

Batched Backpropagation as a Linear Solve

Mikkel Paltor (mpasc@dtu.dk), 7 October 2026

Department of Applied Mathematics and Computer Science, Technical University of Denmark

This post extends Backpropagation as a Linear Solve¹ to batches of inputs. The main change is that the (sparse) matrices of scalars from the previous post become (sparse) matrices whose entries are linear maps.² To readers used to the usual presentation of backpropagation as a pass over a computational graph, matrices may seem an odd choice. The two views are the same: the sparsity pattern below the diagonal of a lower triangular matrix $\mathcal{L}$ is a Directed Acyclic Graph (DAG).

The post ends with a Julia implementation of a neural network whose gradient is $\nabla_{\theta}\mathcal{J} = \mathcal{M}((\mathcal{L}^*)^{-1}\nabla_U\ell) + \nabla_{\theta}\ell$, the same formula as before, written as a linear solve in the style of Edelman, Akyürek, and Wang [1]. The inverse is never formed: because $\mathcal{L}$ is unit lower triangular, $\mathcal{L}^*$ is unit upper triangular, and its inverse can be applied efficiently using back substitution.

1. The batched forward pass

The previous post considered a single input u_0 and a loss $\ell(u_P)$ of its outputs. However, the loss function is often based on minimizing a data-fitting term of a *minibatch* of m input-output pairs $(u_0^{(s)}, y^{(s)})$. Due to the linearity of a sum the full gradient over the m input-output data could be computed by simply summing the gradient contribution for each example. In practice, it is often more efficient to process the data together in a batch.³ The first step is to introduce the batched input and output matrices by stacking the data points as columns, i.e.

$$U_0 = \begin{pmatrix} u_0^{(1)} & \dots & u_0^{(m)} \end{pmatrix}, \quad Y = \begin{pmatrix} y^{(1)} & \dots & y^{(m)} \end{pmatrix}. \quad (1)$$

The only difference between this and the single input case is that the application of layer i now generates m outputs. Writing the outputs with a capital U , as for the input data, layer i gives

$$U_i = \Phi_i(U_{i-1}, \dots, U_0, \theta). \quad (2)$$

We denote the collection of all the states by $U(\theta)$ and the collection of all the layer maps by $\Phi(U, \theta)$. One can think of them as *vectors* whose entries are the states and the layer maps, respectively, so that

$$\begin{aligned} U(\theta) &= (U_1, U_2, \dots, U_N), \\ \Phi(U, \theta) &= (\Phi_1(U_0, \theta), \Phi_2(U_1, U_0, \theta), \dots, \Phi_N(U_{N-1}, \dots, U_0, \theta)). \end{aligned} \quad (3)$$

However, this does not have to be the case, and the only thing we require is that $U(\theta)$ is an element of a vector space $\mathcal{U}$ with an inner product. Similarly, the parameters θ live in a vector space Θ . The next section makes this precise.⁴

2. Generalization to vector spaces

It is standard to equip the space that each layer output lives in with the Frobenius inner product. This means that for $Z, E \in \mathcal{U}$ the inner product is a sum of Frobenius inner products over the layers, i.e.

Contents

1. The batched forward pass
2. Generalization to vector spaces
 - 2.1. The loss
 - 2.2. Adjoint
 - 2.3. Gradients
3. The batched gradient
 - 3.1. The forward relation
 - 3.2. The adjoint equation
 - 3.3. The operators as block (sparse) matrices
4. A code example
5. Conclusion
- Bibliography

¹ Earlier post: *Backpropagation as a Linear Solve*.
mipals.github.io/blog/2026/10-05-backpropagation-linear-solve/

² More often than not the matrices are *sparse*—which is key for the efficiency of automatic differentiation.

³ Batching turns matrix–vector products into matrix–matrix products, which use modern hardware far better. Moreover, because the weights are shared by all the data points, they only have to be loaded into memory once for a batch, whereas processing each example separately would load them once per example.

⁴ See Wikipedia for what is meant by a vector space.

$$\langle Z, E \rangle_{\mathcal{U}} = \sum_{i=1}^N \langle Z_i, E_i \rangle_F = \sum_{i=1}^N \text{tr}(Z_i^\top E_i). \quad (4)$$

The construction of the parameter space is slightly more complicated, as parameters can be shared between layers.⁵ The parameters can still be thought of as a vector whose entries are *parameter blocks*, that is, $\theta = (\theta_k)_{k=1}^K \in \Theta$. Layer i uses only some of the blocks, and several layers may use the same block. A linear selection map $\mathcal{S}_i : \Theta \rightarrow \Theta_{\mathcal{S}_i}$ picks out the blocks of layer i , so that Φ_i depends on θ only through $\theta_{\mathcal{S}_i} = \mathcal{S}_i(\theta)$. In a plain feed-forward network every layer has its own weight block $W_{i,i-1}$ and bias block b_i , and $\mathcal{S}_i(\theta) = (W_{i,i-1}, b_i)$. Similar to $\mathcal{U}$, we equip Θ with the Frobenius inner product block by block, so that for $h, g \in \Theta$ it follows that

$$\langle h, g \rangle_{\Theta} = \sum_{k=1}^K \langle h_k, g_k \rangle_F = \sum_{k=1}^K \text{tr}(h_k^\top g_k). \quad (5)$$

Lastly, we also aim to equip the output space $\mathcal{Y}$ with an inner product. We view elements in $\mathcal{Y}$ as a list of outputs, $A = (A_1, \dots, A_r)$, where again the inner product is a sum of Frobenius inner products over the list,

$$\langle A, B \rangle_{\mathcal{Y}} = \sum_{a=1}^r \langle A_a, B_a \rangle_F = \sum_{a=1}^r \text{tr}(A_a^\top B_a). \quad (6)$$

We then define a linear output map $\mathcal{P} : \mathcal{U} \rightarrow \mathcal{Y}$ such that it maps the full state $U \in \mathcal{U}$ to a corresponding output. That is $\mathcal{P}(U) \in \mathcal{Y}$ and we sometimes simply write $U_{\mathcal{P}} = \mathcal{P}(U)$.⁶

2.1. The loss

We often describe the loss function as a function of the parameters θ only, but it is more natural to view it as a function of the states and the parameters. That is, $\mathcal{J}(\theta) = \ell(U(\theta), \theta)$, where usually

$$\ell(U, \theta) = f(\mathcal{P}(U); Y) + \mathcal{R}(\theta), \quad (7)$$

where f is a data-fitting term that compares the outputs with the targets Y , and $\mathcal{R}$ is a regularization term. The derivation below only needs ℓ and Φ to be differentiable, which also makes $\mathcal{J}$ differentiable. Only the gradients $\nabla_U \ell$ and $\nabla_{\theta} \ell$ of ℓ enter, as the right-hand side of the adjoint equation and as the last term of the gradient.

Example 1 (Squared loss) The natural extension of the scalar loss is the squared Frobenius norm of the residuals, written with the inner product on $\mathcal{Y}$,⁷ together with a regularization of the parameters,

$$f(A; Y) = \frac{1}{2m} \langle A - Y, A - Y \rangle_{\mathcal{Y}}, \quad \mathcal{R}(\theta) = \frac{\alpha}{2} \langle \theta, \theta \rangle_{\Theta}. \quad (8)$$

This is the loss of the examples and the code of this post.

2.2. Adjoints

For a linear map $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Z}$ between two of these spaces, the adjoint $\mathcal{A}^* : \mathcal{Z} \rightarrow \mathcal{X}$ is defined by

$$\langle \mathcal{A}(x), z \rangle_{\mathcal{Z}} = \langle x, \mathcal{A}^*(z) \rangle_{\mathcal{X}} \quad \text{for all } x \in \mathcal{X}, z \in \mathcal{Z}. \quad (9)$$

⁵ For example, in a recurrent neural network.

⁶ Usually $\mathcal{P}$ picks out the last layer, $\mathcal{P}(U) = U_N$. It can also select several layers, or combine them, for example by averaging.

Example 1: Squared loss

⁷ The scalar loss comes from the inner product

$$\langle u - y, u - y \rangle = (u - y)^\top (u - y).$$

Moving an operator across an inner product therefore changes the inner product from the operator's output space to its input space.

Example 2: Adjoint of the selection map

Example 2 (Adjoint of the selection map) Take $N = 3$ layers, each producing a matrix U_i with m columns, one per data point, and let the loss depend on the first and the last layer output. Then

$$U = (U_1, U_2, U_3) \in \mathcal{U}, \quad A = (A_1, A_2) \in \mathcal{Y}, \quad (10)$$

where A_1 and A_2 have the sizes of U_1 and U_3 . The selection map $\mathcal{P} : \mathcal{U} \rightarrow \mathcal{Y}$ is $\mathcal{P}(U) = (U_1, U_3)$. We are free to select other layers, or the same layer more than once, as long as each selected U_i has the size of the corresponding entry of $\mathcal{Y}$. For any $U \in \mathcal{U}$ and $A \in \mathcal{Y}$, writing out the inner product on $\mathcal{Y}$ and adding a zero term for the unselected layer gives

$$\begin{aligned} \langle \mathcal{P}(U), A \rangle_{\mathcal{Y}} &= \langle U_1, A_1 \rangle_F + \langle U_3, A_2 \rangle_F \\ &= \langle U_1, A_1 \rangle_F + \langle U_2, 0 \rangle_F + \langle U_3, A_2 \rangle_F = \langle U, (A_1, 0, A_2) \rangle_{\mathcal{U}} \end{aligned} \quad (11)$$

Hence the adjoint $\mathcal{P}^*$ places the selected blocks back in their original positions, with zeros elsewhere, i.e. $\mathcal{P}^*(A) = (A_1, 0, A_2)$. If a layer is selected more than once, $\mathcal{P}^*$ sums the corresponding blocks of A in its position. The inner product starts in $\mathcal{Y}$, the output space of $\mathcal{P}$, and ends in $\mathcal{U}$, its input space. In the common case where only the last layer is selected, $\mathcal{P}(U) = U_N$ and $\mathcal{P}^*(A) = (0, \dots, 0, A)$.

2.3. Gradients

The gradient is defined through an inner product. For a differentiable scalar function of θ , such as the loss $\mathcal{J}$, the derivative $D\mathcal{J}(\theta) \in \Theta^*$ is a linear map from directions $h \in \Theta$ to numbers.⁸ The Riesz representation theorem⁹ says that such a map is an inner product with a fixed element of Θ . That element is the gradient

$$D\mathcal{J}(\theta)[h] = \langle \nabla_{\theta} \mathcal{J}, h \rangle_{\Theta} \quad \text{for every direction } h. \quad (12)$$

So to find a gradient, we write the derivative in the form $\langle \cdot, h \rangle_{\Theta}$ and read off the other argument. The same definition applies on $\mathcal{U}$ and $\mathcal{Y}$, each with its own inner product. Changing the inner product therefore changes the gradient, but not the derivative.

⁸ Θ^* denotes the *dual* space of Θ , which is the space of all linear functionals on Θ . This is a technicality that we revisit in *Which Way Is Down? First-Order Optimizers as a Choice of Norm*.

⁹ In a finite-dimensional inner product space every linear functional ψ has the form $\psi(h) = \langle g, h \rangle_{\Theta}$ for exactly one $g \in \Theta$, its Riesz representative.

Example 3: Gradients of the loss

Example 3 (Gradients of the loss) A state change $E \in \mathcal{U}$ changes the outputs by $\mathcal{P}(E)$, since $\mathcal{P}$ is linear. Moving $\mathcal{P}$ across the inner product with its adjoint gives

$$D_U \ell[E] = \langle \nabla f, \mathcal{P}(E) \rangle_{\mathcal{Y}} = \langle \mathcal{P}^*(\nabla f), E \rangle_{\mathcal{U}}, \quad \text{so} \quad \nabla_U \ell = \mathcal{P}^*(\nabla f), \quad (13)$$

with ∇f evaluated at $U_{\mathcal{P}}$. For the squared loss, $\nabla f = (U_{\mathcal{P}} - Y)/m$,¹⁰ and $D_{\theta} \ell[h] = \alpha \langle \theta, h \rangle_{\Theta}$ gives $\nabla_{\theta} \ell = \alpha \theta$.

The last step depends on the inner product on Θ . With $\langle g, h \rangle_{\mathcal{A}} = \langle \mathcal{A}(g), h \rangle_{\Theta}$ instead, for a self-adjoint and positive definite $\mathcal{A}$, the same derivative reads

$$D_{\theta} \ell[h] = \alpha \langle \theta, h \rangle_{\Theta} = \alpha \langle \mathcal{A}(\mathcal{A}^{-1}(\theta)), h \rangle_{\Theta} = \langle \alpha \mathcal{A}^{-1}(\theta), h \rangle_{\mathcal{A}}, \quad (14)$$

¹⁰ By the product rule and the symmetry of the inner product, $D\langle a, a \rangle[E] = 2\langle a, Da[E] \rangle$ [2].

so the gradient becomes $\alpha \mathcal{A}^{-1}(\theta)$. In the same way, the gradient of $\mathcal{J}$ becomes the preconditioned $\mathcal{A}^{-1}(\nabla_{\theta} \mathcal{J})$, which Which Way Is Down?¹¹ builds optimizers from.

¹¹ Later post: *Which Way Is Down? First-Order Optimizers as a Choice of Norm.*
mipals.github.io/blog/2026/10-13-steepest-descent/

3. The batched gradient

Note that the parameters enter the loss function $\mathcal{J}(\theta) = \ell(U(\theta), \theta)$ twice: directly as θ , and through the states $U(\theta)$. Differentiating in a parameter direction $h \in \Theta$, the chain rule gives one term for each,

$$\begin{aligned} D\mathcal{J}(\theta)[h] &= D_U \ell[Z_h] + D_{\theta} \ell[h] = \langle \nabla_U \ell, Z_h \rangle_{\mathcal{U}} + \langle \nabla_{\theta} \ell, h \rangle_{\Theta}, \\ Z_h &= DU(\theta)[h] \in \mathcal{U}. \end{aligned} \quad (15)$$

In order to find the gradient $\nabla_{\theta} \mathcal{J}$, we need to write the first term in the form $\langle \cdot, h \rangle_{\Theta}$ so that we can read off the other argument—the Riesz representative of the derivative.

3.1. The forward relation

To relate Z_h to h , differentiate the forward relation $U(\theta) = \Phi(U(\theta), \theta)$ in the direction h . The left side gives Z_h . On the right, θ again enters both directly and through $U(\theta)$, and the chain rule gives

$$Z_h = D_U \Phi[Z_h] + D_{\theta} \Phi[h]. \quad (16)$$

The first term accounts for changes through the states, while the second accounts for the explicit parameter dependence.¹² Moving the state term to the left gives the linear equation

$$\underbrace{(\mathcal{I} - D_U \Phi)}_{\mathcal{L}}(Z_h) = \underbrace{D_{\theta} \Phi}_{\mathcal{M}^*}[h], \quad \mathcal{L} : \mathcal{U} \rightarrow \mathcal{U}, \quad \mathcal{M}^* : \Theta \rightarrow \mathcal{U}, \quad (17)$$

¹² Both partial derivatives are evaluated at the current (U, θ) . In $D_U \Phi$, θ is held fixed; in $D_{\theta} \Phi$, U is held fixed. The total state dependence enters through Z_h .

where $\mathcal{L}$ and $\mathcal{M}^*$ are the operator versions of L and $M^{\top}$ from the previous post. Solving it, $\mathcal{L}(Z_h) = \mathcal{M}^*(h)$, gives Z_h for one direction h . This is forward-mode differentiation, and the full gradient this way would need one solve per direction in a basis of Θ , that is, $\dim \Theta$ solves.

3.2. The adjoint equation

Solving Equation 17 gives $Z_h = \mathcal{L}^{-1}(\mathcal{M}^*(h))$. Substituting this into the first term of Equation 15 and moving first $\mathcal{L}^{-1}$, which maps $\mathcal{U}$ to itself, and then $\mathcal{M}^*$, which maps Θ to $\mathcal{U}$, across the inner product using their respective adjoints gives

$$\langle \nabla_U \ell, \mathcal{L}^{-1}(\mathcal{M}^*(h)) \rangle_{\mathcal{U}} = \langle (\mathcal{L}^{-1})^*(\nabla_U \ell), \mathcal{M}^*(h) \rangle_{\mathcal{U}} = \langle \mathcal{M}(\Lambda), h \rangle_{\Theta}, \quad (18)$$

where $\Lambda = (\mathcal{L}^{-1})^*(\nabla_U \ell)$. Rather than forming the inverse, we compute Λ by solving the adjoint equation¹³

$$\mathcal{L}^*(\Lambda) = \nabla_U \ell, \quad (19)$$

whose right-hand side does not depend on h , so a single solve serves every direction. This is reverse-mode differentiation: as shown below, $\mathcal{L}$ is lower block triangular, so $\mathcal{L}^*$ is upper block triangular, and the solve runs from the last layer back to the first. Collecting the terms,

$$D\mathcal{J}(\theta)[h] = \langle \mathcal{M}(\Lambda) + \nabla_{\theta} \ell, h \rangle_{\Theta} \quad \text{for every } h. \quad (20)$$

¹³ For invertible $\mathcal{L}$, the inverse and adjoint commute:

$$(\mathcal{L}^{-1})^* = (\mathcal{L}^*)^{-1}.$$

Taking adjoints of $\mathcal{L}^{-1} \mathcal{L} = \mathcal{I}$ reverses the order, giving $\mathcal{L}^*(\mathcal{L}^{-1})^* = \mathcal{I}$; the reverse product is also the identity. Moving $\mathcal{L}^{-1}$ across the inner product therefore gives a solve with $\mathcal{L}^*$.

Reading off the Riesz representative of $D\mathcal{J}(\theta)$ gives

$$\nabla_{\theta}\mathcal{J} = \mathcal{M}(\Lambda) + \nabla_{\theta}\ell \in \Theta. \quad (21)$$

3.3. The operators as block (sparse) matrices

Viewing the states and the parameters as lists, $\mathcal{L}$ and $\mathcal{M}^*$ can be written as block matrices whose entries are linear maps. For a state change $E \in \mathcal{U}$, layer i can only respond to the states that come strictly before it, so entry i of $\mathcal{L}(E)$ ¹⁴ is

$$(\mathcal{L}(E))_i = E_i - \sum_{j=1}^{i-1} D_{U_j} \Phi_i[E_j] = \sum_{j=1}^i \mathcal{L}_{ij}(E_j), \quad \mathcal{L}_{ij} = \begin{cases} \mathcal{I} & \text{if } j = i \\ -D_{U_j} \Phi_i & \text{if } j < i \end{cases} \quad (22)$$

so block (i, j) of $\mathcal{L}$ is nonzero only if layer i reads the state U_j . In the same way, for a parameter direction $h \in \Theta$, layer i only sees the parameters $\theta_{S_i} = \mathcal{S}_i(\theta)$, so by the chain rule entry i of $\mathcal{M}^*(h)$ ¹⁵ is

$$(\mathcal{M}^*(h))_i = D_{\theta_{S_i}} \Phi_i[\mathcal{S}_i(h)] = \sum_{k=1}^K \mathcal{M}_{ki}^*(h_k), \quad \mathcal{M}_{ki}^* = D_{\theta_k} \Phi_i, \quad (23)$$

so block (i, k) of $\mathcal{M}^*$ is nonzero only if $\mathcal{S}_i$ selects the parameter block θ_k . The indices of $\mathcal{M}_{ki}^*$ are written in the order ki because it is the adjoint of the block $\mathcal{M}_{ki}$ of $\mathcal{M}$, at position (k, i) , as shown below. Written out, with $\mathcal{O}$ denoting the zero map,

$$\mathcal{L} = \begin{pmatrix} \mathcal{I} & \mathcal{O} & \dots & \mathcal{O} \\ \mathcal{L}_{21} & \mathcal{I} & \dots & \mathcal{O} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{L}_{N1} & \mathcal{L}_{N2} & \dots & \mathcal{I} \end{pmatrix}, \quad \mathcal{M}^* = \begin{pmatrix} \mathcal{M}_{11}^* & \dots & \mathcal{M}_{K1}^* \\ \vdots & \ddots & \vdots \\ \mathcal{M}_{1N}^* & \dots & \mathcal{M}_{KN}^* \end{pmatrix}. \quad (24)$$

Layer i only reads earlier states, so $\mathcal{L}$ is lower block triangular with identities on the diagonal. In practice a layer reads only a few of the states, and most parameter blocks are used by a single layer, so most blocks are $\mathcal{O}$, and $\mathcal{L}$ and $\mathcal{M}^*$ are sparse. Since the inner products on $\mathcal{U}$ and Θ are sums over the layers and the parameter blocks, the adjoints follow the same pattern as for matrices: transpose the block pattern and take the adjoint of each block, $(\mathcal{L}^*)_{ji} = \mathcal{L}_{ij}^*$.¹⁶ This gives

$$\mathcal{L}^* = \begin{pmatrix} \mathcal{I} & \mathcal{L}_{21}^* & \dots & \mathcal{L}_{N1}^* \\ \mathcal{O} & \mathcal{I} & \dots & \mathcal{L}_{N2}^* \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{O} & \mathcal{O} & \dots & \mathcal{I} \end{pmatrix}, \quad \mathcal{M} = \begin{pmatrix} \mathcal{M}_{11} & \dots & \mathcal{M}_{1N} \\ \vdots & \ddots & \vdots \\ \mathcal{M}_{K1} & \dots & \mathcal{M}_{KN} \end{pmatrix}. \quad (25)$$

Since $\mathcal{L}^*$ is upper block triangular, the adjoint equation is solved by back substitution from the last layer to the first. Both this solve and the application of $\mathcal{M}$ only require the action of the adjoint blocks on the entries Λ_i .

¹⁴ Recall that $\mathcal{L} : \mathcal{U} \rightarrow \mathcal{U}$ maps state changes to state changes, so $\mathcal{L}(E)$ is again a list with one entry per layer.

¹⁵ Recall that $\mathcal{M}^* : \Theta \rightarrow \mathcal{U}$ maps parameter changes to the state changes they cause directly: it takes a list over the K parameter blocks to a list over the N layers. Its adjoint $\mathcal{M} : \mathcal{U} \rightarrow \Theta$ goes the other way.

¹⁶ This relies on the inner products treating the blocks separately. An inner product that couples the blocks, such as $\langle h, \mathcal{A}(g) \rangle_{\Theta}$ with the Fisher information $\mathcal{A}$ of natural gradient descent, gives an adjoint that also involves $\mathcal{A}$. In practice one keeps the separable inner product for the derivation and applies $\mathcal{A}^{-1}$ to the resulting gradient afterwards.

Example 4 (Affine layers) Let every layer apply an affine map to the states it reads, followed by an entrywise activation σ . Layer i reads the state U_j through a weight W_{ij} and adds a bias b_i , so that

$$\Phi_i(U, \theta) = \sigma(Z_i), \quad Z_i = \sum_j W_{ij} U_j + b_i \mathbf{1}^\top, \quad \Delta_i = \sigma'(Z_i), \quad (26)$$

where the sum runs over the states that layer i reads, and U_0 is the input data. The selection map $\mathcal{S}_i$ picks out the weights W_{ij} and the bias b_i of layer i among the

Example 4: Affine layers

parameter blocks θ_k , and two layers share a block if, for example, $W_{32} = W_{43}$. A plain feed-forward network only has the weights $W_{i,i-1}$. The product rule and the entrywise linearization of σ give

$$\mathcal{L}_{ij}(E_j) = -\Delta_i \odot (W_{ij}E_j), \quad \mathcal{M}_{ki}^*(h_k) = \begin{cases} \Delta_i \odot (h_k U_j^\top) & \text{if } \theta_k = W_{ij} \\ \Delta_i \odot (h_k \mathbf{1}^\top) & \text{if } \theta_k = b_i \end{cases} \quad (27)$$

where $\odot$ denotes entrywise multiplication.¹⁷ Their adjoints follow from two properties of the Frobenius inner product: entrywise multiplication is self-adjoint, $\langle Z, \Delta \odot X \rangle_F = \langle \Delta \odot Z, X \rangle_F$, and a factor of a matrix product moves to the other side, $\langle Z, AB \rangle_F = \langle A^\top Z, B \rangle_F = \langle ZB^\top, A \rangle_F$.¹⁸ Hence

$$\mathcal{L}_{ij}^*(\Lambda_i) = -W_{ij}^\top (\Delta_i \odot \Lambda_i), \quad \mathcal{M}_{ki}(\Lambda_i) = \begin{cases} (\Delta_i \odot \Lambda_i) U_j^\top & \text{if } \theta_k = W_{ij} \\ (\Delta_i \odot \Lambda_i) \mathbf{1} & \text{if } \theta_k = b_i \end{cases}. \quad (28)$$

4. A code example

The code is adapted from the Julia notebook *Backprop with Backslash* accompanying [1], which fixes a block bidiagonal $\mathcal{L}$ and a block diagonal $\mathcal{M}$. Here both are general sparse matrices of operators, so that a layer can read any earlier state and share parameter blocks with other layers.

We introduce a network of $N = 6$ layers, which produce the states $U_1, \dots, U_6$ from the input U_0 , and include $m = 10$ data points. The layers are those of Example 4. Besides the chain weights $W_{i,i-1}$, layers 3, 5, and 6 also read U_1, U_2 , and U_3 through the skip weights W_{31} , W_{52} , and W_{63} , and layers 3 and 4 share both their chain weight, $W_{32} = W_{43}$, and their bias, $b_3 = b_4$. With 8 weight blocks and 5 bias blocks, $K = 13$.¹⁹ The operator $\mathcal{L}$ then has the pattern

$$\mathcal{L} = \begin{pmatrix} \mathcal{I} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{L}_{2,1} & \mathcal{I} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{L}_{3,1} & \mathcal{L}_{3,2} & \mathcal{I} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{L}_{4,3} & \mathcal{I} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{L}_{5,2} & \mathcal{O} & \mathcal{L}_{5,4} & \mathcal{I} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{L}_{6,3} & \mathcal{O} & \mathcal{L}_{6,5} & \mathcal{I} \end{pmatrix}. \quad (29)$$

Already here one can see that $\mathcal{L}$ is sparse, with only 14 nonzero blocks out of 36, and only 8 of the 15 blocks below the diagonal. The deeper the network, the sparser it becomes, as each layer reads only a few of the earlier states.

Column i of $\mathcal{M}$ has a block in row k for every parameter block θ_k that $\mathcal{S}_i$ selects.²⁰ This gives

¹⁷ The weights W_{i0} read the input data U_0 , which is fixed and not part of $\mathcal{U}$, so they give no block in $\mathcal{L}$, but they still give blocks of $\mathcal{M}^*$. Treating U_0 as a state would add the blocks $\mathcal{L}_{i0}$ and give the gradient with respect to the input.

¹⁸ Both equalities follow from $\langle X, Y \rangle_F = \text{tr}(X^\top Y)$. The first only regroups the product,

$$\text{tr}(Z^\top AB) = \text{tr}((A^\top Z)^\top B),$$

and the second uses the cyclic property of the trace,

$$\text{tr}(Z^\top AB) = \text{tr}(BZ^\top A) = \text{tr}((ZB^\top)^\top A).$$

¹⁹ Each of the 6 layers has a chain weight and a bias, but layers 3 and 4 share theirs, which leaves $6 - 1 = 5$ pairs. Each of the 3 skip connections adds a weight block of its own, so $K = 2(6 - 1) + 3 = 13$.

²⁰ We number the weights first, $\theta_1 = W_{10}$, $\theta_2 = W_{21}$, $\theta_3 = W_{32} = W_{43}$, $\theta_4 = W_{54}$, and $\theta_5 = W_{65}$ along the chain and $\theta_6 = W_{31}$, $\theta_7 = W_{52}$, and $\theta_8 = W_{63}$ for the skip connections, and the biases $\theta_9, \dots, \theta_{13}$ last, so that, for example, $\mathcal{S}_3(\theta) = (\theta_3, \theta_6, \theta_{11})$ and $\mathcal{S}_4(\theta) = (\theta_3, \theta_{11})$.

$$\mathcal{M} = \begin{pmatrix} \mathcal{M}_{1,1} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{M}_{2,2} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{M}_{3,3} & \mathcal{M}_{3,4} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{M}_{4,5} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{M}_{5,6} \\ \mathcal{O} & \mathcal{O} & \mathcal{M}_{6,3} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{M}_{7,5} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{M}_{8,6} \\ \mathcal{M}_{9,1} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{M}_{10,2} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{M}_{11,3} & \mathcal{M}_{11,4} & \mathcal{O} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{M}_{12,5} & \mathcal{O} \\ \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{M}_{13,6} \end{pmatrix}. \quad (30)$$

The rows of the shared weight θ_3 and of the shared bias θ_{11} each hold two blocks, so these two entries of $\mathcal{M}(\Lambda)$ sum the contributions of layers 3 and 4.

In the code, each block is an operator that stores the map and its adjoint, and $\mathcal{L}$ and $\mathcal{M}^*$ are sparse matrices of these operators. The only new methods are the back substitution with $\mathcal{L}^*$ and the application of $\mathcal{M}$, which both visit only the stored blocks. The script prints the patterns of $\mathcal{L}$ and $\mathcal{M}^*$ (the transpose of the pattern of $\mathcal{M}$ above) and compares the gradient with a central difference along a random direction.

julia

```
backbackback.jl
1 using LinearAlgebra, Random, SparseArrays
2 Random.seed!(42)
3
4 # A linear map stored together with its adjoint
5 struct Operator
6     operator
7     adjoint
8     sym::String
9 end
10 Base.adjoint(A::Operator) = Operator(A.adjoint, A.operator, "$(A.sym)'\")
11 Base.*(A::Operator, X) = A.operator(X)
12 Base.show(io::IO, A::Operator) = print(io, A.sym)
13 Base.zero(::Type{Operator}) = Operator(zero, zero, "0") # For printing
14 const I = Operator(identity, identity, "I")
15
16 # Adjoints of sparse matrices whose entries are operators.
17 const AdjointOperators = Adjoint{<:Any, <:SparseMatrixCSC{Operator}}
18
19 # Back substitution with  $\mathcal{L}^*$ :  $\Lambda_j = g_j - \sum_{i>j} \mathcal{L}_{ij}^* \Lambda_i$ .
20 function Base.:\(A::AdjointOperators, g::Vector)
21     L = parent(A) # Get the non-adjoint, but apply adjoint later
22     rows, blocks = rowvals(L), nonzeros(L) # Column j: blocks[r], rows[r]
23     Lambda = copy(g)
24     for j in length(g):-1:1, r in nzrange(L, j)
25         i = rows[r]
26         if i > j
27             Lambda[j] -= blocks[r]' * Lambda[i]
28         end
29     end
30     return Lambda
31 end
32
33 #  $\mathcal{M} = (\mathcal{M}')^*$ : entry k sums the contributions of the layers that use  $\theta_k$ .
34 function Base.*(A::AdjointOperators, Lambda::Vector)
35     M' = parent(A) # Get the non-adjoint, but apply adjoint later
36     rows, blocks = rowvals(M'), nonzeros(M')
```

```

37     return [sum(blocks[r]' * Λ[rows[r]] for r in nzrange(M', k))
38             for k in axes(M', 2)]
39 end
40
41 # Defining the activation function and its derivative
42 σ(x) = exp(-x)
43 σ'(x) = -exp(-x)
44
45 # Layer i reads the state U_j through the weight W_ij = θ_k for every
46 # j => k in W[i], and adds the bias b_i = θ_b[i]. Layers 3 and 4 share
47 # both their weight θ_3 and their bias θ_11.
48 d = [5, 4, 4, 4, 4, 3, 1] # Feature sizes of U_0, ..., U_6
49 W = [[0 => 1], [1 => 2], [2 => 3, 1 => 6], [3 => 3], [4 => 4, 2 => 7],
50       [5 => 5, 3 => 8]]
51 b = [9, 10, 11, 11, 12, 13]
52 N, K = 6, 13 # Layers, parameter blocks
53
54 function network(θ)
55     U, Δ = Matrix{Float64}[], Matrix{Float64}[]
56     state(j) = j == 0 ? U_0 : U[j]
57     for i in 1:N
58         Z = sum(θ[k] * state(j) for (j, k) in W[i]) .+ θ[b[i]]
59         push!(U, σ.(Z))
60         push!(Δ, σ'.(Z))
61     end
62     return U, Δ
63 end
64
65 # The sparse matrices L = I - D_U Φ and M' = D_θ Φ, block by block.
66 function operators(θ, U, Δ)
67     state(j) = j == 0 ? U_0 : U[j]
68     L, M' = spzeros(Operator, N, N), spzeros(Operator, N, K)
69     for i in 1:N
70         L[i, i] = I
71         for (j, k) in W[i]
72             Wij, Uj = θ[k], state(j)
73             if j > 0
74                 L[i, j] = Operator(E -> -(Δ[i] .* (Wij * E)),
75                                     Λ -> -(Wij' * (Δ[i] .* Λ)), "L")
76             end
77             M'[i, k] = Operator(h -> Δ[i] .* (h * Uj),
78                               Λ -> (Δ[i] .* Λ) * Uj', "M'")
79         end
80         M'[i, b[i]] = Operator(h -> Δ[i] .* h,
81                               Λ -> vec(sum(Δ[i] .* Λ, dims=2)), "M'")
82     end
83     return L, M'
84 end
85
86 # The data and the parameters.
87 m = 10 # Data points
88 α = 1e-2 # Weight of the regularization α/2 (θ, θ)
89 init(dims...) = 0.01 * randn(dims...)
90 U_0, Y = init(d[1], m), init(d[end], m)
91 θ = Vector{Array{Float64}}{undef, K}
92 for i in 1:N
93     for (j, k) in W[i]
94         θ[k] = init(d[i+1], d[j+1])
95     end
96     θ[b[i]] = init(d[i+1])
97 end
98
99 # The gradient of J(θ) = ||U_N - Y||^2 / (2m) + α/2 (θ, θ).
100 J(θ) = norm(network(θ)[1][N] - Y)^2 / (2m) + α / 2 * dot(θ, θ)
101 U, Δ = network(θ)
102 L, M' = operators(θ, U, Δ)
103 ∇Uℓ = [i == N ? (U[N] - Y) / m : zero(U[i]) for i in 1:N] # ℙ*(∇f)
104 Λ = L' \ ∇Uℓ
105 ∇J = M' * Λ + α * θ # ℙ(Λ) + αθ
106 display(L)

```

```

107 display(M')
108
109 # Central difference of the loss along a random direction.
110 h = [randn(size(p)...) for p in θ]
111 ε = 1e-6
112 directional = (J(θ + ε * h) - J(θ - ε * h)) / (2ε)
113 versus_finite_difference = abs(directional - dot(∇J, h)) / abs(dot(∇J, h))
114 @show versus_finite_difference

```

5. Conclusion

Going from a single input to a batch only changed the entries of the matrices, from scalars to linear maps. The overall structure stayed the same: $\mathcal{L}$ is a unit lower block triangular matrix whose sparsity pattern is a DAG, and both $\mathcal{L}$ and $\mathcal{M}$ are most often sparse. The formula for the gradient also kept the same form, $\nabla_{\theta}\mathcal{J} = \mathcal{M}((\mathcal{L}^*)^{-1}\nabla_U\ell) + \nabla_{\theta}\ell$,²¹ and can still be computed with a single back substitution and a single sparse product.

Bibliography

- [1] A. Edelman, E. Akyürek, and Y. Wang, “Backpropagation through Back Substitution with a Backslash,” *SIAM Journal on Matrix Analysis and Applications*, vol. 45, no. 1, pp. 429–449, 2024, doi: 10.1137/22M1532871.
- [2] P. Bright, A. Edelman, and S. G. Johnson, “Matrix Calculus (for Machine Learning and Beyond).” [Online]. Available: <https://arxiv.org/abs/2501.14787>

²¹ In the previous post the gradient reads $\nabla_{\theta}\mathcal{J} = ML^{-\top}P^{\top}\nabla_{u_P}\ell + \mu\nabla_{\theta}\mathcal{R}$. The matrices M and L become the operators $\mathcal{M}$ and $\mathcal{L}$, the transpose becomes the adjoint, $P^{\top}\nabla_{u_P}\ell$ becomes $\nabla_U\ell = \mathcal{P}^*(\nabla f)$, and the regularization is part of $\nabla_{\theta}\ell$.