too and is integrated. The reason we cannot use a conventional, non-integrated, recurrent neural network internal state is because the solver algorithm itself might jump back and forth in time; by letting the channel be differential we let the solver handle the intricate details of the time spacing and order of its calculations, just as it does for integrating the other normal channels.

The third channel is absolutely required since it would be impossible for a neural ODE to predict correctly with only the input of two of the three populations and no other information because the correct output depends on all three populations. That is for the same Rabbit and Wolf population the derivative output is a multi-valued function that will differ depending on the bear population. So, to give the neural ODE more information, we grant it a sort of "recurrent channel". It is able to put any scalar at one time step into this channel, and the following time step it will get the integral back as an input, allowing it to potentially learn to insert some useful information into the channel so that it may predict the multi-valued output correctly. This setup makes logical sense because, for a system for which we lack equations, we may not even

realize a factor has any effect in the system. This "recurrent channel" acts as an open slot for variables which we do not know about, but are required for the prediction of the system.

In the second experiment (training the neural ODE with a "recurrent channel" to match two of the three species), the channel is initiated with the starting population of bears. It is true that bears are not fully hidden in this experiment, but this is fixed in the next experiment. This experiment is really just a stepping stone to the next one. If you like, you may imagine it as a partially known variable – you know bears are in a forest affecting the system, but they are only measured once at the start because it is dangerous to repeatedly count bears.

In a neural ODE, the neural network acts as the rule of a differential equation. This means to produce a complete prediction through time, the neural ODE is integrated like an ordinary differential equation, being re-evaluated every time step. *Why not just have a neural net directly predict the entire solution instead?* There are several very important reasons. Doing it this way ensures the solution will have the properties we expect. An ODE forces the curves to be continuous no matter how sparsely sampled (not enforced by an ordinary neural net). We know there is likely some way of describing the system with an ordinary differential equation, however we don't know the rule or the variables of the system. So, enforcing the properties of the system we do know, that it will be in ODE form, speeds up the process and makes finding the solution easier. This can also be used to enforce conservation laws. Although in this example there is no conservation law between wolf and rabbit populations, in other cases this becomes important. This can also ensure the solution obeys consistent behaviors. For example, if all the free variables in a system return to a set of values which they already had, the same behavior should follow now as when it previously happened. If something happens twice, expect the same outcome each time. Direct prediction from a recurrent neural network can violate this principle.

Why didn't we use a parameter-fitting model containing dozens of guesses at possible mathematical combinations of the input (such as a Taylor series) rather than an entire neural network to find the rule? Parameter fitting seemingly has many advantages: much fewer trainable parameters (a parameter fitting model might have a few dozen while a neural net has thousands), and a readable outcome telling you which terms were needed and thus giving the functional form at the end. Recent research has shown that a parameter fitting model just does not work as well as a neural network for doing this job [5]. It is unclear why this is, but for now neural ODEs simply work better than parameter-fitting models, and the best strategy for model inference is to train the neural ODE then use it to generate additional data for parameter fitting.

6 Computer Code

The code and documentation written for this project can be found online at `https://github.com/robertstrauss/nnDiffEq`. Code is written in Julia language version 1.0. Packages used include

7 Acknowledgements

Dr. Charlie Strauss mentored me on coding and advised me on what direction to take my project while I was developing it. Chris Rackauckas was a very helpful influence in the online video tutorials he created [4].

References

[1] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2018.

[2] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[3] Ken ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks*, 6(6):801 – 806, 1993.

[4] Chris Rackauckas. Chris rackauckas, 2020.

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