Deep Declarative Networks

Stephen Gould, *Member, IEEE*, Richard Hartley, *Fellow, IEEE*, and Dylan Campbell, *Member, IEEE*

Abstract—We explore a class of end-to-end learnable models wherein data processing nodes (or network layers) are defined in terms of desired behavior rather than an explicit forward function. Specifically, the forward function is implicitly defined as the solution to a mathematical optimization problem. Consistent with nomenclature in the programming languages community, we name these models *deep declarative networks*. Importantly, it can be shown that the class of deep declarative networks subsumes current deep learning models. Moreover, invoking the implicit function theorem, we show how gradients can be back-propagated through many declaratively defined data processing nodes thereby enabling end-to-end learning. We discuss how these declarative processing nodes can be implemented in the popular PyTorch deep learning software library allowing declarative and imperative nodes to co-exist within the same network. We also provide numerous insights and illustrative examples of declarative nodes and demonstrate their application for image and point cloud classification tasks.

Index Terms—Deep learning, implicit differentiation, declarative networks

1 Introduction

Modern deep learning models are composed of parametrized processing nodes (or layers) organized in a directed graph. There is an entire zoo of different model architectures categorized primarily by graph structure and mechanisms for parameter sharing [38]. In all cases the function for transforming data from the input to the output of a processing node is explicitly defined. End-to-end learning is then achieved by back-propagating an error signal along edges in the graph and adjusting parameters so as to minimize the error. Almost universally, the error signal is encoded as the gradient of some global objective (or regularized loss) function and stochastic gradient descent based methods are used to iteratively update parameters. Automatic differentiation (or hand-derived gradients) is used to compute the derivative of the output of a processing node with respect to its input, which is then combined with the chain rule of differentiation proceeding backwards through the graph. The error gradient can thus be calculated efficiently for all parameters of the model as the signal is passed backwards through each processing node.

In this paper we survey a new class of end-to-end learnable models, which we call *deep declarative networks* (DDNs), and which collects several developments from recent works on differentiable optimization [1], [2], [3], [4], [31] and related ideas [14], [51], [54] into a single framework. Instead of explicitly defining the forward processing function, nodes in a DDN are defined implicitly by specifying behavior. That is, the input–output relationship for a node is defined in terms of an objective and constraints in a mathematical optimization problem, the output being the solution of the problem conditioned on the input (and parameters). Importantly, as we will show, we can still perform back-propagation through a DDN by making use of implicit differentiation. Moreover, the gradient calculation does not require knowledge of the method used to solve the optimization problem, only the form of the objective and constraints, thereby allowing any state-of-the-art solver to be used during the forward pass.

DDNs subsume conventional deep learning models in that any explicitly defined forward processing function can also be defined as a node in a DDN. Furthermore, declaratively defined nodes and explicitly defined nodes can coexist within the same end-to-end learnable model. To make the distinction clear, when both types of nodes appear in the same model we refer to the former as *declarative nodes* and the latter as *imperative nodes*. To this end, we have developed a reference DDN implementation within PyTorch [44], a popular software library for deep learning, supporting both declarative and imperative nodes.

We present some theoretical results that show the conditions under which exact gradients can be computed and the form of such gradients. We also discuss scenarios in which the exact gradient cannot be computed (such as non-smooth objectives) but a descent direction can still be found, allowing stochastic optimization of model parameters to proceed. These ideas are explored through a series of illustrative examples and tested experimentally on the problems of image and point cloud classification using modified ResNet [33] and PointNet [46] architectures, respectively.

One advantage of the declarative view of deep neural networks is that it enables the use of classic constrained and unconstrained optimization algorithms as a modular component within a larger, end-to-end learnable network. This extends the concept of a neural network layer to include, for example, geometric model fitting, such as relative or absolute

- *Stephen Gould and Dylan Campbell are with the Australian Centre for Robotic Vision, Research School of Computer Science, Australian National University, Canberra, ACT 0200, Australia. E-mail: {stephen.gould, dylan.campbell}@anu.edu.au.*
- *Richard Hartley is with the Australian Centre for Robotic Vision, Research School of Engineering, Australian National University, Canberra, ACT 0200, Australia. E-mail: richard.hartley@anu.edu.au.*

Manuscript received 13 May 2020; revised 9 Feb. 2021; accepted 11 Feb. 2021. Date of publication 16 Feb. 2021; date of current version 1 July 2022.
(Corresponding author: Stephen Gould.)
Recommended for acceptance by V. Koltun.
Digital Object Identifier no. 10.1109/TPAMI.2021.3059462

pose solvers or bundle adjustment, model-predictive control algorithms, expectation-maximization, matching, optimal transport, and structured prediction solvers to name a few. Moreover, the change in perspective can help us envisage variations of standard neural network operations with more desirable properties, such as robust feature pooling instead of standard average pooling. There is also the potential to reduce opacity and redundancy in networks by incorporating local model fitting as a component within larger models. For example, we can directly use the (often nonlinear) underlying physical and mathematical models rather than having to re-learn these within the network. Importantly, this allows us to provide guarantees and enforce hard constraints on representations within a model (for example, that a rotation within a geometric model is valid or that a permutation matrix is normalized correctly). Furthermore, the approach is still applicable when no closed form solution exists, allowing sophisticated approaches with non-differentiable steps (such as RANSAC [29]) to be used internally. Global end-to-end learning of network parameters is still possible even in this case.

This new approach to developing deep learning components has attracted much interest over the past few years and several workshops and tutorials have been run in conjunction with major conferences to promote these ideas.[1] Being a new and active area of research, however, there are still challenges to address and we discuss some of these in this paper. Many preliminary and foundational ideas relating to declarative networks appear in previous works, which we summarize below in the context of a general coherent framework.

2 Background and Related Work

The ability to differentiate through declarative nodes relies on the implicit function theorem, which has a long history whose roots can be traced back to works by Descartes, Leibniz, Bernoulli and Euler [50]. It was Cauchy who first placed the theorem on rigorous mathematical grounds and Dini who first presented the theorem in its modern multi-variate form [24], [37]. Roughly speaking, the theorem states conditions under which the derivative of a variable y with respect to another variable x exists for implicitly defined functions, $f(x, y) = 0$, and provides a means for computing the derivative of y with respect to x when it does exist.

In the context of deep learning, it is the derivative of the output of a (declarative) node with respect to its input that facilitates end-to-end parameter learning by back-propagation [38], [47]. In this sense the learning problem is formulated as an optimization problem on a given error metric or regularized loss function. When declarative nodes appear in the network, computing the network output and hence the loss function itself requires solving an inner optimization problem. Formally, we can think of the learning problem as an upper optimization problem and the network output as being obtained from a lower optimization problem within a bi-level optimization framework [7], [53].

Bi-level optimization (and the need for implicit differentiation) has appeared in various settings in the machine learning literature, most notably for the problem of meta-learning. For example, Do *et al.* [22] consider the problem of determining regularization parameters for log-linear models formulated as a bi-level optimization problem. Domke [23] addresses the problem of learning parameters of continuous energy-based models wherein inference (equivalent to the forward pass in a neural network) requires finding the minimizer of a so-called energy function. The resulting learning problem is then necessarily bi-level. More recently, Johnson *et al.* [35] propose to combine conditional Markov random fields with deep learning and computing gradients of a mean field inference objective to facilitate learning. Last, Klatzer and Pock [36] propose a bi-level optimization framework for choosing hyper-parameter settings for support vector machines that avoids the need for cross-validation.

In computer vision and image processing, bi-level optimization has been used to formulate solutions to pixel-labeling problems. Samuel and Tappen [48] propose learning parameters of a continuous Markov random field with bi-level optimization and apply their technique to image denoising and in-painting. The approach is a special case of energy-based model learning [23]. Ochs *et al.* [42] extend bi-level optimization to handle non-smooth lower-level problems and apply their method to image segmentation tasks.

Recent works have started to consider the question of placing specific optimization problems within deep learning models [4], [14], [31], [51], [54]. These approaches can be thought of as establishing the groundwork for DDNs by developing specific declarative components. Gould *et al.* [31] summarize some general results for differentiating through unconstrained, linearly constrained and inequality constrained optimization problems. In the case of unconstrained and equality constrained problems the results are exact whereas for inequality constrained problems they make use of an interior-point approximation. We extend these results to the case of exact differentiation for problems with nonlinear equality and inequality constraints.

Amos and Kolter [4] also show how to differentiate through an optimization problem, for the specific case of quadratic programs (QPs). A full account of the work, including discussion of more general cone programs, appears in Amos [3]. Along the same lines Agrawal *et al.* [2] report results for efficiently differentiating through cone programs with millions of parameters. In both cases (quadratic programs and cone programs) the problems are convex and efficient algorithms exist for finding the minimum. Indeed differentiation can be automated for convex programs by extending disciplined convex programming frameworks [1]. In this paper, we make no restriction on convexity for declarative nodes but still assume that an algorithm exists for evaluating them in the forward pass.

In the case of linear programs (LPs) invoking the implicit function theorem to differentiate the solution, as can be done for other convex problems, breaks down. Here the derivative is not well defined and some recent works look at stochastic smoothing to get around this difficulty [8], [45].

Other works consider the problem of differentiating through discrete submodular problems [21], [51]. These problems have the nice property that minimizing a convex

[1] Such events include the CVPR 2020 Workshop on Deep Declarative Networks (http://cvpr2020.deepdeclarativenetworks.com/), the ECCV 2020 Tutorial on DDNs and Differentiable Optimization (http://eccv2020.deepdeclarativenetworks.com/), and the NeurIPS 2020 Tutorial on Deep Implicit Layers (https://implicit-layers-tutorial.org/).

relaxation still results in the optimal solution to the submodular minimization problem, which allows the derivative to be computed [21]. For submodular maximization there exist polynomial time algorithms to find approximate solutions. Smoothing of these solutions results in a differentiable model as demonstrated in Tschiatschek *et al.* [51]. A technique for evaluating the backward pass for models with general combinatorial optimization components with black box solvers is discussed in Vlastelica *et al.* [52].

Close in spirit to the idea of a deep declarative network is the recently proposed SATNet [54]. Here MAXSAT problems are approximated by solving a semi-definite program (SDP), which is differentiable. A fast solver allows the problem to be evaluated efficiently in the forward pass and thanks to implicit differentiation there is no need to explicitly unroll the optimization procedure, and hence store Jacobians, making it memory and time efficient in the backward pass. The method is applied to solving Sudoku problems presented as images, which requires that the learned network encode logical constraints.

Another interesting class of models that fit into the deep declarative networks framework are deep equilibrium models recently proposed by Bai *et al.* [6]. Here the model executes a sequence of fixed-point iterations $y^{(t)} = f(x, y^{(t-1)})$ until convergence in the forward pass. Bai *et al.* [6] show that rather than back-propagating through the unbounded sequence of fixed-point iterations, the derivative of the solution y with respect to input x can be computed directly via the implicit function theorem by observing that y satisfies implicit function $f(x, y) - y = 0$.

Chen *et al.* [14] show how to differentiate through ordinary differential equation (ODE) initial value problems at some time T using the adjoint sensitivity method, and view residual networks [33] as a discretization of such problems. This approach can be interpreted as solving a feasibility problem with an integral constraint function, and hence an exotic type of declarative node. The requisite gradients in the backward pass are computed elegantly by solving a second augmented ODE. While such problems can be included within our declarative framework, in this paper we focus on results for twice-differentiable objective and constraints functions that can be expressed in closed form.

There have also been several works that have proposed differentiable components based on optimization problems for addressing specific tasks. In the context of video classification, Fernando and Gould [26], [27] show how to differentiate through a rank-pooling operator [28] within a deep learning model, which involves solving a support vector regression problem. The approach was subsequently generalized to allow for a subspace representation of the video and learning on a manifold [15].

Santa Cruz *et al.* [49] propose a deep learning model for learning permutation matrices for visual attribute ranking and comparison. Two variants are proposed both relaxing the permutation matrix representation to a doubly-stochastic matrix representation. The first variant involves iteratively normalizing rows and columns to approximately project a positive matrix onto the set of doubly-stochastic matrices. The second variant involves formulating the projection as a quadratic problem and solving exactly.

Lee *et al.* [39] consider the problem of few-shot learning for visual recognition. They embed a differentiable QP [4] into a deep learning model that allows linear classifiers to be trained as a basis for generalizing to new visual categories. Promising results are achieved on standard benchmarks and they report low training overhead in that solving the QP takes about the same time as image feature extraction.

Deep learning models with differentiable optimization problems have also been considered in applications in geometric computer vision. For example, Chen *et al.* [13] develop a differentiable perspective-n-point (PnP) solver for estimating the pose of a camera within a 3D scene. In concurrent work, Campbell *et al.* [12] solve the blind PnP problem, wherein pixel to 3D point correspondences are not specified, using declarative nodes for both 2D-to-3D point matching and camera pose estimation.

In the context of planning and control, Amos *et al.* [5] propose a differentiable model predictive controller in a reinforcement learning setting. They are able to demonstrate superior performance on classic pendulum and cart-pole problems. De Avila Belbute-Peres *et al.* [17] show how to differentiate through physical models, specifically, the optimal solution of a linear complementarity problem, which allows a simulated physics environment to be placed within an end-to-end learnable system.

Last, imposing hard constraints on the output of a deep neural network using a Krylov subspace method was investigated in Marquez-Neila *et al.* [40]. The approach is applied to the task of human pose estimation and demonstrates the feasibility of training a very high-dimensional model that enforces hard constraints, albeit without improving over a softly constrained baseline.

3 Notation

The results herein require differentiating vector-valued functions with respect to vector arguments. To assist presentation we clarify our notation here. Consider the function $f(x, y, z)$ where both y and z are themselves functions of x. We have

$$\frac{\mathrm{d}}{\mathrm{d}x} f = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}x} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x} + \frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}x}, \tag{1}$$

by the chain rule of differentiation. For functions taking vector arguments, $f : \mathbb{R}^n \to \mathbb{R}$, we write the derivative vector as

$$\mathrm{D} f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right] \in \mathbb{R}^{1\times n}. \tag{2}$$

For vector-valued functions $f : \mathbb{R} \to \mathbb{R}^m$ we write

$$\mathrm{D} f = \left[\frac{\mathrm{d}f_1}{\mathrm{d}x}, \ldots, \frac{\mathrm{d}f_m}{\mathrm{d}x}\right]^{\mathsf{T}} \in \mathbb{R}^{m\times 1}. \tag{3}$$

More generally, we define the derivative $\mathrm{D} f$ of $f : \mathbb{R}^n \to \mathbb{R}^m$ as an $m \times n$ matrix with entries

$$\Big(\mathrm{D} f(x)\Big)_{ij} = \frac{\partial f_i}{\partial x_j}(x). \tag{4}$$

Then the chain rule for $h(x) = g(f(x))$ is simply

$$\mathrm{D}h(x) = \mathrm{D}g(f(x))\mathrm{D}f(x), \tag{5}$$

where the matrices automatically have the right dimensions and standard matrix-vector multiplication applies. When taking partial derivatives we use a subscript to denote the formal variable over which the derivative is computed (the remaining variables fixed), for example $\mathrm{D}_X f(x, y)$. For brevity we use the shorthand $\mathrm{D}^2_{XY} f$ to mean $\mathrm{D}_X(\mathrm{D}_Y f)^{\mathsf{T}}$.

When no subscript is given for multi-variate functions we take D to mean the total derivative with respect to the independent variables. So, with x as the independent variable, the vector version of Equation (1) becomes

$$\mathrm{D}f = \mathrm{D}_X f + \mathrm{D}_Y f\, \mathrm{D}y + \mathrm{D}_Z f\, \mathrm{D}z. \tag{6}$$

4 Deep Declarative Networks

We are concerned with data processing nodes in deep neural networks whose output $y \in \mathbb{R}^m$ is defined as the solution to a constrained optimization problem parametrized by the input $x \in \mathbb{R}^n$. In the most general setting we have

$$y \in \operatorname*{arg\,min}_{u \in C} f(x, u), \tag{7}$$

where $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is an arbitrary parametrized objective function and $C \subseteq \mathbb{R}^m$ is an arbitrary constraint set (that may also be parametrized by x). [2]

To distinguish such processing nodes from processing nodes (or layers) found in conventional deep learning models, which specify an explicit mapping from input to output, we refer to the former as declarative nodes and the latter as imperative nodes, which we formally define as follows.

Definition 4.1 (Imperative Node). *An* imperative *node is one in which the implementation of the forward processing function $\tilde{f}$ is explicitly defined. The output is then defined mathematically as*

$$y = \tilde{f}(x; \theta),$$

where θ are the parameters of the node (possibly empty).

Definition 4.2 (Declarative Node). *A* declarative *node is one in which the exact implementation of the forward processing function is not defined; rather the input–output relationship ($x \mapsto y$) is defined in terms of behavior specified as the solution to an optimization problem*

$$y \in \operatorname*{arg\,min}_{u \in C} f(x, u; \theta),$$

where f is the (parametrized) objective function, θ are the node parameters (possibly empty), and C is the set of feasible solutions, that is, constraints.

Fig. 1 is a schematic illustration of the different types of nodes. Since nodes form part of an end-to-end learnable model we are interested in propagating gradients backwards through them. In the sequel we make no distinction between the inputs and the parameters of a node as these are treated the same for the purpose of computing gradients.

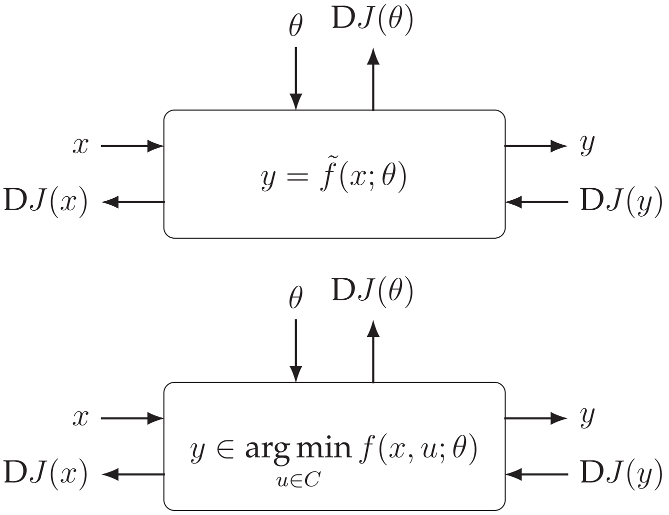


Fig. 1. Parametrized data processing nodes in an end-to-end learnable model with global objective function J. During the forward evaluation pass of an imperative node (top) the input x is transformed into output y based on some explicit parametrized function $\tilde{f}(\cdot; \theta)$. During the forward evaluation pass of a declarative node (bottom) the output y is computed as the minimizer of some parametrized objective function $f(x, \cdot; \theta)$. During the backward parameter update pass for either node type, the gradient of the global objective function with respect to the output $\mathrm{D}J(y)$ is propagated backwards via the chain rule to produce gradients with respect to the input $\mathrm{D}J(x)$ and parameters $\mathrm{D}J(\theta)$.

The definition of declarative nodes (Definition 4.2) is very general and encompasses many subproblems that we would like embedded within a network—robust fitting, projection onto a constraint set, matching, optimal transport, physical simulations, etc. Some declarative nodes may not be efficient to evaluate (that is, solve the optimization problem specifying its behavior), and in some cases the gradient may not exist (e.g., when the feasible set or output space is discrete). Nevertheless, under certain conditions on the objective and constraints, covering a wide range of problems or problem approximations (such as shown by Wang *et al.* [54] for MAXSAT and Berthet *et al.* [8] for LPs), the declarative node is differentiable and can be placed within an end-to-end learnable model as we discuss below.

4.1 Learning

We make no assumption about *how* the optimal solution y is obtained in a declarative node, only that there exists some algorithm for computing it. The consequence of this assumption is that, when performing back-propagation, we do not need to propagate gradients through a complicated algorithmic procedure from output to input. Rather we rely on implicit differentiation to directly compute the gradient of the node's output with respect to its input (or parameters). Within the context of end-to-end learning with respect to some global objective (i.e., loss function) this becomes a bi-level (multi-level) optimization problem [7], [18], which was first studied in the context of two-player leader–follower games by Stackelberg in the 1930s [53]. Formally, we can write

[2] In practical deep learning systems inputs and outputs are generally one or more multi-dimensional tensors. For clarity of exposition we consider inputs and outputs as single vectors without loss of generality.

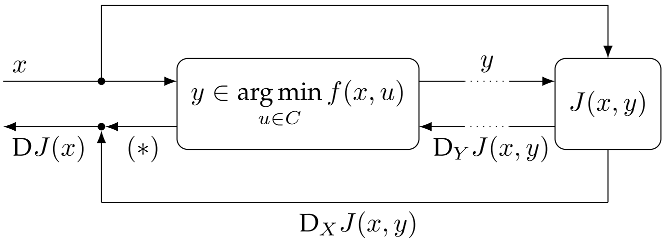


Fig. 2. Bi-level optimization problem showing back-propagation of gradients through a deep declarative node. The quantity $(*)$ is $\mathrm{D}_Y J(x,y)\, Dy(x)$ which when added to $\mathrm{D}_X J(x,y)$ gives $\mathrm{D}J(x,y(x))$. The bypass connections (topmost and bottommost paths) do not exist when the upper-level objective J only depends on x through y. Moreover, if f appears in J as the only term involving y then $\mathrm{D}_Y J(x,y)$ is zero and the backward edge $(*)$ is not required. That is, $\mathrm{D}J(x) = \mathrm{D}_X J(x,y)$.

$$\begin{array}{ll} \text{minimize} & J(x,y) \\ \text{subject to} & y \in \operatorname*{arg\,min}_{u \in C} f(x,u), \end{array} \tag{8}$$

where the minimization is over all tunable parameters in the network. Here $J(x,y)$ may depend on y through additional layers of processing. Note also that y is itself a function of x. As such minimizing the objective via gradient descent requires the following computation

$$\mathrm{D}J(x,y) = \mathrm{D}_X J(x,y) + \mathrm{D}_Y J(x,y)\mathrm{D}y(x), \tag{9}$$

The key challenge for declarative nodes then is the calculation of $\mathrm{D}y(x)$, which we will discuss in Section 4.3. A schematic illustration of a bi-level optimization problem and associated gradient calculations is shown in Fig. 2.

The higher-level objective J is often defined as the sum of various terms (including loss terms and regularization terms) and decomposes over individual examples from a training dataset. In such cases the gradient of J with respect to model parameters also decomposes as the sum of gradients for losses on each training example (and regularizers).

A simpler form of the gradient arises when the lower-level objective function f appears in the upper-level objective J as the only term involving y. Formally, if $J(x,y) = g(x, f(x,y))$ and the lower-level problem is unconstrained (that is, $C = \mathbb{R}^m$), then

$$\begin{aligned} \mathrm{D}J(x,y) &= \mathrm{D}_X g(x,f) + \mathrm{D}_F g(x,f)(\mathrm{D}f + \mathrm{D}_Y f\, \mathrm{D}y) \\ &= \mathrm{D}_X g(x,f) + \mathrm{D}_F g(x,f)\mathrm{D}f, \end{aligned} \tag{10}$$

since $\mathrm{D}_Y f(x,y) = 0$ by virtue of y being an unconstrained minimum of $f(x,\cdot)$. For example, if $J(x,y) = f(x,y)$, then $\mathrm{D}J(x,y) = \mathrm{D}_X f(x,y)$. This amounts to a variant of block coordinate descent on x and y, where y is optimized fully during each step, and in such cases we do not need to propagate gradients backwards through the declarative node. The remainder of this paper is focused on the more general case where y appears in other terms in the loss function (possibly through composition with other nodes) and hence calculation of $\mathrm{D}y$ is required.

4.2 Common Sub-Classes of Declarative Nodes

It will be useful to consider three very common cases of the general setting presented in Equation (7), in increasing levels of sophistication. The first is the unconstrained case,

$$y \in \operatorname*{arg\,min}_{u \in \mathbb{R}^m} f(x,u), \tag{11}$$

where the objective function f is minimized over the entire m-dimensional output space. The second is when the feasible set is defined by p possibly nonlinear equality constraints,

$$\begin{array}{lll} y \in & \arg\min_{u \in \mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h_i(x,u) = 0, \quad i = 1, \ldots, p. \end{array} \tag{12}$$

The third is when the feasible set is defined by p equality and q inequality constraints,

$$\begin{array}{lll} y \in & \arg\min_{u \in \mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h_i(x,u) = 0, \quad i = 1, \ldots, p \\ & & g_i(x,u) \leq 0, \quad i = 1, \ldots, q. \end{array} \tag{13}$$

Note that in all cases we can think of y as an implicit function of x, and will often write $y(x)$ to make the input–output relationship clear. We now show that, under mild conditions, the gradient of y with respect to x can be computed for these and other special cases without having to know anything about the algorithmic procedure used to solve for y in the first place.

In the sequel it will be useful to characterize solutions y using the concept of regularity due to Bertsekas [10, Section 3.3.1].

Definition 4.3 (Regular Point). *A feasible point u is said to be* regular *if the equality constraint gradients $\mathrm{D}_U h_i$ and the active inequality constraint gradients $\mathrm{D}_U g_i$ are linearly independent, or there are no equality constraints and the inequality constraints are all inactive at u.*[3]

4.3 Back-Propagation Through Declarative Nodes

Our results for computing gradients are based on implicit differentiation. We begin with the unconstrained case, which has appeared in different guises in several previous works [31]. The result is a straightforward application of Dini's implicit function theorem [24, p19] applied to the first-order optimality condition $\mathrm{D}_Y f(x,y) = 0$.

Proposition 4.4 (Unconstrained). *Consider a function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$. Let*

$$y(x) \in \operatorname*{arg\,min}_{u \in \mathbb{R}^m} f(x,u).$$

Assume $y(x)$ exists and that f is second-order differentiable in the neighborhood of the point $(x, y(x))$. Set $H = \mathrm{D}^2_{YY} f(x, y(x)) \in \mathbb{R}^{m \times m}$ and $B = \mathrm{D}^2_{XY} f(x, y(x)) \in \mathbb{R}^{m \times n}$. Then for H non-singular the derivative of y with respect to x is

$$\mathrm{D}y(x) = -H^{-1}B.$$

Proof. For any optimal y, the first-order optimality condition requires $\mathrm{D}_Y f(x,y) = 0_{1 \times m}$. The result then follows

[3] An inequality constraint g_i is active for a feasible point u if $g_i(x,u) = 0$ and inactive if $g_i(x,u) < 0$.

from the implicit function theorem: transposing and differentiating both sides with respect to x we have

$$\begin{aligned} 0_{m\times n} &= \mathrm{D}\left(\mathrm{D}_Y f(x,y)\right)^{\mathsf{T}} \\ &= \mathrm{D}^2_{XY} f(x,y) + \mathrm{D}^2_{YY} f(x,y)\mathrm{D}y(x), \end{aligned} \tag{14}$$

which can be rearranged to give

$$\mathrm{D}y(x) = -\left(\mathrm{D}^2_{YY} f(x,y)\right)^{-1}\mathrm{D}^2_{XY} f(x,y), \tag{15}$$

when $\mathrm{D}^2_{YY} f(x,y)$ is non-singular. □

In fact, the result above holds for any stationary point of $f(x,\cdot)$ not just minima. However, for declarative nodes it is the minima that are of interest. Our next result gives the gradient for equality constrained declarative nodes, making use of the fact that the optimal solution of a constrained optimization problem is a stationary point of the Lagrangian associated with the problem.

Proposition 4.5 (Equality Constrained). *Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^p$. Let*

$$\begin{array}{lll} y(x) \in & \arg\min_{u\in\mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h_i(x,u) = 0, \quad i = 1,\ldots,p. \end{array}$$

Assume that $y(x)$ exists, that f and $h = [h_1,\ldots,h_p]^{\mathsf{T}}$ are second-order differentiable in the neighborhood of $(x, y(x))$, and that $\mathrm{rank}(\mathrm{D}_Y h(x,y)) = p$. Then for H non-singular

$$\mathrm{D}y(x) = H^{-1}A^{\mathsf{T}}\left(AH^{-1}A^{\mathsf{T}}\right)^{-1}\left(AH^{-1}B - C\right) - H^{-1}B,$$

where

$$\begin{aligned} A &= \mathrm{D}_Y h(x,y) \in \mathbb{R}^{p\times m} \\ B &= \mathrm{D}^2_{XY} f(x,y) - \sum_{i=1}^{p} \lambda_i \mathrm{D}^2_{XY} h_i(x,y) \in \mathbb{R}^{m\times n} \\ C &= \mathrm{D}_X h(x,y) \in \mathbb{R}^{p\times n} \\ H &= \mathrm{D}^2_{YY} f(x,y) - \sum_{i=1}^{p} \lambda_i \mathrm{D}^2_{YY} h_i(x,y) \in \mathbb{R}^{m\times m}, \end{aligned}$$

and $\lambda \in \mathbb{R}^p$ satisfies $\lambda^{\mathsf{T}}A = \mathrm{D}_Y f(x,y)$.

Proof. By the method of Lagrange multipliers [9] we can form the Lagrangian

$$\mathcal{L}(x,y,\lambda) = f(x,y) - \sum_{i=1}^{p} \lambda_i h_i(x,y). \tag{16}$$

Assume y is optimal for a fixed input x. Since $\mathrm{D}_Y h(x,y)$ is full rank we have that y is a regular point. Then there exists a λ such that the Lagrangian is stationary at the point (y,λ). Here both y and λ are understood to be functions of x. Thus

$$\begin{bmatrix} \left(\mathrm{D}_Y f(x,y) - \sum_{i=1}^{p} \lambda_i \mathrm{D}_Y h_i(x,y)\right)^{\mathsf{T}} \\ h(x,y) \end{bmatrix} = 0_{m+p}, \tag{17}$$

where the first m rows are from differentiating $\mathcal{L}$ with respect to y (that is, $\mathrm{D}_Y\mathcal{L}^{\mathsf{T}}$) and the last p rows are from differentiating $\mathcal{L}$ with respect to λ (that is, $\mathrm{D}_\Lambda\mathcal{L}^{\mathsf{T}}$).

Now observe that at the optimal point y we have that either $\mathrm{D}_Y f(x,y) = 0_{1\times m}$, that is, the optimal point of the unconstrained problem automatically satisfies the constraints, or $\mathrm{D}_Y f(x,y)$ is non-zero and orthogonal to the constraint surface defined by $h(x,y) = 0$. In the first case we can simply set $\lambda = 0_p$. In the second case we have (from the first row in Equation (17))

$$\mathrm{D}_Y f(x,y) = \sum_{i=1}^{p} \lambda_i \mathrm{D}_Y h_i(x,y) = \lambda^{\mathsf{T}}A, \tag{18}$$

for A defined above.

Now, differentiating the gradient of the Lagrangian with respect to x we have

$$\mathrm{D}\begin{bmatrix} \left(\mathrm{D}_Y f(x,y)\right)^{\mathsf{T}} - \sum_{i=1}^{p} \lambda_i \left(\mathrm{D}_Y h_i(x,y)\right)^{\mathsf{T}} \\ h(x,y) \end{bmatrix} = 0. \tag{19}$$

For the first row this gives

$$\begin{aligned} &\mathrm{D}^2_{XY} f + \mathrm{D}^2_{YY} f\mathrm{D}y - \mathrm{D}_Y h^{\mathsf{T}}\mathrm{D}\lambda \\ &\qquad - \sum_{i=1}^{p} \lambda_i\left(\mathrm{D}^2_{XY} h_i + \mathrm{D}^2_{YY} h_i \mathrm{D}y\right) = 0_{m\times n}, \end{aligned} \tag{20}$$

and for the second row we have

$$\mathrm{D}_X h + \mathrm{D}_Y h\mathrm{D}y = 0_{p\times n}. \tag{21}$$

Therefore

$$\begin{aligned} &\begin{bmatrix} \mathrm{D}^2_{YY} f - \sum_{i=1}^{p} \lambda_i \mathrm{D}^2_{YY} h_i & -\mathrm{D}_Y h^{\mathsf{T}} \\ \mathrm{D}_Y h & 0_{p\times p} \end{bmatrix} \begin{bmatrix} \mathrm{D}y \\ \mathrm{D}\lambda \end{bmatrix} \\ &\qquad + \begin{bmatrix} \mathrm{D}^2_{XY} f - \sum_{i=1}^{p} \lambda_i \mathrm{D}^2_{XY} h_i \\ \mathrm{D}_X h \end{bmatrix} = 0_{(m+p)\times n}, \end{aligned} \tag{22}$$

where all functions are evaluated at (x,y). We can now solve by variable elimination [11] to get

$$\mathrm{D}\lambda(x) = \left(AH^{-1}A^{\mathsf{T}}\right)^{-1}\left(AH^{-1}B - C\right) \tag{23}$$

$$\mathrm{D}y(x) = H^{-1}A^{\mathsf{T}}\left(AH^{-1}A^{\mathsf{T}}\right)^{-1}\left(AH^{-1}B - C\right) - H^{-1}B, \tag{24}$$

with A, B, C, and H as defined above. □

The Lagrange multipliers λ in Proposition 4.5 are often made available by the method used to solve for $y(x)$, e.g., in the case of primal-dual methods for convex optimization problems. Where it is not provided by the solver it can be computed explicitly solving $\lambda^{\mathsf{T}}A = \mathrm{D}_Y f(x,y)$, which has unique analytic solution

$$\lambda = (AA^{\mathsf{T}})^{-1}A\left(\mathrm{D}_Y f\right)^{\mathsf{T}}. \tag{25}$$

Thus given optimal y we can find λ.

Although the result looks expensive to compute, much of the computation is shared since $H^{-1}A^{\mathsf{T}}$ is independent of the coordinate of x for which the gradient is being computed. Moreover, H need only be factored once and then reused in computing $H^{-1}B$ for each input dimension, that is, column of B. And for many problems H has structure that can be further exploited to speed calculation of $\mathrm{D}y$.

Inequality constrained problems encompass a richer class of deep declarative nodes. Gould *et al.* [31] addressed the question of computing gradients for these problems by approximating the inequality constraints with a logarithmic barrier [11] and leveraging the result for unconstrained problems. However, it is also possible to calculate the gradient without resorting to approximation by leveraging the KKT optimality conditions [10] as has been shown in previous work for the special case of convex optimization problems [2], [4]. Here we present a result for the more general non-convex case.

We first observe that for a smooth objective function f the optimal solution $y(x)$ to the inequality constrained problem in Equation (13) is also a local (possibly global) minimum of the problem with any or all inactive inequality constraints removed. Moreover, the derivative $\mathrm{D}y(x)$ is the same for both problems. As such, without loss of generality, we can assume that all inequality constraints are active.

Proposition 4.6 (Inequality Constrained). *Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$, $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^p$ and $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$. Let*

$$\begin{array}{lll} y(x) \in & \operatorname{arg\,min}_{u \in \mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h_i(x,u) = 0, \quad i = 1,\ldots,p \\ & & g_i(x,u) \le 0, \quad i = 1,\ldots,q. \end{array}$$

Assume that $y(x)$ exists, that f, h and g are second-order differentiable in the neighborhood of $(x, y(x))$, and that all inequality constraints are active at $y(x)$. Let $\tilde{h} = [h_1,\ldots,h_p, g_1,\ldots,g_q]$ and assume $\operatorname{rank}(\mathrm{D}_Y\tilde{h}(x,y)) = p+q$. Then for H non-singular

$$\mathrm{D}y(x) = H^{-1}A^{\mathsf{T}}\left(AH^{-1}A^{\mathsf{T}}\right)^{-1}\left(AH^{-1}B - C\right) - H^{-1}B,$$

where

$$\begin{aligned} A &= \mathrm{D}_Y\tilde{h}(x,y) \in \mathbb{R}^{(p+q)\times m} \\ B &= \mathrm{D}^2_{XY}f(x,y) - \sum_{i=1}^{p+q}\lambda_i\mathrm{D}^2_{XY}\tilde{h}_i(x,y) \in \mathbb{R}^{m\times n} \\ C &= \mathrm{D}_X\tilde{h}(x,y) \in \mathbb{R}^{(p+q)\times n} \\ H &= \mathrm{D}^2_{YY}f(x,y) - \sum_{i=1}^{p+q}\lambda_i\mathrm{D}^2_{YY}\tilde{h}_i(x,y) \in \mathbb{R}^{m\times m}, \end{aligned}$$

and $\lambda \in \mathbb{R}^{p+q}$ satisfies $\lambda^{\mathsf{T}}A = \mathrm{D}_Y f(x,y)$ with $\lambda_i \le 0$ for $i = p+1,\ldots,p+q$. The gradient $\mathrm{D}y(x)$ is one-sided whenever there exists an $i > p$ such that $\lambda_i = 0$.

Proof. The existence of λ and non-negativity of λ_i for $i = p+1,\ldots,p+q$ comes from the fact that $y(x)$ is a regular point (because $\mathrm{D}_Y\tilde{h}(x,y)$ is full rank). The remainder of the proof follows Proposition 4.5. One-sidedness of $\mathrm{D}y(x)$ results from the fact that $\lambda_i(x)$, for $i = p+1,\ldots,p+q$, is not differentiable at zero. □

The main difference between the result for inequality constrained problems (Proposition 4.6) and the simpler equality constrained case (Proposition 4.5) is that the gradient is discontinuous at points where any of the Lagrange multipliers for (active) inequality constraints are zero. We illustrate this by analysing a deep declarative node with a single inequality constraint,

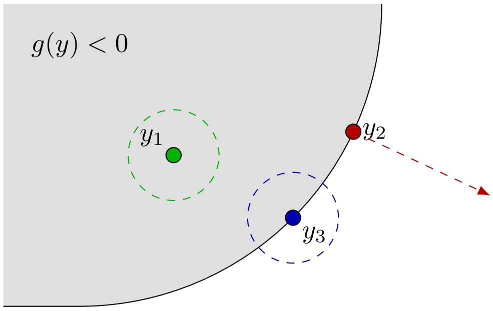


Fig. 3. Illustration of different scenarios for the solution to inequality constrained deep declarative nodes. In the first scenario (y_1) the solution is a local minimum strictly satisfying the constraints. In the second scenario (y_2) the solution is on the boundary of the constraint set with the negative gradient of the objective pointing outside of the set. In the third scenario (y_3) the solution is on the boundary of the constraint set and is also a local minimum.

$$\begin{array}{lll} y \in & \operatorname{arg\,min}_{u \in \mathbb{R}^m} & f(x,u) \\ & \text{subject to} & g(x,u) \le 0. \end{array} \tag{26}$$

where both f and g are smooth.

Consider three scenarios (see Fig. 3). First, if the constraint is inactive at solution y, that is, $g(x,y) < 0$, then we must have $\mathrm{D}_Y f(x,y) = 0$ and can take $\mathrm{D}y(x)$ to be the same as for the unconstrained case. Second, if the constraint is active at the solution, that is, $g(x,y) = 0$ but $\mathrm{D}_Y f(x,y) \neq 0$ then in must be that $\lambda \neq 0$, and from Proposition 4.6, the gradient $\mathrm{D}y(x)$ is the same as if we had replaced the inequality constraint with an equality. Last, if the constraint is active at y and $\mathrm{D}_Y f(x,y) = 0$ then $\mathrm{D}y(x)$ is undefined. In this last scenario we can choose either the unconstrained or constrained gradient in order to obtain an update direction for back-propagation similar to how functions such as rectified linear units are treated in standard deep learning models.

4.4 Simpler Equality Constraints

Often there is only a single fixed equality constraint that does not depend on the inputs—we consider such problems in Section 5.2—or the equality constraints are all linear. The above result can be specialized for these cases.

Corollary 4.7. *Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^m \to \mathbb{R}$. Let*

$$\begin{array}{lll} y(x) \in & \operatorname{arg\,min}_{u \in \mathbb{R}^m} & f(x,u) \\ & \text{subject to} & h(u) = 0. \end{array}$$

Assume that $y(x)$ exists, that $f(x,u)$ and $h(u)$ are second-order differentiable in the neighborhood of $(x, y(x))$ and $y(x)$, respectively, and that $\mathrm{D}_Y h(y(x)) \neq 0$. Then

$$\mathrm{D}y(x) = \left(\frac{H^{-1}aa^{\mathsf{T}}H^{-1}}{a^{\mathsf{T}}H^{-1}a} - H^{-1}\right)B,$$

where

$$a = (\mathrm{D}_Y h(y))^{\mathsf{T}} \in \mathbb{R}^m,$$
$$B = \mathrm{D}^2_{XY} f(x,y) \in \mathbb{R}^{m\times n},$$
$$H = \mathrm{D}^2_{YY} f(x,y) - \lambda \mathrm{D}^2_{YY} h(y) \in \mathbb{R}^{m\times m}, \text{ and}$$
$$\lambda = \left(\frac{\partial h}{\partial y_i}(y)\right)^{-1} \frac{\partial f}{\partial y_i}(x,y) \in \mathbb{R} \text{ for any } \frac{\partial h}{\partial y_i}(y) \neq 0.$$

Proof. Follows from Proposition 4.5 with $p=1$, $\mathrm{D}_X h \equiv 0_{1\times n}$ and $\mathrm{D}^2_{XY} h \equiv 0_{m\times n}$. □

Observation 4.8. *When the constraint function is independent of x we have $\mathrm{D}y(x) \perp \mathrm{D}_Y h(y)$, from Equation (21) with $\mathrm{D}_X h = 0$. In other words, y can only change (as a function of x) in directions that maintain the constraint $h(y)=0$.*

Given this simpler form of the gradient for declarative nodes with a single equality constraint, one might be tempted to naively combine multiple equality constraints into a single constraint, for example $\tilde{h}(x,u) \triangleq \sum_{i=1}^p h_i^2(x,u) = 0$ (or any other function of the h_i's that is identically zero if any only if the h_i's are all zero). However, this will not work as it violates the assumptions of the method of Lagrange multipliers, namely that $\mathrm{D}_Y \tilde{h}(x,y) \neq 0$ at the optimal point.

The following result is for the common case of multiple fixed linear equality constraints that do not depend on the inputs (which has also been reported previously [31]).

Corollary 4.9. *Consider a function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and let $A \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$ with $\operatorname{rank}(A) = p$ define a set of p under-constrained linear equations $Au = d$. Let*

$$\begin{array}{rl} y(x) \in \text{arg min}_{u\in\mathbb{R}^m} & f(x,u) \\ \text{subject to} & Au = d. \end{array}$$

Assume that $y(x)$ exists and that $f(x,u)$ is second-order differentiable in the neighborhood of $(x, y(x))$. Then

$$\mathrm{D}y(x) = \left(H^{-1}A^{\mathsf{T}}(AH^{-1}A^{\mathsf{T}})^{-1}AH^{-1} - H^{-1}\right)B,$$

where $H = \mathrm{D}^2_{YY} f(x,y)$ and $B = \mathrm{D}^2_{XY} f(x,y)$.

Proof. Follows from Proposition 4.5 with $h(x,u) \triangleq Au - d$ so that $\mathrm{D}_X h \equiv 0_{p\times n}$, $\mathrm{D}^2_{XY} h \equiv 0_{m\times n}$ and $\mathrm{D}^2_{YY} h \equiv 0_{m\times m}$. □

4.5 Geometric Interpretation

Consider, for simplicity, a declarative node with scalar input x ($n=1$) and a single equality constraint ($p=1$). An alternative form for the gradient from Proposition 4.5, which lends itself to geometric interpretation as shown in Fig. 4, is given by

$$\mathrm{D}y(x) = H^{-\frac{1}{2}}\left(\tilde{b} - (\hat{a}^{\mathsf{T}}\tilde{b})\hat{a} + \frac{c}{\|\tilde{a}\|}\hat{a}\right), \tag{27}$$

where $\tilde{a} = -H^{-\frac{1}{2}}a$, $\tilde{b} = -H^{-\frac{1}{2}}b$ and $\hat{a} = \|\tilde{a}\|^{-1}\tilde{a}$. First, consider the case when the constraints do not depend on the input ($c=0$). The unconstrained derivative $\mathrm{D}y_{\text{unc}}(x) = H^{-\frac{1}{2}}\tilde{b}$ (equality when $\lambda = 0$) is corrected by removing the component perpendicular to the constraint surface. That is, the unconstrained gradient is projected to the tangent plane of the constraint surface, as shown in Fig. 4. The gradient is therefore parallel to the constraint surface, ensuring that gradient descent with an infinitesimal step size does not push the solution away from the constraint surface. Thus, the equation encodes the following procedure: (i) pre-multiply the unconstrained gradient vector $\mathrm{D}y_{\text{unc}}(x)$ by $H^{\frac{1}{2}}$; (ii) project onto the linearly transformed (by $H^{-\frac{1}{2}}$) tangent plane of the constraint surface; and (iii) pre-multiply by $H^{-\frac{1}{2}}$. Observe that the equation compensates for the curvature H before the projection. The same intuition applies to the case where the constraints depend on the inputs ($c \neq 0$), with an additional bias term accounting for how the constraint surface changes with the input x.

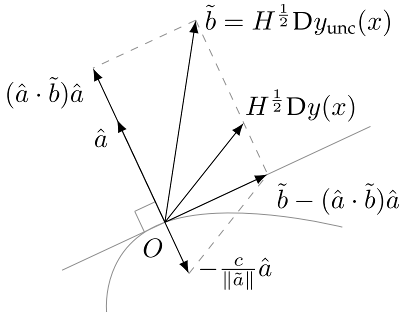

$\tilde{b} = H^{\frac{1}{2}}\mathrm{D}y_{\text{unc}}(x)$
$(\hat{a}\cdot\tilde{b})\hat{a}$
$H^{\frac{1}{2}}\mathrm{D}y(x)$
$\hat{a}$
$\tilde{b} - (\hat{a}\cdot\tilde{b})\hat{a}$
O
$-\frac{c}{\|\tilde{a}\|}\hat{a}$

Fig. 4. Geometry of the gradient for an equality constrained optimization problem. The unconstrained gradient $\mathrm{D}y_{\text{unc}}(x)$ is corrected to ensure that the solution remains on the constraint surface after gradient descent with an infinitesimal step size.

4.6 Relationship to Conventional Neural Networks

Deep declarative networks subsume traditional feed-forward and recurrent neural networks in the sense that any layer in the network that explicitly defines its outputs in terms of a differentiable function of its inputs can be reduced to a declarative processing node. A similar observation was made by Amos and Kolter [4], and the notion is formalized in the following proposition.

Proposition 4.10. *Let $\tilde{f} : \mathbb{R}^n \to \mathbb{R}^m$ be an explicitly defined differentiable forward processing function for a neural network layer. Then we can alternatively define the behavior of the layer declaratively as*

$$y = \operatorname*{arg\,min}_{u\in\mathbb{R}^m} \frac{1}{2}\|u - \tilde{f}(x)\|^2.$$

Proof. The objective function $f(x,u) = \frac{1}{2}\|u - \tilde{f}(x)\|^2$ is strongly convex. Differentiating it with respect to u and setting to zero we get $y = \tilde{f}(x)$. By Proposition 4.4 we have $H = I$ and $B = -\mathrm{D}_X\tilde{f}(x)$, giving $\mathrm{D}y(x) = \mathrm{D}\tilde{f}(x)$. □

Of course, despite the appeal of having a single mathematical framework for describing processing nodes, there is no reason to do this in practice since imperative and declarative nodes can co-exist in a network.

Composability. One of the key design principles in deep learning is that models should define their output as the composition of many simple functions (such as $y = \tilde{f}(\tilde{g}(x))$). Just like conventional neural networks deep declarative networks can also be composed of many optimization problems arranged in levels as the following example shows:

$$\begin{aligned} y \in \quad & \arg\min_u \quad f(z,u) \\ & \text{subject to} \quad z \in \arg\min_v g(x,v). \end{aligned} \tag{28}$$

Here the gradients combine by the chain rule of differentiation as one might expect

$$\begin{aligned} \mathrm{D}z(x) &= -H_g^{-1}B_g(x,z) \\ \mathrm{D}y(z) &= -H_f^{-1}B_f(z,y) \\ \mathrm{D}y(x) &= \mathrm{D}y(z)\mathrm{D}z(x), \end{aligned} \tag{29}$$

where $H_g = \mathrm{D}^2_{ZZ}g(x,z)$, etc. Here, during back propagation in computing $\mathrm{D}J(x)$ from $\mathrm{D}J(y)$ for training objective J, it may be more efficient to first compute $\mathrm{D}J(z)$ as

$$\mathrm{D}J(x) = \underbrace{(\mathrm{D}J(y)\mathrm{D}y(z))}_{\mathrm{D}J(z)}\mathrm{D}z(x), \tag{30}$$

which we will discuss further in Section 6.1.

4.7 Extensions and Discussion

4.7.1 Feasibility Problems

Feasibility problems, where we simply seek a solution $y(x)$ that satisfies a set of constraints (themselves functions of x), can be formulated as optimization problems [11], and hence declarative nodes. Here the objective is constant (or infinite for an infeasible point) but the problem is more naturally written as

$$\begin{aligned} \text{find} \quad & u \\ \text{subject to} \quad & h_i(x,u) = 0, \quad i = 1,\ldots,p \\ & g_i(x,u) \leq 0, \quad i = 1,\ldots,q. \end{aligned} \tag{31}$$

The gradient can be derived by removing inactive inequality constraints and maintaining the invariant $\tilde{h}(x,y(x)) = 0$ where $\tilde{h} = [h_1,\ldots,h_p,g_1,\ldots,g_q]$, i.e., following the constraint surface. Implicit differentiation gives

$$A\mathrm{D}y(x) + C = 0, \tag{32}$$

where $A = \mathrm{D}_Y\tilde{h}(x,y)$ and $C = \mathrm{D}_X\tilde{h}(x,y)$ as defined in Proposition 4.6. This is the same set of linear equations as the second row of the Lagrangian in our proof of Proposition 4.5, and has a unique solution whenever $\mathrm{rank}(A) = m$. [4]

4.7.2 Non-Smooth Objective and Constraints

Propositions 4.5 and 4.6 showed how to compute derivatives of the solution to parametrized equality constrained optimization problems with second-order differentiable objective and constraint functions (in the neighborhood of the solution). However, we can also define declarative nodes where the objective and constraints are non-smooth (that is, non-differentiable at some points). To enable back-propagation learning with such declarative nodes all we require is a local descent direction (or Clarke generalized gradient [16]). This is akin to the approach taken in standard deep learning models with non-smooth activation functions such as the rectified linear unit, $y = \max\{x, 0\}$, which is non-differentiable at $x = 0$. [5] We explore this in practice when we consider robust pooling and projecting onto the boundary of L_p-balls in our illustrative examples and experiments below.

4.7.3 Non-Regular Solutions

The existence of the Lagrange multipliers λ (and their uniqueness) in our results above was guaranteed by our assumption that $y(x)$ is a regular point. Moreover, the fact that $A = \mathrm{D}_Y h$ is full rank (and H non-singular) ensures that $\mathrm{D}y(x)$ can be computed. Under other constraint qualifications, such as Slater's condition [11], the λ are also guaranteed to exist but may not be unique. Moreover, we may have $\mathrm{rank}(A) \neq p + q$. If $p + q > m$ but A is full rank, which can happen if multiple constraints are active at $y(x)$, then we can directly solve the over-determined system of equations $A\mathrm{D}y(x) = -C$. On the other hand, if A is rank deficient then we have redundant constraints and must reduce A to a full rank matrix by removing linearly dependent rows (and fixing the corresponding λ_i to zero) or resort to using a pseudo-inverse to approximate $\mathrm{D}y(x)$. This can happen if two or more (nonlinear inequality) constraints are active at $y(x)$ and tangent to each other at that point.

4.7.4 Non-Unique Solutions

In presenting deep declarative nodes we have made no assumption about the uniqueness of the solution to the optimization problem (i.e., output of the node). Indeed, many solutions may exist and our gradients are still valid. Needless to say, it is essential to use the same value of y in the backward pass as was computed during the forward pass (which is automatically handled in modern deep learning frameworks by caching the forward computations). But there are other considerations too.

Consider for now the unconstrained case (Equation (11)). If $\mathrm{D}^2_{YY}f(x,y) \succ 0$ then f is strongly convex in the neighborhood of y. Hence y is an isolated minimizer (that is, a unique minimizer within its neighborhood), and the gradient derived in Proposition 4.4 holds for all such points. Nevertheless, when performing parameter learning in a deep declarative network it is important to be consistent in solving optimization problems with multiple minima to avoid oscillating between solutions as we demonstrate with the following pathological example. Consider

$$\begin{aligned} y \in \quad & \arg\min_{u \in \mathbb{R}} \quad 0 \\ & \text{subject to} \quad (u-1)^2 - x^2 = 0, \end{aligned} \tag{33}$$

which has solution $y = 1 \pm x$ and, by inspection,

$$\mathrm{D}y = \begin{cases} -1, & \text{for } y = 1 - x \\ +1, & \text{for } y = 1 + x \end{cases}. \tag{34}$$

Depending on the choice of optimal solution the gradient of the loss being propagated backwards through the node will point in opposite directions, which is problematic for end-to-end learning.

[4] In Proposition 4.5 we assumed that $\mathrm{rank}(A) = p \leq m$. Typically we have $p < m$ and so $A\mathrm{D}y(x) = -C$ is under-determined, and the curvature of the objective resolves $\mathrm{D}y(x)$.

[5] Incidentally, the (elementwise) rectified linear unit can be defined declaratively as $y = \arg\min_{u \in \mathbb{R}^n_+} \frac{1}{2}\|u - x\|_2^2$.

Singular Hessians. Now, if $\mathrm{D}^2_{YY}f(x,y)$ is singular then y may not be an isolated minimizer. This can occur, for example, when y is an over-parametrized descriptor of some physical property such as an unnormalized quaternion representation of a 3D rotation. In such cases we cannot use Proposition 4.4 to find the gradient. In fact the gradient is undefined. There are three strategies we can consider for such points. First, reformulate the problem to use a minimal parametrization or introduce constraints to remove degrees of freedom thereby making the solution unique. Second, make the objective function strongly convex around the solution, for example by adding the proximal term $\frac{\delta}{2}\|u-y\|^2$ for some small $\delta > 0$. Third, use a pseudo-inverse to solve the linear equation $\mathrm{D}^2_{YY}f(x,y)\Delta y = -\mathrm{D}^2_{XY}(x,y)$ for Δy and take Δy as a descent direction. Supposing $H^\dagger$ is a pseudo-inverse of $\mathrm{D}^2_{YY}f(x,y)$, we can compute an entire family of solutions as

$$\Delta y \in \{-H^\dagger B + (I - H^\dagger H)Z \mid Z \in \mathbb{R}^{m\times n}\}, \tag{35}$$

where $H = \mathrm{D}^2_{YY}f(x,y)$ and $B = \mathrm{D}^2_{XY}f(x,y)$.

As a toy example, motivated by the quaternion representation, consider the following problem that aligns the output vector with an input vector $x \neq 0$ in $\mathbb{R}^4$,

$$y \in \operatorname*{arg\,min}_{u\in\mathbb{R}^4} f(x,u) \quad \text{with } f(x,u) \triangleq \frac{-x^\mathsf{T}u}{\|u\|_2}, \tag{36}$$

which has solution $y = \alpha x$ for arbitrary $\alpha > 0$. Here we have $\mathrm{D}^2_{YY}f(x,y) = \alpha^{-2}\|x\|_2^{-1}\left(I - \|x\|_2^{-2}xx^\mathsf{T}\right)$, which is singular (by the matrix inversion lemma [30]). Fixing one degree of freedom resolves the problem, which in this case is easily done by forcing the output vector to be normalized

$$\begin{array}{rll} y \in & \operatorname*{arg\,min}_{u\in\mathbb{R}^4} & -x^\mathsf{T}u \\ & \text{subject to} & \|u\|_2 = 1. \end{array} \tag{37}$$

Alternatively we can compute the Moore–Penrose pseudo-inverse [41] of $\mathrm{D}^2_{YY}f(x,y)$ to obtain

$$\mathrm{D}y = \alpha\left(I - \frac{1}{\|x\|_2^2}xx^\mathsf{T}\right), \tag{38}$$

which is the same gradient as the constrained case where the solution is fixed to have $\alpha = 1/\|x\|_2$ for this problem.

5 Illustrative Examples

In this section we provide examples of unconstrained and constrained declarative nodes that illustrate the theory developed above. For each example we begin with a standard operation in deep learning models, which is usually implemented as an imperative node. We show how the operations can be equivalently implemented in a declarative framework, and then generalize the operations to situations where an explicit implementation is not possible, but where the operation results in more desirable model behavior.

5.1 Robust Pooling

Average (or mean) pooling is a standard operation in deep learning models where multi-dimensional feature maps are averaged over one or more dimensions to produce a summary statistic of the input data. For the one-dimensional case, $x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$, we have

$$y = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{39}$$

and

$$\mathrm{D}y = \left[\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n}\right] = \frac{1}{n}1^\mathsf{T}. \tag{40}$$

As a declarative node the mean pooling operator is

$$y \in \operatorname*{arg\,min}_{u\in\mathbb{R}} \sum_{i=1}^{n} \frac{1}{2}(u - x_i)^2. \tag{41}$$

While the solution, and hence gradients, can be expressed in closed form, it is well-known that as a summary statistic the mean is very sensitive to outliers. We can make the statistic more robust by replacing the quadratic penalty function $\phi^{\text{quad}}(z) = \frac{1}{2}z^2$ with one that is less sensitive to outliers. The pooling operation can then be generalized as

$$y \in \operatorname*{arg\,min}_{u\in\mathbb{R}} \sum_{i=1}^{n} \phi(u - x_i; \alpha), \tag{42}$$

where ϕ is a penalty function taking scalar argument and controlled by parameter α.

For example, the Huber penalty function defined as

$$\phi^{\text{huber}}(z;\alpha) = \begin{cases} \frac{1}{2}z^2 & \text{for } |z| \leq \alpha \\ \alpha(|z| - \frac{1}{2}\alpha) & \text{otherwise.} \end{cases} \tag{43}$$

is more robust to outliers. The Huber penalty is convex and hence the pooled value can be computed efficiently via Newton's method or gradient descent [11]. However, no closed-form solution exists. Moreover, the solution set may be an interval of points. The pseudo-Huber penalty [32],

$$\phi^{\text{pseudo}}(z;\alpha) = \alpha^2\left(\sqrt{1 + \left(\frac{z}{\alpha}\right)^2} - 1\right), \tag{44}$$

has similar behaviour to the Huber but is strongly convex so the optimal solution (pooled value) is unique.

The Welsch penalty function [20] is even more robust to outliers flattening out to a fixed cost as $|z| \to \infty$,

$$\phi^{\text{welsch}}(z;\alpha) = 1 - \exp\left(-\frac{z^2}{2\alpha^2}\right). \tag{45}$$

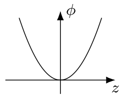
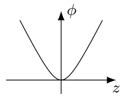
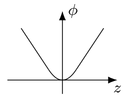
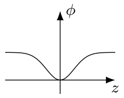
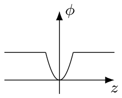
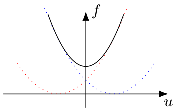
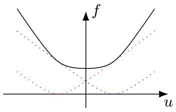
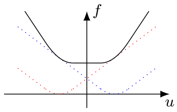
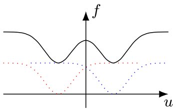
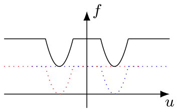
However it is non-convex and so obtaining the solution $y(x)$ is non-trivial. Nevertheless, *given a solution* we can use Proposition 4.4 to compute a gradient for back-propagation parameter learning.

Going one step further we can defined the truncated quadratic penalty function,

$$\phi^{\text{trunc}}(z;\alpha) = \begin{cases} \frac{1}{2}z^2 & \text{for } |z| \leq \alpha \\ \frac{1}{2}\alpha^2 & \text{otherwise.} \end{cases}$$

In addition to being non-convex, the function is also non-smooth. The solution amounts to finding the maximal set of

TABLE 1
The Gradient of the Estimate for a Robust Mean Over a Vector of Values for Various Penalty Functions $\phi(z;\alpha)$ When it Exists

	QUADRATIC	PSEUDO-HUBER	HUBER	WELSCH	TRUNC. QUAD.
$\phi(z;\alpha)$	$\frac{1}{2}z^2$	$\alpha^2\left(\sqrt{1+\left(\frac{z}{\alpha}\right)^2}-1\right)$	$\begin{cases}\frac{1}{2}z^2 & \text{for } \lvert z\rvert \le \alpha \\ \alpha(\lvert z\rvert - \frac{1}{2}\alpha) & \text{otherwise.}\end{cases}$	$1-\exp\left(-\frac{z^2}{2\alpha^2}\right)$	$\begin{cases}\frac{1}{2}z^2 & \text{for } \lvert z\rvert \le \alpha \\ \frac{1}{2}\alpha^2 & \text{otherwise.}\end{cases}$
					
					
	closed-form, convex, smooth, unique solution	convex, smooth, unique solution	convex, non-smooth (C^1), non-isolated solutions	non-convex, smooth, isolated solutions	non-convex, non-smooth (C^0), isolated solutions
$\mathrm{D}^2_{YY}f(x,y)$	n	$\sum_{i=1}^n\left(1+\left(\frac{y-x_i}{\alpha}\right)^2\right)^{-3/2}$	$\sum_{i=1}^n \llbracket \lvert y-x_i\rvert \le \alpha \rrbracket$	$\sum_{i=1}^n \frac{\alpha^2-(y-x_i)^2}{\alpha^4}\exp\left(-\frac{(y-x_i)^2}{2\alpha^2}\right)$	$\sum_{i=1}^n \llbracket \lvert y-x_i\rvert \le \alpha \rrbracket$
$\mathrm{D}^2_{XY}f(x,y)$	-1_n^T	$\mathbf{vec}\left\{-\left(1+\left(\frac{y-x_i}{\alpha}\right)^2\right)^{-3/2}\right\}^T$	$\mathbf{vec}\left\{-\llbracket \lvert y-x_i\rvert \le \alpha \rrbracket\right\}$	$\mathbf{vec}\left\{\frac{(y-x_i)^2-\alpha^2}{\alpha^4}\exp\left(-\frac{(y-x_i)^2}{2\alpha^2}\right)\right\}^T$	$\mathbf{vec}\left\{-\llbracket \lvert y-x_i\rvert \le \alpha \rrbracket\right\}^T$
$\mathrm{D}y(x)$	$\frac{1}{n}1_n^T$	$\mathbf{vec}\left\{\frac{w_i}{\sum_{j=1}^n w_j}\right\}^T$ where $w_i=\left(1+\left(\frac{y-x_i}{\alpha}\right)^2\right)^{-3/2}$	$\mathbf{vec}\left\{\frac{\llbracket \lvert y-x_i\rvert \le \alpha \rrbracket}{\sum_{j=1}^n \llbracket \lvert y-x_j\rvert \le \alpha \rrbracket}\right\}^T$	$\mathbf{vec}\left\{\frac{w_i}{\sum_{j=1}^n w_j}\right\}^T$ where $w_i=\frac{\alpha^2-(y-x_i)^2}{\alpha^4}\exp\left(-\frac{(y-x_i)^2}{2\alpha^2}\right)$	$\mathbf{vec}\left\{\frac{\llbracket \lvert y-x_i\rvert \le \alpha \rrbracket}{\sum_{j=1}^n \llbracket \lvert y-x_j\rvert \le \alpha \rrbracket}\right\}^T$

The robust estimate is found as $y(x)=\arg\min_{u\in\mathbb{R}} f(x,u)$ with $f(x,u)=\sum_{i=1}^n \phi(u-x_i;\alpha)$. The first plot (row 2) shows the penalty function, the second plot (row 3) shows an example $f(x,u)$ for $x=(-2\alpha,2\alpha)\in\mathbb{R}^2$. Plots have been drawn at different scales to enhance visualization of shape differences between the penalty functions. Despite not being able to solve the problem in closed form for most penalty functions, given a solution the gradient $\mathrm{D}y(x)$ can still be calculated. Notice the similarity in gradient forms due to the objective f being composed of a sum of independent penalties, each penalty symmetric in x and y. Moreover, the Huber and truncated quadratic have exactly the same gradient form (when it exists) even though their solutions may be different. Under some conditions Huber and Welsch may result in a zero $\mathrm{D}^2_{YY}f$ and, hence, an undefined gradient. However, we did not see this in practice. More importantly, computation of the Welsch gradient is sensitive to numerical underflow and care should be taken to appropriately scale the w_i's before dividing by their sum.

values all within some fixed distance α from the mean. The objective $f(x,u)=\sum_{i=1}^n \phi^{\text{trunc}}(u-x_i;\alpha)$ is not differentiable at points where there exists an i such that $|y(x)-x_i|=\alpha$, in the same way that the rectified linear unit is non-differentiable at zero. Nevertheless, we can still compute a gradient almost everywhere (and take a one-sided gradient at non-differentiable points).

The various penalty functions are depicted in Table 1. When used to define a robust pooling operation there is no closed-form solution for all but the quadratic penalty. Yet the gradient of the solution with respect to the input data for all robust penalties can be calculated using Proposition 4.4 as summarized in the table.

5.2 L_p-Sphere or L_p-Ball Projection

euclidean projection onto an L_2-sphere, equivalent to L_2 normalization, is another standard operation in deep learning models. For $x\in\mathbb{R}^n \mapsto y\in\mathbb{R}^n$, we have

$$y=\frac{1}{\|x\|_2}x, \tag{47}$$

and

$$\mathrm{D}y(x)=\frac{1}{\|x\|_2}\left(I-\frac{1}{\|x\|_2^2}xx^{\mathsf{T}}\right). \tag{48}$$

As a declarative node the L_2-sphere projection operator is

$$\begin{array}{rll} y\in & \underset{u\in\mathbb{R}^n}{\arg\min} & \frac{1}{2}\|u-x\|_2^2 \\ & \text{subject to} & \|u\|_2=1. \end{array} \tag{49}$$

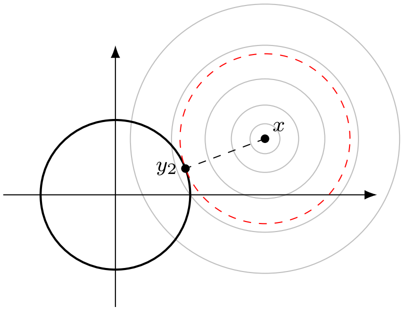
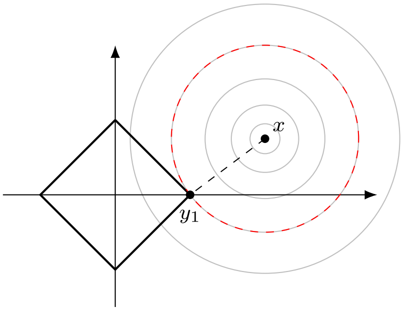
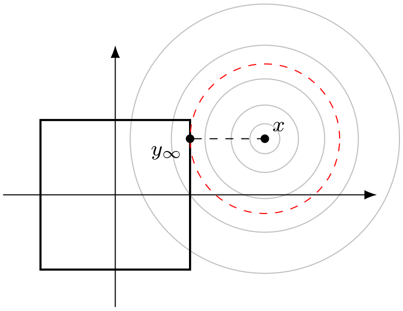
While the solution, and hence gradients, can be expressed in closed-form, it may be desirable to use other L_p-spheres or balls, for regularization, sparsification, or to improve generalization performance [43], as well as for adversarial robustness. The projection operation onto an L_p-sphere can then be generalized as

$$\begin{array}{rll} y_p\in & \underset{u\in\mathbb{R}^n}{\arg\min} & \frac{1}{2}\|u-x\|_2^2 \\ & \text{subject to} & \|u\|_p=1, \end{array} \tag{50}$$

where $\|\cdot\|_p$ is the L_p-norm.

For example, projecting onto the L_1-sphere has no closed-form solution, however Duchi *et al.* [25] provide an efficient $O(n)$ solver. Similarly, projecting onto the L_∞-sphere has an efficient (trivial) solver. Given a solution from one of these solvers, we can use Proposition 4.5 to compute a gradient. For both cases, the constraint functions h are non-smooth and so are not differentiable whenever the optimal projection y lies on an $(n-k)$-face for $2\le k\le n$. For example, when y lies on a vertex, changes in x may not have any effect on y. While the

TABLE 2
The Gradient of the Euclidean Projection Onto Various L_p-Spheres With Constraint Functions h, When it Exists

	L_2	L_1	L_∞
$h(u)$	$\lVert u\rVert_2 - 1 = \sqrt{\sum_{i=1}^n u_i^2} - 1$	$\lVert u\rVert_1 - 1 = \sum_{i=1}^n \lvert u_i\rvert - 1$	$\lVert u\rVert_\infty - 1 = \max_i\{\lvert u_i\rvert\} - 1$
			
	closed-form, smooth, unique[†] solution	non-smooth (C^0), isolated solutions	non-smooth (C^0), isolated solutions
$\mathrm{D}_Y h(y)$	y	$\mathbf{vec}\,\{\mathbf{sign}\,(y_i)\}^T$	$\mathbf{vec}\,\{[\![i \in I^\star]\!]\,\mathbf{sign}\,(y_i)\}^T$ $I^\star = \{i \mid \lvert y_i\rvert \geq \lvert y_j\rvert\ \forall j\}$
$\mathrm{D}_{YY}^2 h(y)$	$I - yy^T$	$0_{n\times n}$	$0_{n\times n}$
λ	$1 - \lVert x\rVert_2$	$\mathbf{sign}\,(y_i)\,(y_i - x_i)\ \forall i$	$[\![i \in I^\star]\!]\,\mathbf{sign}\,(y_i)\,(y_i - x_i)\ \forall i \in I^\star$
$\mathrm{D}y(x)$	$\frac{1}{\lVert x\rVert_2}\left(I - yy^T\right)$	$I - \frac{\mathrm{D}_Y h(y)^T \mathrm{D}_Y h(y)}{\mathrm{D}_Y h(y) \mathrm{D}_Y h(y)^T}$	$I - \frac{\mathrm{D}_Y h(y)^T \mathrm{D}_Y h(y)}{\mathrm{D}_Y h(y) \mathrm{D}_Y h(y)^T}$
$\mathrm{D}^{\mathrm{z}} y(x)$	$\frac{1}{\lVert x\rVert_2}\left(I - yy^T\right)$	$\mathbf{diag}\,(\lvert \mathrm{D}_Y h(y)\rvert) - \frac{\mathrm{D}_Y h(y)^T \mathrm{D}_Y h(y)}{\mathrm{D}_Y h(y) \mathrm{D}_Y h(y)^T}$	$I - \mathbf{diag}\,(\lvert \mathrm{D}_Y h(y)\rvert)$

The projection is found as $y(x) = \arg\min_{u\in\mathbb{R}^n} f(x,u)$ subject to $h(u) = 0$, with $f(x,u) = \frac{1}{2}\|u - x\|_2^2$. In all cases, $B = \mathrm{D}_{XY}^2 f(x,y) = -I$, and $\mathrm{D}_{YY}^2 f(x,y) = I$. The plots show the euclidean projection of an example point $x \in \mathbb{R}^2$ onto L_p-spheres for $p = 1, 2$ and ∞. Despite not being able to solve the problem in closed form for L_1 and L_∞ constraints, given a solution the gradient $\mathrm{D}y(x)$ can still be calculated. While $\mathrm{D}y(x)$ provides a valid local descent direction in all cases, the gradient can be altered to prefer a zero gradient along plateau dimensions by zeroing the rows and columns corresponding to the zero (L_1) or non-zero (L_∞) elements of $\mathrm{D}_Y h(y)$. This gradient $\mathrm{D}^{\mathrm{z}} y(x)$ is obtained by masking $\mathrm{D}y(x)$ with pp^T for L_1 or $\bar{p}\bar{p}^T$ for L_∞, where $p = |\mathrm{D}_Y h(y)|$ is treated as a Boolean vector. [†]Except for $x = 0$ where the solution is non-isolated (the entire sphere). For L_1 and L_∞ the solution is unique when x is outside the ball but can be non-unique for x inside the ball.

gradient obtained from Proposition 4.5 provides a valid local descent direction (Clarke generalized gradient [16]), it can be modified to prefer a zero gradient at these plateau dimensions by masking the gradient, and thereby becoming identical to the gradient obtained by numerical differentiation methods.

The various constraint functions and gradients are given in Table 2. When used to define a projection operation, only the L_2 case has a closed-form solution. Nonetheless, the gradient of the solution with respect to the input data for all constraint functions can be calculated using Proposition 4.5 as summarized in the table.

We can also consider a declarative node that projects onto the unit L_p-ball with output defined as

$$y_p^\circ \in \begin{array}{cl} \underset{u\in\mathbb{R}^n}{\arg\min} & \frac{1}{2}\|u - x\|_2^2 \\ \text{subject to} & \|u\|_p \leq 1, \end{array} \tag{51}$$

where we now have an inequality constrained convex optimization problem (for $p \geq 1$). Here we take the gradient $\mathrm{D}y_p^\circ$ to be zero if $\|x\|_p < 1$ and $\mathrm{D}y_p$ otherwise. In words, we set the gradient to zero if the input already lies inside the unit ball. Otherwise we use the gradient obtained from projecting onto the L_p-sphere.

6 Implementation Considerations

In this section we provide some practical considerations relating to the implementation of deep declarative nodes.

6.1 Vector–Jacobian Product

As discussed above, the key challenge for declarative nodes is in computing $\mathrm{D}y(x)$ for which we have provided linear algebraic expressions in terms of first- and second-order derivatives of the objective and constraint functions. In computing the gradient of the loss function, J, for the end-to-end model we compute

$$\mathrm{D}J(x) = \mathrm{D}J(y)\mathrm{D}y(x), \tag{52}$$

and the order in which we evaluate this expression can have a dramatic effect on computational efficiency.

For simplicity consider the unconstrained case (Proposition 4.4) and let $v^{\mathsf{T}} = \mathrm{D}J(y) \in \mathbb{R}^{1\times m}$. We have

$$\mathrm{D}J(x) = -v^{\mathsf{T}} H^{-1} B, \tag{53}$$

where $H \in \mathbb{R}^{m\times m}$ and $B \in \mathbb{R}^{m\times n}$. The expression can be evaluated in two distinct ways—either as $(v^{\mathsf{T}} H^{-1})B$ or as $v^{\mathsf{T}}(H^{-1}B)$ where parentheses have been used to highlight

TABLE 3
The Effect of Robust Pooling Layers on Point Cloud Classification Results for the ModelNet40 Dataset [55], With Varying Outlier Percentages (O) and the Same Rate of Outliers Seen During Training and Testing

O	Top-1 Accuracy %						Mean Average Precision ×100					
%	[46]	Q	PH	H	W	TQ	[46]	Q	PH	H	W	TQ
0	**88.4**	84.7	84.7	86.3	86.1	85.4	**95.6**	93.8	95.0	95.4	95.0	93.8
10	79.4	84.3	85.6	85.5	**86.6**	85.5	89.4	94.3	94.6	**95.1**	94.6	94.7
20	76.2	84.8	84.8	85.2	**86.3**	85.5	87.8	94.8	95.0	**95.0**	94.8	95.0
50	72.0	84.0	83.1	83.9	**84.3**	83.9	83.3	93.8	93.5	94.3	94.8	**94.8**
90	29.7	61.7	63.4	63.1	**65.3**	61.8	38.9	76.8	78.7	78.5	**79.1**	76.6

Outliers points are uniformly sampled from the unit ball. PointNet [46] is compared to our variants that replace max pooling with robust pooling: quadratic (Q), pseudo-Huber (PH), Huber (H), Welsch (W), and truncated quadratic (TQ), all trained from scratch. Top-1 accuracy and mean average precision are reported.

calculation order. Assuming that H^{-1} has been factored (and contains no special structure), the cost of evaluating $\mathrm{D}\,J(x) \in \mathbb{R}^{1\times n}$ is $O(m^2 + mn)$ for the former and $O(m^2 n)$ for the latter, and thus there is a computational advantage to evaluating Equation (53) from left to right.

Moreover, it is often the case that $n \gg m$ in deep learning models, such as for robust pooling. Here, rather than precomputing B, which would require $O(mn)$ bytes of precious GPU memory to store, it is much more space efficient to compute the elements of $\mathrm{D}\,J(x)$ iteratively as

$$\mathrm{D}\,J(x)_i = \tilde{v}^{\mathsf{T}} b_i, \tag{54}$$

where $\tilde{v} = -H^{-1}v$ is cached and b_i is the ith column of B computed on the fly and therefore only requiring only $O(m)$ bytes.

6.2 Automatic Differentiation

The forward processing function for some deep declarative nodes (such as those defined by convex optimization problems) can be implemented using generic solvers. Indeed, Agrawal *et al.* [1] provide a general framework for specifying (convex) deep declarative nodes through disciplined convex optimization and providing methods for both the forward and backward pass. However, many other interesting nodes will require specialized solvers for their forward functions (e.g., the coordinate-based SDP solver proposed by Wang *et al.* [54]). Even so, the gradient calculation in the backward pass can be implemented using generic automatic differentiation techniques whenever the objective and constraint functions are twice continuously differentiable (in the neighborhood of the solution). For example, the following Python code makes use of the `autograd` package to compute the gradient of an arbitrary unconstrained deep declarative node with twice differentiable, first-order objective `f` at minimum `y` given input `x`.

Python Code

```python
import autograd.numpy as np
from autograd import grad, jacobian

def gradient(f, x, y):
  fY = grad(f, 1)
  fYY = jacobian(fY, 1)
  fXY = jacobian(fY, 0)

  return -1.0 * np.linalg.solve(fYY(x,y), fXY(x,y))
```

If called repeatedly (such as during learning) the partial derivative functions `fY`, `fYY` and `fXY` can be pre-processed and cached. And of course the gradient can instead be manually coded for special cases and when the programmer chooses to introduce memory and speed efficiencies. The implementation for equality and inequality constrained problems follows in a straightforward way and is omitted here for brevity. Naturally, the code above is (executable but) illustrative and does not include various optimizations for efficient GPU computation and exploiting problem structure (such as the vector–Jacobian product or the block diagonal nature of mini-batches). We provide full Python and PyTorch reference implementations for generic problems and examples in the software repository accompanying this paper (see http://deepdeclarativenetworks.com). This library facilitates rapid prototyping since hand-coding of problem-specific gradients is not required, and is sufficiently optimized for real computer vision applications (e.g., [12]).

7 Experiments

We conduct experiments on standard image and point cloud classification tasks to assess the viability of applying declarative networks to challenging computer vision problems. Our goal is not to obtain state-of-the-art results. Rather we aim to validate the theory presented above and demonstrate how declarative nodes can easily be integrated into non-trivial deep learning pipelines. To this end, we implement the pooling and projection operations from Section 5. For efficiency, we use the symbolic derivatives obtained in Tables 1 and 2, and never compute the Jacobian directly, instead computing the vector–Jacobian product to reduce the memory overhead. All code is available at http://deepdeclarativenetworks.com.

For the point cloud classification experiments with robust pooling, we use the ModelNet40 CAD dataset [55], with 2048 points sampled per object, normalized into a unit ball [46]. Each model is trained using stochastic gradient descent for 60 epochs with an initial learning rate of 0.01, decaying by a factor of 2 every 20 epochs, momentum (0.9), and a batch size of 24. All models, variations of PointNet [46] implemented in PyTorch, have 0.8M parameters. Importantly, pooling layers do not add additional parameters.

The results are shown in Tables 3 and 4, and Fig. 5, for a varying fraction of outliers seen during training and testing (Table 3) or only during testing, that is, trained without outliers (Table 4). We report top-1 accuracy and mean Average Precision (mAP) on the test set. Outlier points are sampled uniformly from the unit ball, randomly replacing existing inlier points, following the same protocol as Qi *et al.* [46].

TABLE 4
The Effect of Robust Pooling Layers on Point Cloud Classification Results for the ModelNet40 Dataset [55], With Varying Outlier Percentages (O) and No Outliers Seen During Training

O	Top-1 Accuracy %						Mean Average Precision ×100					
%	[46]	Q	PH	H	W	TQ	[46]	Q	PH	H	W	TQ
0	**88.4**	84.7	84.7	86.3	86.1	85.4	**95.6**	93.8	95.0	95.4	95.0	93.8
1	32.6	84.9	84.7	**86.4**	86.2	85.3	48.6	93.8	95.1	**95.3**	95.1	93.0
10	6.47	83.9	84.6	85.3	**86.0**	85.9	8.20	93.4	94.8	94.4	**94.9**	93.9
20	5.95	79.6	82.8	81.1	84.7	**84.9**	7.73	91.9	93.4	92.7	94.2	**94.6**
30	5.55	70.9	74.2	72.2	77.6	**83.2**	6.00	87.8	89.5	85.1	90.9	**92.8**
40	5.35	55.3	59.1	55.4	63.1	**75.6**	6.41	77.6	80.2	72.7	83.2	**90.6**
50	4.86	32.9	36.0	34.6	44.1	**57.9**	5.68	62.3	60.2	60.1	66.4	**85.3**
60	4.42	14.5	16.2	18.1	27.1	**30.6**	5.08	39.1	36.3	38.5	42.7	**68.5**
70	4.25	5.03	6.33	7.95	**14.1**	11.9	4.66	22.5	19.3	18.4	25.7	**47.9**
80	3.11	4.10	4.51	5.64	**8.88**	5.11	4.21	10.8	8.91	8.98	14.9	**26.7**
90	3.72	4.06	4.06	4.30	**5.68**	4.22	4.49	8.20	5.98	5.80	8.37	**9.78**

During testing, outlier points are uniformly sampled from the unit ball. Models are identical to those in Table 3. Top-1 accuracy and mean average precision are reported.

The robust pooling layer replaces the max pooling layer in the model, with the quadratic penalty function being denoted by Q, pseudo-Huber by PH, Huber by H, Welsch by W, and truncated quadratic by TQ. The optimizer for the last two (non-convex) functions is initialized to both the mean and the median, and then the lowest local minimum selected as the solution. It is very likely that RANSAC [29] search would produce better results, but this was not tested. For all experiments, the robustness parameter α (loosely, the inlier threshold) is set to one.

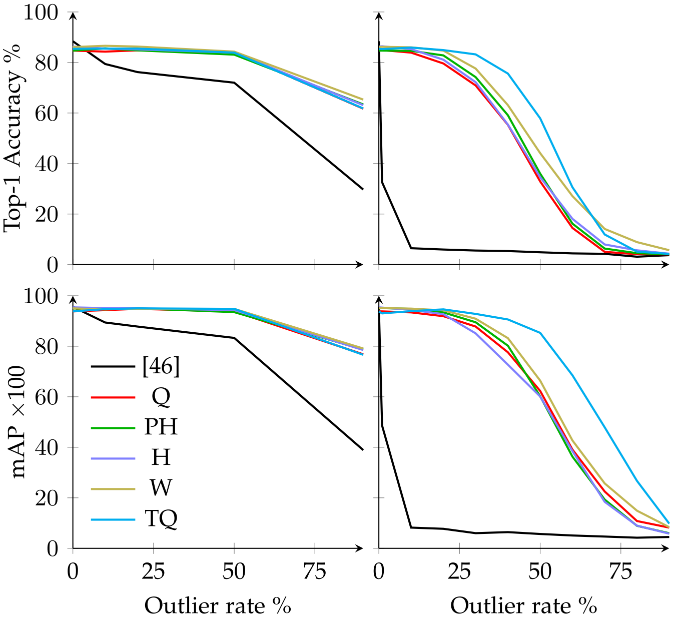


Fig. 5. The effect of robust pooling layers on point cloud classification results for the ModelNet40 dataset [55], with varying outlier rates (O). **Left:** the same rate of outliers is seen during training and testing, corresponding to results in Table 3. **Right:** no outliers are seen during training, corresponding to results in Table 4. Outliers points are uniformly sampled from the unit ball. PointNet [46] is compared to our variants that replace max pooling with robust pooling: quadratic (Q), pseudo-Huber (PH), Huber (H), Welsch (W), and truncated quadratic (TQ), all trained from scratch. Top-1 accuracy (**top**) and mean average precision (**bottom**) are reported.

We observe that the robust pooling layers perform significantly better than max pooling and quadratic (average) pooling when outliers are present, especially when not trained with the same outlier rate. There is a clear trend that, as the outlier rate increases, the most accurate model tends to be more robust model (further towards the right). This can be easily seen in Fig. 5 where we have plotted accuracy against outlier rate.

For the image classification experiments with L_p-sphere and L_p-ball projection, we use the ImageNet 2012 dataset [19], with the standard single central 224×224 crop protocol. Each model is trained using stochastic gradient descent for 90 epochs with an initial learning rate of 0.1, decaying by a factor of 10 every 30 epochs, weight decay (1e-4), momentum (0.9), and a batch size of 256. All models, variations of ResNet-18 [33] implemented in PyTorch, have 11.7M parameters. As with the robust pooling layers, projection layers do not add additional parameters.

The results are shown in Table 6, with top-1 accuracy, top-5 accuracy, and mean Average Precision (mAP) reported on the validation set. The projection layer, prepended by a batch normalization layer with no learnable parameters, is inserted before the final fully-connected output layer of the network. The features are pre-scaled according to the formula $\frac{2}{3}\text{median}_i \|f_i\|_p$ where f_i are the batch-normalized training set features of the penultimate layer of the pre-trained ResNet model. This corresponds to scaling factors of 250, 15, and 3 for $p = 1$, 2, and ∞ respectively, and can be thought of as varying the radius of the L_p-ball. The chosen scale ensures that the projection affects most features, but is not too aggressive.

The results indicate that feature projection improves the mAP significantly, with a more modest increase in top-1 and top-5 accuracy. This suggests that projection encourages a more appropriate level of confidence about the predictions, improving the calibration of the model.

To quantify the additional computation required when including declarative nodes within a network we report running time for training the different variants of our models

TABLE 5
Training Runtime (in ms / Point Cloud) of the Point Cloud Classification Network, Divided into Data Loading, Forward Pass, and Backward Pass, for the ModelNet40 Dataset [55], With Varying Outlier Fractions (O) and the Same Rate of Outliers Seen During Training and Testing

	Outlier fraction: 0.0						Outlier fraction: 0.9					
	[46]	Q	PH	H	W	TQ	[46]	Q	PH	H	W	TQ
Data	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
Fwd	**0.18**	0.80	3.43	5.14	19.2	50.0	**0.18**	0.80	2.75	3.85	10.5	32.6
Bwd	**0.27**	0.31	0.57	0.62	2.00	0.75	**0.27**	0.33	0.56	0.65	0.75	1.11

PointNet [46] is compared to our variants that replace max pooling with robust pooling: quadratic (Q), pseudo-Huber (PH), Huber (H), Welsch (W), and truncated quadratic (TQ), all trained from scratch.

averaged over the number of training instances (Tables 5 and 6 for robust pooling on point clouds and euclidean projection on image features, respectively). We have separated time for (i) data loading, (ii) evaluating the model in the forward pass, which requires solving an optimization problem for each training instance, and (iii) computing gradients and updating parameters in the backward pass.

Training of the ResNet-18 model for image classification is dominated by the time to load data with only a small overhead of 1–5μs per image needed to solve the projection problem in the forward pass. This reflects the efficiency of the euclidean projection algorithms. Surprisingly, the backward pass is slowest for the model without projection. We note that the gradient calculations are extremely simple (once the optimization problem has been solved and intermediate calculations cached), and attribute the variation between different models to timing tolerance and subtle changes in GPU resource utilization on different computation graphs.

Robust pooling for point cloud classification (Table 5) tells a different story. Here the data loading time is small and the effect of solving non-convex optimization problems in the forward pass can be clearly seen. Notice that the backward pass does not exhibit the same trend highlighting, as with projection, that the gradient calculations are efficient once the optimization problem is solved in the forward pass.

TABLE 6
The Effect of Projection Layers on Image Classification Results for the ImageNet 2012 Dataset [19]

Model	Top-1	Top-5	mAP	Data	Fwd	Bwd
	Acc. %	Acc. %	×100	μs / image		
ResNet-18	69.80	89.26	58.97	1358	**9**	17
ResNet-18-pt	69.76	89.08	53.76	–	–	–
ResNet-18-L_1Sphere	69.92	89.38	62.72	1362	12	15
ResNet-18-L_2Sphere	**70.66**	89.60	71.97	1358	14	**13**
ResNet-18-L_∞Sphere	70.03	89.22	63.98	1362	10	**13**
ResNet-18-L_1Ball	70.17	89.47	61.26	1362	11	15
ResNet-18-L_2Ball	70.59	**89.70**	**72.43**	1363	12	14
ResNet-18-L_∞Ball	70.06	89.29	63.33	1359	13	14

Top-1 and top-5 accuracy, mean Average Precision (mAP), and runtime (divided into data loading, forward pass, and backward pass) are reported. All models are trained from scratch, with the exception of ResNet-18-pt, which uses the PyTorch pre-trained weights.

8 Conclusion

In the preceding sections we have presented some theory and practice for deep declarative networks. On the one hand we developed nothing new—argmin can simply be viewed as yet another function and all that is needed is to work out the technical details of how to compute its derivative. On the other hand deep declarative networks offers a new way of thinking about network design with expressive processing functions and constraints but without having to worry about implementation details (or having to back-propagate gradients through complicated algorithmic procedures).

By facilitating the inclusion of declarative nodes into end-to-end learnable models, network capacity can be directed towards identifying features and patterns in the data rather than having to re-learn an approximation for theory that is already well-established (for example, physical system models) or enforce constraints that we know must hold (for example, that the output must lie on a manifold). As such, with deep declarative networks, we have the potential to create more robust models that can generalize better from less training data (and with fewer parameters) or, indeed, approach problems that could not be tackled by deep learning previously as demonstrated in recent works such as Amos and Kolter [4] and Wang *et al.* [54].

As with any new approach there are some shortcomings and much still to do. For example, it is possible for problems with parametrized constraints to become infeasible during learning, so care should be taken to avoid parameterizations that may result in an empty feasible set. More generally, optimization problems can be expensive to solve and those with multiple solutions can create difficulties for end-to-end learning. Also more theory needs to be developed around non-smooth objective functions. We have presented some techniques for dealing with problems with inequality constraints (when we may only have a single-sided gradient) or when the KKT optimality conditions cannot be guaranteed but there are many other avenues to explore. For example, recent work by Berthet *et al.* [8] that looks at averaging over perturbed optimization problems. Along these lines it would also be interesting to consider the trade-off in finding general descent directions rather than exact gradients and efficient approximations to the forward processing function, especially in the early stages of learning.

The work of Chen *et al.* [14] that shows how to differentiate through an ordinary differential equation also presents interesting directions for future work. Viewing such models as declarative nodes with differential constraints suggests many extensions including coupling ordinary differential equations with constrained optimization problems, which may be useful for physical simulations or modeling dynamical systems.

Finally, the fact that deep declarative nodes provide some guarantees on their output suggests that a principled approach to analyzing the end-to-end behavior of a deep declarative network might be possible. This would be especially useful in deploying deep learned models in safety critical applications such as autonomous driving or complex control systems. And while there are many challenges yet to overcome, the hope is that deep declarative networks will deliver solutions to problems faced by existing deep neural networks leading to better robustness and interpretability.

Acknowledgments

The authors would like to thanks to Bob Williamson and John Lloyd for helpful discussions, and anonymous reviewers for insightful suggestions. They also like to thank Itzik Ben-Shabat for recommending experiments on point cloud classification. This work was supported in part by the Australian Research Council Centre of Excellence in Computer Vision (CE140100016).

References

[1] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter, "Differentiable convex optimization layers," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019

[2] A. Agrawal, S. Barratt, S. Boyd, E. Busseti, and W. M. Moursi, "Differentiating through a cone program," *J. Appl. Numer. Optim.*, vol. 1, no. 2, pp. 107–115, Apr. 2019.

[3] B. Amos, "Differentiable optimization-based modeling for machine learning," PhD thesis, Comput. Sci. Dept., Carnegie Mellon Univ., Pittsburgh, PA, 2019.

[4] B. Amos and J. Z. Kolter, "OptNet: Differentiable optimization as a layer in neural networks," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 136–145.

[5] B. Amos, I. D. J. Rodriguez, J. Sacks, B. Boots, and J. Z. Kolter, "Differentiable MPC for end-to-end planning and control," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 8299–8310.

[6] S. Bai, Z. Kolter, and V. Koltun, "Deep equilibrium models," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 5238–5250.

[7] J. F. Bard, *Practical Bilevel Optimization: Algorithms and Applications*. Norwell, MA, USA: Kluwer Academic Press, 1998.

[8] Q. Berthet, M. Blondel, O. Teboul, M. Cuturi, J.-P. Vert, and F. Bach, "Learning with differentiable perturbed optimizers," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 9508–9519.

[9] D. P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*. Cambridge, MA, USA: Academic Press, 1982.

[10] D. P. Bertsekas, *Nonlinear Programming*. Belmont, MA, USA: Athena Scientific, 2004.

[11] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[12] D. Campbell, L. Liu, and S. Gould, "Solving the blind perspective-n-point problem end-to-end with robust differentiable geometric optimization," in *Proc. Eur. Conf. Comput. Vis.*, 2020, pp. 244–261.

[13] B. Chen, A. Parra, J. Cao, N. Li, and T.-J. Chin, "End-to-end learnable geometric vision by backpropagating PnP optimization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 8097–8106.

[14] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 6572–6583.

[15] A. Cherian, B. Fernando, M. Harandi, and S. Gould, "Generalized rank pooling for action recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 1581–1590.

[16] F. H. Clarke, "Generalized gradients and applications," *Trans. Amer. Math. Soc.*, vol. 205, pp. 247–262, 1975.

[17] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter, "End-to-end differentiable physics for learning and control," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*. 2018, pp. 7178–7189.

[18] S. Dempe and S. Franke, "On the solution of convex bilevel optimization problems," *Comput. Optim. Appl.*, vol. 63, pp. 685–703, 2016.

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[20] J. E. Dennis Jr and R. E. Welsch, "Techniques for nonlinear least squares and robust regression," *Commun. Statist. - Simul. Comput.*, vol. 7, no. 4, pp. 345–359, 1978.

[21] J. Djolonga and A. Krause, "Differentiable learning of submodular models," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1014–1024.

[22] C. B. Do, C.-S. Foo, and A. Y. Ng, "Efficient multiple hyperparameter learning for log-linear models," in *Proc. 20th Int. Conf. Neural Inf. Process. Syst.*, 2007, pp. 377–384.

[23] J. Domke, "Generic methods for optimization-based modeling," in *Proc. 15th Int. Conf. Artif. Intell. Statist.*, 2012, pp. 318–326.

[24] A. L. Dontchev and R. T. Rockafellar, *Implicit Functions and Solution Mappings: A View from Variational Analysis*, 2nd ed. Berlin, Germany: Springer-Verlag, 2014.

[25] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra, "Efficient projections onto the $l1$-ball for learning in high dimensions," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 272–279.

[26] B. Fernando and S. Gould, "Learning end-to-end video classification with rank-pooling," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1187–1196.

[27] B. Fernando and S. Gould, "Discriminatively learned hierarchical rank pooling networks," *Int. J. Comput. Vis.*, vol. 124, pp. 335–355, Sep. 2017.

[28] B. Fernando, E. Gavves, J. M. Oramas, A. Ghodrati, and T. Tuytelaars, "Modeling video evolution for action recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 5378–5387.

[29] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[30] G. H. Golub and C. F. van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD, USA: Johns Hopkins Univ. Press, 1996.

[31] S. Gould, B. Fernando, A. Cherian, P. Anderson, R. Santa Cruz, and E. Guo, "On differentiating parameterized argmin and argmax problems with application to bi-level optimization," Australian National University, Canberra, Australia, Jul. 2016, *arXiv:1607.05447*.

[32] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[34] S. Jiang, D. Campbell, M. Liu, S. Gould, and R. Hartley, "Joint unsupervised learning of optical flow and egomotion with bi-level optimization," in *Proc. Int. Conf. 3D Vis.*, 2020, pp. 682–691.

[35] M. J. Johnson, D. K. Duvenaud, A. B. Wiltschko, R. P. Adams, and S. R. Datta, "Composing graphical models with neural networks for structured representations and fast inference," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 2954–2962.

[36] T. Klatzer and T. Pock, "Continuous hyper-parameter learning for support vector machines," in *Proc. Comput. Vis. Winter Workshop*, 2015, pp. 39–47.

[37] S. G. Krantz and H. R. Parks, *The Implicit Function Theorem: History, Theory, and Applications*. Berlin, Germany: Springer, 2013.

[38] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[39] K. Lee, S. Maji, A. Ravichandran, and S. Soatto, "Meta-learning with differentiable convex optimization," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 10649–10657.

[40] P. Márquez-Neila, M. Salzmann, and P. Fua, "Imposing hard constraints on deep networks: Promises and limitations," in *Proc. CVPR Workshop Negative Results Comput. Vis.*, 2017.

[41] C. D. Meyer, "Generalized inversion of modified matrices," *SIAM J. Appl. Math.*, vol. 24, no. 3, pp. 315–323, 1973.

[42] P. Ochs, R. Ranftl, T. Brox, and T. Pock, "Bilevel optimization with nonsmooth lower level problems," in *Proc. Int. Conf. Scale Space Variational Methods Comput. Vis.*, 2015, pp. 654–665.

[43] S. Oymak, "Learning compact neural networks with regularization," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018, pp. 3963–3972.

[44] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. NeurIPS Autodiff Workshop*, 2017.

[45] M. B. Paulus, D. Choi, D. Tarlow, A. Krause, and C. J. Maddison, "Gradient estimation with stochastic softmax tricks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2020, pp. 5691–5704.

[46] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep learning on point sets for 3D classification and segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 77–85.

[47] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[48] K. G. G. Samuel and M. F. Tappen, "Learning optimized MAP estimates in continuously-valued MRF models," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 477–484.

[49] R. Santa Cruz, B. Fernando, A. Cherian, and S. Gould, "Visual permutation learning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 41, no. 12, pp. 3100–3114, Dec. 2019.

[50] G. M. Scarpello and D. Ritelli, "Historical outline of the theorem of implicit functions," *Divulgaciones Matematica*, vol. 10, pp. 171–180, 2002.

[51] S. Tschiatschek, A. Sahin, and A. Krause, "Differentiable submodular maximization," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 2731–2738.

[52] M. Vlastelica, A. Paulus, V. Musil, G. Martius, and M. Rolínek, "Differentiation of BlackBox combinatorial solvers," in *Proc. Int. Conf. Learn. Representations*, 2020.

[53] H. von Stackelberg, D. Bazin, L. Urch, and R. R. Hill, *Market Structure and Equilibrium*. Berlin, Germany: Springer, 2011.

[54] P.-W. Wang, P. L. Donti, B. Wilder, and Z. Kolter, "SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver," in *Proc. 36th Int. Conf. Mach. Learn.*, 2019, pp. 6545–6554.

[55] Z. Wu *et al.*, "3D ShapeNets: A deep representation for volumetric shapes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1912–1920.

Stephen Gould (Member, IEEE) received the BSc degree in mathematics and computer science and BE degree in electrical engineering from the University of Sydney, Australia, in 1994 and 1996, respectively, and the MS degree in electrical engineering from Stanford University, Stanford, California, in 1998. After working in industry for several years he returned to PhD studies and received the PhD degree from Stanford University, Stanford, California, in 2010. He is currently a professor with the Research School of Computer Science, Australian National University (ANU), Australia. He is a former ARC postdoctoral fellow, microsoft faculty fellow and principal research scientist at Amazon Inc. He is currently a director of the Australian Centre for Robotic Vision. He has broad interests in the areas of computer and robotic vision, machine learning, deep learning, structured prediction, and optimization.

Richard Hartley (Fellow, IEEE) is professor and has been a member of the Computer Vision Group, Research School of Engineering, Australian National University, Acton, Australia, since January 2001. He is a member of the Computer Vision Research Group, Data61, a government funded research laboratory. He was with the General Electric Research and Development Center from 1985 to 2001, where he was involved in VLSI design and later in computer vision. He became involved with Image Understanding and Scene Reconstruction working with GEs Simulation and Control Systems Division. He is the author of the book Multiple View Geometry in Computer Vision with Andrew Zisserman.

Dylan Campbell (Member, IEEE) received the BE degree in mechatronic engineering (Hons) from the University of New South Wales, Australia, and the PhD degree in engineering from Australian National University, Australia, in 2018 while concurrently working as a research assistant in the Cyber-Physical Systems Group, Data61–CSIRO. He is a research fellow with the Research School of Computer Science, Australian National University (ANU), Australia, and the Australian Research Council Centre of Excellence in Robotic Vision. His research interests cover a range of computer vision topics and their underlying geometry and optimization problems.