

EECS 182 Deep Neural Networks

Fall 2026 Gireeja Ranade

Discussion 2

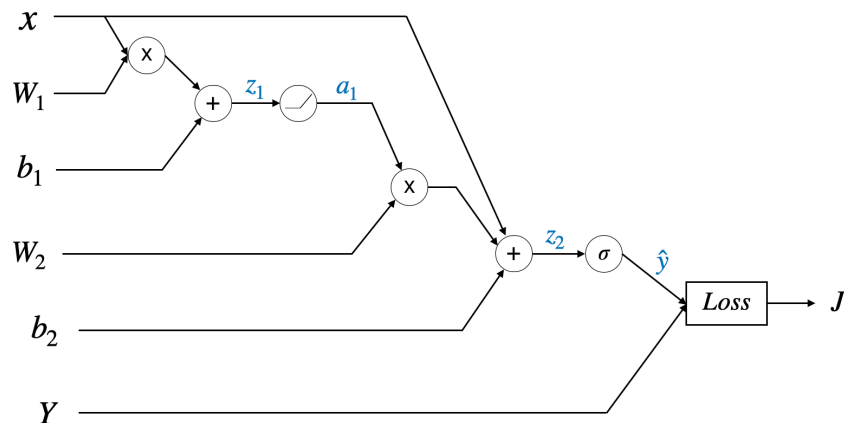
1. Computation Graph Review

One of the reasons why modern neural networks are growing much larger than ten years ago is due to the advent of automatic differentiation softwares, also known as *Autodiff*. Autodiff softwares can compute the gradients of any differentiable function by constructing a *computation graph* of such functions, which allows researchers to simply focus on building models with effective information propagation and not to worry about the complex back prop.

Here is the computation graph of an one hidden layer MLP with *skip connection*, which basically means adding the input to the output of some later layers (this is a commonly used design that we will cover in detail later in the course). σ denotes the sigmoid function and J is the final loss. Assume that we use binary cross-entropy as our loss function with a ground truth label y , ie. $loss = -y \ln \hat{y} - (1 - y) \ln(1 - \hat{y})$.

You may find this quantity useful:

- $\frac{\partial J}{\partial z_2} = \hat{y} - y$



(a) Express $\hat{y}$ as a function of x , W_1 , b_1 , W_2 and b_2 .

Solution:

$$\begin{aligned}
 \hat{y} &= \sigma(z_2) \\
 &= \sigma(W_2 a_1 + b_2 + x) \\
 &= \sigma(W_2 \cdot \text{ReLU}(W_1 x + b_1) + b_2 + x)
 \end{aligned} \tag{1}$$

(b) Compute the derivatives $\frac{\partial J}{\partial W_2}$, $\frac{\partial J}{\partial b_2}$.

Solution:

- $\frac{\partial J}{\partial W_2} = \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} = (\hat{y} - y) \cdot a_1$
- $\frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} = \hat{y} - y$

- (c) **Compute the derivatives** $\frac{\partial J}{\partial W_1}, \frac{\partial J}{\partial b_1}, \frac{\partial J}{\partial x}$.

Solution:

Note how $\frac{\partial J}{\partial x}$ relies on two streams of derivatives.

$$\begin{aligned} \bullet \frac{\partial J}{\partial W_1} &= \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} = (\hat{y} - y) \cdot W_2 \cdot \frac{\partial a_1}{\partial z_1} \cdot x \\ \bullet \frac{\partial J}{\partial b_1} &= \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} = (\hat{y} - y) \cdot W_2 \cdot \frac{\partial a_1}{\partial z_1}, \quad \text{where} \end{aligned}$$

$$\frac{\partial a_1}{\partial z_1} = \begin{cases} 1, & \text{if } z_1 > 0 \\ 0, & \text{if } z_1 < 0 \end{cases}$$

$$\bullet \frac{\partial J}{\partial x} = \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial x} + \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial x} = (\hat{y} - y) \cdot W_2 \cdot \frac{\partial a_1}{\partial z_1} \cdot W_1 + (\hat{y} - y)$$

- (d) **[Jump to this in xx:30 (skip previous ones if you haven't finished)]** What **intermediate variables do we need to cache** in this case? **Why is it more efficient to have an additional backward pass** on top of the forward pass for the computation of derivatives?

Solution:

We cache the input x , the output $\hat{y}$, and the intermediate variables z_1 , a_1 , and z_2 . These values were already computed during the forward pass and are needed to evaluate the local derivatives during the backward pass. For example, z_1 determines the ReLU gate $\mathbf{1}\{z_1 > 0\}$, while a_1 is needed for $\partial J / \partial W_2$.

The backward pass is more efficient because it computes each shared downstream derivative once and reuses it for every parameter that depends on it. In this graph, first compute

$$\delta_2 := \frac{\partial J}{\partial z_2} = \hat{y} - y.$$

The same δ_2 is then reused to obtain

$$\frac{\partial J}{\partial W_2} = \delta_2 a_1, \quad \frac{\partial J}{\partial b_2} = \delta_2.$$

Next, propagate it through the hidden path once:

$$\delta_1 := \frac{\partial J}{\partial z_1} = \delta_2 W_2 \mathbf{1}\{z_1 > 0\}.$$

This single value gives

$$\frac{\partial J}{\partial W_1} = \delta_1 x, \quad \frac{\partial J}{\partial b_1} = \delta_1, \quad \frac{\partial J}{\partial x} = \delta_1 W_1 + \delta_2,$$

where the two terms in $\partial J / \partial x$ are accumulated from the hidden path and the skip connection.

Without a backward pass, computing each requested derivative separately from J would repeatedly recompute common chain-rule factors such as δ_2 and the ReLU derivative. Reverse-mode backpropagation instead traverses the graph once in reverse order, caching and reusing these factors. Thus, for a scalar loss with many parameters, all parameter gradients can be computed in roughly the cost of one additional pass through the graph rather than one separate pass per parameter.

- (e) **Quantify the memory used by this AutoDiff framework.**

Every named quantity here is a scalar. Count the parameter; input/label; forward values; parameter gradients; and backward deltas. Each is stored in `float32` and occupies **4 bytes**.

For each storage group, compute its memory in bytes and sum the total peak memory.

Solution: Because each scalar uses 4 bytes, the memory accounting is

Storage group	Number of scalars	Memory
Parameters W_1, b_1, W_2, b_2	4	16 bytes
Input and label x, y	2	8 bytes
Forward values $z_1, a_1, z_2, \hat{y}, J$	5	20 bytes
Four parameter-gradient slots	4	16 bytes
Backward adjoints δ_1, δ_2	2	8 bytes
Total	17	68 bytes

Therefore, the total peak memory under this accounting is

$$16 + 8 + 20 + 16 + 8 = \boxed{68 \text{ bytes}}.$$

- (f) Under the memory accounting in part (e), can a naive reverse-mode AutoDiff implementation that retains every listed value perform direct gradient backpropagation with only **64 bytes** of memory? If not, briefly discuss alternatives that can still produce the exact gradients.

Solution: Not under the naive accounting from part (e): retaining all 17 float32 scalars requires $17 \times 4 = 68$ bytes, which exceeds the 64-byte limit.

One alternative is **liveness-aware memory reuse**. After the forward pass has reported J , backpropagation does not need the numerical value of J itself; it needs the adjoint initialized at that output. Releasing or reusing that 4-byte slot reduces the live storage from 68 to 64 bytes, so this toy example can fit while still performing exact reverse-mode differentiation.

If a larger graph still does not fit, use **checkpointing/rematerialization**: retain selected activations, discard the rest, and recompute discarded values when backward needs them. Other alternatives include offloading saved tensors to CPU memory, using lower-precision storage when numerically safe, and reducing the microbatch size while accumulating gradients. These approaches preserve AutoDiff, but trade additional computation, data movement, or numerical precision for lower peak memory.

2. Visualizing Derivatives

Consider a simple neural network that takes a scalar real input, has one hidden layer with $k = 2$ units and a ReLU nonlinearity for those units, and has an output linear (affine) layer.

We can algebraically write any function that it represents in matrix form as

$$y = W^{(2)} \max(\mathbf{0}, W^{(1)}x + \mathbf{b}^{(1)}) + b^{(2)}.$$

Here $x, y \in \mathbb{R}$, $W^{(1)} \in \mathbb{R}^{2 \times 1}$, $W^{(2)} \in \mathbb{R}^{1 \times 2}$, $\mathbf{b}^{(1)} \in \mathbb{R}^{2 \times 1}$, and $b^{(2)} \in \mathbb{R}$. The superscripts are indices, not exponents and the $\max$ given two vector arguments applies the $\max$ on corresponding pairs and returns a vector. Let's assume

$$W^{(1)} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} -1 \\ -2 \end{bmatrix}, \quad W^{(2)} = \begin{bmatrix} 0.5 & 2.0 \end{bmatrix}.$$

Calculate each requested partial derivative. Then sketch **seven separate figures**, one for each of the follow-

ing gradients as a function of x :

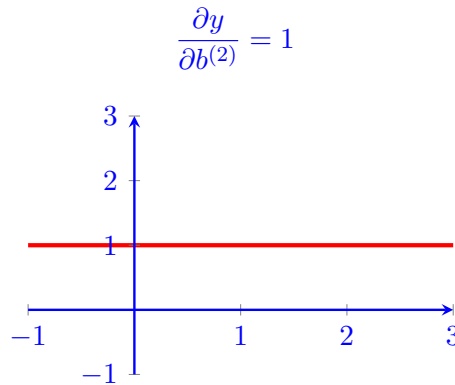
$$\frac{\partial y}{\partial b^{(2)}}, \quad \frac{\partial y}{\partial w_1^{(2)}}, \quad \frac{\partial y}{\partial w_2^{(2)}}, \quad \frac{\partial y}{\partial b_1^{(1)}}, \quad \frac{\partial y}{\partial b_2^{(1)}}, \quad \frac{\partial y}{\partial w_1^{(1)}}, \quad \frac{\partial y}{\partial w_2^{(1)}}.$$

Clearly label all discontinuities, kinks, and segment slopes.

Solution: The plots below use the assumed parameter values above. For hidden unit i , the ReLU transition occurs at $x = -b_i^{(1)}/w_i^{(1)}$: specifically, at $x = 0.5$ for unit 1 and at $x = 1$ for unit 2.

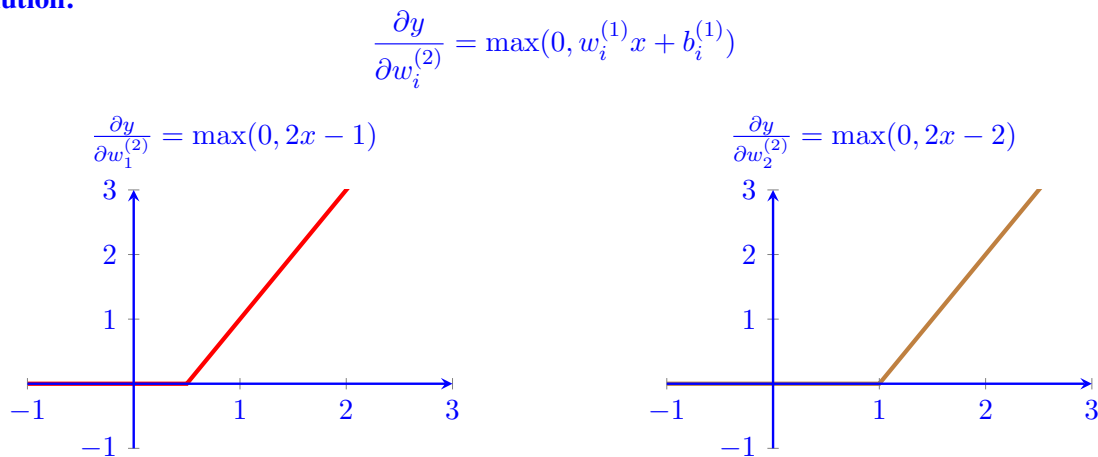
- (a) Calculate $\frac{\partial y}{\partial b^{(2)}}(x)$ and draw its figure.

Solution:



- (b) Calculate $\frac{\partial y}{\partial w_1^{(2)}}(x)$ and $\frac{\partial y}{\partial w_2^{(2)}}(x)$, and draw one separate figure for each gradient.

Solution:

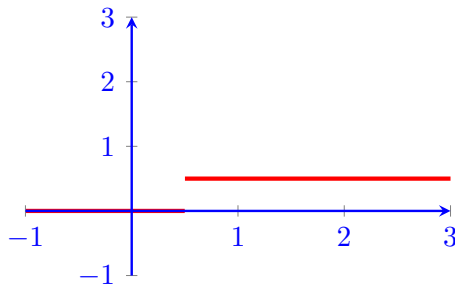


- (c) Calculate $\frac{\partial y}{\partial b_1^{(1)}}(x)$ and $\frac{\partial y}{\partial b_2^{(1)}}(x)$, and draw one separate figure for each gradient.

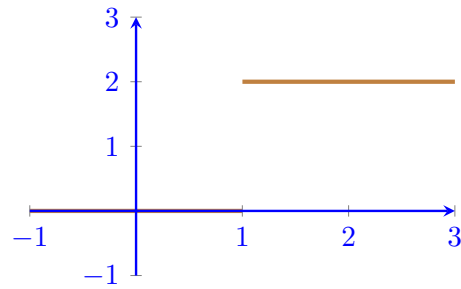
Solution:

$$\frac{\partial y}{\partial b_i^{(1)}} = \begin{cases} w_i^{(2)}, & \text{if } w_i^{(1)}x + b_i^{(1)} > 0 \\ 0, & \text{if } w_i^{(1)}x + b_i^{(1)} < 0 \end{cases}$$

$$\frac{\partial y}{\partial b_1^{(1)}} = \begin{cases} 0, & x < 0.5 \\ 0.5, & x > 0.5 \end{cases}$$



$$\frac{\partial y}{\partial b_2^{(1)}} = \begin{cases} 0, & x < 1 \\ 2, & x > 1 \end{cases}$$

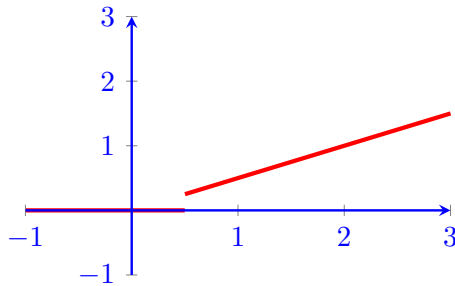


- (d) Calculate $\frac{\partial y}{\partial w_1^{(1)}}(x)$ and $\frac{\partial y}{\partial w_2^{(1)}}(x)$, and draw one separate figure for each gradient.

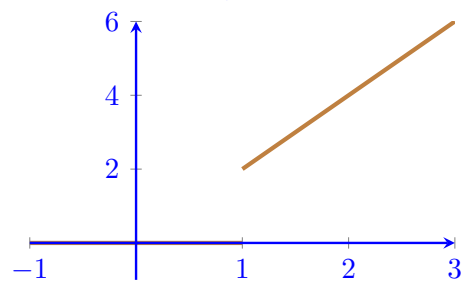
Solution:

$$\frac{\partial y}{\partial w_i^{(1)}} = \begin{cases} w_i^{(2)} x, & \text{if } w_i^{(1)} x + b_i^{(1)} > 0 \\ 0, & \text{if } w_i^{(1)} x + b_i^{(1)} < 0 \end{cases}$$

$$\frac{\partial y}{\partial w_1^{(1)}} = \begin{cases} 0, & x < 0.5 \\ 0.5x, & x > 0.5 \end{cases}$$



$$\frac{\partial y}{\partial w_2^{(1)}} = \begin{cases} 0, & x < 1 \\ 2x, & x > 1 \end{cases}$$



- (e) When you fine-tune your network, you observe a coupled motion in your parameters as gradient descent adjusts them. In particular, whenever the bias $b_1^{(1)}$ on the first hidden unit moves up by a small amount $\alpha > 0$, the weight $w_2^{(1)}$ on the second hidden unit moves up by 2α . Design a 1-dimensional synthetic feature (function of x) showing how this coupled parameter movement affects output y by this synthetic feature.

Hint: Fix all the parameters in the network except for $b_1^{(1)}$ and $w_2^{(1)}$ and consider how the output will change due to small changes in these two parameters. The quantities $\frac{\partial f}{\partial \theta}(x)$ are what we have just plotted in the previous parts and are features (functions of x) of the neural network locally. We can approximate the output using

$$\theta = \begin{bmatrix} w_2^{(1)} \\ b_1^{(1)} \end{bmatrix}, \quad \Delta\theta = \alpha \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad (2)$$

$$y = f_{\theta_0 + \Delta\theta}(x) \approx f_{\theta_0}(x) + \frac{\partial f}{\partial \theta}(x) \Delta\theta. \quad (3)$$

Solution: Let $\alpha > 0$ be a small amount. The observed coupled parameter increments are

$$\Delta\theta = \begin{bmatrix} \Delta w_2^{(1)} \\ \Delta b_1^{(1)} \end{bmatrix} = \begin{bmatrix} 2\alpha \\ \alpha \end{bmatrix} = \alpha \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

Using the first-order approximation from the hint, the corresponding change in the output is

$$\Delta y(x) \approx \begin{bmatrix} \frac{\partial y}{\partial w_2^{(1)}}(x) & \frac{\partial y}{\partial b_1^{(1)}}(x) \end{bmatrix} \Delta\theta \quad (4)$$

$$= \alpha \left(2 \frac{\partial y}{\partial w_2^{(1)}}(x) + \frac{\partial y}{\partial b_1^{(1)}}(x) \right). \quad (5)$$

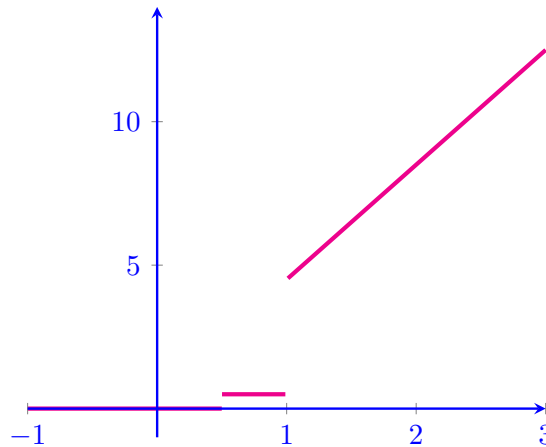
The factor α controls how far the parameters move, while the fixed vector $[2 \ 1]^\top$ controls their shared direction. Therefore, the expression in parentheses is the one-dimensional synthetic feature describing how that coupled direction changes the output:

$$s(x) = 2 \frac{\partial y}{\partial w_2^{(1)}}(x) + \frac{\partial y}{\partial b_1^{(1)}}(x) \quad (6)$$

$$= 2 (2x \mathbf{1}\{x > 1\}) + 0.5 \mathbf{1}\{x > 0.5\} \quad (7)$$

$$= \begin{cases} 4x + 0.5, & x > 1, \\ 0.5, & 0.5 < x < 1, \\ 0, & x < 0.5. \end{cases} \quad (8)$$

At $x = 0.5$ and $x = 1$, the relevant ReLU derivative is not uniquely defined; the plot therefore shows jumps at those locations. Moreover, $\Delta y(x) \approx \alpha s(x)$, so $s(x)$ captures the shape of the output change independently of the particular small step size α .



Why calculate this synthetic feature? If the two parameters moved independently, their local effect on the output would be

$$\Delta y(x) \approx \Delta w_2^{(1)} \frac{\partial y}{\partial w_2^{(1)}}(x) + \Delta b_1^{(1)} \frac{\partial y}{\partial b_1^{(1)}}(x),$$

with two independently chosen coefficients. Coupling means those coefficients cannot be chosen independently: here they must satisfy $\Delta w_2^{(1)} = 2\Delta b_1^{(1)}$. Thus every allowed update lies on the one-dimensional line $\Delta\theta = \alpha[2 \ 1]^\top$ in the two-dimensional parameter space. The network maps this single parameter-space direction to the single function

$$s(x) = 2\frac{\partial y}{\partial w_2^{(1)}}(x) + \frac{\partial y}{\partial b_1^{(1)}}(x),$$

and every such coupled update changes the output by $\Delta y(x) \approx \alpha s(x)$. We calculate $s(x)$ so that one feature describes the entire recurring two-parameter movement.

Contributors:

- Matteo Guarrera.
- Mert Cemri.
- Sizhe Chen.
- Kevin Li.
- Saagar Sanghavi.
- Anant Sahai.
- Qiyang Li.