Chapter 7

Differentiable cvxpy Optimization Layers

This is an excerpt of Chapter 7 from https://github.com/bamos/thesis

In this chapter, we show how to turn the cvxpy modeling language [DB16] into a differentiable optimization layer and implement our method in PyTorch [Pas+17]. This allows users to express convex optimization layers in the intuitive cvxpy modeling language without needing to manually implement the backward pass.

7.1 Introduction

This thesis has presented differentiable optimization layers as a powerful class of operations for end-to-end learning that allow more specialized domain knowledge to be integrated into the modeling pipeline in a differentiable way. Convex optimization layers can be represented as

$$z_{i+1} = \operatorname*{argmin}_{z} f_\theta(z, z_i) \text{ s.t. } z \in \mathcal{C}_\theta(z_i) \tag{7.1}$$

where z_i is the previous layer, f is a convex objective function parameterized by θ, and $\mathcal{C}$ is a convex constraint set. From the perspective of end-to-end learning, convex optimization layers can be seen as a module that outputs z_{i+1} and has parameters θ that can be learned with gradient descent. We note that the convex case captures many of the applications above, and can be used as a building block for differentiable non-convex optimization problems.

Implementing optimization layers can be non-trivial as explicit closed-form solutions typically do not exist. The forward pass needs to call into an optimization problem solver and the backward pass typically *cannot* leverage automatic differentiation. The backwards pass is usually implemented by implicitly differentiating the KKT conditions of the optimization problem as done in bilevel optimization [Gou+16; KP13], sensitivity analysis [Ber99; FI90; BS13], and in our OptNet approach **??**. Thus to implement an optimization layer, users have to manually implement the backwards pass, which is cumbersome and error-prone, or use an existing optimization problem layer such as the differentiable QP layer from **??**, which is not capable of exploiting problem-specific structures, uses dense operations, and requires the user to manually transform their problem into a standard form.

We make `cvxpy` differentiable with respect to the `Parameter` objects provided to the optimization problem by making internal `cvxpy` components differentiable. This involves differentiating the reduction from the `cvxpy` language to the problem data of a cone program in standard form and then differentiating through the cone program. We show how to differentiate through cone programs by implicitly differentiating the residual map from Busseti, Moursi, and Boyd [BMB18], which is of standalone interest as this shows how to differentiate through optimization problems with non-polytope constraints.

7.2 Background

7.2.1 The `cvxpy` modeling language

`cvxpy` [DB16] is a domain-specific modeling language based on disciplined convex programming [GBY06] that allows users to express optimization problems in a more natural way than the standard form required my most optimization problem solvers. `cvxpy` works by transforming the optimization problem from their domain-specific language to a standard (or canonical) form that is passed into a solver. This inner canonicalized problem is then solved and the results are returned to the user. In this chapter, we focus on the canonicalization to a cone program, which is one of the most commonly used modes as most convex optimization problems can be expressed as a cone program, although we note that our method can be applied to other `cvxpy` solvers. Figure 7.1 overviews the relevant `cvxpy` components.

7.2.2 Cone Preliminaries

A set $\mathcal{K}$ is a *cone* if for all $x \in \mathcal{K}$ and $t > 0$, $tx \in \mathcal{K}$. The *dual cone* of a cone $\mathcal{K}$ is

$$\mathcal{K}^* = \left\{ y \;\middle|\; \inf_{x \in \mathcal{K}} y^\top x \geq 0 \right\}.$$

Commonly used cones include the nonnegative orthant $\left\{ x \;\middle|\; x \geq 0 \right\}$, second-order cone $\left\{ (x,t) \in \mathbb{R}^n_+ \;\middle|\; t \geq ||x||_2 \right\}$, positive semidefinite cone $\{X = X^\top \succeq 0\}$, and exponential cone

$$\left\{ (x,y,z) \;\middle|\; y > 0, ye^{x/y} \leq z \right\} \cup \left\{ (x,0,z) \;\middle|\; x \leq 0, z \geq 0 \right\} \tag{7.2}$$

We can also create a cone from the Cartesian products of simpler cones as $\mathcal{K} = \mathcal{K}_1 \times \ldots \times \mathcal{K}_p$.

7.2.3 Cone Programming

Most convex optimization problems can be represented and efficiently solved as a cone program that uses the nonnegative orthant, second-order cone, positive semidefinite cone, and exponential cones. This applicability makes them a commonly used internal solver for `cvxpy`, which implements many of the well-known transformations from problems to their conic form. In the following we state properties of cone programs and useful definitions for this chapter. More details about cone programming can be found in Boyd and Vandenberghe [BV04], Ben-Tal and Nemirovski [BN01], Busseti, Moursi, and Boyd [BMB18], O'Donoghue, Chu, Parikh, and Boyd [ODo+16], Lobo, Vandenberghe, Boyd, and Lebret [Lob+98], and Alizadeh and Goldfarb [AG03].

In their primal (P) and dual (D) forms, cone programs can be represented as

$$\text{(P)}\quad x^\star, s^\star = \begin{array}{ll} \operatorname{argmin}_{x,s} & c^\top x \\ \text{subject to} & Ax + s = b \\ & s \in \mathcal{K} \end{array} \qquad \text{(D)}\quad y^\star = \begin{array}{ll} \operatorname{argmax}_{y} & b^\top y \\ \text{subject to} & A^\top y + c = 0 \\ & y \in \mathcal{K}^* \end{array} \tag{7.3}$$

where $x \in \mathbb{R}^n$ is the *primal variable*, $s \in \mathbb{R}^m$ is the *primal slack variable*, $y \in \mathbb{R}^m$ is the *dual variable.* and $\mathcal{K}$ is a nonempty, closed, convex cone with dual cone $\mathcal{K}^*$.

The KKT optimality conditions. The Karush–Kuhn–Tucker (KKT) conditions for the cone program in Equation (7.3) provide necessary and sufficient conditions for optimality and are defined by

$$Ax + s = b, \quad A^\top y + c = 0, \quad s \in \mathcal{K}, \quad y \in \mathcal{K}^*, \quad s^\top y = 0. \tag{7.4}$$

The complimentary slackness condition $s^\top y = 0$ can alternatively be captured with a condition that makes the duality gap zero $c^\top x + b^\top y = 0$.

The homogenous self-dual embedding. Ye, Todd, and Mizuno [YTM94] converts the primal and cone dual programs in Equation (7.3) into a single feasibility problem called the homogenous self-dual embedding, which is defined by

$$Qu = v, \quad u \in \boldsymbol{\mathcal{K}}, \quad v \in \boldsymbol{\mathcal{K}}^*, \quad u_{m+n+1} + v_{m+n+1} > 0, \tag{7.5}$$

where

$$\boldsymbol{\mathcal{K}} = \mathbb{R}^n \times \mathcal{K}^* \times \mathbb{R}_+, \quad \boldsymbol{\mathcal{K}}^* = \{0\}^n \times \mathcal{K} \times \mathbb{R}_+,$$

and Q is the skew-symmetric matrix

$$Q = \begin{bmatrix} 0 & A^\top & c \\ -A & 0 & b \\ -c^\top & -b^\top & 0 \end{bmatrix}.$$

A solution to this embedding problem $(u^\star, v^\star)$ can be used to determine the solution of a conic problem, or to certify the infeasibilty of the problem if a solution doesn't exist. If a solution exists, then $u^\star = (x^\star/\tau, y^\star/\tau, \tau)$ and $v^\star = (0, s^\star/\tau, 0)$.

The conic complementarity set. The *conic complementarity set* is defined by

$$\mathcal{C} = \left\{(u, v) \in \boldsymbol{\mathcal{K}} \times \boldsymbol{\mathcal{K}}^* \;\middle|\; u^\top v = 0\right\}. \tag{7.6}$$

We denote the Euclidean projection onto $\boldsymbol{\mathcal{K}}$ with Π and the Euclidean projection onto $-\boldsymbol{\mathcal{K}}^*$ with Π^*. Moreau [Mor61] shows that $\Pi^* = I - \Pi$.

Minty's parameterization of the complementarity set. Minty's parameterization $M : \mathbb{R}^{m+n+1} \to \mathcal{C}$ of $\mathcal{C}$ is defined by $M(z) = (\Pi z, -\Pi^* z)$. This parameterization is invertible with $M^{-1}(u, v) = u - v$. See Rockafellar [Roc70, Corollary 31.5.1] and Bauschke and Combettes [BC17, Remark 23.23(i)] for more details. The homogeneous self-dual embedding can be expressed using Minty's parameterization as $-\Pi^* z = Q\Pi z$ where $z_{m+n+1} \neq 0$.

The residual map of Minty's parameterization. Busseti, Moursi, and Boyd [BMB18] defines the *residual map* of Minty's parameterization $\mathcal{R} : \mathbb{R}^{m+n+1} \to \mathbb{R}^{m+n+1}$ as

$$\mathcal{R}(z) = Q\Pi z + \Pi^* z = ((Q - I)\Pi + I)z. \tag{7.7}$$

and shows how to compute the derivative of it when Π is differentiable at z as

$$\mathsf{D}_z \mathcal{R}(z) = (Q - I)\mathsf{D}_z \Pi(z) + I, \tag{7.8}$$

where $z \in \mathbb{R}^{m+n+1}$. The cone projection differentiation $\mathsf{D}_z \Pi(z)$ can be computed as described in Ali, Wong, and Kolter [AWK17].

The Splitting Conic Solver (SCS). SCS [ODo+16] is an efficient way of solving general cone programs by using the alternating direction method of multipliers (ADMM) [Boy+11] and is a commonly used solver with `cvxpy`. In the simplified form, each iteration of SCS consists of the following three steps:

$$\begin{aligned} \tilde{u}^{k+1} &= (I + Q)^{-1}(u^k + v^k) \\ u^{k+1} &= \Pi\left(\tilde{u}^{k+1} - v^k\right) \\ v^{k+1} &= v^k - \tilde{u}^{k+1} + u^{k+1}. \end{aligned} \tag{7.9}$$

The first step projects onto an affine subspace, the second projects onto the cone and the last updates the dual variable. In this paper we will mostly focus on solving the affine subspace projection step. O'Donoghue, Chu, Parikh, and Boyd [ODo+16, Section 4.1] shows that the affine subspace projection can be reduced to solving linear systems of the form

$$\begin{bmatrix} I & -A^\top \\ -A & -I \end{bmatrix} \begin{bmatrix} z_x \\ -z_y \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \end{bmatrix}, \tag{7.10}$$

which can be re-written as

$$z_x = (I + A^\top A)^{-1}(w_x - A^\top w_y), \quad z_y = w_y + Az_x. \tag{7.11}$$

7.3 Differentiating `cvxpy` and Cone Programs

We have created a differentiable `cvxpy` layer by making the relevant components differentiable, which we visually show in Figure 7.1. We make the transformation from the problem data in the original form to the problem data of the canonicalized cone problem differentiable by replacing the numpy operations for this component with PyTorch [Pas+17] operations. We then pass this data into a differentiable cone program solver, which we show how to create in Section 7.3.1 by implicitly differentiating the residual map of Minty's parameterization for the backward pass. The solution of this cone program can then be mapped back up to the original problem space in a differentiable way and returned.

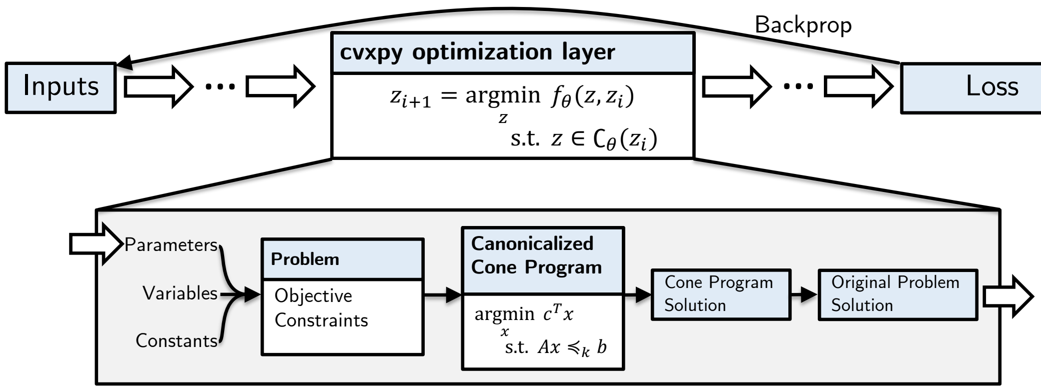

Backprop
Inputs
cvxpy optimization layer
$z_{i+1} = \text{argmin}_z \; f_\theta(z, z_i)$
s.t. $z \in C_\theta(z_i)$
Loss
Parameters
Variables
Constants
Problem
Objective
Constraints
Canonicalized Cone Program
$\text{argmin}_x \; c^T x$
s.t. $Ax \preccurlyeq_k b$
Cone Program Solution
Original Problem Solution

Figure 7.1: Summary of our differentiable `cvxpy` layer that allows users to easily turn most convex optimization problems into layers for end-to-end machine learning.

7.3.1 Differentiating Cone Programs

We consider the argmin of the primal cone program in Equation (7.3) as a function that maps from the problem data $\theta = \{c, A, b\}$ to the solution $x^\star$. The approach from **??** that differentiates through convex quadratic programs by implicitly differentiating the KKT conditions is difficult to use for cone programs. This is because the cone constraints in the KKT conditions in Equation (7.4) make it difficult to form a set of implicit functions. Instead of implicitly differentiating the KKT conditions, we show how to similarly apply implicit differentiation to the residual map of Minty's parameterization shown in Busseti, Moursi, and Boyd [BMB18] to compute the derivative $\partial x^\star/\partial\theta$. Furthermore for backpropagation, the full Jacobian is expensive and unnecessary to form and we show how to efficiently compute $\partial\ell/\partial\theta$ given $\partial\ell/\partial x^\star$.

Implicit differentiation of the residual map

We assume that we have solved the forward pass of the cone program in Equation (7.3) and have a solution $x^\star, s^\star, y^\star$. We now show how to compute $\partial x^\star/\partial\theta$. This derivation was concurrently considered and done by Agrawal, Barratt, Boyd, Busseti, and Moursi [Agr+19].

We construct $u^\star = (x^\star, y^\star, 1)$, and $v^\star = (0, s^\star, 0)$, and $z^\star = u^\star - v^\star$. The residual map of Minty's parameterization is zero, $R(z^\star) = 0$, and forms a set of implicit equations that describe the solution mapping. Implicit differentiation can be done as described in Dontchev and Rockafellar [DR09] with

$$\mathsf{D}_\theta z^\star = -\left(\mathsf{D}_z \mathcal{R}(z^\star)\right)^{-1} \mathsf{D}_\theta \mathcal{R}(z^\star). \tag{7.12}$$

$\left(\mathsf{D}_z \mathcal{R}(z^\star)\right)^{-1}$ can be computed as described in [BMB18] and $\mathsf{D}_\theta \mathcal{R}(z^\star)$ can be analytically computed. We consider the scaling factor $\tau = z_{m+n+1} = 1$ to be a constant because a solution to the cone program exists. Finally, applying the chain rule to $u^\star = \Pi z^\star$ gives

$$\mathsf{D}_\theta u^\star = (\mathsf{D}_z \Pi z)\mathsf{D}_\theta z^\star. \tag{7.13}$$

We note that implicitly differentiating the residual map captures implicit differentiation of the KKT conditions as a special case for simple cones such as the zero cone and non-negative orthant.

The linear system in Equation (7.12) can be expensive to solve. In special cases such as quadratic programs and LQR problems that we discussed in **??** and **??**, respectively, this system can be interpreted as the solution to another convex optimization problem and efficiently solved with a method similar to the forward pass. This connection is made by interpreting the linear system solve as a KKT system solve that represents another optimization problem. However for general cone programs it is more difficult to interpret this linear system as a KKT system because of the cone projections and therefore it is more difficult to interpret this linear system solve as an optimization problem.

7.4 Implementation

7.4.1 Forward Pass: Efficiently solving batches of cone programs with SCS and PyTorch

Naïvely implemented optimization layers can become computational bottlenecks when used in a machine learning pipeline that requires efficiently processing minibatches of data. This is because most other parts of the modeling pipeline involve operations such as linear maps, convolutions, and activation functions that can quickly be executed on the GPU to exploit data parallelism across the minibatch. Most off-the-shelf optimization problem solvers are designed for the setting of solving a single problem at a time and are not easily able to be plugged into the batched setting required when using optimization layers.

To overcome the computational challenges of solving batches of cone programs concurrently, we have created a batched PyTorch and potentially GPU-backed backend for the Splitting Conic Solver (SCS) [ODo+16]. The bottleneck of the SCS iterates in Equation (7.9) is typically in the subspace projection part that solves linear systems of the form

$$\tilde{u}^{k+1} = (I+Q)^{-1}(u^k + v^k) \tag{7.14}$$

We have added a new linear system solver backend to the official SCS C implementation that calls back up into Python to solve this linear system.

Our cone program layer implementation offers the following modes for solving a batch of N cone programs represented in standard form as in Equation (7.3) with as $\theta_i = \{A_i, b_i, c_i\}$ for $i \in \{1, \ldots, N\}$ with SCS. As common in practice, we assume that the cone programs have the structure and use the same cones but have different problem data θ_i. We empirically compare these modes in Section 7.6.

Vanilla SCS, serialized. This is a baseline mode that is the easiest to implement and sequentially iterates through the problems θ_i. This lets us use the vanilla SCS sparse direct and indirect linear system solvers on the CPU and CUDA, but does not take advantage of data parallelism.

Vanilla SCS, batched. This is another baseline mode that comes from observing that a batch of cone programs can be represented as a single cone program in standard form as in Equation (7.3) with variables $x = [x_1^\top, \ldots, x_N^\top]^\top$ and data $A = \mathrm{diag}(A_1, \ldots, A_N)$, $b = [b_1^\top, \ldots, b_N^\top]^\top$, and $c = [c_1^\top, \ldots, c_N^\top]^\top$. This exploits the knowledge that all of the cone programs can be solved concurrently. The bottleneck of this mode is still typically in the linear system solve portion of SCS, which happens using sparse operations on the CPU or GPU.

SCS+PyTorch, batched. In this mode we represent the batch of cone programs as a single batched cone program use SCS will callbacks up into Python so that we can use PyTorch to efficiently solve the linear system. This allows us to keep the A data in PyTorch and potentially on the GPU without converting/transferring it and passing it into the SCS. Specifically we use dense operations and have implemented direct and indirect methods to solve Equation (7.11) in PyTorch and then pass the result back down into SCS for the rest of the operations. Our direct method uses PyTorch's batched LU factorizations and solves and our indirect method uses a batched conjugate gradient (CG) implementation. These custom linear system solvers are able to explicitly take advantage of the independence present in the linear systems that the sparse linear system solvers may not recognize automatically, and the dense solvers are also useful for dense cone programs, which come up in the context of differentiable optimization layers when large portions of the constraints are being learned.

7.4.2 Backward pass: Efficiently solving the linear system

When using cone programs as layers in end-to-end learning systems with some scalar-valued loss function ℓ, the full Jacobian $\mathsf{D}_\theta x^\star$ is expensive and unnecessary to form and requires solving $|\theta|$ linear systems. The Jacobian is only used when applying the chain rule to obtain $\mathsf{D}_\theta \ell = (\mathsf{D}_{x^\star}\ell)\mathsf{D}_\theta x^\star$. We can directly compute $\mathsf{D}_\theta \ell$ without computing the intermediate Jacobian by solving a single linear system. Following the method of **??**, we set up the system

$$\mathsf{D}_z \mathcal{R}(z^\star) \begin{bmatrix} d_{z_1} \\ d_{z_2} \\ 0 \end{bmatrix} = - \begin{bmatrix} \nabla_{x^\star}\ell \\ 0 \\ 0 \end{bmatrix}. \tag{7.15}$$

Applying the chain rule to $u^\star = \Pi z^\star$ gives $d_x = d_{z_1}$ and $d_y = (\mathsf{D}_z \Pi z) d_{z_2}$. We then compute the relevant backpropagation derivatives as

$$\nabla_c \ell = d_x \qquad \nabla_A \ell = d_y \otimes x^\star + y^\star \otimes d_x \qquad \nabla_b \ell = -d_y \tag{7.16}$$

Solving Equation (7.15) is still challenging to implement in practice as $\mathsf{D}_z \mathcal{R}(z^\star)$ can be large and sparse and doesn't have obviously exploitable properties such as symmetry or anti-symmetry. In addition to directly solving this linear system, we also explore the use of LSQR [PS82] as an iterative indirect method of solving this system in Section 7.6.2. Our LSQR implementation uses the implementation from Ali, Wong, and Kolter [AWK17] to compute $\mathsf{D}_z \Pi(z)$ in the form of an abstract linear operator so the full matrix does not need to be explicitly formed.

7.5 Examples

This section provides example use cases of our `cvxpy` optimization layer. All of these use the preamble

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

7.5.1 The ReLU, sigmoid, and softmax

We will start with basic examples and revisit the optimization views of the ReLU, sigmoid, and softmax from **??**. These can be implemented with our `cvxpy` layer in a few lines of code.

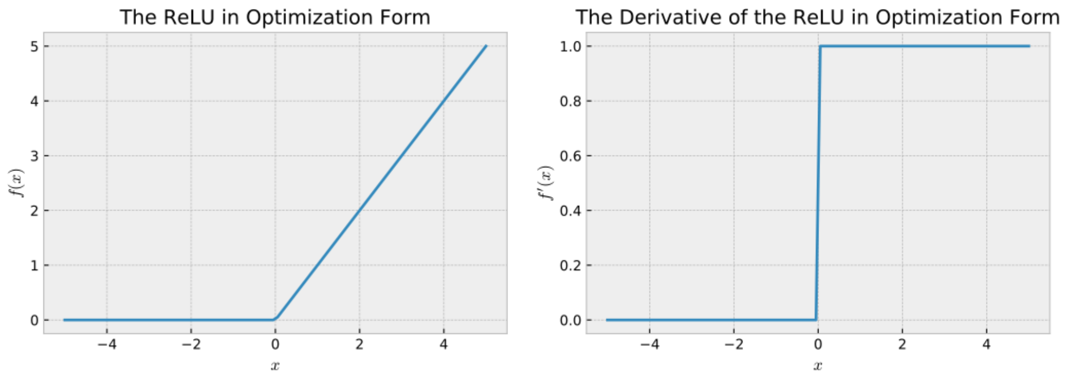
The ReLU. Recall from **??** that the optimization view is

$$f(x) = \operatorname*{argmin}_{y} \ \frac{1}{2}||x-y||_2^2 \ \text{ s. t. } \ y \geq 0.$$

We can implement this layer with:

```python
x = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.Minimize(cp.sum_squares(y-x))
cons = [y >= 0]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, params=[x], out_vars=[y])
```

This layer can be used and differentiated through just as any other PyTorch layer. Here is the output and derivative for a single dimension, illustrating that this is indeed performing the same operation as the ReLU.



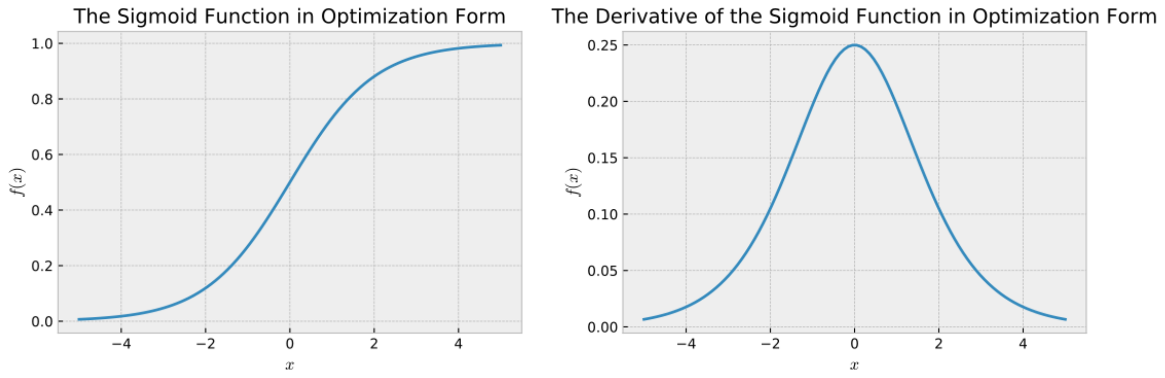
The sigmoid. Recall from **??** that the optimization view is

$$f(x) = \operatorname*{argmin}_{0<y<1} \quad -x^\top y - H_b(y).$$

We can implement this layer with:

```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])
```

We can also check that the output and derivative matches what we expect from the usual sigmoid function:



The softmax. Lastly recall from **??** that the optimization view is

$$f(x) = \operatorname*{argmin}_{0<y<1} \quad -x^\top y - H(y) \quad \text{s.t.} \quad 1^\top y = 1$$

We can implement this layer with:

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

7.5.2 The OptNet QP

We can re-implement the OptNet QP layer from **??** with our differentiable `cvxpy` layer in a few lines of code. The OptNet layer is represented as a convex quadratic program of the form

$$\begin{aligned} x^\star = \underset{x}{\text{argmin}} \quad & \frac{1}{2}x^\top Q x + p^\top x \\ \text{subject to} \quad & Ax = b \\ & Gx \le h \end{aligned} \tag{7.17}$$

where $x \in \mathbb{R}^n$ is our optimization variable $Q \in \mathbb{R}^{n\times n} \succeq 0$ (a positive semidefinite matrix), $p \in \mathbb{R}^n$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $G \in \mathbb{R}^{p\times n}$ and $h \in \mathbb{R}^p$ are problem data. We can implement this with:

```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])
```

This layer can then be used by passing in the relevant parameter values:

```python
Lval = torch.randn(nx, nx, requires_grad=True)
Qval = Lval.t().mm(Lval)
pval = torch.randn(nx, requires_grad=True)
Aval = torch.randn(ncon_eq, nx, requires_grad=True)
bval = torch.randn(ncon_eq, requires_grad=True)
Gval = torch.randn(ncon_ineq, nx, requires_grad=True)
hval = torch.randn(ncon_ineq, requires_grad=True)
y = layer(Qval, pval, Aval, bval, Gval, hval)
```

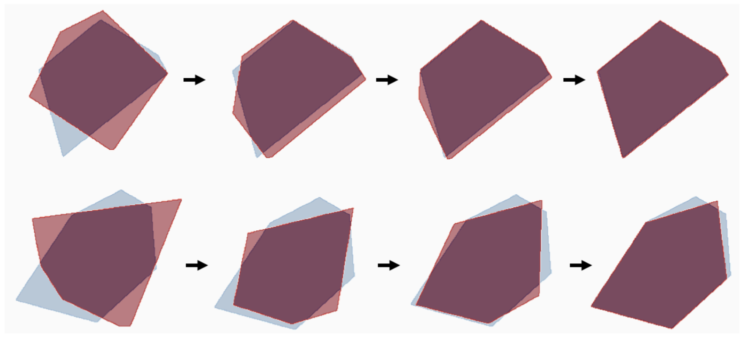


Figure 7.2: Learning polyhedrally constrained problems.

7.5.3 Learning Polyhedral Constraints

We demonstrate how gradient-based learning can be done with a `cvxpy` layer in this synthetic example. Consider the polyhedrally constrained projection problem

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

Suppose we don't know the polytope's parameters $\theta = \{G, h\}$ and want to learn them from data. Then using the MSE for ℓ, we can randomly initialize ellipsoids θ and learn them with gradient steps $\nabla_\theta \ell$. We note that this problem is meant for illustrative purposes and could be solved by taking the convex hull of the input data points. However our approach would still work if this was over a latent and unobserved part of the model, of if you want to take an approximate convex hull that limits the number of polytope edges.

We can implement this layer with the following code. Figure 7.2 shows the results of learning on two examples. Each problem has a true known polytope that we show in blue and the model's approximation is in red. Learning starts on the left with randomly initialized polytopes that are updated with gradient steps, which are shown in the images progressing to the right.

```python
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])
```

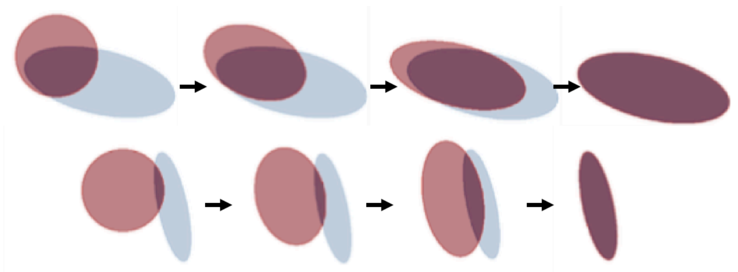


Figure 7.3: Learning ellipsoidally constrained problems.

7.5.4 Learning Ellipsoidal Constraints

In addition to learning polyhedral constraints, we can easily learn any parameterized convex constraint set. Suppose instead that we want to learn an ellipsoidal projection of the form

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

with ellipsoid parameters $\theta = \{A, z\}$ This is an interesting optimization problem to consider because it is an example of doing learning with a non-polytope cone program (a SOCP), which prior approaches such as OptNet could not easily handle.

We can implement this layer with the following code. Figure 7.3 visualizes the learning process on two examples. As before, each problem has a true known ellipsoid that we show in blue and the model's approximation is in red. Learning starts on the left with randomly initialized ellipsoids that are updated with gradient steps, which are shown in the images progressing to the right.

```
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])
```

7.6 Evaluation

In this section we analyze the runtime of our layer's forward and backward passes compared to hand-written implementations for commonly used optimization layers. We will focus on three tasks:

Task 1: Dense QP. We consider a QP layer of the form Equation (7.17) with a dense quadratic objective and dense inequality constraints. Our default experiment uses a QP with 100 latent variables, 100 inequality constraints, and a minibatch size of 128 examples. We chose this task to understand how the performance of our `cvxpy` layer compares to the `qpth` implementation from **??**, which we use as a comparison point. The problem size we consider here is comparable to the QP problem sizes considered in **??**. The backwards pass of `qpth` is optimized to use a single batched, pre-factorized linear system solve.

Task 2: Box QP. We consider a QP layer of the form Equation (7.17) with a dense quadratic objective constrained to the box $[-1, 1]^n$. Our default experiment uses a QP with 100 latent variables and a minibatch size of 128 examples. We chose this task to study the impacts of sparsity on the runtime. We again use `qpth` as the comparison point for these experiments.

Task 3: Linear Quadratic Regulator (LQR). We consider a continuous-state-action, discrete-time, finite-horizon LQR problem of the form

$$\tau^\star_{1:T} = \operatorname*{argmin}_{\tau_{1:T}} \sum_t \frac{1}{2}\tau_t^\top C_t \tau_t + c_t^\top \tau_t \quad \text{subject to} \quad x_1 = x_{\text{init}},\ x_{t+1} = F_t \tau_t + f_t. \tag{7.18}$$

where $\tau_{1:T} = \{x_t, u_t\}_{1:T}$ is the nominal trajectory, T is the horizon, x_t, u_t are the state and control at time t, $\{C_t, c_t\}$ parameterize a convex quadratic cost, and $\{F_t, f_t\}$ parameterize an affine system transition dynamics. We consider a control problem with 10 states, 2 actions, and a horizon of 5. We compare to the differentiable model predictive control (MPC) solver from [Amo+18], which uses batched PyTorch operations to solve a batch of LQR problems with the Riccati equations, and then implements the backward pass with another, simpler, LQR solve with the Riccati equations.

For each of these tasks we have measured the forward and backward pass execution times for our layer in comparison to the specialized solvers. We have run these experiments on an unloaded system with an NVIDIA GeForce GTX 1080 Ti GPU and a four-core 2.60GHz Intel Xeon E5-2623 CPUs hyper-threaded to eight cores. We set the number of OpenMP threads to 8 for our experiments. For numerical stability, we use 64-bits for all of our implementations and baselines. For `qpth` and our implementation, we use an iteration stopping condition of $\epsilon = 10^{-3}$.

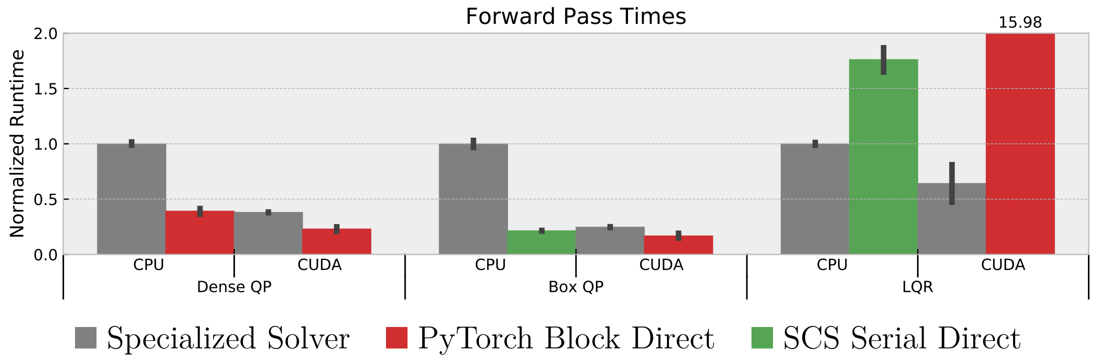


Figure 7.4: Forward pass execution times. For each task we run ten trials on an unloaded system and normalize the runtimes to the CPU execution time of the specialized solver. The bars show the 95% confidence interval. For our method, we show the best performing mode.

7.6.1 Forward pass profiling

Figure 7.4 summarizes our main forward pass execution times. Figure 7.6 shows the runtimes of all of the modes and batch sizes, and Figure 7.5 illustrates the speedup of our best mode compared to the specialized solvers. We have implemented and run every mode from Section 7.4.1 and our summary presents the best-performing mode, which in every case on the GPU is our block direct solver. On the CPU, serializing SCS calls is competitive for problems with more sparsity. For dense and sparse QPs on the CPU and GPU, our batched SCS+PyTorch direct cone solver is faster than the `qpth` solver, which likely comes from the acceleration, convergence, and normalization tricks in SCS that are not present in `qpth`. The LQR task presents a sparse problem that illustrates the challenges to using a general cone program formulation. Our specialized solver that solves the Riccati equations in batched form exploits the sparsity pattern of the problem that is extremely difficult for the general cone program formulation we consider here to take advantage of. If the correct mappings to the cone program exist, our layer could be modified to accept an optimized user-provided solver for the forward pass so that users can still take advantage of our backward pass implementation.

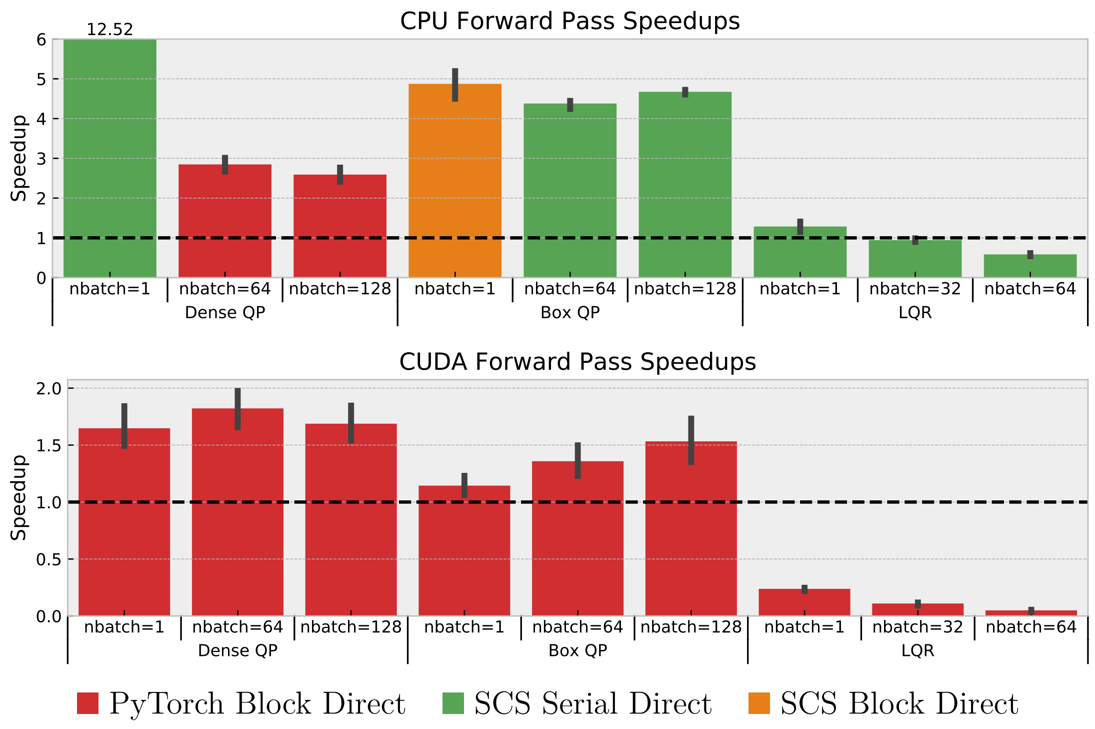


Figure 7.5: Forward pass execution time speedups of our best performing method in comparison to the specialized solver's execution time. For each task we run ten trials on an unloaded system. The bars show the 95% confidence interval.

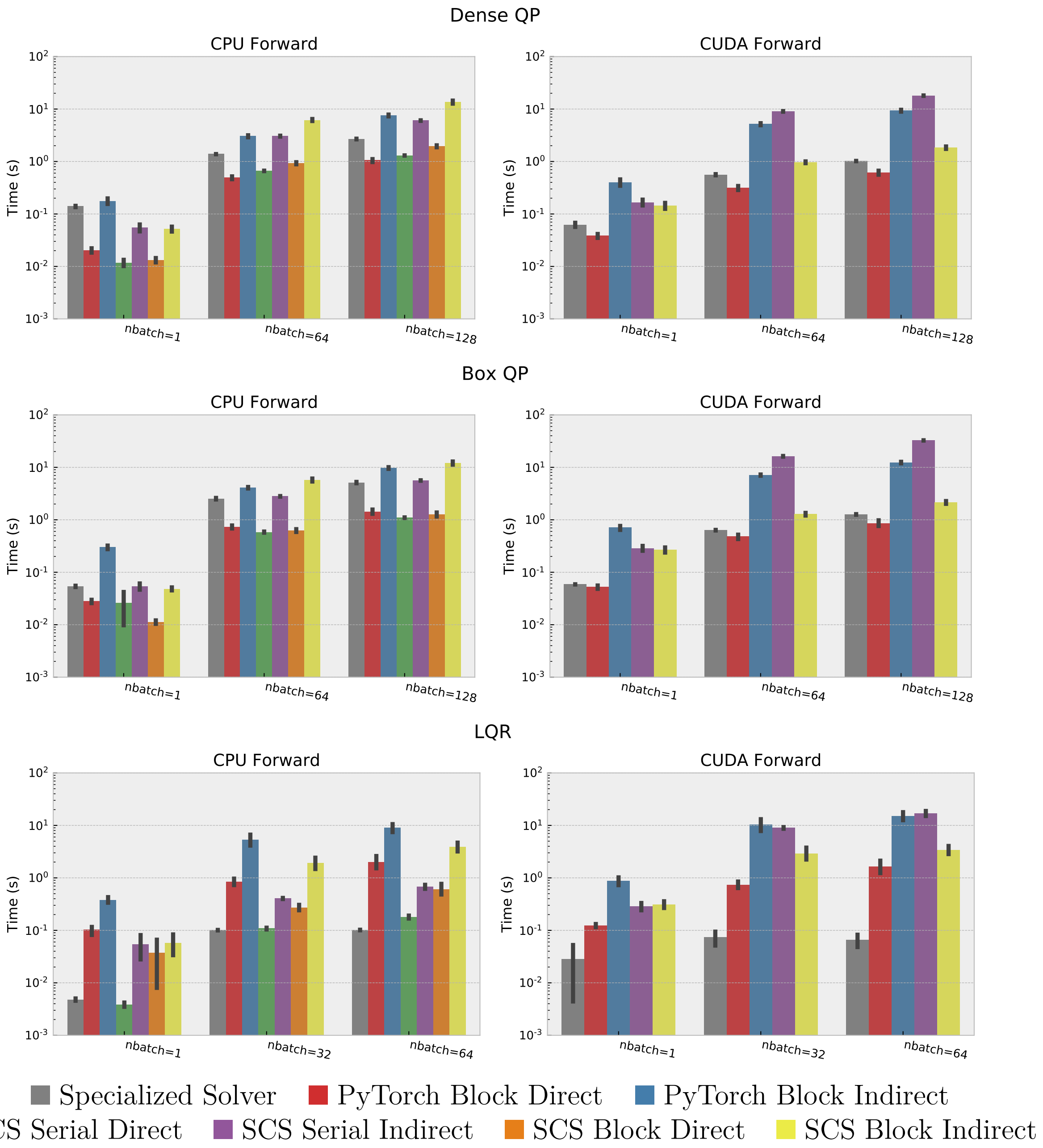


Figure 7.6: Full data for the forward pass execution times. For each task we run ten trials on an unloaded system. The bars show the 95% confidence interval.

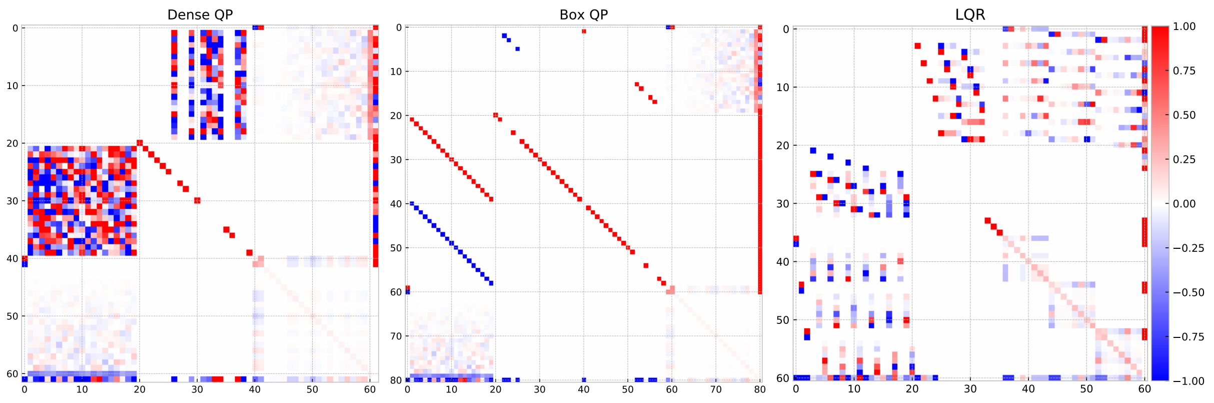


Figure 7.7: Sample linear system coefficients for the backward pass system in Equation (7.15) on smaller versions of the tasks we consider. The tasks we consider are approximately five times larger than these systems.

7.6.2 Backward pass profiling

In this section we compare the backward pass times of our layer in comparison to the specialized solvers on the same three tasks as before: the dense QP, the box QP, and LQR. We show that differentiating through our conic solver is competitive in comparison to the specialized solver. As a comparison point, the `qpth` solver exploits the property that the linear system for the backward pass is the same as the linear system in the forward pass and can therefore do the backward pass with a single pre-factorized solve. The LQR solver exploits the property that the backward pass for LQR can be interpreted as another LQR problem that can efficiently be solved with the Riccati recursion.

These comparisons are important because the linear system for differentiating cone programs in Equation (7.15) is a more general form and cannot leverage the same exploits as the specialized solvers. To get an intuition of what these linear systems look like on our tasks, we plot sample maps on smaller problems of the coefficient matrix in Figure 7.7. This illustrates the sparsity that is typically present in the linear system that needs to be solved, but also illustrates that beyond sparsity, there is no other common property that can be exploited between the tasks.

To understand how many LSQR iterations are necessary to solve our task, we compare the approximate derivatives computed by LSQR to the derivatives obtained by directly solving the linear system in Figure 7.8. This shows that typically 500-1000 LSQR iterations are necessary for the tasks that we consider. In some cases such as $\partial x^\star/\partial A$ for LQR, the approximate gradient computed by LSQR does never converges exactly to the true gradient.

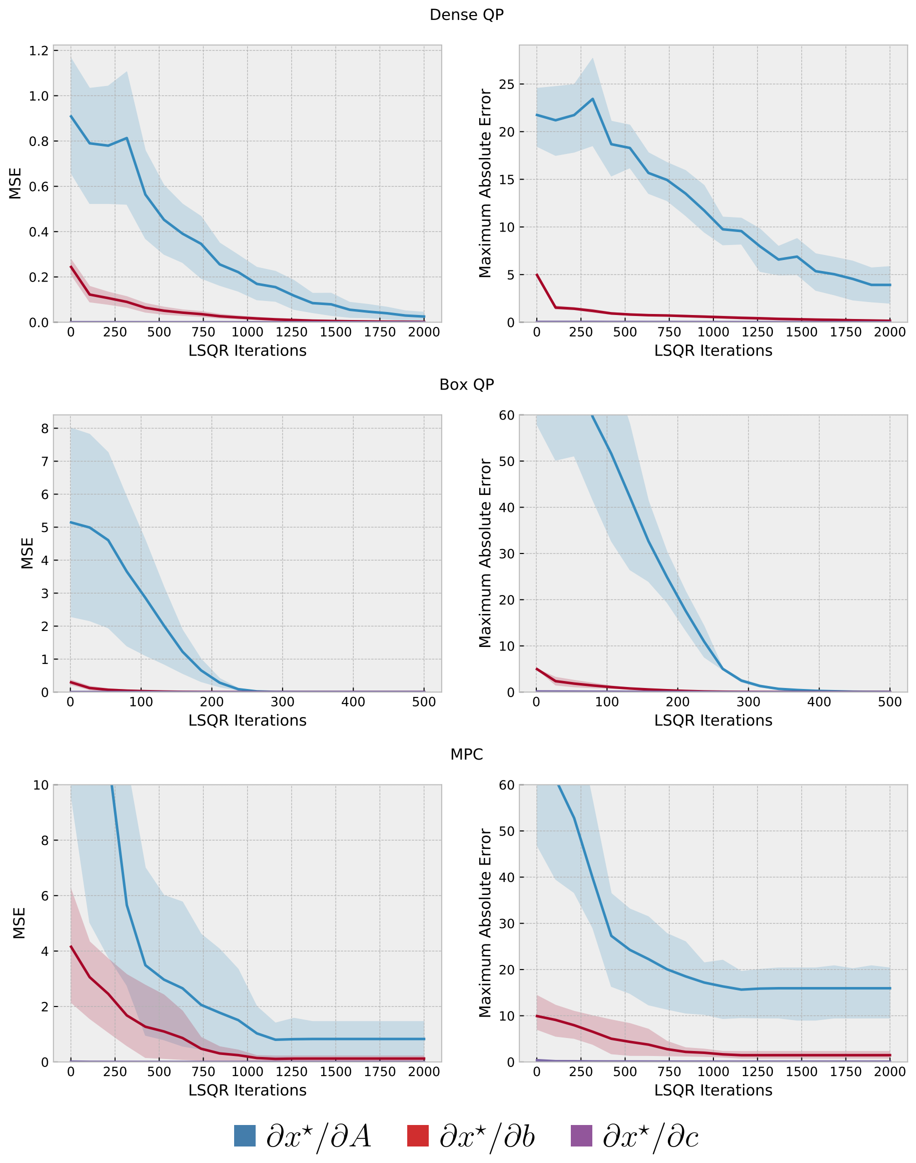


Figure 7.8: LSQR convergence for the backward pass systems. The shaded areas show the 95% confidence interval across ten problem instances.

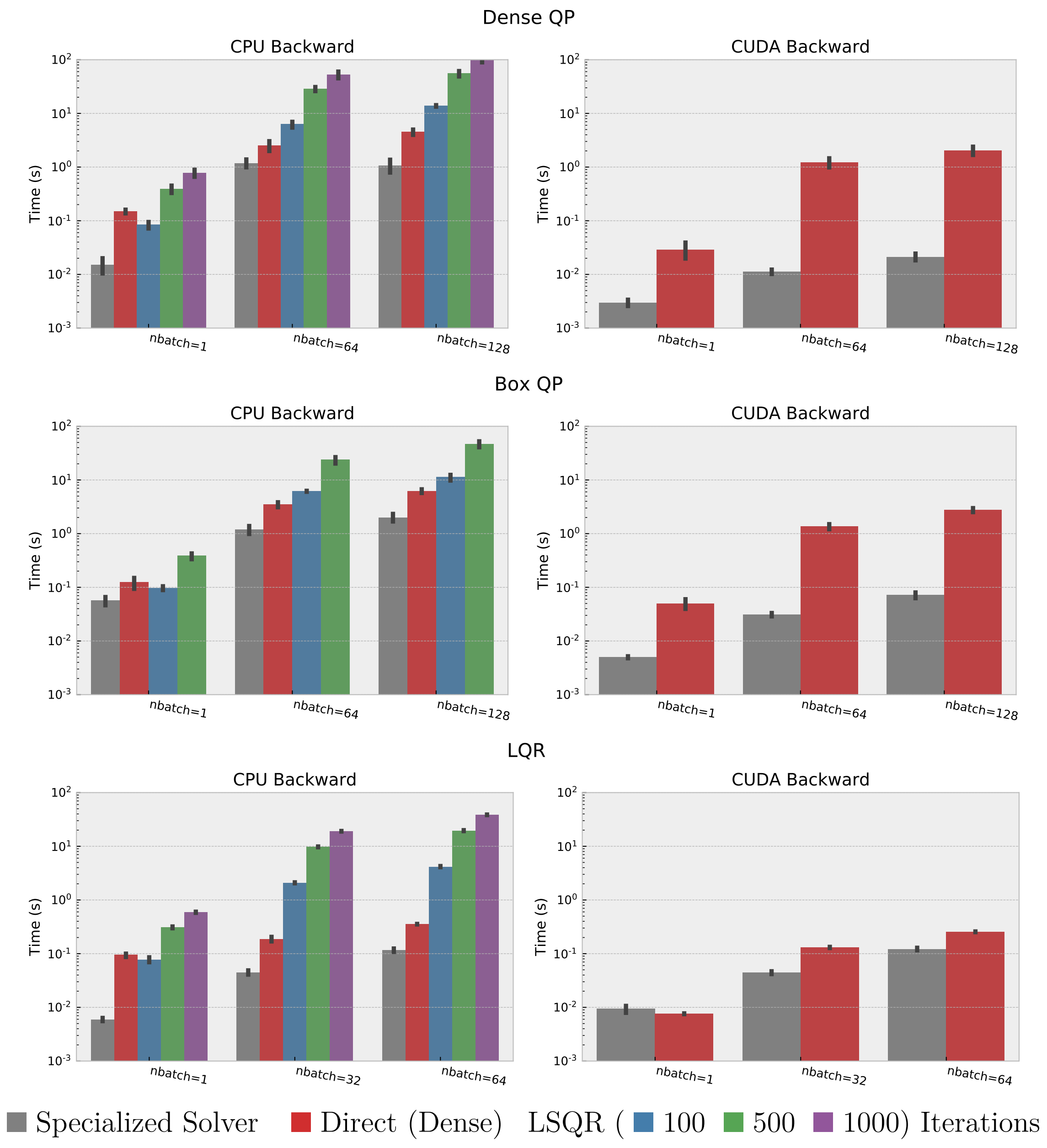


Figure 7.9: Backward pass execution times. For each task we run ten trials on an unloaded system. The bars show the 95% confidence interval.

Figure 7.9 compares our backward pass times to the specialized solvers for the QP and LQR tasks. This shows that there is a slight computational overhead in comparison to the specialized solvers, but that solving the linear system is still tractable for these tasks. The LSQR runtime is serialized across the batch, and is currently only implemented on the CPU. We emphasize that if the backward pass time becomes a bottleneck, the runtime can be further improved by further exploiting the sparsity by investigating other direct and indirect solvers for the systems, or by exploiting the property that we mentioned earlier in Section 7.3.1 that for simple cones like the free and non-negative cones, parts of the system become the same as parts of the KKT system.

7.7 Conclusion

This section has presented a way of differentiating through cone programs that enabled us to create a powerful prototyping tool for differentiable convex optimization layers. Practitioners can use this library in place of hand-implementing a solver and implicitly differentiating the KKT conditions. The speed of our tool is competitive with the speed of specialized solvers, even in the batched setting required for machine learning.

Bibliography

This bibliography contains 24 references.

[AG03] Farid Alizadeh and Donald Goldfarb. “Second-order cone programming”. In: *Mathematical programming* 95.1 (2003), pp. 3–51.

[Agr+19] Akshay Agrawal, Shane Barratt, Stephen Boyd, Enzo Busseti, and Walaa M. Moursi. “Differentiating Through a Conic Program”. In: *arXiv e-prints*, arXiv:1904.09043 (Apr. 2019), arXiv:1904.09043. arXiv: `1904.09043 [math.OC]`.

[Amo+18] Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J. Zico Kolter. “Differentiable MPC for End-to-end Planning and Control”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 8299–8310.

[AWK17] Alnur Ali, Eric Wong, and J. Zico Kolter. “A semismooth Newton method for fast, generic convex programming”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 70–79.

[BC17] Heinz H Bauschke and Patrick L Combettes. “Convex Analysis and Monotone Operator Theory in Hilbert Spaces, second edition”. In: (2017).

[Ber99] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

[BMB18] Enzo Busseti, W Moursi, and Stephen Boyd. “Solution Refinement at Regular Points of Conic Problems”. In: *arXiv preprint arXiv:1811.02157* (2018).

[BN01] Ahron Ben-Tal and Arkadi Nemirovski. *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. Vol. 2. Siam, 2001.

[Boy+11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. “Distributed optimization and statistical learning via the alternating direction method of multipliers”. In: *Foundations and Trends® in Machine Learning* 3.1 (2011), pp. 1–122.

[BS13] J Frédéric Bonnans and Alexander Shapiro. *Perturbation analysis of optimization problems*. Springer Science & Business Media, 2013.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[DB16] Steven Diamond and Stephen Boyd. “CVXPY: A Python-embedded modeling language for convex optimization”. In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 2909–2913.

[DR09] Asen L Dontchev and R Tyrrell Rockafellar. “Implicit functions and solution mappings”. In: *Springer Monogr. Math.* (2009).

[FI90] Anthony V Fiacco and Yo Ishizuka. "Sensitivity and stability analysis for nonlinear programming". In: *Annals of Operations Research* 27.1 (1990), pp. 215–235.

[GBY06] Michael Grant, Stephen Boyd, and Yinyu Ye. "Disciplined convex programming". In: *Global optimization.* Springer, 2006, pp. 155–210.

[Gou+16] Stephen Gould, Basura Fernando, Anoop Cherian, Peter Anderson, Rodrigo Santa Cruz, and Edison Guo. "On Differentiating Parameterized Argmin and Argmax Problems with Application to Bi-level Optimization". In: *arXiv preprint arXiv:1607.05447* (2016).

[KP13] Karl Kunisch and Thomas Pock. "A bilevel optimization approach for parameter learning in variational models". In: *SIAM Journal on Imaging Sciences* 6.2 (2013), pp. 938–983.

[Lob+98] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret. "Applications of second-order cone programming". In: *Linear algebra and its applications* 284.1-3 (1998), pp. 193–228.

[Mor61] Jean Jacques Moreau. "Décomposition orthogonale d'un espace hilbertien selon deux cônes mutuellement polaires". In: *Comptes rendus hebdomadaires des séances de l'Académie des sciences* 255 (1961), pp. 238–240.

[ODo+16] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. "Conic optimization via operator splitting and homogeneous self-dual embedding". In: *Journal of Optimization Theory and Applications* 169.3 (2016), pp. 1042–1068.

[Pas+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch". In: *NIPS Autodiff Workshop* (2017).

[PS82] Christopher C Paige and Michael A Saunders. "LSQR: An algorithm for sparse linear equations and sparse least squares". In: *ACM Transactions on Mathematical Software (TOMS)* 8.1 (1982), pp. 43–71.

[Roc70] R Tyrrell Rockafellar. "Convex Analysis Princeton University Press". In: *Princeton, NJ* (1970).

[YTM94] Yinyu Ye, Michael J Todd, and Shinji Mizuno. "An $O(\sqrt{n}L)$-iteration homogeneous and self-dual linear programming algorithm". In: *Mathematics of Operations Research* 19.1 (1994), pp. 53–67.