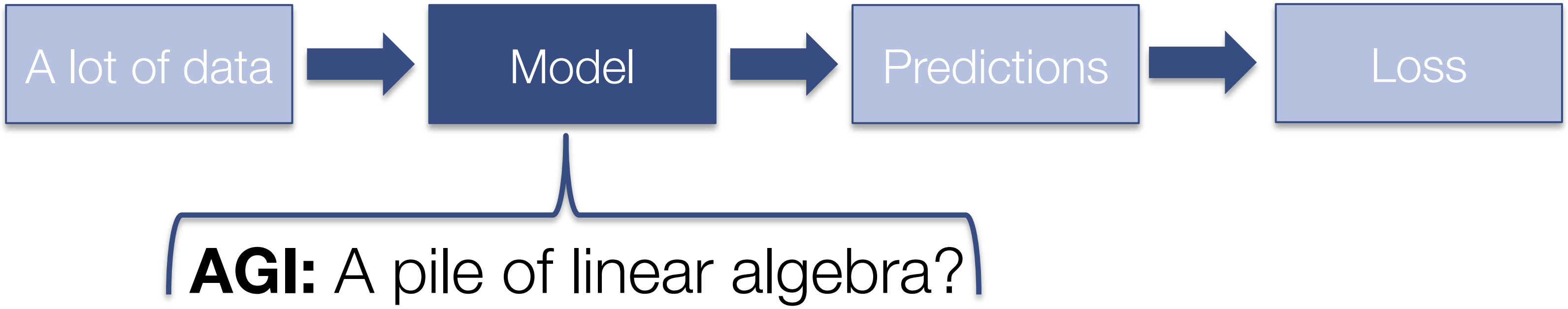
Differentiable optimization-based modeling for machine learning

Brandon Amos • Meta AI (FAIR)

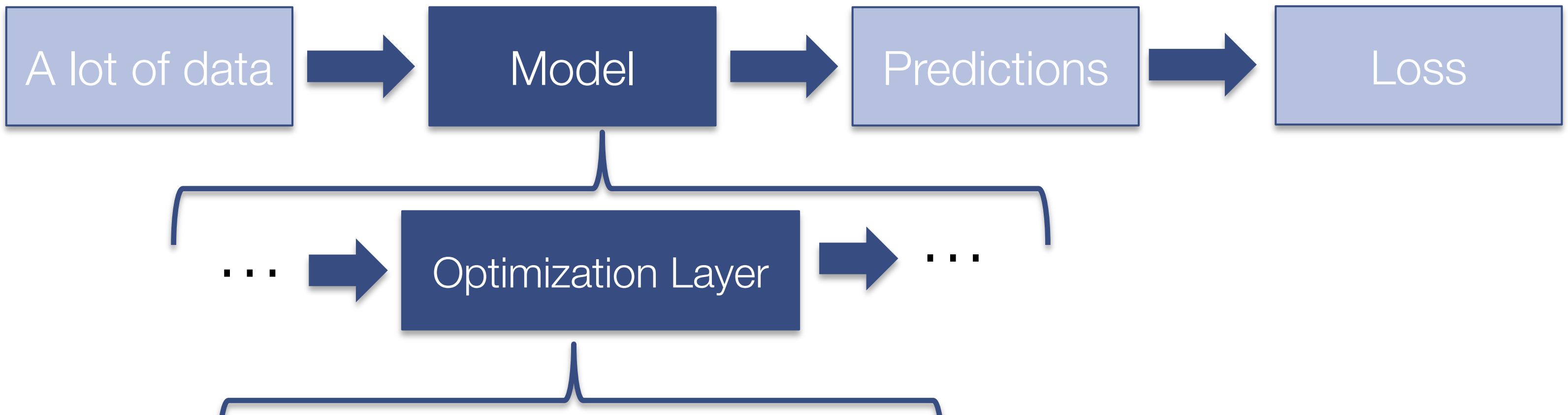
Joint with Akshay Agrawal, Shane Barratt, Byron Boots, Stephen Boyd, Roberto Calandra, Steven Diamond, Priya Donti, Ivan Jimenez, Zico Kolter, Nathan Lambert, Jacob Sacks, Omry Yadan, and Denis Yarats

Can we throw big neural networks at every problem?

(Maybe) Neural networks are **soaring** in vision, RL, and language



Optimization-based modeling for machine learning

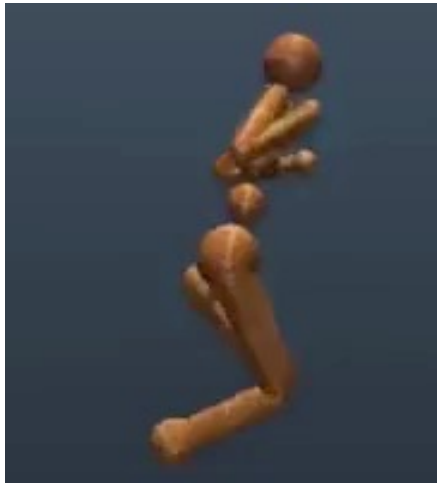


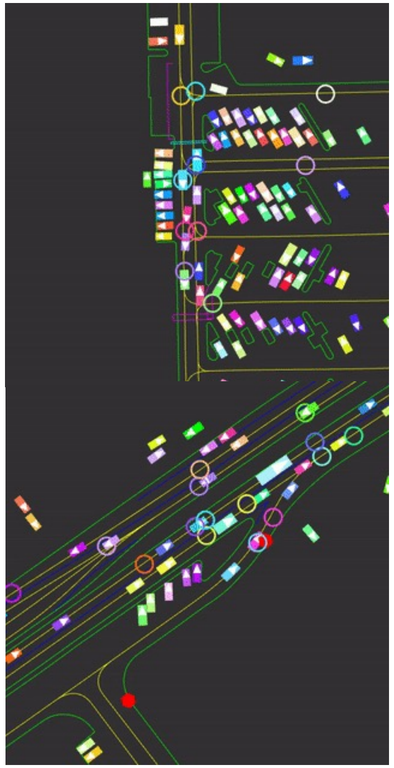
$$
\begin{aligned}
z_{i+1} = \ & \underset{z}{\operatorname{argmin}} \ f_\theta(z, z_i) \\
& \text{subject to} \ z \in C_\theta(z, z_i)
\end{aligned}
$$

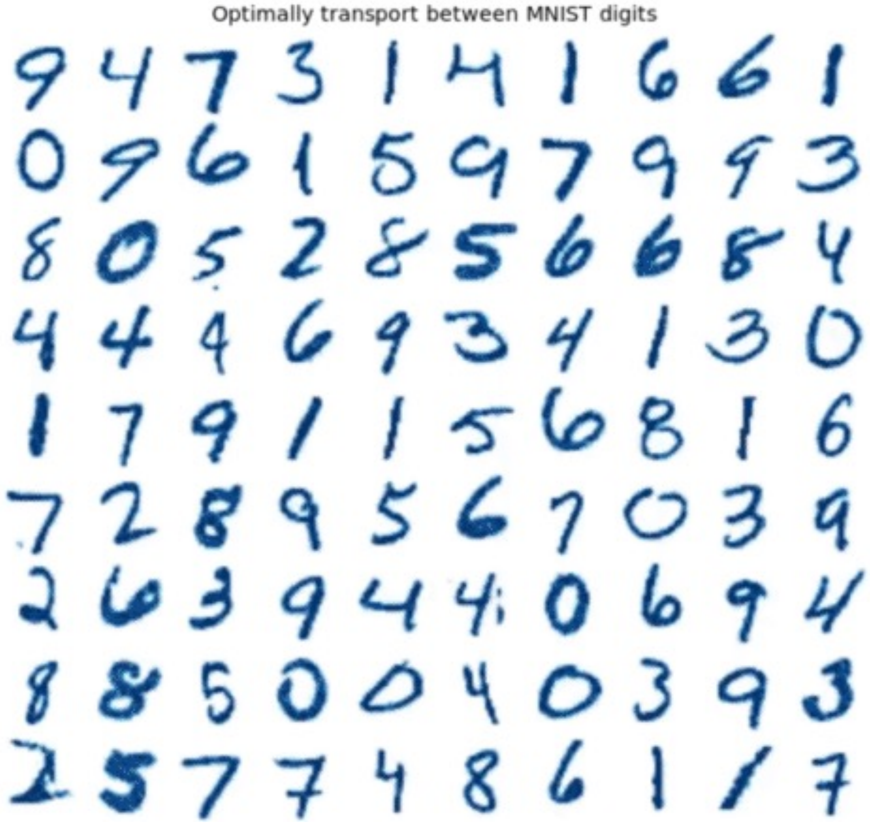
Adds **domain knowledge** and **hard constraints** to your **modeling** pipeline
Integrates and **trains** nicely with your other **end-to-end** modeling components
Applications in **RL, control, meta-learning, game theory, optimal transport**

Why optimization-based modeling?

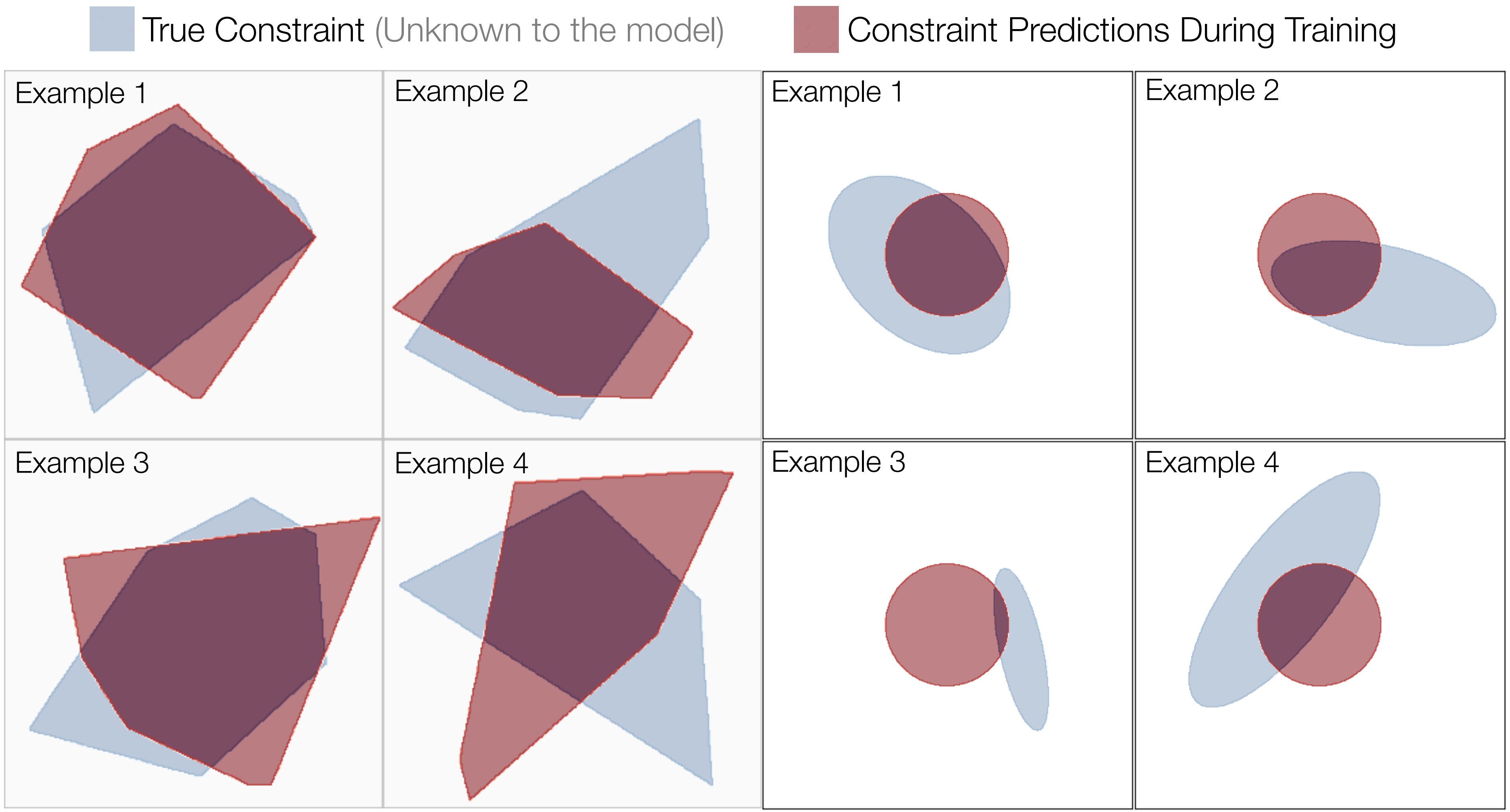
Non-trivial reasoning operations are **fundamentally optimization problems**
Why **unnecessarily** approximate them? (e.g. with a neural network)
Explicitly **model the optimization components** and **learn the rest** (when possible)







Optimization layers model hard constraints



This talk: differentiable optimization-based models

Standard operations as convex optimization layers — warmup

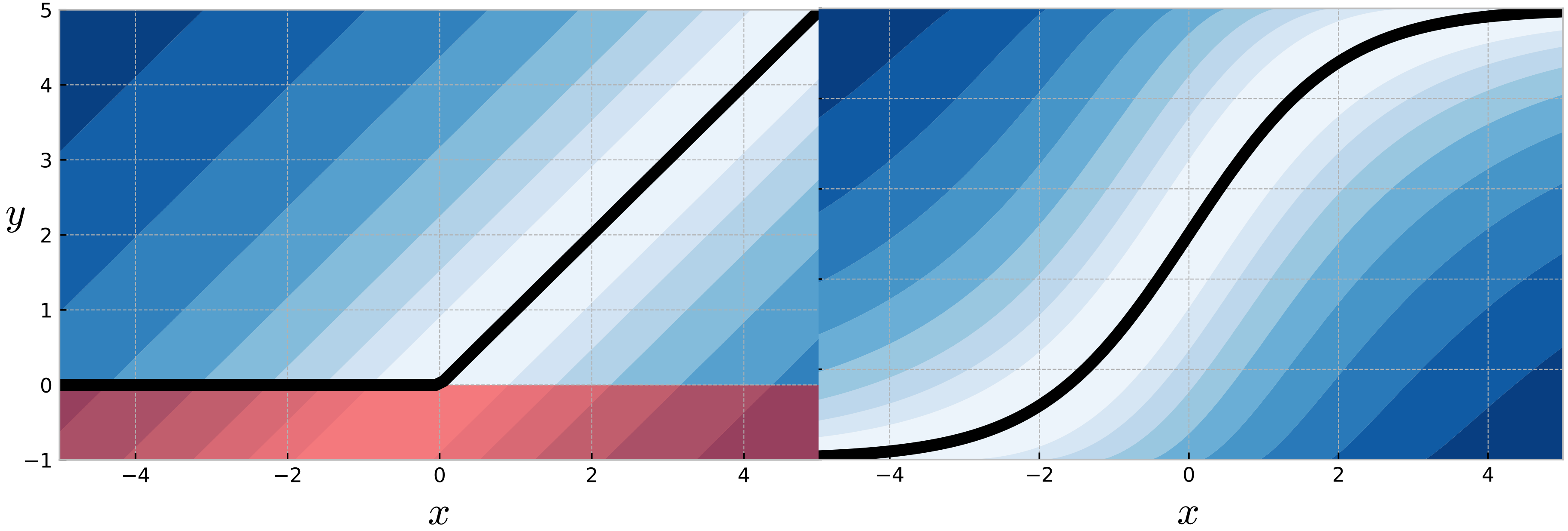
Differentiable optimization theory and practice — core

Differentiable control and objective mismatch — focus application

Convex optimization is expressive

The **argmin** of a convex optimization problem is **non-convex** and expressive
Standard non-linearities to be seen as **solutions** to convex optimization problems
We'll start simple for **intuition** and **motivation to generalize beyond these**

$$y^\star(x) = \operatorname*{argmin}_y \; f(y;x) \;\; \text{subject to} \;\; y \in C(x)$$

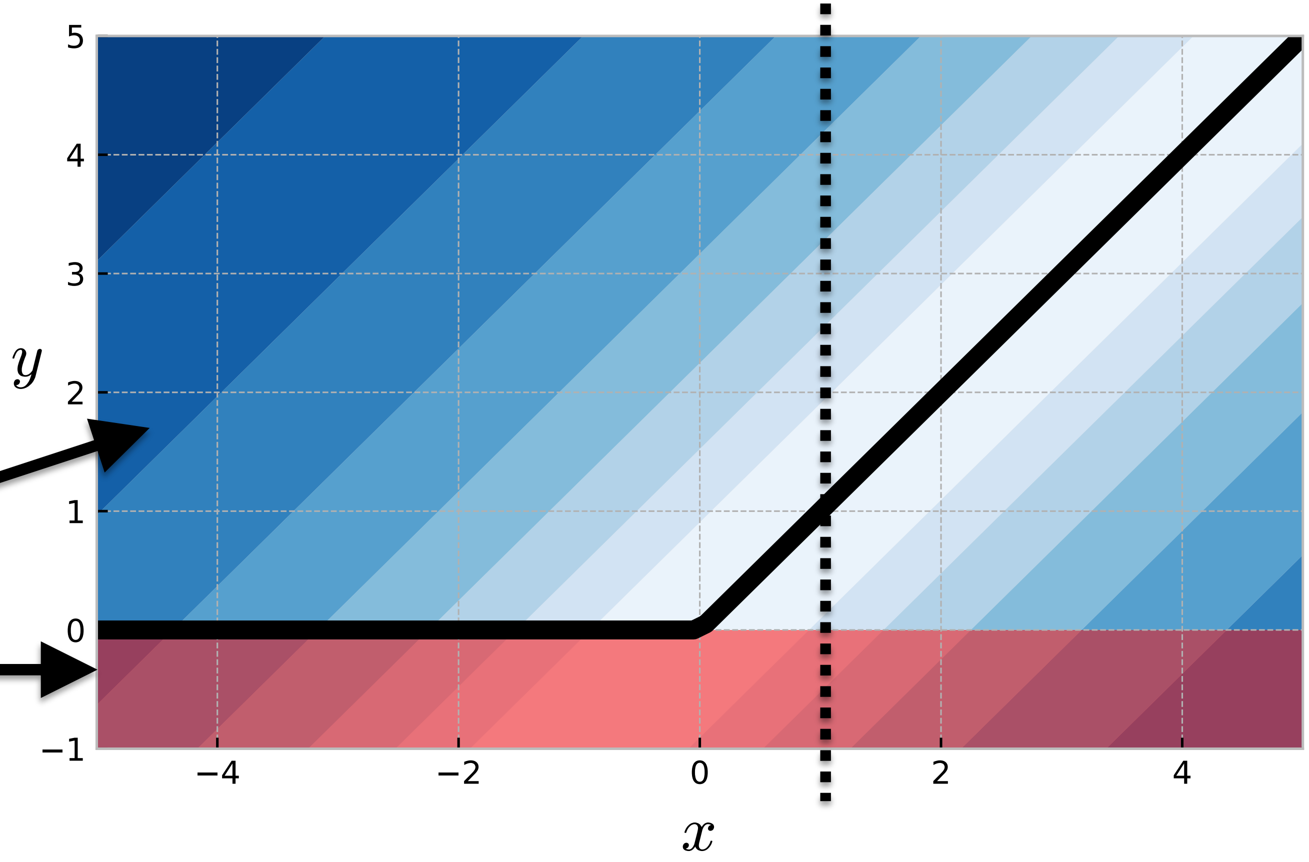


The ReLU is a convex optimization layer

Proof: Comes from first-order optimality (section 2 of my thesis)

$$\mathrm{ReLU}(x) = \max\{0, x\}$$

$$\mathrm{ReLU}(x) = \operatorname*{argmin}_{y} \ \|y - x\|_2^2 \quad \text{s.t.} \quad y \geq 0$$



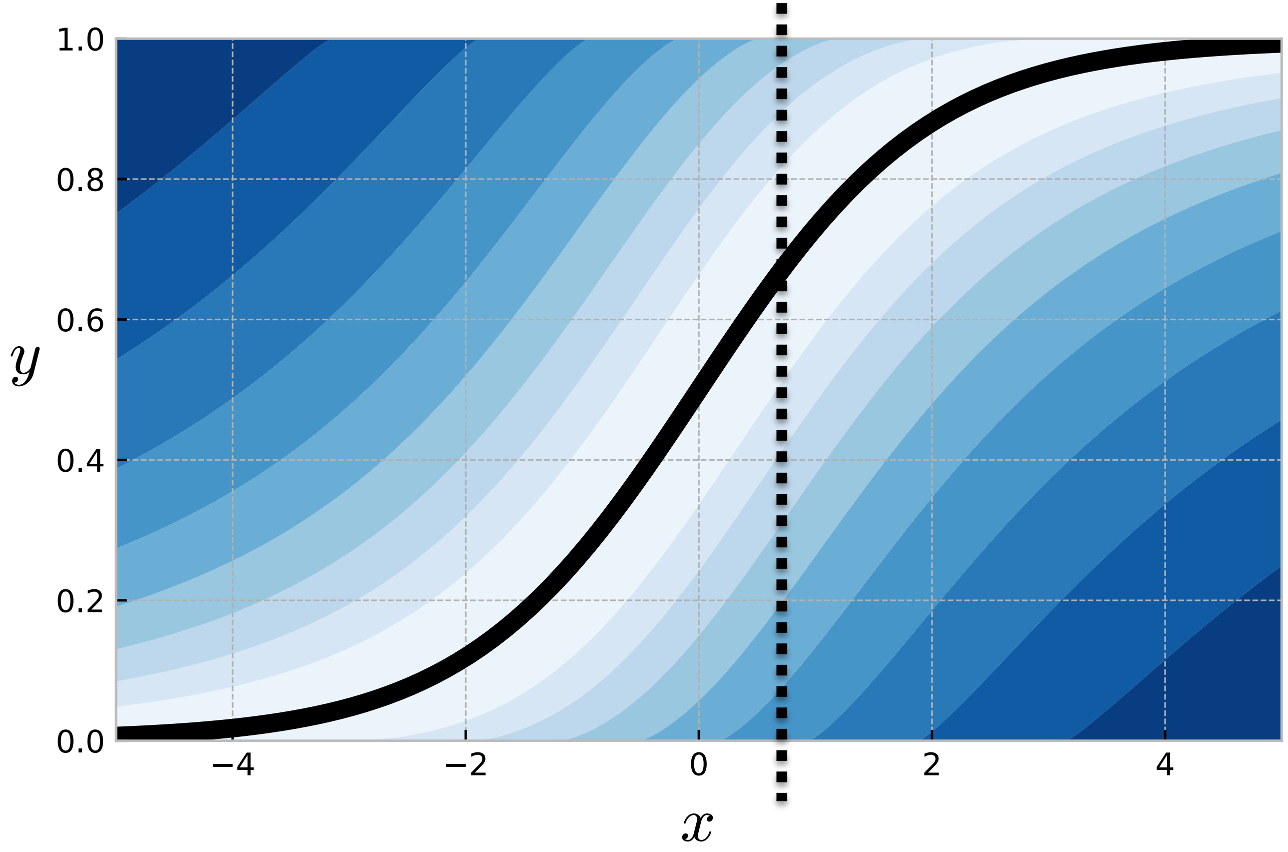
The sigmoid is a convex optimization layer

Proof: Comes from first-order optimality (section 2 of my thesis)

$$\sigma(x) = \frac{1}{1 + \exp\{-x\}}$$

$$\Updownarrow$$

$$\sigma(x) = \underset{y}{\operatorname{argmin}} \; -y^\top x - H_b(y) \quad \text{s.t.} \; 0 \leq y \leq 1$$



The soft-argmax is a convex optimization layer

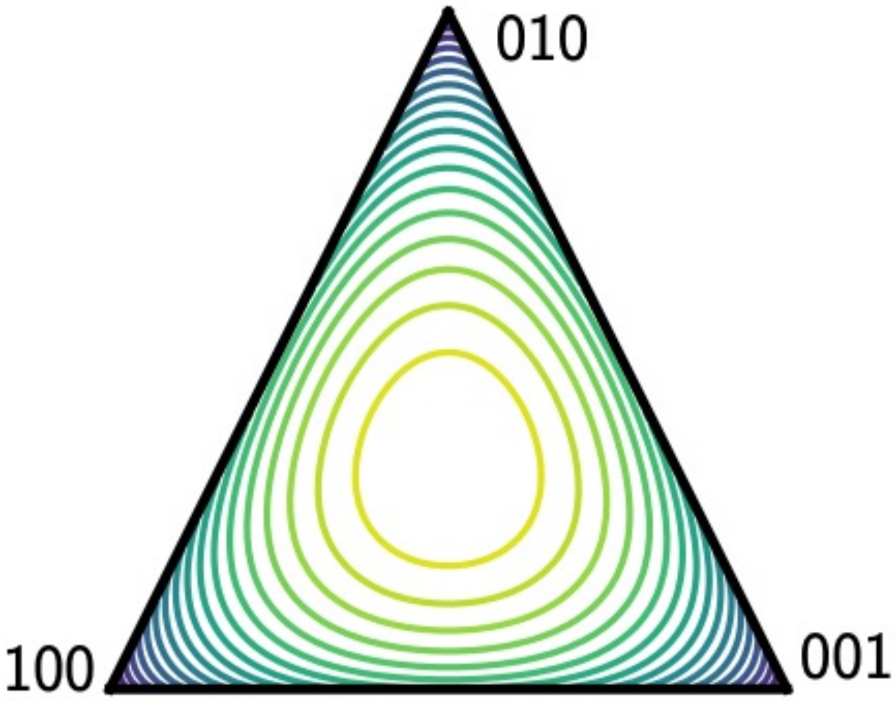
Proof: Comes from first-order optimality (section 2 of my thesis)

$$\pi_\Delta(x/\tau) = \frac{\exp\{x/\tau\}}{\sum_i \exp\{x_i/\tau\}}$$

$$\updownarrow$$

$$\pi_\Delta(x/\tau) = \underset{y}{\operatorname{argmax}} \; y^\top x + \tau H(y) \quad \text{s.t.} \; 0 \le y \le 1, \; 1^\top y = 1$$

(approaches the argmax when $\tau \to 0$)



Contours of the entropy $H(y)$ over the simplex

How can we generalize this?

$$z_{i+1}(z_i) = \underset{z}{\operatorname{argmin}}\ f_\theta(z, z_i)\ \text{subject to}\ z \in C_\theta(z, z_i)$$

Derivatives and backpropagation

For learning, we **differentiate** or backpropagate through these layers — **differentiable optimization**

Easy if the optimization problem has an **explicit, closed-form solution** (often standard differentiation)

Otherwise, need to use **implicit differentiation,** which is also used for sensitivity analysis

This talk: differentiable optimization-based models

Standard operations as convex optimization layers — warmup

Differentiable optimization theory and practice — core

Differentiable control and objective mismatch — focus application

The Implicit Function Theorem

[Dini 1877, Dontchev and Rockafellar 2009]

Given an **implicit function** $f(x)\colon \mathbb{R}^n \to \mathbb{R}^m$
defined by $f(x) \in \{y\colon g(x,y) = 0\}$ where
$g(x,y)\colon \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$

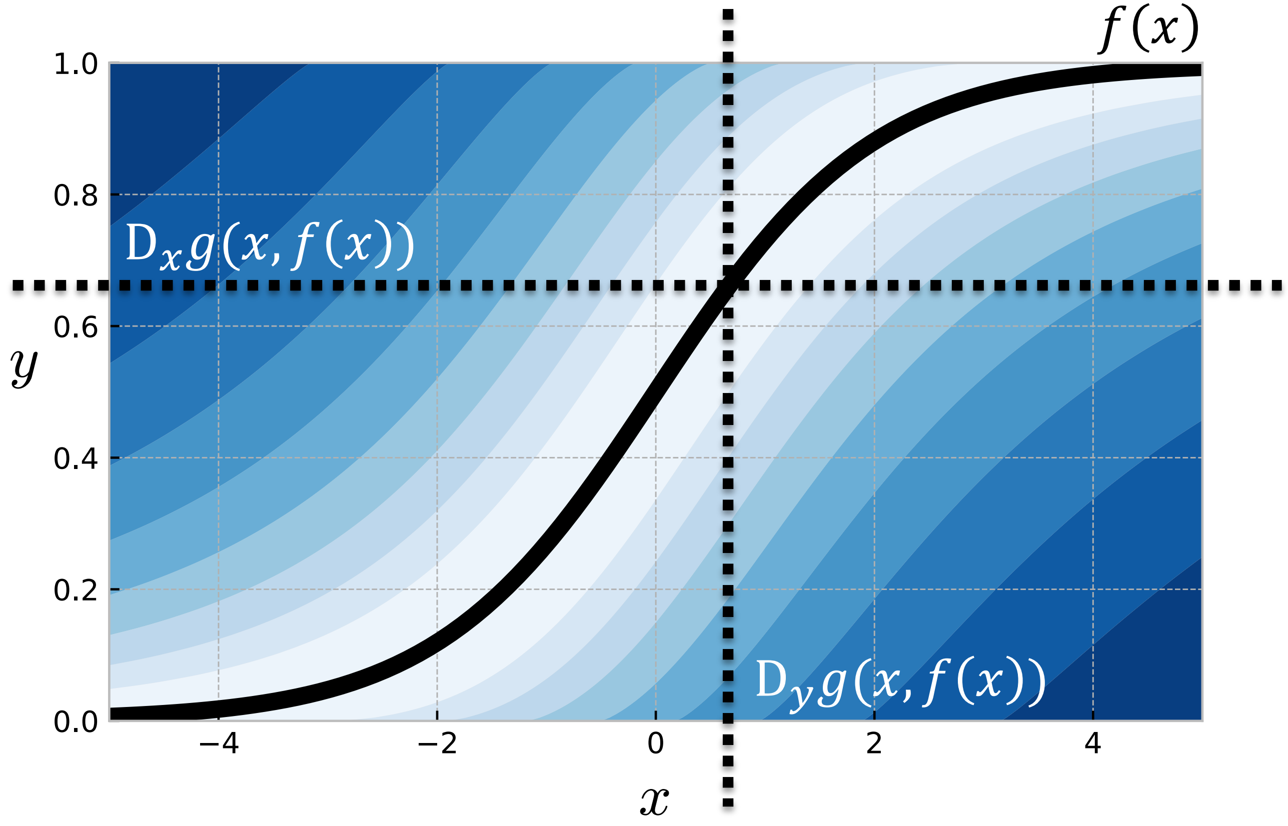
How can we compute $\mathrm{D}_x f(x)$?

The **Implicit Function Theorem** gives

$$\mathrm{D}_x f(x) = -\mathrm{D}_y g\big(x, f(x)\big)^{-1} \mathrm{D}_x g\big(x, f(x)\big)$$

under mild assumptions

Contour of $g(x,y)$ defining an implicit function



Implicitly differentiating a convex quadratic program

Original problem considered in OptNet

$$x^\star = \underset{x}{\operatorname{argmin}} \; \frac{1}{2} x^\top Q x + p^\top x$$
$$\text{subject to } \; Ax = b \quad Gx \leq h$$

$$\Downarrow$$

KKT Optimality

Find $z^\star$ s.t. $\mathcal{R}(z^\star, \theta) = 0$ where $z^\star = [x^\star, \ldots]$ and $\theta = \{Q, p, A, b, G, h\}$

$$\Downarrow$$

Implicitly differentiating $\mathcal{R}$ gives $D_\theta(z^\star) = -\left(D_z \mathcal{R}(z^\star)\right)^{-1} D_\theta \mathcal{R}(z^\star)$

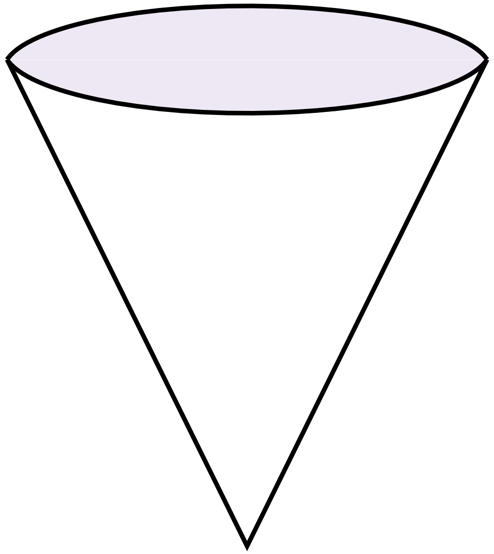
Background: cones and conic programs

Most convex optimization problems can be transformed into a (convex) conic program

$$x^\star = \operatorname*{argmin}_x \; c^\top x \quad \text{subject to } b - Ax \in \mathcal{K}$$

Zero: $\{0\}$
Free: $\mathbb{R}^n$
Non-negative: $\mathbb{R}^n_+$
Second-order (Lorentz): $\{(t, x) \in \mathbb{R}_+ \times \mathbb{R}^n \,|\, \|x\|_2 \leq t\}$
Semidefinite: $\mathbb{S}^n_+$
Exponential: $\{(x, y, z) \in \mathbb{R}^3 \,|\, y e^{x/y} \leq z, y > 0\} \cup \mathbb{R}_- \times \{0\} \times \mathbb{R}_+$

Cartesian Products: $\mathcal{K} = \mathcal{K}_1 \times \cdots \times \mathcal{K}_p$



Implicitly differentiating a conic program

Section 7 of my thesis

$$x^\star = \underset{x}{\operatorname{argmin}} \; c^\top x \quad \text{subject to} \;\; b - Ax \in \mathcal{K}$$

Conic Optimality

Find $z^\star$ s.t. $\mathcal{R}(z^\star, \theta) = 0$ where $z^\star = [x^\star, \ldots]$ and $\theta = \{A, b, c\}$

Implicitly differentiating $\mathcal{R}$ gives $D_\theta(z^\star) = -\left(D_z\mathcal{R}(z^\star)\right)^{-1} D_\theta\mathcal{R}(z^\star)$

Applications of differentiable convex optimization

Learning **hard constraints** (Sudoku from data)

Modeling **projections** (ReLU, sigmoid, softmax; differentiable top-k, and sorting)

Game theory (differentiable equilibrium finding)

RL and control (differentiable control-based policies, enforcing safety constraints)

Meta-learning (differentiable SVMs and optimizers, implicit MAML)

Energy-based learning and **structured prediction** (differentiable inference with, e.g., ICNNs)

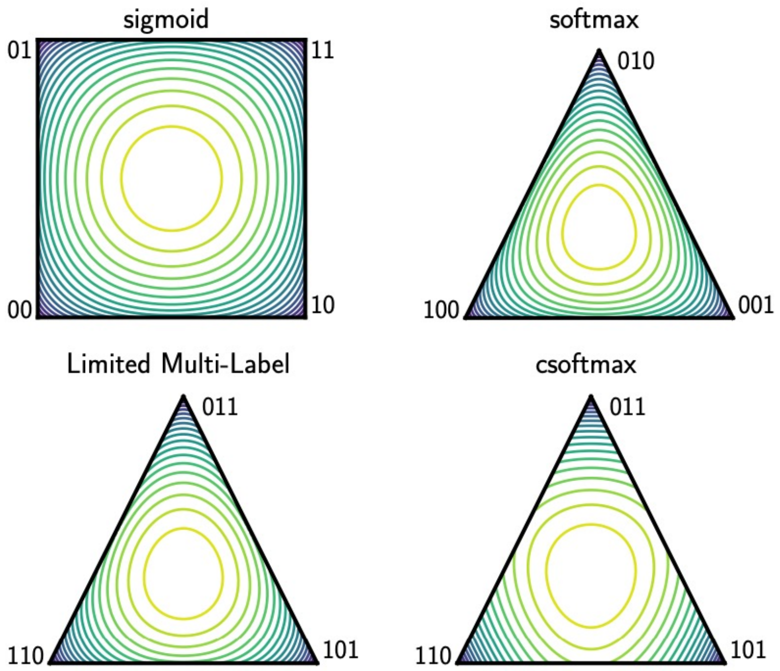
Amortized optimization (as models or for enforcing constraints via differentiable projections)

From the softmax to soft/differentiable top-k

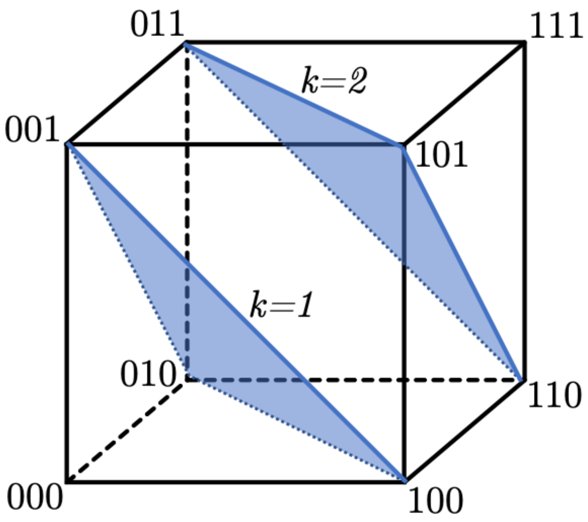
Constrained softmax, constrained sparsemax, Limited Multi-Label Projection

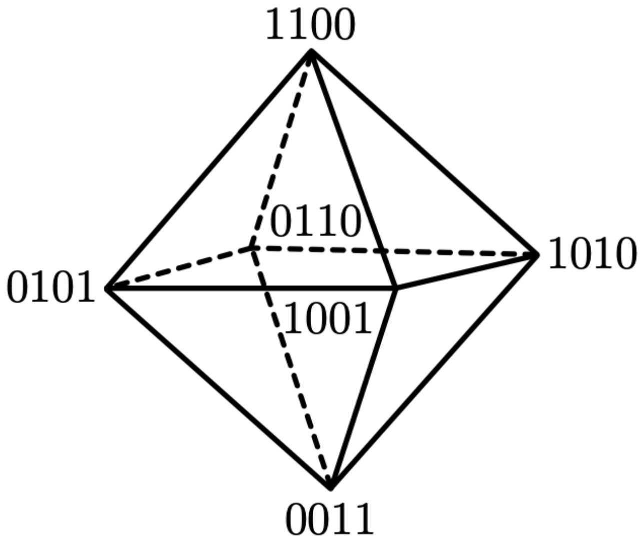
$$y^\star = \underset{y}{\operatorname{argmin}} \; -y^\top x - H(y) \quad \text{subject to} \quad 0 \le y \le 1, \quad 1^\top y = 1$$

$$y^\star = \underset{y}{\operatorname{argmin}} \; -y^\top x - H_b(y) \quad \text{subject to} \quad 0 \le y \le 1, \quad 1^\top y = k$$



Contours of the entropy penalties



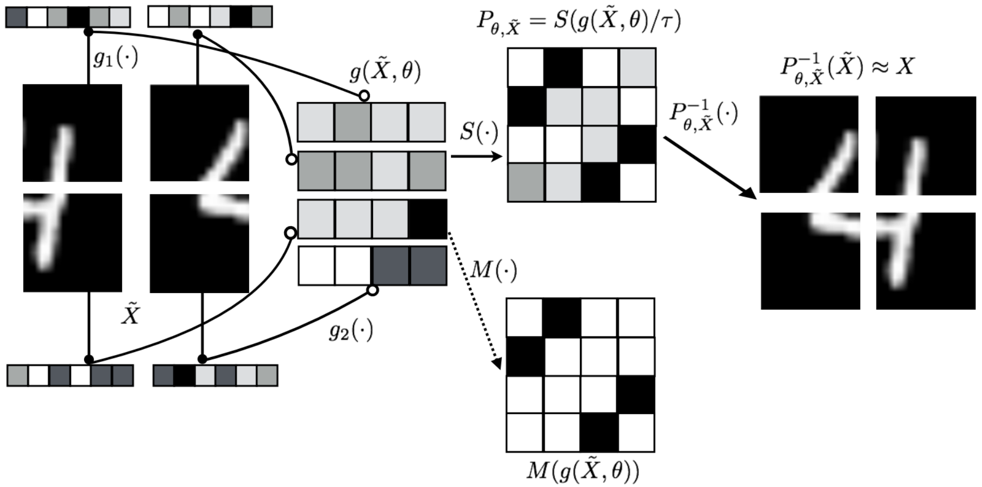


Differentiable permutations, sorting and SVMs

Differentiable permutations and sorting (Gumbel-Sinkhorn)
Projection onto the **Birkhoff polytope** $\mathcal{B}_N$:

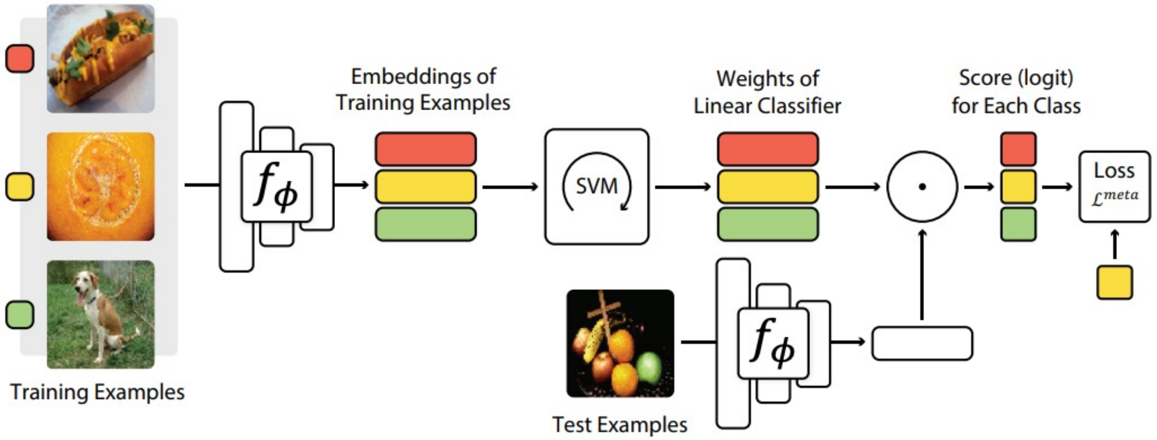
$$S\left(\frac{X}{\tau}\right) = \underset{P \in \mathcal{B}_N}{\operatorname{argmax}} \ \langle P, X \rangle_F + \tau H(P)$$

$$\mathcal{B}_N = \{X : X \geq 0, \Sigma_i X_{ij} = \Sigma_j X_{ij} = 1\}$$



Differentiable SVMs (MetaOptNet)
Differentiate the decision boundary w.r.t. the dataset

$$w^\star = \underset{w}{\operatorname{argmin}} \ \|w\|^2 + C \sum_i \max\{0, 1 - y_i f(x_i)\}$$



Optimization layers need to be carefully implemented

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

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

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

```
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)
```

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

$$
\begin{aligned}
\nabla_Q \ell &= \frac{1}{2}(d_z z^T + z d_z^T) & \nabla_q \ell &= d_z \\
\nabla_A \ell &= d_\nu z^T + \nu d_z^T & \nabla_b \ell &= -d_\nu \\
\nabla_G \ell &= D(\lambda^\star)(d_\lambda z^T + \lambda d_z^T) & \nabla_h \ell &= -D(\lambda^\star) d_\lambda
\end{aligned}
$$

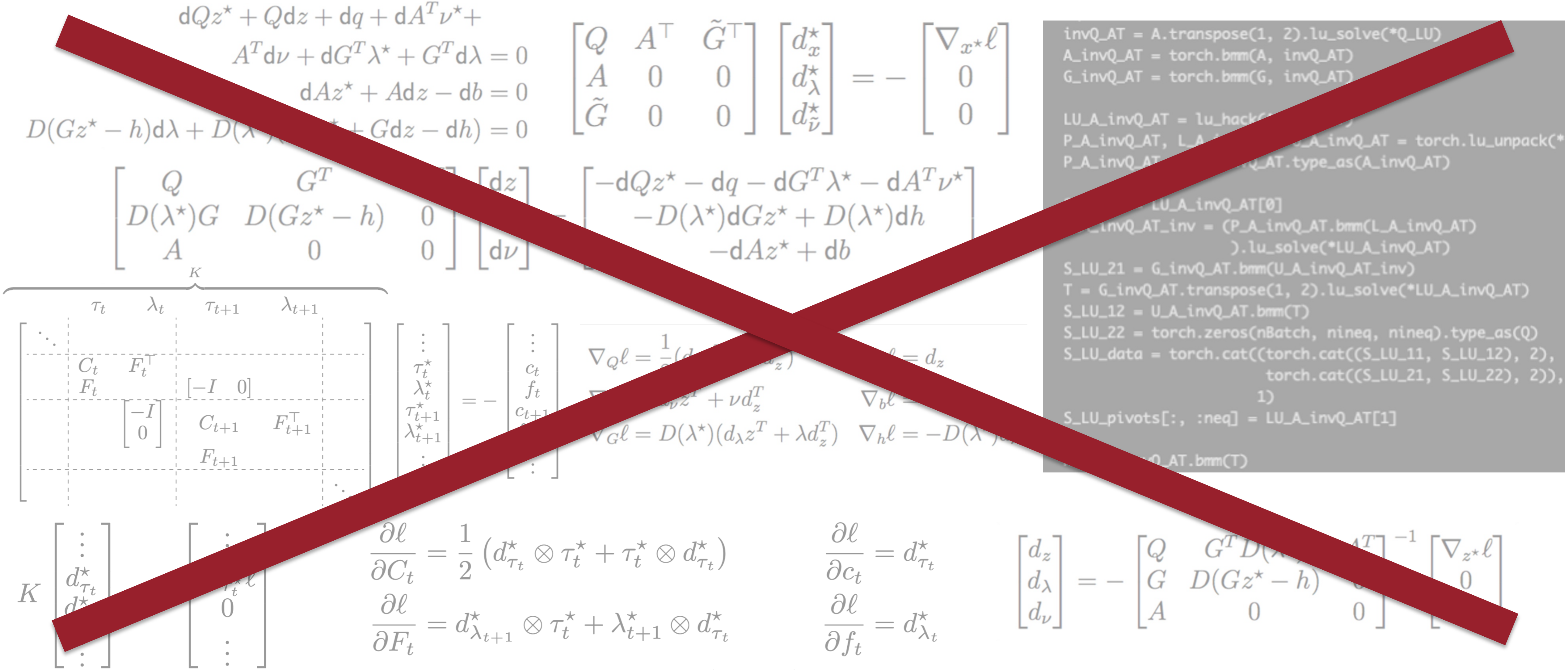
$$
K \begin{bmatrix} \vdots \\ d_{\tau_t}^\star \\ d_{\lambda_t}^\star \\ \vdots \end{bmatrix} = - \begin{bmatrix} \vdots \\ \nabla_{\tau_t^\star}\ell \\ 0 \\ \vdots \end{bmatrix}
$$

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

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

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

Why should practitioners care?



Differentiable convex optimization layers

NeurIPS 2019 and officially in CVXPY!
Joint work with A. Agrawal, S. Barratt, S. Boyd, S. Diamond, J. Z. Kolter

CVXpy

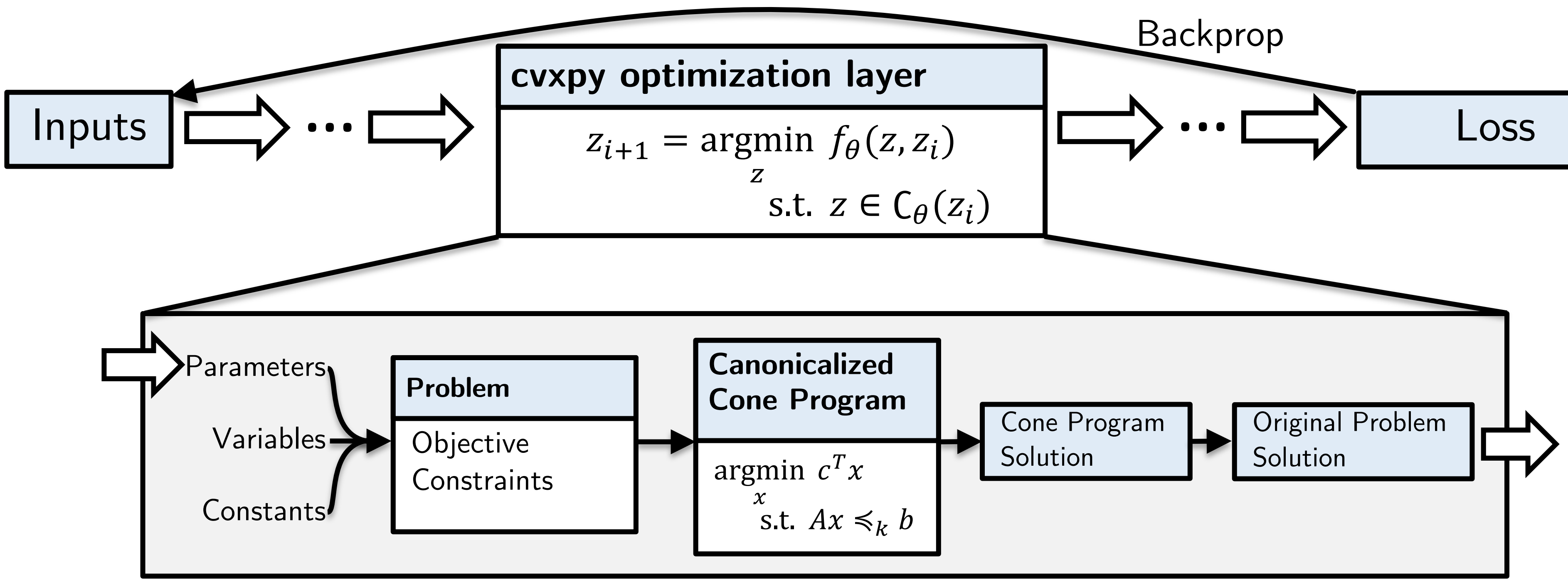
$$\begin{aligned} x^\star(\theta) = \ & \operatorname*{argmin}_x \ f(x;\theta) \\ & \text{subject to } \ g(x;\theta) \leq 0 \\ & \phantom{\text{subject to }} \ h(x;\theta) = 0 \end{aligned}$$

PyTorch

TensorFlow

`locuslab.github.io/2019-10-28-cvxpylayers`

A new way of rapidly prototyping optimization layers



Code example: OptNet QP

Before: 1k lines of code → **Now:** 10 lines of code

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

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

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

```python
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])
```

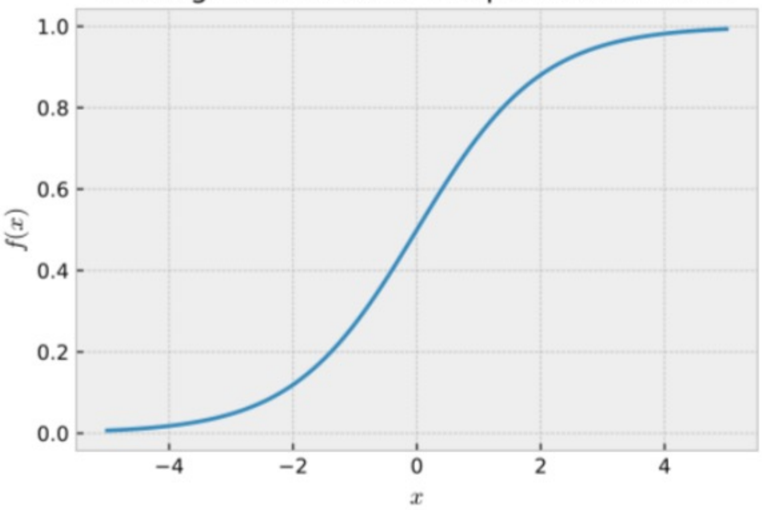
Code example: the sigmoid

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

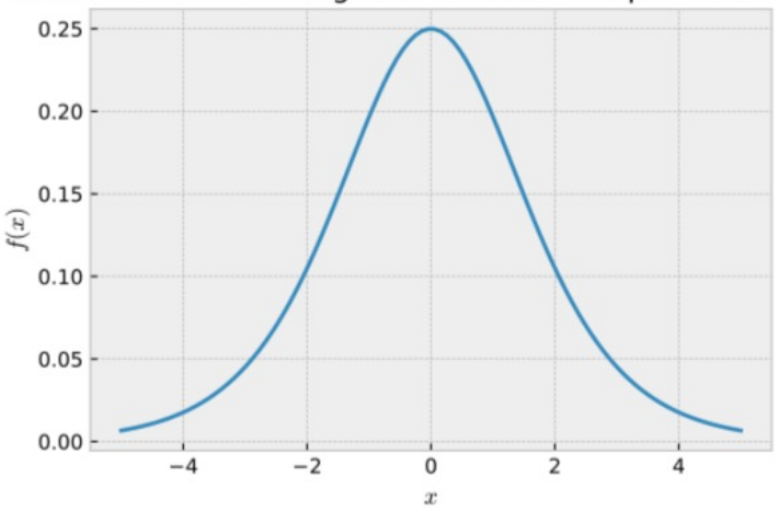
$$y^\star = \operatorname*{argmin}_y \; -y^\top x - H_b(y) \quad \text{subject to} \quad 0 \le y \le 1$$

```
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])
```

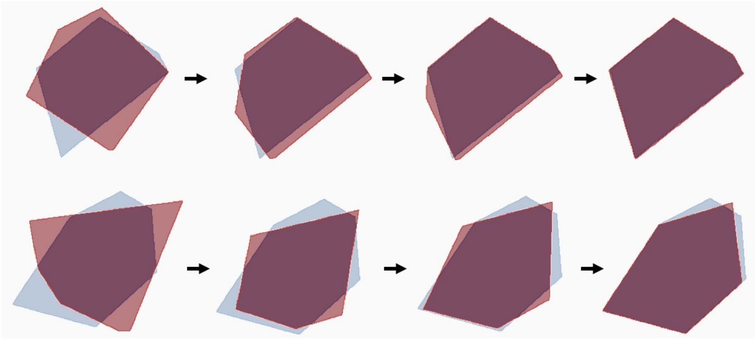
The Sigmoid Function in Optimization Form



The Derivative of the Sigmoid Function in Optimization Form

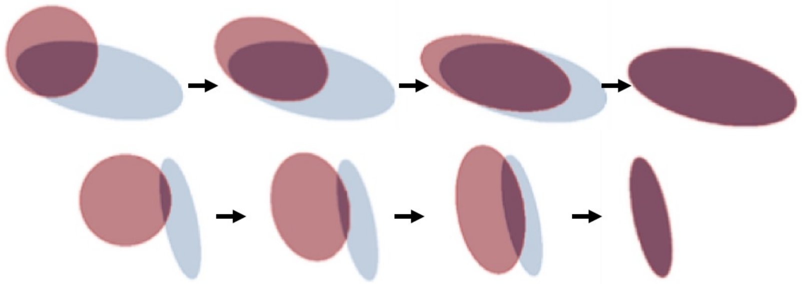


Code example: constraint modeling



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

```
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])
```



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

```
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])
```

Connections to sensitivity and perturbation analysis

Adjoint derivatives for optimization problems have been studied for decades
We have focused on uses for learning, but also widely used for **sensitivity analysis**

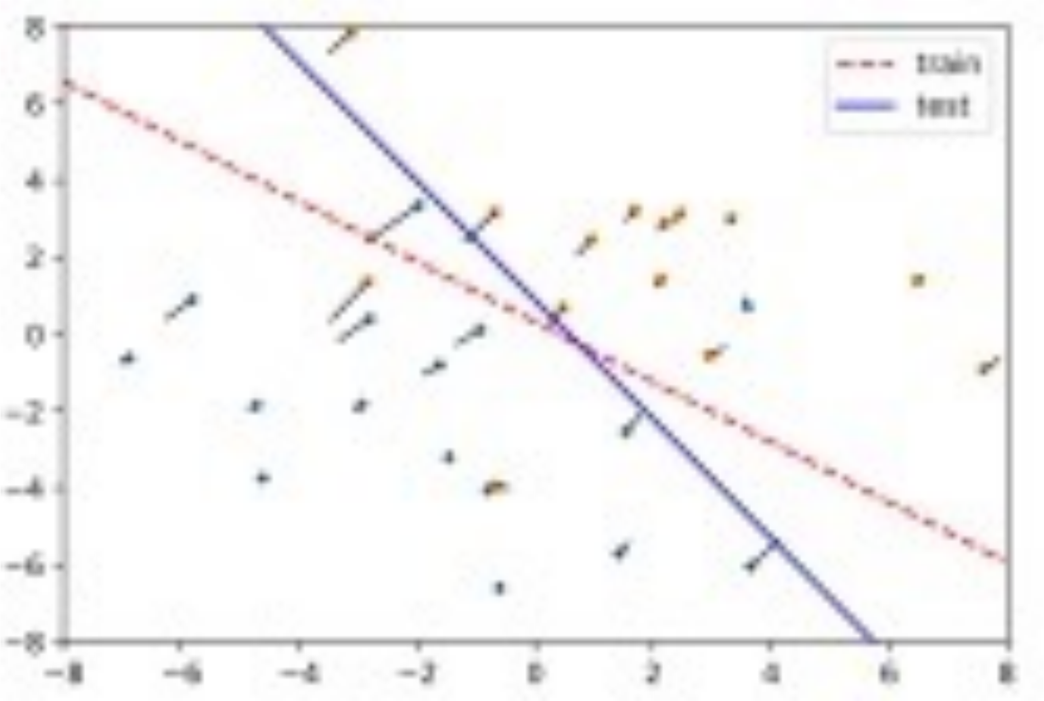
Logistic regression example
Find optimal decision boundary:

$$\theta^\star \in \operatorname*{argmax}_\theta \sum_i \log p_\theta(y_i \mid x_i)$$

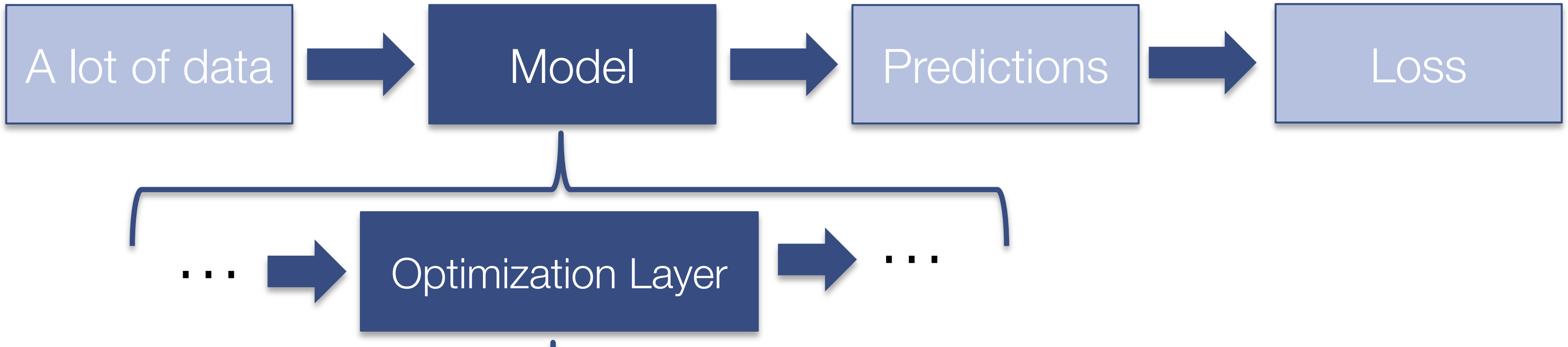
Use derivatives for **sensitivity** to the data points:

$$\frac{\partial \theta^\star}{\partial x_i}$$

How much the data impacts the decision boundary



How do we handle non-convex optimization layers?



$$
\begin{aligned}
z_{i+1} = \quad & \underset{z}{\operatorname{argmin}} \; f_\theta(z, z_i) \\
& \text{subject to} \;\; z \in C_\theta(z, z_i)
\end{aligned}
$$

If non-convex:

1. **Implicitly differentiate** the fixed-point of a non-convex solver
   - Form a **locally convex approximation** to the problem
2. **Unroll gradient steps** $\nabla_z f$ if unconstrained (MAML)
3. **Unroll** steps of another optimizer (differentiable cross-entropy method)

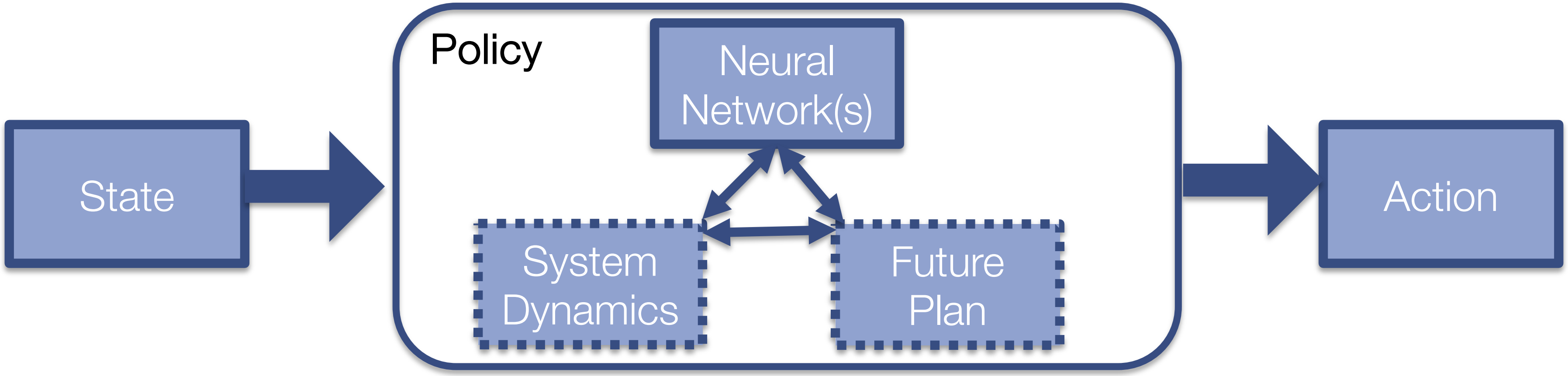
This talk: differentiable optimization-based models

Standard operations as convex optimization layers — warmup

Differentiable optimization theory and practice — core

Differentiable control and objective mismatch — focus application

Should RL policies have a system dynamics model or not?



Model-free RL

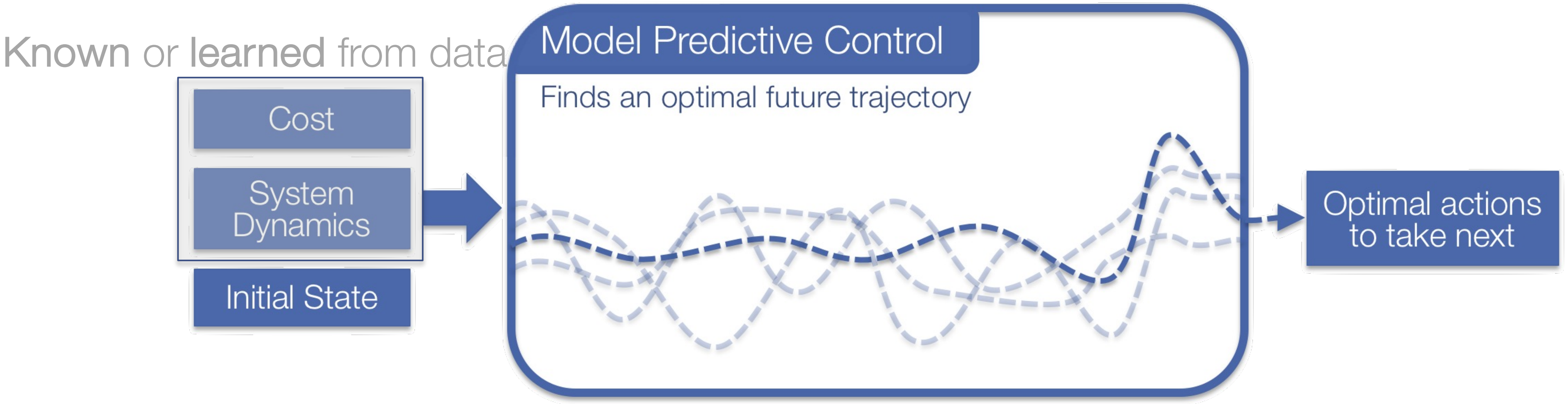
More general, doesn't make as many assumptions about the world

Rife with poor data efficiency and learning stability issues

Model-based RL (or control)

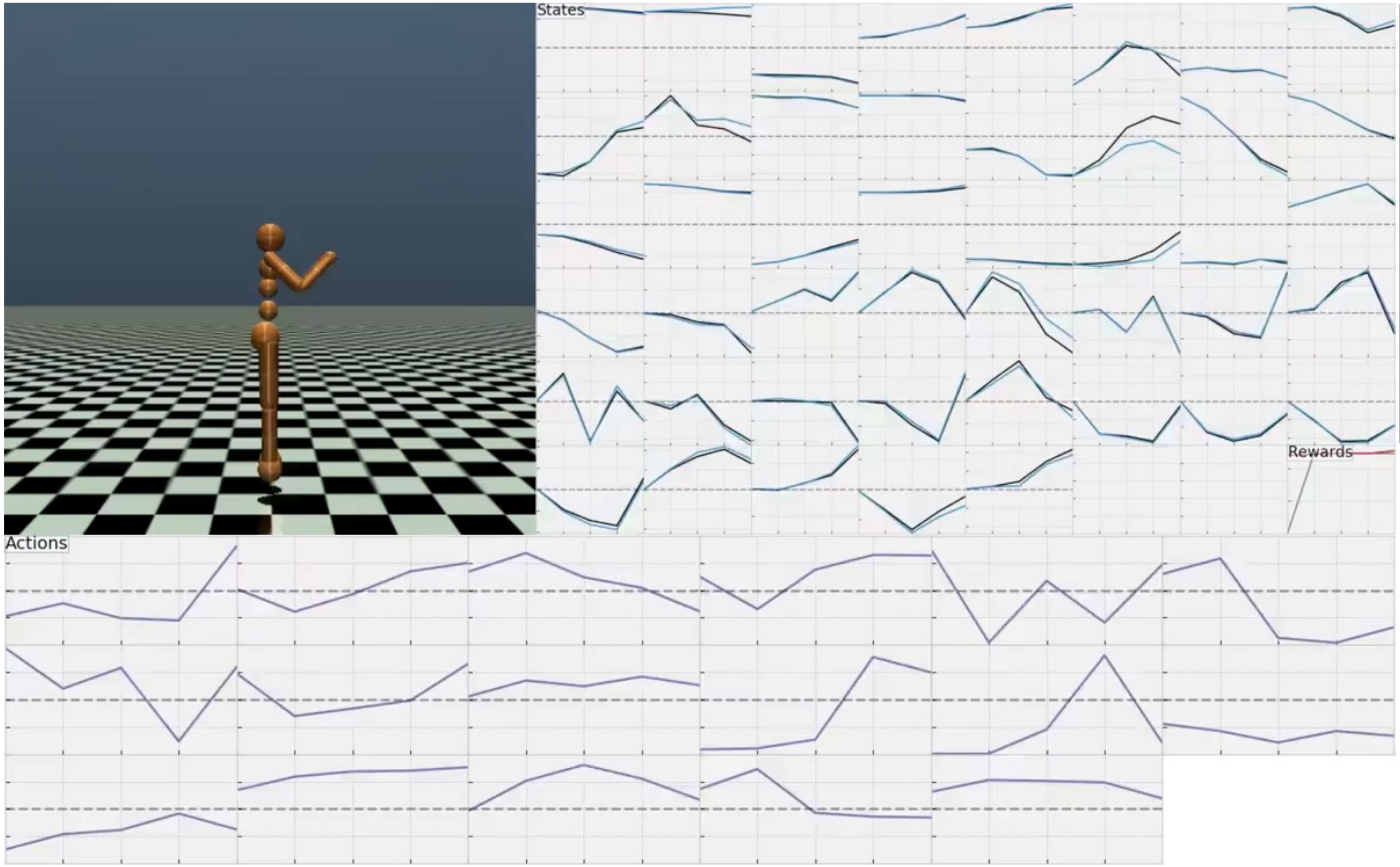
A useful prior on the world if it lies within your set of assumptions

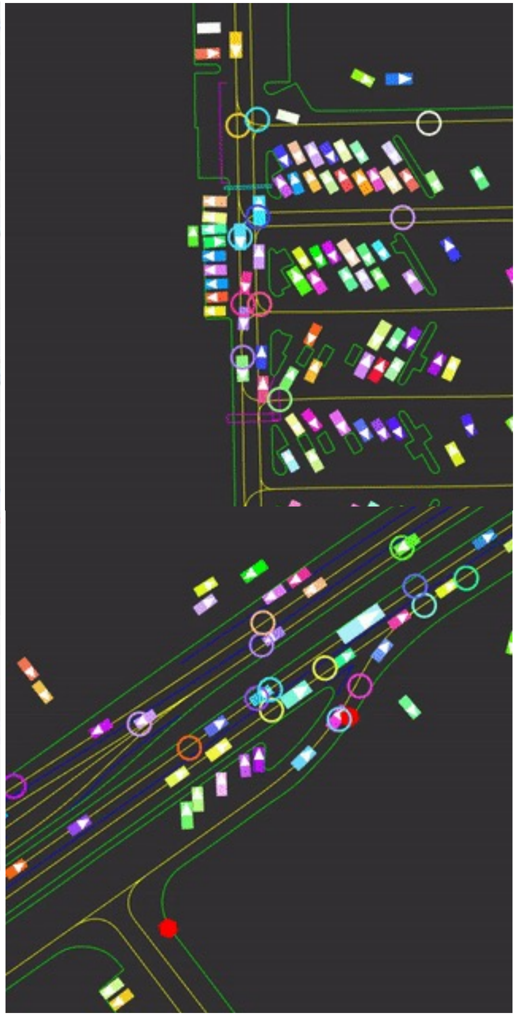
Model Predictive Control



Why model predictive control?

Powerfully deployed in robotic systems, autonomous vehicles, aerospace settings, and beyond





Model Predictive Control

A pure **planning problem** given (potentially non-convex) **cost** and **dynamics**:

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

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

Model Predictive Control with SQP

The standard way of solving MPC is to use **sequential quadratic programming (SQP)**
Form approximations to the cost and dynamics around the current iterate
Repeat until a fixed point is reached, then **implicitly differentiate the fixed point**

… → QP Iterate i → …

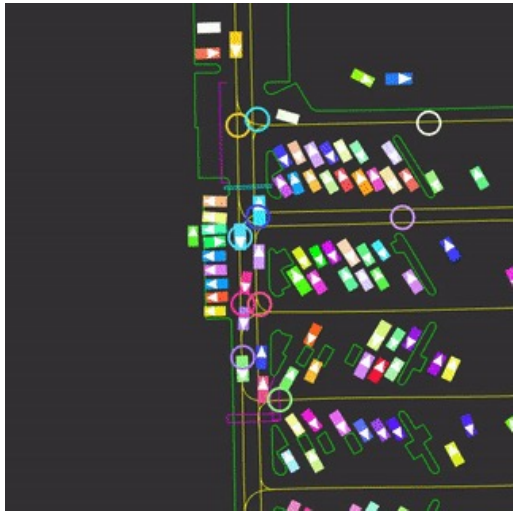
$$
\begin{aligned}
\tau_{1:T}^{i} = \operatorname*{argmin}_{\tau_{1:T}} \quad & \sum_t \tilde{C}_\theta^i(\tau_t) \\
\text{subject to} \quad & x_1 = x_{init} \\
& x_{t+1} = \tilde{f}_\theta^i(\tau_t) \\
& \underline{u} \le u \le \overline{u}
\end{aligned}
$$

Challenge: complex systems are difficult to model

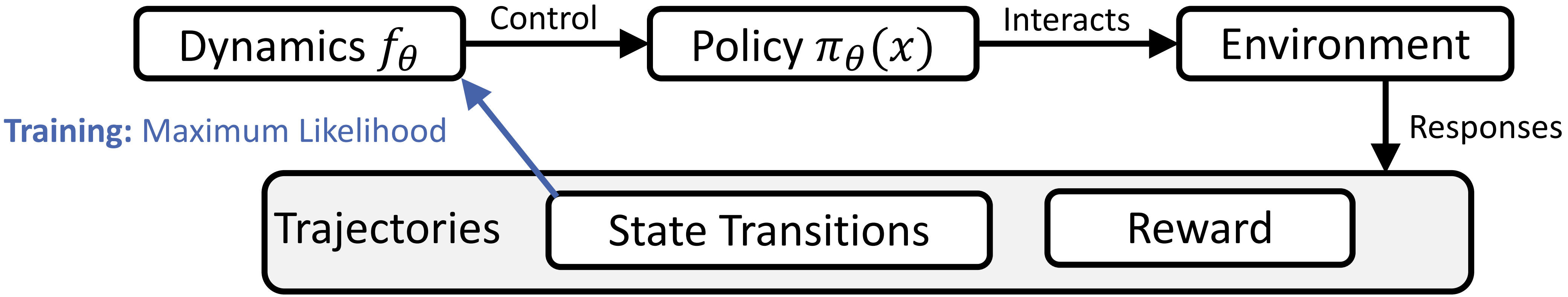
Modeling complex systems in the world is challenging
Often resort to **data-driven approaches** and **learning** to estimate unknown parts

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

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

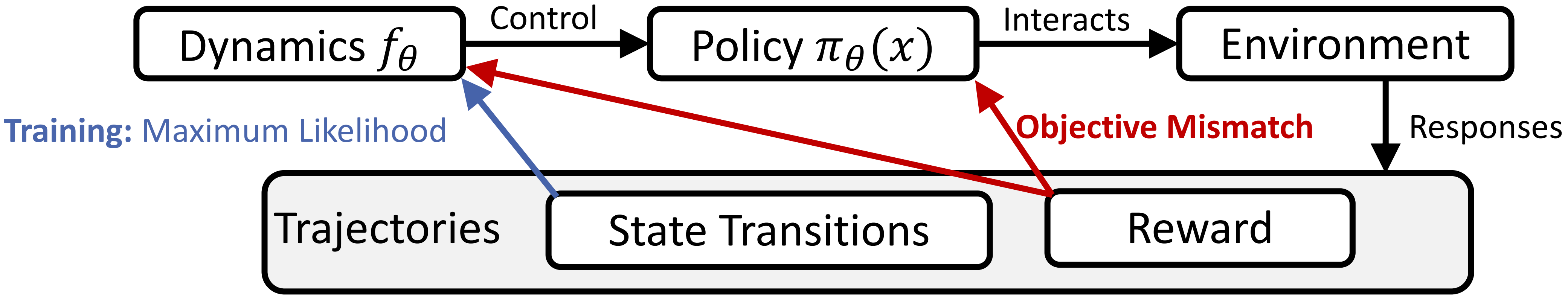


Standard model-based control training pipeline

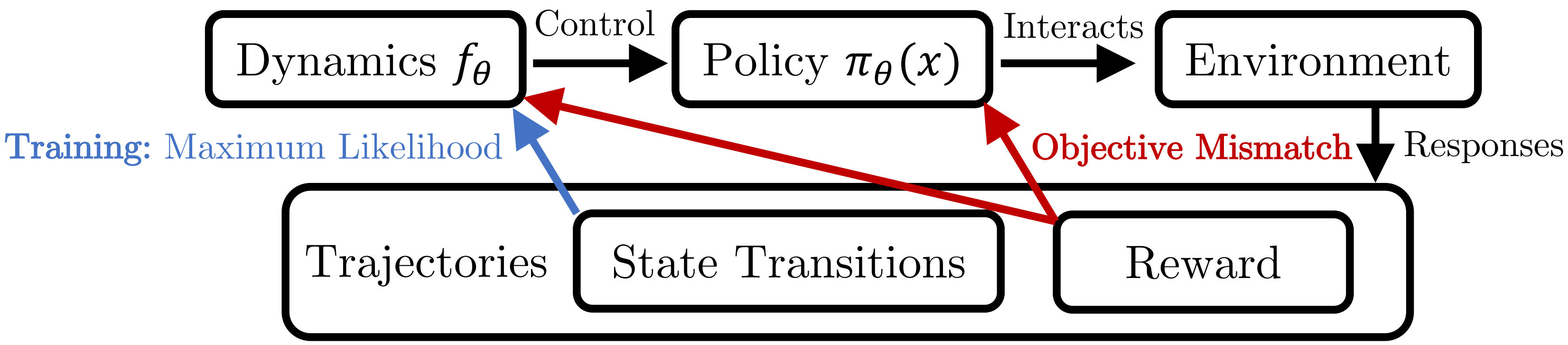


Standard model-based control training pipeline
objective mismatch: dynamics unaware of reward

Similar to problems arising in predict then optimize settings



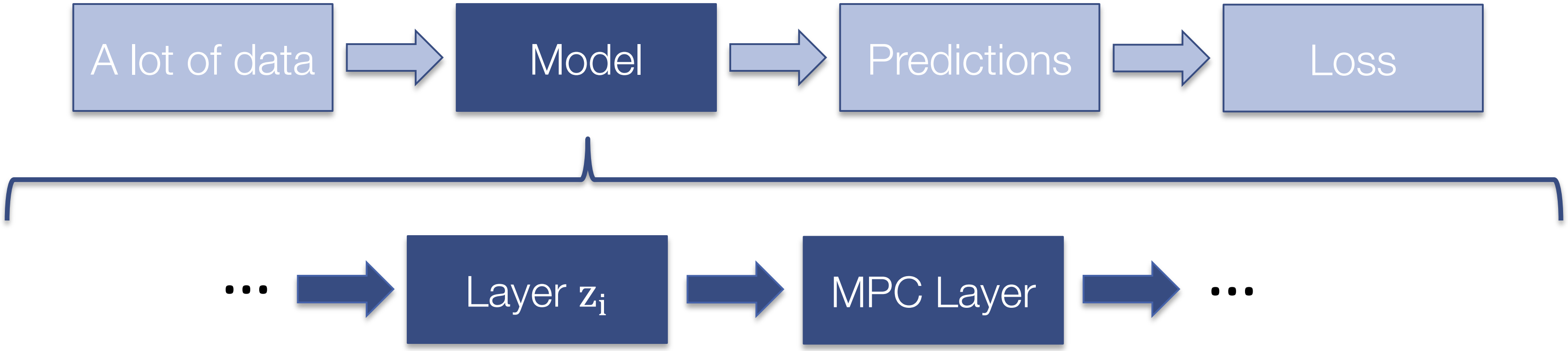
Potential solutions to objective mismatch



1. Re-weight states to focus on **high-value** or high-advantage regions

2. This talk: use **differentiable optimization** to connect the dynamics and reward signal

Differentiable Model Predictive Control



What can we do with this?

Augment neural network policies in model-free algorithms with MPC policies
Replace the unrolled controllers in other settings (hindsight plan, universal planning networks)
Fight objective mismatch by end-to-end learning dynamics
The cost can also be end-to-end learned! No longer need to hard-code in values

Differentiating LQR control is easy

Definition: Linear quadratic regulator

$$\min_{\tau=\{x_t,u_t\}} \sum_t \tau_t^T C_t \tau_t + c_t \tau_t$$
$$\text{s.t.} \quad x_{t+1} = F_t \tau_t + f_t \quad x_0 = x_{\text{init}}$$

Riccati recursion solves the **KKT system:**

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

Backward pass: implicitly **differentiate** the LQR KKT conditions:

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

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!

Differentiating LQR control is easy

Definition: Linear quadratic regulator

$$\min_{\tau=\{x_t,u_t\}} \sum_t \tau_t^T C_t \tau_t + c_t \tau_t$$
$$\text{s.t.} \quad x_{t+1} = F_t \tau_t + f_t \quad x_0 = x_{\text{init}}$$

Riccati recursion solves the **KKT system:**

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

Backward pass: implicitly **differentiate** the LQR KKT conditions:

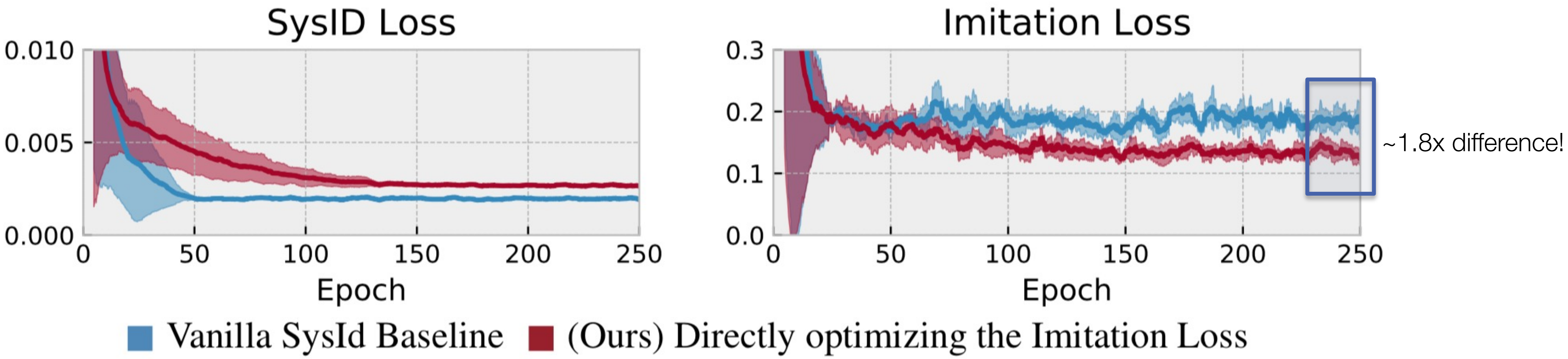
$$\frac{\partial \ell}{\partial C_t} = \frac{1}{2}\left(d_{\tau_t}^\star \otimes \tau_t^\star + \tau_t^\star \otimes d_{\tau_t}^\star\right) \qquad \frac{\partial \ell}{\partial c_t} = d_{\tau_t}^\star \qquad \frac{\partial \ell}{\partial x_{\text{init}}} = d_{\lambda_0}^\star$$
$$\frac{\partial \ell}{\partial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

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

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

True system: pendulum with noise (damping and a wind force)
Approximate model: pendulum without the noise terms

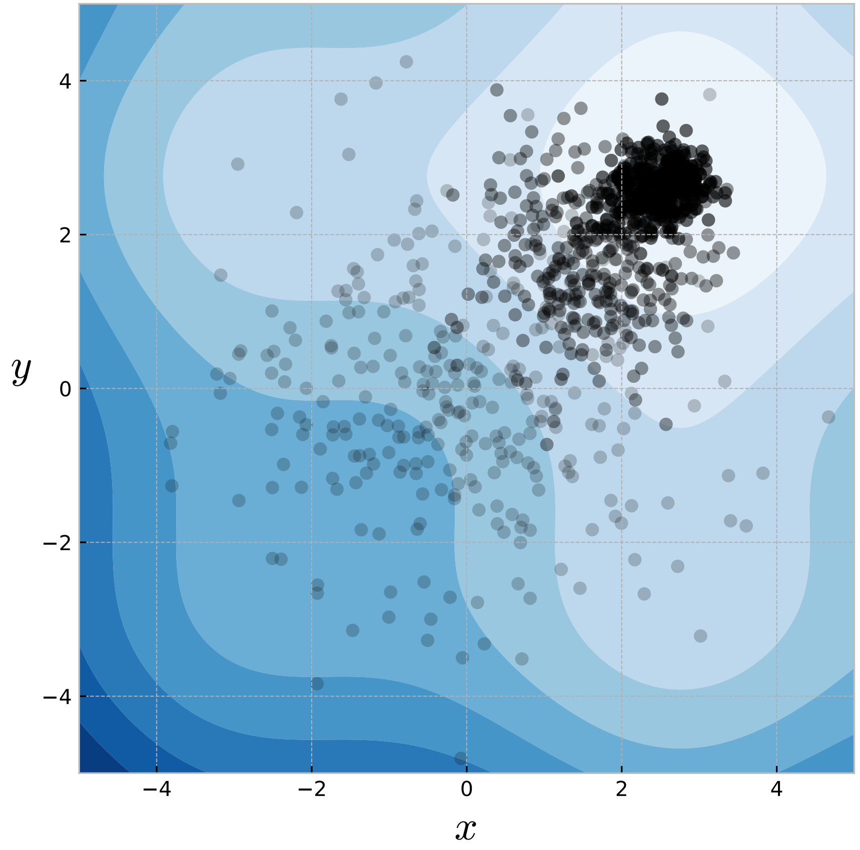


Another control optimizer: the cross-entropy method

Iterative sampling-based optimizer that:
1. **Samples** from the domain
2. **Observes** the function's values
3. **Updates** the sampling distribution

Powerful optimizer for **control** and **model-based RL**

CEM iteratively refining Gaussians



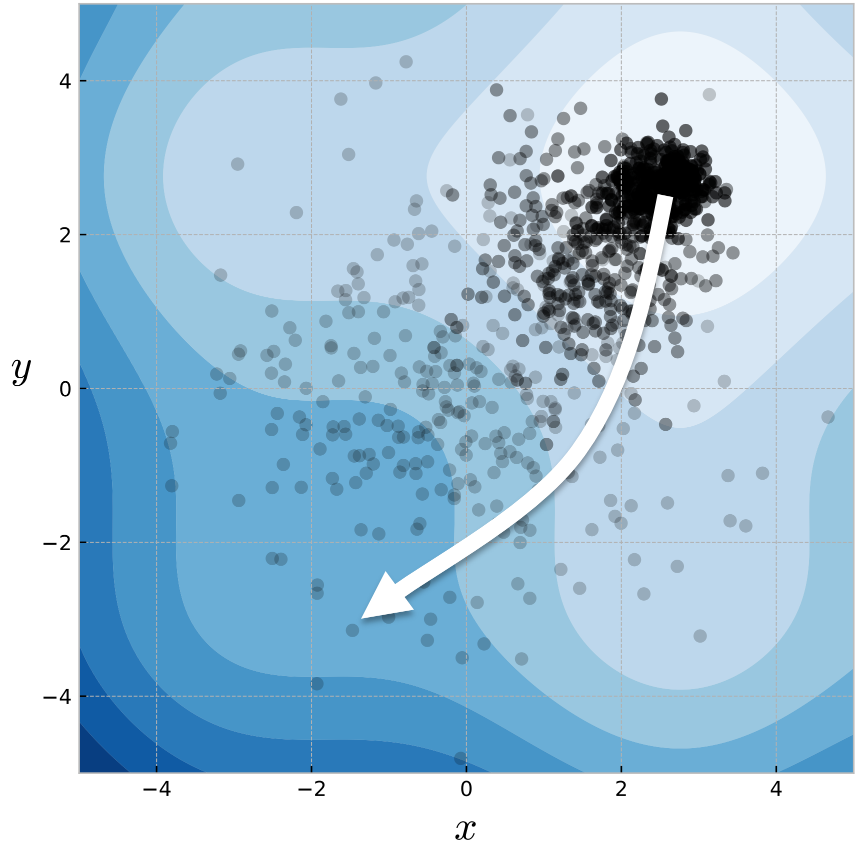
The Differentiable Cross-Entropy Method (DCEM)

Differentiate backwards through the sequence of samples
Using **differentiable top-k** (LML) and **reparameterization**

Useful when a fixed point is **hard to find,** or when unrolling gradient descent hits a local optimum

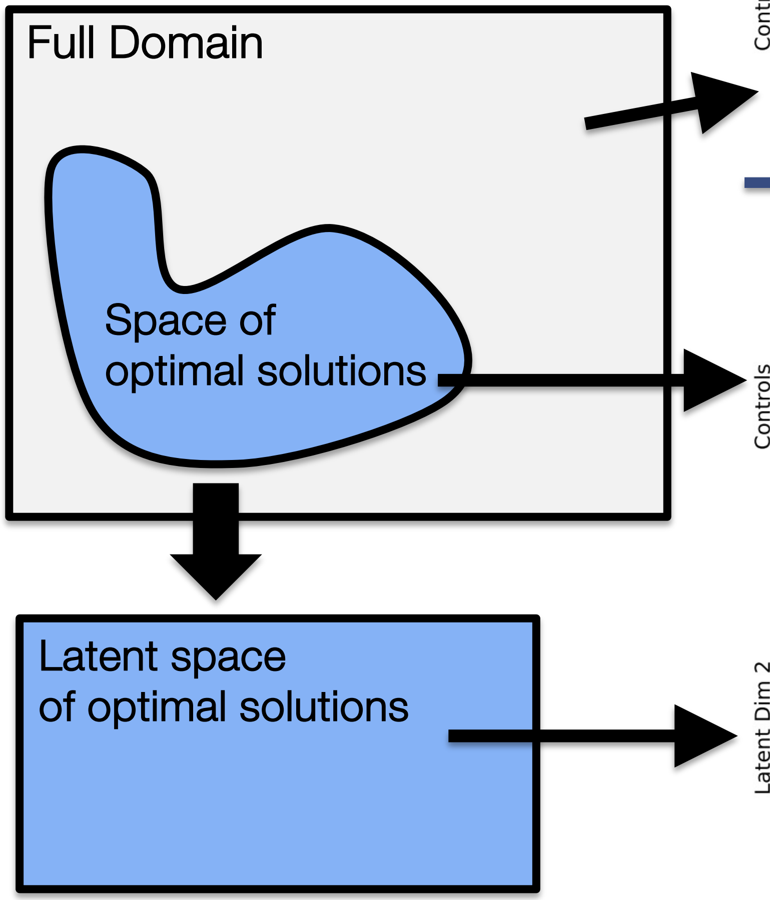
A **differentiable controller** in the RL setting

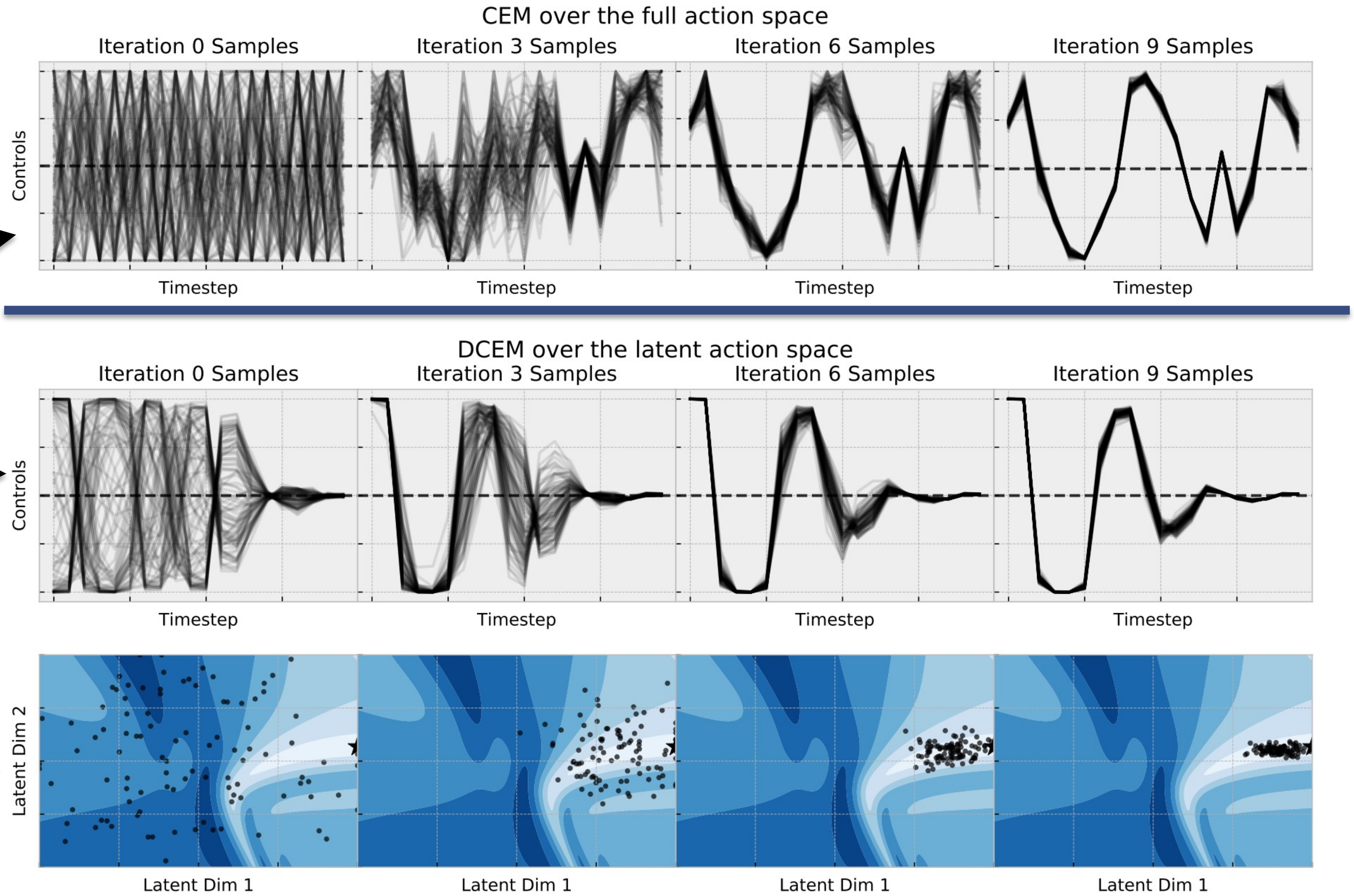
CEM iteratively refining Gaussians



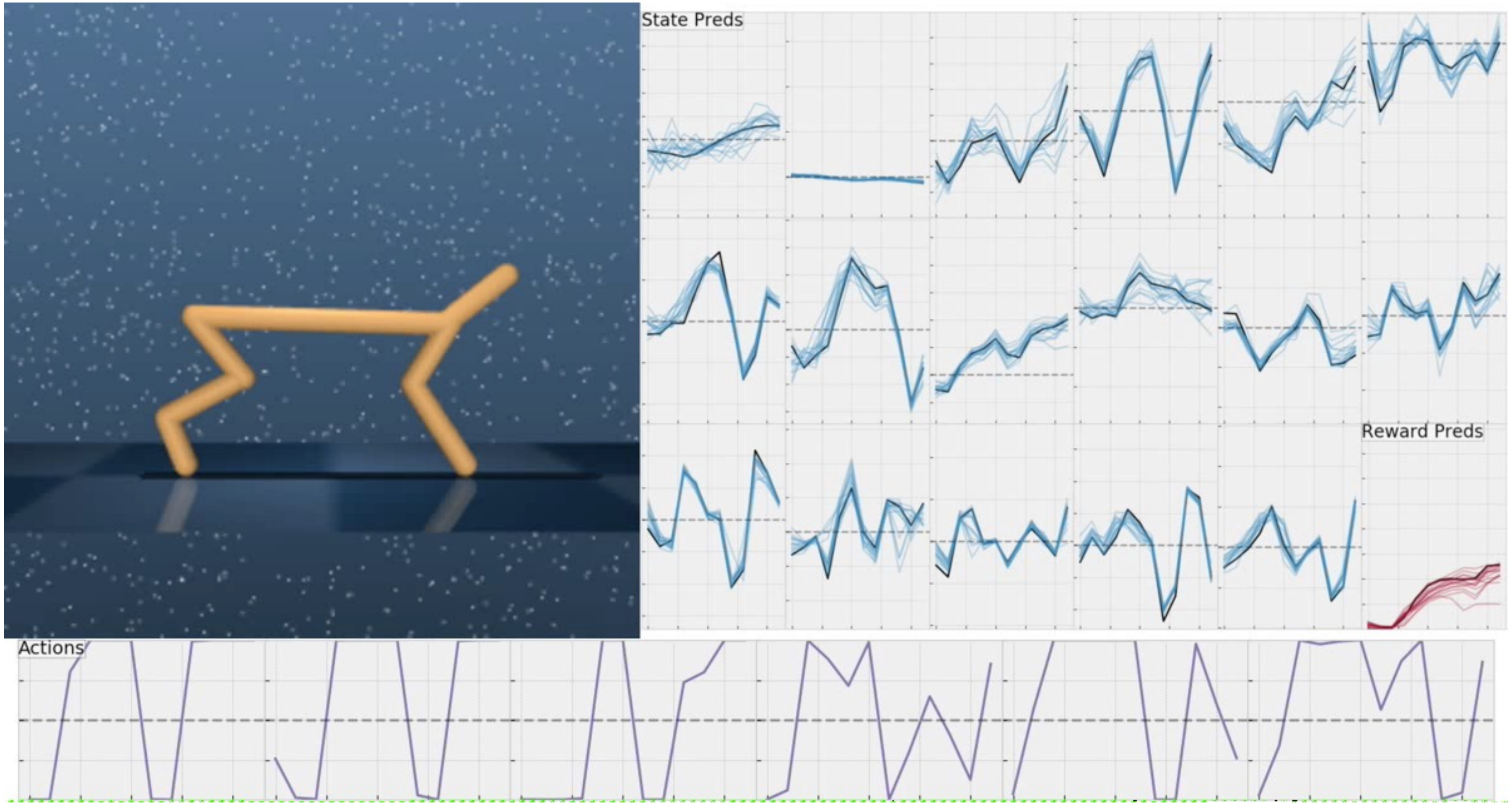
DCEM can learn the solution space structure

$$x^\star = \underset{x \in [0,1]^N}{\operatorname{argmin}} \; f(x)$$





DCEM fine-tunes highly non-convex controllers



sites.google.com/view/diff-cross-entropy-method

Closing thoughts and future directions

Differentiable optimization is a **powerful primitive** to use within larger systems

- **Theoretical** and **engineering** foundations are here
- Can be **propagated through and learned**, just like any layer
- Provides a **perspective to analyze** existing models and layers

Applicable where **optimization expresses non-trivial modeling operations** including game theory, geometry, RL/control, meta-learning, energy-based learning, structured prediction

Extendable far beyond the (mostly convex) continuous Euclidean settings considered here

Differentiable optimization-based modeling for machine learning

Brandon Amos • Meta AI (FAIR)

brandondamos bamos.github.io

Differentiable QPs: OptNet [ICML 2017]
Differentiable Stochastic Opt: Task-based Model Learning [NeurIPS 2017]
Differentiable MPC for End-to-end Planning and Control [NeurIPS 2018]
Differentiable Convex Optimization Layers [NeurIPS 2019]
Differentiable Optimization-Based Modeling for ML [Ph.D. Thesis 2019]
Differentiable Top-k and Multi-Label Projection [arXiv 2019]
Generalized Inner Loop Meta-Learning [arXiv 2019]
Objective Mismatch in Model-based Reinforcement Learning [L4DC 2020]
Differentiable Cross-Entropy Method [ICML 2020]
Differentiable Combinatorial Optimization: CombOptNet [ICML 2021]

Joint with Akshay Agrawal, Shane Barratt, Byron Boots, Stephen Boyd, Roberto Calandra, Steven Diamond, Priya Donti, Ivan Jimenez, Zico Kolter, Nathan Lambert, Jacob Sacks, Omry Yadan, and Denis Yarats