Neural Differential Equations as a Basis for Scientific Machine Learning

CHRIS RACKAUCKAS

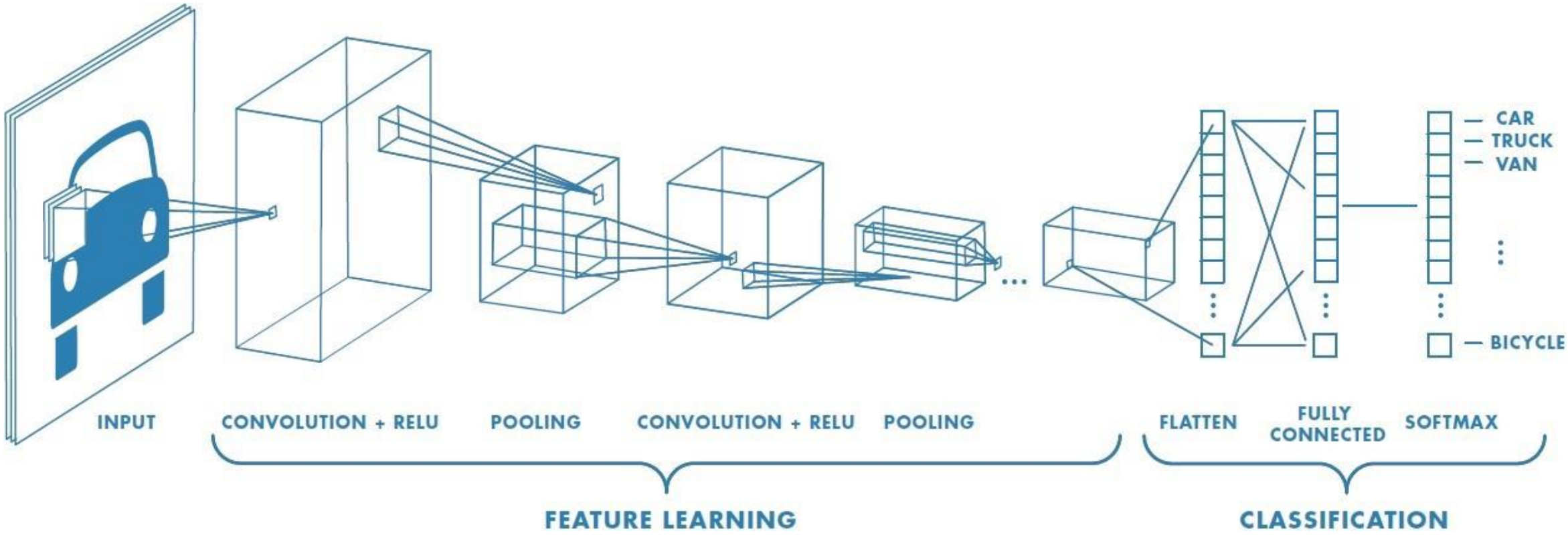
MASSACHUSETTS INSTITUTE OF TECHNOLOGY, DEPARTMENT OF MATHEMATICS

UNIVERSITY OF MARYLAND, BALTIMORE, SCHOOL OF PHARMACY, CENTER FOR TRANSLATIONAL MEDICINE

The major advances in machine learning were due to encoding more structure into the model

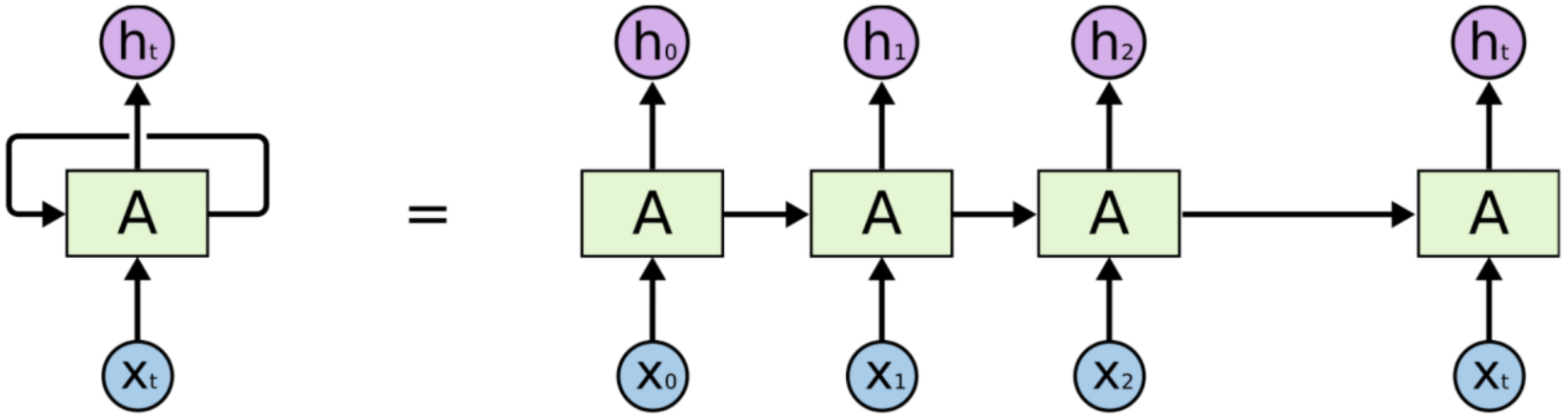
MORE STRUCTURE = FASTER AND BETTER FITS FROM LESS DATA

Convolutional Neural Networks Encode (Spatial) Structure



A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, Sumit Saha

Recurrent Neural Networks Encode Time Dependency Assumptions



Everything you need to know about Recurrent Neural Networks, Prajjwal

Now let's generalize this idea to scientific structures

WHAT'S THE STRUCTURE?

Scientific Structure: X changes as a function of (Y,Z,…)

THIS IS JUST A DIFFERENTIAL EQUATION

Ecology Example: Lotka-Volterra Equations

$$\frac{d\,🐇}{dt} = \alpha\,🐇 - \beta\,🐇🦊$$

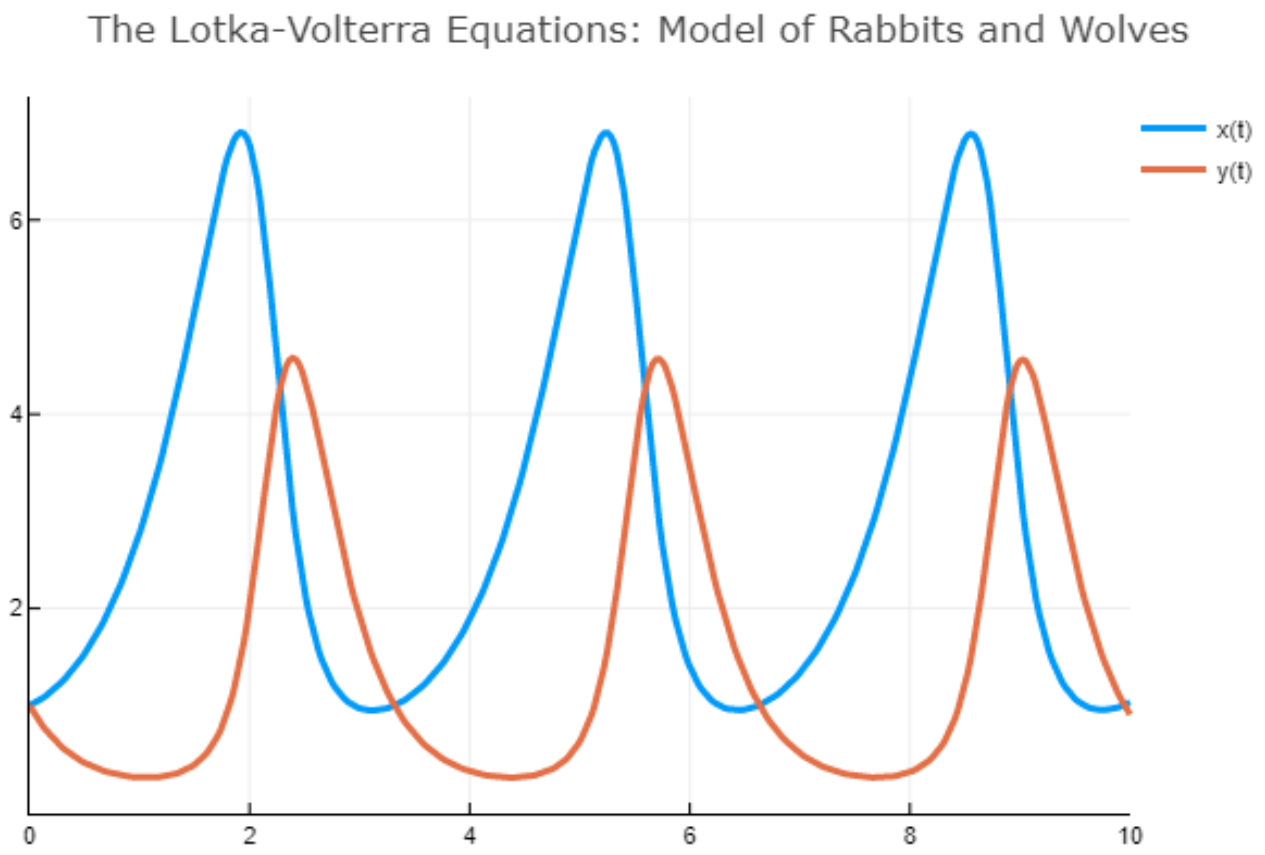
Exponential growth

Gets eaten by wolves

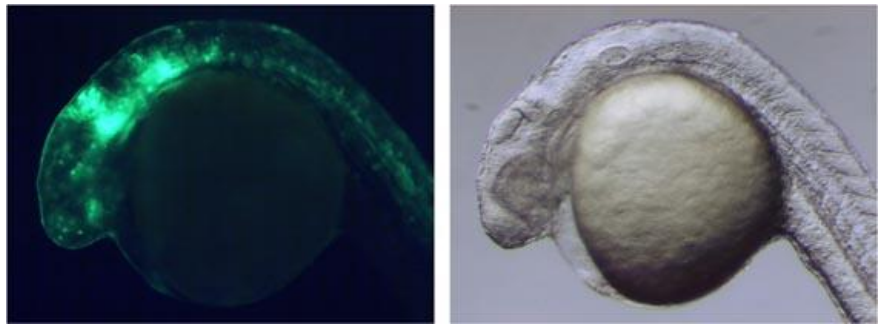
$$\frac{d\,🦊}{dt} = \delta\,🐇🦊 - \gamma\,🦊$$

Increases with more food

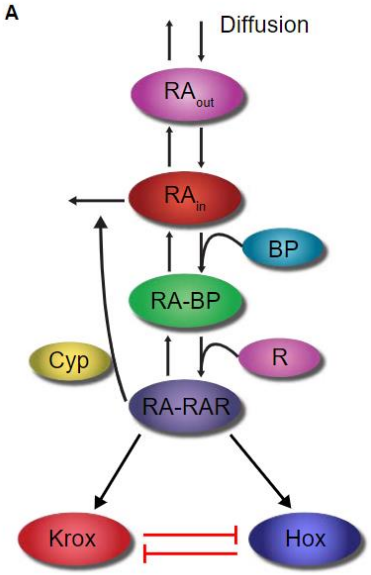
Decreases with competition



Developmental Biology Example: Hindbrain Development



Structure From Scientific Experiments



Statistical mechanical principles imply an evolution of:

$$d[RA_{out}] = (\beta - b[RA_{out}] + c[RA_{in}])dt,$$

$$d[RA_{in}] = \left(b[RA_{out}] + \delta[RA - BP] - \left(\gamma[BP] + \eta + \frac{\alpha[RA - RAR]}{\omega + [RA - RAR]} - c\right)[RA_{in}]\right)dt + \sigma dW_t,$$

$$d[\mathrm{RA - BP}] = (\gamma[BP][RA_{in}] + \lambda[BP][RA - RAR] - (\delta + \nu[RAR])[RA - BP])dt,$$

$$d[RA - RAR] = (\nu[RA - BP][RAR] - \lambda[BP][RA - RAR])dt,$$

$$d[RAR] = (\zeta - \nu[RA - BP][RAR] + \lambda[BP][RA - RAR] - r[RAR])dt,$$

$$d[BP] = (a - \lambda[BP][RA - RAR] - \gamma[BP][RA_{in}] + (\delta + \nu[RAR])[RA - BP] - u[BP])dt,$$

But what is the mathematical structure of machine learning?

Neural Networks = Function Approximation

- Polynomial: $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots$
- Nonlinear: $e^x = 1 + \frac{2\tanh\left(\frac{x}{2}\right)}{1-\tanh\left(\frac{x}{2}\right)}$
- Neural Network: $e^x \approx W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3$

Neural Networks = Nonlinear Regression

- Polynomial: $e^x = a_1 + a_2 x + a_3 x^2 + \cdots$
- Nonlinear: $e^x = 1 + \frac{a_1 \tanh(a_2 x)}{a_3 x - \tanh(a_4 x)}$
- Neural Network: $e^x \approx W_3 \sigma(W_2 \sigma(W_1 x + b_1) + b_2) + b_3$. Train the weights (W, b)

NEURAL NETWORKS CAN GET ϵ CLOSE TO ANY $R^n \rightarrow R^m$ FUNCTION

Universal Approximation Theorem

Other Universal Approximators

- Polynomials: easy but numerically unstable and Runge's Phenomenon
- Fourier/Chebyshev Series: great if smooth, but Gibb's Phenomenon.
- Going to higher dimensions usually isn't rotationally invariant: mixed derivatives give issues.
  - Tensor product spaces
  - Sparse Grids
- Radial Basis Functions (RBFs) work well on low dimensional manifolds in high dimensional spaces

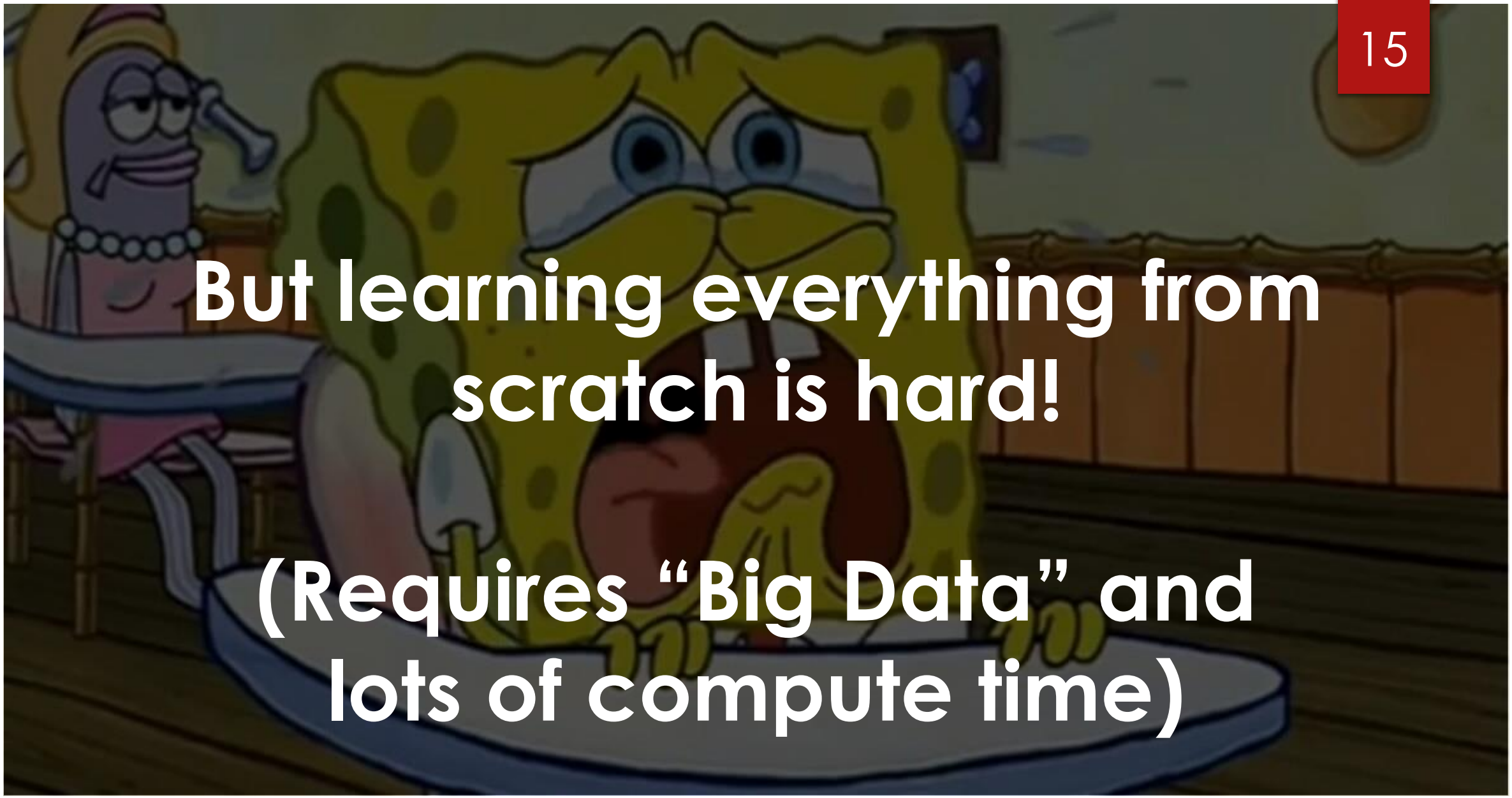
Nothing is really a silver bullet for arbitrary functions in high dimensions.

[1] L. N. Trefethen, Cubature, approximation, and isotropy in the hypercube, *SIAM Review*

[2] https://www.cs.cmu.edu/afs/cs/academic/class/15883-f17/slides/rbf.pdf

Neural Networks are universal approximators which work well in high dimensions

NEURAL NETWORKS OVERCOME "THE CURSE OF DIMENSIONALITY"


But learning everything from scratch is hard!
(Requires “Big Data” and lots of compute time)

What happens when we combine universal approximators and scientific models?

Latent (Neural) Differential Equations

NEURAL ORDINARY DIFFERENTIAL EQUATION:

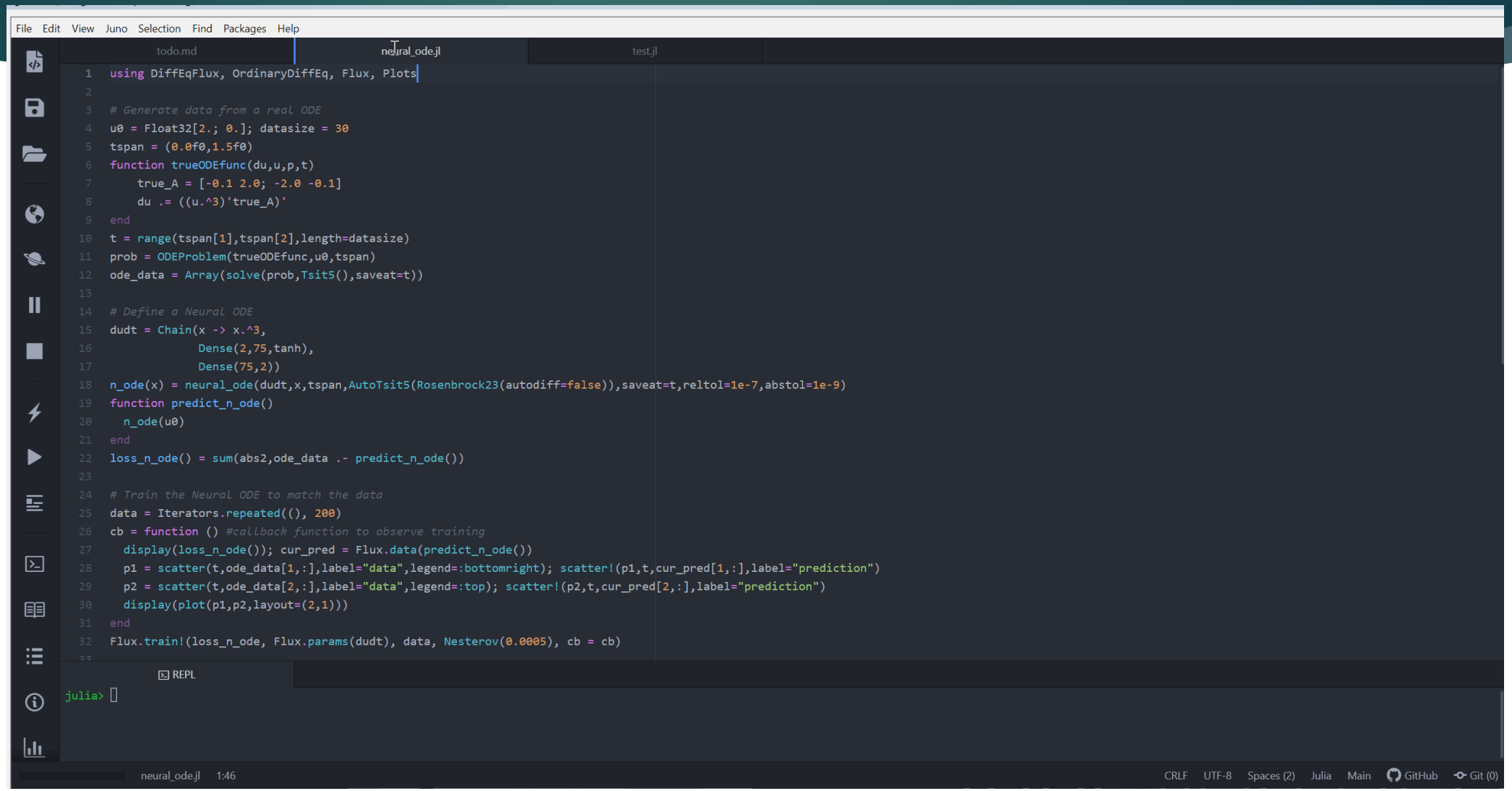
$$u' = f(u, p, t)$$

LET f BE A NEURAL NETWORK

Training a neural differential equation

- Solve the differential equation
- Compute the gradient of the solution with respect to the parameters defining the neural network
  - Adjoint sensitivity analysis
  - Differentiable programming
- Update the neural network and repeat

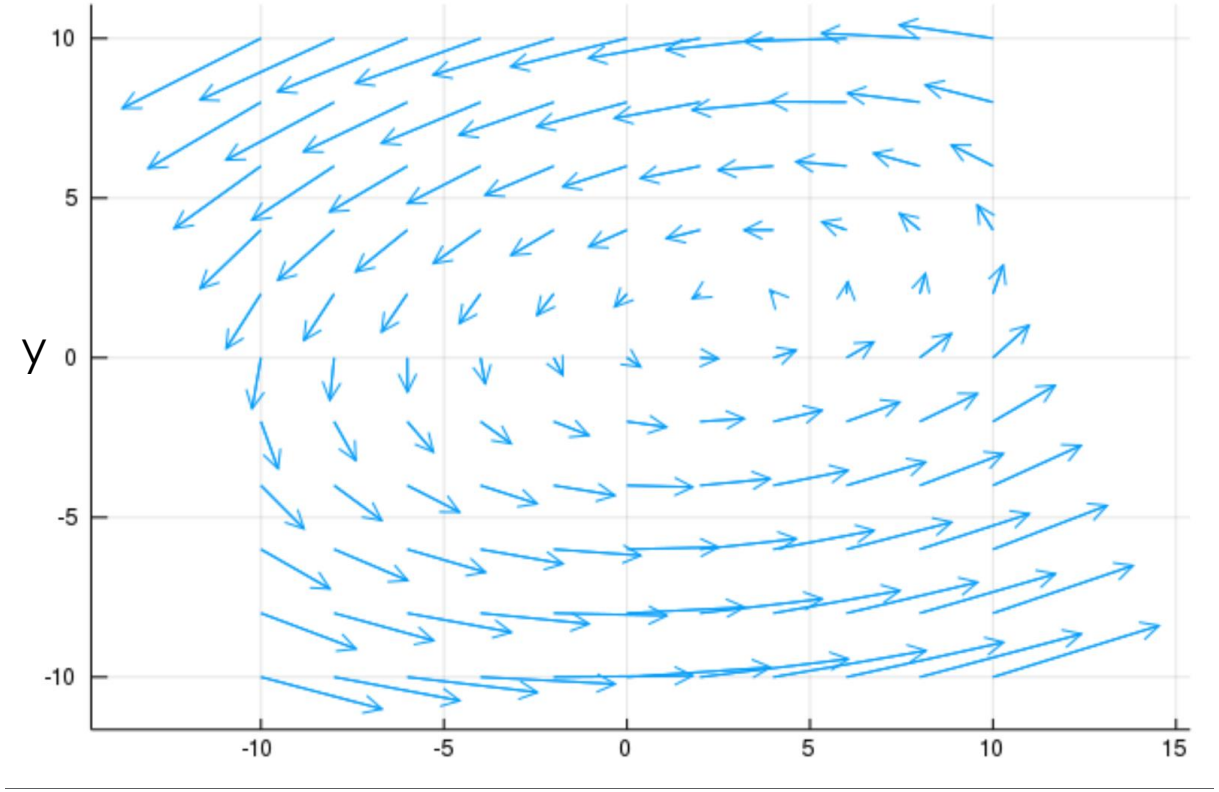
Automatically Learning the Model

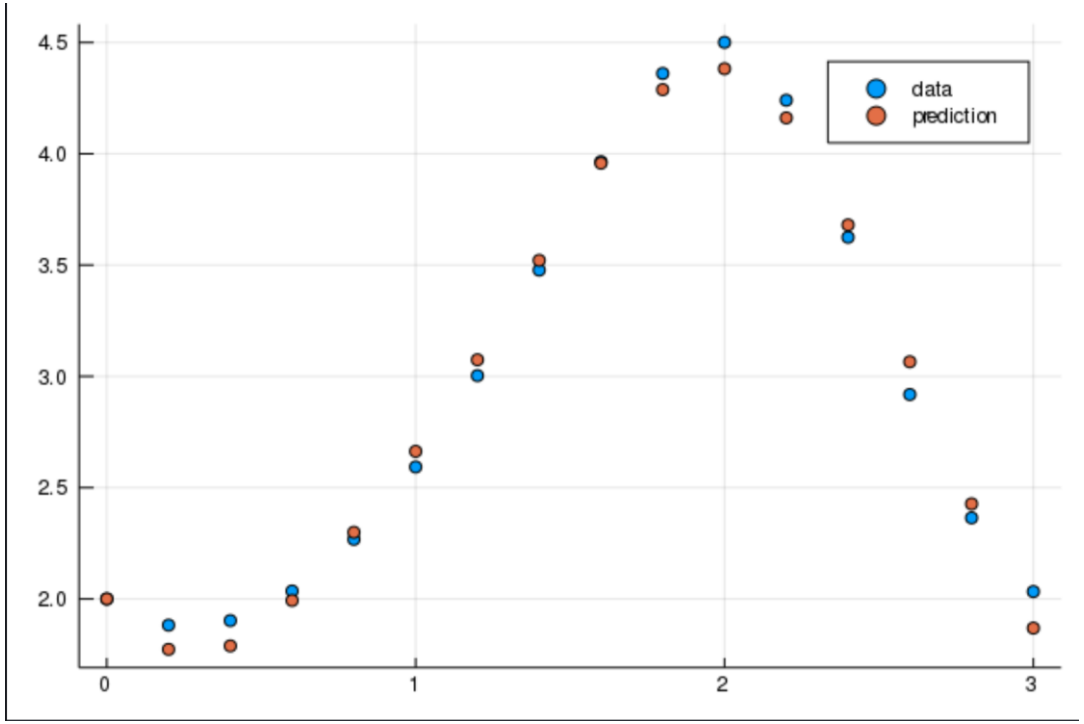


The Neural Differential Equation doesn't learn how to predict a timeseries, it learns phase space.

Direct Learning of ODEs from data: Lotka-Volterra from 16 data points

David Koplow (@DavidKoplow)





The cyclic phase space is learned, allowing correct extrapolation in time

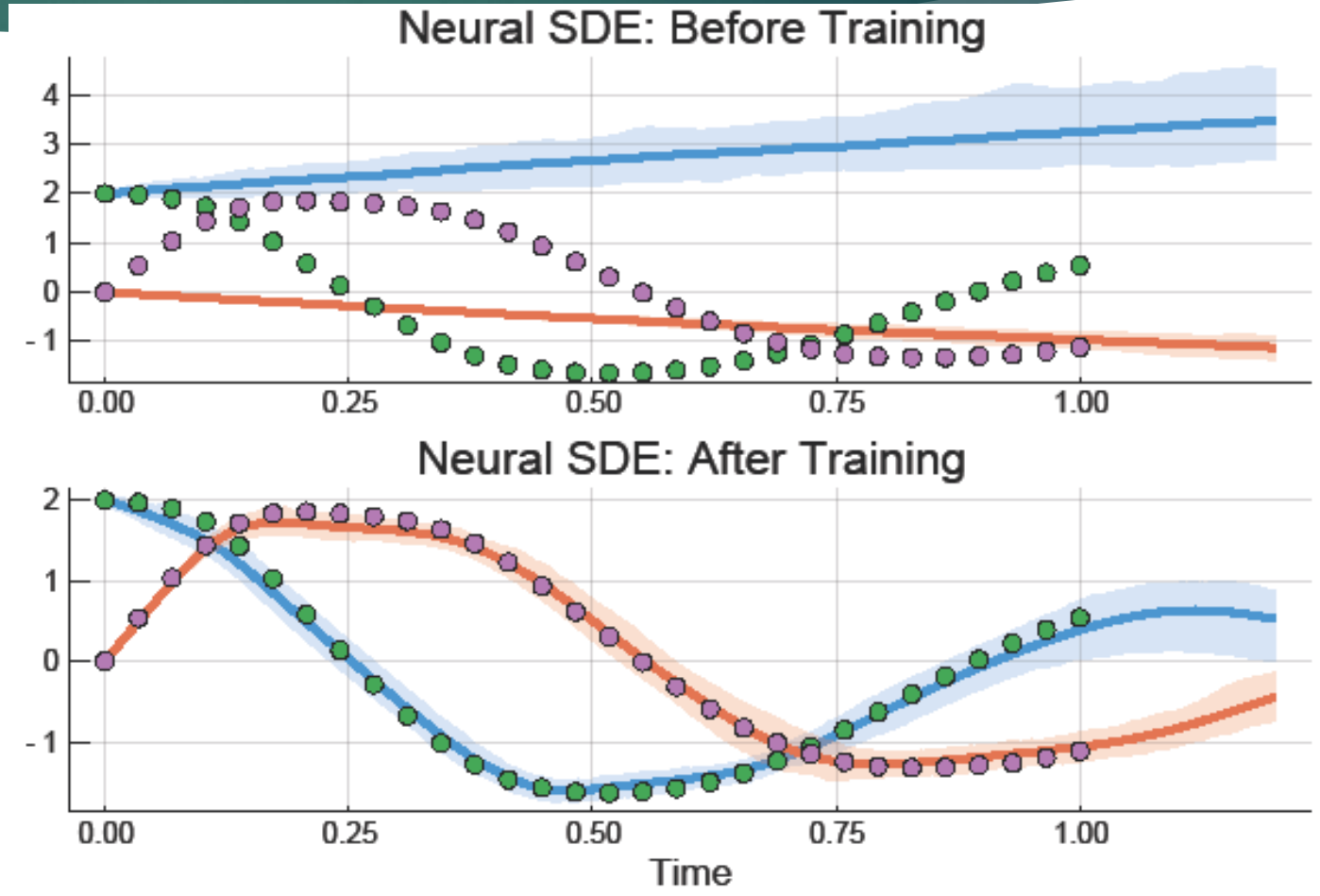
Latent (Neural) Stochastic Differential Equations

$$du = f(u,p,t)dt + g(u,p,t)dW_t$$

LET f AND g BE A NEURAL NETWORK

Neural SDEs: Dynamical Extrapolation with Constrained Variation

- If the number of rabbits ever gets too high, then the number of wolves increases which brings it back down.
- Learning the dynamical system makes it learn these feedback effects, which constrains the output to by physical and extrapolate variance as well

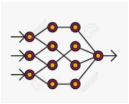


While using this as a full training method is great…
The real power comes from incorporating known structure into the ML framework
(Mixed Neural Differential Equation)

Mix Neural Networks Into DiffEqs!

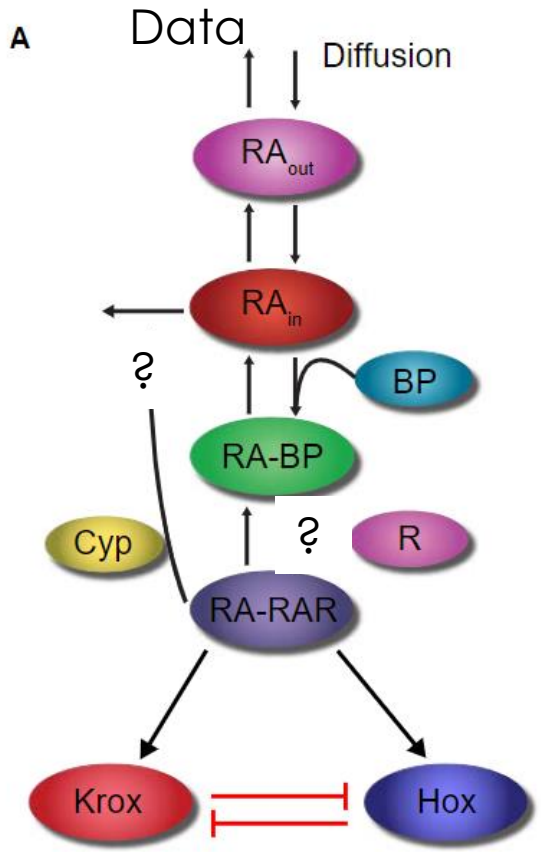
```
using DiffEqFlux, Flux, OrdinaryDiffEq

u0 = param(Float32[0.8; 0.8])
tspan = (0.0f0,25.0f0)
ann = Chain(Dense(2,10,tanh), Dense(10,1))

p1 = Flux.data(DiffEqFlux.destructure(ann))
p2 = Float32[-2.0,1.1]
p3 = param([p1;p2])
ps = Flux.params(p3,u0)

function dudt_(du,u,p,t)
    x, y = u
    du[1] = DiffEqFlux.restructure(ann,p[1:41])(u)[1]
    du[2] = p[end-1]*y + p[end]*x
end
prob = ODEProblem(dudt_,u0,tspan,p3)
function predict_adjoint()
  diffeq_adjoint(p3,prob,Tsit5(),u0=u0,saveat=0.0:0.1:25.0)
end
loss_adjoint() = sum(abs2,x-1 for x in predict_adjoint())
Flux.train!(loss_adjoint, ps, Iterators.repeated((), 10), ADAM(0.1))
```

$$\frac{dx}{dt} = \quad \frac{dy}{dt} = p_1 y + p_2 x$$



Fit the "mixed neural differential equation" using the same method!

ML-Assisted Model Discovery



The chemical reactions imply an evolution of:

$$d[RA_{out}] = (\beta - b[RA_{out}] + c[RA_{in}])dt,$$

$$d[RA_{in}] = \left(b[RA_{out}] + \delta[RA - BP] - \left(\gamma[BP] + \eta + \ ?\ \ - c\right)[RA_{in}]\right)dt + \sigma dW_t,$$

$$d[\mathrm{RA} - \mathrm{BP}] = (\gamma[BP][RA_{in}] + \lambda[BP][RA - RAR] - \ ?\ \ dt,$$

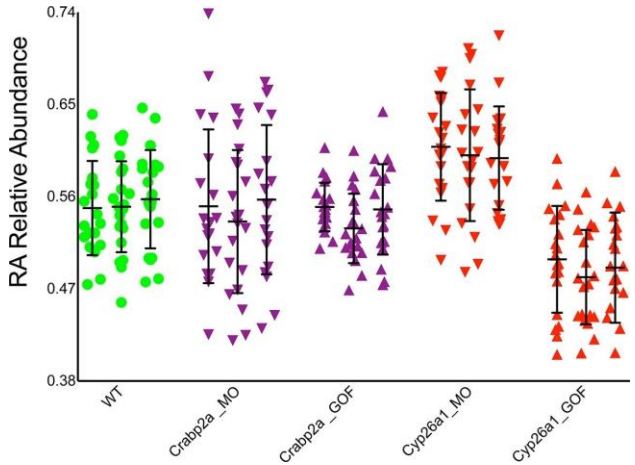
$$d[RA - RAR] = \ ?\ \ - \lambda[BP][RA - RAR])dt,$$

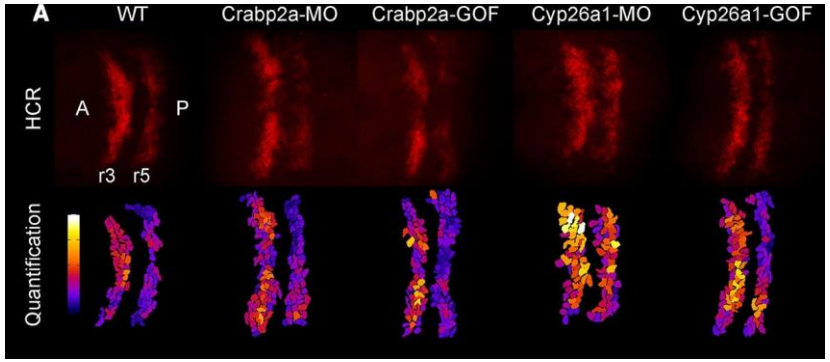
$$d[RAR] = \ ?\ \ dt,$$

$$d[BP] = (a - \lambda[BP][RA - RAR] - \gamma[BP][RA_{in}] + (\delta + \nu[RAR])[RA - BP] - u[BP])dt,$$

Biologically-Informed Neural Network

Data





Find neural networks so the model matches the data

$$d[RA_{out}] = (\beta - b[RA_{out}] + c[RA_{in}])dt,$$

$$d[RA_{in}] = \left(b[RA_{out}] + \delta[RA - BP] - \left(\gamma[BP] + \eta + \text{NN} - c\right)[RA_{in}]\right)dt + \sigma dW_t,$$

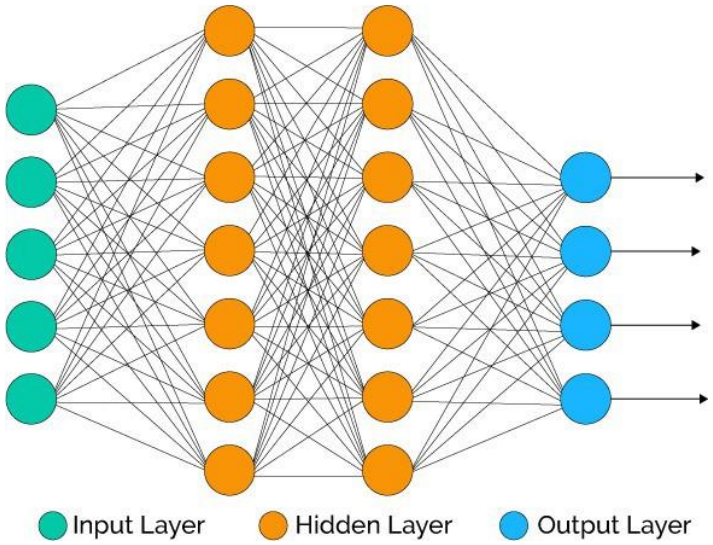
$$d[\mathrm{RA - BP}] = (\gamma[BP][RA_{in}] + \lambda[BP][RA - RAR] - \text{NN})\,dt,$$

$$d[RA - RAR] = \text{NN} - \lambda[BP][RA - RAR])dt,$$

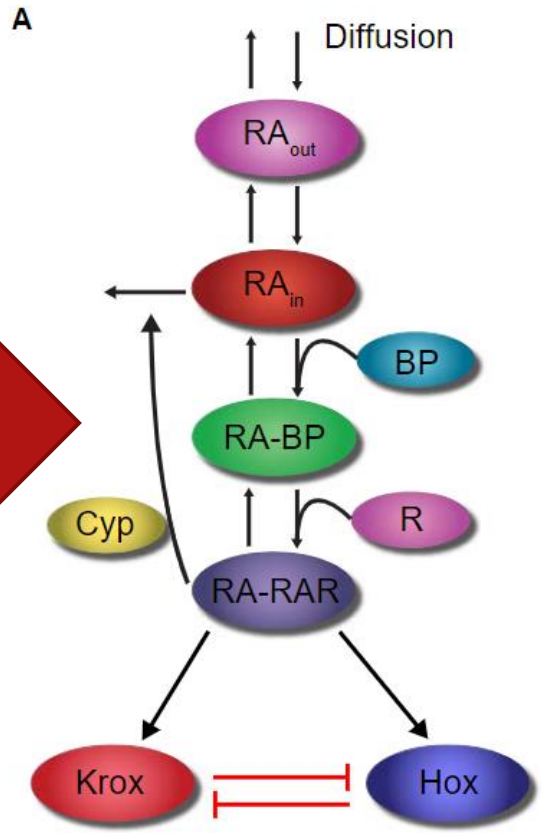
$$d[RAR] = \text{NN}\,dt,$$

$$d[BP] = (a - \lambda[BP][RA - RAR] - \gamma[BP][RA_{in}] + (\delta + \nu[RAR])[RA - BP] - u[BP])dt,$$

Interpretability of Neural Differential Equations



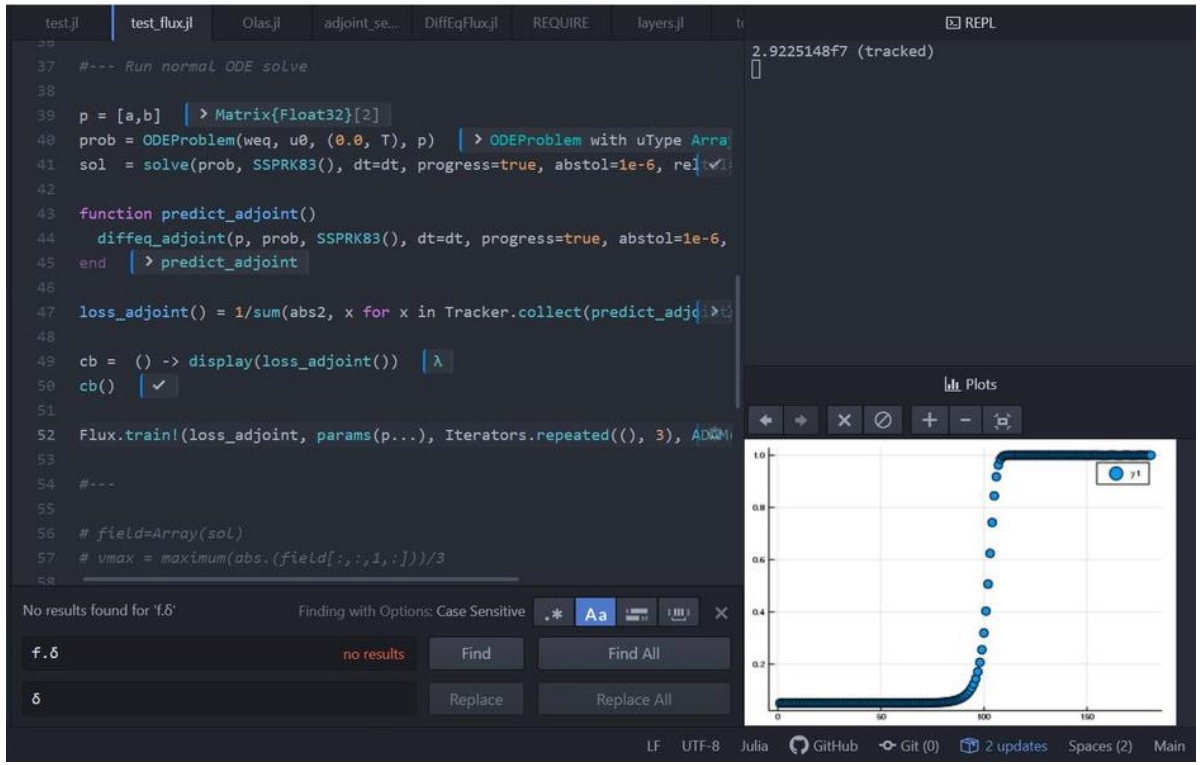
Analyze the Jacobian/Hessian



Nonlinear Optimal Control as a Mixed Neural ODE

- $x' = f(x(t), u(t), t)$
- Minimize $J = \Phi\left(x(t_0), t_0, x\left(t_f\right), t_f\right) + \int_{t_0}^{t_f} L(x(t), u(t), t)dt$
- Example: $x(t)$ is the location of an automated drone, $u(t)$ is the controller, find what the controller should be such that the vehicle goes to the right place for the least energy.
- Neural ODE Approach: Make $u(t)$ be a neural network. Find the neural network s.t. $x(t)$ correctly evolves

Neural PDEs for Acceleration: Automated Quasilinear Approximations



- Boussinesq Equations (Navier-Stokes) are used in climate models

$$\nabla \cdot \mathbf{u} = 0$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \mathrm{Pr}\nabla^2 \mathbf{u} + b\hat{z}$$

$$\frac{\partial b}{\partial t} + \mathbf{u} \cdot \nabla b = \nabla^2 b + Fe^z$$

- People attempt to solve this by "parameterizing", i.e. getting a 1-dimensional approximation through averaging:

$$\left(\frac{\partial}{\partial t} + \mathbf{u}.\nabla - \nabla^2\right)c' - \frac{\partial}{\partial z}\overline{w'c'} = -w'\frac{\partial \bar{c}}{\partial z}$$

where $\overline{w'c'}$ is unknown.

- Instead of picking a form for $\overline{w'c'}$(the current method), replace it with a neural network and learn it from small scale simulations! Discretize. Result: Neural ODE.

There seems to be a pattern going on here…

ARE OTHER RECENT BREAKTHROUGHS ALSO NEURAL DIFFERENTIAL EQUATIONS?

Solving 1000 dimensional PDEs: Hamilton-Jacobi-Bellman, Nonlinear Black-Scholes

- Semilinear Parabolic Form (Diffusion-Advection Equations, Hamilton-Jacobi-Bellman, Black-Scholes)

$$\frac{\partial u}{\partial t}(t,x)+\frac{1}{2}\mathrm{Tr}\left(\sigma\sigma^{\mathrm{T}}(t,x)(\mathrm{Hess}_x u)(t,x)\right)+\nabla u(t,x)\cdot\mu(t,x) \\ +f\left(t,x,u(t,x),\sigma^{\mathrm{T}}(t,x)\nabla u(t,x)\right)=0 \qquad [1]$$

Then the solution of Eq. 1 satisfies the following BSDE (cf., e.g., refs. 8 and 9):

$$u(t,X_t)-u(0,X_0) \\ =-\int_0^t f\left(s,X_s,u(s,X_s),\sigma^{\mathrm{T}}(s,X_s)\,\nabla u(s,X_s)\right)ds \qquad [3] \\ +\int_0^t [\nabla u(s,X_s)]^{\mathrm{T}}\,\sigma(s,X_s)\,dW_s.$$

- Make $(\sigma^T \nabla \mathrm{u})(t,X)$ a neural network.
- Solve the resulting SDEs and learn $\sigma^T \nabla \mathrm{u}$ via:

$$l(\theta)=\mathbb{E}\left[\left|g(X_{t_N})-\hat{u}\left(\{X_{t_n}\}_{0\le n\le N},\{W_{t_n}\}_{0\le n\le N}\right)\right|^2\right].$$

Simplified:

- Transform it into a Backwards SDE: a stochastic boundary value problem. The unknown is a function!
- Learn the unknown function via neural network.
- Once learned, the PDE solution is known.

Solving high-dimensional partial differential equations using deep learning, 2018, PNAS, Han, Jentzen, E

Stochastic RNN Formulation

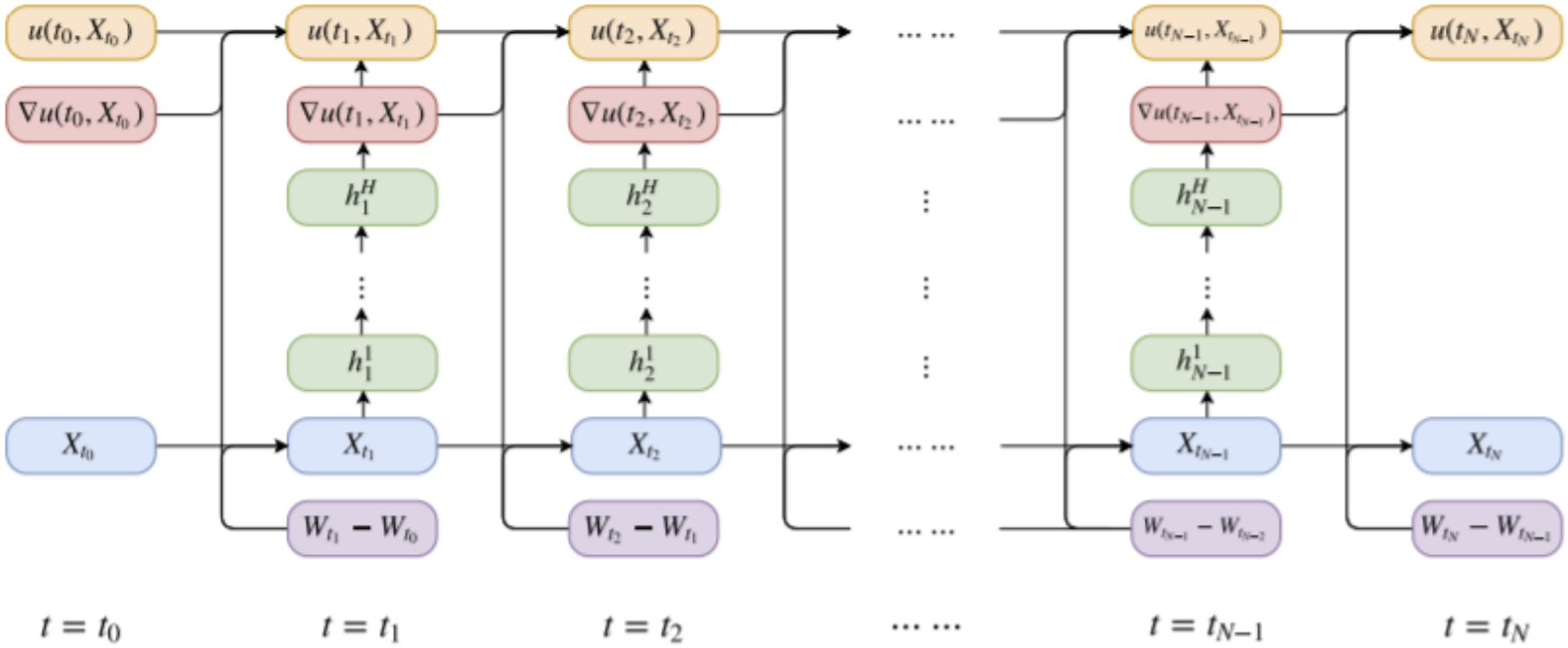
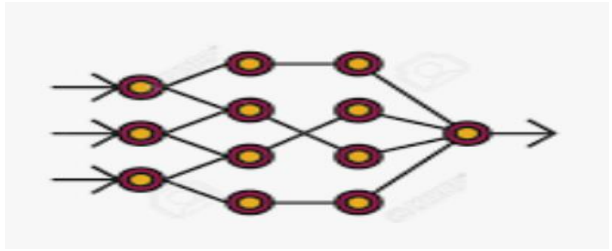


Fig. 4. Illustration of the network architecture for solving semilinear parabolic PDEs with H hidden layers for each subnetwork and N time intervals. The whole network has $(H+1)(N-1)$ layers in total that involve free parameters to be optimized simultaneously. Each column for $t = t_1, t_2, \ldots, t_{N-1}$ corresponds to a subnetwork at time t. $h_n^1, \ldots, h_n^H$ are the intermediate neurons in the subnetwork at time $t = t_n$ for $n = 1, 2, \ldots, N-1$.

But this method can be represented as a neural SDE

- That method is the fixed time-step Euler-Maruyama method on

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t,$$

$$dU_t = f(t, X_t, U_t, \sigma^T(t, X_t)\nabla u(t, X_t))dt + \text{  } dW_t,$$

- As a neural SDE, we can solve with higher order (less neural network evaluations), adaptivity, etc.

Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations

- M.Raissi, P.Perdikaris, & G.E.Karniadakis

- "neural networks that are trained to solve supervised learning tasks while respecting any given laws of physics described by general nonlinear partial differential equations"

The Discrete PINN Method

- General nonlinear partial differential equation: $u_t + N(u) = 0$
- Discretize with Runge-Kutta:

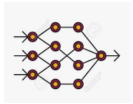
$$u^{n+c_i} = u^n - \Delta t \sum_{j=1}^{q} a_{ij} \mathcal{N}[u^{n+c_j}], \quad i = 1, \ldots, q,$$
$$u^{n+1} = u^n - \Delta t \sum_{j=1}^{q} b_j \mathcal{N}[u^{n+c_j}].$$

- Make your neural networks be:

$$[u_1^n(x), \ldots, u_q^n(x), u_{q+1}^n(x)]$$

- As a prior, train the neural network to be the PDE's solution, then match to data.

Discrete PINN as a Neural Differential Equation

- $u_t + N(u) +$  $= 0$ where the neural network NN is initialized to zero.
- Discretize using a fixed timestep Runge-Kutta method

This makes it clear how to generalize…

- Other PDE discretizations
- Adaptive solver
- Methods other than Runge-Kutta

Neural Networks mixed with scientific models tend to give Neural Differential Equations, which:

- ALLOW FOR AUTOMATICALLY LEARNING MODELS, USING KNOWN EQUATIONS AS A PRIOR
- SOLVE OPTIMAL CONTROL PROBLEMS
- ACCELERATE THE SOLUTION OF PDES
- SOLVE PDES WHICH WERE PREVIOUSLY UNSOLVABLE

Scientific Machine Learning requires efficient solution of Neural Differential Equations

DiffEqFlux.jl

The first (mixed) Neural Differential Equation solver. Supports:

- Neural ODEs
- Neural SDEs (SDDEs)
- Neural DAEs
- Neural DDEs
- Stiff Equations
- Hybrid Equations
- Adjoints via reverse-mode AD and adjoint sensitivity analysis

DiffEqFlux.jl uses code generation to make Flux.jl neural networks compose with the DifferentialEquations.jl solvers

ALL OF THE FEATURES ARE AVAILABLE

Some DifferentialEquations.jl Features

- MPI+GPU Compatibility
- Implicit, IMEX, multirate, symplectic, exponential integrators, etc.
- Adaptive high order methods for stochastic differential equations
- Stiff state-dependent delay differential equation discontinuity tracking
- Mix in Gillespie simulation (Continuous-Time Markov Chains)
- Automatic sparsity detection and optimization
- Arbitrary code injection through callbacks

And it's routinely benchmarks as one of the fastest libraries in most categories

DiffEqFlux.jl has the bells and whistles to solve "real" problems

Neural ODE with batching on the GPU (without internal data transfers) with high order adaptive implicit ODE solvers for stiff equations using matrix-free Newton-Krylov via preconditioned GMRES and trained using checkpointed adjoint equations.

```
using OrdinaryDiffEq, Flux, DiffEqFlux, DiffEqOperators, CuArrays
x = Float32[2.; 0.]|>gpu
tspan = Float32.((0.0f0,25.0f0))
dudt = Chain(Dense(2,50,tanh),Dense(50,2))|>gpu
p = DiffEqFlux.destructure(dudt)
dudt_(du,u::TrackedArray,p,t) =  u .= DiffEqFlux.restructure(dudt,p)(u)
dudt_(du,u::AbstractArray,p,t) = u .= Flux.data(DiffEqFlux.restructure(dudt,p)(u))
ff = ODEFunction(dudt_,jac_prototype = JacVecOperator(dudt_,x))
prob = ODEProblem(ff,x,tspan,p)
diffeq_adjoint(p,prob,KenCarp4(linsolve=LinSolveGMRES());u0=x,
               saveat=0.0:0.1:25.0,backsolve=false)
```

Conclusion: ML can be improved by using scientific knowledge

- ML with scientific knowledge is Neural Differential Equations
- DiffEqFlux.jl is the only Neural Differential Equation solver, and it has the bells and whistles to handle large stiff stochastic delay equations

Do you want to research numerical differential equations and Scientific ML?

- Looking for collaborators for Julia-wide scientific ecosystem development grants
- Students are welcome! Contact me for Google Summer of Code, Julia Seasons of Contributions, or Pumas development. No Julia experience is required. Just jump right into the chat channels (https://gitter.im/JuliaDiffEq/Lobby) and we can find you an appropriate project.
- https://julialang.org/soc/ideas-page
- If you're a hobbyist… join our chats! Fit models! Stop by the MIT Julia Lab!

Appendix

Automatic Differentiation in a nutshell

- Numerical differentiation is numerically bad because you're dividing by a small number. Can this be avoided?
- Early idea: instead of using a real-valued difference, when f is real-valued but complex analytic, use the following identity:

$$f'(x) \approx \Im\left\{\frac{f(x+ih)}{h}\right\}.$$

- Claim: the numerical stability of this algorithm matches that of f
- Automatic differentiation then scales this idea to multiple dimensions
- One implementation: use Dual numbers $\mathrm{x} = a + b\epsilon$ where $\epsilon^2 = 0$ (smooth infinitesimal arithmetic). Define $f(x) = f(a) + f'(a)b\epsilon$ (chain rule).

Differentiable Programming: Derivative Calculations as Non-standard Interpretation

$$(x + x'\varepsilon) + (y + y'\varepsilon) = x + y + (x' + y')\varepsilon$$
$$(x + x'\varepsilon) \cdot (y + y'\varepsilon) = xy + xy'\varepsilon + yx'\varepsilon + x'y'\varepsilon^2 = xy + (xy' + yx')\varepsilon$$

- Claim: if you recompiled your entire program to do Dual arithmetic, then the output of your program is a Dual number which computes both the original value and derivative simultaneously (to machine accuracy).
- As described, this is known as operator overloading forward-mode automatic differentiation (AD). There are also computational graph and AST-based AD implementations. In addition, there are "adjoint" or reverse-mode automatic differentiation which specifically produce gradients of cost functions with better scaling properties
- "Backpropogation" of neural networks is simple reverse-mode AD on some neural network program.

Analytical Solutions (Sensitivity Analysis) VS AD (Adjoints)

A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions

Christopher Rackauckas, Yingbo Ma, Vaibhav Dixit,
Xingjian Guo, Mike Innes, Jarrett Revels,
Joakim Nyberg, Vijay Ivaturi

$$\frac{d}{dt}\left(\frac{\partial u}{\partial p_i}\right) = \frac{\partial f}{\partial u}\frac{\partial u}{\partial p_i} + \frac{\partial f}{\partial p_i}$$

$$\frac{d\lambda^*}{dt} = -\lambda^* \frac{\partial f(u(t), p, t)}{\partial u}$$

- General conclusion: AD already is more efficient than traditional sensitivity analysis techniques until you get to 100 parameters (PDEs) where continuous adjoint sensitivity analysis still shines.
- Mixing analytical solutions and AD is currently the best.
- More work to be done on reverse-mode AD to get the scaling advantage with less overhead. Tracing-based AD (Jax, PyTorch, Flux, etc.) is unable to scale on these problems due to scalar options in nonlinear code.
- Source-to-source AD (Zygote) doesn't have these issues, so it's our next goal.