EE5311 Differentiable and Probabilistic Computing

Adjoint Sensitivity Method for Neural Differential Equations

Wang Yilong
A0279793A
E1143606@u.nus.edu

Li Peiyu
A0289598W
E1323341@u.nus.edu

Xu Keying
A0298475H
E1352613@u.nus.edu

Abstract

Neural Differential Equations (NDEs) integrate differential equations with deep learning to model data through continuous-time dynamics, thus overcoming the limits of traditional discrete networks. Direct differentiation suffers from high memory usage and numerical errors during backpropagation. The Adjoint Sensitivity Method optimizes gradient computation, reducing both complexity and memory demands. This report details the core concepts of NDEs, the challenges of direct differentiation, and the principles of the Adjoint Sensitivity Method as implemented in the Julia-based NeuralODE solver.

Keywords: Neural Differential Equations, Adjoint Sensitivity, Julia.

Contents

1 Overview of Neural Differential Equations **2**
- 1.1 Fundamentals of Ordinary Differential Equations (ODE) . . . 2
  - 1.1.1 Definition and Physical Examples of ODEs . . . 2
  - 1.1.2 Numerical Methods for Solving ODEs . . . 2
- 1.2 Definition and Structure of Neural Differential Equations . . . 3

2 Limitations of Direct Differentiation Methods **4**
- 2.1 Memory Consumption Issues . . . 4
- 2.2 Error Accumulation Issues . . . 4

3 Theoretical Foundations of the Adjoint Sensitivity Method **6**

4 Reverse-Mode Derivatives for the IVP of ODE **8**
- 4.1 Forward Pass . . . 8
- 4.2 Adjoint Variable Initialization . . . 8
- 4.3 Backward Pass of the Adjoint Equation . . . 9
- 4.4 Computation of the Gradient with Respect to θ . . . 9

5 Memory Cost and Error Accumulation **10**
- 5.1 Advantages in Memory Management . . . 10
- 5.2 Error Control Mechanisms . . . 10
  - 5.2.1 Sources of Error in Reverse Integration . . . 10
  - 5.2.2 Error Control Strategies . . . 10

6 Application Example of the NeuralODE Solver in Julia **12**

7 Experimental Design and Discussion of Results **15**

8 Conclusions and Future Prospects **18**
- 8.1 Summary . . . 18
- 8.2 Future Prospects . . . 18

1 Overview of Neural Differential Equations

1.1 Fundamentals of Ordinary Differential Equations (ODE)

1.1.1 Definition and Physical Examples of ODEs

An ordinary differential equation describes the relationship between an unknown function and its derivatives. The standard form is given by

$$\frac{dy(t)}{dt} = f(t,\, y(t)) \tag{1}$$

For example, consider the free-fall problem of a ball under the influence of gravity and linear resistance:

- Let $y(t)$ denote the height and $v(t) = \frac{dy}{dt}$ denote the velocity
- The model is formulated as

$$v'(t) = g - \mu\, v(t) \tag{2}$$

  where g represents the gravitational acceleration and $\mu = \frac{k}{m}$ is the drag coefficient

Using the integrating factor method, the general solution is obtained as

$$v(t) = v_0 e^{-\mu t} + \frac{g}{\mu}\left(1 - e^{-\mu t}\right) \tag{3}$$

with the velocity asymptotically approaching the terminal velocity g/μ as $t \to \infty$

1.1.2 Numerical Methods for Solving ODEs

Since most ODEs cannot be solved analytically, numerical methods become essential tools. Two commonly used methods are described below:

1. **Euler's Method**

   Euler's method is a first-order explicit method. For discrete time points t_n with a step size $h = t_{n+1} - t_n$, the iterative formula is given by

$$y_{n+1} = y_n + h\, f(t_n, y_n) \tag{4}$$

   - Local truncation error: $O(h^2)$
   - Global error: $O(h)$

   Although straightforward to implement, Euler's method requires extremely small step sizes for stiff problems or when high accuracy is demanded.

2. **Classical Fourth-Order Runge–Kutta Method (RK4)**

   The RK4 method achieves higher accuracy by sampling multiple points within each step. Its procedure is as follows:

   (a) **Compute four slopes:**

$$\begin{aligned} k_1 &= f(t_n,\, y_n), \\ k_2 &= f\Big(t_n + \tfrac{h}{2},\, y_n + \tfrac{h}{2}\, k_1\Big), \\ k_3 &= f\Big(t_n + \tfrac{h}{2},\, y_n + \tfrac{h}{2}\, k_2\Big), \\ k_4 &= f(t_n + h,\, y_n + h\, k_3) \end{aligned} \tag{5}$$

   (b) **Update the state:**

$$y_{n+1} = y_n + \frac{h}{6}\,(k_1 + 2\,k_2 + 2\,k_3 + k_4) \tag{6}$$

   - Local truncation error: $O(h^5)$
   - Global error: $O(h^4)$

Although RK4 provides superior accuracy and stability compared to Euler's method for a given step size, small step sizes remain necessary for stiff problems. In contemporary applications, adaptive step size methods (e.g., Dormand–Prince RK45) are frequently employed to dynamically balance accuracy and efficiency.

1.2 Definition and Structure of Neural Differential Equations

The essence of Neural Differential Equations lies in employing neural networks as constituent modules within continuous-time dynamical systems. The specific model can be expressed in the form of an ordinary differential equation:

$$\frac{d\mathbf{h}(t)}{dt} = f\big(\mathbf{h}(t),\, t,\, \theta\big) \tag{7}$$

where:

- $\mathbf{h}(t)$ denotes the hidden state at time t
- f is a function parameterized by the neural network parameters θ
- Given the initial state $\mathbf{h}(0) = \mathbf{h}_{\text{in}}$, the final output is defined as

$$\mathbf{h}_{\text{out}} = \mathbf{h}(T) \tag{8}$$

  with T being the terminal time

Residual networks can be regarded as the discrete form of Neural Differential Equations, with the update rule for each layer given by

$$h_{t+1} = h_t + f(h_t, \theta) \tag{9}$$

Given an input $\mathbf{h}_{\text{in}}$, Neural Differential Equations compute the output $\mathbf{h}_{\text{out}}$ by solving the following initial value problem:

$$\frac{d\mathbf{h}(t)}{dt} = f_\theta(\mathbf{h}(t), t), \quad \mathbf{h}(0) = \mathbf{h}_{\text{in}} \tag{10}$$

After solving this differential equation, the output is assigned as

$$\mathbf{h}_{\text{out}} = \mathbf{h}(T) \tag{11}$$

Neural Differential Equations thereby generalize the approach to continuous time, describing the evolution of the state through differential equations.

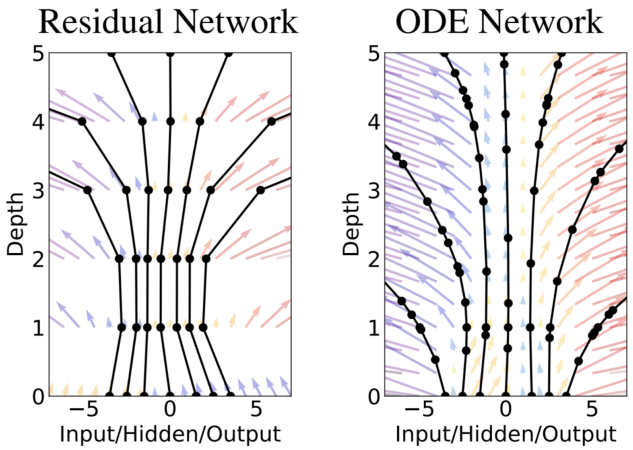

Figure 1: ResNet vs ODE

2 Limitations of Direct Differentiation Methods

In training Neural Differential Equations, computing the gradient of the loss function with respect to the model parameters θ is of paramount importance. The direct differentiation method (Standard Backpropagation Through ODE Solvers) computes gradients by performing forward integration followed by backward propagation via the chain rule

During the forward propagation phase, a numerical integrator (such as RK4, Euler's method, or an adaptive method) is employed to integrate the initial state $\mathbf{h}(0) = \mathbf{h}_{\text{in}}$ to the terminal time T, thereby obtaining the state trajectory

$$\{\mathbf{h}(t_1), \mathbf{h}(t_2), \ldots, \mathbf{h}(T)\} \tag{12}$$

This process is equivalent to solving the ordinary differential equation

$$\frac{d\mathbf{h}(t)}{dt} = f\big(\mathbf{h}(t), t, \theta\big) \tag{13}$$

In the backward propagation phase, the chain rule is applied to compute the gradients based on all intermediate states recorded during the forward pass. Let the loss function be defined as $\mathcal{E} = \ell\big(\mathbf{h}(T)\big)$. The gradient computation can then be expressed as

$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{k=1}^{K} \frac{\partial \mathcal{E}}{\partial \mathbf{h}(t_k)} \cdot \frac{\partial \mathbf{h}(t_k)}{\partial \theta} \tag{14}$$

where K denotes the number of discrete time steps during forward integration. In discrete deep networks (e.g., residual networks), a similar chain propagation occurs:

$$\frac{\partial \mathcal{E}}{\partial \mathbf{x}_\ell} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_\ell} = \frac{\partial \mathcal{E}}{\partial \mathbf{x}_L} \left(1 + \frac{\partial}{\partial \mathbf{x}_\ell} \sum_{i=\ell}^{L-1} F(\mathbf{x}_i)\right) \tag{15}$$

where $\mathbf{x}_\ell$ denotes the input to the ℓth layer and $F(\cdot)$ represents the function of the residual block

This method necessitates the storage of all intermediate states during forward propagation to facilitate gradient computation during backpropagation

Although the direct differentiation method directly leverages ODE solvers for gradient computation, it exhibits notable limitations when applied to long time series and high-resolution integration problems, particularly in terms of memory consumption and error accumulation

2.1 Memory Consumption Issues

In standard backpropagation, the computation of gradients requires the retention of all discrete states from the forward propagation phase:

- If each state vector has a dimension D and there are K discrete time steps, the memory requirement is approximately $O(K \times D)$
- When T is extended or the state dimension is high, memory consumption increases dramatically, potentially leading to insufficient hardware resources

This issue is particularly acute in continuous-time modeling and Neural Differential Equations, since ensuring numerical accuracy often necessitates subdividing the time steps, thereby substantially increasing memory overhead

2.2 Error Accumulation Issues

Numerical integration inherently introduces truncation errors, with each integration step contributing a small error. During backpropagation, these errors are gradually amplified through the chain rule. For instance, consider a prediction task

$$y(t) = \sin(t) \tag{16}$$

If each step introduces an error ϵ, the model prediction may be expressed as

$$y_{\text{pred}}(t) = \sin(t) + \epsilon \cdot t \tag{17}$$

As t increases, the cumulative error $\epsilon \cdot t$ may severely affect the stability of model training, necessitating the use of extremely small step sizes to mitigate the error

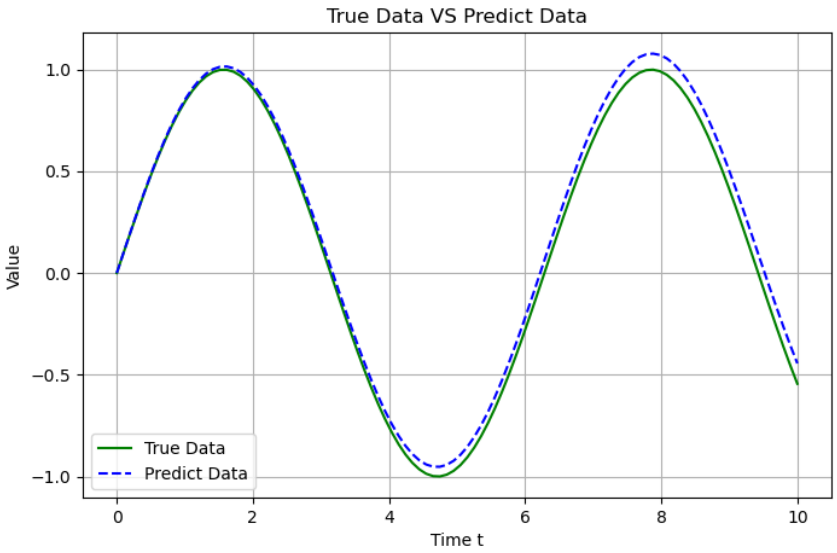

Figure 2: Schematic representation of error accumulation

3 Theoretical Foundations of the Adjoint Sensitivity Method

Training neural differential equations presents a fundamental challenge in computing gradients via ODE solvers. Due to the operations of ODE solvers (for example, multiple internal Runge–Kutta steps) not forming a standard layer-by-layer computational graph, automatic differentiation frameworks cannot process them directly. Although one may unroll the ODE solver (treating each small step as a layer and performing backpropagation through every step), such a *direct differentiation* approach is extremely memory-intensive—requiring either the storage of all intermediate states or the use of checkpointing techniques to recompute them. Moreover, differentiating through adaptive step size logic is nontrivial and may introduce additional numerical errors. Consequently, Chen et al. adopted the *adjoint sensitivity method*, a classical technique originating from continuous optimization and optimal control (tracing back to the work of Pontryagin et al. in 1962). The adjoint method obtains gradients by solving a second-order ordinary differential equation in reverse time, thereby avoiding backpropagation through each solver step

Consider the neural differential equation system

$$\frac{dh}{dt} = f(h(t), t; \theta) \tag{18}$$

with the initial state $h(0)$ and terminal state $h(T)$. Suppose there exists a scalar loss function $L = \mathcal{L}(h(T))$ (for instance, $h(T)$ may be fed into a cost function for comparison with a target). Our objective is to compute the gradient $\nabla_\theta L$ efficiently. Since $h(T)$ depends on θ, the chain rule yields

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial h(T)} \frac{\partial h(T)}{\partial \theta} \tag{19}$$

where $\frac{\partial h(T)}{\partial \theta}$ denotes the sensitivity of the ODE solution with respect to the parameters. This sensitivity can be obtained by differentiating the state dynamic equation with respect to θ. Differentiating

$$\frac{dh}{dt} = f(h, t; \theta) \tag{20}$$

with respect to θ results in the *sensitivity equation*

$$\frac{d}{dt}\left(\frac{\partial h}{\partial \theta}\right) = \frac{\partial f}{\partial h}(h(t), t; \theta) \frac{\partial h}{\partial \theta} + \frac{\partial f}{\partial \theta}(h(t), t; \theta) \tag{21}$$

with the initial condition $\frac{\partial h(0)}{\partial \theta} = 0$ (assuming $h(0)$ is independent of θ). Here, $\frac{\partial h}{\partial \theta}$ is a matrix (the Jacobian of the state with respect to the parameters). This constitutes a first-order ODE that describes sensitivity. Integrating it forward along with the state equation yields $\frac{\partial h(T)}{\partial \theta}$, which can then be substituted into the chain rule. However, for high-dimensional problems the size of $\frac{\partial h}{\partial \theta}$ may be extremely large (of dimension [state dimension] $\times$ [parameter dimension]), rendering the computational cost of propagation prohibitively high. The adjoint method circumvents this issue by introducing a vector (the co-state) to transmit the necessary information in a more compact form

Define the *adjoint state* $a(t) := \frac{\partial L}{\partial h(t)}$, which is a row vector of the same dimension as h and represents the gradient of the loss function with respect to the state $h(t)$. Intuitively, $a(t)$ indicates how variations in $h(t)$ affect the final loss. By differentiating $a(t)$ in reverse, one can derive an ODE that describes its evolution. Employing the chain rule (analogous to differentiating future costs in optimal control), one obtains

$$\frac{da(t)}{dt} = -a(t) \frac{\partial f(h(t), t; \theta)}{\partial h} \tag{22}$$

with the *terminal condition* $a(T) = \frac{\partial L}{\partial h(T)}$. This equation, known as the *adjoint equation*, propagates gradients backward from time T to 0. Essentially, it is the continuous counterpart of backpropagation, in which $\frac{\partial f}{\partial h}$ plays a role analogous to the Jacobian of a layer at time t, and the current gradient $a(t)$ is multiplied by it (the negative sign indicating reverse time propagation). This equation can be derived by differentiating the inner product $a(t) \cdot h(t)$ along the true trajectory so that $dL = 0$, or more formally through the Lagrange multiplier method under ODE constraints. The key insight is that, given the terminal condition $a(T)$, one can compute $a(t)$ by solving the ODE backward in time

Once $a(t)$ is obtained, the gradient with respect to the parameters can be computed through an additional integration. Differentiating the loss function with respect to θ (while taking into account the dependence of the integration of $h(t)$ on θ) yields

$$\frac{dL}{d\theta} = \int_0^T a(t) \frac{\partial f(h(t), t; \theta)}{\partial \theta} \, dt \tag{23}$$

This equation indicates that the contributions to the parameter gradient are accumulated along the trajectory, with the weighting determined by the adjoint state $a(t)$. In practice, one may integrate an augmented state together with $a(t)$ to obtain $\frac{dL}{d\theta}$. For example, one may introduce an auxiliary variable $b(t)$ that satisfies

$$\frac{db}{dt} = a(t) \frac{\partial f(h(t), t; \theta)}{\partial \theta} \tag{24}$$

with the terminal condition $b(T) = 0$. By construction, $b(0)$ is equal to

$$\int_0^T a(t) \frac{\partial f(h(t), t; \theta)}{\partial \theta} \, dt = \frac{dL}{d\theta} \tag{25}$$

(as a row vector). Therefore, by augmenting the adjoint system (and similarly augmenting to accumulate gradients with respect to the initial conditions), all required gradients can be computed in a single backward integration

4 Reverse-Mode Derivatives for the IVP of ODE

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$
$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$ ▷ Define initial augmented state
def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$): ▷ Define dynamics on augmented state
return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T} \frac{\partial f}{\partial \theta}]$ ▷ Compute vector-Jacobian products
$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$ ▷ Return gradients

Assume that we are given the following initial value problem (IVP)

$$\begin{cases} \frac{dz(t)}{dt} = f\big(z(t),\ t,\ \theta\big), \quad t \in [t_0, t_1] \\ z(t_0) = z_0(\theta) \end{cases} \tag{26}$$

where

- $z(t) \in \mathbb{R}^n$ denotes the state variable
- $\theta \in \mathbb{R}^p$ is the parameter vector to be learned or optimized
- $f(\cdot)$ specifies the dynamical equations of the system

We define a loss function L (alternatively denoted as ℓ) that depends on the final state $z(t_1)$

$$L = \ell\big(z(t_1)\big) \tag{27}$$

Our objective is to compute the gradient of the loss with respect to the parameters θ, namely

$$\frac{\partial L}{\partial \theta} \tag{28}$$

4.1 Forward Pass

1. Given the initial condition $z(t_0) = z_0(\theta)$, the ODE is numerically solved to obtain the trajectory $z(t)$ over the interval $[t_0, t_1]$
2. In particular, the state $z(t_1)$ at the final time is obtained

4.2 Adjoint Variable Initialization

Define the adjoint variable (or dual variable) $a(t)$, which represents the sensitivity of the loss L with respect to the state $z(t)$

$$a(t) \ = \ \frac{\partial L}{\partial z(t)} \tag{29}$$

At the final time t_1, since $L = \ell\big(z(t_1)\big)$, it follows that

$$a(t_1) \ = \ \frac{\partial \ell\big(z(t_1)\big)}{\partial z(t_1)} \tag{30}$$

This serves as the initialization of the adjoint variable (note that it is initialized at the final time and then integrated backward)

4.3 Backward Pass of the Adjoint Equation

The adjoint variable satisfies the following differential equation, which propagates backward from t_1 to t_0

$$\frac{d\,a(t)}{dt} \;=\; -\left(\frac{\partial f}{\partial z}(t,\; z(t),\; \theta)\right)^{T} a(t), \quad a(t_1) = \frac{\partial \ell\big(z(t_1)\big)}{\partial z(t_1)} \tag{31}$$

4.4 Computation of the Gradient with Respect to θ

By the chain rule, one obtains

$$\frac{\partial L}{\partial \theta} \;=\; \int_{t_0}^{t_1} a(t)^{T}\, \frac{\partial f}{\partial \theta}\big(t,\; z(t),\; \theta\big)\, dt \;+\; \left(\frac{\partial z_0(\theta)}{\partial \theta}\right)^{T} a(t_0) \tag{32}$$

where

- $\displaystyle\int_{t_0}^{t_1} a(t)^{T}\, \frac{\partial f}{\partial \theta}\, dt$ represents the accumulated inner product of the adjoint variable with the partial derivative of the dynamics with respect to θ
- $\displaystyle\left(\frac{\partial z_0(\theta)}{\partial \theta}\right)^{T} a(t_0)$ accounts for the dependency of the initial condition on θ (this term is zero if the initial condition is independent of θ)

  First, perform a forward integration to obtain $z(t)$; next, initialize $a(t_1)$ using the sensitivity of the final state; then, integrate the adjoint equation backward to compute $a(t)$; finally, accumulate the integration of $a(t)$ with $\frac{\partial f}{\partial \theta}$ to obtain $\frac{\partial L}{\partial \theta}$

5 Memory Cost and Error Accumulation

5.1 Advantages in Memory Management

In a conventional "automatic differentiation + ODE solver" workflow, if one employs discrete time-step backpropagation (i.e., *forward accumulation* combined with backpropagation through the unrolled computational graph) to compute gradients, it requires:

1. **Storing all intermediate states from the forward pass**, including the state $z(t)$ at each time step and the corresponding intermediate variables
2. During the backward pass, sequentially retrieving these states in order to compute the gradients with respect to the parameters

For long time series or large-scale systems, the memory requirement of this approach increases almost linearly with the number of discrete steps, and in some cases may even render training or solving infeasible due to insufficient memory

In contrast, the adjoint sensitivity method theoretically requires only:

- **Storing the initial state and the state at the final time** (as well as the necessary adjoint variables)
- Dynamically retrieving information such as $\frac{\partial f}{\partial z}$ and $\frac{\partial f}{\partial \theta}$ during the reverse integration

Compared with directly storing the entire forward trajectory, it is unnecessary to explicitly save every time step's state and intermediate computation graph. The key idea is that:

1. **Forward pass**: Integrate the initial state $z(t_0)$ to t_1 to obtain the final state $z(t_1)$
2. **Backward pass**: Starting from t_1, integrate the adjoint equation backward in time. In this process, there is no need to access all stored intermediate data, as the ODE solver can "regenerate" (or approximate) the required states during the reverse integration

   In this manner, the memory consumption is mainly concentrated in the ODE solver itself (and the storage of the necessary adjoint variables), eliminating the need for explicit storage of the entire forward trajectory

5.2 Error Control Mechanisms

5.2.1 Sources of Error in Reverse Integration

Although the adjoint sensitivity method offers significant advantages in memory management, it also faces issues of numerical error accumulation:

1. **Discretization error**: Since ODE solvers typically employ numerical methods (such as Runge–Kutta, Adams, BDF, etc.), reverse integration inevitably introduces numerical discretization errors
2. **Floating-point precision error**: In the cumulative computations over long time series or large-scale systems, rounding errors in floating-point arithmetic may gradually amplify
3. **Discrepancies in regenerating the forward trajectory**: During the reverse integration, the adjoint equation requires both $\frac{\partial f}{\partial z}$ and $z(t)$ itself; if the precise forward state is not stored but instead approximated via interpolation or re-integration, additional bias may be introduced

5.2.2 Error Control Strategies

1. **Selection of an Appropriate ODE Solver and Tolerance Settings**
   - Common adaptive step size algorithms (e.g., Dormand–Prince 5(4), 8(7), etc.) can dynamically adjust based on the local truncation error, ensuring that the overall error remains within a specified tolerance
   - During reverse integration, it is equally necessary to set proper absolute and relative tolerances to avoid errors or unnecessary computational expense due to over- or under-precision

2. **Checkpointing (Segmented Storage)**
    - At key time points, one may store precise forward states to avoid relying on re-integration from the beginning during the reverse pass
    - By performing forward integration repeatedly between several segments, the dependency on the entire trajectory is effectively reduced, thereby mitigating cumulative errors while balancing memory usage and numerical precision
3. **Bidirectional Comparison or Contrast Metrics**
    - Some studies perform an additional forward integration (starting from the obtained $z(t_0)$) after reverse integration to compare the final error, thereby verifying the numerical stability and consistency of the adjoint variable
    - If the error is found to be excessive, adjustments to the ODE solver's step size or tolerance may be warranted
4. **Utilization of Higher-Precision Floating-Point Computations**
    - In scenarios demanding extreme precision, one may consider mixed precision or double precision (64-bit) computations, albeit with a careful evaluation of the associated computational cost

In practical applications, the configuration of the adjoint method (including step size, adaptive tolerance, checkpoint intervals, etc.) should be flexibly adjusted based on the task requirements (such as the acceptable error range and available hardware resources) in order to achieve an optimal balance among memory usage, computational cost, and numerical precision

6 Application Example of the NeuralODE Solver in Julia

Neural differential equations can be implemented in various frameworks; here we present a simple implementation using **Julia**, which leverages the SciML/DiffEqFlux libraries to handle differential equations and machine learning. We construct a neural differential equation model to learn the free-fall dynamics introduced earlier. Specifically, we first generate a synthetic dataset of a free-falling ball $(t, y(t))$ and then fit the data using a neural differential equation model that internally employs the adjoint method.

Setup: We assume that `DifferentialEquations.jl` and `DiffEqFlux.jl` are already installed in Julia. Next, we generate a synthetic dataset of a free-falling ball's $(t, y(t))$ and build a neural differential equation to fit these data.

Data Generation and True Dynamics

```julia
using DifferentialEquations, DiffEqFlux, Flux

# True dynamics of free fall with drag (used for data generation)
const g = 9.8        # gravitational acceleration (m/s^2)
const m = 1.0        # mass (kg)
const k = 0.5        # drag coefficient (kg/s)
μ = k/m              # constant for ease of calculation

# Define the true physical ODE: state u = [y, v] (position and velocity)
function freefall!(du, u, p, t)
    # u[1] = y (position), u[2] = v (velocity)
    du[1] = u[2]                        # dy/dt = v
    du[2] = g - μ * u[2]                # dv/dt = g - (k/m)*v
end

# Initial conditions: free fall starting from y = 0 and v = 0
u0 = [0.0, 0.0]
tspan = (0.0, 5.0)                     # simulate for 5 seconds
tsteps = range(tspan[1], tspan[2], length=100)  # 100 observation points

# Solve the ODE to generate synthetic position data
prob = ODEProblem(freefall!, u0, tspan)
true_solution = solve(prob, Tsit5(); saveat=tsteps)  # using the Tsit5 solver (a Runge–Kutta method)
y_data = [sol[1] for sol in true_solution]           # extract the position trajectory (length 100)
```

In the code above, the ODE solver (`Tsit5` — a Runge–Kutta method) is used to generate `y_data`, which represents the true height of the free-falling object as a function of time. Next, we construct a neural differential equation model.

NeuralODE Model Construction

Note:

- `NeuralODE(nn, tspan, Tsit5(); saveat=tsteps)` creates an ODE problem with *nn* serving as the right-hand side function $f(u)$. By default, this method employs the *adjoint sensitivity method* to compute gradients (DiffEqFlux internally invokes the continuous adjoint method during optimization)
- The function `predict_positions` solves the ODE from $t = 0$ to 5 s using the current parameters p and returns the predicted position values
- The loss function computes the sum of squared errors between the predicted values and the true positions

Now, we train the neural differential equation. Flux's ADAM optimizer (or another optimizer) is used to minimize the loss function. In each iteration, a forward pass is first executed to compute the predictions and loss, followed by reverse differentiation via the adjoint method to update the parameters.

```julia
# Define a neural network for the right-hand side function f(u, t; θ)
# We use a simple Dense network with tanh activation (Dense is provided by Flux)
nn = Chain(
    Dense(2, 16, tanh),   # 2 inputs (y, v) -> 16 hidden units
    Dense(16, 2)          # 16 -> 2 outputs (predicting du/dt)
)

# Build the NeuralODE structure
tspan = (0.0, 5.0)
neural_ode = NeuralODE(nn, tspan, Tsit5(); saveat=tsteps)

# Define the loss function to match the predicted positions with y_data
function predict_positionsθ(p)
    sol = neural_ode(u0, p)       # solve the neural ODE with current parameters
    # neural_ode(u0, p) returns a solution object; extract the array output:
    y_pred = sol[1, :]            # extract the first component (position) at all time steps
    return y_pred
end

loss(p) = sum((predict_positionsθ(p) .- y_data).^2)  # mean squared error over all time points

# Retrieve the initial parameters of the neural network
p = Flux.params(nn)    # set of trainable parameters
opt = ADAM(0.05)       # ADAM optimizer with a learning rate of 0.05

# Train for a number of epochs
for epoch in 1:200
    grads = Flux.gradient(p) do        # compute gradients of the loss
        loss_val = loss(p)
        loss_val
    end
    Flux.Optimise.update!(opt, p, grads)   # update the parameters using ADAM
    if epoch % 50 == 0
        println("Epoch $epoch, loss = ", loss(p))
    end
end
```

In each gradient computation, `Flux.gradient` performs backpropagation on $loss(p)$, which requires differentiating the process of solving $neural\_ode(u0, p)$. DiffEqFlux accomplishes this by solving the *adjoint ODE* (i.e., the reverse propagation process) rather than backpropagating through every solver step. This ensures that even if $neural\_ode$ contains many computational steps, memory usage remains low. The loss is printed every 50 epochs to monitor training progress (in practical applications, more advanced training routines or callbacks provided by SciML may be used to simplify the procedure, but the loop above clearly demonstrates the entire process)

After training, the parameters of the neural differential equation should be adjusted such that the predicted trajectory matches the true data. We then evaluate the training results.

Evaluating the Trained Model

```julia
# Evaluate the trained model
trained_positions = predict_positionsθ(p)
println("Final training loss: ", loss(p))
println("True vs Predicted positions at t=5s: ", y_data[end], " vs ", trained_positions[end])

plt1 = plot(tsteps, y_data, label="True positions", xlabel="Time (s)", ylabel="Position (m)",
    title="Freefall Dynamics: True vs Predicted Positions", lw=2)
plot!(plt1, tsteps, trained_positions, label="Predicted positions", lw=2, ls=:dash)
plt2 = plot(1:epochs, loss_list, label="Loss", xlabel="Epoch", ylabel="Loss",
    title="Training Loss Curve", lw=2)

plot(plt1, plt2, layout=(1,2), size=(1200,400))
```

Upon convergence, we expect the loss to be very small and the predicted heights to closely follow the true heights

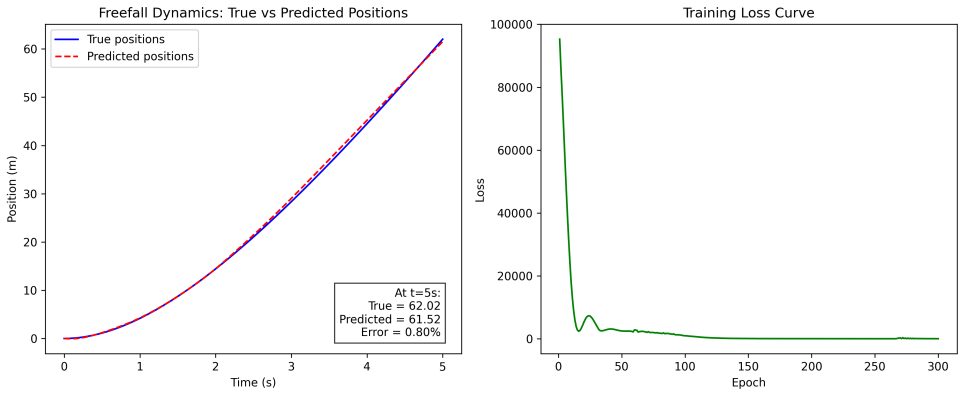

Figure 3: Comparison of true and predicted free-fall trajectories

7 Experimental Design and Discussion of Results

The following data compare the computational time and gradient accuracy between direct differentiation methods and the adjoint sensitivity method [10]. In this context, the Finite Difference Method (FDM) and Forward Sensitivity Analysis (FSA) are categorized as direct differentiation methods (DDM)

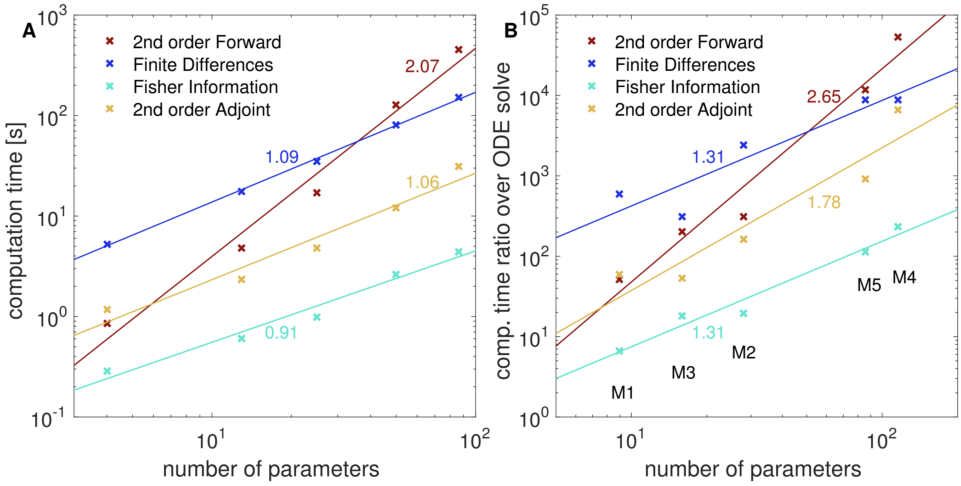

Figure 4: Computational time for computing the Hessian matrix using different methods

Figure 4 presents a comparison of the computational time required by various methods to compute the Hessian matrix. It is evident that the adjoint method is an efficient choice for Hessian computation because its computational time increases at a lower rate, rendering it suitable for large-scale parameter problems. In comparison with Forward and Finite Differences, the computational burden of the adjoint method is significantly lower, especially when the number of parameters is large.

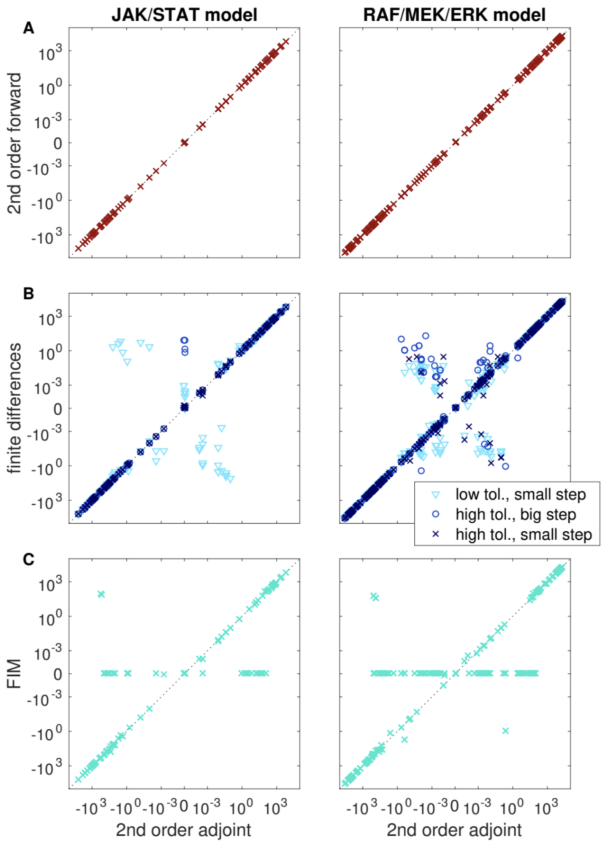

Figure 5: Accuracy of different methods for computing the Hessian matrix

The figure compares various methods for computing or approximating the Hessian at the global optimum for models M2 and M3. Each point represents a Hessian entry calculated by two methods:

1. Second order forward analysis vs. second order adjoint analysis.
2. Finite differences (under varying step sizes and solver tolerances) vs. second order adjoint analysis.
3. Fisher information matrix vs. second order adjoint analysis.

In this figure, second order adjoint analysis is taken as the baseline, with the left and right columns representing results for the JAK/STAT and RAF/MEK/ERK models, respectively. For the comparison between second order forward and adjoint analyses, the scatter points lie nearly perfectly along the diagonal. This strong alignment indicates that both methods produce nearly identical Hessian values, though the forward approach tends to be more computationally expensive than the adjoint method. In the case of finite differences, the points generally cluster around the diagonal but show greater dispersion—especially when using low tolerances and large step sizes. These deviations suggest that the accuracy of finite differences is highly sensitive to the chosen parameters, resulting in noticeable numerical errors when compared to the adjoint method.

Similarly, when comparing the Fisher information matrix with the second order adjoint analysis, most points align along the diagonal; however, a few regions show systematic deviations. This indicates that while the Fisher information matrix can serve as an approximation of the Hessian, it may introduce bias for certain components.

Overall, the results clearly demonstrate that the second order adjoint method offers an optimal balance between accuracy and computational efficiency. It closely matches the forward analysis results while avoiding the parameter sensitivity of finite differences and the potential bias of the Fisher information matrix.

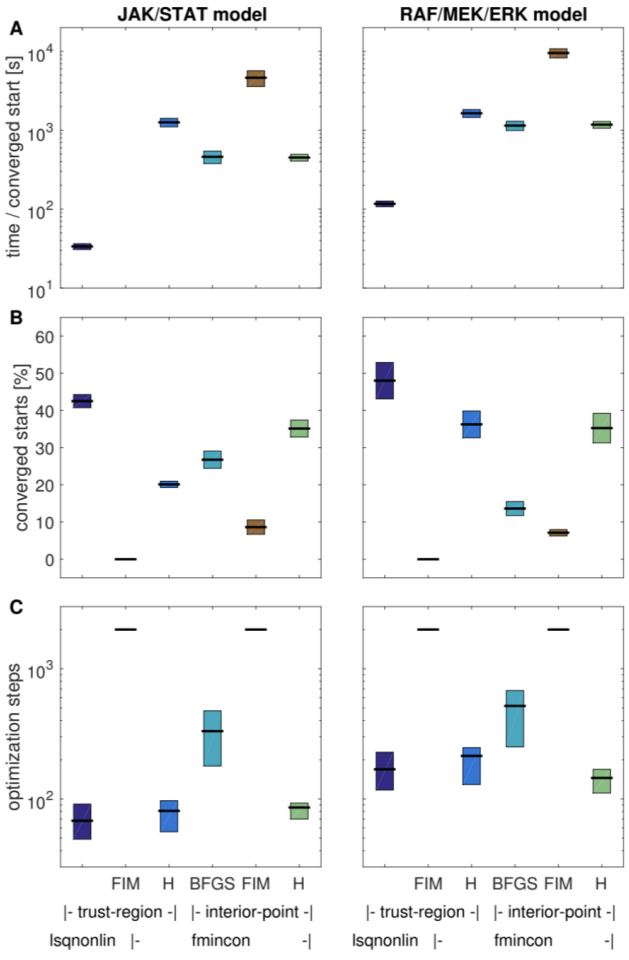

Figure 6: Performance of different methods in optimization algorithms

To assess the performance of different methods in optimization algorithms, the MATLAB `fmincon` optimizer was employed, using Hessians computed by various methods. The Trust-Region Reflective and Interior-Point algorithms were compared. As shown in Figure 6, the method based on second order adjoint sensitivity for computing the Hessian exhibited superior convergence and the shortest optimization time. The Hessian computation via the second order adjoint sensitivity method enhances the stability of the optimizer, resulting in fewer optimization steps and faster convergence compared to other methods

Furthermore, in the official Julia blog post [7], a comparison between direct differentiation and the adjoint sensitivity method was presented. Although detailed numerical tables were not provided, it was clearly stated that the primary issue with direct differentiation is the necessity to store the entire computational trajectory, which leads to high memory overhead. In contrast, the adjoint method is more memory-efficient and well-suited for long time series modeling. The author also noted that this does not imply that the adjoint sensitivity method is always the best choice; since direct differentiation yields more accurate gradients, it may be faster and more precise when the ODE time steps are sufficiently small

Discussion

NeuralODE has demonstrated advantages over other algorithms across several tasks

In supervised learning tasks, researchers have applied NeuralODE to the MNIST handwritten digit recognition task [2]. Experimental results indicate that the NeuralODE model achieves accuracy comparable to that of residual networks (ResNet) while using fewer parameters. Furthermore, NeuralODE exhibits constant spatial complexity; compared with traditional deep neural networks, it requires fewer parameters and lower memory consumption. NeuralODE can therefore serve as a substitute for ResNet, achieving similar performance while incorporating the feature of continuous depth

The advent of NeuralODE has enabled the development of Continuous Normalizing Flows (CNF) [3]. CNF experiments, which represent flows as continuous-time dynamical systems, can more stably fit complex distributions by avoiding the limitations associated with computing Jacobian determinants in traditional discrete normalizing flows. In time series modeling experiments [9], ODE-RNN outperformed standard RNN in both interpolation and extrapolation tasks. In terms of computational time, the computational load of ODE-RNN is 60% higher than that of conventional models. This is because NeuralODE models the evolution of the hidden state as a continuous-time differential equation, thereby overcoming the irregular sampling issues encountered by traditional recurrent neural networks (RNN) when processing such data. This makes NeuralODE more suitable for physical modeling, biosignal analysis, and related tasks. A recent study also demonstrated the advantages of NODE in autonomous driving scenarios [1]. Using the NGSIM highway dataset, the prediction performance of NODE was compared with that of TCN, ResNet, and CNN. The NODE model outperformed the others under all loss functions, achieving a test accuracy of 0.972, compared to 0.965 for ResNet and CNN and 0.904 for TCN. This highlights the extensive application potential of NODE in autonomous driving systems

8 Conclusions and Future Prospects

8.1 Summary

In this report, we rigorously defined ordinary differential equations, presented a concrete example with error and stability analyses, and formally described neural differential equations—demonstrating their equivalence with residual networks in the limit. We derived the adjoint sensitivity method in a stepwise manner, first introducing the necessary differentiability assumptions and then showing how gradients are obtained by solving the adjoint ODE, effectively illustrating the continuous analogue of backpropagation (inspired by Pontryagin's principle). We discussed the benefits of this method in reducing memory and computational costs, while also identifying potential numerical error sources.

The Julia implementation concretizes these ideas: although the code examples are simplified for clarity, they employ a black-box ODE solver for the forward pass and utilize DiffEqFlux's adjoint method for gradient computation, thus clearly translating theory into practice.

8.2 Future Prospects

With ongoing advances in computational hardware, automatic differentiation, and hybrid modeling approaches, the adjoint sensitivity method (ASM) is expected to play an increasingly important role in deep learning, scientific computing, and engineering optimization.

(1) Applications in Large-Scale Deep Learning

ASM is currently applied to small-scale neural differential equations. However, as deep neural networks and physical simulations scale up (e.g., spatio-temporal modeling for weather forecasting or continuous-time reinforcement learning for robotics), further optimization of ASM for large-scale systems is an important research direction [2].

(2) Efficient Numerical Computation and Hardware Optimization

Current ASM computations mainly rely on CPUs or GPUs. With the emergence of TPUs and specialized accelerators, future work should focus on:

- **Solver Optimization:** Enhancing ODE solvers (such as `DifferentialEquations.jl` or `torchdiffeq`) to better handle high-dimensional gradient computations.
- **Parallel Computing:** Developing efficient strategies to parallelize ASM across multiple GPUs/TPUs for distributed training.

(3) Applications in Physical Sciences and Engineering

ASM can improve modeling accuracy in areas like:

- **Computational Fluid Dynamics:** Solving parameter sensitivity in the Navier–Stokes equations to enhance simulation efficiency [8].
- **Biomedical Modeling:** Reducing computational burdens in pharmacokinetic/pharmacodynamic and neural signal modeling [4].

Opportunities and Challenges

Despite its advantages, ASM faces challenges including high computational cost in high-dimensional systems [2], numerical instability, and difficulties in real-time processing. Potential remedies include adaptive step-size control, advanced integration methods, distributed training algorithms [6], and scalable gradient computation techniques [5, 8, 4, 6].

Contributions of Group Members

Member	Contributions
Xu Keying	Abstract, Section 1 – 2
Wang Yilong	Section 3 – 6
Li Peiyu	Section 6 – 8

Furthermore, all group members participated in literature review and discussion, as well as in the composition of the article layout and the references section.

References

[1] Berk Agin, Pelin Oksuz, Semih Gulum, Ozan Caldiran, and Ali Nagaria. Social aware maneuver prediction based on neuralode for autonomous driving. In *2023 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pages 1–6. IEEE, 2023.

[2] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[3] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. *arXiv preprint arXiv:1810.01367*, 2018.

[4] Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural controlled differential equations for irregular time series. *Advances in neural information processing systems*, 33:6696–6707, 2020.

[5] Xuechen Li, Ting-Kam Leonard Wong, Ricky TQ Chen, and David Duvenaud. Scalable gradients for stochastic differential equations. In *International Conference on Artificial Intelligence and Statistics*, pages 3870–3882. PMLR, 2020.

[6] Michael Lutter, Christian Ritter, and Jan Peters. Deep lagrangian networks: Using physics as model prior for deep learning. *arXiv preprint arXiv:1907.04490*, 2019.

[7] Chris Rackauckas, Yingbo Ma, Vaibhav Dixit, Kirill Zubov, Rohit Goswami, Mohamed Tarek, and Ali Vahdati. Neural ordinary differential equations. `https://julialang.org/blog/2019/01/fluxdiffeq/`, 2019. Accessed: 2023-10-10.

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

[9] Yulia Rubanova, Ricky TQ Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. *Advances in neural information processing systems*, 32, 2019.

[10] Paul Stapor, Fabian Froehlich, and Jan Hasenauer. Optimization and uncertainty analysis of ode models using 2nd order adjoint sensitivity analysis. *bioRxiv*, page 272005, 2018.