Thesis Defense

Differentiable Optimization-Based Modeling for Machine Learning

Brandon Amos • Carnegie Mellon University

Thesis Committee:
J. Zico Kolter, Chair
Barnabás Póczos
Jeff Schneider
Vladlen Koltun (Intel Labs)

The source code behind all of my work is free and publicly available:
http://github.com/bamos/thesis

My Ph.D. Contributions

Secondary Contribution

[CMU 2016] OpenFace

[ICML 2016] Collapsed Variational Inference for SPNs

[ICML 2017] Input Convex Neural Networks

[ICML 2017] OptNet

[NeurIPS 2017] Task-Based Model Learning

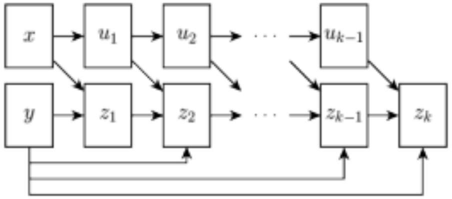
[ICLR 2018] Learning Awareness Models

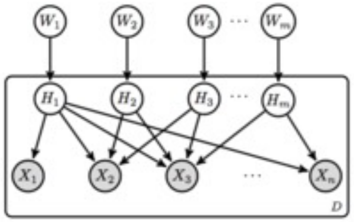
[NeurIPS 2018] Imperfect-Information Game Solving

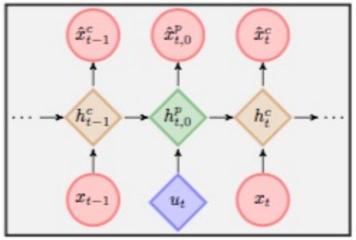
[NeurIPS 2018] Differentiable MPC

The Limited Multi-Label Projection Layer

Differentiable `cvxpy` Optimization Layers







This Talk

Secondary Contribution

[CMU 2016] OpenFace

[ICML 2016] Collapsed Variational Inference for SPNs

[ICML 2017] Input Convex Neural Networks

[ICML 2017] OptNet

[NeurIPS 2017] Task-Based Model Learning

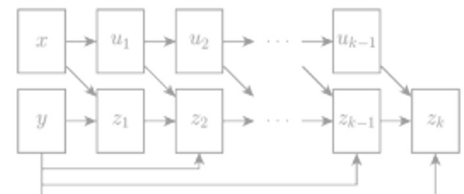
[ICLR 2018] Learning Awareness Models

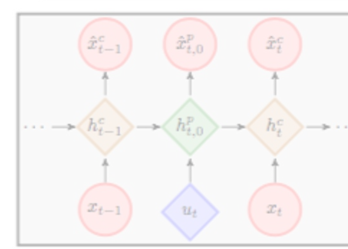
[NeurIPS 2018] Imperfect-Information Game Solving

[NeurIPS 2018] Differentiable MPC

The Limited Multi-Label Projection Layer

Differentiable `cvxpy` Optimization Layers

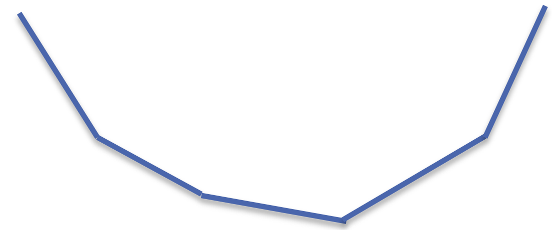




Input Convex Neural Networks

A quick glimpse

Input Convex Neural Networks (ICNNS)



Definition Scalar-valued network $f(x, y; \theta)$ such that f is **convex** in y for all values of x (note that these networks are still **not convex** in $\theta = \{W_i, b_i\}$)

We can efficiently **optimize over some inputs** to the network **given other inputs**

Efficiently captures dependencies in the output space for prediction

It turns out, we don't need very many restrictions on the network to achieve this property

How to achieve input convexity?

Most networks can be "trivially" modified to **guarantee input convexity**

Consider a simple **feedforward ReLU network**:

$$z_{i+1} = \max\{0, W_i z_i + b_i\}, \qquad i = 1, \ldots, k$$
$$f(y; \theta) = z_{k+1},\ z_1 = y$$

Proposition. f is convex in y provided that the W_i are non-negative for $i > 1$

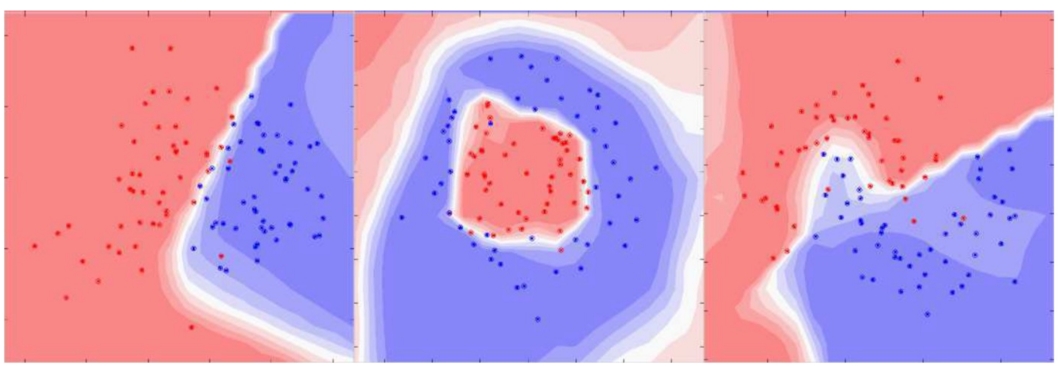
More generally, any activation function that is convex and non-decreasing also has this property.

Is convexity restrictive?

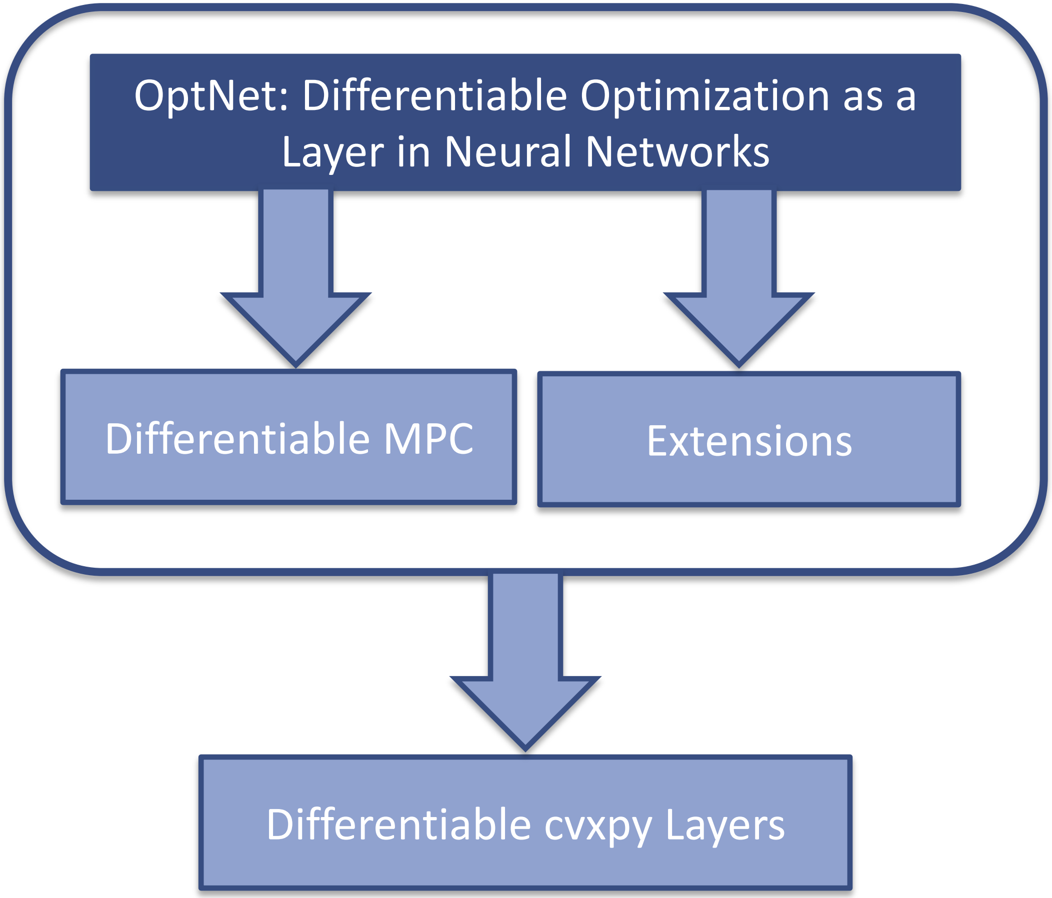
Yes (**by definition**, the functions are restricted to be convex), but **not that bad** in practice

Proposition. ICNNs trivially subsume any feedforward network $\tilde{f}(x)$ with the network $f(x,y) = \left(y - \tilde{f}(x)\right)^2$

More complex convex portion **adds additional structure** over y, which can still be "easily" optimized over



Overview for the remainder of this talk



Today's Machine Learning Systems

A lot of data → Model → Predictions → Loss

Current Primitive Operations: Linear maps, convolutions, activation functions, random sampling, simple projections (e.g. onto the simplex or Birkhoff polytope)

How can the modeling part be improved?

Black-box neural networks don't work everywhere and when they fail, task-specific domain knowledge can provide useful modeling priors

My work mostly focuses on ways to **use optimization to inject domain knowledge into the modeling process**

Optimization and Machine Learning

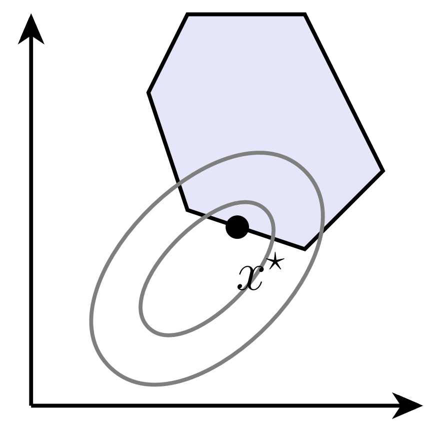
Non-convex optimization is thriving in machine learning for parameter optimization and architecture search. **This is not what this talk covers.**

In this talk, we argue that optimization is also a useful operation for inference and control.

We consider optimization as another potential **layer**, to be **composed with others**

Why? Optimization is an extremely powerful paradigm for decision-making.

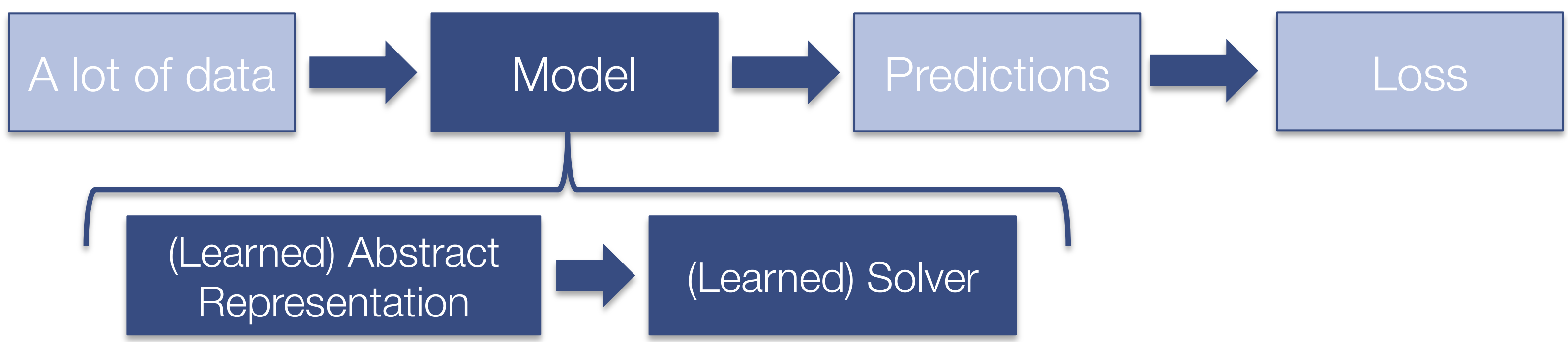
- Applications in **finance** (Markowitz portfolio optimization), **machine learning** (support vector machines), **control** (linear-quadratic model predictive control), **geometry** (projections onto polyhedra)



Why is optimization a useful primitive operation in learning systems?

We have incomplete domain knowledge about what we want to model

- Fill in parts of the optimization problem that we know
- Use data to learn the parts that we don't



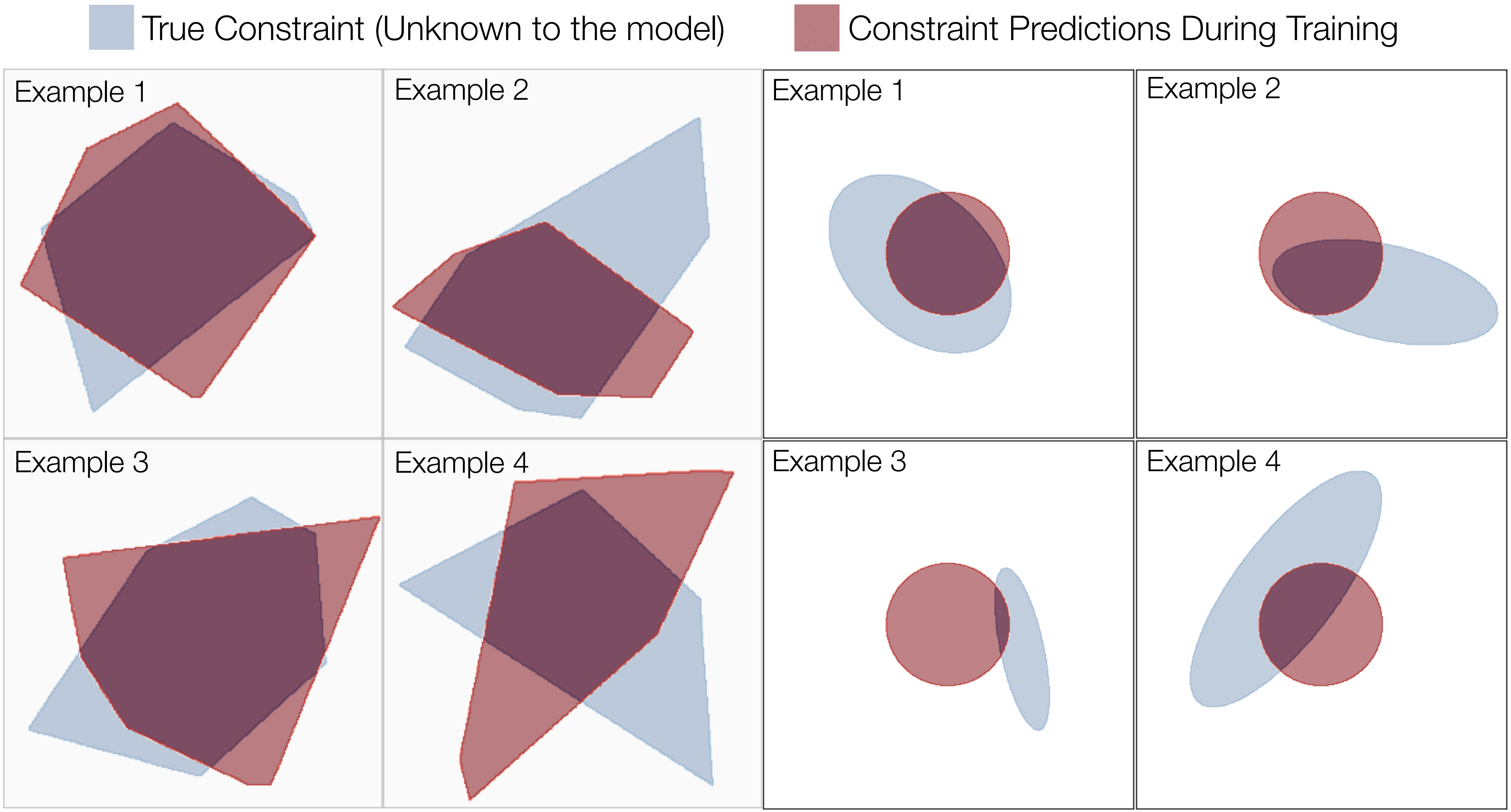
Also subsumes many standard layers (ReLU, sigmoid, softmax)

- We will show this later

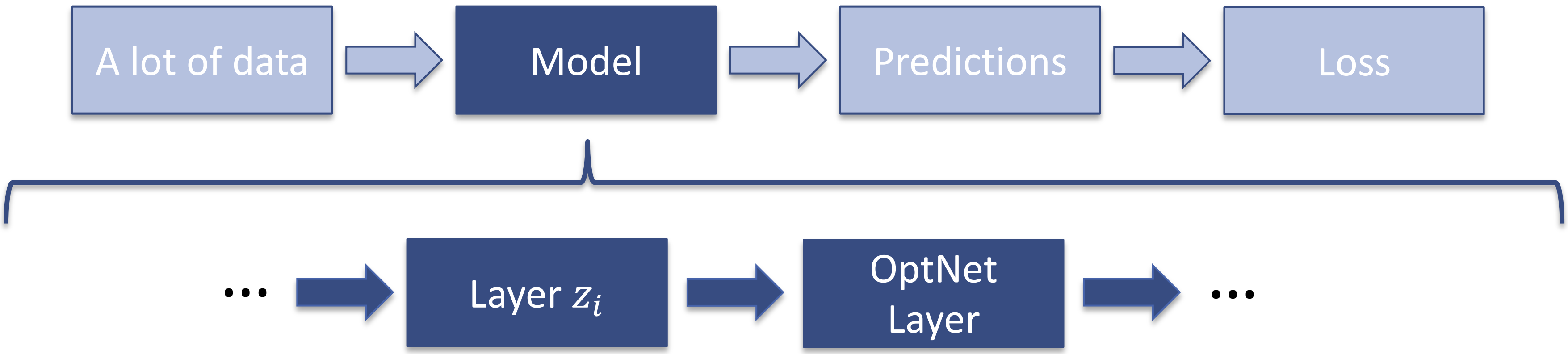
Convex optimization viewpoint of standard layers

ReLU	$y = \max\{0, x\}$	$$\begin{aligned} y^\star = \ & \underset{y}{\operatorname{argmin}} \ \|y - x\|_2^2 \\ & \text{subject to} \ \ y \geq 0 \end{aligned}$$
Sigmoid	$y = \dfrac{1}{1 + e^{-x}}$	$$\begin{aligned} y^\star = \ & \underset{y}{\operatorname{argmin}} \ -y^T x - H_b(y) \\ & \text{subject to} \ \ 0 \leq y \leq 1 \end{aligned}$$
Softmax	$y_j = \dfrac{e^{x_j}}{\sum e^{x_k}}$	$$\begin{aligned} y^\star = \ & \underset{y}{\operatorname{argmin}} \ -y^T x - H(y) \\ & \text{subject to} \ \ 0 \leq y \leq 1 \\ & \qquad\qquad\quad 1^T y = 1 \end{aligned}$$

OptNet Application: Modeling Constraints



The OptNet Layer



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

Parameters/Submodules : Q, q, A, b, G, h

The matrix $Q(z_i)$ depends on the previous layer z_i

Differentiating a quadratic argmin

Consider the optimization problem:

$$z^\star = \operatorname*{argmin}_z \frac{1}{2} z^T Q z + q^T z$$
$$\text{subject to } \; Az = b, Gz \le h$$

From convex optimization theory, the **Karush-Kuhn-Tucker conditions** provide necessary and sufficient equations for optimality.

stationarity	$Qz^\star + q + A^T \nu^\star + G^T \lambda^\star = 0$
primal feasibility	$Az^\star - b = 0$
complementary slackness	$D(\lambda^\star)(Gz^\star - h) = 0$

To obtain $\partial z^\star / \partial \theta$ **implicitly differentiate the KKT conditions.**

This also works for any convex optimization problem (not just QPs)

Implicitly differentiating the KKT conditions

Implicitly differentiate them (using differentials here):

$$\mathrm{d}Q z^* + q\,\mathrm{d}z + \mathrm{d}q + \mathrm{d}A^T \nu^* + A^T \mathrm{d}\nu + \mathrm{d}G^T \lambda^* + G^T \mathrm{d}\lambda = 0$$
$$\mathrm{d}A z^* + A\,\mathrm{d}z - \mathrm{d}b = 0$$
$$D(Gz^* - h)\mathrm{d}\lambda + D(\lambda^\star)(\mathrm{d}G z^* + G\,\mathrm{d}z - \mathrm{d}h) = 0$$

Fill in desired differentials, form a linear system, solve for unknowns

If done naively, takes **many** linear system solves
If done correctly, just requires a single solve to compute **all** gradients

A Simple Application: Sudoku

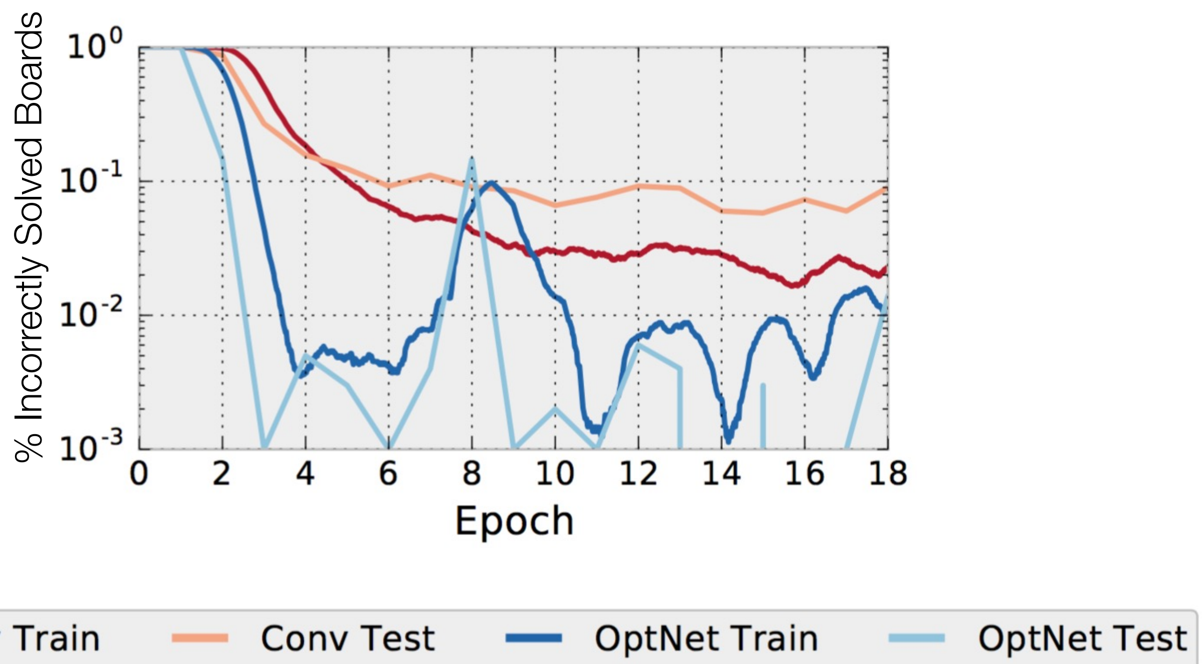
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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

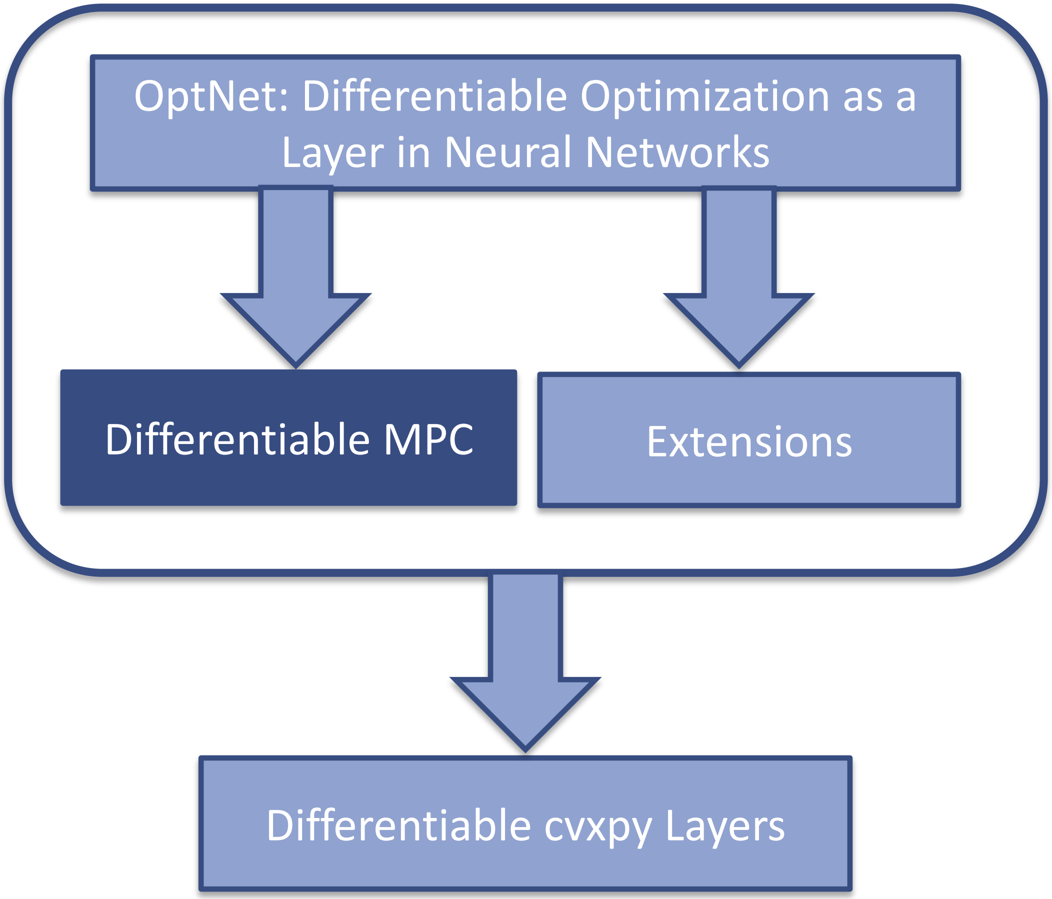
OptNet Learns Sudoku

$$x^\star = \operatorname*{argmin}_x \ \mathrm{dist}(x, p)$$
$$\text{subject to } \ Ax = b$$

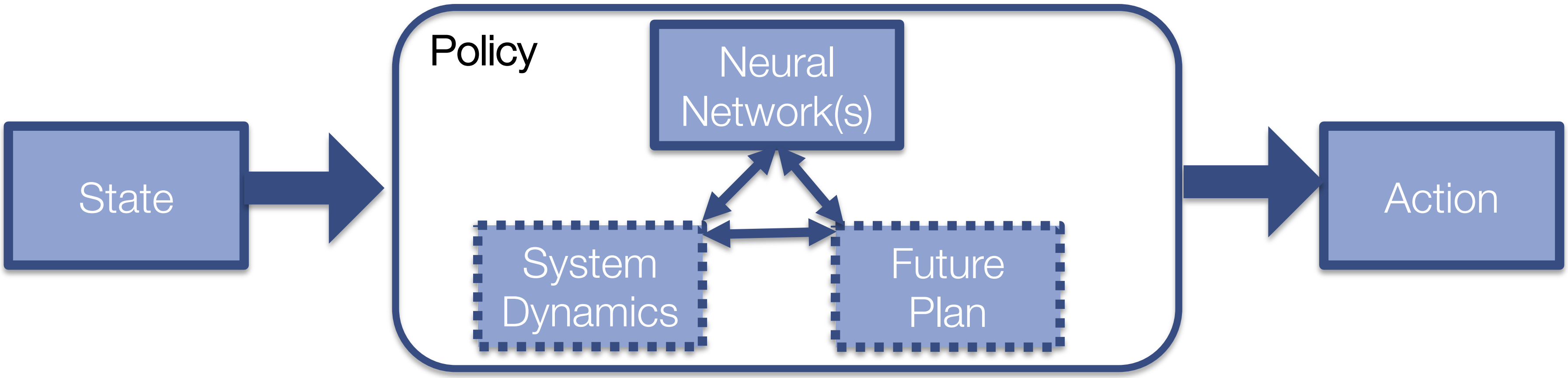
The OptNet layer exactly learns the mini-Sudoku constraints from data!
Baseline: A deep convolutional feed-forward network



Overview for the remainder of this talk



Should RL policies have a system dynamics model or not?



Model-free RL

More general, doesn't make as many assumptions about the world

Rife with poor data efficiency and learning stability issues

Model-based RL (or control)

A useful prior on the world if it lies within your set of assumptions

Combining model-based and model-free RL

Recently there has been a lot of interest in model-based priors for model-free reinforcement learning:

> Among others: Dyna-Q (Sutton, 1990), GPS (Levine and Koltun, 2013), Imagination-Augmented Agents (Weber et al., 2017), Value Iteration Networks (Tamar et al., 2016), TreeQN (Farquhar et al., 2017)

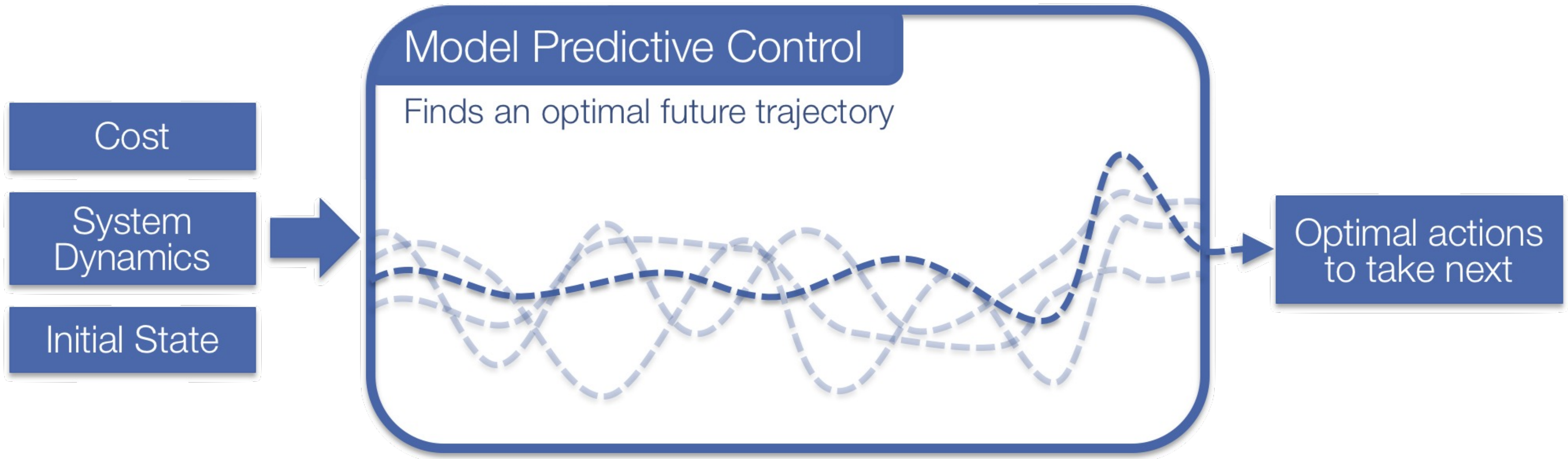
These typically involve:

1. **Using an RNN:** Efficient but not as expressive and general as MPC/iLQR
2. **Unrolling an LQR or gradient-based solver:** Expressive/general but inefficient

Our approach: Differentiable Model-Predictive Control

- **Explicitly** solves a control problem

Our Approach: Model Predictive Control



Our Approach: Model Predictive Control

Traditionally viewed as a pure **planning problem** given known (potentially non-convex) **cost** and **dynamics**:

$$
\begin{aligned}
\tau^\star_{1:T} = \operatorname*{argmin}_{\tau_{1:T}} \; & \sum_t \boxed{C_\theta(\tau_t)} \text{ Cost} \\
\text{subject to } \; & x_1 = x_{init} \\
& x_{t+1} = \boxed{f_\theta(\tau_t)} \text{ Dynamics} \\
& \underline{u} \le u \le \overline{u}
\end{aligned}
$$

where $\tau_t = \{x_t, u_t\}$

Execute u_1 in the environment, observe the next observation, and repeat.

Cost and dynamics explicitly represented and learned.

Model Predictive Control with SQP

- The standard way of solving MPC is to use **sequential quadratic programming (SQP),** using LQR in most cases
- **Form approximations** to the cost and dynamics around the current iterate
- Repeat until a **fixed point** is reached and **differentiate through it**

… → QP Iterate i → …

QP Iterate i

$$
\begin{aligned}
\tau_{1:T}^{i} = \operatorname*{argmin}_{\tau_{1:T}} \; & \sum_t \tilde{C}_\theta^i(\tau_t) \\
\text{subject to} \; & x_1 = x_{init} \\
& x_{t+1} = \tilde{f}_\theta^i(\tau_t) \\
& \underline{u} \le u \le \overline{u}
\end{aligned}
$$

LQR, KKT Systems, and Differentiation

Solving **LQR** with **dynamic Riccati recursion** efficiently solves the **KKT system**

$$
\overbrace{\begin{array}{c}
\begin{array}{cccc} \tau_t & \lambda_t & \tau_{t+1} & \lambda_{t+1} \end{array} \\
\left[\begin{array}{c|cc|cc|c}
\ddots & & & & & \\ \hline
 & C_t & F_t^\top & & & \\
 & F_t & & [-I \quad 0] & & \\ \hline
 & & \begin{bmatrix} -I \\ 0 \end{bmatrix} & C_{t+1} & F_{t+1}^\top & \\
 & & & F_{t+1} & & \\ \hline
 & & & & & \ddots
\end{array}\right]
\end{array}}^{K}
\begin{bmatrix} \vdots \\ \tau_t^\star \\ \lambda_t^\star \\ \tau_{t+1}^\star \\ \lambda_{t+1}^\star \\ \vdots \end{bmatrix}
= -
\begin{bmatrix} \vdots \\ c_t \\ f_t \\ c_{t+1} \\ f_{t+1} \\ \vdots \end{bmatrix}
$$

Backwards Pass: Use the OptNet approach from [Amos and Kolter, 2017] to implicitly **differentiate** the LQR KKT conditions:

$$
\frac{\partial \ell}{\partial C_t} = \frac{1}{2}\left(d_{\tau_t}^\star \otimes \tau_t^\star + \tau_t^\star \otimes d_{\tau_t}^\star\right) \qquad \frac{\partial \ell}{\partial c_t} = d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial x_{\mathrm{init}}} = d_{\lambda_0}^\star
$$

$$
\frac{\partial \ell}{\partial F_t} = d_{\lambda_{t+1}}^\star \otimes \tau_t^\star + \lambda_{t+1}^\star \otimes d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial f_t} = d_{\lambda_t}^\star
$$

where $K \begin{bmatrix} \vdots \\ d_{\tau_t}^\star \\ d_{\lambda_t}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \nabla_{\tau_t^\star} \ell \\ 0 \\ \vdots \end{bmatrix}$

Just another LQR problem!

LQR, KKT Systems, and Differentiation

Solving **LQR** with **dynamic Riccati recursion** efficiently solves the **KKT system**

$$
\overbrace{
\begin{array}{cccccc}
 & \tau_t & \lambda_t & \tau_{t+1} & \lambda_{t+1} & \\
\end{array}
}^{K}
$$

$$
\begin{bmatrix}
\ddots & & & & & \\
 & C_t & F_t^\top & & & \\
 & F_t & & [-I \quad 0] & & \\
 & & \begin{bmatrix} -I \\ 0 \end{bmatrix} & C_{t+1} & F_{t+1}^\top & \\
 & & & F_{t+1} & & \\
 & & & & & \ddots
\end{bmatrix}
\begin{bmatrix} \vdots \\ \tau_t^\star \\ \lambda_t^\star \\ \tau_{t+1}^\star \\ \lambda_{t+1}^\star \\ \vdots \end{bmatrix}
= -
\begin{bmatrix} \vdots \\ c_t \\ f_t \\ c_{t+1} \\ f_{t+1} \\ \vdots \end{bmatrix}
$$

Backwards Pass: Use the OptNet approach from [Amos and Kolter, 2017] to implicitly **differentiate** the LQR KKT conditions:

$$
\frac{\partial \ell}{\partial C_t} = \frac{1}{2}\left(d_{\tau_t}^\star \otimes \tau_t^\star + \tau_t^\star \otimes d_{\tau_t}^\star\right) \qquad \frac{\partial \ell}{\partial c_t} = d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial x_{\mathrm{init}}} = d_{\lambda_0}^\star
$$

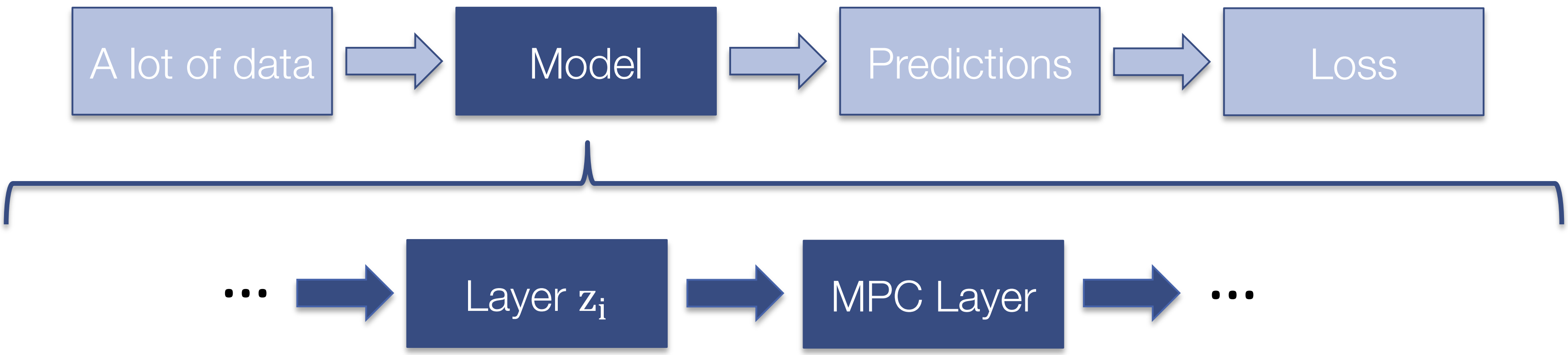
$$
\frac{\partial \ell}{\partial F_t} = d_{\lambda_{t+1}}^\star \otimes \tau_t^\star + \lambda_{t+1}^\star \otimes d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial f_t} = d_{\lambda_t}^\star
$$

where $K \begin{bmatrix} \vdots \\ d_{\tau_t}^\star \\ d_{\lambda_t}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \nabla_{\tau_t^\star} \ell \\ 0 \\ \vdots \end{bmatrix}$

Just another LQR problem!

A Differentiable MPC Module

We can differentiate through (non-convex) MPC with a single (convex) LQR solve by differentiating the SQP fixed point



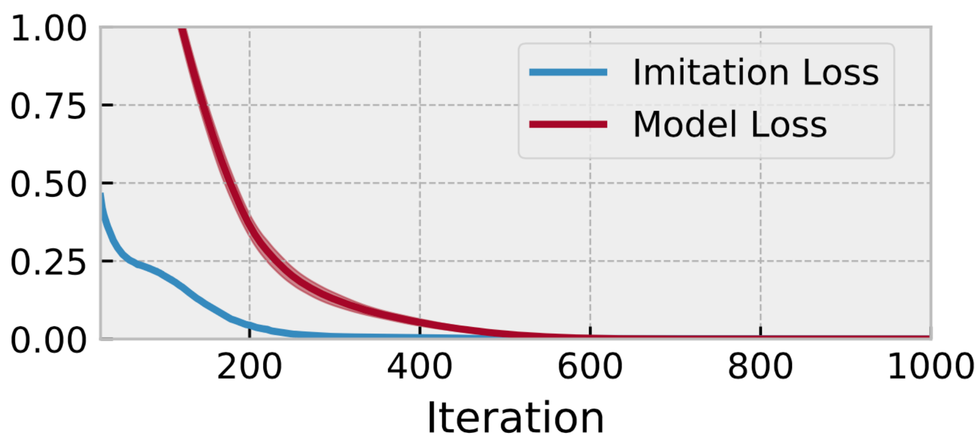
What can we do with this now?

Replace neural network policies in model-free algorithms with MPC policies, and also **replace the unrolled controllers** in other settings (hindsight plan, universal planning networks)

The cost can also be learned! No longer have to hard-code in a known value.

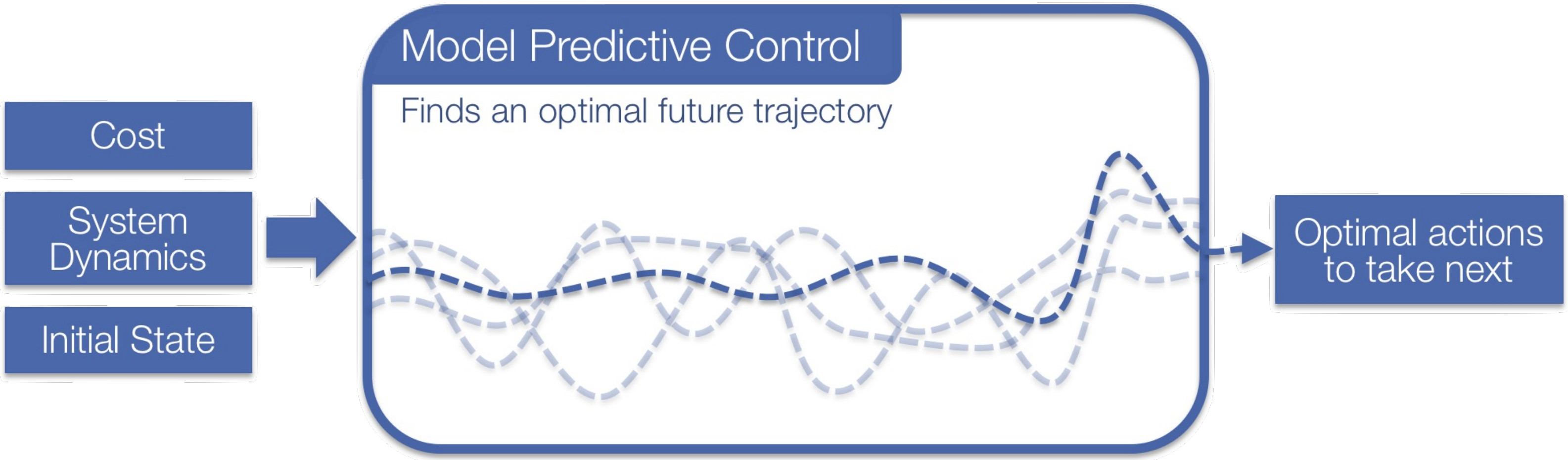
Imitation learning with a linear model

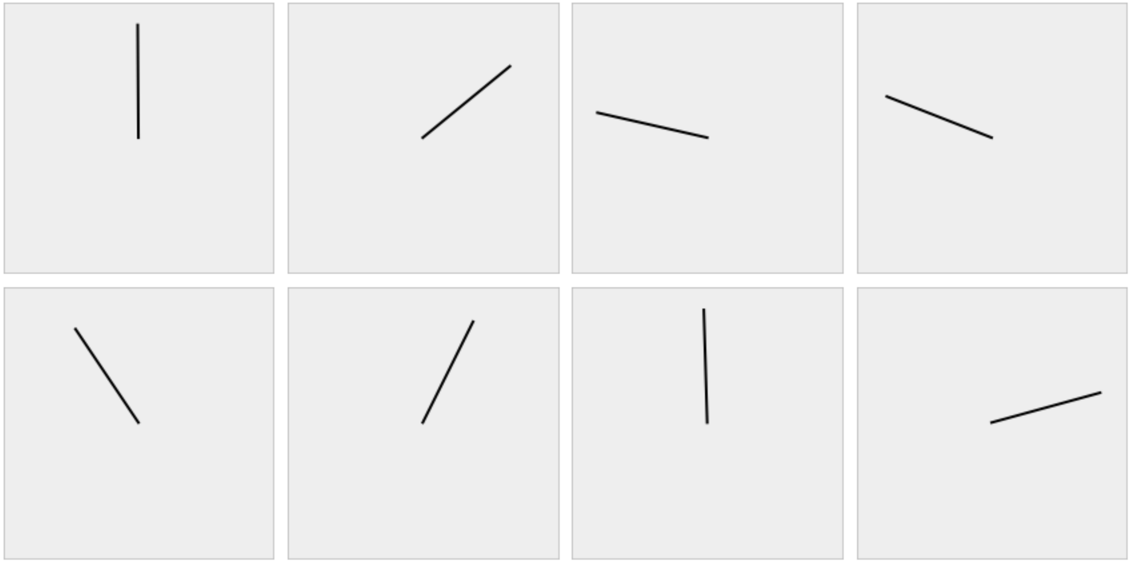
Linear dynamics: $f(x_t, u_t) = Ax_t + Bu_t$
Parameters: $\theta = \{A, B\}$
Trajectory: $\tau_\theta(x_{\text{init}})$ obtained by MPC
Given known θ and sample trajectories, learn $\hat{\theta}$
Trajectory (Training) Loss: $\text{MSE}(\tau_\theta(x_{\text{init}}), \tau_{\hat{\theta}}(x_{\text{init}}))$
Model Loss: $\text{MSE}(\theta, \hat{\theta})$



Not guaranteed to converge, but a good sanity check that it does in small cases.

Simple Pendulum Control

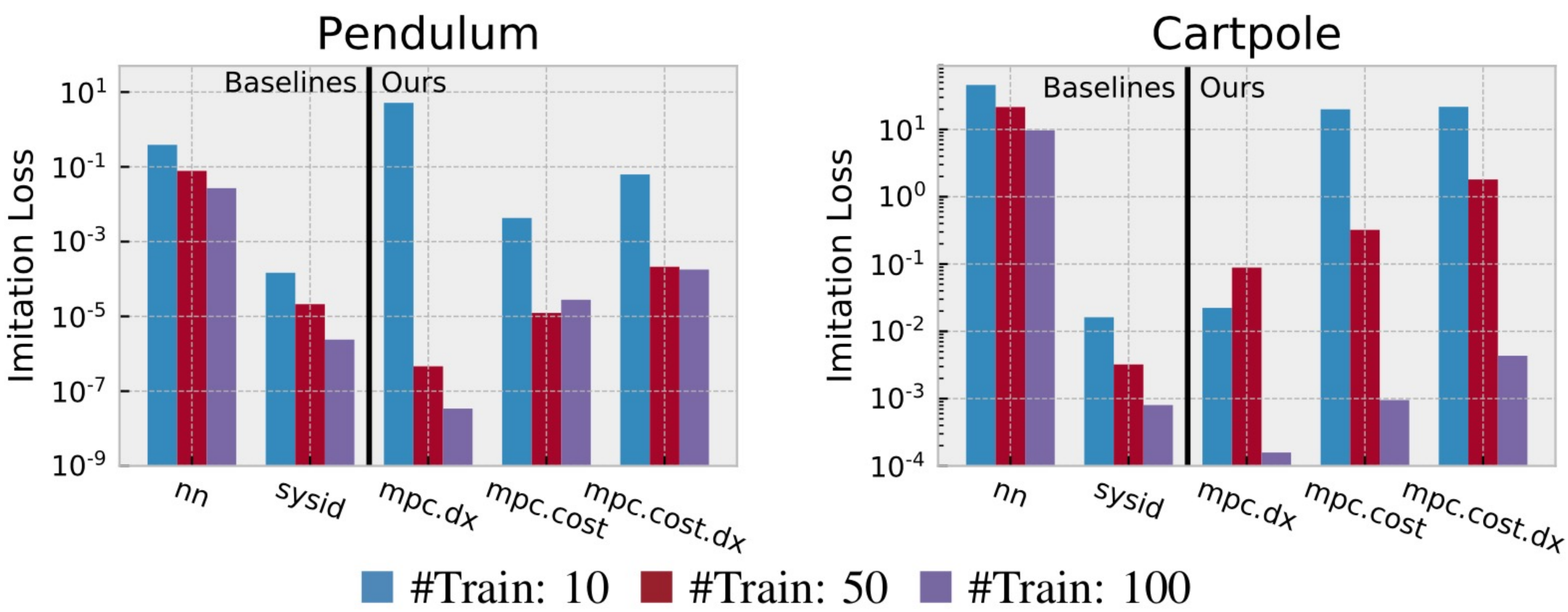




Imitation learning with the pendulum/cartpole

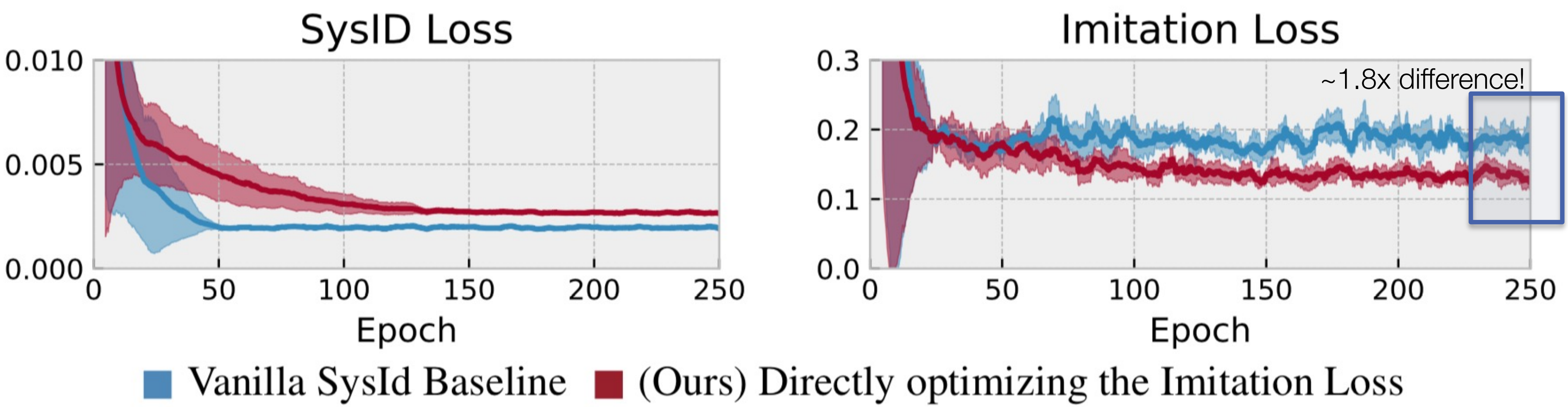
Again optimizes the imitation loss with respect to the controller's parameters

Using **only action trajectories** we can recover the true parameters



Optimizing the task loss is often better than SysID in the unrealizable case

True System: Pendulum environment with noise (damping and a wind force)
Approximate Model: Pendulum without the noise terms



A PyTorch MPC Solver

https://locuslab.github.io/mpc.pytorch

mpc.pytorch

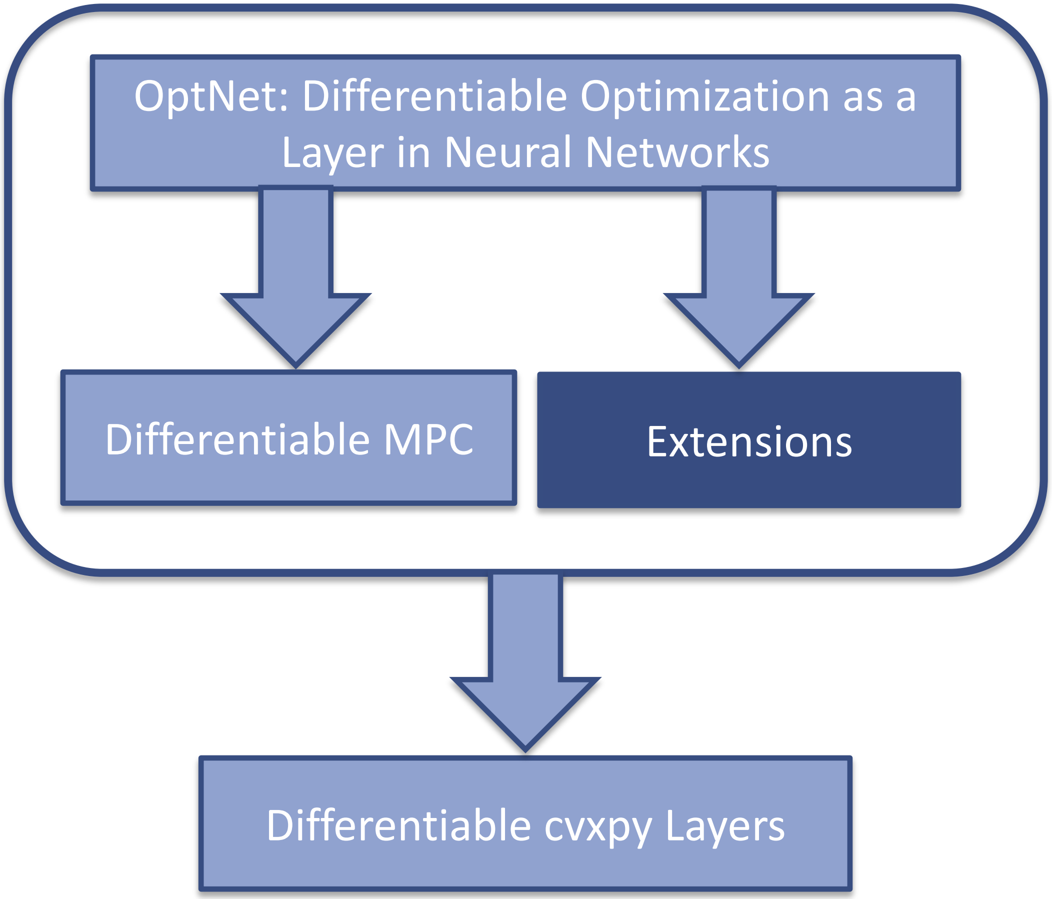
A fast and differentiable model predictive control (MPC) solver for PyTorch. Crafted by Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J. Zico Kolter. For more context and details, see our ICML 2017 paper on OptNet and our (forthcoming) NIPS 2018 paper on differentiable MPC.

View On GitHub

Control is important!

Optimal control is a widespread field that involve finding an optimal sequence of future actions to take in a system or environment. This is the most useful in domains when you can analytically model your system and can easily define a cost to optimize over your system. This project focuses on solving model predictive control (MPC) with the box-DDP heuristic. MPC is a powerhouse in many real-world domains ranging from short-time horizon robot control tasks to long-time horizon control of chemical processing plants. More recently, the reinforcement learning community, strife with poor sample-complexity and instability issues in model-free learning, has been actively searching for useful model-based applications and priors.

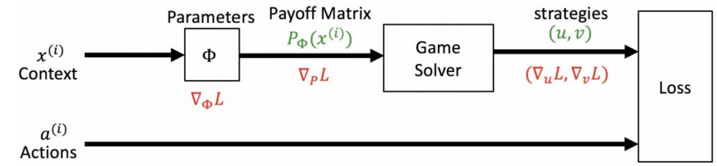
Overview for the remainder of this talk



Extensions

Section 2 and Section 8 of my thesis document contain a more complete set of references

Game Theory [Ling, Fang, and Kolter; IJCAI 2017]: Distinguished Paper Award

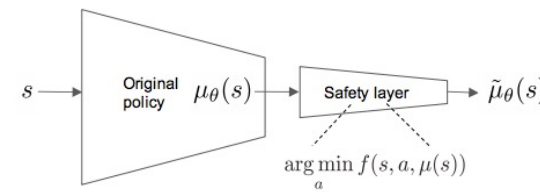


Stochastic optimization and end-to-end learning
[Donti, Amos, and Kolter; NeurIPS 2017]

$$\begin{bmatrix} 0 \\ 0 \\ a \\ \sigma \\ \zeta \end{bmatrix} - \begin{bmatrix} \mathcal{M} & -\mathcal{J}_e & -\mathcal{J}_c & -\mathcal{J}_f & 0 \\ \mathcal{J}_e & 0 & 0 & 0 & 0 \\ \mathcal{J}_c & 0 & 0 & 0 & 0 \\ \mathcal{J}_f & 0 & 0 & 0 & E \\ 0 & 0 & \mu & -E^T & 0 \end{bmatrix} \begin{bmatrix} v_{t+dt} \\ \lambda_e \\ \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} = \begin{bmatrix} \mathcal{M}v_t + dt f_t \\ 0 \\ c \\ 0 \\ 0 \end{bmatrix}$$

$$\text{subject to} \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix} \geq 0, \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} \geq 0, \begin{bmatrix} a \\ \sigma \\ \zeta \end{bmatrix}^T \begin{bmatrix} \lambda_c \\ \lambda_f \\ \gamma \end{bmatrix} = 0,$$

Reinforcement learning and control
Safety [Dalal et al. 2018], physics-based modeling [Peres et al. NeurIPS 2018], inverse cost and reward learning, multi-agent systems, learnable embeddings



Discrete, combinatorial, and submodular optimization
[Djolonga and Krause 2017, Niculae and Blondel 2017, Mensch and Blondel 2018]

$$\mathbf{y}^* = \min_{x \in \mathcal{B}(G)} \frac{1}{2}\|\mathbf{y} - \mathbf{y}'\|, \text{ where } \mathbf{y}' = \arg\min_{\mathbf{y}} f(\mathbf{y}) + \frac{1}{2}\|\mathbf{y} - \mathbf{z}\|^2.$$

$$\Pi_\Omega(\boldsymbol{x}) := \arg\max_{\boldsymbol{y} \in \Delta^d} \boldsymbol{y}^\mathrm{T}\boldsymbol{x} - \gamma\Omega(\boldsymbol{y}) = \nabla\mathrm{max}_\Omega(\boldsymbol{x})$$

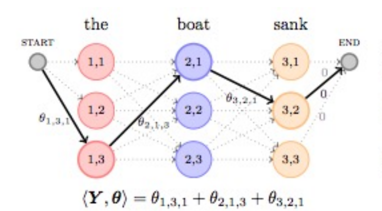
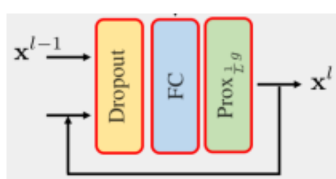
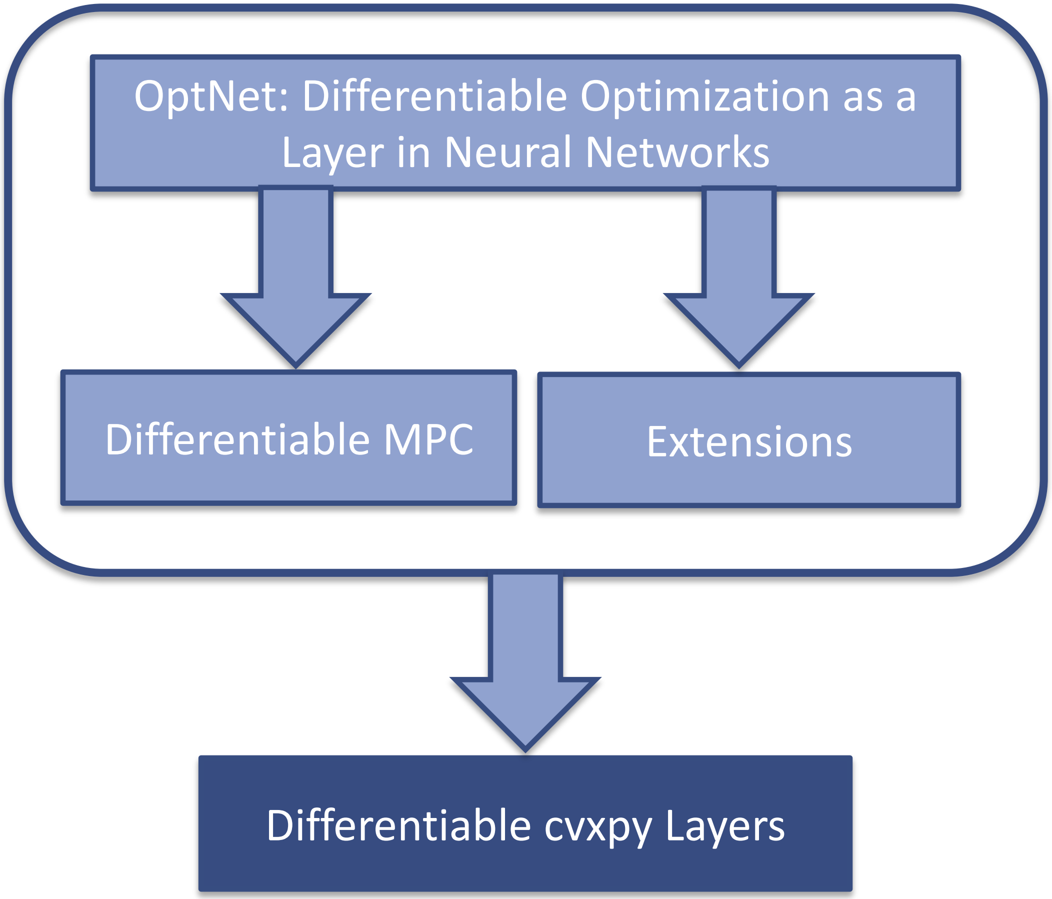


Figure 2. Computational graph of the Viterbi algorithm.

Optimization viewpoints of standard components
[Bibi et al. ICLR 2019]



Overview for the remainder of this talk



Hand-Implementing Optimization Layers is Hard

$$\nabla_Q \ell = \frac{1}{2}(d_x^\star \otimes x^\star + x^\star \otimes d_x^\star) \qquad \nabla_p \ell = d_x^\star$$

$$\nabla_A \ell = d_\lambda^\star \otimes x^\star + \lambda^\star \otimes d_x^\star \qquad \nabla_b \ell = -d_\lambda^\star$$

$$\mathsf{d}Qz^\star + Q\mathsf{d}z + \mathsf{d}q + \mathsf{d}A^T\nu^\star + A^T\mathsf{d}\nu + \mathsf{d}G^T\lambda^\star + G^T\mathsf{d}\lambda = 0$$

$$\mathsf{d}Az^\star + A\mathsf{d}z - \mathsf{d}b = 0$$

$$D(Gz^\star - h)\mathsf{d}\lambda + D(\lambda^\star)(\mathsf{d}Gz^\star + G\mathsf{d}z - \mathsf{d}h) = 0$$

$$\begin{bmatrix} Q & A^\top & \tilde{G}^\top \\ A & 0 & 0 \\ \tilde{G} & 0 & 0 \end{bmatrix} \begin{bmatrix} d_x^\star \\ d_\lambda^\star \\ d_{\tilde{\nu}}^\star \end{bmatrix} = - \begin{bmatrix} \nabla_{x^\star}\ell \\ 0 \\ 0 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^\star)G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix}}_{K} \begin{bmatrix} \mathsf{d}z \\ \mathsf{d}\lambda \\ \mathsf{d}\nu \end{bmatrix} = \begin{bmatrix} -\mathsf{d}Qz^\star - \mathsf{d}q - \mathsf{d}G^T\lambda^\star - \mathsf{d}A^T\nu^\star \\ -D(\lambda^\star)\mathsf{d}Gz^\star + D(\lambda^\star)\mathsf{d}h \\ -\mathsf{d}Az^\star + \mathsf{d}b \end{bmatrix}$$

$$\nabla_Q \ell = \frac{1}{2}(d_z z^T + z d_z^T) \qquad \nabla_q \ell = d_z$$

$$\nabla_A \ell = d_\nu z^T + \nu d_z^T \qquad \nabla_b \ell = -d_\nu$$

$$\nabla_G \ell = D(\lambda^\star)(d_\lambda z^T + \lambda d_z^T) \qquad \nabla_h \ell = -D(\lambda^\star) d_\lambda$$

```
invQ_AT = A.transpose(1, 2).lu_solve(*Q_LU)
A_invQ_AT = torch.bmm(A, invQ_AT)
G_invQ_AT = torch.bmm(G, invQ_AT)

LU_A_invQ_AT = lu_hack(A_invQ_AT)
P_A_invQ_AT, L_A_invQ_AT, U_A_invQ_AT = torch.lu_unpack(*
P_A_invQ_AT = P_A_invQ_AT.type_as(A_invQ_AT)

S_LU_11 = LU_A_invQ_AT[0]
U_A_invQ_AT_inv = (P_A_invQ_AT.bmm(L_A_invQ_AT)
                   ).lu_solve(*LU_A_invQ_AT)
S_LU_21 = G_invQ_AT.bmm(U_A_invQ_AT_inv)
T = G_invQ_AT.transpose(1, 2).lu_solve(*LU_A_invQ_AT)
S_LU_12 = U_A_invQ_AT.bmm(T)
S_LU_22 = torch.zeros(nBatch, nineq, nineq).type_as(Q)
S_LU_data = torch.cat((torch.cat((S_LU_11, S_LU_12), 2),
                       torch.cat((S_LU_21, S_LU_22), 2)),
                      1)
S_LU_pivots[:, :neq] = LU_A_invQ_AT[1]

R -= G_invQ_AT.bmm(T)
```

$$\begin{array}{c} \begin{array}{cccc} \tau_t & \lambda_t & \tau_{t+1} & \lambda_{t+1} \end{array} \\ \left[\begin{array}{c:cc:cc:c} \ddots & & & & & \\ \hdashline & C_t & F_t^\top & & & \\ & F_t & & [-I \quad 0] & & \\ \hdashline & & \begin{bmatrix} -I \\ 0 \end{bmatrix} & C_{t+1} & F_{t+1}^\top & \\ & & & F_{t+1} & & \\ \hdashline & & & & & \ddots \end{array}\right] \end{array} \begin{bmatrix} \vdots \\ \tau_t^\star \\ \lambda_t^\star \\ \tau_{t+1}^\star \\ \lambda_{t+1}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ c_t \\ f_t \\ c_{t+1} \\ f_{t+1} \\ \vdots \end{bmatrix}$$

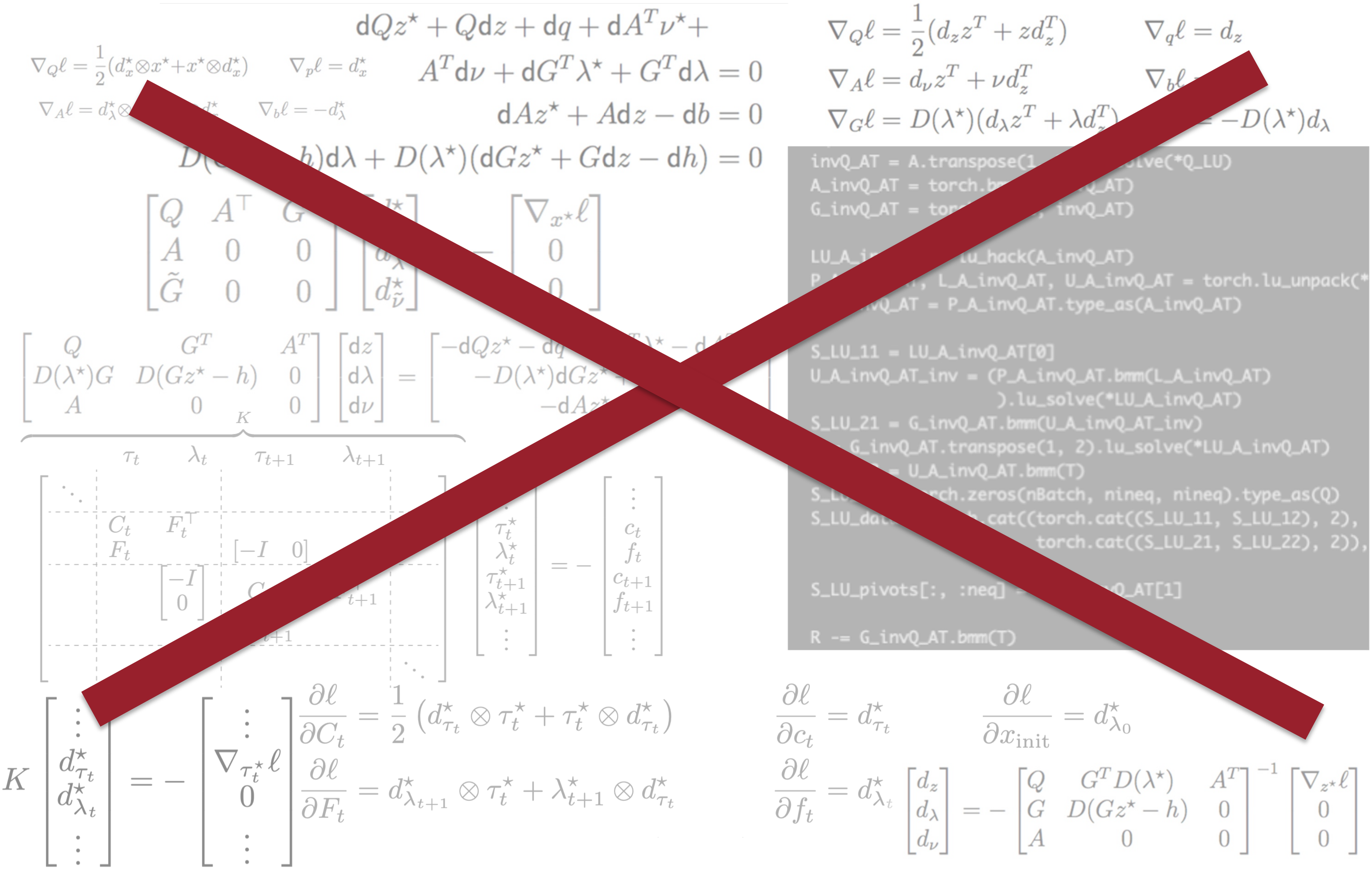
$$K \begin{bmatrix} \vdots \\ d_{\tau_t}^\star \\ d_{\lambda_t}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \nabla_{\tau_t^\star}\ell \\ 0 \\ \vdots \end{bmatrix}$$

$$\frac{\partial \ell}{\partial C_t} = \frac{1}{2}\left(d_{\tau_t}^\star \otimes \tau_t^\star + \tau_t^\star \otimes d_{\tau_t}^\star\right) \qquad \frac{\partial \ell}{\partial c_t} = d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial x_{\text{init}}} = d_{\lambda_0}^\star$$

$$\frac{\partial \ell}{\partial F_t} = d_{\lambda_{t+1}}^\star \otimes \tau_t^\star + \lambda_{t+1}^\star \otimes d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial f_t} = d_{\lambda_t}^\star$$

$$\begin{bmatrix} d_z \\ d_\lambda \\ d_\nu \end{bmatrix} = - \begin{bmatrix} Q & G^T D(\lambda^\star) & A^T \\ G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \nabla_{z^\star}\ell \\ 0 \\ 0 \end{bmatrix}$$

Why should practitioners care?



cvxpy

http://cvxpy.org

(constrained LASSO)

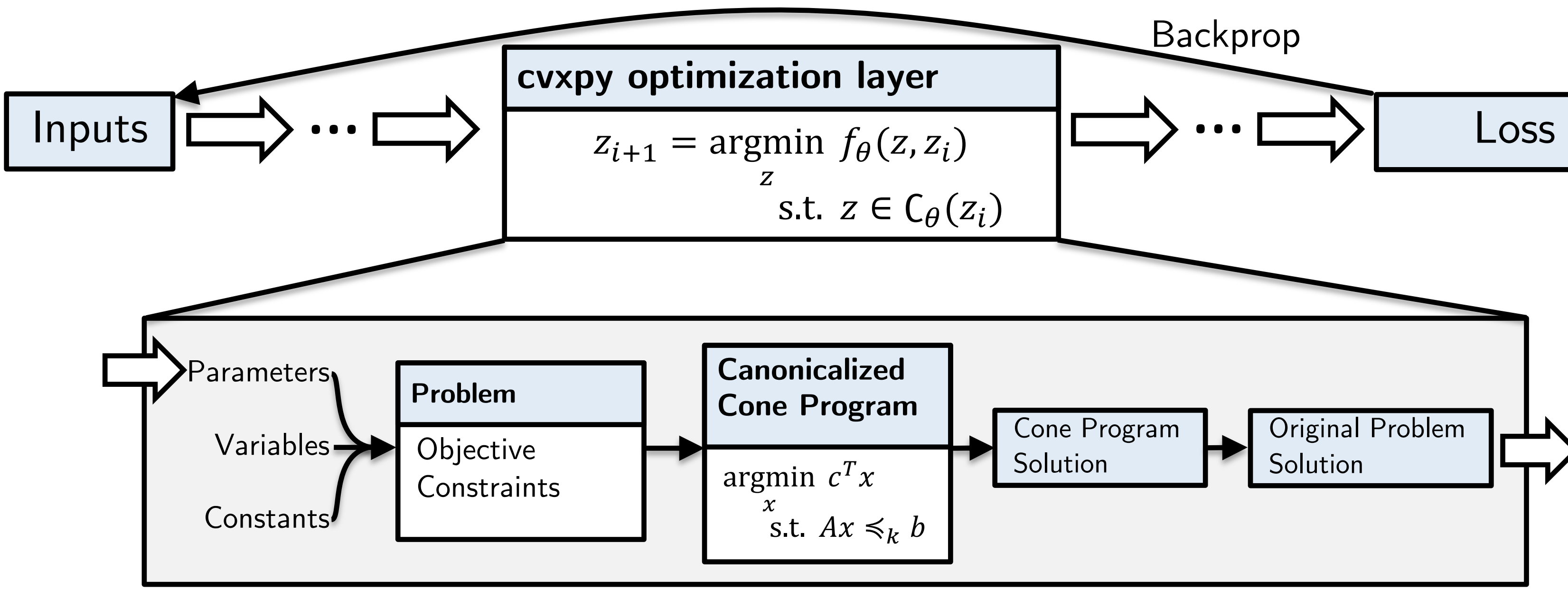
[Diamond2018]

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 + \gamma\|x\|_1 \\ \text{subject to} & \mathbf{1}^T x = 0, \quad \|x\|_\infty \leq 1 \end{array}$$

with variable $x \in \mathbf{R}^n$

```
from cvxpy import *
x = Variable(n)
cost = sum_squares(A*x-b) + gamma*norm(x,1)
obj = Minimize(cost)
constr = [sum_entries(x) == 0, norm(x,"inf") <= 1]
prob = Problem(obj, constr)
opt_val = prob.solve()
solution = x.value
```

A new way of rapidly prototyping optimization layers



Code example: OptNet QP

Before: 1k lines of code
Hand-implemented and optimized PyTorch GPU-capable batched primal-dual interior point method

Now: 10 lines of code
Same speed

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

Parameters/Submodules : Q, q, A, b, G, h

```
import cvxpy as cp
from cvxpyth import CvxpyLayer
```

```
Q = cp.Parameter((n, n), PSD=True)
p = cp.Parameter(n)
A = cp.Parameter((m, n))
b = cp.Parameter(m)
G = cp.Parameter((p, n))
h = cp.Parameter(p)
x = cp.Variable(n)
obj = cp.Minimize(0.5*cp.quad_form(x, Q) + p.T * x)
cons = [A*x == b, G*x <= h]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[Q, p, A, b, G, h], out=[x])
```

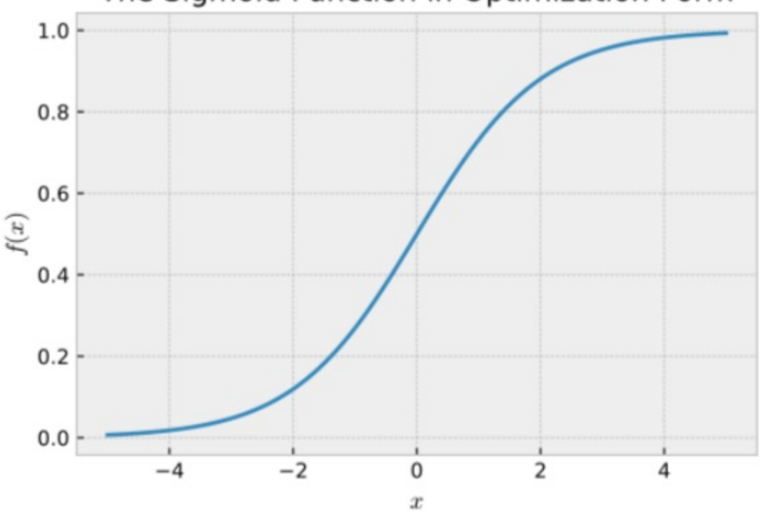
Code example: The sigmoid

$$y = \frac{1}{1+e^{-x}}$$

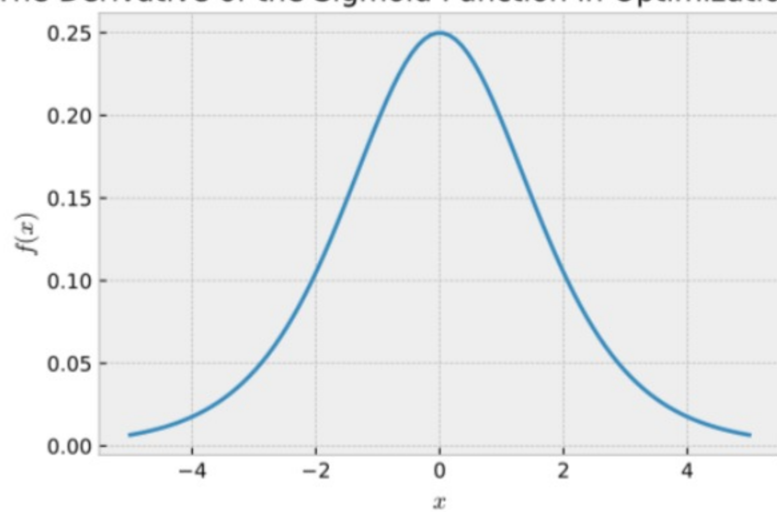
$$y^\star = \begin{array}{ll} \underset{y}{\operatorname{argmin}} & -y^T x - H_b(y) \\ \text{subject to} & 0 \le y \le 1 \end{array}$$

```python
x = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(-x.T*y - cp.sum(cp.entr(y) + cp.entr(1.-y)))
prob = cp.Problem(obj)
layer = CvxpyLayer(prob, params=[x], out_vars=[y])
```

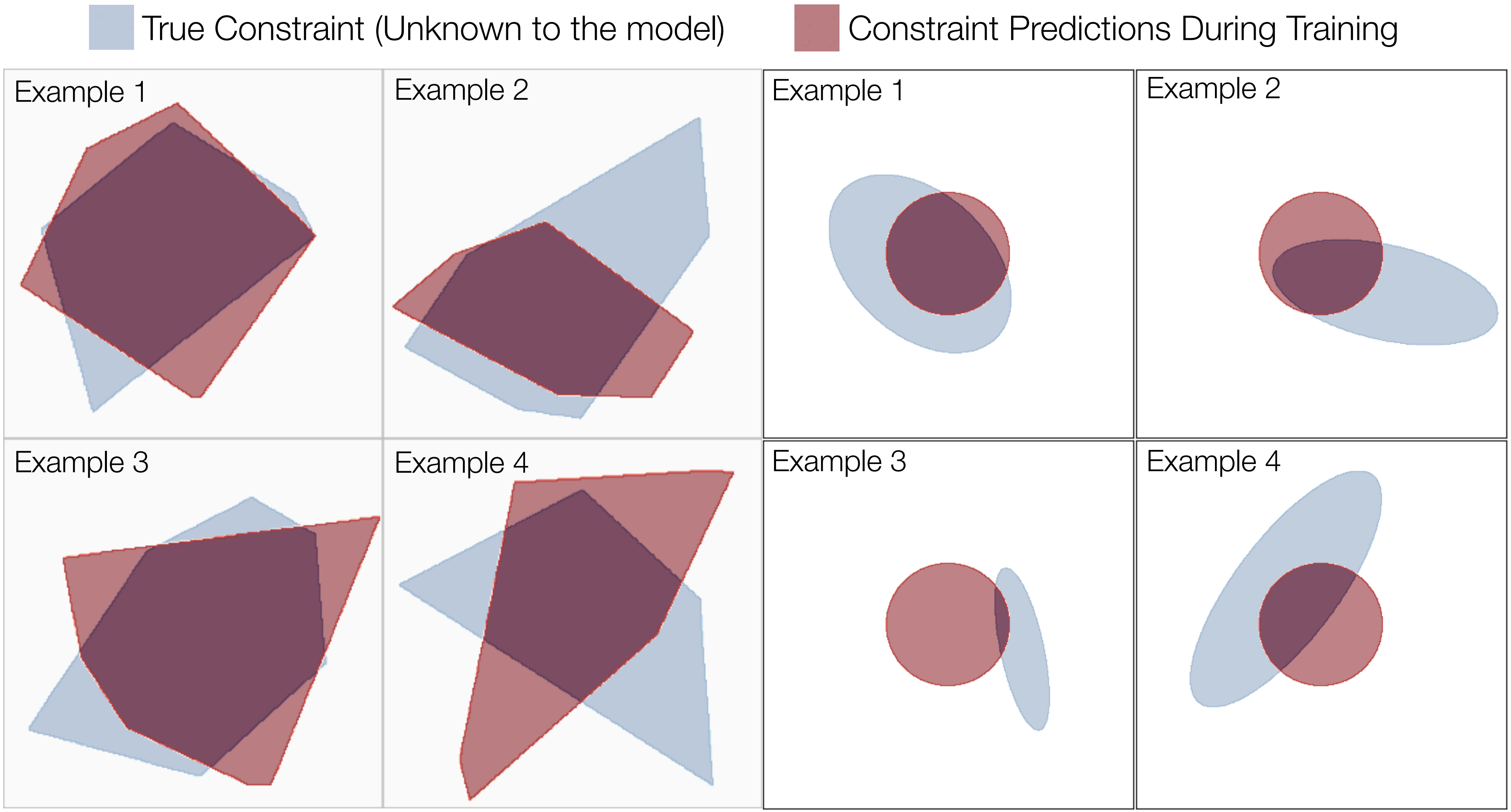
The Sigmoid Function in Optimization Form



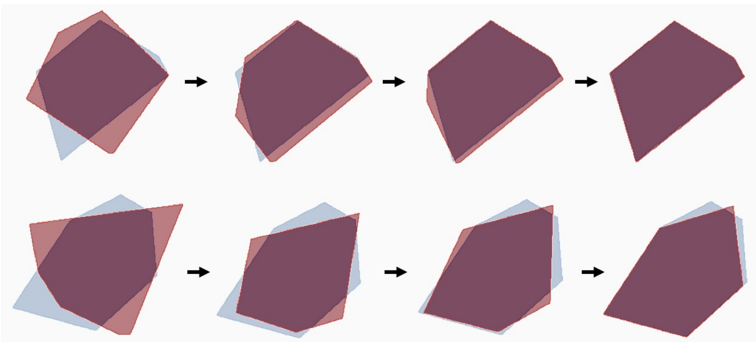
The Derivative of the Sigmoid Function in Optimization Form



OptNet Application: Modeling Constraints

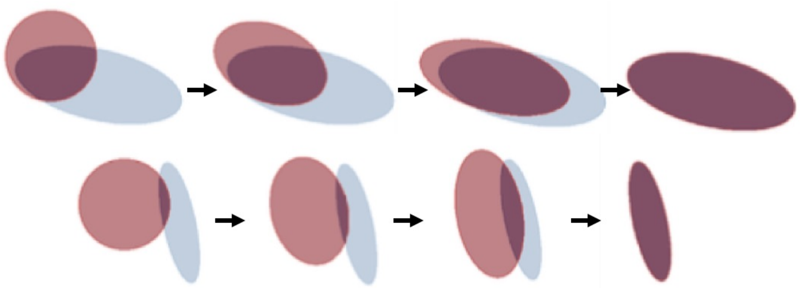


Code example: Constraint modeling



$$\hat{y} = \underset{y}{\operatorname{argmin}} \ \frac{1}{2}||p-y||_2^2 \quad \text{s.t.} \ \ Gy \leq h$$

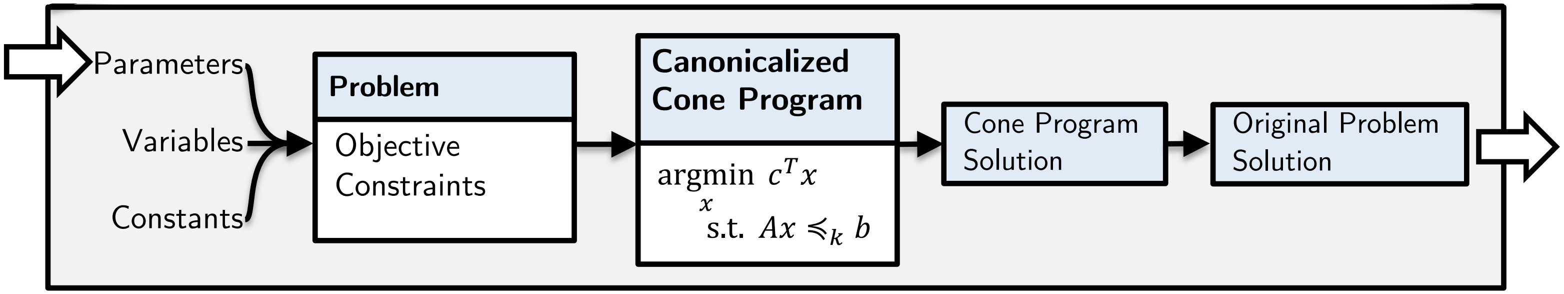
```
G = cp.Parameter((m, n))
h = cp.Parameter(m)
p = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(0.5*cp.sum_squares(y-p))
cons = [G*y <= h]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[p, G, h], out=[y])
```



$$\hat{y} = \underset{y}{\operatorname{argmin}} \ \frac{1}{2}||p-y||_2^2 \quad \text{s.t.} \ \ \frac{1}{2}(y-z)^\top A(y-z) \leq 1$$

```
A = cp.Parameter((n, n), PSD=True)
z = cp.Parameter(n)
p = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(0.5*cp.sum_squares(y-p))
cons = [0.5*cp.quad_form(y-z, A) <= 1]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[p, A, z], out=[y])
```

What's going on behind the scenes?



Cone Program Differentiation

Much **more general** than the QPs we considered in OptNet

Question from my thesis proposal: How to differentiate non-polyhedral cones?

Non-trivial because we can't easily differentiate the KKT conditions of cone programs because of non-trivial cone constraints

Cone Program Differentiation

Take the homogenous self-dual embedding of the cone program

$$Qu = v \qquad \text{where} \qquad Q = \begin{bmatrix} 0 & A^\top & c \\ -A & 0 & b \\ -c^\top & -b^\top & 0 \end{bmatrix} \qquad \begin{array}{l} u \in \boldsymbol{\mathcal{K}}, \quad v \in \boldsymbol{\mathcal{K}}^*, \quad u_{m+n+1} + v_{m+n+1} > 0, \\ \boldsymbol{\mathcal{K}} = \mathbb{R}^n \times \mathcal{K}^* \times \mathbb{R}_+, \quad \boldsymbol{\mathcal{K}}^* = \{0\}^n \times \mathcal{K} \times \mathbb{R}_+, \end{array}$$

Definition: Minty's projection onto the embedding space
$M\colon \mathbb{R}^{m+n+1} \to \mathcal{C} \quad M(z) = (\Pi z, -\Pi^* z)$ where $\mathcal{C} = \{(u, v) \in \boldsymbol{\mathcal{K}} \times \boldsymbol{\mathcal{K}}^* \mid u^\mathrm{T} v = 0\}$

Take the **residual map** of Minty's parameterization:

$$\mathcal{R}(z) = Q\Pi z + \Pi^* z$$

Implicitly differentiate $\mathcal{R}$:

$$D_\theta(z) = -\left(D_z \mathcal{R}(z^*)\right)^{-1} D_\theta R(z^*)$$

Captures KKT differentiation as a special case

Closing Thoughts And Future Directions

Optimization is a **powerful primitive** to use within larger systems

- This thesis has uncovered **theoretical** and **engineering** foundations
- Can be **propagated through and learned**, just like any layer
- Provides a **perspective to analyze** existing models and layers
- Can be used to **project onto sets in a differentiable way**
  Even if a closed form solution doesn't exist

Applications in:

- **Model-based RL and control**
  - In the **policy** or for **exploration**
  - Inverse control, **cost learning**
  - **Learning embedded state spaces** for planning
  - **Multi-agent systems**
    Interpret other agents as solving optimization problems
- **Meta-Learning**
- **Energy-based learning and structured prediction**

Closing Thoughts And Future Directions

Optimization is a **powerful primitive** to use within larger systems

- This thesis has uncovered **theoretical** and **engineering** foundations
- Can be **propagated through and learned**, just like any layer
- Provides a **perspective to analyze** existing models and layers
- Can be used to **project onto sets in a differentiable way**
  Even if a closed form solution doesn't exist

Applications in:

- **Model-based RL and control**
  - In the **policy** or for **exploration**
  - Inverse control, **cost learning**
  - **Learning embedded state spaces** for planning
  - **Multi-agent systems**
    Interpret other agents as solving optimization problems
- **Meta-Learning**
- **Energy-based learning and structured prediction**

Thesis Defense

Differentiable Optimization-Based Modeling for Machine Learning

Brandon Amos • Carnegie Mellon University

brandondamos
bamos.github.io

The source code behind all of my work is free and publicly available:
http://github.com/bamos/thesis

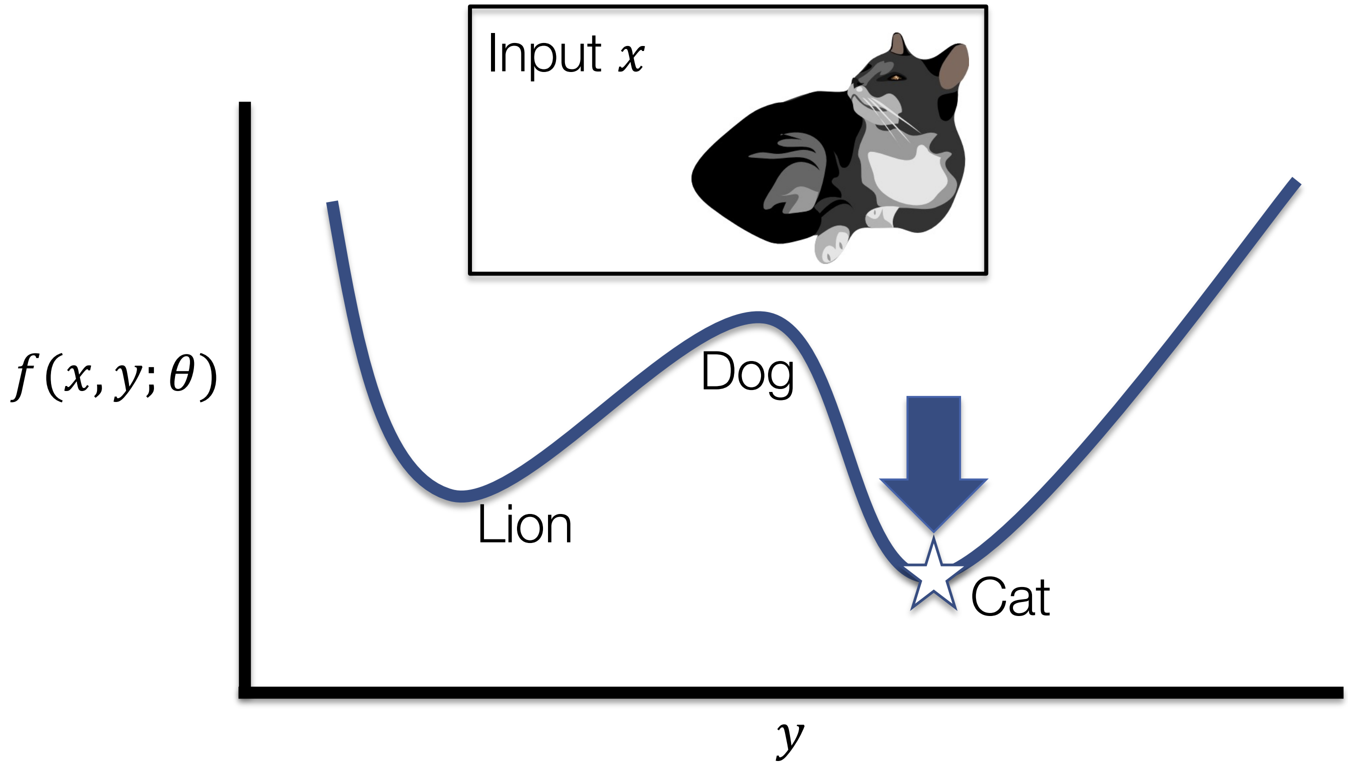
Extra Slides

Optimization-Based Inference

Structured prediction: define a network over $\mathcal{X}\times\mathcal{Y}$ and predict via

$$\hat{y}(x) = \mathrm{argmin}_y\, f(x, y; \theta)$$

*This is also called **energy-based modeling**



Structured prediction models nicely capture dependencies in the output space

Especially useful for high-dimensional, correlated output spaces

- Multi-label classification
- Semantic segmentation
- Scene-graph generation

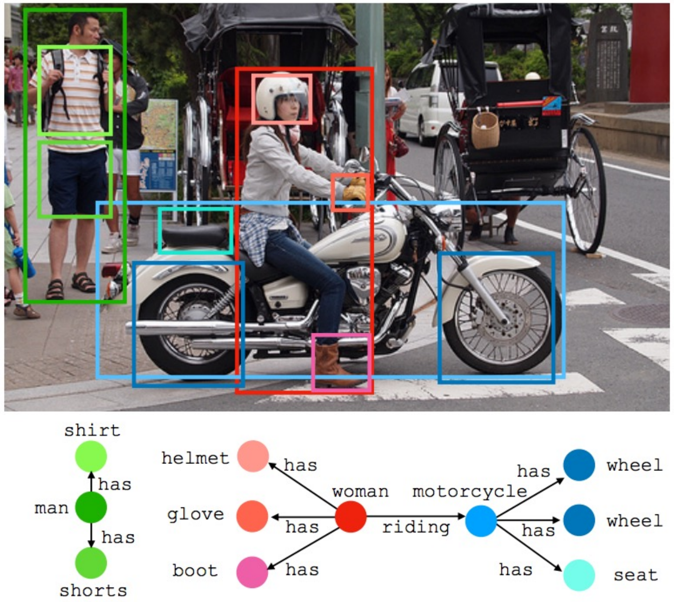
Difficult to capture with most feed-forward models

Intractable in many graphical models if a special structure is not imposed

- Like in MRFs/CRFs

Easy with energy-based models

- Just add them to the energy $f_\theta(x, y)$

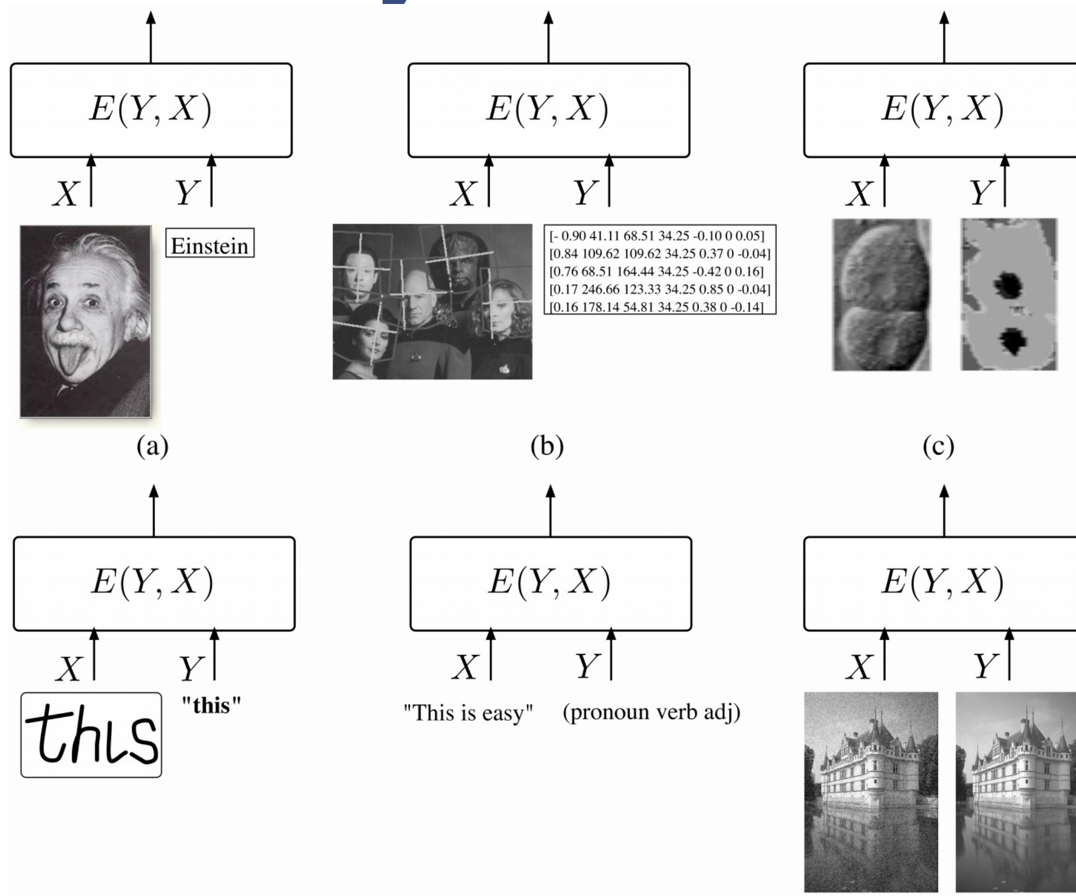


[Zellers2018]

Energy-based models have historically been used for many tasks

Historically these have relied on **shallow energy functions** and **hand-engineered features**

We show how to use a **deep convex energy-based model** with **learned features**



[LeCun2006]

Optimization-Based Inference

Structured prediction: define a network over $\mathcal{X} \times \mathcal{Y}$ and predict via

$$\hat{y}(x) = \operatorname{argmin}_y f(x, y; \theta)$$

Data imputation: build a network over only over $\mathcal{Y}$, given $y_\mathcal{J}$ populate the remaining entries via

$$\hat{y}_{\bar{\mathcal{J}}} = \operatorname{argmin}_{y_{\bar{\mathcal{J}}}} f(y_{\bar{\mathcal{J}}}, y_\mathcal{J}; \theta)$$

Continuous action reinforcement learning: Represent Q function as $Q^\star(s, a) = -f(s, a; \theta)$, policy becomes

$$\pi^\star(s) = \operatorname{argmin}_a f(s, a; \theta)$$

ICNN Portion Overview

Our Contribution: Input Convex Neural Networks

Challenges: Inference and Learning

Experiments
- Synthetic
- Multi-label Classification
- Image Completion
- Continuous–Action Q-Learning

Input Convex Neural Networks (ICNNS)

Definition Scalar-valued network $f(x, y; \theta)$ such that f is **convex** in y for all values of x (note that these networks are still **not convex** in $\theta = \{W_i, b_i\}$)

We can efficiently **optimize over some inputs** to the network **given other inputs**

Efficiently captures dependencies in the output space for prediction

It turns out, we don't need very many restrictions on the network to achieve this property

Applications of Optimization for Inference

With ICNNs: All of these problems are convex, "easy" to solve globally

Structured prediction: define a network over $\mathcal{X}\times\mathcal{Y}$ and predict via

$$\hat{y}(x) = \mathrm{argmin}_y \, f(x, y; \theta)$$

Data imputation: build a network over only over $\mathcal{Y}$, given $y_{\mathcal{J}}$ populate the remaining entries via

$$\hat{y}_{\bar{\mathcal{J}}} = \mathrm{argmin}_{y_{\bar{\mathcal{J}}}} \, f(y_{\bar{\mathcal{J}}}, y_{\mathcal{J}}; \theta)$$

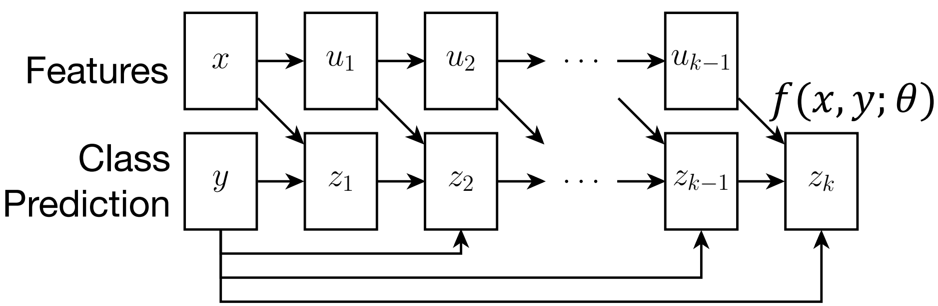
Continuous action reinforcement learning: Represent Q function as $Q^\star(s, a) = -f(s, a; \theta)$, policy becomes

$$\pi^\star(s) = \mathrm{argmin}_a f(s, a; \theta)$$

Example Networks

ICNN for structured prediction:

$$\hat{y}(x) = \operatorname{argmin}_y f(x,y;\theta)$$



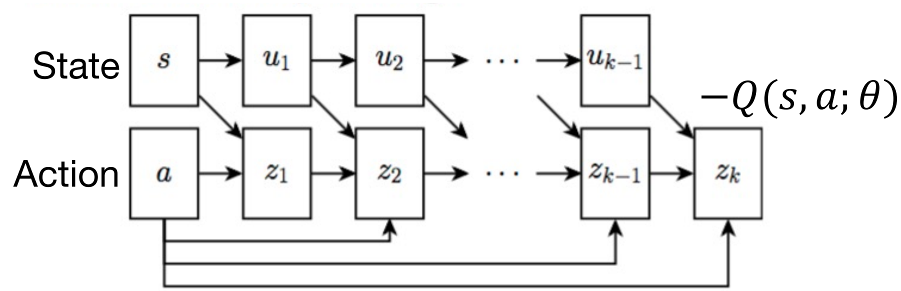
$$u_{i+1} = \tilde{g}_i(\tilde{W}_i u_i + \tilde{b}_i)$$

$$z_{i+1} = g_i\left(W_i^{(uz)}(u_i \circ z_i) + W_i^{(u)} u_i + W_i^{(z)} z_i + W_i^{(y)} y_i + b_i\right)$$

$$f(x,z;\theta) = z_k$$

ICNN for Q learning:

$$\pi^\star(s) = \operatorname{argmin}_a - Q(s,a;\theta)$$



$$u_{i+1} = \tilde{g}_i(\tilde{W}_i u_i + \tilde{b}_i)$$

$$z_{i+1} = g_i\left(W_i^{(z)}\left(z_i \circ [W_i^{(zu)} u_i + b_i^{(z)}]_+\right) + W_i^{(a)}\left(a \circ (W_i^{(au)} u_i + b_i^{(a)})\right) + W_i^{(u)} u_i + b_i\right)$$

$$-Q(s,a;\theta) = f(s,a;\theta) = z_k,\; u_0 = s,\; z_0 = a$$

How to achieve input convexity?

Most networks can be "trivially" modified to **guarantee input convexity**

Consider a simple **feedforward ReLU network**:

$$z_{i+1} = \max\{0, W_i z_i + b_i\}, \qquad i = 1, \dots, k$$
$$f(y;\theta) = z_{k+1},\ z_1 = y$$

Proposition. f is convex in y provided that the W_i are non-negative for $i > 1$

More generally, any activation function that is convex and non-decreasing also has this property.

Is convexity restrictive?

Yes (**by definition**, the functions are restricted to be convex), but **not that bad** in practice

Proposition. ICNNs trivially subsume any feedforward network $\tilde{f}(x)$ with the network $f(x, y) = \left(y - \tilde{f}(x)\right)^2$

More complex convex portion **adds additional structure** over y, which can still be "easily" optimized over

We'll see more evidence for this later

ICNN Portion Overview

Our Contribution: Input Convex Neural Networks

Challenges: Inference and Learning

Experiments
- Synthetic
- Multi-label Classification
- Image Completion
- Continuous–Action Q-Learning

Challenges for ICNNs

Inference: how do we efficiently perform the optimization?

$$y^\star(x;\theta) = \mathrm{argmin}_y\, f(x, y; \theta)$$

Learning: How do we train the network (find θ) such that it gives good predictions?

$$\mathrm{minimize}_\theta \sum_{i=1}^{n} \ell\big(y_i, y^\star(x_i;\theta)\big)$$

Inference in ICNNs

In theory, inference in ICNNs is **just a linear program**

$$\min_y f(y;\theta) = \min_{y,z} z_{k+1}$$
$$\text{s.t.}\ \ z_{i+1} \geq W_i z_i + b_i$$
$$z_i \geq 0 \ \text{ for } i > 1$$
$$z_1 = y$$

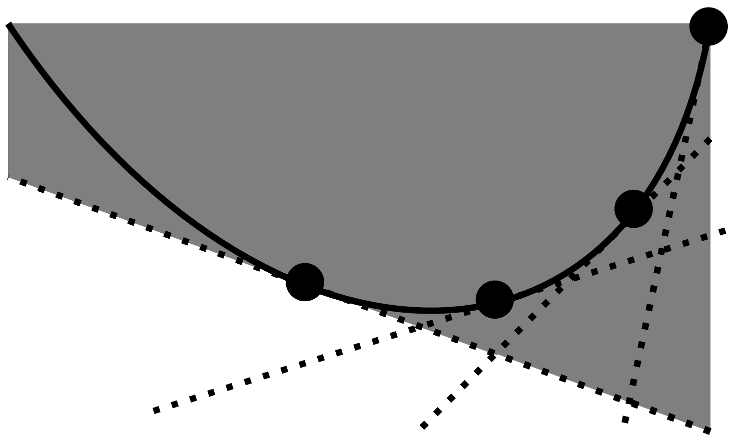
This program has **as many variables as hidden units** in the network, exact solution methods require that we invert the $W_i^T W_i$ matrices

Instead, exploit the fact that we can easily compute the gradient of $f(x, y; \theta)$ with respect to y (this is just **backprop**), and **optimize using gradient-based methods**

We found that the **bundle method** (defined on the next slide) performs better than gradient descent in some cases

Inference with the Bundle Method

Repeatedly **minimize a lower bound** on the function



Uses convexity to minimize more quickly than gradient descent

Boundary constraints are difficult, so we actually use an entropy penalty

$$\tilde{f}(x, y; \theta) + y \log y + (1 - y)\log(1 - y)$$

ICNN Learning

Two possibilities for training networks

1. **Max-margin structured prediction**: enforce constraint that
$$f(x_i, y_i; \theta) \leq \mathrm{argmin}_y \big(f(x_i, y; \theta) + \Delta(y, y_i)\big)$$
Common structured prediction approach
Margin-scaling term $\Delta(y, y_i)$ can be finicky

2. **Argmin differentiation**, directly compute
$$\nabla_\theta \ell\big(y_i, y^\star(x_i; \theta)\big)$$
Can be approximated by unrolling an optimization procedure
Plays nicely with bundle method and approximate optimization
May require some differential calculus (nothing too nasty)

ICNN Portion Overview

Our Contribution: Input Convex Neural Networks

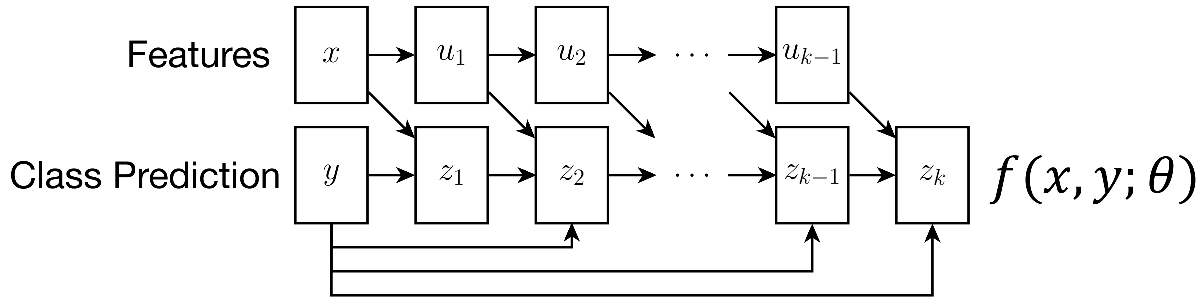
Challenges: Inference and Learning

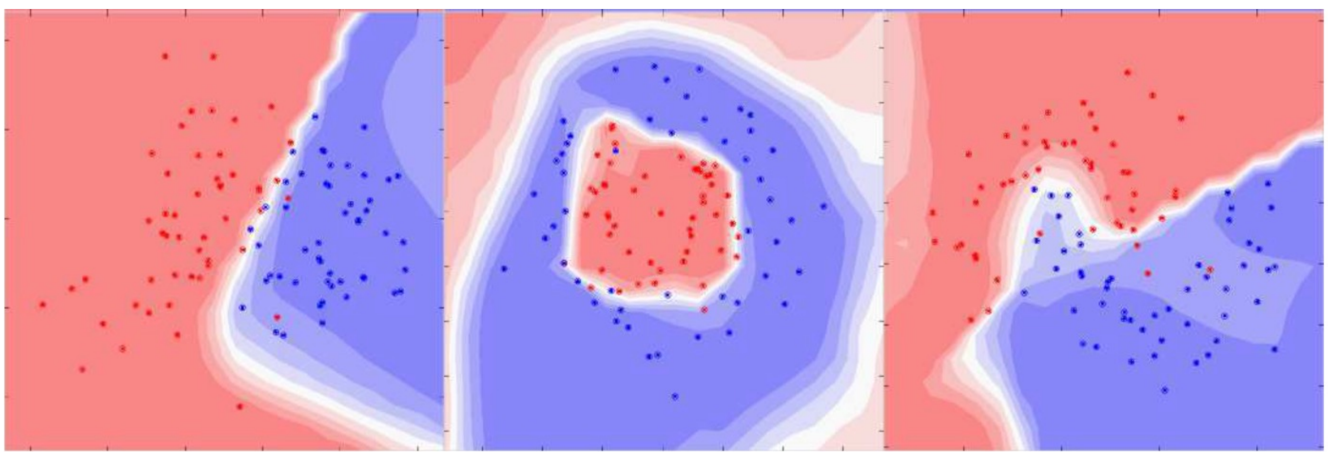
Experiments
- Synthetic
- Multi-label Classification
- Image Completion
- Continuous–Action Q-Learning

Results: toy example

Partially input convex neural network trained to **classify points in 2D space**

$$\hat{y}(x) = \operatorname{argmin}_y f(x, y; \theta)$$





Only point to remember from this: **convex energy function does *not* imply a convex decision boundary**; argmin operator is a powerful one

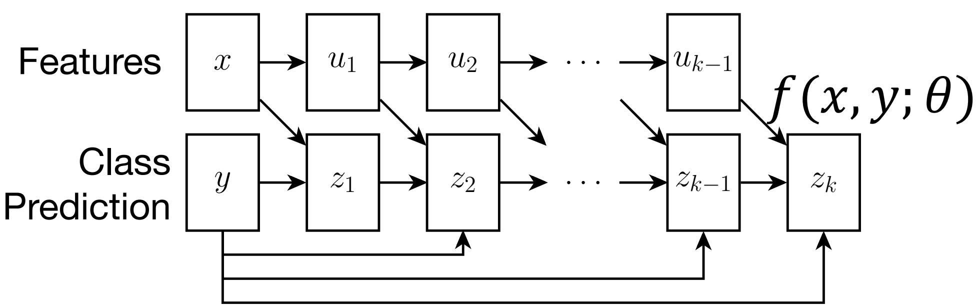
Results: multi-label classification

Task: Predict tags for bibtex entries from bag of words features

Used in Belanger and McCallum, 2016: Structured Prediction Energy Networks

ICNNs almost recover the same performance as SPENs despite the convexity restrictions

$$\hat{y}(x) = \text{argmin}_y\, f(x, y; \theta)$$



(Higher = Better)

Method	Test Macro-F1
NN (Baseline)	0.396
SPEN	**0.422**
ICNN	0.415

Results: image completion

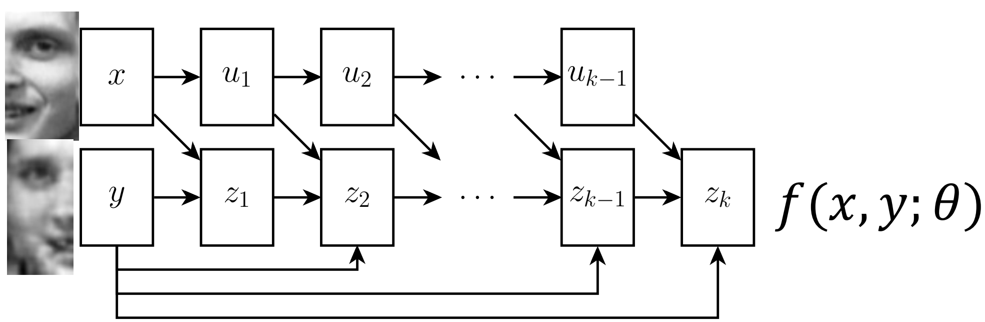
Task: Predict the left side of the image given the right side. Used in Poon and Domingos 2011; Sum-Product Networks

ICNN: DQN-like network over both input and output

$$\hat{y}(x) = \operatorname{argmin}_y f(x, y; \theta)$$

(Given) Right Face Half

(Predicted) Left Face Half



ICNN Test Set Completions

Method	MSE
Sum-Product Network Baseline [PD11]	942.0
Dilated CNN Baseline [YK15]	800.0
FCN Baseline [LSD15]	**795.4**
ICNN - Bundle Entropy	833.0
ICNN - Gradient Decent	872.0
ICNN - Nonconvex	850.9

Input Convex Neural Networks

Brandon Amos Lei Xu J. Zico Kolter
Carnegie Mellon University
School of Computer Science

Our Contribution: **Input Convex Neural Networks**

Challenges: Inference and Learning

Experiments

1. Synthetic
2. Multi-label Classification
3. Image Completion
4. Continuous–Action Q-Learning

The full TensorFlow source code to reproduce all of our experiments is available online at https://github.com/locuslab/icnn