Deep Learning, Models & Optimization
Review of *OptNet: Differentiable Optimization as a Layer in Neural Networks (Amos, Kolter)* and *Differentiating Through a Conic Program* (Agrawal, Barratt, Boyd, Busseti, Moursi)

Pierre Personnat*
ENSAE Paris
pierre.personnat@ensae.fr

Mathieu Garrouty*
ENSAE Paris
mathieu.garrouty@ensae.fr

1 OptNet

Abstract

OptNet stages a new neural network implementation by incorporating differentiable optimization layers. These layers consist in quadratic programming (QP) solvers and enable end-to-end training of deep learning models with the added capability of learning optimization-based decision-making. The QP layer takes the parameters of a quadratic programming problem as inputs and outputs the optimal solution of the problem. In the article, this approach is demonstrated on various applications, including image denoising, model predictive control, and resource allocation, showcasing the advantages of integrating optimization techniques into deep learning architectures.

1.1 Introduction

Traditional deep learning models have focused on using layers such as convolutional, recurrent, and fully connected layers to learn complex functions and representations. However, many real-world problems involve decision-making processes that can be modeled using optimization techniques. OptNet presents a new method for integrating optimization techniques, specifically quadratic programming, into deep learning models. By incorporating QP solvers as layers in neural networks, OptNet allows the network to learn optimization-based decision-making for various applications, bridging the gap between optimization and deep learning. The output of the QP layer is the optimal solution of the quadratic programming problem, which can be used as input to following layers or as the final output for decision-making in various applications.

* stands for equal contribution

1.2 QP Problem Formulation

First of all, the authors present the quadratic programming problem and formulate it as a differentiable layer. They define the forward pass as solving the QP problem, and the backward pass involves computing the gradient of the layer's output with respect to its input. Consider a neural network with a QP layer, where the inputs to the QP layer are the parameters of a quadratic programming problem: the positive semidefinite matrix $P \in \mathbb{R}^{n \times n}$, the vector $q \in \mathbb{R}^n$, the matrices $G \in \mathbb{R}^{m \times n}$ and $A \in \mathbb{R}^{p \times n}$, and the vectors $h \in \mathbb{R}^m$ and $b \in \mathbb{R}^p$. These inputs can either be outputs from previous layers in the neural network or external inputs.

The QP layer aims to solve the following quadratic programming problem:

$$\begin{aligned} \min_{x} \quad & \frac{1}{2} x^T P x + q^T x \\ \text{s.t.} \quad & Gx \leq h \\ & Ax = b \end{aligned} \tag{1}$$

The output of the QP layer is the optimal solution $x^* \in \mathbb{R}^n$ of the QP problem, which results from minimizing the quadratic objective function subject to the linear inequality and equality constraints.

The optimal solution x^* serves as the output of the QP layer and can be used as input to subsequent layers in the neural network or as the final output for decision-making in various applications.

Backward pass

To compute the gradient of the QP layers during the backpropagation phase, the authors use the KKT (Karush-Kuhn-Tucker) conditions based on the Lagrangian of the QP problem. The KKT con-

ditions for the given QP problem are as follows:

$$\begin{aligned}
&\text{(i) Stationarity:} Px + q + G^T\lambda + A^T\nu = 0 \\
&\text{(ii) Primal feasibility:} Gx \leq h \quad \text{and } Ax = b \\
&\text{(iii) Dual feasibility:} \lambda \geq 0 \\
&\text{(iv) Complementary slackness:} \lambda_i(G_i x - h_i) = 0 \text{for all } i
\end{aligned} \tag{2}$$

where λ and ν are the dual variables associated with the inequality and equality constraints. To compute the gradients of the primal variable x with respect to the inputs (P, q, G, h, A, b), the authors differentiate the KKT conditions with respect to the inputs. Here follow the three derivative conditions from KKT:

$$\begin{aligned}
&dQz^* + Qdz + dq + dA^T\nu^* + A^T d\nu \\
&+ dG^T\lambda^* + G^T d\lambda = 0 \\
&dAz^* + Adz - db = 0 \\
&D(Gz^* - h)d\lambda + D(\lambda^*)(dGz^* + Gdz - dh) = 0
\end{aligned} \tag{3}$$

or equivalently written in matrix form:

$$\begin{bmatrix} Q & G^T & A^T \\ D(\lambda)G & D(Gz^* - h) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} dz \\ d\lambda \\ d\nu \end{bmatrix} = \tag{4}$$

$$\begin{bmatrix} -dQz^* - dq - dA^T\nu^* - dG^T\lambda^* \\ -dGz^* + dh + D(\lambda)d\lambda \\ -dAz^* + db \end{bmatrix} \tag{5}$$

From these total derivative conditions of the KKT system, the authors are aiming to compute the gradients of the output x^* with respect to the problem parameters (P, q, G, h, A, b).They mention that during the backpropagation algorithm, they do not want to explicitly form the actual Jacobian matrices. Instead, they aim to compute the product of the Jacobian matrix with the previous backward pass vector $\partial l \in \mathbb{R}^n$, i.e., $\frac{\partial l}{\partial z^*}$.

The reason behind this is that computing the full Jacobian matrix can be computationally expensive, especially for large-scale problems. Moreover, in practice, it is often unnecessary to compute the entire Jacobian matrix, as the ultimate goal is to compute the product of the Jacobian with the gradients from the previous layer during backpropagation.

Therefore, instead of explicitly forming the Jacobian, the authors focus on computing the matrix-vector product $\partial l, \partial z^*$ directly. This approach is more efficient computationally and is sufficient for updating the problem parameters during the end-to-end training of the neural network using gradient-based optimization algorithms.

After obtaining the derivative of the total loss of the network with respect to the different component of the QP problem, the article describes how these minima can be efficiently computed with an interior point method.

Forward pass

Indeed, one of the key challenge in solving QP problems in a neural network is to solve them simultaneously as the layers would process a batch of inputs at once. The authors are using a batched QP solver, specifically a "a GPU-based primal-dual inte- rior point method (PDIPM)". This batched approach is particularly beneficial in the context of GPU computation, as GPUs can parallelize the computation across multiple QP problems, leading to substantial performance improvements.

The Primal-Dual Interior Point Method (PDIPM) is an iterative algorithm for solving QP problems. At each iteration, the algorithm computes a search direction by solving a linear system derived from the KKT conditions. The search direction is used to update the primal and dual variables while staying close to the central path. In order to understand how this method works, let's break it down into its mains steps.

1. Form the KKT system for the QP problem:

$$\begin{bmatrix} Q & A^T & G^T \\ A & 0 & 0 \\ G & 0 & -D(\lambda) \end{bmatrix} \begin{bmatrix} z \\ \nu \\ \lambda \end{bmatrix} = \begin{bmatrix} q \\ b \\ h \end{bmatrix} \tag{6}$$

2. Compute the Newton step by differentiating the KKT system and solving the linear system:

$$\begin{bmatrix} Q & A^T & G^T \\ A & 0 & 0 \\ G & 0 & -D(\lambda) \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta\nu \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -r_z \\ -r_b \\ -r_c \end{bmatrix} \tag{7}$$

   where r_z, r_b, and r_c are the primal, dual, and complementary residuals, respectively.

3. Update the primal and dual variables with the computed search direction:

$$z^{k+1} = z^k + \alpha_p \Delta z \tag{8}$$
$$\nu^{k+1} = \nu^k + \alpha_d \Delta \nu \tag{9}$$
$$\lambda^{k+1} = \lambda^k + \alpha_d \Delta \lambda \tag{10}$$

where α_p and α_d are the primal and dual step lengths chosen to ensure that the iterates remain feasible and close to the central path.

The PDIPM algorithm iterates these steps until a suitable stopping criterion is met, such as a small duality gap or low residuals. This method stages the KKT matrix seen before and performs a factorization in step 2, which is a crucial intermediate result as it can be used to compute the gradients of the output z^* with respect to the problem parameters during the backward pass.
By reusing the KKT matrix factorization obtained during the forward pass (i.e., when solving the QP problems), the authors can efficiently compute the gradients required for the backward pass. This approach avoids redundant computations and further improves the overall efficiency of the OptNet layer.

1.3 Properties of OptNet

The article shows that OptNet layers can learn to approximate a wide range of nonlinear functions by learning the parameters of the QP problem. The authors also emphasize the benefits of incorporating constraints in the layers, which allow the model to learn complex relationships between the input and output while satisfying specific requirements.
More precisely, the authors present three theorems the representational power and properties of OptNet layers.
Theorem 1 establishes that the output of an OptNet layer, $z^*(\theta)$, is subdifferentiable everywhere, under certain conditions on the problem parameters. This property ensures that gradient-based optimization algorithms, such as stochastic gradient descent (SGD), can be applied effectively to train the neural network containing OptNet layers. The theorem also implies that the Jacobian, which is needed for the backward pass, exists and is unique for almost all points θ. This result is crucial for ensuring the stability and convergence of gradient-based optimization algorithms during training.
Theorem 2 establishes a connection between the OptNet layer and ReLU activation function. The authors show that under certain conditions, the solution of a QP problem with an OptNet layer can be expressed as a ReLU-activated linear function. This theorem demonstrates that OptNet layers can represent standard neural network layers, such as those with ReLU activation functions.
Theorem 3 discusses the relationship between OptNet layers and linear dynamical systems. The authors show that an OptNet layer can simulate the behavior of a linear time-invariant (LTI) system. This result further illustrates the representational power of OptNet layers and their ability to model complex relationships between input and output.

Concerning the limitations of this network architecture, one must insist on the computational complexity of solving QP problems. Although the batched PDIPM solver presented in the paper is efficient, solving QP problems can still be computationally expensive, especially for large-scale problems.

Another limitation is the choice of the QP problem's structure. The structure of the QP problem may not be suitable for all types of deep learning problems, and in some cases, the problem may need to be reformulated or approximated to fit within the OptNet framework. Furthermore, the OptNet approach requires tuning the QP problem's parameters, which can be challenging, especially in the presence of multiple layers and complex deep learning architectures.

2 Differentiating Through a Conic Program

Abstract

In this paper, authors presents a method for efficiently computing the derivative of the solution map of a convex cone program (when it exists and is unique). The authors achieve this by implicitly differentiating the residual map for its homogeneous self-dual embedding and solving the linear systems of equations using an iterative method. This enables the efficient computation of the derivative operator and its adjoint, which can be used for perturbation analysis and gradient computation. The method is scalable to large problems with millions of coefficients. They have also created an open-source Python library to implement their methods.

Some useful definitions

Primal Form

In convex optimization, we can define a conic program through its Primal form (P) :

$$\begin{aligned} \min_{x} \quad & c^\intercal\, x \\ \text{s.t.} \quad & Ax + s = b \\ & s \in \mathbf{K} \end{aligned} \tag{11}$$

where $\mathbf{K} \subset \mathbb{R}^m$ (convex cone), $A \in \mathbb{R}^{m*n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ are the *problem data*. x is the *primal variable*.

Dual Form

The problem can also be under its dual form (D) :

$$\begin{aligned} \min_{b} \quad & c^\intercal\, x \\ \text{s.t.} \quad & A^\intercal\, y + c = 0 \\ & y \in \mathbf{K}^* \end{aligned} \tag{12}$$

where $\mathbf{K}^* \subset \mathbb{R}^m$ is the dual cone of $\mathbf{K}$. These two systems are referred to as the **primal-dual conic program**.

Tools introduced

We define the set :

$$\mathcal{Q} = \left\{ Q = \begin{bmatrix} 0 & A^\intercal & c \\ -A & 0 & b \\ -c^\intercal & -b^\intercal & 0 \end{bmatrix}, Q \in \mathbb{R}^{N*N} \right\}$$

where $N = m + n + 1$. It will be used for the homogeneous self-embedding of our problem.

Solution of the problem

(x, y, z) is a solution to the **primal-dual conic program** if and only if :
$Ax + s = b$, $A^\intercal\, y + c = 0$, $s \in \mathbf{K}$ and $y \in \mathbf{K}^*$.

Solution mapping

In the rest of the paper, authors focus on the case where, for a given problem data, **the primal-dual conic programs has a single solution**. In most general cases, it could have 0, an unique or multiple solutions. Under this assumption, we can define a function

$$S : \mathbb{R}^{m*n} * \mathbb{R}^m * \mathbb{R}^n \longrightarrow \mathbb{R}^{m+2n}$$

taking the problem data as input and returning its only solution : $(x, y, z) = S(A, b, c)$. We construct this function by composition : $S = \phi \circ s \circ Q$

- $Q : (A, b, c) \mapsto \begin{bmatrix} 0 & A^\intercal & c \\ -A & 0 & b \\ -c^\intercal & -b^\intercal & 0 \end{bmatrix}$
- $S : \mathcal{Q} \longrightarrow \mathbb{R}^n$, giving a solution of the self-dual embedded problem.
- $\phi : \mathbb{R}^N \longrightarrow \mathbb{R}^{n+2m}$, transforming the self-dual solution into a primal-dual solution.

Methodology

Aim of the article

For a given Conic Program, under the assumption of existence and unicity of the solution, the authors study the consequence of variation (dA, db, dc) on the solution (x, y, z), and the reverse impact. To do so, they compute the derivative of S, and of its adjoint.
If S is differentiable, we have

$$DS(A, b, c) = D\phi(z)Ds(Q)DQ(A, b, c)$$

We have $DQ(A, b, c) = Q(A, b, c)$ as it is a linear function of its component. They then compute the two remaining terms.

Derivative of S respect to Q

Using work from [**?**], they introduce the *normalized residual map* function :

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

where $z \in \mathbb{R}^N$ is partitioned in $z = (u, v, w) \in \mathbb{R}^n * \mathbb{R}^m * \mathbb{R}$. z can be used to form a solution of the primal-dual problem if and only if $N(z, Q) = 0$

and $w > 0$. We can compute the derivative $D_z N(z, Q)$:

$$((Q-I)D\Pi(z)+I)/w - sign(w)((Q-I)\Pi+I)\frac{z}{w^2}e^\intercal$$

If z is a solution of our problem, second term vanishes, and we have (we can remove absolute value because $w > 0$ if z is a solution) :

$$D_z N(z, Q) = ((Q - I)D\Pi(z) + I)/w$$

As $\mathcal{N}$ is an affine function of Q, its derivative always exists and is :

$$D_Q \mathcal{N}(z, Q)^\intercal \ (y) = y(\Pi(z/|w|))^\intercal$$

We know also suppose that $D_z N(z, Q)$ is reversible at z (which is still supposed solution). By the **implicit function theorem**, we can find a neighborhood of $Q \in \mathcal{Q}$ in which $\mathcal{N}(s(Q), Q) = 0$, and where :

$$D_s(Q) = -(Dz\mathcal{N}(s(Q), Q))^{-1} D_Q \mathcal{N}(s(Q), Q)$$

Derivative of ϕ respect to z

The function ϕ construct a solution of the primal-dual pair from a solution $z = (u, v, w)$ of the homogeneous self-dual embedding. It is given by

$$\phi(z) = (u, \Pi_{\mathcal{K}^*}(v), \Pi_{\mathcal{K}^*}(v) - v)/w$$

It is differentiable if $\Pi_{\mathcal{K}^*}$ is, and we have :

$$D\phi(z) = \begin{bmatrix} I & 0 & -x \\ 0 & D\Pi_{\mathcal{K}^*}(v) & -y \\ 0 & D\Pi_{\mathcal{K}^*}(v) - I & -s \end{bmatrix}$$

We know are able to compute each terms of the derivative of S, and hence study the impact of perturbation on program data :

$$(dx, dy, ds) = D\phi(z)Ds(Q)DQ(A, b, c)(dA, db, dc)$$

Compute the derivative of the adjoint

Once we have computed these terms, we can easily study the impact of perturbation on the the solution of the problem, as :

$$(dA, db, dc) = DQ(A, b, c)^\intercal(Q)^\intercal \phi(z)^\intercal dx, dy, ds)$$

Implementation

Compute the derivative

The author choose to consider A as a sparse matrix. They basically follow the guideline announced above. They first compute dQ (linear in (A, b, c)). Then, they compute

$$g = D_Q \mathcal{N}(s(Q), dQ) = dQ\Pi(z/|w|)$$

$$dz = -M^{-1}g$$

where $M = ((Q - I)D\Pi(z) + I)/w$. To do so efficiently, they propose to use the LSQR algorithm (efficient when working with sparse matrix, and if Q is sparse, M is sparse as I and Π are also sparses). It gives us $dz = (du, dv, dw)$. Eventually, they compute

$$\begin{bmatrix} dx \\ dy \\ ds \end{bmatrix} = \begin{bmatrix} du - (dw)x \\ D\Pi_{\mathcal{K}^*}(v)dv - (dw)y \\ D\Pi_{\mathcal{K}^*}(v)dv - dv - (dw)s \end{bmatrix}$$

Compute the adjoint of the derivative

The method is quite similar. From

$$(dA, db, dc) = DQ(A, b, c)^\intercal \ Ds(Q)^\intercal \ D\phi(z)^\intercal \ (dx, dy, ds)$$

We first compute

$$dz = D\phi(z)^\intercal \ (dx, dy, ds) = \begin{bmatrix} dx \\ D\Pi^\intercal_{\mathcal{K}^*} \ (v)(dy + ds) - ds \\ -x^\intercal \ dx - y^\intercal \ dy - s^\intercal \ ds \end{bmatrix}$$

We then form (using LSQR again)

$$g = -M^{-\intercal} \ dz$$

then

$$dQ = g(\Pi(z/|w|))^\intercal$$

We only compute itz non-zero entries (to accelerate the computation). Eventually, we obtain :

$$dQ = \begin{bmatrix} dQ_{11} & dQ_{12} & dQ_{13} \\ dQ_{21} & dQ_{22} & dQ_{23} \\ dQ_{31} & dQ_{32} & dQ_{33} \end{bmatrix}$$

And the final expressions of our variations :

$$dA = -dQ^\intercal_{12} + dQ_{21}$$

$$db = -dQ_{23} + dQ^\intercal_{32}$$

$$dc = -dQ_{13} + dQ^\intercal_{31}$$

Experimental results

2.1 Simple OptNet implementation

In order to create a simple neural network with a differentiable optimization layer like in the article, we used the QP function defined in the authors' GitHub that implements the solver of such a problem for a specific layer and some other functions specific to the problem. As the code was written in 2012, it was not compatible anymore with PyTorch version; we had to adapt some parts of it. The network consists in a simple fully connected linear layer that maps the input to a new feature space and then only the QP layer that solves the QP problem using the outputs from the linear layer.
We simulated a simple dataset containing 10 numeric features and two numeric targets and below is the evolution of the loss for the training on this dataset.

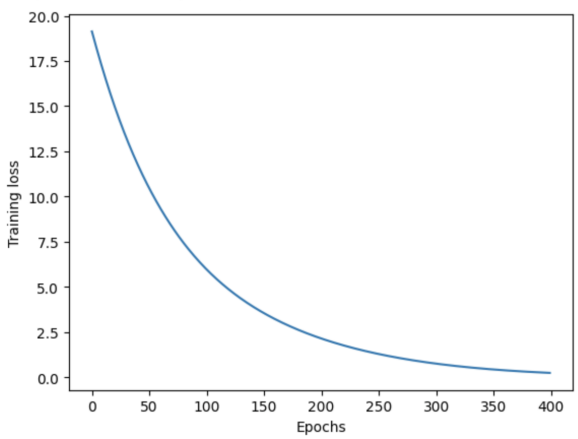

Figure 1: Evolution of the loss wrt to the epochs

2.2 Conic Program Implementation

We tried to use the same methods as above for computing the derivative of a conic program. We combined several python functions made by the authors to reproduce the method of the paper. We decided to see how computation time evolves respect to the size of the matrix A. To keep the results as clear as possible, we chose to work only with square matrix, in order to only have a two-dimensional graph. Here are our main results :

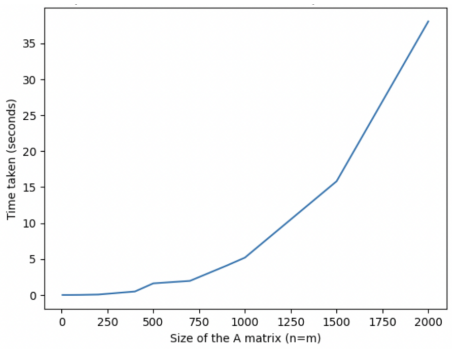

Figure 2: Computation time of the derivative respect to the shape of A.

We obtain a quadratic curves, as the number of computation increases with the non-zeros values of A, likely to increase in its number of cells.

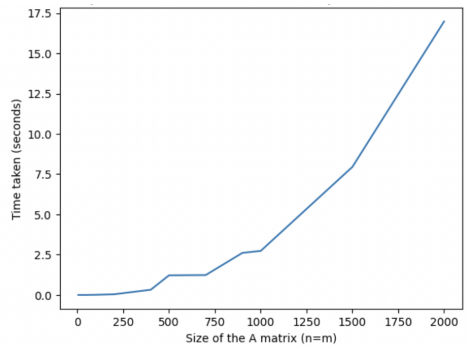

Figure 3: Computation time of the adjoint of the derivative respect to the shape of A

We can see here that the computation time of the adjoint of the derivative seems to be twice as small as the one of the derivative.