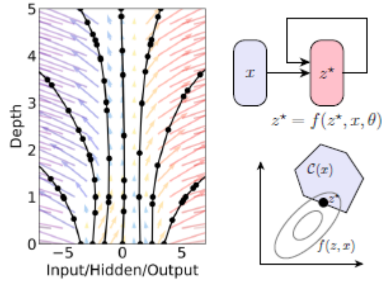
Deep Implicit Layers

2021-1-7

손형욱

hyounguk.shon@kaist.ac.kr

Based on the NIPS 2020 tutorial



Deep Implicit Layers:
Neural ODEs, Equilibrium Models, and Beyond

http://implicit-layers-tutorial.org

David Duvenaud
University of Toronto and Vector Institute

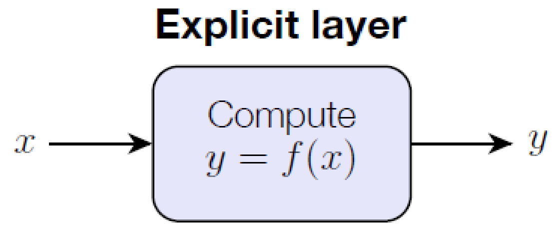
J. Zico Kolter
Carnegie Mellon and Bosch Center for AI

Matt Johnson
Google Brain

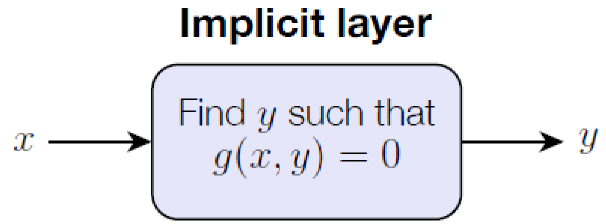
Learning implicit functions

- Implicit functions in Deep Learning:
  - Deep implicit layers
  - Implicit neural representations
- Implicit function opens a new search space for deep learning.
  - Novel architectures for learning implicit functions
  - Novel representations that simplify difficult problems
- Deep implicit layers
  - A layer is a differentiable parametric function.
  - An emerging family of neural networks. (roots back to 1980s)
- Why implicit layers?
  - More representation power
  - Extreme memory efficiency
  - Simple architecture
  - Isolates the layer's behavior from its computation.

- Occupancy Networks
- Neural Radiance Field (NeRF)



$$y = \sqrt{1 - x^2}$$



$$x^2 + y^2 = 1$$

Deep Learning Toolbox

powerful new models!

DEQ

Neural ODE

CNN

Transformer

RNN
LSTM

GNN

BNN

GAN

EBM

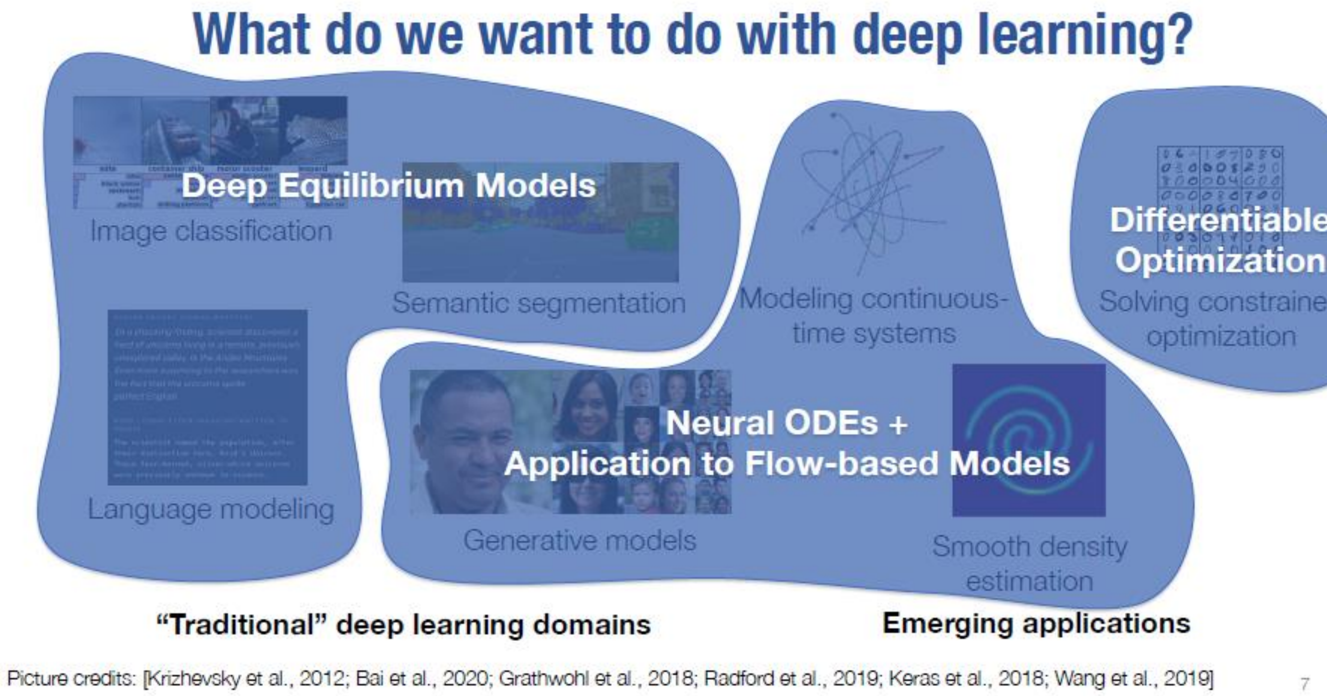
VAE

Normalizing Flows

Overview of the topics

- Deep Equilibrium Models (DEQs)
- Neural Ordinary Differential Equations (Neural ODEs)
- Rules for forward and backward pass
- Applications
- Future directions

Chen et al., Neural Ordinary Differential Equations. NIPS 2018.
Bai et al., Deep Equilibrium Models. NIPS 2019.



Deep Equilibrium Models

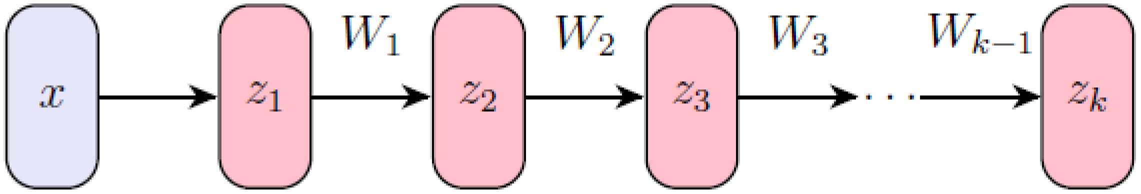
Shaojie Bai, Zico Kolter, Vladlen Koltun. *Deep Equilibrium Models*. NeurIPS 2019.

Shaojie Bai, Vladlen Koltun, Zico Kolter. *Multiscale Deep Equilibrium Models*. NeurIPS 2020.

Examples where weight-tying was used for improving performance:
- Dehghani et al., Universal Transformers. ICLR 2019.
- Lan et al., ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. ICLR 2020.

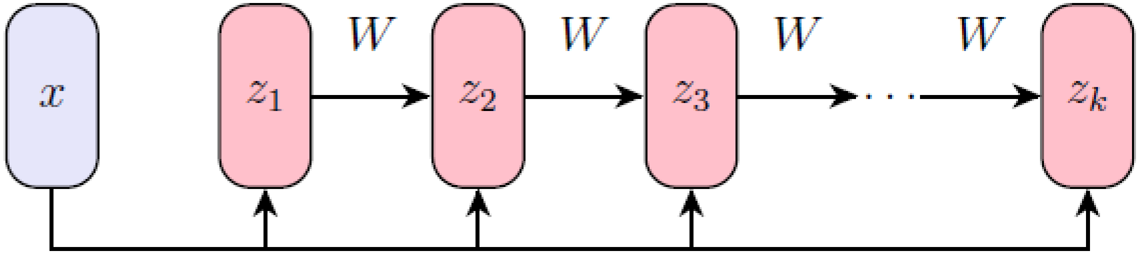
A simple example: Fixed-Point Layer

- A deep neural network:



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

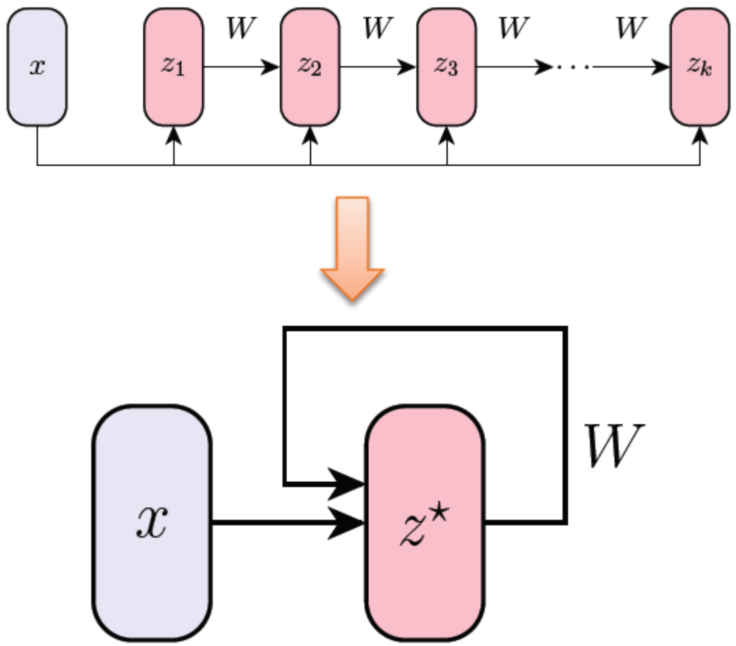
- Our fixed-point network:
  1. Parameter W is shared across layers.
  2. The input $\boldsymbol{x}$ is injected to every layers.



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

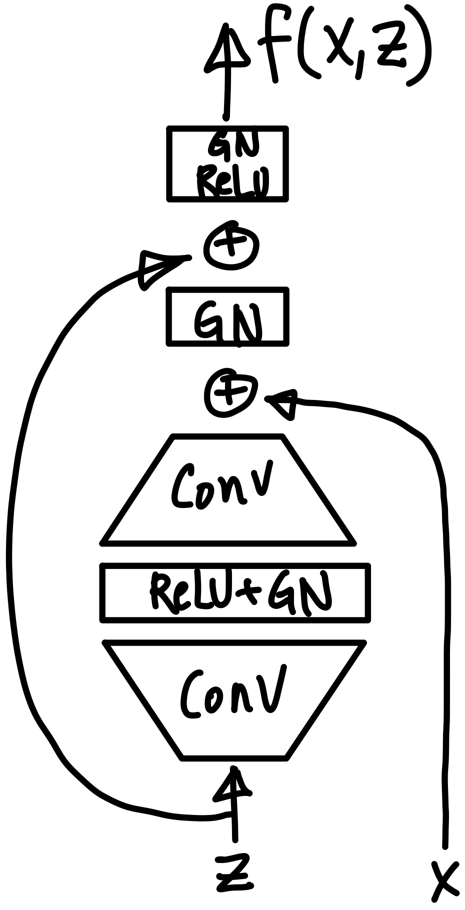
Fixed-Point Layer: forward and backward

- Fixed-Point Layer
  - We repeatedly apply the same transformation to hidden units.
  - At $k \to \infty$, the hidden unit converges to some fixed-point $\boldsymbol{z}^*$ such that $\boldsymbol{z}^* = \sigma(W\boldsymbol{z}^* + \boldsymbol{x})$
  - We use $\boldsymbol{z}^*$ as the output of our fixed-point layer.
- Forward pass is fixed-point computation
  1. Input: $\boldsymbol{x}$
  2. Iterate $\boldsymbol{z}_{i+1} = f(\boldsymbol{z}_i, \boldsymbol{x})$ until $\|\boldsymbol{z}_{i+1} - \boldsymbol{z}_i\| < \varepsilon$.
  3. Output: $\boldsymbol{z}^*$
- Backward pass
  - Backpropagation through time (it works, but inefficient)
- DEQ uses efficient forward and backward algorithms.
  - Fast forward computation
  - Fast and very memory-efficient backward computation



Deep Equilibrium Models

- Deep Equilibrium Models
  - So far, $f_\theta(\boldsymbol{x}, \boldsymbol{z}) = \sigma(W\boldsymbol{z} + \boldsymbol{x})$ was just a single layer.
  - DEQ incorporates more expressive architectures for $f_\theta(\boldsymbol{x}, \boldsymbol{z})$.
- Forward pass through a DEQ
  1. Input: $\boldsymbol{x}$
  2. DEQ uses fixed-point solvers that are substantially more efficient.
  3. Output: $\boldsymbol{z}^*$
- Accelerating convergence of fixed-point iterations
  - Root finding: Solve $f(\boldsymbol{z}, \boldsymbol{x}) - \boldsymbol{z} = \boldsymbol{0}$ (e.g. Broyden's method)
  - Anderson acceleration: Extrapolate $\boldsymbol{z}_{i+1}$ from previous iterations.
- Anderson acceleration method:
  - $\boldsymbol{z}_{i+1} = \alpha_1 f(\boldsymbol{z}_i) + \alpha_2 f(\boldsymbol{z}_{i-1}) + \cdots + \alpha_M f(\boldsymbol{z}_{i-M+1})$
  - $\sum \alpha_m = 1$
  - Choose $\boldsymbol{\alpha} = \mathrm{argmin}|\boldsymbol{G}\boldsymbol{\alpha}|$, $\boldsymbol{G} = [f(\boldsymbol{z}_i) - \boldsymbol{z}_i \quad \cdots \quad f(\boldsymbol{z}_{i-M+1}) - \boldsymbol{z}_{i-M+1}]$

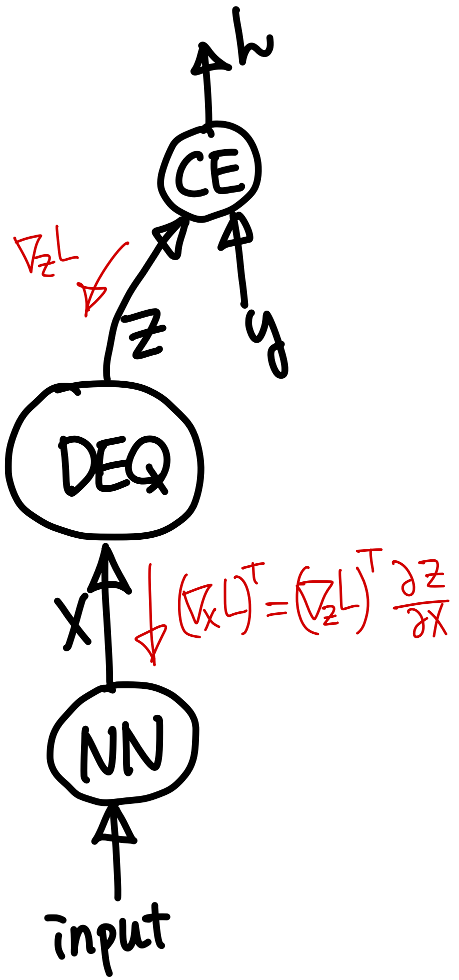


A residual cell

Backward pass through a DEQ

Takeaways:
1. Only need to record the final fixed-point z^*.
2. Backward have only $\mathcal{O}(1)$ memory footprint.
3. Backward solves another fixed-point problem.

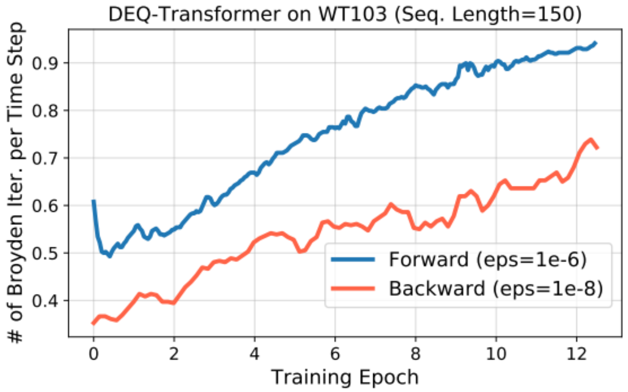
- The implicit differentiation rule
  - $f(x,z) - z = 0$
  - $\frac{\partial f(x,z)}{\partial x} + \frac{\partial f(x,z)}{\partial z}\frac{\partial z}{\partial x} - \frac{\partial z}{\partial x} = 0$
  - $\frac{\partial z}{\partial x} = \left[\mathrm{I} - \frac{\partial f(x,z)}{\partial z}\right]^{-1} \frac{\partial f(x,z)}{\partial x}$
  - Only requires the output z^*, not the trajectory $z_1, z_2, \dots, z_t$.
- Backpropagation through DEQ
  - We want to compute the vector-Jacobian product: $(\nabla_x \mathcal{L})^\intercal = (\nabla_z \mathcal{L})^\intercal \frac{\partial z}{\partial x}$
  - $(\nabla_z \mathcal{L})^\intercal \frac{\partial z}{\partial x} = (\nabla_z \mathcal{L})^\intercal \left[\mathrm{I} - \frac{\partial f(x,z)}{\partial z}\right]^{-1} \frac{\partial f(x,z)}{\partial x}$
  - Let $v^\intercal = (\nabla_z \mathcal{L})^\intercal \left[\mathrm{I} - \frac{\partial f(x,z)}{\partial z}\right]^{-1}$
  - $v^\intercal = v^\intercal \frac{\partial f(x,z)}{\partial z} + (\nabla_z \mathcal{L})^\intercal$ (this is a fixed-point equation)
  - Backward pass is another fixed-point problem!



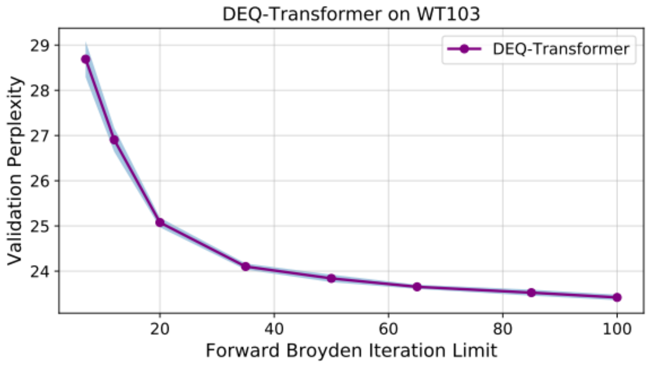
*same works for the parameter gradient,
$(\nabla_z \mathcal{L})^\intercal \frac{\partial z}{\partial \theta} = (\nabla_z \mathcal{L})^\intercal \left[\mathrm{I} - \frac{\partial f(x,z)}{\partial z}\right]^{-1} \frac{\partial f(x,z)}{\partial \theta}$

Deep Equilibrium Models: Summary

- Inference and training DEQs
  - DEQs = infinite-depth neural networks
  - Forward pass runs a fixed-point solver.
  - Backward pass runs another fixed-point solver.
  - Fixed-points are computed efficiently via acceleration methods.
- Training DEQs (vs. traditional neural networks)
  - $\mathcal{O}(1)$ memory training regardless of depth of layers. (vs. $\mathcal{O}(L)$ memory)
  - No need to store the activations of each intermediate layer.
- Why does DEQs matter?
  - Generally performs better when model sizes are similar.
  - Requires much less GPU memory for training
  - Provides a mechanism to trade off accuracy vs. latency at test-time
- Current shortcomings
  - Training DEQs typically takes 2x – 3x longer.
  - Perhaps we can regularize DEQs to be faster to solve.



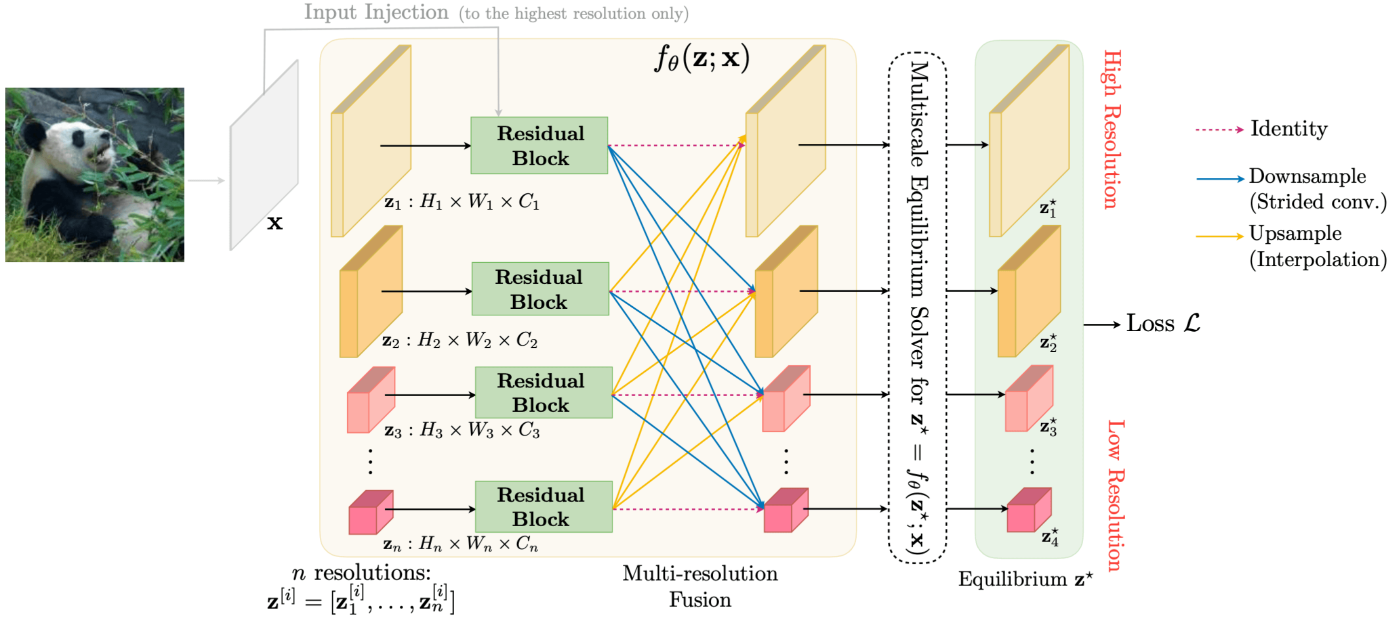
Finding the fixed-point becomes harder as the model fits the dataset. (the model learns to become 'deep')

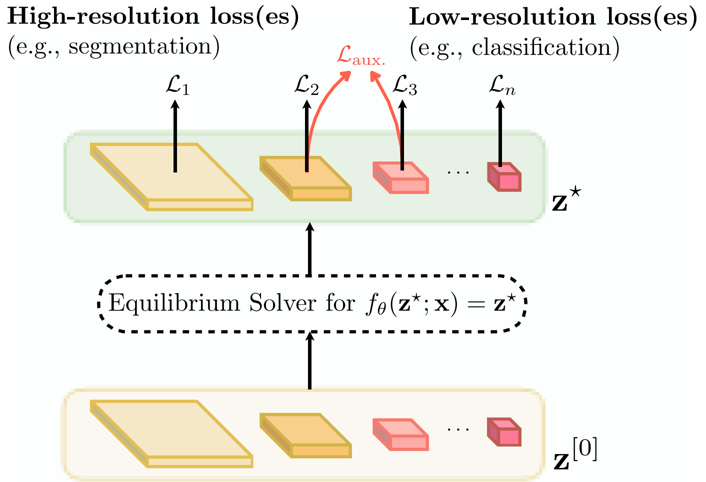


Perplexity vs. latency trade-off

Deep Equilibrium Models: applications

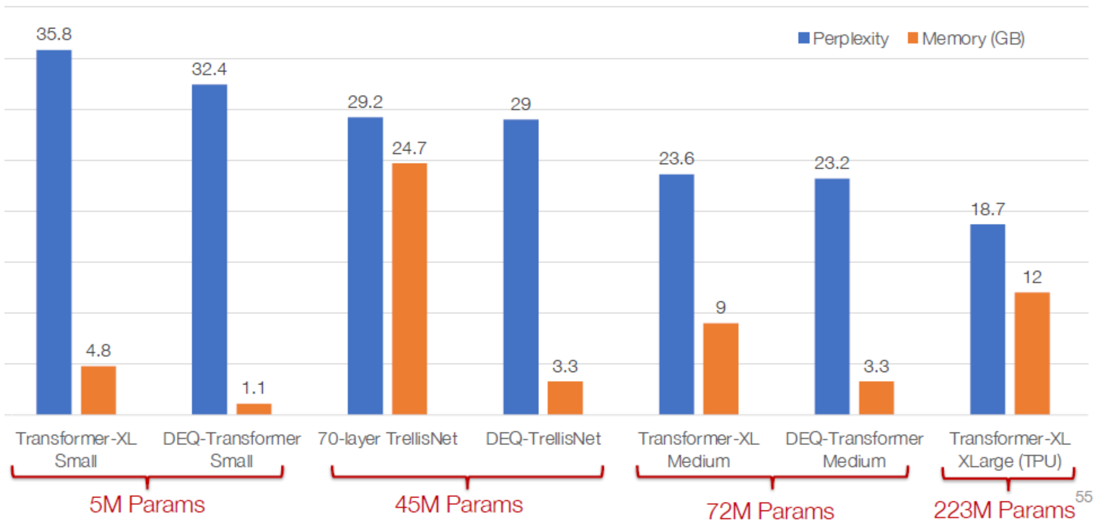
- Advantages
  - Represent modern deep networks using a single implicit layer.
  - Adaptive computation
  - Competitive performance in large-scale NLP tasks.
- Multiscale DEQ
  - Competitive performance in large-scale visual tasks.



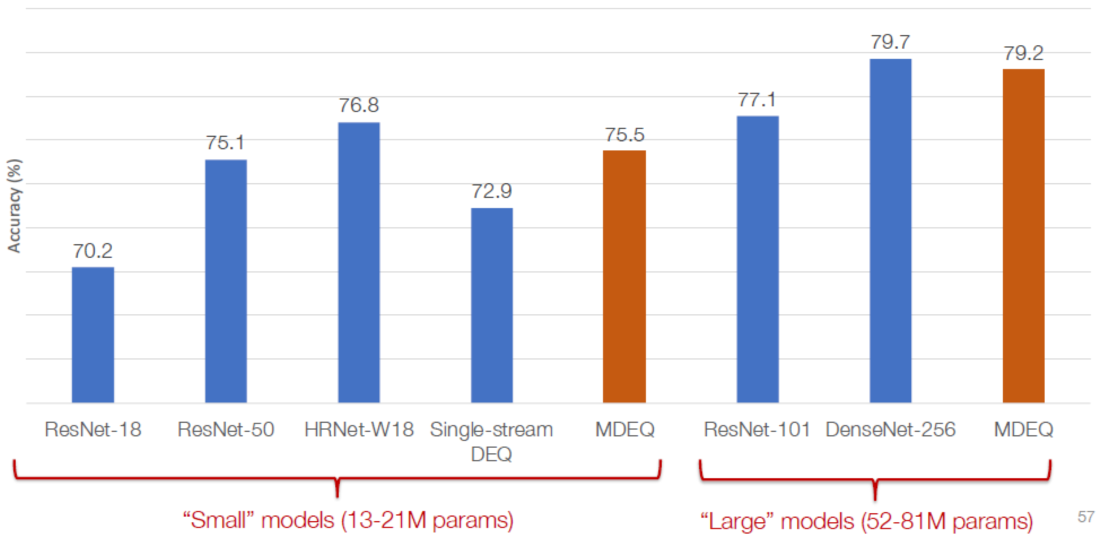


Bai, Koltun, Kolter. Multiscale Deep Equilibrium Models, NeurIPS 2020

Language modeling: WikiText-103



ImageNet Top-1 Accuracy



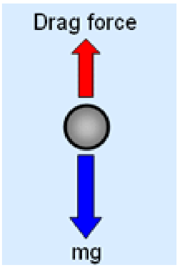
Citiscapes mIoU

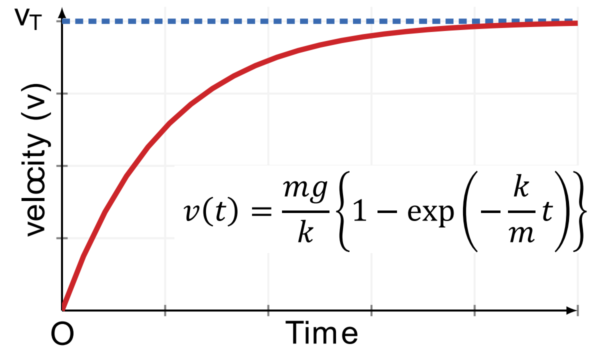
Bai, Koltun, Kolter. Multiscale Deep Equilibrium Models, NeurIPS 2020

Neural ODEs

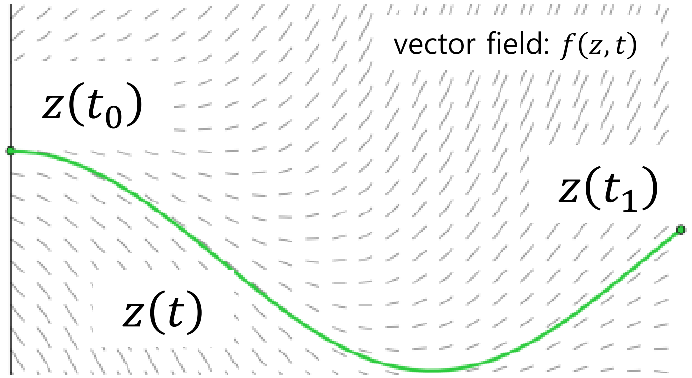
Ricky Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud. *Neural Ordinary Differential Equations*. NeurIPS 2018.

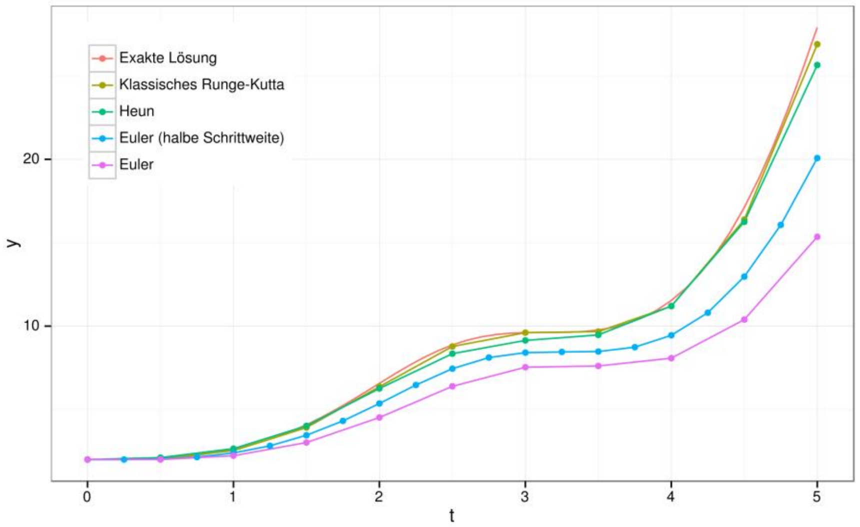
Background: Ordinary Differential Equations





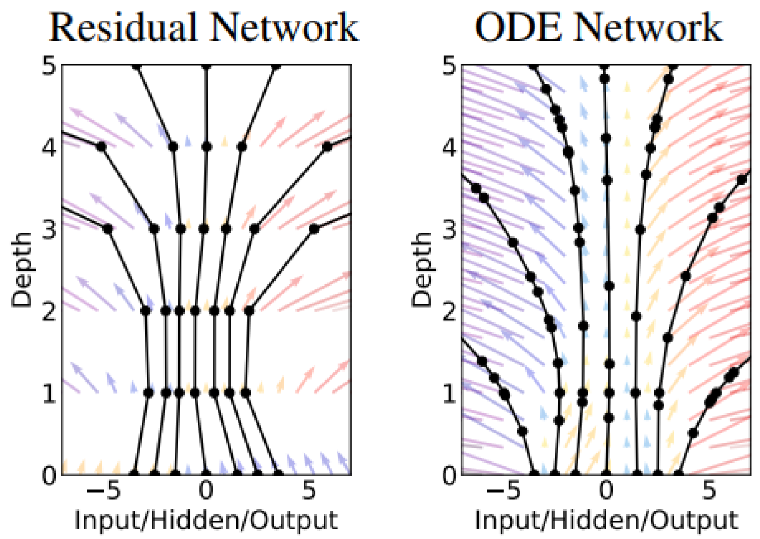
- Velocity of a free-falling object
  - ODE specifies a continuous-time state variable by its rate of change.
  - State variable: $v(t)$
  - State dynamics: $m\frac{dv}{dt} = mg - kv$
- Ordinary Differential Equations (1st order)
  - The state vector $z(t)$ follows the dynamics $f(z,t)$.
  - $\begin{cases} \frac{dz}{dt} = f(z,t) \\ z(t_0) = z_0 \end{cases}$
  - Dynamics + initial state = Initial value problem
- Solving an ODE means finding $z(t)$.
  - We can evaluate $z(t_1)$ by integrating $f(z,t)$ for $t \in [t_0, t_1]$.
  - $z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z,t)dt$
  - We have numerical ODE solvers. (e.g., Euler, Runge-Kutta, Dopri5)
  - Modern solvers adaptively adjust the number of function evaluations according to the complexity of the dynamics.





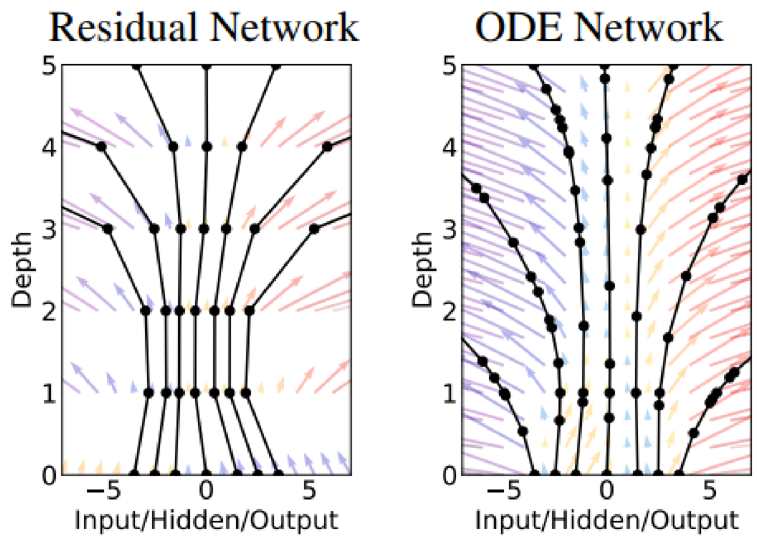
Weight-tied Residual Network

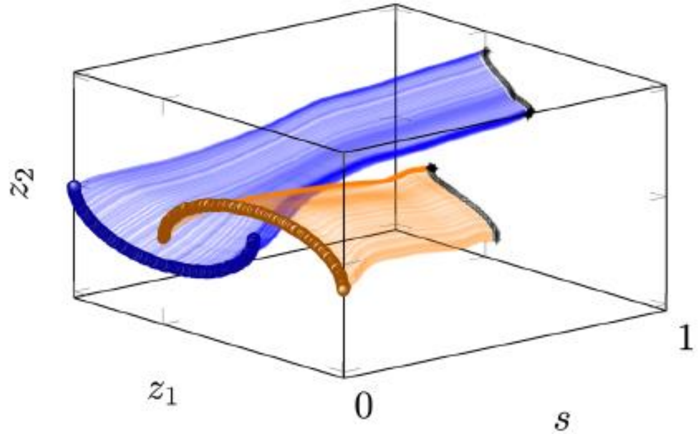
- Neural ODE is a depth-continuous ResNet.
  - Residual block: $z_{i+1} = z_i + f_\theta(z_i, i)$
  - Neural ODE: $dz = f_\theta(z, t)dt$



Neural ODE

- Parametrizes the dynamics $f(z,t)$ by a neural network.
  - $\frac{dz}{dt} = f_\theta(z,t)$
  - $z(t)$ is the hidden state vector.
  - f_θ is just a MLP. ($\mathbb{R}^{|z|+1} \to \mathbb{R}^{|z|}$)
- Forward pass is an initial value problem.
  - Use numerical solver: $z(t_1) = \text{ODEint}(f_\theta, z_0, [t_0, t_1])$
  - z_0 : layer input
  - $z(t_1)$: layer output
- Neural ODE is a depth-continuous ResNet.
  - Residual block: $z_{i+1} = z_i + f_\theta(z_i, i)$
  - Neural ODE: $dz = f_\theta(z,t)dt$
- What are Neural ODEs good for?
  - Can be used anywhere a ResNet can.
  - Flexible density estimation and time-series models.
  - Whenever knowing the trajectory is important.





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

Backward pass through Neural ODE

- Don't backpropagate through time – there's a better way!
  - Things we want to compute: $\frac{\partial \mathcal{L}}{\partial z(0)}$ and $\frac{\partial \mathcal{L}}{\partial \theta}$
- Adjoint Sensitivity method
  - Differentiating Neural ODE ends up with another ODE problem.
  - Again, we use an ODE solver to compute the gradients.
  - Only memorize the final state $z(t_1)$, then $z(t)$ is exactly reproducible.

Differentiation in discrete-time (weight-tied ResNet)

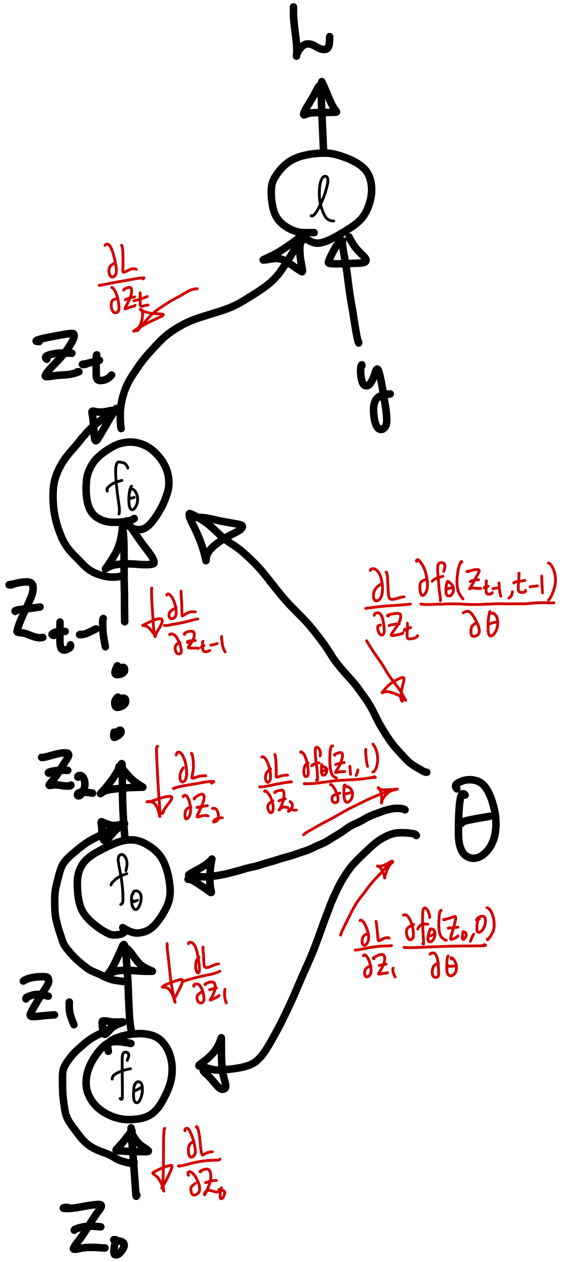
$$\frac{\partial \mathcal{L}}{\partial z_t} = \frac{\partial \mathcal{L}}{\partial z_{t+1}} + \frac{\partial \mathcal{L}}{\partial z_{t+1}} \frac{\partial f_\theta(z_t, t)}{\partial z_t}$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_t \frac{\partial \mathcal{L}}{\partial z_{t+1}} \frac{\partial f_\theta(z_t, t)}{\partial \theta}$$

Differentiation in continuous-time (Neural ODE)

$$\frac{\partial}{\partial t} \frac{\partial \mathcal{L}}{\partial z(t)} = -\frac{\partial \mathcal{L}}{\partial z(t)} \frac{\partial f_\theta(z, t)}{\partial z}$$

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



Weight-tied ResNet

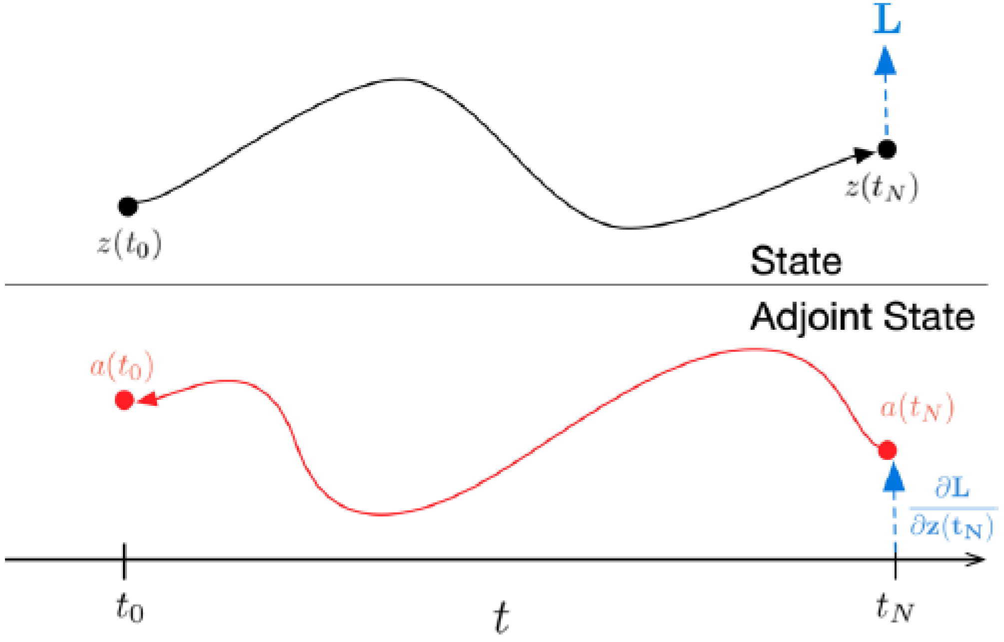
Backward pass through Neural ODE

Takeaways:
1. Only need to record the final state $z(t_1)$.
2. Therefore, backward uses $\mathcal{O}(1)$ memory.
3. Backward solves another ODE problem.

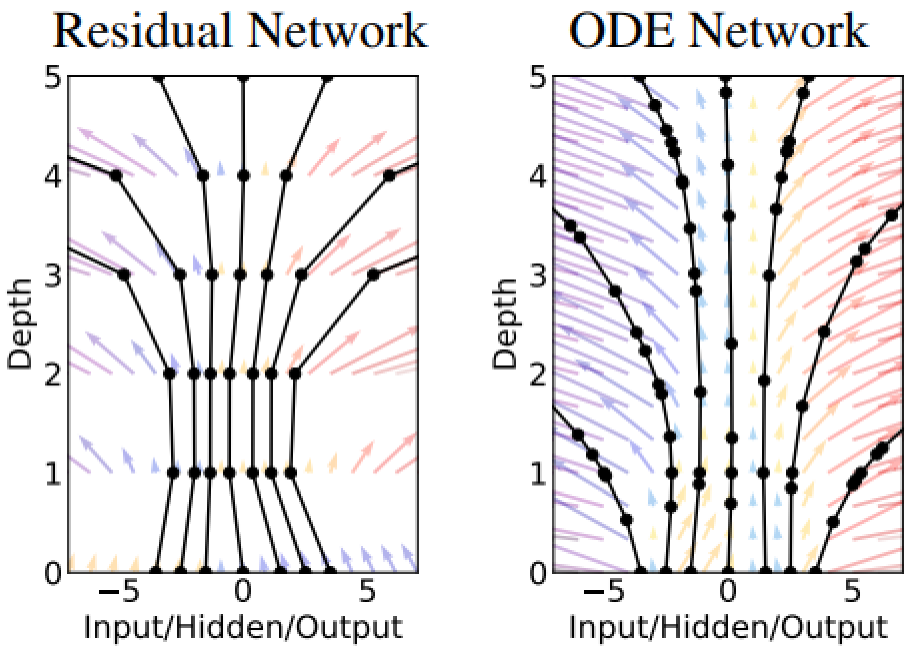
- Adjoint sensitivities
  - $\frac{\partial z(t)}{\partial t} = f_\theta(z,t),\ z(1) = z(1)$
  - $\frac{\partial a(t)}{\partial t} = -a(t) \cdot \frac{\partial f_\theta(z,t)}{\partial z},\ a(1) = \frac{\partial \mathcal{L}}{\partial z(1)}$
  - $\frac{\partial d(t)}{\partial t} = -a(t) \cdot \frac{\partial f_\theta(z,t)}{\partial \theta},\ d(1) = 0$

$z(t)$	hidden state
$a(t) = -\frac{\partial \mathcal{L}}{\partial z(t)}$	adjoint state
$d(t) = \frac{\partial \mathcal{L}}{\partial \theta}$	parameter gradient

- Solve the augmented ODE.
  - Define the augmented state $s(t) = \begin{bmatrix} z(t) \\ a(t) \\ d(t) \end{bmatrix}$
  - $\frac{\partial s(t)}{\partial t} = \begin{bmatrix} \frac{\partial z(t)}{\partial t} \\ \frac{\partial a(t)}{\partial t} \\ \frac{\partial d(t)}{\partial t} \end{bmatrix} = \begin{bmatrix} f_\theta(z,t) \\ -a(t) \cdot \frac{\partial f_\theta(z,t)}{\partial z} \\ -a(t) \cdot \frac{\partial f_\theta(z,t)}{\partial \theta} \end{bmatrix}$
  - $s(1) = \begin{bmatrix} z(1) \\ a(1) \\ d(1) \end{bmatrix} = \begin{bmatrix} z_1 \\ \frac{\partial \mathcal{L}}{\partial z_1} \\ \mathbf{0} \end{bmatrix}$
  - Solve the IVP: $s(0) = \mathrm{ODEint}\left(\frac{\partial s(t)}{\partial t}, s(1), [1,0]\right)$

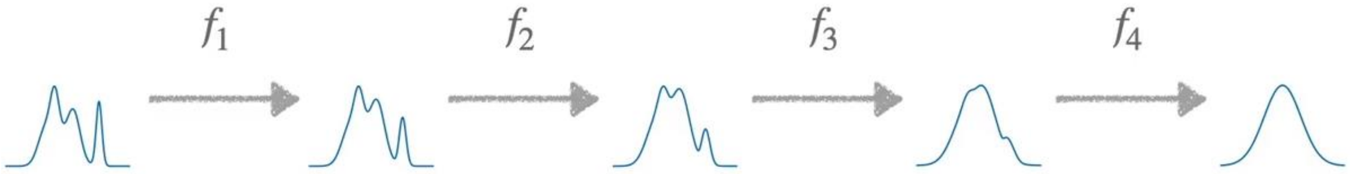


Neural ODE: Summary



- Inference and training Neural ODEs
  - Neural ODEs = depth-continuous residual networks
  - Forward pass runs an ODEsolve.
  - Backward pass is also an ODEsolve. (Adjoint sensitivity method)
- Training Neural ODEs (vs. traditional neural networks)
  - Constant memory regardless of depth of layers. (vs. $\mathcal{O}(L)$ memory)
  - No need to store the activations of each intermediate layer.
  - Gradient computation runs an ODEsolve in reversed time t.
- When to use Neural ODEs
  - Trajectory of the feature vector is important (e.g., continuous time series)
  - Using normalizing flows (easier change of variables)
- DEQ and Neural ODE
  - $\mathcal{O}(1)$ memory training.
  - Provides a mechanism to balance between numerical precision vs. latency at test-time.
  - Infinite / adjustable depth
  - Adaptive computation depending on the complexity of the problem.

Background: Normalizing Flows

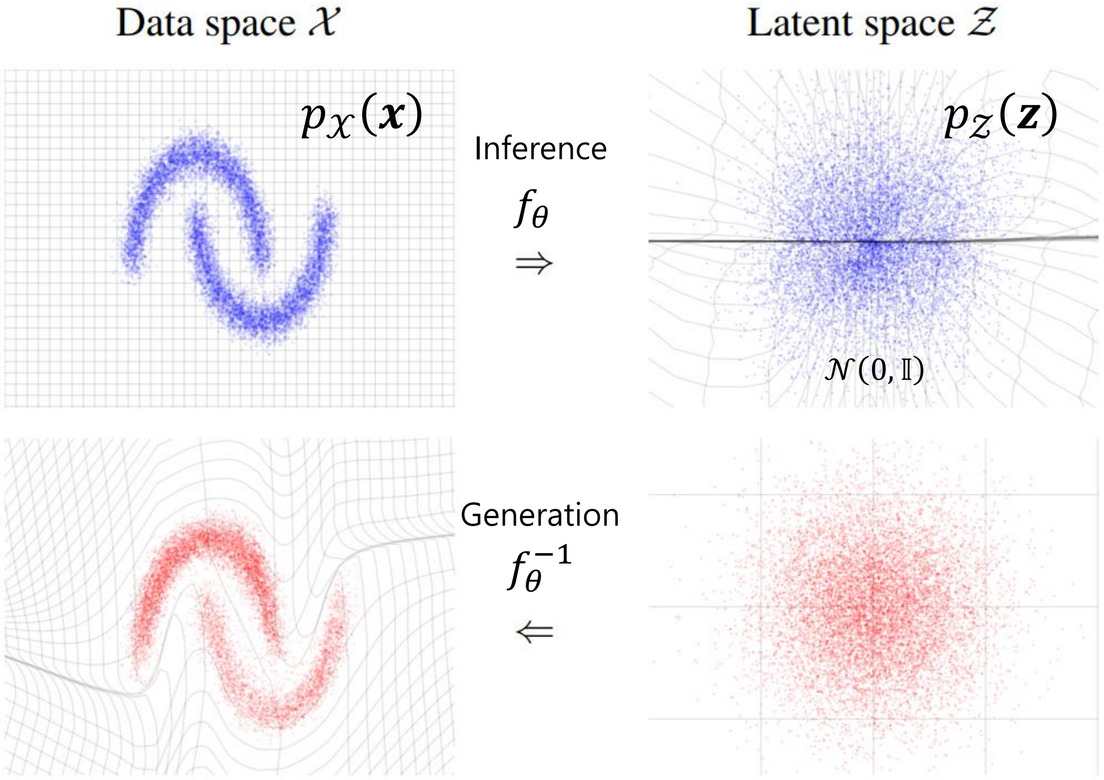


Brubaker et al., Introduction to Normalizing Flows (ECCV2020 Tutorial)

- Normalizing Flow
  - learns mapping from a complex distribution $p_{\mathcal{X}}$ into a simple distribution $p_{\mathcal{Z}}$.
  1. Uses invertible nonlinear transformations. ("flow")
     - $\boldsymbol{z} = f_\theta(\boldsymbol{x})$
     - $\boldsymbol{x} = f_\theta^{-1}(\boldsymbol{z})$

     *Requires special invertible architectures.
  2. Change of variables formula
     - $p_{\mathcal{X}}(\boldsymbol{x}) = p_{\mathcal{Z}}(\boldsymbol{z}) \cdot \left|\det\left(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}\right)\right|$
     - Typically, $p_{\mathcal{Z}}$ is fixed to a unit gaussian distribution.
     - Gives ability to compute $p(\boldsymbol{x})$ directly.

     *Requires Jacobians to be structured.
  3. Increases the flexibility by stacking layers of flow.
     - $f_\theta = f_K \circ \cdots \circ f_2 \circ f_1$
  4. Training normalizing flows for density estimation
     - $\theta^* = \underset{\theta}{\operatorname{argmax}} \log p(\boldsymbol{x}) = \underset{\theta}{\operatorname{argmax}} \left\{\log p\big(f_\theta(\boldsymbol{x})\big) + \log\left|\det\left(\frac{\partial f_\theta(\boldsymbol{x})}{\partial \boldsymbol{x}}\right)\right|\right\}$

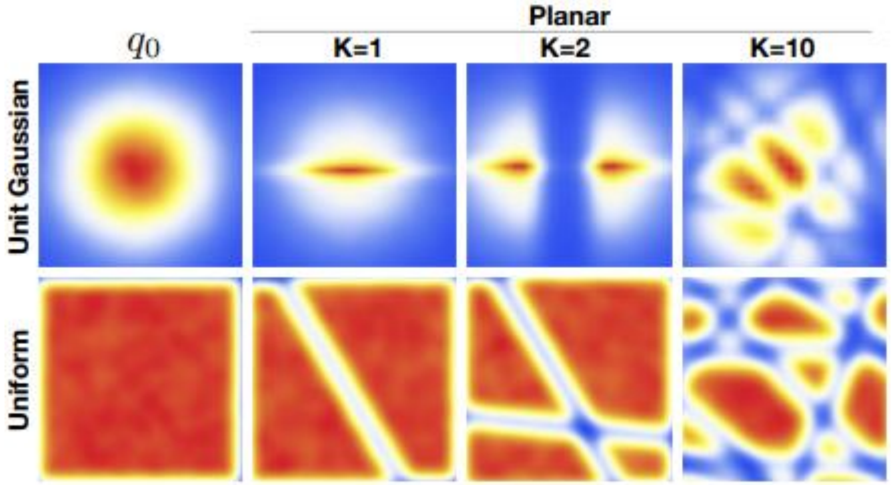


Density Estimation using Real NVP. Dinh, Sohl-Dickstein, Bengio (2017)

Background: Normalizing Flows

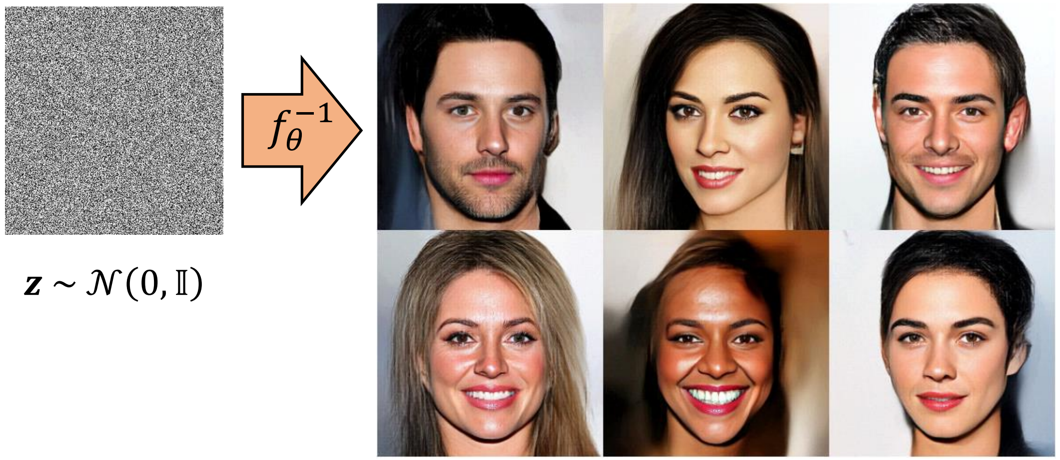
- Why should you care about NF?
  - Can learn tractable probabilistic density model of data $p(\boldsymbol{x})$.
  - Can fit complex posterior distributions.
  - Useful for density estimation, variational inference, generative models.

Learning flexible variational distributions



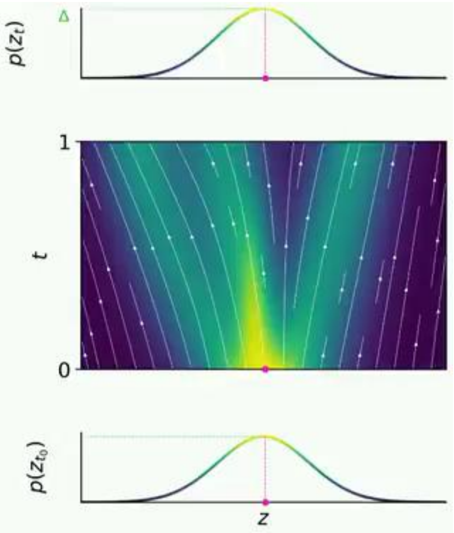
Rezende et al., Variational Inference with Normalizing Flows. ICML 2015

Generating from high-dimensional data distributions



Kingma et al., Glow: Generative Flow with Invertible 1×1 Convolutions. NIPS 2018.

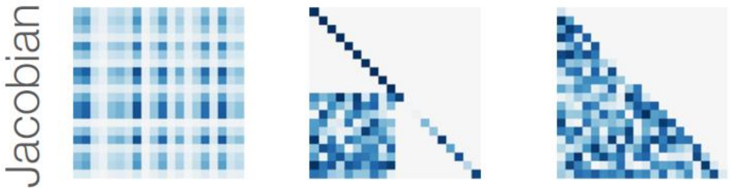
Grathwohl et al., FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models. ICLR 2019



Continuous Normalizing Flows

Normalizing Flow

- $z_K = f_K \circ \cdots \circ f_2 \circ f_1(z_0)$
- Evaluating the density
  - Change of variables
  - $\log p(z_K) = \log p(z_0) - \sum \log \left| \det \left(\frac{\partial z_{i+1}}{\partial z_i} \right) \right|$
- Challenges with NF
  - Inverting transformation is expensive – $\mathcal{O}(D^3)$
  - Jacobian determinant is expensive – $\mathcal{O}(D^3)$
  - Consequently, flow layers must be limited to have structured Jacobians.



(Low rank) (Sparse) (Lower triangular)

Continuous Normalizing Flow

- $\frac{\partial z(t)}{\partial t} = f_\theta(z, t)$
- Evaluating the density
  - Instantaneous change of variables
  - $\frac{\partial \log p(z)}{\partial t} = -\mathrm{tr}\left(\frac{\partial f(z,t)}{\partial z} \right)$
  - $\log p(z) = \log p(z_0) - \int_0^T \mathrm{tr}\left(\frac{\partial f(z,t)}{\partial z} \right) dt$
- Advantages of CNF
  - Inverting is cheap. (solving reversed dynamics)
  - Trace is cheap – $\mathcal{O}(D)$ cost
  - Free-form Jacobian.
- Hutchinson trace estimator
  - $\mathrm{tr}(A) = \mathbb{E}_{v \sim \mathcal{N}(0,1)}[v^\top A v]$
  - $\int_0^T \mathrm{tr}\left(\frac{\partial f_\theta(z,t)}{\partial z} \right) dt = \mathbb{E}_{v \sim \mathcal{N}(0,1)} \left[\int_0^T v^\top \frac{\partial f(z,t)}{\partial z} v \, dt \right]$

VJP is cheaper than Jacobian products.

Continuous Normalizing Flows

Takeaways:
1. Neural ODE greatly reduces computational cost of NF.
2. More flexible flows by removing architectural restriction.

- Continuous Normalizing Flows
  - Tractable probability with $\mathcal{O}(D)$ change of variables
  - Able to scale normalizing flow models
  - Recent score-based training algorithms scale to 1024x1024

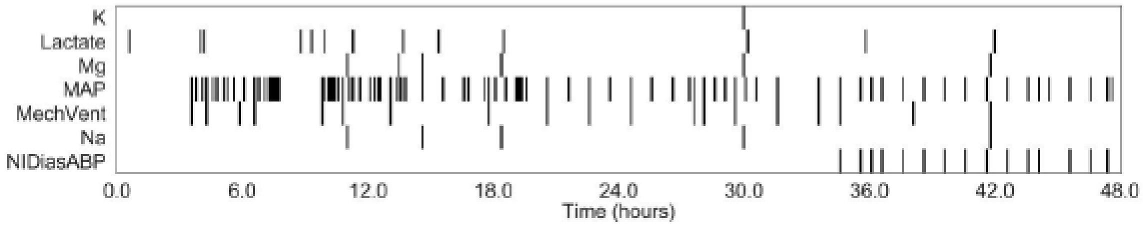
[Song, Sohl-Dickstein, Kingma, Abhishek, Ermon, Poole. Score-Based Generative Modeling through Stochastic Differential Equations, 2020]

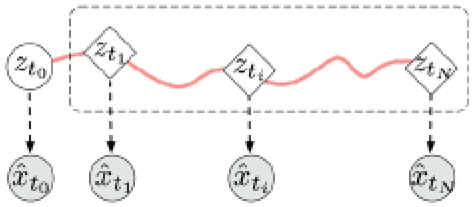
Conditional inpainting and colorization without retraining.

Requires iterative sampling procedure.

Neural ODEs: Applications

- Continuous time-series modeling
  - Can learn from datapoints sampled with irregular intervals.
- Meta-Learning
  - Rajeswaran et al., Meta-Learning with Implicit Gradients. NIPS 2019

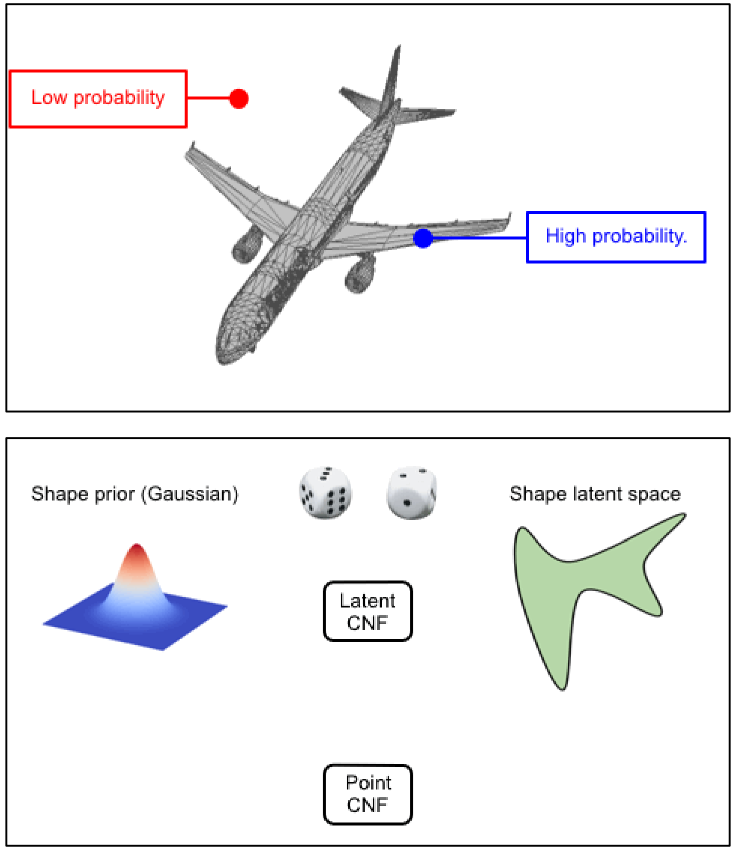


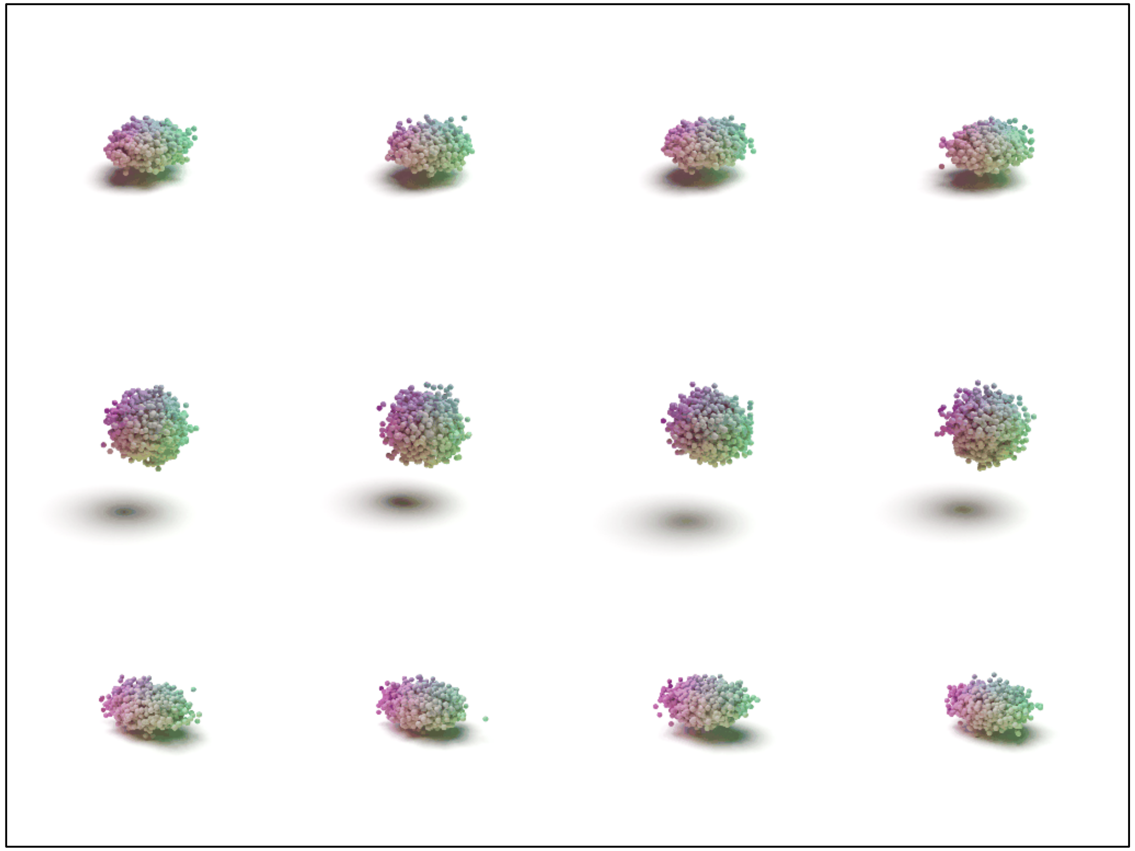


Latent ODEs for Irregularly-Sampled Time Series. Rubanova, Chen, Duvenaud (2020
Neural Controlled Differential Equations for Irregular Time Series.
Kidger, Morrill, Foster, Lyons (2020)
GRU-ODE-Bayes: Continuous modeling of sporadically-observed time series. de Brouwer, Simm, Arany, Moreau. (2020)

Neural ODEs: Applications

- Learning non-intersecting transformations (homeomorphisms) using normalizing flows

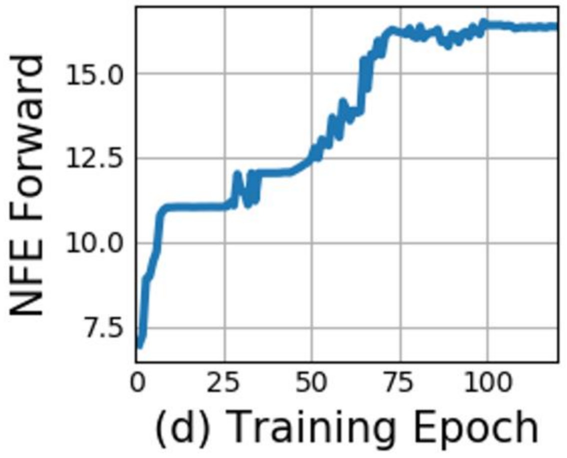




Yang et al., PointFlow: 3D Point Cloud Generation with Continuous Normalizing Flows. ICCV 2019.

Neural ODEs: Summary

- Computational advantages
  - Constant memory cost in backwards
  - Adaptive computation (trade off speed-precision flexibly)
- Modeling advantages
  - Tractable probabilistic generative models
  - Time-series models for irregularly-sampled data
  - Learning smooth homeomorphisms
- Computational disadvantages
  - Speed (future works → regularizing ODEs to be easier to solve)



Neural ODE adapts computation time according to complexity of the dynamics.

When to use DEQs vs. Neural ODEs

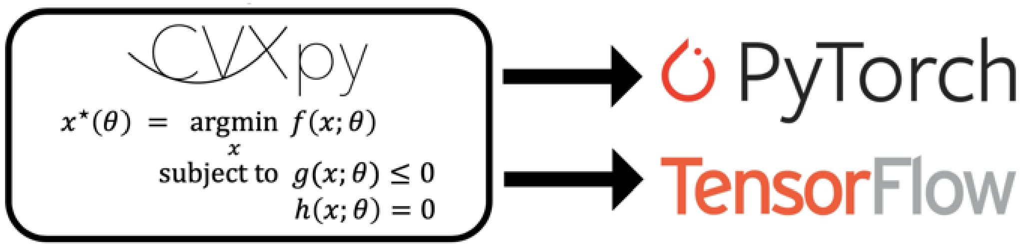
- Use DEQs for:
  - Drop-in implicit replacement for deep models.
  - Supervised learning (CNNs, Transformers)
  - Unsupervised learning (language modeling)
  - Shown success in large-scale tasks.
- Use Neural ODEs for:
  - Continuous-time series modeling
  - Flexible density modeling
  - Modeling homeomorphism

Simulate infinite depth with constant memory footprint

What's beyond

Open problems and future directions

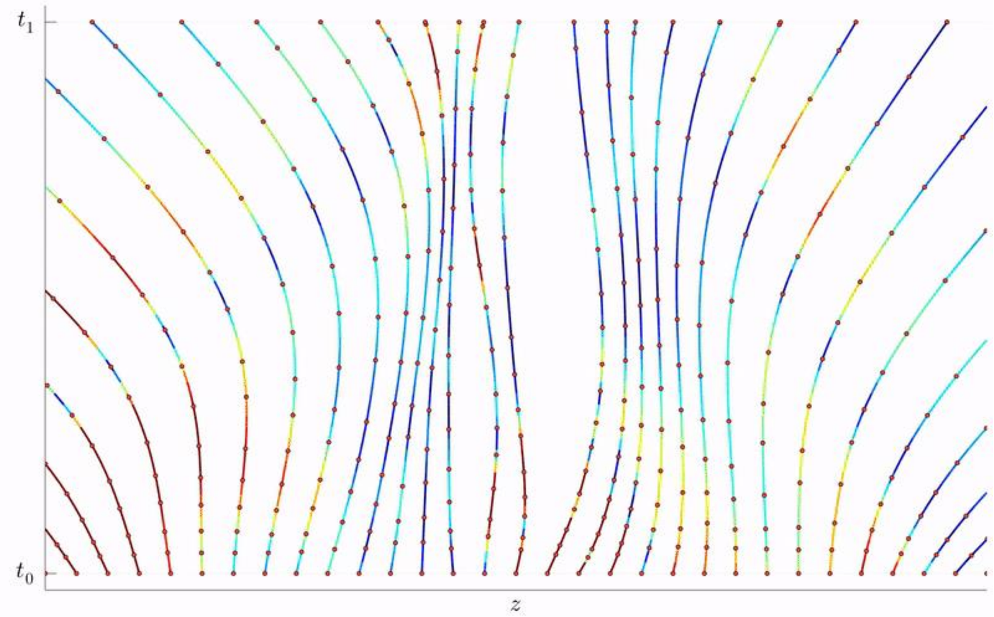
- Regularizing DEQs and Neural ODEs to be faster to solve.
- Re-architecting the models to take advantage of memory efficiency.
- Scaling and application of latent stochastic differential equations.
- Partial differential equation (PDE) solution as a layer
- Differentiable optimization problem as a layer
  - Wang and Kolter et al. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. ArXiv 1905.12149
  - Diamond and Boyd. CVXPY: A Python-embedded modeling language for convex optimization. JMLR Vol. 17 (2016)
  - Agarwal and Kolter et al. Differentiable convex optimization layers. NIPS 2019.

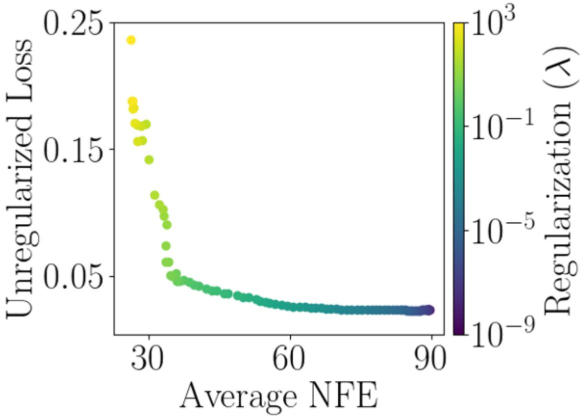


https://github.com/cvxgrp/cvxpy

Regularizing implicit models to be easy to solve

- Controlling flexibility vs. speed of implicit models
  - Trade model quality for number of function evaluations (NFEs).
  - How to regularize implicit models?
- Idea so far for ODEs: regularize the dynamics to have small magnitudes.





Learning Differential Equations that are Easy to Solve. Kelly, Bettencourt, Johnson, Duvenaud. (NeurIPS 2020)

How to Train Your Neural ODE: the World of Jacobian and Kinetic Regularization. Finlay, Jacobsen, Nurbekyan, Oberman. (ICML 2020)

References

- NeurIPS 2020 Tutorial on Deep Implicit Layers: http://implicit-layers-tutorial.org/
- Deep Equilibrium Models
  - Shaojie Bai, Zico Kolter, Vladlen Koltun. Deep Equilibrium Models. NeurIPS 2019.
  - Shaojie Bai, Vladlen Koltun, Zico Kolter. Multiscale Deep Equilibrium Models. NeurIPS 2020.
- Neural ODEs
  - Ricky Chen, Yulia Rubanova, Jesse Bettencourt, David Duvenaud. Neural Ordinary Differential Equations. NeurIPS 2018.