

# OptNet: Differentiable Optimization as a Layer in Neural Networks

Brandon Amos<sup>1</sup> J. Zico Kolter<sup>1</sup>

## Abstract

This paper presents OptNet, a network architecture that integrates optimization problems (here, specifically in the form of quadratic programs) as individual layers in larger end-to-end trainable deep networks. These layers allow complex dependencies between the hidden states to be captured that traditional convolutional and fully-connected layers are not able to capture. In this paper, we develop the foundations for such an architecture: we derive the equations to perform exact differentiation through these layers and with respect to layer parameters; we develop a highly efficient solver for these layers that exploits fast GPU-based batch solves within a primal-dual interior point method, and which provides backpropagation gradients with virtually no additional cost on top of the solve; and we highlight the application of these approaches in several problems. In one particularly standout example, we show that the method is capable of learning to play Sudoku given just input and output games, with no a priori information about the rules of the game; this task is virtually impossible for other neural network architectures that we have experimented with, and highlights the representation capabilities of our approach.

## 1. Introduction

In this paper, we lay the foundation and provide algorithms for treating constrained, exact optimization as itself a layer within a deep learning architecture. Unlike traditional feed-forward networks, where the output of each layer is a relatively simple (though non-linear) function of the previous layer, our optimization framework allows for individual layers to capture much richer behavior, expressing complex operations that in total can reduce the overall depth

of the network while preserving richness of representation. Specifically, we build a framework where the output of the  $i + 1$ th layer in a network is the *solution* to a constrained optimization problem based upon previous layers. This framework naturally encompasses a wide variety of inference problems expressed within a neural network, allowing for the potential of much richer end-to-end training for complex tasks that require such inference procedures.

Concretely, in this paper we specifically consider the task of solving small quadratic programs as individual layers. That is, we consider layers of the form

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

where  $z$  is the optimization variable,  $Q(z_i)$ ,  $p(z_i)$ ,  $A(z_i)$ ,  $b(z_i)$ ,  $G(z_i)$ , and  $h(z_i)$  are parameters of the optimization problem. As the notation suggests, these parameters can depend in any differentiable way on the previous layer  $z_i$ , and which can eventually be optimized just like any other weights in a neural network.

To implement layers of this form, we present two chief innovations. First, we show that it is possible backpropagate the relevant gradient information through the  $\operatorname{argmin}$  operator for a strictly convex quadratic program. By taking matrix differentials of the KKT conditions of the optimization problem at its solution, we derive backpropagation rules that allow us to compute all the relevant derivatives with respect to all the relevant parameters and through the layer.

Second, in order to make the approach actually practical for large networks, we develop a custom solver which can simultaneously solve multiple small QPs in batch form. We do so by developing a custom interior point primal dual method tailored specifically to dense batch operations on a GPU. Unlike virtually all existing related work that we are aware of, our architecture does *not* simply unroll an optimization procedure like gradient descent as a computation graph, but instead computes the gradients analytically at the solution to the optimization problem, which can then be computed much more efficiently and exactly. In total, the solver can solve batches of quadratic programs over 100 times faster than existing highly tuned quadratic pro-

<sup>1</sup>School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA. Correspondence to: Brandon Amos <bamos@cs.cmu.edu>, J. Zico Kolter <zkolter@cs.cmu.edu>.

gramming solvers such as Gurobi and CPLEX. One crucial algorithmic insight in the solver is that by using a specific factorization of the interior point primal-dual update, we can obtain a backward pass over the optimization layer virtually “for free” (i.e., requiring no additional factorization once the optimization problem itself has been solved). Together, these innovations make it practical to include parameterized optimization problems directly within the architecture of existing deep networks.

We begin by highlighting background and related work, and then present our optimization layer itself. Using matrix differentials we derive rules for computing all the necessary backpropagation updates. We then detail our specific solver for these quadratic programs, based upon a state-of-the-art primal-dual interior point architecture, and highlight the novel elements as they apply to our formulation, such as the aforementioned fact that we can compute backpropagation at very little additional cost. We then provide experimental results that demonstrate the capabilities of the architecture, highlighting potential tasks that these architectures can solve, and illustrating improvements upon existing approaches.

## 2. Background and related work

### General optimization and quadratic programming

Optimization plays a key role in modeling complex phenomena and providing concrete decision making processes in complex environments. For example, many control tasks are naturally represented as the solution to optimization problems such as those that occur in model prediction control frameworks (Morari & Lee, 1999); problems in rigid body dynamics like predicting contact forces are expressed and solved with quadratic programming (Löfstedt, 1984); and numerous statistical and mathematical formalisms are easily described via the solution to optimization problems (Boyd & Vandenberghe, 2004). Our work is a step towards learning optimization problems behind real-world processes from data that can be learned end-to-end rather than requiring human specification and intervention.

**Optimization within deep architectures** In addition to its general prominence, optimization is playing an increasingly important role in deep architectures. For instance, recent work on structured prediction (Belanger & McCallum, 2016; Amos et al., 2016) has used optimization within energy-based neural network models, and (Metz et al., 2016) used unrolled optimization within a network to stabilize the convergence of generative adversarial networks (Goodfellow et al., 2014).

These architectures typically introduce an optimization procedure such as gradient descent into the inference procedure. The optimization procedure is unrolled automati-

cally or manually (Domke, 2012) to obtain derivatives during training that incorporate the effects of these in-the-loop optimization procedures. However, unrolling the computation of a method like gradient descent typically requires a substantially larger network, and adds substantially to the computation complexity of the network. Crucially, in this paper, we do **not** unroll an optimization procedure but instead solve (constrained) optimization problems *exactly* (to numerical precision) using an interior point methods (Wright, 1997), and then analytically derive expressions for the gradients using the matrix differential of the KKT conditions.

**Structured prediction and MAP inference** Our work also draws some connection to MAP-inference-based learning and approximate inference. There are two broad classes of learning approaches in structured prediction: methods that use probabilistic inference techniques (typically exploiting the fact that the gradient of the log likelihood is given by the actual feature expectations minus their expectation under the learned model (Koller & Friedman, 2009, Ch 20)), and methods that rely solely upon MAP inference (such as max-margin structured prediction (Taskar et al., 2005; Tschantz et al., 2005)). MAP inference in particular also has close connections to optimization, as various convex relaxations of the general MAP inference problem often perform well in practice or in theory. The proposed methods can be viewed as an extreme case of this second class of algorithm, where inference is based *solely* upon a convex optimization problem that may not have any probabilistic semantics at all.

**Argmin differentiation** Most closely related to our own work, there have been several recent papers that propose some form of differentiation through argmin operators. In the case of (Gould et al., 2016), the authors describe general techniques for differentiation through optimization problems, but only describe the case of exact equality constraints rather than both equality and inequality constraints (in the case inequality constraints, they add these via a barrier function). Other work (Johnson et al., 2016; Amos et al., 2016) (the later with some inequality constraints) uses argmin differentiation within the context of a specific optimization problem, but doesn’t consider a particularly general setting; similarly, the older work of (Mairal et al., 2012) considered argmin differentiation for a LASSO problem, deriving specific rules for this case, and presenting an efficient algorithm based upon our ability to solve the LASSO problem efficiently. However, to the best of our knowledge none of these approaches consider how to handle and differentiate through general types of exact inequality and equality constraints, nor have they developed methods to make these approaches practical within the context of existing architectures.

### 3. OptNet: solving optimization within a neural network

In the most general form, an OptNet layer is an optimization problem of the form

$$\begin{aligned} z_{i+1} &= \underset{z}{\operatorname{argmin}} f_\theta(z, z_i) \\ \text{subject to } g_\phi(z, z_i) &\leq 0 \quad h_\varphi(z, z_i) = 0 \end{aligned} \quad (2)$$

where  $z_i$  and  $z_{i+1}$  are the previous and current layer,  $z$  is the optimization variable, and  $\theta$ ,  $\phi$ , and  $\varphi$  are parameters. Although most of the techniques we present here can easily be extended to the case of any convex optimization problem, the remainder of this paper will focus on the special case, mentioned in the introduction, where (2) is a convex quadratic program. More general convex optimization problems can be solved, for instance, using sequential quadratic programming that makes a relatively small number of calls to our QP optimizer within an inner loop over the larger optimization procedure.

#### 3.1. QP layers and backpropagation

To formally define our layer, we slightly simplify the previous notation, and consider the convex quadratic program

$$\begin{aligned} \underset{z}{\operatorname{minimize}} \quad & \frac{1}{2} z^T Q z + p^T z \\ \text{subject to} \quad & A z = b, \quad G z \leq h \end{aligned} \quad (3)$$

where  $z \in \mathbb{R}^n$  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. As is well-known, these problems can be solved in polynomial time using a variety of methods; if one desires exact (to numerical precision) solutions to these problems, then primal-dual interior point methods, as we will use in a later section, are the current state of the art in solution methods.

In the neural network setting, the *optimal solution* (or more generally, a *subset of the optimal solution*) of this optimization problems becomes the output of our layer, denoted  $z_{i+1}$ , and any of the problem data  $Q, p, A, b, G, h$  can depend on the value of the previous layer  $z_i$ . The forward pass in our OptNet architecture thus involves simply setting up and finding the solution to the above optimization problems.

Training deep architectures, however, requires that we not just have a forward pass in our network but also a backward pass. This requires that we compute the derivative of the solution to the QP with respect to its input parameters. Although the previous papers mentioned above have considered similar argmin differentiation techniques (Gould et al., 2016), to the best of our knowledge this is the first case of a general formulation for argmin differentiation in the

presence of exact equality and inequality constraints. To obtain these derivatives, we differentiate the KKT conditions (sufficient and necessary condition for optimality) of (3) at a solution to the problem, using techniques from matrix differential calculus (Magnus & Neudecker, 1988). Our analysis here can be extended to more general convex optimization problems.

The Lagrangian of (3) is given by

$$L(z, \nu, \lambda) = \frac{1}{2} z^T Q z + p^T z + \nu^T (A z - b) + \lambda^T (G z - h) \quad (4)$$

where  $\nu$  are the dual variables on the equality constraints and  $\lambda \geq 0$  are the dual variables on the inequality constraints. The KKT conditions for stationarity, primal feasibility, and complementary slackness are

$$\begin{aligned} Q z^* + p + A^T \nu^* + G^T \lambda^* &= 0 \\ A z^* - b &= 0 \\ D(\lambda^*)(G z^* - h) &= 0, \end{aligned} \quad (5)$$

where  $D(\cdot)$  creates a diagonal matrix from a vector. Taking the differentials of these conditions gives the equations

$$\begin{aligned} dQ z^* + Q dz + dp + dA^T \nu^* + \\ A^T d\nu + dG^T \lambda^* + G^T d\lambda &= 0 \\ dA z^* + A dz - db &= 0 \\ D(G z^* - h) d\lambda + D(\lambda^*)(dG z^* + G dz - dh) &= 0 \end{aligned} \quad (6)$$

or written more compactly in matrix form

$$\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^*)G & D(G z^* - h) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} dz \\ d\lambda \\ d\nu \end{bmatrix} = \begin{bmatrix} -dQ z^* - dp - dG^T \lambda^* - dA^T \nu^* \\ -D(\lambda^*)dG z^* + D(\lambda^*)dh \\ -dA z^* + db \end{bmatrix}. \quad (7)$$

Using these equations, we can form the Jacobians of  $z^*$  (or  $\lambda^*$  and  $\nu^*$ , though we don't consider this case here), with respect to any of the data parameters. For example, if we wished to compute the Jacobian  $\frac{\partial z^*}{\partial b} \in \mathbb{R}^{n \times m}$ , we would simply substitute  $db = I$  (and set all other differential terms in the right hand side to zero), solve the equation, and the resulting value of  $dz$  would be the desired Jacobian.

In the backpropagation algorithm, however, we never want to explicitly form the actual Jacobian matrices, but rather want to form the left matrix-vector product with some previous backward pass vector  $\frac{\partial \ell}{\partial z^*} \in \mathbb{R}^n$ , i.e.,  $\frac{\partial \ell}{\partial z^*} \frac{\partial z^*}{\partial b}$ . We can do this efficiently by noting the solution for the  $(dz, d\lambda, d\nu)$  involves multiplying the *inverse* of the left-hand-side matrix in (7) by some right hand side. Thus, if multiply the backward pass vector by the transpose of the

differential matrix

$$\begin{bmatrix} d_z \\ d_\lambda \\ d_\nu \end{bmatrix} = \begin{bmatrix} Q & G^T D(\lambda^*) & A^T \\ G & D(Gz^* - h) & 0 \\ A & 0 & 0 \end{bmatrix}^{-1} \begin{bmatrix} \left(\frac{\partial \ell}{\partial z^*}\right)^T \\ 0 \\ 0 \end{bmatrix} \quad (8)$$

then the relevant gradients with respect to all the QP parameters can be given by

$$\begin{aligned} \frac{\partial \ell}{\partial p} &= d_z & \frac{\partial \ell}{\partial b} &= -d_\nu \\ \frac{\partial \ell}{\partial p} &= d_z & \frac{\partial \ell}{\partial b} &= -d_\nu \\ \frac{\partial \ell}{\partial h} &= -D(\lambda^*)d_\lambda & \frac{\partial \ell}{\partial Q} &= \frac{1}{2}(d_x x^T + x d_x^T) \\ \frac{\partial \ell}{\partial A} &= d_\nu x^T + \nu d_x^T & \frac{\partial \ell}{\partial G} &= D(\lambda^*)(d_\lambda x^T + \lambda d_x^T) \end{aligned} \quad (9)$$

where as in standard backpropagation, all these terms are at most the size of the parameter matrices. As we will see in the next section, the solution to an interior point method in fact already provides a factorization we can use to compute these gradient efficiently.

### 3.2. An efficient batched QP solver

Deep networks are typically trained in mini-batches to take advantage of efficient data-parallel GPU operations. Without mini-batching on the GPU, many modern deep learning architectures become intractable for all practical purposes. However, today’s state-of-the-art QP solvers like Gurobi and CPLEX do not have the capability of solving multiple optimization problems on the GPU in parallel across the entire minibatch. This makes larger OptNet layers become quickly intractable compared to a fully-connected layer with the same number of parameters.

To overcome this performance bottleneck in our quadratic program layers, we have implemented a GPU-based primal-dual interior point method (PDIPM) based on (Mattingley & Boyd, 2012) that solves a batch of quadratic programs, and which provides the necessary gradients needed to train these in an end-to-end fashion. Our performance experiments in Section 4.1 shows that our solver is significantly faster than the standard non-batch solvers Gurobi and CPLEX.

Following the method of (Mattingley & Boyd, 2012), our solver introduces slack variables on the inequality constraints and iteratively minimizes the residuals from the KKT conditions over the primal and dual variables. Each iteration computes the affine scaling directions by solving

$$K \begin{bmatrix} \Delta z^{\text{aff}} \\ \Delta s^{\text{aff}} \\ \Delta \lambda^{\text{aff}} \\ \Delta \nu^{\text{aff}} \end{bmatrix} = \begin{bmatrix} -(A^T \nu + G^T \lambda + Qz + p) \\ -S\lambda \\ -(Gz + s - h) \\ -(Az - b) \end{bmatrix} \quad (10)$$

where

$$K = \begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & D(\lambda) & D(s) & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix},$$

then centering-plus-corrector directions by solving

$$K \begin{bmatrix} \Delta z^{\text{cc}} \\ \Delta s^{\text{cc}} \\ \Delta \lambda^{\text{cc}} \\ \Delta \nu^{\text{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma \mu 1 - D(\Delta s^{\text{aff}}) \Delta \lambda^{\text{aff}} \\ 0 \\ 0 \end{bmatrix}, \quad (11)$$

where  $\alpha$  is the largest step size that maintains dual feasibility and  $\sigma > 0$ . Each variable  $v$  is updated with  $\Delta v = \Delta v^{\text{aff}} + \Delta v^{\text{cc}}$  using an appropriate step size.

We solve these iterations for every example in our mini-batch by solving a symmetrized version of these linear systems with

$$K_{\text{sym}} = \begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & D(\lambda/s) & I & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix}, \quad (12)$$

where  $D(\lambda/s)$  is the only portion of  $K_{\text{sym}}$  that changes between iterations. We analytically decompose these systems into smaller symmetric systems and pre-factorize portions of them that don’t change (i.e. that don’t involve  $D(\lambda/s)$  between iterations.

We have implemented a batched version of this method with the PyTorch library<sup>1</sup> and have released it as an open source library at <https://github.com/locuslab/qpth>. It uses a custom CUBLAS extension that provide an interface to solve multiple matrix factorizations and solves in parallel, and which provides the necessary backpropagation gradients for their use in an end-to-end learning system.

#### 3.2.1. EFFICIENTLY COMPUTING GRADIENTS

A key point of the particular form of primal-dual interior point method that we employ is that it is possible to compute the backward pass gradients “for free” after solving the original QP, without an additional matrix factorization or solve. Specifically, at each iteration in the primal-dual interior point, we are computing an LU decomposition of the matrix  $K_{\text{sym}}$ .<sup>2</sup> This matrix is essentially a symmetrized

<sup>1</sup><https://pytorch.org>

<sup>2</sup>We actually perform an LU decomposition of a certain subset of the matrix formed by eliminating variables to create only a  $p \times p$  matrix (the number of inequality constraints) that needs to be factor during each iteration of the primal-dual algorithm, and one  $m \times m$  and one  $n \times n$  matrix once at the start of the primal-dual algorithm, though we omit the detail here. We also

version of the matrix needed for computing the backpropagated gradients, and we can similarly compute the  $d_{z,\lambda,\nu}$  terms by solving the linear system

$$K_{\text{sym}} \begin{bmatrix} d_z \\ d_s \\ \tilde{d}_\lambda \\ d_\nu \end{bmatrix} = \begin{bmatrix} \left(-\frac{\partial \ell}{\partial z_{i+1}}\right)^T \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad (13)$$

where  $\tilde{d}_\lambda = D(\lambda^*)d_\lambda$  for  $d_\lambda$  as defined in (8). Thus, all the backward pass gradients can be computed using the factored KKT matrix at the solution. Crucially, because the bottleneck of solving this linear system is computing the factorization of the KKT matrix (cubic time as opposed to the quadratic time for solving via backsubstitution once the factorization is computed), the additional time requirements for computing all the necessary gradients in the backward pass is virtually nonexistent compared with the time of computing the solution. To the best of our knowledge, this is the first time that this fact has been exploited in the context of learning end-to-end systems.

### 3.3. Limitation of the method

Although, as we will show shortly, the OptNet layer has several strong points, we also want to highlight the potential drawbacks of this approach. First, although, without an efficient batch solver, integrating an OptNet layer into existing deep learning architectures is potentially practical, we do note that solving optimization problems exactly as we do here has cubic complexity in the number of variables and/or constraints. This contrasts with the quadratic complexity of standard feedforward layers. This means that we are ultimately limited to settings where the number of hidden variables in an OptNet layer is not too large (less than 1000 dimensions seems to be the limits of what we currently find to be practical, and substantially less if one wants real-time results for an architecture).

Secondly, there are many improvements to the OptNet layers that are still possible. Our QP solver, for instance, uses fully dense matrix operations, which makes it solve very efficiently for GPU solutions, and which also makes sense for our general setting where the coefficients of the quadratic problem can be learned. However, for setting many real-world optimization problems (and hence for architectures that wish to more closely mimic some real-world optimization problem), there is often substantial structure (e.g., sparsity), in the data matrices that can be exploited for efficiency. There is of course no prohibition of incorporating sparse matrix methods into the fast custom solver, but

use an LU decomposition as this routine is provided in batch form by CUBLAS, but could potentially use a (faster) Cholesky factorization if and when the appropriate functionality is added to CUBLAS).

doing so would require substantial added complexity, especially regarding efforts like finding minimum fill orderings for different sparsity patterns of the KKT systems.

Lastly, we note that while the OptNet layers can be trained just as any neural network layer, since they are a new creation and since they have manifolds in the parameter space which have no effect on the resulting solution (e.g., scaling the rows of a constraint matrix and its right hand side does not change the optimization problem), there is admittedly more tuning required to get these to work. This situation is common when developing new neural network architectures, and our hope is that techniques for overcoming some of the challenges in learning these layers will be developed in future work.

## 4. Experimental results

In this section, we present several experimental results that highlight the capabilities of the OptNet layer. Specifically we look at 1) computational efficiency over existing solvers; 2) the ability to improve upon existing convex problems such as those used in signal denoising; 3) integrating the architecture into an generic deep learning architectures; and 3) performance of our approach on a problem that is very challenging for current approaches. performance of our approach can sometimes vastly improve upon existing deep architectures architectures. In particular, we want to emphasize the results of our system on learning the game of (4x4) Sudoku, a well-known logical puzzle; to the best of our knowledge, ours in the first type of end-to-end differentiable architecture that can learn problems such as this one based solely upon examples with no a priori knowledge of the rules of Sudoku. The code and data for our experiments are open sourced at <https://github.com/locuslab/optnet>.

### 4.1. Batch QP solver performance

Our first experiment illustrates why standard baseline QP solvers like CPLEX and Gurobi without batch support are too computationally expensive for OptNet layers to be tractable. We run our solver on an unloaded Titan X GPU and Gurobi on an unloaded quad-core Intel Core i7-5960X CPU @ 3.00GHz. We set up the same random QP of the form (1) across all three frameworks and vary the number



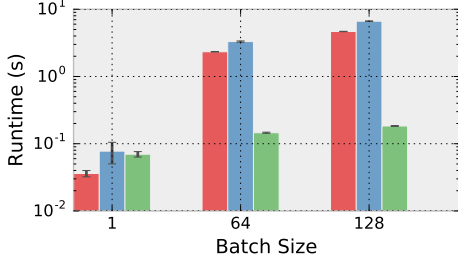


Figure 1. Performance of the Gurobi (red), qpth single (ours, blue), and qpth batched (ours, green) QP solvers.

of variable, constraints, and batch size.<sup>3</sup>

Figure 1 shows the means and standard deviations of running each trial 10 times, showing that our solver outperforms Gurobi, itself a highly tuned solver, in all batched instances. For the minibatch size of 128, we solve all problems in an average of 0.18 seconds, whereas Gurobi takes an average of 4.7 seconds. In the context of training a deep architecture this type of speed difference for a single minibatch can make the difference between a practical and a completely unusable solution.

#### 4.2. Total variation denoising

Our next experiment studies how we can use the OptNet architecture to *improve* upon signal processing techniques that currently use convex optimization as a basis. Specifically, our goal in this case is to denoise a noisy 1D signal given training data consistency of noisy and clean signals generated from the same distribution. Such problems are often addressed by convex optimization procedures, and (1D) total variation denoising is a particularly common and simple approach. Specifically, the total variation denoising approach attempts to smooth some noisy observed signal  $y$  by solving the optimization problem

$$\arg\min_z \frac{1}{2} \|y - z\|^2 + \lambda \|Dz\|_1 \quad (14)$$

where  $D$  is the first order differencing operation. Penalizing the  $\ell_1$  norm of the signal *difference* encourages this difference to be sparse, i.e., the number of changepoints of the signal is small, and we end up approximating  $y$  by a (roughly) piecewise constant function.

To test this approach and competing ones on a denoising

<sup>3</sup>Experimental details: we sample entries of a matrix  $U$  from a random uniform distribution and set  $Q = U^T U + 10^{-3} I$ , sample  $G$  with random normal entries, and set  $h$  by selecting generating some  $z_0$  random normal and  $s_0$  random uniform and setting  $h = Gz_0 + s_0$  (we didn't include equality constraints just for simplicity, and since the number of inequality constraints in the primary driver of complexity for the iterations in a primal-dual interior point method). The choice of  $h$  guarantees the problem is feasible.

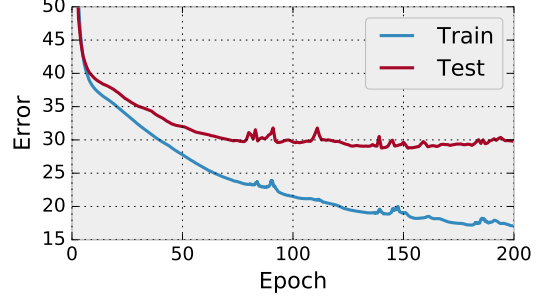


Figure 2. Performance of the fully connected network on the denoising task.

task, we generate piecewise constant signals (which are the desired outputs of the learning algorithm) and corrupt them with independent Gaussian noise (which form the inputs to the learning algorithm). The summary training and testing performance of the four approaches we describe are shown in Table 1.

##### 4.2.1. BASELINE: TOTAL VARIATION DENOISING

To establish a baseline for denoising performance with total variation, we run the above optimization problem varying values of  $\lambda$  between 0 and 100. The procedure performs best with a choice of  $\lambda \approx 13$ , and achieves a minimum test MSE on our task of about 16.5 (the units here are unimportant, the only relevant quantity is the relative performances of the different algorithms).

##### 4.2.2. BASELINE: LEARNING WITH A FULLY-CONNECTED NEURAL NETWORK

An alternative approach to denoising is by learning from data. A function  $f_\theta(x)$  parameterized by  $\theta$  can be used to predict the original signal. The optimal  $\theta$  can be learned by using the mean squared error between the true and predicted signals. Denoising is typically a difficult function to learn and Figure 2 shows that a fully connected neural network perform substantially worse on this denoising task than the convex optimization problem.

##### 4.2.3. LEARNING THE DIFFERENCING OPERATOR WITH OPTNET

Between the feedforward neural network approach and the convex total variation optimization, we could instead use a generic OptNet layers that effectively allowed us to solve (14) using *any* denoising matrix, which we randomly initialize. While the accuracy here is substantially lower than even the fully connected case, this is largely the result of learning an over-regularized solution to  $D$ . This is indeed a point that should be addressed in future work (we refer back to our comments in the previous section on the po-

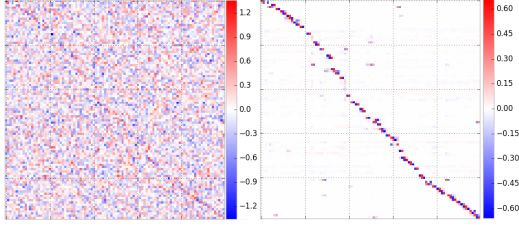


Figure 3. Initial and learned difference operators for denoising.

Method	Train MSE	Test MSE
FC Net	18.5	29.8
Pure OptNet	52.9	53.3
Total Variation	16.3	16.5
OptNet Tuned TV	13.8	<b>14.4</b>

Table 1. Performance of different methods on the denoising task.

tential challenges of training these layers), but the point we want to highlight here is that the OptNet layer seems to be learning something very interpretable and understandable. Specifically, Figure 3 shows the  $D$  matrix of our solution before and after learning (we permute the rows to make them ordered by the magnitude of where the large-absolute-value entries occurs). What is interesting in this picture is that the learned  $D$  matrix typically captures exactly the same intuition as the  $D$  matrix used by total variation denoising: a mainly sparse matrix with a few entries of alternating sign next to each other. This implies that for the data set we have, total variation denoising is indeed the “right” way to think about denoising the resulting signal, but if some other noise process were to generate the data, then we can learn that process instead. We can then attain lower actual error for the method (in this case similar though slightly higher than the TV solution), by fixing the learned sparsity of the  $D$  matrix and then fine tuning.

#### 4.2.4. FINE-TUNING AND IMPROVING THE TOTAL VARIATION SOLUTION

To finally highlight the ability of the OptNet methods to *improve* upon the results of a convex program, specifically tailoring to the data. Here, we use the same OptNet architecture as in the previous subsection, but initialize  $D$  to be the differencing matrix as in the total variation solution. As shown in Figure 4, the procedure is able to improve both the training and testing MSE over the TV solution, specifically improving upon test MSE by 12%.

### 4.3. MNIST

In this section we consider the integration of OptNet layers into a traditional fully connected network for the MNIST problem. The results here show only very marginal im-

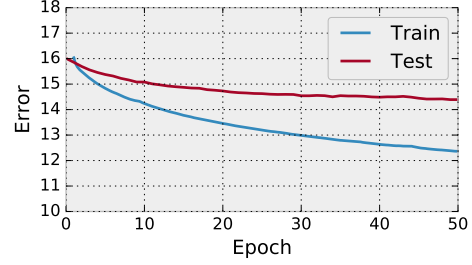


Figure 4. Error rate from fine-tuning the TV solution

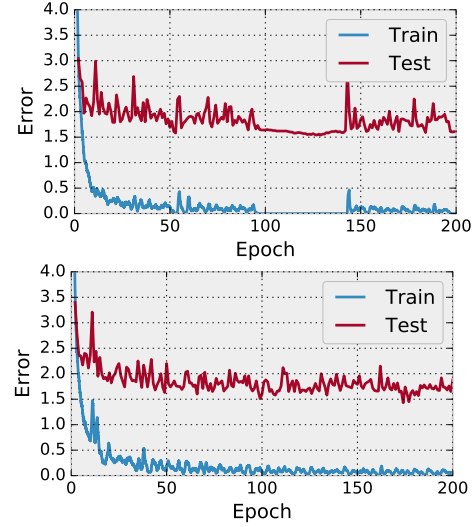


Figure 5. Training performance on MNIST; top: fully connected network; bottom: OptNet as final layer.)

provement if any over a fully connected layer (MNIST, after all, is very fairly well-solved by a fully connected network, let alone a convolution network). But our main point of this comparison is simply to illustrate that we can include these layers within existing network architectures and efficiently propagate the gradients through the layer. Specifically we use a FC600-FC10-FC10-SoftMax fully connected network and compare to a FC600-FC10-Optnet10-SoftMax network, where the numbers after each layer indicate the layer size, and the OptNet network in this case includes only inequality constraints. As shown in Figure 5, the results are similar for both networks with slightly lower error and less variance in the OptNet network.

### 4.4. Sudoku

Finally, we present the main illustrative example of the representational power of our approach, the task of learning the game of Sudoku. Sudoku is a popular logical puzzle, where a (typically 9x9) grid of points must be arranged given some initial point, so that each row, each column, and each 3x3 grid of points must contain one of each num-

			3
1			
		4	
4			1

2	4	1	3
1	3	2	4
3	1	4	2
4	2	3	1

Figure 6. Example 4x4 Sudoku puzzle, showing initial problem and solution.

ber 1 through 9. We consider the slightly simpler case of 4x4 Sudoku puzzles, with number in 1 through 4, as shown in Figure 4.3.

Sudoku is fundamentally a constraint satisfaction problem, and is trivial for computers to solve when told the rules of the game. However, if we do not know the rules of the game, but are only presented with examples of unsolved and the corresponding solved puzzle, this is a challenging task. We consider this to be an interesting benchmark task for algorithms that seek to capture complex strict relationships between all input and output variables. The input to the algorithm consists of a 4x4 grid (really a 4x4x4 tensor with a one-hot encoding for known entries and all zeros for unknown entries), and the desired output is a 4x4x4 tensor of the one-hot encoding of the solution.

This is a problem where traditional neural networks fail completely: as a baseline we implemented a multilayer feedforward network to attempt to solve Sudoku problems (specifically, we report results for a FC100-FC100-F100-FC100-Softmax network, though we tried other architectures as well), and found them completely unable to achieve an error rate lower than 99% on at test set of 1000 examples, where error here is interpreted as whether or not the puzzle is solved correctly if assign cell to whichever index is largest in the predicted encoding). This performance is shown in Figure 7, showing that the network *is* able to decrease the loss function (squared error) somewhat, but produces no boost in correctly solved puzzles.

We contrast this with the performance of the OptNet network. Here we learn a completely generic QP in so-called “standard form” with only positivity inequality constraints but an arbitrary constraint matrix  $Ax = b$ , a small  $Q = 0.1I$  to make sure the problem is strictly feasible, and with the linear term  $p$  simply being the input one-hot encoding of the Sudoku problem. We know that Sudoku *can* be approximated well with a linear program (indeed, integer programming is a typical solution method for such problems), but the model here is told nothing about the rules of Sudoku. Despite this, as shown in Figure 8 after just three epochs, the algorithm has effectively learned the game, and can get virtually zero test error with just minor noise in the learning process. This represents a substantial advance over the fully connected layers, and we believe highlights the ability of the OptNet layers to learn com-

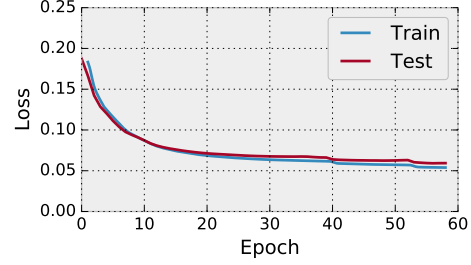


Figure 7. Training process of Sudoku with a fully connected network. Error is not show, but never goes below 0.99.

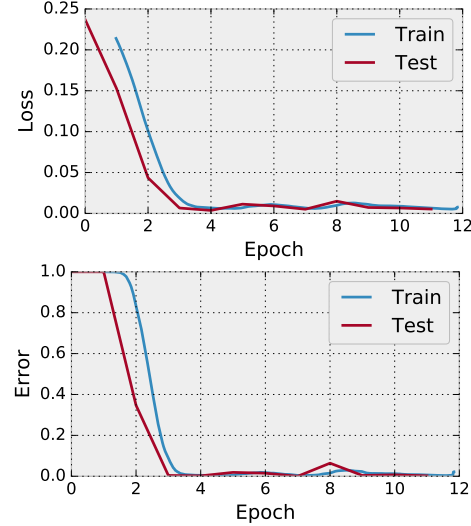


Figure 8. Training process of Sudoku with OptNet. After 3 epochs, the algorithm has effectively learned the game.

plex phenomena that currently elude neural networks. We know of no other machine learning algorithm that can learn a game like this solely from data.

## 5. Conclusion

We have presented OptNet, a neural network architecture where we use optimization problems as a single layer in the network. We have derived the algorithmic formulation for differentiating through these layers, allowing for backpropagating in end-to-end architectures. We have also developed an efficient batch solver for these optimizations based upon a primal-dual interior point method, and developed a method for attaining the necessary gradient information “for free” from this approach. Our experiments highlight the potential power of these networks, showing that they can solve problems where existing networks are very poorly suited, such as learning Sudoku problems purely from data. There are many future directions of research for these approaches, but we feel that they add another important primitive to the toolbox of neural network practitioners.



## References

- Amos, Brandon, Xu, Lei, and Kolter, J Zico. Input convex neural networks. *arXiv preprint arXiv:1609.07152*, 2016.
- Belanger, David and McCallum, Andrew. Structured prediction energy networks. In *Proceedings of the International Conference on Machine Learning*, 2016.
- Boyd, Stephen and Vandenberghe, Lieven. *Convex optimization*. Cambridge university press, 2004.
- Domke, Justin. Generic methods for optimization-based modeling. In *AISTATS*, volume 22, pp. 318–326, 2012.
- Goodfellow, Ian, Pouget-Abadie, Jean, Mirza, Mehdi, Xu, Bing, Warde-Farley, David, Ozair, Sherjil, Courville, Aaron, and Bengio, Yoshua. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.
- Gould, Stephen, Fernando, Basura, Cherian, Anoop, Anderson, Peter, Santa Cruz, Rodrigo, and Guo, Edison. On differentiating parameterized argmin and argmax problems with application to bi-level optimization. *arXiv preprint arXiv:1607.05447*, 2016.
- Johnson, Matthew, Duvenaud, David K, Wiltchko, Alex, Adams, Ryan P, and Datta, Sandeep R. Composing graphical models with neural networks for structured representations and fast inference. In *Advances in Neural Information Processing Systems*, pp. 2946–2954, 2016.
- Koller, Daphne and Friedman, Nir. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.
- Lötstedt, Per. Numerical simulation of time-dependent contact and friction problems in rigid body mechanics. *SIAM journal on scientific and statistical computing*, 5 (2):370–393, 1984.
- Magnus, X and Neudecker, Heinz. *Matrix differential calculus*. New York, 1988.
- Mairal, Julien, Bach, Francis, and Ponce, Jean. Task-driven dictionary learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):791–804, 2012.
- Mattingley, Jacob and Boyd, Stephen. Cvxgen: A code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012.
- Metz, Luke, Poole, Ben, Pfau, David, and Sohl-Dickstein, Jascha. Unrolled generative adversarial networks. *arXiv preprint arXiv:1611.02163*, 2016.
- Morari, Manfred and Lee, Jay H. Model predictive control: past, present and future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.
- Taskar, Ben, Chatalbashev, Vassil, Koller, Daphne, and Guestrin, Carlos. Learning structured prediction models: A large margin approach. In *Proceedings of the 22nd International Conference on Machine Learning*, pp. 896–903. ACM, 2005.
- Tsochantaridis, Ioannis, Joachims, Thorsten, Hofmann, Thomas, and Altun, Yasemin. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research*, 6:1453–1484, 2005.
- Wright, Stephen J. *Primal-dual interior-point methods*. Siam, 1997.