Differentiable Convex Optimization Layers

Akshay Agrawal  Brandon Amos  Shane Barratt  Stephen Boyd
Steven Diamond  J. Zico Kolter

Stanford University  Carnegie Mellon University  Facebook AI
Convex optimization
Convex optimization problems

\[
\begin{align*}
\text{minimize} & \quad f(x; \theta) \\
\text{subject to} & \quad g(x; \theta) \leq 0 \\
& \quad A(\theta)x = b(\theta)
\end{align*}
\]

with variable \( x \in \mathbb{R}^n \)

- objective and inequality constraints \( f_0, \ldots, f_m \) are convex
  \( i.e., \) graphs of \( f_i \) curve upward
- equality constraints are linear
- find a value for \( x \) that minimizes objective, while satisfying constraints
Why convex optimization?

- beautiful, fairly complete, and useful theory
- solution algorithms that work well, in theory and practice
- **many applications** in
  - machine learning, statistics
  - control
  - signal, image processing
  - networking
  - engineering design
  - finance
- ... and many more
How do you solve a convex optimization problem?

use someone else’s (‘standard’) solver
  - your problem *must* be written in a standard form
  - analogous to writing machine code

write your own (custom) solver
  - lots of work, but can take advantage of special structure

use a domain-specific language
  - transforms user-friendly format into solver-friendly standard form
  - extends reach of problems solvable by standard solvers
Domain-specific languages (DSLs)

- DSLs make it easy to specify and solve convex problems
- Grammar and semantics based on a rule from convex analysis \cite{GBY06}
- Examples: CVXPY, CVXR, Convex.jl, CVX
Example

CVXPY is a Python-embedded DSL [DB16; AVD\textsuperscript{+}18]

```python
import cvxpy as cp
import numpy as np

m, n = 30, 20
A = np.random.randn(m, n)
b = np.random.randn(m)

x = cp.Variable(n)
objective = cp.Minimize(cp.sum_squares(A @ x - b))
constraints = [0 <= x, x <= 1]
problem = cp.Problem(objective, constraints)
problem.solve()
```
Neural networks?
Differentiable programming

- Deep learning uses derivatives ("backpropagation") to train neural networks
- A special case of differentiable programming
  - Library of parametrized atomic functions
  - Each atomic function is differentiable
  - A differentiable program is a composition of atomic functions
  - Use chain rule to tune parameters to better achieve some goal
Differentiable programming

This talk: (analytically) differentiating through CVXPY

Why?

- encode prior knowledge (e.g., physics) into a differentiable program
- implement hard constraints or specialized operations in a neural network
- sensitivity analysis
- learn the structure of convex problems
  - learn to control a vehicle
  - model utility functions for agents
  - tune portfolio optimization policies
and more . . .
Differentiating through convex optimization problems
A convex optimization problem with variable $x \in \mathbb{R}^n$ can be parametrized by numerical data $\theta \in \mathbb{R}^p$:

\[
\begin{align*}
\text{minimize} & \quad f_0(x; \theta) \\
\text{subject to} & \quad f_i(x; \theta) \leq 0, \quad i = 1, \ldots, m \\
& \quad A(\theta)x = b(\theta),
\end{align*}
\]

(here, $A$ and $b$ are functions of $\theta$).
Parametrized convex optimization problems

Toy example:

$$\text{minimize } (x - 2\theta)^2,$$

with variable $x \in \mathbb{R}$ and parameter $\theta \in \mathbb{R}$.

- Each choice of $\theta$ induces a new optimization problem.
Parametrized convex optimization problems

Toy example, in CVXPY:

```python
import cvxpy as cp
theta = cp.Parameter()
x = cp.Variable()
objective = (x - 2*theta)**2
problem = cp.Problem(cp.Minimize(objective))

# solve an instance of problem with theta == 3.0
theta.value = 3.0
problem.solve()
```
Solution mapping of a convex optimization problem

- A convex problem can be viewed as a map from parameters to solutions

\[ x^*(\theta) = \arg\min f_0(x; \theta) \]
\[ \text{subject to } f_i(x; \theta) \leq 0, \quad i = 1, \ldots, m \]
\[ A(\theta)x = b(\theta) \]

- \( x^*(\theta) = \{2\theta\} \) for the problem of minimizing \((x - 2\theta)^2\)
- For most problems \( x^*(\theta) \) cannot be written down analytically
When $x^*(\theta)$ is single-valued, we can compute its derivative \cite{ABB+19}

- requires implicitly differentiating optimality conditions of a cone program

For our toy example, $x^*(\theta) = 2\theta$, $\frac{dx^*}{d\theta}(\theta) = 2$

- Can efficiently differentiate through problems, even when solution is not analytical
Differentiating through CVXPY

Solution map of a parametrized CVXPY problem: $x^*(\theta) = (R \circ S \circ C)(\theta)$
- Problem is *canonicalized* (C) to a standard form
- The canonicalized problem is *solved* (S)
- A solution for the original problem is *retrieved* (R)
Differentiating through CVXPY

We can efficiently differentiate through $C$, $S$, and $R$ [AAB+19]

- in fact, we ensure $C$ and $R$ are affine
- we call this *affine-solver-affine* (ASA) form
- we introduce a grammar that ensures problem is in ASA form
Differentiating through CVXPY

Differentiate through the solver ($S$) by differentiating through a \textit{cone program}

- every convex program can be written as a convex cone program
- solving a cone program equivalent to finding a 0 of a map $\mathcal{N}$
- a vector $z$ can be used to construct a solution of a cone program if and only if $\mathcal{N}(z, Q) = 0$, where $Q$ is an embedding of problem data
- if technical conditions are satisfied, the solution $z$ is given by a function of $Q$, and we can compute its derivative
  - application of implicit function theorem
- details in [ABB$^+$19]
Differentiating through CVXPY

```python
import cvxpy as cp

theta = cp.Parameter()
x = cp.Variable()
objective = (x - 2*theta)**2
problem = cp.Problem(cp.Minimize(objective))

theta.value = 3.0
problem.solve(requires_grad=True)
problem.backward()  # backpropagate through solution
print(theta.gradient)  # theta.gradient now equals 2.0
```
Exporting to PyTorch and TensorFlow

cvxpylayers: an open-source library for exporting CVXPY problems to PyTorch and TensorFlow

\[ x^*(\theta) = \underset{x}{\text{argmin}} \ f(x; \theta) \]
subject to \( g(x; \theta) \leq 0 \)
\( h(x; \theta) = 0 \)
Exporting to PyTorch and TensorFlow

```python
from cvxpylayers.torch import CvxpyLayer
import torch

# export to torch (or tensorflow) with just one line
layer = CvxpyLayer(problem, parameters=[theta], variables=[x])

theta_tch = torch.tensor(3.0, requires_grad=True)
soln = layer(theta_tch)[0]
soln.backward()
print(theta_tch.grad)
```
Examples
Learning convex-optimization control policies

Consider a stochastic control problem

\[
\begin{align*}
\text{minimize} \quad & \lim_{T \to \infty} \mathbb{E} \left[ \frac{1}{T} \sum_{t=0}^{T-1} \|x_t\|_2^2 + \|\phi(x_t)\|_2^2 \right] \\
\text{subject to} \quad & x_{t+1} = Ax_t + B\phi(x_t) + \omega_t, \quad t = 0, 1, \ldots, \\
& \|\phi(x_t)\|_\infty \leq u_{\text{max}}, \quad t = 0, 1, \ldots,
\end{align*}
\]

- \(x_t \in \mathbb{R}^n\) is the state, \(\phi(x_t) \in \mathbb{R}^m\) the control, \(\omega_t\) is random
- expectation is over \(\omega_t\) and \(x_0\)
- optimization is over states \(x_t\) and policy \(\phi : \mathbb{R}^n \to \mathbb{R}^m\)
- this problem is computationally intractable
Learning convex-optimization control policies

- Common heuristic for stochastic control is approximate dynamic programming (ADP), which parametrizes $\phi$
- ADP replaces minimization over functions $\phi$ with minimization over parameters
- Differentiable convex optimization layers can be used in an ADP method
Learning convex-optimization control policies

- Take $\phi$ to be the solution of a convex optimization problem:

$$
\phi(x_t) = \arg\min_u \left\| P^{1/2}(Ax_t + Bu) \right\|_2^2 + \left\| u \right\|_2^2
\quad \text{subject to} \quad \left\| u \right\|_{\infty} \leq u_{\text{max}}
$$

- Here, the parameter is $P^{1/2} \in \mathbb{R}^{n \times n}$
- Learning method:
  - simulate the system with the policy in the loop
  - approximate expected cost
  - update $P^{1/2}$ using gradient descent
Learning convex-optimization control policies

![Graph showing expected cost over iterations for different policies: clf_lqr, clipped LQR, clf_lb (lower bound), and lower bound. The graph plots expected cost on a log scale against iteration number.]
Learning convex-optimization control policies

- Many real-world applications, including
  - finance
  - vehicle control
  - supply-chain management

- See our paper: “Learning convex-optimization control policies”
Data poisoning attack

- given data \((x_i, y_i), i = 1, \ldots, m\)
  - \(x_i \in \mathbb{R}^n\) are feature vectors
  - \(y_i \in \{0, 1\}\) are associated boolean outcomes
- linear classifier: \(\hat{y} = 1[\beta^T x \geq 0]\)
- find optimal weights \(\beta^*\) by minimizing

\[
\frac{1}{m} \sum_i L(\beta; x_i, y_i) + r(\beta)
\]

- \(L(\beta; x_i, y_i) = \log(1 + \exp(\beta^T x_i + b)) - y_i \beta^T x_i\) is the logistic loss
- \(r(\beta) = 0.1\|\beta\|_1 + 0.1\|\beta\|_2^2\) is elastic-net regularization
- adversary seeks to increase test loss \(L^{\text{test}}(\beta^*)\) by (just barely) perturbing training data
Data poisoning attack

- $\beta^*$ is a solution to a convex problem parametrized by $x_i$
- $\nabla_{x_i} \mathcal{L}^{\text{test}}(\beta^*)$ gives direction $x_i$ should be moved to achieve greatest increase in test loss
Data poisoning attack

```python
X = cp.Parameter((m, n))
beta = cp.Variable(n)
log_likelihood = cp.sum(
    cp.multiply(Y, X @ beta) - cp.logistic(X @ beta)
)
r = 0.1*cp.norm(beta, 1) + 0.1*cp.norm(beta, 2)**2)
problem = cp.Problem(cp.Minimize(-log_likelihood/m + r))
fit_logreg = CvxpyLayer(problem, parameters=[X], variables=[beta])
beta_star = fit_logreg(X_train)[0]
test_loss = compute_loss(bbeta_star, X_test, Y_test)
test_loss.backward()
```
Summary

Our software makes it easy to differentiate through DSLs for convex optimization, letting you pair

- convex optimization
  - rich modelling capabilities
  - large number of applications
  - efficient solution algorithms
  - mature, high-level software libraries
- with machine learning
Software

https://github.com/cvxgrp/cvxpylayers

https://github.com/cvxgrp/cvxpy


Additional examples
Tuning a Markowitz policy

Figure: left: untuned; right: policy with tuned constraints, mean and covariance
Tracking a vehicle trajectory

Figure: left: untrained path; middle: trained path; right: expected cost histogram.