Some more use cases of neural differential equations and universal differential equations

Chris Rackauckas
Director of Modeling and Simulation,
Julia Computing

Research Affiliate, Co-PI of Julia Lab,
Massachusetts Institute of Technology, CSAIL

Director of Scientific Research,
Pumas-AI

Final Note: Using Compilers and Transformations Beyond Differentiation

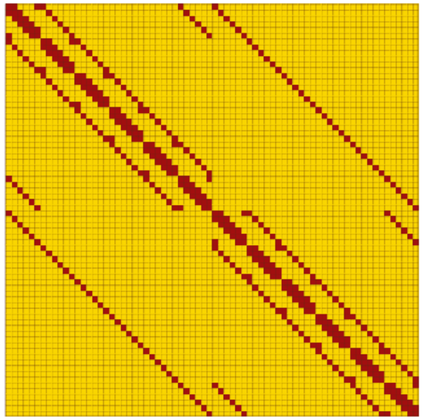


Figure 1: Sparsity pattern of the Jacobian of the Brusselator code in Listing 1 with input and output tensors of size $6 \times 6 \times 2 = 72$.

Table 1: Hessian sparsity construction for a program taking as input a vector of length 4. The 4×4 sparsity pattern for each intermediate value is shown. The provenance polynomial has the same hessian sparsity pattern.

code fragment	polynomial	sparsity
`deg2rad(x[1])`	x_1	
`log(x[1])`	x_1^2	
`x[1] + x[4]`	$x_1 + x_4$	
`x[1] * x[4]`	$x_1 x_4$	
`q = x[1]/x[4]`	$x_1 x_4^2$	
`asin(q)*x[3]`	$(x_1^2 x_4^2) x_3$	

Automatic Sparsity Detection

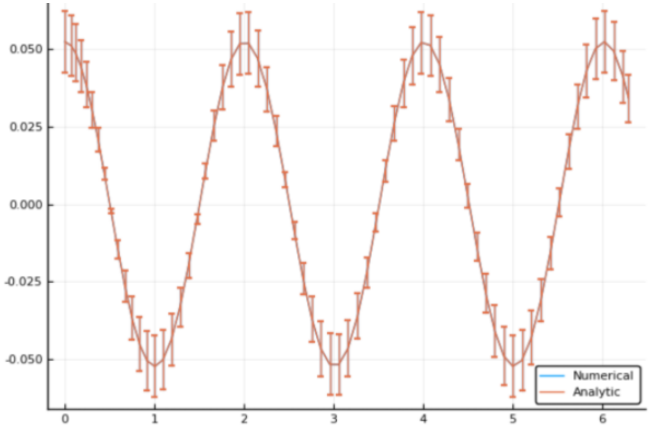
```
g = 9.79 ± 0.02 # Gravitational constants
L = 1.00 ± 0.01 # Length of the pendulum

# Initial speed & angle, time span
u₀ = [0 ± 0, π/60 ± 0.01]
tspan = (0.0, 6.3)

# Define the problem
function pendulum(du, u, p, t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

# Pass to solvers
prop = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g/L) .* sol.t)
```



Rackauckas et al. DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. 2017. ([Journal of Open Research Software](#))

Giordano. Uncertainty propagation with functionally correlated quantities ([arXiv:1610.08716](#))

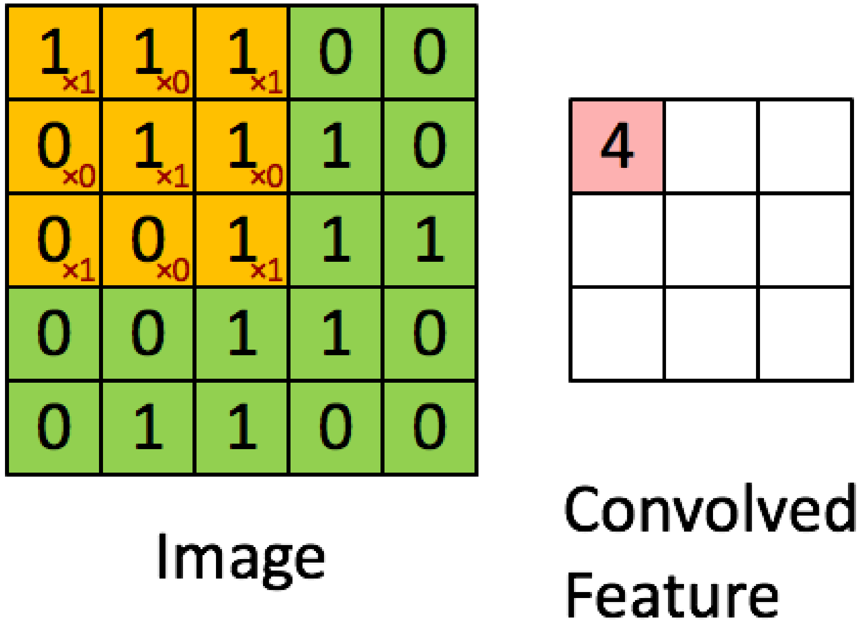
Compiler-Based Intrusive Uncertainty Quantification

[Generalizing Automatic Differentiation to Automatic Sparsity Uncertainty, Stability, and Parallelism](#), StochasticLifestyle.cor

Discretized PDE Operators are Convolutions

$$\frac{u(x+\Delta x, y) - 2u(x,y) + u(x-\Delta x, y)}{\Delta x^2} + \frac{u(x, y+\Delta y) - 2u(x,y) + u(x-x, y-\Delta y)}{\Delta y^2}$$

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

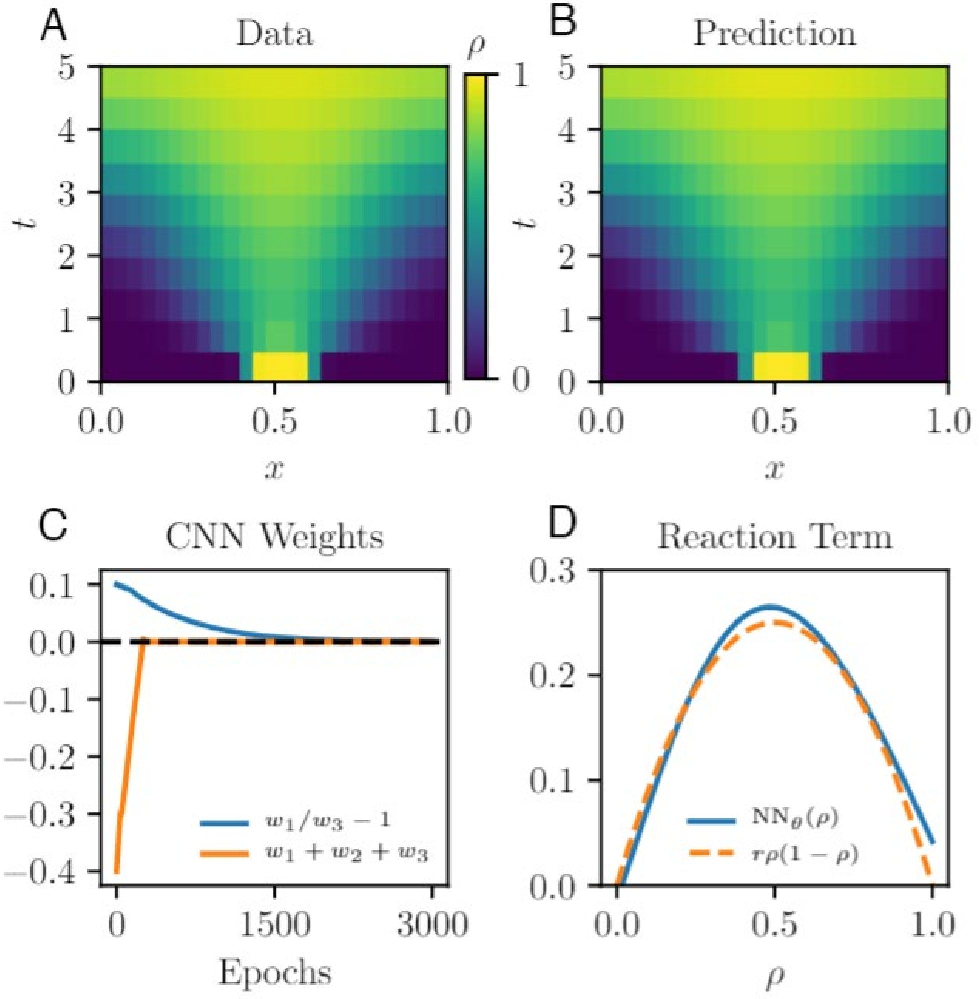


Automatically Learning PDEs from Data: Universal PDEs for Fisher-KPP

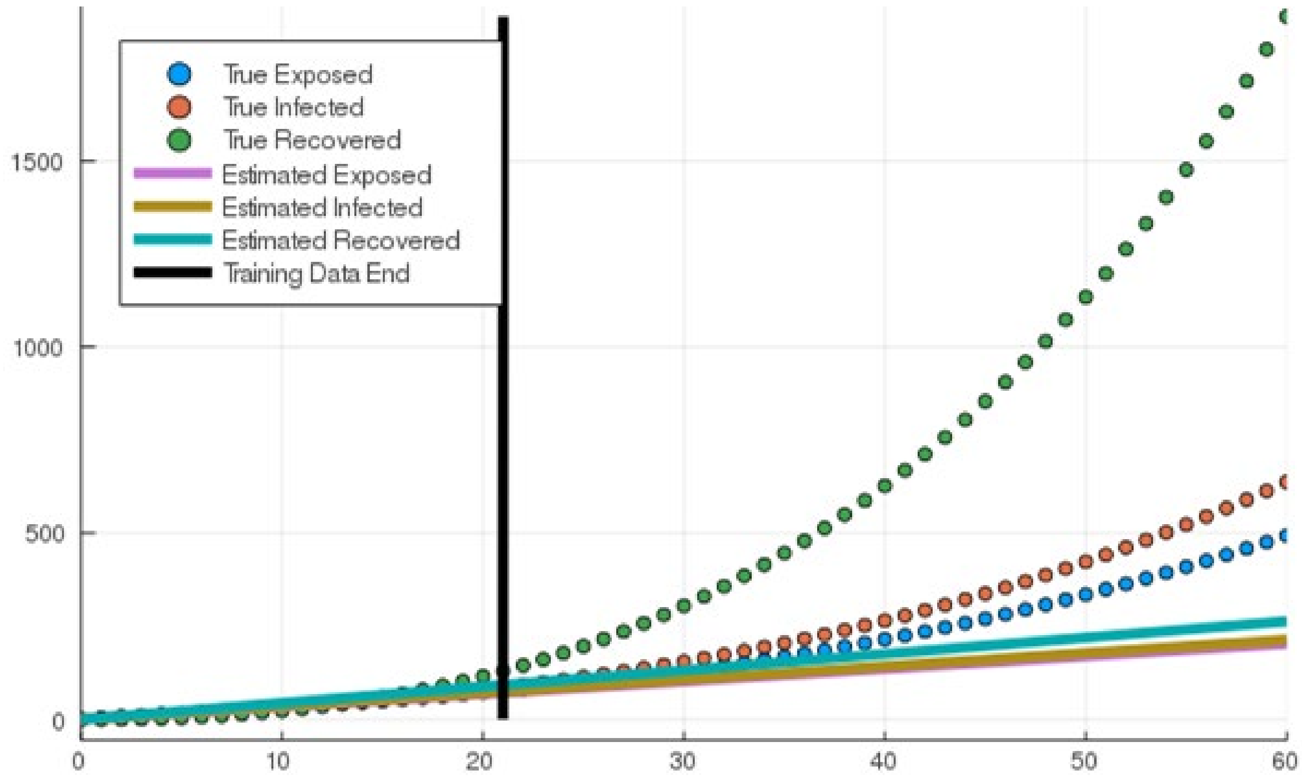
$$\rho_t = \mathrm{NN}_\theta(\rho) + D\,\mathrm{CNN}(\rho),$$

$$\rho_t = r\rho(1-\rho) + D\rho_{xx},$$

Note: due to the dimensionality of the operator, it's more efficient to use a non-neural network operator here!



Neural ODE: Learn the whole model



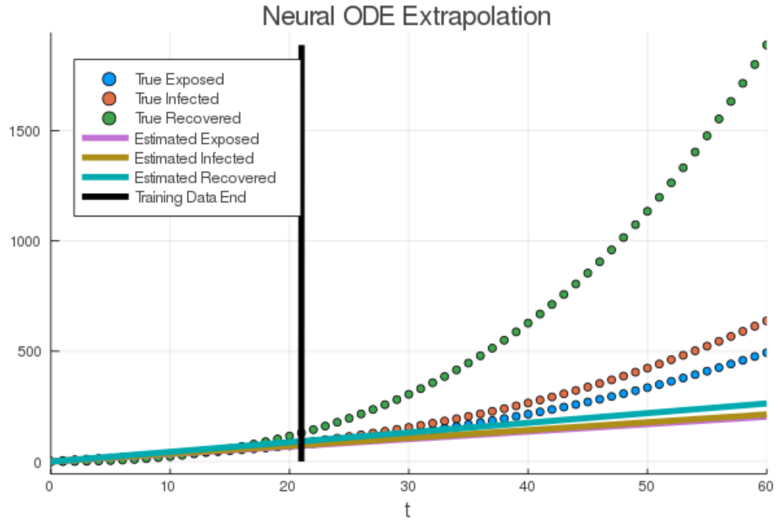
u’=NN(u) trained on 21 days of data

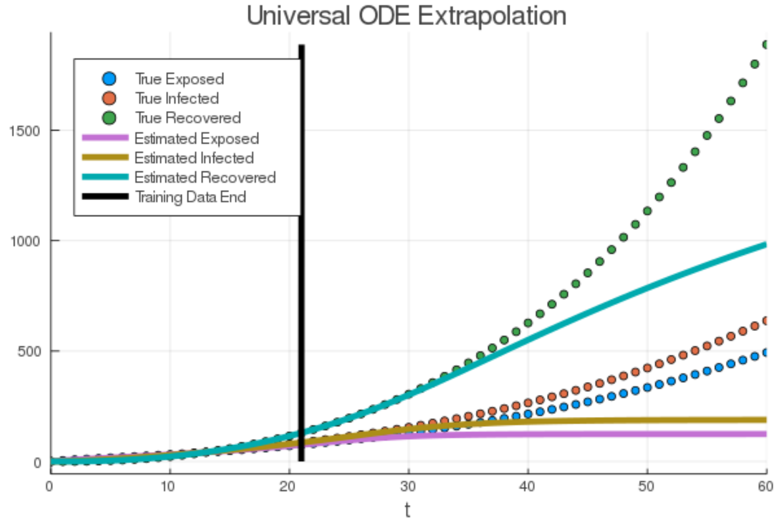
Can fit, but not enough information to accurately extrapolate

Does not have the correct asymptotic behavior

Universal ODE

$$
\begin{aligned}
S' &= -\frac{\beta_0 S F}{N} - \text{Replace Unknown Portion} - \mu S,\\
E' &= \frac{\beta_0 S F}{N} + \text{Replace Unknown Portion} - \left(\sigma + \mu\right) E,\\
I' &= \sigma E - (\gamma + \mu) I,\\
R' &= \gamma I - \mu R,\\
N' &= -\mu N,\\
D' &= d\,\gamma I - \lambda D, \quad \text{and}\\
C' &= \sigma E,
\end{aligned}
$$





SInDy – Sparse Identification of Dynamical Systems

sparse vectors of coefficients $\Xi = [\xi_1 \; \xi_2 \; \cdots \; \xi_n]$ that determine which nonlinearities are active:

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi. \qquad [3]$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{array}{c} \xrightarrow{\text{state}} \\ \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix} \end{array} \downarrow \text{time}$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^T(t_1) \\ \dot{\mathbf{x}}^T(t_2) \\ \vdots \\ \dot{\mathbf{x}}^T(t_m) \end{bmatrix} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \cdots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \cdots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \cdots & \dot{x}_n(t_m) \end{bmatrix}.$$

Next, we construct a library $\Theta(\mathbf{X})$ consisting of candidate nonlinear functions of the columns of $\mathbf{X}$. For example, $\Theta(\mathbf{X})$ may consist of constant, polynomial, and trigonometric terms:

$$\Theta(\mathbf{X}) = \begin{bmatrix} | & | & | & | & & | & | & \\ 1 & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \cdots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \cdots \\ | & | & | & | & & | & | & \end{bmatrix}. \qquad [2]$$

Brunton, Steven L., Joshua L. Proctor, and J. Nathan Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems." *Proceedings of the national academy of sciences* 113.15 (2016): 3932-3937.

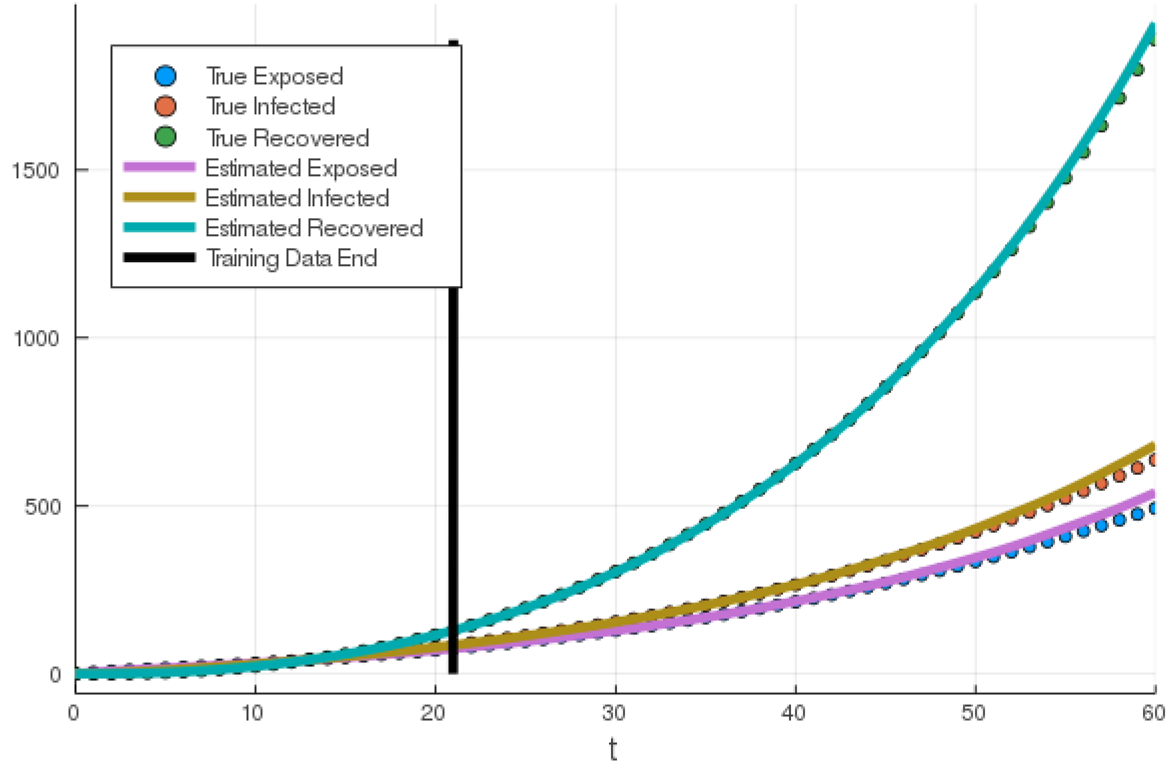
- Operation[cos(u_1) * -0.0013108600297508188 + cos(u_2) * 0.001048733466930909 + sin(u_3) * 0.002524237642240494 + 4.582000697122147 + u_3 * 48.22745315102507 + u_3 ^ 2 * -0.5293305992835255 + u_2 * 39.085961651678964 + u_2 * u_3 * -0.6742175940650399 + u_2 * u_3 ^ 2 * 0.0018086945606415868 + u_2 ^ 2 * -0.7760315827702667 + u_2 ^ 2 * u_3 * -0.00827007707292397 + u_2 ^ 2 * u_3 ^ 2 * -4.8420203054602525e-5 + u_1 * 0.6927075862062384 + u_1 * u_3 * 2.5477896384187675 + u_1 * u_3 ^ 2 * -0.007633697801342265 + u_1 * u_2 * -0.8050223920175605 + u_1 * u_2 * u_3 * -0.005893734488035572 + u_1 * u_2 * u_3 ^ 2 * -4.205818407350913e-5 + u_1 * u_2 ^ 2 * 0.05154776022562611 + u_1 * u_2 ^ 2 * u_3 * 0.00011401535262358879 + u_1 * u_2 ^ 2 * u_3 ^ 2 * -1.8409670007515867e-7 + u_1 ^ 2 * -1.480917344589218 + u_1 ^ 2 * u_3 * 0.022834435321810845 + u_1 ^ 2 * u_3 ^ 2 * -7.10505011605666e-5 + u_1 ^ 2 * u_2 * -0.0811262292209696 + u_1 ^ 2 * u_2 * u_3 * 1.2503710381374686e-5 + u_1 ^ 2 * u_2 * u_3 ^ 2 * -1.5835869421530206e-7 + u_1 ^ 2 * u_2 ^ 2 * 0.0003756078420420898 + u_1 ^ 2 * u_2 ^ 2 * u_3 * 2.0403671083190194e-6 + u_1 ^ 2 * u_2 ^ 2 * u_3 ^ 2 * -4.0790059067580516e-10, cos(u_1) * 0.0018236630124880049 + sin(u_3) * -0.002857556410244201 + 0.738713743952307 + u_3 * -45.316633125282735 + u_3 ^ 2 * 0.4976552341495027 + u_2 * -36.669905096040644 + u_2 * u_3 * 0.63405194300575 + u_2 * u_3 ^ 2 * -0.001699189499009162 + u_2 ^ 2 * 0.7292234161358288 + u_2 ^ 2 * u_3 * 0.007782847250932861 + u_2 ^ 2 * u_3 ^ 2 * 4.5537832343115385e-5 + u_1 * -0.662837140886116 + u_1 * u_3 * -2.3955577736237044 + u_1 * u_3 ^ 2 * 0.007174813124917316 + u_1 * u_2 *
- 0.7564652530371222 + u_1 * u_2 * u_3 * 0.005539740817006857 + u_1 * u_2 * u_3 ^ 2 * 3.952859749575076e-5 + u_1 * u_2 ^ 2 * -0.04846972496409705 + u_1 * u_2 ^ 2 * u_3 * -0.00010714683124587004 + u_1 * u_2 ^ 2 * u_3 ^ 2 * 1.7315253185547634e-7 + u_1 ^ 2 * 1.3922758705496125 +
- u_1 ^ 2 * u_3 * -0.021478161074782457 + u_1 ^ 2 * u_3 ^ 2 * 6.675620535553527e-5 + u_1 ^ 2 * u_2 * 0.07628907557295377 + u_1 ^ 2 * u_2 * u_3 * -1.174623626431566e-5 + u_1 ^ 2 * u_2 * u_3 ^ 2 * 1.4858536352836396e-7 + u_1 ^ 2 * u_2 ^ 2 * -0.0003531614272747699 + u_1 ^ 2 * u_2
- ^ 2 * u_3 * -1.9178976768869506e-6 + u_1 ^ 2 * u_2 ^ 2 * u_3 ^ 2 * 3.8405659245262027e-10, -0.04932474700217403 + u_2 * 0.17406814677977456 + u_1 ^ 2 * u_2 * -1.4594144102122378e-6]

Universal ODE -> Internal Sparse Regression

Sparse Identification on only the missing term
Operation[u_2 * 0.10234428543435758 + u_1 * u_2 * 0.11371750552005416 + u_1 ^ 2 * u_2 * 0.12635459799855597] of u=(S/N,I,D/N)

Sparsity improves generalizability!

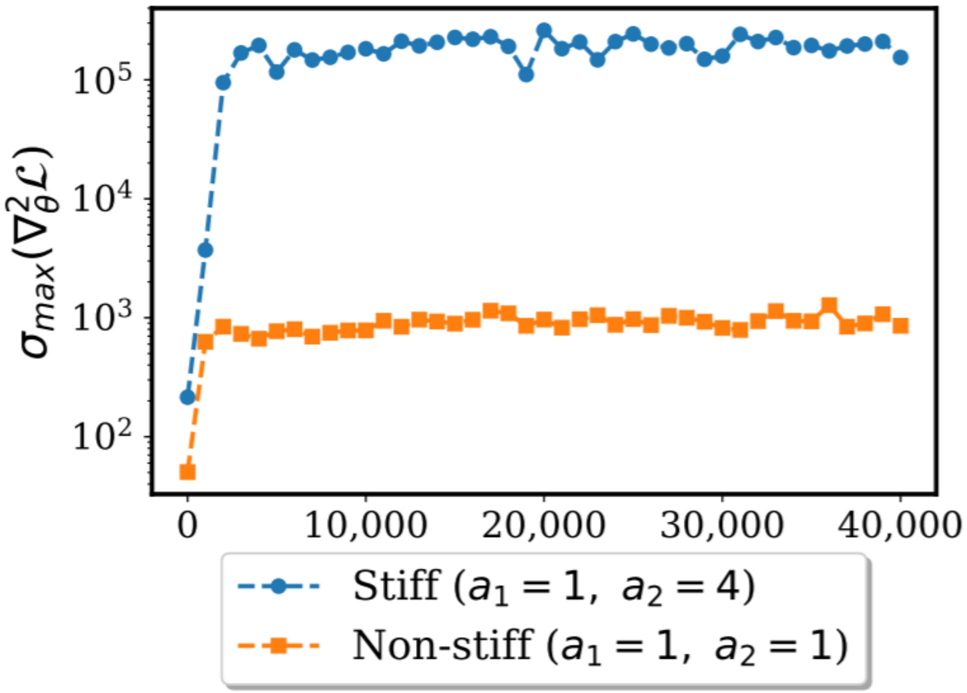
$$
\begin{aligned}
S' &= -\frac{\beta_0 S F}{N} - \text{Replace Unknown Portion} - \mu S,\\
E' &= \frac{\beta_0 S F}{N} + \text{Replace Unknown Portion} - \Big(\sigma + \mu\Big) E,\\
I' &= \sigma E - (\gamma + \mu) I,\\
R' &= \gamma I - \mu R,\\
N' &= -\mu N,\\
D' &= d\,\gamma I - \lambda D, \quad \text{and}\\
C' &= \sigma E,
\end{aligned}
$$

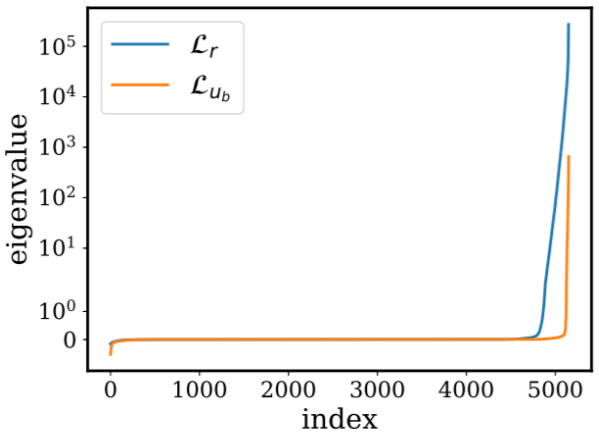


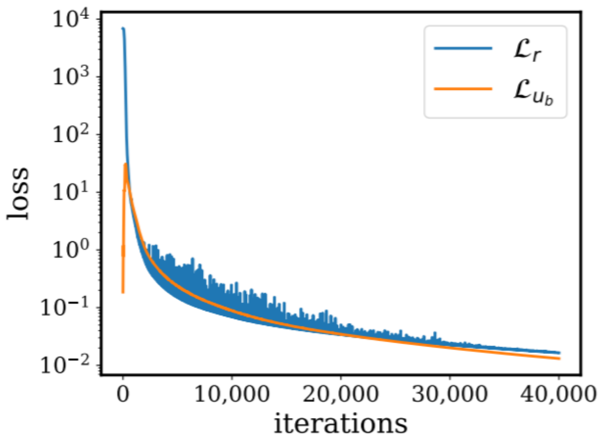
Why is a new foundation needed? Because off-the-shelf ML tools will not work

Understanding and mitigating gradient pathologies in physics-informed neural networks

Sifan Wang, Yujun Teng, Paris Perdikaris







Universal Differential Equations are Powerful Abstractions: Solving 1000 dimensional Hamilton-Jacobi-Bellman via Universal SDEs on a laptop

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

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

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

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

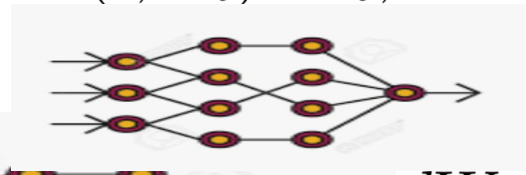
- Make $(\sigma^T \nabla \mathrm{u})(t, X)$ a neural network.
- Solve the resulting SDEs and learn $\sigma^T \nabla \mathrm{u}$ via:

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

Forward-Backward Stochastic Neural Networks: Deep Learning of High-dimensional Partial Differential Equations
Maziar Raissi

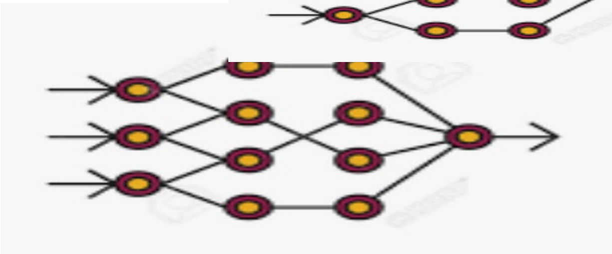
$$dX_t = \mu(t,X_t)dt+\sigma(t,X_t)dW_t,$$

$$dU_t = f(t,X_t,U_t,$$



$$)dt$$

$$+$$



$$dW_t,$$

Use high order, implicit, adaptive SDE solvers
Train a solution in minutes

Using non-adaptive explicit 0.5th order Euler-Maruyama matches the state-of-the-art deep BSDE methods from the literature

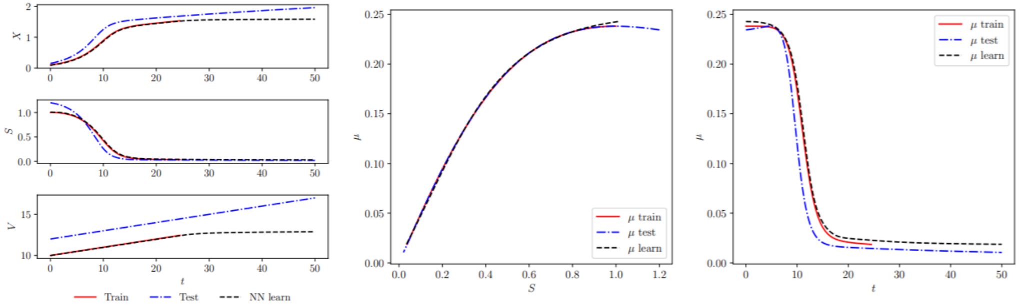
Solving high-dimensional partial differential equations using deep learning
Jiequn Han, Arnulf Jentzen, and Weinan E

UDE Methods Cover Accelerated Physics-Informed Neural Network Methods

$$l_n = \sum_{m=0}^{M} [\alpha_m y_{n-m} + \Delta t \beta_m f^{NN}(y_{n-m})], \quad n = M, \cdots, N.$$

$$loss(f^{NN}(y^{NN}(t))) = \frac{1}{N_y} \sum_{n=1}^{N_y} \left(y^{NN}(t_n) - y^*(t_n)\right)^2$$

$$+ \frac{1}{N_f} \sum_{n=1}^{N_f} \left(\frac{dy^{NN}}{dt}(t_n) - f^{NN}(y^{NN}(t_n)) \right)^2,$$



This methodology can be seen as a universal differential equation with a multistep integrator where adaptive=false

The UDE methodology thus gives an generalization to:

- Implicit methods, SSP methods
- Runge-Kutta-Chebyshev methods
- SDEs, DAEs, DDEs, etc.

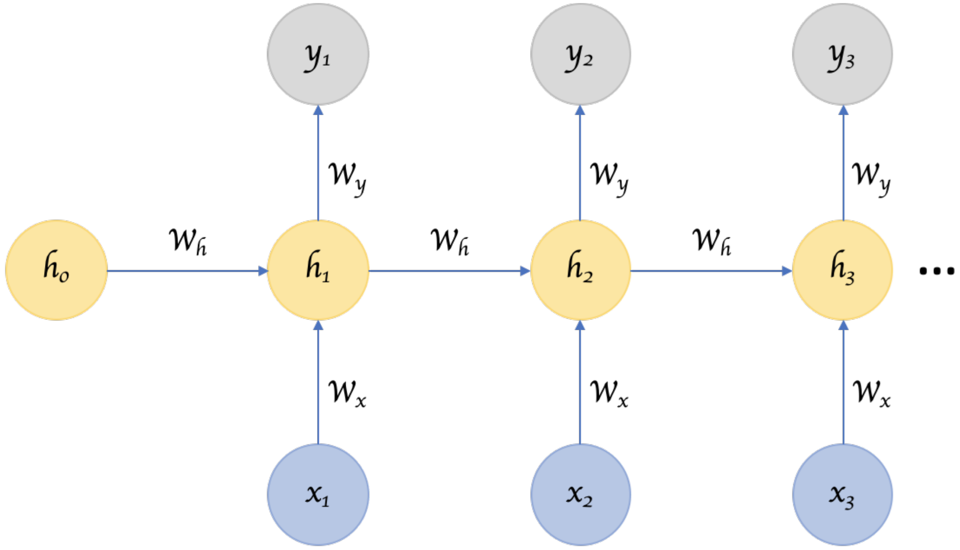
A comparative study of physics-informed neural network models for learning unknown dynamics and constitutive relations Ramakrishna Tipireddy, Paris Perdikaris, Panos Stinis and Alexandre Tartakovsky

Our results indicate that the accuracy of the trained neural network models is much higher for the cases where we only have to learn a constitutive relation instead of the whole dynamics.

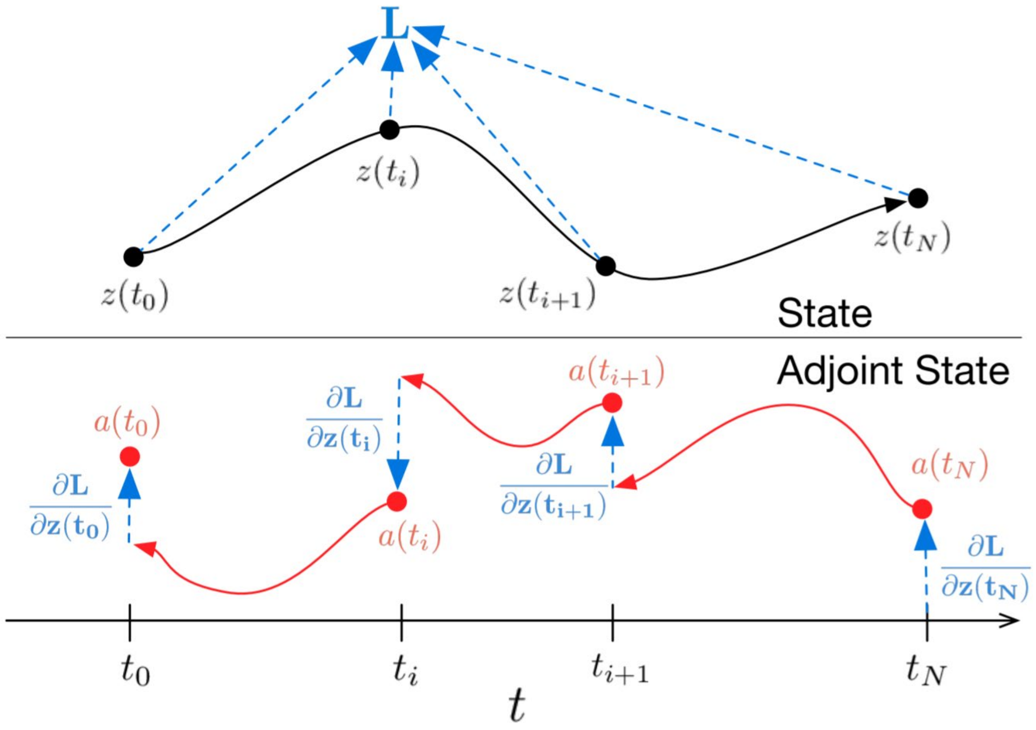
Multistep Neural Networks for Data-driven Discovery of Nonlinear Dynamical Systems Maziar Raissi, Paris Perdikaris , and George Em Karniadakis

Neural ODEs as Adaptive Layer Methods

RNN: Euler's method with fixed number of layers



Neural ODE: Adaptive ODE solver chooses number of layers

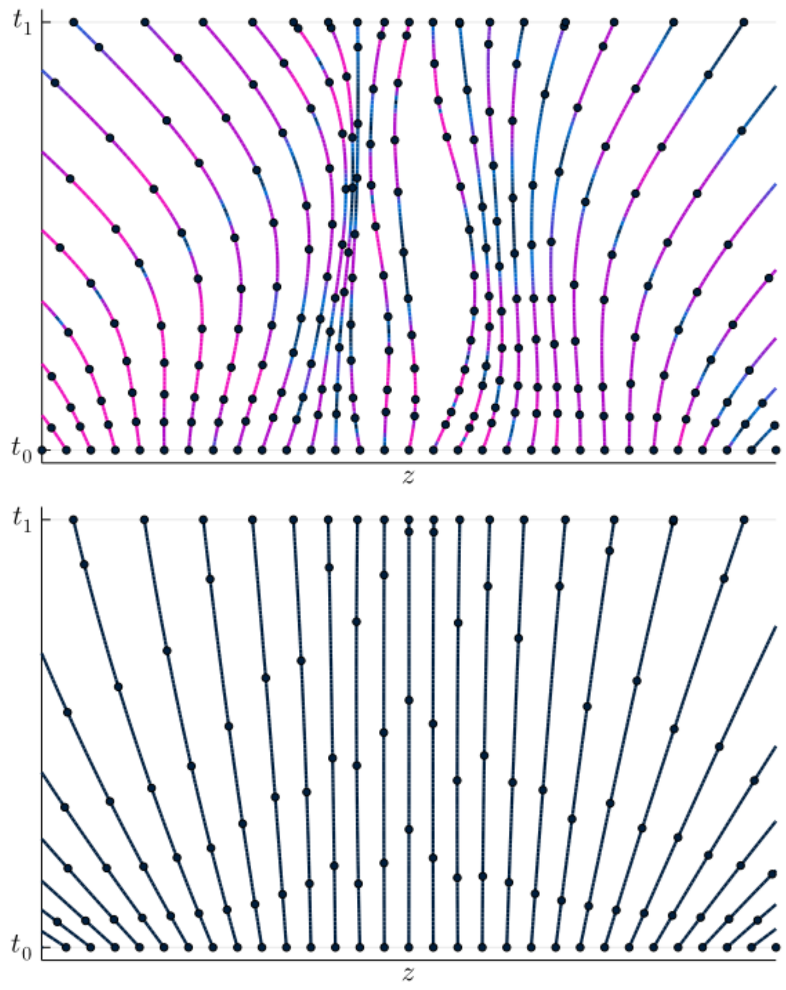


Neural ODEs can be used on classical machine learning problems to automatically learn the required number of layers

ML Layer: Value is initial condition of ODE, output is solution of ODE at final time

Chen, R. T., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018, December). Neural ordinary differential equations. In Proceedings of the 32nd International Conference on Neural Information Processing Systems (pp. 6572-6583).

If you only care about the end, why not learn the easiest dynamics you can?



Adaptive ODE solvers are correct to $K-1$ derivatives and control error on the Kth.

Use Taylor mode (high order) automatic differentiation to calculate:

$$R_K(\theta) = \int_{t_0}^{t_1} \left\| \frac{d^K z(t)}{dt^K} \right\| dt$$

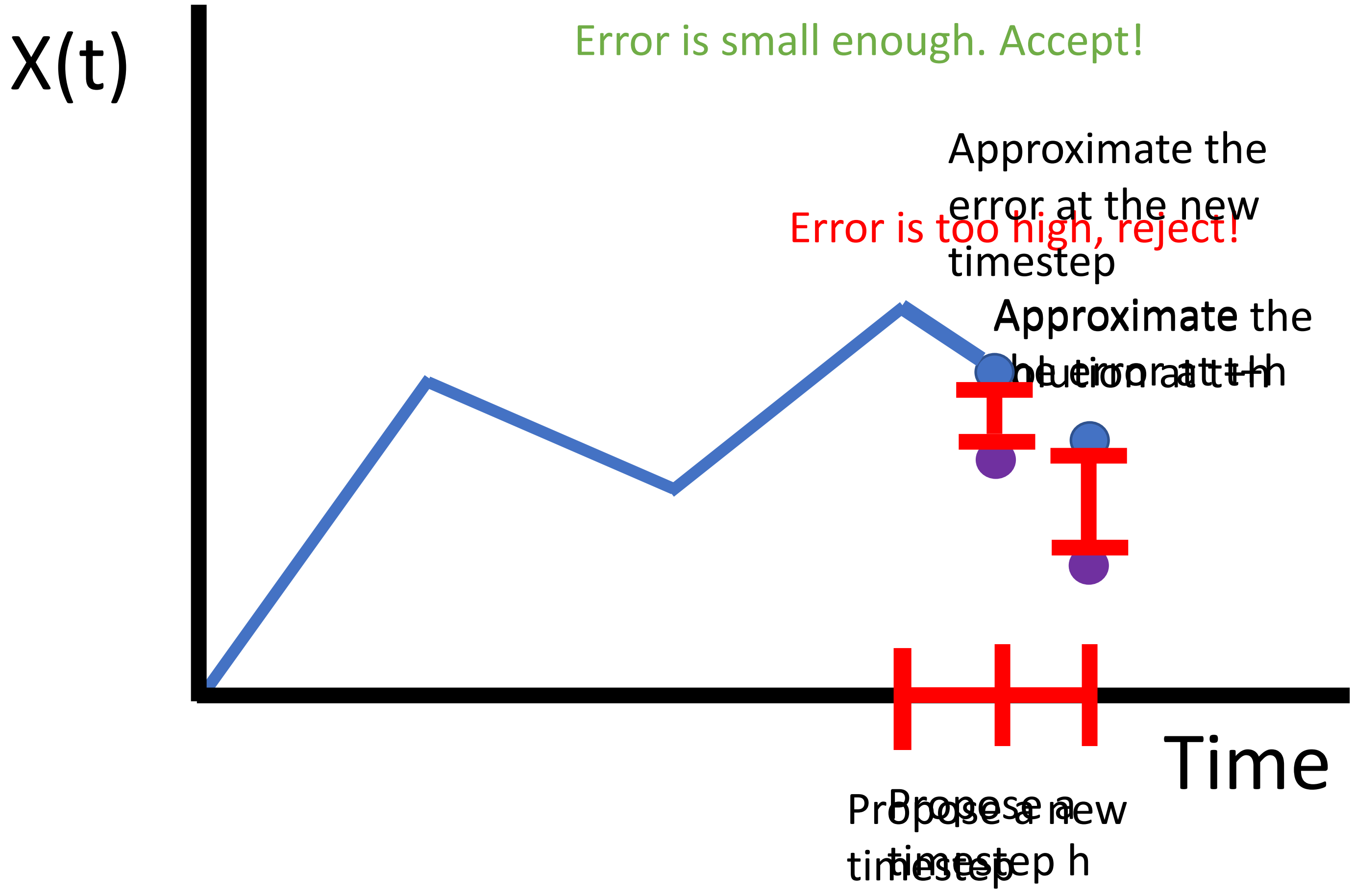
and regularize:

$$L_{reg}(\theta) = L(\theta) + R_K(\theta)$$

It does accelerate the learned dynamics, but training is expensive (10x slower!) because higher order automatic differentiation is exponentially expensive.

Kelly, J., Bettencourt, J., Johnson, M.J. and Duvenaud, D., 2020. Learning differential equations that are easy to solve. arXiv preprint arXiv:2007.04504.

But Solvers "know" a lot about the equation!



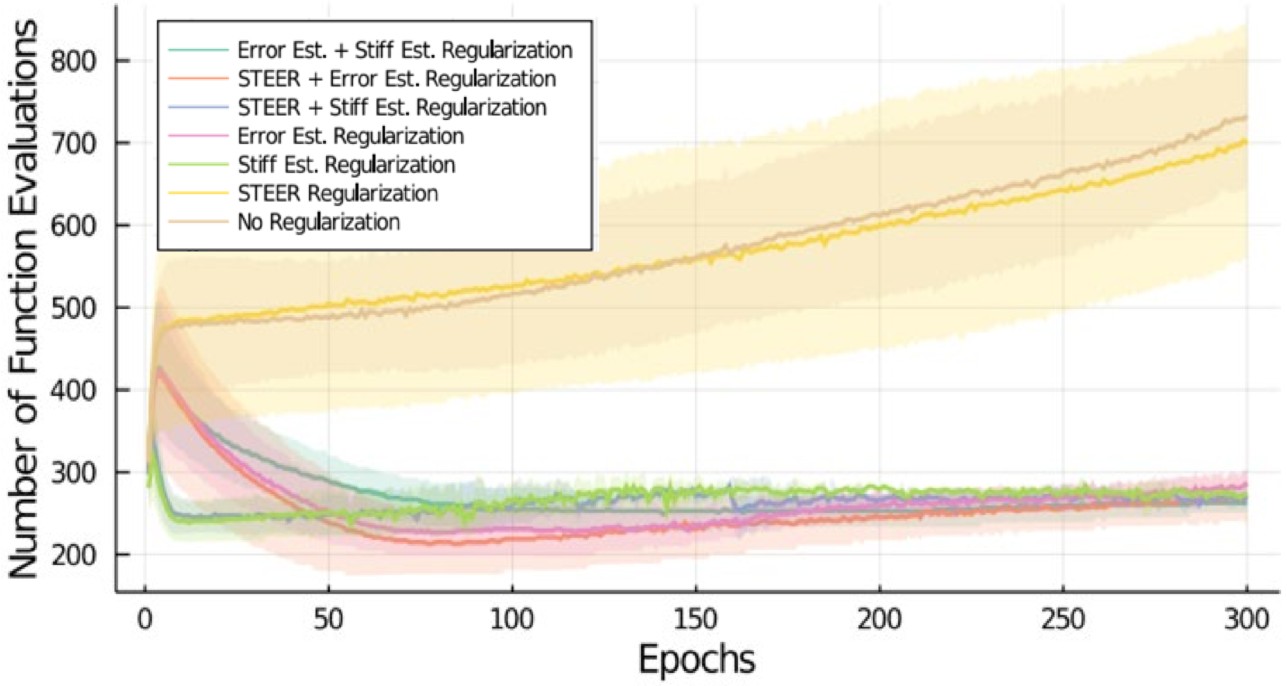
Adaptive ODE solvers already know "free" error estimates and stiffness estimators!

Idea: use the solver's internal heuristics to regularize out "hard" dynamics

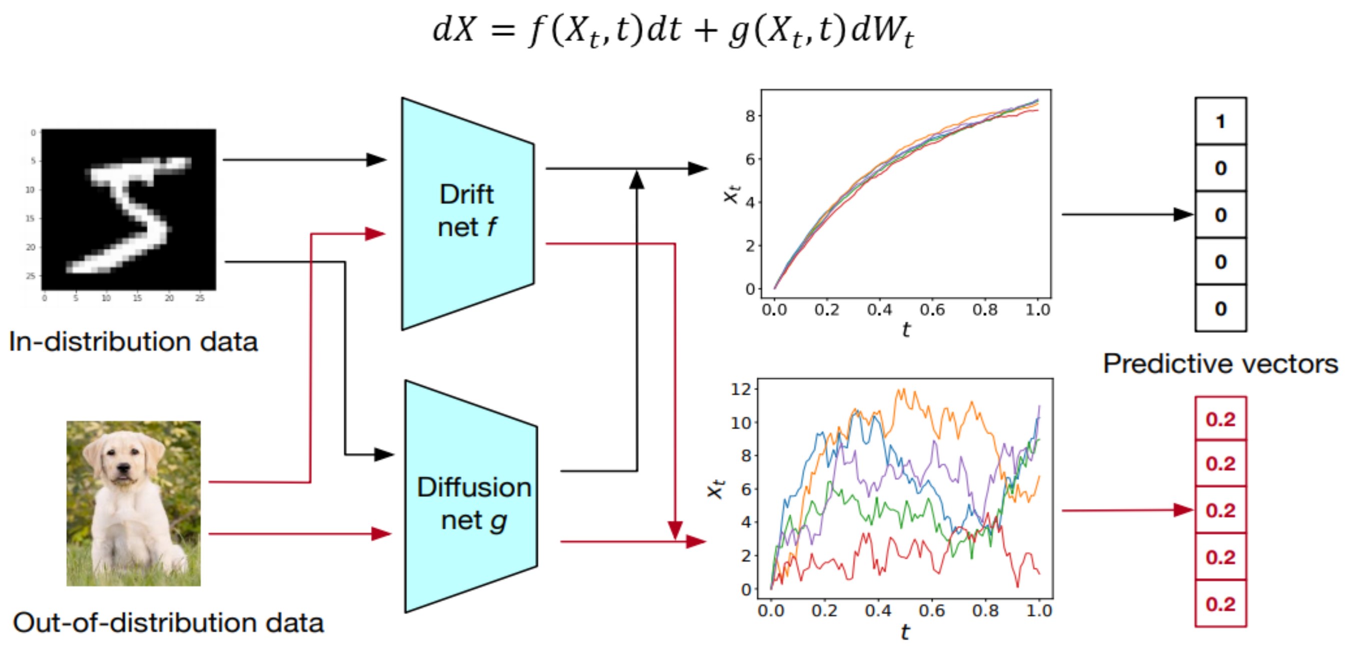
How to improve by an order of magnitude: use knowledge of numerical methods!

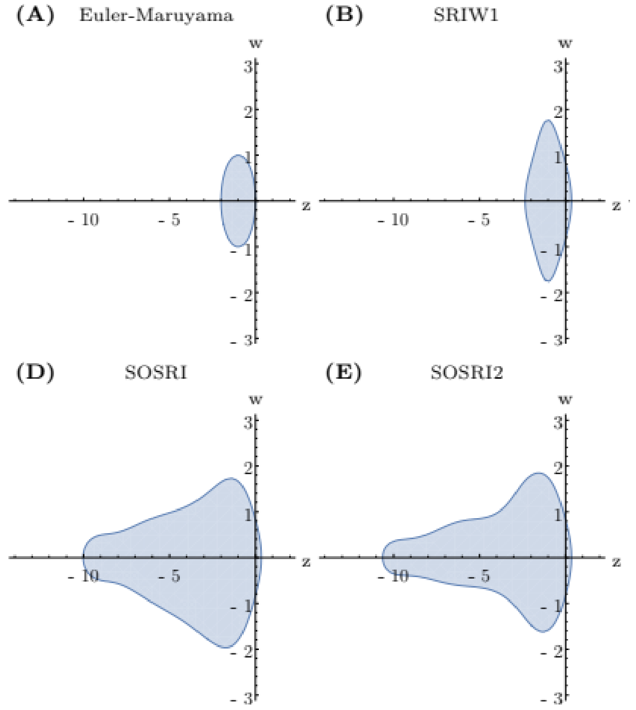
10x Neural ODE training vs previous regularization, 2x faster prediction time vs vanilla neural ODE

Method	Train Loss	Test Loss	Train Time (hr)	Prediction Time (s)
Vanilla NODE	3.48	3.55	1.75	0.53
TayNODE	4.21	4.21	12.3	0.22
SRNODE	**3.52**	**3.58**	**0.87**	**0.20**



Neural SDEs improve generalization. Can we improve them?





Add noise and uncertainty quantification to continuous layer methods via stochastic differential equations

New improved stability SDE solvers with adaptivity and automatic stiffness detection

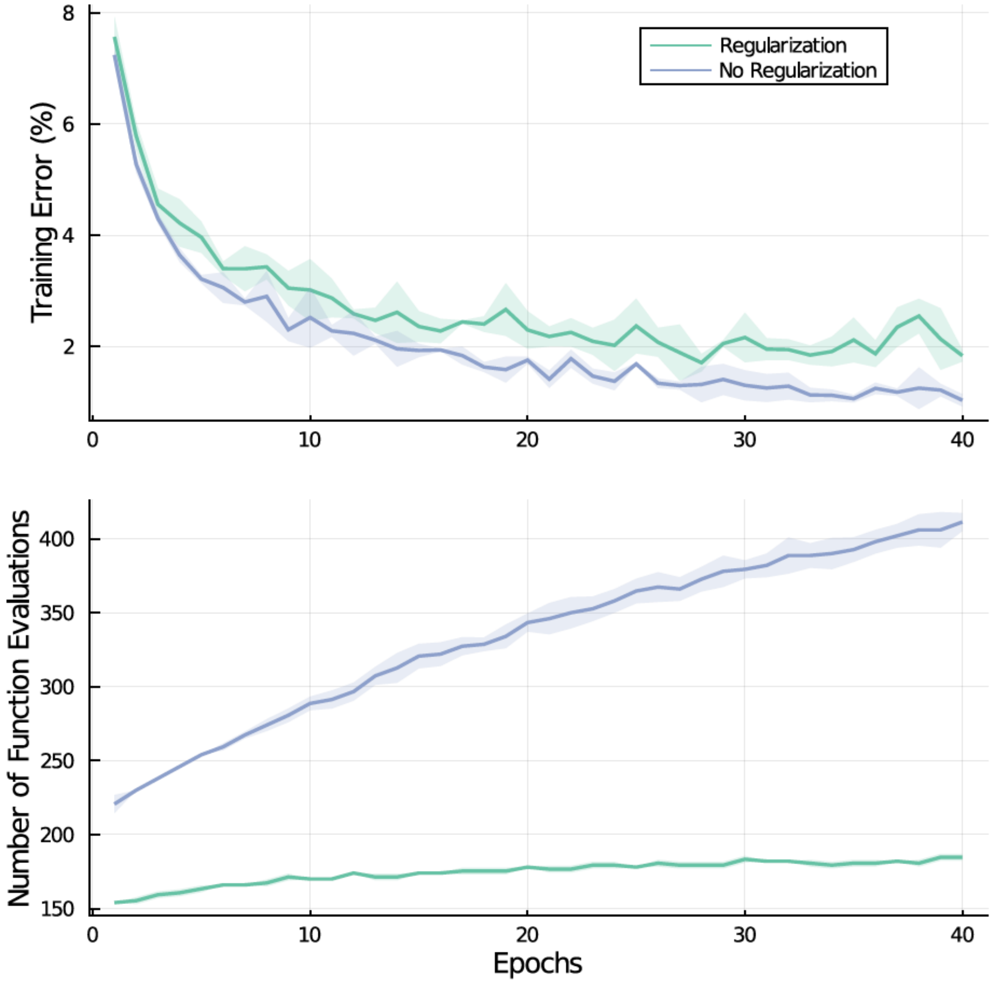
Liu, X., Si, S., Cao, Q., Kumar, S. and Hsieh, C.J., 2019. Neural sde: Stabilizing neural ode networks with stochastic noise. *arXiv preprint arXiv:1906.02355.*

Rackauckas, C. and Nie, Q., 2020, September. Stability-optimized high order methods and stiffness detection for pathwise stiff stochastic differential equations. In 2020 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1-8). IEEE. **(Quality Submission Award)**

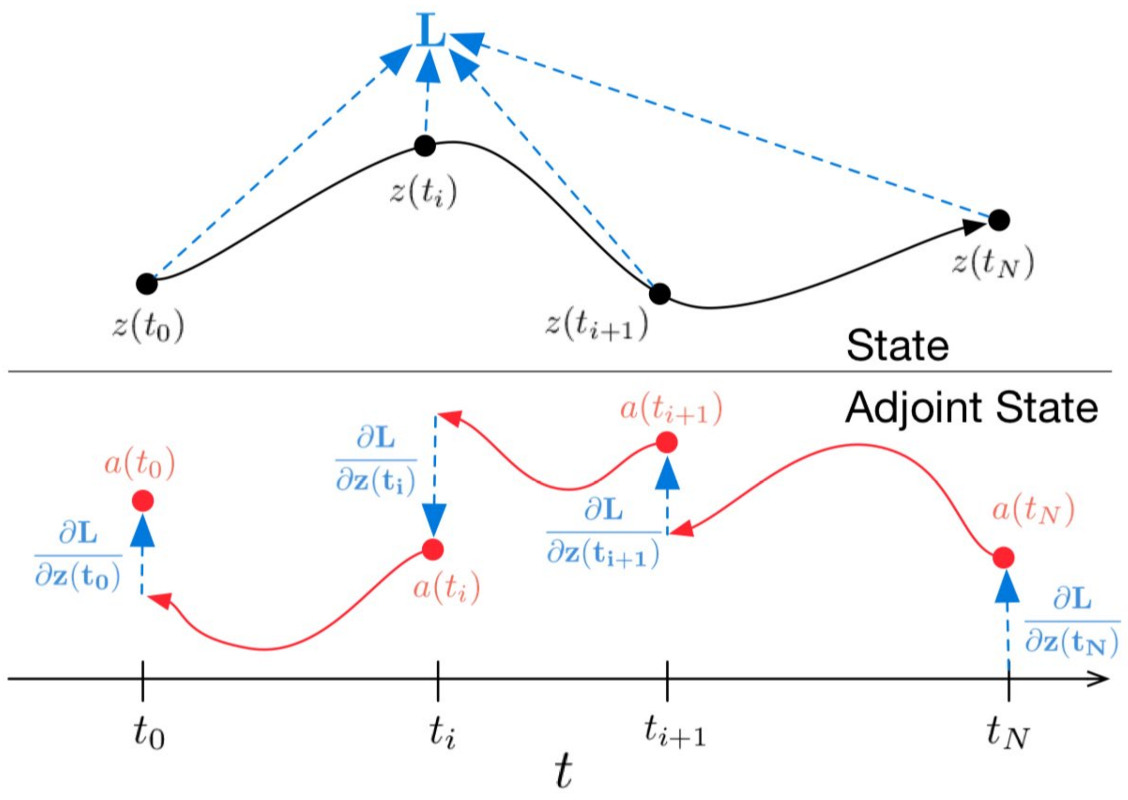
Major improvements to Neural SDEs on MNIST

Method	Train Accuracy (%)	Test Accuracy (%)	Train Time (hr)	Prediction Time (s)
Vanilla NSDE	98.97	96.95	6.32	15.07
RegNSDE	**98.16**	**96.27**	**4.19**	**7.23**

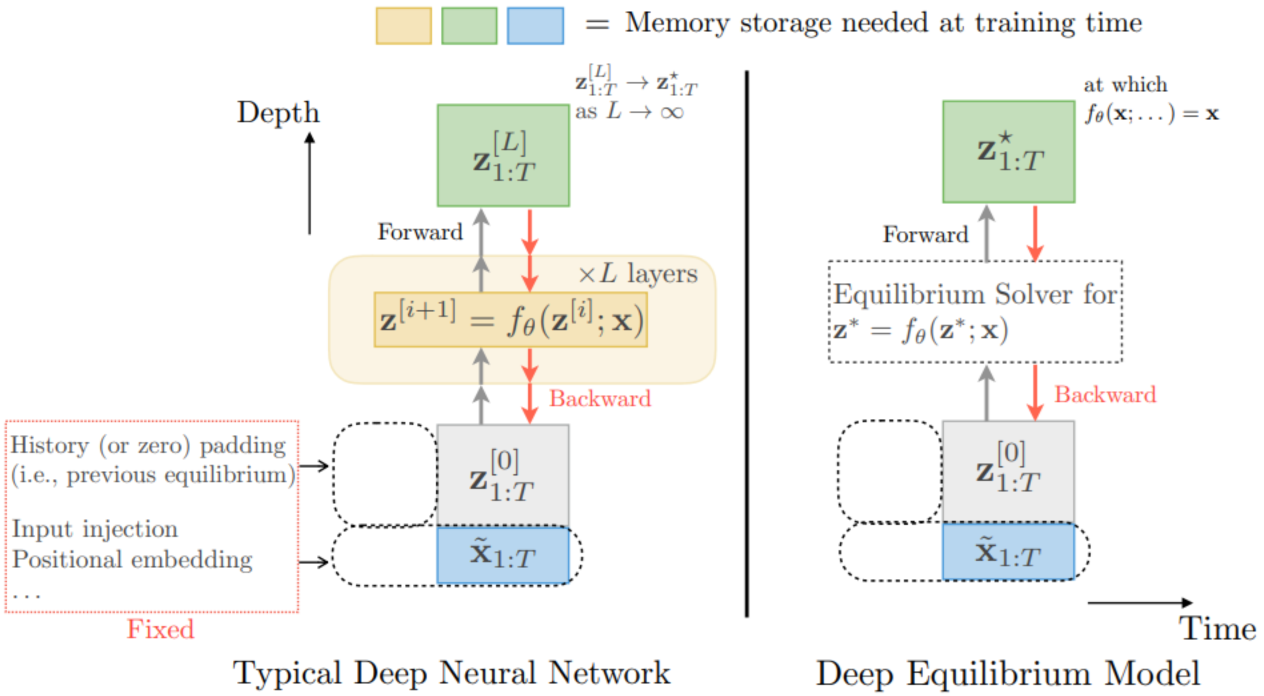
Double Neural SDE prediction speed!



Implicit Layer Machine Learning



Neural ODE



Deep Equilibrium Model

Implicit ML: Neural ODEs, Deep Equilibrium Models (DEQs), etc.

Infinite-Time Neural ODEs... Faster?

- **Forward Pass:** Solve the Steady State Equation

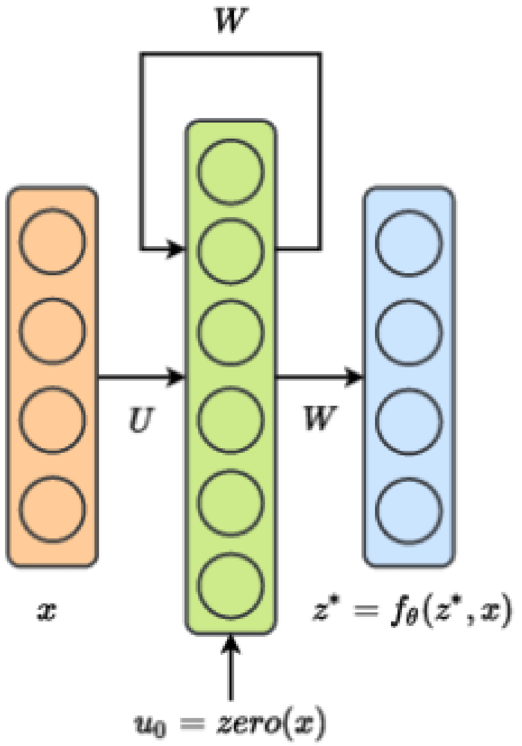
$$\frac{dz}{dt} = f_\theta(z;x) - z = 0$$

with the initial condition $u_0 = 0$ and parameters θ.

- **Backward Pass:** Solve the Linear Equation

$$\left(I - \left(\frac{\partial f_\theta(z^*;x)}{\partial z^*}\right)^T\right) g = y$$

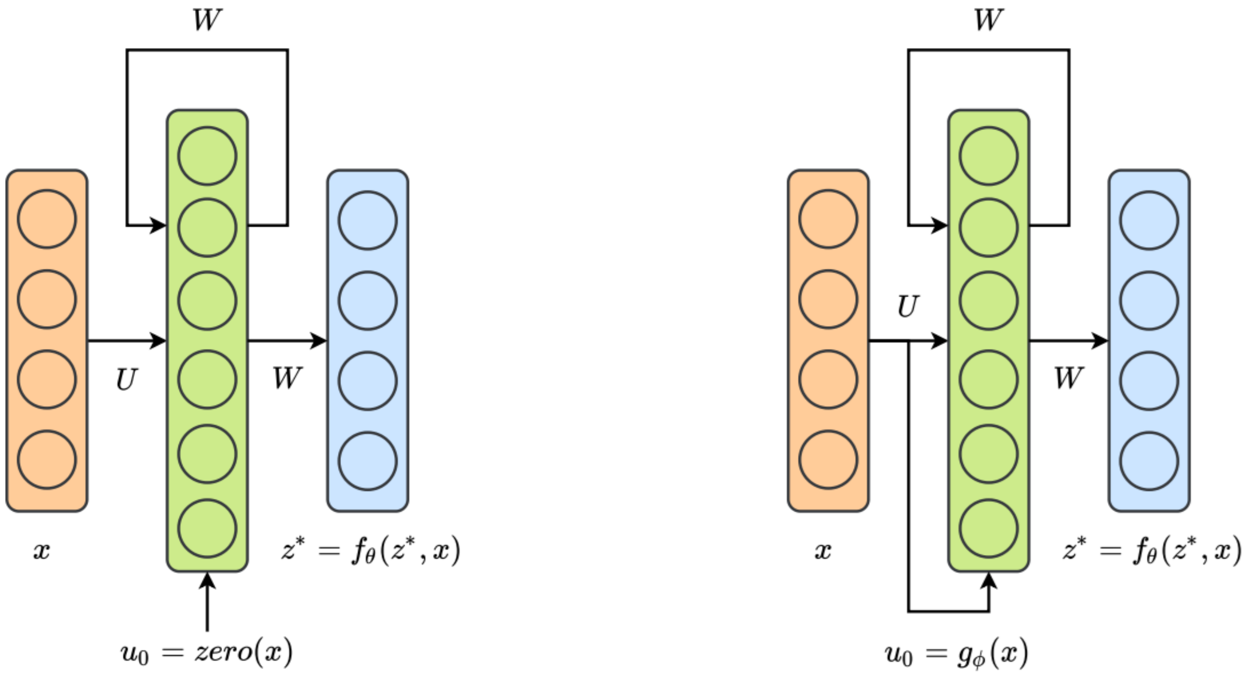
  - Simple old VJPs (Reverse Mode Autodiff) + Newton Krylov Solvers (Fast!!)
  - No need to store intermediate computations (Memory Efficient!!)



Blog post with starter code:
https://julialang.org/blog/2021/10/DEQ/

Infinite Neural ODEs are paradoxically easier to train

From Implicit to Implicit-Explicit Machine Learning



(a) Vanilla DEQ (b) Skip DEQ

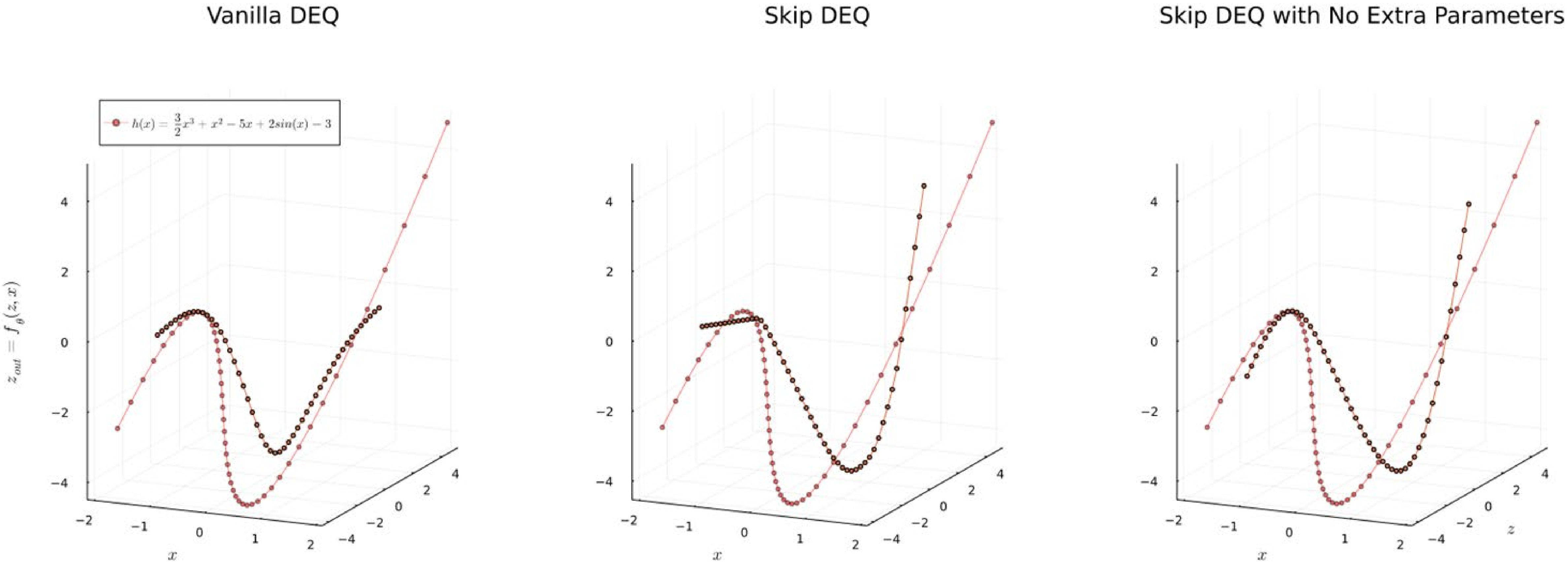
problem $u_0 = g_\phi(x)$ (See Figure 2). We jointly optimize for $\{\theta, \phi\}$ by adding an auxiliary loss function:

$$\mathcal{L}_{skip} = \lambda_{skip} \| f_\theta(z^*, x) - g_\phi(x) \|_1$$

Intuitively, our explicit model g_ϕ better predicts a value closer to the steady-state (over the training iterations), and hence we need to perform fewer iterations during the forward pass. Given that its prediction is relatively free in

Making a combined ML architecture gives the benefits of both

Animations Show It Works



https://github.com/SciML/FastDEQ.jl

Cool animations

Continuous+Skip DEQ: Much Faster and Robust ML Training

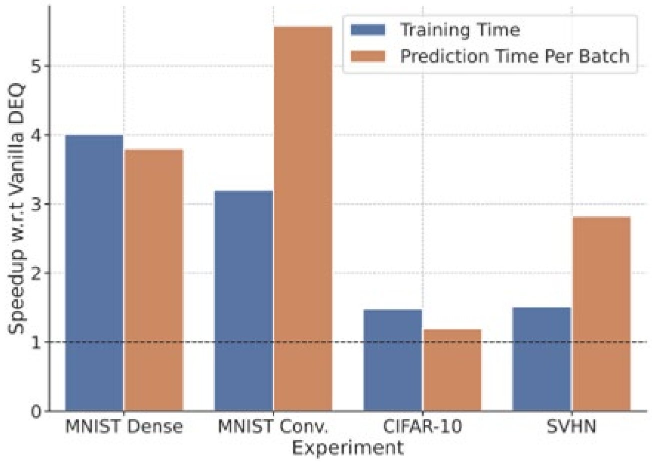


Figure 1. **Training and Prediction Speedups for Skip DEQ**: The best Skip DEQ model is on average **2.55x faster to train** and **3.349x faster during prediction time.**

Model	Dynamical System	Trainable Parameters	Testing Accuracy (%)	Testing NFE	Convergence Depth	Training Time (hr)	Prediction Time Per Batch (s)
DEQ	Continuous	**171K**	77.946 ± 1.617	126.000 ± 0.000	✗	7.862 ± 0.461	2.62 ± 0.164
Skip DEQ	Continuous	229K	79.466 ± 1.502	$\mathbf{95.320 \pm 3.093}$	$\mathbf{14.886 \pm 0.515}$	5.653 ± 0.655	2.190 ± 0.092
Skip DEQ V2	Continuous	**171K**	$\mathbf{80.466 \pm 1.817}$	96.025 ± 0.045	15.004 ± 0.007	$\mathbf{5.314 \pm 0.185}$	$\mathbf{2.187 \pm 0.031}$
DEQ	Discrete	**171K**	81.756 ± 0.525	$\mathbf{23.000 \pm 0.000}$	✗	$\mathbf{3.825 \pm 0.096}$	$\mathbf{0.801 \pm 0.010}$
Skip DEQ	Discrete	229K	81.956 ± 0.555	$\mathbf{23.000 \pm 0.000}$	✗	4.060 ± 0.096	0.818 ± 0.018
Skip DEQ V2	Discrete	**171K**	$\mathbf{83.386 \pm 0.235}$	$\mathbf{23.000 \pm 0.000}$	✗	4.695 ± 0.262	0.824 ± 0.018

Table 3. **CIFAR-10 Classification**: Skip DEQs generalize better to the testing data outperforming DEQ by **0.2% - 2.52%**. Continuous Skip DEQs **converge to the steady state** and **reduce the training time by 1.39 - 1.47x** and **prediction time by 1.2x**. Discrete forms are unable to converge to the steady state and hence using Skip DEQ has a detrimental effect on the training and prediction timings.

Want to Dig Deeper into the Trade-Offs?

Engineering Trade-Offs in Automatic Differentiation: from TensorFlow and PyTorch to Jax and Julia

December 25 2021 in Julia, Programming, Science, Scientific ML | Tags: automatic differentiation, compilers, differentiable programming, jax, julia, machine learning, pytorch, tensorflow, XLA | Author: Christopher Rackauckas

Check out this blog post!

Blog post for more information