The Hessian of tall-skinny networks is easy to invert

Ali Rahimi

January 14, 2026

Abstract

We describe an exact algorithm to solve linear systems of the form $Hx = b$ where H is the Hessian of a deep net. The method computes Hessian-inverse-vector products without storing the Hessian or its inverse. It requires time and storage that scale linearly in the number of layers. This is in contrast to the naive approach of first computing the Hessian, then solving the linear system, which takes storage and time that are respectively quadratic and cubic in the number of layers. The Hessian-inverse-vector product method scales roughly like Pearlmutter's algorithm for computing Hessian-vector products.

1 Introduction

The Hessian of a deep net is the matrix of second-order mixed partial derivatives of its loss with respect to its parameters. Decades ago, when deep nets had only hundreds or thousands of parameters, the Hessian matrix could be inverted to implement optimizers that converged much faster than gradient descent [11, 1]. But for large modern deep nets, relying on the Hessian has become impractical: The Hessian of a model with a billion parameters would have a quintillion entries, which is far larger than can be stored, multiplied, or inverted even in the largest data centers. A common workaround is to approximate the Hessian as a low-rank matrix [12, 6] or as a diagonal matrix [2, 3, 5]. Such approximations make it easier to apply the inverse of the Hessian to a vector. This article shows how to compute and apply the inverse of the Hessian exactly without storing the Hessian or its inverse. The Hessian-inverse-vector product can be computed in time and storage that scale linearly with the number of layers in the model, and cubically in the number of parameters and activations in each layer.

Pearlmutter [9] showed how to compute the product of the Hessian with a fixed vector (the so-called Hessian-vector product) in time and storage that scale linearly with the number of layers in the network. This is much faster than the cubic scaling of the naive algorithm that first computes the Hessian matrix and then multiplies it by the vector. His method transforms the original network into a new network whose gradient is the desired Hessian-vector product. To compute the Hessian-vector product, one then just applies backpropagation to the new network. Given a way to compute the Hessian-vector product, one can indirectly compute the Hessian-inverse-vector product via, say Krylov iterations like Conjugate Gradient as proposed by Pearlmutter and more recently re-investigated [8, 10]. However, the quality of the result would then depend on the conditioning of the Hessian, which is notoriously poor for deep nets [4]. Unfortunately, there seems to exist no variant of Pearlmutter's trick to compute the Hessian-inverse-vector products directly.

The proposed Hessian-inverse-vector product algorithm takes advantage of a deep net's layerwise structure. Regardless of the specific operations in each layer, the Hessian is a second order matrix polynomial that involves the first order and second order mixed derivatives of each layer, and the inverse of a block bi-diagonal operator that represents the backpropagation algorithm. Multiplying a vector by this structure computes the Hessian-vector product using exactly the same operations as Pearlmutter's algorithm (see Appendix B). It also leads to a way to compute Hessian-inverse-vector product that does not require forming or storing the full Hessian. To formally characterize the storage and running time of the algorithm, assume an oracle that offers the first order and second order mixed derivatives of each layer. For an L-layer deep net where each layer has at most p parameters and generates at most a activations, naively storing the Hessian would require $O(L^2p^2)$ memory, and solving a linear system would require $O\left(L^3p^3\right)$ operations in addition to the oracle queries. In contrast, we will show how to perform these operations using only $O(L\max(a,p)^2)$

storage and $O(L \max(a,p)^3)$ computation in addition to the oracle queries. The dependence on the number of activations and parameters in each layer remains cubic, but the dependence on the number of layers is only linear. This makes operating on the Hessian of tall and skinny networks more efficient than the Hessian of short and fat networks.

Although modern networks are typically short and wide, our method runs faster than the naive Hessian-inverse-vector product algorithm on tall and skinny networks, since it ameliorates the naive algorithm's cubic dependence on depth to a mild linear dependence on depth. A follow-up article will explore training tall-skinny networks using the Hessian-inverse-vector product as a preconditioner. We expect that a significant training speedup will motivate a return to deeper network designs.

2 Overview

Our objective is to efficiently solve linear systems of the form $Hx = b$, where H is the Hessian of a deep neural network, without forming H explicitly. To do this, we employ the following strategy:

1. Write down the gradient of the deep net in matrix form, as a block bi-diagonal system of linear equations. Solving this system uses back-substitution, which in this case coincides exactly with the computations carried out by backpropagation.

2. Differentiate this matrix form to obtain an expression for the Hessian. This expression involves a second-order polynomial in the inverse of the aforementioned block bi-diagonal matrix.

3. Refactor the expression wit hthe help of auxiliarry variables. These lift the polynomial into a higher-dimensional linear form. After pivoting, this linear form becomes block tri-diagonal.

4. Factorize the resulting block-tri-diagonal into a lower block bi-diagonal matrix, a block diagonal matrix, and an upper block diagonal matrix using the LDU factorization.

5. Solve the resulting system using forward and backward substitution. This is similar to running back-propagation on a modified version of the original network that's designed to compute the Hessian-inverse-vector product.

6. Un-pivot the result, and report the un-lifted solution.

Algorithm 2 summarizes the steps above.

Algorithm 1 Compute the hessian-inverse-vector product by solving $(H + \epsilon I)x = b$.

Require: A vector b, damping constant ϵ.
Ensure: $x = (H + \epsilon I)^{-1} b$
1: Define the sparse block matrix $\mathcal{K}$ and the augmented system as in Equation 13.
2: Pivot into permuted system $\mathcal{K}'x' = b'$ where $\mathcal{K}' = \Pi\mathcal{K}\Pi^\top$ is block-tri-diagonal following Equation (14).
3: Solve $\mathcal{K}'x' = b'$ using block-LDU decomposition.
4: Recover the solution x from the permuted vector $\Pi \begin{bmatrix} x \\ y \\ z \end{bmatrix}'$.

The algorithm is straightforward to implement with a suitable library of block matrix operations.[1]

[1] `https://github.com/a-rahimi/hessian`

3 Notation

We write a deep net as a pipeline of functions $\ell = 1, \ldots, L$,

$$\begin{aligned} z_1 &= f_1(z_0; x_1) \\ &\cdots \\ z_\ell &= f_\ell(z_{\ell-1}; x_\ell) \\ &\cdots \\ z_L &= f_L(z_{L-1}; x_L) \end{aligned} \tag{1}$$

The vectors $x_1, \ldots, x_L$ are the parameters of the pipeline. The vectors $z_1, \ldots, z_L$ are its intermediate activations, and z_0 is the input to the pipeline. The last layer f_L computes the final activations and their training loss, so the scalar z_L is the loss of the model on the input z_0. To make this loss's dependence on z_0 and the parameters explicit, we sometimes write it as $z_L(z_0; x)$. This formalization deviates slightly from the traditional deep net formalism in two ways: First, the training labels are subsumed in z_0, and are propagated through the layers until they're used in the loss. Second, the last layer fuses the loss (which has no parameters) and the last layer (which does).

We'll assume the first and partial derivatives of each layer with respect to its parameters and its inputs exist. This poses some complications with ReLU activations and other non-differentiable operations in modern networks. Notably, for the Hessian to be symmetric, it must be differentiable everywhere. We'll largely ignore this complication and assume that differentiable approximations to these operations are used.

At the end of each section, we'll count the number of floating point operations required to compute various expressions. While the derivations do not impose any restrictions on the shape of the layers, for the purpose of this accounting, we'll assume all but the last L layers have a-dimensional activations ($z_\ell \in \mathbb{R}^a$) and p-dimensional parameters ($x_\ell \in \mathbb{R}^p$).

4 Backpropagation, the matrix way

We would like to fit the vector of parameters $x = (x_1, \ldots, x_L)$ given a training dataset, which we represent by a stochastic input z_0 to the pipeline. Training the model proceeds by gradient descent steps along the stochastic gradient $\partial z_L(z_0; x)/\partial x$. The components of this direction can be computed by the chain rule with a backward recursion:

$$\frac{\partial z_L}{\partial x_\ell} = \underbrace{\frac{\partial z_L}{\partial z_\ell}}_{b_\ell} \underbrace{\frac{\partial z_\ell}{\partial x_\ell}}_{\nabla_x f_\ell} \tag{2}$$

$$\frac{\partial z_L}{\partial z_\ell} = \underbrace{\frac{\partial z_L}{\partial z_{\ell+1}}}_{b_{\ell+1}} \underbrace{\frac{\partial z_{\ell+1}}{\partial z_\ell}}_{\nabla_z f_{\ell+1}}. \tag{3}$$

The identification $b_\ell \equiv \frac{\partial z_L}{\partial z_\ell}$, $\nabla_x f_\ell \equiv \frac{\partial z_\ell}{\partial x_\ell}$, and $\nabla_z f_\ell \equiv \frac{\partial z_\ell}{\partial z_{\ell-1}}$ turns this recurrence into

$$\frac{\partial z_L}{\partial x_\ell} = b_\ell \cdot \nabla_x f_\ell \tag{4}$$

$$b_\ell = b_{\ell+1} \cdot \nabla_z f_{\ell+1}, \tag{5}$$

with the base case $b_L = 1$, a scalar. These two equations can be written in vector form as

$$\frac{\partial z_L}{\partial x} = \begin{bmatrix} \frac{\partial z_L}{\partial x_1} & \cdots & \frac{\partial z_L}{\partial x_L} \end{bmatrix} = \underbrace{\begin{bmatrix} b_1 & \cdots & b_L \end{bmatrix}}_{\equiv b} \underbrace{\begin{bmatrix} \nabla_x f_1 & & \\ & \ddots & \\ & & \nabla_x f_L \end{bmatrix}}_{\equiv D_x}, \tag{6}$$

and

$$\begin{bmatrix} b_1 & b_2 & b_3 & \cdots & b_{L-1} & b_L \end{bmatrix} \underbrace{\begin{bmatrix} I & & & & & \\ -\nabla_z f_2 & I & & & & \\ & -\nabla_z f_3 & I & & & \\ & & \ddots & \ddots & & \\ & & & & -\nabla_z f_L & 1 \end{bmatrix}}_{\equiv M} = \underbrace{\begin{bmatrix} 0 & \cdots & 1 \end{bmatrix}}_{\equiv e_L}. \tag{7}$$

Solving for b and substituting back gives

$$\frac{\partial z_L}{\partial x} = e_L M^{-1} D_x. \tag{8}$$

The matrix M is block bi-diagonal. Its diagonal entries are identity matrices, and its off-diagonal matrices are the gradient of the intermediate activations with respect to the layer's parameters. The matrix D_x is block diagonal, with the block as the derivative of each layer's activations with respect to its inputs. M is invertible because the spectrum of a triangular matrix can be read off its diagonal, which in this case is all ones.

5 The Hessian

To obtain the Hessian, we use similar techniques to compute the gradient of Equation (8) with respect to x. The gradient we computed in Equation (8) is the unique vector g such that $dz_L \equiv z_L(x + dx) - z_L(dx) \to g(x)^\top dx$ as $dx \to 0$. Similarly, the Hessian H of z_L with respect to the parameters is the unique matrix $H(x)$ such that $dg \equiv g(x + dx) - g(x) \to H(x)\, dx$ as $dx \to 0$. Appendix A shows that the Hessian has the following form:

Claim 1. *The Hessian of the loss z_L with respect to the vector of parameters x is given by*

$$H = D_D D_{xx} + D_D D_{zx} P M^{-1} D_x + D_x^\top M^{-\top} P^\top D_M D_{xz} + D_x^\top M^{-\top} P^\top D_M D_{zz} P M^{-1} D_x, \tag{9}$$

with the following matrices:

$$D_D \equiv \begin{bmatrix} \underbrace{I \otimes b_1}_{p \times ap} & & \\ & \ddots & \\ & & I \otimes b_L \end{bmatrix}, \qquad D_M \equiv \begin{bmatrix} \underbrace{I \otimes b_1}_{a \times a^2} & & \\ & \ddots & \\ & & I \otimes b_L \end{bmatrix},$$

$$P \equiv \begin{bmatrix} 0 & & \\ I & 0 & \\ & \ddots & \\ & I & 0 \end{bmatrix}, \qquad M \equiv \begin{bmatrix} I & & & & \\ \underbrace{-\nabla_z f_2}_{a \times a} & I & & & \\ & -\nabla_z f_3 & I & & \\ & & \ddots & \ddots & \\ & & & -\nabla_z f_L & 1 \end{bmatrix},$$

$$D_x \equiv \begin{bmatrix} \underbrace{\nabla_x f_1}_{a \times p} & & \\ & \ddots & \\ & & \nabla_x f_L \end{bmatrix}, \qquad D_{xx} \equiv \begin{bmatrix} \underbrace{\nabla_{xx} f_1}_{ap \times p} & & \\ & \ddots & \\ & & \nabla_{xx} f_L \end{bmatrix}, \qquad D_{xz} \equiv \begin{bmatrix} \underbrace{\nabla_{xz} f_1}_{a^2 \times p} & & \\ & \ddots & \\ & & \nabla_{xz} f_L \end{bmatrix},$$

$$D_{zx} \equiv \begin{bmatrix} \underbrace{\nabla_{zx} f_1}_{ap \times a} & & \\ & \ddots & \\ & & \nabla_{zx} f_L \end{bmatrix}, \qquad D_{zz} \equiv \begin{bmatrix} \underbrace{\nabla_{zz} f_1}_{a^2 \times a} & & \\ & \ddots & \\ & & \nabla_{zz} f_L \end{bmatrix}.$$

Given a vector $x \in \mathbb{R}^{Lp}$, the formula above allows us to compute Hx in $O\left(Lap^2 + La^2p + La^3\right)$ operations without forming H. This cost is dominated by multiplying by the D_{xx}, D_{zx}, and D_{zz} matrices. Appendix B shows that these operations are exactly the operations performed in Pearlmutter's trick to compute the Hessian-vector product.

To solve systems of the form $Hx = b$, one could use Krylov methods to repeatedly multiply by H without forming H (a possibility Pearlmutter considered [9]). This would require applying H some number of times that depends on the condition number of H. However, the next section shows how to solve systems of the form $H^{-1}x = b$ with only $\max(a, p)$ times more operations than are needed to compute Hx.

6 Applying the inverse of the Hessian

The above shows that the Hessian is a second order matrix polynomial in M^{-1}. While M itself is block-bidiagonal, M^{-1} is dense, so H is dense. Nevertheless, this polynomial can be lifted into a higher order object whose inverse is easy to compute:

$$H = D_D D_{xx} + D_D D_{zx} P M^{-1} D_x + D_x^\top M^{-\top} P^\top D_M D_{xz} + D_x^\top M^{-\top} P^\top D_M D_{zz} P M^{-1} D_x.$$

In general, H is singular. We wish to solve the system $(H + \epsilon I)x = g$ for x. We can convert this dense system involving inverses into a larger, sparse system by introducing auxiliary variables. Define

$$y \equiv M^{-1} D_x x, \tag{10}$$

which implies $My - D_x x = 0$. Define a second auxiliary variable

$$z \equiv M^{-\top}\left(P^\top D_M D_{xz} x + P^\top D_M D_{zz} P y\right), \tag{11}$$

which implies $M^\top z - P^\top D_M D_{xz} x - P^\top D_M D_{zz} P y = 0$. With these substitutions, $(H + \epsilon I)x = g$ becomes

$$\left(D_D D_{xx} + \epsilon I\right) x + D_D D_{zx} P y + D_x^\top z = g. \tag{12}$$

Collecting these three linear equations gives us a unified system

$$\underbrace{\begin{bmatrix} D_D D_{xx} + \epsilon I & D_D D_{zx} P & D_x^\top \\ -D_x & M & 0 \\ -P^\top D_M D_{xz} & -P^\top D_M D_{zz} P & M^\top \end{bmatrix}}_{\mathcal{K}} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} g \\ 0 \\ 0 \end{bmatrix}. \tag{13}$$

Finding $x = \left(H + \epsilon I\right)^{-1} g$ is equivalent to solving this block-linear system and reporting the resulting x. The benefit of doing this is that this system can be pivoted into a block-tri-diagonal system, which can be solved more efficiently than Gaussian elimination on $\mathcal{K}$ or H.

The pivoting we'll apply reorders the blocks of the variables x, y, z from $x_1, \ldots, x_L, y_1, \ldots, y_L, z_1, \ldots, z_L$ to $x_1, y_1, z_1, \ldots, x_L, y_L, z_L$. This pivoting operation acts as a kind of transpose operation on block matrices (its generalization is called a "commutation matrix" in [7]). When applied to $\mathcal{K}$, it turns it into a block-tri-diagonal matrix. To get a better feel for this operation, denote the ijth block of $\mathcal{K}$ by $\mathcal{K}_{ij}$ and the uvth sub-block of this block by $\mathcal{K}_{ij,uv}$. Similarly, x denote a vector conformant with $\mathcal{K}$, with x_j denoting the column vector that multiplies each block $\mathcal{K}_{\cdot j}$ and denote the vth sub-block of x_j by x_{jv}, which multiplies $\mathcal{K}_{\cdot j,\cdot v}$. We say that x is j-major because as we traverse the entries of x from top to bottom, the index j increments more slowly than the v index.

Denote by Π the permutation that transposes the blocks with the sub-blocks. Applying this permutation matrix Π to x reorders it from j-major to d-major. Applying it to the rows and columns of $\mathcal{K}$ results in a block matrix $\mathcal{K}' \equiv \Pi\mathcal{K}\Pi^\top$ that satisfies $\mathcal{K}_{ij,uv} = \mathcal{K}'_{uv,ij}$. Section C illustrates this operator with some examples.

Observation 1. *The permutation Π is involutory, meaning $\Pi^{-1} = \Pi$.*

This implies that to solve a system $\mathcal{K}x = b$, we can instead solve $\mathcal{K}'x' = \Pi b$ for x', then report $x = \Pi x'$.

This permutation reduces the bandwidth of certain block matrices:

Observation 2. *When the blocks of $\mathcal{K}$ are banded with bandwidth w, $\mathcal{K}'$ is block-banded with bandwidth w.*

The blocks in $\mathcal{K}$ are either block-diagonal, block-upper-bi-diagonal, or block-lower-bi-diagonal, so $\mathcal{K}'$ is at most block-tri-diagonal. Denote the ijth block of $\mathcal{K}'$ as B_{ij}:

$$B_{ij} \equiv \begin{bmatrix} K_{11,ij} & K_{12,ij} & K_{13,ij} \\ K_{21,ij} & K_{22,ij} & K_{23,ij} \\ K_{31,ij} & K_{32,ij} & K_{33,ij} \end{bmatrix}.$$

Then $\mathcal{K}'$ can be written as the block-tri-diagonal matrix

$$\mathcal{K}' = \begin{bmatrix} B_{11} & B_{12} & 0 & \cdots & & & 0 \\ B_{21} & B_{22} & B_{23} & \ddots & & & \vdots \\ 0 & B_{32} & B_{33} & B_{34} & \ddots & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & B_{L-1,L-2} & B_{L-1,L-1} & B_{L-1,L} \\ 0 & \cdots & 0 & 0 & 0 & B_{L,L-1} & B_{L,L} \end{bmatrix}. \tag{14}$$

That means we can solve the system $\mathcal{K}'x' = b'$ by first decomposing $\mathcal{K}'$ via block LDU decomposition into the product of a lower-block-bi-diagonal matrix, a block-diagonal matrix, and an upper-block-bi-diagonal matrix as $LDUx' = b'$, then solve for $Ly = b'$ using back substitution, and finally solve for $DUx' = y$ using forward substitution. Because these systems are block-bi-diagonal, solving them looks like forward and backward propagation on a chain. The running time for these solvers is linear in the depth of the network and cubic in the dimension of each block B_{ij}. Depending on the block, the dimension of each of these blocks is either in the number of parameters or the number of activations in the corresponding layer. In all,assuming a and p are the largest activation and parameter count for a layer, the running time for solving this system is $O\left(L \max(a,p)^3\right)$ operations.

7 Conclusion

We have described a method to compute the product of the inverse Hessian of a deep neural network with a vector. Compared to the naive method, which stores the Hessian and requires computation cubic in the depth of the network, this method does not store the Hessian and requires computation linear in the depth of the network.

In its final stage, the method relies on a forward and backward substitution to solve a block tridiagonal system. This step bears a similarity to Pearlmutter's method for computing the Hessian-vector product: It can be interpreted as running backpropagation on a modified version of the original network whose gradient is the desired Hessian-inverse-vector product. The downside of this similarity is that solving large linear systems with LDU factorization is prone to numerical instability. An improvement on the proposal is to solve the block tridiagonal system with an off-the-shelf banded solver.

Our hope is to use this technique as a preconditioner to speed up stochastic gradient descent. Since the method's speedup is greatest when the network is tall and skinny, we hope it might rekindle interest in extremely deep architectures.

References

[1] Etienne Barnard. Optimization for training neural nets. *IEEE Transactions on Neural Networks*, 3(2):232–240, March 1992.

[2] Suzanna Becker and Yann LeCun. Improving the convergence of back-propagation learning with second order methods. *Proceedings of the 1988 Connectionist Models Summer School*, pages 29–37, 1989.

[3] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[4] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An investigation into neural net optimization via hessian eigenvalue density. *CoRR*, abs/1901.10159, 2019.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Quoc V. Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 265–272, Bellevue, Washington, USA, 2011. Omnipress.

[7] Jan R Magnus and Heinz Neudecker. *Matrix Differential Calculus with Applications in Statistics and Econometrics*. John Wiley & Sons, 3rd edition, 2019.

[8] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 735–742. Omnipress, 2010.

[9] Barak A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6(1):147–160, 1994.

[10] Oriol Vinyals and Daniel Povey. Krylov subspace descent for deep learning. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, volume 22, pages 1261–1268. PMLR, 2012.

[11] Richard L. Watrous. Learning algorithms for connectionist networks: Applied gradient methods of nonlinear optimization. In *Proceedings of the First IEEE International Conference on Neural Networks*, volume 2, pages 619–627, San Diego, CA, June 1987. IEEE.

[12] Andrew R. Webb and David Lowe. A hybrid optimisation strategy for adaptive feed-forward layered networks. Technical Report RSRE Memorandum 4193, Royal Signals and Radar Establishment (RSRE), Malvern, UK, 1988.

A The Hessian

We use the facts that $dM^{-1} = -M^{-1}(dM)M^{-1}$ and $b = e_L M^{-1}$ to write

$$\begin{aligned} dg &= g(x+dx) - g(x) \qquad (15)\\ &= d(e_L M^{-1} D_x) \\ &= e_L M^{-1}(dD_x) + e_L \left(dM^{-1}\right) D_x \\ &= b \cdot dD_x - e_L M^{-1}(dM)M^{-1} D_x \\ &= b \cdot dD_x - b \cdot (dM) M^{-1} D_x \qquad (16)\end{aligned}$$

We compute each of these terms separately. As part of this agenda, we will rely on the gradient of tensor-valued functions $g : \mathbb{R}^d \to \mathbb{R}^{o_1 \times \cdots \times o_k}$. Define this gradient $\nabla_x g(x) \in \mathbb{R}^{(o_1 \cdots o_k) \times d}$ as the unique matrix-valued function that satisfies

$$\text{vec}\left(g(x+dx) - g(x)\right) \to \nabla_x g(x) \cdot dx \qquad (17)$$

as $dx \to 0$. This convention readily implies the Hessian of a vector-valued function: If $g : \mathbb{R}^d \to \mathbb{R}^o$, then $\nabla_{xx} g(x) \in \mathbb{R}^{o \times d^2}$ is the unique matrix such that $\text{vec}\left(\nabla_x g(x+dx) - \nabla_x g(x)\right) \to \nabla_{xx} g(x)\ dx$. This convention also readily accommodates the chain rule. For example, the gradient of $h(x) \equiv f(g(x))$ for matrix-valued f and g can be written as $\nabla f \nabla g$ as expected. It also implies partial derivatives like $\nabla_{yz} g$ for $g : \mathbb{R}^{|x|} \to \mathbb{R}^{|g|}$. If $y \in \mathbb{R}^{|y|}$ and $z \in \mathbb{R}^{|z|}$ are restrictions of $x \in \mathbb{R}^{|x|}$ to some $|y|$ and $|z|$-dimensional subsets, then $\nabla_z g(x) \in \mathbb{R}^{|g| \times |z|}$, and $\nabla_{yz} g(x) = \nabla_y \nabla_z g(x) \in \mathbb{R}^{|g||z| \times |y|}$. See Chapter 6 of Magnus & Neudecker [7] for a deeper treatment of higher order derivatives of vector-valued functions.

A.1 The term involving dD_x

The matrix D_x is block-diagonal with its ℓth diagonal block containing the matrix $D_\ell \equiv \nabla_x f_\ell$. Using the facts that $\mathrm{vec}(ABC) = \left(C^\top \otimes A\right)\mathrm{vec}(B)$, and $(A \otimes B)^\top = A^\top \otimes B^\top$, we get

$$\begin{aligned} b \cdot (dD_x) &= \begin{bmatrix} b_1 & \cdots & b_L \end{bmatrix} \begin{bmatrix} dD_1 & & \\ & \ddots & \\ & & dD_L \end{bmatrix} \\ &= \begin{bmatrix} b_1 \cdot dD_1 & \cdots & b_L \cdot dD_L \end{bmatrix} \\ &= \begin{bmatrix} \mathrm{vec}\,(dD_1)^\top \left(I \otimes b_1^\top\right) & \cdots & \mathrm{vec}\,(dD_L)^\top \left(I \otimes b_L^\top\right) \end{bmatrix} \\ &= \begin{bmatrix} \mathrm{vec}\,(dD_1) \\ \vdots \\ \mathrm{vec}\,(dD_L) \end{bmatrix}^\top \begin{bmatrix} I \otimes b_1^\top & & \\ & \ddots & \\ & & I \otimes b_L^\top \end{bmatrix} \end{aligned} \tag{18}$$

Observe that $\mathrm{vec}\,(dD_\ell) = d\,\mathrm{vec}\nabla_x f_\ell(z_{\ell-1}; x_\ell)$ varies with dx through both its arguments x_ℓ and $z_{\ell-1}$. Using mixed partials of vector-valued functions described above, we get

$$\mathrm{vec}\,(dD_\ell) = d\,\mathrm{vec}\,(\nabla_x f_\ell) = (\nabla_{xx} f_\ell)\ dx_\ell + (\nabla_{zx} f_\ell)\ dz_{\ell-1}. \tag{19}$$

Stacking these equations gives

$$\begin{bmatrix} \mathrm{vec}\,(dD_1) \\ \vdots \\ \mathrm{vec}\,(dD_L) \end{bmatrix} = \begin{bmatrix} \nabla_{xx} f_1 & & \\ & \ddots & \\ & & \nabla_{xx} f_L \end{bmatrix} dx + \begin{bmatrix} \nabla_{zx} f_1 & & \\ & \ddots & \\ & & \nabla_{zx} f_L \end{bmatrix} \begin{bmatrix} dz_0 \\ \vdots \\ dz_{L-1} \end{bmatrix}. \tag{20}$$

Each vector dz_ℓ in turn varies with dx via $dz_\ell = (\nabla_x f_\ell)dx_\ell + (\nabla_z f_\ell)dz_{\ell-1}$, with the base case $dz_0 = 0$, since the input z_0 does not vary with dx. Stacking up this recurrence gives

$$\begin{bmatrix} I & & & \\ -\nabla_z f_2 & I & & \\ & & \ddots & \\ & & -\nabla_z f_L & 1 \end{bmatrix} \begin{bmatrix} dz_1 \\ \vdots \\ dz_{L-1} \\ dz_L \end{bmatrix} = \begin{bmatrix} \nabla_x f_1 & & \\ & \ddots & \\ & & \nabla_x f_L \end{bmatrix} dx. \tag{21}$$

We can solve for the vector $\begin{bmatrix} dz_1 \\ \vdots \\ dz_L \end{bmatrix} = M^{-1} D_x dx$ and use the downshifting matrix

$$P \equiv \begin{bmatrix} 0 & & \\ I & 0 & \\ & \ddots & \\ & I & 0 \end{bmatrix} \tag{22}$$

to plug back the vector $\begin{bmatrix} dz_0 \\ \vdots \\ dz_{L-1} \end{bmatrix} = PM^{-1} D_x dx$:

$$\begin{bmatrix} \mathrm{vec}\,(dD_1) \\ \vdots \\ \mathrm{vec}\,(dD_L) \end{bmatrix} = \left(\begin{bmatrix} \nabla_{xx} f_1 & & \\ & \ddots & \\ & & \nabla_{xx} f_L \end{bmatrix} + \begin{bmatrix} \nabla_{zx} f_1 & & \\ & \ddots & \\ & & \nabla_{zx} f_L \end{bmatrix} PM^{-1} D_x \right) dx. \tag{23}$$

A.2 The term involving dM

The matrix dM is lower-block-diagonal with $dM_2, \ldots, dM_L$, and $dM_\ell \equiv d\nabla_z f_\ell$. Similar to the above, we can write

$$b \cdot (dM) M^{-1} D_x = \begin{bmatrix} b_1 & \cdots & b_{L-1} & b_L \end{bmatrix} \begin{bmatrix} 0 & & & \\ -dM_2 & 0 & & \\ & & \ddots & \\ & & -dM_L & 0 \end{bmatrix} M^{-1} D_x \tag{24}$$

$$= - \begin{bmatrix} b_2 \cdot dM_2 & \cdots & b_L \cdot dM_L & 0 \end{bmatrix} M^{-1} D_x \tag{25}$$

$$= - \begin{bmatrix} \operatorname{vec}(dM_2)^\top \left(I \otimes b_2^\top\right) & \cdots & \operatorname{vec}(dM_L)^\top \left(I \otimes b_L^\top\right) & 0 \end{bmatrix} M^{-1} D_x \tag{26}$$

$$= - \begin{bmatrix} \operatorname{vec}(dM_1) \\ \vdots \\ \operatorname{vec}(dM_L) \end{bmatrix}^\top \begin{bmatrix} 0 & & & \\ I \otimes b_2^\top & 0 & & \\ & & \ddots & \\ & & I \otimes b_L^\top & 0 \end{bmatrix} M^{-1} D_x \tag{27}$$

$$= - \begin{bmatrix} \operatorname{vec}(dM_1) \\ \vdots \\ \operatorname{vec}(dM_L) \end{bmatrix}^\top \begin{bmatrix} I \otimes b_1^\top & & \\ & \ddots & \\ & & I \otimes b_L^\top \end{bmatrix} P M^{-1} D_x. \tag{28}$$

Each matrix $dM_\ell = d\nabla_z f_\ell(z_{\ell-1}; x_\ell)$ varies with dx through both x_ℓ and $z_{\ell-1}$ as $d \operatorname{vec}(M_\ell) = (\nabla_{xz} f_\ell)\, dx_\ell + (\nabla_{zz} f_\ell)\, dz_{\ell-1}$. Following the steps of the previous section gives

$$\begin{bmatrix} \operatorname{vec}(dM_1) \\ \vdots \\ \operatorname{vec}(dM_L) \end{bmatrix} = \left(\begin{bmatrix} \nabla_{xz} f_1 & & \\ & \ddots & \\ & & \nabla_{xz} f_L \end{bmatrix} + \begin{bmatrix} \nabla_{zz} f_1 & & \\ & \ddots & \\ & & \nabla_{zz} f_L \end{bmatrix} P M^{-1} D_x \right) dx. \tag{29}$$

A.3 Putting it all together

We have just shown that the Hessian of the deep net has the form

$$H \equiv \frac{\partial^2 z_L}{\partial x^2} = D_D \left(D_{xx} + D_{zx} P M^{-1} D_x\right) + D_x^\top M^{-T} P^\top D_M \left(D_{xz} + D_{zz} P M^{-1} D_x\right) \tag{30}$$

$$= D_D D_{xx} + D_D D_{zx} P M^{-1} D_x + D_x^\top M^{-\top} P^\top D_M D_{xz} + D_x^\top M^{-\top} P^\top D_M D_{zz} P M^{-1} D_x. \tag{31}$$

B Equation (9) is Pearlmutter's Hessian-vector multiplication algorithm

We showed that Equation (9) makes it possible to compute Hessian-vector product Hv in time linear in L. These operations are equivalent to Pearlmutter's [9] algorithm, a framework to compute Hessian-vector products in networks with arbitrary topologies. This section specializes Pearlmutter's machinery to the pipeline topology, and shows that the operations it produces coincide exactly with those of Equation (9).

Consider a set of vectors $v_1, \ldots, v_L$ that match the dimensions of the parameter vectors $x_1, \ldots, x_L$. Just as $z_L(x_1, \ldots, x_L)$ denotes the loss under the parameters w, we'll consider the perturbed loss $z_L(x_1 + \alpha v_1, \ldots, x_L + \alpha v_L)$ with a scalar α. By the chain rule,

$$\frac{\partial}{\partial \alpha} z_L(x_1 + \alpha_1 v_1, \ldots, x_L + \alpha_L v_L)\bigg|_{\alpha=0} = \nabla_x z_L(x_1, \ldots, x_L) \cdot v. \tag{32}$$

Applying ∇_x to both sides gives

$$\nabla_x \frac{\partial}{\partial \alpha} z_L(x_1 + \alpha_1 v_1, \ldots, x_L + \alpha_L v_L)\bigg|_{\alpha=0} = \nabla_x^2 z_L(x_1, \ldots, x_L) \cdot v. \tag{33}$$

In other words, to compute the Hessian-vector product $\nabla_x^2 z_L \cdot v$, it suffices to compute the gradient of $\frac{\partial z_L}{\partial \alpha}$ with respect to x. We can do this by applying standard backpropagation to $\frac{\partial z_L}{\partial \alpha}$. At each stage ℓ during its backward pass, backpropagation produces $\frac{\partial}{\partial x_\ell}\frac{\partial z_L}{\partial \alpha} = \nabla_{x_\ell, x} z_L \cdot v$, yielding a block of rows in $\nabla_x^2 z_L \cdot v$.

To see that this generates the same operations as applying Equation (9) to v, we'll write the backprop operations from Equation (2) against $\frac{\partial z_L}{\partial \alpha}$ explicitly. We'll use again the fact that z_ℓ depends on α through both $z_{\ell-1}$ and $x_\ell + \alpha v_\ell$ to massage the backward recursion for $\frac{\partial z_L}{\partial \alpha}$ into a format that matches Equation (2):

$$b'_\ell \equiv \frac{\partial}{\partial z_\ell}\frac{\partial z_L}{\partial \alpha} = \frac{\partial}{\partial \alpha}\frac{\partial z_L}{\partial z_\ell} = \frac{\partial}{\partial \alpha} b_\ell = \frac{\partial}{\partial \alpha}\left[b_{\ell+1} \cdot \nabla_z f_{\ell+1}\right] \tag{34}$$

$$= b'_{\ell+1} \cdot \nabla_z f_{\ell+1} + \left[(I \otimes b_{\ell+1})\frac{\partial}{\partial \alpha}\mathrm{vec}\left(\nabla_z f_{\ell+1}\right)\right]^\top \tag{35}$$

$$= b'_{\ell+1} \cdot \nabla_z f_{\ell+1} + \left[(I \otimes b_{\ell+1})\left(\nabla_{zz} f_{\ell+1} \cdot \frac{\partial z_\ell}{\partial \alpha} + \nabla_{xz} f_{\ell+1} \cdot v_{\ell+1}\right)\right]^\top. \tag{36}$$

During the backward pass, from b_ℓ and b'_ℓ, we compute

$$\frac{\partial}{\partial x_\ell}\frac{\partial z_L}{\partial \alpha} = (\nabla_{x_\ell, x} z_L) \cdot v = \frac{\partial}{\partial \alpha}\frac{\partial z_L}{\partial x_\ell} = \frac{\partial}{\partial \alpha}\left[\frac{\partial z_L}{\partial z_\ell}\frac{\partial z_\ell}{\partial x_\ell}\right] = \frac{\partial}{\partial \alpha}\left[b_\ell \cdot \nabla_x f_\ell\right] \tag{37}$$

$$= b'_\ell \cdot \nabla_x f_\ell + \left[(I \otimes b_\ell)\frac{\partial}{\partial \alpha}\mathrm{vec}\left(\nabla_x f_\ell\right)\right]^\top \tag{38}$$

$$= b'_\ell \cdot \nabla_x f_\ell + \left[(I \otimes b_\ell)\left(\nabla_{zx} f_\ell \cdot \frac{\partial z_{\ell-1}}{\partial \alpha} + \nabla_{xx} f_\ell \cdot v_\ell\right)\right]^\top. \tag{39}$$

Stacking these backward equations horizontally with $b' \equiv \left[\begin{smallmatrix} b'_1 & \cdots & b'_L \end{smallmatrix}\right]$, $g_\ell^\alpha \equiv \frac{\partial z_\ell}{\partial \alpha}$ and $g^\alpha \equiv \begin{bmatrix} g_1^\alpha \\ \vdots \\ g_L^\alpha \end{bmatrix}$, then transposing, gives

$$\begin{aligned} M^\top \left(b'\right)^\top &= P D_M \left(D_{zz} g^\alpha + D_{xz} v\right) \\ \nabla_x^2 z_L \cdot v &= D_x^\top \left(b'\right)^\top + D_D \left(D_{zx} P g^\alpha + D_{xx} v\right). \end{aligned} \tag{40}$$

g_ℓ^α can be computed during the forward pass via

$$g_\ell^\alpha \equiv \frac{\partial z_\ell}{\partial \alpha} = \nabla_z f_\ell \cdot \frac{\partial z_{\ell-1}}{\partial \alpha} + \nabla_x f_\ell \cdot v_\ell = \nabla_z f_\ell \cdot g_{\ell-1}^\alpha + \nabla_x f_\ell \cdot v_\ell, \tag{41}$$

which when stacked up, gives $M g^\alpha = D_x v$. Plugging g^α back into Equation (40) and solving for b' gives

$$\nabla_x^2 z_L \cdot v = D_x^\top M^{-\top} P D_M \left(D_{zz} M^{-1} D_x v + D_{xz} v\right) + D_D \left(D_{zx} P M^{-1} D_x v + D_{xx} v\right). \tag{42}$$

This coincides with Equation (9), showing that the two algorithms are equivalent.

C Examples of Pivoting Matrices to Block-Banded Form

Example 1. *Suppose each block D_{ij} of $\mathcal{K}$ is diagonal. Then the full 4×4 matrix is*

$$\mathcal{K} = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} = \begin{bmatrix} a & 0 & c & 0 \\ 0 & b & 0 & d \\ e & 0 & g & 0 \\ 0 & f & 0 & h \end{bmatrix},$$

where

$$D_{11} = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}, \quad D_{12} = \begin{bmatrix} c & 0 \\ 0 & d \end{bmatrix}, \quad D_{21} = \begin{bmatrix} e & 0 \\ 0 & f \end{bmatrix}, \quad D_{22} = \begin{bmatrix} g & 0 \\ 0 & h \end{bmatrix}.$$

The vector $x = \begin{bmatrix} x_1^1 \\ x_1^2 \\ x_2^1 \\ x_2^2 \end{bmatrix}$ *(in j-major order) is reordered by* Π *into* $\begin{bmatrix} x_1^1 \\ x_2^1 \\ x_2^2 \\ x_1^2 \end{bmatrix}$ *(v-major order). Then*

$$\Pi\mathcal{K}\Pi = \begin{bmatrix} D_{11,11} & D_{12,11} & D_{11,12} & D_{12,12} \\ D_{21,11} & D_{21,11} & D_{21,12} & D_{21,12} \\ D_{11,21} & D_{12,21} & D_{11,22} & D_{12,22} \\ D_{21,21} & D_{21,21} & D_{21,22} & D_{21,22} \end{bmatrix} = \begin{bmatrix} a & c & 0 & 0 \\ e & g & 0 & 0 \\ 0 & 0 & b & d \\ 0 & 0 & f & h \end{bmatrix}$$

Notice that the resulting matrix has become block-diagonal, whereas $\mathcal{K}$ had a bandwidth of 2.

Example 2. *Now endow each block D_{ij} with an upper off-diagonal entry:*

$$D_{11} = \begin{bmatrix} a & \alpha \\ 0 & b \end{bmatrix}, \quad D_{12} = \begin{bmatrix} c & \beta \\ 0 & d \end{bmatrix}, \quad D_{21} = \begin{bmatrix} e & \delta \\ 0 & f \end{bmatrix}, \quad D_{22} = \begin{bmatrix} g & \gamma \\ 0 & h \end{bmatrix}.$$

The full 4×4 matrix is

$$\mathcal{K} = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} = \begin{bmatrix} a & \alpha & c & \beta \\ 0 & b & 0 & d \\ e & \delta & g & \gamma \\ 0 & f & 0 & h \end{bmatrix}.$$

After applying the same permutation Π as before, we obtain

$$\Pi\mathcal{K}\Pi^\top = \begin{bmatrix} a & c & \alpha & \beta \\ e & g & \delta & \gamma \\ 0 & 0 & b & d \\ 0 & 0 & f & h \end{bmatrix}.$$

The off-diagonal entries in each block D_{ij} are collected in the upper-right block of the permuted matrix. The permuted matrix is block-banded with bandwidth 2, with a dense upper-right block and a sparse lower-left block.