INTRODUCTION TO THE DISCRETE ADJOINT METHOD AND NEURAL ORDINARY DIFFERENTIAL EQUATIONS

Mathematics Senior Seminar Paper

Garrett Gilliom
Tulane University
Department of Mathematics
Spring 2023

Contents

1 Abstract

A primary objective of applied mathematics is to represent the world by using equations that allow for the prediction, control of, and analysis of natural processes. With the advent of computers, large computational models long thought to be intractable may now accurately represent a variety of complex physical systems. Many of these models are based on differential equations where a function is related to its derivatives. However, finding solutions to these equations is often challenging, as real-world model equations may not have analytical solutions. Moreover, these models often require thousands of parameter values to be discovered before being deemed accurate. In this work, these challenges faced by models based on ordinary differential equations (ODEs) are reviewed and approaches to circumvent them are introduced, specifically the method of gradient descent and the discrete adjoint method. An example of these methods' application on a classification problem using neural networks by creating a neural ODE is also discussed.

2 Introduction and Motivation

2.1 Applied Mathematical Models

A primary objective of applied mathematics is to represent the world by using equations that allow for the prediction, control of, and analysis of natural processes. With the advent of computers, large computational models long thought to be intractable may now accurately represent a variety of complex physical systems. Many of these models are based on differential equations where a function is related to its derivatives.

The focus of this work will be on ordinary differential equations (ODEs) of the form

$$\frac{d}{dt}y(t) = y' = f(y), \tag{1}$$

which are commonly used within science and engineering to model real-world systems. Radioactive decay, natural population change, and economic growth are just a few of the wide range of systems that have corresponding equations to model them [1]. A key observation is that nearly all of these models rely on a set of parameters that depend on the specific details of the underlying system; such systems and parameters will be represented as

$$y' = f(y; \theta) \tag{2}$$

where θ represents a collection of parameters.

2.2 Applications of ODEs

Many ODE model parameters are determined from an understanding of what the model represents. For example, the ODE

$$y' = -ky \tag{3}$$

$$y(t_0) = y_0 \tag{4}$$

models radioactive decay of a substance. Here, parameter k represents its decay constant, which is dependent on its specific half-life. Given this parameter k and an initial substance amount y_0 at time t_0, the amount remaining at some future time t_{final} may be computed by solving this ODE.

However, the parameters of ODEs are not always measurable. In these cases, a data-driven approach is taken by collecting data relevant to those systems, such as population growth, financial trading, economic growth, or voting records, and then determining which parameter values best align the ODE model with those data. To compare how well a model aligns with, or "fits," a dataset, an error function $E(\theta)$ must be defined to quantify the model's inaccuracy based on parameters θ. There are many ways to define $E(\theta)$ and it is often done so on a case-by-case basis, depending on the type of model. Parameters θ^* may be described as "optimal" if $E(\theta^*)$ is minimized (i.e., there is no other θ that produces a lower error). To determine θ^*, a process called "parameter tuning" is undergone; there are many parameter tuning techniques, each of which has its own strengths and weaknesses.

2.3 Challenges Faced by ODE Models

Finding solutions to ODE models is often difficult. Many forms of ODEs exist – and only some of them are solvable by hand, meaning an analytical solution to the ODE exists. For example, some of the most simple ODEs may be solved by a technique specifically designed for so-called"Separable ODEs" [1]. However, many ODEs which are well-known to model and represent the real-world mathematically do not have analytical solutions. For example, the Lotka-Volterra equations that describe the predator-prey dynamics between two species' populations do not have an analytical solution [1]. For models and equations like these, numerical solution methods must be used to find an approximate solution across a specified time span, given an initial condition.

Many ODE models also contain multiple parameters θ (e.g., thousands) compared to the single parameter k in (3). To accurately model the physical system they are meant to represent, each parameter must be tuned to θ^* to minimize the error $E(\theta)$. Furthermore, many of these parameters θ may directly affect one another or drastically affect the solution curve in different ways, making it difficult to determine their values individually. Because computers are capable of performing computational tasks much faster than humans, parameter tuning techniques are programmed and ran by machines to optimize parameters θ and find θ^*.

3 Time Integration

3.1 Euler's Method Overview

Numerical methods that approximate a solution to an ODE at discrete time steps over a specified time interval are called time integrators. One of the simplest time integrators is Euler's method, an iterative method capable of producing approximate solutions of ODEs of the form

$$y' = f(y; \theta) \tag{5}$$

with initial condition

$$y(t_0) = y_0, \tag{6}$$

where θ is a list of parameters for $f(y)$. This method leverages what's known about the solution curve $y(t)$ based on what's given:

- The solution curve begins at point (t_0, y_0)
- The slope of the solution curve at time t_0 is $f(y_0; \theta)$

Using this information, an equation for the tangent line at time t_0 along solution curve $y(t)$ may be written as

$$y(t) = f(y_0; \theta) \cdot (t - t_0) + y_0. \tag{7}$$

An approximation for $y(t)$ may then be made at some other time $t = t_1$ such that $y(t_1) = y_1$. This new point (t_1, y_1) and slope $f(y_1; \theta)$ at t_1 may then be used to approximate y_2, which helps approximate y_3, and so on. As such, approximations along the solution curve $y(t)$ at discrete points of time t according to time-step size h may be found by following the recursive method

$$y_{n+1} = y_n + h \cdot f(y_n; \theta) \tag{8}$$

where $y_n = y(t_n)$ and $t_n = t_0 + n \cdot h$. This process also requires that values for parameters θ are provided, as they are necessary to compute a numerical value of $f(y_n; \theta)$.

3.2 Euler's Method Example and Implementation

Figure 1 illustrates an implementation of Euler's method in the Python programming language, where the ODE being approximated is

$$y' = f(y; \theta) = \theta y, \tag{9}$$

where $y(0) = 1$ and $\theta = 3$. Figure 2 shows a graph comparing its true solution,

$$y(t) = e^{3t}, \tag{10}$$

to an approximated solution produced by running Euler's method with $N = 100$ timesteps; increasing N would result in a more accurate approximation, as the time-step length between $t_0 = 0$ and $t_{final} = 1$ would shrink.

```python
### Euler's Method
def forward_solve_euler_method(y_0,f,N,T,theta):
    y_ns = [y_0] # initialize with initial value y_0
    h = T/N # timestep length
    # Calculate solutions
    for i in range(N):
        y_i_plus_1 = y_ns[i] + h * f(y_ns[i],theta)
        y_ns.append(y_i_plus_1)
    return y_ns
```

Figure 1: Euler's method in Python

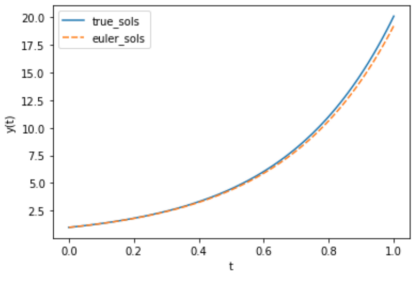

Figure 2: Graph comparing true to Euler's method solutions

4 Gradient Descent

4.1 Motivation and Overview

If the parameters of an ODE $y' = f(t;\theta)$ are chosen at random, the model will not be meaningful. The accuracy of how a chosen set of parameters for an ODE's solution compares to data may be measured by an error function $E(\theta)$. This error function requires a solution $y(t)$ to the ODE $y' = f(y;\theta)$ and parameters θ for said solution. It's necessary that $E(\theta)$ is defined such that a lower error value means that the input parameters θ produces a more accurate solution curve $y(t;\theta)$ than a higher error value. Given data in the form of $\{(\tau_k, \hat{y}_k)\}$ for $k = 1, \ldots, K$, the error function may be defined as

$$Error(y;\theta) = E(\theta) = \sum_{k=1}^{K} (y(\tau_k;\theta) - \hat{y}_k)^2. \tag{11}$$

With this, parameters θ^* are deemed optimal if they minimize $E(\theta)$.

Gradient descent is an iterative optimization algorithm that's meant to find parameters which minimize a differentiable function, such as θ^* for $E(\theta)$. This process requires the negative gradient of $E(\theta)$, or

$$\frac{\partial}{\partial\theta} E(\theta) = [\frac{\partial}{\partial\theta_i} E(\theta_i)]^T, \tag{12}$$

which represents the slope direction of the steepest descent for $E(\theta)$ with respect to each parameter θ_i. By multiplying this negative gradient by a small learning constant, α, and adding it to the previously used θ, successive small adjustments to parameters θ are made to push θ closer and closer to values θ^* that minimize $E(\theta)$. A single update of parameters θ across M iterations of gradient descent is done by repeatedly following the equation

$$\theta = \theta - \alpha \cdot \frac{\partial}{\partial\theta} E(\theta). \tag{13}$$

4.2 Process and Example

For ODEs with a simple analytical solution, calculating the gradient $\frac{\partial}{\partial\theta} E(\theta)$ necessary for gradient descent is a simple process. Consider again equation (9), namely

$$y' = \theta y \tag{14}$$

where $y(0) = 1$ and θ should be selected to best fit data $(\tau, \hat{y}) = (1, 3)$. This setup serves as a simple example to illustrate how gradient descent may be applied to find the specific value for θ which best matches our data. By method of separable ODEs, the solution $y(t;\theta)$ to $y' = \theta y$ where $y(0) = 1$ is found to be $y(t) = e^{\theta t}$ [1]. Thus, because we only have one data point at $\tau = 1$, the error function specified by (11) may be expressed as

$$E(\theta) = (y(1) - 3)^2 = (e^{\theta} - 3)^2. \tag{15}$$

As this ODE only consists of a single parameter θ, the gradient $\frac{\partial}{\partial\theta}E(\theta)$ may be computed as

$$\frac{\partial}{\partial\theta}E(\theta) = 2\cdot(e^{\theta}-3)\cdot e^{\theta} \tag{16}$$

following the chain rule of derivation.

From this, the following steps must be followed to apply the method of gradient descent to find the θ^* to best fit the solution curve $y(t) = e^{\theta t}$ to data $(1, 3)$:

(1) Select the number of iterations of gradient descent M, learning constant α, and initial parameter guess θ_0

(2) Evaluate the gradient of the error as $\frac{\partial}{\partial\theta}E(\theta) = 2\cdot(e^{\theta}-3)\cdot e^{\theta}$

(3) Substitute the selected α and θ_0 and found $\frac{\partial}{\partial\theta}E(\theta)$ into $\theta = \theta - \alpha\cdot\frac{\partial}{\partial\theta}E(\theta)$ to find a new θ

(4) Repeat steps (2) and (3) $M-1$ more times using the previous iteration's θ as the new θ guess

An implementation of gradient descent in Python for this specific problem may be found in Figure 3; a more generalized implementation may be found in Figure 4.

The Python function provided in Figure 3 may be run without needing more code to be written or functions to be defined, while the function in Figure 4 requires a `get_gradient(...)` function to compute the gradient $\frac{\partial}{\partial\theta}E(\theta)$. This distinction illustrates the massive difference in difficulty of performing gradient descent for an ODE with a simple analytical solution and few parameters relative to an ODE with no analytical solution and many parameters. For the former, the gradient can be explicitly evaluated by deriving $\frac{\partial}{\partial\theta}E(\theta)$ analytically, while this cannot be done for ODEs with only numerical solutions – and it's much more challenging for ODEs with many parameters. Because ODEs that model real-world physical systems are often complicated, consider many parameters, and rarely have analytical solutions, other non-analytical approaches to computing the gradient $E(\theta)$ such as the discrete adjoint method must be taken.

```python
import math
# Gradient descent for y'= theta * y, data={(1,3)}
def gradient_descent_specific(theta_init,M,alpha):
    theta = theta_init
    error_partial_theta = 0 # initialize gradient variable
    for i in range(M): # iterate M times
        error_partial_theta = (2 * (math.exp(theta)-3) * math.exp(theta))
        theta = theta - alpha * error_partial_theta
    return theta
```

Figure 3: Gradient descent (specific) in Python

```python
# Generalized gradient descent; may require more parameters
def gradient_descent_general(theta_init,M,alpha,data,y_0,...):
    theta = theta_init
    error_partial_theta = 0 # initialize gradient variable
    for i in range(M): # iterate M times
        #get_gradient(...) is a n arbitrary function
        error_partial_theta = get_gradient(theta,data,y_0,...)
        theta = theta - (alpha * error_partial_theta)
    return theta
```

Figure 4: Gradient descent (general) in Python

5 Discrete Adjoint Method

5.1 Defining an Optimal Control Problem

An optimal control problem aims to find parameters θ^* for an ODE $y' = f(y;\theta)$ that minimizes a cost function $E(y(\theta))$. Gradient descent may be used to iteratively solve an optimal control problem by repeatedly improving upon an initial guess for θ using the gradient $\frac{\partial E}{\partial \theta}$. More generally, the optimal control problem is an optimization problem where the goal is to minimize $E(y(\theta))$ subject to the constraint $y' = f(y;\theta)$.

Optimal control problems are common in science and engineering because ensuring efficiency is necessary in many fields. For example, it's important for a spacecraft to travel along an optimal flight trajectory to save time and avoid wasting fuel. Financial firms, on the other hand, may aim to develop optimal investment strategies to accrue large returns, edge-out competition, and circumvent losses. These scenarios illustrate just a few examples of when it's important for human-created systems to be optimal and efficient according to some quantifiable and measurable error.

In this section, a process for efficiently computing the gradient $\frac{\partial E}{\partial \theta}$ for gradient descent called the discrete adjoint method will be discussed. In 5.2 and 5.3, the method of Lagrange multipliers is introduced as an approach to solving optimization problems subject to constraints and casts it as an optimal control problem. The discrete adjoint method and its application on gradient descent are then covered in subsections 5.4 and 5.5.

5.2 Lagrange Multipliers

The Lagrange multiplier method is a strategy for finding local extrema for a function f subject to constraints g_j. Let X represent parameters $x_1, x_2, \ldots, x_m$. This problem may be expressed as:

$$\text{minimize } f(X) \tag{17}$$

$$\text{subject to } g_j(X) = 0, j = 1, \ldots, N. \tag{18}$$

This method requires the introduction of the Lagrange multiplier λ and its accompanying Lagrangian $\mathcal{L}(X, \lambda)$. In the case of a single constraint equation $g(X) = 0$, the Lagrangian is formulated as

$$\mathcal{L}(X, \lambda) = f(X) + \lambda \cdot g(X), \tag{19}$$

where only a single Lagrange multiplier λ is necessary. However, if the function $f(X)$ is being optimized according to multiple constraints, the Lagrangian is formulated as

$$\mathcal{L}(X, \lambda_1, \ldots, \lambda_N) = f(X) + \sum_{j=1}^{N} \lambda_j \cdot g_j(X). \tag{20}$$

Here, each of the N constraint equations $g_j(X)$ correspond to a single Lagrange multiplier λ_j, resulting in N Lagrange multipliers $\lambda_1, \ldots, \lambda_N$. Using this method, parameters X^* optimize the given function $f(X)$ based on the given constraints $g_j(X) = 0$ if all partial derivatives of L with respect to each parameter and Lagrange multiplier are equal to 0.

Stated explicitly, the optimal parameters satisfy

$$\frac{\partial \mathcal{L}}{\partial x_j} = 0 \text{ for } j = 1, \ldots, m \tag{21}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = 0 \text{ for } j = 1, \ldots, N. \tag{22}$$

5.3 Lagrange Multipliers for an Optimal Control Problem

The optimal control problem may be cast as a Lagrange multiplier problem if the continuous ODE is replaced by a discrete approximation computed using a time integrator such as Euler's method. In this scenario, the function being optimized is the error function, $E(\theta)$, and the constraints $g_j(X)$ are the steps of the time integrator. Using only the single point of data $(\tau_N, \hat{y})$ for the solution curve $y(t)$ to fit to, this problem is expressed as

$$\text{minimize } E(\theta) = (y_N - \hat{y})^2 \tag{23}$$

$$\text{subject to } 0 = y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta) \text{ for } j = 1, \ldots, N \tag{24}$$

where $y_N = y(\tau_N; \theta)$. (24) is equivalent to stating that each solution y_j is approximated using Euler's method as in (8). Thus, the Lagrangian may be rewritten as

$$\mathcal{L}(y_1, \ldots, y_N, \lambda_1, \ldots, \lambda_N, \theta) = (y_N - \hat{y})^2 + \sum_{j=1}^{N} \lambda_j \cdot (y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta)). \tag{25}$$

It should be noted that although y_0 and $\hat{y}$ are used in $\mathcal{L}$, they are given values for the problem and thus not changeable parameters.

Because the method of Lagrange multipliers requires that optimal parameters must make all partial derivatives of $\mathcal{L}$ equal to 0, the following conditions must be met for achieving the optimal parameters θ^*:

$$\frac{\partial \mathcal{L}}{\partial y_j} = 0 \tag{26}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = 0 \tag{27}$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = 0 \tag{28}$$

for $j = 1, \ldots, N$. As the evaluations of parameters $y_1, \ldots, y_N$ and $\lambda_1, \ldots, \lambda_N$ are both explicitly dependent on the choice of parameter θ for $y' = f(y; \theta)$, θ is the only true parameter which should be actively tuned, or changed.

In general, solving all three equations (26), (27), and (28) is not computationally tractable. The discrete adjoint method, which is presented in the next section, solves only (26) and (27) and then uses $\frac{\partial \mathcal{L}}{\partial \theta}$ to compute the gradient $\frac{\partial E}{\partial \theta}$. This gradient can then be used by gradient descent to find optimal parameters θ^*.

5.4 Discrete Adjoint Method

Of the conditions (26), (27), and (28), (26) and (27) may each be achieved by a straightforward iterative method, while finding θ which satisfies (28) is generally too difficult to solve for directly. It will be shown that guaranteeing (27) is equivalent to solving each y_j for $j = 1, \ldots, N$ using Euler's method and guaranteeing (26) gives an iterative method for computing each λ_j for $j = 1, \ldots, N$, while guaranteeing (28) will be accomplished by gradient descent.

Using the Lagrangian (25), it may be shown that

$$\frac{\partial \mathcal{L}}{\partial \lambda} = 0 \iff y_j = y_{j-1} + h \cdot f(y_{j-1}; \theta) \tag{29}$$

$$\frac{\partial \mathcal{L}}{\partial y_j} = 0 \iff \begin{cases} \lambda_N = 2 \cdot (\hat{y} - y_N) \text{ if } j = N \\ \lambda_{j-1} = \lambda_j + \lambda_j \cdot h \cdot \frac{\partial}{\partial y_j} f(y_j; \theta) \text{ if } j < N \end{cases}, \tag{30}$$

thereby satisfying (27) and (26), respectively; see Appendix A for details on the derivation. Because each λ_j is calculated descending from $N, \ldots, 1$ and λ_j requires y_j, it's clear that all approximate solutions y_j must first be computed forward to understand how θ affects the ODE's solution before computing all the Lagrange multipliers λ_j backward. A Python implementation of this process to find all the Lagrange multipliers given data $\hat{y}$, time-step size h, approximate solutions y_i, $\frac{\partial}{\partial y_j} f(y_j; \theta)$, and θ is shown in Figure 5.

```
# data is of form [(t,y_hat)]
# y_n is of form [(t,y_i)] where y_i is an approx. sol
# f_partial_y is the partial derivative of the function f w.r.t. y
def backward_solve_lagrange(data, y_ns, h, f_partial_y, theta):
    y_hat = data[0][1]
    y_N = y_ns[-1] # use last y_n as y_N to compute lambda_N
    lambda_N = 2 * (y_hat - y_N) ## Define lambda_N from error of (y_N - y^hat)^2
    lambda_ns = [lambda_N] ## Initialize lambda list
    y_ns_rev = y_ns[:-1][::-1] # Remove y_N; reverse y_ns to work backward
    for y_j in y_ns_rev:
        lambda_j = lambda_ns[0] # lambda_j is first element of list
        lambda_j_minus_1 = lambda_j + (lambda_j * h * f_partial_y(y_j,theta))
        lambda_ns = [lambda_j_minus_1] + lambda_ns # Add lambda_j_minus_1 to beginning of lambda_ns
    return lambda_ns
```

Figure 5: λ_j computation implementation in Python

The methods in (29) and (30) calculate each λ_j and y_j for $j = 1, \ldots, N$ such that conditions (27) and (26) are satisfied. Then, by computing $\frac{\partial \mathcal{L}}{\partial \theta}$ using λ_j and y_j from (30) and (29), $\frac{\partial E}{\partial \theta}$ may be computed directly with

$$\frac{\partial E}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta} = -\sum_{j=1}^{N} \lambda_j \cdot h \cdot \frac{\partial}{\partial \theta} f(y_{j-1}; \theta), \tag{31}$$

the details of which may be found in Appendix A and in reference [2].

5.5 Pairing Gradient Descent with the Discrete Adjoint Method

With this calculation of $\frac{\partial E}{\partial \theta}$ using the forward-solved y_j and backward-solved λ_j values and a given θ, a gradient for the error $E(\theta)$ has finally been found. This gradient can then be used to perform gradient descent M times using the formula provided in (13); a Python implementation of this may be found in Figure 4. It's important to note, however, that this straightforward approach to computing the $\frac{\partial E}{\partial \theta}$ at each step of gradient descent is only directly applicable to scenarios where the solution curve $y(t;\theta)$ is being fit to only a single point of data $(\tau, \hat{y})$ and there is only a single parameter θ. Increasing the number of data points to the ODE $y' = f(y;\theta)$ and its solution $y(t;\theta)$ changes the error function and will not be covered here. Additional complexities arise if the number of parameters is increased or if the dimension of the ODE is increased, too.

Finding the gradient $\frac{\partial E}{\partial \theta}$ using this process is called the discrete adjoint method, which is used to efficiently compute a cost gradient of a numerical solution to a model with respect to its parameters. Here, the model is the ODE $y' = f(y;\theta)$, the parameters are θ, the numerical solution is $y(t;\theta)$, and the cost and its respective gradient is $E(\theta)$ and $\frac{\partial E}{\partial \theta}$. To summarize, the following steps make up the discrete adjoint method to find $\frac{\partial E}{\partial \theta}$ for performing gradient descent to find θ^* which best matches solution $y(t;\theta)$ to data $(\tau, \hat{y})$:

1. Approximate solution curve $y(t;\theta)$ to obtain each y_j for $j = 1, \ldots, N$ given initial guess θ using Euler's Method; other time integrators may also be used
2. Find the Lagrange multipliers λ_j using which satisfy (30) the Lagrangian condition that $\frac{\partial \mathcal{L}}{\partial y_j} = 0$ for $j = 1, \ldots, N$ using an iterative method
3. Compute the gradient $\frac{\partial E}{\partial \theta}$ from $\frac{\partial E}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta}$ using the found y_j and λ_j values
4. Adjust θ according to $\frac{\partial E}{\partial \theta}$, learning step α, and the initial θ to perform a single iteration of gradient descent using (13)
5. Repeat M-1 times to determine a θ^* which best fits solution curve $y(t;\theta)$ to data $(\tau, \hat{y})$

6 Neural Ordinary Differential Equations

6.1 Classification Problem

Classification is an example of a common machine learning problem. A machine classifier typically completes a supervised training process where it learns to associate certain data characteristics with certain labels, thus "learning" how to label new instances of data. For example, given the colored data in Figure 6, the goal or objective of classification would be for the classifier to determine whether a data point is a part of the blue group or the orange group [3].

This classifier may be viewed as an arbitrary function $B(\mathcal{X})$ that takes in data of the form of $[x, y]^T$ and returns some 2-dimensional vector as output. Thus, black box solver B is a function that maps an input in $\mathbb{R}^2$ to an output in $\mathbb{R}^2$. A graphic showing the flow of input data to black box solver to output is in Figure 7. When the classifier makes a decision, the

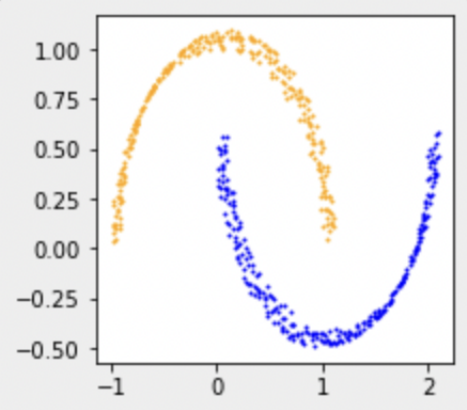

Figure 6: Starting data for an example classification problem
[3]

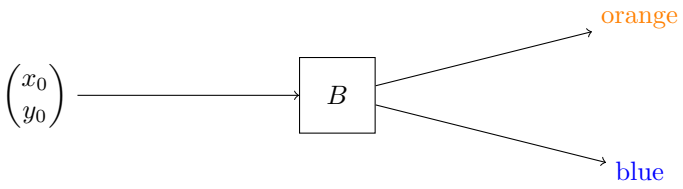

Figure 7: Black box solver function for classification of data

evaluation of $B(\mathcal{X})$ should to map all the blue points to a single location and all the orange points to a different location.

6.2 Classification Using an ODE

Consider the situation where the black box solver B is the solution to a general ODE whose initial value is the input. Because the input and output vectors in this example are 2-dimensional, the ODE itself will be a 2-dimensional system where solutions $x(t)$ and $y(t)$ make up the 2-dimensional vector $\mathcal{X}(t) = [x(t), y(t)]^T$. Therefore, the function B applied to the input $\mathcal{X}_0 = [x_0, y_0]^T$ produces

$$B(\mathcal{X}_0) = \mathcal{X}(t_{final}) \tag{32}$$

where $\mathcal{X}(t)$ is the solution to the general ODE

$$\mathcal{X}' = G(\mathcal{X}) \tag{33}$$

$$\mathcal{X}(t_0) = \mathcal{X}_0 \tag{34}$$

where input data $\mathcal{X}_0 = [x_0, y_0]^T$ is the initial condition. Note that $\mathcal{X}(t_{final})$ is the specific solution to (33) at time t_{final}. This means that the black box solver B solves (33) from time t_0 to time t_{final} using the input as the initial condition; it's common to let $t_0 = 0$.

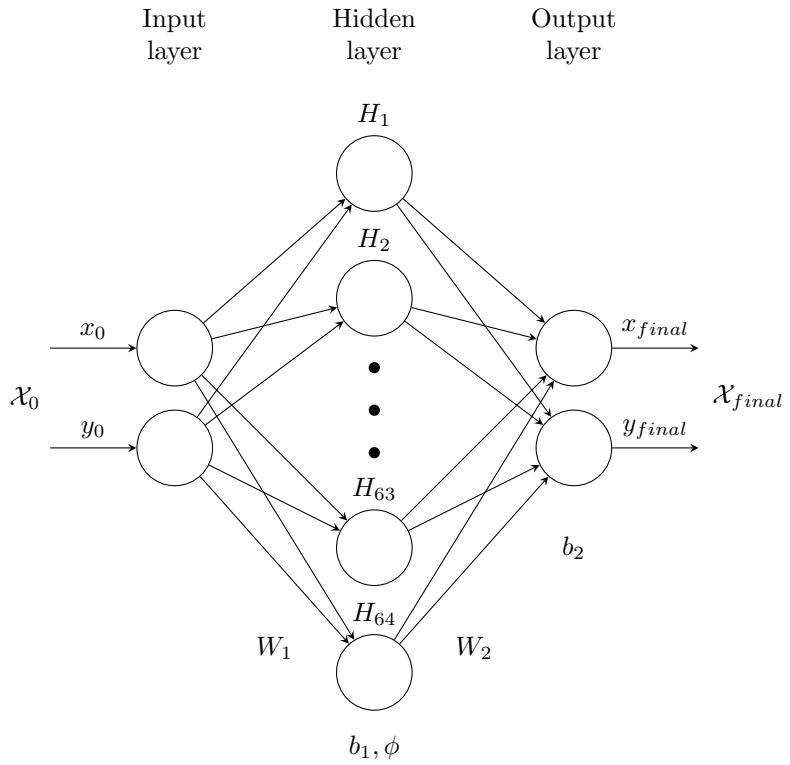

Figure 8: Torchdyn example's neural network illustration

6.3 Neural Ordinary Differential Equations

To determine what this black box solver B is, (33) must be solved, which requires a choice of what $G(\mathcal{X})$ should be. To obtain a neural ODE, the assignment

$$G(\mathcal{X}) = f(\mathcal{X}; \theta) \tag{35}$$

is made, where $f(\mathcal{X}; \theta)$ is a neural network that takes in $\mathcal{X}_0$ to input layer and outputs $\mathcal{X}_{final}$ at its output layer [4]. For this example, let $f(\mathcal{X}; \theta)$ be a neural network with a single hidden layer of 64 nodes; an illustration to depict this neural network is in Figure 8.

The neural network in Figure 8 acts as a function from input vector $\mathcal{X}_0 = [x_0, y_0]^T$ to output vector $\mathcal{X}_{final} = [x_{final}, y_{final}]^T$. This is done by applying a series of weights and biases with matrix arithmetic through the hidden and output layers, all of which will be denoted as parameters θ. An activation function is also applied at the hidden layer. This function may be represented analytically as

$$f(\mathcal{X}; \theta) = W_2 \phi(W_1 \mathcal{X} + b_1) + b_2 \tag{36}$$

where $\theta = [W_1, W_2, b_1, b_2]$. Here, input vector $\mathcal{X}$ is a 2×1 matrix, weight W_1 is a 64×2 matrix, bias b_1 is a 64×1 matrix, W_2 is a 2×64 matrix, and b_2 is a 2×1 matrix, thus its output is also a 2×1 matrix; the activation function $\phi(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is also applied pointwise to the vector $W_1 \mathcal{X} + b_1$. With equations (33), (34), and (35), the entire neural

ODE can be expressed as

$$\mathcal{X}' = W_2\phi(W_1\mathcal{X} + b_1) + b_2 \tag{37}$$

$$\mathcal{X}(0) = \mathcal{X}_0 \tag{38}$$

whose solution $\mathcal{X}(t;\theta)$ should map $\mathcal{X}_0$ to $\mathcal{X}_{final}$ to best classify the data in Figure 6.

To determine solution curve $\mathcal{X}(t;\theta)$ which best fits the data in Figure 6, parameters $\theta = [W_1, W_2, b_1, b_2]$ must be determined to minimize error between the labeled data and the output of the neural ODE given θ according to some chosen error function. Here, gradient descent may be used in conjunction with the discrete adjoint method to tune θ from an initial random guess θ_0 to optimal θ^* which minimizes error. The Cross-Entropy Loss function, which is an error function commonly used to train neural networks on labeled data, may also be used [5]. Once θ_0 has been chosen, the steps of the adjoint method may be followed to compute the gradient of the error function and perform gradient descent as in (13): Euler's method or another time integrator is used to approximate $\mathcal{X}(t;\theta)$ from t_0 to t_{final} to satisfy $\frac{\partial \mathcal{L}}{\partial \lambda_j} = 0$, the Lagrange multipliers λ_j are found to satisfy $\frac{\partial \mathcal{L}}{\partial y_j} = 0$, the gradient $\frac{\partial E}{\partial \theta}$ is found from $\frac{\partial \mathcal{L}}{\partial \theta}$, and then θ is adjusted according to that gradient and learning step size α. Repeat this M times to determine a θ^* which best fits solution curve $\mathcal{X}(t;\theta)$ to the data in Figure 6.

Figure 9 displays an example of using test data as input to the neural ODE after no training on initial random parameters θ_0; it's clear that the solution curves, or flows, of $\mathcal{X}(t)$ do not map each $\mathcal{X}_0$ to a $\mathcal{X}_{final}$ which easily distinguishes the two classes of the data. However, after training the neural ODE to find θ^*, the resulting flows and their end points at time t_{final} make classifying the two types of data much simpler; Figure 10 illustrates this.

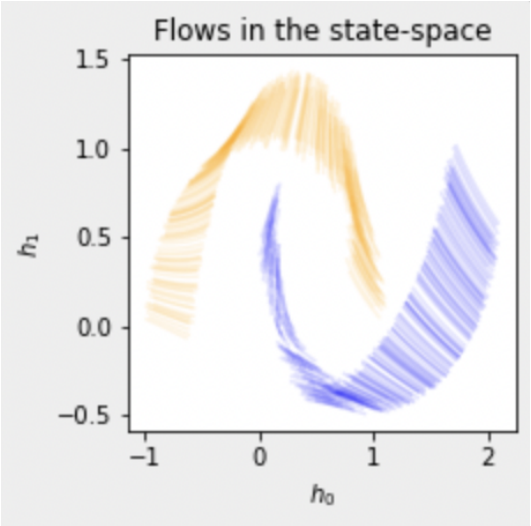

Figure 9: Low-training solution flows [3]

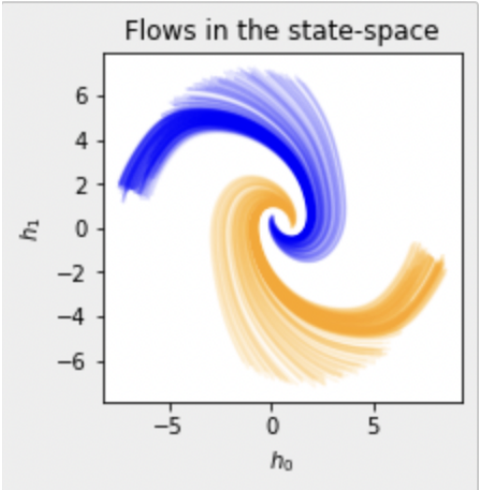

Figure 10: High-training solution flows [3]

7 Bibliography

References

[1] W. E. Boyce, R. C. DiPrima, and D. B. Meade, *Elementary Differential Equations and Boundary Value Problems*, 11th ed. Wiley, 2017.

[2] M. Betancourt, C. C. Margossian, and V. Leos-Barajas, "The discrete adjoint method: Efficient derivatives for functions of discrete sequences," 2020.

[3] M. Poli, S. Massaroli, A. Yamashita, H. Asama, J. Park, and S. Ermon, "Torchdyn: Implicit models and neural numerical methods in pytorch."

[4] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," *arXiv preprint arXiv:1806.07366*, 2018.

[5] I. J. Good, "Rational decisions," *Journal of the Royal Statistical Society*, 1952.

A Lagrangian Derivations

Defining $\mathcal{L}$ as $\mathcal{L}(y_1, \ldots, y_N, \lambda_1, \ldots, \lambda_N, \theta)$ from (25), (27) is shown as such:

$$\mathcal{L} = (y_N - \hat{y})^2 + \sum_{j=1}^{N} \lambda_j \cdot (y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta))$$

$$\mathcal{L} = (y_N - \hat{y})^2 + \sum_{j=2}^{N} \lambda_j \cdot (y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta)) + \lambda_1 \cdot (y_1 - y_0 - h \cdot f(y_0; \theta))$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_1} = 0 = 0 + y_1 - y_0 - h \cdot f(y_0; \theta)$$

$$y_1 = y_0 + h \cdot f(y_0; \theta)$$

Similarly,

$$\frac{\partial \mathcal{L}}{\partial \lambda_2} = 0 \iff y_2 = y_1 + h \cdot f(y_0; \theta)$$

$$\vdots$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_N} = 0 \iff y_N = y_{N-1} + h \cdot f(y_{N-1}; \theta)$$

may be shown similarly. Thus, generalizing across each $\frac{\partial \mathcal{L}}{\partial \lambda_j}$ for $j = 1, 2, \ldots, N$, it's clear that

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = 0 \iff y_j = y_{j-1} + h \cdot f(y_{j-1}; \theta) \iff y_{j+1} = y_j + h \cdot f(y_j; \theta), \tag{39}$$

where y_0 is defined similarly to the initial condition of the ODE in equation (6). (39) is identical to the formula for Euler's method given in (8); therefore, if each solution $y_j = y(t_j)$ is computed using Euler's method at each time t_j, then (27) will be guaranteed.

To show that (26) gives an iterative method to computing each λ_j, y_N should be defined as the final solution of $y(t)$ at time $t_{final} = t_N$ and $\mathcal{L}$ as $\mathcal{L}(y_1, \ldots, y_N, \lambda_1, \ldots, \lambda_N, \theta)$ as in (25). First, the partial derivative of $\mathcal{L}$ should be taken to compute a direct formula for λ_N based on $\hat{y}$ and y_N:

$$\mathcal{L} = (y_N - \hat{y})^2 + \sum_{j=1}^{N} \lambda_j \cdot (y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta))$$

$$\mathcal{L} = (y_N - \hat{y})^2 + \sum_{j=1}^{N-2} \lambda_j \cdot (y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta)) + \lambda_{N-1} \cdot (y_{N-1} - y_{N-2} - h \cdot f(y_{N-1}; \theta)) + \lambda_N \cdot (y_N - y_{N-1} - h \cdot f(y_N; \theta))$$

$$\frac{\partial \mathcal{L}}{\partial y_N} = 0 = 2 \cdot (y_N - \hat{y}) \cdot 1 + 0 + 0 + \lambda_N - 0 - 0$$

$$\lambda_N = 2 \cdot (\hat{y} - y_N).$$

Using this newfound λ_N, the partial derivatives of $\mathcal{L}$ with respect to y_{N-1}, y_{N-2}, ..., y_1 may be computed to find the rest of the Lagrangian multipliers:

$$\frac{\partial \mathcal{L}}{\partial y_{N-1}} = 0 = 0 + 0 + \lambda_{N-1} - 0 - 0 + 0 - \lambda_N - \lambda_N \cdot h \cdot \frac{\partial}{\partial y_{N-1}} f(y_{N-1}; \theta)$$
$$\lambda_{N-1} = \lambda_N + \lambda_N \cdot \frac{\partial}{\partial y_{N-1}} f(y_{N-1}; \theta),$$
$$\frac{\partial \mathcal{L}}{\partial y_{N-2}} = 0 = 0 + 0 + \lambda_{N-2} - 0 - 0 + 0 - \lambda_{N-1} - \lambda_{N-1} \cdot h \cdot \frac{\partial}{\partial y_{N-2}} f(y_{N-2}; \theta)$$
$$\lambda_{N-2} = \lambda_{N-1} + \lambda_{N-1} \cdot \frac{\partial}{\partial y_{N-2}} f(y_{N-2}; \theta),$$

and so on until

$$\frac{\partial \mathcal{L}}{\partial y_1} = 0 = 0 + 0 + \lambda_1 - 0 - 0 + 0 - \lambda_2 - \lambda_2 \cdot h \cdot \frac{\partial}{\partial y_1} f(y_1; \theta)$$
$$\lambda_1 = \lambda_2 + \lambda_2 \cdot \frac{\partial}{\partial y_1} f(y_1; \theta).$$

Generalizing, an iterative formula for each λ_j from setting (26) may be reached:

$$\frac{\partial \mathcal{L}}{\partial y_j} = 0 \iff \lambda_{j-1} = \lambda_j + \lambda_j \cdot h \cdot \frac{\partial}{\partial y_{j-1}} f(y_{j-1}; \theta) \tag{40}$$

where $j = 1, \ldots, N$ and $\lambda_N = 2 \cdot (\hat{y} - y_N)$.

The proof of $\frac{\partial E}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta}$ is more rigorous than what is stated here, but the following provides the general intuition behind its process and why it's important for the discrete adjoint method.

By computing $\frac{\partial \mathcal{L}}{\partial \theta}$ using using λ_j and y_j found from guaranteeing (26) and (27), respectively, and using a given θ, $\frac{\partial E}{\partial \theta}$ may be computed directly by equivalently solving for $\frac{\partial \mathcal{L}}{\partial \theta}$:

$$\mathcal{L} = (y_N - \hat{y})^2 + \sum_{j=1}^{N} \lambda_j \cdot (y_j - y_{j-1} - h \cdot f(y_{j-1}; \theta))$$
$$\frac{\partial \mathcal{L}}{\partial \theta} = 0 + 0 - 0 - \sum_{j=1}^{N} \lambda_j \cdot h \cdot \frac{\partial}{\partial \theta} f(y_{j-1}; \theta)$$
$$\frac{\partial \mathcal{L}}{\partial \theta} = -\sum_{j=1}^{N} \lambda_j \cdot h \cdot \frac{\partial}{\partial \theta} f(y_{j-1}; \theta).$$

Thus,

$$\frac{\partial E}{\partial \theta} = -\sum_{j=1}^{N} \lambda_j \cdot h \cdot \frac{\partial}{\partial \theta} f(y_{j-1}; \theta) \tag{41}$$

as stated in 31. The full proof may be found in reference [2].