

Backpropagation is the Adjoint Method

Mikkel Paltor (mpasc@dtu.dk), 25 October 2025

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

Backpropagation is often introduced as something developed within the field of machine learning. However, the reality is that backpropagation is just a special case of the adjoint method where the inversion of the partial derivative $\frac{\partial f}{\partial u}$ can be done efficiently due to its lower-triangular structure. This note is in two parts. In the first part we review the adjoint method while in the second part we describe how backpropagation is a special case of the adjoint method with a structure that results in scalable computational algorithms.

A key insight is that while the adjoint method is seemingly limited to equality constrained problems, it turns out that all computer programs are trivially equality constrained problems with equalities coming from each assignment¹. In this sense, as shown in a later section, optimizing a neural network is actually an equality constrained optimization problem with constraints coming from each layer of the network. This is opposed to how optimizing a neural network is usually presented as an unconstrained optimization problem.

1. The Adjoint Method

In many engineering applications we are interested in solving equality constrained optimization problems where the optimization variable (θ) implicitly defines another variable (u) that then appears in the objective function². Written out this means solving optimization problems of the following form

$$\begin{aligned} \min_{\theta \in \mathbb{R}^{n_\theta}} \quad & L(u, \theta), \quad (\text{some objective function}) \\ \text{subject to} \quad & f(u, \theta) = 0, \quad (\text{some equality constraint}) \\ & u \in \mathbb{R}^{n_u}. \end{aligned} \quad (1)$$

Most methods for solving equality constrained optimization problems require the computation of the gradient of the objective function with respect to the optimization variable θ . This is where the problem comes in: Naively computing the gradient requires the computation of the sensitivities of the implicitly defined variable u with respect to the optimization variable θ (i.e. $\frac{du}{d\theta} \in \mathbb{R}^{n_u \times n_\theta}$). This results in a computational bottleneck as forming the sensitivities scales as $\mathcal{O}(n_u n_\theta)$, meaning that adding a new parameter adds n_u additional sensitivities (and similarly adding one more variable would add n_θ additional sensitivities).

To resolve this computational bottleneck the adjoint method was introduced. The first step in the derivation of the adjoint method is to introduce the Lagrangian of the objective function, i.e.

$$\mathcal{L}(u, \theta, \lambda) = L(u, \theta) + \lambda^\top f(u, \theta). \quad (2)$$

It is easy to see that whenever u satisfies the equality constraint $f(u, \theta) = 0$, then the Lagrangian is equal to the original objective function. The reason why the introduction of the additional term in the Lagrangian is useful is that λ can be set to

Contents

1. The Adjoint Method
2. Backpropagation and the Adjoint Method
 - 2.1. How to compute the elements of L and $M^\top$
- Bibliography

¹ All lines of code are essentially “variable = something” which can be written as “variable – something = 0”

² A famous example of this is Topology Optimization

be anything. In particular, we aim to set it in a way so that we can avoid computing the otherwise computationally expensive sensitivities $\frac{du}{d\theta}$. We start by computing the total derivative of the Lagrangian with respect to θ^3

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial u} \frac{du}{d\theta} + \lambda^\top \left(\frac{\partial f}{\partial \theta} + \frac{\partial f}{\partial u} \frac{du}{d\theta} \right). \quad (3)$$

Now we collect the terms that depend on the undesired $\frac{du}{d\theta}$, resulting in

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial L}{\partial \theta} + \lambda^\top \frac{\partial f}{\partial \theta} + \left(\frac{\partial L}{\partial u} + \lambda^\top \frac{\partial f}{\partial u} \right) \frac{du}{d\theta}. \quad (4)$$

As we can choose λ freely a natural idea is to set it in a way such that the term in front of the undesired term vanishes. This means that we can choose λ as the solution to the equation

$$\frac{\partial L}{\partial u} + \lambda^\top \frac{\partial f}{\partial u} = 0 \Rightarrow \lambda^\top = -\frac{\partial L}{\partial u} \left(\frac{\partial f}{\partial u} \right)^{-1}. \quad (5)$$

Inserting $\lambda^\top$ back into the equation we find that the gradient of the Lagrangian with respect to θ is given by

$$\frac{d\mathcal{L}}{d\theta} = \frac{\partial L}{\partial \theta} - \underbrace{\frac{\partial L}{\partial u} \left(\frac{\partial f}{\partial u} \right)^{-1}}_{\lambda^\top} \frac{\partial f}{\partial \theta}. \quad (6)$$

To conclude: The adjoint method is a simple way to avoid the computational bottleneck of computing the sensitivities $\frac{du}{d\theta}$ by cleverly computing the so-called adjoint variable λ in a way that eliminates the need to compute the problematic sensitivities.

³ The total derivative is the transpose of the gradient i.e. $\left(\frac{dL}{d\theta}\right)^\top = \nabla_\theta L$. That is multiplying the change in θ with the total derivative (rather than the transpose of the gradient) gives the change in the function L .

Example 1 (Linearly constrained problem with structure) In order to illustrate the adjoint method we consider a simple linearly constrained problem of the form (inspiration from problem 38 in [1])

$$\begin{aligned} L(u, \theta) &= (c^\top u(\theta))^2, \\ f(u, \theta) &= A(\theta)u - b = 0, \end{aligned} \quad (7)$$

where $A(\theta) \in \mathbb{R}^{n \times n}$ is a symmetric tridiagonal matrix that depends on the parameters $\theta \in \mathbb{R}^{2n-1}$ as

$$A = \begin{bmatrix} \theta_1 & \theta_{n+1} & & & 0 \\ \theta_{n+1} & \theta_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \theta_{n-1} & \theta_{2n-1} \\ 0 & & & \theta_{2n-1} & \theta_n \end{bmatrix}. \quad (8)$$

Now in order to compute the gradient of interest we start by computing the adjoint variable λ as

$$\lambda^\top = -\frac{\partial L}{\partial u} \left(\frac{\partial f}{\partial u} \right)^{-1} = -2(c^\top u)c^\top A(\theta)^{-1}. \quad (9)$$

Example 1: Linearly constrained problem with structure

Note that in practice we do not form $A(\theta)^{-1}$ explicitly but rather compute λ by solving the linear system $A(\theta)^\top \lambda = -2c(c^\top u)$. Using the adjoint variable we can compute the gradient of L with respect to θ as

$$\frac{dL}{d\theta} = \underbrace{\frac{\partial L}{\partial \theta}}_{=0} + \lambda^\top \frac{\partial f}{\partial \theta} = \lambda^\top \frac{\partial A}{\partial \theta} u. \quad (10)$$

In the concrete case of the derivatives of A with respect to θ_i we have that

$$\frac{\partial A}{\partial \theta_i} = \begin{bmatrix} \delta_{i,1} & \delta_{i,n+1} & & & 0 \\ \delta_{i,n+1} & \delta_{i,2} & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \delta_{i,n-1} & \delta_{i,2n-1} \\ 0 & & & \delta_{i,2n-1} & \delta_{i,n} \end{bmatrix}. \quad (11)$$

Using this it follows that the i th component of the gradient can be computed as

$$\lambda^\top \frac{\partial A}{\partial \theta_i} u = \begin{cases} \lambda_i u_i, & i \leq n, \\ \lambda_{i+1-n} u_{i-n} + \lambda_{i-n} u_{i+1-n}, & i > n. \end{cases} \quad (12)$$

Below is a Julia implementation of the above example along with a test against ForwardDiff.jl to verify the correctness of the implementation.

julia

```
1 using LinearAlgebra
2 using ForwardDiff
3 using Random
4 using Test
5
6 Random.seed!(42)
7 n = 1000 # Number of linear constraints
8 c = rand(n) # Objective vector
9 b = rand(n) # Linear equality vector
10 θ = rand(2n-1) # Parameters of the equality constraint
11
12 # Parametrizing linear equality constraint matrix
13 A(θ) = SymTridiagonal(θ[1:n], θ[n+1:end])
14
15 # Objective function
16 f(θ) = (c' * (A(θ) \ b))^2
17
18 # Partial derivatives
19 partial(i,n,λ,u) = i <= n ? λ[i]*u[i] : λ[i+1-n]*u[i-n] + λ[i-n]*u[i+1-n]
20
21 # Gradient computation
22 function ∇f(θ)
23     n = length(b)
24     M = A(θ) # Defining constraint matrix
25     u = M \ b # Computing solution
26     λ = M \ (-2*c*(c'*u)) # Adjoint variables
27     return [partial(i,n,λ,u) for i = 1:2n-1]
28 end
29
30 # Testing against ForwardDiff
31 @test ForwardDiff.gradient(f,θ) ≈ ∇f(θ)
```

2. Backpropagation and the Adjoint Method

Section 5.5 of [2] includes an example of how the adjoint method and backpropagation of neural networks are similar. In neural networks we are often interested in minimizing a loss function of the form

$$L(\theta; u_0) = \| \Phi_N(\theta; u_0) - y \|_2^2 + \mu \mathcal{R}(\theta), \quad (13)$$

with $\theta \in \mathbb{R}^k$, $u_0 \in \mathbb{R}^m$, $y \in \mathbb{R}^n$ and the notation “; u_0 ” is used in order to highlight that u_0 is a constant input (i.e. most often the “data”) and $\mathcal{R}(\theta)$ is some regularization function (e.g. $\mathcal{R}(\theta) = \| \theta \|_2^2$). Furthermore $\Phi_N(\theta; u_0) : \mathbb{R}^k \rightarrow \mathbb{R}^n$ is a neural network with N layers that given a set of constant inputs u_0 maps the parameters θ to an output. Now, the above objective function does not include any equality constraints. However, one should realize that an N -layer neural network is nothing more than a composition of N functions

$$\Phi_N(\theta; u_0) = \Phi_N(\Phi_{N-1}(\dots(\Phi_1(\theta; u_0)\dots, \theta; u_0), \theta; u_0). \quad (14)$$

Now using the notation that u_i is the output of the i th layer of the network we can write the propagation through the network as a large nonlinear system of equations that have to be satisfied

$$\begin{aligned} f(u, \theta) &= u - \Phi(u, \theta) \\ &= \underbrace{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{bmatrix}}_u - \underbrace{\begin{bmatrix} \Phi_1(\theta; u_0) \\ \Phi_2(u_1, \theta; u_0) \\ \vdots \\ \Phi_N(u_1, \dots, u_{N-1}, \theta; u_0) \end{bmatrix}}_{\Phi(u, \theta)} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \end{aligned} \quad (15)$$

Using this notation (and removing the explicit dependence on u_0) we see that optimizing a neural network is really nothing more than solving the following equality constrained optimization problem

$$\begin{aligned} \min_{\theta} \quad & L(u, \theta) = \| u_N - y \|_2^2 + \mu \mathcal{R}(\theta), \\ \text{subject to} \quad & f(u, \theta) = 0. \end{aligned} \quad (16)$$

We can now use the adjoint method to compute the gradient of L with respect to θ . As a reminder this means that

$$\frac{dL}{d\theta} = \frac{\partial L}{\partial \theta} - \underbrace{\frac{\partial L}{\partial u} \left(\frac{\partial f}{\partial u} \right)^{-1}}_{\lambda^\top} \frac{\partial f}{\partial \theta}, \quad (17)$$

where λ is referred to as the adjoint variable. Let n_i be the dimension of the output u_i of layer i , so that the stacked state u has dimension $d = \sum_{i=1}^N n_i$. For simplicity, we assume that the final output u_N is scalar, i.e. $n_N = 1$. Then $u_N = e_d^\top u$, where e_d is the last canonical basis vector in $\mathbb{R}^d$. In this case we have that

$$\frac{\partial L}{\partial u} = 2(e_d^\top u - y)e_d^\top = 2(u_N - y)e_d^\top = g_N^\top. \quad (18)$$

Now what is left to compute is $\frac{\partial f}{\partial u}$ and $\frac{\partial f}{\partial \theta}$. Starting with $\frac{\partial f}{\partial u}$ we note that because f only propagates the u_i 's forwards the resulting partial derivative is a lower block triangular matrix of the form

$$\frac{\partial f}{\partial u} = \begin{bmatrix} I & 0 & \dots & 0 \\ -\frac{\partial \Phi_2}{\partial u_1} & I & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{\partial \Phi_N}{\partial u_1} & \dots & -\frac{\partial \Phi_N}{\partial u_{N-1}} & I \end{bmatrix} = L. \quad (19)$$

Now is a good place to stop and think of a key concept of backpropagation: The resulting computational graph must be a Directed Acyclic Graph (DAG). This is indeed equivalent to stating that $\frac{\partial f}{\partial u}$ should be a lower triangular matrix. This is an important property as the adjoint method requires us to invert $\frac{\partial f}{\partial u}$, which can be done cheaply for a lower triangular matrix. Even more important is that the diagonal blocks are the identity, meaning that forward/backward substitutions can be done without any matrix inversions at all.

What is left is to compute $\frac{\partial f}{\partial \theta}$, which standardly is done as

$$\frac{\partial f}{\partial \theta} = - \underbrace{\begin{bmatrix} \frac{\partial \Phi_1}{\partial \theta} \\ \vdots \\ \frac{\partial \Phi_N}{\partial \theta} \end{bmatrix}}_{M^\top} = -M^\top. \quad (20)$$

Now, in case of the layers not sharing any parameters the matrix $M^\top$ will have a block diagonal structure, i.e.

$$M^\top = \text{blkdiag} \left(\frac{\partial \Phi_1}{\partial \theta_1}, \dots, \frac{\partial \Phi_N}{\partial \theta_N} \right), \quad \theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_N \end{bmatrix}, \quad (21)$$

where θ_i are the parameters of layer i .

Putting everything together we find that the gradient of L with respect to θ is given by

$$\begin{aligned} \frac{d\mathcal{L}}{d\theta} &= \frac{\partial L}{\partial \theta} - \frac{\partial L}{\partial u} \left(\frac{\partial f}{\partial u} \right)^{-1} \frac{\partial f}{\partial \theta} \\ &= \mu \frac{\partial \mathcal{R}}{\partial \theta} - (g_N^\top L^{-1} (-M^\top)) \\ &= \mu \frac{\partial \mathcal{R}}{\partial \theta} + 2(u_N - y)e_d^\top L^{-1} M^\top. \end{aligned} \quad (22)$$

As noted previously both L^{-1} and $M^\top$ are structured, meaning that the above can be computed efficiently.

2.1. How to compute the elements of L and $M^\top$

For a standard linear layer an activation function σ is usually applied element wise. In practice this means that we really should look at the rows of Φ_i separately. For the j th row the gradients are easily seen to be

$$\begin{aligned} \nabla_{u_{i-1}} \sigma(w_j^\top u_{i-1} + b_j) &= \sigma'(w_j^\top u_{i-1} + b_j) w_j, & \text{Goes into } L \\ \nabla_{w_j} \sigma(w_j^\top u_{i-1} + b_j) &= \sigma'(w_j^\top u_{i-1} + b_j) u_{i-1}, & \text{Goes into } M^\top \\ \nabla_{b_j} \sigma(w_j^\top u_{i-1} + b_j) &= \sigma'(w_j^\top u_{i-1} + b_j), & \text{Goes into } M^\top \end{aligned} \quad (23)$$

Using the above we see that

$$\frac{\partial \Phi_i}{\partial u_{i-1}} = \text{diag}(\sigma'(W_i u_{i-1} + b_i)) W_i, \quad (24)$$

where $\sigma'(\cdot)$ is applied element wise. Now for $\frac{\partial \Phi_i}{\partial \theta_i}$ we have to pick a way of how to vectorize the parameters. Here we choose

$$\theta_i = \begin{bmatrix} \text{vec}(W_i) \\ b_i \end{bmatrix}, \quad (25)$$

where $\text{vec}(W)$ vectorizes by stacking the columns of W . Using the vectorization we can write the sought after sensitivity as

$$\frac{\partial \Phi_i}{\partial \theta_i} = \text{diag}(\sigma'(W_i u_{i-1} + b_i)) ([u_{i-1}^\top \ 1] \otimes I_{n_i}). \quad (26)$$

While the Kronecker product above does result in a sparse matrix, it is even more structured. Using the standard property of Kronecker products, $(A \otimes B)v = \text{vec}(B \text{ mat}(v) A^\top)$, where $\text{mat}(v)$ reverses the column-stacking operation vec with dimensions compatible with A and B , its products with vectors can be computed without forming it, e.g. $([u_{i-1}^\top \ 1] \otimes I_{n_i})\theta_i = W_i u_{i-1} + b_i$.

A final remark here is that the diagonal matrices that go into the elements of L and $M^\top$ are the same. That is if we define

$$\begin{aligned} D &= \text{diag}(\sigma'(W_1 u_0 + b_1), \dots, \sigma'(W_N u_{N-1} + b_N)) \\ &= \text{blkdiag}(\text{diag}(\sigma'(W_1 u_0 + b_1)), \dots, \text{diag}(\sigma'(W_N u_{N-1} + b_N))), \end{aligned} \quad (27)$$

then it follows that

$$\begin{aligned} L &= I - D \text{blkdiag}(W_2, \dots, W_N, -1), \\ M^\top &= D \text{blkdiag}([u_0^\top \ 1] \otimes I_{n_1}, \dots, [u_{N-1}^\top \ 1] \otimes I_{n_N}), \end{aligned} \quad (28)$$

with the slightly unusual notation that $\text{blkdiag}(\cdot, -1)$ denotes the lower block diagonal matrix. Here its $(i, i-1)$ block is W_i for $i = 2, \dots, N$, with all other blocks zero and block row and column sizes $n_1, \dots, n_N$. The weight W_1 acts on the fixed input u_0 and therefore does not appear in L .

Below is a Julia implementation of the above backpropagation/adjoint method for a fully connected feedforward neural network with scalar output. In this example we set $\mu = 0$, so the objective contains only the squared error. The implementation uses `BlockDiagonalMatrices.jl` and `Kronecker.jl` in order to efficiently represent the involved matrices.

julia

```
1 using BlockDiagonalMatrices
2 using LinearAlgebra
3 using Kronecker
4 using Random
5 using Test
6
7 Random.seed!(42)
8
9 h(x) = exp(-x) # activation function
10 ∇h(x) = -exp(-x) # derivative of activation function
11
12 # Forward pass including the derivatives
13 function forward_pass(u0, layers)
14     u = u0
15     _, b1 = first(layers)
16     derivs = typeof(b1)[]
17     kron_blocks = typeof([u; 1]' * I(length(b1)))[]
18     for (W, b) in layers
19         push!(kron_blocks, [u; 1]' * I(length(b))) # Lazy Kronecker product
20         z = W*u + b
21         u = h.(z)
22         push!(derivs, ∇h.(z))
```

```

23     end
24     return kron_blocks, derivs, u
25 end
26 # Block back substitution: solves  $L^T y = b$ , i.e.  $y = L^{(-T)} b$ 
27 function backsub(derivs, weights, b)
28     y = Matrix(b)
29     stop = length(y)
30     @views for (d,Wt) in zip(reverse(derivs), reverse(transpose(weights)))
31         nprev, n = size(Wt)
32         src = stop-n+1:stop
33         dst = stop-n-nprev+1:stop-n
34         tmp = d .* y[src]
35         # Equivalent to: y[dst] .+= Wt * (d .* y[src])
36         mul!(y[dst], Wt, tmp, 1, 1)
37         stop -= n
38     end
39     return y
40 end
41 # Helper function that unpacks a vector  $\theta$  into Ws and bs
42 function unpack_θ(θ, layer_sizes)
43     Ws, bs = Matrix{eltype(θ)}[], Vector{eltype(θ)}[]
44     i = 1
45     for (n_in,n_out) in zip(layer_sizes[1:end-1], layer_sizes[2:end])
46         j = i + n_out*n_in
47         push!(Ws, reshape(θ[i:j-1], n_out, n_in))
48         i = j + n_out
49         push!(bs, θ[j:i-1])
50     end
51     return Ws, bs
52 end
53 # Evaluate the network output using the forward pass
54 function network_output(θ, layer_sizes, u0)
55     Ws, bs = unpack_θ(θ, layer_sizes)
56     _, _, uN = forward_pass(u0, zip(Ws,bs))
57     return only(uN)
58 end
59 # Gradient computation using the adjoint method
60 function ∇L(θ, layer_sizes, u0; y=0.0)
61     # First unpack the parameter vector
62     Ws, bs = unpack_θ(θ, layer_sizes)
63     # Forward pass includes derivative information
64     kron_blocks, derivs, uN = forward_pass(u0, zip(Ws, bs))
65     # S has subdiagonal blocks  $W_2, \dots, W_N$ ;  $L = I - D \cdot S$  and  $M^T = D \cdot K$ 
66     D = Diagonal(vcat(derivs...))
67     K = BlockDiagonal(kron_blocks)
68     g = zeros(eltype(θ), sum(layer_sizes[2:end]))
69     g[end] = 2*(only(uN) - y)
70     # The final step is to evaluate the gradient from the right
71     grad_adjoint = (backsub(derivs[2:end], Ws[2:end], g') * D) * K
72     return grad_adjoint'
73 end
74 # Forward difference to test the adjoint gradient implementation
75 function fd(θ, layer_sizes, u0, i; e=1e-6, y=2.0)
76     uN0 = network_output(θ, layer_sizes, u0)
77     θ[i] += e
78     uN1 = network_output(θ, layer_sizes, u0)
79     θ[i] -= e
80     return ((uN1 - y)^2 - (uN0 - y)^2)/e
81 end
82
83 # Setting up parameters
84
85 layer_sizes = [50,40,30,20,10,1]
86 N = length(layer_sizes) - 1
87 init(sizes...) = 0.01*randn(sizes...)
88 Ws = [init(layer_sizes[i+1], layer_sizes[i]) for i=1:N]
89 bs = [init(layer_sizes[i+1]) for i = 1:N]
90 u0 = init(layer_sizes[1])
91 layers = zip(Ws, bs)
92

```

```

93 # First we compute the forward pass
94  $\theta$ vec = vcat([W[:]; b] for (W, b) in layers...)
95
96 ## Testing the gradient
97 y = 3.0 # We aim to have the final output be 3.0
98 grad_adjoint =  $\nabla$ L( $\theta$ vec, layer_sizes, u0; y=y)
99 idx = 4000
100 grad_fd = fd( $\theta$ vec, layer_sizes, u0, idx; e=1e-5, y=y)
101 @test grad_fd  $\approx$  grad_adjoint[idx] atol=1e-6
102
103 # Optimizing to get the desired output y
104 for _ = 1:1000
105     grad =  $\nabla$ L( $\theta$ vec, layer_sizes, u0; y=y)
106      $\theta$ vec .-= 0.001 .* grad
107 end
108 @test network_output( $\theta$ vec, layer_sizes, u0)  $\approx$  y
109 final_gradient =  $\nabla$ L( $\theta$ vec, layer_sizes, u0; y=y)
110 @test final_gradient  $\approx$  zeros(length( $\theta$ vec)) atol=1e-10

```

Bibliography

- [1] P. Bright, A. Edelman, and S. G. Johnson, “Matrix Calculus (for Machine Learning and Beyond).” [Online]. Available: <https://arxiv.org/abs/2501.14787>
- [2] 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.