Integrating optimization, constraints, and control within deep networks

Zico Kolter
Carnegie Mellon University and Bosch Center for AI

Work with: **Brandon Amos, Po-wei Wang, Priya Donti, Gaurav Manek,** Bryan Wilder, Akshay Agarwal, Shane Barratt, Steven Diamond, Stephen Boyd

Who's afraid of convex loss functions?

Who is afraid of non-convex loss functions?

Yann LeCun
The Courant Institute of Mathematical Sciences
New York University

Are we really free of convex constraints?

?

Things like constraints, robustness, interpretability, data efficiency, etc…, really do seem to matter

Outline

Structured implicit layers in deep models

Incorporating optimization as a layer

Differentiable SAT solving

Learning stable dynamical systems

Final thoughts

Outline

Structured implicit layers in deep models

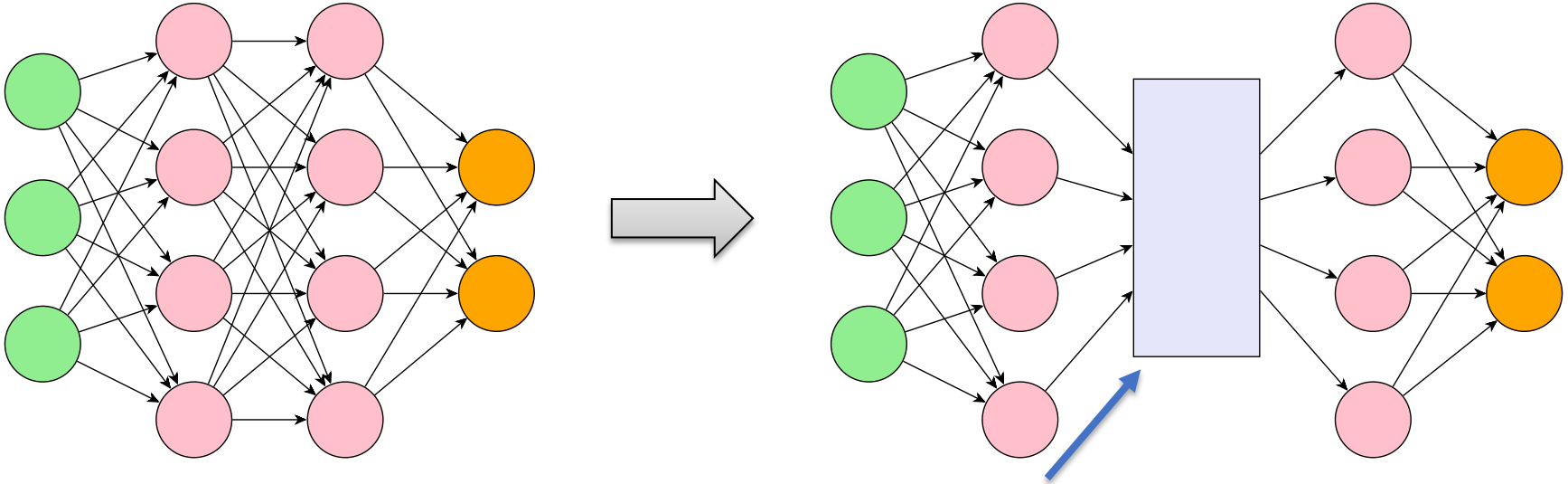
Incorporating optimization as a layer

Differentiable SAT solving

Learning stable dynamical systems

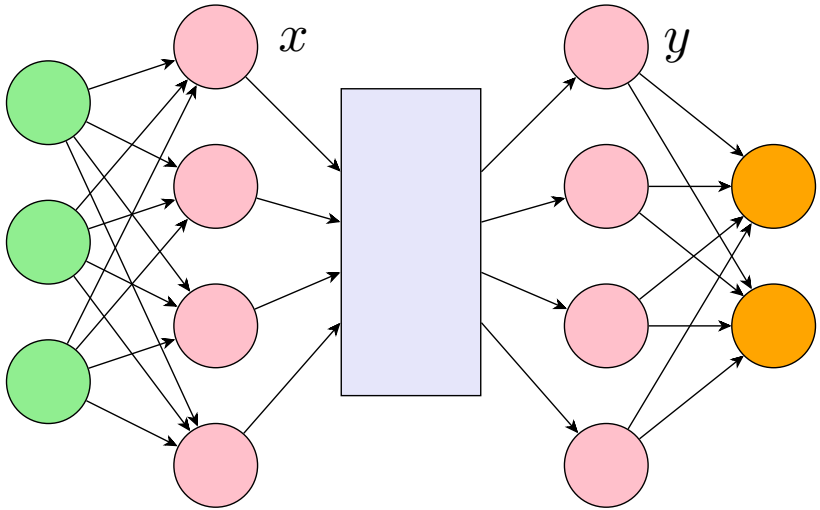
Final thoughts

The move to structured models



Optimizer [Amos and Kolter, ICML 2017]
(MPC) Controller [Amos et al., NeurIPS 2018]
Physics engine [Peres et al., NeurIPS 2018]
Game solver [Ling et al., IJCAI 2018]
(Smoothed) SAT Solver [Wang et al., ICML 2019]

The nature of structured layers



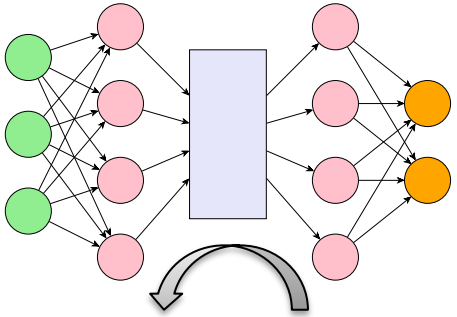
Optimizer
(MPC) Controller
Physics engine
Game solver
(Smoothed) SAT Solver

$$\text{Find } y \text{ such that } f(x, y; \theta) = 0$$

(Implicit-form layer)

Backprop through an implicit layer?

How do we compute relevant Jacobians, i.e., $\frac{\partial y}{\partial (\cdot)}$?



$$f(x, y(\cdot)) = 0$$

$$\frac{\partial f\big(x, y(\cdot)\big)}{\partial (\cdot)} = 0$$

$$\frac{\partial f(x,y)}{\partial (\cdot)} + \frac{\partial f(x,y)}{\partial y} \cdot \frac{\partial y}{\partial (\cdot)} = 0$$

$$\frac{\partial y}{\partial (\cdot)} = -\left(\frac{\partial f(x,y)}{\partial y}\right)^{-1} \frac{\partial f(x,y)}{\partial (\cdot)}$$

…Implicit differentiation, goes back many decades (centuries?)

Important note: “Unrolling” solutions?

Important point: we are *not* advocating to “unroll” the solution procedure of the implicit function, then backprop through that

- Can work, but it’s memory intensive, creates long graphs

Instead, find the exact solution, and backprop (analytically) through it

- Much more memory efficient
- Often effectively “free” after computing forward pass

Outline

Structured implicit layers in deep models

Incorporating optimization as a layer

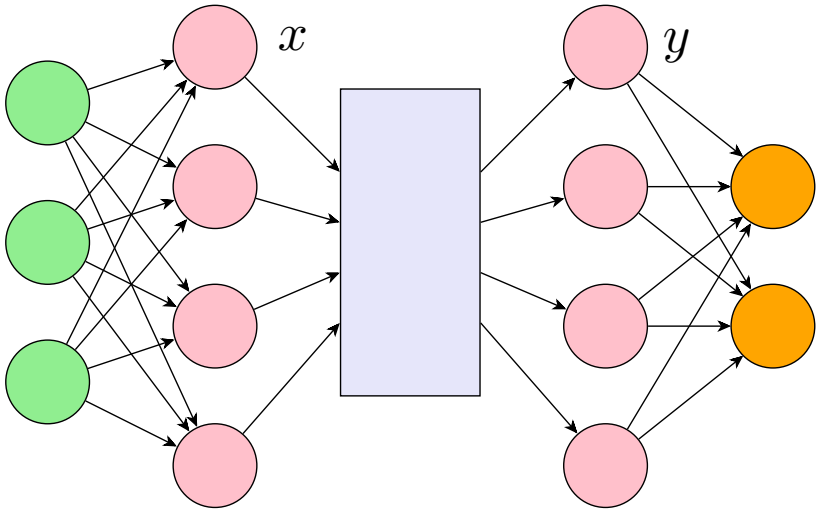
Differentiable SAT solving

Learning stable dynamical systems

Final thoughts

[Agarwal, Amos, Barratt, Boyd, Diamond, Kolter, “Differentiable Convex Optimization Layers”, NeurIPS 2019]

Convex optimization as a layer



$$y = \underset{z \in C(x)}{\operatorname{argmin}} f(z, x)$$

We can consider very general forms of (convex) optimization as the inner layer

Allows for very expressive functions/constraints

Differentiating through convex optimization

Consider the cone problem (here c, A, b all functions of layer input x):

$$\underset{z}{\text{minimize}} \;\; c^T z\,, \qquad \text{subject to } Az - b \in \mathcal{K}$$

Solving this problem equivalent to finding a zero of a certain operator e.g.,

$$\big((Q - I)\Pi + I\big)u = 0, Q = \begin{bmatrix} 0 & A^T & c \\ -A & 0 & b \\ -c^T & -b^T & 0 \end{bmatrix}, \Pi = \text{Proj}_{\mathbb{R}^n \times \mathcal{K}^\star \times \mathbb{R}_+}$$

We can differentiate this fixed point to compute the needed derivatives of $z^\star$ with respects to c, A, b, x

The problem with cone programs

While cone programs are extremely general, it is cumbersome to put problems into standard form

For traditional optimization problems, packages such as cvxpy have revolutionized the ease with which we can write and solve medium-scale convex optimization problems

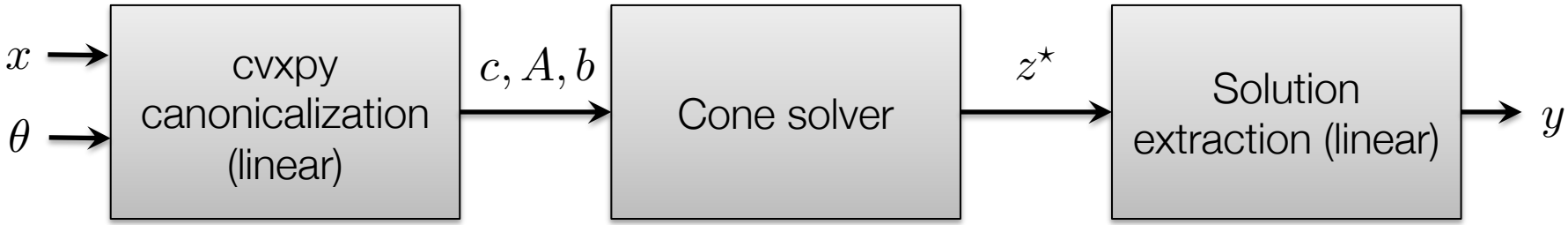
$$\begin{array}{ll} \underset{x}{\text{minimize}} & \|Ax - b\|_1 \\ \text{subject to} & x \geq 0 \end{array}$$

```
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)
```

Our work: differentiable cvxpy

In collaboration with convex optimization group at Stanford, we have developed a differentiable version of cvxpy

Basic process:



All elements of the process can be differentiated through

PyTorch and Tensorflow interfaces

(Perhaps most importantly), easily integrate cvxpy problems as layers within automatic differentiation toolkits such as PyTorch and Tensorflow

```
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()
```

https://github.com/cvxgrp/cvxpylayers

Simple example: data poisoning

Fit ML algorithm to data by convex problem

$$\theta^\star(x,y) = \operatorname*{argmin}_\theta \frac{1}{m}\sum_{i=1}^{m} \ell(\theta^T x_i, y_i) + \lambda\|\theta\|$$

Adversary perturbs the data points within some ball to increase loss on test set $\{x_i', y_i'\}$

$$\max_{\|\delta_{1:m}\|\leq\epsilon} \sum_{i=1}^{m'} \ell(\theta^\star(x+\delta, y)^T x\_i', y_i')$$

Solve via gradient descent over δ

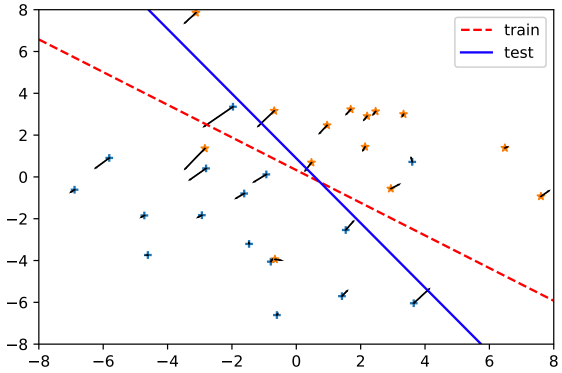


Figure 1: Gradients (black lines) of the logistic test loss with respect to the training data.

Outline

Structured implicit layers in deep models

Incorporating optimization as a layer

Differentiable SAT solving

Learning stable dynamical systems

Final thoughts

[Wang, Donti, Wilder, Kolter, “SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver”, ICML 2019]

Combinatorial optimization?

What about discrete optimization problems, e.g. (MAX)SAT solvers, (existence of) contacts, mixed integer programs?

Inherently non-differentiable (and “simple” relaxations like linear programs are *also* non-differentiable)

Our argument: higher-order relaxations (e.g., semidefinite relaxations) are *perfect* fit for differentiable approximations to combinatorial problems

Also: [Djolonga and Krause, 2017; Tschiatschek, Sahin, Krause 2018]

MAXSAT Problem

MAXSAT is the optimization variant of SAT solving (find assignment that maximizing the number of satisfied clauses)

$$\underset{x \in \{-1,+1\}^d}{\text{maximize}} \sum_{i=1}^{m} 1 \left\{ \sum_{j=1}^{n} s_{ij} x_j > 0 \right\}$$

Where $s_{ij} \in \{-1, 0, +1\}$ denotes sign of ith variable in jth clause

Semidefinite relaxation (Goemans-Williamson, 1995), $X \approx xx^T$

$$\underset{X \succcurlyeq 0,\ \mathrm{diag}(X)=1}{\text{minimize}} \quad \langle S^T S, X \rangle$$

Fast solutions to MAXSAT SDP approximation

Efficiently solve via low-rank factorization $X = V^T V$, $V \in \mathbb{R}^{k \times n}$,$\|v_i\| = 1$ (a.k.a. Burer-Monteiro method), and block coordinate descent updates

$$v_i := -\text{normalize}(V S^T s_i - \|s_i\|_2^2 v_i)$$

For $k > (2n)^{\frac{1}{2}}$, (non-convex) iterates are guaranteed to converge to global optima of SDP [Wang et al., 2018; Erdogdu et al., 2018]

The fixed point of the iteration above also provides an implicit function definition of solution (can backprop efficiently with a similar CD method)

SATNet: MAXSAT SDP as a layer

MAXSAT as a layer: fix some (relaxed) inputs v_I into the SDP solver, use coordinate descent to determine the remaining outputs $v_\mathcal{O}$

Randomized rounding or smoothed probabilities as outputs

Code available: http://github.com/locuslab/SATNet

Illustration: learning parity from data

Parity problem is surprisingly hard for most deep networks to learn [Shalev-Swartz et al., 2017]

Chained (recurrent) SATNet-based network learns parity functions for up to length 40 strings from 10K examples

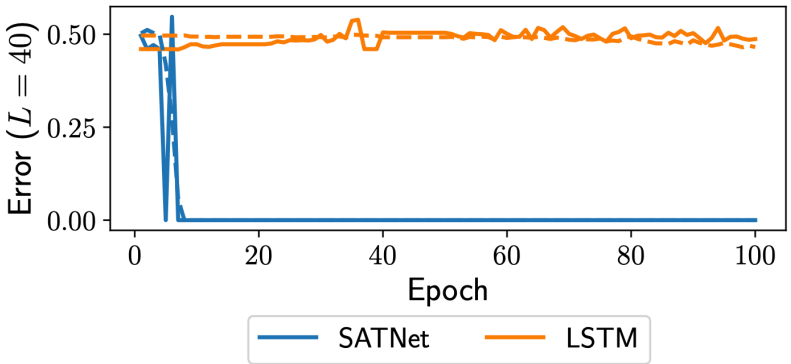
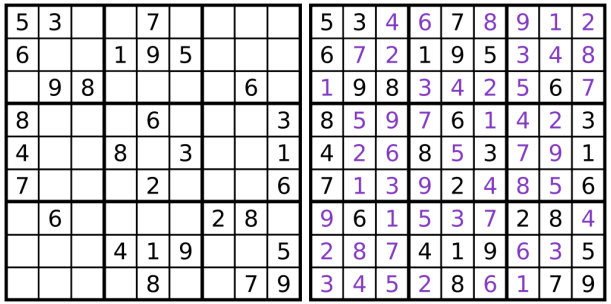


Illustration: Learning Sudoku



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Learning 9x9 Sudoku only from examples

Single SATNet layer on one-hot-encoded input puzzle; free parameters are S matrix of clauses, randomly initialized

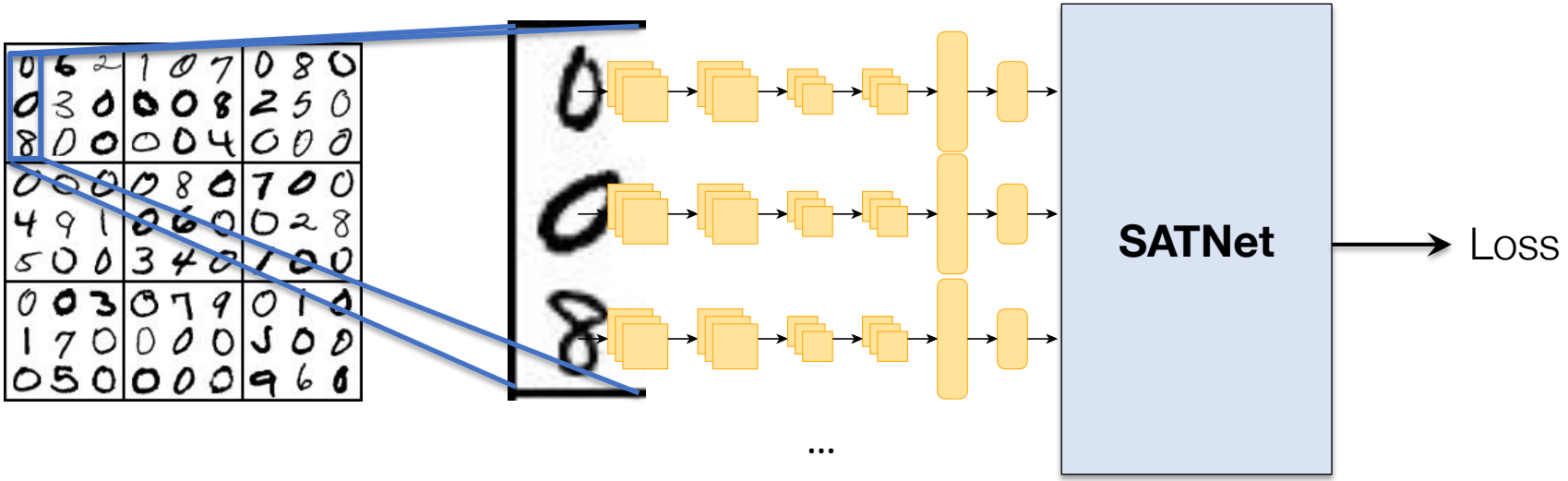
Model	Train	Test
ConvNet	72.6%	0.04%
ConvNetMask	91.4%	15.1%
SATNet (ours)	99.8%	**98.3%**

Standard Sudoku

Model	Train	Test
ConvNet	0%	0%
ConvNetMask	0.01%	0%
SATNet (ours)	99.7%	**98.3%**

Permuted Sudoku

Illustration: MNIST Sudoku



Model	Train	Test
ConvNet	0.31%	0%
ConvNetMask	89%	0.1%
SATNet (ours)	93.6%	**63.2%**

*Note that getting an example "correct" requires correct Sodoku solution *and* predicting 81 MNIST test digits correctly

Outline

Structured implicit layers in deep models

Incorporating optimization as a layer

Differentiable SAT solving

Learning stable dynamical systems

Final thoughts

[Manek, Kolter, “Learning Stable Deep Dynamics Models”, NeurIPS 2019]

Learning stable dynamical systems

One more example of the power of incorporating (even simple) constraints into neural networks: learning *stable* dynamics

Consider learning continuous-time dynamical system $f: \mathbb{R}^n \to \mathbb{R}^n$

$$\dot{x} = f(x)$$

such that f is globally exponentially stable around $x = 0$

A hard (non-convex) problem even for linear dynamics…

Enforcing stability via constrained layers

We know f is globally exponentially stable iff there exists a Lyapunov function $V: \mathbb{R}^n \to \mathbb{R}$ (satisfying some conditions) such that for $x \in \mathbb{R}^n$

$$\nabla V(x)^T f(x) \leq -\alpha V(x)$$

Let's just *enforce* this condition via projection, given "nominal" dynamics $g: \mathbb{R}^n \to \mathbb{R}^n$ defined by a deep network, define dynamics f as

$$\begin{aligned} f(x) &= \text{Proj}(g(x), \{y: \nabla V(x)^T y \leq -\alpha V(x)\}) \\ &= g(x) - \text{ReLU}\Big(\nabla V(x)^T g(x) + \alpha V(x)\Big) \nabla V(x) / \|\nabla V(x)\|_2^2 \end{aligned}$$

Key insight: we can fit both g *and* V from data, using AD libraries

Some additional details

Can't choose any V, needs to be be unimodal with minimum at $x = 0$, be differentiable everywhere

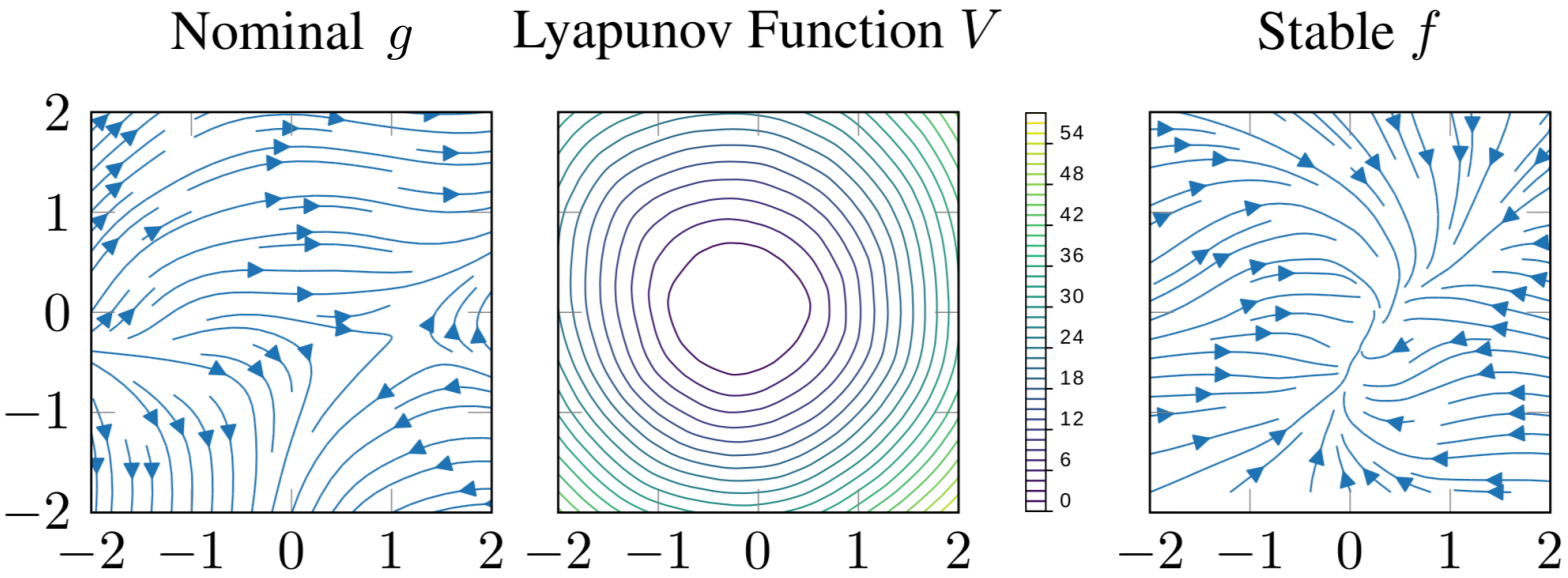
We enforce this by representing V using an input convex neural network (ICNN), shifted to zero, using smooth activation functions, e.g.

$$h(x) = \sigma(W_2\sigma(W_1 x + b) + b_2)$$
$$V(x) = \sigma\big(h(x) - h(0)\big) + \epsilon\|x\|_2^2$$

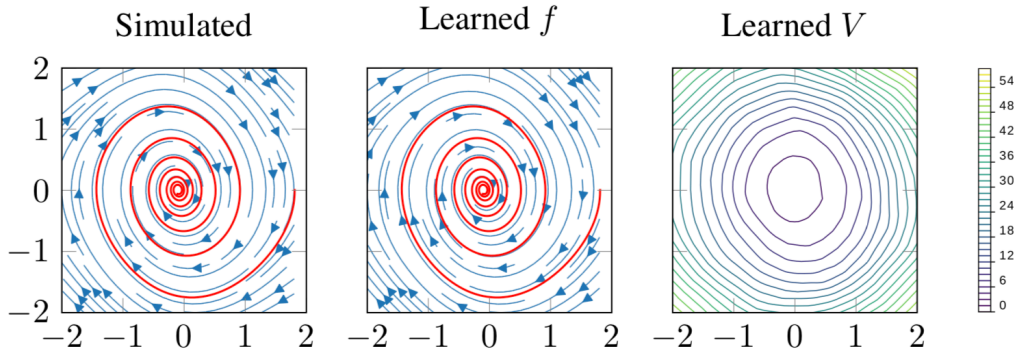
with σ smooth, monotonic, convex, and $W_2 \geq 0$

Example: random networks

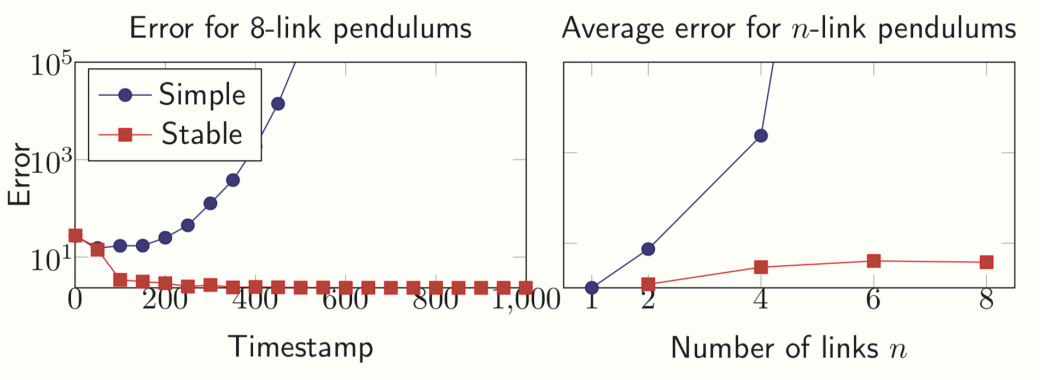
We can randomly initialize g and V networks, and observe that this leads to stable nonlinear dynamics



Example: multi-link pendulum

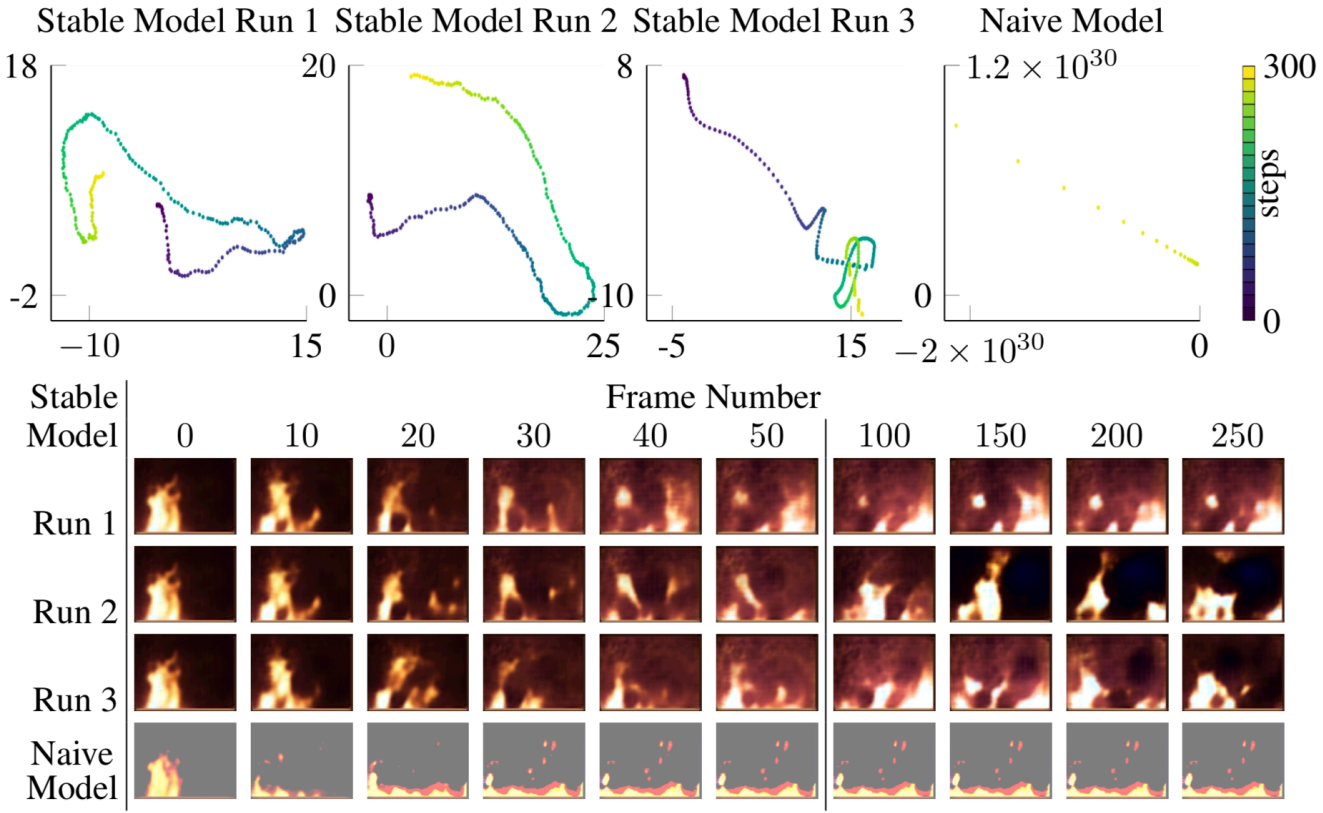


Learned dynamics and Lyapunov function of simple one-link pendulum



Simulation error for naïve neural network and our stable approach on n-link pendulum

Stable VAE system for video textures



Outline

Structured implicit layers in deep models

Incorporating optimization as a layer

Differentiable SAT solving

Learning stable dynamical systems

Final thoughts

The zen of abandoning (outer loop) convexity

Many of the constraints/models we want to enforce in learning, control, and other settings can be directly incorporated into the “implicit bias” of deep networks

The tools of convex optimization, semidefinite relaxations, Lyapunov stability, etc have immense applications not just in understanding “traditional” deep learning, but in extending the nature of deep models themselves

Papers and code available at:
http://zicokolter.com