# A The canonicalization map

In this appendix, we provide a proof of Lemma 1. We compute $Q$ and $R$ via a reduction on the affine expression trees that represent the canonicalized problem. Let $f$ be the root node with arguments (descendants) $g_1, \dots, g_n$. Then we obtain tensors $T_1, \dots, T_n$ representing the (linear) action of $f$ on each argument. We recurse on each subtree $g_i$ and obtain tensors $S_1, \dots, S_n$. Due to the DPP rules, for $i = 1, \dots, n$, we either have $(T_i)_{j,k,\ell} = 0$ for $\ell \neq p+1$ or $(S_i)_{j,k,\ell} = 0$ for $\ell \neq p+1$. We define an operation $\psi(T_i, S_i)$ such that in the first case, $\psi(T_i, S_i) = \sum_{\ell=1}^{p+1} (T_i)_{[:,:,p+1]} (S_i)_{[:,:,\ell]}$, and in the second case $\psi(T_i, S_i) = \sum_{\ell=1}^{p+1} (T_i)_{[:,:,\ell]} (S_i)_{[:,:,p+1]}$. The tree rooted at $f$ then evaluates to $S_0 = \psi(T_1, S_1) + \cdots + \psi(T_n, S_n)$.

The base case of the recursion corresponds to the tensors produced when a variable, parameter, or constant node is evaluated. (These are the leaf nodes of an affine expression tree.)

- A variable leaf $x \in \mathbf{R}^d$ produces a tensor $T \in \mathbf{R}^{d \times n+1 \times 1}$, where $T_{i,j,1} = 1$ if $i$ maps to $j$ in the vector containing all variables, 0 otherwise.
- A parameter leaf $p \in \mathbf{R}^d$ produces a tensor $T \in \mathbf{R}^{d \times 1 \times p+1}$, where $T_{i,1,j} = 1$ if $i$ maps to $j$ in the vector containing all parameters, 0 otherwise.
- A constant leaf $c \in \mathbf{R}^d$ produces a tensor $T \in \mathbf{R}^{d \times 1 \times 1}$, where $T_{i,1,1} = c_i$ for $i = 1, \dots, d$.

# B Derivative of a cone program

In this appendix, we show how to differentiate through a cone program. We first present some preliminaries.

**Primal-dual form of a cone program.** A (convex) cone program is given by

$$\begin{array}{lll} \text{(P)} & \text{minimize} & c^T x \\ & \text{subject to} & Ax + s = b \\ & & s \in \mathcal{K}, \end{array} \qquad \begin{array}{lll} \text{(D)} & \text{minimize} & b^T y \\ & \text{subject to} & A^T y + c = 0 \\ & & y \in \mathcal{K}^*. \end{array} \tag{10}$$

Here $x \in \mathbf{R}^n$ is the *primal* variable, $y \in \mathbf{R}^m$ is the *dual* variable, and $s \in \mathbf{R}^m$ is the primal *slack* variable. The set $\mathcal{K} \subseteq \mathbf{R}^m$ is a nonempty, closed, convex cone with *dual cone* $\mathcal{K}^* \subseteq \mathbf{R}^m$. We call $(x, y, s)$ a solution of the primal-dual cone program (10) if it satisfies the KKT conditions:

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

Every convex optimization problem can be reformulated as a convex cone program.

**Homogeneous self-dual embedding.** The homogeneous self-dual embedding reduces the process of solving (10) to finding a zero of a certain residual map [76]. Letting $N = n+m+1$, the embedding uses the variable $z \in \mathbf{R}^N$, which we partition as $(u, v, w) \in \mathbf{R}^n \times \mathbf{R}^m \times \mathbf{R}$. The *normalized residual map* introduced in [24] is the function $\mathcal{N} : \mathbf{R}^N \times \mathbf{R}^{N \times N} \to \mathbf{R}^N$, defined by

$$\mathcal{N}(z, Q) = \big((Q - I)\Pi + I\big)(z/|w|),$$

where $\Pi$ denotes projection onto $\mathbf{R}^n \times \mathcal{K}^* \times \mathbf{R}_+$, and $Q$ is the skew-symmetric matrix

$$Q = \begin{bmatrix} 0 & A^T & c \\ -A & 0 & b \\ -c^T & -b^T & 0 \end{bmatrix}. \tag{11}$$

If $\mathcal{N}(z, Q) = 0$ and $w > 0$, we can use $z$ to construct a solution of the primal-dual pair (10) as

$$(x, y, s) = (u, \Pi_{\mathcal{K}^*}(v), \Pi_{\mathcal{K}^*}(v) - v)/w, \tag{12}$$

where $\Pi_{\mathcal{K}^*}(v)$ denotes the projection of $v$ onto $\mathcal{K}^*$. From here onward, we assume that $w = 1$. (If this is not the case, we can scale $z$ such that it is the case.)

**Differentiation.** A conic solver is a numerical algorithm for solving (10). We can view a conic solver as a function $\psi : \mathbf{R}^{m\times n} \times \mathbf{R}^m \times \mathbf{R}^n \to \mathbf{R}^{n+2m}$ mapping the problem data $(A, b, c)$ to a solution $(x, y, s)$. (We assume that the cone $\mathcal{K}$ is fixed.) In this section we derive expressions for the derivative of $\psi$, assuming that $\mathcal{S}$ is in fact differentiable. Interlaced with our derivations, we describe how to numerically evaluate the adjoint of the derivative map, which is necessary for backpropagation.

Following [1] and [4, Section 7], we can express $\psi$ as the composition $\phi \circ s \circ Q$, where

- $Q : \mathbf{R}^{m\times n} \times \mathbf{R}^m \times \mathbf{R}^n \to \mathbf{R}^{N\times N}$ maps the problem data to $Q$, given by (11),
- $s : \mathbf{R}^{N\times N} \to \mathbf{R}^N$ solves the homogeneous self-dual embedding, which we can implicitly differentiate, and
- $\phi : \mathbf{R}^N \to \mathbf{R}^n \times \mathbf{R}^m \times \mathbf{R}^m$ maps $z$ to the primal-dual pair, given by (12).

To backpropagate through $\psi$, we need to compute the adjoint of the derivative of $\psi$ at $(A, b, c)$ applied to the vector $(\mathsf{d}x, \mathsf{d}y, \mathsf{d}s)$, or

$$(\mathsf{d}A, \mathsf{d}b, \mathsf{d}c) = \mathsf{D}^T\psi(A, b, c)(\mathsf{d}x, \mathsf{d}y, \mathsf{d}s) = \mathsf{D}^T Q(A, b, c)\mathsf{D}^T s(Q)\mathsf{D}^T\phi(z)(\mathsf{d}x, \mathsf{d}y, \mathsf{d}s).$$

Since our layer only outputs the primal solution $x$, we can simplify the calculation by taking $\mathsf{d}y = \mathsf{d}s = 0$. By (12),

$$\mathsf{d}z = \mathsf{D}^T\phi(z)(\mathsf{d}x, 0, 0) = \begin{bmatrix} \mathsf{d}x \\ 0 \\ -x^T\mathsf{d}x \end{bmatrix}.$$

We can compute $\mathsf{D}s(Q)$ by implicitly differentiating the normalized residual map:

$$\mathsf{D}s(Q) = -(\mathsf{D}_z\mathcal{N}(s(Q), Q))^{-1}\mathsf{D}_Q\mathcal{N}(s(Q), Q). \tag{13}$$

This gives

$$\mathsf{d}Q = \mathsf{D}^T s(Q)\mathsf{d}z = -(M^{-T}\mathsf{d}z)\Pi(z)^T,$$

where $M = (Q - I)\mathsf{D}\Pi(z) + I$. Computing $g = M^{-T}\mathsf{d}z$ via a direct method (*i.e.*, materializing $M$, factorizing it, and back-solving) can be impractical when $M$ is large. Instead, one might use a Krylov method like LSQR [65] to solve

$$\underset{g}{\text{minimize}} \quad \|M^T g - \mathsf{d}z\|_2^2, \tag{14}$$

which only requires multiplication by $M$ and $M^T$. Instead of computing $\mathsf{d}Q$ as an outer product, we only obtain its nonzero entries. Finally, partitioning $\mathsf{d}Q$ as

$$\mathsf{d}Q = \begin{bmatrix} \mathsf{d}Q_{11} & \mathsf{d}Q_{12} & \mathsf{d}Q_{13} \\ \mathsf{d}Q_{21} & \mathsf{d}Q_{22} & \mathsf{d}Q_{23} \\ \mathsf{d}Q_{31} & \mathsf{d}Q_{32} & \mathsf{d}Q_{33} \end{bmatrix},$$

we obtain

$$\begin{aligned} \mathsf{d}A &= -\mathsf{d}Q_{12}^T + \mathsf{d}Q_{21} \\ \mathsf{d}b &= -\mathsf{d}Q_{23} + \mathsf{d}Q_{32}^T \\ \mathsf{d}c &= -\mathsf{d}Q_{13} + \mathsf{d}Q_{31}^T. \end{aligned}$$

**Non-differentiability.** To implicitly differentiate the solution map in (13), we assumed that the $M$ was invertible. When $M$ is not invertible, we approximate $\mathsf{d}Q$ as $-g^{\text{ls}}\Pi(z)^T$, where $g^{\text{ls}}$ is a least-squares solution to (14).

## C Examples

This appendix includes code for the examples presented in §6.

**Logistic regression.** The code for the logistic regression problem is below:

```python
import cvxpy as cp
from cvxpylayers.torch import CvxpyLayer

beta = cp.Variable((n, 1))
b = cp.Variable((1, 1))
X = cp.Parameter((N, n))

log_likelihood = (1. / N) * cp.sum(
    cp.multiply(Y, X @ beta + b) - cp.logistic(X @ beta + b)
)
regularization = -0.1 * cp.norm(beta, 1) -0.1 *
    cp.sum_squares(beta)

prob = cp.Problem(cp.Maximize(log_likelihood + regularization))
fit_logreg = CvxpyLayer(prob, parameters=[X], variables=[beta,
    b])
```

**Stochastic control.** The code for the stochastic control problem (7) is below:

```python
import cvxpy as cp
from cvxpylayers.torch import CvxpyLayer

x_cvxpy = cp.Parameter((n, 1))
P_sqrt_cvxpy = cp.Parameter((m, m))
P_21_cvxpy = cp.Parameter((n, m))
q_cvxpy = cp.Parameter((m, 1))

u_cvxpy = cp.Variable((m, 1))
y_cvxpy = cp.Variable((n, 1))

objective = .5 * cp.sum_squares(P_sqrt_cvxpy @ u_cvxpy) +
    x_cvxpy.T @ y_cvxpy + q_cvxpy.T @ u_cvxpy
prob = cp.Problem(cp.Minimize(objective),
  [cp.norm(u_cvxpy) <= .5, y_cvxpy == P_21_cvxpy @ u_cvxpy])

policy = CvxpyLayer(prob,
  parameters=[x_cvxpy, P_sqrt_cvxpy, P_21_cvxpy, q_cvxpy],
  variables=[u_cvxpy])
```

## D TensorFlow layer

In §5, we showed how to implement the problem (4) using our PyTorch layer. The below code shows how to implement the same problem using our TensorFlow 2.0 layer.

```python
import tensorflow as tf
from cvxpylayers.tensorflow import CvxpyLayer

F_t = tf.Variable(tf.random.normal(F.shape))
g_t = tf.Variable(tf.random.normal(g.shape))
lambd_t = tf.Variable(tf.random.normal(lambd.shape))
layer = CvxpyLayer(problem, parameters=[F, g, lambd],
    variables=[x])
with tf.GradientTape() as tape:
  x_star, = layer(F_t, g_t, lambd_t)
dF, dg, dlambd = tape.gradient(x_star, [F_t, g_t, lambd_t])
```

# E Additional examples

In this appendix we provide additional examples of constructing differentiable convex optimization layers using our implementation. We present the implementation of common neural networks layers, even though analytic solutions exist for some of these operations. These layers can be modified in simple ways such that they do *not* have analytical solutions. In the below problems, the optimization variable is $y$ (unless stated otherwise). We also show how prior work on differentiable convex optimization layers such as OptNet [6] is captured by our framework.

The **ReLU**, defined by $f(x) = \max\{0, x\}$, can be interpreted as projecting a point $x \in \mathbf{R}^n$ onto the non-negative orthant as

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}||x - y||_2^2 \\ \text{subject to} & y \geq 0. \end{array}$$

We can implement this layer with:

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

The **sigmoid** or **logistic** function, defined by $f(x) = (1 + e^{-x})^{-1}$, can be interpreted as projecting a point $x \in \mathbf{R}^n$ onto the interior of the unit hypercube as

$$\begin{array}{ll} \text{minimize} & -x^\top y - H_b(y) \\ \text{subject to} & 0 < y < 1, \end{array}$$

where $H_b(y) = -\left(\sum_i y_i \log y_i + (1 - y_i)\log(1 - y_i)\right)$ is the binary entropy function. This is proved, *e.g.*, in [4, Section 2.4]. 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, parameters=[x], variables=[y])
```

The **softmax**, defined by $f(x)_j = e^{x_j} / \sum_i e^{x_i}$, can be interpreted as projecting a point $x \in \mathbf{R}^n$ onto the interior of the $(n-1)$-simplex $\Delta_{n-1} = \{p \in \mathbf{R}^n \mid 1^\top p = 1 \;\text{ and }\; p \geq 0\}$ as

$$\begin{array}{ll} \text{minimize} & -x^\top y - H(y) \\ \text{subject to} & 0 < y < 1, \\ & 1^\top y = 1, \end{array}$$

where $H(y) = -\sum_i y_i \log y_i$ is the entropy function. This is proved, *e.g.*, in [4, Section 2.4]. 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, parameters=[x], variables=[y])
```

The **sparsemax** [58] does a Euclidean projection onto the simplex as

$$\begin{array}{ll} \text{minimize} & ||x-y||_2^2 \\ \text{subject to} & 1^\top y = 1, \\ & 0 \leq y \leq 1. \end{array}$$

We can implement this layer with:

```
x = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.sum_squares(x-y)
cons = [cp.sum(y) == 1, 0. <= y, y <= 1.]
prob = cp.Problem(cp.Minimize(obj), cons)
layer = CvxpyLayer(prob, [x], [y])
```

The **constrained softmax** [59] solves the optimization problem

$$\begin{array}{ll} \text{minimize} & -x^\top y - H(y) \\ \text{subject to} & 1^\top y = 1, \\ & y \leq u, \\ & 0 < y < 1. \end{array}$$

We can implement this layer with:

```
x = cp.Parameter(n)
y = cp.Variable(n)
obj = -x*y-cp.sum(cp.entr(y))
cons = [cp.sum(y) == 1., y <= u]
prob = cp.Problem(cp.Minimize(obj), cons)
layer = CvxpyLayer(prob, [x], [y])
```

The **constrained sparsemax** [55] solves the optimization problem

$$\begin{array}{ll} \text{minimize} & ||x-y||_2^2, \\ \text{subject to} & 1^\top y = 1, \\ & 0 \leq y \leq u. \end{array}$$

We can implement this layer with:

```
x = cp.Parameter(n)
y = cp.Variable(n)
obj = cp.sum_squares(x-y)
cons = [cp.sum(y) == 1., 0. <= y, y <= u]
prob = cp.Problem(cp.Minimize(obj), cons)
layer = CvxpyLayer(prob, [x], [y])
```

The **Limited Multi-Label (LML)** layer [7] solves the optimization problem

$$\begin{array}{ll} \text{minimize} & -x^\top y - H_b(y) \\ \text{subject to} & 1^\top y = k, \\ & 0 < y < 1. \end{array}$$

We can implement this layer with:

```
x = cp.Parameter(n)
y = cp.Variable(n)
obj = -x*y-cp.sum(cp.entr(y))-cp.sum(cp.entr(1.-y))
cons = [cp.sum(y) == k]
prob = cp.Problem(cp.Minimize(obj), cons)
layer = CvxpyLayer(prob, [x], [y])
```

**The OptNet QP.** We can re-implement the OptNet QP layer [6] in a few lines of code. The OptNet layer is a solution to a convex quadratic program of the form

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^\top Qx + q^\top x \\ \text{subject to} & Ax = b, \\ & Gx \leq h, \end{array}$$

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

```
Q_sqrt = cp.Parameter((n, n))
q = 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.sum_squares(Q_sqrt*x) + q.T @ x)
cons = [A @ x == b, G @ x <= h]
prob = cp.Problem(obj, cons)
layer = CvxpyLayer(prob, parameters=[Q_sqrt, q, A, b, G, h],
    variables=[x])
```

Note that we take the matrix square-root of $Q$ in PyTorch, outside the CVXPY layer, to get the derivative with respect to $Q$. DPP does not allow the quadratic form atom to be parametrized, as discussed in §4.1.