ECE 228 Spring 2025 Lecture 11: Neural ODEs

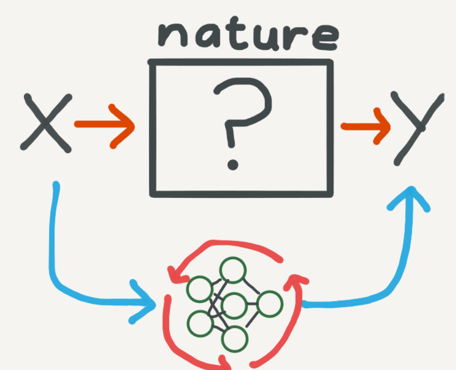
Luke Bhan

May 8, 2025

Review of the classic machine learning pipeline

Typically, the main paradigm for (supervised) machine learning is the following.

1. Collect *n* pairs of data with their measured output labels.
2. Take each input pair, and perform matrix multiplication on that input many times sprinkling in some nonlinearity
3. Backpropagate on the over-parameterized matrices according to the difference between the model output and the target.



On matrix multiplication based models

Matrix multiplication combined with carefully chosen nonlinearities (e.g. transformers) can produce remarkable results (just look at ChatGPT). But is this truly the best, or only, long-term approach to building intelligent systems? Should we think about other ways?

- LLMs are super expensive to train both from an energy and monetary standpoint.
- These things have little interpretability (See the entire AI safety community)
- There is a very fair question about whether our brain's work in this manner and whether this approach is a fair way to *generate new ideas*.

Today, we are going to explore a new paradigm - Neural ODEs - for training machine learning models that mimic the same structure we use to model the physical world.

Motivation for studying Neural ODEs

- They introduce a new way to think about ”machine learning” by introducing a *continuous* time structure that is closer to physical principles of the real world
- Neural ODEs enable efficient physical solvers; thus they have been used in weather/seismic forecasting, fluid dynamics, and even chemical kinetics
- Lastly, if you just want to impress your friends, Neural ODEs are the backbone of almost all generative AI algorithms

A quick review of ordinary differential equations (ODEs)

Recall, *dynamical systems* are the mathematical language for describing how kinematics in our world behave! Typically, they arise in the form of ordinary differential equations:

$$\frac{dz(t)}{dt} = f(z(t), t)\,, \quad z(0) = \bar{z}_0\,.$$

Why study these objects? Well, they describe pretty much everything:

- Robot kinematics
- Weather and climate modeling
- Epidemiology (spread of diseases - essential for modeling the spread of COVID-19)
- The stock market (for those who want to buy a Yacht)
- Acoustics in music (and are the backbone for AI generative music!)

Crash course on what's important in ODEs

ODEs describe how a system evolves over time. Thus, they model how a system behaves given an initial state.

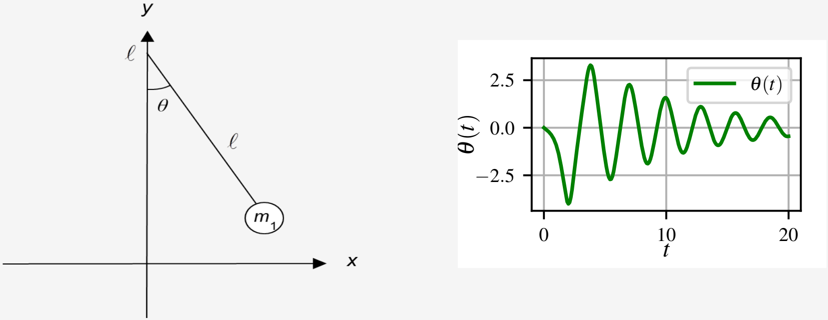


Figure: Pendulum system and its corresponding trajectory as an example of an ODE.

Crash course on what's important in ODEs

ODE solutions are *unique*. This may appear to be just some mathematical property, but is extremely useful when we want to design machine learning models that are *interpretable* (More on this later).

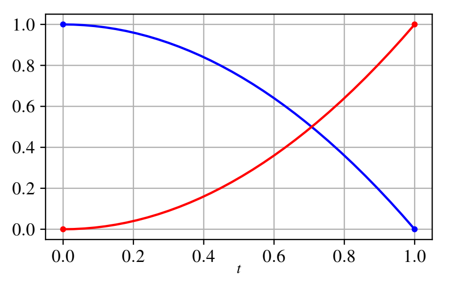


Figure: Consider the red and blue lines to be two ODE trajectories. Then, since they cross, their underlying dynamics f *must be different!*

Crash course on what's important in ODEs

Numerically, we can solve almost any kind of ODE through finite differencing or Runge-Kutta methods. However, these are computationally expensive for hard (stiff) problems!

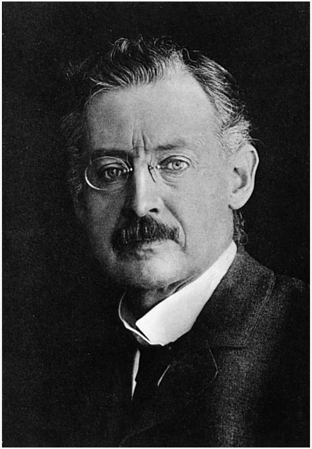


Figure: On the left is pictured Carl Runge and on the right is Martin Kutta.

Euler Integration: Part 1

Recall that ODEs are given in *continuous representations*:

$$\frac{dz(t)}{dt} = f(z(t), t), \quad z(0) = \bar{z}_0 .$$

Yet, computers **require** *discrete representations.* Thus, we need an approximation for $\frac{dz(t)}{dt}$!

The simplest method for Numerical approximation of this is to use a first-order Taylor expansion:

$$z(t + \delta t) = z(t) + \delta t \times \frac{dz(t)}{dt} + \mathcal{O}(\delta t^2),$$

Then, with some algebra, we obtain

$$\frac{dz(t)}{dt} = \frac{z(t + \delta t) - z(t)}{\delta t} + \mathcal{O}(\delta t^2).$$

Euler Integration: Part 2

Plugging

$$\frac{dz(t)}{dt} = \frac{z(t+\delta t) - z(t)}{\delta t} + \mathcal{O}(\delta t^2)\,.$$

into our continuous representation

$$\frac{dz(t)}{dt} = f(z(t), t)\,, \quad z(0) = \bar{z}_0\,.$$

and rearranging for $z(t+\delta t)$ yields

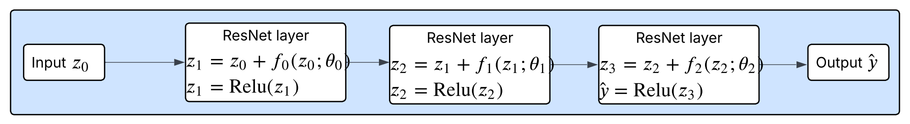
$$z(t+\delta t) \approx z(t) + \delta t \times f(z(t), t)\,,$$

Since we know $z(0)$, we can use this to estimate $z(t+\delta t)$ recursively in a computer as long as δt is small (ensuring our approximation error is small enough).

This is known as Euler's method.

ResNet to Neural ODE

Fundamentally, ODEs enable us to design new machine learning architectures.
To explain this, let us revisit the ResNet:



Input z_0

ResNet layer
$z_1 = z_0 + f_0(z_0;\theta_0)$
$z_1 = \mathrm{Relu}(z_1)$

ResNet layer
$z_2 = z_1 + f_1(z_1;\theta_1)$
$z_2 = \mathrm{Relu}(z_2)$

ResNet layer
$z_3 = z_2 + f_2(z_2;\theta_2)$
$\hat{y} = \mathrm{Relu}(z_3)$

Output $\hat{y}$

Recall Euler's method was of the form:

$$z_{k+1} = z_k + \delta t \times f(z_k, k)$$

This ResNet structure closely resembles Euler's method! Thus, one can think of a ResNet as *learning the Euler approximation of an ODE.*

Remember, what were the con's of ResNet

- Requires many layers (Recall the popular ResNet 50 architecture has of course, 50 layers)
- Due to the many layers, backprop is expensive!
- We observed in ResNet that more layers is better. "What about having infinite many layers?"

Idea: Instead of learning discrete networks $f_1, f_2, \ldots, f_n$, why don't we just learn a single ODE representation?

That is, learn the ODE dynamics $\hat{f}$ where we have

$$\frac{dz(t)}{dt} = \hat{f}(z(t), t; \theta)\,, \quad z(0) = \bar{z}_0$$

Then, the input of your neural network becomes $\bar{z}_0$ and the output is $z(T)$ for some chosen time T. What do you think T represents in a sense?

Neural ODEs

T is analogous to the network depth in ResNet! Thus, a larger T lends itself to capture more rich dynamics. This architecture is called a **Neural ODE**!

However, two things are not obvious:

- Say we have some $\hat{f}$ given by a MLP. Then, the ODE we have is in a continuous representation. How do we actually compute our output $z(T)$?
- Let say we compute $z(T)$, how to do backpropagation?
- Are either of these steps actually faster then just using a ResNet with many layers?

How to compute $z(T)$?

Answer: ResNet was akin to using Euler's method - *the most naive ODE solver*. Luckily, numerical analysts have been working on simulating ODEs for years and thus, we can use an extremely accurate and fast ODE solver such as Dopri 5 (A RK solver with an adaptive δt) to generate $z(T)$.

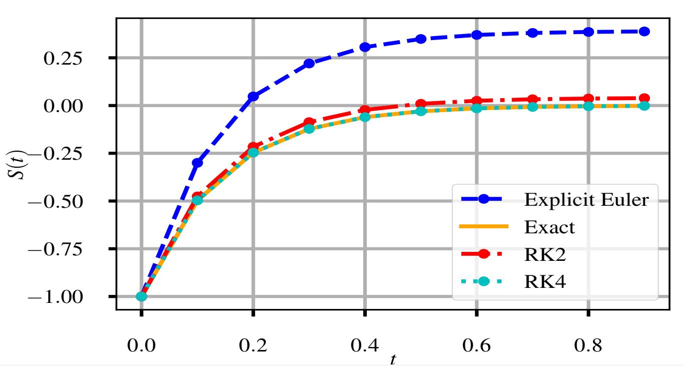


Figure: Comparison of trajectories $S(t)$ over time with various solvers. (Modified from a UCSD Phys 141 course.)

How to back propagate through our chosen ODE solver?

Let us consider the standard loss function case:

$$\mathcal{L}(y, \hat{y}) = \|y - \hat{y}\|_2\,, \text{ where } \hat{y} = z(T)\,.$$

Then, using the chain rule, we need to compute $\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial z(T)} \frac{\partial z(T)}{\partial \theta}$ where the second term requires us to compute the backprop through each calculation in our ODE solver. Explicitly, let us consider the example where we use the *simplest* approach Euler's to simulate our PDE. Then, we have that

$$\begin{aligned}
z(T) &= z(T - \delta t) + \delta t \cdot f(z(T - \delta t), t - \delta t; \theta)\,, \\
z(T - \delta t) &= z(T - 2\delta t) + \delta t \cdot f(z(T - 2\delta t), t - 2\delta t; \theta)\,, \\
&\vdots \\
z(\delta t) &= z(0) + \delta t \cdot f(z(0), 0; \theta)\,, \\
z(0) &= \bar{z}_0 \text{ (given as input)}\,.
\end{aligned}$$

Backpropagation continued...

To compute our gradients, recall we needed to compute:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial z(T)} \frac{\partial z(T)}{\partial \theta}$$

and to compute the second term, plugging in our Euler discretization:

$$z(T) = z(T - \delta t) + \delta t \cdot f(z(T - \delta t), t - \delta t; \theta)\,,$$

we obtain,

$$\frac{\partial z(T)}{\partial \theta} = \frac{\partial z(T - \delta t)}{\partial \theta} + \delta t \frac{\partial f(z(T - \delta t), t - \delta t; \theta)}{\partial \theta}$$

Now, notice the second term is a multi-variable function with $z(T - \delta t)$ and f depending on θ. Thus, we apply the chain rule yielding

$$\frac{\partial f(z(T - \delta t), t - \delta t; \theta)}{\partial \theta} = \frac{\partial f}{\partial z(T - \delta t)} \times \frac{\partial z(T - \delta t)}{\partial \theta} + \frac{\partial f}{\partial \theta}$$

Backpropagation continued...

Resulting in the final calculation

$$\frac{\partial z(T)}{\partial \theta} = (\text{Identity} + \delta t \cdot \frac{\partial f}{\partial \theta}) \cdot \frac{\partial z(T - \delta t)}{\partial \theta} + \delta t \cdot \frac{\partial f}{\partial \theta}$$

Notice that, we now have a recursion where we need to compute $\frac{\partial z(T-\delta t)}{\partial \theta}$ and repeat this all the way until time 0. We also need to repeat a similar calculation for

> As you can see, for even the simplest integration method (Euler's, of which in practice one probably uses an RK method) this is both expensive in memory and computational time for large T which controls the expressivity of our network!

There must be a better way...

Adjoint method

To showcase an alternative approach to compute $\frac{\partial \mathcal{L}}{\partial \theta}$, let us consider treating θ as following the constant ODE

$$\frac{d\hat{\theta}(t)}{dt} = 0\,, \hat{\theta}(0) = \theta$$

which of course has a solution $\hat{\theta}(t) = \theta$ for all t. Thus, we can consider the augmented ODE:

$$\frac{d}{dt}\begin{bmatrix} z \\ \theta \end{bmatrix}(t) = \begin{bmatrix} f(z(t), t; \theta) \\ \theta \end{bmatrix}$$

Let us define the function

$$\begin{bmatrix} a \\ a_\theta \end{bmatrix}(t) = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z(t)} \\ \frac{\partial \mathcal{L}}{\partial \theta(t)} \end{bmatrix}$$

which is exactly the gradients we need to compute for our backpropagation.

Adjoint method continued...

We call the state $a(t)$ the **adjoint**:

$$\begin{bmatrix} a \\ a_\theta \end{bmatrix}(t) = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial z(t)} \\ \frac{\partial \mathcal{L}}{\partial \theta(t)} \end{bmatrix}$$

The key idea is that, we can differentiate the function $[a, a_\theta]$ with respect to time (apply chain rule) obtaining an **ODE**:

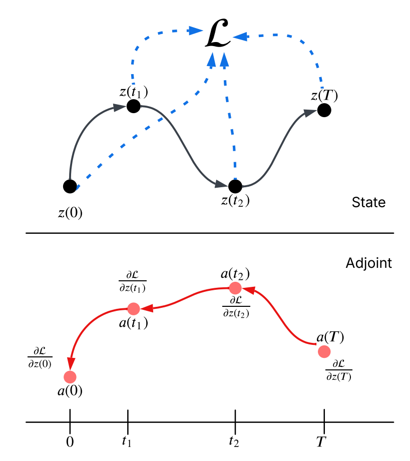
$$\frac{d}{dt}\begin{bmatrix} a \\ a_\theta \end{bmatrix}(t) = \begin{bmatrix} -a(t)^T \frac{\partial f(z(t),t;\theta)}{\partial z(t)} dt \\ -a(t)^T \frac{\partial f(z(t),t;\theta)}{\partial \theta(t)} dt \end{bmatrix}$$

$$a(T) = \frac{\partial \mathcal{L}}{\partial z(T)} \text{(easy to compute)}$$

$$a_\theta(T) = 0 \ (\mathcal{L} \text{ has no } \theta(T) \text{ dependence})$$

Thus to find $a_\theta(0)$, which are the required gradients, we just need to solve the following *backwards* ODE using our favorite integration method!

Visualization of the adjoint method



$\mathcal{L}$
$z(t_1)$
$z(T)$
$z(0)$
$z(t_2)$
State
Adjoint
$\frac{\partial \mathcal{L}}{\partial z(t_1)}$
$a(t_2)$
$\frac{\partial \mathcal{L}}{\partial z(t_2)}$
$a(t_1)$
$a(T)$
$\frac{\partial \mathcal{L}}{\partial z(0)}$
$a(0)$
$\frac{\partial \mathcal{L}}{\partial z(T)}$
0
t_1
t_2
T

- In the top, we compute the forward pass ODE to obtain $z(T)$.
- To get our gradients, we solve a separate ODE backwards for $a(0)$ which gives us $a_\theta(0) = \frac{\partial L}{\partial \theta}$.

Advantages of the adjoint method

Turn's out the adjoint method is much faster since:

- It avoids storing the full trajectory of $[a, a_\theta](t)$, enabling constant memory.
- We can reuse standard ODE solvers (e.g., Runge–Kutta) to compute gradients during the backward pass, avoiding costly backpropagation through each solver step.

Can you think of any disadvantages?

It assumes the backwards ODE is not stiff or numerically instable which is not necessarily guaranteed!

A new neural network architecture: Neural ODEs summarized

- Neural ODEs are a continuous formulation of ResNets which learn the dynamics f of an ODE:

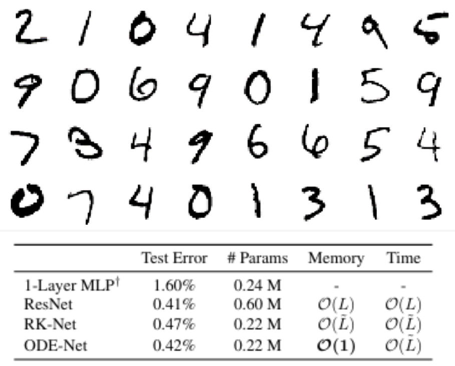
$$\frac{dz(t)}{dt} = f(z(t), t; \theta), \quad z(0) = \bar{z}_0$$

- Unlike ResNets, they don't require many layers, but just a larger T to improve expressiveness
- Furthermore, the adjoint method can be used to quickly compute backpropagation through the ODE solver.
- In essence, by using our understanding of dynamical systems, we developed a *more efficient* neural network architecture.

Do they really work in practice?

Turn’s out they are really useful...

When first introduced in 2018, achieved SOTA over ResNet on MNINST classification.



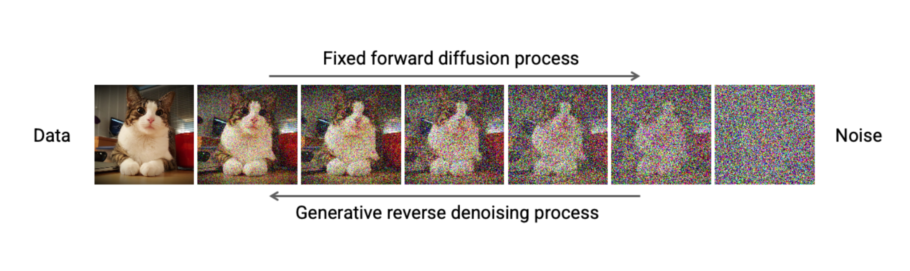
	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(\tilde{L})$

Figure: MNIST dataset example and Neural ODE performance compared to ResNet.

Since... they have taken over generative modeling

Generative modeling (the entire field is controlled by some form of neural ODE now):

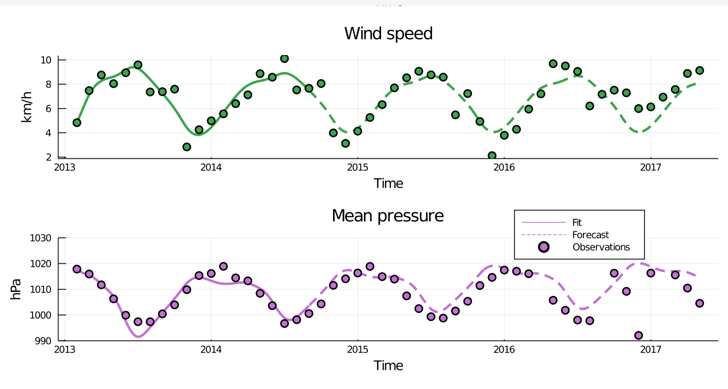
- Continuous normalizing flows (older)
- Diffusion process can be rewritten as a stochastic differential equation and thus diffusion models are a form of Neural ODE
- Using the idea of neural ODEs, flow matching has now emerged as the go to method for generative models which similarly uses Neural ODEs


Fixed forward diffusion process
Data
Noise
Generative reverse denoising process

Other applications...

- Learning physical systems from data (fluids, chemical reactions, COVID-19 dynamics, etc.)
- Learning based control
- Time-series modeling

For example, they can learn weather patterns:



Practical considerations when training Neural ODEs

- The core neural network is typically an MLP, but also can be a CNN or fixed length transformer.
- Typically best to initialize the neural network with a small vector field close to 0 to avoid backprop issues in the adjoint solve
- Technically activation functions should be C^1 to ensure the solution for the adjoint exists, but ReLU still works well in practice.
- For classification, one will typically take the output layer $z(T) \in \mathbb{R}^d$ and project to a scalar with either a linear layer or softmax.

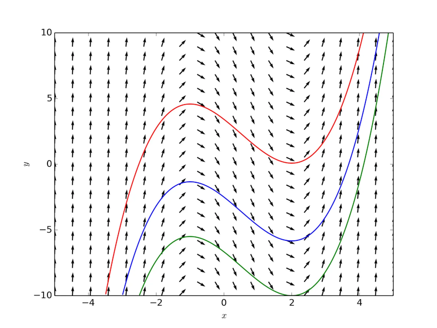
$$\frac{dz(t)}{dt} = f(z(t), t; \theta), \quad z(0) = \bar{z}_0$$

$$\text{Output} = g(z(T)), \text{where } g \text{ is an MLP}.$$

Interpretability of Neural ODEs

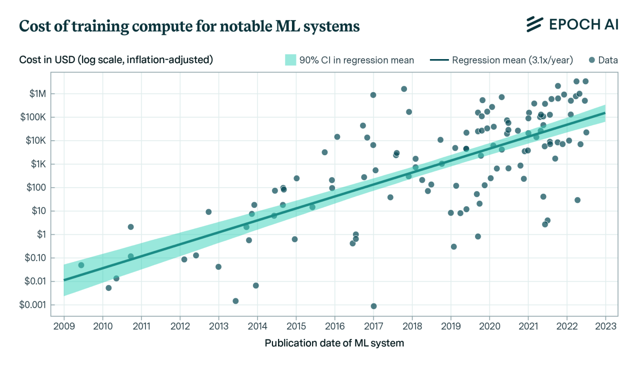
While Neural ODEs are not directly interpretable, their ODE structure gives a good playground to build interpretable dynamics.
Advantages include:

- Visualize the vector field dynamics
- Understand their behavior through the lens of dynamical systems characterizing components such as stability, fixed-points, regions of attraction, etc.
- Because NeuralODEs typically have less parameterization, we can explore the direct effect of weight-matrix perturbations



Limitations of Neural ODEs

- The structure of an ODE is an inherent limitation in expressiveness (Explore in Homework 3)
- Still suffers from gradient issues with large T due to the backward ODE solve being stiff or challenging.
- Scaling to higher dimensional problems: Due to the use of an ODE solver, it can be computational (and thus monetarily) expensive to train a neural ODE at scale.



Cost of training compute for notable ML systems

Extensions and future reading

- Augmented Neural ODEs (See Homework 3)
- Stabilizing Neural ODEs `https://arxiv.org/abs/1906.02355`
- Neural ODEs that continuously receive new input (Controlled Neural ODEs, Latent Neural ODEs) `https://arxiv.org/abs/2005.08926`, `https://arxiv.org/abs/1907.03907`
- Robustness of Neural ODEs to attacks `https://arxiv.org/abs/1910.05513`