An Introduction To the Field of Scientific Machine Learning

s174356 and s184335

February 2021

Contents

1 Introduction

There are two main branches of technical computing: machine learning and scientific computing.

The field of scientific computing is an area of science which spans many disciplines, but at its core, it involves the development of models and simulations to understand natural systems. This often involve the modelling language provided by dynamic systems which usually rely on differential equations.

Machine learning is the study of computer algorithms that improve automatically through experience. Machine learning algorithms build a model based on sample data, in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as email filtering and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks.

One problem with machine learning is that often one needs a lot of training data to make reliable predictions. Furthermore, you often also see that they can have a hard time extrapolating out of the domain where trained.

Differential equations on the other hand are often modeled by use of domain knowledge. Therefore the practitioner can set up equations to modeled observed changes of a system. These often extrapolate fairly well and do not need data to function.

However, there has been a recent convergence of the two disciplines. This field, scientific machine learning, has been showcasing results like how partial differential equation simulations can be accelerated with neural networks. New methods, such as probabilistic and differentiable programming, have started to be developed specifically for enhancing the tools of this domain. In the following we will explore methods from scientific computing and extend these methods to be more data-driven. At last we will end up discussing and implementing crucial parts of the NeuralODE.

If one after having read this paper still fell like more we recommend the online course "Parallel Computing and Scientific Machine Learning".

2 Explicit Runge-Kutta Methods

The first topic we will explore is explicit Runge-Kutta solvers. Runge-Kutta solvers comes in many different forms suited to solve different differential equations. Explicit Runge-Kutta solvers are made for solving non stiff ordinary differential equations of the form.

$$u' = f(u, t) \tag{1}$$

The following section in based on chapter 2 in [1], week 7 of the course [2] and this lecture from Manchester Metropolitan University.

2.1 The general framework

The simplest Runge-Kutta solver is the Euler method.

$$u_{n+1} = u_n + \Delta t f(u_n, t_n) \tag{2}$$

where

$$f(u, t) = u' = \frac{\mathrm{d}u}{\mathrm{d}t} \approx \frac{\Delta u}{\Delta t} \tag{3}$$

The method approximates the next state at time $t+1$ by stepping one step forward by use the differential of u at time t. This is a very easy method to understand and implement, but frankly also quite inaccurate. We can quantify the inaccuracy by looking at the Taylor expansion for 1 at time t.

$$u(t + \Delta t) = u(t) + \Delta t u'(t) + \frac{\Delta t^2}{2} u''(t) + \dots \tag{4}$$

We see that 2 matches the first two terms of the Taylor expansion.

$$\begin{aligned} u(t + \Delta t) &= u(t) + \Delta t f(u, p, t) + \mathcal{O}\left(\Delta t^2\right) \Rightarrow \\ \frac{u(t + \Delta t) - u(t)}{\Delta t} &= f(u, p, t) + \mathcal{O}(\Delta t) \end{aligned}$$

We hence see that our local error is linear in Δt. We can increase the accuracy of a given Runge-Kutta method by covering higher order terms in the Taylor expansion. The number of terms a method is covering is called the order of the method.

Definition 1. *A Runge-Kutta method has order p if for sufficiently smooth problems 1,*

$$u(t_n + \Delta t) - u_n \leq \mathcal{O}\left(\Delta t^p\right)$$

i.e., if the Taylor series for the exact solution $u(t_n + \Delta t)$ and for u_n coincide up to (and including) the term Δt^p.

To obtain these higher order methods, we evaluate the differential equation multiple times within the interval Δt. These intermediate evaluations are called stages and are more formally defined by:

Definition 2. *Let s be the number of stages and $a_{21}, a_{31}, a_{32}, \ldots, a_{s1}, a_{s2}, \ldots, a_{s,s-1}$, $b_1, \ldots, b_s$, $c_2, \ldots, c_s$ be real coefficients. Then the method*

$$\begin{aligned}
k_1 &= f\left(t_n, u_n\right) \\
k_2 &= f\left(t_n + c_2\Delta t, u_n + \Delta t a_{21} k_1\right) \\
k_3 &= f\left(t_n + c_3\Delta t, u_n + \Delta t\left(a_{31}k_1 + a_{32}k_2\right)\right) \\
&\ldots \\
k_s &= f\left(t_n + c_s\Delta t, u_n + \Delta t\left(a_{s1}k_1 + \ldots + a_{s,s-1}k_{s-1}\right)\right) \\
u_{n+1} &= u_n + \Delta t\left(b_1 k_1 + \ldots + b_s k_s\right)
\end{aligned}$$

is called an s-stage explicit Runge-Kutta method.

To see these definitions in action, we will derive a second order method.

2.2 An explicit Runge-Kutta method of second order

We will abbreviate the second order method RK2 in the following, where we will also use Δt interchangeably with $\mathrm{d}t$. The RK2 method is given as

$$u_{n+1} = u_n + (b_1 k_1 + b_2 k_2)\mathrm{d}t \tag{5}$$

where

$$k_1 = f(t_n, u_n) \tag{6}$$
$$k_2 = f(t_n + c_2\mathrm{d}t, u_n + a_{21}k_1\mathrm{d}t) \tag{7}$$

To obtain a second order method, we need to determine the values of b_1, b_2, c_2 and a_{21} such that we cover all the second order terms of the Taylor expansion. Consider the Taylor expansion at u_{n+1}:

$$\begin{aligned}
u_{n+1} &= u_n + f(t_n, u_n) + \frac{f'(t_n, u_n)}{2!}\mathrm{d}t^2 + \mathcal{O}(\mathrm{d}t^3) \Rightarrow \\
u_{n+1} &= u_n + f(t_n, u_n) + \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial u}\frac{du}{\mathrm{d}t}\right)\frac{\mathrm{d}t^2}{2!} + \mathcal{O}(\mathrm{d}t^3)
\end{aligned} \tag{8}$$

To determine the constants, we will do a little trick. First, we will expand 7 by use of the multivariate Taylor expansion which for a function g(x,y) is given as:

$$g(x+r, y+s) = g(x,y) + r\frac{\partial g}{\partial x} + s\frac{\partial g}{\partial y} + ...$$

We expand 7

$$f(t_n + c_2\mathrm{d}t, u_n + a_{21}k_1\mathrm{d}t) = f(t_n, u_n) + c_2\mathrm{d}t\frac{\partial f}{\partial t} + a_{21}k_1\frac{\partial f}{\partial u} + \mathcal{O}(\mathrm{d}t^2) \quad (9)$$

We can now substitute 6 and 9 into 5

$$u_{n+1} = u_n + b_1\mathrm{d}t f(t_n, u_n) + b_2\mathrm{d}t f(t_n, u_n) + b_2c_2\mathrm{d}t^2\frac{\partial f}{\partial t} + b_2a_{21}\mathrm{d}t^2 f(t_n, u_n)\frac{f}{u} + O(\mathrm{d}t^3)$$

We sort terms by order

$$\begin{aligned} u_{n+1} = u_n &+ \left[b_1 f(t_n, u_n) + b_2 f(t_n, u_n)\right]\mathrm{d}t \\ &+ \left[b_2c_2\frac{\partial f}{\partial t} + b_2a_{21}f(t_n, u_n)\frac{\partial f}{\partial u}\right]\mathrm{d}t^2 + \mathcal{O}(\mathrm{d}t^3) \end{aligned} \quad (10)$$

We can now by comparing 8 and 10 determine the coefficients b_1, b_2, c_2 and a_{21}.
First we compare first order terms.

$$\begin{aligned} b_1 f(t_n, u_n) + b_2 f(t_n, u_n) = f(t_n, u_n) \Leftrightarrow \\ b_1 + b_2 = 1 \end{aligned} \quad (11)$$

We then determine second order terms w.r.t. t

$$\begin{aligned} b_2c_2\frac{\partial f}{\partial t} &= \frac{1}{2!}\frac{\partial f}{\partial t} \\ \implies \quad b_2c_2 &= \frac{1}{2} \end{aligned} \quad (12)$$

Lastly we determine second order terms w.r.t. u

$$\begin{aligned} b_2a_{21}f(t_n, u_n)\frac{\partial f}{\partial u} &= \frac{1}{2!}\frac{\partial f}{\partial u}\frac{du}{\mathrm{d}t} \\ \implies \quad b_2a_{21}f(t_n, u_n)\frac{\partial f}{\partial u} &= \frac{1}{2!}f(t_n, u_n)\frac{\partial f}{\partial u} \\ \implies \quad b_2a_{21} &= \frac{1}{2} \end{aligned} \quad (13)$$

We can now solve the system of equations and get

$$\begin{aligned} b_1 = b_2 = \frac{1}{2} \\ c_2 = a_{21} = 1 \end{aligned}$$

These are the coefficients of the second order method RK2.

2.3 Higher order methods

A convenient way to write the coefficients for a Runge-Kutta method is by use of the Butcher Tableau which are written as in table 1.

c_1	a_{11}	a_{12}	$\cdots$
c_2	a_{21}	a_{22}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$
	b_1	b_2	$\cdots$

Table 1: A general Butcher tableau

So for the RK2 method we have the Butcher tableau 2.

0	0	0
1	1	0
	$\frac{1}{2}$	$\frac{1}{2}$

Table 2: The Butcher tableau for the RK2 method

Probably the most famous Runge-Kutta method is the 4th order RK4 method or the classic Runge-Kutta method. In table 3 the Butcher tableau for the RK4 method is given. We see that the 4th order method has 4 stages.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Table 3: The Butcher tableau for the classic RK4 method

Methods of order 4 is the highest order methods for which we have algorithms with the same number of stages. For methods of order $p = 5$ or higher, there are no algorithms with $s = p$ stages. [thm 5.1 [1]]
One very popular 5th order method is the DOPRI54 method. It is in detail explained how to derive the method on page 175-179 in [1] but we will only state the Butcher tableau here.

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
y_1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0
$\widehat{y}_1$	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

Table 4: The Butcher tableau for the DOPRI54 method

We see in table 4 that two b-rows are present. The first row indicated by y_1 is a 5th order which is used to step forward. The second row, $\hat{y}_1$, is a 4th order method which is used to get a local error estimate at every step. An implementation in Julia can be seen below.

```
function solve_system_save_Dormand_Prince_54(f,u0,p,n,dt,error_est=false)
  u = Vector{typeof(u0)}(undef,n)
  if error_est
    e = Vector{typeof(u0)}(undef,n)
    @inbounds e[1] = 0.0*u0
  end
  k1 = @SVector[0.0,0.0]
  k2 = @SVector[0.0,0.0]
  k3 = @SVector[0.0,0.0]
  k4 = @SVector[0.0,0.0]
  k5 = @SVector[0.0,0.0]
  k6 = @SVector[0.0,0.0]
  @inbounds u[1] = u0
  @inbounds for i in 1:n-1
      k1 = f(u[i],p)
      k2 = f((@muladd u[i]+dt*(1/5*k1)),p)
      k3 = f((@muladd u[i]+dt*(3/40*k1+9/40*k2)),p)
      k4 = f((@muladd u[i]+dt*(44/45*k1-56/15*k2+32/9*k3)),p)
      k5 = f((@muladd u[i]+dt*(19372/6561*k1-25360/2187*
           k2+64448/6561*k3-212/729*k4)),p)
      k6 = f((@muladd u[i]+dt*(9017/3168*k1-355/33*k2+
           46732/5247*k3+49/176*k4-5103/18656*k5)),p)
      @muladd u[i+1] = u[i]+dt*(35/384*k1+500/1113*k3+
            125/192*k4-2187/6784*k5+11/84*k6)
      if error_est
        k7 = k6 = f(u[i].+dt,p)
        uhat = u[i]+dt*(5179/57600*k1+7571/16695*k3+
               393/640*k4-92097/339200*k5+187/2100*k6+1/40*k7)
        e[i+1] = abs.(u[i+1]-uhat)
```

```
        end
    end
    if error_est
        return u,e
    else
        return u
    end
end
```

To test the implementation, we will solve the Lotka-Volterra equations which are stated in 14. They can be used to model the dynamics of interacting spices and often give rise to oscillating systems.

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha x - \beta xy \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= \delta xy - \gamma y \end{aligned} \tag{14}$$

We solved the system with parameters $\alpha = 1.5$, $\gamma = 3.0$, $\beta = 1.0$ and $\delta = 1.0$, $t \in [0, 10]$, $\Delta t = \frac{1}{4}$, and $u(0) = [x(0), y(0)] = [1, 1]$.

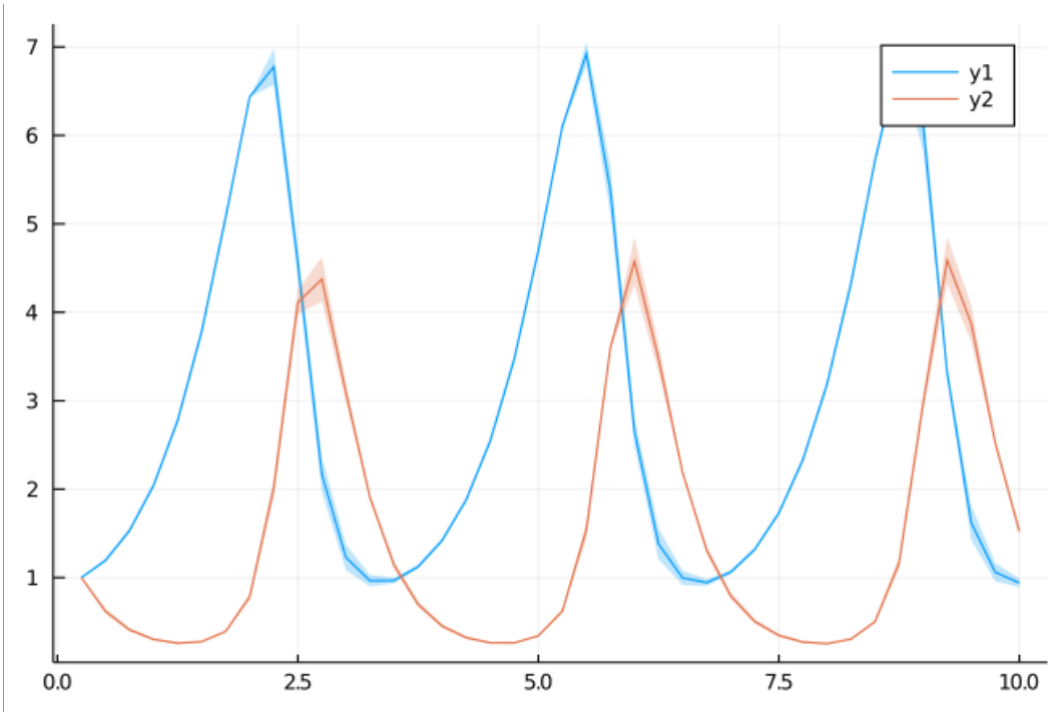


Figure 1: The Lotka-Volterra model described by the differential equations 14. We see our solver catches the pattern and the error estimated by the embedded 4th order method is shown as the shaded area around the lines.

We see the DOPRI54 method accurately solves the system with a step size of $\Delta t = \frac{1}{4}$. We though also see that the solution is pretty jagged and if we increase the step size to $\frac{1}{3}$, the solver would diverge. If we had used an explicit solver with an even higher order we could maybe have solved the system with a step size of $\frac{1}{3}$. The reason behind this, can be explained by what is known as stability of the method. We will hence devote the last part of this section to this topic.

2.4 Stability

To examine linear stability of a method we will use *the test equation*

$$u'(t) = \lambda u(t) \tag{15}$$

When we solve 15 with a Runge-Kutta solver the method generates a discrete dynamical system. We want the system to converge to the true solution of 15. We will illustrate this by solving 15 with Eulers method.

$$\begin{aligned} u_{n+1} &= u_n + \Delta t f(u_n, \Delta t) \Rightarrow \\ &= u_n + \Delta t \lambda u_n \Rightarrow \\ &= (1 + \Delta t \lambda) u_n \Rightarrow \\ &= (1 + z) u_n \end{aligned} \tag{16}$$

We see that z must be in the set $\{z \in \mathbb{C} | \operatorname{Re}(z) \in [-2, 0] \cup \operatorname{Im}(z) \in [-1, 1]\}$, for the Euler method to converge. This set is illustrated in figure 2.

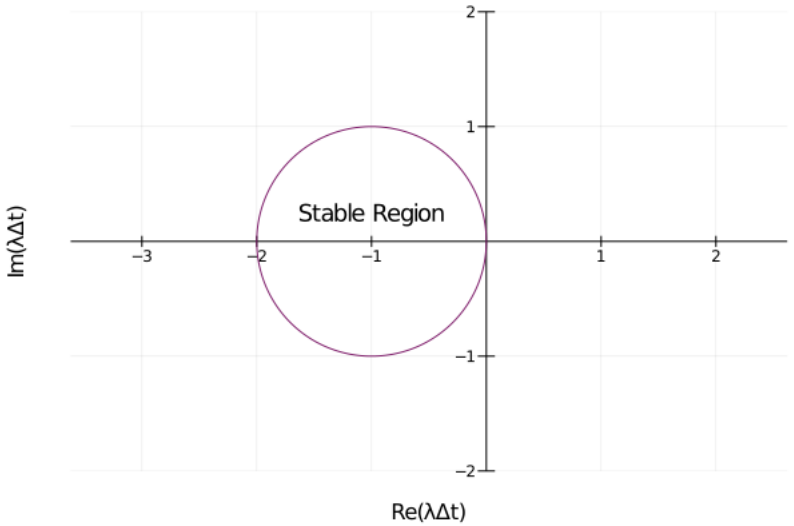


Figure 2: The region of absolute stability for the Euler method.

To see what happens when the step size Δt gets out of the stability region, we solved the problem $u' = -2.3u$ with the Euler method in figure 3. First with a step size of $\Delta t = 0.7$ which gives $z = \lambda \Delta t = -2.3 \cdot 0.7 = -1.61$ that is within the stable region. Afterwards, we tried with a step size of $\Delta t = 1$ which gives $z = -2.3$ that is outside the stable region. It is also clear from 3 that $\Delta t = 0.7$ converges and $\Delta t = 1$ diverges.

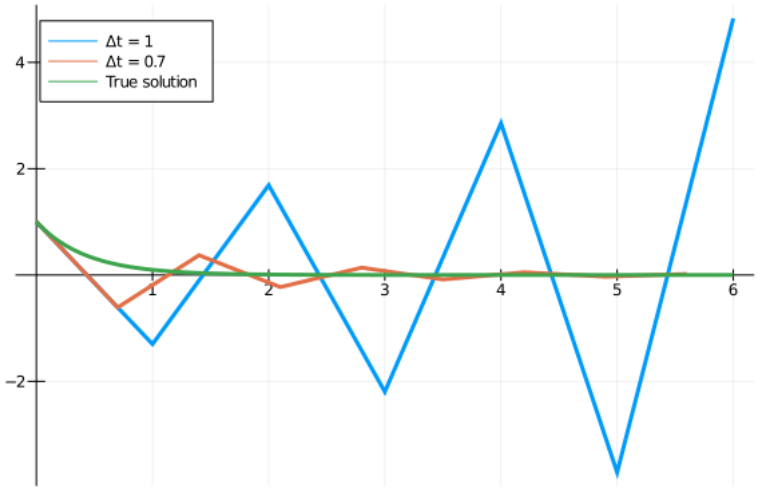


Figure 3: The problem $u'(t) = -2.3u(t)$ solved by the Euler method with different step sizes.

To generalize this concept to higher order methods we define a stability function $R(z)$.

$$u_{n+1} = R(z)u_n$$

where the stable region is given by

$$\{z \in \mathbb{C} | \ |R(z)| \leq 1\}$$

Ignoring the c's we have that a general Runge-Kutta method is given by

$$\begin{aligned} k_1 &= y_0 + a_{11}k_1 + a_{12}k_2 + \cdots + a_{1s}k_s, \\ k_2 &= y_0 + a_{21}k_1 + a_{22}k_2 + \cdots + a_{2s}k_s, \\ &\vdots \\ k_s &= y_0 + a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{ss}k_s. \end{aligned}$$

We now let $U = (k_1, k_2, \ldots, k_s)^T$ and $\mathbf{1} = (1, 1, \ldots, 1)^T$, then

$$\begin{aligned} U &= \mathbf{1}u_0 + \Delta t \lambda A U \Rightarrow \\ U &= (I - zA)^{-1}\mathbf{1}u_0 \end{aligned}$$

We can now substitute into 16.

$$\begin{aligned} u_1 &= u_0 + \Delta t \lambda b^T U \\ &= u_0 + z b^T (I - zA)^{-1}\mathbf{1}u_0 \\ &= (1 + z b^T (I - zA)^{-1}\mathbf{1})u_0 \end{aligned}$$

We hence see that for a general Runge-Kutta method, the stability function $R(z)$ is given by

$$R(z) = 1 + z b^T (I - zA)^{-1}\mathbf{1} \tag{17}$$

Using the geometric series of matrices, $(I - T)^{-1} = \sum_{i=0}^{n} T^i$, we can rewrite 17 as

$$R(z) = 1 + z b^T (I + zA + z^2 A^2 + z^3 A^3 + ...)\mathbf{1}$$

For explicit methods we can further rewrite by applying

$$b^T A^i \mathbf{1} = b^T A^{i-1} c$$

Which gives

$$R(z) = 1 + z b^T \mathbf{1} + z^2 b^T A^0 c + z^3 b^T A^1 c + ... \tag{18}$$

The representation, 18, gives us a way to verify the order of an arbitrary Butcher Tableau. The solution to the test problem 15 is given by

$$\begin{aligned} u(t) &= \exp(z) \\ &= \sum_{p=0}^{\infty} \frac{z^p}{p!} \\ &= 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4 + ... \end{aligned} \tag{19}$$

The order of a given Runge-Kutta method is determined by how many of the fractions in 19 the $R(z)$ function associated to the Runge-Kutta method, matches. To see this we will try with the 5th order method in DOPRI54.

$$R_{DOPRI54}(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4 + \frac{1}{120}z^5 + \frac{1}{600}z^6 \tag{20}$$

We compare with 19

$$u(t) \approx \sum_{p=0}^{6} \frac{z^p}{p!}$$
$$= 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4 + \frac{1}{120}z^5 + \frac{1}{720}z^6$$

We hence see that as expected the 5th order method in DOPRI54 covers the fractions up to fifth order term and partially covers the sixth order term. In figure 4 we have plotted the region of absolute stability for the 5th order method in DOPRI54 with a 6th order method given by the $p = 0..6$ terms in 19.

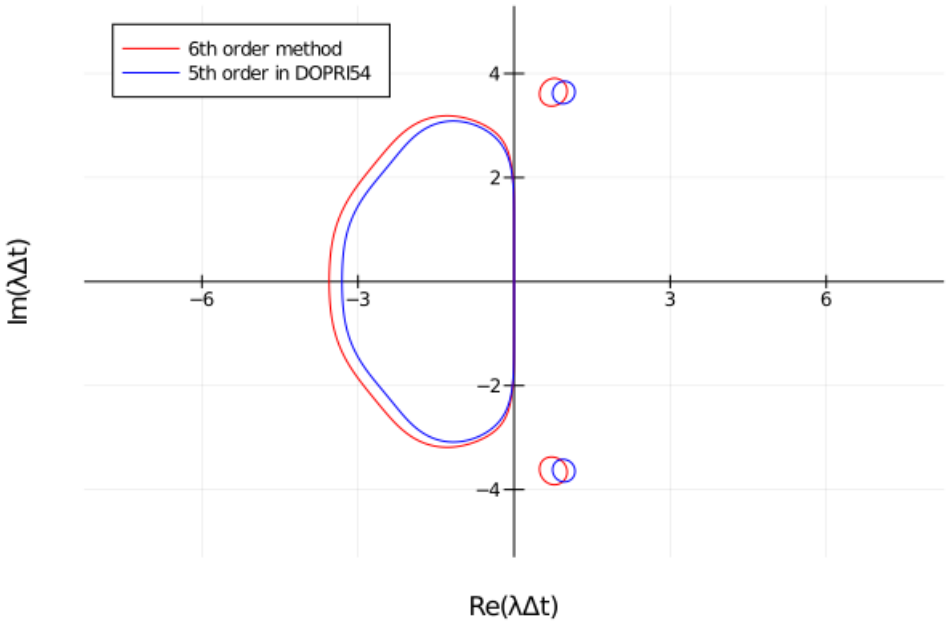


Figure 4: The stable region for the 5th order method in DOPRI54 togther with a 6th order method given by the $p = 0..6$ terms in 19.

3 Forward Sensitivity and Forward-Mode Automatic Differentiation

In this section we will introduce the forward sensitivity analysis and Forward-Mode Automatic Differentiation. We will also explore how sensitivity analysis can be used to quantify the effect a change in initial conditions and parameters will have on the trajectories.
First, we will consider a simple system to understand the basics of sensitivity analysis, then we will move into dual numbers and thereby forward mode automatic differentiation.

3.1 An introduction to sensitivity analysis

Consider a simple initial value problem.

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x), \qquad x(0) = x_0 \tag{21}$$

We want to investigate how much a change in the initial condition x_0 affects the solution, x. As a measure of this change, we introduce the sensitivity coefficient $\frac{\partial x}{\partial x_0}$. As x changes over time, so does $\frac{\partial}{\partial x_0}x(t)$. To understand this, we can imagine a system where the initial condition means a lot in the beginning and after some time, the trajectories converge to some common attractor.

We are interested in how $\frac{\partial x}{\partial x_0}(t)$ evolves over time. Therefore, we consider the time derivative of the sensitivity coefficient:

$$\frac{1}{\partial t}\left(\frac{\partial x}{\partial x_0}\right) \tag{22}$$

We assume that the derivatives of x are sufficiently smooth functions s.t.

$$\begin{aligned}\frac{1}{\partial t}\left(\frac{\partial x}{\partial x_0}\right) &= \frac{\partial^2 x}{\partial t \partial x_0} \\ &= \frac{\partial}{\partial x_0}\frac{\partial x}{\partial t}\end{aligned}$$

We now recognize $\frac{\partial x}{\partial t} = f(x)$. As a consequence of the chain rule and as f depends on x

$$\begin{aligned}\frac{1}{\partial t}\left(\frac{\partial x}{\partial x_0}\right) &= \frac{\partial f}{\partial x_0} \\ &= \frac{\partial f}{\partial x}\frac{\partial x}{\partial x_0}\end{aligned} \tag{23}$$

equation (23) is called the variational equation and must be solved simultaneous with system (21) to obtain $\partial x/\partial x_0\,(t)$
If f depends directly on some parameters, then the expression is slightly different. By the chain rule, the sensitivity coefficient for the parameter would be

$$\frac{\partial f(x,p)}{\partial p} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial f}{\partial p}$$

Example For the Lotka-Volterra equation, we can apply the analysis for the sensitivity of α on x. Let $\dot{x} = f_1$ and consult equation (14) to see

$$\begin{aligned}\frac{1}{\partial t}\left(\frac{\partial x}{\partial \alpha}\right) &= \frac{\partial f_1}{\partial x}\frac{\partial x}{\partial \alpha} + \frac{\partial f_1}{\partial \alpha} \\ &= \frac{\partial\left(\alpha x - \beta x y\right)}{\partial x}\frac{\partial x}{\partial \alpha} + \frac{\partial\left(\alpha x - \beta x y\right)}{\partial \alpha} \\ &= \left(\alpha - \beta y\right)\frac{\partial x}{\partial \alpha} + x\end{aligned} \tag{24}$$

usually $\partial x/\partial\alpha(0) = 0$ as the initial condition is independent of the parameter.
In equation (24) we see that the change in the sensitivity coefficient depends on the sensitivity of $\frac{\partial x}{\partial \alpha}$, the solution of y and x.
As mentioned before, we need to solve the above equation simultaneously with the normal Lotka-Volterra system of equation and the variational equation for each the parameters. Therefore, we should also setup the system for $\frac{\partial x}{\partial \beta}$, $\frac{\partial y}{\partial \delta}$, $\frac{\partial y}{\partial \gamma}$ and the initial conditions. This is a tedious process and the number of of differential equations we have to solve simultaneous explodes for more complex systems. To resolve this issue, we introduce forward-mode automatic differentiation.

3.2 Forward-Mode Automatic Differentiation

Forward-Mode Automatic Differentiation is a way to calculate the sensitivities fast using dual numbers and maintain machine precision on the derivatives.

To understand why dual numbers are important in the calculation of derivatives, consider the finite difference method:

$$f'(x) = \lim_{\epsilon \to \infty} \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{25}$$

If we use floating point numbers, then we can store round 16 digits accurately. If $\epsilon = 10^{-5}$, then $f(x+\epsilon)$ and $f(x)$ are approximately equal in the first 5 digits. When we do the subtraction in equation (25), we would be left with only 11 digits of precision in the result.
Then we divide by $\epsilon = 10^{-5}$ which brings up the 11 digits to relative size of 16 digits but 5 of them are insignificant digits. If we continue this procedure, we would end up manipulating numbers of no significance.

Dual numbers is a way to do better. We bring in this concept by use of complex numbers. Consider some function f that is complex analytic and some $x \in \mathbb{R}$. We want to calculate f' as a real derivative in an efficient way. We expand our function as a Taylor series but purely in the direction of the complex plane

$$f(x+ih) = f(x) + ihf'(x) + \mathcal{O}(h^2)$$

isolate $if'(x)$

$$if'(x) = \frac{f(x+ih) - f(x)}{h} + \mathcal{O}(h) \tag{26}$$

we take out the imaginary part on both sides

$$\begin{aligned} f'(x) &= \operatorname{Im}\left(\frac{f(x+ih) - f(x)}{h} + \mathcal{O}(h)\right) \\ &= \left(\operatorname{Im}\frac{f(x+ih)}{h}\right) + \operatorname{Im}(\mathcal{O}(h)) \qquad \text{as } \operatorname{Im}(f(x)) = 0 \\ &= \frac{\operatorname{Im}\left(f(x+ih)\right)}{h}\mathcal{O}(h) \end{aligned}$$

If we take h sufficiently small, we do a complex step differentiation. Compare with equation (25) and notice that there is no subtraction in the complex differentiation. This means that we maintain a high accuracy.

With this intuition, we instead take elements from smooth infinitesimal analysis to come up with dual numbers. Consider again f and this time some small perturbation ϵ. Then we see how much f is affected by ϵ

$$f(x+\epsilon) = f(a) + \epsilon f'(a) + \mathcal{O}(\epsilon)$$

we invoke parts of nilpotent algebra and stipulate $\epsilon^2 = 0$. Each function will be represent by the value of $f(a)$ and the derivative $f'(a)$ which would be encoded as coefficients of a Taylor polynomial of degree 1:

$$f \rightsquigarrow f(a) + \epsilon f'(a) \tag{27}$$

With this representation and in this setting, we have to define how each arithmetic operation should be performed. Therefore, we introduce two generic values

$$\begin{aligned} f &\rightsquigarrow f(a) + \epsilon f'(a) \\ g &\rightsquigarrow g(a) + \epsilon g'(a) \end{aligned} \tag{28}$$

We define addition and multiplication

$$\begin{aligned} (f+g) &= [f(a) + g(a)] + \epsilon\,[f'(a) + g'(a)] \\ (f \cdot g) &= [f(a) \cdot g(a)] + \epsilon\,[f(a) \cdot g'(a) + g(a) \cdot f'(a)] \end{aligned} \tag{29}$$

This we can easily encode in julia

```
struct MultiDual{N,T}
    val::T
    derivs::SVector{N,T}
end
```

Then we define addition with another dual number, a dual and real number and real and dual number in julia

```
function +(f::MultiDual{N,T}, g::MultiDual{N,T}) where {N,T}
    return MultiDual{N,T}(f.val + g.val, f.derivs + g.derivs)
end
function +(f::MultiDual{N,T}, α::Number) where {N,T}
    return MultiDual{N,T}(f.val + α, f.derivs)
end
function +(α::Number, f::MultiDual{N,T}) where {N,T}
    return f + α
end
```

Likewise we define all arithmetic and powers as:

```
function -(f::MultiDual{N,T}, g::MultiDual{N,T}) where {N,T}
    return MultiDual{N,T}(f.val - g.val, f.derivs - g.derivs)
end
```

```
function -(f::MultiDual{N,T}, α::Number) where {N,T}
    return MultiDual{N,T}(f.val - α, f.derivs)
end
function -(α::Number, f::MultiDual{N,T}) where {N,T}
    return f - α
end
function -(f::MultiDual{N,T}) where {N,T}
    return -1*f
end

# Multiplication, Importent to note is that for the derivative part of
#   the dual number multiplication is difined by the product rule.
function *(f::MultiDual{N,T}, g::MultiDual{N,T}) where {N,T}
    return MultiDual{N,T}(f.val * g.val, f.val .* g.derivs + g.val .* f.derivs)
end
function *(α::Number, g::MultiDual{N,T}) where {N,T}
    return MultiDual{N,T}(α * g.val, α * g.derivs)
end
function *(g::MultiDual{N,T}, α::Number) where {N,T}
    return α * g
end


# Division, Importent to note is that for the derivative part of
#   the dual number division is difined by the quotient rule.
function /(f::MultiDual{N,T}, g::MultiDual{N,T}) where {N,T}
    return MultiDual{N,T}(f.val/g.val,
                (f.derivs .* g.val - f.val .* g.derivs)/(g.val.^2))
end
function /(α::Number, g::MultiDual{N,T}) where {N,T}
    return MultiDual{N,T}(α/g.val, -α*g.derivs ./ g.val^2)
end
function /(g::MultiDual{N,T}, α::Number) where {N,T}
    return g * inv(α)
end

# Power
function ^(g::MultiDual{N,T}, α::Number) where {N,T}
    return MultiDual{N,T}(g.val^α, α.*g.derivs.^(α-1))
end
```

3.2.1 Higher dimensional dual numbers

Moving to higher dimensional example, we first introduce a dual number and a $f(x)$:

$$x = x_0 + v_1\epsilon_1 + \ldots + v_m\epsilon_m$$

and likewise

$$f(x) = f(x_0) + f'(x_0)v_1\epsilon_1 + \ldots + f'(x_0)v_m\epsilon_m$$

here $f'(x_0)v_i$ would be the directional derivative in the direction of v_i. In most use cases, we want to compute the gradient of the input. In that case,

$v_1, v_2, ..., v_m$ is the standard basis for $\mathbb{R}^m$. Formally $v_i = e_i$ where $e \in \mathbb{R}^m$ and $[e_j] = 0$ for all $j \neq i$ and $[e_j] = 1$ for $j = i$. For instance, $e_1 = \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix}^T$. This need for higher dimensional dual numbers is best shown with an example.

3.3 Derivatives of the parameters for Lotka-Volterra

In our case we have $\dot{x} = f(x; P)$ hence depends f both on the state, x, and the parameters P. We assume that we have k parameters in P and we want to calculate the sensitivity coefficients for each of the parameters as in sub-section 3.1. To do this using dual numbers, we introduce:

$$\begin{aligned} x &= x_0 + 0\epsilon_1 + \ldots + 0\epsilon_k \\ y &= y_0 + 0\epsilon_1 + \ldots + 0\epsilon_k \\ P &= p + e_1\epsilon_1 + \ldots + e_k\epsilon_k \end{aligned} \tag{30}$$

we have extended P with k ϵs - one for each parameter. The neat property emerge when we evaluate f with P

$$\begin{aligned} f(x; P) &= f(x; p) + \frac{\partial f}{\partial p_1}\epsilon_1 + \ldots + \frac{\partial f}{\partial p_k}\epsilon_k \\ f(y; P) &= f(y; p) + \frac{\partial f}{\partial p_1}\epsilon_1 + \ldots + \frac{\partial f}{\partial p_k}\epsilon_k \end{aligned}$$

This means that in this new framework of dual number arithmetic, we can compute the dynamics of our system and on the fly calculate the sensitivity coefficients without the use of variational equations. We reduce the time needed to setup the system, we reduce computational complexity and the derivative are calculated with machine precision.
This is the essence of the method and tool called *forward automatic differentiation*.

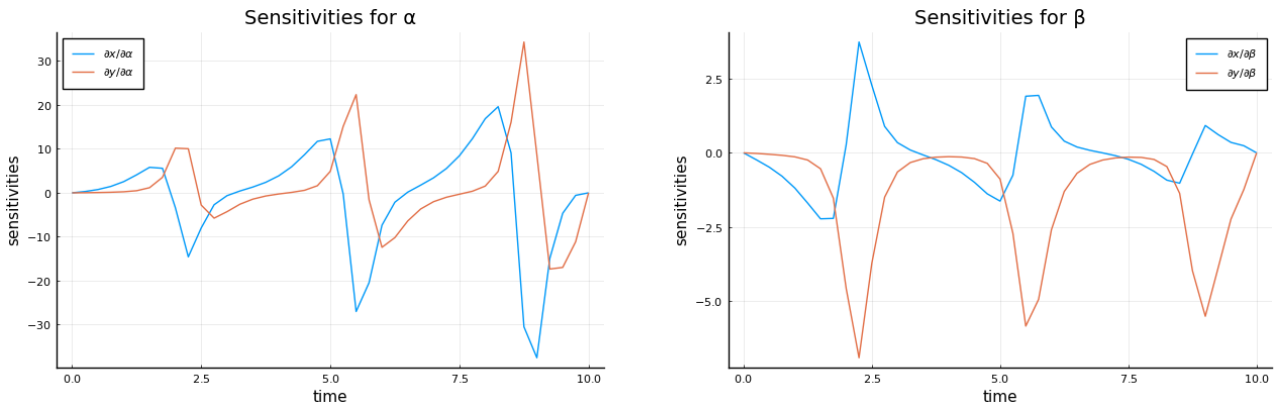
This is straigt forward to implement in *Julia*:

```
(x0,y0) = (1,1)
dx0 = MultiDual(x0, SVector(0.0, 0.0, 0.0, 0.0))
dy0 = MultiDual(y0, SVector(0.0, 0.0, 0.0, 0.0))
u0 = [dx0,dy0]

(p1, p2, p3, p4) = (1.5, 1.0, 3.0, 1.0)
dp1 = MultiDual(p1, SVector(1.0, 0.0, 0.0, 0.0))
dp2 = MultiDual(p2, SVector(0.0, 1.0, 0.0, 0.0))
dp3 = MultiDual(p3, SVector(0.0, 0.0, 1.0, 0.0))
dp4 = MultiDual(p4, SVector(0.0, 0.0, 0.0, 1.0)
p = [dp1, dp2, dp3, dp4]
```

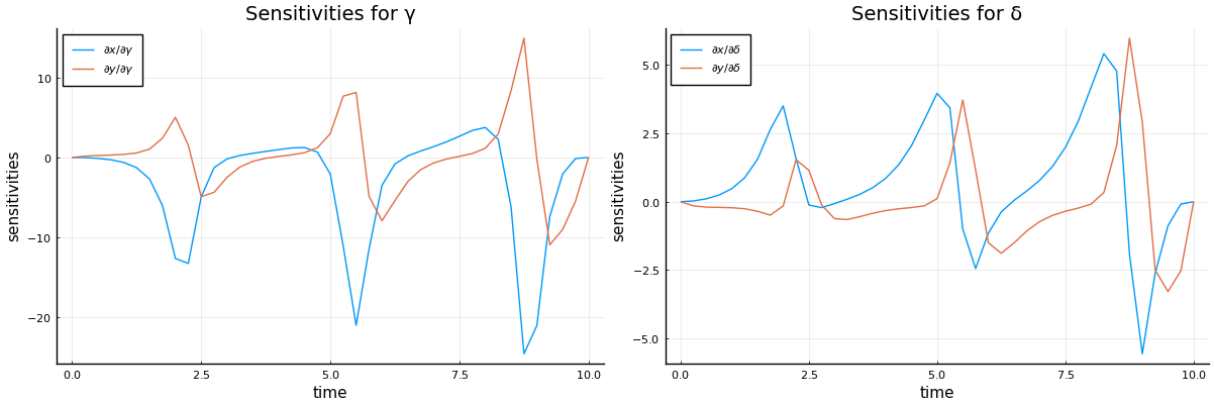
We can now perform the simulation of Lotka-Volterra just as in section (2). For each iteration, we now save both x and y and all the derivatives w.r.t. each of the parameters. We can now easily take out the first parameter α and plot:

```
x_sens_α = [DP5sens[i][1].derivs[1] for i in 1:length(DP5sens)]
y_sens_α = [DP5sens[i][2].derivs[1] for i in 1:length(DP5sens)]
```



(a) Sensitivities for the parameter α

(b) Sensitivities for the parameter β



(c) Sensitivities for the parameter γ

(d) Sensitivities for the parameter δ

Figure 5: Sensitivities for all the parameters

This is the intuition behind how forward automatic differentiation works. Below are just some short comments on how to understand the calculated sensitivities of the system. This is used in other areas of dynamical systems.

3.3.1 Interpretation of sensitivities

Consider figure 5a with the sensitivities for α. In the first time interval $\tau_1 = [0, \approx 2]$, the sensitivities are both positive which means an increase in α would lead to an increase in both x and y. This is only in the timespan τ_1, however, for $t > \tau_1$ then $\frac{\partial y}{\partial \alpha}$ starts to decrease and becomes negative. When the sensitives are negative, it means that an increase in α would lead to a decrease in y. Sometimes one would normalize the sensitivity coefficients using the ratio between parameter and variable to better asses how a relative change will affect the solution. This is encapsulated in the elasticity coefficients:

$$\epsilon_{y,\alpha} = \frac{\partial y}{\partial \alpha} \frac{\alpha}{y} \tag{31}$$

These can be a very valuable tool to investigate how and when parameters affects the solution for e.g. disease control. This subsection was only to give a brief understanding how to interpret the sensitivity coefficients. In many applications, they are used as a tool to find optimal parameters of a system.

3.4 Parameter Estimation

We now apply forward sensitivity analysis with forward automatic differentiation to estimate parameters for an ODE system.
If we can derive governing equations from either first principals or other reasoning, then we can infer the parameters. To do this, we will come up with an initial guess of the parameters and then measure the deviation of the simulated system to the measured system. Then we can calculate the gradient with respect to each of the parameters and iterate until we find a set of parameters within some margin of error.
We will use ℓ_2-loss and utilize forward mode automatic differentiation by use of dual numbers to automatically compute the gradients for each time we update the loss function. The specific implementation in Julia is shown in Listing 1.

Listing 1: The ℓ_2 norm implemented with dual numbers

```
function dual_l2norm(g::MultiDual{N,T}, α::Number) where {N,T}
    return MultiDual{N,T}(sqrt((g.val-α)^2), 2*(g.val-α)*g.derivs)
end
```

To show a specific use case, we will setup the rest of the framework and do parameter estimation on a simulated example. The parameters and code

we used to simulate the data can be seen in listing 2. We make use of the introduced DOPRI54 method from section 2.

Listing 2: Code to generate simulated data

```
p_y = (1.5,1.0,3.0,1.0)
u0 = @SVector[1.0, 1.0]
resolution = 4
y = Vector{typeof(u0)}(undef,resolution*10)
@inbounds y[1] = u0
@inbounds for i in 1:(resolution*10)-1
  y[i+1] = y[i]+(1/resolution)*Lotka_Volterra(y[i],p_y)
end
```

With the simulated data, we will now try to estimate the parameters using this new framework. We initialize the parameters at $[\alpha = 1.2, \beta = 0.8, \gamma = 2.8, \delta = 0.8]$. We then do 200 iterations to improve the parameters

Listing 3: The framework for estimating parameters

```
function l2Loss(y,p,u0,f,x,n,dt)
    @inbounds x[1] = u0
    @inbounds for i in 1:(n)-1
      x[i+1] = y[i]+dt*f(y[i],p)
    end
    loss = [[dual_l2norm(x[i][j],y[i][j])
                for j in 1:length(x[i])] for i in 1:length(x)]
    run = loss[1]
    for i in 2:n
        run += loss[i]
    end
    return(sum(run))
end

function estParam(f, loss, y, p, u0, n, dt, lambda, iter, plotting=true)
    x = Vector{typeof(u0)}(undef,n)

    valse = [0.0 for i in 1:(iter)]
    for k in 1:iter
        loss_i = loss(y,p,u0,f,x,n,dt)
        valse[k] = loss_i.val
        p = p - lambda * loss_i.derivs
    end
    if plotting
        display(plot(valse, title = "Loss per iterration",
                ylabel = "L2-loss", xlabel = "Iterrations", legend = false))
    end
    return p

end
# initial parameter guess and u0
(p1_s, p2_s, p3_s, p4_s) = (1.2,0.8,2.8,0.8)
dp1_s = MultiDual(p1_s, SVector(1.0, 0.0, 0.0, 0.0))
dp2_s = MultiDual(p2_s, SVector(0.0, 1.0, 0.0, 0.0))
```

```
dp3_s = MultiDual(p3_s, SVector(0.0, 0.0, 1.0, 0.0))
dp4_s = MultiDual(p4_s, SVector(0.0, 0.0, 0.0, 1.0))
p_s = [dp1_s,dp2_s,dp3_s,dp4_s]
u0 = @SVector[MultiDual(1.0, SVector(0.0, 0.0, 0.0, 0.0)),
        MultiDual(1.0, SVector(0.0, 0.0, 0.0, 0.0))]

p_out = estParam(Lotka_Volterra, l2Loss, y, p_s, u0, resolution*10,
        1/resolution, 0.1, 200)
```

This gives the parameter estimates $[\alpha = 1.5, \beta = 1, \gamma = 3, \delta = 1]$ which is exactly the parameters originally used to simulate the data. This can also be seen from the L_2-loss for each iterate in the figure 6.

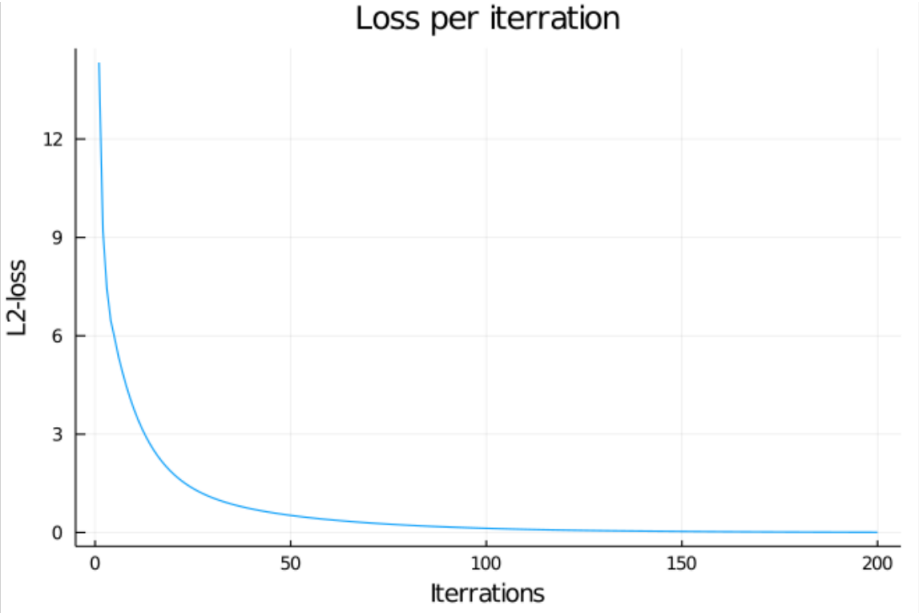


Figure 6: L_2-loss for each iteration of the parameter estimation

The presented method showcases how one can use forward sensitivity analysis to find the correct parameters of some system. To find optimal parameters for more advanced systems one should consider the global optimizer package which is used for the Lotka-Volterra in this blog.

This is a great tool if one has an idea of the equations that could govern the distribution from which data was generated. However, in many applications we have no idea and this is where the neural network comes into the picture. These heavily rely on the strong method of reverse automatic differentiation which is introduced in the next section.

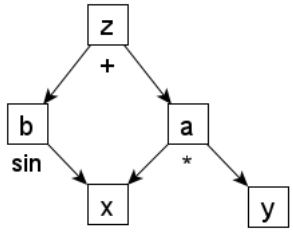
4 Reverse Automatic Differentiation

The following is inspired by [3] and we will start with the simple example below in a forward-mode AD setting.

$$z = xy + \sin(x) \tag{32}$$

If we want to calculate the derivative of z w.r.t. x and y, we would have to run the "program" 32 twice. Once with $\mathrm{d}x = 1, \mathrm{d}y = 0$ and once with $\mathrm{d}x = 0, \mathrm{d}y = 1$. We see that in a forward-mode AD framework we would have to run our program of interest once for every parameter we would like the derivative of w.r.t. the outputs of the program. More formally our complexity grows $\mathcal{O}(m)$ where m is the number of input variables. This is definitely not viable for something like a neural network where we have many input variables and parameters we want to optimize regarding to the output of some loss function.

Therefore, we need some other way to deal with this problem. Lets look at the computational graph of 32.



And then write up the forward-mode AD approach to calculate $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$, where t is the input variable we differentiate w.r.t.

$$\begin{aligned}
\tfrac{\partial x}{\partial t} &= ? \\
\tfrac{\partial y}{\partial t} &= ? \\
\tfrac{\partial a}{\partial t} &= y \cdot \tfrac{\partial x}{\partial t} + x \cdot \tfrac{\partial y}{\partial t} \\
\tfrac{\partial b}{\partial t} &= \cos(x) \cdot \tfrac{\partial x}{\partial t} \\
\tfrac{\partial z}{\partial t} &= \tfrac{\partial a}{\partial t} + \tfrac{\partial b}{\partial t}
\end{aligned} \tag{33}$$

which can be written up more generally as

$$\begin{aligned}
\frac{\partial w}{\partial t} &= \sum_i \left(\frac{\partial w}{\partial u_i} \cdot \frac{\partial u_i}{\partial t} \right) \\
&= \frac{\partial w}{\partial u_1} \cdot \frac{\partial u_1}{\partial t} + \frac{\partial w}{\partial u_2} \cdot \frac{\partial u_2}{\partial t} + \cdots
\end{aligned} \tag{34}$$

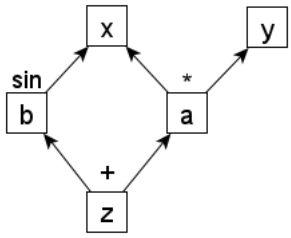
where w is some output variable of the program and the u_i's are input variables w depends on. We can now obtain either $\frac{\partial z}{\partial x}$ or $\frac{\partial z}{\partial y}$ by setting t to either x or y.
The key observation here, which will lead to reverse AD, is that we could have multiple outputs for which we could calculate the derivative w.r.t. t without reseeding the program. This property we can reverse by the symmetry of the chain rule. We write up 34 in reverse.

$$\begin{aligned}\frac{\partial s}{\partial u} &= \sum_i \left(\frac{\partial w_i}{\partial u} \cdot \frac{\partial s}{\partial w_i} \right) \\ &= \frac{\partial w_1}{\partial u} \cdot \frac{\partial s}{\partial w_1} + \frac{\partial w_2}{\partial u} \cdot \frac{\partial s}{\partial w_2} + \cdots\end{aligned} \tag{35}$$

Here u is some input, w_i is the outputs u depends on and s is now the output variable we want to differentiate w.r.t. all the inputs.
We can now rewrite 33 by use of the reverse chain rule.

$$\begin{aligned}\frac{\partial s}{\partial z} &= ? \\ \frac{\partial s}{\partial b} &= \frac{\partial s}{\partial z} \\ \frac{\partial s}{\partial a} &= \frac{\partial s}{\partial z} \\ \frac{\partial s}{\partial y} &= x \cdot \frac{\partial s}{\partial a} \\ \frac{\partial s}{\partial x} &= y \cdot \frac{\partial s}{\partial a} + \cos(x) \cdot \frac{\partial s}{\partial b}\end{aligned} \tag{36}$$

To visualize this reverse dependency structure we can rewrite the computational graph.



Returning to 36 we see that if we set $s = z$ and set $\frac{\partial s}{\partial z} = 1$ we obtain $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ with a complexity of $\mathcal{O}(1)$ which is a huge improvement compared to FAD's $\mathcal{O}(m)$. Furthermore we observe that if we had multiple outputs as

$$\begin{cases} z = 2x + \sin(x) \\ v = 4x + \cos(x) \end{cases} \tag{37}$$

We would now have to run the program for every output to obtain the sensitivities of the all the inputs with w.r.t. to every output.
So the complexity of forward mode AD grows linearly with inputs and constant w.r.t. outputs while reverse AD is the other way around.

4.1 Implementation

From [4] and lecture 11 in [2] we know that one problem we have when doing reveres mode AD is implementation complexity. In forward mode AD, the differentiated program and the ordinary program shared the same program flow. In reverse automatic differentiation, the ordinary program still flows forward in the program while the differentiated program flows in reveres while still being dependent on all the ordinary calculations.
Therefore, one needs to forward ordinary computations and saving them doing so. Afterwards one can pull back the gradient w.r.t. the desired output.
To illustrate this, we will implement a reveres AD frame work based on the principle tracing which is used in many known libaries as PyThorch, Tensorflow and Theano.
Tracing builds upon a Wengert list which represents the computational graph of our program. With such a list we can build the differentiated version and pull back our gradients.
One problem of building this Wengert list is control flow in programs. The computational graph of a program with control flow can change depending on the specific input as shown in figure 7.

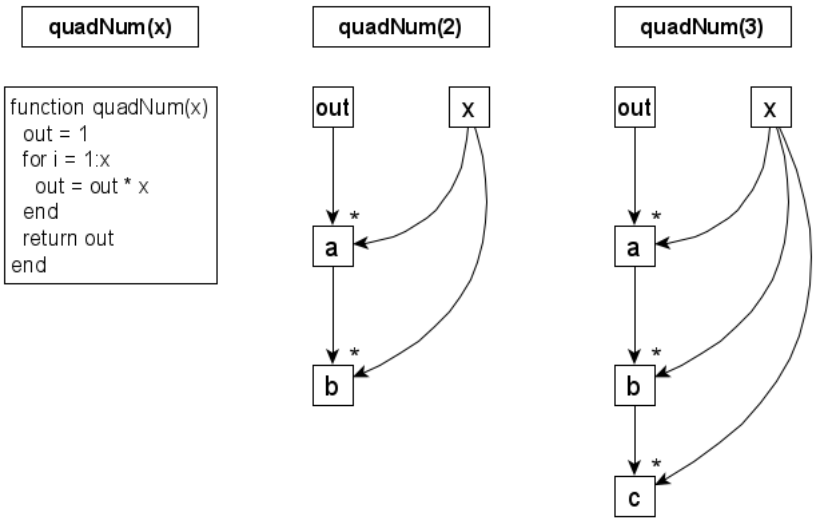


Figure 7: Input dependent computational graph

To solve this issue, we trace the specific inputs through our program and builds the Wengert list dynamically.

We have implemented 32 in the Julia code below.

Listing 4: Tracing framework for revers AD

```
# We define a variable
mutable struct Var
    value::Real
    grad::Any
    children::Vector{Tuple{Real,Var}}

    Var(val) = new(val, nothing, Vector{Tuple{Real,Var}}())
end

# Addition
function +(x::Var, y::Var)
    z = Var(x.value + y.value)
    append!(x.children, [(1,z)])
    append!(y.children, [(1,z)])
    return z
end

# Multiplication
function *(x::Var, y::Var)
    z = Var(x.value * y.value)
    append!(x.children, [(y.value,z)])
    append!(y.children, [(x.value,z)])
    return z
end

# Trigonometric functions
function sin(x::Var)
    z = Var(sin(x.value))
    append!(x.children, [(cos(x.value),z)])
    return z
end

# Reverse pass
function grad_value(x::Var)
    if x.grad == nothing
        x.grad = sum(w*grad_value(v) for (i,(w,v)) in enumerate(x.children))
    end
    return x.grad
end

x = Var(4)
y = Var(3)
z = x*y+sin(x)
z.grad = 1
```

We then pull back to obtain $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$

```
julia> grad_value(x)
2.346356379136388

julia> grad_value(y)
4
```

4.2 Source To Source reveres AD

When we compare the implementation using tracing to the implementation of forward AD by using of dual numbers, tracing is much slower. For the forward mode the compiler could do a full optimization on both the ordinary program and the differentiated one. This is not possible when doing tracing because traces though the same program can be different due to input dependent control flow. Therefore, we are not able to JIT-compile the pull back using this technique.
One way to handle this issue is to instead of building a differentiated computational graph we can differentiate the control flow graph (CFG), i.e. differentiate the source code itself. This is called source to source reveres AD and gives an input independent structure the compiler is able to optimize.

4.2.1 Zygote.jl

The specific implementation of this technique in Julia is called Zygote and is described in the paper [5]. It works by utilizing the LLVM compiler framework in Julia
LLVM has a front end, a middel end, and a back end as shown in figure 8.

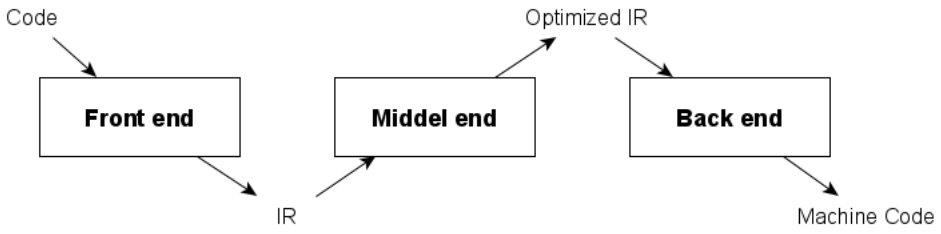


Figure 8: The LLVM frame work

The intermediate representation (IR) is a CFG and in LLVM based on static single assignments (SSA). This form allows for the middel end to optimize and analyze the source code.
In the paper behind Zygote, they come up with a framework to transform this IR in SSA-form into the differentiated source code.

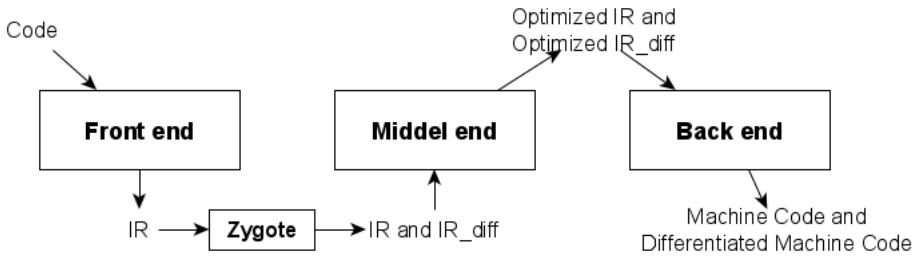


Figure 9: The Zygote frame work

As illustrated in figure 9 this now gives us a fully optimized ordinary program and differentiated program.

5 Neural Networks

One use case of reveres AD is in neural networks. The famous work horse behind the success of neural networks is the algorithm backpropagation which is just another name for reverse AD. We will here explore neural networks where most information is from chapter 2 in [6].

5.1 Architecture

A neural network is build of computation nodes connected by edges with an associated weight or bias as shown below.

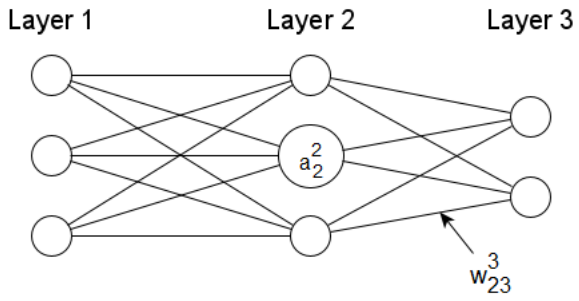


Figure 10: A basic neural network architecture

Each weight associated with an edge is indexed by w_{jk}^l. The weight w_{jk}^l is going from node k in layer $l-1$ to node j in layer l. Biases b_j^l is going to node j in layer l.
Furthermore, we have a nonlinear activation node a_j^l for which the indexing follows the same convention as for the biases. These nonlinear activation functions are defined as follows.

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{38}$$

Where σ is the nonlinear function. These nonlinearities are the reason behind neural network's nonlinear predictive abilities.
We now assume σ works element-wise such that 38 simplifies to

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right) \tag{39}$$

To evaluate the predictive performance of a neural network, we apply what is known as a cost function C. It takes as input a target y and the networks prediction a^L or $\hat{y}$. The cost function then outputs a real number which describes the discrepancy between a^L and y. If one is doing regression one choice of cost function is the L_2-loss.

$$C = \frac{1}{2N} \sum_{n=1}^{N} \left\| y(x_n) - a^L(x_n) \right\|_2 \tag{40}$$

where x is the training data and n is the number of training data. If the task is classification a typical choice is binary cross entropy.

$$C = -\frac{1}{N} \sum_{n=1}^{N} \left[y(x_n) \log a^L(x_n) + (1 - y(x_n)) \log \left(1 - a^L(x_n)\right) \right] \tag{41}$$

5.2 Backpropagation

To be able to adjust the weights and biases of the network w.r.t. to C, we need the derivatives $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$. To get these we will use reverse AD or backpropagation.
Instead of deriving $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$ directly, we will define the argument of the activation function a as

$$z^l = w^l a^{l-1} + b^l \tag{42}$$

We will then start by deriving the last layer in the network, $\frac{\partial C}{\partial z^L_j}$.

$$\begin{aligned} C(y_j, a^L_j) &= (y_j - a^L_j(z^L_j))^2 \\ &= (y_j - \sigma(z^L_j))^2 \end{aligned} \tag{43}$$

So by the chain rule we get

$$\frac{\partial C}{\partial z^L_j} = \sum_k \frac{\partial C}{\partial a^L_k} \frac{\partial a^L_k}{\partial z^L_j} \tag{44}$$

We know from the architecture that a^L_k only depends on z^L_j when $j = k$ so all other terms go to zero leaving us.

$$\begin{aligned} \frac{\partial C}{\partial z^L_j} &= \frac{\partial C}{\partial a^L_j} \frac{\partial a^L_j}{\partial z^L_j} \\ &= \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j) \end{aligned} \tag{45}$$

We will simplify the notation for for this derivative as

$$\frac{\partial C}{\partial z_j^L} = \delta_j^L \tag{46}$$

Further, we can generalize the derivative to a layer wide derivative by the elementwise multiplication operator $\odot$ called the Hadamard or Schur operator.

$$\begin{aligned}\delta^L &= \nabla_a C \odot \sigma'(z^L) \\ &= (a^L - y) \odot \sigma'(z^L)\end{aligned} \tag{47}$$

Next we need to derive the same derivative but for all the intermediate layers $\frac{\partial C}{\partial z_j^l}$.

$$\begin{aligned}\delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}\end{aligned} \tag{48}$$

We determine $\frac{\partial z_k^{l+1}}{\partial z_j^l}$

$$\begin{aligned}z_k^{l+1} &= \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} \\ &= \sum_j w_{kj}^{l+1} \sigma\left(z_j^l\right) + b_k^{l+1}\end{aligned} \tag{49}$$

We differentiate w.r.t. z_j^l

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \tag{50}$$

We substitute back into 48 and get

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{51}$$

Again we can generalize to a layer wide derivative by use of the Hadamard operator.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{52}$$

Now we know all the derivatives for each node in the network but what we really want is $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$. So we will use the chain rule on the δ^l's. We start with the weights.

$$\begin{aligned} \frac{\partial C}{\partial w^l_{jk}} &= \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} \\ &= \delta^l_j \frac{\partial z^l_j}{\partial w^l_{jk}} \end{aligned} \tag{53}$$

z^l_j is.

$$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j \tag{54}$$

So $\frac{\partial z^l_j}{\partial w^l_{jk}}$ becomes.

$$\frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k \tag{55}$$

We substitute back into 53

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{56}$$

Similarly for the biases we have

$$\begin{aligned} \frac{\partial C}{\partial b^l_j} &= \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial b^l_j} \\ &= \delta^l_j \frac{\partial z^l_j}{\partial b^l_j} \end{aligned} \tag{57}$$

from 54 we derive that

$$\frac{\partial z^l_j}{\partial b^l_j} = 1 \tag{58}$$

So

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{59}$$

We have now derived the structure of a neural network and all its derivatives. We will now proceed with the implementation of a general neural network.

5.3 Implementation

We will now implement a neural network and the associated backpopagation algorithm. Our implementation is inspired by this implementation where the backpopagation algorithm is implemented by use of static tracing as decribed in the previous section. If one is only interested in efficiency one should go with the native Julia library Flux for machine learning. It makes use of Zygote for the backpopagation which ensures a very fast implementation.

We start by implementing the data structure representing the single layers.

Listing 5: Layer structure

```
struct Layer{WS, BS, Z, F}
    W::WS # Weights
    b::BS # Biases
    z::Z  # intermediate state, w*a+b
    σ::F  # Non linear activation
end

Layer(in::Int, out::Int, σ::Function) =
    Layer(rand(Float32, out, in) .- 0.5f0, #Initialize the layer
          zeros(Float32, out),
          Array{Float32}[],
          σ)

function (l::Layer)(a) # propagate forward through the layer
    W, b, z, σ = l.W, l.b, l.z, l.σ
    temp = W * a .+ b
    empty!(z)
    push!(z, temp)
    return σ.(temp)
end
```

We can now define the single layers and popagate data through the layer. We will now define the network which combines the single layers into a neural network.

Listing 6: Neural network structure

```
struct Network{Layers, NodeValues} # Data structure of the whole network
    layers::Layers #Architecture of the network
    a::NodeValues #Values of every node in the network

    Network(layers...) = new{typeof(layers), Vector{Array{Float32}}}(layers, [])
end


function (m::Network)(X) # propagate forward through the whole network
```

```
    empty!(m.a)
    push!(m.a, X)

    for layer in m.layers
        push!(m.a, layer(m.a[end]))
    end

    return pop!(m.a) # return the values of the last nodes (aL)
end
```

We have now constructed a neural network. We can built a neural network and propagate data through as we do in 7 but this is not to much use right now. The network just does some random non-linear perturbation of the data and outputs a random number. Before the network is of any use for us we need to adjust all the weights and biases in the network to fit some data.

Listing 7: Definition of a neural network and a forward pass of data

```
sigmoid(x) = 1 / (1 + exp(-x)) # The sigmoid activation function

relu(x) = max(0, x) # The Relu activation function

neural_network = Network( # A 2-10-10-1 network
    Layer(2, 10, relu),
    Layer(10, 10, relu),
    Layer(10, 1, sigmoid))

x = [3,4]
neural_network(x) -> 0.36292547 #We propagate x=[3,4] through the network
```

We will therefore implement the backpropagation algorithm. To back-propagate the network we need to first differentiate the last layer as given in 47.

$$\begin{aligned}\frac{\partial C}{\partial z^L} &= \delta^L = \nabla_a C \odot \sigma'(z^L) \\ &= (a^L - y) \odot \sigma'(z^L)\end{aligned}$$

Next, we can differentiate all the following layers by using 52, 56 and 59.

$$\begin{aligned}\frac{\partial C}{\partial z^l} &= \delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \\ \frac{\partial C}{\partial w^l_{jk}} &= a^{l-1}_k \delta^l_j \\ \frac{\partial C}{\partial b^l_j} &= \delta^l_j\end{aligned}$$

We see from the above that we need to define the derivative of our cost

function, activation functions, and the derivatives of layers and weights. To do this we will overload a function derive.

Listing 8: Relevant derivatives

```
# Derivatives of layer, weight and biases
function derive(l::Layer, ∂Cost∂a_out, a_in)
    dσ = derive(l.σ)

    ∂Cost∂z = ∂Cost∂a_out .* dσ.(l.z[1])

    ∂W(∂Cost∂z, a_in) = ∂Cost∂z * a_in'

    ∂Cost∂W = sum(∂W.(eachcol(∂Cost∂z), eachcol(a_in)))

    ∂Cost∂b = sum(eachcol(∂Cost∂z))

    ∂Cost∂a_in = l.W' * ∂Cost∂z

    return ∂Cost∂W, ∂Cost∂b, ∂Cost∂a_in
end

# Derivatives of activation function
sigmoid(x) = 1 / (1 + exp(-x))
derive(::typeof(sigmoid)) = x -> sigmoid(x) * (1 - sigmoid(x))

relu(x) = max(0, x)
derive(::typeof(relu)) = x -> (x ≥ 0 ? 1 : 0)

# Derivative of cost function
bcentropy(ŷ, y) = -y* log(ŷ + 1e-7) - (1 - y) * log(1 - ŷ + 1e-7)
derive(::typeof(bcentropy)) = (ŷ, y) -> -y/(ŷ + 2e-7) + (1 - y)/(1 - ŷ + 1e-7)

Cost(Ŷ, Y) = sum(bcentropy.(Ŷ, Y)) / size(Y, 2)
derive(::typeof(Cost)) = (Ŷ, Y) -> derive(bcentropy).(Ŷ, Y) / size(Y, 2)
```

We see in the above that we have used the binary cross entropy because as an illustration we will do a simple classification. We can now define a function , backpopagte, to differentiate a layer and a whole network.

Listing 9: Backpopagation

```
function update!(l::Layer, dW, db, η)
    l.W .-= η * dW
    l.b .-= η * db
end

function back_propagate!(l::Layer, ∂Cost∂a_out, a_in, η)
    ∂Cost∂W, ∂Cost∂b, ∂Cost∂a_in = derive(l, ∂Cost∂a_out, a_in)
    update!(l, ∂Cost∂W, ∂Cost∂b, η)
    return ∂Cost∂a_in
end

function back_propagate!(m::Network, ∂Cost∂aL, η)
    ∂Cost∂a_out = ∂Cost∂aL
```

```
    for layer in reverse(m.layers)
        a_in = pop!(m.a)
        ∂Cost∂a_out = back_propagate!(layer, ∂Cost∂a_out, a_in, η)
    end
end
```

Lastly, we need a train function which stitch it all together

Listing 10: The train function

```
function train!(m::Network, Cost, dataset, η)
    costs = Float32[]
    dCost = derive(Cost)

    X, Y = dataset
    out = m(X)

    cost = Cost(out, Y)
    push!(costs, cost)

    ∂Cost∂out = dCost(out, Y)
    back_propagate!(m, ∂Cost∂out, η)

    return sum(costs) / length(dataset)
end
```

We will now train a neural network to classify data with the decision boundary given by $f(x) = x^3 - 3x + 2$. We then train the model for 5000 iterations and obtain the decision boundary given in figure 11

```
η = 0.05
iterations = 5000
for i in 1:iterations
    # Train model
    cost = train!(neural_network, Cost, dataset, η)
end
```

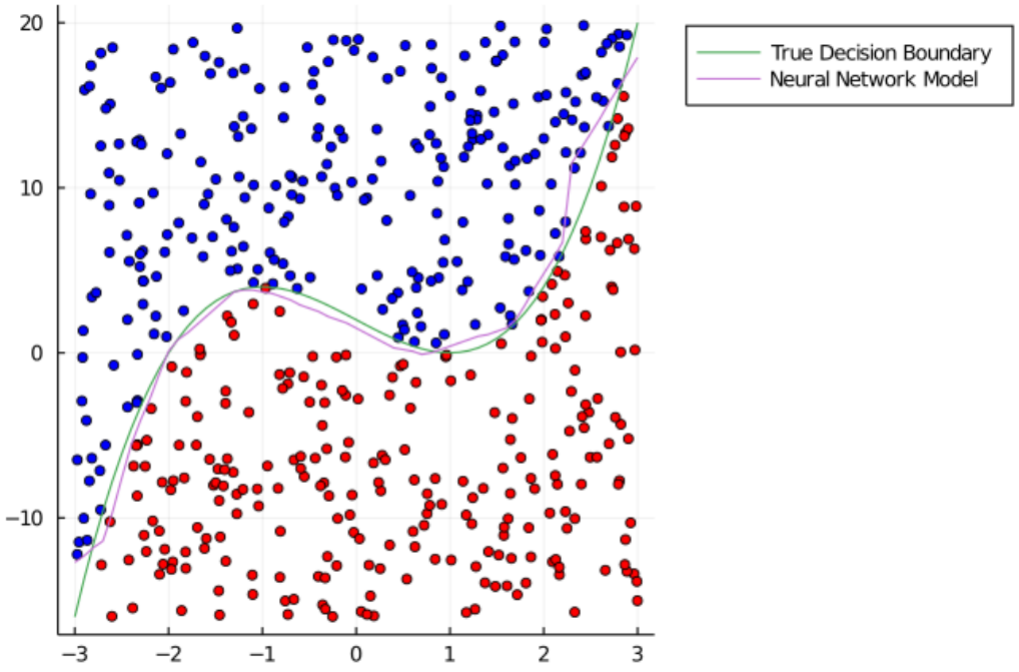


Figure 11: A decision boundary

6 Neural ODE and the adjoint method

We will now collect ideas from all of the previous sections and introduce a new machinery. Consider a differential equation $\mathrm{d}x/\mathrm{d}t = f$ and now let the function that proscribes the dynamics be a neural network

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \mathrm{NN}(x)$$

From the *universal approximation theorem* we know that there exist such a neural network that would be able represent any continuous function, f, to arbitrary precision. This opens up a world of data-driven dynamical systems where we can find any underlying function if we have enough data and assume that the data source is generated from some dynamical system.

The foundation is build on the framework of numerical integration using an ODE solver which calls the neural network at each necessary point in time. From section 2 and 5 we can draw important aspects to consider for this machinery to work. In figure 12, we see that our model does not interpolate all data points and the ODE solver seems to take large steps in time. From this we can derive that we have to make sure that

1. Make sure that the ODE solver uses sufficiently small steps sizes, Δt, when the systems exhibit rapid shifts.

2. The neural network should have parameters that minimizes the loss over all the available data points from $t = 0$ to $t = T$.

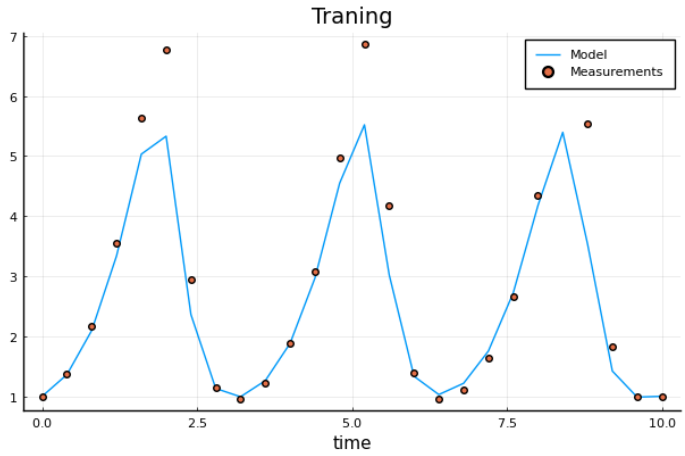


Figure 12: Training process with data points and a sub-optimal model

To meet the above requirements, the ODE solve have to do many function evaluation of the neural network for us to propagate precisely from $t = 0$ to $t = T$. At $t = T$, we have all the constituent parts to calculate the loss function. With the loss function, we can now adjust the parameters of our neural network with a backward pass as described in section 5. This is not trivial as the number of function calls from $t = 0$ to $t = T$ by the ODE solver is enormous. At each function call we do a forward pass which we have to store and back propagate from $t = T$ to $t = 0$. This computational graph is huge and would use all available memory quickly.

To make this machinery work, we use the adjoint method which rephrase the problem such that we essentially solve the ODE backwards in time and avoid the need to store huge intermediate computations. In the following we will introduce the adjoint method for ODEs and then extend the framework to neural ODEs.

6.1 Adjoint method for ODEs

Consider the general problem of ODE integration:

$$\begin{aligned} \underset{\theta}{\text{argmin}} \quad & L(x, \theta) \\ \text{subject to} \quad & \\ & F\left(\dot{u}(t), u(t), \theta, t\right) = \dot{u}(t) - f\left(u(t), \theta, t\right) = 0 \\ & u(t_0) = u_{t_0}, \qquad t_0 < t_1 \end{aligned} \tag{60}$$

- L is a cost function evaluated in the entire time-span $L(x,p) \equiv \int_{t_0}^{t_1} \ell(x,\theta,t)\mathrm{d}t$
- f is the neural network with parameters, θ
- $u(t)$ is the state at time $t \in [t_0, t_1]$ with initial condition $u(t_0)$
- $\dot{u}(t) = \mathrm{d}z/\mathrm{d}t$ is the time derivative at time t

the constraint $F\left(\dot{u}(t), u(t), \theta, t\right) = \dot{u}(t) - f\left(u(t), \theta, t\right) = 0$ is imposed to ensure that our ODE solver takes sufficiently small steps.

The aim is to calculate $\frac{\mathrm{d}\partial L(u)}{\mathrm{d}\theta}$ in a memory efficient way. The problem is obvious if we take the total derivative using the chain rule

$$\frac{\mathrm{d}L(u(\theta))}{\mathrm{d}\theta} = \int_{t_0}^{t_1} \left[\frac{\partial \ell}{\partial \theta}\frac{\mathrm{d}u}{\mathrm{d}\theta} + \frac{\partial \ell}{\partial \theta}\right] \mathrm{d}t \tag{61}$$

$\mathrm{d}u/\mathrm{d}\theta$ is the derivative of each variable with respect to each parameter, i.e., we calculate the Jacobian as in section 3. We do the computation for each state, u_k and $u_{k+\delta t}$, we reach with the ODE solver. These nested computations would quickly ramp-up memory when we solve $t = 0$ to $t = T$. We want to get rid of them.

6.2 The Lagrangian

We introduce the Lagrangian of the optimization problem

$$\begin{aligned} \mathcal{L} &\equiv L\left(u(\theta)\right) - \int_{t_0}^{t_1} \lambda(t) F\left(\dot{u}(t), u(t), \theta, t\right) \mathrm{d}t \\ &= L\left(u(\theta)\right) - \int_{t_0}^{t_1} \lambda(t) \left[\dot{u}(t) - f\left(u(t), \theta, t\right)\right] \mathrm{d}t \end{aligned} \tag{62}$$

where $\lambda(t)$ is a time dependent Lagrangian multiplier. Notice as $F = 0$ we actually have $\mathcal{L} = L$ and likewise $\mathrm{d}\mathcal{L}/\mathrm{d}\theta = \mathrm{d}L/\mathrm{d}\theta$ which is a property we will use below. *Note: if u is multi-dimensional and real, then one must use $\lambda^\intercal(t)$.*

To simplify notation, we omit writing explicit dependencies and consider only the last term in equations (62). We apply integration by parts

$$\begin{aligned}\int_{t_0}^{t_1} \lambda\left[\dot{u}-f\right]\mathrm{d}t &= \int_{t_0}^{t_1}\lambda\dot{u}\,\mathrm{d}t - \int_{t_0}^{t_1}\lambda f\mathrm{d}t \\ &= \lambda u|_{t_0}^{t_1} - \int_{t_0}^{t_1}\dot{\lambda}u\mathrm{d}t - \int_{t_0}^{t_1}\lambda f\mathrm{d}t \\ &= \lambda_{t_1}z_{t_1} - \lambda_{t_0}u_{t_0} - \int_{t_0}^{t_1}\left(\dot{\lambda}u+\lambda f\right)\mathrm{d}t\end{aligned}$$

We now take the total derivative of the above with respect to θ

$$\frac{\mathrm{d}}{\mathrm{d}\theta}\left[\int_{t_0}^{t_1}\lambda F\mathrm{d}t\right] = \frac{\mathrm{d}}{\mathrm{d}\theta}\lambda_{t_1}z_{t_1} - \int_{t_0}^{t_1}\frac{\mathrm{d}}{\mathrm{d}\theta}\left(\dot{\lambda}u+\lambda f\right)\mathrm{d}t \tag{63}$$

as the initial condition is independent of the choice of parameters then $\mathrm{d}/\mathrm{d}\theta u_{t_0}=0$. Apply the chain rule for $\mathrm{d}f/\mathrm{d}\theta$:

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}\theta}\left[\int_{t_0}^{t_1}\lambda F\mathrm{d}t\right] &= \frac{\mathrm{d}}{\mathrm{d}\theta}\lambda_{t_1}u_{t_1} - \int_{t_0}^{t_1}\frac{\mathrm{d}}{\mathrm{d}\theta}\left(\dot{\lambda}u+\lambda f\right)\mathrm{d}t \\ &= \frac{\mathrm{d}}{\mathrm{d}\theta}\lambda_{t_1}u_{t_1} - \int_{t_0}^{t_1}\frac{\mathrm{d}}{\mathrm{d}\theta}\dot{\lambda}u + \lambda\left(\frac{\partial f}{\partial u}\frac{\mathrm{d}u}{\mathrm{d}\theta}+\frac{\partial f}{\partial\theta}\right)\mathrm{d}t \\ &= \lambda_{t_1}\frac{\mathrm{d}u_{t_1}}{\mathrm{d}\theta} - \int_{t_0}^{t_1}\left(\dot{\lambda}+\lambda\frac{\partial f}{\partial u}\right)\frac{\mathrm{d}u}{\mathrm{d}\theta}\mathrm{d}t - \int_{t_0}^{t_1}\lambda\frac{\partial f}{\partial\theta}\mathrm{d}t\end{aligned} \tag{64}$$

we invoke equation (64) above and equation (61) to write the total derivative of the Lagrangian w.r.t. θ

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} =& \frac{\mathrm{d}L(u(\theta))}{\mathrm{d}\theta} - \frac{\mathrm{d}}{\mathrm{d}\theta}\left[\int_{t_0}^{t_1}\lambda F\mathrm{d}t\right] \\ =& \int_{t_0}^{t_1}\left[\frac{\partial\ell}{\partial\theta}\frac{\mathrm{d}u}{\mathrm{d}\theta}+\frac{\partial\ell}{\partial\theta}\right]\mathrm{d}t - \lambda_{t_1}\frac{\mathrm{d}u_{t_1}}{\mathrm{d}\theta} + \int_{t_0}^{t_1}\left(\dot{\lambda}+\lambda\frac{\partial f}{\partial u}\right)\frac{\mathrm{d}u}{\mathrm{d}\theta}\mathrm{d}t + \int_{t_0}^{t_1}\lambda\frac{\partial f}{\partial\theta}\mathrm{d}t \\ =& \int_{t_0}^{t_1}\left[\frac{\partial\ell}{\partial\theta}+\lambda\frac{\partial f}{\partial\theta}\right]\mathrm{d}t + \int_{t_0}^{t_1}\left(\frac{\partial\ell}{\partial\theta}+\dot{\lambda}+\lambda\frac{\partial f}{\partial u}\right)\frac{\mathrm{d}u}{\mathrm{d}\theta}\mathrm{d}t - \lambda_{t_1}\frac{\mathrm{d}u_{t_1}}{\mathrm{d}\theta}\end{aligned}$$

As mentioned before the computational expensive derivative to avoid is the Jacobian $\frac{\mathrm{d}u}{\mathrm{d}\theta}$. Therefore we require

$$\begin{aligned}\dot{\lambda} &= -\lambda\frac{\partial f}{\partial u} - \frac{\partial\ell}{\partial\theta} \\ \lambda(t_1) &= 0\end{aligned} \tag{65}$$

which is a new initial value problem. Notice we impose $\lambda(t_1) = 0$ to get rid of $\lambda_{t_1} \mathrm{d}u_{t_1}/\mathrm{d}\theta$. Further notice, that we will solve the above in reverse time which is a property we consider further below.
Now use the property $\mathrm{d}\mathcal{L}/\mathrm{d}\theta = \mathrm{d}L/\mathrm{d}\theta$ and equation (65) to obtain

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \frac{\mathrm{d}L}{\mathrm{d}\theta} = \int_{t_0}^{t_1} \left[\frac{\partial \ell}{\partial \theta} + \lambda \frac{\partial f}{\partial \theta}\right] \mathrm{d}t + \lambda_{t_0} \frac{\mathrm{d}}{\mathrm{d}u} L_{t_0} \tag{66}$$

notice we have found a way to avoid computing the Jacobian, $\frac{\mathrm{d}u}{\mathrm{d}\theta}$.
This is only possible if we solve the IVP in (65).

We can further leverage the similarities between (65) and (66). When we solve (65) by numerical integration, we can also solve $\mathrm{d}L/\mathrm{d}\theta$ one the fly. For each step we calculate λ_{t_k} and in the same time $\frac{\partial \ell}{\partial \theta} + \lambda_{t_k} \frac{\partial f}{\partial \theta}$ and do this for $t_k \in]t_0, t_1[$.

The major computationally deficiencies have been overcome. However, we also need to consider how to solve the ODE in reverse:

6.3 Concerns on ODE in reverse

We want to solve the ODE in reverse

$$\begin{aligned} \dot{\lambda} &= -\lambda \frac{\partial f}{\partial u} - \frac{\partial \ell}{\partial \theta} \\ \lambda(t_1) &= 0 \end{aligned}$$

but $\frac{\partial f}{\partial u}$ depends on $u(t)$. $u(t)$ is usually calculated during the forward pass but now we have 3 ways of calculating the above:

1. solve the ODE $\dot{u} = f(u, \theta, t)$ in reverse time. This is possible but as mentioned in [2] it is also numerically unstable. Further, there is no numerical guarantee that trajectories of u will be the same when calculated in reverse.

2. solve the forward ODE for $u(t)$ with high precision and interpolate each time $u(t_k)$ is needed. This is very fast, however, we again have to store the entire $u(t)$ solution.

3. during the forward pass save $u(t)$ finite number of times, $t = \{k_0, k_1, ...\}$. When we need $u(t)$ $t \in]k_n, k_{n+1}[$ during the backward pass, we resolve the ODE from $u(t_{k_n})$ to $u(t_{k_{n+1}})$. The idea is a trade of between computational and memory efficiency. The idea is illustrated in figure (13).

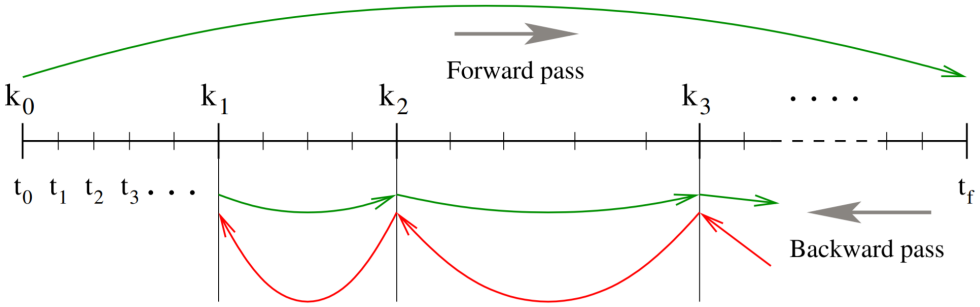


Figure 13: Find checkpoints $u(t)$ for $t \in \{k_0, k_1, ..\}$ during the forward. During the backward pass, resolve the ODE from $u(t_{k_n})$ to $u(t_{k_{n+1}})$

6.4 Data-driven loss function

Assume we have data $\mathcal{D}$ in some time interval $t \in [t_0, t_1]$. Let d_0 denote the data point at $t = t_0$, d_i denotes the i'th data point and d_N the data point at $t = t_1$. Let $u(t, \theta)$ be the NODE solution with parameters θ and use an ℓ_2-loss function:

$$\ell_2\text{-loss}(\theta) = \sum_{i=1}^{N} \|d_i - u(t_i, \theta)\|_2^2$$

The loss function need to be continuous for it to fit the framework of section 6.2. Here we utilize the properties of the Dirac delta function

$$L(x, p) = \int_0^T \sum_{i=1}^{N} \|d_{t_i} - x(t_i, p)\|_2^2 \, \delta(t_i - t) \mathrm{d}t \tag{67}$$

where

$$\delta(t) = \begin{cases} \infty, & t = 0, \\ 0, & \text{otherwise}, \end{cases} \tag{68}$$

one might recognize the trick above from the periodically kicked oscillator, otherwise Professor Gilbert Strang explains the concept in great detail.
For each d_i we can now calculate the contribution to the loss function

$$\frac{\mathrm{d}}{\mathrm{d}u} \ell\left(t_i, \theta\right) = 2\left(d_i - u\left(t_i, \theta\right)\right) \tag{69}$$

In practice, this means we only consider the loss function at discrete timepoints where data is available t_i. Between data points, we just solve the ODE $\dot{u}$.

6.5 Generalization and the adjoint method

In section 6.2, we introduced the Lagrangian multiplier, however, to simplify the notation we did only consider the scalar case. In A we do a more general introduction to the adjoint method for interested readers. An alternative derivation can be found in [7] and in appendix B of [8].

6.6 Implementation

We now consider how the forward and backward pass of the Neural ODE could be implemented in *Julia*. We will limit our attention and not cover how to implement a complete framework. This would require in-dept explanation of custom adjoints for Zygote and the contruction of a costume struct to represent the Neural ODE in a training framework in Flux. Interested readers can consult this GitHub repository. For simplicity, we will consider the second method introduced in section 6.3.

The forward pass of a neural network through an ODE solver can be described in pseudo-code as shown in algorithm 1

Algorithm 1: Forward pass of an Neural ODE

Input: $u(t_0), tspan, f, \theta$

$u(t_1) = \text{ODEsolve}(u(t_0), f, tspan, \theta)$;

return $u(t_1)$

where f is a neural network. The implementation in *Julia* is

```
function forward(u_t0, tspan, f, θ)

    function dudt(du, u, θ, t)
        du .= f(u, θ, t)
    end

    problem = ODEProblem(dudt, u_t0, tspan, θ)
    solution = solve(problem, DP5())

    u = solution.u # An array of x evaluated at each timestep

    return u
end
```

The backward pass After the forward pass, the computed trajectory u is used to calculate the loss function, L. Using the adjoint method, we

back-propagate the loss as described in appendix C in [8]. The algorithm is modified slightly to fit the notation used in section 6.1. At $t = t_1$ we compute $\mathrm{d}L/\mathrm{d}u_1$ and use u from the forward pass

Algorithm 2: Backward pass of an Neural ODE

Input: $u(t_0), tspan, f, \theta, \mathrm{dL}/\mathrm{d}u_{t_1}$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial u_{t_1}} \frac{\partial u_{t_1}}{\partial t_1} = \frac{\partial L}{\partial x(t_1)} f\left(u(t_1), t_1, \theta\right)$

$\lambda(t_1) = \begin{bmatrix} u(t_1) & \frac{\mathrm{d}L}{\mathrm{d}u_{t_1}} & 0_{|p|} & \frac{\partial L}{\partial t_1} \end{bmatrix}$

Function $\lambda\mathtt{dynamics}\left(\begin{bmatrix} u(t) & \lambda(t) & \cdot \end{bmatrix}, t, \theta\right)$**:**

  return $\begin{bmatrix} f(u(t), t, p) & \lambda(t)\frac{\partial f}{\partial u} & \lambda(t)\frac{\partial f}{\partial \theta} & \lambda(t)\frac{\partial f}{\partial t} \end{bmatrix}$

$\begin{bmatrix} x(t_0) & \frac{\partial f}{\partial x(t_0)} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t_0} \end{bmatrix} = \mathrm{ODEsolve}(\lambda(t_1), \lambda\mathrm{dynamics}, tspan, \theta)$

return $\frac{\partial f}{\partial x(t_0)}, \frac{\partial f}{\partial \theta}, \frac{\partial f}{\partial t_0}, \frac{\partial f}{\partial t_1}$

Algorithm 2 can be implemented in Julia as shown below

```
function backward(x, ∂L∂x_t1, θ, t0, t1, f)

    ∂L∂t1 = ∂L∂z_t1[:]' * f(x[end], θ, t1)[:]

    λ_t1 = ArrayPartition(∂L∂x_t1, zero(θ), [-∂L∂t1])

    function dλdt(dλ, λ, θ, t)

        _, back = Zygote.pullback(f, sol(t), θ, t)

        d = back(-λ.x[1])

        get_derivative(Δ, x) = (Δ == nothing ? zero(x) : Δ)
        Δλ = get_derivative.(d, dλ.x[:])

        for i in 1:3
            dλ.x[i] .= Δλ[i]
        end
    end

    problem = ODEProblem(dλdt, λ_t1, (t1, t0), θ)
    solution = solve(problem, DP5())
    λ_t0 = solution[end]

    return (λ_t0.x[1], λ_t0.x[2], -λ_t0.x[3][1], ∂L∂t1)
end
```

In practice, we will use the framework of [9]. Here we pass a neural network architecture, and ODE solver, a loss function and training data to a function that do all the underlying computations efficiently. In the section below, we will provide links to notebooks he have made using the framework of [9].

7 Application and further work

The sections above serve as a formal introduction to the concept of neural ordinary differential equations. To get intuitive understanding of the methods, application and properties, we think the format of notebooks and online web pages are superior. Below we will provide the links to *GitHub* repositories and HTML pages to explore:

Properties of NODE and introduction of ANODE In this notebook we show how we can use and setup the NODE in the framework from [9]. With and example we illustrate how the existence and uniqueness theorem also applies in the setting of neural ordinary differential equations. Then we move on an introduce the augmented neural ordinary differential equation introduced in [10]. Here extra dimensions are added to improve training and produce more stable solutions after training.

ANODE dynamics Here we investigate what the dynamics in the augmented NODE say about the system and properties of the ANODE. Here we use ideas from [11] and implement them in the framework of [9]. The agenda in the notebook is straight forward

1. simulate a system of 3 variables
2. construct a ANODE with one added dimension but train it only on two of the variables of the simulated system. In this way we can check if the ANODE can train without knowledge of the full system
3. we add one dimension to the NODE with our ANODE. Will this added dimension exhibit the dynamics of the variables that is was missing during training?
4. investigate stability in terms of switch in initial condition and simulations over longer periods

8 Related network constellations

In the following we will show how we can think NODE as a parametrization of hidden states of other network structures. We will mainly consider residual networks and recurrent neural networks.

8.1 Residual neural networks, ResNets

Consider a simple setup where we want to determine a flowmap from x_0 to x_1 in some time interval $t \in [0, T]$. Let $F : \mathbb{R}^d \mapsto \mathbb{R}^d$ be the flowmap s.t.

$$x_1 = F^t(x_0) \tag{70}$$

As with the Runge Kutta method of section 2, we can take a sequence of intermediate steps h_τ, $\tau \in [0, T]$ to get from x_0 to x_1 as seen in figure 14

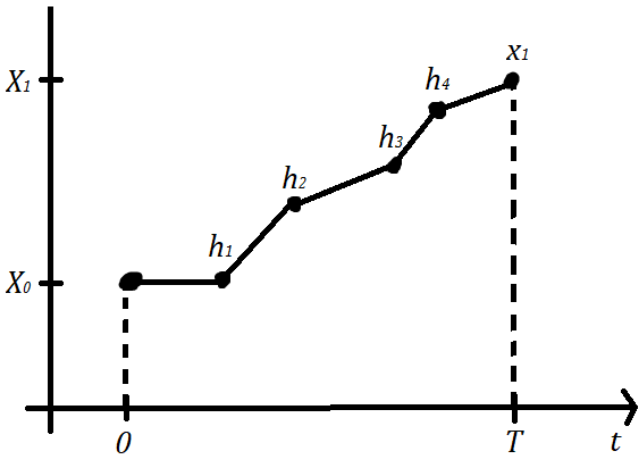


Figure 14: Flow map from x_0 to x_1 with hidden nodes h_τ

The intermediate steps h_τ are not explicit inputs and outputs of the network but assembles the values of the activations inside the neural network. These are very intuitive when we use a ResNet

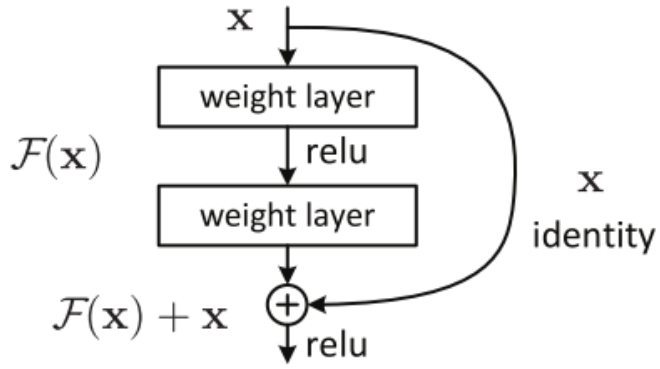


Figure 15: ResNet architecture (from figure 2, [12])

ResNet introduced in [12] is a neural network that tries to alleviate the problem of vanishing/exploding gradients when neural networks become to deep. To do that, identity mappings (see figure 15) are introduced that skip layers to more rapidly pass information through the network. Let all layers have the same dimension and let $h_\tau \in \mathbb{R}^n$ denote the activation in the layer at time τ, then we can describe the ResNet architecture as

$$h_{\tau+1} = \mathcal{F}(\tau, \theta_t) + h_\tau \tag{71}$$

where $\mathcal{F} : \mathbb{R}^d \mapsto \mathbb{R}^d$ is a continues function. The difference between the hidden states $h_{\tau+1} - h_\tau$ we can interpret as the discretisation of a continues transformation. If we add more layers, then the time difference between the layers, Δt, would decrease $\Delta t \to 0$. Let $\mathbf{h}_{t+\Delta t} - \mathbf{h}_t$ be the difference between two consecutive layers with time difference Δt then

$$\lim_{\Delta t \to 0} \frac{\mathbf{h}_{t+\Delta t} - \mathbf{h}_t}{\Delta t} = \frac{\mathrm{d}\mathbf{h}(t)}{\mathrm{d}t} = \mathcal{F}(\mathbf{h}(t), t) \tag{72}$$

i.e. we can parameterize the hidden states by an ODE and solve the IVP

$$\frac{\mathrm{d}\mathbf{h}(t)}{\mathrm{d}t} = \mathcal{F}(\mathbf{h}(t), t), \qquad h(0) = x_0 \tag{73}$$

and we want to find $\mathcal{F}$ s.t. $h(T) = x_1$. This is exactly the definition of the problem for a NODE between to points in time x_0 and x_1

To rephrase, with a ResNet we map x_0 to x_1 by a forward pass of a neural network through a discrete number of hidden states. In NODEs, we we map x_0 to x_1 with an ODE that starts at x_0 and our task is to find the function $\mathcal{F}$ that prescribes the right dynamics to get the correct continuous transformation of h in the time interval $t \in [0, T]$.

Graphical example The discrete transformations of hidden states for the ResNet and the continuous transformation of the NODE can graphically be seen in figure 16.

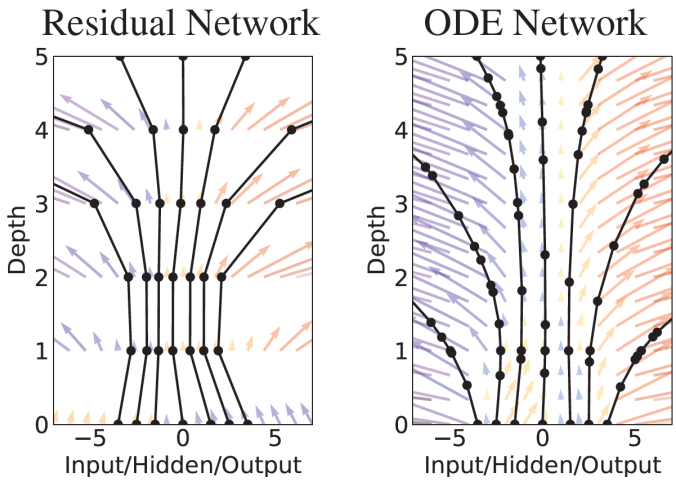


Figure 16: ResNet makes discrete steps and the NODE defines a vector field that continuously transform the input to the output (from figure 1, [8])

8.2 Recurrent neural networks, RNN

For a generic RNN we have an input series $x_t = \{x_0, x_1, ..., x_N\}$ with corresponding outputs $y_t = \{y_0, y_1, ..., y_N\}$ at equidistant timestamps in some time interval. We use the same neural network to map input to output and to carry the history of the series, we introduce a hidden state, h_k. In figure 17 we see how the information from the hidden state is passed on when we encounter new inputs. We can formulate this as a recurrence relation

$$y_k = F^\tau\left(x_k, h_{k-1}\right) \tag{74}$$

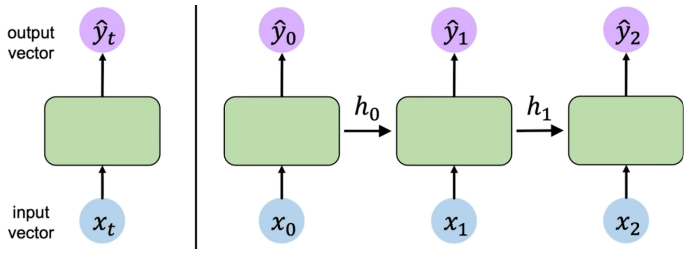


Figure 17: Two graphical representation of a recurrent neural network (from slide 15, lecture 2, [13])

For a dynamical systems, the outputs would often be the value at the next time stamp hence $x_t = \{x_0, x_1, ..., x_{N-1}\}$ and $y_t = \{x_1, x_2, ..., x_N\}$ hence the recurrence relation (74) would be a flow map

$$x_{k+1} = F^{\tau}(x_k, h_{k-1}) \tag{75}$$

We can again see the hidden state we pass on as the last point of a discrete sequence of finite transformations that maps x_k to x_{k+1}. Here the NODE would instead define a continuous transformation that maps x_k to x_{k+1} in the time-span Δt.

8.2.1 When to prefer NODE over RNN?

NODE has the obvious advantage that it is defined in all time points in $t \in [0, T]$ whereas RNNs are defined only at discrete time points and optimally at regularly-sampled time series data.

For training this means that NODEs can handle data points with varying Δt, whereas for RNNs must devide the time span into equidistant time steps and impute or aggregate data points [14] which potentially break up information kept in the data points. If data is sampled from **stiff systems**, then RNNs would be unable to handle that whereas the NODE with an adaptive ODE solver should handle this.

The adaptive NODE can become a problem if we want to simulate over long time spans and have enough training data. If at $t = 0$ and we want to determine $t = k$, $k >> 0$, then the NODE must take many intermediate steps bounded by the Δt allowance of the ODE solve. There could exist a flow map f s.t. $x_k = f^t(x_0)$ which we could learn using an RNN and then we avoid all the intermediate steps. The flow map could also be superior in physics at multiple scales as it would capture general structures and the NODE could encapsulate and struggle with neglectable minor structures.

References

[1] E. Hairer, S. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I Nonstiff problems*. Springer, first ed., 1993.

[2] C. Rackauckas, "18.337j/6.338j: Parallel computing and scientific machine learning," 2020.

[3] P. Ruffwind, "Reverse-mode automatic differentiation: a tutorial," 2016.

[4] M. J. Innes, "Diff zoo," 2020.

[5] M. J. Innes, "Sense sensitivities: The path to general-purpose algorithmic differentiation," 2020.

[6] M. Nielsen, *Neural Networks and Deep Learning*. 2019.

[7] V. Patel, "Deriving the adjoint equation for neural odes using lagrange multipliers," 2020.

[8] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," 2019.

[9] C. Rackauckas, Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and A. Ramadhan, "Universal differential equations for scientific machine learning," *arXiv preprint arXiv:2001.04385*, 2020.

[10] E. Dupont, A. Doucet, and Y. W. Teh, "Augmented neural odes," *arXiv preprint arXiv:1904.01681*, 2019.

[11] R. Strauss, "Augmenting neural differential equations to model unknown dynamical systems with incomplete state information," *arXiv preprint arXiv:2008.08226*, 2020.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[13] A. Amini and A. Soleimany, "Mit 6.s191: Introduction to deep learning," 2021.

[14] Y. Rubanova, R. T. Chen, and D. Duvenaud, "Latent odes for irregularly-sampled time series," *arXiv preprint arXiv:1907.03907*, 2019.

[15] S. G. Johnson, "Notes on adjoint methods for 18.336." Online at `https://github.com/mitmath/18335/blob/master/notes/adjoint/adjoint.p` Spring 2016, updated January 14, 2021.

[16] S. R. Garcia and R. A. Horn, *A second course in linear algebra*. Cambridge University Press, 2017.

Appendices

A The adjoint method

In the following we will show how the adjoint method is derived for general functions. The section is mostly based on [15].

First we need to know what the adjoint method is for some simple function and then we will extend the concept.

A.1 Derivation with links to report

Consider some state variables $x \in \mathbb{F}^{n_x}$ and parameters $p \in \mathbb{F}^{n_p}$ where denotes all $\mathbb{F}$ complex numbers. Define two functions, $g(x) : \mathbb{F}^{n_x} \times \mathbb{F}^{n_p} \to \mathbb{F}^{n_x}$ and $f(x) : \mathbb{F}^{n_p} \to \mathbb{F}$.

In this setup, $g(x, p)$ could be the constraint of section 6.2 that ensures the ODE is solution is computed correctly hence $g(x, p) = 0$. Likewise $f(x, p)$ could the be the real function that computes the loss with the data and parameters.

To minimize the error between our simulation and the data we will then try to find the parameters that minimizes f:

$$p_{\text{opt}} = \arg\min_{p} \ f(x)$$

To do that, we need to calculate $\frac{\partial f}{\partial p}$ as in section 6.2. In the following we denote $f_p = \frac{\partial f}{\partial p}$ likewise $f_x = \frac{f}{\partial x}$ now consider

$$\begin{aligned} f_p = f(x(p))_p) &= \frac{\partial f}{\partial x}\frac{\partial x}{\partial p} \\ &= f_x x_p \end{aligned} \tag{76}$$

Further, as $g(x, p) = 0$ everywhere, then we have $g_p = 0$ hence

$$g_p = g_x x_p + g_p = 0$$

Then we can solve for x_p to obtain; $x_p = -g_x^{-1} g_p$. We can then substitute this into equation 76 and we get:

$$f_p = -f_x g_x^{-1} g_p$$

lets consider each of the elements above.

- f_x: is a $1 \times n_x$ row-vector:

$$\begin{bmatrix} \frac{\partial f}{x_1} & \frac{\partial f}{x_2} & \cdots & \frac{\partial f}{x_n} \end{bmatrix}$$

- g_x: is the $n_x \times n_x$ Jacobian matrix

$$\begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_{n_p}} \\ \frac{\partial g_2}{\partial x_1} & & \vdots \\ \vdots & \ddots & \\ \frac{\partial g_{n_x}}{\partial x_1} & \cdots & \frac{\partial g_{n_x}}{\partial x_{n_x}} \end{pmatrix}$$

- g_p is a $n_x \times n_p$:

$$\begin{pmatrix} \frac{\partial g_1}{\partial p_1} & \vdots & \frac{\partial g_1}{\partial p_{n_p}} \\ \frac{\partial g_2}{\partial p_1} & & \vdots \\ \vdots & \ddots & \\ \frac{\partial g_{n_x}}{\partial p_1} & \cdots & \frac{\partial g_{n_x}}{\partial p_{n_p}} \end{pmatrix}$$

It would be very expensive to do the calculation as:

$$f_p = -f_x \left(g_x^{-1} g_p \right)$$

expecially because we have to save a $n_x \times n_p$ matrix from $g_x^{-1} g_p$. Instead we calculate:

$$f_p = \left(-f_x g_x^{-1} \right) g_p$$

and see that that $-f_x g_x^{-1}$ is actually a solution to the linear systems:

$$g_x^* \lambda = -f_x^*$$

Notice the conjugate transpose because f_x is a row-vector and g_x is $n_x \times n_x$. We introduce these s.t. we can directly plug back in

$$f_p = \lambda^* g_p$$

We just used the adjont method to find the particular function of interest. The following section is readers that come from linear algebra and might have encountered the adjoint method there.

A.2 Adjoint methods derived from inner products in linear algebra

We first review how we define the standard inner product with the notation from [16]. Consider $\mathcal{V} = \mathbb{F}^n$ as a vectorspace over $\mathbb{F}$. Then define two vectors $\mathbf{u} = [u_i]$, $\mathbf{v} = [v_i] \in \mathcal{V}$. We define the standard innerproduct as:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{v}^* \mathbf{u} = \sum_{i=1}^{n} u_i \bar{v_i}$$

Here $\mathbf{v}^*$ is the conjugate transpose of $\mathbf{v}$. The equation above might be more familiar in the case where $\mathbb{F} = \mathbb{R}$ and $n = 2$ where we have $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u} \cdot \mathbf{v}$ is the dot product in $\mathbb{R}^2$. For further introduction, consult [16] p. 88.
Let $T_A : \mathcal{V} \to \mathcal{V}$ be a linear transformation induced by a matrix $A \in M(\mathbb{F})_n$ s.t. $T_A \mathbf{u} = A\mathbf{u}$. Consider

$$\langle A\mathbf{u}, \mathbf{v} \rangle = \mathbf{v}^* (A\mathbf{u}) = (A^* \mathbf{v})^* \, \mathbf{u} = \langle \mathbf{u}, A^* \mathbf{v} \rangle \tag{77}$$

then we define T_A^* as the adjoint of T_A. Therefore we see that the adjoint of the matrix A is directly equal to the conjugate transpose. It is this property we use to solve our original problem in terms of the adjoint matrix to simply computation. Explicitly, we can now review the equation

$$f_p = -f_x g_x^{-1} g_p = \left\langle g_x^{-1} g_p, -f_x^* \right\rangle$$

g_x^{-1} is a $n_x \times n_x$ and then we can define $T_{g_x^{-1}} : \mathcal{V} \to \mathcal{V}$ as the linear transform induced by g_x^{-1}. Then as with equation (77) there exist an adjoint matrix $(g_x^{-1})^*$ s.t.

$$\left\langle g_x^{-1} g_p, -f_x^* \right\rangle = -f_x \left(g_x^{-1} g_p \right) = \left[\left(-g_x^{-1} \right)^* f_x^* \right]^* g_p = \left\langle g_p, \left(-g_x^{-1} \right)^* f_x^* \right\rangle$$

Computationally it is much smarter to work with the adjoint transformation induced $(g_x^{-1})^*$ as we can solve the equation without allocation of memory to store huge matrices. Instead solve for λ in

$$\left(-g_x^{-1} \right)^* \lambda = -f_x^* \tag{78}$$

and then substitute it back into the computation.
The equation (78) is the *adjoint equation*. The vector λ is called the vector of *adjoint varibles*.