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:

1. It is of intellectual value to replace traditional treatments of automatic differentiation with a (left acting) operator theoretic, graph-based approach.
2. Operators can be readily placed in matrices in software in programming languages such as Julia as an implementation option.
3. We introduce a novel notation, "transpose dot" operator "$\{\}^{T\bullet}$" that allows the reversal of operators.

We demonstrate the elegance of the operators approach in a suitable programming language consisting of generic linear algebra operators such as Julia [3], 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, and without rewriting any code, the software carries through to completion giving the correct answer.

1. Preface: Summary and the Challenge. This paper provides the mathematics to show how an operator theoretic, graph-based approach can realize backpropogation using backsubstitution of a matrix whose elements are operators.

As a showecase result, one can backpropogate to compute the gradient on feed-forward neural networks with

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

We then set up a challenge to ourselves. Could we vivify the linear algebra math expressed in the above equation by simply typing the command (after basic setup)

```
∇J = M' * (ImL' \ g)
```

?

We demonstrate that indeed the backpropogated gradient can be computed, almost by magic, in a programming language that allows for generic programming as 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, Autodiff, 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 reverse modes [2, 10]. 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 [5, 8]. Who would have thought that one of the most routine college course material would now be the subject of much renewed interest both in applied mathematics and computer science?

*Department of Mathematics and CSAIL, MIT, Cambridge, MA (edelman@mit.edu).
†Department of EECS and CSAIL, MIT, Cambridge, MA (akyurek@mit.edu)
‡AWS AI Labs, Santa Clara, CA (yuyawang@amazon.com). Work done prior joining Amazon.

This paper introduces the notion that AD is best understood with a matrix based approach. The chain-rule explanation takes on the role of a distraction or 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 to implementation 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 for software. 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. For example, we theoretically connect backpropagation to the back substitution method for triangular systems of equations. Similarly, forward substitution corresponds to forward mode calculation of automatic differentiation. As is well documented in the preface to the book Graph Algorithms in the Language of Linear Algebra [7] there have been many known benefits to formulate mathematically a graph algorithm in linear algebraic terms. One of the known benefits is cognitive complexity.

The ability to implement these abstractions and retain performance is demonstrated in 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 simple 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 vs backwards (reverse) mode in AutoDiff 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 Graph. Consider a directed acyclic graph (DAG) with edge weights as in Figure 3.1 where nodes 1 and 2 are sources (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

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

where L^T 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 $(I - L^T)^{-1} = I + L^T + \ldots + (L^T)^{n-1}$ then counts path weights of all lengths.

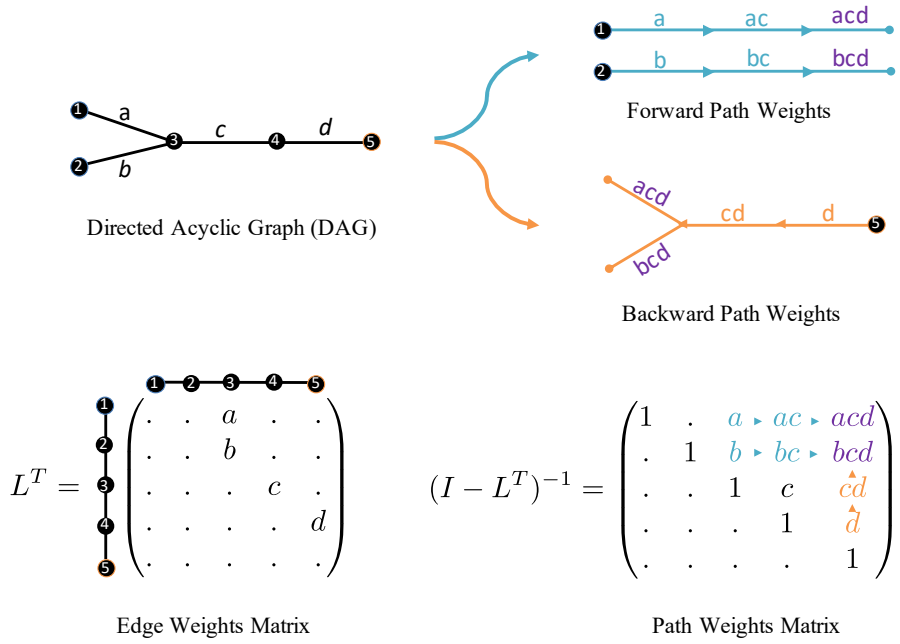

Fig. 3.1: (Upper Left:) Multiply the weights along the paths from source node 1 to sink node 5 and also source node 2 to sink node 5 to obtain acd and bcd. (Right Blue:) The obvious forward method. (Right Orange:) A backward method that requires one fewer multiplication. (Below:) A matrix method: if L^T_{ij} = the weight on edge ij, then $(I - L^T)^{-1}$ simultaneously exhibits the forward and backward methods.
Color Coding : Purple: target weights, Blue:forward computation, Orange: backwards computation.

Moreover, 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 exactly the same algorithms on the graph which we summarize in Figure 3.2.

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 approach, the backwards ap-

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 approaches compared: Both are seen equivalently as a choice of parenthesizing Equation 3.1 or as forward substitution vs. 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.

proach, the mixed-modes approaches, 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 [11], that there are a Catalan number, $C_5 = 42$, ways to parenthesize the above expression. One of which evaluates left to right; this is forward substitution which computes the graph weights forward. Another evaluates from right to left, backward substitution. There are three other "mixed-modes" [9] 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.

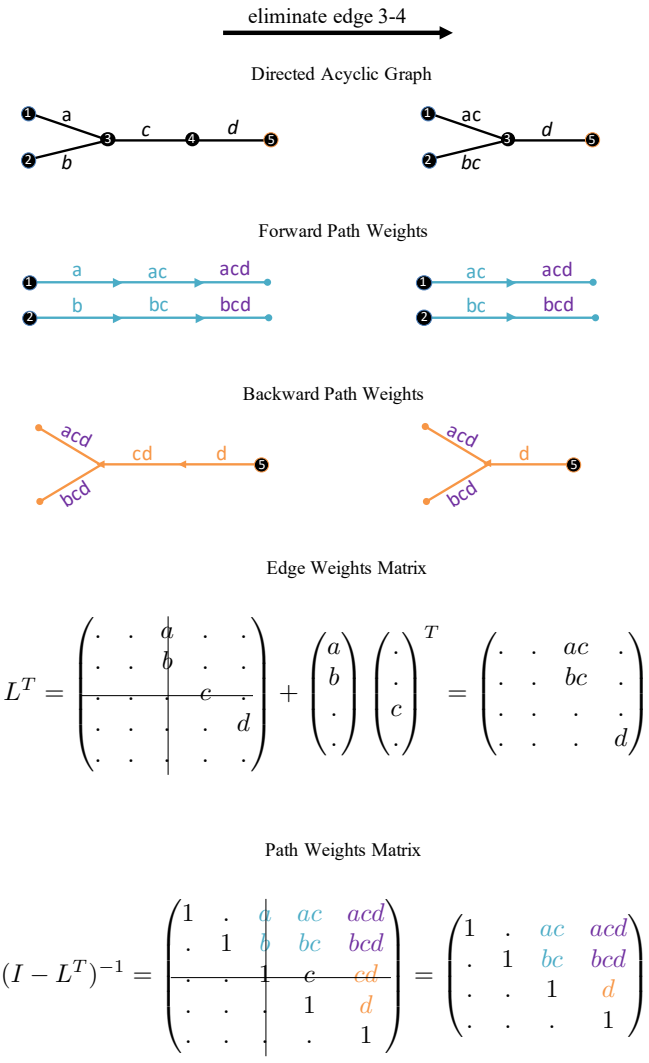

Fig. 3.3: Elimination of the edge from node 3 to node 4 on the graphs and with matrices. The matrix versions involve a rank one update in the case of the Edge Weights Matrix and a deletion of a row and column in the Path Weights Matrix.

3.1.4. Edge addition at the Sink Node. For reasons that shall become clear when we introduce loss functions, 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 add an edge "e":

$$\text{original path weights} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T}_{\text{sources}} \underbrace{(I - L^T)^{-1}}_{\text{path weights matrix}} \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{sink}} \tag{3.2}$$

which updates by augmenting by one node to become

$$\text{updated path weights} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}^T}_{\text{sources}} \underbrace{\begin{pmatrix} (I - L^T)^{-1} & . \\ . & 1 \end{pmatrix} \begin{pmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & e \\ . & . & . & . & . & 1 \end{pmatrix}}_{\text{updated path weights matrix}} \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}}_{\text{sink}}. \tag{3.3}$$

The update from the path weights matrix in Equation 3.2 to the updated path weights matrix in Equation 3.3 can probably be verified in many ways. One simple way is to look at the explicit elements of the path weights matrix before and after noticing that the new matrix has a column with one more factor of e augmented with a 1. Another way, is to update the edge weights matrix L and compute the block inverse.

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

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

3.2. Examples of DAGs and Weighted Paths.

3.2.1. The "Complete DAG" and Weighted Paths. Consider as an example (Figure 3.4) the complete DAG on four nodes with graph weights evaluated through the forward and reverse mechanism. 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 mode folds in the edges labelled "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.

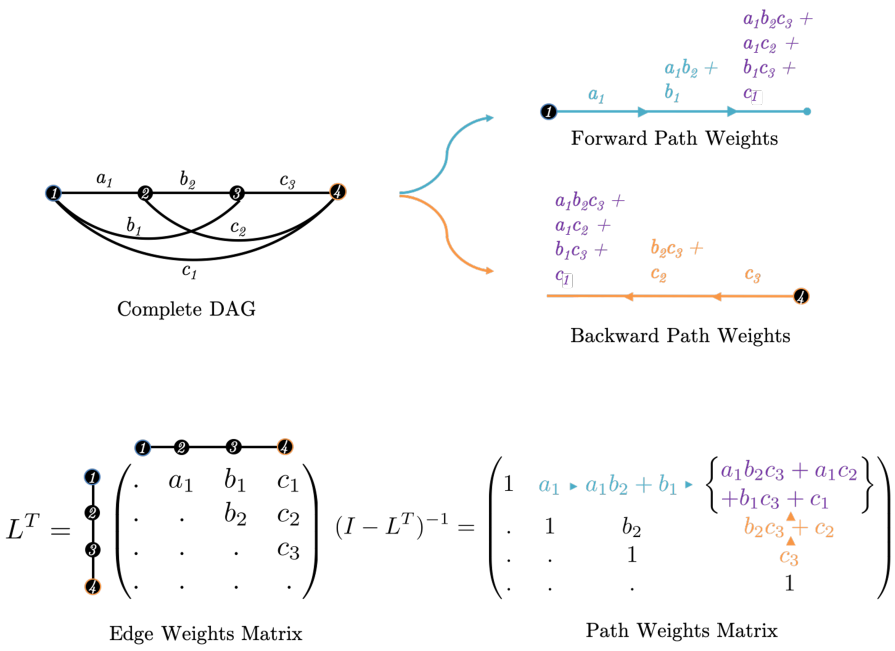

Fig. 3.4: The complete DAG on four nodes illustrates a symmetric situation where forward and reverse have the same complexity but arrive at the same answer through different operations.

3.2.2. The "multi-layer perceptron DAG" and Weighted Paths. Figure 3.5 is the DAG for the derivatives in a deep neural network. 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

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

and therefore

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

Notice that the matrix M corresponds to connections between the sources and internal nodes. In this example M 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 $5, 6, 7, 8$.

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

$$\text{Path weights} = M(I - \tilde{L}^T)^{-1} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}, \tag{3.6}$$

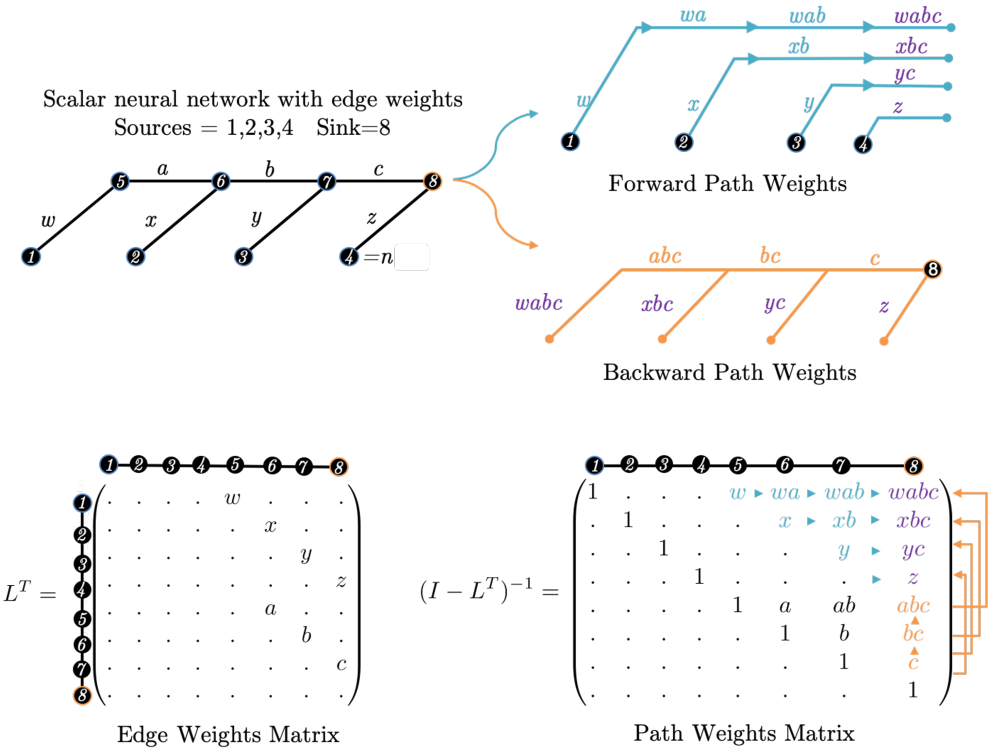

Fig. 3.5: This diagram contains most of what is needed to understand forward and reverse propagation of derivatives through a deep neural network. The details of what the weights look like will come later. If we take $n = 4$ for the pictured network, the sources are labeled $1 : (n + 1)$ and the sink is labeled $2n$. Forward mode requires $n(n-1)/2$ multiplications while reverse mode requires $2n - 3$ multiplications.

We can now take a close look at Figure 3.5 and fully grasp the path weight structure. The spine consisting of $a, b, c, 1$ requires the computation of the cumulative suffix product $1, c, bc, abc$. What follows is an element-wise multiplication by z, y, x, w, and

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

from which we can calculate the last column of $M(I - \tilde{L}^T)^{-1}$ or

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

3.3. Computational Graphs, Derivative Graphs, and their superposition. Many treatments of automatic differentiation 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

DEFINITION 1. *A computational graph is a node labelled DAG, where leaf nodes consist of variable names, and non-leaf 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.

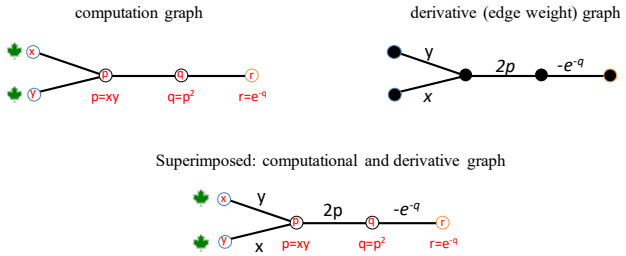

Fig. 3.6: Computational (node) graph, derivative (edge) graph, and their "superposition."

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.
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 ate 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's known as a loss function in modern applications.

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 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 p is exactly the path weight defined in (3.6),

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

That's it! Equation (3.8) is all that is about adjoint method (discussed in the subsequent section) in scientific computation and backpropagation in deep learning.

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

Algorithm 3.3 Path weights (Chained derivatives)

1: $dr/dx = y \times 2p \times (-e^{-q})$ (Chain lines 1,3, and 4 of Algorithm 2.2)
2: $dr/dy = x \times 2p \times (-e^{-q})$ (Chain lines 2,3, and 4 of Algorithm 2.2)

4. Key Idea: 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. This section will set the mathematical stage.

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

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 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 1 indicates some operators of interest.

Consider the inner product $\langle X, Y \rangle = \text{trace}(X^T Y)$. 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 left multiplication by A operator is left multiplication by A^T.

The operator transposes are

$(A_L)^T = (A^T)_L$
$(B_R)^T = (B^T)_R$
$(M_H)^T = M_H$ (symmetric)

	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	$G^{T\bullet}$	$X \mapsto \text{trace}(G^T X)$	$\text{vec}(G)^T$	$1 \times mn$

(We overload $A \otimes B$ to be both the operator and the matrix.)

Table 1: Matrix Operators and the size of their dense representations assuming $X : m \times n, \quad A : n_1 \times n, \quad B : m_1 \times m, \quad M : m \times n, \quad G : m \times n.$

DEFINITION 2. *We wish to propose a very carefully thought out notation for another useful operator, the matrix inner (or dot) product with G. We will denote $G^{T\bullet}$ ("G transpose dot.") This operator takes a matrix X of the same size as G and returns the scalar, $G^{T\bullet}X \equiv Tr(G^T X) = vec(G)^T vec(X) = \sum G_{ij} X_{ij}$.*

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 use 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 "eats" vectors and returns scalars. In the same way we propose that one might denote a linear function of a matrix $X \mapsto tr(G^T X)$ with the operator notation $X \mapsto G^{T\bullet} X$, an operator that "eats" matrices and returns scalars.

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,

$$(A_L^T G)^{T\bullet} = X \mapsto tr((A^T G)^T X)$$

and

$$G^{T\bullet} A_L = X \mapsto tr(G^T A X),$$

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.)

	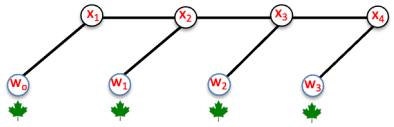
Gradient	$\nabla_p \mathcal{L} = \quad M^T \quad \times \quad (I-L)^{-1} \quad \times \quad g$ $= \begin{pmatrix} m_1 & & & \\ & m_2 & & \\ & & m_3 & \\ & & & m_4 \end{pmatrix}^T \times \begin{pmatrix} I & & & \\ -l_2 & I & & \\ & -l_3 & I & \\ & & -l_4 & I \end{pmatrix}^{-T} \times \begin{pmatrix} . \\ . \\ . \\ g_4 \end{pmatrix}$
(i) Scalar $p = w$	$m_i = \delta_i x_{i-1} \qquad l_i = \delta_i w_i \qquad g_4 = \mathcal{L}'(x_4)$
(ii) Vector $p = [w, b]$	$m_i = [\delta_i x_{i-1} \ \delta_i] \qquad$ '' '' $\qquad$ '' ''
(iii) Matrices $p = [W, B]$	$m_i = [\Delta_{iH} \circ X_{i-1R} \ \ \Delta_{iH}] \qquad l_i = \Delta_{iH} \circ W_{iL} \qquad g_4 = \nabla_{X_4} \mathcal{L}$ ↑ Operators ↑

Table 2: Algebraic Structure for a basic neural network when the parameters are (i) only scalar weights (ii) a weight/bias vector, (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.

5. Operator Methodology.

5.1. Matrices of scalars.

Algorithm 5.1 Scalar Neural Net Without Bias

input x_0
for i = 1 to N **do**
    input w_i
    $x_i \leftarrow h_i(w_i x_{i-1})$
    $(\delta_i \leftarrow h_i'(w_i x_{i-1}))$
end for
output x_N

The simple case of scalar neural networks 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.)

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

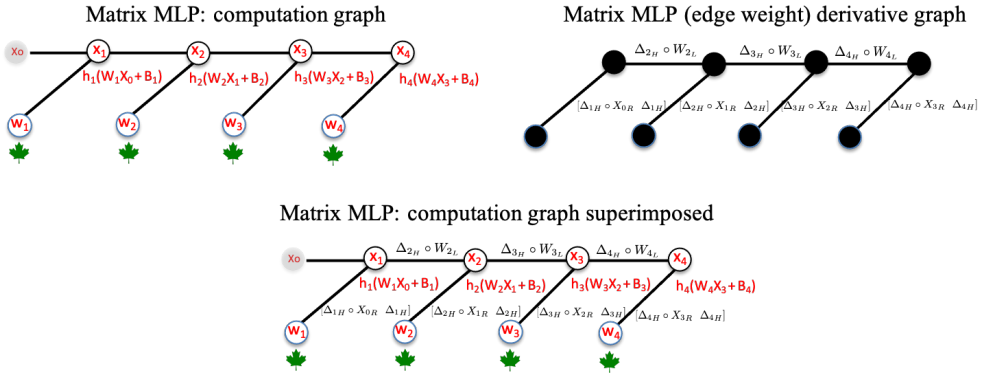

Fig. 5.1: Top left: computation graph of a scalar MLP. This computation, which has nothing to do with derivatives, is often referred to as forward propagation because of its direction. Evaluation must generally necessarily go from left to right. Top right: derivative edge weights. Since derivatives are linear, multiple directions are possible to evaluate the products. Bottom: the superimposed graph showing both the forward computation and backward derivative propagation.

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

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

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 and discuss its format.

$\nabla_{[w,b]} \mathcal{L} =$

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

We see we have an ordinary matrix backsubstitution 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.

5.3. Matrices of operators.
Letting $\mathcal{I}$ denote the identity operator and empty space the zero operator:
$\nabla_{[W,B]}\mathcal{L} =$

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

for the matrix neural network in Algorithm 5.2

Algorithm 5.2 Matrix Neural Network

1: input X_0 $(n_0 \times k)$
2: **for** i := 1 to N **do**
3: input $W_i(n_i \times n_{i-1}), B_i(n_i \times k)$
4: $X_i \leftarrow h_i(W_i * X_{i-1} + B_i)$.
5: $(\Delta_i \leftarrow h_i'(W_i * X_{i-1} + B_i))$
6: **end for**
7: $J \leftarrow \mathcal{L}(X_N, y)$

The entries of our matrix of operators may be read immediately from the differential of line 4 of Algorithm 5.2:

$$dX_i = (\Delta_{iH} \circ X_{i-1R})dW_i + \Delta_{iH}dB_i + (\Delta_{iH} \circ W_{iL})dX_{i-1}. \tag{5.1}$$

5.4. The Power of Notation. We read directly off the edge weight graph in Figure 5.2

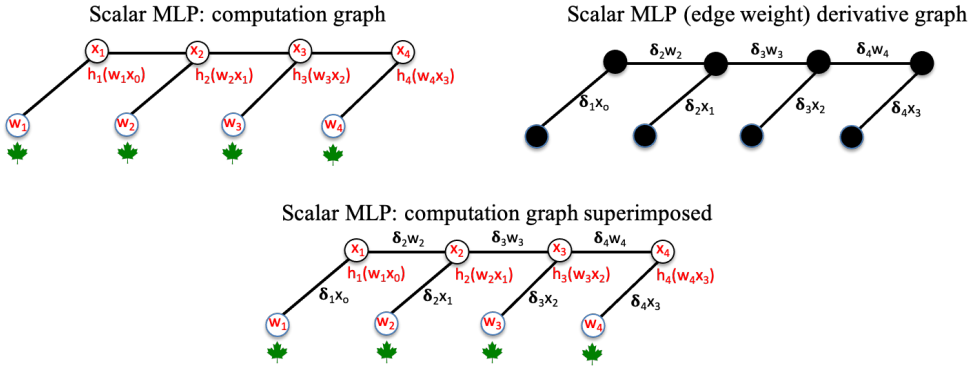

Fig. 5.2: Computation, derivative graph of a matrix MLP with their superimposed version. Comparing to Figure 5.1 of the scalar MLP, all remains the same except that the players in computational graph become matrices while the edges in the derivative graph being replaced by operators.

that for a matrix neural network we have

- FORWARD Mode Operators (right to left)

$$\frac{\partial \mathcal{L}}{\partial W_i} = G^{T_\bullet}(\Delta_N)_H(W_N)_L \dots (W_{i+2})_L(\Delta_{i+1})_H(W_{i+1})_L(\Delta_i)_H(X_{i-1})_R \tag{5.2}$$

$$\frac{\partial \mathcal{L}}{\partial B_i} = G^{T_\bullet}(\Delta_N)_H(W_N)_L \dots (W_{i+2})_L(\Delta_{i+1})_H(W_{i+1})_L(\Delta_i)_H \tag{5.3}$$

or going the other way we have

- REVERSE Mode Operators (right to left)

$$\left(\frac{\partial \mathcal{L}}{\partial W_i}\right)^{T_\bullet} = \left\{(X_{i-1}^T)_R(\Delta_i)_H(W_{i+1}^T)_L(\Delta_{i+1})_H(W_{i+2}^T)_L \dots (W_N^T)_L(\Delta_N)_H G\right\}^{T_\bullet} \tag{5.4}$$

$$\left(\frac{\partial \mathcal{L}}{\partial B_i}\right)^{T_\bullet} = \left\{(\Delta_i)_H(W_{i+1}^T)_L(\Delta_{i+1})_H(W_{i+2}^T)_L \dots (W_N^T)_L(\Delta_N)_H G\right\}^{T_\bullet} \tag{5.5}$$

Understanding these operators. The forward operators in Equations (5.2) and (5.3) 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 $\frac{\partial \mathcal{L}}{\partial W_i}(\Delta W_i)$. Alternative each operator can be written out as a (large) matrix, and ultimately a gradient can be computed.

The reverse 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 equations (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 [6] and [4].

We find it satisfying that the graph theoretic interpretation of reverse mode autodifferentiation and the adjoint method of scientific computing can be seen to give the same answer from two very different viewpoints.

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

Algorithm 5.3 General Computation

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

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 2.1, the $\frac{d\Phi_i}{dx_j}$ and $\frac{d\Phi_i}{dp_j}$ are general cases of what is seen in Algorithm 2.2, and the $\frac{dx_i}{dp_j}$ generalize what is seen in Algorithm 2.3.

We note that the adjoint method literature tends to consider a yet more general implicit approach. Placing Section 3 of [6] in an explicit setting, we define a function f such that $f(x,p)=0$. To this end let

$$f(x,p) = x - \Phi(x,p) := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} - \begin{pmatrix} \Phi_1(;p;x_0) \\ \Phi_2(x_1;p;x_0) \\ \vdots \\ \Phi_N(x_1,\ldots,x_{N-1};p;x_0) \end{pmatrix}. \tag{5.6}$$

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 Equation (3.8), which is the derivative of x_N wrt to the parameter p.

Let us first consider the derivation for x_p, which is the derivative of x, implicitly defined by $f(x,p)=0$, wrt 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 an 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 Equation (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 Equation 5.6:

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

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:

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

6. Julia, the power of language.

6.1. The challenge. Here is a complete realization of the challenge of the preface (Section 1). The question we asked is whether we could vivify the linear algebra math expressed in

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

by typing the command

```
∇J = M' * (ImL' \ g)
```

and compute the backpropogated gradient of a matrix neural network almost by magic?

We remark that it is common to see code or pointers to 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 backpropogating through back substitution with a backslash:

```
## The diagonal matrix
M = Diagonal([ [ℋ(δ[i]) ∘ ℛ(X[i-1])  ℋ(δ[i])] for i=1:N])

## The lower triangular matrix (I-L)
ImL = Bidiagonal([ℐ() for i in 1:N], -[ℋ(δ[i]) ∘ ℒ(Ws_and_bs[i][1]) for i=2:N] , :L)

g = [ fill(𝒪,N-1) ; [ℓ′(X[N],y)] ] # gradient of the loss function

∇J = M' * (ImL' \ g) # The gradient
```

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

```
using LinearAlgebra

struct Operator  # Linear Matrix Operators from Matrices to Matrices (and the operator adjoint)
    op
    adj
    sym
end

## Operators
ℒ(A::Matrix)  = Operator(X->A*X    , X->A'*X, "ℒ$(size(A))"  ) # left multiply by A (X → AX)
ℛ(A::Matrix)  = Operator(X->X*A    , X->X*A', "ℛ$(size(A))")   # right multiply by A (X → XA)
ℋ(A::Matrix)  = Operator(X->X.*A   , X->X.*A, "ℋ$(size(A))")   # Hadamard product
ℐ()  =          Operator(X->X      ,    X->X,    "I")         # identity operator
𝒪()  =          Operator(X->zero(X) , X->zero(X),"𝒪")         # zero operator

import Base:  zero, show, adjoint, *, \, ∘, +, -
show(io::IO, M::Operator) = print(io, M.sym)  # pretty printing
zero(::Any) = 𝒪() # Let's make any undefined zero the 𝒪 operator
adjoint(A::Operator) = Operator(A.adj, A.op,  "("*A.sym*")'")
adjoint(B::Bidiagonal) = Bidiagonal(adjoint.(B.dv),adjoint.(B.ev),(B.uplo == 'U') ? :L : :U)
-(A::Operator) = Operator(X->-A.op(X), X->-A.adj(X),"-"*A.sym)
-(::typeof(𝒪), X::Matrix) = -X # 𝒪 - X should be -X
*(A::Operator, X::Matrix) = A.op(X)
\(ℐ::typeof(ℐ()), A::Matrix) = A
∘(A::Operator, B::Operator) = Operator(A.op ∘ B.op, B.adj ∘ A.adj, A.sym*"∘"*B.sym)
# We need [A;B]*C to somehow magically be [AC;BC]
*(M::Adjoint{Operator, Matrix{Operator}},v::Array) = M .* [v]
+(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.

For completeness we list lines 29 through 63 which constitute the setup of a basic forward pass through a matrix neural net. We remark th at lines 30 and 38 allow an index origin of 0.

For reference, we present the matrices with operators below:

$$M = \begin{pmatrix} [\Delta_{1H} \circ X_{0R}\ \Delta_{1H}] & & \\ & [\Delta_{2H} \circ X_{1R}\ \Delta_{2H}] & \\ & & [\Delta_{3H} \circ X_{2R}\ \Delta_{3H}] \\ & & & [\Delta_{4H} \circ X_{3R}\ \Delta_{4H}] \end{pmatrix}$$

$$(I-L) = \begin{pmatrix} \mathcal{I} & & & \\ -\Delta_{2H} \circ W_{2L} & \mathcal{I} & & \\ & -\Delta_{3H} \circ W_{3L} & \mathcal{I} & \\ & & -\Delta_{4H} \circ W_{4L} & \mathcal{I} \end{pmatrix}$$

$$g = \begin{pmatrix} \cdot \\ \cdot \\ \cdot \\ \nabla_{X_4} \mathcal{L} \end{pmatrix}.$$

```
using OffsetArrays

h(x) =   exp(-x) # sample activation function
h'(x) = -exp(-x)

function neural_net(params,X₀;h=h,h'= h')
    T = Matrix{Float64}
    N = length(params)
    X = OffsetArray(Vector{T}(undef,N+1),0:N)
    Δ = Vector{T}(undef, N)
    X[0] = X₀
    W = first.(params)
    B = last.(params)

    for i=1:N
          X[i] =  h.(W[i]*X[i-1] .+ B[i])
          Δ[i] =  h'.(W[i]*X[i-1] .+ B[i])
    end
    X,Δ
end

n = [5,4,3,1]  # this contains [n₀...n_N]
k = 3 # batchsize
N = length(n)-1 #should be positive
init(sizes...) = 0.01randn(sizes...)
# The second part of the pair is a vector here:
Ws_and_bs =[ [init(n[i+1],n[i]) , init(n[i+1])]  for i=1:N]
X₀ = init(n[1],k)
y  =  init(n[end],k); #  y is what we will compare X_N against
X,δ = neural_net(Ws_and_bs,X₀) # This has all the X's and δ's

ℓ(x,y) = sum(abs2,x-y)/2 #loss
ℓ'(x,y) = x.-y;

X,δ = neural_net(Ws_and_bs,X₀) # Run the neural net
```

6.2. Modern Computer Science meets Linear Algebra. The venerable position of numerical linear algebra libraries can not be undersold. Years of rigorous mathematical and algorithmic research culminating in the modern LAPACK library [1] have lead to a crowning achievement of value to a huge number of users calling LAPACK perhaps from, for example, Julia, NumPy, or MATLAB in most cases 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 derivative. We believe this would be impossible in, say, 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 the software on the first 28 lines of code. The backslash, however, is not LAPACK's backlash, as the LAPACK library is constrained to floating point real and complex numbers. Julia's 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 back propagation, replacing the "for loops" with the backlash 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 number 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 1x2 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.

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

A quick word about 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.
`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.

Summary: For software to efficiently achieve the goals set out of this paper, modern elements not found in older languages that are important include:

- Generic Programming (generic operators)
- Abstract Representations
- Fast Performance without waste
- Multiple dispatch
- Aggressive type system

7. Acknowledgments. We wish to thank David Sanders and Jeremy Kepner for helpful conversations. This material is based upon work supported by the National Science Foundation under grant no. OAC-1835443, grant no. SII-2029670, grant no. ECCS-2029670, grant no. OAC-2103804, and grant no. PHY-2021825. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0001211 and DE-AR0001222. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Agree-

ment No HR00112290091. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This material was supported by The Research Council of Norway and Equinor ASA through Research Council project "308817 - Digital wells for optimal production and drainage". Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al., *LAPACK users' guide*, SIAM, 1999.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: a survey*, Journal of Machine Learning Research, 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 review, 59 (2017), pp. 65–98.

[4] A. M. Bradley, *PDE-constrained optimization and the adjoint method*, 2010, https://cs.stanford.edu/~ambrad/adjoint_tutorial.pdf.

[5] 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*, 2019, http://arxiv.org/abs/1907.07587.

[6] S. G. Johnson, *Notes on adjoint methods for 18.335*, 2006, https://math.mit.edu/~stevenj/18.336/adjoint.pdf.

[7] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*, SIAM, 2011.

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

[9] J. Revels, T. Besard, V. Churavy, B. D. Sutter, and J. P. Vielma, *Dynamic automatic differentiation of GPU broadcast kernels*, 2018, https://arxiv.org/abs/arXiv:1810.08297.

[10] J. Revels, M. Lubin, and T. Papamarkou, *Forward-mode automatic differentiation in Julia*, arXiv preprint arXiv:1607.07892, (2016).

[11] R. P. Stanley, *Catalan numbers*, Cambridge University Press, 2015.