A primer on neural ODEs

Patrick Kidger
Google X

Me:

- **PhD on neural differential equations.**
- **Got hired at Google X to apply NDEs to biotech.**
- **Now doing scientific ML + open-source software!**

01

From traditional parameterised modelling to neural ODEs

ODEs are really really (really) good models.

🐰(0) = 2

🐱(0) = 1

$$\frac{d🐰}{dt} = 🐰 - 🐰🐱$$

$$\frac{d🐱}{dt} = -🐱 + 🐰🐱$$

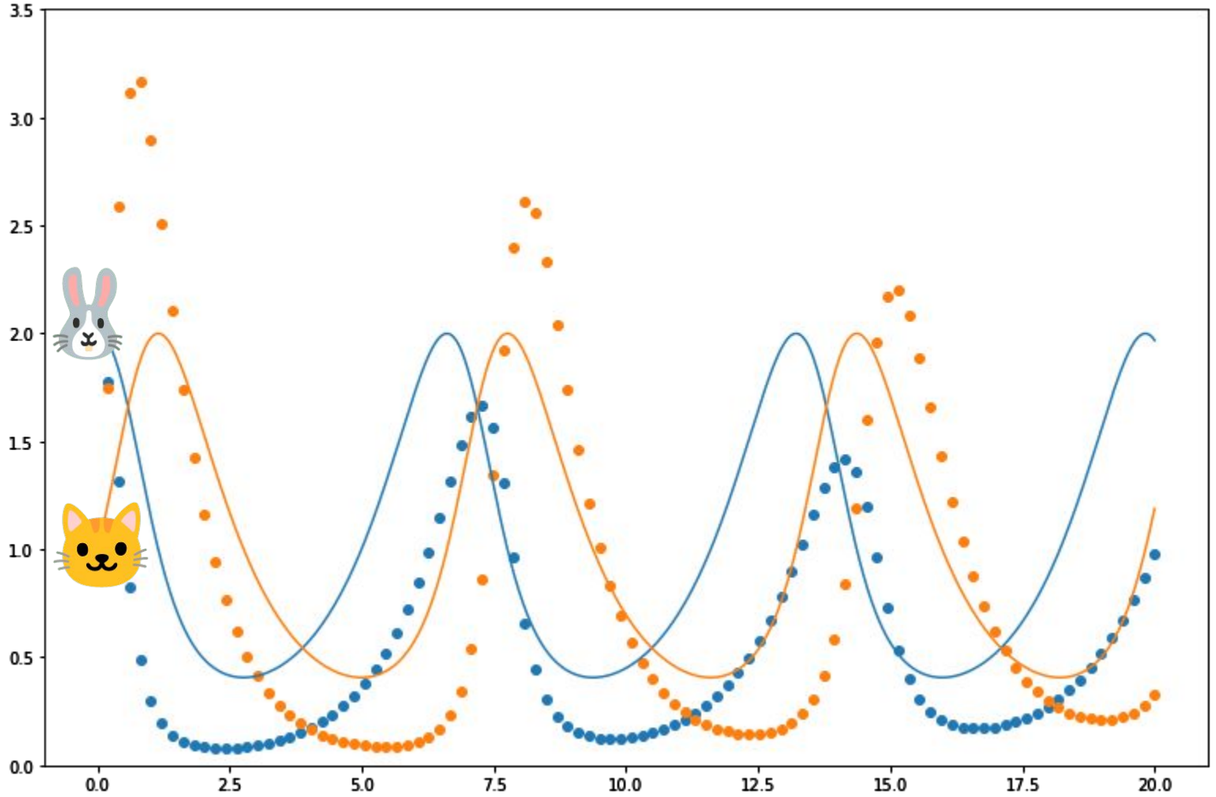
Good for:

- Population modelling;
- Motion of the planets;
- Structural integrity of a bridge;
- Fluid dynamics;
- …

Simple idea: just write down how a value:

- Starts
- And how it changes over time.

Lotka—Volterra (Predator–Prey)



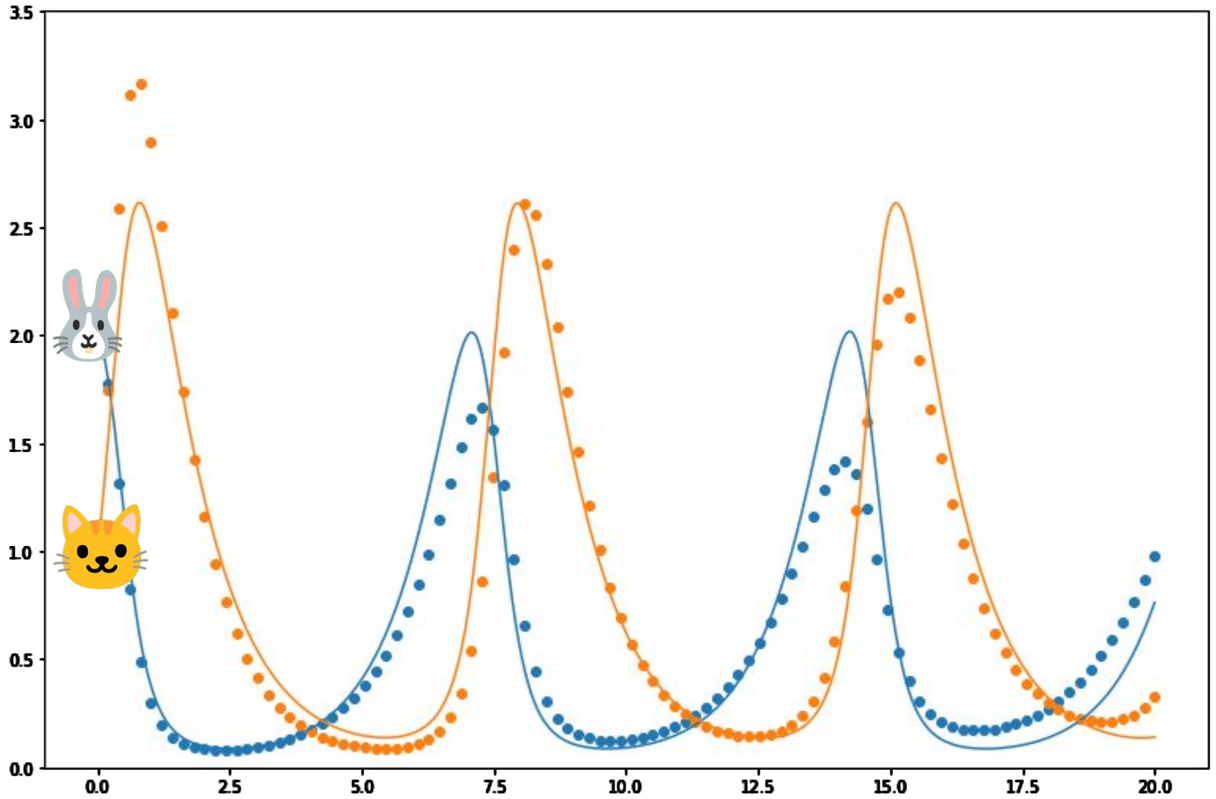
Parameterised ODEs

$$🐰(0) = 2 \qquad \frac{d🐰}{dt} = \alpha 🐰 - \beta 🐰 🐱$$

$$🐱(0) = 1 \qquad \frac{d🐱}{dt} = -\gamma 🐱 + \delta 🐰 🐱$$

- Initialise
  α=1
  β=1
  γ=1
  δ=1

- Set up a loss function: (data - model)^2
- Backpropagate (through the ODE solver)
- Train via gradient descent

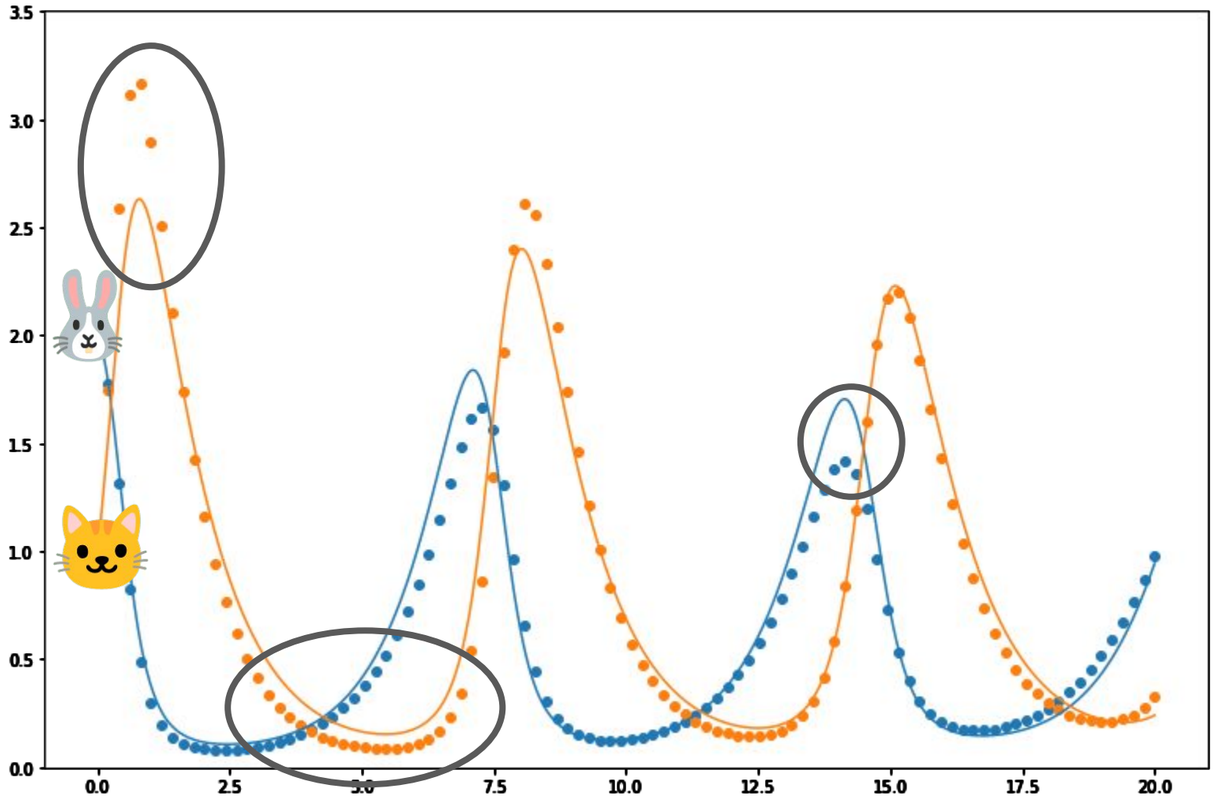


Neural ODEs

🐰(0) = 2 $\quad d🐰/_{dt} = \alpha 🐰 - \beta 🐰 🐱 + NN_\theta(🐰, 🐱)$

🐱(0) = 1 $\quad d🐱/_{dt} = -\gamma 🐱 + \delta 🐰 🐱$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$
- Set up a loss function: $(\text{data} - \text{model})^2$
- Backpropagate (through the ODE solver)
- Train via gradient descent

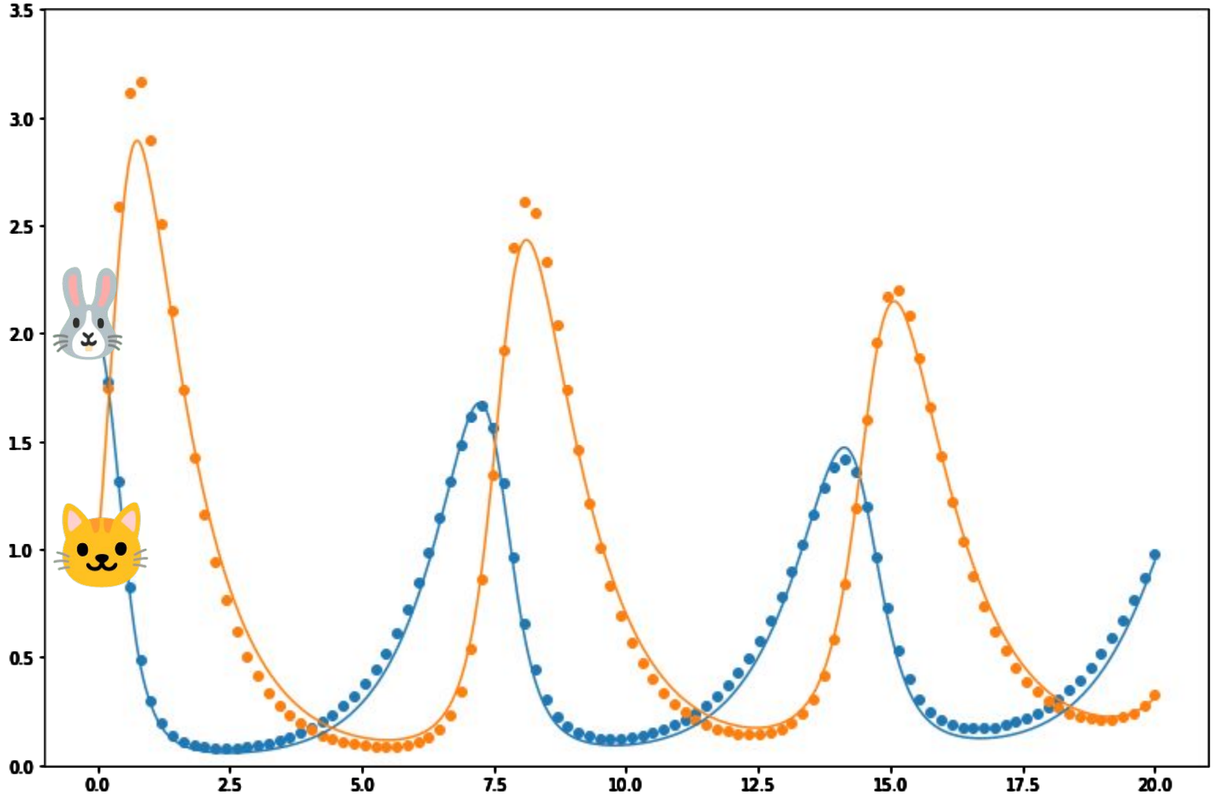


Neural ODEs: version 2

- Initialise
  α=1
  β=1
  γ=1
  δ=1
  $\theta \sim N(0, \sigma^2)$
- Set up a loss function: $(\text{data} - \text{model})^2$
- Backpropagate (through the ODE solver)
- Train via gradient descent

🐰(0) = 2 $\quad d🐰/_{dt} = \alpha 🐰 - \beta 🐰 🐱$

🐱(0) = 1 $\quad d🐱/_{dt} = -\gamma 🐱 + \delta 🐰 (🐱 + NN_\theta(🐰, 🐱))$



Summary of part 1:

- Neural ODE trained in the same way as the mechanistic ODE.
  - In both cases: a parameterised vector field trained with backprop+SGD.
  - Easy to implement: software for NN, ODEs, SGD, …

- Keep the physical model if you think it's good.
  - Put the NN on the bit you think isn't well-modelled.
  - Here the NN is tiny: 4 neurons wide, 1 layer deep.
  - Physical parameters trained first; them kept fixed whilst training NN.

- "Neural ODE" ≈ "hybrid ODE" ≈ "universal ODE" ≈ "parameterised ODE"

02 Next steps: tips, tricks, and nitty details

Symbolic regression

Let's try to interpret NN_θ.

$$d🐰/dt = 1.1🐰 - 1.3🐰🐱$$
$$d🐱/dt = -0.98🐱 + 1.6🐰(🐱 + NN_\theta(🐰, 🐱))$$

Generate a dataset of ~10k samples:

$$\{(🐰_1, 🐱_1, y_1), \ldots, (🐰_n, 🐱_n, y_n)\} \text{ with } y_i = NN_\theta(🐰_i, 🐱_i).$$

(Sample $🐰_i$ and $🐱_i$ in your favourite way, e.g. points from typical ODE trajectories.)

Symbolically regress y_i against $(🐰_i, 🐱_i)$:

$$\{\text{PySR, Lasso, ...}\} \Rightarrow y = 🐱^{0.9} - 🐱 + 0.5🐱^2$$

Retrain all parameters differentiably:

$$d🐰/dt = 0.9🐰 - 1.1🐰🐱$$
$$d🐱/dt = -1.2🐱 + 2.1🐰🐱^{0.95}$$

vs SINDy?

SINDy performs Lasso of $d(🐰, 🐱)/dt$ against $(🐰, 🐱)$. Thus it requires an estimate of the derivative. That's not needed here!

Also, we're not constrained to just Lasso: other symbolic regression (e.g. PySR i.e. regularised evolution) can be used.

Conditions + Learnt Initialisations

Recall our general ODE formulation.

$$(🐰, 🐱)(0) = g \qquad d(🐰, 🐱)/dt = f_\theta(🐰, 🐱)$$

Both f and g may depend on additional data, e.g., the latitude L that the 🐰 and 🐱 live at.

$$(🐰, 🐱)(0) = g(L) \qquad d(🐰, 🐱)/dt = f_\theta(🐰, 🐱, L)$$

Recall that an ODE is determined by:

- An initial condition;
- A vector field.

So far we've tried learning the vector field.

$$(🐰, 🐱)(0) = g_\theta(L) \qquad d(🐰, 🐱)/dt = f_\theta(🐰, 🐱, L)$$

We can also learn the initial condition.

Generalised observables

We might observe things that aren't part of the evolving state of our ODE, but which still somehow give us information.

For example, we might observe the amount of rabbit... output 💩.

Now fit this to the data as well.

Even if we don't care about 💩, this can help us fit the 🐰, 🐱 parts better.

Important extreme case: "hidden state".

$$(🐰, 🐱)(0) = g_\theta(L) \qquad d(🐰, 🐱)/dt = f_\theta(🐰, 🐱, L)$$

$$💩 = h_\theta(🐰, L)$$

$$\text{loss} = \|🐰 - 🐰_{data}\|^2 + \|🐱 - 🐱_{data}\|^2 + \|💩 - 💩_{data}\|^2$$

$$y(0) = g_\theta(c) \qquad dy/dt = f_\theta(y, c) \qquad (🐰, 🐱, 💩) = h_\theta(y, c)$$

This is actually the the most typical formulation of a neural ODE!

Tricks-of-the-trade for nODEs

- Continuum learning:
  - first train on the first 5 timepoints of your time series.
  - then use 10 timepoints.
  - then use 15… etc.
  - **This helps avoid local minima in the optimisation landscape.**
- *Tune your hyperparameters*.
  - Change neural network size.
  - Change learning rate.
  - …
  - **This is always necessary to get good results with neural networks.**
- Initialise your neural network (before training) with very very very small weights.
  - **More precisely: start your learnt correction near zero.**

Tricks-of-the-trade for nODEs

- Train the physical parameters first, then freeze them – and only then introduce the neural network.
  - **So the neural network doesn't also capture the known physics.**
- Typically use (recursively checkpointed) discretise-then-optimise.
  - *Not optimise-then-discretise!! This only gives approximate gradients.*
  - **The efficacy of "the continuous adjoint method" has been gradiently overstated.**
- Try different numerical methods.
  - Use a solver that makes more/fewer function evaluations;
  - Use tighter/looser step size tolerances.
  - **Balance training speed with getting good solutions.**

NDEs vs PINNs

NDEs are a *modelling approach*:

- Learn θ in the model $^{dy}/_{dt}(t) = f_\theta(t, y(t))$, so that you match your data.
- Loss function: $\min_\theta \| y(t) - data(t) \|^2$

PINNs are a *numerical method:*

- Learn φ in $y(t)=y_\varphi(t)$, so that it matches the diffeq $^{dy}/_{dt}(t) = f(t, y(t))$.
- Loss function: $\min_\varphi \| ^{dy}/_{dt}(t) - f(t, y(t)) \|^2$

These terms have precise meaning, but the terms often muddled up, as sometimes both techniques are done at the same time!

- Loss function: $\min_{\theta,\varphi}(\| y(t) - data(t) \|^2 + \| ^{dy}/_{dt}(t) - f(t, y(t)) \|^2)$

PINNs are useful when traditional numerical methods fail (high-dimensional PDEs; nonlocal integral operators.). But mostly… just use traditional numerical methods.

NDEs vs residual networks

Consider the neural ODE:

$$\frac{dy}{dt}(t) = f_\theta(t, y(t))$$

Approximate:

$$\frac{dy}{dt}(t) \approx \frac{(y(t + h) - y(t))}{h}$$

Rearrange:

$$y(t + h) \approx y(t) + h\, f_\theta(t, y(t))$$

We've essentially recovered the formulation of a residual network!

$$y_{n+1} = y_n + f_{\theta,n}(y_n)$$

Can expand on this:

- Residual networks ⇔ neural ODEs
- Recurrent networks ⇔ neural CDEs
- Invertible networks, normalising flows ⇔ reversible differential equation solvers
- …

03 Wrap-up: open problems & software

Open problems!

Neural ODEs

- Just applying neural ODEs (+symbolic regression!) to current problems in science.
- Neural ODEs for multiscale dynamics.

Neural {CDEs, SDEs, PDEs, …}

- Neural CDEs accept time series input. These still need some squaring away: handling causality, efficient vector fields, …
  - Connections to RL and control theory?
- Neural SDEs are generative. These are difficult to train stably.
  - Connections to score-based diffusions?
- Neural PDEs!! Mostly not explored.
  - E.g. neural advection-reaction-diffusion equations?
  - Connections to Mixer architectures, some computer graphics applications, …?

On software: here's a quick soapbox moment...

Please write better code!

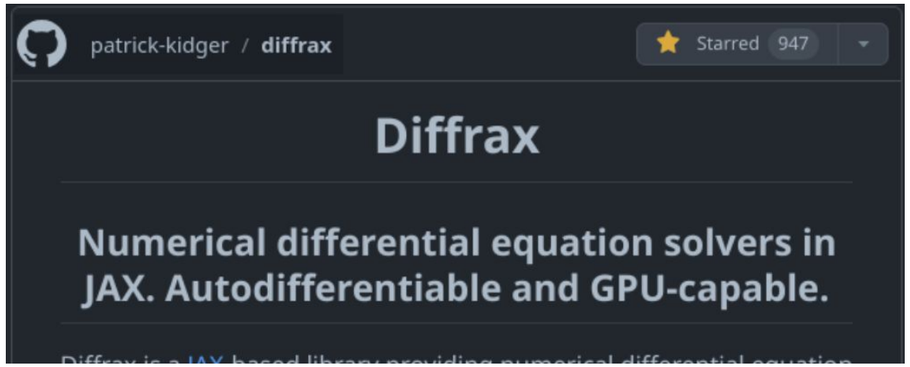
- Format with Black.
- Lint (=check for common errors) with Ruff.
- Type-check with pyright.
- and run them automatically with pre-commit.

(Feel free to steal my config from https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml)

Software

We've got a pretty good JAX ecosystem going now!

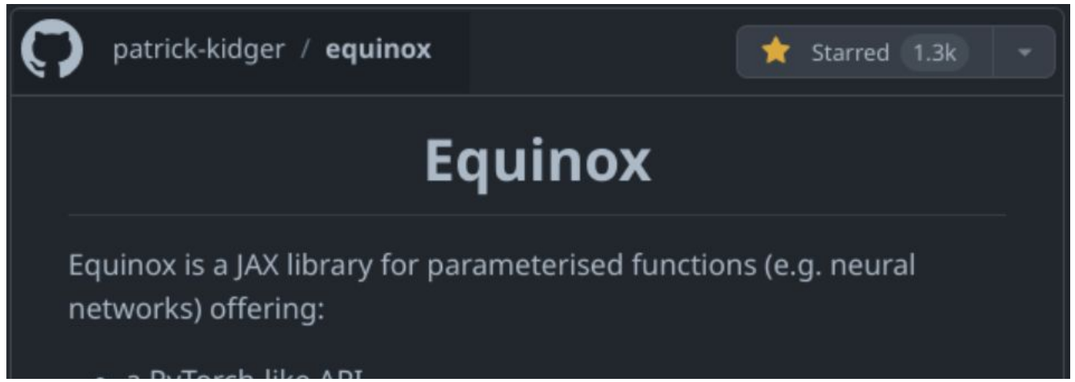
Toy example – solving a small neural ODE.




```python
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                 key=jax.random.PRNGKey(0))

def solve_neural_ode(y0: Float[Array, "4"]) -> Float[Array, "10 4"]:
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: mlp(y)),
                          dfx.Tsit5(), t0=0, t1=1, dt0=0.1, y0=y0
                          saveat=dfx.SaveAt(ts=jax.numpy.linspace(0, 1, 10))
    return sol.ys
```



…see tomorrow for more about JAX!

Also supports PyTorch!

Congratulations!

You are now neural ODE experts :)

Further links:

- The neural ODE example we saw in these slides: https://colab.research.google.com/drive/1ZlK36VgWy1vBjBNXjSUg6Cb-7zeoa3jh

- Reference for this material: Chapter 1, Section 2.2.2, Section 2.3, and Section 6.1 of *On Neural Differential Equations* (https://arxiv.org/abs/2202.02435).

- JAX ecosystem: https://github.com/patrick-kidger/diffrax (+links within)

- Questions? kidger@google.com; twitter.com/PatrickKidger