BACKPROPAGATION THROUGH BACK SUBSTITUTION WITH A BACKSLASH*

ALAN EDELMAN†, EKIN AKYÜREK‡, AND YUYANG WANG§

Abstract. We present a linear algebra formulation of backpropagation which allows the calculation of gradients by using a generically written "backslash" or Gaussian elimination on triangular systems of equations. Generally, the matrix elements are operators. This paper has three contributions: (i) it is of intellectual value to replace traditional treatments of automatic differentiation with a (left acting) operator theoretic, graph-based approach; (ii) operators can be readily placed in matrices in software in programming languages such as Julia as an implementation option; (iii) we introduce a novel notation, "transpose dot" operator "$\{\}^{T\bullet}$" that allows for the reversal of operators. We further demonstrate the elegance of the operators approach in a suitable programming language consisting of generic linear algebra operators such as Julia [Bezanson et al., *SIAM Rev.*, 59 (2017), pp. 65–98], and that it is possible to realize this abstraction in code. Our implementation shows how generic linear algebra can allow operators as elements of matrices. In contrast to "operator overloading," where backslash would normally have to be rewritten to take advantage of operators, with "generic programming" there is no such need.

1. Preface: Summary and the challenge. This paper provides the mathematics to show how an operator theoretic, graph-based approach can realize backpropagation by applying back substitution to a matrix whose elements are operators.

As a showcase result, one can back-propagate to compute the gradient on feedforward neural networks (or multilayer perceptron (MLP)) [5] with

$$(1.1) \qquad \nabla J = M^T((I - \tilde{L})^T \backslash g),$$

where M (source to nonsource nodes) and $\tilde{L}$ (within nonsource nodes) are blocks of the adjacency matrix of the computational graph (see section 3.2.2 for precise definitions), g is the vector of gradients of the loss function, and I is the identity matrix. For readers unfamiliar with the backslash notation, an equivalent expression of (1.1) is $\nabla J = M^T (I - \tilde{L})^{-T} g$.

We then set up a challenge to ourselves. Could we correctly implement (1.1) by simply typing the command (after basic setup but *without overloading of backslash*)

$$\boxed{\texttt{∇J = M' * (ImL' \textbackslash\ g)}}\ ?$$

We demonstrate that indeed the backpropagation can be achieved, almost by magic, in a programming language armed with generic programming allowing for operators as elements of matrices. The software in section 6.1 is by itself interesting not for the usual reasons of what it does, but in this case how it does it: how a powerful language with generic programming and multiple dispatch can allow this abstract mathematical formulation to be realized.

2. Introduction: Linear algebra, graphs, automatic differentiation, operators, and Julia. Automatic differentiation (AD) is fundamental to gradient-based optimization of neural networks and is used throughout scientific computing. There are two popular approaches to AD:, namely, forward and backward (reverse) modes [6, 7, 2, 13], the latter of which is also known as *backpropagation* in the machine learning literature.[1] A common high-level description of AD is that it is really "only" the chain rule. The centuries old technology of taking derivatives is taking on a modern twist in the form of differentiable programming [8, 11]. Who would have thought that one of the most routine college course subjects would now be the subject of much renewed interest both in applied mathematics and computer science?

This paper introduces the notion that AD is best understood with a matrix-based approach. The chain-rule explanation, in retrospect, feels to us as a distraction or at least extra baggage. We suspect that while the chain rule is well known, it is understood mechanically rather than deeply by most of us. We argue that a linear algebra based framework for AD, while mathematically equivalent to other approaches, provides a simplicity of understanding, and equally importantly a viable approach worthy of further study.

Regarding software, while most high-level languages allow for matrices whose elements are scalars, the ability to work with matrices whose elements might be operators without major changes to the elementwise software is an intriguing abstraction. We discuss a Julia implementation that makes this step particularly mathematically natural.

It is our view that a linear algebraic approach sheds light on how backpropagation works in its essence. We theoretically connect backpropagation to the back

---

[1] Despite similar terminology, the term forward propagation (or forward pass) in machine learning has no connection to forward mode AD. Instead, it refers to the process where a neural network calculates its output by sequentially passing input data through each layer, applying weighted sums and activation functions, until it reaches the output layer. In later sections, Algorithm 5.1 and 5.2 illustrate such a procedure. Whereas "backpropagation" (backward pass or reverse mode AD) is so named because information flows backwards through the network during this process.

substitution method for triangular systems of equations. Similarly, forward substitution corresponds to the forward mode calculation of AD. As is well documented in the preface to the book Graph Algorithms in the Language of Linear Algebra [10], there have been many known benefits to formulate mathematically a graph algorithm in linear algebraic terms.

The ability to implement these abstractions while retaining performance is demonstrated using Julia, a language that facilitates abstractions, multiple dispatch, the type system, and which offers generic operators.

3. A matrix method for weighted paths.

3.1. "Forward and back" through graphs and linear algebra. In the spirit of reaching the mathematical core, let us strip away the derivatives, gradients, Jacobians, the computational graphs, and the "chain rule" that clutter the story of how is it possible to compute the same thing forwards and backwards. We set ourselves the goal of explaining the essence of forward mode versus backward mode in AD with a single figure. Figure 3.1 is the result. Note that "forward mode" differentiation is not to be confused with the forward computation of the desired quantity.

3.1.1. Path weights on directed graphs. Consider a directed acyclic graph (DAG) with edge weights as in Figure 3.1 where nodes 1 and 2 are sources
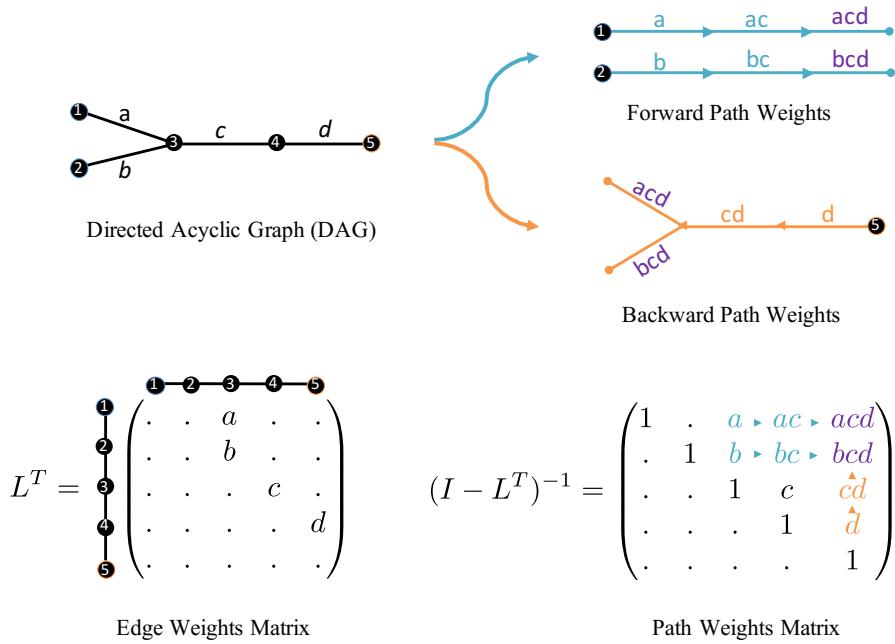
FIG. 3.1. *Legend: Purple: target weights; blue: forward computation; orange: backward computation. The dots in matrices denote zeros. (Upper left) Multiply the weights along the paths from source node 1 to sink node 5 and also source node 2 to sink node 5 to obtain acd and bcd. (Right blue) The obvious forward method. (Right orange) A backward method that requires one fewer multiplication. (Below) A matrix method: if $L_{ij}^T =$ the weight on edge ij, then $(I - L^T)^{-1}$ simultaneously exhibits the forward (i.e., $a \to ac \to acd$ and $b \to bc \to bcd$) and backward methods (i.e., $d \to cd \to bcd \to acd/bcd$). (Note: color appears only in the online article.)*

Two Equivalent Ways to Compute the Path Weights in Figure 3.1:	
Forward Substitution:	$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}^T \left[(I-L)^{-1} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \right]$
Back Substitution:	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T \left[(I-L^T)^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right]$

FIG. 3.2. *The forward and backward methods compared: Both are seen equivalently as a choice of parenthesizing* (3.1) *or as forward substitution versus back substitution. Generally speaking, when the number of sources is larger than the number of sinks, one might expect the backward method to have less complexity.*

(starting nodes), and node 5 is a sink (end node). The problem is to compute the *path weights*, which we define as the products of the weights from every start node to every sink node.

Evidently, the path weights that we seek in Figure 3.1 may be obtained by calculating

$$(3.1) \qquad \text{path weights} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T}_{\text{sources}} (I-L^T)^{-1} \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{sink}},$$

where L^T, the adjacency matrix (edge weight matrix), is displayed in the lower left of Figure 3.1. One explanation of why (3.1) works for calculating the path weights is that $(L^T)^k_{ij}$ sums the path weights of length k from node i to node j and $(I-L^T)^{-1} = I + L^T + \cdots + (L^T)^{n-1}$ then counts path weights of all lengths from i to j.

If one follows step by step the linear algebra methods of forward substitution for lower triangular matrices or back substitution for upper triangular matrices, one obtains path weights algorithms as summarized in Figure 3.2. We remind the reader that forward and back substitutions are the standard methods to solve lower and upper triangular systems, respectively.

3.1.2. Generalizing forward and back to a Catalan number of possibilities. Continuing with the same L matrix from section 3.1, we can begin to

understand all of the possibilities including the forward method, the backward method, the mixed-modes methods, and even more possibilities:

$$
\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T (I - L^T)^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}
$$

$$
= \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T
\begin{pmatrix} 1 & . & a & . & . \\ . & 1 & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & . \\ . & . & . & . & 1 \end{pmatrix}
\begin{pmatrix} 1 & . & . & . & . \\ . & 1 & b & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & . \\ . & . & . & . & 1 \end{pmatrix}
\begin{pmatrix} 1 & . & . & . & . \\ . & 1 & . & . & . \\ . & . & 1 & c & . \\ . & . & . & 1 & . \\ . & . & . & . & 1 \end{pmatrix}
\begin{pmatrix} 1 & . & . & . & . \\ . & 1 & . & . & . \\ . & . & 1 & . & . \\ . & . & . & 1 & d \\ . & . & . & . & 1 \end{pmatrix}
\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.
$$

It is well known [14] that there are a Catalan number, $C_5 = 42$, ways to parenthesize the above expression. One of the 42 choices evaluates left to right; this is forward substitution which computes the graph weights forward. Another evaluating from right to left is backward substitution. There are three other "mixed-modes" [12] which combine forward and backward methods. The remaining 37 methods require matrix-matrix multiplication as a first step. We encourage the reader to work out some of these on the graph. Partial products correspond to working through subgraphs. Perhaps readers might find cases where working from the middle outward can be useful. For example it would be possible to go from the middle outward using the Example of Figure 3.1: we would go from c to cd then compute acd and bcd.

3.1.3. Edge elimination. It is possible to eliminate an edge (and preserve the path weights) by moving the weight of the edge to the weights of the incoming edges. We illustrate this in Figure 3.3 by eliminating the edge from node 3 to node 4, moving the weight c to the incoming edges by multiplication. The corresponding linear algebra operation on L^T is the deletion of column 3 and row 3 and the rank 1 update based on this column and row with the (3,3) element deleted. The corresponding linear algebra operation on $(I - L^T)^{-1}$ is merely the deletion of column 3 and row 3. This example is representative of the general case.

3.1.4. Edge addition at the sink node. We will be interested in the case where the edge weight graph is modified by adding one edge to the sink node. Continuing our example from Figure 3.3, we will add an edge "e" by starting with

$$
(3.2) \qquad \text{path weights} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T}_{\text{sources}} \underbrace{(I - L^T)^{-1}}_{\text{path weights matrix}} \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{sink}}
$$
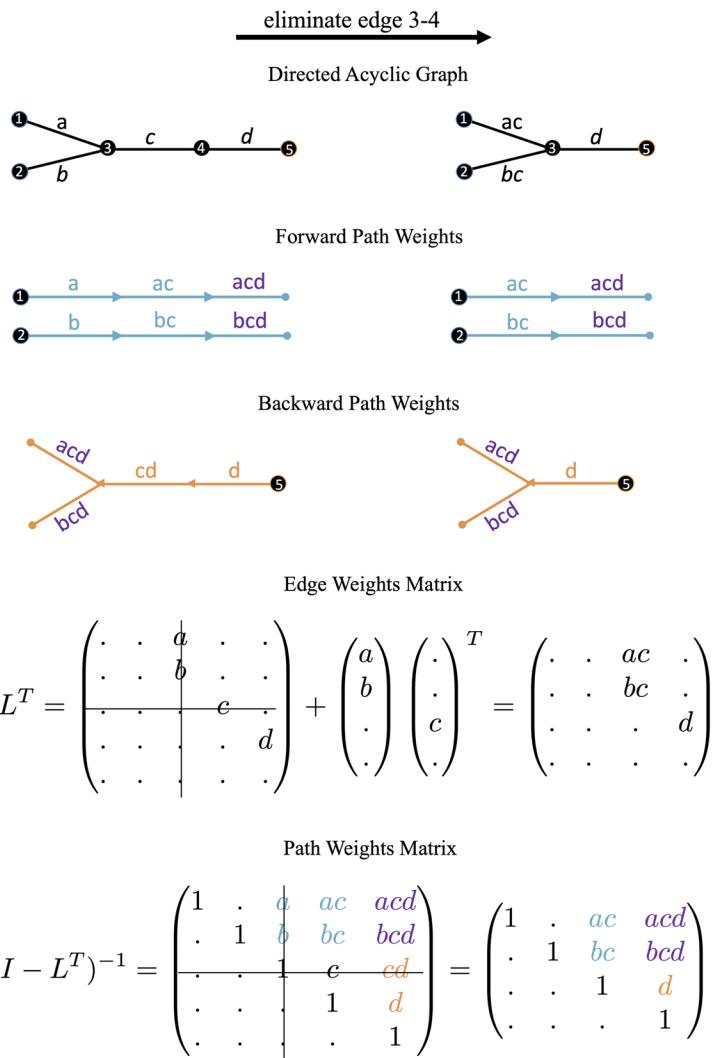
FIG. 3.3. *Elimination of the edge from node* 3 *to node* 4 *on the graphs and with matrices. The matrix versions involve a rank one update to a row and column deleted matrix in the case of the edge weights matrix and only a deletion of a row and column in the path weights matrix. (Note: color appears only in the online article.)*

and then updating by augmenting the graph with one end node to become

(3.3)

$$
\text{updated path weights} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T}_{\text{sources}} \underbrace{\begin{pmatrix} (I-L^T)^{-1} & . \\ . & 1 \end{pmatrix} \begin{pmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & e \\ . & . & . & . & . & 1 \end{pmatrix}}_{\text{updated path weights matrix}} \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{sink}}.
$$

The update from the path weights matrix in (3.2) to the updated path weights matrix in (3.3) can be verified in many ways. One simple way is to look at the explicit elements of the path weights matrix before and after and then notice that the new matrix has a column with one more element e augmented with a 1.

It is an easy exercise in linear algebra to show that (3.3) is the same as (3.4) which folds the added edge e multiplicatively into the sink vector:

$$(3.4) \qquad \text{updated path weights } = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}}_{\text{sources}}^{T} (I - L^T)^{-1} \underbrace{\begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ e \end{pmatrix}}_{\text{sink}}.$$

3.2. Examples of DAGs and weighted paths.

3.2.1. The "complete DAG" and weighted paths. Consider as an example in Figure 3.4, the complete DAG on four nodes with graph weights evaluated through a forward and backward method. There is one source and one sink. We find that this complete DAG example reveals most clearly the equivalence between path weights and the inverse matrix.

We see that the forward path weights folds in the edges labeled "a," then "b," then "c." This works through the matrix L^T by columns. The backward mode folds in the edges with subscript "3," then "2," then "1." This works through the matrix L^T by rows from bottom to top.
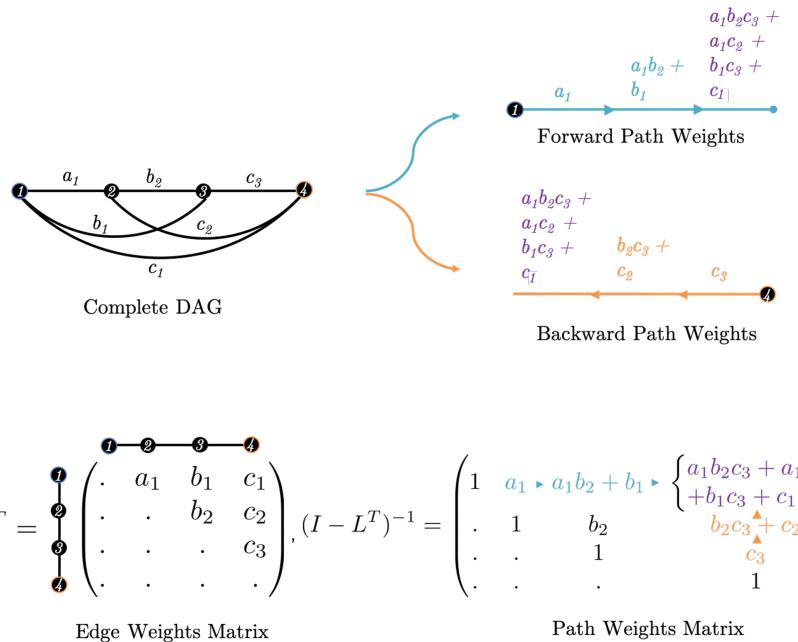

FIG. 3.4. *The complete DAG on four nodes illustrates a symmetric situation where forward and backward have the same complexity but arrive at the same answer through different operations. (Note: color appears only in the online article.)*
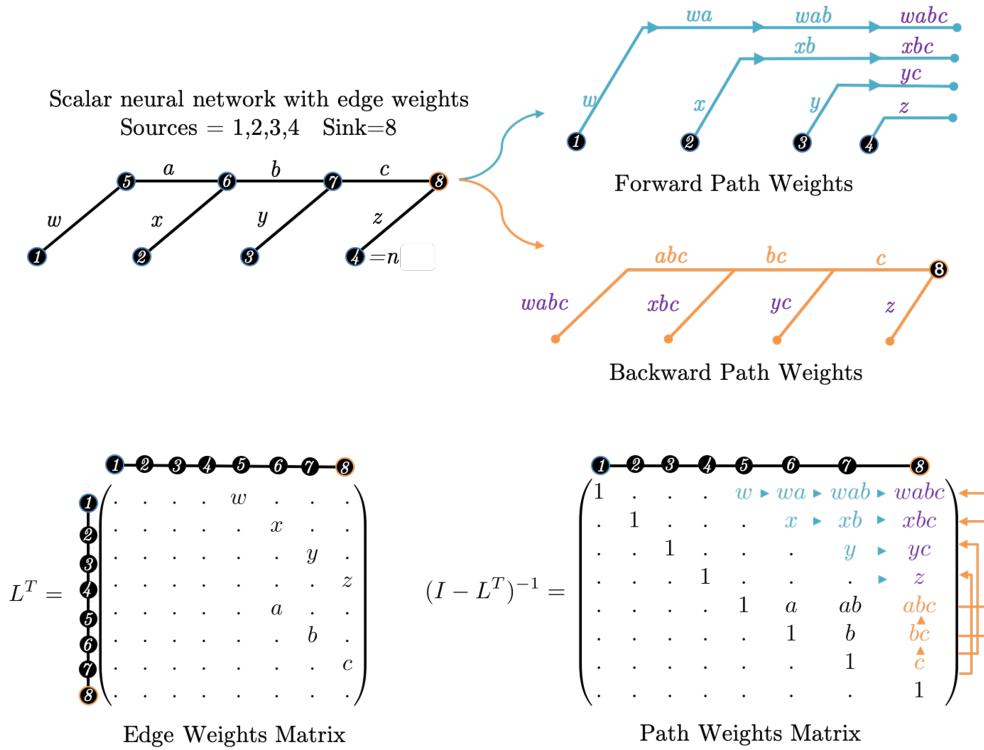
FIG. 3.5. *This diagram contains most of what is needed to understand forward and backward propagation of derivatives through an MLP. The details of what the weights look like will come later. If we take $n = 4$ for the pictured network, the sources are labeled $1:n$ and the sink is labeled $2n$. Forward mode requires $n(n-1)/2$ multiplications while backward mode requires $2n-3$ multiplications. (Note: color appears only in the online article.)*

3.2.2. The "MLP DAG" and weighted paths. Figure 3.5 is the DAG for the derivatives in a MLP. It may be thought of as a spine with feeds for parameters (nodes 1,2,3, and 4 in the figure).

If sources are labeled $1, \ldots, s$ (in Figure 3.5, $s = 4$), then the top left s by s matrix in L^T is the zero matrix as there are no connections. We can then write

$$(3.5) \qquad L^T = \begin{pmatrix} 0 & M^T \\ 0 & \tilde{L}^T \end{pmatrix},$$

where

$$M^T = \begin{pmatrix} w & . & . & . \\ . & x & . & . \\ . & . & y & . \\ . & . & . & z \end{pmatrix}, \tilde{L}^T = \begin{pmatrix} . & a & . & . \\ . & . & b & . \\ . & . & . & c \\ . & . & . & . \end{pmatrix},$$

where the matrix M^T corresponds to connections between the sources and internal nodes, and $\tilde{L}^T$ corresponds to internal connections. In this example M^T is diagonal corresponding to a bipartite matching between nodes $1, 2, 3, 4$ and $5, 6, 7, 8$. The $\tilde{L}^T$ matrix represents internal connections; in this case it is the "spine" linearly connecting nodes $5, 6, 7, 8$.

Now we have

$$(I - L^T) = \begin{pmatrix} I & -M^T \\ 0 & I - \tilde{L}^T \end{pmatrix} \text{ and } (I - L^T)^{-1} = \begin{pmatrix} I & M^T(I - \tilde{L}^T)^{-1} \\ 0 & (I - \tilde{L}^T)^{-1} \end{pmatrix}.$$

If the last node is the one unique sink, then we obtain the useful formula

$$(3.6) \qquad \text{path weights} = M^T(I - \tilde{L}^T)^{-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

We can now take a close look at Figure 3.5 and fully grasp the path weight structure. The spine consisting of a, b, c, and 1 (understood) requires the computation of the cumulative suffix product $1, c, bc, abc$. What follows is an elementwise multiplication by z, y, x, w, from which we can calculate the last column of $M^T(I - \tilde{L}^T)^{-1}$:

$$(3.7) \qquad M^T(I - \tilde{L}^T)^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} wabc \\ xbc \\ yc \\ z \end{pmatrix}.$$

3.3. Computational graphs, derivative graphs, and their superposition. Many treatments of AD introduce computational graphs at the start of the discussion. Our treatment shows that this is not necessary. However, in the end the key application of edge weights will be as derivatives of computed quantities. To this end, we define the following.

DEFINITION 3.1. *A computational graph is a node labeled DAG, where leaf nodes consist of variable names, and nonleaf nodes contain variable names and formulas that depend on incoming variables.*

We remark that there are variations on where the variable names and formulas live on a computational graph, but we believe the definition here is the cleanest when wishing to incorporate derivative information.

3.3.1. The chain rule, derivatives, and Jacobians. Here we say explicitly how the edge weights and path weights relate to derivatives in a computation.

Consider the computation from Figure 3.6; the next three algorithms show the computation, the derivatives of each line of code, and the overall derivatives. We see that the one step derivatives are edge weights and the overall derivatives are path weights.

If the final output is a scalar, we immediately have that the gradient with respect to the source x and y (in Figure 3.6) is exactly the path weight defined in (3.6),

$$(3.8) \qquad \text{gradient} = \text{the last column of } M^T(I - \tilde{L}^T)^{-1},$$

which corresponds to the output in Algorithm 3.8 with

$$M^T = \begin{pmatrix} y & \cdot & \cdot \\ x & \cdot & \cdot \end{pmatrix}, \quad \tilde{L} = \begin{pmatrix} \cdot & 2p & \cdot \\ \cdot & \cdot & -e^{-q} \\ \cdot & \cdot & \cdot \end{pmatrix}.$$

Equation (3.8) fully describes backpropogation. Table 1 summarizes the transition from the graph elements to AD. For completeness, the term "forward propagation" (forward pass) describes the process of executing the computational graph in a forward direction (left to right), storing the intermediate values that are subsequently utilized in backpropagation (backward pass).
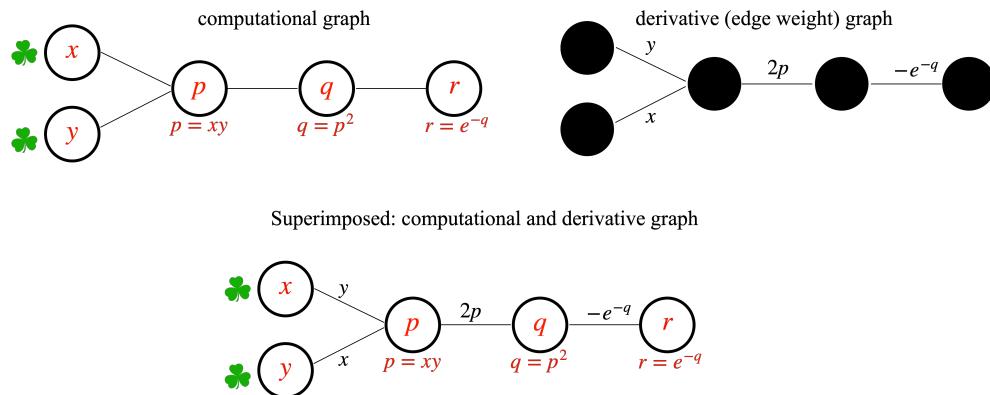
FIG. 3.6. *An example of computational (node) graph, derivative (edge) graph, and their "superposition." (Note: color appears only in the online article.)*

TABLE 1
A dictionary translating graph elements to AD (cf. Figure 3.6*).*

Player	Description
Edge weight from node i and j	These are the derivatives of one step of a computation. These can be scalars but in general these are Jacobian matrices (or operators).
Path weight from node i to j	These are the derivatives that reach back into a chain of computations. The chain rule states that if you multiply ("chain together") the derivatives at each step you get the dependence of one variable on an earlier variable.
Source	The sources in the graph are typically parameters in real computations, as many modern applications are interested in the derivatives with respect to the input parameters.
Sink	The sink is usually what is known as a loss function in modern applications.

Algorithm 3.1. Simple algorithm example from Figure 3.6.

1: $p \leftarrow \text{multiply}(x, y)$
2: $q \leftarrow \text{square}(p)$
3: $r \leftarrow \text{exp\_neg}(q)$
4: output r

Algorithm 3.2. Edge weights (derivatives of one line of code).

1: $d\{\text{multiply}(x, y)\}/dx = y \quad (= \frac{\partial p}{\partial x})$
2: $d\{\text{multiply}(x, y)\}/dy = x \quad (= \frac{\partial p}{\partial y})$
3: $d\{\text{square}(p)\}/dp = 2p \quad (= \frac{\partial q}{\partial p})$
4: $d\{\text{exp\_neg}(q)\}/dr = -e^{-q} \quad (= \frac{\partial r}{\partial q})$

Algorithm 3.3. Path weights (chained derivatives).

1: $dr/dx = y \times 2p \times (-e^{-q})$ (chain lines 1,3, and 4 of Algorithm 3.2)
2: $dr/dy = x \times 2p \times (-e^{-q})$ (chain lines 2,3, and 4 of Algorithm 3.2)

4. Linear operators as elements of matrices. We will illustrate in section 6.1 the value of software that allows linear operators as elements of matrices. Here we set the mathematical stage, starting with a question.

Consider a matrix transformation of X such as $T_{A,B} : X \mapsto BXA^T$, how should we represent the Jacobian $\partial T_{A,B}/\partial X$?

Before we answer, we remind the reader how the Kronecker product works. One view of the Kronecker product $A \otimes B$ of two matrices is that it multiplies every element in A times every element of B placing the elements in such a way that we have the identity

$$(A \otimes B)\text{vec}(X) = \text{vec}(BXA^T),$$

where *vec* denotes the flattening of a matrix X into a vector by stacking its columns. We may abuse notation when there is no confusion and write

$$(A \otimes B)(X) = BXA^T$$

for the linear operator $T_{A,B}$ that sends X to BXA^T. Identifying the matrix $A \otimes B$ with the operator is more than a handy convenience, it makes computations practical in a software language that allows for this. Table 2 defines some operators of interest.

Consider the inner product (matrix dot product) $\langle X, Y \rangle = \text{Tr}(X^T Y)$. The identity $\langle X, AY \rangle = \langle A^T X, Y \rangle$ implies$(A_L)^T = (A^T)_L$; in words, the operator adjoint with respect to the operator A_L (left multiplication by A) is left multiplication by A^T. The operator transposes are $(A_L)^T = (A^T)_L$, $(B_R)^T = (B^T)_R$, and $(M_H)^T = M_H$ (symmetric).

We wish to propose a carefully thought out notation for another useful operator, $G^{T\bullet}$ ("G transpose dot"), the matrix inner (or dot) product with G.

DEFINITION 4.1. *Let $G^{T\bullet}$ denote the matrix inner (or dot) product with G. This operator takes a matrix X of the same size as G and returns the scalar, $G^{T\bullet}X := Tr(G^T X) = vec(G)^T vec(X) = \sum_{i,j} G_{ij} X_{ij}$.*

Many communities choose a notation where small Roman letters denote a column vector, so that $x \mapsto g^T x$ denotes a linear function of x. Those who are used to this notation no longer "see" the transpose so much as turning a column into a row, but rather they see the linear function g^T as an object that acts on ("eats") vectors and returns scalars. In the same way we propose that one might denote a linear function of a matrix $X \mapsto \text{Tr}(G^T X)$ with the operator notation $X \mapsto G^{T\bullet}X$, an operator that eats matrices and returns scalars.

TABLE 2

Matrix operators and the size of their dense representations assuming $X : m \times n$, $A : n_1 \times n$, $B : m_1 \times m$, $M : m \times n$, and $G : m \times n$. We overload $A \otimes B$ to be both the operator and the matrix.

	Symbol	Definition	Dense representation	
Kronecker product of A, B	$A \otimes B$	$X \mapsto BXA^T$	$A \otimes B$	$m_1 n_1 \times mn$
Left multiplication by B	B_L	$X \mapsto BX$	$I \otimes B$	$m_1 n \times mn$
Right multiplication by A	A_R	$X \mapsto XA$	$A^T \otimes I$	$mn_1 \times mn$
Hadamard product with M	M_H	$X \mapsto M.*X$	$\text{diag}(\text{vec}(M))$	$mn \times mn$
Matrix inner product with G	$G^{T\bullet}$	$X \mapsto \text{Tr}(G^T X)$	$\text{vec}(G)^T$	$1 \times mn$

LEMMA 4.1. *If the superscript "$()^T$" is overloaded to denote real operator adjoint or matrix transpose as appropriate, $\mathcal{L}$ is a linear operator and G is a matrix, then we have the operator identity: $(\mathcal{L}^T G)^{T\bullet} = G^{T\bullet}\mathcal{L}$. Notice that if we pretend all letters are just matrices and you ignore the dot, the notation has the appearance of the familiar transpose rule.*

Proof. We have that for all X,

$$(\mathcal{L}^T G)^{T\bullet} X = \langle \mathcal{L}^T G, X \rangle = \langle G, \mathcal{L}X \rangle = G^{T\bullet}\mathcal{L}X,$$

showing that as operators $(\mathcal{L}^T G)^{T\bullet} = G^{T\bullet}\mathcal{L}$.   □

As an example, we have

$$(A_L^T G)^{T\bullet} = X \mapsto \mathrm{Tr}((A^T G)^T X) \quad \text{and} \quad G^{T\bullet} A_L = X \mapsto \mathrm{Tr}(G^T A X),$$

which shows that $(A_L^T G)^{T\bullet} = G^{T\bullet} A_L$. We encourage the reader to follow the matrices A, G, A^T and the operators $A_L^T, A_L, (A_L^T G)^{T\bullet}, G^{T\bullet}$. (See section 5.4 for why this notation can be valuable.)

5. Operator methodology. We proceed from matrices of scalars to matrices of vectors to matrices of operators in sections 5.1, 5.2, and 5.3. ultimately taking advantage of Julia's capabilities. We encourage the reader to compare the matrices in each of these sections. Section 5.4 illustrates the power of the $G^{T\bullet}$ notation, while section 5.5 shows the relationship to the adjoint method that is well known in the field of scientific computing.

5.1. Matrices of scalars. The simple case of scalar neural networks (shown in Algorithm 5.1 and Figure 5.1) without bias shows the power of the graph approach. However, the full power is revealed in the coming sections. Here we remind the reader of the algorithm, draw the graphs, and instantly write down the linear algebra that provides the gradients through backpropogation. (The graphs and matrices are illustrated for $N = 4$ for ease of presentation.)

Starting with

$$\tilde{L}^T = \begin{pmatrix} . & \delta_2 w_1 & . & . \\ . & . & \delta_3 w_2 & . \\ . & . & . & \delta_4 w_3 \\ . & . & . & . \end{pmatrix}, M = \begin{pmatrix} \delta_1 x_0 & . & . & . \\ . & \delta_2 x_1 & . & . \\ . & . & \delta_3 x_2 & . \\ . & . & . & \delta_4 x_3 \end{pmatrix},$$

it is an immediate consequence of our graph theory methodology which concluded with (3.4) and (3.7) that the backpropagated gradient is computed by evaluating efficiently

---

Algorithm 5.1. Scalar MLP without bias (forward propagation).

1: Input data x_0, initial weights $w_i, \quad i = 1, \ldots, N$
2: Select activation functions $h_i(\cdot)$ such as sigmoid, tanh, ReLU, etc.
3: **for** $i = 1$ to N **do**
4:      $x_i \leftarrow h_i(w_i x_{i-1})$
5:      $(\delta_i \leftarrow h_i'(w_i x_{i-1}))$
6: **end for**
7: Output x_N

Scalar MLP: computational graph

Scalar MLP: (edge weight) derivative graph

Scalar MLP: computational graph superimposed
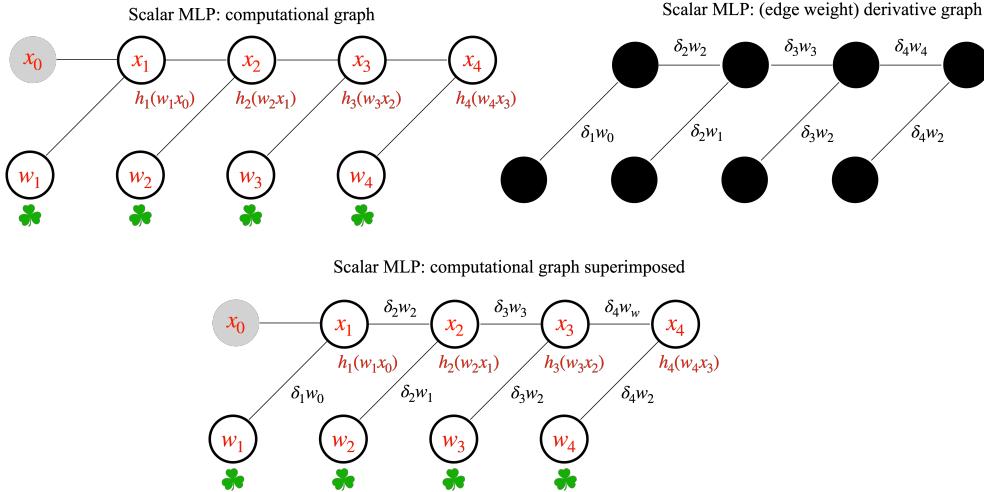
FIG. 5.1. *Top left: computational graph of a scalar MLP. This computation, which has nothing to do with derivatives, is often referred to as forward propagation because of its direction. Evaluation must generally necessarily go from left to right. Top right: derivative edge weights. Since derivatives are linear, multiple directions are possible to evaluate the products. Bottom: the superimposed graph showing both the forward computation and the derivative edge weights. (Note: color appears only in the online article.)*

$$
\nabla_w \mathcal{L} = \begin{pmatrix} \delta_1 x_0 & & & \\ & \delta_2 x_1 & & \\ & & \delta_3 x_2 & \\ & & & \delta_4 x_3 \end{pmatrix} \begin{pmatrix} 1 & & & \\ -\delta_2 w_1 & 1 & & \\ & -\delta_3 w_2 & 1 & \\ & & -\delta_4 w_3 & 1 \end{pmatrix}^{-T} \begin{pmatrix} . \\ . \\ . \\ \mathcal{L}'(x_4) \end{pmatrix}.
$$

5.2. Matrices of vectors. As a baby step towards the matrices of operators approach, we show how one can (optionally) group weights and biases that appear in a neuron. Algorithm 5.1 is modified so that $w_i x_{i-1}$ is replaced with $w_i x_{i-1} + b_i$. In the interest of space, we will simply write the answer of $\nabla_{[w,b]}\mathcal{L}$ and discuss its format:

$$
\begin{pmatrix} [\delta_1 x_0 \ \delta_1] & & & \\ & [\delta_2 x_1 \ \delta_2] & & \\ & & [\delta_3 x_2 \ \delta_3] & \\ & & & [\delta_4 x_3 \ \delta_4] \end{pmatrix}^T \begin{pmatrix} 1 & & & \\ -\delta_2 w_1 & 1 & & \\ & -\delta_3 w_2 & 1 & \\ & & -\delta_4 w_3 & 1 \end{pmatrix}^{-T} \begin{pmatrix} . \\ . \\ . \\ \mathcal{L}'(x_4) \end{pmatrix}.
$$

We see we have an ordinary matrix back substitution followed by multiplication by a diagonal matrix of row vectors of length 2 so that the result is a vector of column vectors of length 2 which nicely packages the gradients with respect to the weight and bias in each neuron. We remark that the transpose applies recursively in the diagonal matrix. The transpose is overkill in this case but is critical in the next section.

Algorithm 5.2. Matrix MLP (forward propagation).

1: Input data X_0 ($n_0 \times k$), and initial weight matrices and corresponding bias terms $W_i(n_i \times n_{i-1}), B_i(n_i \times k)$
2: Select activation functions $h_i(\cdot)$ such as sigmoid, tanh, ReLU, etc.
3: **for** $i := 1$ to N **do**
4: $\quad X_i \leftarrow h_i(W_i * X_{i-1} + B_i)$.
5: $\quad (\Delta_i \leftarrow h_i'(W_i * X_{i-1} + B_i))$
6: **end for**
7: output X_N

5.3. Matrices of operators. Letting $\mathcal{I}$ denote the identity operator and empty space the zero operator, we have the following,

$$
\nabla_{[W,B]}\mathcal{L}
$$
$$
= \begin{pmatrix} [\Delta_{1H} \circ X_{0R}\ \Delta_{1H}] & & & \\ & [\Delta_{2H} \circ X_{1R}\ \Delta_{2H}] & & \\ & & [\Delta_{3H} \circ X_{2R}\ \Delta_{3H}] & \\ & & & [\Delta_{4H} \circ X_{3R}\ \Delta_{4H}] \end{pmatrix}^{T}
$$
$$
\times \begin{pmatrix} \mathcal{I} & & & \\ -\Delta_{2H} \circ W_{2L} & \mathcal{I} & & \\ & -\Delta_{3H} \circ W_{3L} & \mathcal{I} & \\ & & -\Delta_{4H} \circ W_{4L} & \mathcal{I} \end{pmatrix}^{-T} \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \nabla_{X_4}\mathcal{L} \end{pmatrix}
$$

for the matrix neural network in Algorithm 5.2. The entries of our matrix of operators may be read immediately from the differential of line 4 of Algorithm 5.2,

$$
\begin{aligned}
dX_i &= d\left[h_i(W_i X_{i-1} + B_i\right] \\
&= (\Delta_{iH} \circ X_{i-1R})dW_i + \Delta_{iH}dB_i + (\Delta_{iH} \circ W_{iL})dX_{i-1},
\end{aligned}
$$

where Δ_i is the gradient matrix, and the definitions of the operators Δ_{iH}, W_{iL}, and X_{i-1R} are given in Table 2.

5.4. The power of notation. We read directly off the edge weight graph in Figure 5.2 that for a matrix neural network we have

FORWARD MODE OPERATORS (RIGHT TO LEFT)

(5.1) $\qquad \dfrac{\partial \mathcal{L}}{\partial W_i} = G^{T\bullet}(\Delta_N)_H(W_N)_L \ldots (W_{i+2})_L(\Delta_{i+1})_H(W_{i+1})_L(\Delta_i)_H(X_{i-1})_R,$

$\qquad \dfrac{\partial \mathcal{L}}{\partial B_i} = G^{T\bullet}(\Delta_N)_H(W_N)_L \ldots (W_{i+2})_L(\Delta_{i+1})_H(W_{i+1})_L(\Delta_i)_H$

or going the other way we have

BACKWARD MODE OPERATORS (RIGHT TO LEFT)

(5.2) $\left[\dfrac{\partial \mathcal{L}}{\partial W_i}\right]^{T\bullet} = \left\{(X_{i-1}^T)_R(\Delta_i)_H(W_{i+1}^T)_L(\Delta_{i+1})_H(W_{i+2}^T)_L \ldots (W_N^T)_L(\Delta_N)_H G\right\}^{T\bullet},$

$\qquad \left[\dfrac{\partial \mathcal{L}}{\partial B_i}\right]^{T\bullet} = \left\{(\Delta_i)_H(W_{i+1}^T)_L(\Delta_{i+1})_H(W_{i+2}^T)_L \ldots (W_N^T)_L(\Delta_N)_H G\right\}^{T\bullet}.$
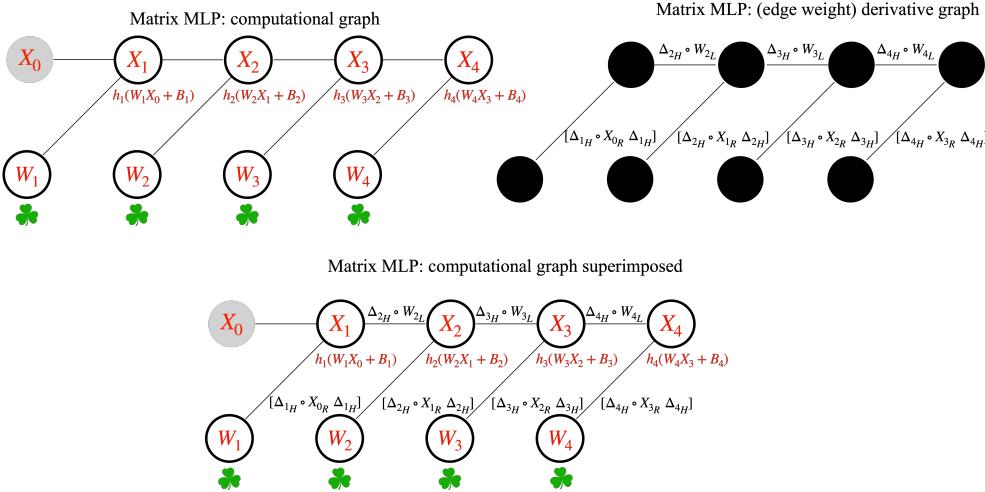
FIG. 5.2. *Computational and derivative graphs of a matrix MLP with their superimposed version. Compared to Figure* 5.1 *of the scalar MLP, everything remains the same except for two changes: the elements in the computational graph are now matrices, and the edges in the derivative graph have been replaced by operators. (Note: color appears only in the online article.)*

Understanding these operators. The forward operators in (5.1) may be thought of as sensitivity operators or as a means of computing the full gradient. As a sensitivity operator, one can state that the directional derivative of $\mathcal{L}$ in the direction ΔW_i is $\frac{\partial \mathcal{L}}{\partial W_i}(\Delta W_i)$. Alternatively, each operator can be written out as a (large) matrix and, ultimately, a gradient can be computed. The backward operator is intended to be evaluated from right to left inside the braces. Doing so computes the gradient directly.

We hope the reader appreciates the power of the "$T_{\bullet}$" notation, whereby one feels we are taking transposes of matrices and reversing order, but in fact we are transposing the operators. Either way the operators can be read right off the graphs.

5.5. Relationship to the adjoint method of scientific computing. We will show how to derive (3.8) and (3.7) using the adjoint method so-named because of its focus on the transpose (the adjoint) of the Jacobian. We encourage interested readers to see [9] and [4] to learn more about adjoint methods in numerical computation.

We find it satisfying that the graph theoretic interpretation of backward mode AD and the adjoint method of scientific computing yield the same answer from two very different viewpoints.

Consider a general computation with known constant input $x_0 \in \mathbb{R}$ and parameters $p = [p_1, \ldots, p_k]$.

Algorithm 5.3 is an explicit computation. The *function* ϕ_i computes the value of the *variable* x_i. The notation $\frac{d\Phi_i}{dx_j}$ or $\frac{d\Phi_i}{dp_j}$ gives the partial derivatives of one step of the algorithm. By contrast, the notation $\frac{dx_i}{dp_j}$ gives the partial derivatives across multiple steps of the algorithm. Algorithm 5.3 is the general case of Algorithm 3.1, the $\frac{d\Phi_i}{dx_j}$ and $\frac{d\Phi_i}{dp_j}$ are general cases of what is seen in Algorithm 3.2, and the $\frac{dx_i}{dp_j}$ generalize what is seen in Algorithm 3.3.

Algorithm 5.3. General Computation.

1: Input constant x_0
2: Input parameters $p_1, \ldots, p_k$
3: $x_1 \leftarrow \Phi_1(; p_1, \ldots, p_k; x_0)$
4: $x_2 \leftarrow \Phi_2(x_1; p_1, \ldots, p_k; x_0)$
5:          $\vdots$ $\vdots$
6: $x_N \leftarrow \Phi_N(x_1, \ldots, x_{N-1}; p_1, \ldots, p_k; x_0)$
7: Output x_N

We note that the adjoint method literature tends to consider a yet more general implicit approach. Placing section 3 of [9] in an explicit setting, we define a function f such that $f(x, p) = 0$. To this end, let

$$
(5.3) \qquad f(x, p) = x - \Phi(x, p) := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} - \begin{pmatrix} \Phi_1(; p; x_0) \\ \Phi_2(x_1; p; x_0) \\ \vdots \\ \Phi_N(x_1, \ldots, x_{N-1}; p; x_0) \end{pmatrix}.
$$

Clearly, given p, the computed $x = (x_1, \ldots, x_N)$ from Algorithm 5.3 is a solution to $f(x, p) = 0$. Our goal is to reproduce (3.8), which is the derivative of x_N w.r.t. to the parameter p.

Let us first consider the derivation for x_p, which is the derivative of x, implicitly defined by $f(x, p) = 0$, w.r.t. to p. To connect the viewpoints a table of notation for various Jacobians is helpful.

Adjoint method	Nabla notation	Matrix	Size
f_x	$\nabla_x f$	$I - \tilde{L}$	$N \times N$
f_p	$\nabla_p f$	$-M^T$	$N \times k$
x_p	$\nabla_p x$	$(I - \tilde{L})^{-1} M^T$	$N \times k$

The matrices themselves are explicitly

$$
\tilde{L} = \left[\frac{\partial \Phi_i}{\partial x_j} \right]_{i,j}, \quad i > j, \ j = 1, \ldots, N-1,
$$

and

$$
M^T = \left[\frac{\partial \Phi_i}{\partial p_j} \right]_{i,j}, \quad \nabla_p x = \left[\frac{\partial x_i}{\partial p_j} \right]_{i,j}, \quad i \in 1, \ldots, N, j \in 1, \ldots, k.
$$

The matrix $\tilde{L}$ that contains the partials $\partial \Phi_j / \partial x_j$ is strictly lower triangular exactly because Algorithm 5.3 is an explicit computation, whereas an implicit function would generally have a dense Jacobian. Since $f(x, p) = x - \Phi(x, p)$, the Jacobian $\nabla_x f = I - \tilde{L}$. Differentiating $0 = f(x, p)$ with respect to p we get $0 = f_x x_p + f_p$ or $x_p = -f_x^{-1} f_p$ which is $(I - \tilde{L})^{-1} M^T$ in matrix notation explaining the bottom row of the above table.

If $g(x)$ is any scalar function of x, then the key adjoint equation is

$$
\nabla_p g = g_x x_p = -g_x f_x^{-1} f_p := -\lambda^T f_p,
$$

where λ satisfies the so-called adjoint equation $f_x^T \lambda = g_x^T$. Since g_x is a 1 by k vector, by computing the adjoint λ first, we reduce the computation of a matrix-matrix multiplication and a matrix-vector multiplication to two matrix-vector multiplications.

If we take $g(x) = x_N$ then $g_x = [0, \ldots, 0, 1]$. The gradient is then

$$\nabla_p g(x) = [0, \ldots, 0, 1](I - \tilde{L})^{-1} M^T,$$

achieving our goal of reproducing (3.7).

So much is happening here that it is worth repeating with other notation. We can use the Jacobian of f with respect to x and p to differentiate (5.3):

$$0 = \begin{pmatrix} dx_1 \\ dx_2 \\ \vdots \\ dx_N \end{pmatrix} - \begin{pmatrix} 0 & 0 & \cdots & 0 \\ \frac{\partial \Phi_2}{\partial x_1} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \frac{\partial \Phi_N}{\partial x_1} & \cdots & \frac{\partial \Phi_N}{\partial x_{N-1}} & 0 \end{pmatrix} \begin{pmatrix} dx_1 \\ dx_2 \\ \vdots \\ dx_N \end{pmatrix} - \begin{pmatrix} \frac{\partial \Phi_1}{\partial p_1} & \cdots & \frac{\partial \Phi_1}{\partial p_k} \\ \vdots & \vdots & \vdots \\ \frac{\partial \Phi_N}{\partial p_1} & \cdots & \frac{\partial \Phi_N}{\partial p_k} \end{pmatrix} \begin{pmatrix} dp_1 \\ dp_2 \\ \vdots \\ dp_k \end{pmatrix},$$

which can be solved to obtain

$$\begin{pmatrix} dx_1 \\ dx_2 \\ \vdots \\ dx_N \end{pmatrix} = \left(I - \begin{pmatrix} 0 & 0 & \cdots & 0 \\ \frac{\partial \Phi_2}{\partial x_1} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \frac{\partial \Phi_N}{\partial x_1} & \cdots & \frac{\partial \Phi_N}{\partial x_{N-1}} & 0 \end{pmatrix} \right)^{-1} \begin{pmatrix} \frac{\partial \Phi_1}{\partial p_1} & \cdots & \frac{\partial \Phi_1}{\partial p_k} \\ \vdots & \vdots & \vdots \\ \frac{\partial \Phi_N}{\partial p_1} & \cdots & \frac{\partial \Phi_N}{\partial p_k} \end{pmatrix} \begin{pmatrix} dp_1 \\ dp_2 \\ \vdots \\ dp_k \end{pmatrix}.$$

Some readers unfamiliar with the notation of differentials might prefer what amounts to a notational change, but avoids the notation of differentials:

$$\begin{pmatrix} \frac{\partial x_1}{\partial p_1} & \cdots & \frac{\partial x_1}{\partial p_k} \\ \vdots & \vdots & \vdots \\ \frac{\partial x_N}{\partial p_1} & \cdots & \frac{\partial x_N}{\partial p_k} \end{pmatrix} = \left(I - \begin{pmatrix} 0 & 0 & \cdots & 0 \\ \frac{\partial \Phi_2}{\partial x_1} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots \\ \frac{\partial \Phi_N}{\partial x_1} & \cdots & \frac{\partial \Phi_N}{\partial x_{N-1}} & 0 \end{pmatrix} \right)^{-1} \begin{pmatrix} \frac{\partial \Phi_1}{\partial p_1} & \cdots & \frac{\partial \Phi_1}{\partial p_k} \\ \vdots & \vdots & \vdots \\ \frac{\partial \Phi_N}{\partial p_1} & \cdots & \frac{\partial \Phi_N}{\partial p_k} \end{pmatrix}.$$

6. Julia, the power of language.

6.1. The challenge. This section provides a complete realization of the challenge described in the preface (section 1). The question we asked is whether we could bring to life the linear algebra mathematics expressed in

$$\nabla J = M^T((I - L)^T \backslash g)$$

by typing the command

```
∇J = M' * (ImL' \ g)
```

and computing the backpropagated gradient of a matrix neural network almost by magic?

We remark that it is common to see code in papers. Code can serve the purpose of specifying details, facilitating reproducibility, and verifiability. Code can also allow users to adapt methods to their own situations. In addition to all of the above, we have a further purpose. We believe the code example we provide shows the power, elegance, and utility of the Julia programming language in ways that may be difficult or impossible to imagine in other languages.

At the risk of showing the end of the code before the start, 63 lines of setup culminate in exactly what we wanted: code which looks just like the math of matrices with operators that correctly calculates the gradient fulfilling our title goal of backpropagating through back substitution with a backslash:

```
64
65    ## The diagonal matrix
66    M = Diagonal([ [ℋ(δ[i]) ∘ ℛ(X[i-1])  ℋ(δ[i])] for i=1:N])
67
68    ## The lower triangular matrix (I-L)
69    ImL = Bidiagonal([ℐ() for i in 1:N], -[ℋ(δ[i]) ∘ ℒ(Ws_and_bs[i][1]) for i=2:N] , :L)
70
71    ## gradient of the loss function
72    g = [ fill(𝒪,N-1) ; [ℓ′(X[N],y)] ]
73
74    ## The gradient
75    ∇J = M' * (ImL' \ g)
```

The first 28 lines elegantly set up the mathematics very much like a mathematician defining operators and algebraic axioms:

```
1     using LinearAlgebra
2
3     struct Operator  # Linear Matrix Operators from Matrices to Matrices (and the operator adjoint)
4         op
5         adj
6         sym
7     end
8
9     ## Operators
10    ℒ(A::Matrix)  = Operator(X->A*X    , X->A'*X, "ℒ$(size(A))"  )   # left multiply by A (X → AX)
11    ℛ(A::Matrix)  = Operator(X->X*A    , X->X*A', "ℛ$(size(A))")     # right multiply by A (X → XA)
12    ℋ(A::Matrix)  = Operator(X->X.*A   , X->X.*A, "ℋ$(size(A))")     # Hadamard product (elementwise product)
13    ℐ()  =              Operator(X->X    ,    X->X,    "I")      # identity operator
14    𝒪()  =              Operator(X->zero(X) , X->zero(X),"𝒪")# zero operator
15
16    import Base:  zero, show, adjoint, *, \, ∘, +, -
17    show(io::IO, M::Operator) = print(io, M.sym)  # pretty printing
18    zero(::Any) = 𝒪() # Let's make any undefined zero the 𝒪 operator
19    adjoint(A::Operator) = Operator(A.adj, A.op,  "("*A.sym*")'")
20    adjoint(B::Bidiagonal) = Bidiagonal(adjoint.(B.dv),adjoint.(B.ev),(B.uplo == 'U') ? :L : :U) # lower to upper
21    -(A::Operator) = Operator(X->-A.op(X), X->-A.adj(X),"-"*A.sym)
22    -(::typeof(𝒪), X::Matrix) = -X # 𝒪 - X should be -X
23    *(A::Operator, X::Matrix) = A.op(X)
24    \(ℐ::typeof(ℐ()), A::Matrix) = A
25    ∘(A::Operator, B::Operator) = Operator(A.op ∘ B.op, B.adj ∘ A.adj, A.sym*"∘"*B.sym)
26    # We need [A;B]*C to somehow magically be [AC;BC]
27    *(M::Adjoint{Operator, Matrix{Operator}},v::Array) = M .* [v]
28    +(A::Array,x::Number)=A.+x
```

Lines 10–14 above define matrix operators and their adjoints. Lines 16–28 define various math operations, such as the negative operator on line 21, or the composition of operators on line 25.

```
29
30    using OffsetArrays
31
32    h(x) =   exp(-x) # sample activation function
33    h'(x) = -exp(-x)
34
35    function neural_net(params,X₀;h=h,h'= h')
36        T = Matrix{Float64}
37        N = length(params)
38        X = OffsetArray(Vector{T}(undef,N+1),0:N)
39        Δ = Vector{T}(undef, N)
40        X[0] = X₀
41        W = first.(params)
42        B = last.(params)
43
44        for i=1:N
45            X[i] =  h.(W[i]*X[i-1] .+ B[i])
46            Δ[i] =  h'.(W[i]*X[i-1] .+ B[i])
47        end
48        X,Δ
49    end
50
51    n = [5,4,3,1]  ## this contains [n₀...n_N]
52    k = 3 # batchsize
53    N = length(n)-1 #should be positive
54    init(sizes...) = 0.01randn(sizes...)
55    Ws_and_bs =[ [init(n[i+1],n[i]) , init(n[i+1])]  for i=1:N]
56    X₀ = init(n[1],k)
57    y  = init(n[end],k); #  y is what we will compare X_N against
58    X,δ = neural_net(Ws_and_bs,X₀) # This has all the X's and δ's
59
60    ℓ(x,y) = sum(abs2,x-y)/2 #loss
61    ℓ'(x,y) = x.-y;
62
63
```

For completeness we list lines 29 through 63 which constitute the setup of a basic forward pass through a matrix neural net. We remark that lines 30 and 38 allow an index origin of 0. The readers are encouraged to try the code at https://github.com/alanedelman/BackBackBack.

6.2. Modern computer science meets linear algebra. The venerable position of numerical linear algebra libraries cannot be undersold. Years of rigorous mathematical and algorithmic research have culminated in the modern LAPACK library [1] which represents a crowning achievement of value to a huge number of users who call LAPACK perhaps from, for example, Julia, NumPy, or MATLAB. In most cases the users are unaware of the scientific bedrock of which they are beneficiaries.

Continuing this grand tradition, we wish to highlight some of the computer science innovations that allow for the code in section 6.1 to look so deceptively simple.

Generic programming or how can the backslash just work? We invite the reader to consider how the innocent backslash on line 75 of the code in section 6.1 could possibly perform a backpropogation of a derivative. We believe this would be impossible in, for example, NumPy or MATLAB as these packages currently exist. From a computer science point of view, Julia's multiple dispatch mechanism and generic programming features allow the generic backslash to work with matrices and vectors whose elements are operators and compositions of operators. We remind the reader that the operators are not language constructs, but are created in software on the first 28 lines of code. The backslash, however, is not the LAPACK backslash, as the LAPACK library is constrained to floating point real and complex numbers.

TABLE 3

Algebraic structure for an MLP when the parameters (i.e., the set of leaf nodes collectively referred to as p) are (i) *only scalar weights,* (ii) *a weight/bias vector, and* (iii) *a vector of weight/bias matrices. We emphasize the common algebraic structure and the benefit of software that can represent matrices of vectors and matrices of operators. (Note: color appears only in the online article.)*
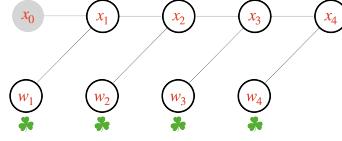
Gradient w.r.t. parameters p (leaf nodes)	$\nabla_p \mathcal{L} =$ M^T	$\times$	$(I-L)^{-1}$	$\times$ g
	$= \begin{pmatrix} m_1 & & & \\ & m_2 & & \\ & & m_3 & \\ & & & m_4 \end{pmatrix}^T$	$\times$	$\begin{pmatrix} I & & & \\ -l_2 & I & & \\ & -l_3 & I & \\ & & -l_4 & I \end{pmatrix}^{-T}$	$\times$ $\begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ g_4 \end{pmatrix}$
(i) Scalar $p = \{w_i\}$	$m_i = \delta_i x_{i-1}$		$l_i = \delta_i w_i$	$g_4 = \mathcal{L}'(x_4)$
(ii) Vector $p = \{[w_i, b_i]\}$	$m_i = [\delta_i x_{i-1} \ \ \delta_i]$		" "	" "
(iii) Matrices $p = \{[W_i, B_i]\}$	$m_i = [\Delta_{iH} \circ X_{i-1R} \ \ \Delta_{iH}]$		$l_i = \Delta_{iH} \circ W_{iL}$	$g_4 = \nabla_{X_4} \mathcal{L}$
	$\llcorner$ Operators $\lrcorner$			

The Julia backslash currently runs LAPACK when dispatched by matrices of floats, but, as is the case here, the generic algorithm is called. We are fascinated by the fact that the author of the generic algorithm would not have imagined how it might be used. We are aware of backslash being run on quaternion matrices, block matrices, matrices over finite fields, and now matrices with operators. Such is the mathematical power of abstraction and what becomes possible if software is allowed to be generic. In the context of backpropagation, replacing the "for loops" with the backslash helps us see backpropogation from a higher viewpoint.

The significance of transpose all the way down. Not without controversy, Julia implements transpose recursively. We believe this is the preferred behavior. This means a block matrix of block matrices of matrices (etc.) will transpose in the expected manner. Similarly matrices of complex numbers or quaternions will perform conjugate transposes as expected. In this work the M as seen in line 66 of the code in section 6.1 is diagonal, but is not symmetric. In line 75 we are transposing a diagonal matrix of 1×2 matrices of composed operators `M'` while in that same line we are also transposing a bidiagonal matrix of operators. Because the operator adjoint is defined on lines 10–14 of the code and the adjoint for a composed operator is defined on line 25, Julia's generic implementation, again, just works. We are not aware of any other linear algebra system whereby the transpose would just work this readily. The page

`https://discourse.julialang.org/t/why-is-transpose-recursive/2550` documents some of the controversy. We are extremely grateful that the recursive definition won the day.

A quick word about performance. There is nothing in the backslash formulation that would impede performance.

Possible extensions to the example code in section 6.1. We deliberately only used as an example the matrix neural network. We also have implemented a fully connected neural network where the matrix $I - L$ is a Julia triangular type, whereas the reference example was bidiagonal. We also implemented a square case where the W parameter was constant from one iteration to the next. We also conceived of the case of being restricted to a manifold. We thus stress that we did not build a fully functional package at this time, and thus emphasize that this could be future research, but we have not yet seen any roadblock to this methodology.

Concluding moral. Exciting new innovations in numerical algorithms are emerging from software developments. Critical elements for creativity include generic programming (generic operators), abstract representations, fast performance without waste, multiple dispatch, and an aggressive type system.

Abstraction matters. Software matters. Language matters.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, 1999.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: A survey*, J. Mach. Learn. Res., 18 (2018), pp. 1–43, http://jmlr.org/papers/v18/17-468.html.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM Rev., 59 (2017), pp. 65–98.

[4] A. M. Bradley, *PDE-Constrained Optimization and the Adjoint Method*, https://cs.stanford.edu/~ambrad/adjoint_tutorial.pdf (2010).

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, MA, 2016, http://www.deeplearningbook.org.

[6] A. Griewank, *A mathematical view of automatic differentiation*, Acta Numer., 12 (2003), pp. 321–398.

[7] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2008.

[8] M. Innes, A. Edelman, K. Fischer, C. Rackauckus, E. Saba, V. Shah, and W. Tebbutt, *∂P: A Differentiable Programming System to Bridge Machine Learning and Scientific Computing*, preprint, https://arxiv.org/abs/1907.07587, 2019.

[9] S. G. Johnson, *Notes on Adjoint Methods for* 18.335, https://math.mit.edu/~stevenj/18.336/adjoint.pdf (2006).

[10] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, Software Environ. Tools 22, SIAM, Philadelphia, 2011.

[11] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, *Differentiable programming for image processing and deep learning in halide*, ACM Trans. Graph., 37 (2018), pp. 1–13.

[12] J. Revels, T. Besard, V. Churavy, B. D. Sutter, and J. P. Vielma, *Dynamic Automatic Differentiation of GPU Broadcast Kernels*, https://arxiv.org/abs/1810.08297, 2018.

[13] J. Revels, M. Lubin, and T. Papamarkou, *Forward-Mode Automatic Differentiation in Julia*, preprint, arXiv:1607.07892, 2016.

[14] R. P. Stanley, *Catalan Numbers*, Cambridge University Press, Cambridge, 2015.