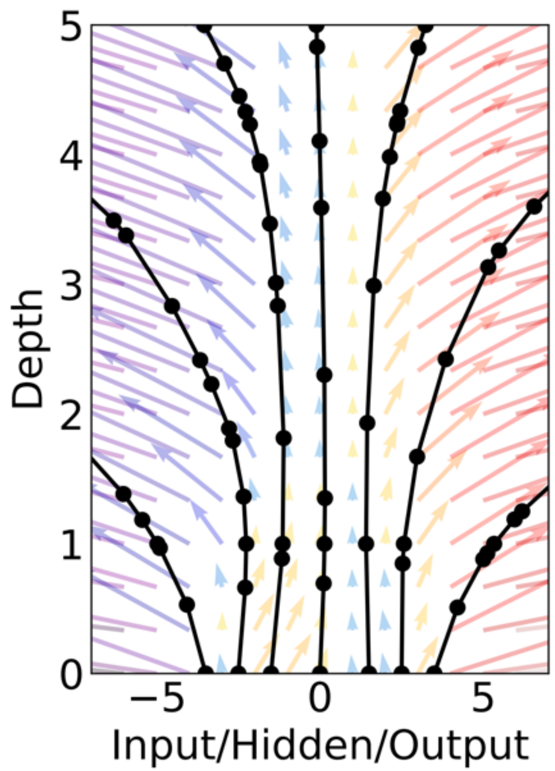
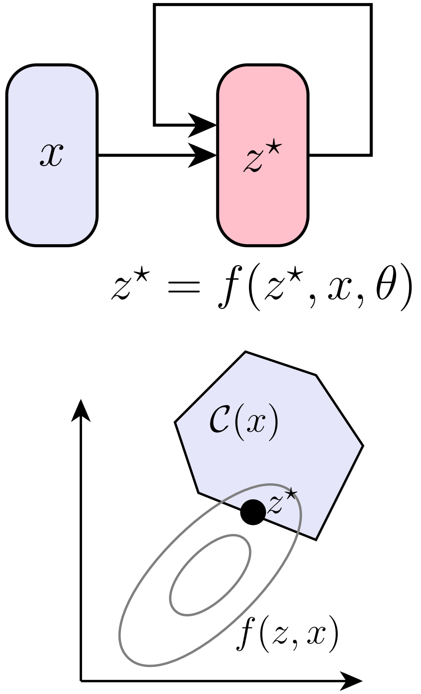




Deep Implicit Layers:

Neural ODEs, Equilibrium Models, and Beyond

http://implicit-layers-tutorial.org

David Duvenaud
University of Toronto
and Vector Institute

J. Zico Kolter
Carnegie Mellon and
Bosch Center for AI

Matt Johnson
Google Brain

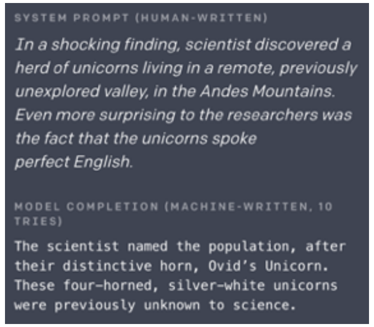
What do we want to do with deep learning?

Image classification

Semantic segmentation

Modeling continuous-time systems

Solving constrained optimization



SYSTEM PROMPT (HUMAN-WRITTEN)

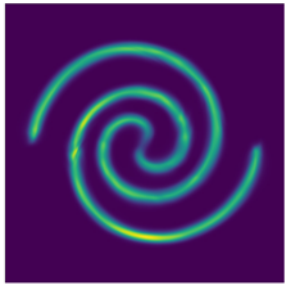
In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

`The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.`

Language modeling

Generative models



Smooth density estimation

“Traditional” deep learning domains

Emerging applications

Picture credits: [Krizhevsky et al., 2012; Bai et al., 2020; Grathwohl et al., 2018; Radford et al., 2019; Keras et al., 2018; Wang et al., 2019]

What do we want to do with deep learning?

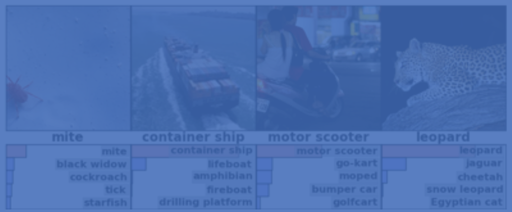


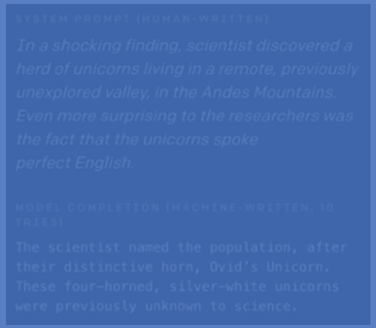
Image classification

Semantic segmentation

Modeling continuous-time systems

Solving constrained optimization

Implicit Layers



SYSTEM PROMPT (HUMAN-WRITTEN)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Language modeling

Generative models

Smooth density estimation

“Traditional” deep learning domains

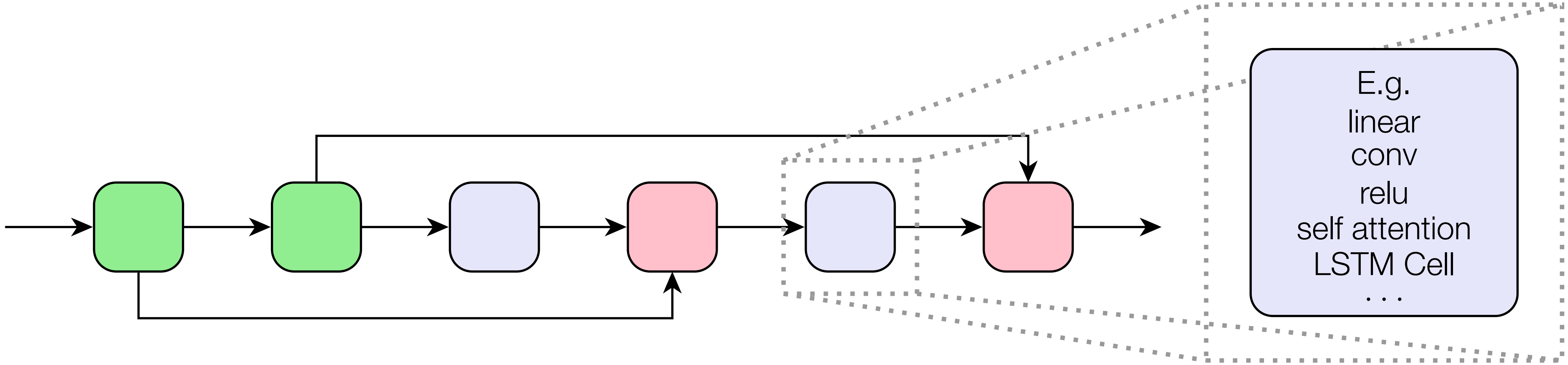
Emerging applications

Picture credits: [Krizhevsky et al., 2012; Bai et al., 2020; Grathwohl et al., 2018; Radford et al., 2019; Keras et al., 2018; Wang et al., 2019]

What is a “layer”?

A layer, for the purposes of this tutorial, is a ***differentiable parametric function***

Deep learning architectures are typically constructed by composing together many such layers, then training the complete system end-to-end via backpropagation

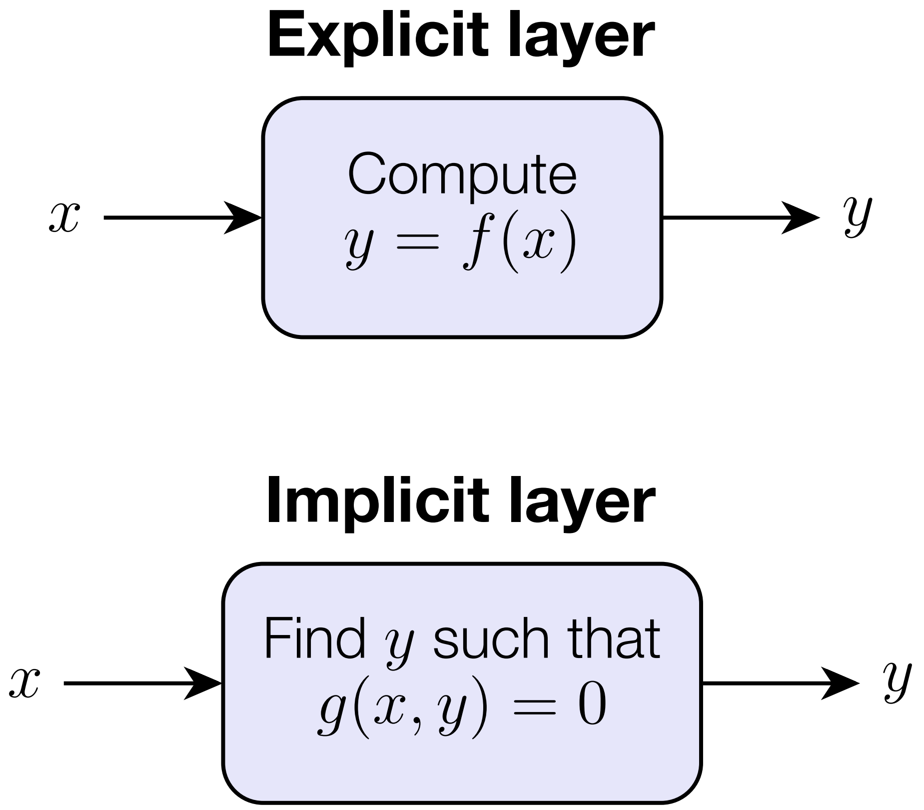


Explicit vs. Implicit layers

Virtually all commonly-used layers are ***explicit***, in that they provide a computation graph for computing the forward pass, and backprop through that computation graph

Implicit layers, in contrast, define a layer in terms of ***satisfying some joint condition of the input and output***

- *Many* examples: differential equations, fixed point iteration, optimization solutions, etc



Why use implicit layers?

1. **Powerful representations:** compactly represent complex operations such as integrating differential equations, solving optimization problems, etc

2. **Memory efficiency:** no need to backpropagate through intermediate components, via implicit function theorem

3. **Simplicity:** Ease and elegance of designing architectures

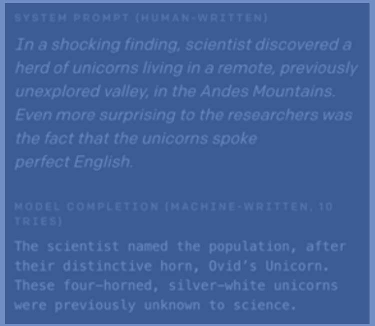
4. **Abstraction:** Separate “what a layer should do” from “how to compute it”, an abstraction that has been extremely valuable in many other settings

What do we want to do with deep learning?

Deep Equilibrium Models

Image classification

Semantic segmentation


SYSTEM PROMPT (HUMAN-WRITTEN)

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

MODEL COMPLETION (MACHINE-WRITTEN, 10 TRIES)

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

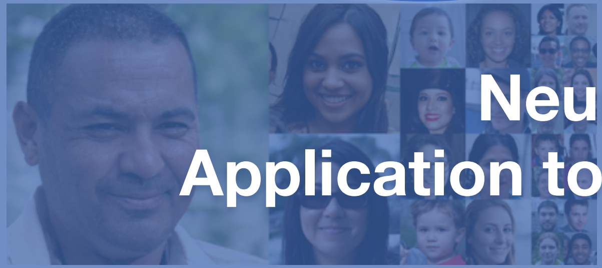
Language modeling

Modeling continuous-time systems

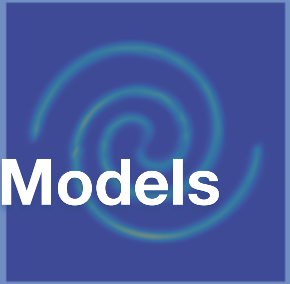
Differentiable Optimization

Solving constrained optimization

Neural ODEs + Application to Flow-based Models



Generative models



Smooth density estimation

"Traditional" deep learning domains

Emerging applications

Picture credits: [Krizhevsky et al., 2012; Bai et al., 2020; Grathwohl et al., 2018; Radford et al., 2019; Keras et al., 2018; Wang et al., 2019]

This tutorial

Goal of this tutorial is to provide you with **an understanding of the techniques, motivations, and applications for implicit layers in modern deep learning**

Heavy focus on:

- Mathematical foundations of implicit layers + automatic differentiation
- Examples including Neural ODEs, deep equilibrium models, differentiable optimization
- Starter code and highlights of future directions

Detailed notes + code available in companion website:
http://implicit-layers-tutorial.org

Outline

Background and applications of implicit layers

The mathematics of implicit layers

Deep Equilibrium Models

Neural ODEs

Differentiable optimization

Future directions

Outline

Background and applications of implicit layers

The mathematics of implicit layers

Deep Equilibrium Models

Neural ODEs

Differentiable optimization

Future directions

Myth: Implicit layers are new to neural networks

Reality: The history of implicit layers in deep learning goes back to the late 80s, highlighted by the papers of [Pineda, 1987] and [Almeida, 1987] (papers below, respectively), going by the name *recurrent backpropagation*

Example 1: Recurrent backpropagation with first order units

Consider a dynamical system whose state vector **x** evolves according to the following set of coupled differential equations

$$dx_i/dt = -x_i + g_i\Big(\sum_j w_{ij}x_j\Big) + I_i \qquad (1)$$

where i=1,...,N. The functions g_i are assumed to be differentiable and may have different forms for various populations of neurons. In this paper we shall make no

A differential equation layer!

2. BACKPROPAGATION IN FEEDBACK PERCEPTRONS

Consider a graded perceptron network, and designate by x_k the external inputs ($k = 1,\dots,K$), by y_i the outputs of the units ($i = 1,\dots,N$), by s_i the result of the sum performed at the input of unit i, and by o_p the external outputs ($p \in O$, where O is the set of units producing external outputs). The static equations of the perceptron network are

$$s_i = \sum_{n=1}^{N} a_{ni}\, y_n + \sum_{k=1}^{K} b_{ki}\, x_k + c_i \qquad i = 1,\dots,N \qquad (1)$$

$$y_i = S_i(s_i) \qquad i = 1,\dots,N \qquad (2)$$

$$o_p = y_p \qquad p \in O \qquad (3)$$

where a_{ni} and b_{ki} are weights, c_i is a bias term, and S_i is the nonlinear function in unit i (usually a sigmoid). In a feedforward perceptron, the

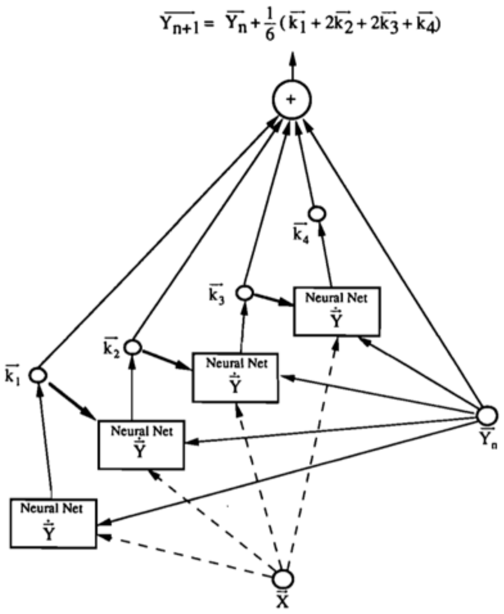
A fixed point equation layer!

Largely fell out of use in favor of explicit network structure

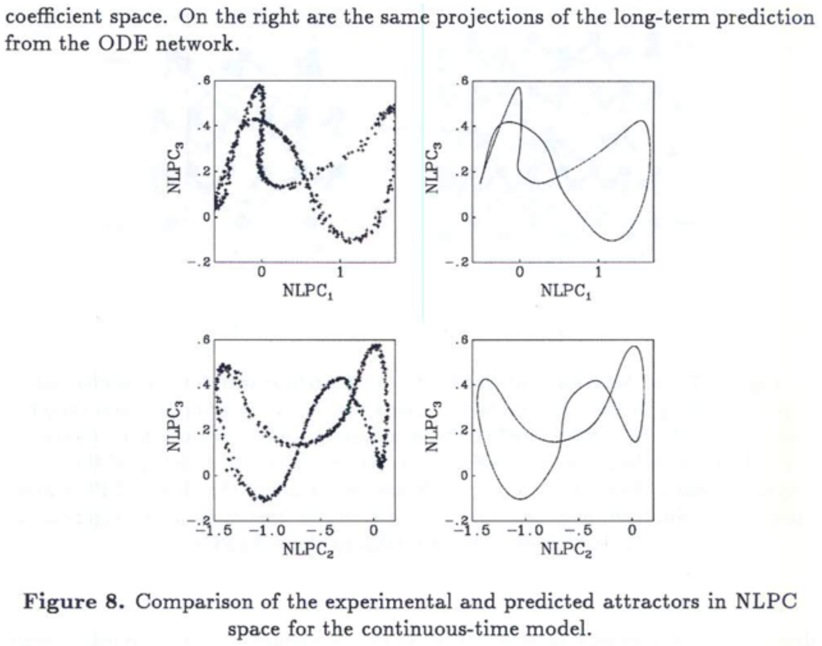
Much of the current efforts are a revisiting of this idea, using the tools and techniques of modern architectures and automatic differentiation tools

The “Implicit Layer Winter”

Although implicit layers were not prominent in ML, they did find a great number of use cases within applied engineering domains in the 90s, 2000s

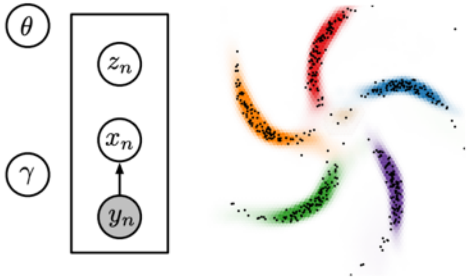


[Rico-Martinez et al., 1992]
Runge-Kutta integrator with neural network dynamics

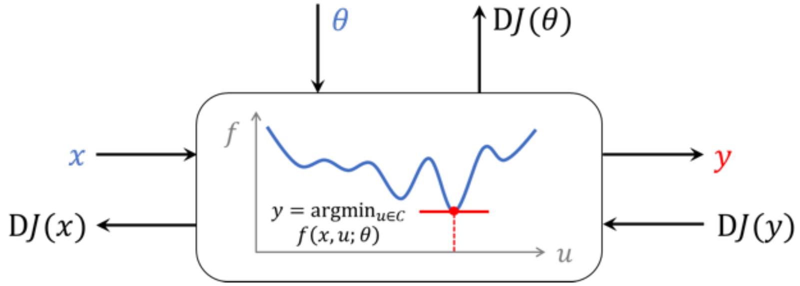


[Rico-Martinez and Kevrekidis, 1995]
Modeling carbon monoxide crystallization using differentiable implicit trapezoidal integration

Differentiable optimization

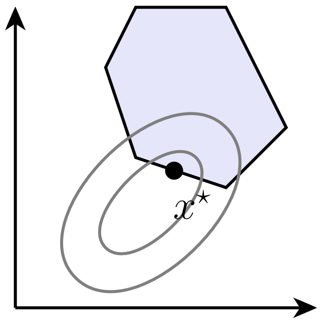


Structured Variational Autoencoder
[Johnson et al., 2016]
Differentiate through graphical model inference

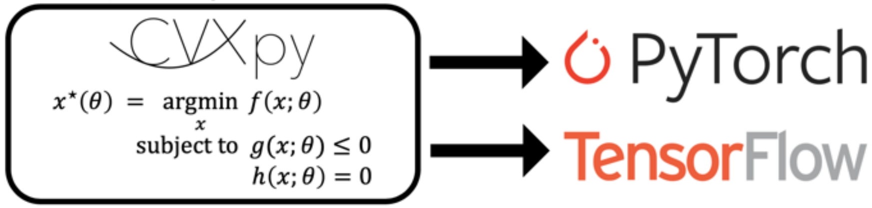


Deep Declarative Networks
[Gould et al., 2019; Gould et al., 2016]
Parameterize layers as general (non-convex) optimization problems

$$
\begin{aligned}
z_{i+1} = \quad & \underset{z}{\text{argmin}} \ \frac{1}{2} z^T Q(z_i) z + p(z_i)^T z \\
\text{subject to} \quad & A(z_i) z = b(z_i) \\
& G(z_i) z \le h(z_i)
\end{aligned}
$$

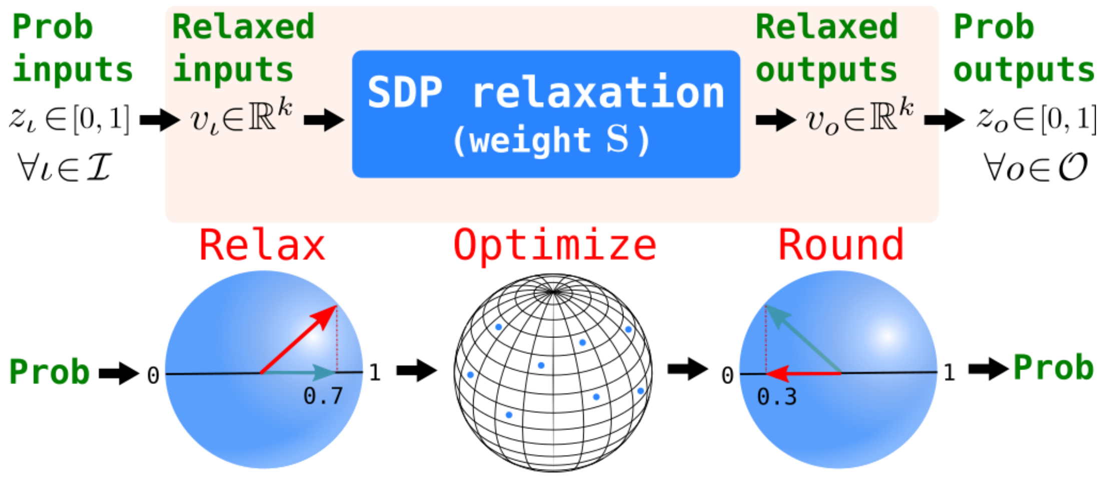


OptNet
[Amos and Kolter, 2017]
Differentiable quadratic programming layer

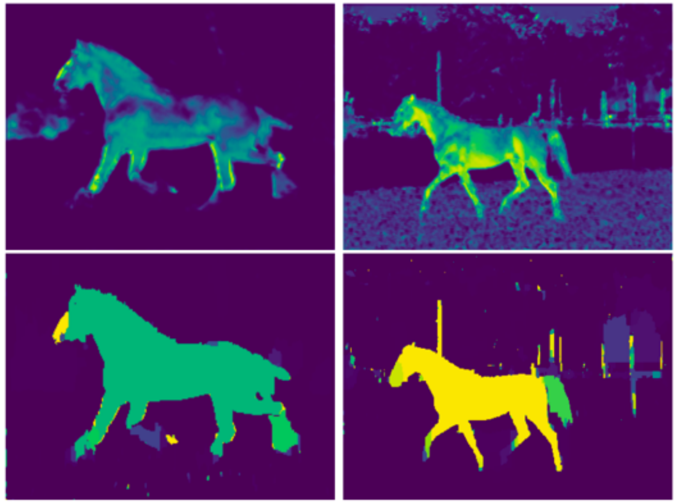


CvxpyLayers
[Agarwal et al.,, 2019]
Differentiable convex optimization easily integrated with automatic differentiation libraries

(Smoothed) combinatorial optimization



SatNet
[Wang et al., 2019]
Solve a smoothed version of a MAXSAT satisfiability problem via differentiable semidefinite programming



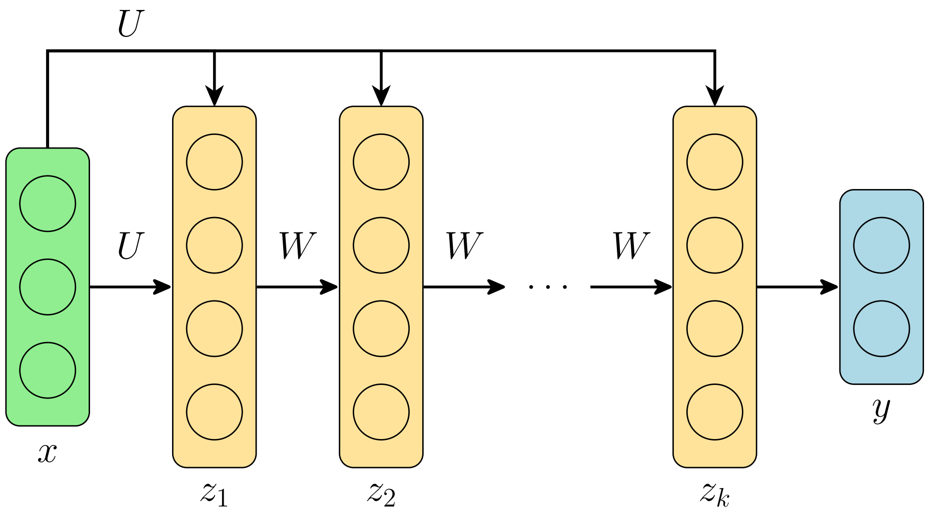
Differentiable submodular optimization
[Djolonga and Krause, 2017]
Differentiate through submodular minimization problems, such as graph cuts (application to image segmentation)

Deep equilibrium models

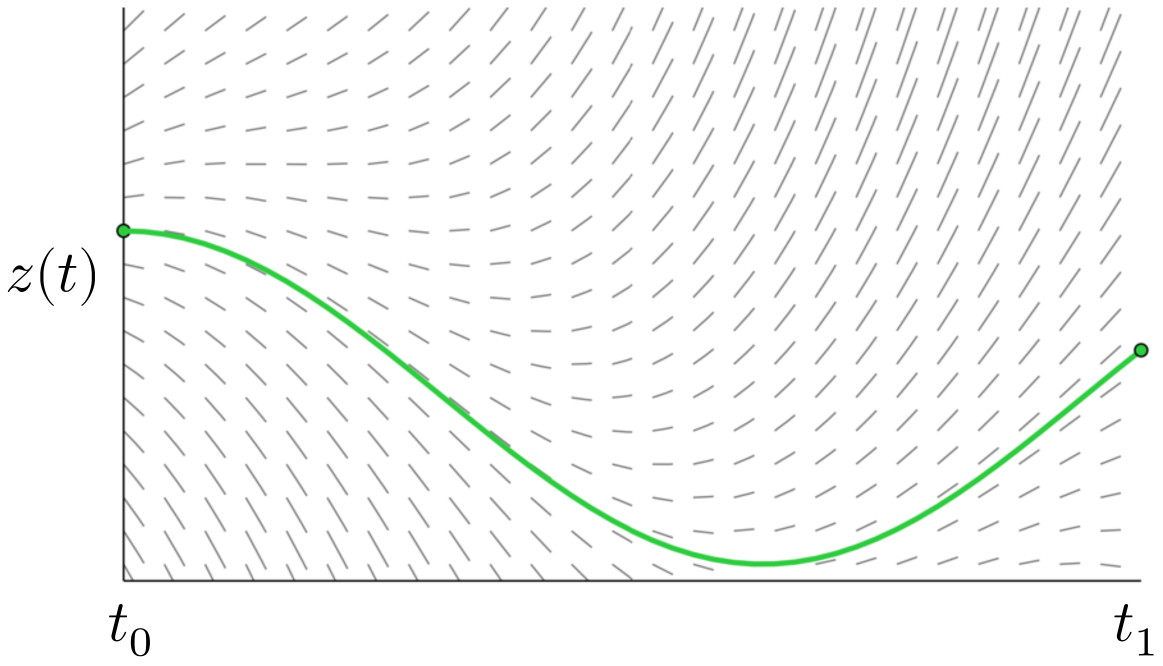
[Bai et al., 2019; Bai et al., 2020]

Represent modern deep networks using a single implicit layer

Near state of the art performance in large scale NLP and vision tasks such as semantic segmentation (using similar training approaches / network sizes)



Ordinary Differential Equations



If a vector z follows dynamics f:

$$\frac{dz}{dt} = f(z(t), t, \theta)$$

Can find $z(t_1)$ by starting at $z(t_0)$ and integrating until time t_1:

$$z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta) dt$$

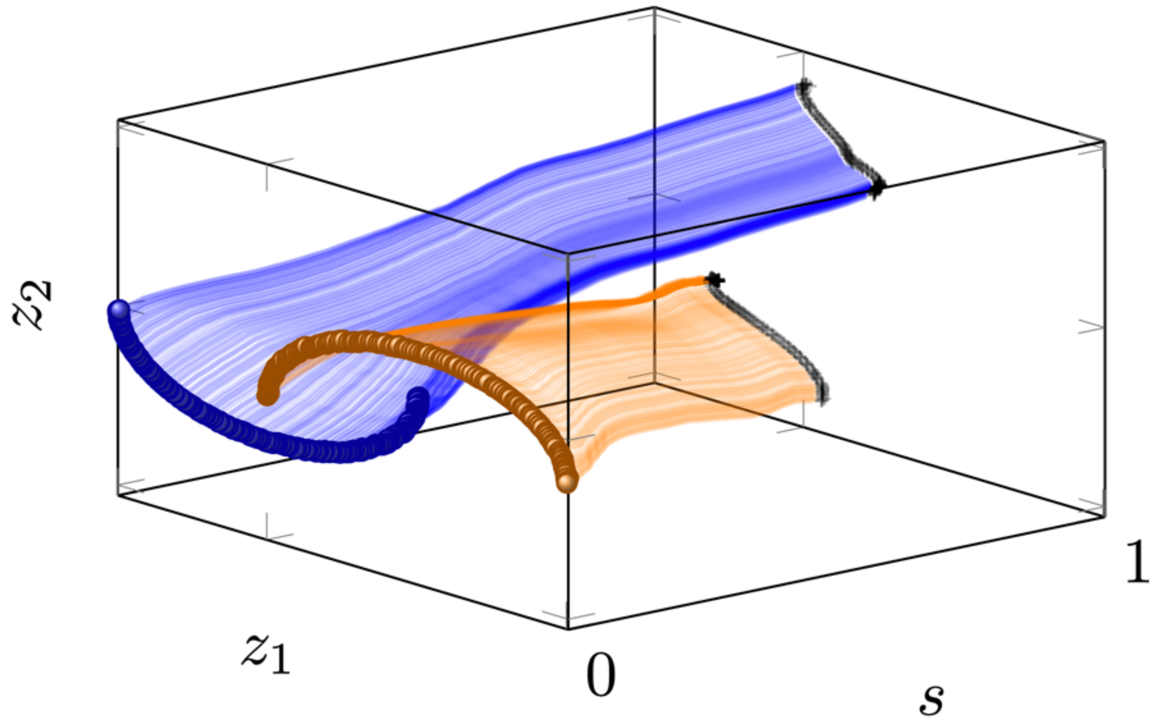
An implicit layer: $y = \mathrm{odeint}(f, x, t_0, t_1, \theta)$

What are Neural ODEs good for?

Equivalent to a resnet with infinitely many layers, each making an infinitesimal change.

Can be used anywhere a ResNet can.

In classifiers, data should be separable at output.



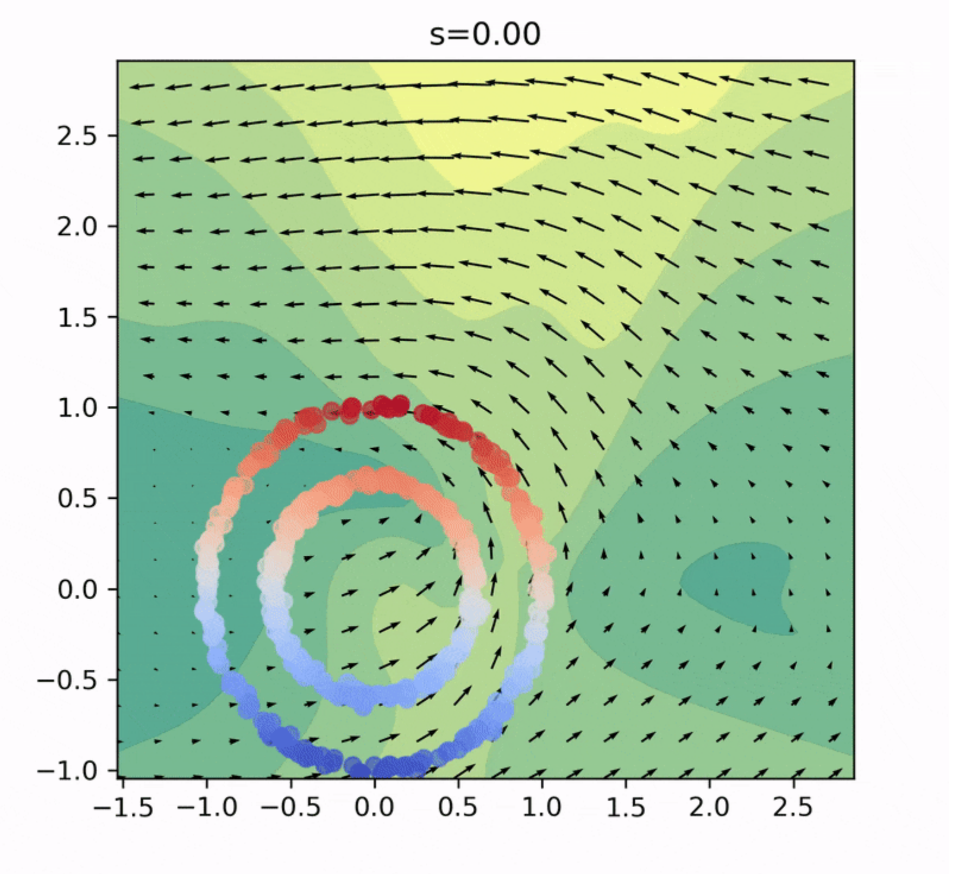
Dissecting Neural ODEs. Massaroli, Poli, Park, Yamashita, Asama (2020)

What are Neural ODEs good for?

Equivalent to a resnet with infinitely many layers, each making an infinitesimal change.

Can be used anywhere a ResNet can.

In classifiers, data should be separable at output.

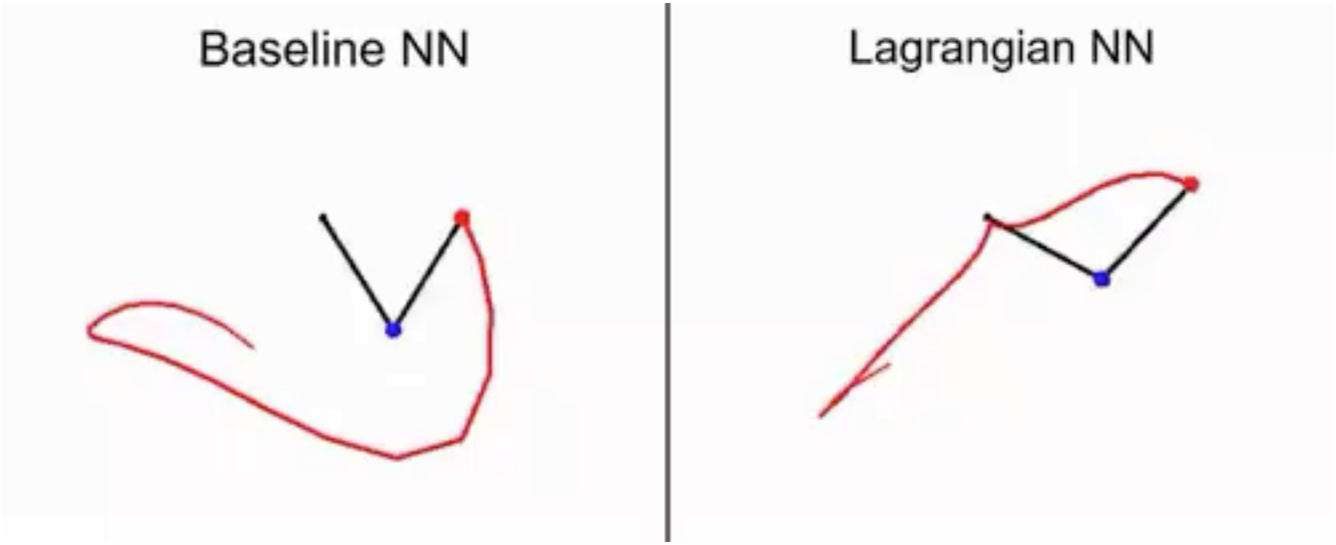


TorchDyn: A Neural Differential Equations Library.
Poli, Massaroli, Yamashita, Asama Park (2020)

Continuous-time Physical Models

Incorporate known structure or constraints, e.g. Hamiltonians, Lagrangians

$$\ddot{q} = \left(\frac{\partial^2 \mathcal{L}}{\partial \dot{q}^2}\right)^{-1} \left(\frac{\partial \mathcal{L}}{\partial q} - \dot{q}\frac{\partial^2 \mathcal{L}}{\partial q \partial \dot{q}}\right)$$

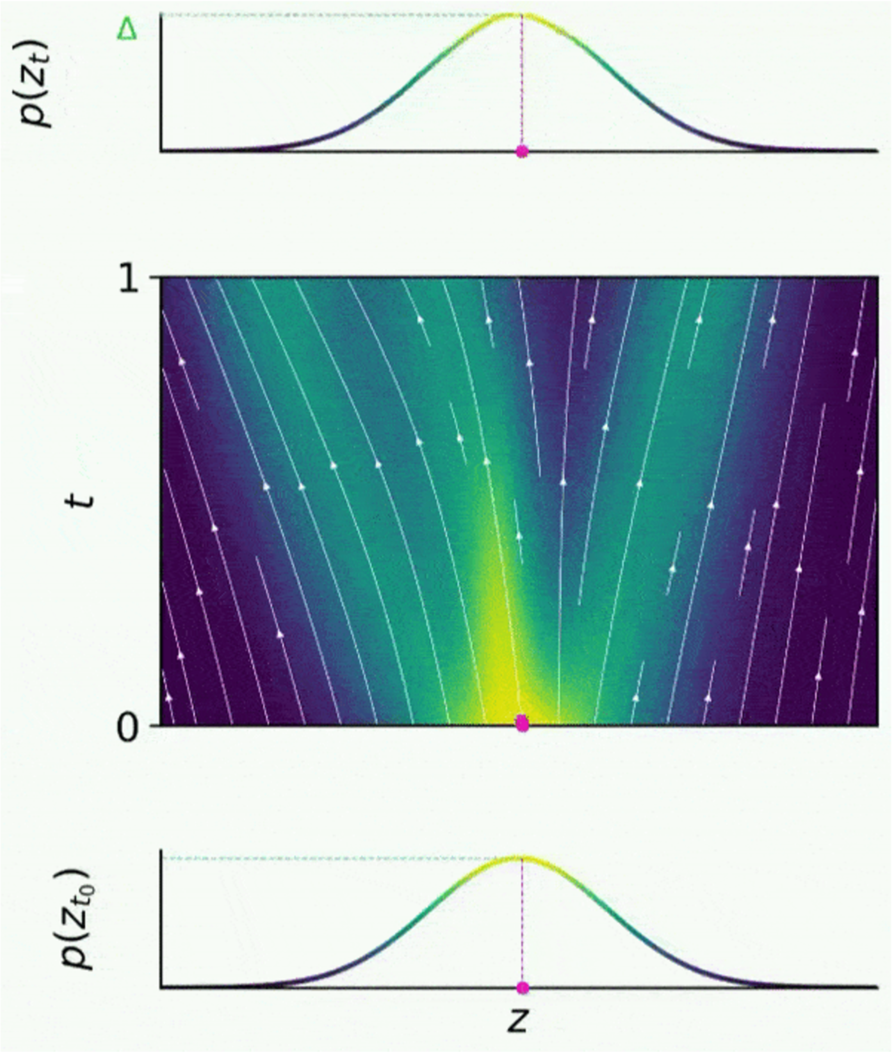


Hamiltonian Graph Networks with ODE Integrators. Sanchez-Gonzalez, Bapst, Cranmer, Battaglia (2019)
Lagrangian Neural Networks. Cranmer, Greydanus, SHoyer, Battaglia, Spergel, Ho (2020)
Differentiable Molecular Simulations for Control and Learning. Wang, Axelrod, Gómez-Bombarelli (2020)
Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control. Zhong, Dey, Chakraborty (2020)

Continuous Normalizing Flows

Transforms a simple density into a complex parametric density.

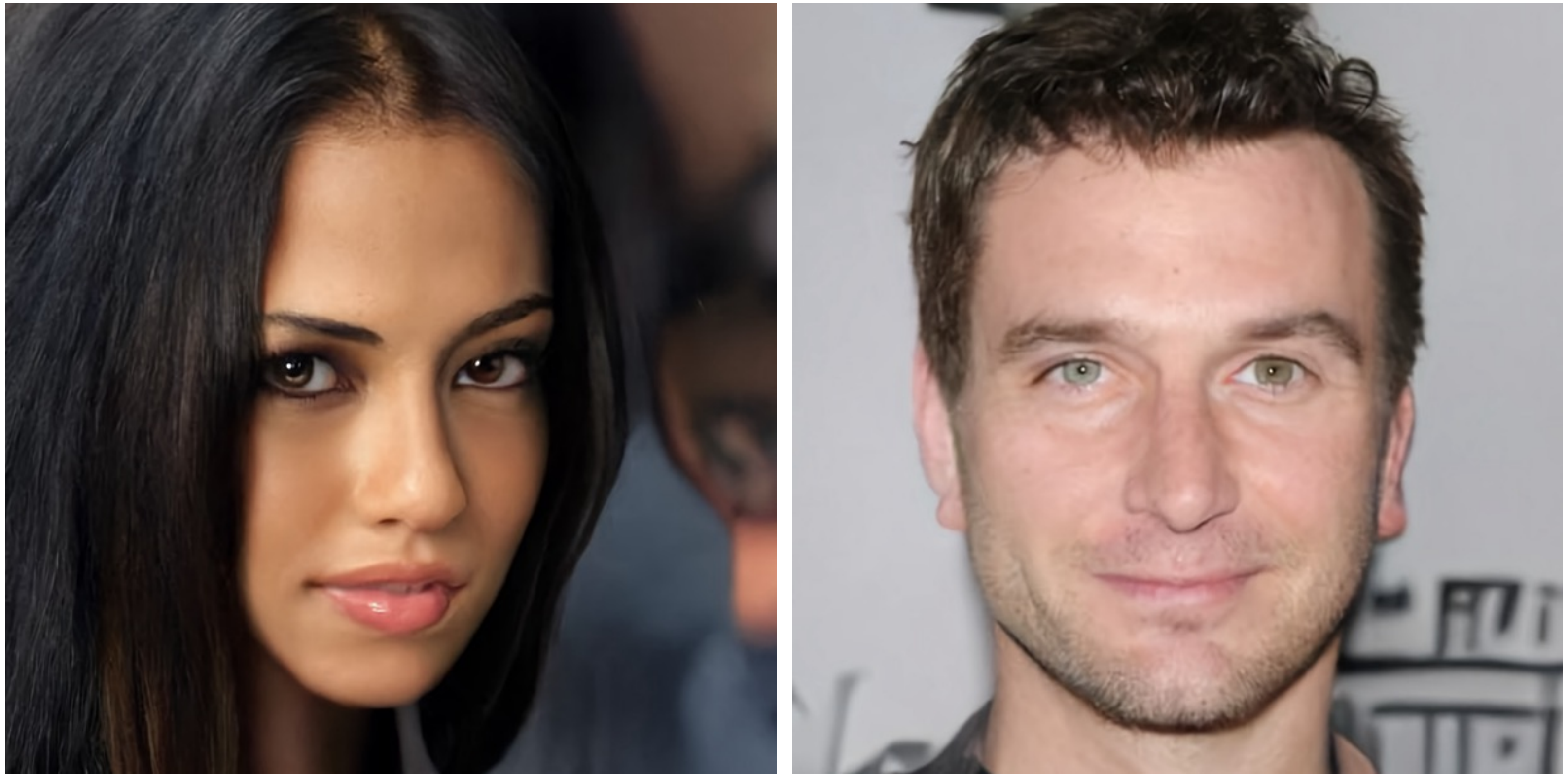
Change of variables formula easier to compute instantaneously.



Continuous Normalizing Flows

Score-based training scales to 1024x1024

Exact density available, but expensive



[Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. Score-Based Generative Modeling through Stochastic Differential Equations, 2020]

Continuous Normalizing Flows

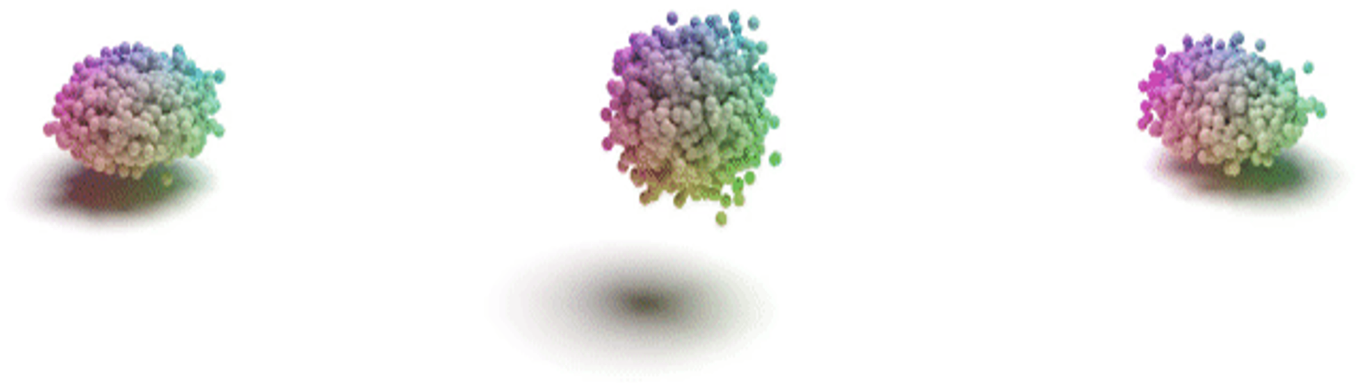
Conditional inpainting and colorization without retraining.

Requires iterative sampling procedure.

Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. Score-Based Generative Modeling through Stochastic Differential Equations. 2020

Continuous Normalizing Flows

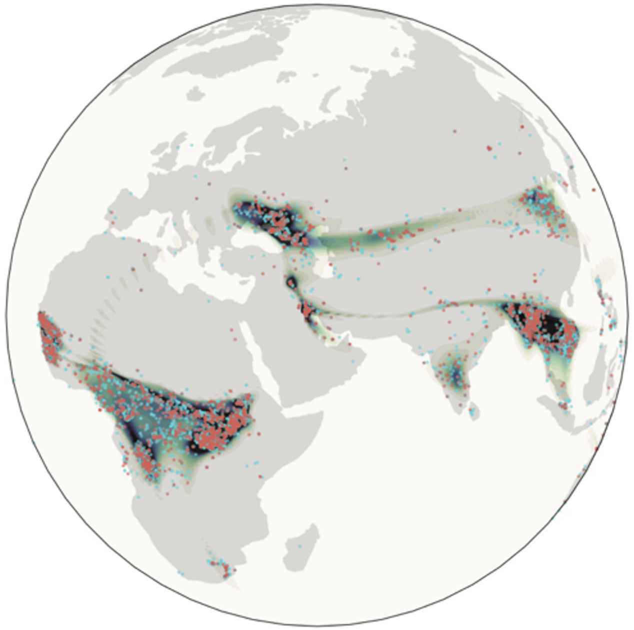
Can also parameterize
homeomorphisms
(non self-intersecting maps)



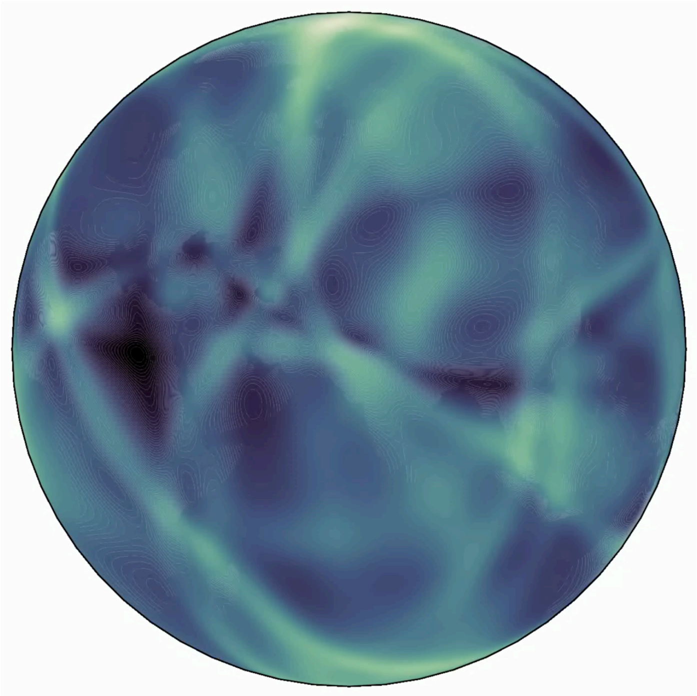
PointFlow: 3D Point Cloud Generation with
Continuous Normalizing Flows.
Yang, Huang, Hao, Liu, Belongie, Hariharan (2019).

Continuous Normalizing Flows

Can build flexible parametric density models on manifolds (e.g. spheres)



Fire



Thanks to Emile Mathieu

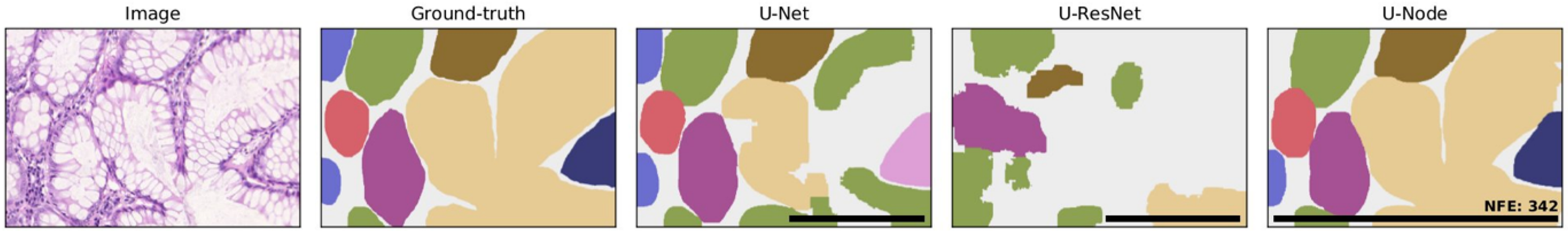
Riemannian Continuous Normalizing Flows [Mathieu and Nickel, 2020]
Neural Ordinary Differential Equations on Manifolds. [Falorsi and Forré, 2020]
Neural Manifold Ordinary Differential Equations. [Lou et al., 2020]

Applications in biology

Used for modeling cellular development trajectories.

[Tong et al. “TrajectoryNet: A Dynamic Optimal Transport Network for Modeling Cellular Dynamics”, 2020]

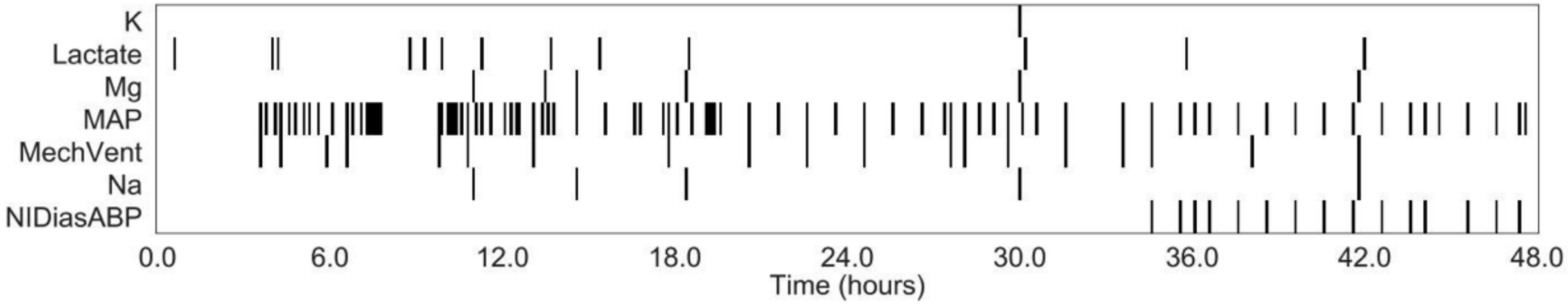
Used in convolutional u-net segmentation for colon screening.

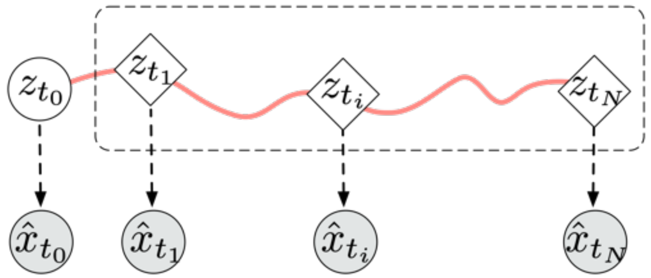


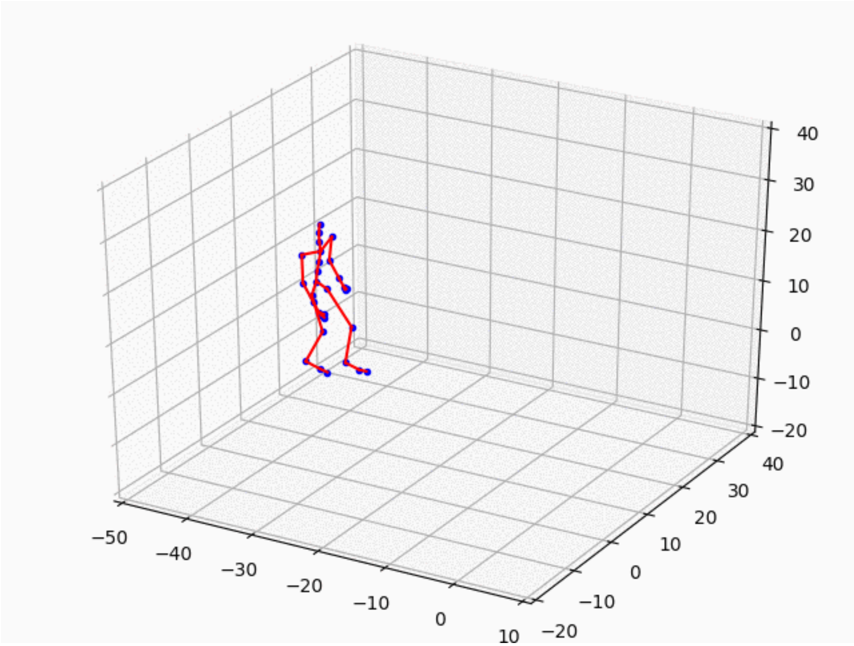
Neural Ordinary Differential Equations for Semantic Segmentation of Individual Colon Glands. [Pinckaers and Litjens, 2019].

Continuous-time Time Series Models

Can deal with data collected at irregular intervals natively.







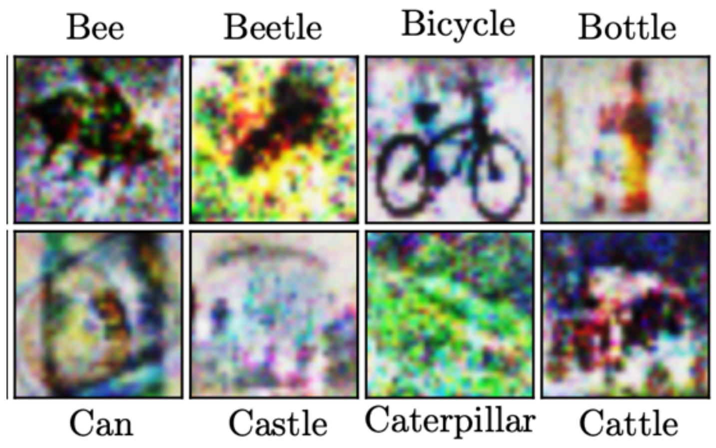
Latent ODEs for Irregularly-Sampled Time Series. Rubanova, Chen, Duvenaud (2020)
Neural Controlled Differential Equations for Irregular Time Series.
Kidger, Morrill, Foster, Lyons (2020)
GRU-ODE-Bayes: Continuous modeling of sporadically-observed time series. de Brouwer, Simm, Arany, Moreau. (2020)

Other Uses of Implicit Gradients

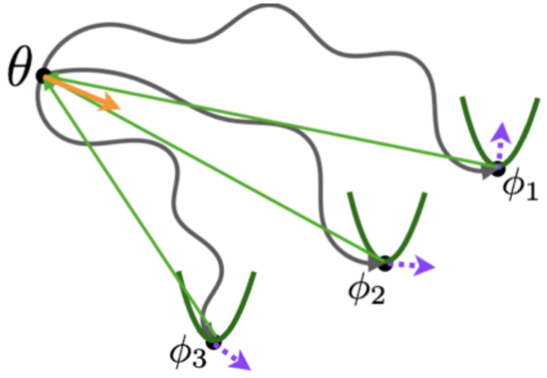
Can use implicit gradients to tune millions of hyperparameters.

Can also be used for meta-learning if inner loop is trained to convergence.

Dataset Distillation



Meta-learning (iMAML)



Meta-Learning with Implicit Gradients. Rajeswaran, Finn, Kakade, Levine (2019)
Optimizing Millions of Hyperparameters by Implicit Differentiation. Jonathan Lorraine, Paul Vicol, David Duvenaud (2019)
Gradient-Based Optimization of Hyper-Parameters. Yoshua Bengio. (2000)

Outline

Background and applications of implicit layers

The mathematics of implicit layers

Deep Equilibrium Models

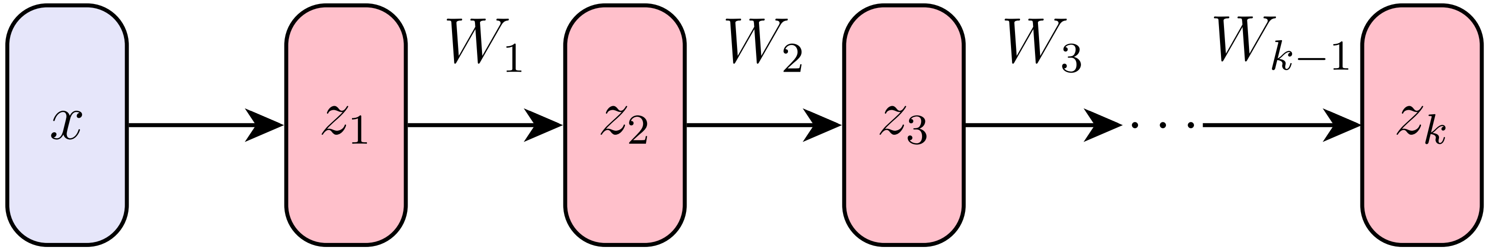
Neural ODEs

Differentiable optimization

Future directions

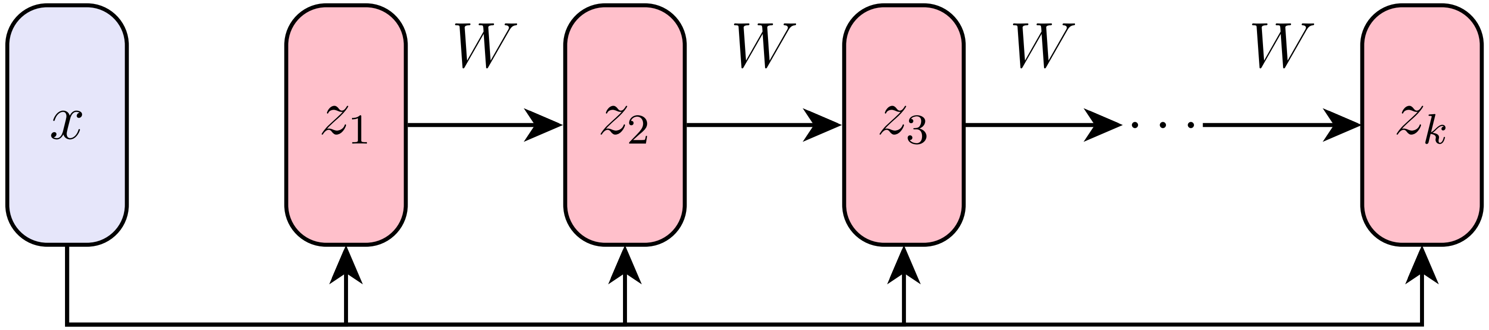
Motivating a simple implicit layer

Consider a traditional deep network applied to an input x



$$z_{i+1} = \sigma(W_i z_i + b_i)$$

We now modify this network in two ways: by re-injecting the input at each step, and by applying the *same* weight matrix at each iteration (weight tying)



$$z_{i+1} = \sigma(W z_i + x)$$

Iterations of deep weight-tied models

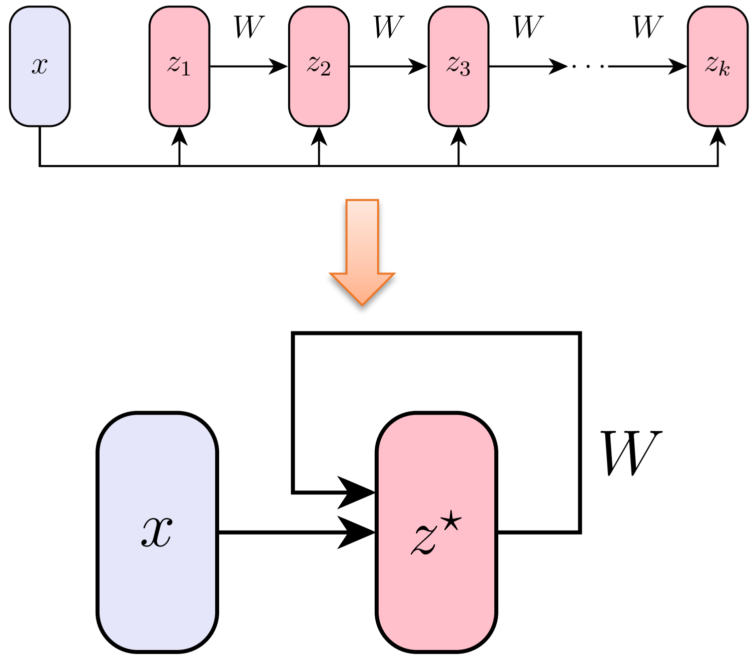
With a weight-tied model of this form, we are applying the *same* function repeatedly to the hidden units

$$z_{i+1} = \sigma(W z_i + x)$$

In many situations, we can design the network such that this iteration will converge to some *fixed point*, or *equilibrium point*

$$z^\star = \sigma(W z^\star + x)$$

This is precisely a recurrent backpropagation network, or a (minimal) deep equilibrium model



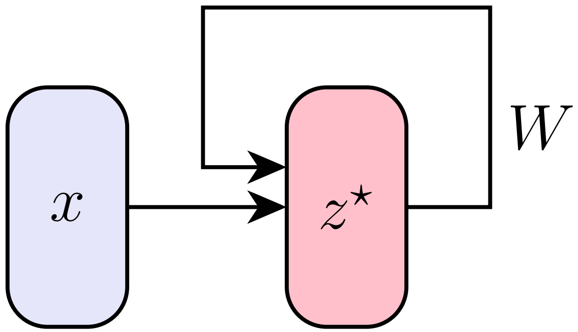
Simple instantiation: A tanh fixed point iteration

Let's consider a very simple form of such a fixed point layer, iterating:

$$z_{i+1} = \tanh(W z_i + x)$$

How do we compute the fixed point?

$$z^\star = \tanh(W z^\star + x)$$

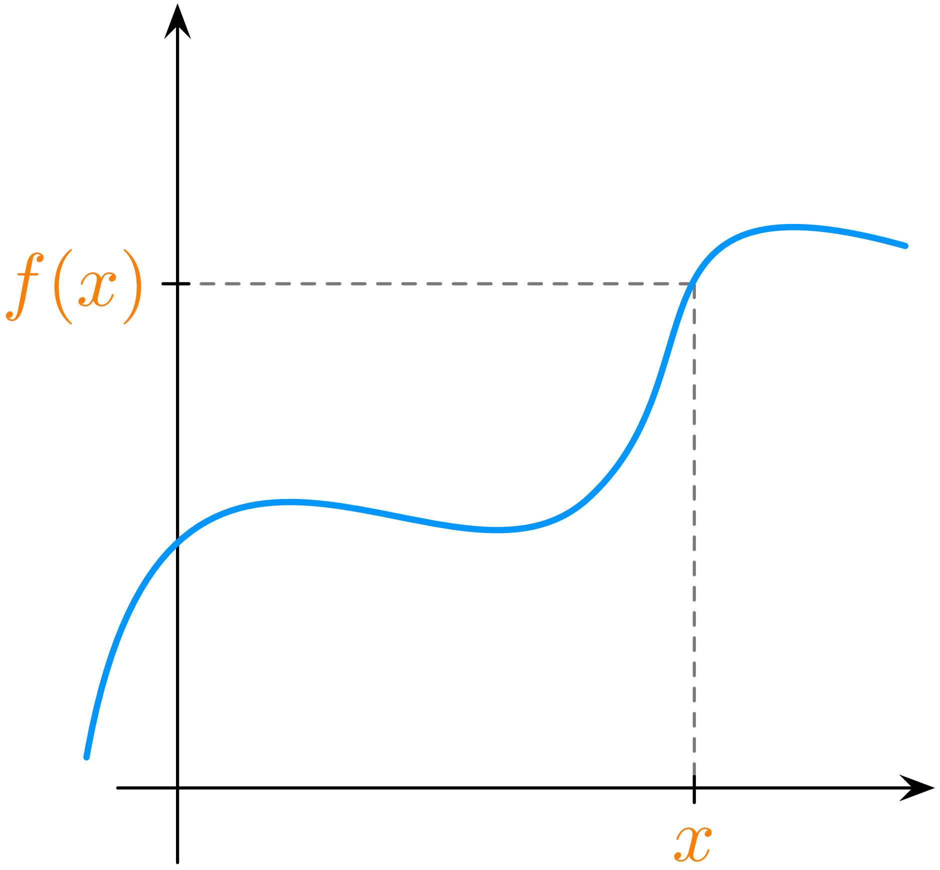


How do we integrate such a layer with backprop? Does the derivative exist?

To answer this, let's see a quick demo

Differentiation notation

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

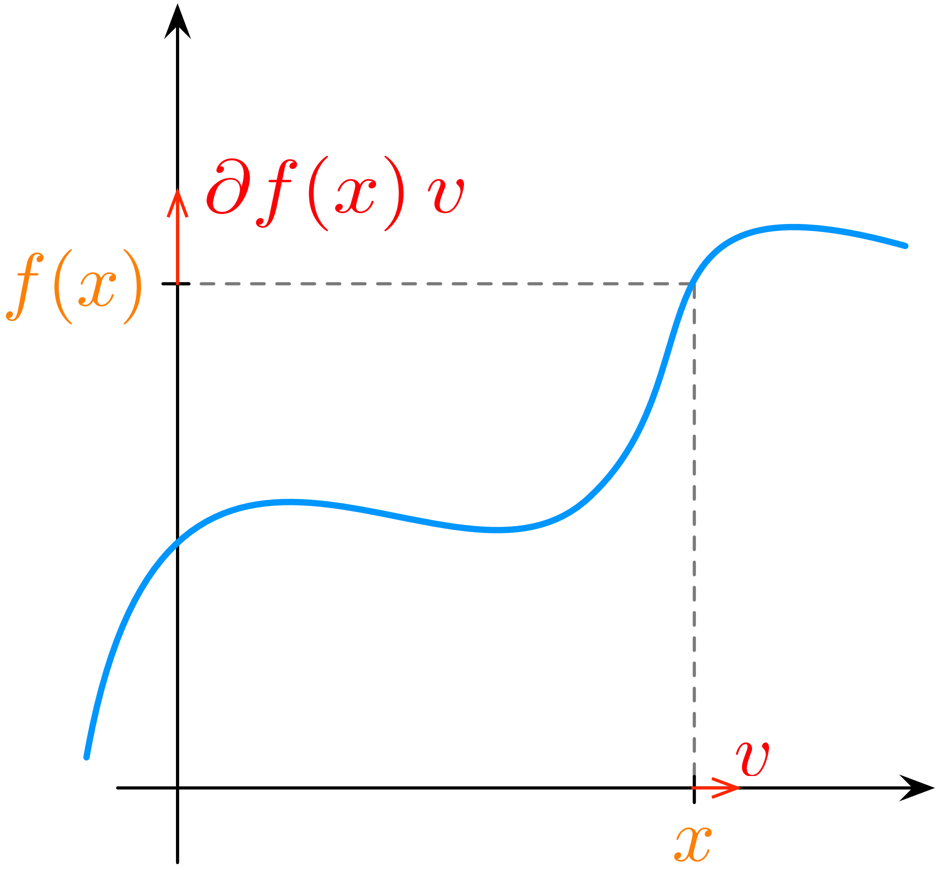


Differentiation notation

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m$$

$$\partial f(x) \in \mathbb{R}^{m \times n}$$



Differentiation notation

$$\partial_0 f(x, y) \equiv \partial g(x) \text{ where } g(x) = f(x, y)$$

$$\partial_1 f(x, y) \equiv \partial g(y) \text{ where } g(y) = f(x, y)$$

The implicit function theorem

Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p$, $z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and

2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,

2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and

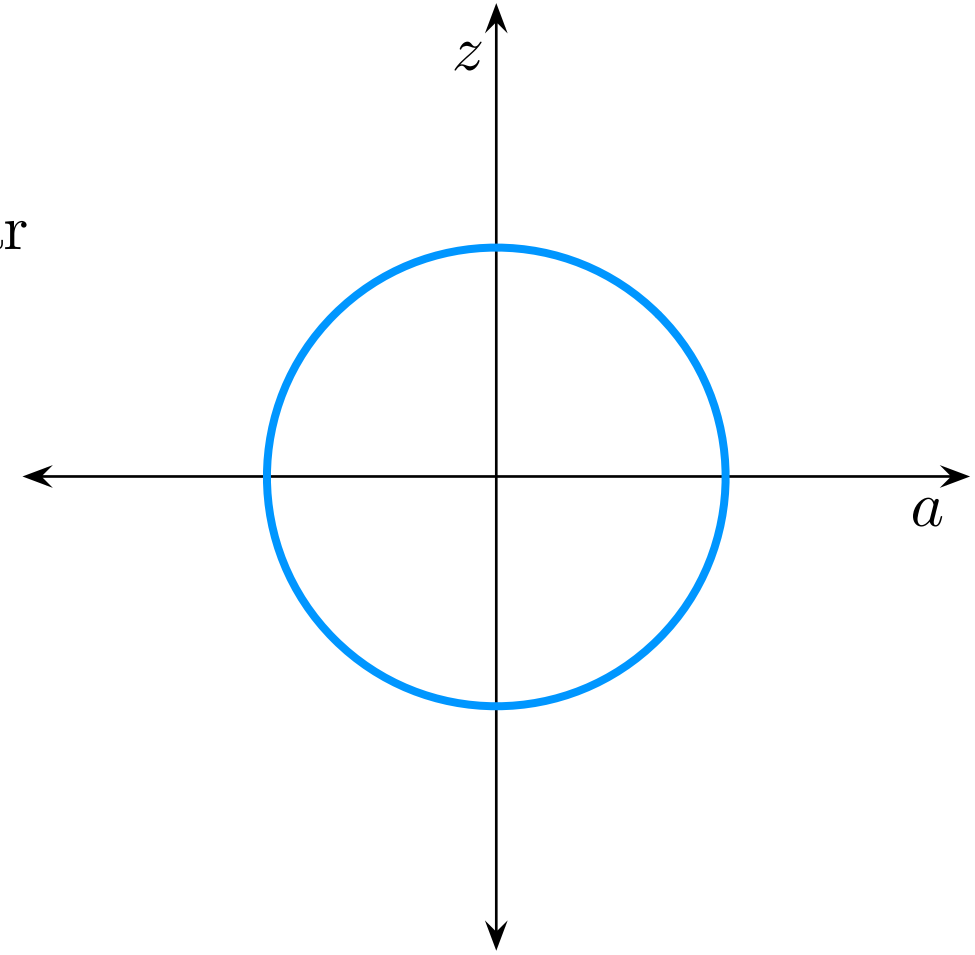
3. z^* is differentiable on S_{a_0}.

The implicit function theorem

$f(a, z) = a^2 + z^2 - 1 = 0$

Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p$, $z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.



Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

The implicit function theorem

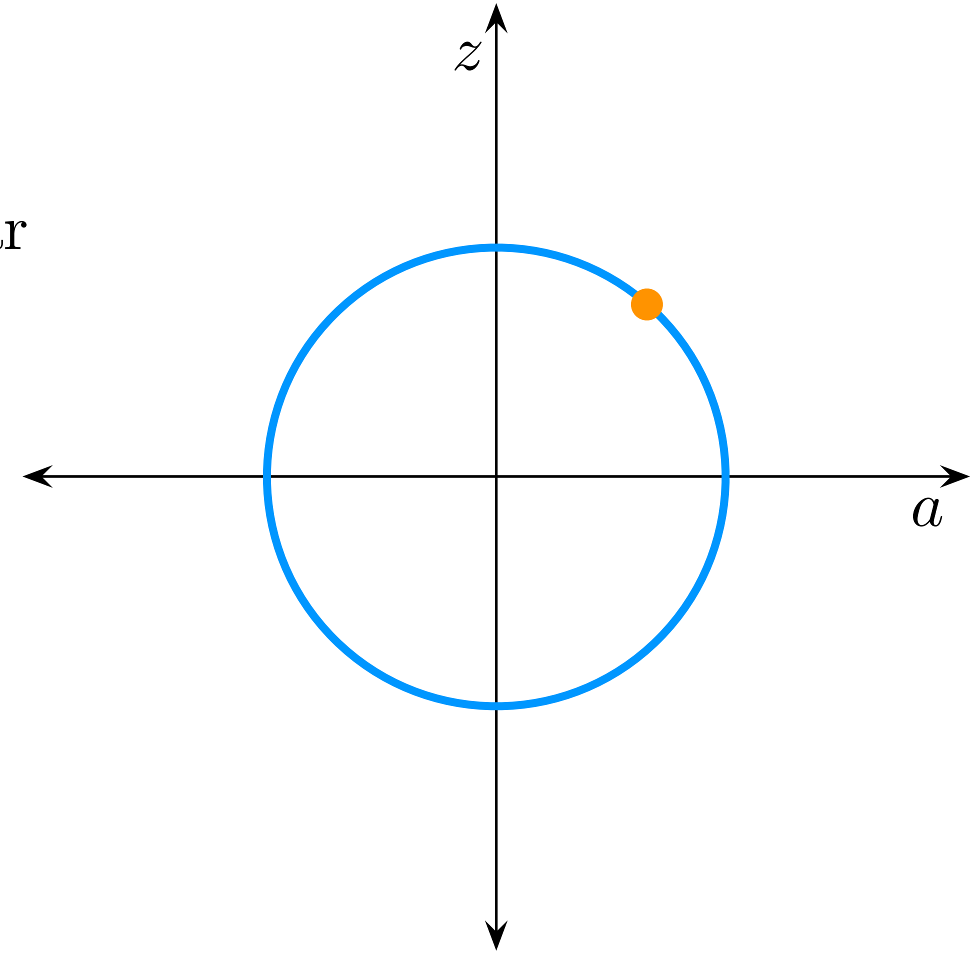
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$$f(a, z) = a^2 + z^2 - 1 = 0$$



The implicit function theorem

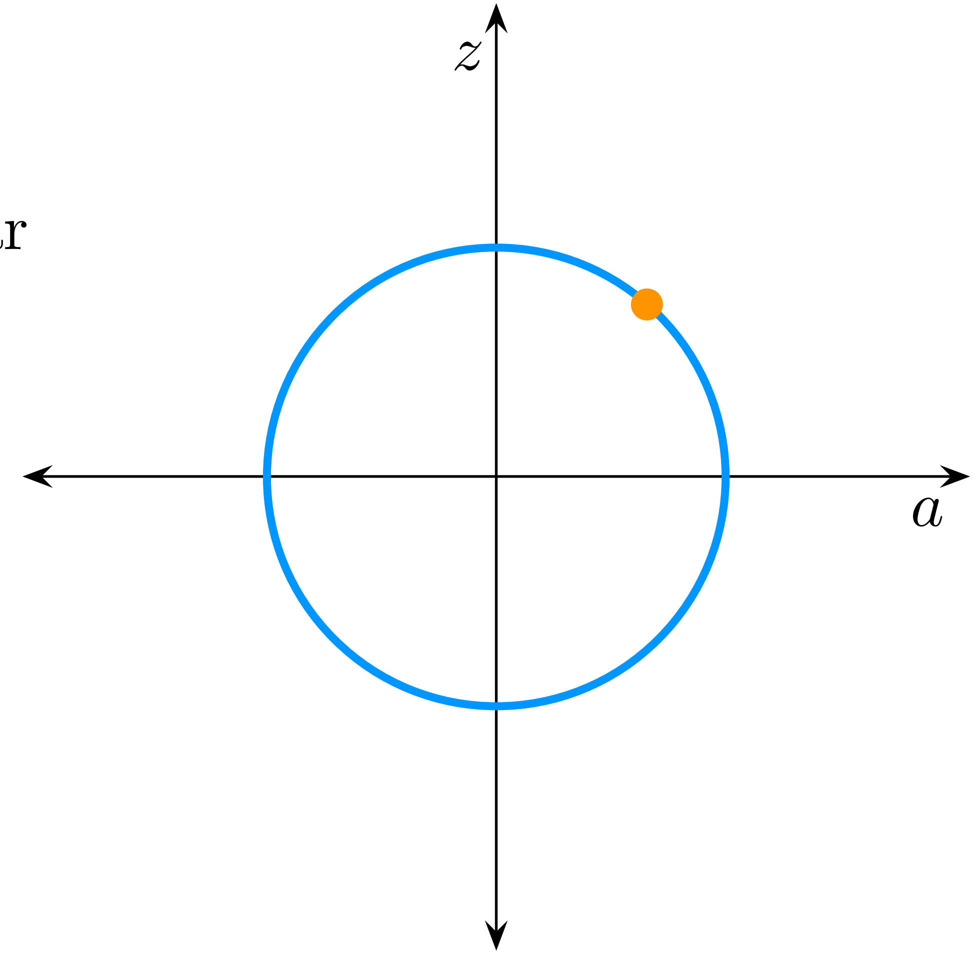
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$f(a, z) = a^2 + z^2 - 1 = 0$



The implicit function theorem

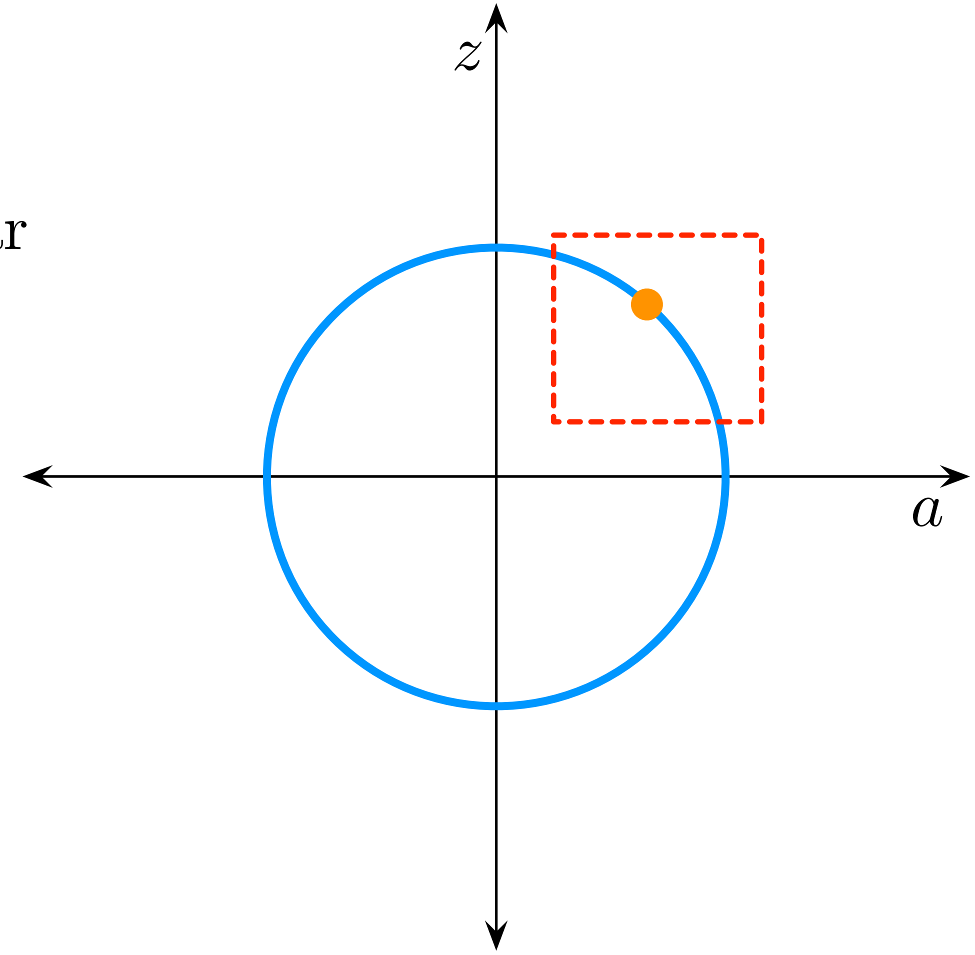
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$f(a, z) = a^2 + z^2 - 1 = 0$



The implicit function theorem

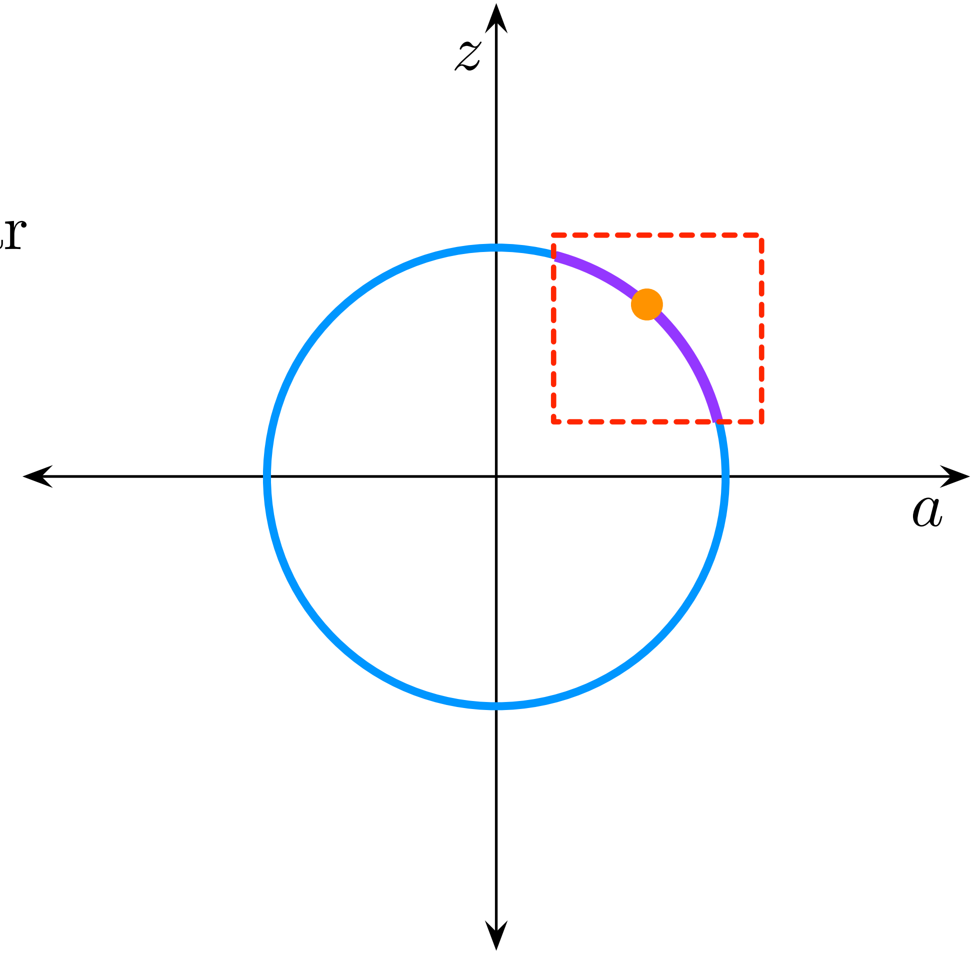
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$$f(a, z) = a^2 + z^2 - 1 = 0$$



The implicit function theorem

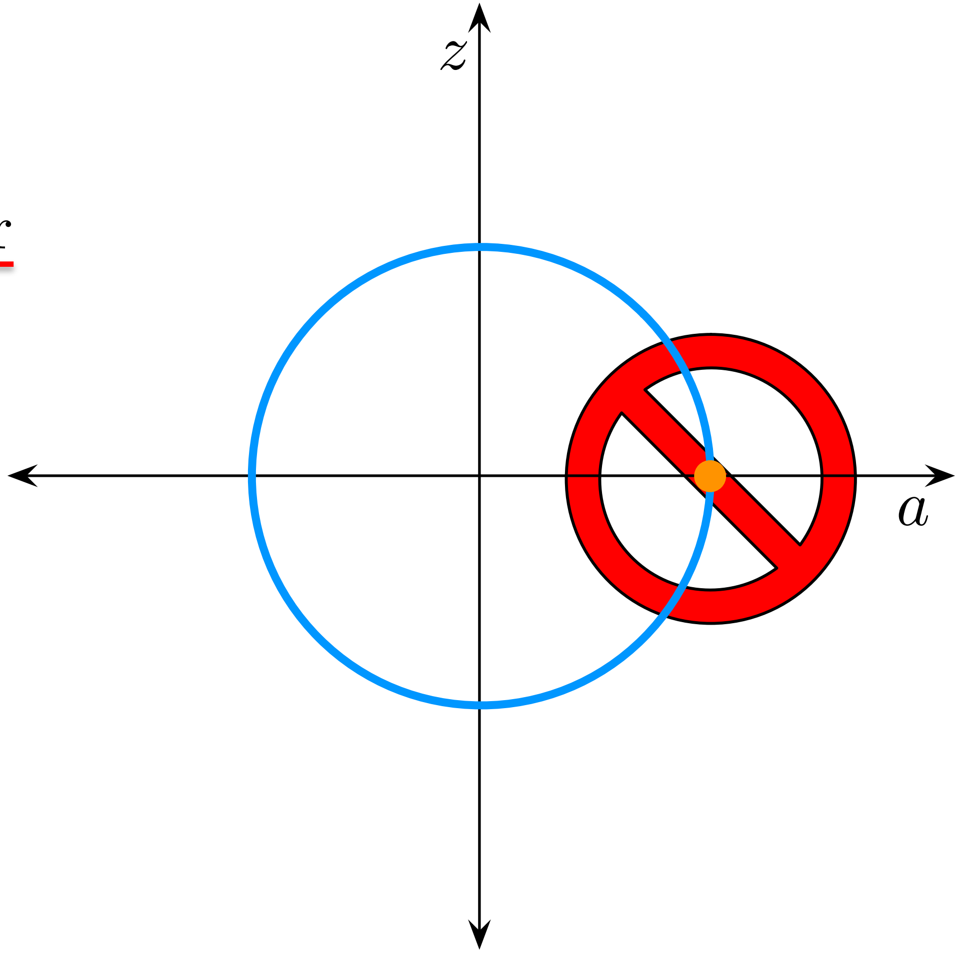
Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $a_0 \in \mathbb{R}^p,\ z_0 \in \mathbb{R}^n$ be such that

1. $f(a_0, z_0) = 0$, and
2. f is continuously differentiable with non-singular Jacobian $\partial_1 f(a_0, z_0) \in \mathbb{R}^{n \times n}$.

Then there exist open sets $S_{a_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing a_0 and z_0, respectively, and a unique continuous function $z^* : S_{a_0} \to S_{z_0}$ such that

1. $z_0 = z^*(a_0)$,
2. $f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$, and
3. z^* is differentiable on S_{a_0}.

$f(a, z) = a^2 + z^2 - 1 = 0$



The implicit function theorem

4.1 Ordinary Differential Equations

There is a strong connection between the implicit function theorem and the theory of differential equations. This is true even from the historical point of view, for Picard's iterative proof of the existence theorem for ordinary differential equations inspired Goursat to give an iterative proof of the implicit function theorem (see Goursat [Go 03]). In the mid-twentieth century, John Nash pioneered the use of a sophisticated form of the implicit function theorem in the study of partial differential equations. We will discuss Nash's work in Section 6.4. In this section, we limit our attention to ordinary (rather than partial) differential equations because the technical details are then so much simpler. Our plan is first to show how a theorem on the existence of solutions to ordinary differential equations can be used to prove the implicit function theorem. Then we will go the other way by using a form of the implicit function theorem to prove an existence theorem for differential equations.

The Implicit Function Theorem: History, Theory, and Applications. Krantz and Parks (2002)

The implicit function theorem: derivative expression

$$f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$$

The implicit function theorem: derivative expression

$$f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$$

$$\partial_0 f(a, z^*(a)) + \partial_1 f(a, z^*(a)) \partial z^*(a) = 0 \quad \forall a \in S_{a_0}$$

$$\partial_0 f(a_0, z_0) + \partial_1 f(a_0, z_0) \partial z^*(a_0) = 0$$

The implicit function theorem: derivative expression

$$f(a, z^*(a)) = 0 \quad \forall a \in S_{a_0}$$

$$\partial_0 f(a, z^*(a)) + \partial_1 f(a, z^*(a)) \partial z^*(a) = 0 \quad \forall a \in S_{a_0}$$

$$\partial_0 f(a_0, z_0) + \partial_1 f(a_0, z_0) \partial z^*(a_0) = 0$$

$$\partial z^*(a_0) = -[\partial_1 f(a_0, z_0)]^{-1} \partial_0 f(a_0, z_0)$$

Punchline: can express Jacobian matrix of solution mapping z^* in terms of Jacobian matrices of f at solution point (a_0, z_0).

Differentiation of fixed point solution mappings

$$z_0 = f(z_0, a_0)$$

$$z^*(a) = f(z^*(a), a)$$

$$\partial z^*(a_0) = \partial_0 f(z_0, a_0) \partial z^*(a_0) + \partial_1 f(z_0, a_0)$$

Differentiation of fixed point solution mappings

$$z_0 = f(z_0, a_0)$$

$$z^*(a) = f(z^*(a), a)$$

$$\partial z^*(a_0) = \partial_0 f(z_0, a_0) \partial z^*(a_0) + \partial_1 f(z_0, a_0)$$

$$\partial z^*(a_0) = [I - \partial_0 f(z_0, a_0)]^{-1} \partial_1 f(z_0, a_0)$$

Connecting to automatic differentiation

1. Jacobian-vector products: $v \mapsto \partial f(x)\, v$

   JVP / push-forward / forward-mode

   build Jacobian one column at a time

2. vector-Jacobian products: $w \mapsto w^{\mathsf{T}} \partial f(x)$

   VJP / pull-back / reverse-mode

   build Jacobian one row at a time

VJPs for fixed point solution mappings

$$\partial z^*(a_0) = [I - \partial_0 f(z_0, a_0)]^{-1} \partial_1 f(z_0, a_0)$$

$$w^\mathsf{T} \partial z^*(a_0) = w^\mathsf{T} [I - \partial_0 f(z_0, a_0)]^{-1} \partial_1 f(z_0, a_0)$$

$$= \boxed{u^\mathsf{T} \partial_1 f(z_0, a_0)}$$

VJPs!

$$\text{where } u^\mathsf{T} = w^\mathsf{T} + \boxed{u^\mathsf{T} \partial_0 f(z_0, a_0)}$$

Punchline: backward pass solve is a (linear) fixed point in terms of VJPs!

Outline

Background and applications of implicit layers

The mathematics of implicit layers

Deep Equilibrium Models

Neural ODEs

Differentiable optimization

Future directions

Deep Equilibrium Models

The simple recurrent backpropagation cell we used previously was quite limited, in practice we want to find an equilibrium point of a more complex “cell”, and use this as our *entire* model (plus one additional linear layer)

$$z^\star = \sigma(W z^\star + x) \quad \Rightarrow \quad z^\star = f(z^\star, x, \theta)$$

Residual block, Transformer block, LSTM cell, etc ($\theta \equiv$ parameters of layers)

As motivated by the discussion on implicit differentiation, we additionally do not care *how* we solve for the equilibrium point, and can use any non-linear root finding algorithm to do so (and also to solve the backward pass)

[Bai, Kolter, Koltun “Deep Equilibrium Models”, NeurIPS 2019]

How to train your DEQ

Forward pass:

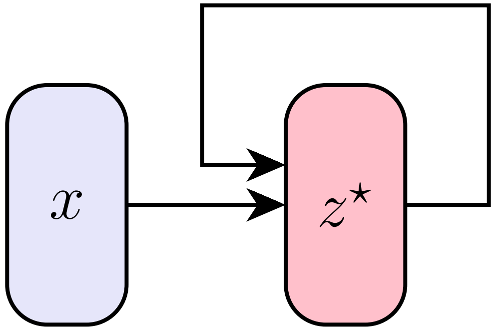
- Given (x, y), compute equilibrium point $z^\star$

$$z^\star = f(z^\star, x, \theta)$$

- Compute loss as some function of $z^\star$, $\ell(z^\star, y)$

Backward pass: Compute gradients using implicit function theorem:

$$\partial \ell(\theta) = \partial_0 \ell(z^\star, y) \underbrace{\left(I - \partial_0 f(z^\star, x, \theta)\right)^{-1} \partial_2 f(z^\star, x, \theta)}_{\text{Implicit differentiation-based solution, solve via indirect method}}$$



$$z^\star = f(z^\star, x, \theta)$$

More details: how to compute the fixed point?

In practice, how do we compute the fixed point $z^\star = f(z^\star, x, \theta)$ (and the linear fixed point for the backward pass)?

Many possible approaches, but one method that works well in practice is *Anderson Acceleration* [Anderson, 1965; Walker and Ni, 2011], a generic method for accelerating fixed point iterations

For the backward (linear) pass, Anderson acceleration is equivalent to the GMRES indirect method

DEQs "One (implicit) layer is all you need"

Theorem 1: A single-layer DEQ can represent any feedforward deep network

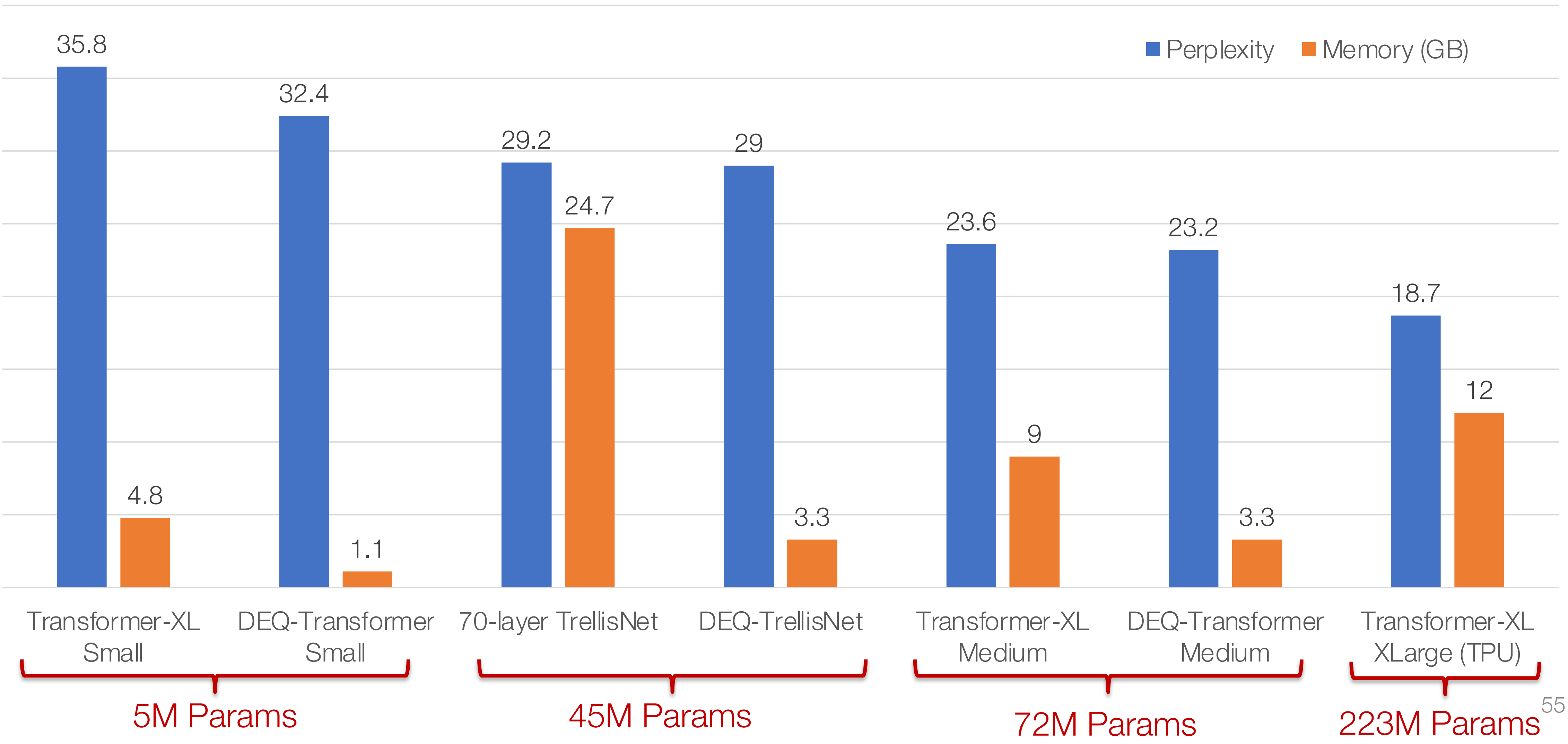
Proof intuition: "Stack" all hidden layers together, and let f be "shifted" application of all layers (**important note:** just for theory, *not* what is done in practice)

Theorem 2: A single-layer DEQ can represent any multi-layer DEQ

Proof intuition: Two equilibrium models can again be represented as a single equilibrium model with layer again "stacked" together

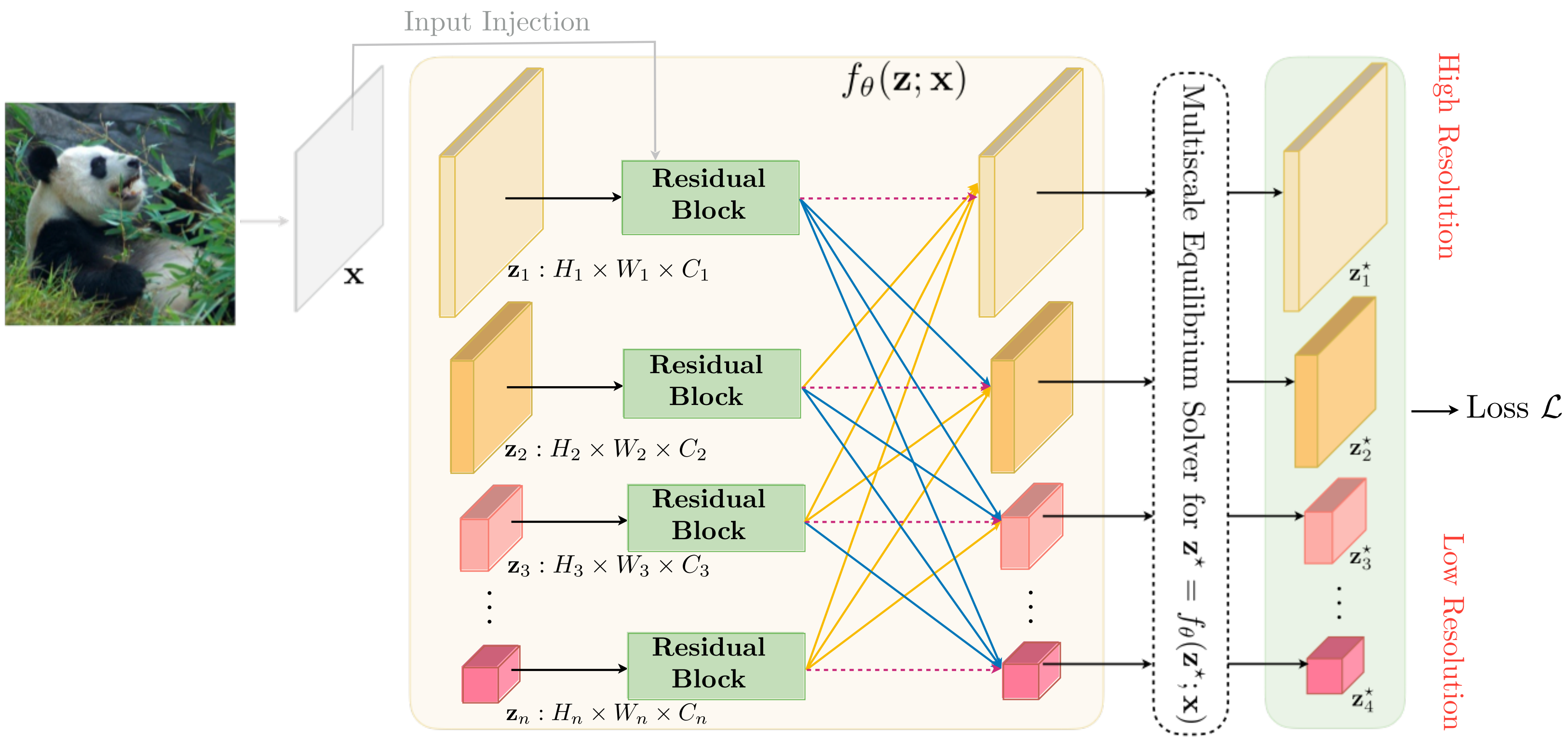
But doesn't address… existence of equilibrium point? uniqueness? stability?

Language modeling: WikiText-103



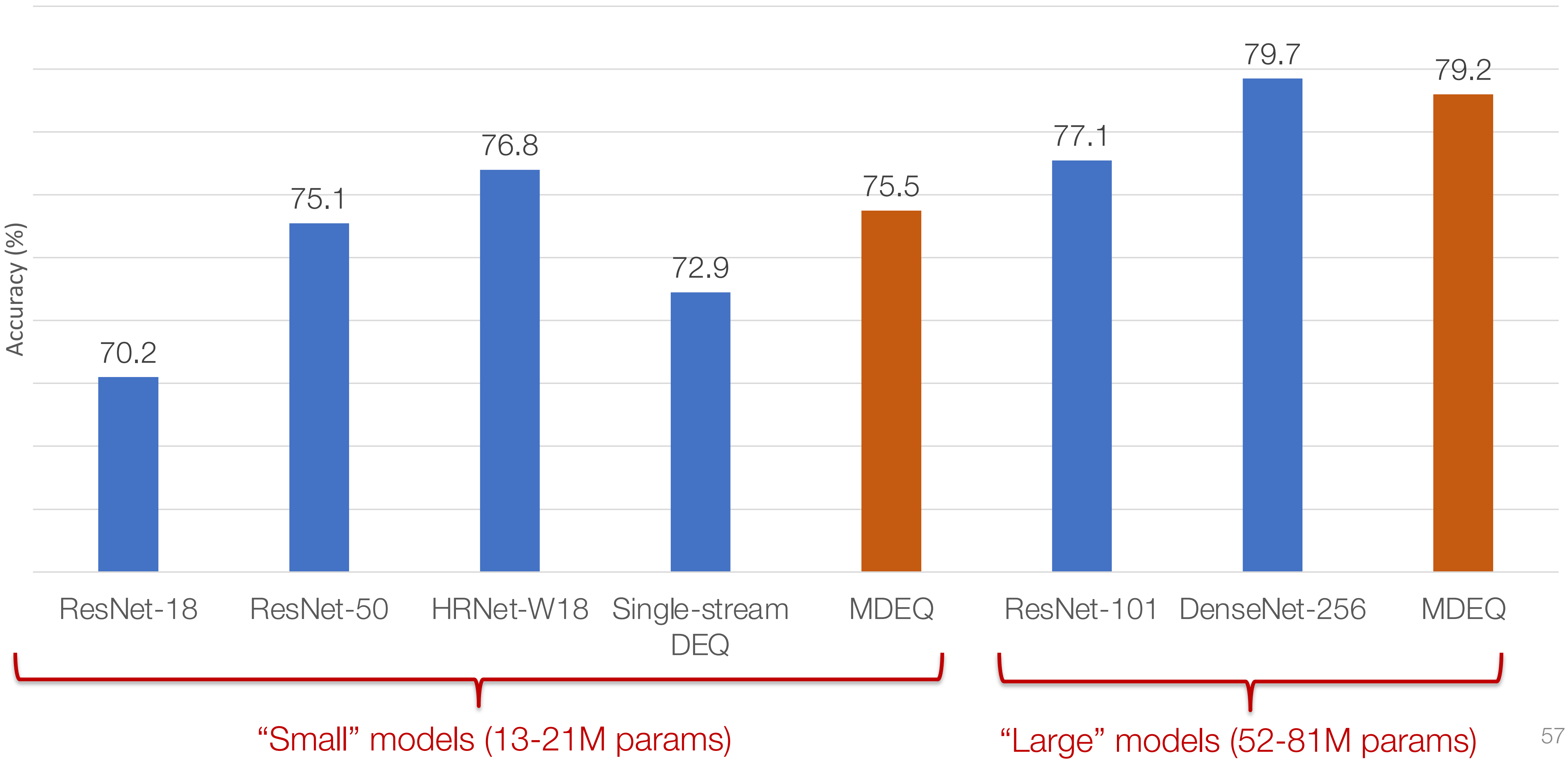
Multiscale deep equilibrium models

Key idea: maintain multiple spatial scales within the hidden unit of a DEQ model, and *simultaneously* find equilibrium point for all of them



[Bai, Koltun, Kolter "Multiscale Deep Equilibrium Models", NeurIPS 2020]

ImageNet Top-1 Accuracy



Citiscapes mIoU

Visualization of Segmentation

Outline

Background and applications of implicit layers

The mathematics of implicit layers

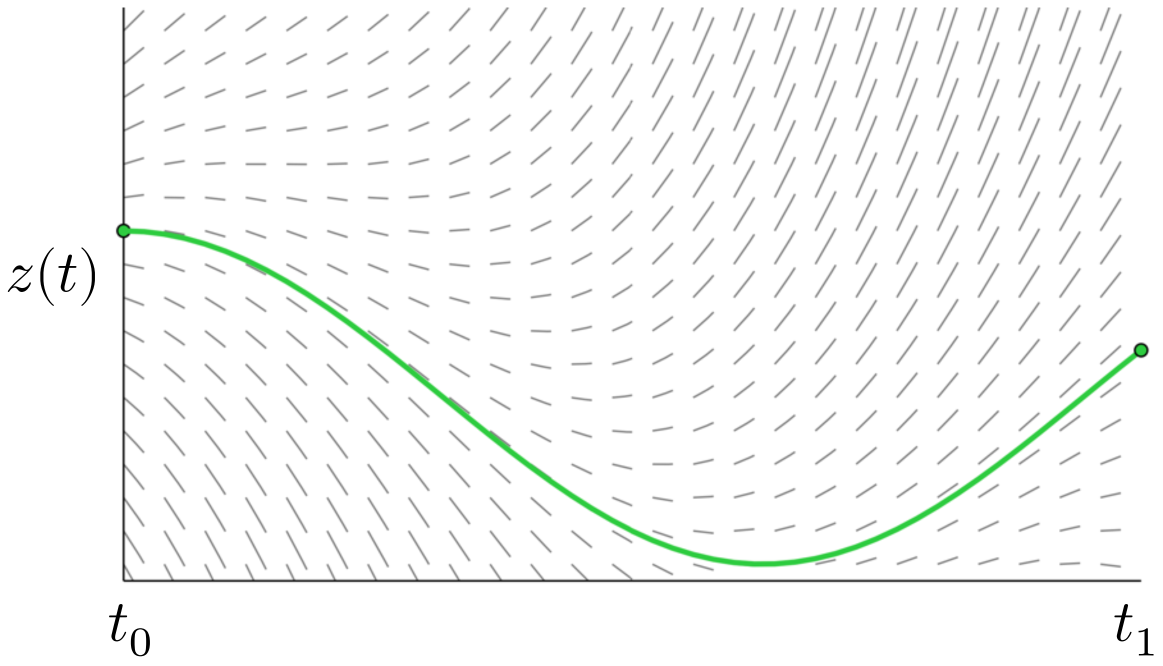
Deep Equilibrium Models

Neural ODEs

Differentiable optimization

Future directions

Ordinary Differential Equations



If a vector z follows dynamics f:

$$\frac{dz}{dt} = f(z(t), t, \theta)$$

Can find $z(t_1)$ by starting at $z(t_0)$ and integrating until time t_1:

$$z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta) dt$$

An implicit layer: $y = \mathrm{odeint}(f, x, t_0, t_1, \theta)$

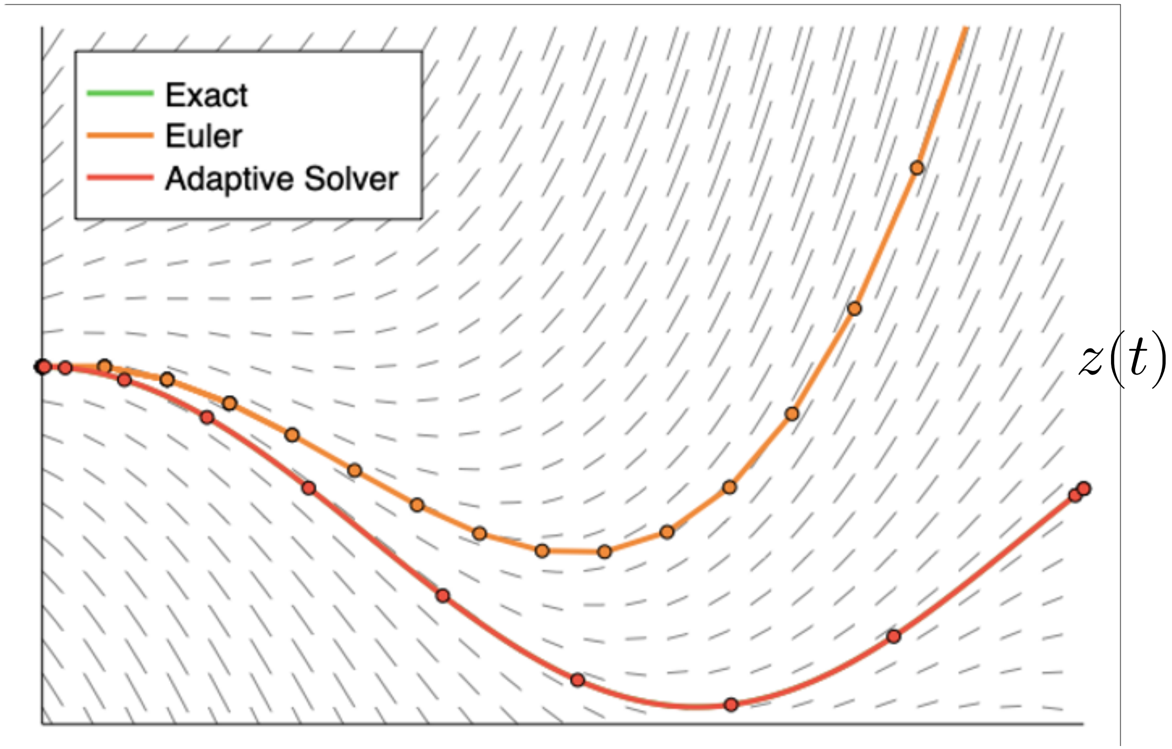
For continuously differentiable and Lipshitz f, gradients always exist. (no `relu`, but `tanh` fine)

How to Solve ODEs?

Simplest way: Euler's method. Take steps of size h in direction of f

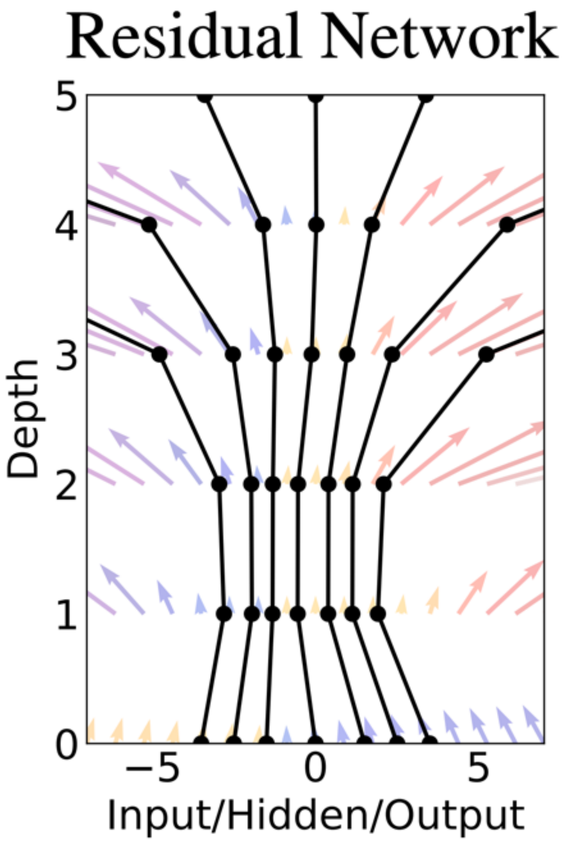
$$z_{i+1} = z_i + hf(z_i, t_i, \theta)$$

Looks just like a residual network!



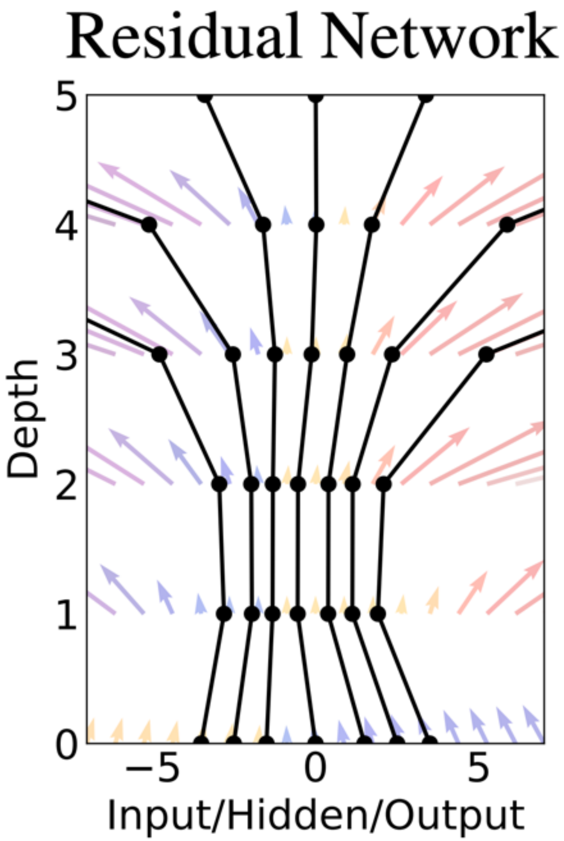
From ResNets to ODE-Nets

```
def f(z, t, θ):
  return nnet(z, θ[t])

def resnet(z, θ):
  for t in [1:T]:
    z = z + f(z, t, θ)
  return z
```



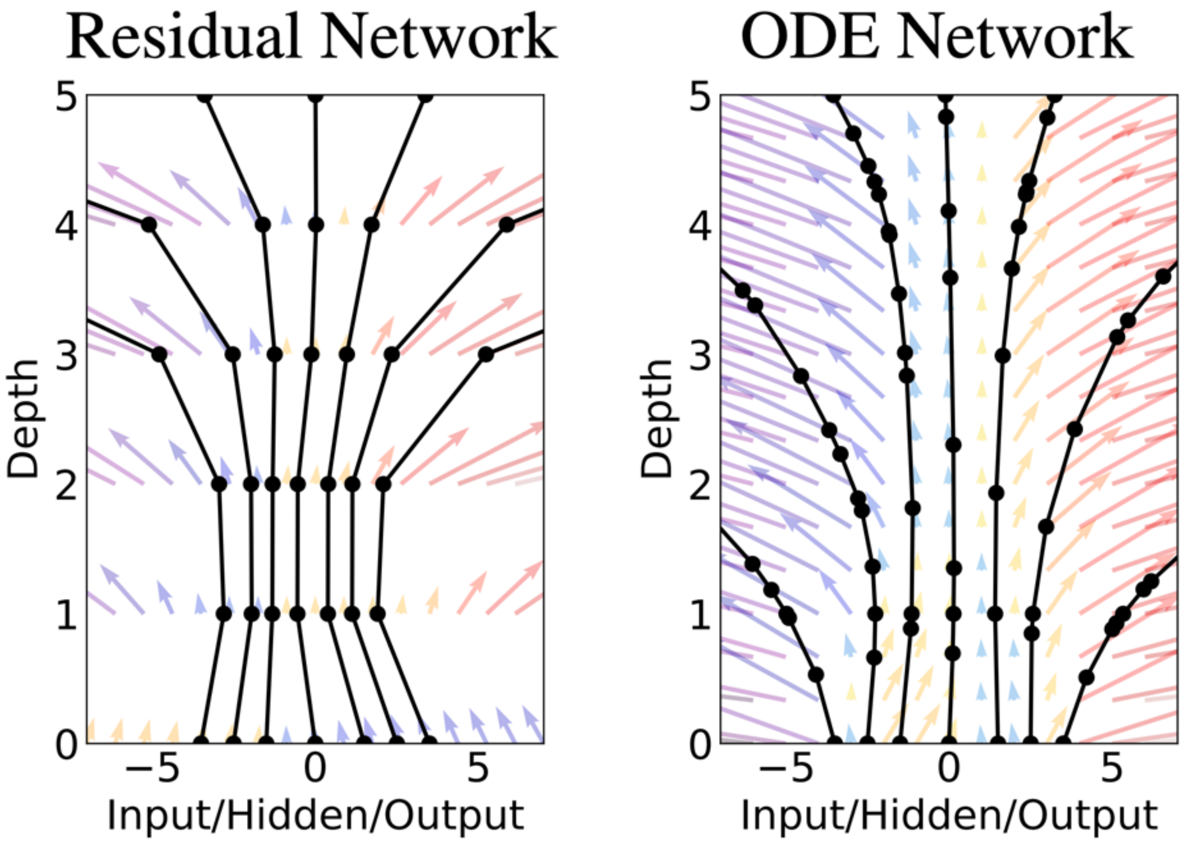
From ResNets to ODE-Nets

```
def f(z, t, θ):
  return nnet([z, t], θ)

def resnet(z, θ):
  for t in [1:T]:
    z = z + f(z, t, θ)
  return z
```



From ResNets to ODE-Nets

```
def f(z, t, θ):
    return nnet([z, t], θ)

def ODEnet(z, θ):
    return ODESolve(f, z, 0, 1, θ)
```

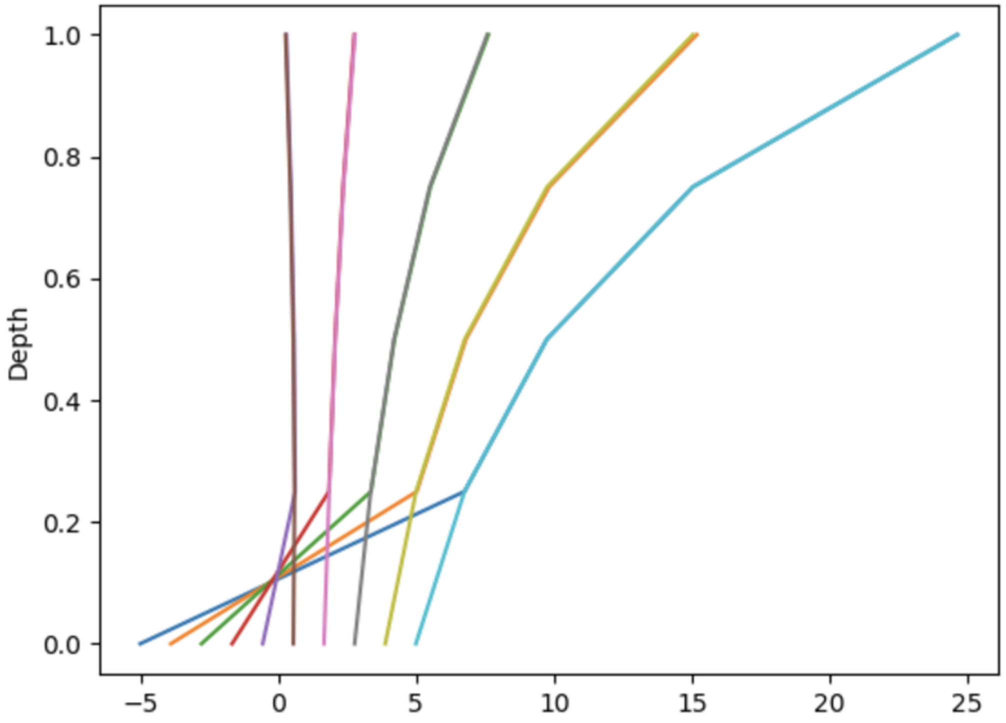


Residual Networks vs ODE solutions

Example: Fit $y = x^2$

ResNet can learn non-bijective transformations.

Output y



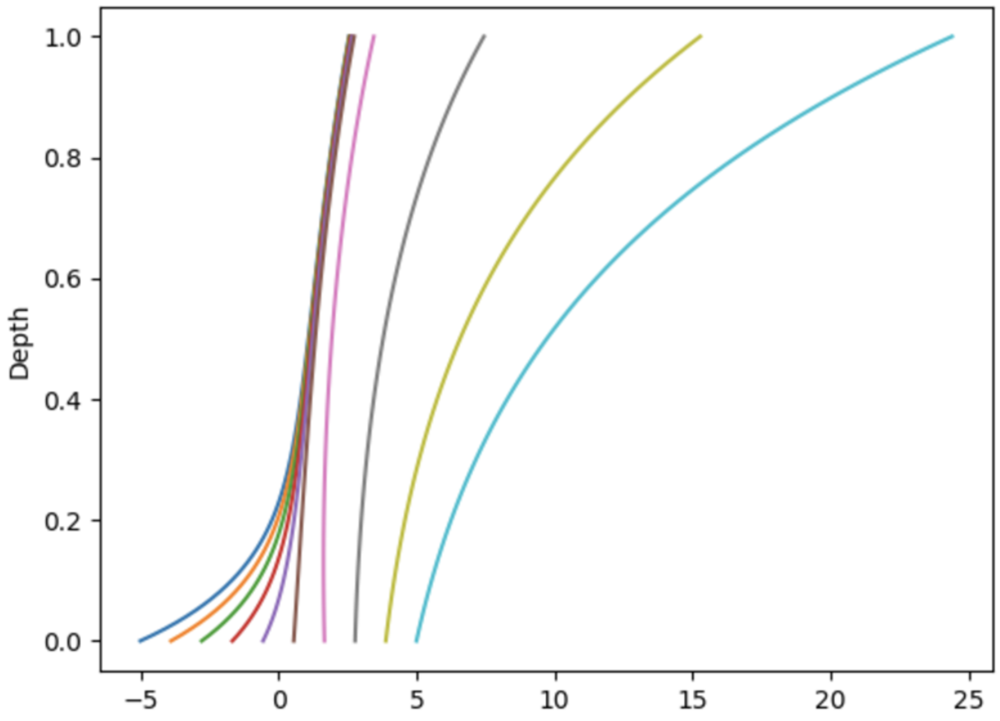
Input x

Residual Networks vs ODE solutions

Example: Fit $y = x^2$

Ode-net can only learn bijective transformations.

Output y

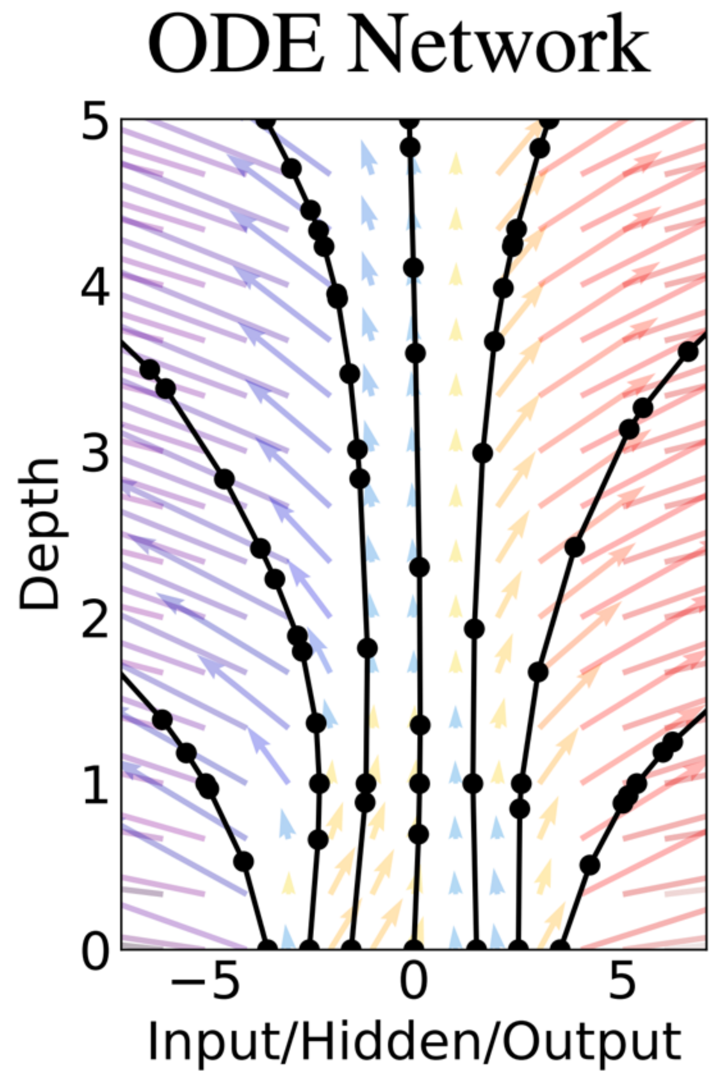


Input x

Adaptive ODE Solvers

Adaptive solvers:

- Usually fit a local polynomial to dynamics
- Try to estimate extrapolation error
- Need fewer evaluations of dynamics function f when dynamics are simple / well-approximated
- Can adjust tolerance / precision of solver at any time

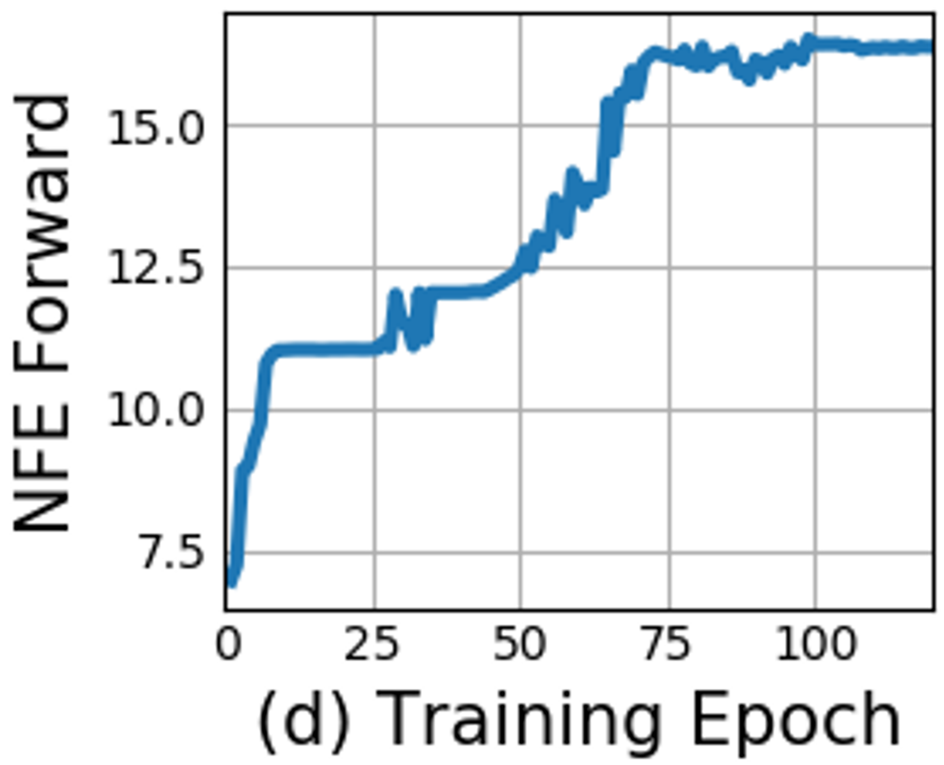


Dynamics Become Increasingly Complex in Training

Dynamics become more demanding to compute during training.

Adapts computation time according to complexity of dynamics.

Also happens in DEQs



How to train an ODE net?

Can backprop through solver operations, but high memory cost.

$$L(\theta) = L\left(\int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right)$$

$$\frac{\partial L}{\partial \theta} = ?$$

Continuous-time Backpropagation

Standard Backprop:

$$\frac{\partial L}{\partial \mathbf{z}_t} = \frac{\partial L}{\partial \mathbf{z}_{t+1}} \frac{\partial f(\mathbf{z}_t, \theta)}{\partial \mathbf{z}_t}$$

$$\frac{\partial L}{\partial \theta} = \sum_t \frac{\partial L}{\partial \mathbf{z}_t} \frac{\partial f(\mathbf{z}_t, \theta)}{\partial \theta}$$

Adjoint sensitivities:
(Pontryagin et al., 1962):

$$\frac{\partial}{\partial t} \frac{\partial L}{\partial \mathbf{z}(t)} = \frac{\partial L}{\partial \mathbf{z}(t)} \frac{\partial f(\mathbf{z}(t), \theta)}{\partial \mathbf{z}}$$

$$\frac{\partial L}{\partial \theta} = \int_{t_1}^{t_0} \frac{\partial L}{\partial \mathbf{z}(t)} \frac{\partial f(\mathbf{z}(t), \theta)}{\partial \theta} dt$$

Continuous-time Backpropagation

Can build adjoint dynamics with autodiff, compute all gradients with another ODE solve:

```
def f_and_a([z, a, d], t):
  return [f, -a*df/da, -a*df/dθ)

[z0, dL/dx, dL/dθ] =
   ODESolve(f_and_a,
[z(t1), dL/dz(t), 0], t1, t0)
```

Adjoint sensitivities:
(Pontryagin et al., 1962):

$$\frac{\partial}{\partial t}\frac{\partial L}{\partial \mathbf{z}(t)} = \frac{\partial L}{\partial \mathbf{z}(t)}\frac{\partial f(\mathbf{z}(t),\theta)}{\partial \mathbf{z}}$$

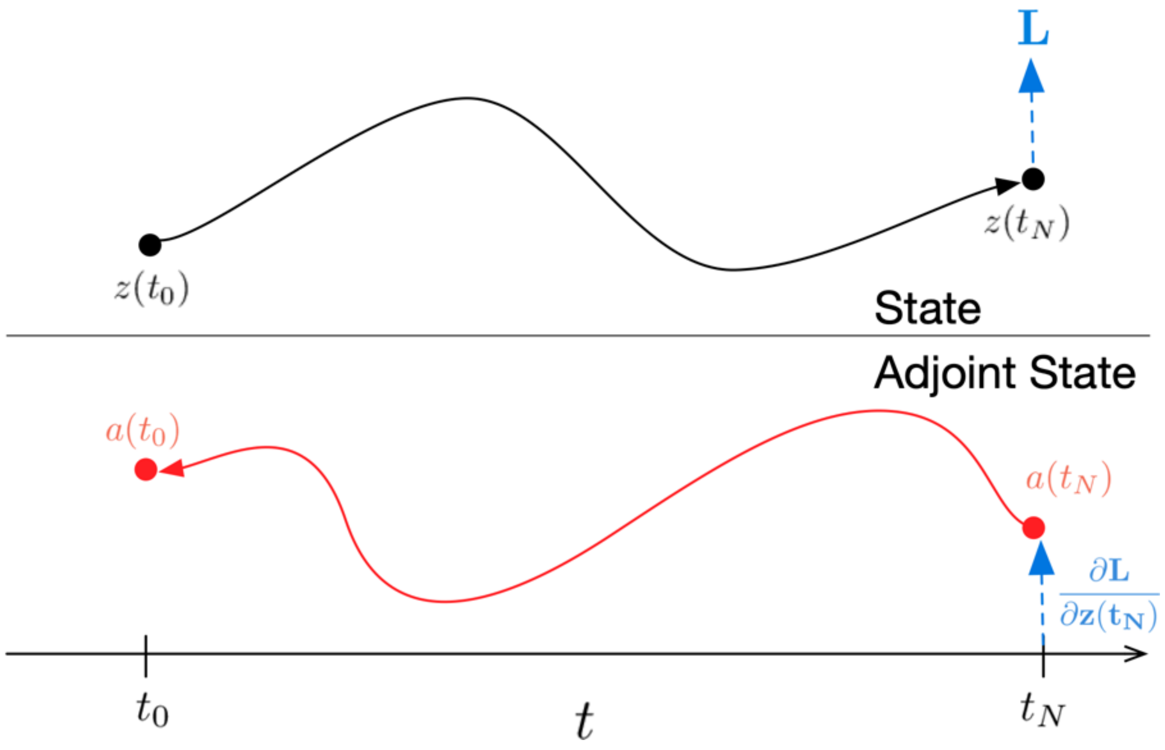
$$\frac{\partial L}{\partial \theta} = \int_{t_1}^{t_0}\frac{\partial L}{\partial \mathbf{z}(t)}\frac{\partial f(\mathbf{z}(t),\theta)}{\partial \theta}dt$$

O(1) Memory Gradients

No need to store activations, just run dynamics backwards from output.

Can do similar trick with Reversible ResNets (Gomez et al., 2018), but must restrict architecture.

This introduces extra numerical error. if mismatch is detected, can use checkpointing to force a better match.



Deep Equilibrium Models vs Neural ODEs

Both have:

- Constant memory training
- Adjustable compute vs precision at test time
- Infinite / adjustable depth

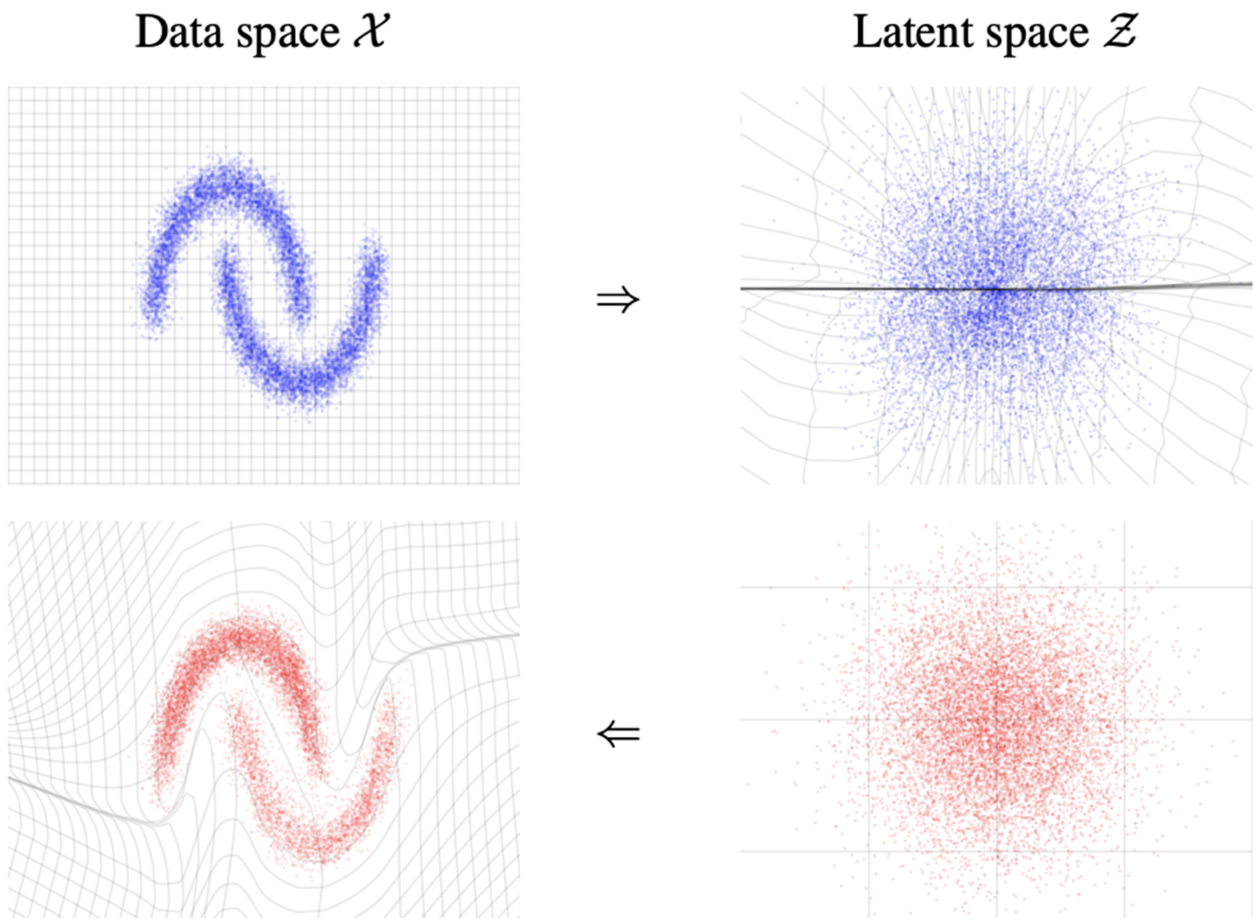
Use neural ODEs when:

- You care about the trajectory (continuous time series, physics)
- Building normalizing flows (easier change of variable sometimes)

Normalizing Flows

Tractable probabilistic models based on change of variables

Requires an invertible transformation



Density Estimation using Real NVP. Dinh, Sohl-Dickstein, Bengio (2017)

Continuous(-time) Normalizing Flows

Change of variables theorem:

$$x_1 = F(x_0) \implies p(x_1) = p(x_0) \left| \det \frac{\partial F}{\partial x_0} \right|^{-1}$$

Determinant is $O(D^3)$ cost

Must design architectures to have structured Jacobian

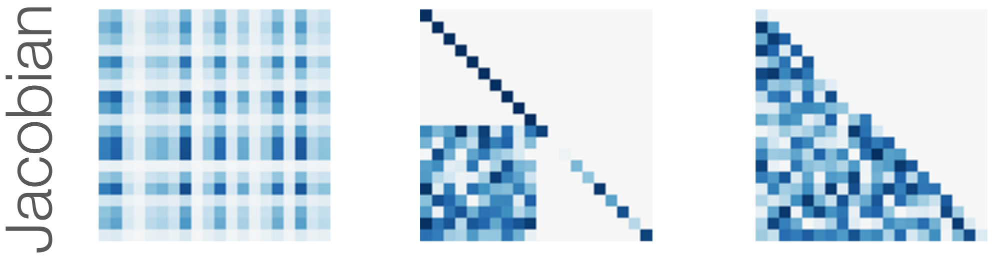
Continuous(-time) Normalizing Flows

Change of variables theorem:

$$x_1 = F(x_0) \implies p(x_1) = p(x_0) \left| \det \frac{\partial F}{\partial x_0} \right|^{-1}$$

Determinant is $O(D^3)$ cost

Must design architectures to have structured Jacobian



(Low rank) (Sparse) (Lower triangular)

Continuous(-time) Normalizing Flows

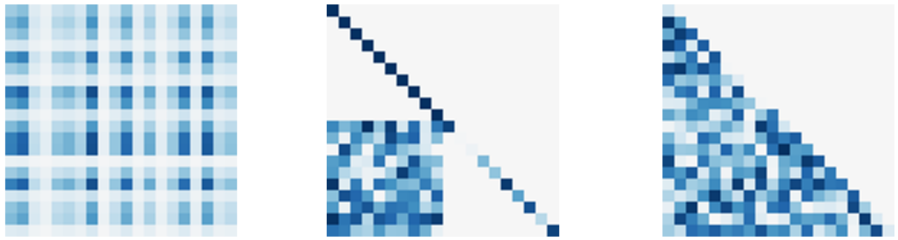
Change of variables theorem:

$$x_1 = F(x_0) \implies p(x_1) = p(x_0) \left| \det \frac{\partial F}{\partial x_0} \right|^{-1}$$

Determinant is $O(D^3)$ cost

Must design architectures to have structured Jacobian

Jacobian



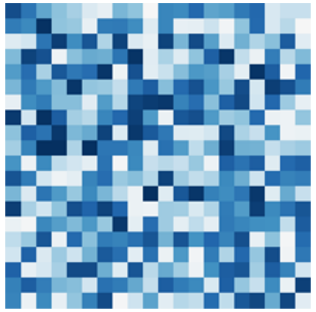
(Low rank) (Sparse) (Lower triangular)

<u>Instantaneous</u> change of variables:

$$\frac{dx}{dt} = f(x(t), t) \implies \frac{\partial \log p(x(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial x(t)} \right)$$

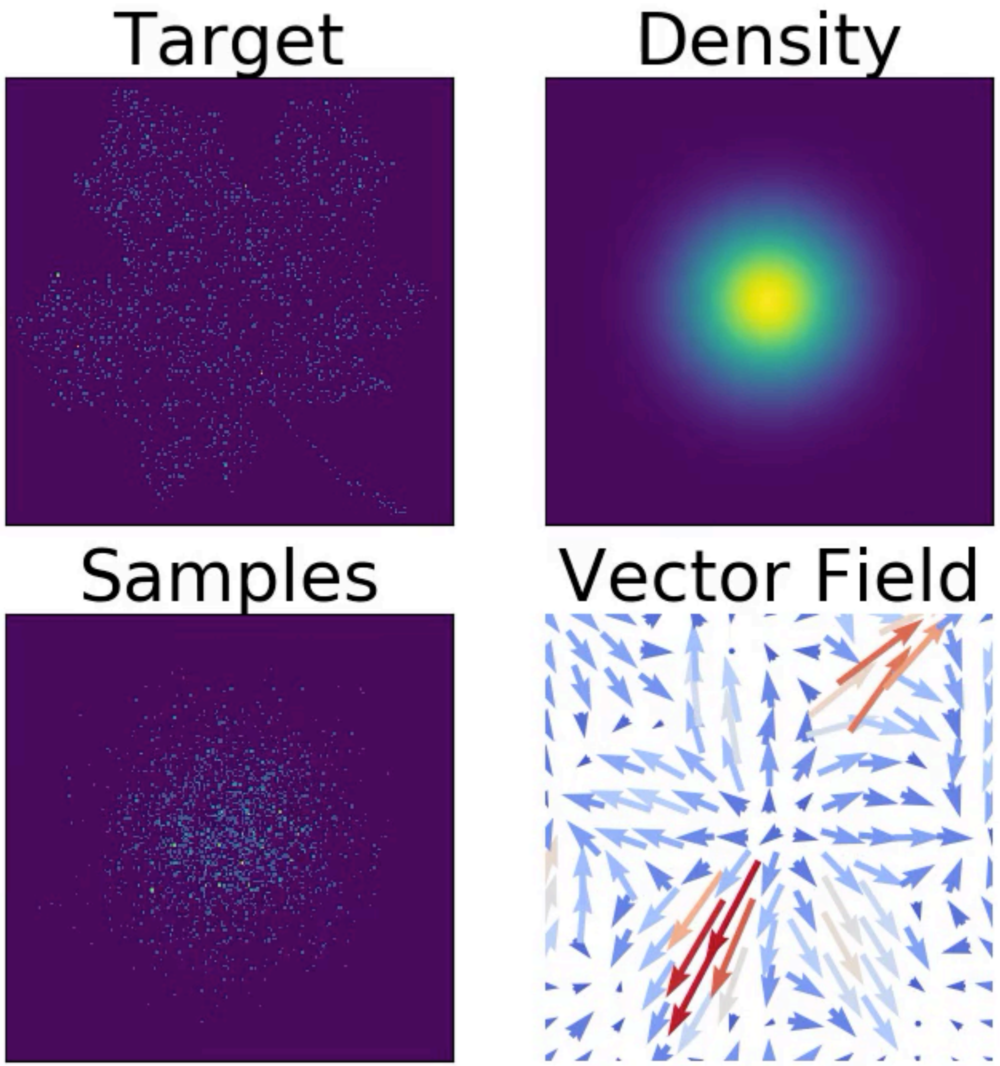
Trace is always $O(D)$ cost.

Trace allows flows at **linear cost**.



(Arbitrary)

Continuous(-time) Normalizing Flows

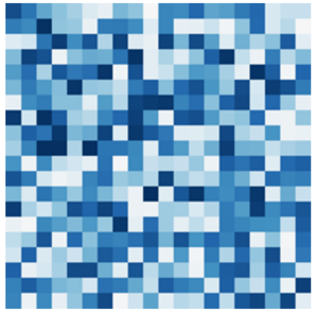


<u>Instantaneous</u> change of variables:

$$\frac{dx}{dt} = f(x(t), t) \implies \frac{\partial \log p(x(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial x(t)}\right)$$

Trace is always $O(D)$ cost.

Trace allows flows at **linear cost**.



(Arbitrary)

Stochastic Estimation for CNFs

Divergence of a neural network can be be computationally expensive

$$\log p(x) = \log p(z) + \int_0^T \operatorname{div} f \; dt$$
$$= \log p(z) + \int_0^T \operatorname{tr}(J_f) \; dt$$

trace of Jacobian is expensive

Stochastic Estimation for CNFs

Divergence of a neural network can be be computationally expensive

$$\begin{aligned}\log p(x) &= \log p(z) + \int_0^T \operatorname{div} f \; dt \\ &= \log p(z) + \int_0^T \operatorname{tr}(J_f)\; dt\end{aligned}$$

trace of Jacobian is expensive

$$\operatorname{tr}(A) = E_{v \sim \mathcal{N}(0,1)}[v^T A v]$$

(Hutchinson's trace estimator)

Stochastic Estimation for CNFs

Divergence of a neural network can be be computationally expensive

$$\begin{aligned}\log p(x) &= \log p(z) + \int_0^T \operatorname{div} f \, dt \\ &= \log p(z) + \int_0^T \operatorname{tr}(J_f) \, dt \\ &= \log p(z) + \mathbb{E}_{v \sim \mathcal{N}(0,1)} \left[\int_0^T v^T J_f v \, dt \right]\end{aligned}$$

vector-Jacobian products are cheap

$$\operatorname{tr}(A) = E_{v \sim \mathcal{N}(0,1)} [v^T A v]$$

(Hutchinson's trace estimator)

Stochastic Estimation for CNFs

Divergence of a neural network can be be computationally expensive

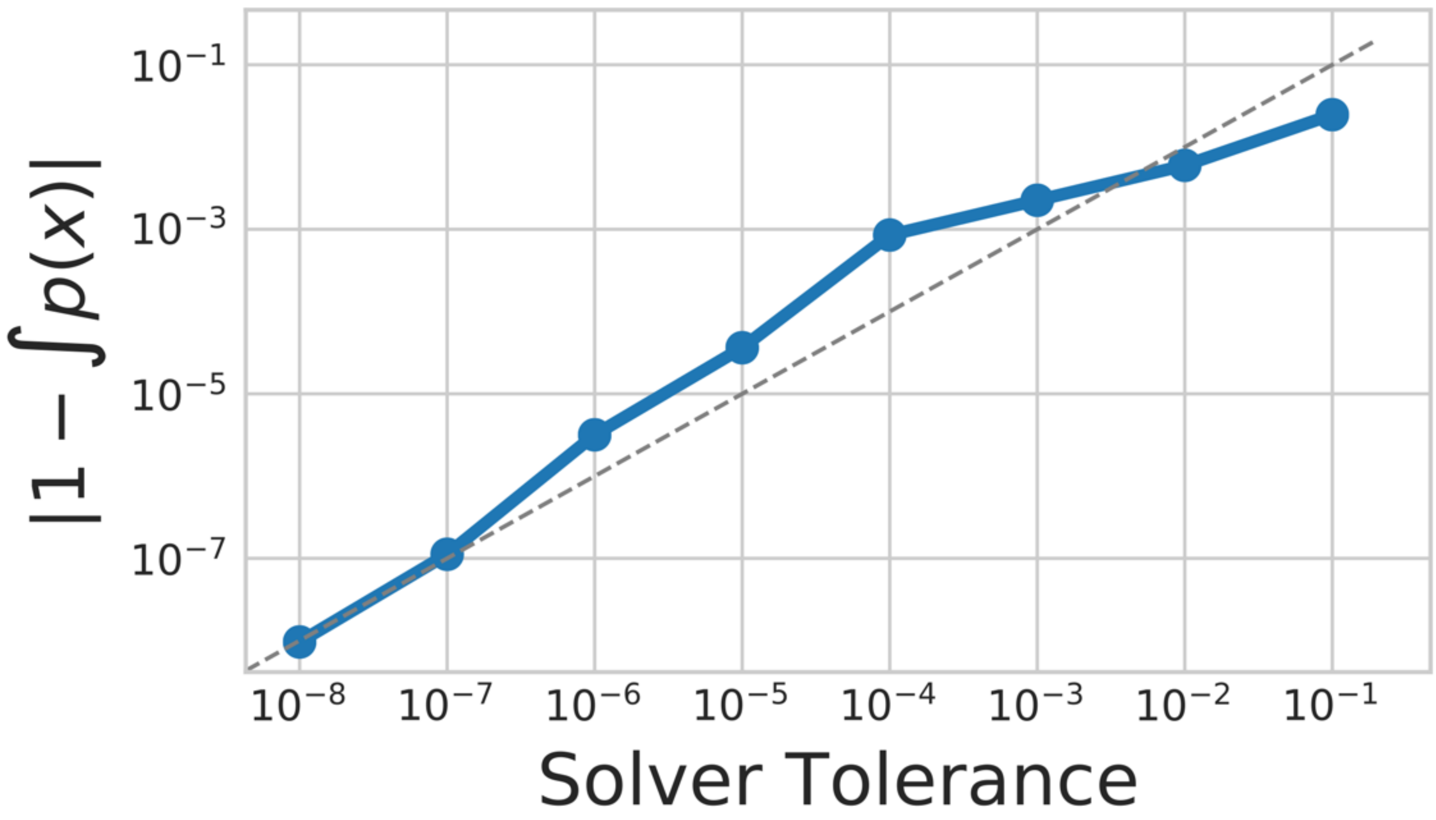
$$\begin{aligned}\log p(x) &= \log p(z) + \int_0^T \operatorname{div} f \; dt \\ &= \log p(z) + \int_0^T \operatorname{tr}(J_f) \; dt \\ &= \log p(z) + \mathbb{E}_{v \sim \mathcal{N}(0,1)} \left[\int_0^T v^T J_f v \; dt \right]\end{aligned}$$

vector-Jacobian products are cheap

What about numerical error?

Is density accurate?

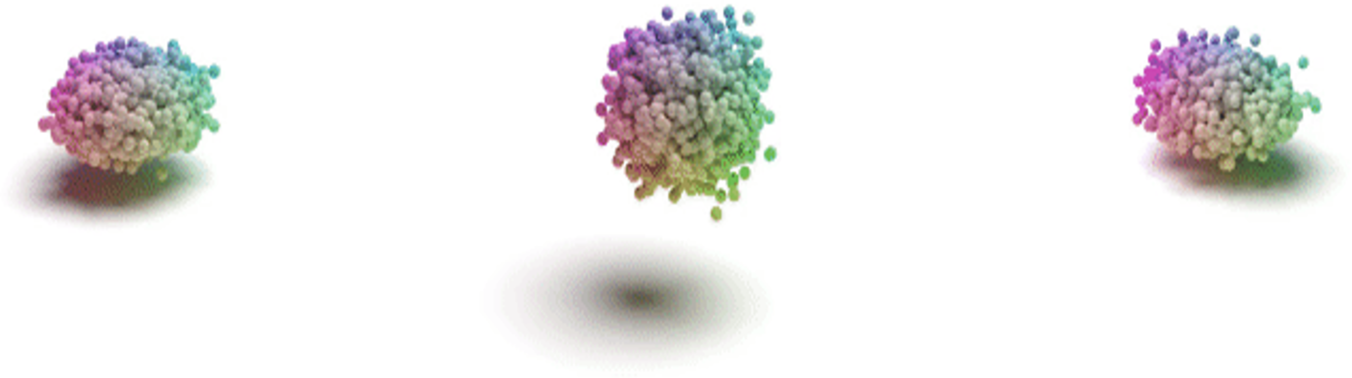
Only up to solver prevision, but can choose precision at test time.



FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models
Grathwohl, Chen, Bettencourt, Sutskever, Duvenaud

Continuous Normalizing Flows

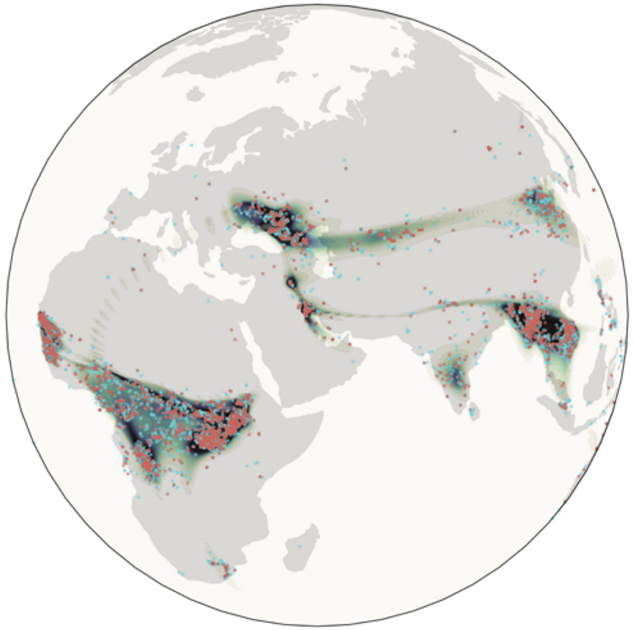
Can also parameterize homeomorphisms
(non self-intersecting maps)



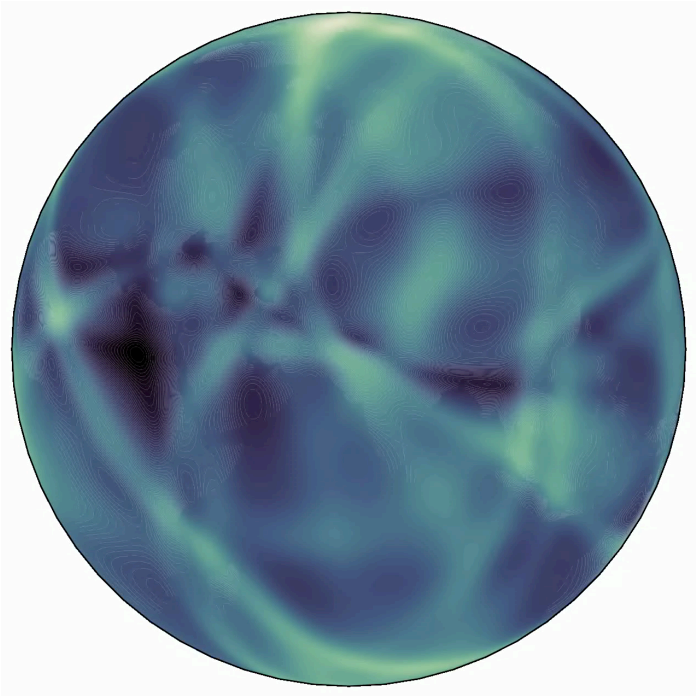
PointFlow: 3D Point Cloud Generation with Continuous Normalizing Flows.
Yang, Huang, Hao, Liu, Belongie, Hariharan (2019).

Continuous Normalizing Flows

Can build flexible parametric density models on manifolds (e.g. spheres)



Fire

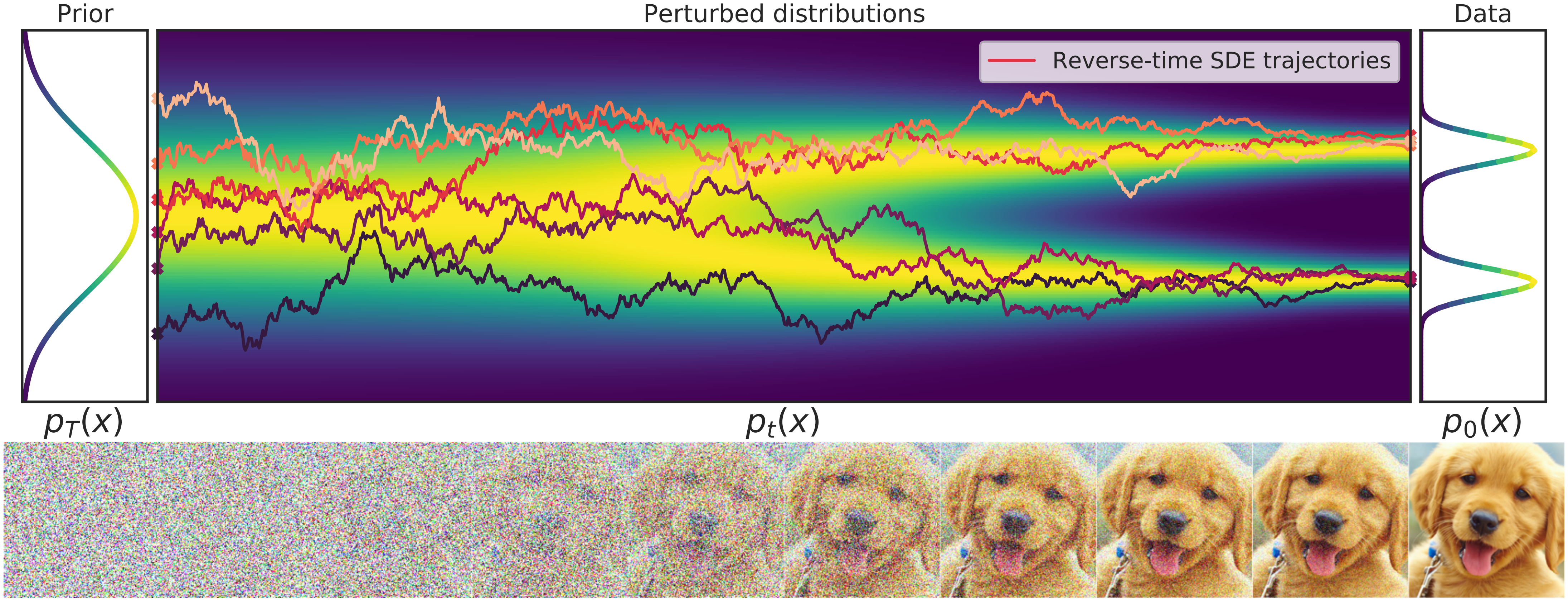


Thanks to Emile Mathieu

Riemannian Continuous Normalizing Flows [Mathieu and Nickel, 2020]
Neural Ordinary Differential Equations on Manifolds. [Falorsi and Forré, 2020]
Neural Manifold Ordinary Differential Equations. [Lou et al., 2020]

Slide credit: Yang Song

Score-based generative modeling via SDEs



$$\mathrm{d}\mathbf{x} = \sigma(t)\mathrm{d}\mathbf{w} \quad \xrightarrow{\text{Time reversal}} \quad \mathrm{d}\mathbf{x} = -\sigma^2(t)\nabla_{\mathbf{x}} \log p_t(\mathbf{x})\mathrm{d}t + \sigma(t)\mathrm{d}\bar{\mathbf{w}}$$

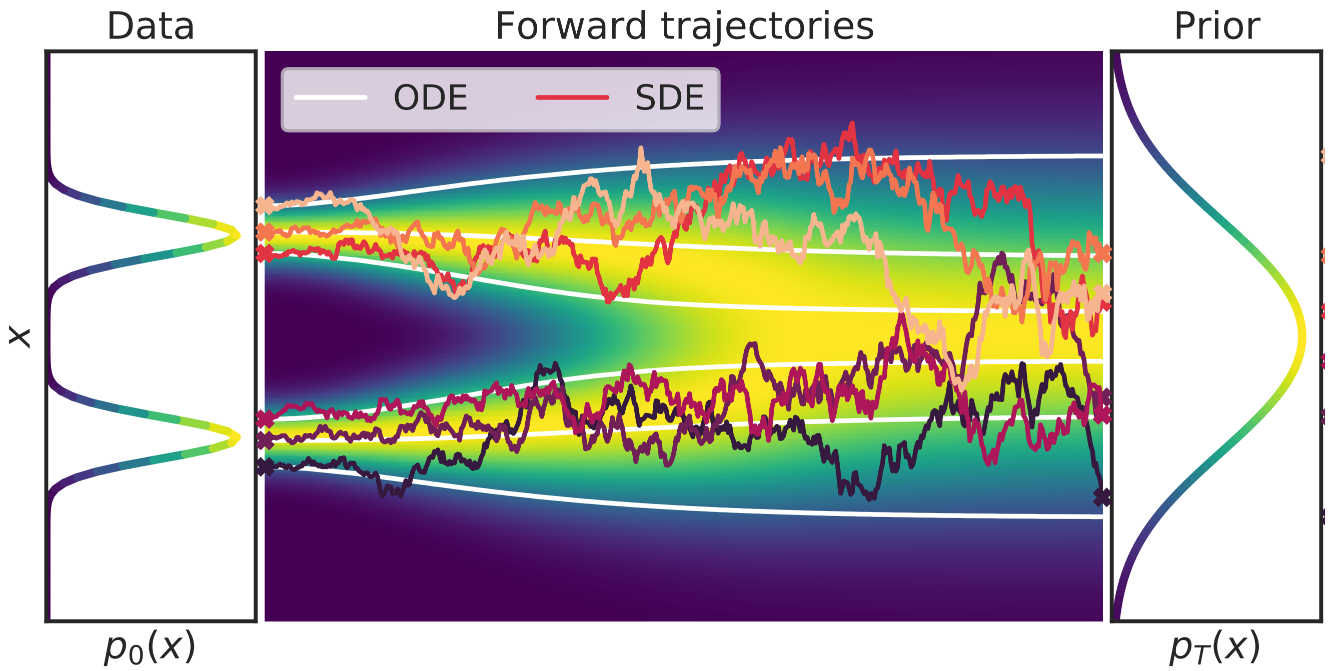
Score-Based Generative Modeling through Stochastic Differential Equations. Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. (2020)

Slide credit: Yang Song

Turning a reverse diffusion SDE into ODE

Probability flow ODE (ordinary differential equation)

$$\mathrm{d}\mathbf{x} = \sigma(t)\mathrm{d}\mathbf{w} \underset{\{p_t(\mathbf{x})\}_{t=0}^{T}}{=} \mathrm{d}\mathbf{x} = -\frac{1}{2}\sigma(t)^2 \nabla_{\mathbf{x}} \log p_t(\mathbf{x}) \mathrm{d}t$$



Score-Based Generative Modeling through Stochastic Differential Equations. Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. (2020)

Score-based Continuous Normalizing Flows

Score-based training scales to 1024x1024

Exact density available, but expensive

[Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. Score-Based Generative Modeling through Stochastic Differential Equations, 2020]

Score-based Continuous Normalizing Flows

Conditional inpainting and colorization without retraining.

Requires iterative sampling procedure.

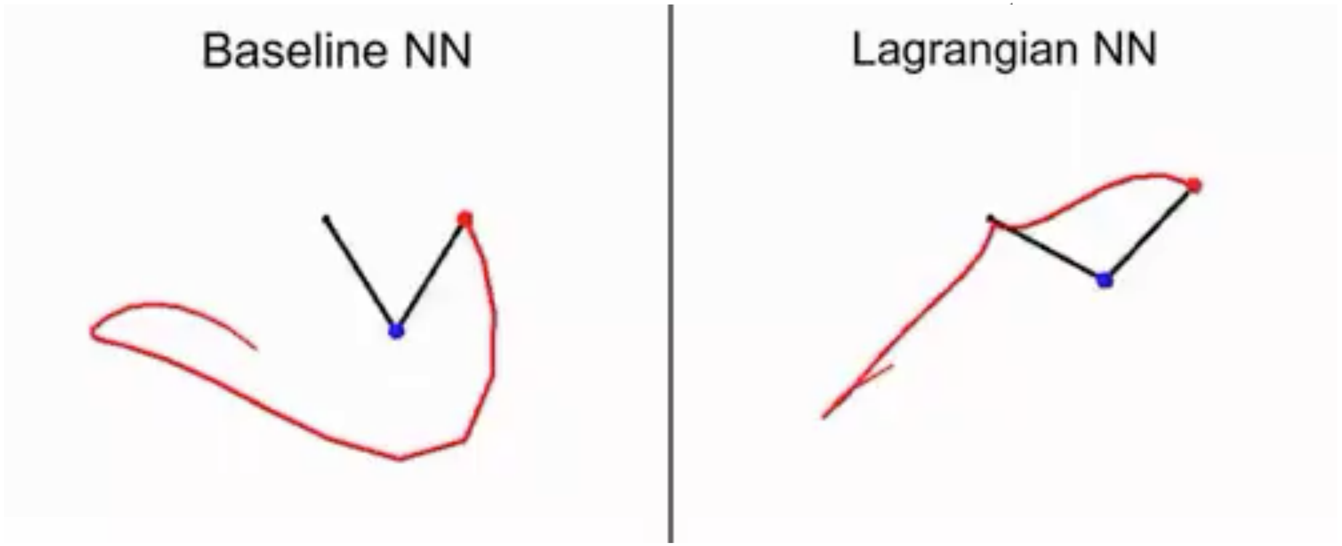
Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. Score-Based Generative Modeling through Stochastic Differential Equations. 2020

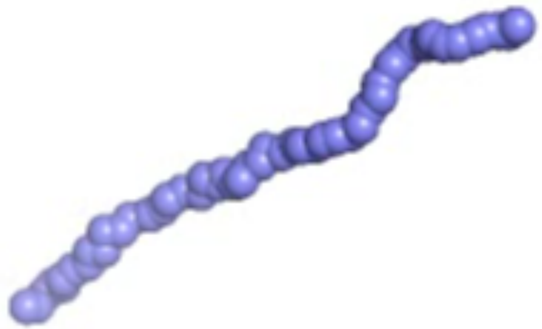
Neural ODEs for Time Series

Continuous-time Physical Models

Incorporate known structure or constraints, e.g. Hamiltonians, Lagrangians

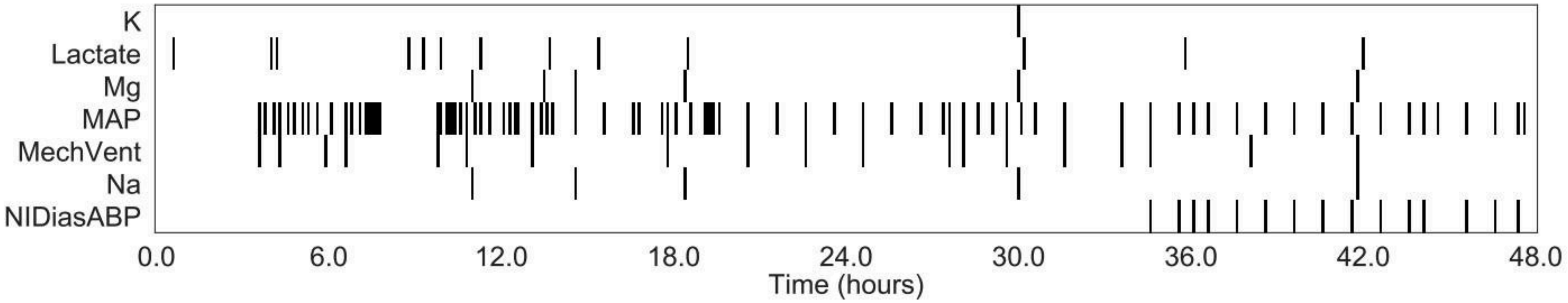
$$\ddot{q} = \left(\frac{\partial^2 \mathcal{L}}{\partial \dot{q}^2}\right)^{-1} \left(\frac{\partial \mathcal{L}}{\partial q} - \dot{q}\frac{\partial^2 \mathcal{L}}{\partial q \partial \dot{q}}\right)$$





Hamiltonian Graph Networks with ODE Integrators. Sanchez-Gonzalez, Bapst, Cranmer, Battaglia (2019)
Lagrangian Neural Networks. Cranmer, Greydanus, SHoyer, Battaglia, Spergel, Ho, (2020)
Differentiable Molecular Simulations for Control and Learning. Wang, Axelrod, Gómez-Bombarelli (2020)
Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control. Zhong, Dey, Chakraborty (2020)

Irregularly-timed datasets



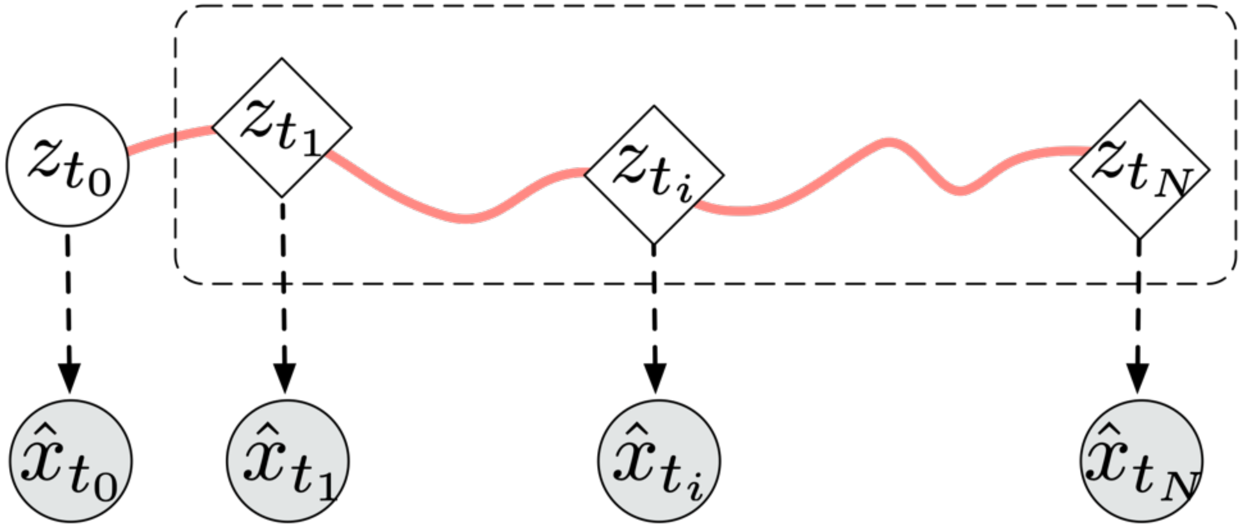
Most patient data, business data irregularly sampled through time.

Most large parametric models in ML are discrete time: RNNs, HMMs, DKFs

How to handle these data without binning?

Continuous-time Time Series Models

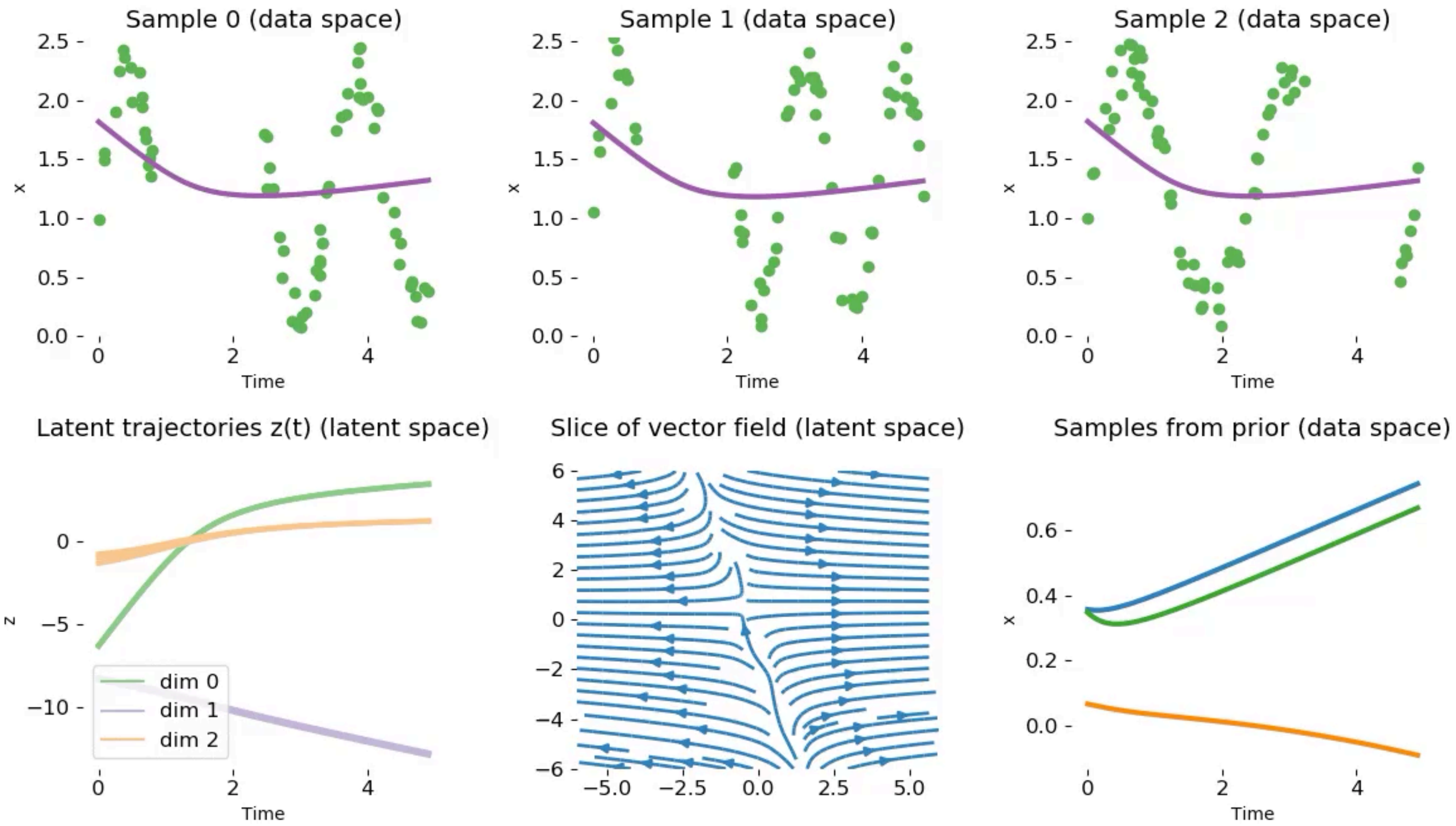
Can deal with data collected at irregular intervals natively.



Can jointly train dynamics, likelihood, and recognition network as a VAE.

Latent ODEs for Irregularly-Sampled Time Series. Rubanova, Chen, Duvenaud (2020)
Neural Controlled Differential Equations for Irregular Time Series. Kidger, Morrill, Foster, Lyons (2020)
GRU-ODE-Bayes: Continuous modeling of sporadically-observed time series. de Brouwer, Simm, Arany, Moreau. (2020)

Continuous-time Time Series Models



Latent ODEs for Irregularly-Sampled Time Series. Rubanova, Chen, Duvenaud. 2020

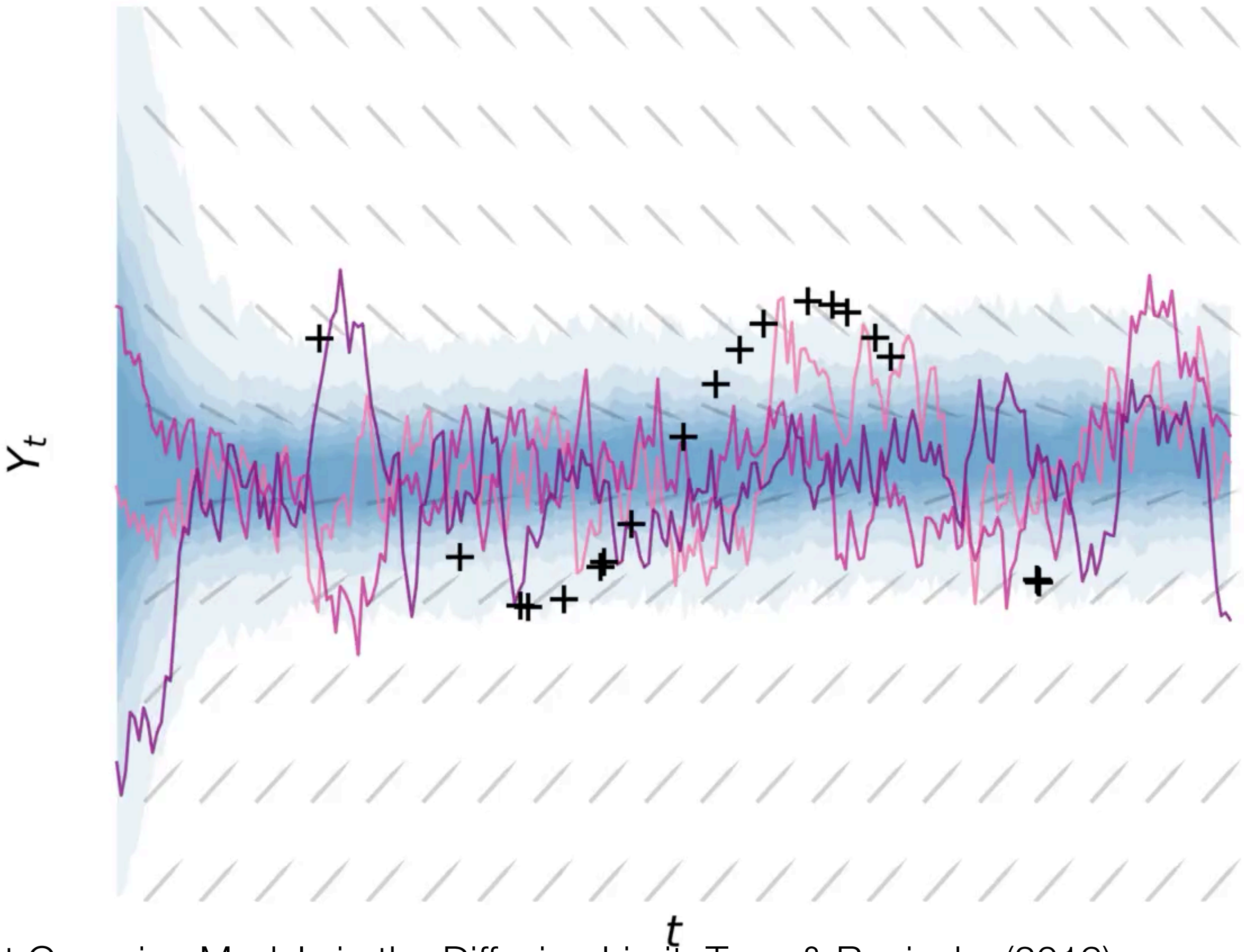
Neural Stochastic Differential Equations

Recently generalized to stochastic differential equations

Still O(1) memory and can use adaptive SDE solvers

Bayesian model with prior and approximator posterior SDEs

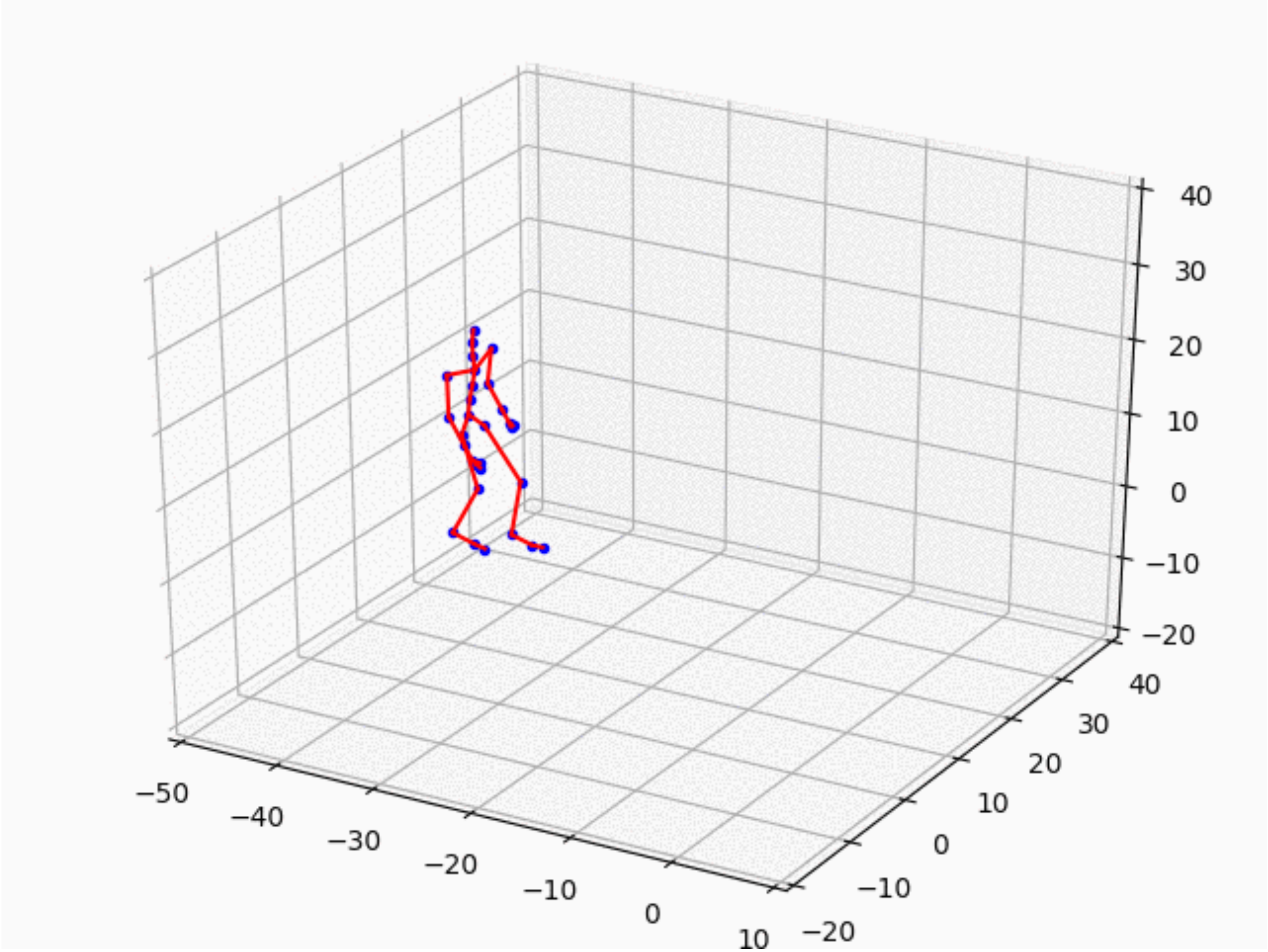
Handles unseen interventions



Neural Stochastic Differential Equations: Deep Latent Gaussian Models in the Diffusion Limit. Tzen & Raginsky (2019)
Scalable Gradients for Stochastic Differential Equations. Li, Wong, Chen, Duvenaud (2020)

Neural Stochastic Differential Equations

Trains with stochastic variational inference, scalable in number of parameters and state dimension



Neural Stochastic Differential Equations: Deep Latent Gaussian Models in the Diffusion Limit. Tzen & Raginsky (2019)
Scalable Gradients for Stochastic Differential Equations. Li, Wong, Chen, Duvenaud (2020)

Outline

Background and applications of implicit layers

The mathematics of implicit layers

Deep Equilibrium Models

Neural ODEs

Differentiable optimization

Future directions

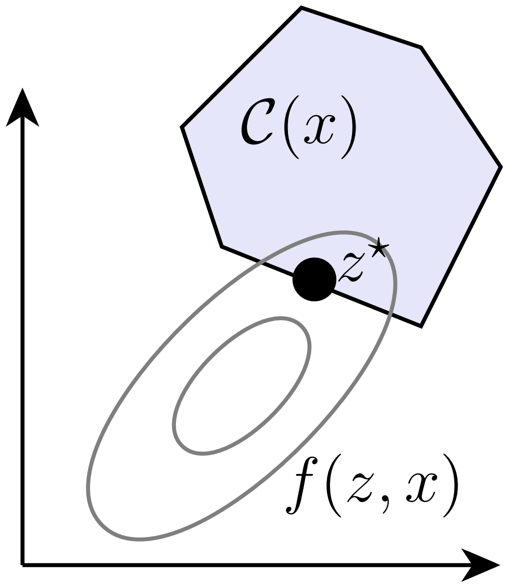
Differentiable optimization

DEQs and Neural ODEs both impose substantial structure on the nature of the layer, in order to gain substantial representational power

Other common strategy for imposing a different (but related) kind of structure is that of differentiable optimization

Layer of the form

$$z^\star = \operatorname*{argmin}_{z \in \mathcal{C}(x)} f(z, x)$$



Differentiating optimization problems

How do we differentiate through a layer?

$$z^\star = \operatorname*{argmin}_{z \in \mathcal{C}(x)} f(z, x)$$

Finding a solution to constrained optimization is equivalent to finding the solution of a of a set of nonlinear equations called KKT conditions

$$\begin{aligned} z^\star = \quad & \operatorname{argmin} \ \frac{1}{2} z^T Q(x) z + p(x)^T z \\ & \text{subject to} \ A(x) z = b(x), \\ & \qquad\qquad\quad G(x) z \leq h(x) \end{aligned}$$

$$\begin{aligned} & Find \ (z^\star, \nu^\star, \lambda^\star) \ s.t. \\ & 1. \, Az^\star = b \\ & 2. \, Gz^\star \leq h \\ & 3. \, \lambda^\star \geq 0 \\ & 4. \, \lambda^\star \circ (Gz^\star - h) = 0 \\ & 5. \, Qz^\star + p + A^T \nu^\star + G^T \lambda^\star = 0 \end{aligned}$$

Differentiating through optimization problems

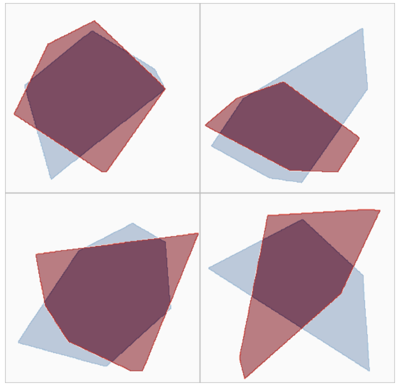
Alternatively, we can view virtually any optimization procedure as a fixed point iteration; e.g. for projected gradient descent

$$z_{k+1} = \mathrm{Proj}_{C(x)}[z_k - \alpha \partial_0 f(z_k, x)]$$

(But also true of much more sophisticated optimization approaches)

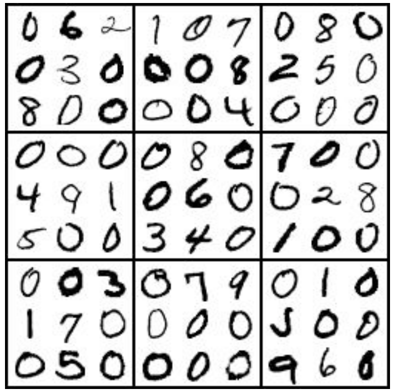
Therefore, can use differentiation of fixed point iteration to differentiate through optimization problems!

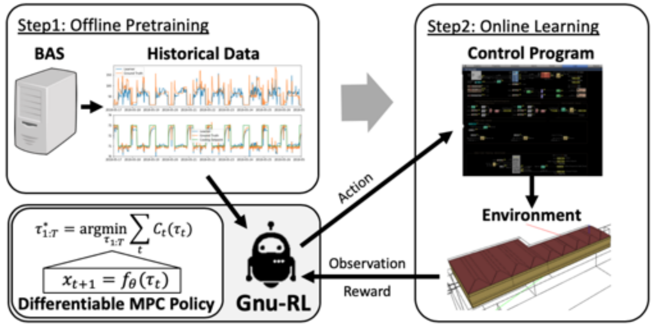
Some example applications



Learning a convex polytope from data
[Amos and Kolter., 2018]

Solving Sudoku (w/ MNIST digits) using
differentiable SDP solver [Wang et al., 2019]





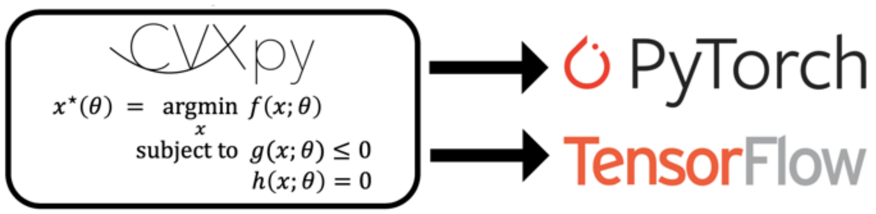
Controlling HVAC systems with differentiable
MPC controllers [Chen et al., 2019]

CvxpyLayers: Differentiable convex modeling

Differentiable optimization traditionally involved implementing the (potentially complex) optimization solution method

cvxpylayers tool allows one to easily write generic optimization problems using the **cvxpy** library, export directly as Tensorflow/PyTorch layers

https://github.com/cvxgrp/cvxpylayers




```
import cvxpy as cp
import torch
from cvxpylayers.torch import CvxpyLayer

n, m = 2, 3
x = cp.Variable(n)
A = cp.Parameter((m, n))
b = cp.Parameter(m)
constraints = [x >= 0]
objective = cp.Minimize(0.5 * cp.pnorm(A @ x - b, p=1))
problem = cp.Problem(objective, constraints)
assert problem.is_dpp()

cvxpylayer = CvxpyLayer(problem, parameters=[A, b], variables=[x])
A_tch = torch.randn(m, n, requires_grad=True)
b_tch = torch.randn(m, requires_grad=True)

# solve the problem
solution, = cvxpylayer(A_tch, b_tch)

# compute the gradient of the sum of the solution with respect to A, b
solution.sum().backward()
```

Outline

Background and applications of implicit layers

The mathematics of implicit layers

Deep Equilibrium Models

Neural ODEs

Differentiable optimization

Future directions

When to use DEQs vs Neural ODEs

Use DEQs for:

- Drop-in implicit replacement for deep models.
- Supervised learning
    - convnets, resnets
    - transformers
- Standard unsupervised learning
    - E.g. Language models

Use Neural ODEs if you need:

- Continuous-time series models
    - Irregular-sampled time series
    - physics models
- Flexible density models
    - E.g. manifolds
- A homeomorphism
    - warping a 3d shape

Open Problems and Future Directions

1. Regularizing DEQs and Neural ODEs to be faster to solve

2. Re-architecting models to take advantage of memory advantages

3. Scaling and application of latent SDEs

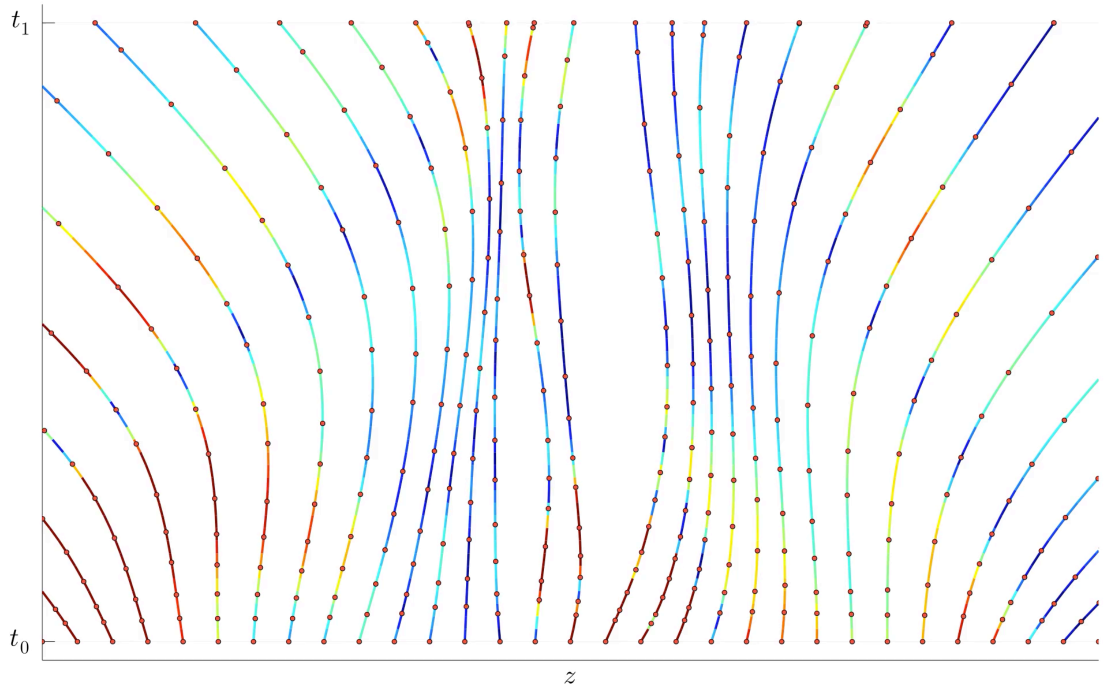
4. Partial differential equation (PDE) solutions as a layer

Future Direction: Regularizing to be Easy to Solve

How to control number of function evaluations?

Idea so far for ODEs: Regularize dynamics to have small derivatives.

Can trade model quality for speed.



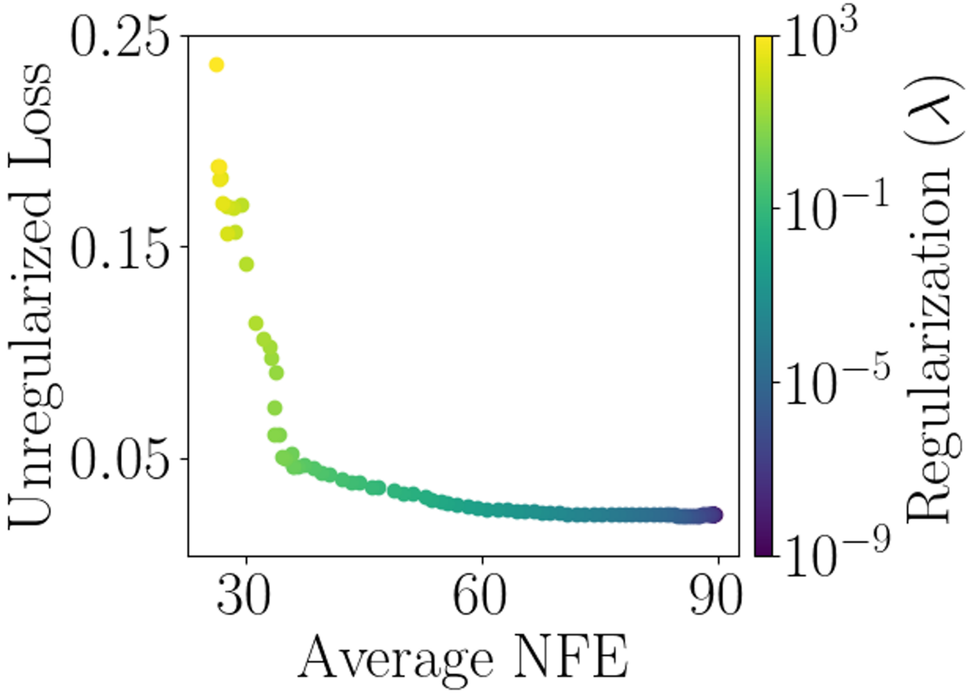
- How to train your neural ODE: the world of Jacobian and kinetic regularization. Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, Adam M Oberman. (2020)
- Learning Differential Equations that are Easy to Solve. Kelly, Bettencourt, Johnson, Duvenaud. (2020)

Future Direction: Regularizing to be Easy to Solve

How to control number of function evaluations?

Idea so far for ODEs: Regularize dynamics to have small derivatives.

Can trade model quality for speed.

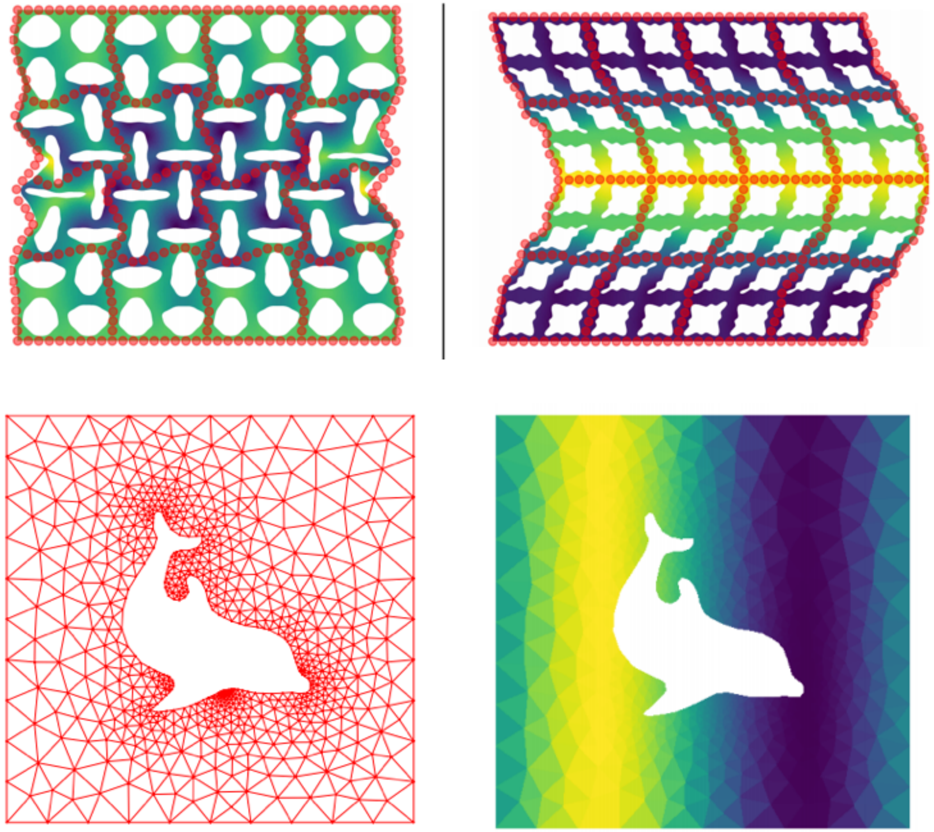


- How to train your neural ODE: the world of Jacobian and kinetic regularization. Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, Adam M Oberman. (2020)
- Learning Differential Equations that are Easy to Solve. Kelly, Bettencourt, Johnson, Duvenaud. (2020)

Future Direction: Neural Partial Differential Equations

Similar adjoint equations for differentiating through PDEs.

Can amortize solution of PDE simultaneously while optimizing its parameters.



- Learning Composable Energy Surrogates for PDE Order Reduction. Beatson, Ash, Roeder, Xue, Adams (2020)
- Amortized Finite Element Analysis for Fast PDE-Constrained Optimization.Xue, Beatson, Adriaenssens, Adams (2020)-
- Learning Neural PDE Solvers with Convergence Guarantees. Hsieh, Zhao, Eismann, Mirabella, Ermon (2020)
- Fourier Neural Operator for Parametric Partial Differential Equations. Zongyi Li, Nikola Kovachki, Azizzadenesheli, Liu, Bhattacharya, Stuart, Anandkumar (2020)

Additional Code

http://github.com/rtqichen/torchdiffeq - General code for ODEs in PyTorch

http://github.com/YuliaRubanova/latent_ode - PyTorch latent ODEs

http://github.com/jacobjinkelly/easy-neural-ode/ - Jax latent ODEs, FFJORD

http://github.com/google-research/torchsde/ - PyTorch latent SDEs

http://github.com/locuslab/deq - Deep Equilibrium Models

http://github.com/locuslab/mdeq - Multiscale DEQs

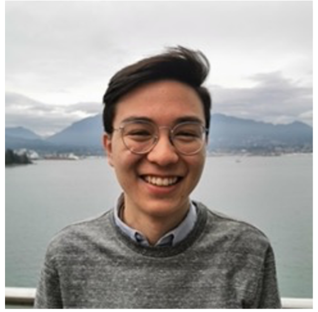
http://github.com/cvxgrp/cvxpylayers - Convex optimization as a layer

Thank you to all our collaborators and beyond



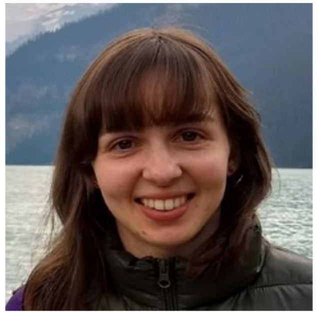


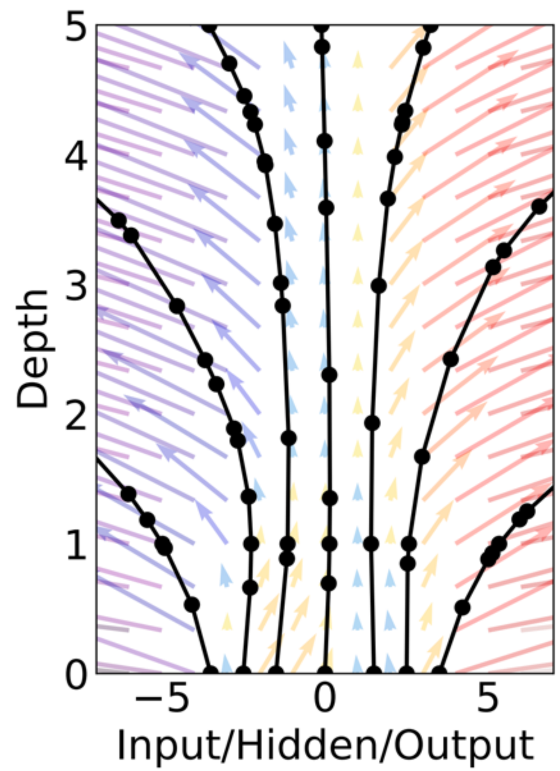


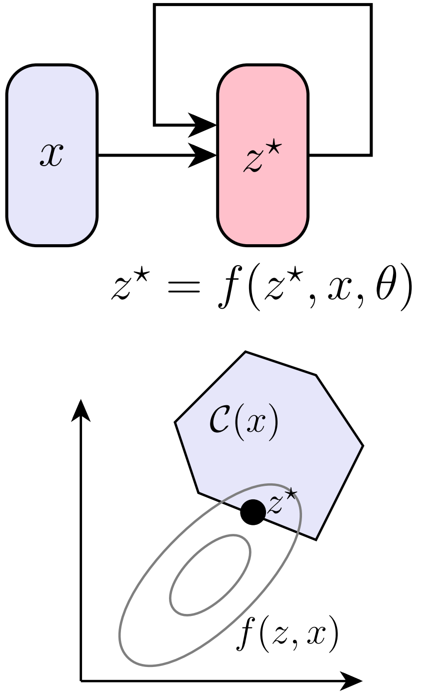













Deep Implicit Layers:

Neural ODEs, Equilibrium Models, and Beyond

http://implicit-layers-tutorial.org

David Duvenaud
University of Toronto
and Vector Institute

J. Zico Kolter
Carnegie Mellon and
Bosch Center for AI

Matt Johnson
Google Brain