Exploiting Problem Structure in Deep Declarative Networks: Two Case Studies

Stephen Gould[1], **Dylan Campbell**[2], **Itzik Ben-Shabat**[1,3], **Chamin Hewa Koneputugodage**[1], **Zhiwei Xu**[1]

[1] Australian National University, Canberra, Australia
[2] University of Oxford, Oxford, United Kingdom
[3] Technion, Haifa, Israel

Abstract

Deep declarative networks and other recent related works have shown how to differentiate the solution map of a (continuous) parametrized optimization problem, opening up the possibility of embedding mathematical optimization problems into end-to-end learnable models. These differentiability results can lead to significant memory savings by providing an expression for computing the derivative without needing to unroll the steps of the forward-pass optimization procedure during the backward pass. However, the results typically require inverting a large Hessian matrix, which is computationally expensive when implemented naively. In this work we study two applications of deep declarative networks—robust vector pooling and optimal transport—and show how problem structure can be exploited to obtain very efficient backward pass computations in terms of both time and memory. Our ideas can be used as a guide for improving the computational performance of other novel deep declarative nodes.

Introduction

Deep declarative networks, also known as differentiable optimization or implicit layers (Gould, Hartley, and Campbell 2021; Agrawal et al. 2019; Amos and Kolter 2017), are deep learning models that support propagating (exact) gradients backwards through the solution of a continuous optimization problem. This is achieved by applying the implicit function theorem to the optimality conditions of the problem at a given solution. The advantage of this approach is that intermediate results produced by the (typically iterative) optimization algorithm need not be cached for use in the backward pass. Indeed, non-differentiable steps can be applied during the forward pass and details of the optimization algorithm do not even need to be known for calculating the gradient in the backward pass.

Specifically, an expression for the Jacobian $\mathrm{D}y(x)$ of the output y with respect to the input x can be formulated knowing only the optimality conditions for the problem at hand and the current solution. Moreover, given a software implementation of the objective and constraints (or the optimality condition directly) the gradient can be computed without additional coding by automatic differentiation (Paszke et al. 2017; Blondel et al. 2021). However, notwithstanding the significant savings in development time, automatic differentiation can in some situations lead to suboptimal computations, and implemented poorly the result may be even slower and more memory intensive than unrolling and back-propagating through the forward pass optimization loop.

The core operation performed by a deep learning node or layer during the backward pass is to calculate the gradient of the loss function (or global objective) $\mathrm{D}J(x)$ with respect to the node's inputs (or parameters) given the gradient of the loss function with respect to its outputs $\mathrm{D}J(y)$. The calculation is an instance of the chain rule for differentiation:

$$\mathrm{D}J(x) = \mathrm{D}J(y) \cdot \mathrm{D}y(x), \tag{1}$$

where J is the loss function and $\mathrm{D}y(x)$ is the gradient of the output with respect to the input. In PyTorch this is the role of the `backward` method of `autograd.Function` (Paszke et al. 2017) that then allows gradients to back-propagate through the entire network.

Gould, Hartley, and Campbell (2021) consider deep declarative nodes defined by second-order differentiable, equality constrained, optimization problems parametrized by an n-dimensional input x of the form

$$y(x) \in \begin{array}{ll} \arg\min_{u \in \mathbb{R}^m} & f(x,u) \\ \text{subject to} & h_i(x,u) = 0,\ i = 1,\dots,p \end{array} \tag{2}$$

and give an expression for $\mathrm{D}y(x)$ as

$$H^{-1}A^{\mathsf{T}}\left(AH^{-1}A^{\mathsf{T}}\right)^{-1}\left(AH^{-1}B - C\right) - H^{-1}B, \tag{3}$$

where A, B, C and H are objects (matrices or tensors) of first- and second-order (mixed) partial derivatives of the objective and constraint functions with respect to $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. Specifically,

$$A = \mathrm{D}_Y h(x,y) \in \mathbb{R}^{p \times m}$$

$$B = \mathrm{D}^2_{XY} f(x,y) - \sum_{i=1}^{p} \lambda_i \mathrm{D}^2_{XY} h_i(x,y) \in \mathbb{R}^{m \times n}$$

$$C = \mathrm{D}_X h(x,y) \in \mathbb{R}^{p \times n}$$

$$H = \mathrm{D}^2_{YY} f(x,y) - \sum_{i=1}^{p} \lambda_i \mathrm{D}^2_{YY} h_i(x,y) \in \mathbb{R}^{m \times m}$$

and $\lambda \in \mathbb{R}^p$ satisfies $\lambda^{\mathsf{T}} A = \mathrm{D}_Y f(x,y)$. Here, the notation comes from Gould, Hartley, and Campbell (2021) with

D_Z denoting partial derivatives with respect to variables Z. Naive implementation of Eqn. 3 requires $O(\max\{m^3, p^3\})$ operations due to the matrix inversions.

In general, the loss function J is scalar-valued and summed over each training example in a mini-batch. As such gradients of J with respect to each node's inputs and outputs decompose over elements of the mini-batch, and Equations 1 and 3 can be evaluated independently (and in parallel) for each training example of the mini-batch. Let b be the size of the mini-batch, n be the size of the input and m be the size of the output. Then storage for $\mathrm{D}J(y)$, $\mathrm{D}y(x)$, and $\mathrm{D}J(x)$ requires $O(bm)$, $O(bnm)$ and $O(bn)$ bytes, respectively. However, for many optimization problems we do not need to construct $\mathrm{D}y(x)$ explicitly and can instead exploit its structure to save both computation and memory.

Deep declarative networks provide a powerful and flexible tool that has been applied to a growing number of applications including video classification (Fernando et al. 2016), visual Sudoku (Amos and Kolter 2017; Wang et al. 2019), blind PnP (Campbell, Liu, and Gould 2020; Chen et al. 2020) and meta-learning (Lee et al. 2019). The contribution of this paper is to provide case studies that demonstrate general principles for implementing efficient backward pass computation in deep declarative nodes so as not to be a bottleneck. Based on the case studies and our experience, we conclude with tips and advice for implementing new declarative nodes.

Background and Related Work

Automatic differentiation is the backbone of modern deep learning software frameworks such as PyTorch (Paszke et al. 2017). It allows rapid experimentation with different network architectures and implementation of new differentiable processing nodes, where the forward pass can be explicitly implemented as a sequence of steps, themselves differentiable expressions. Deep declarative networks (Gould, Hartley, and Campbell 2021) introduced a new form of processing node as the solution to an optimization problem, where the algorithm for implementing the forward pass is not explicitly defined, but where back-propagation through the node is still possible.

Early examples of such declarative nodes in deep networks (Amos and Kolter 2017; Gould et al. 2016; Fernando et al. 2016) relied on hand-coded implementations of the backward pass. Later works show that automatic differentiation techniques can also be applied in the case of deep declarative nodes by differentiating the optimality conditions for the problem at hand (Agrawal et al. 2019; Diamond and Boyd 2016; Gould, Hartley, and Campbell 2021; Blondel et al. 2021), dramatically simplifying the implementation of these nodes. However, this automatic approach is less able to exploit structure that may exist in the problem, and as a result is suboptimal. Thus, it is sometimes desirable to revert to carefully crafted manual implementations.

Early work that exploits problem structure includes Fernando and Gould (2016) for the case of differentiable rank pooling, where the Sherman–Morrison formula (Horn and Johnson 1991) was used to efficiently compute the inverse of a Hessian matrix required during the backward pass. The same work and others suggest applying approximations to simplify the backward pass, e.g., taking the diagonal of the Hessian (Fernando and Gould 2016), ignoring constraints, or heavily regularising to reduce the number of iterations in the forward pass (Asano, Rupprecht, and Vedaldi 2020). We provide further examples showing general patterns for exploiting structure and opportunities for approximation.

Case Studies

We present two case studies of deep declarative nodes—one unconstrained and one constrained. The case studies follow a generic recipe for implementing deep declarative nodes: (i) Write out the mathematical expressions for the objective and constraints; (ii) Derive the relevant partial derivatives needed in Eqn. 3; (iii) Inspect the components for structure and consider how to implement them efficiently; (iv) Code and test the forward and backward passes. Experiments profiling memory and running time are included for each example, and full PyTorch source code is available.[1]

Robust Vector Pooling

Consider the problem of computing a robust estimate for the mean of a set of m-dimensional points $\mathcal{X} = \{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\}$. That is, we assume that our data $\mathcal{X}$ is noisy and wish to find the point $y \in \mathbb{R}^m$ that best approximates the mean of the noise-free data. If we knew the noise model then this amounts to solving a maximum-likelihood problem. For example, under an isotropic Gaussian noise model (or no noise) the best approximation is the *sample mean*, $y = \frac{1}{n}\sum_{i=1}^{n} x_i$. In other situations, we may want to reduce the effect of outliers, and do so by finding a point y that minimizes the sum of costs for the distance to each point x_i,

$$y \in \arg\min_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \phi(\|u - x_i\|_2; \alpha), \tag{4}$$

where $\phi : \mathbb{R} \to \mathbb{R}_+$ is a penalty function parametrized by α. For the one-dimensional case ($m = 1$) this is an instance of the *penalty function approximation problem* (Boyd and Vandenberghe 2004). When using a quadratic penalty function, $z \mapsto \frac{1}{2}z^2$, the solution is the sample mean. However, this is not robust to outliers and many other penalty functions have been proposed (e.g., see Tab. 1).[2]

The objective function for the robust vector pooling optimization problem (Eqn. 4) is

$$f(\mathcal{X}, u) = \sum_{i=1}^{n} \phi(\|u - x_i\|_2; \alpha) = \sum_{i=1}^{n} \phi(z_i; \alpha), \tag{5}$$

[1] All results are reported using PyTorch 1.8.1 with robust vector pooling running on NVIDIA GeForce RTX 2080 GPU and optimal transport on NVIDIA GeForce RTX 3090.

[2] Note that Gould, Hartley, and Campbell (2021) consider the one-dimensional case, applying the penalty function to $u - x_i$, which is computationally more straightforward since H and B are scalars. Here we generalize to the vector case and apply the penalty function to $\|u - x_i\|_2$, which requires more care in implementing operations on m-by-m matrices.

where we have written $z_i = \|u - x_i\|_2$. Since the problem is unconstrained, the gradient of the minimizer y with respect to each of the x_j reduces to (Gould, Hartley, and Campbell 2021, Proposition 4.4)

$$\mathrm{D}_{X_j} y = -H^{-1} B, \tag{6}$$

where $H = \mathrm{D}^2_{YY} f$ and $B = \mathrm{D}^2_{X_j Y} f$. Since f decomposes as a sum of penalty functions ϕ, it suffices to just consider $\mathrm{D}^2_{YY}\phi$ and $\mathrm{D}^2_{XY}\phi$. Let us start by computing $\mathrm{D}_Y \phi$ for the i-th data point,

$$\mathrm{D}_Y \phi(z_i; \alpha) = \phi'(z_i; \alpha)\mathrm{D}_Y z_i = \frac{\phi'(z_i;\alpha)}{z_i}(y - x_i)^\mathsf{T}, \tag{7}$$

where ϕ' is the first derivative of ϕ. Computing second derivatives, we have

$$\begin{aligned}\mathrm{D}^2_{YY}\phi(z_i;\alpha) = &\frac{\phi'(z_i;\alpha)}{z_i} I_{m\times m} + \\ &\left(\frac{\phi''(z_i;\alpha)}{z_i^2} - \frac{\phi'(z_i;\alpha)}{z_i^3}\right)(y - x_i)(y - x_i)^\mathsf{T} \\ = &\kappa_1(z_i) I_{m\times m} + \kappa_2(z_i)(y - x_i)(y - x_i)^\mathsf{T},\end{aligned} \tag{8}$$

where κ_1 and κ_2 are quantities that depend on the penalty function and z_i (see Tab. 1). By anti-symmetry of x_i and y in Eqn. 7, we have $\mathrm{D}^2_{X_j Y}\phi(z_j;\alpha) = -\mathrm{D}^2_{YY}\phi(z_j;\alpha)$. We can therefore write the following expression for $\mathrm{D}_{X_j} y$,

$$\underbrace{\left(\sum_{i=1}^n \kappa_1(z_i) I + \kappa_2(z_i)(y - x_i)(y - x_i)^\mathsf{T}\right)^{-1}}_{H^{-1}} \underbrace{\left(\kappa_1(z_j) I + \kappa_2(z_j)(y - x_j)(y - x_j)^\mathsf{T}\right)}_{-B}. \tag{9}$$

A naive implementation of this expression would be prohibitively expensive since B is an m-by-m matrix that must to be computed separately for each point $x_j \in \mathcal{X}$ (or stored if computed in batch during the construction of H requiring $O(nm^2)$ memory). It is preferable to compute $\mathrm{D}_{X_j} y$ for all j at the same time, i.e., in batch, to make use of GPU parallelization, which further exacerbates the memory problem. A better approach is to evaluate the entire expression for the gradient of the loss function (Eqn. 1) from left-to-right.

Let $v^\mathsf{T} = \mathrm{D}J(y)$ be the derivative of the loss function with respect to the output, i.e., the incoming backward gradient. Our goal is to compute $\mathrm{D}J(x_i)$ for $i = 1, \ldots, n$. We have, $\mathrm{D}J(x_i) = v^\mathsf{T} H^{-1} B$. Letting $w^\mathsf{T} = v^\mathsf{T} H^{-1}$ be obtained by solving $v = Hw$ using Cholesky factorization and back substitution. Note that this can be computed once for all points in the input as it is independent of which x_i we are taking the derivative with respect to. We then have

$$\mathrm{D}J(x_i) = \kappa_1(z_i) w^\mathsf{T} + \kappa_2(z_i) w^\mathsf{T}(y - x_i)(y - x_i)^\mathsf{T}. \tag{10}$$

Taking the inner product $w^\mathsf{T}(y - x_i)$ first, instead of the outer product $(y - x_i)(y - x_i)^\mathsf{T}$, results in significant memory and computational savings, requiring only $O(nm)$ bytes of storage when processed in batch versus $O(nm^2)$. Note also that some penalty functions have $\kappa_2 \equiv 0$, e.g., quadratic, thus avoiding this computation entirely (see Tab. 1).

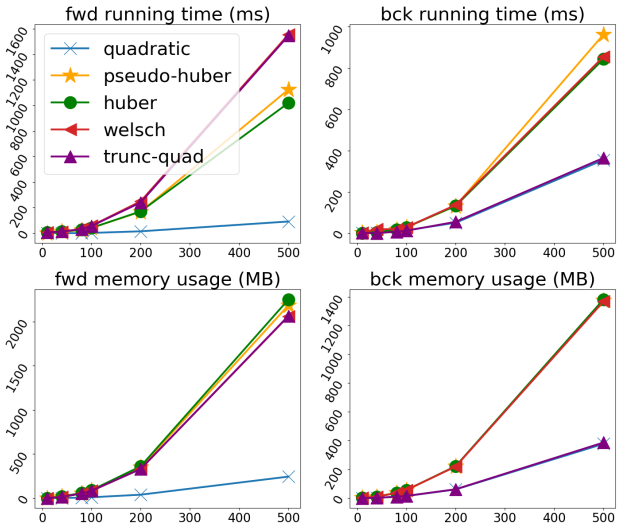


Figure 1: Time (top) and memory (bottom) requirements for forward and backward passes of robust vector pooling on the CPU. The 2D feature map ($\sqrt{n} \times \sqrt{n}$) has $m = 128$ channels, and the batch size one. We use L-BFGS in the forward pass except for quadratic, which has a closed-form solution. For non-convex penalties we take the best solution from two different initializations. The backward pass is implemented by implicit differentiation following the expression in Eqn. 10 (code in Fig. 2).

Fig. 2 shows PyTorch source code for the backward pass. The code handles both the case of $\kappa_2 = 0$ (Lines 9–10) and the case of $\kappa_2 \neq 0$ (Lines 12–20), and follows a batch implementation of the expression above. Profiling of forward and backward passes for different size problems and different penalty functions is shown in Fig. 1. Observe that memory for the forward and backward passes is comparable.

Optimal Transport

Optimal transport is a very popular algorithm in machine learning for measuring the distance between two probability distributions. It can also be used to find matches between sets of objects (e.g., in solving the blind PnP problem (Campbell, Liu, and Gould 2020)). The entropy regularized optimal transport problem can be written as the linearly constrained mathematical program,

$$\begin{array}{ll}\text{minimize (over } P \in \mathbb{R}_+^{m\times n}) & \langle P, M\rangle + \frac{1}{\gamma}\mathrm{KL}(P\|rc^\mathsf{T}) \\ \text{subject to} & P1 = r \\ & P^\mathsf{T}1 = c,\end{array} \tag{11}$$

where $M \in \mathbb{R}^{m\times n}$ is the input cost matrix, r and c are positive vectors of row and column sums (with $1^\mathsf{T} r = 1^\mathsf{T} c = 1$), and $\gamma > 0$ controls the strength of the regularization term.

	$\phi(z;\alpha)$	$\kappa_1(z;\alpha)$	$\kappa_2(z;\alpha)$
QUADRATIC	$\frac{1}{2}z^2$	1	0
PSEUDO-HUBER	$\alpha^2\left(\sqrt{1+\left(\frac{z}{\alpha}\right)^2}-1\right)$	$\left(1+\left(\frac{z}{\alpha}\right)^2\right)^{-1/2}$	$-\frac{1}{\alpha^2}\left(1+\left(\frac{z}{\alpha}\right)^2\right)^{-3/2}$
HUBER	$\begin{cases}\frac{1}{2}z^2 & \text{for } \lvert z\rvert \le \alpha \\ \alpha(\lvert z\rvert - \frac{1}{2}\alpha) & \text{otherwise}\end{cases}$	$\begin{cases}1 & \text{for } \lvert z\rvert \le \alpha \\ \alpha/\lvert z\rvert & \text{otherwise}\end{cases}$	$\begin{cases}0 & \text{for } \lvert z\rvert \le \alpha \\ -\alpha/\lvert z\rvert^3 & \text{otherwise}\end{cases}$
WELSCH	$1-\exp\left(-\frac{z^2}{2\alpha^2}\right)$	$\frac{1}{\alpha^2}\exp\left(-\frac{z^2}{2\alpha^2}\right)$	$-\frac{1}{\alpha^4}\exp\left(-\frac{z^2}{2\alpha^2}\right)$
TRUNC. QUAD.	$\begin{cases}\frac{1}{2}z^2 & \text{for } \lvert z\rvert \le \alpha \\ \frac{1}{2}\alpha^2 & \text{otherwise}\end{cases}$	$\begin{cases}1 & \text{for } \lvert z\rvert \le \alpha \\ 0 & \text{otherwise}\end{cases}$	0

Table 1: Parameters κ_1 and κ_2 for various robust penalty functions ϕ where $\kappa_1(z) = \phi'(z)/z$ and $\kappa_2(z) = (\phi''(z) - \kappa_1(z))/z^2$. In the case of robust vector pooling the argument z is non-negative and the absolute value calculations can be omitted.

```
def backward(ctx, v):
    x, y = ctx.saved_tensors
    b, m = x.shape[:2]

    y_minus_x = y.view(b,m,1) - x.view(b,m,-1)
    z = linalg.norm(y_minus_x, dim=1, keepdim=True) + 1.0e-9

    k1, k2 = ctx.penalty.kappa(z, ctx.alpha)
    if all(k2 == 0.0):
        return (k1 * (y_grad / k1.sum(dim=2)).view(b,m,1)).reshape(x.shape)

    H = k1.sum(dim=2).view(b,1,1) * eye(m).view(1,m,m) + \
        einsum("bik,bjk->bij", y_minus_x, k2 * y_minus_x)

    L = cholesky(H)
    w = cholesky_solve(v.view(b,m,-1), L).view(b,m)

    u = einsum("bi,bik->bk", w, k2 * y_minus_x)

    return (k1 * w.view(b,m,1) + einsum("bk,bik->bik", u, y_minus_x)).reshape(x.shape)
```

Figure 2: Implementation of the backward pass for robust vector pooling. It is assumed that the b-by-m-by-n input and b-by-m output are cached in the forward pass. Global pooling is done over m-dimensional features for each of the b batches independently. The function `ctx.penalty.kappa` computes κ_1 and κ_2 for the given penalty function (see Tab. 1). Full source code available at http://deepdeclarativenetworks.com.

What makes this formulation attractive from a computational perspective is that it can be solved very efficiently by the Sinkhorn algorithm, an iterative algorithm that performs successive row and column normalizations (Cuturi 2013).

In computing derivatives we arrive at similar expressions to Luise et al. (2018), who present an algorithm for differentiating with respect to r and c, but where we directly use the results for deep declarative nodes (Eqn. 3). To find $\mathrm{D}_M P$, the Jacobian of P with respect to M, let $f(M,P) = \sum_{ij} M_{ij}P_{ij} + \frac{1}{\gamma}\sum_{ij} P_{ij}\left(\log P_{ij} - \log r_i c_j\right)$ be the entropy regularized optimal transport objective. We can then write the following partial derivatives,

$$\frac{\partial f}{\partial P_{ij}} = M_{ij} + \frac{1}{\gamma}\log P_{ij} + \frac{1}{\gamma} - \frac{1}{\gamma}\log r_i c_j, \tag{12}$$

$$H_{ij,kl} = \frac{\partial^2 f}{\partial P_{ij}\partial P_{kl}} = \begin{cases}\frac{1}{\gamma P_{ij}} & \text{if } (i,j) = (k,l) \\ 0 & \text{otherwise,}\end{cases} \tag{13}$$

$$B_{ij,kl} = \frac{\partial^2 f}{\partial P_{ij}\partial M_{kl}} = \begin{cases}1 & \text{if } (i,j) = (k,l) \\ 0 & \text{otherwise,}\end{cases} \tag{14}$$

for $i = 1,\dots,m$ and $j = 1,\dots,n$. Flattening the input M and output P into vectors rowwise we have $H^{-1} = \mathbf{diag}(\gamma P_{ij})$ and $B = I_{mn\times mn}$. That these are diagonal makes sense, since if not for the linear equality constraints each P_{ij} would only depend on M_{ij} and not any other M_{kl} for $kl \neq ij$. Moreover, since B is the identity matrix we can ignore it from any calculations.

The primary challenge now is computing the term $(AH^{-1}A^{\mathsf{T}})^{-1}$ in Eqn. 3. Here the matrix A of partial derivatives of the constraint functions with respect to the output is formed as the coefficients of the P_{ij} in the constraint functions of Problem 11,

$$h : \left\{ \begin{array}{ll} \sum_{j=1}^{n} P_{ij} - r_i & \text{for } i = 1,\dots,m \\ \sum_{i=1}^{m} P_{ij} - c_j & \text{for } j = 1,\dots,n \end{array} \right\} = 0. \tag{15}$$

Note that the set contains a redundant constraint; if any $m+n-1$ constraints are satisfied then the remaining constraint will also be satisfied. To apply Eqn. 3 we must remove one constraint otherwise A will not be full rank (Gould, Hartley, and Campbell 2021, Corollary 4.9). Removing the

first constraint and where P_{ij} has again been flattened row-wise, we have

$$A = \begin{bmatrix} 0_n^\mathsf{T} & 1_n^\mathsf{T} & \cdots & 0_n^\mathsf{T} \\ \vdots & \vdots & \ddots & \vdots \\ 0_n^\mathsf{T} & 0_n^\mathsf{T} & \cdots & 1_n^\mathsf{T} \\ I_{n\times n} & I_{n\times n} & \cdots & I_{n\times n} \end{bmatrix} \in \mathbb{R}^{(m+n-1)\times mn}. \tag{16}$$

It is straightforward to show that

$$AH^{-1}A^\mathsf{T} = \begin{bmatrix} \mathbf{diag}\left(\sum_{j=1}^n H^{-1}_{pj,pj} \mid p = 2,\ldots,m\right) & (H^{-1}_{ij,ij})_{i=2,\ldots,m\times j=1,\ldots,n} \\ (H^{-1}_{ij,ij})_{j=1,\ldots,n\times i=2,\ldots,m} & \mathbf{diag}\left(\sum_{i=1}^m H^{-1}_{ip,ip} \mid p = 1,\ldots,n\right) \end{bmatrix} \tag{17}$$

$$= \gamma \begin{bmatrix} \mathbf{diag}(r_{2:m}) & P_{2:m,1:n} \\ P^\mathsf{T}_{2:m,1:n} & \mathbf{diag}(c) \end{bmatrix} \tag{18}$$

by considering the (p,q)-th entry of $AH^{-1}A^\mathsf{T}$ for $p,q \in 1,\ldots,m+n-1$ as,

$$(AH^{-1}A^\mathsf{T})_{pq} = \sum_{i=1}^m \sum_{j=1}^n \frac{A_{p,ij}A_{q,ij}}{H_{ij,ij}} \tag{19}$$

and substituting γP_{ij} for $H^{-1}_{ij,ij}$, and $r_{2:m}$ and c for their corresponding sums.

Now we can directly compute $(AH^{-1}A^\mathsf{T})^{-1}$ in $O((m+n-1)^3)$ time or make use of more efficient block matrix inversion (Horn and Johnson 1991) results to compute in $O((m-1)^3)$ time,[3]

$$\begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^\mathsf{T} & \Lambda_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{diag}(r_{2:m}) & P_{2:m,1:n} \\ P^\mathsf{T}_{2:m,1:n} & \mathbf{diag}(c) \end{bmatrix}^{-1}, \tag{20}$$

where each block is calculated as

$$\Lambda_{11} = \left(\mathbf{diag}(r_{2:m}) - P_{2:m,1:n}\mathbf{diag}(c)^{-1}P^\mathsf{T}_{2:m,1:n}\right)^{-1} \tag{21}$$

$$\Lambda_{12} = -\Lambda_{11}P_{2:m,1:n}\mathbf{diag}(c)^{-1} \tag{22}$$

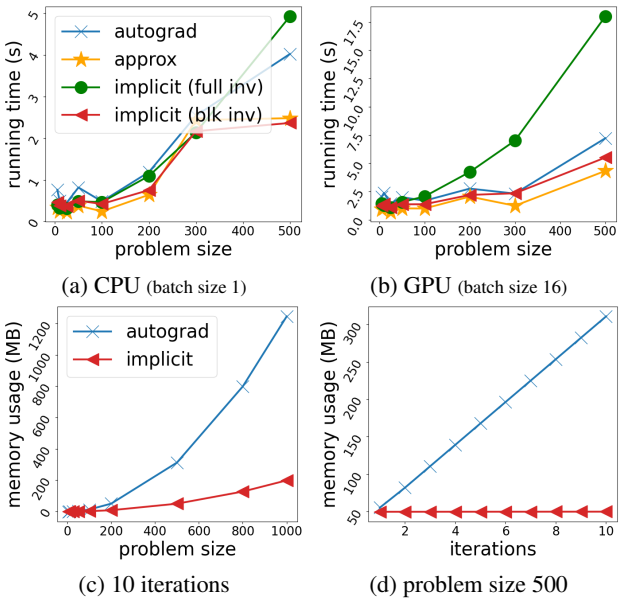
$$\Lambda_{22} = \mathbf{diag}(c)^{-1}\left(I - P^\mathsf{T}_{2:m,1:n}\Lambda_{12}\right) \tag{23}$$

and we use Cholesky factorization to multiply by Λ_{11} rather than inverting explicitly.

A PyTorch implementation for the gradient is shown in Fig. 4. Here we evaluate the expression for the gradient from left-to-right and replace explicit multiplication by A with corresponding summations of terms in the multiplicand (see Line 9). Rather than flattening P we keep it in tensor form. Line 6 initializes the calculation of $\mathrm{D}_M J$ with $-v^\mathsf{T}H^{-1}B$. This can be seen as an approximation to the gradient with constraints ignored and is close to the true gradient when only a small number of Sinkhorn iterations is needed in the forward pass.

Profiling this approximation is included in our experiments, where we also compare block inverse of $AH^{-1}A^\mathsf{T}$ versus the full inverse (see Fig. 3). Important to observe is that unrolling Sinkhorn (autograd) and the implicit differentiation approach with block inverse have approximately the same running time (Fig. 3(a) and (b)) whereas the latter is much more memory efficient, improving over unrolling Sinkhorn beyond four iterations for problems of size 500-by-500 (Fig. 3(d)).

[3]Or in $O(n^3)$ time if $n < m$ using an alternative formula for the block inverse.



(a) CPU (batch size 1) (b) GPU (batch size 16)

(c) 10 iterations (d) problem size 500

Figure 3: Time and memory comparison for optimal transport. Our block-inverse implicit differentiation is much faster than full-inverse version and uses less memory than autograd.

We can similarly back propagate through r and c (c omitted for brevity). Here we note that

$$\frac{\partial f}{\partial r_i} = -\frac{1}{\gamma r_i}\sum_{j=1}^n P_{ij} = -\frac{1}{\gamma} \implies \frac{\partial^2 f}{\partial r_i \partial P_{kl}} = 0. \tag{24}$$

As such $B = 0$ and the expression in Eqn. 3 reduces to

$$-H^{-1}A^\mathsf{T}\left(AH^{-1}A^\mathsf{T}\right)^{-1}C, \tag{25}$$

where, by inspection of the constraint function in Eqn. 15,

$$C = \left[\frac{\partial h_p}{\partial r_i} \mid p = 1,\ldots,m+n-1,\ i = 1,\ldots,m\right] \tag{26}$$

$$= -\begin{bmatrix} I_{m-1\times m} \\ 0_{n\times m} \end{bmatrix} \in \mathbb{R}^{(m+n-1)\times m}. \tag{27}$$

The calculation of $v^\mathsf{T}H^{-1}A^\mathsf{T}(AH^{-1}A^\mathsf{T})^{-1}$ can be reused for the gradients associated with M, r and c as done in Lines 22 and 23 of the code. Note that taking a step in the (negative) gradient direction may destroy normalization of r (or c) required by the optimal transport problem. One way to ensure normalization is preserved is to define r in terms of another positive vector $\tilde{r}$ as $r = \tilde{r}/1^\mathsf{T}\tilde{r}$. The backward going gradient would then need to be post-multiplied by $\mathrm{D}r(\tilde{r}) = (I_{n\times n} - r1^\mathsf{T})/1^\mathsf{T}\tilde{r}$, omitted in Fig. 4 for simplicity of exposition.

```
def backward(ctx, dJdP):
    M, r, c, P = ctx.saved_tensors
    b, m = M.shape[:2]

    # initialize backward gradients (-v^T H^{-1} B with v = dJdP and B = I)
    dJdM = -1.0 * ctx.gamma * P * dJdP

    # compute [vHAt1, vHAt2] = v^T H^{-1} A^T as two blocks
    vHAt1, vHAt2 = dJdM[:, 1:m].sum(dim=2), dJdM.sum(dim=1)

    # compute [v1,v2] = -v^T H^{-1} A^T (A H^{-1} A^T)^{-1} by block inverse
    PdivC = P[:, 1:m] / c.view(b, 1, -1)
    block_11 = cholesky(diag_embed(r[:, 1:m]) - einsum("bij,bkj->bik", P[:, 1:m], PdivC))
    block_12 = cholesky_solve(PdivC, block_11)
    block_22 = diag_embed(1/c) + einsum("bji,bjk->bik", block_12, PdivC)

    v1 = cholesky_solve(vHAt1.view(b,m-1,1), block_11).view(b,m-1) - \
        einsum("bi,bji->bj", vHAt2, block_12)
    v2 = einsum("bi,bij->bj", vHAt2, block_22) - einsum("bi,bij->bj", vHAt1, block_12)

    # compute v^T H^{-1} A^T (A H^{-1} A^T)^{-1} A H^{-1} B - v^T H^{-1} B
    dJdM[:, 1:m] -= v1.view(b, m-1, 1) * P[:, 1:m]
    dJdM -= v2.view(b, 1, -1) * P

    # compute -v^T H^{-1} A^T (A H^{-1} A^T)^{-1} C for r and c
    dJdr = -1.0 / ctx.gamma * cat((zeros(b, 1), v1), dim=1)
    dJdc = -1.0 / ctx.gamma * v2

    return dJdM, dJdr, dJdc
```

Figure 4: Implementation of the backward pass for optimal transport. Assumes that the inputs M, r and c, and output P are cached by the forward pass. Input M and output P consist of b batches of m-by-n matrices. Full source code available at http://deepdeclarativenetworks.com.

Discussion

In this paper we studied two examples of deep declarative nodes and showed how to implement an efficient backward pass by exploiting problem structure. This results in better utilization of memory and compute than can be achieved from automatic differentiation (autodiff) or unrolling the forward pass optimization loop. However, for other problems unrolling or autodiff may be satisfactory for a given task despite being computationally more expensive. We now summarize several key practical implementation considerations for developing new deep declarative nodes if compute is an issue, using our case studies as a guide.

It is judicious to first implement and experiment with the declarative node using a generic automatic differentiation approach. Several open-source tools make this easy (Gould, Hartley, and Campbell 2021; Agrawal et al. 2019; Blondel et al. 2021). Moreover, having such an implementation allows for rapid testing of new ideas and will facilitate debugging of future specialized code in addition to the use of numerical gradient checking (e.g., `autograd.gradcheck`).

Next, inspect the required derivatives for structure and use this to simplify the computation. For example, efficient algorithms exist for inverting certain Hessian matrices (diagonal or block, triangular, etc.), and multiplication by 0-1 matrices can be replaced with summations. Importantly, when the objective of the problem decomposes elementwise over the optimization variables, such as in optimal transport, then the Hessian matrix will be diagonal. Related to this is thinking about the order of operations in Eqn. 3, which can dramatically affect the memory required for storing intermediate results. The vector-Jacobian product used for computing the loss in the backward pass is a good example of this, as is the left-to-right evaluation of the outer products required for robust vector pooling, which is common when norms appear in objective or constraint functions.

Other standard considerations include saving calculations in the forward pass (if tractable to do so); disabling autodiff in the forward pass, which avoids unnecessary construction of the computation graph; performing inline operations to reuse memory buffers; and batch operations for better parallelism. Numerical stability can also be an issue, especially when the (locally) optimal solution is not isolated or the Hessian is almost singular. Here, linear system solvers (e.g., Cholesky) should be used instead of inverting matrices and trust-region approaches (or regularization of the Hessian) can be used to improve stability (Toso, Campbell, and Russell 2019; Gould, Hartley, and Campbell 2021).

Finally, reparametrizing the problem can give different computational trade-offs, e.g., removing constraints to make a problem unconstrained or adding variables (and associated constraints) so that the Hessian is structured. Alternatively, taking a hybrid approach where structure is exploited for some terms and autodiff used for the rest. This is particularly attractive when the optimality conditions can be written as the composition of many functions (as was done for example in Campbell, Liu, and Gould (2020)). Moreover, it presents an exciting future research direction to see whether some of these techniques can be applied automatically.

References

Agrawal, A.; Amos, B.; Barratt, S.; Boyd, S. P.; Diamond, S.; and Kolter, Z. 2019. Differentiable Convex Optimization Layers. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Amos, B.; and Kolter, Z. 2017. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *Proc. of the International Conference on Machine Learning (ICML)*.

Asano, Y. M.; Rupprecht, C.; and Vedaldi, A. 2020. Self-labelling via simultaneous clustering and representation learning. In *Proc. of the International Conference on Learning Representations (ICLR)*.

Blondel, M.; Berthet, Q.; Cuturi, M.; Frostig, R.; Hoyer, S.; Llinares-Lopez, F.; Pedregosa, F.; and Vert, J.-P. 2021. Efficient and Modular Implicit Differentiation. Technical report, Google (arXiv:2105.15183).

Boyd, S. P.; and Vandenberghe, L. 2004. *Convex Optimization*. Cambridge.

Campbell, D.; Liu, L.; and Gould, S. 2020. Solving the Blind Perspective-n-Point Problem End-To-End with Robust Differentiable Geometric Optimization. In *Proc. of the European Conference on Computer Vision (ECCV)*.

Chen, B.; Parra, A.; Cao, J.; Li, N.; and Chin, T.-J. 2020. End-to-End Learnable Geometric Vision by Backpropagating PnP Optimization. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Cuturi, M. 2013. Sinkhorn Distances: Lightspeed Computation of Optimal Transport. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Diamond, S.; and Boyd, S. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83): 1–5.

Fernando, B.; Anderson, P.; Hutter, M.; and Gould, S. 2016. Discriminative Hierarchical Rank Pooling for Activity Recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Fernando, B.; and Gould, S. 2016. Learning End-to-end Video Classification with Rank-Pooling. In *Proc. of the International Conference on Machine Learning (ICML)*.

Gould, S.; Fernando, B.; Cherian, A.; Anderson, P.; Santa Cruz, R.; and Guo, E. 2016. On Differentiating Parameterized Argmin and Argmax Problems with Application to Bi-level Optimization. Technical report, Australian National University (arXiv:1607.05447).

Gould, S.; Hartley, R.; and Campbell, D. 2021. Deep Declarative Networks. *IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI)*.

Horn, R. A.; and Johnson, C. R. 1991. *Topics in Matrix Analysis*. Cambridge University Press.

Lee, K.; Maji, S.; Ravichandran, A.; and Soatto, S. 2019. Meta-Learning with Differentiable Convex Optimization. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Luise, G.; Rudi, A.; Pontil, M.; and Ciliberto, C. 2018. Differential Properties of Sinkhorn Approximation for Learning with Wasserstein Distance. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic Differentiation in PyTorch. In *NeurIPS Autodiff Workshop*.

Toso, M.; Campbell, N.; and Russell, C. 2019. Fixing Implicit Derivatives: Trust-Region Based Learning of Continuous Energy Functions. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Wang, P.-W.; Donti, P. L.; Wilder, B.; and Kolter, Z. 2019. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *Proc. of the International Conference on Machine Learning (ICML)*.