Neural Ordinary Differential Equations: Theory and Implementation on Spiral datasets

Nishant Padmanabhan

June 2025

Abstract

Neural Ordinary Differential Equations (Neural ODEs) provide a framework for modeling continuous dynamics with neural networks. This report aims to explore the logic, mathematical underpinnings and implementation of Neural ODEs, in the context of recognizing and predicting spiral-shaped differential equations.

1 Introduction

Ordinary Differential Equations (ODEs) are equations that relate a function to its derivatives. In recent years, they have inspired the development of continuous-depth neural networks called Neural ODEs. These models offer an elegant alternative to traditional feedforward networks by treating hidden layers as a continuous transformation parameterized by an ODE solver.

Consider a Residual network type model (*ResNet*), with skip connections. The mathematical equation behind such a model would look like the following : $\mathbf{h}_2 = \mathrm{x} + \mathrm{f}(\mathbf{h}_1)$, where x is the initial output, and f is a function on the previous layer, usually something of the form $\tanh{(h_1 * w + b_i)}$.By a happy coincidence, this is very similar to the structure of a differential equation, as shown in the next section.

2 . Background: From ODEs to Neural ODEs

To define any differential equation, one needs to know two things; the initial state and the bounded gradient at each point, i.e the input (x), and the gradient $\frac{dy}{dx}$ $\forall$ x $\subset$ $(0, n\pi)$. The logic behind using Neural ODEs for modeling differential equations is that it assumes $\mathbf{h}(t_0)$ as the input, $\mathbf{h}(t_1)$ as the output for any two consecutive layers $\mathbf{h}(t_0), \mathbf{h}(t_1)$.

Let $\mathbf{h}(t)$ be the hidden state at time t. A Neural ODE models the evolution of this state as:

$$\mathbf{h}(t_1) - \mathbf{h}(t_0) = f(\mathbf{h}(t), t; \theta) \tag{1}$$

If the difference between $\mathbf{h}(t_0)$ and $\mathbf{h}(t_1)$ is small, one can assume the difference to be the derivative of $\mathbf{h}(t_0)$ at that point

$$\mathbf{h}(t_1) - \mathbf{h}(t_0) = \frac{d\mathbf{h}(t)}{dt} \tag{2}$$

And hence

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t; \theta) \tag{3}$$

where f is a neural network and θ its parameters. Instead of stacking layers, the final state is obtained by integrating this differential equation:

$$\mathbf{h}(t_1) = \mathbf{h}(t_0) + \int_{t_0}^{t_1} f(\mathbf{h}(t), t; \theta)\, dt \tag{4}$$

3 . Adjoint Sensitivity Method

To train a Neural ODE, we must differentiate through the ODE solver. Instead of backpropagating through all solver steps, the adjoint sensitivity method solves a backward ODE to compute gradients efficiently.

The following is a proof on the calculation of the derivative of the adjoint state, which is far simpler computationally than regular backpropagation.

3.1 Proof of Instantaneous Change of Variables Formula

To prove the theorem,

$$\frac{\partial \log p(\mathbf{z}(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial \mathbf{z}}(t)\right) \tag{5}$$

we take the infinitesimal limit of finite changes of $\log p(\mathbf{z}(t))$ over time.

Let $\mathbf{z}(t+\varepsilon) = T_\varepsilon(\mathbf{z}(t))$ represent the transformation over an ε increment in time. Assume f is Lipschitz continuous in $\mathbf{z}(t)$ and continuous in t, ensuring a unique solution by Picard's theorem. Assume $\mathbf{z}(t)$ is bounded, and so are f, T_ε, and $\frac{\partial T_\varepsilon}{\partial \mathbf{z}}$.

We begin from the change of variables formula:

$$\begin{aligned}
\frac{\partial \log p(\mathbf{z}(t))}{\partial t} &= \lim_{\varepsilon \to 0^+} \frac{\log p(\mathbf{z}(t)) - \log\left|\det \frac{\partial T_\varepsilon(\mathbf{z}(t))}{\partial \mathbf{z}}\right| - \log p(\mathbf{z}(t))}{\varepsilon} && (6)\\
&= -\lim_{\varepsilon \to 0^+} \frac{\log\left|\det \frac{\partial T_\varepsilon(\mathbf{z}(t))}{\partial \mathbf{z}}\right|}{\varepsilon} && (7)\\
&= -\lim_{\varepsilon \to 0^+} \frac{\partial}{\partial \varepsilon} \log\left|\det \frac{\partial T_\varepsilon(\mathbf{z}(t))}{\partial \mathbf{z}}\right| && (8)
\end{aligned}$$

Using the Jacobian Formula $\frac{d}{d\varepsilon}\log\det A = \mathrm{tr}(A^{-1}\frac{dA}{d\varepsilon})$:

$$\begin{aligned}
\frac{\partial \log p(\mathbf{z}(t))}{\partial t} &= -\lim_{\varepsilon \to 0^+} \mathrm{tr}\left[\mathrm{adj}\left(\frac{\partial T_\varepsilon(\mathbf{z}(t))}{\partial \mathbf{z}}\right) \cdot \frac{\partial}{\partial \varepsilon}\left(\frac{\partial T_\varepsilon(\mathbf{z}(t))}{\partial \mathbf{z}}\right)\right] && (9)\\
&= -\mathrm{tr}\left(\lim_{\varepsilon \to 0^+} \frac{\partial}{\partial \varepsilon} \frac{\partial T_\varepsilon(\mathbf{z}(t))}{\partial \mathbf{z}}\right) && (10)
\end{aligned}$$

Substituting T_ε with its Taylor series expansion and taking the limit:

$$\begin{aligned}
\frac{\partial \log p(\mathbf{z}(t))}{\partial t} &= -\text{tr}\left(\lim_{\varepsilon \to 0^+} \frac{\partial}{\partial \varepsilon}\frac{\partial}{\partial \mathbf{z}}\left[\mathbf{z} + \varepsilon f(\mathbf{z}(t), t) + \mathcal{O}(\varepsilon^2) + \mathcal{O}(\varepsilon^3) + \ldots\right]\right) && (11)\\
&= -\text{tr}\left(\lim_{\varepsilon \to 0^+} \frac{\partial}{\partial \varepsilon}\left[I + \varepsilon \frac{\partial f(\mathbf{z}(t), t)}{\partial \mathbf{z}} + \mathcal{O}(\varepsilon^2) + \ldots\right]\right) && (12)\\
&= -\text{tr}\left(\lim_{\varepsilon \to 0^+} \left[\frac{\partial f(\mathbf{z}(t), t)}{\partial \mathbf{z}} + \mathcal{O}(\varepsilon) + \ldots\right]\right) && (13)\\
&= -\text{tr}\left(\frac{\partial f(\mathbf{z}(t), t)}{\partial \mathbf{z}}\right) && (14)
\end{aligned}$$

3.2 Continuous Backpropagation and Adjoint Equation

Let $\mathbf{z}(t)$ follow the differential equation:

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta) \tag{15}$$

Define the adjoint state:

$$\mathbf{a}(t) = \frac{d\mathcal{L}}{d\mathbf{z}(t)} \tag{16}$$

Then it follows the differential equation:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{17}$$

The adjoint state is the gradient with respect to the hidden state at a specified time t. In standard neural networks, the gradient of a hidden layer h_t depends on the next layer h_{t+1} via the chain rule:

$$\frac{d\mathcal{L}}{dh_t} = \frac{d\mathcal{L}}{dh_{t+1}} \cdot \frac{dh_{t+1}}{dh_t} \tag{18}$$

For continuous hidden states, the transformation after an ε time change is:

$$\mathbf{z}(t+\varepsilon) = \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta)dt + \mathbf{z}(t) = T_\varepsilon(\mathbf{z}(t), t) \tag{19}$$

Applying the chain rule:

$$\frac{d\mathcal{L}}{\partial \mathbf{z}(t)} = \frac{d\mathcal{L}}{d\mathbf{z}(t+\varepsilon)} \cdot \frac{d\mathbf{z}(t+\varepsilon)}{d\mathbf{z}(t)} \Rightarrow \mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \cdot \frac{\partial T_\varepsilon(\mathbf{z}(t), t)}{\partial \mathbf{z}(t)} \tag{20}$$

Now we compute the derivative:

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon)-\mathbf{a}(t)}{\varepsilon} \tag{21}$$

$$= \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon)-\mathbf{a}(t+\varepsilon)\cdot \frac{\partial}{\partial \mathbf{z}(t)}T_\varepsilon(\mathbf{z}(t))}{\varepsilon} \tag{22}$$

$$= \lim_{\varepsilon\to 0^+} \frac{\mathbf{a}(t+\varepsilon)\left[I-\left(I+\varepsilon\frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}+\mathcal{O}(\varepsilon^2)\right)\right]}{\varepsilon} \tag{23}$$

$$= \lim_{\varepsilon\to 0^+} -\mathbf{a}(t+\varepsilon)\cdot\left(\frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}+\mathcal{O}(\varepsilon)\right) \tag{24}$$

$$= -\mathbf{a}(t)\cdot\frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)} \tag{25}$$

This ODE for the adjoint state is solved backward in time. The initial condition is specified at the final time point:

$$\mathbf{a}(t_N) = \frac{d\mathcal{L}}{d\mathbf{z}(t_N)} \tag{26}$$

And the value at t_0 is computed via:

$$\mathbf{a}(t_0) = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)\cdot\frac{\partial f(\mathbf{z}(t),t,\theta)}{\partial \mathbf{z}(t)}dt \tag{27}$$

If $\mathcal{L}$ depends on intermediate time points as well (e.g., $t_1, t_2, \ldots$), the adjoint step is repeated on each interval $[t_{i-1}, t_i]$ in reverse, and gradients are accumulated..

3.3 Gradients with Respect to Parameters

To compute $\frac{d\mathcal{L}}{d\theta}$, we use:

$$\frac{d\mathcal{L}}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T\frac{\partial f(\mathbf{h}(t),t;\theta)}{\partial\theta}dt \tag{28}$$

This follows from the identity for total derivatives and interchanging the order of differentiation and integration.

Additional gradients (e.g., w.r.t. θ) can be computed as:

$$\frac{d\mathcal{L}}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^T\frac{\partial f(\mathbf{h}(t),t;\theta)}{\partial\theta}dt \tag{29}$$

The adjoint sensitivity method reduces the calculation time taken when used for highly complex codes, and is advised to be used when dealing with a very large dataset. However, in the implementation of the code, we have omitted this method, for smaller datasets like our spiral dataset, the adjoint method may take as much time or even be slower than regular backpropagation.

4 My Project

The aim of my experiment was to use the *torchdiffeq* package and use it to train a model off of a spiral dataset (x = $\cos t$, y = $\sin t$) and test it on different spirals, by altering the spiral and damping rate, to see if it learns to pick up on spiral dynamics to generate any spiral by calculating its gradient at each point.

The hyperparameters in setting up this project were the dimension of the hidden layer(256), the choice of the ODE solver at each step of training, validation and testing (rk4 was chosen due to GPU constraints as Dopri-5 requires more computation time), relative tolerance (*rtol*) and absolute tolerance (*atol*) (chosen as 10^-9 and 10^-10 respectively to maintain accuracy), the number of epochs, the patience (number of epochs trained on with no improvement in validation loss before an early stop) and learning rate (10^-3). The method of choosing these was through gridsearch.

5 Code Snippets

5.1 ODE Function Definition

```
class GeneralODEFunc(nn.Module):
    def __init__(self, input_dim, param_dim=0, hidden_dim=128, n_layers=3)
        :
        super().__init__()
        self.input_dim = input_dim
        self.param_dim = param_dim

        layers = []
        layers.append(nn.Linear(input_dim + param_dim + 1, hidden_dim))
        layers.append(nn.Tanh())
        layers.append(nn.Dropout(0.1))

        for _ in range(n_layers - 2):
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(nn.Tanh())
            layers.append(nn.Dropout(0.1))

        layers.append(nn.Linear(hidden_dim, input_dim))

        self.net = nn.Sequential(*layers)
        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)

    def forward(self, t, y_with_params):
        y, params = y_with_params[:, :self.input_dim], y_with_params[:,
            self.input_dim:]
        t_expanded = t.expand(y.shape[0], 1)
        inp = torch.cat([y, params, t_expanded], dim=1)
        return self.net(inp)
```

5.2 Parameter integration helper

```
def integrate_with_params(func, y0, params, t, input_dim, **kwargs):

  if y0.dim() == 1: y0 = y0.unsqueeze(0)
  if params.dim() == 1: params = params.unsqueeze(0)

  if y0.shape[0] > 1 and params.shape[0] == 1:
    params = params.expand(y0.shape[0], -1)

  y0_aug = torch.cat([y0, params], dim=1) # Shape: (batch_size, input_dim
     + param_dim)

  pred_y_aug = odeint(func, y0_aug, t, **kwargs) # Shape: (T, batch_size,
     input_dim + param_dim)
  return pred_y_aug[:, :, :input_dim]
```

5.3 Model Integration and Training

```
pred_y = integrate_with_params(self.func, y0_train, params, t_train,
   input_dim, ...)
# Training loop omitted for brevity
```

6 Results

In this particular exercise, we hoped to train a model on a spiral dataset and to use that in generating other, different spirals. One may note in the results that when tested on a completely different spiral to that it was trained on (as in *fig-2*), it starts badly, but does better towards the end of the spiral. This could mean that if it were trained using the more computationally expensive Doprimand - 5 method and on a more varied dataset, the model might have done far better than in the status quo.

7 Conclusions and Future Enhancements

Neural ODEs provide a flexible and interpretable framework for modeling continuous-time dynamics. With the adjoint sensitivity method, these models become scalable and differentiable, opening up new possibilities in time-series modeling and physics-inspired learning.

When looking at the results, one can see how the model does substantially worse on the testing model than the training set, and the root cause for this is overfitting and lack of generalization.

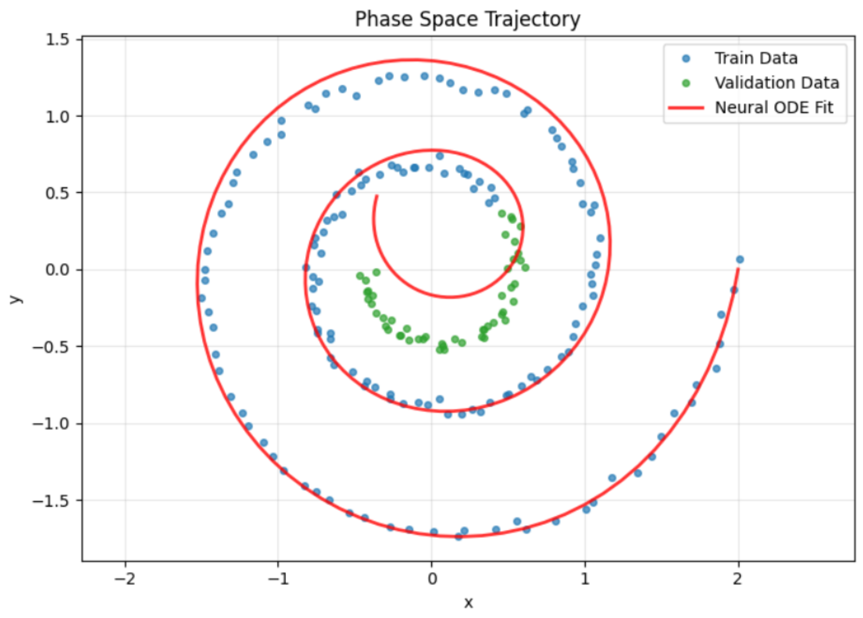

Figure 1: Neural ODE trajectory vs. actual data during training and validation.

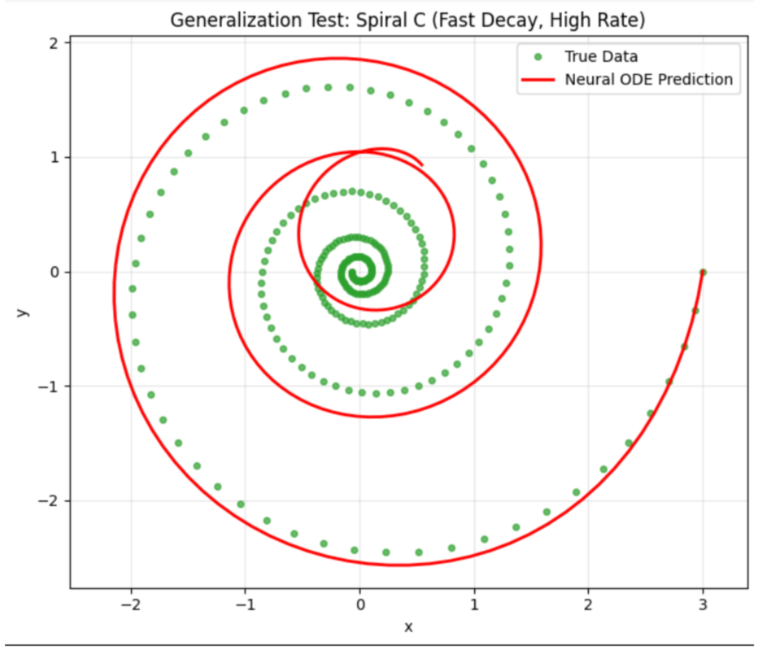

Figure 2: Neural ODE predictions vs true spiral trajectories during testing on a completely different spiral.

The core issue is that the model has memorized the path of the training data rather than learning the general physical laws of a spiral. A spiral can be described by the system of ordinary differential equations (ODEs): $\frac{dx}{dt} = ax - by; \frac{dy}{dt} = bx + ay$. The model performed well on the specific 'a' and 'b' from the training data but did not do as well when presented with a new set of parameters.

Future suggestions

1. Data-Centric Enhancements (Most Recommended) The most effective way to improve generalization is to train the model on a more diverse dataset. Instead of training on a single spiral trajectory, generate training data from a wide variety of spirals by randomizing their underlying parameters (‘a‘ and ‘b‘ in the equations above); vary Decay Rates, Angular Velocity, Initial Conditions. By seeing many different examples, the Neural ODE will be forced to learn the relationship between the system's state ‘(x, y)‘ and its derivative ‘$(dx/dt, dy/dt)$‘ in a way that is independent of a single trajectory. 2. Physics-Informed Priors: Since the general form of the ODE is known, one can bake this knowledge into the model. Instead of having the network learn an arbitrary function f, it could learn the parameters a and b of the linear system, perhaps as a function of some input conditions. This is a form of a Physics-Informed Neural Network (PINN) and can lead to excellent generalization. By implementing these suggestions, particularly by diversifying training data, the Neural ODE should evolve from simply memorizing a path to truly learning the underlying system of equations that governs spiral dynamics.

References

- Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). Neural ordinary differential equations. *NeurIPS.*
- Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software.*
- torchdiffeq: https://github.com/rtqichen/torchdiffeq
- Sinai, J. (2019). Understanding Neural ODE's. *jontysinai.github.io/posts/*
- Gibson, K. (2018). Neural networks as Ordinary Differential Equations. *rkevingibson.github.io/*
- Hu, P. (2018). A note on the adjoint method for neural ordinary differential equation network *arxiv.org/2402.15141v1.*
- Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations *NASA Reference Publication 1327*
- Nair, A., Dr.Kadam, S.D, Dr.Menon, R., Shende, P.C, (2024). Neural Differential Equations: A Comprehensive Review and Applications. *Advances in Nonlinear Variational Inequalities*