Backpropagation through Back substitution with a Backslash

Ekin Akyurek, Alan Edelman, Bernie Wang

MIT, AWS AI Labs

Abstract and Contributions

- This paper introduces the notion that AutoDiff (AD) is best understood with a matrix based approach. We argue that a linear algebra based framework for AD, while mathematically equivalent to other approaches, provides a simplicity of understanding, and an approach towards implementation.
- In particular we show that with the correct abstractions, backpropagation can be viewed simply as a "backslash" operation on triangular matrices, i.e. backpropagation equals backward substitution. The gradient of interests dp can be calculated by solving the following linear system

$$dx = (I - L)^{-1} D dp,$$

where D denote the diagonal matrix, L the lower triangular matrix, dx the vector of intermediate differentials, and dp the vector of parameter differentials.
- By extending the matrix above to linear operators, one could readily generalize the framework to other matrix manifolds without the need to explicitly changing the AutoDiff algorithm.
- Julia enables the implementation of these abstractions while retaining performance through its facility for multiple dispatch and its generic operators.

AutoDiff with Backslash

Consider a directed acyclic graph (DAG) with edge weights as in Figure 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.

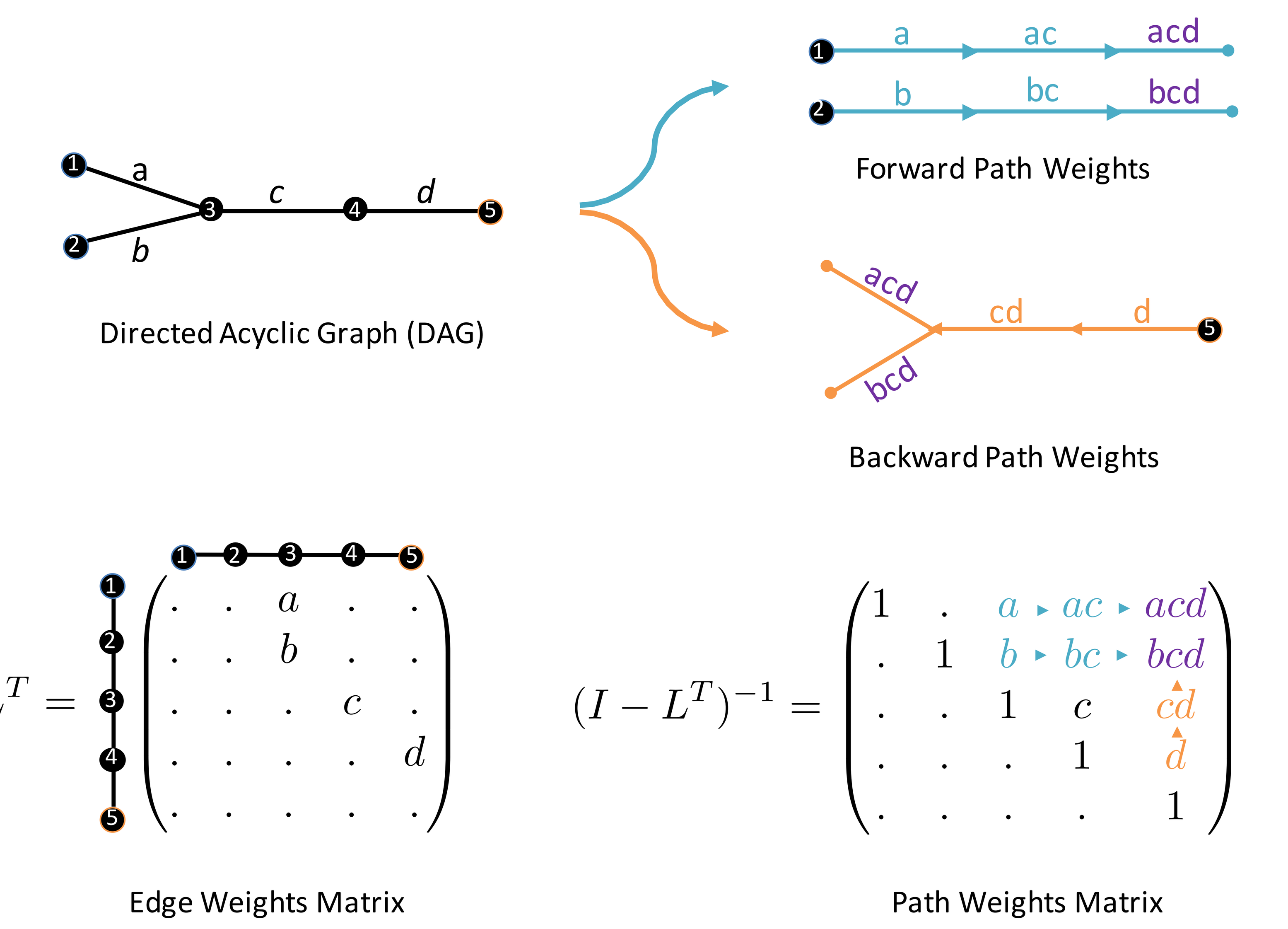

Figure 1: (Upper Left:) Multiply the weights along the paths from source node 1 to sink node 5 and also source node 2 to sink node 5 to obtain acd and bcd. (Right Blue:) The obvious forward method. (Right Orange:) A backward method that requires one fewer multiplication. (Below:) A matrix method: if 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.

Evidently, the path weights that we seek in Figure 1 may be obtained by calculating

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

where L^T is displayed in the lower left of Figure 1.

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

The forward and backward approaches are seen equivalently as a choice of parenthesizing Equation 1 or as forward substitution or back substitution.

Scalar MLP as an Example

We start with the simple iterative equation of a scalar MLP

$$x_i \leftarrow h_i(w_i x_{i-1} + b_i), \quad i = 1, \cdots, N, \tag{2}$$

Denoting $\delta_i := h_i'(w_i x_{i-1} + b_i)$,

$$dx_i = \delta_i (x_{i-1} dw_i + w_i dx_{i-1} + db_i), \quad i = 1, \cdots, N.$$

$$\begin{pmatrix} dx_1 \\ dx_2 \\ \vdots \\ dx_{N-1} \\ dx_N \end{pmatrix} = \overbrace{\begin{pmatrix} [\delta_1 x_0 \ \delta_1] & & & & \\ & [\delta_2 x_1 \ \delta_2] & & & \\ & & \ddots & & \\ & & & [\delta_{N-1} x_{N-2} \ \delta_{N-1}] & \\ & & & & [\delta_N x_{N-1} \ \delta_N] \end{pmatrix}}^{D} \overbrace{\begin{pmatrix} [dw_1; db_1] \\ [dw_2; db_2] \\ \vdots \\ [dw_{N-1}; db_{N-1}] \\ [dw_N; db_N] \end{pmatrix}}^{dp} + \underbrace{\begin{pmatrix} 0 & \dots & 0 & 0 & 0 \\ \delta_2 w_2 & \dots & 0 & 0 & 0 \\ & \ddots & & & \\ & & \delta_{N-1} w_{N-1} & 0 & 0 \\ & & & \delta_N w_N & 0 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} dx_1 \\ dx_2 \\ \vdots \\ dx_{N-1} \\ dx_N \end{pmatrix}}_{dx}. \tag{3}$$

Letting D denote the diagonal matrix, L the lower triangular matrix, dx the vector of intermediate differentials, and dp the vector of parameter differentials,

$$dx = Ddp + Ldx$$

and thus we can solve for dx:

$$dx = (I - L)^{-1} D dp.$$

AutoDiff for MLP: Backslash with Linear Operators

- Input and parameters: X_0 $(n_0 \times k)$, $W_i(n_i \times n_{i-1})$, $B_i(n_i \times k)$
- Feedforward relations

$$X_i \leftarrow h_i(W_i X_{i-1} + B_i)$$
$$\Delta_i \leftarrow h_i'(W_i X_{i-1} + B_i)$$

$$dX_i = \Delta_i \odot (W_i dX_{i-1} + dW_i X_{i-1} + dB_i), \quad i = 1, \dots, N. \tag{4}$$

where $\odot$ denotes element-wise multiplication.

Description	Notation	Mapping		Size
Kronecker Product of A, B	$A \otimes B$	$X \mapsto BXA^T$		$m_1 n_1 \times mn$
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	$X \mapsto \text{trace}(G^T X)$	$\text{vec}(G)^T \text{vec}(X)$	scalar

$$\begin{pmatrix} dX_1 \\ dX_3 \\ \vdots \\ dX_{N-1} \\ dX_N \end{pmatrix} = \begin{pmatrix} [(\Delta_{1_H} X_{0_R}) \ \Delta_{1_H}] & & & & \\ & [(\Delta_{2_H} X_{1_R}) \ \Delta_{2_H}] & & & \\ & & \ddots & & \\ & & & [(\Delta_{N-1_H} X_{N-2_R}) \ \Delta_{N-1_H}] & \\ & & & & [(\Delta_{N_H} X_{N-1_R}) \ \Delta_{N_H}] \end{pmatrix} \begin{pmatrix} [dW_1; dB_1] \\ [dW_2; dB_2] \\ \vdots \\ [dW_{N-1}; dB_{N-1}] \\ [dW_N; dB_N] \end{pmatrix} + \begin{pmatrix} 0 & \dots & 0 & 0 & 0 \\ (\Delta_{2_H} W_{2_L}) & \dots & 0 & 0 & 0 \\ & \ddots & & & \\ & & (\Delta_{N-1_H} W_{N-1_L}) & 0 & 0 \\ & & & (\Delta_{N_H} W_{N_L}) & 0 \end{pmatrix} \begin{pmatrix} dX_1 \\ dX_2 \\ \vdots \\ dX_{N-1} \\ dX_N \end{pmatrix}$$

The form of the linear system stays the same with linear operators instead of matrices.

Julia: the power of language

A natural question to ask is why we go through a seemingly complicated framework to compute the gradient. The answer is that it allows us to generalize immediately to other matrix manifolds, and all one needs to do is define the necessary linear operators, and invoke the backslash. 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.

```julia
## 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 (elementwise product)
𝒥()  =         Operator(X->X,    X->X,    "I")      # identity operator
𝒪()  =         Operator(X->zero(X), X->zero(X),"𝒪")# zero operator
```

With the operator defined as above, all we need is the following piece of code.

```julia
X,δ = neural_net(Ws_and_bs,X₀) # This has all the X's and δ's

## The diagonal matrix
D = Diagonal([ [ℋ(δ[i]) ∘ ℛ(X[i-1])  ℋ(δ[i]) ∘ ℛ(ones(1,k))] for i=1:N])

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

## gradient of the loss function
g = [ fill(𝒪,N-1) ; [ℓ'(X[N],y)] ]
∇J = D' * (ImL'\g)
```