Vectorized Formulations of Deep Networks

Sui Jiet Tay CS.JIET@GMAIL.COM

Keywords: deep neural networks; feedforward neural networks; multilayer perceptrons (MLPs); fully connected layers; forward pass; backpropagation; multivariate chain rule; vectorization; matrix calculus; matrix-vector products; outer products; Jacobians; diagonal Jacobians; coordinate-wise (entrywise) activations; Kronecker delta; 3-tensor Jacobians; tensor contraction; parameter gradients; weight gradients; bias gradients; pre-activations (logits); post-activations; computational graphs; automatic differentiation; softmax; cross-entropy loss

Contents

1. Motivation

How fast can you backpropagate by hand? It turns out it is essentially as fast as you can evaluate the required algebraic operators — mainly matrix multiplications, matrix-calculus derivatives, and entrywise function transformations —*correctly*.

A complex neural network computational graph can look intimidating, even tracing a few layers yields a large expression that is difficult to manage systematically. However, it turns out we can still express the network in a closed analytical form for both inference and backpropagation, which makes theoretical analysis significantly more tractable.

In this writing, I show how, with the tools of linear algebra, we can derive clean expressions for backpropagation. In particular, training a neural network *by hand* boils down to knowing (i) the evaluated formulas for the four *global* gradients we care about, together with (ii) the corresponding *local* Jacobians that allow those global gradients to be propagated backward via the chain rule. With these in hand, we can execute one complete update cycle entirely by hand: one forward pass, zeroing out the gradients, one backward pass, and one optimization step. Repeating this cycle over the entire dataset constitutes one epoch, and repeating across epochs is what we call training for multiple epochs.

I also provide code snippets showing how to extract the *exact* vectorized gradients used here from PyTorch, which can serve as a sanity check as you begin to compute them by hand.

2. Notations

(Dimensions / widths)

$d_0, \cdots, d_L \in \mathbb{N}$ (input width, hidden widths, output width)

(Layer index)

$\ell \in \{1, \ldots, L\}$ (affine/activation layer index)

(Neuron / coordinate indices)

$j \in \{1, \ldots, d_\ell\}$ (neuron index (coordinate) in layer ℓ)

$k \in \{1, \ldots, d_{\ell-1}\}$ (input-coordinate index to layer ℓ)

$m \in \{1, \ldots, d_\ell\}$ (row (output-neuron) index of $W^{(\ell)}$)

(Vectors)

$\mathbf{x} \in \mathbb{R}^{d_0}$ (input vector)

$\mathbf{h}^{(0)} := \mathbf{x}$

$\mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell}$ (**pre-activation vector** at layer ℓ)

$\mathbf{h}^{(\ell)} \in \mathbb{R}^{d_\ell}$ (**post-activation vector** at layer ℓ)

$\mathbf{z}_j^{(\ell)} \in \mathbb{R}$ (j-th pre-activation scalar in layer ℓ)

$\mathbf{h}_j^{(\ell)} \in \mathbb{R}$ (j-th post-activation scalar in layer ℓ)

(Parameters)

$W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ (weight matrix at layer ℓ (rows index neurons))

$\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}$ (bias vector at layer ℓ)

$W_{j,:}^{(\ell)} \in \mathbb{R}^{1 \times d_{\ell-1}}$ (weight row-vector of neuron j in layer ℓ)

$W_{m,k}^{(\ell)} \in \mathbb{R}$ (entry in row m, column k of $W^{(\ell)}$)

$\boldsymbol{\theta} := \left(W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}, \ldots, W^{(L)}, \mathbf{b}^{(L)}\right)$

(ordered sequence of weights and biases for affine layers $\ell \in \{1, \ldots, L\}$)

(Activations)

$\sigma^{(\ell)} : \mathbb{R} \to \mathbb{R}$ (scalar nonlinearity used in layer ℓ)

$\Sigma^{(\ell)} : \mathbb{R}^{d_\ell} \to \mathbb{R}^{d_\ell}$ (coordinatewise extension of $\sigma^{(\ell)}$)

$\Sigma_j^{(\ell)} : \mathbb{R}^{d_\ell} \to \mathbb{R}$ (j-th coordinate map of $\Sigma^{(\ell)}$)

$\Sigma_j^{(\ell)}\big(\mathbf{z}^{(\ell)}\big) = \sigma^{(\ell)}\big(\mathbf{z}_j^{(\ell)}\big)$ (coordinatewise activation at neuron j in layer ℓ)

(Derivative / indexing conventions)

$\dfrac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{d_\ell}$ (gradient w.r.t. the vector $\mathbf{z}^{(\ell)}$ (viewed as a column vector))

$\Big(\dfrac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\Big)_j = \dfrac{\partial \mathcal{L}}{\partial \mathbf{z}_j^{(\ell)}}$ (j-th coordinate of the gradient)

$\dfrac{\partial \mathcal{L}}{\partial W^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ (matrix of partial derivatives w.r.t. entries of $W^{(\ell)}$)

$\Big(\dfrac{\partial \mathcal{L}}{\partial W^{(\ell)}}\Big)_{m,k} = \dfrac{\partial \mathcal{L}}{\partial W_{m,k}^{(\ell)}}$ (entrywise meaning of the weight gradient)

$$\Big(\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}\Big)_{m,k} = \sum_{j=1}^{d_\ell} \Big(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\Big)_j \Big(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}\Big)$$ (chain rule expanded in coordinates)

(Variant notation) $\Big(\dfrac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\Big)_{j,m,k} := \dfrac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}$

(3-index object: output coordinate j vs. parameter indices (m,k))

(Variant notation) $\Big(\dfrac{\partial \mathbf{z}^{(\ell)}}{\partial W_{m,k}^{(\ell)}}\Big)_j := \dfrac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}$

(treat (m,k) as fixed; index only the output coordinate j)

Remark (why the specific shape convention for the weight matrix $W^{(\ell)}$). We choose $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ so that indexing a neuron is immediate: the j-th pre-activation is the dot product of the j-th weight row with the previous-layer activation vector,

$$\mathbf{z}_j^{(\ell)} = W_{j,:}^{(\ell)}\, \mathbf{h}^{(\ell-1)} = \Big\langle \big(W_{j,:}^{(\ell)}\big)^\top, \mathbf{h}^{(\ell-1)} \Big\rangle,$$

which avoids repeatedly writing transposes when passing between the layer map $W^{(\ell)}\mathbf{h}^{(\ell-1)}$ and the neuron-wise dot-product view.

3. Practical setup

3.1. Backpropagation by hand (one training iteration)

Goal. Given one input $\mathbf{x} = \mathbf{h}^{(0)} \in \mathbb{R}^{d_0}$, target $\mathbf{Y}$, and parameters $\{W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}},\ \mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}\}_{\ell=1}^{L}$, compute $\left\{ \frac{\partial \mathcal{L}}{\partial W^{(\ell)}},\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} \right\}_{\ell=1}^{L}$ and then take one gradient step.

One full update cycle (one training iteration).

1. **Forward pass** For $\ell = 1, \ldots, L$:

$$\mathbf{z}^{(\ell)} := W^{(\ell)} \mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$
$$\mathbf{h}^{(\ell)} := \begin{cases} \Sigma^{(\ell)}\big(\mathbf{z}^{(\ell)}\big), & \ell < L \\ \mathbf{z}^{(L)}, & \ell = L \quad \text{(logits (no hidden-layer activation))} \end{cases}$$

   Then compute the loss

$$\mathcal{L} := \mathcal{L}\big(\mathbf{z}^{(L)}, \mathbf{Y}\big)$$

   What to track: $\mathbf{h}^{(\ell-1)}$ and $\mathbf{z}^{(\ell)}$ for every layer.

2. **Initialize the backward pass at the output.** Compute

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} \qquad \text{directly from the loss definition.}$$

3. **Output layer gradients.**

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}}$$
$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} \right) \big(\mathbf{h}^{(L-1)}\big)^\top$$

4. **Backpropagate through hidden layers ($\ell = L-1, \ldots, 1$).** For $\ell = L-1, L-2, \ldots, 1$, compute in this order:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} = \left(W^{(\ell+1)}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell+1)}}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} = \left(\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$$
$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right) \left(\mathbf{h}^{(\ell-1)}\right)^\top$$

   Remark. The Jacobian $\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}$ is the coordinatewise derivative of the activation (typically a diagonal matrix).

5. **Optimization step (one gradient step).** For $\ell = 1, \ldots, L$:

$$W^{(\ell)} \leftarrow W^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(\ell)}}$$
$$\mathbf{b}^{(\ell)} \leftarrow \mathbf{b}^{(\ell)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}$$

6. **Repeat.** Repeating this update cycle over all training examples (or mini-batches) forms one epoch.

That's all you need!

Note. When calculating by hand, we can expect small numerical differences between hand calculations and PyTorch outputs due to finite-precision floating-point arithmetic and rounding at intermediate steps. By default, PyTorch performs computations in `float32`, with specific number of significant decimal digits of precision. Similarly, any rounding scheme we choose during hand calculation will incur the same penalty. This means that each arithmetic operation we do may introduce a small *rounding error*, and these errors can accumulate along the chain of operations during backpropagation.

However, even if our hand calculations round intermediate quantities to the same number of significant digits, PyTorch may still produce slightly different numerical results. Two reasons this can happen are that it may use different *reduction conventions* (e.g., summing versus averaging over a batch), and that in parallel settings it may evaluate floating-point operations in different orders.

Since rounding occurs at each floating-point operation, changing the evaluation order changes where rounding happens, which can slightly change the final result. These implementation choices are inherent to PyTorch's design and reflect how the library optimizes for speed and scalability during training and inference.

Consequently, floating-point results can be non-deterministic in practice, because floating-point addition is *not associative*. That is, changing the order of evaluation can change when rounding occurs, and thus change the final answer in the least significant digits. If you have taken a course on parallelism and worked through reductions by hand (or traced multithreaded code), you have likely

seen that there are many valid execution orders, which can yield slightly different floating-point results from run to run. Hence, when comparing hand calculations to PyTorch, the right expectation is agreement *up to floating-point rounding error*, rather than exact equality.

4. Mathematical Formulation of a Layer

Formally, it is helpful to separate a deep network into three conceptually different parts. First, the *input layer* is simply the data and introduces no learnable parameters. Next, the *hidden layers* are the learnable affine transformations, each typically consisting of an affine map followed by a nonlinearity. Finally, the *output head* is a task-dependent map applied to the final logits (e.g., identity for regression, sigmoid for binary classification, or softmax for multiclass classification).

4.1. Input layer

Note (common misconception). The *input layer* is often illustrated as an independent layer in network diagrams. But unlike the hidden and output layers, it is not a learnable map: it is simply the data, i.e., $\mathbf{h}^{(0)} := \mathbf{x}$.

4.2. Hidden layer

4.2.1. AFFINE MAP (PRE-ACTIVATION AT LAYER ℓ)

Affine map/ Pre-activation

$$A^{(\ell)} : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}$$
$$A^{(\ell)}(\mathbf{u}) := W^{(\ell)}\mathbf{u} + \mathbf{b}^{(\ell)} \qquad (\mathbf{u} \in \mathbb{R}^{d_{\ell-1}})$$

Shape convention. Let $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ and $\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}$.

Neuron-weight convention. For each $j \in \{1, \dots, d_\ell\}$, the incoming weight vector for neuron j is the j-th *row* of $W^{(\ell)}$ (viewed as a row vector), namely

$$W^{(\ell)}_{j,:} \in \mathbb{R}^{1 \times d_{\ell-1}}$$

Equivalently, if we prefer a column-vector view of the same weights, we write

$$\left(W^{(\ell)}_{j,:}\right)^\top \in \mathbb{R}^{d_{\ell-1}}$$

(We may refer to $\left(W^{(\ell)}_{j,:}\right)^\top$ as the weight vector of neuron j.)

Remark (potential ambiguity). Introducing a shorthand such as $\mathbf{w}^{(\ell)}_j := \left(W^{(\ell)}_{j,:}\right)^\top$ can be dangerous: the subscript j is very often read as "the j-th *coordinate* of a vector" (a scalar), whereas here j labels an entire *neuron weight vector*. If these two meanings are silently mixed, dimension checks can fail without warning. For this reason, it is often safer to keep the row/transpose notation explicit, i.e., to write $W^{(\ell)}_{j,:} \in \mathbb{R}^{1 \times d_{\ell-1}}$ or $\left(W^{(\ell)}_{j,:}\right)^\top \in \mathbb{R}^{d_{\ell-1}}$.

For any hidden layer output $\mathbf{h}^{(\ell-1)}$

$$\begin{aligned}
&\mathbf{z}^{(\ell)} := A^{(\ell)}\big(\mathbf{h}^{(\ell-1)}\big) \\
&\mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell} \\
&\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \qquad \big(\mathbf{h}^{(\ell-1)} \in \mathbb{R}^{d_{\ell-1}}\big)
\end{aligned}$$

Expanded component form (four equivalent views).

$$\begin{aligned}
\mathbf{z}^{(\ell)} &= W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)} \\
&= \begin{bmatrix} W^{(\ell)}_{1,:}\,\mathbf{h}^{(\ell-1)} \\ \vdots \\ W^{(\ell)}_{d_\ell,:}\,\mathbf{h}^{(\ell-1)} \end{bmatrix} + \mathbf{b}^{(\ell)} \\
&= \begin{bmatrix} \Big\langle \big(W^{(\ell)}_{1,:}\big)^\top, \mathbf{h}^{(\ell-1)} \Big\rangle \\ \vdots \\ \Big\langle \big(W^{(\ell)}_{d_\ell,:}\big)^\top, \mathbf{h}^{(\ell-1)} \Big\rangle \end{bmatrix} + \mathbf{b}^{(\ell)} \\
&= \sum_{k=1}^{d_{\ell-1}} h^{(\ell-1)}_k\, W^{(\ell)}_{:,k} \;+\; \mathbf{b}^{(\ell)} \quad \text{(linear combination formulation (uncommon interpretation))}
\end{aligned}$$

For the last formulation, it applies the matrix–vector product definition, but now read as a column-wise linear combination; this can be confusing since the earlier two displays emphasize a *row-wise* dot-product (neuron) view.

Index-wise notation (four equivalent scalars). For each $j \in \{1, \dots, d_\ell\}$,

$$\begin{aligned}
\mathbf{z}^{(\ell)}_j &= \big(W^{(\ell)}\mathbf{h}^{(\ell-1)}\big)_j + \mathbf{b}^{(\ell)}_j \\
&= W^{(\ell)}_{j,:}\,\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}_j \\
&= \Big\langle \big(W^{(\ell)}_{j,:}\big)^\top, \mathbf{h}^{(\ell-1)} \Big\rangle + \mathbf{b}^{(\ell)}_j
\end{aligned}$$

Notice that each layer contains a weight matrix, which can be interpreted as a "**learnable *linear map***",

$$W^{(\ell)}$$

Hence, we may study it using the same tools as in linear algebra: we can define its kernel $Ker\big(W^{(\ell)}\big)$ and image $Im\big(W^{(\ell)}\big)$, discuss bases for these subspaces, and apply the singular value decomposition (SVD) to analyze its principal directions of stretch (i.e., its dominant input and output directions and their associated singular values).
For the bias offset,

$$\mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}$$

Unlike $W^{(\ell)}$, the bias is not a linear map; it is a translation vector. In other words, together

$$A^{(\ell)}(\mathbf{u}) \coloneqq W^{(\ell)}\mathbf{u} + \mathbf{b}^{(\ell)}$$

defines an *affine* map, not a linear one.
We can study $\mathbf{b}^{(\ell)}$ by viewing it as the layer's **learned *offset***. Since it adds directly to the pre-activation, it sets the baseline pre-activation when the input is zero:

$$\mathbf{z}^{(\ell)}\big|_{\mathbf{h}^{(\ell-1)}=\mathbf{0}} = \mathbf{b}^{(\ell)}$$

It also acts as to shift each neuron's "firing" threshold (e.g., for ReLU, it shifts where the unit becomes active). Recall that this is the *evaluation-under-a-condition* bar $\big(\big|\big)$ notation (widely used in calculus when evaluating an antiderivative at the endpoints of a definite integral). Here it means: first write the symbolic expression for $\mathbf{z}^{(\ell)}$, and then evaluate it at the specific input $\mathbf{h}^{(\ell-1)} = \mathbf{0}$. We write it this way to emphasize that $\mathbf{z}^{(\ell)}$ is defined as a function of a variable input $\mathbf{h}^{(\ell-1)}$, not as a single constant value, and avoid writing the full expression.

4.2.2. ACTIVATION MAP (POST-ACTIVATION AT LAYER ℓ)

Convention (coordinates). Vectors are denoted with $\cdot$. For a vector $\mathbf{x} \in \mathbb{R}^d$, the notation $\mathbf{x}_i \in \mathbb{R}$ denotes its i-th scalar coordinate.

(Vector activation as a coordinatewise lift of a scalar nonlinearity). Let $\sigma^{(\ell)} : \mathbb{R} \to \mathbb{R}$ be the scalar activation function at layer ℓ. Define its coordinatewise extension

$$\Sigma^{(\ell)} : \mathbb{R}^{d_\ell} \to \mathbb{R}^{d_\ell}$$

For any $\mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell}$, set

$$\mathbf{h}^{(\ell)} := \Sigma^{(\ell)}\big(\mathbf{z}^{(\ell)}\big) \in \mathbb{R}^{d_\ell}$$

where, by definition,

$$\begin{aligned}\Sigma^{(\ell)}\big(\mathbf{z}^{(\ell)}\big) &:= \begin{bmatrix} \Sigma_1^{(\ell)}\big(\mathbf{z}_1^{(\ell)}\big) \\ \vdots \\ \Sigma_{d_\ell}^{(\ell)}\big(\mathbf{z}_{d_\ell}^{(\ell)}\big) \end{bmatrix} \\ &= \begin{bmatrix} \sigma^{(\ell)}\big(\mathbf{z}_1^{(\ell)}\big) \\ \vdots \\ \sigma^{(\ell)}\big(\mathbf{z}_{d_\ell}^{(\ell)}\big) \end{bmatrix} \qquad \big(\mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell}\big)\end{aligned}$$

4.2.3. LAYER MAP (ACTIVATION $\circ$ AFFINE COMPOSITE LAYER MAP AT LAYER ℓ)

$$T^{(\ell)} : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}, \qquad T^{(\ell)} := \Sigma^{(\ell)} \circ A^{(\ell)}$$

$$
\begin{aligned}
\mathbf{h}^{(\ell)} &:= T^{(\ell)}\big(\mathbf{h}^{(\ell-1)}\big) \in \mathbb{R}^{d_\ell} \\
\mathbf{h}^{(\ell)} &= \Sigma^{(\ell)}\big(\mathbf{z}^{(\ell)}\big) \\
\mathbf{h}^{(\ell)} &= \Sigma^{(\ell)}\big(A^{(\ell)}(\mathbf{h}^{(\ell-1)})\big) \\
\mathbf{h}^{(\ell)} &= \Sigma^{(\ell)}\Big(W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\Big)
\end{aligned}
$$

Index-wise notation. For each $j \in \{1, \dots, d_\ell\}$,

$$
\mathbf{h}_j^{(\ell)} = \sigma^{(\ell)}\big(\mathbf{z}_j^{(\ell)}\big) \qquad \big(\mathbf{z}^{(\ell)} := W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\big)
$$

4.3. Output layer

Note (common misconception). By convention, the output layer is also drawn as a layer of neurons in network diagrams. However, it is often treated differently from the input layer and the hidden layers in two ways: (i) it is typically the final *learnable* affine map, and (ii) it typically does *not* apply the hidden-layer activation. Concretely, the *output layer* is the final affine map $A^{(L)}$, which produces the *logits* $\mathbf{z}^{(L)}$ and contains the learnable parameters (weights and biases) of the final layer. In other words, while hidden layers use the composite map $T^{(\ell)} = \Sigma^{(\ell)} \circ A^{(\ell)}$, the output layer is often taken to be $A^{(L)}$ itself.

5. The Multilayer Perceptron (MLP) / Fully-Connected Feedforward Neural Network (L-Layer Map)

Convention (coordinates). Vectors are denoted with $\cdot$. For $\mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell}$, the notation $\mathbf{z}_j^{(\ell)} \in \mathbb{R}$ denotes the j-th scalar coordinate (neuron output) of $\mathbf{z}^{(\ell)}$

Nested composition (evaluation form).

$$
f_{\boldsymbol{\theta}}(\mathbf{x}) := T^{(L)}\Big(T^{(L-1)}\big(\cdots T^{(2)}(T^{(1)}(\mathbf{x}))\cdots\big)\Big)
$$

Recursive definition (forward pass)

$$
\begin{aligned}
\mathbf{h}^{(0)} &:= \mathbf{x}, \\
\mathbf{h}^{(\ell)} &:= T^{(\ell)}\big(\mathbf{h}^{(\ell-1)}\big) \qquad \big(\ell \in \{1, \dots, L\}\big), \\
f_{\boldsymbol{\theta}}(\mathbf{x}) &:= \mathbf{h}^{(L)}
\end{aligned}
$$

Composition of maps (definition).

$$
f_{\boldsymbol{\theta}}(\mathbf{x}) := \big(T^{(L)} \circ T^{(L-1)} \circ \cdots \circ T^{(1)}\big)(\mathbf{x})
$$

5.0.1. CODE THAT PERFORMS A FORWARD PASS

```
import torch
```

```python
import torch.nn as nn


class MLP(nn.Module):
    def __init__(self, activation: nn.Module = nn.ReLU(), use_bias: bool = True):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(5, 3, bias=use_bias),
            activation,
            nn.Linear(3, 4, bias=use_bias),
            activation,
            nn.Linear(4, 3, bias=use_bias),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)


model = MLP()
X = torch.randn(8, 5)   # X \in R^{8 x 5}: batch of 8 inputs, each in R^5
Y = model(X)            # Y \in R^{8 x 3}: batch of 8 outputs, each in R^3
```

6. Output head

An *output head* is a (task-dependent) function that maps the network's final pre-activation vector, often called the *logits* $\mathbf{z}^{(L)}$, to the desired output space.

Depending on the task, one may apply a specific head to $\mathbf{z}^{(L)}$ (e.g., the identity map if its a regression task, a sigmoid for binary classification, or a softmax for multiclass classification).

6.1. Softmax

Convention The final pre-activation vector is $\mathbf{z}^{(L)} \in \mathbb{R}^{d_L}$.

$$p_{\boldsymbol{\theta}}(c \mid \mathbf{x}) := \frac{\exp\big(\mathbf{z}_c^{(L)}\big)}{\sum_{j=1}^{d_L} \exp\big(\mathbf{z}_j^{(L)}\big)} \qquad \big(c \in \{1, \dots, d_L\}\big)$$

7. Post-hoc calibration

This is a somewhat specialized topic in settings where confidence scores are interpreted as probabilities and must be as accurate as possible. This domain of research designs a *post-hoc calibration map*, which is an additional transformation applied to a model's existing output scores (e.g., logits

or softmax probabilities) in order to make the reported probabilities "better aligned" with observed empirical frequencies.

From the perspective of backpropagation, such a calibration layer is simply another map composed on top of the network. As long as this map is differentiable (or, more generally, differentiable almost everywhere), we can backpropagate through it by the chain rule. Even if the calibration rule is non-smooth at a small set of points (e.g., piecewise-linear operations), gradients still propagate using the derivative wherever it exists. The main exception is truly discontinuous post-processing (e.g., a hard `argmax` or hard thresholding), which breaks ordinary gradient-based backpropagation unless one replaces it with a smooth relaxation or a surrogate-gradient method.

8. Loss function

8.1. Cross-entropy loss

$$\ell\big(\mathbf{x}, c\big) := -\log\Big(p_{\boldsymbol{\theta}}(c \mid \mathbf{x})\Big) \qquad \big(c \in \{1, \dots, d_L\}\big)$$

9. Backpropagation

9.1. Motivation and setup of backpropagation

We need to compute a small set of derivatives (the *backpropagation ingredients*) that together produce the gradients of the learnable parameters. Conceptually, the network contains the objects

$$\mathcal{L},\ \mathbf{z}^{(1)}, \dots, \mathbf{z}^{(L)},\ \mathbf{h}^{(0)}, \dots, \mathbf{h}^{(L)},\ W^{(1)}, \dots, W^{(L)},\ \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(L)},\ \mathbf{x}$$

For training, the final quantities we want are the *parameter gradients*

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}, \qquad \ell \in \{1, \dots, L\},$$

since these are exactly what gradient-based optimization uses to update the weights and biases.

However, to compute $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}$, we also compute *intermediate (hidden-variable) gradients* that propagate sensitivity information backward through the network:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}}, \qquad \ell \in \{1, \dots, L\}$$

These are not parameters we optimize directly. Instead, they are the *carriers* that move gradient information backward via the chain rule.

Concretely, once $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$ is known, the layer-ℓ parameter gradients follow immediately from the affine map $\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$:

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} \qquad \text{are obtained by combining} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \text{ with } \mathbf{h}^{(\ell-1)}$$

This motivates the "four derivatives" viewpoint (per layer): two are the *targets* (parameter gradients), and the other two are the *links* (hidden-variable gradients) that allow us to chain the computation.

In order to compute these *global* gradients, we also need a small set of *local* derivatives (local Jacobians) coming from the two building blocks of each layer: the activation map $\mathbf{h}^{(\ell)} = \Sigma^{(\ell)}(\mathbf{z}^{(\ell)})$ and the affine map $\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$. In our transpose-backprop convention, the local derivatives we will repeatedly use are

$$\underbrace{\Big(\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}\Big)^{\top}}_{\text{local activation Jacobian}} \qquad \text{and} \qquad \Big(\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}}\Big)^{\top},$$

together with the affine-map Jacobians

$$\underbrace{\Big(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\Big)}_{\text{local (3-tensor) Jacobian, immediately contracted}} \qquad \text{and} \qquad \Big(\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}}\Big)^{\top}$$

We will compute each of these local objects explicitly (or, in the case of $\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}$, we will compute its contraction form directly), and then assemble the global gradients by systematic applications of the chain rule.

In principle, one could also compute $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$, since $\mathbf{x} = \mathbf{h}^{(0)}$ is an input to the computational graph. However, in standard supervised training we do *not* update the data $\mathbf{x}$, so $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$ is usually omitted. (It becomes relevant in other settings, e.g. adversarial examples, input optimization, and saliency maps.)

Why do we compute these derivatives at all? A partial derivative such as $\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}$ formalizes the following sensitivity statement: if we perturb the parameter $W^{(\ell)}$ by an infinitesimally small amount, how does the scalar loss $\mathcal{L}$ change to first order? This is the same core idea that appears throughout classical optimization (including convex optimization in statistics): use local sensitivity to decide a direction that decreases the objective.

What is "new" in neural networks is not the idea of gradients, but the *scalability of the chain rule*. In a single-layer linear model, the objective depends on one affine map. In a deep network, the loss depends on a composition of many affine maps and nonlinearities. Backpropagation is simply the systematic application of the multivariate chain rule through this layered composition, allowing us to compute all required intermediate gradients and, from them, all parameter gradients efficiently even when the network has many layers and many neurons per layer

Convention. $\mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell}$ and $\mathbf{h}^{(\ell-1)} \in \mathbb{R}^{d_{\ell-1}}$ are vectors that collect neuron outputs. Thus, $\mathbf{z}_i^{(\ell)} \in \mathbb{R}$ denotes the *scalar* output (pre-activation) of the i-th neuron in layer ℓ, and $\mathbf{h}_j^{(\ell-1)} \in \mathbb{R}$ denotes the *scalar* output (activation) of the j-th neuron in layer $\ell - 1$. Equivalently, $\mathbf{z}_i^{(\ell)} = \big(\mathbf{z}^{(\ell)}\big)_i$ and $\mathbf{h}_j^{(\ell-1)} = \big(\mathbf{h}^{(\ell-1)}\big)_j$.

(Weight-shape convention for this section). We take

$$W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}, \qquad \mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell},$$

so that the affine map is

$$\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

In particular, the j-th neuron weight row is $W_{j,:}^{(\ell)} \in \mathbb{R}^{1\times d_{\ell-1}}$

Convention (Jacobian layout). For a vector-valued map $\mathbf{u} : \mathbb{R}^n \to \mathbb{R}^m$, we define the Jacobian

$$\frac{\partial \mathbf{u}}{\partial \mathbf{v}} \in \mathbb{R}^{m\times n}$$

by the entrywise rule

$$\left(\frac{\partial \mathbf{u}}{\partial \mathbf{v}}\right)_{i,j} = \frac{\partial \mathbf{u}_i}{\partial \mathbf{v}_j}, \qquad i \in \{1,\dots,m\},\ \ j \in \{1,\dots,n\},$$

where $\mathbf{v} \in \mathbb{R}^n$ is the input vector and $\mathbf{u}(\mathbf{v}) \in \mathbb{R}^m$ is the output vector.

9.2. Code that performs a backward pass

```python
import torch

loss = ... # real scalar-valued loss (a torch.Tensor with shape ()), computed from the forward pass)

model.zero_grad() # so that it does not accumulate gradients from previous backward passes
loss.backward()
```

9.3. Derivative 1: Pre-activation gradient (Derivative of the loss w.r.t pre-activation vector)

Derivatives of scalars w.r.t. vectors (the gradient vector):

Derivative 1) *Derivative of the loss w.r.t pre-activation vector*

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{d_\ell},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} = \nabla_{\mathbf{z}^{(\ell)}}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1^{(\ell)}}, \cdots, \frac{\partial \mathcal{L}}{\partial \mathbf{z}_{d_\ell}^{(\ell)}}\right)$$

(Scalar coordinate of the upstream gradient). For each $i \in \{1,\dots,d_\ell\}$,

$$\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_i = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_i^{(\ell)}} \in \mathbb{R}$$

9.3.1. THE CHAIN RULE

How do we calculate Derivative 1?: The chain rule

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} = \underbrace{\Big(\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}\Big)^{\top}}_{\text{local gradient}} \quad \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}}}_{\text{Global upstream gradient}}$$

Derivatives of vectors w.r.t. vectors (the Jacobian):
Local gradient

$$\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_\ell}$$

Full matrix illustration (showing what is in each entry).

$$\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}} = \begin{bmatrix} \frac{\partial \mathbf{h}_1^{(\ell)}}{\partial \mathbf{z}_1^{(\ell)}} & \frac{\partial \mathbf{h}_1^{(\ell)}}{\partial \mathbf{z}_2^{(\ell)}} & \cdots & \frac{\partial \mathbf{h}_1^{(\ell)}}{\partial \mathbf{z}_{d_\ell}^{(\ell)}} \\ \frac{\partial \mathbf{h}_2^{(\ell)}}{\partial \mathbf{z}_1^{(\ell)}} & \frac{\partial \mathbf{h}_2^{(\ell)}}{\partial \mathbf{z}_2^{(\ell)}} & \cdots & \frac{\partial \mathbf{h}_2^{(\ell)}}{\partial \mathbf{z}_{d_\ell}^{(\ell)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{h}_{d_\ell}^{(\ell)}}{\partial \mathbf{z}_1^{(\ell)}} & \frac{\partial \mathbf{h}_{d_\ell}^{(\ell)}}{\partial \mathbf{z}_2^{(\ell)}} & \cdots & \frac{\partial \mathbf{h}_{d_\ell}^{(\ell)}}{\partial \mathbf{z}_{d_\ell}^{(\ell)}} \end{bmatrix}$$

Entrywise notation

For each $i \in \{1, \ldots, d_\ell\}$ and $j \in \{1, \ldots, d_\ell\}$,

$$\left(\frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}\right)_{i,j} = \frac{\partial \mathbf{h}_i^{(\ell)}}{\partial \mathbf{z}_j^{(\ell)}}$$

(Coordinatewise activation special case). If $\Sigma^{(\ell)}$ acts coordinatewise, i.e.,

$$\mathbf{h}_i^{(\ell)} = \Sigma_i^{(\ell)}\big(\mathbf{z}^{(\ell)}\big) = \sigma^{(\ell)}\big(\mathbf{z}_i^{(\ell)}\big), \qquad i \in \{1, \ldots, d_\ell\},$$

then for each $i, j \in \{1, \ldots, d_\ell\}$,

$$\frac{\partial \mathbf{h}_i^{(\ell)}}{\partial \mathbf{z}_j^{(\ell)}} = \begin{cases} \frac{\partial \Sigma_i^{(\ell)}\left(\mathbf{z}^{(\ell)}\right)}{\partial \mathbf{z}_i^{(\ell)}} = \sigma^{(\ell)\prime}\big(\mathbf{z}_i^{(\ell)}\big), & i = j, \\ 0, & i \neq j, \end{cases}$$

$$\Rightarrow \quad \frac{\partial \mathbf{h}^{(\ell)}}{\partial \mathbf{z}^{(\ell)}} = \operatorname{diag}\Big(\frac{\partial \Sigma_1^{(\ell)}\left(\mathbf{z}^{(\ell)}\right)}{\partial \mathbf{z}_1^{(\ell)}}, \ldots, \frac{\partial \Sigma_{d_\ell}^{(\ell)}\left(\mathbf{z}^{(\ell)}\right)}{\partial \mathbf{z}_{d_\ell}^{(\ell)}}\Big) \quad \text{(Leibniz's notation)}$$

$$= \operatorname{diag}\Big(\sigma^{(\ell)\prime}\big(\mathbf{z}_1^{(\ell)}\big), \ldots, \sigma^{(\ell)\prime}\big(\mathbf{z}_{d_\ell}^{(\ell)}\big)\Big) \quad \text{(Lagrange's notation)}$$

9.3.2. CODE TO GET THE INTERMEDIATE EXPRESSION

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, activation: nn.Module = nn.ReLU(),
        ↪  use_bias: bool = True):
        super().__init__()
        self.lin1 = nn.Linear(5, 3, bias=use_bias)
        self.lin2 = nn.Linear(3, 4, bias=use_bias)
        self.lin3 = nn.Linear(4, 3, bias=use_bias)
        self.act = activation

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        self.z1 = self.lin1(x)
        self.z1.retain_grad()
        self.h1 = self.act(self.z1)
        self.h1.retain_grad()

        self.z2 = self.lin2(self.h1)
        self.z2.retain_grad()
        self.h2 = self.act(self.z2)
        self.h2.retain_grad()

        # logits
        self.z3 = self.lin3(self.h2)
        self.z3.retain_grad()

        return self.z3

model = MLP(activation=nn.ReLU())
x, y = torch.randn(5), torch.tensor(1) # single sample, shape (5,)
    ↪  and class index in {0,1,2}

logits = model(x) # this is z^(3)
loss = ... # your chosen loss function
loss.backward()

print("dL/dz3 =", model.z3.grad)
print("dL/dh2 =", model.h2.grad)
print("dL/dz1 =", model.z1.grad)
```

Note. In practice, automatic differentiation systems do propagate these intermediate gradients internally, but they typically *do not retain* and expose them by default. This is because retaining many

intermediate tensors (and their gradients) increases memory usage, and these intermediates are typically not needed once we have computed the gradients with respect to the learnable parameters (weights and biases).

9.4. Derivative 2: Post-activation gradient (Derivative of the loss w.r.t post-activation vector)

Derivatives of scalars w.r.t. vectors (the gradient vector):

Derivative 2) *Derivative of the loss w.r.t post-activation vector*

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} \in \mathbb{R}^{d_\ell},$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} = \nabla_{\mathbf{h}^{(\ell)}} \mathcal{L} = \Big(\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1^{(\ell)}}, \cdots, \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{d_\ell}^{(\ell)}} \Big)$$

9.4.1. THE CHAIN RULE

How to calculate Derivative 2?: The chain rule

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} = \Big(\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}} \Big)^\top \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell+1)}}}_{\text{Global upstream gradient}}$$

Remark (two equivalent ways to backpropagate to $\mathbf{h}^{(\ell)}$): They are not different rules: they are the *same* chain rule written with a different choice of the "next node" in the computation graph.

- **Case 1:** step to the next affine node $\mathbf{z}^{(\ell+1)}$

  This form is always valid (output layer or hidden layer), and is usually the cleanest:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} = \left(\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}} \right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell+1)}}$$

- **Case 2:** step to the next activation node $\mathbf{h}^{(\ell+1)}$

If one prefers to treat the entire next layer map $T^{(\ell+1)}$ at once, one may write:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} &= \left(\frac{\partial \mathbf{h}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}}\right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell+1)}} \\ &= \Big(\frac{\partial \mathbf{h}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell+1)}} \frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}}\Big)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell+1)}} \\ &= \left(\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}}\right)^{\top} \underbrace{\left(\frac{\partial \mathbf{h}^{(\ell+1)}}{\partial \mathbf{z}^{(\ell+1)}}\right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell+1)}}}_{\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell+1)}}} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(\ell)}} &= \left(\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}}\right)^{\top} \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell+1)}}\end{aligned}$$

which collapses to the definition of the upstream gradient at the affine node.

Derivatives of vectors w.r.t. vectors (the Jacobian).
Local gradient

$$\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}} \in \mathbb{R}^{d_{\ell+1} \times d_\ell}$$

Entrywise notation

For each $i \in \{1, \ldots, d_{\ell+1}\}$ and $j \in \{1, \ldots, d_\ell\}$,

$$\left(\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}}\right)_{i,j} = \frac{\partial \mathbf{z}_i^{(\ell+1)}}{\partial \mathbf{h}_j^{(\ell)}}$$

Full matrix illustration (showing what is in each entry)

$$\frac{\partial \mathbf{z}^{(\ell+1)}}{\partial \mathbf{h}^{(\ell)}} = \begin{bmatrix} \frac{\partial \mathbf{z}_1^{(\ell+1)}}{\partial \mathbf{h}_1^{(\ell)}} & \frac{\partial \mathbf{z}_1^{(\ell+1)}}{\partial \mathbf{h}_2^{(\ell)}} & \cdots & \frac{\partial \mathbf{z}_1^{(\ell+1)}}{\partial \mathbf{h}_{d_\ell}^{(\ell)}} \\ \frac{\partial \mathbf{z}_2^{(\ell+1)}}{\partial \mathbf{h}_1^{(\ell)}} & \frac{\partial \mathbf{z}_2^{(\ell+1)}}{\partial \mathbf{h}_2^{(\ell)}} & \cdots & \frac{\partial \mathbf{z}_2^{(\ell+1)}}{\partial \mathbf{h}_{d_\ell}^{(\ell)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{z}_{d_{\ell+1}}^{(\ell+1)}}{\partial \mathbf{h}_1^{(\ell)}} & \frac{\partial \mathbf{z}_{d_{\ell+1}}^{(\ell+1)}}{\partial \mathbf{h}_2^{(\ell)}} & \cdots & \frac{\partial \mathbf{z}_{d_{\ell+1}}^{(\ell+1)}}{\partial \mathbf{h}_{d_\ell}^{(\ell)}} \end{bmatrix}$$

9.5. Derivative 3: Weight matrix gradient (Derivative of the loss w.r.t weight matrix)

Derivative of scalars w.r.t. a matrix:
Weight matrix gradient

Derivative 3) *Derivative of the loss w.r.t weight matrix*

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$$

9.5.1. THE CHAIN RULE

How to calculate Derivative 3 (weight gradient): chain rule + immediate contraction.

We conceptually start from the chain rule

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)}_{\text{Global upstream gradient}} \cdot \underbrace{\left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\right)}_{\text{Local gradient}}$$

where $\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}$ is a *vector-with-respect-to-matrix* derivative. Formally, this object is a *3-tensor* (its entries are indexed by one output coordinate and two weight-matrix coordinates), so writing it out explicitly is possible but usually unhelpful and notationally heavy.

In backpropagation we never need to *store* this 3-tensor: we immediately *contract* it with the upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{d_\ell}$, which produces a matrix-shaped gradient. Carrying out this contraction yields the standard closed form

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right) \left(\mathbf{h}^{(\ell-1)}\right)^\top \qquad \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$$

(We provide an exposition below on how to interpret the corresponding 3-tensor Jacobian and how the subsequent contraction yields the matrix-shaped gradient.)

Derivation: Conceptualizing the 3-tensor Jacobian and performing the contraction

$$\left(\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}\right)_{m,k} = \sum_{j=1}^{d_\ell} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_j \left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}\right)$$

(Index-level chain rule via a 3-tensor Jacobian).

Notice $\left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\right)$ is a derivative of a vector w.r.t. a matrix. Intuitively, we perturb one entry $W_{m,k}^{(\ell)}$ and ask how each component $\mathbf{z}_j^{(\ell)}$ changes.

Here

$$j \in \{1, \ldots, d_\ell\}, \qquad m \in \{1, \ldots, d_\ell\}, \qquad k \in \{1, \ldots, d_{\ell-1}\},$$

so $\left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\right)$ has $d_\ell \cdot d_\ell \cdot d_{\ell-1}$ entries, i.e. it lives in $\mathbb{R}^{d_\ell \times d_\ell \times d_{\ell-1}}$. This is the "3-tensor" Jacobian.

We commonly do not write this out, because this object does not correspond to a matrix or a vector. In practice, we do not store this object (it is large and unnecessary), instead we immediately

contract it with the upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$ using

$$\left(\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}\right)_{m,k} = \sum_{j=1}^{d_\ell} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_j \left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}\right)$$

You can think of $\left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\right)$ as d_ℓ many matrices stacked together:

$$\left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\right) = \Big(\underbrace{\frac{\partial \mathbf{z}_1^{(\ell)}}{\partial W^{(\ell)}}}_{\in \mathbb{R}^{d_\ell \times d_{\ell-1}}}, \cdots, \underbrace{\frac{\partial \mathbf{z}_{d_\ell}^{(\ell)}}{\partial W^{(\ell)}}}_{\in \mathbb{R}^{d_\ell \times d_{\ell-1}}} \Big) \in \mathbb{R}^{d_\ell \times d_\ell \times d_{\ell-1}}$$

Recall in a single layer, we have an affine map prior to passing it to a nonlinearity:

$$\mathbf{z}^{(\ell)} \;=\; W^{(\ell)}\mathbf{h}^{(\ell-1)} \;+\; \mathbf{b}^{(\ell)}$$

In index-wise notation, we can write out the explicit definition of the matrix-vector product:

$$\mathbf{z}_j^{(\ell)} \;=\; \sum_{t=1}^{d_{\ell-1}} W_{j,t}^{(\ell)}\, \mathbf{h}_t^{(\ell-1)} \;+\; \mathbf{b}_j^{(\ell)}$$

We define the 3-tensor Jacobian $J^{(\ell)} \in \mathbb{R}^{d_\ell \times d_\ell \times d_{\ell-1}}$ by

$$J_{j,m,k}^{(\ell)} \;:=\; \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}$$

Therefore we have

$$J_{j,m,k}^{(\ell)} \;=\; \frac{\partial}{\partial W_{m,k}^{(\ell)}} \left(\sum_{t=1}^{d_{\ell-1}} W_{j,t}^{(\ell)}\, \mathbf{h}_t^{(\ell-1)} \;+\; \mathbf{b}_j^{(\ell)} \right)$$

Now compute this derivative explicitly:

$$J_{j,m,k}^{(\ell)} \;=\; \sum_{t=1}^{d_{\ell-1}} \frac{\partial}{\partial W_{m,k}^{(\ell)}} \left(W_{j,t}^{(\ell)}\right) \mathbf{h}_t^{(\ell-1)} \;+\; \underbrace{\frac{\partial}{\partial W_{m,k}^{(\ell)}} \left(\mathbf{b}_j^{(\ell)}\right)}_{0}$$

Since $\mathbf{b}^{(\ell)}$ does not depend on $W^{(\ell)}$, it becomes 0.
And since the entries of $W^{(\ell)}$ are independent coordinates,

$$\frac{\partial}{\partial W_{m,k}^{(\ell)}} \left(W_{j,t}^{(\ell)}\right) = \delta_{j,m}\, \delta_{t,k}$$

Therefore,

$$\begin{aligned} J_{j,m,k}^{(\ell)} &= \sum_{t=1}^{d_{\ell-1}} \delta_{j,m}\, \delta_{t,k}\, \mathbf{h}_t^{(\ell-1)} \\ &= \delta_{j,m}\, \mathbf{h}_k^{(\ell-1)} \end{aligned}$$

The Kronecker delta $\delta_{j,m}$ is there to ensure that only the output coordinate $\mathbf{z}_j^{(\ell)}$ that *actually depends* on the row m of $W^{(\ell)}$ contributes: if $j \neq m$, then

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}} = 0$$

Hence, if we think of the 3-tensor Jacobian in the "stacked matrix" view (9.5.1), then for each fixed $j \in \{1, \ldots, d_\ell\}$ we can regard

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$$

as the matrix whose (m, k) entry is $\left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}}\right)_{m,k} = \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}} = \delta_{j,m}\, \mathbf{h}_k^{(\ell-1)}$

Equivalently, written out as a full matrix:

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}} = \begin{bmatrix} \delta_{j,1}\, \mathbf{h}_1^{(\ell-1)} & \delta_{j,1}\, \mathbf{h}_2^{(\ell-1)} & \cdots & \delta_{j,1}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \delta_{j,2}\, \mathbf{h}_1^{(\ell-1)} & \delta_{j,2}\, \mathbf{h}_2^{(\ell-1)} & \cdots & \delta_{j,2}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{j,d_\ell}\, \mathbf{h}_1^{(\ell-1)} & \delta_{j,d_\ell}\, \mathbf{h}_2^{(\ell-1)} & \cdots & \delta_{j,d_\ell}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \end{bmatrix}, \quad \forall j$$

Remark: ***Why we cannot "write" the*** 3***-tensor Jacobian!***
The object

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}$$

is *not* a matrix! It is a *third-order tensor* whose entries are indexed by

$$(j,m,k) \in \{1,\dots,d_\ell\} \times \{1,\dots,d_\ell\} \times \{1,\dots,d_{\ell-1}\}, \qquad \left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}}\right)_{j,m,k} := \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}$$

A matrix has *two* indices (row, column). This object has *three*. Therefore, there is no canonical single pair of brackets $[\cdot]$ that truthfully represents it without first choosing an *arbitrary flattening rule* (e.g., "merge (m,k) into one super-index," or "merge (j,m)," etc.). Different flattenings give different matrix shapes, and silently committing to one can break dimension checks later.

A tempting but misleading "nested matrix" picture. To emphasize why it is intractable, one might be tempted to draw it as "a vector of matrices," namely "stack the slices $\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}}$ over j." Formally, for each fixed j we have a matrix

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}, \qquad \left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}}\right)_{m,k} = \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}} = \delta_{j,m}\, \mathbf{h}_k^{(\ell-1)}$$

So the "stacked matrix view" would look like a tuple of d_ℓ matrices:

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}} \equiv \Bigg(\underbrace{\frac{\partial \mathbf{z}_1^{(\ell)}}{\partial W^{(\ell)}}}_{\in \mathbb{R}^{d_\ell \times d_{\ell-1}}}, \underbrace{\frac{\partial \mathbf{z}_2^{(\ell)}}{\partial W^{(\ell)}}}_{\in \mathbb{R}^{d_\ell \times d_{\ell-1}}}, \dots, \underbrace{\frac{\partial \mathbf{z}_{d_\ell}^{(\ell)}}{\partial W^{(\ell)}}}_{\in \mathbb{R}^{d_\ell \times d_{\ell-1}}} \Bigg)$$

Hence a "large nested matrix" depiction (that we usually do not illustrate in the following way) would be something like

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial W^{(\ell)}} \overset{\text{(NOT a matrix)}}{=} \left[\left(\frac{\partial \mathbf{z}_1^{(\ell)}}{\partial W^{(\ell)}}\right), \left(\frac{\partial \mathbf{z}_2^{(\ell)}}{\partial W^{(\ell)}}\right), \cdots, \left(\frac{\partial \mathbf{z}_{d_\ell}^{(\ell)}}{\partial W^{(\ell)}}\right) \right]$$

$$= \left[\begin{bmatrix} \delta_{1,1}\, \mathbf{h}_1^{(\ell-1)} & \cdots & \delta_{1,1}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \delta_{1,2}\, \mathbf{h}_1^{(\ell-1)} & \cdots & \delta_{1,2}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \vdots & \ddots & \vdots \\ \delta_{1,d_\ell}\, \mathbf{h}_1^{(\ell-1)} & \cdots & \delta_{1,d_\ell}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \end{bmatrix}, \cdots, \begin{bmatrix} \delta_{d_\ell,1}\, \mathbf{h}_1^{(\ell-1)} & \cdots & \delta_{d_\ell,1}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \delta_{d_\ell,2}\, \mathbf{h}_1^{(\ell-1)} & \cdots & \delta_{d_\ell,2}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \vdots & \ddots & \vdots \\ \delta_{d_\ell,d_\ell}\, \mathbf{h}_1^{(\ell-1)} & \cdots & \delta_{d_\ell,d_\ell}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \end{bmatrix} \right]$$

This hypothetical display is a *warning*. It visually shows that we are no longer in the world of matrix calculus, because we are stacking matrices inside a larger bracket. There is no natural multiplication rule for this "object" unless we first specify a contraction (tensor-times-vector, tensor-times-matrix, etc.).

Why we do not store the 3-tensor Jacobian in backprop? In practice, the 3-tensor Jacobian is never formed explicitly. Instead, we immediately perform a *contraction* with the upstream gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{d_\ell}$$

to obtain the matrix-shaped parameter gradient

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$$

This contraction is precisely what avoids ever needing to manipulate or store the nested tensor structure above. Moreover, the 3-tensor would be prohibitively large to store even for moderately sized layers, and it would be contracted immediately in the very next step of the computation anyway, so we might as well perform the contraction directly.

A useful mental picture *before contraction* is to view it as a *stack of matrix slices*, one slice per output coordinate.

Concretely, for each fixed $j \in \{1, \dots, d_\ell\}$, the derivative of the scalar $\mathbf{z}_j^{(\ell)}$ with respect to the matrix $W^{(\ell)}$ is a *matrix* in $\mathbb{R}^{d_\ell \times d_{\ell-1}}$:

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}} = \begin{bmatrix} \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{1,1}^{(\ell)}} & \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{1,2}^{(\ell)}} & \cdots & \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{1,d_{\ell-1}}^{(\ell)}} \\ \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{2,1}^{(\ell)}} & \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{2,2}^{(\ell)}} & \cdots & \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{2,d_{\ell-1}}^{(\ell)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{d_\ell,1}^{(\ell)}} & \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{d_\ell,2}^{(\ell)}} & \cdots & \frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{d_\ell,d_{\ell-1}}^{(\ell)}} \end{bmatrix} \qquad \forall j \in \{1, \dots, d_\ell\}$$

Using the affine definition $\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$, we have, for every $(m, k) \in \{1, \dots, d_\ell\} \times \{1, \dots, d_{\ell-1}\}$,

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}} = \delta_{j,m}\, \mathbf{h}_k^{(\ell-1)}$$

Therefore the slice matrix can be written explicitly as

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W^{(\ell)}} = \begin{bmatrix} \delta_{j,1}\, \mathbf{h}_1^{(\ell-1)} & \delta_{j,1}\, \mathbf{h}_2^{(\ell-1)} & \cdots & \delta_{j,1}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \delta_{j,2}\, \mathbf{h}_1^{(\ell-1)} & \delta_{j,2}\, \mathbf{h}_2^{(\ell-1)} & \cdots & \delta_{j,2}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{j,d_\ell}\, \mathbf{h}_1^{(\ell-1)} & \delta_{j,d_\ell}\, \mathbf{h}_2^{(\ell-1)} & \cdots & \delta_{j,d_\ell}\, \mathbf{h}_{d_{\ell-1}}^{(\ell-1)} \end{bmatrix}, \qquad \forall j \in \{1, \dots, d_\ell\}$$

9.5.2. CONTRACTION: ENTRY-LEVEL CHAIN RULE

Contraction is the *index-summation* operation that generalizes matrix–vector and matrix–matrix multiplication to higher-order tensors, producing the result obtained by evaluating the corresponding indexed expression. It is effectively a multiplication rule for tensor-valued objects in which we multiply entries and then *sum over a shared index*. The term *contraction* reflects that summing over a shared index *reduces the number of free indices* in the resulting object.

Recall the elementary definition for

Matrix–vector multiplication as contraction

$$(A\mathbf{x})_j := \sum_{r=1}^{d} A_{j,r}\, x_r, \qquad A \in \mathbb{R}^{d\times d},\ \mathbf{x} \in \mathbb{R}^d$$

Matrix–matrix multiplication as contraction

$$(AB)_{j,k} := \sum_{r=1}^{d} A_{j,r}\, B_{r,k}, \qquad A, B \in \mathbb{R}^{d\times d}$$

Entry-level chain rule as contraction

$$\left(\frac{\partial \mathbf{h}}{\partial \mathbf{w}}\right)_{j,k} := \sum_{r=1}^{d} \left(\frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right)_{j,r} \left(\frac{\partial \mathbf{z}}{\partial \mathbf{w}}\right)_{r,k}, \qquad j \in \{1,\dots,d\},\ \ k \in \{1,\dots,d\},$$

$$\frac{\partial h_j}{\partial w_k} = \sum_{r=1}^{d} \frac{\partial h_j}{\partial z_r}\, \frac{\partial z_r}{\partial w_k}, \qquad j \in \{1,\dots,d\},\ \ k \in \{1,\dots,d\}$$

In our setup this translates to

For each $(m,k) \in \{1,\dots,d_\ell\} \times \{1,\dots,d_{\ell-1}\}$,

$$\left(\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}\right)_{m,k} = \sum_{j=1}^{d_\ell} \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_j}_{\in\mathbb{R}} \underbrace{\left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}\right)}_{=\,\delta_{j,m}\,\mathbf{h}_k^{(\ell-1)}}$$

$$= \sum_{j=1}^{d_\ell} \underbrace{\left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_j^{(\ell)}}\right)}_{\in\mathbb{R}} \underbrace{\left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial W_{m,k}^{(\ell)}}\right)}_{=\,\delta_{j,m}\,\mathbf{h}_k^{(\ell-1)}}$$

Substituting the explicit derivative,

$$\left(\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}\right)_{m,k} = \sum_{j=1}^{d_\ell} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_j \left(\delta_{j,m}\,\mathbf{h}_k^{(\ell-1)}\right)$$

Because the Kronecker delta $\delta_{j,m}$ forces $j = m$, the sum collapses to a single term:

$$\left(\frac{\partial \mathcal{L}}{\partial W^{(\ell)}}\right)_{m,k} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_m \mathbf{h}_k^{(\ell-1)}$$

Therefore (matrix closed form, after contraction). Equivalently, collecting the entries (m, k) into a matrix,

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right) \left(\mathbf{h}^{(\ell-1)}\right)^\top \qquad \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$$

9.5.3. CODE TO EXTRACT WEIGHT GRADIENT MATRIX OF A SPECIFIC LAYER

```python
import torch
import torch.nn as nn

# assume loss.backward() has already been called

layer = ...  # index of an nn.Linear inside model.net
W = model.net[layer].weight         # parameter tensor
dL_dW = model.net[layer].weight.grad  # gradient tensor (same
    ↪  shape as W)
```

9.6. Derivative 4: The bias gradient (Derivative of the loss w.r.t bias vector)

Derivative of scalars w.r.t. a vector:
Bias gradient

Derivative 4) *Derivative of the loss w.r.t bias vector*

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} \in \mathbb{R}^{d_\ell}$$

9.6.1. THE CHAIN RULE

How to calculate Derivative 4 (bias gradient): chain rule + immediate contraction.

The chain rule

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$$

Final closed-form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \qquad \in \mathbb{R}^{d_\ell}$$

Derivation

Setup (affine map).

$$\mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \qquad \mathbf{z}^{(\ell)} \in \mathbb{R}^{d_\ell},\ \mathbf{b}^{(\ell)} \in \mathbb{R}^{d_\ell}$$

Jacobian of a vector w.r.t. a vector (no 3-tensors needed).

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_\ell}$$

Entrywise definition: for each $i, j \in \{1, \ldots, d_\ell\}$,

$$\left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}}\right)_{i,j} = \frac{\partial \mathbf{z}_i^{(\ell)}}{\partial \mathbf{b}_j^{(\ell)}}$$

Compute the local derivative explicitly. From

$$\mathbf{z}_i^{(\ell)} = \sum_{t=1}^{d_{\ell-1}} W_{i,t}^{(\ell)}\, \mathbf{h}_t^{(\ell-1)} + \mathbf{b}_i^{(\ell)},$$

we get

$$\frac{\partial \mathbf{z}_i^{(\ell)}}{\partial \mathbf{b}_j^{(\ell)}} = \delta_{i,j}$$

Therefore the full Jacobian is the identity:

$$\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}} = \begin{bmatrix} \delta_{1,1} & \delta_{1,2} & \cdots & \delta_{1,d_\ell} \\ \delta_{2,1} & \delta_{2,2} & \cdots & \delta_{2,d_\ell} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{d_\ell,1} & \delta_{d_\ell,2} & \cdots & \delta_{d_\ell,d_\ell} \end{bmatrix} = I_{d_\ell}$$

The Chain rule (vector form, with transpose convention)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \left(\frac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}}\right)^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$$

Substitute $\dfrac{\partial \mathbf{z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}} = I_{d_\ell}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = (I_{d_\ell})^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}$$

Coordinate-wise derivation (same result).

For each $m \in \{1, \ldots, d_\ell\}$, apply the scalar chain rule:

$$\left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}\right)_m = \sum_{j=1}^{d_\ell} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_j \left(\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial \mathbf{b}_m^{(\ell)}}\right)$$

As established from above, we get

$$\frac{\partial \mathbf{z}_j^{(\ell)}}{\partial \mathbf{b}_m^{(\ell)}} = \delta_{j,m}$$

Substitute and collapse:

$$\left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}\right)_m = \sum_{j=1}^{d_\ell} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_j \delta_{j,m}$$
$$\left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}\right)_m = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}}\right)_m$$

Therefore, in vector form,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \qquad \in \mathbb{R}^{d_\ell}$$

9.6.2. CODE TO EXTRACT BIAS GRADIENT VECTOR OF A SPECIFIC LAYER

```python
import torch
import torch.nn as nn

# assume loss.backward() has already been called

layer = ...  # index of an nn.Linear inside model.net
b = model.net[layer].bias           # parameter tensor (or None if
    ↪ bias=False)
dL_db = model.net[layer].bias.grad  # gradient tensor (same shape
    ↪ as b)
```

10. Optimization step

Once we have computed the global gradients of the loss with respect to the weights and biases,

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} \in \mathbb{R}^{d_\ell \times d_{\ell-1}} \qquad \text{and} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} \in \mathbb{R}^{d_\ell}, \qquad \forall \ell,$$

we now understand how an *infinitesimally small* perturbation to individual entries of the weight matrices and bias vectors changes the resulting value of the loss function. The remaining task is to "update" (i.e., *optimize*) the weights and biases so as to minimize the loss. We minimize the loss because, in supervised settings, it directly reflects the discrepancy between the model's predictions and the ground-truth labels.

There are several principled algorithms for doing so, which we will delve into, but before introducing them we first need to clarify why gradients are central to this idea.

Geometrically, in the high-dimensional loss landscape, the gradient encodes a *first-order (local linear) approximation* of the loss.

We can see this from the first-order Taylor expansion. Let $f : \mathbb{R}^d \to \mathbb{R}$ be differentiable, let $\mathbf{x} \in \mathbb{R}^d$ be a point, and let $\mathbf{h} \in \mathbb{R}^d$ be a direction. Then, as $t \to 0$,

$$f(\mathbf{x} + t\mathbf{h}) = f(\mathbf{x}) + t\,\mathbf{h}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) + o(|t|)$$

Equivalently, since $\|t\mathbf{h}\|_2 = |t|\|\mathbf{h}\|_2$ by homogeneity of norms,

$$f(\mathbf{x} + t\mathbf{h}) = f(\mathbf{x}) + t\,\mathbf{h}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) + o\big(\|t\mathbf{h}\|_2\big)$$

Discounting the *remainder term* gives the usual first-order *approximation*

$$f(\mathbf{x} + t\mathbf{h}) \approx f(\mathbf{x}) + t\,\mathbf{h}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$$

Intuitively, this expression tell us that near $\mathbf{x}$, the function behaves like a hyperplane whose slope in direction $\mathbf{h}$ is $\mathbf{h}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$.

Indeed, expanding the Euclidean inner product gives $\mathbf{h}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) = \sum_{i=1}^{d} \mathbf{h}_i \left(\frac{\partial f(\mathbf{x})}{\partial x_i}\right)$. Thus the first-order approximation consists of a constant term $f(\mathbf{x})$ plus a linear function of $\mathbf{h}$. Therefore the resulting first-order model is an affine function (i.e., a hyperplane in $\mathbb{R}^d$).

This "first-order approximation" hyperplane is a powerful object because it encodes the local rate of change of the loss function as we move along the independent coordinate directions (i.e., along the basis vectors). By contrast, directly probing the loss surface by evaluating $\mathcal{L}$ at many nearby points is not a principled strategy in a continuous domain: even an arbitrarily small neighborhood contains *uncountably infinitely many* points, so any exhaustive "testing" heuristic is intractable. First-order information avoids this by "summarizing" the local behavior using only the directional rates of change along the basis directions, yielding a tractable local linear approximation.

The step size t controls how far we move; larger steps typically incur larger approximation error, which is why this is only a local approximation. This provides a heuristic for how to traverse the landscape even when the overall objective is nonconvex (and potentially nonsmooth at some points, depending on the chosen loss and activations).

For completeness, there is also the second-order Taylor expansion. If f is twice differentiable, then as $t \to 0$,

$$f(\mathbf{x} + t\mathbf{h}) = f(\mathbf{x}) + t\,\mathbf{h}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) + \frac{1}{2} t^2\,\mathbf{h}^\top \nabla_{\mathbf{x}}^2 f(\mathbf{x})\,\mathbf{h} + o(t^2)$$

The additional quadratic term depends on the Hessian $\nabla_{\mathbf{x}}^2 f(\mathbf{x})$ and captures local curvature. In principle, this can provide a richer traversal heuristic than first-order information alone because it describes how the slope itself changes.

This becomes especially helpful in "sticky" situations near *critical points*, where the gradient collapses to zero:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{0}$$

That can happen at local minima, local maxima, and saddle points. In such regions, purely gradient-based heuristics may provide little guidance about *which way to move next*. Second-order information, through the Hessian will supply us with curvature information that can help us tell whether we are sitting in a bowl (positive curvature), on a hill (negative curvature), or on a saddle (mixed curvature), and it is the extra ingredient used by second-order optimization methods to choose more informed steps.

In practice, for modern deep networks, explicitly forming and using full Hessian information is typically too expensive at scale, so most implementations rely on first-order methods (and their variants), which are often sufficient to obtain good convergence behavior in large models.

10.1. Optimization algorithm 1: Vanilla Gradient Descent

"Vanilla Gradient Descent"

$$W_{j,k}^{(n+1)} := \underbrace{W_{j,k}^{(n)} - \eta \left(\frac{\partial \mathcal{L}}{\partial W_{j,k}^{(n)}} \right)}_{\in \mathbb{R}}, \qquad \forall\, j, k,\ \forall\, n \in \mathbb{N},$$

$$b_{j}^{(n+1)} := \underbrace{b_{j}^{(n)} - \eta \left(\frac{\partial \mathcal{L}}{\partial b_{j}^{(n)}} \right)}_{\in \mathbb{R}}, \qquad \forall\, j,\ \forall\, n \in \mathbb{N}$$

In words, this rule says: at iteration n, we update each parameter $W_{j,k}^{(n)}$ and $b_j^{(n)}$ (each called an *iterate*) by subtracting from the current iterate by the product of the step size η and the corresponding partial derivative, $\frac{\partial \mathcal{L}}{\partial W_{j,k}^{(n)}}$ or $\frac{\partial \mathcal{L}}{\partial b_j^{(n)}}$.

Viewing $\mathcal{L}$ as a function of (W, b), we "take a step" of size η in the *negative* gradient direction. Since the gradient measures the local rate of increase of the loss with respect to each parameter, subtracting it decreases the loss to first order.

10.1.1. WHY STEP IN THE NEGATIVE GRADIENT DIRECTION?

An informal justification for why we step in the *negative* gradient direction comes from studying the local behavior of the loss landscape using the Taylor expansion in the first-order.

For convenience, collect all parameters into a single vector:

$$\mathbf{w} := \big(\mathrm{vec}(W), \mathbf{b}\big)$$

Then we may view the loss as a scalar function of $\mathbf{w}$, i.e., $\mathcal{L} = \mathcal{L}(\mathbf{w})$. This "flattening" is valid because stacking the entries of $(W, \mathbf{b})$ into a vector is an isomorphism — it preserves the underlying degrees of freedom and only changes how we arrange them. Hence we may treat $\mathbf{w}$ as an ordinary vector parameter.

Previously we wrote the Taylor expansion for a generic scalar objective $f(\mathbf{w})$. In the learning setting, the objective we actually seek to minimize is the loss $\mathcal{L}(\mathbf{w})$. Thus we rewrite the same first-order expansion by replacing f with $\mathcal{L}$:

$$\mathcal{L}\big(\mathbf{w} + \underbrace{t\mathbf{h}}_{\text{step vector}}\big) \approx \mathcal{L}(\mathbf{w}) + t\, \mathbf{h}^\top \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}), \qquad \text{for small } t$$

Gradient descent is the specific choice of a step vector that is proportional to the negative gradient. Concretely, it sets the update to

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + \underbrace{\Big(-\eta\, \nabla_{\mathbf{w}} \mathcal{L}\big(\mathbf{w}^{(n)}\big)\Big)}_{\text{Gradient descent step vector}}$$

which corresponds to choosing $t\mathbf{h} = -\eta\, \nabla_{\mathbf{w}} \mathcal{L}\big(\mathbf{w}^{(t)}\big)$.

To understand why this choice makes sense, we start from the Taylor approximation and treat $t\mathbf{h}$ as the local "move" we are allowed to choose. The Taylor expansion separates *direction* from *magnitude*: $\mathbf{h}$ is the direction vector, and $t > 0$ controls how far we move along that direction. By contrast, in vanilla gradient descent we commit to the specific direction $\mathbf{h} = -\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ and keep only a single scalar step size η to control the overall magnitude of the update.

To make $\mathcal{L}(\mathbf{w} + t\mathbf{h})$ decrease, we want the *first-order change* predicted by the Taylor model, $(t\,\mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}))$, to be negative. For a fixed small $t > 0$, the sign of this first-order change is determined by the inner product $\mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ appearing in

$$\mathcal{L}\big(\mathbf{w} + t\mathbf{h}\big) \approx \mathcal{L}(\mathbf{w}) + t\,\underbrace{\mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})}_{\text{inner product}}, \qquad \text{for small } t$$

Thus, to make the first-order change negative (for a fixed small $t > 0$), we should choose $\mathbf{h}$ so that

$$\mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) < 0$$

At a fixed parameter point $\mathbf{w}$, the gradient $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ is fixed. Hence our only freedom is the choice of direction $\mathbf{h}$. To decide which choice of $\mathbf{h}$ makes the most sense (we do not discuss the optimal step size t here), we focus on the inner product

$$\mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}),$$

because it determines the sign and magnitude of the first-order change.

However, without any constraint, this quantity is not meaningfully comparable across choices of $\mathbf{h}$: if $\mathbf{h}$ makes the inner product negative, then scaling it by a large constant makes the inner product arbitrarily large in magnitude (and still negative). Therefore, to make a fair comparison of *directions*, we *fix the step length* $\|\mathbf{h}\|_2$ and study directionality alone. Intuitively, this holds the "amount of movement" constant and lets us compare directions fairly; otherwise we could make the inner product arbitrarily large in magnitude simply by scaling $\mathbf{h}$.

With $\|\mathbf{h}\|_2$ fixed, we apply Cauchy–Schwarz to upper bound the magnitude of the inner product:

$$\big|\mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})\big| \le \|\mathbf{h}\|_2\, \|\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})\|_2$$

Equivalently,

$$-\|\mathbf{h}\|_2\, \|\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})\|_2 \le \mathbf{h}^\top \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) \le \|\mathbf{h}\|_2\, \|\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})\|_2$$

So the *most negative* first-order change (i.e., the best local decrease for a fixed step length) occurs when $\mathbf{h}$ points exactly opposite to the gradient. Concretely, equality on the left is attained when

$$\mathbf{h} = -\alpha\, \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) \qquad \text{for some } \alpha > 0$$

Therefore, if we choose a single scalar step size and absorb it into the direction by setting $t\alpha = \eta$, the resulting update takes the familiar gradient descent form

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \eta\, \nabla_{\mathbf{w}}\mathcal{L}\big(\mathbf{w}^{(n)}\big)$$

This is the informal Taylor-expansion reason we step in the *negative* gradient direction: among all directions of the same step length, it makes the first-order term as negative as possible, hence it decreases $\mathcal{L}$ at $\mathbf{w}$ the fastest *to first order*.

With the canonical feedforward neural network now fully specified, we are ready to extend a few classical ideas that can be expressed cleanly using the formulations we have developed.

11. Feed forward Neural Network with Dropout

11.1. Dropout in neural networks

Dropout is a *regularization* technique for neural networks that is typically applied only to the input and hidden layers, and is specified independently for each layer.

Regularization refers to a family of techniques designed to mitigate a common failure mode in neural network learning called *overfitting*, in which a model fits the training data extremely well but performs poorly on prediction tasks for previously unseen data drawn from the same underlying data-generating distribution. The guiding idea is that a network should distill salient structure from finite data and apply that knowledge to predict new examples it has not seen. This directly concerns the model's ability to *generalize* predictions, i.e., to make accurate predictions beyond the training set.

Overfitting is undesirable because the practical goal of a good predictive model is to perform well on new data. If it fails to do so, we may describe it (somewhat anthropomorphically) as merely "memorizing" the training set.

Regularization addresses this by deliberately modifying the learning procedure's *inductive bias* — rather than relying on whatever solution the model would prefer over another on its own. A classical example from statistics is to add an explicit penalty on the weights in the loss function (as in penalized linear regression), so that training favors "simpler" or more stable solutions, i.e., solutions that are less sensitive to small perturbations in the data.

Dropout regularizes in a different way from explicit weight penalties. Instead of directly penalizing the parameter values, dropout modifies the *forward pass* during training by randomly *masking* (setting to zero) a subset of neuron *activations*. This induces each training step to use a randomly *"thinned" subnetwork* (terminology used in the original dropout work Srivastava et al. (2014)) *in the computation graph* (via *masked activations*, which we will formalize mathematically later), even though the underlying parameter tensors (weights and biases) are unchanged and are shared across all such thinned subnetworks. This effectively discourages the model from relying too heavily on any particular subset of neurons (or features) during learning and promotes more distributed, redundant representations.

At test time, dropout is turned off, and the weights are multiplied by the probability p.
Procedurally, this corresponds to two phases:

1. **Training (dropout on).**

   Fix a keep probability $p^{(\ell-1)} \in [0, 1]$.

   We adopt the convention that any symbol with a tilde denotes a random variable.

   For each $\ell \in \{1, 2, \ldots, L\}$, define the *dropout mask vector* $\tilde{\mathbf{d}}^{(\ell-1)} \in \{0, 1\}^{d_{\ell-1}}$ as follows.

   (a) **If dropout is enabled**. It randomly turns off/on coordinates of the *post-activation vector* $\mathbf{h}^{(\ell-1)}$ by applying a Hadamard product with a random mask $\tilde{\mathbf{d}}^{(\ell-1)}$,

   $$\tilde{\mathbf{d}}_j^{(\ell-1)} \overset{\text{i.i.d.}}{\sim} \text{Bernoulli}\big(p^{(\ell-1)}\big), \qquad j = 1, \ldots, d_{\ell-1}$$

   where $p^{(\ell-1)} \in (0, 1]$ is the keep probability.

(b) **If dropout is disabled** on $\mathbf{h}^{(\ell-1)}$, set

$$\tilde{\mathbf{d}}^{(\ell-1)} := \mathbf{1}_{d_{\ell-1}}$$

(the all-ones mask), so that masking leaves activations unchanged.

Each mask has length $d_{\ell-1}$ because dropout is applied to the post-activation vector $\mathbf{h}^{(\ell-1)} \in \mathbb{R}^{d_{\ell-1}}$, so the Hadamard product $\mathbf{h}^{(\ell-1)} \odot \tilde{\mathbf{d}}^{(\ell-1)}$ is well-defined.

The output layer (logits) is typically not subjected to dropout, since we usually want the interface into the loss (the logits or predicted outputs) to be computed deterministically from the current parameters during that forward pass, rather than being randomly zeroed in some coordinates. Thus, the logits passed to the loss remain a task-defined set of scores rather than being randomly masked.

Then, at layer ℓ, the forward-pass computation with *masked activations* is

$$\mathbf{h}^{(\ell)} = \Sigma^{(\ell)}\Big(W^{(\ell)}\big(\mathbf{h}^{(\ell-1)} \underbrace{\odot \tilde{\mathbf{d}}^{(\ell-1)}}_{\text{random activation masking}}\big) + \mathbf{b}^{(\ell)}\Big)$$

Here the previous layer's post-activation vector $\mathbf{h}^{(\ell-1)}$ is multiplied elementwise by a Bernoulli mask $\tilde{\mathbf{d}}^{(\ell-1)}$ via the Hadamard product $(\odot)$. Mathematically, relative to the standard layer computation, dropout introduces exactly this additional masking operation on the previous layer's post-activation vector at the layers where it is enabled.

Note. The Hadamard product $(\odot)$ is an elementwise multiplication. For any fixed realization of the mask $\tilde{\mathbf{d}}^{(\ell-1)}$, the masking map (written in assignment-rule notation) is

$$\mathbf{h}^{(\ell-1)} \mapsto \mathbf{h}^{(\ell-1)} \odot \tilde{\mathbf{d}}^{(\ell-1)},$$

which is an elementwise multiplication (Hadamard product).

is linear in $\mathbf{h}^{(\ell-1)}$. Equivalently, for a fixed mask vector $\mathbf{d}^{(\ell-1)} \in \{0,1\}^{d_{\ell-1}}$, we may represent the same masking operation as multiplication by a diagonal matrix:

$$\mathbf{h}^{(\ell-1)} \odot \mathbf{d}^{(\ell-1)} = \operatorname{diag}\big(\mathbf{d}^{(\ell-1)}\big)\, \mathbf{h}^{(\ell-1)}$$

This form is nice because it makes the Jacobian w.r.t. $\mathbf{h}^{(\ell-1)}$ immediate:

$$\frac{\partial(\mathbf{h}^{(\ell-1)} \odot \mathbf{d}^{(\ell-1)})}{\partial \mathbf{h}^{(\ell-1)}} = \operatorname{diag}\big(\mathbf{d}^{(\ell-1)}\big)$$

2. **Testing (dropout off).** In the original paper, there is a scale factor $p^{(\ell-1)} \in [0,1]$ applied to the post-activation vector $\mathbf{h}^{(\ell-1)}$ when dropout is used. To see why this factor is imperative, we take an expectation over the dropout mask to characterize the average behavior of the forward pass across many training iterations.

   Our first attempt is to study the randomness of the full layer map. Since during training we repeatedly sample dropout masks (with i.i.d. Bernoulli coordinates) across iterations, a

natural way to summarize the "average behavior" of the random hidden output is to take an expectation over the dropout randomness. Concretely, with dropout applied to $\mathbf{h}^{(\ell-1)}$, the resulting quantity $\tilde{\mathbf{d}}^{(\ell-1)} \odot \mathbf{h}^{(\ell-1)}$ is a random vector because the dropout mask $\tilde{\mathbf{d}}^{(\ell-1)}$ is random (and $\mathbf{h}^{(\ell-1)}$ is itself random as it is a function of earlier random masks). The random pre-activation and post-activation at layer ℓ therefore satisfy

$$\tilde{\mathbf{z}}^{(\ell)} = W^{(\ell)}\big(\tilde{\mathbf{d}}^{(\ell-1)} \odot \mathbf{h}^{(\ell-1)}\big) + \mathbf{b}^{(\ell)}, \qquad \tilde{\mathbf{h}}^{(\ell)} = \Sigma^{(\ell)}\big(\tilde{\mathbf{z}}^{(\ell)}\big)$$

Thus the expected post-activation output is

$$\begin{aligned}\mathbb{E}\Big[\tilde{\mathbf{h}}^{(\ell)}\Big] &= \mathbb{E}\Big[\Sigma^{(\ell)}\big(\tilde{\mathbf{z}}^{(\ell)}\big)\Big] \\ &= \mathbb{E}\Big[\Sigma^{(\ell)}\Big(W^{(\ell)}\big(\tilde{\mathbf{d}}^{(\ell-1)} \odot \mathbf{h}^{(\ell-1)}\big) + \mathbf{b}^{(\ell)}\Big)\Big]\end{aligned}$$

However, without imposing additional assumptions on the particular activation function $\Sigma^{(\ell)}$, this expression cannot be simplified further in general. The impediment is that $\Sigma^{(\ell)}$ is nonlinear, so the expectation operator $\mathbb{E}$ does not, in general, commute with $\Sigma^{(\ell)}$:

$$\mathbb{E}\Big[\Sigma^{(\ell)}(X)\Big] \neq \Sigma^{(\ell)}(\mathbb{E}[X])$$

Hence, the next best thing we can do while remaining activation-agnostic —is to study the mean of the pre-activation $\tilde{\mathbf{z}}^{(\ell)}$, i.e. the mean of the random input fed into the activation function. Since $\mathbf{h}^{(\ell-1)}$ is treated as fixed when taking expectation over the dropout mask at layer ℓ, it is most precise to write this as a conditional expectation, conditioning on a fixed realization of the (possibly random) input activation vector $\tilde{\mathbf{h}}^{(\ell-1)}$:

$$\begin{aligned}\mathbb{E}\Big[\tilde{\mathbf{z}}^{(\ell)} \,\Big|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\Big] &= \mathbb{E}\Big[W^{(\ell)}\big(\tilde{\mathbf{d}}^{(\ell-1)} \odot \tilde{\mathbf{h}}^{(\ell-1)}\big) + \mathbf{b}^{(\ell)} \,\Big|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\Big] \\ &= W^{(\ell)}\,\mathbb{E}\Big[\tilde{\mathbf{d}}^{(\ell-1)} \odot \tilde{\mathbf{h}}^{(\ell-1)} \,\Big|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\Big] + \mathbf{b}^{(\ell)}\end{aligned}$$

(by linearity of conditional expectation)

$$= W^{(\ell)}\Big(\mathbb{E}\Big[\tilde{\mathbf{d}}^{(\ell-1)} \,\Big|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\Big] \odot \mathbf{h}^{(\ell-1)}\Big) + \mathbf{b}^{(\ell)}$$

If, as in standard dropout, the mask $\tilde{\mathbf{d}}^{(\ell-1)}$ is sampled independently of $\tilde{\mathbf{h}}^{(\ell-1)}$, then

$$\mathbb{E}\Big[\tilde{\mathbf{d}}^{(\ell-1)} \,\Big|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\Big] = \mathbb{E}\Big[\tilde{\mathbf{d}}^{(\ell-1)}\Big]$$

Since the mask coordinates are i.i.d. Bernoulli with $\mathbb{E}[\tilde{d}_j^{(\ell-1)}] = p^{(\ell-1)}$, we have

$$\mathbb{E}\Big[\tilde{\mathbf{d}}^{(\ell-1)}\Big] = p^{(\ell-1)}\mathbf{1}_{d_{\ell-1}},$$

and therefore

$$\mathbb{E}\Big[\tilde{\mathbf{z}}^{(\ell)} \,\Big|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\Big] = W^{(\ell)}\big(p^{(\ell-1)}\mathbf{h}^{(\ell-1)}\big) + \mathbf{b}^{(\ell)}$$

This can be interpreted as follows: under (vanilla) dropout training, the activation function at layer ℓ is, on average over dropout-mask realizations, fed a pre-activation whose contribution from the previous layer is effectively scaled by $p^{(\ell-1)}$ during the forward pass.

However, if at test time we disable dropout and apply the standard forward pass without any compensating scale factor, then the pre-activation at layer ℓ becomes

$$\mathbf{z}_{\text{test}}^{(\ell)} = W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}$$

By contrast, the mean pre-activation seen during training under dropout is

$$\mathbb{E}\left[\tilde{\mathbf{z}}^{(\ell)} \,\middle|\, \tilde{\mathbf{h}}^{(\ell-1)} = \mathbf{h}^{(\ell-1)}\right] = W^{(\ell)}\big(p^{(\ell-1)}\mathbf{h}^{(\ell-1)}\big) + \mathbf{b}^{(\ell)}$$

Notice that these two expressions differ only by a factor of $p^{(\ell-1)}$ in the contribution from $\mathbf{h}^{(\ell-1)}$.

This means that, without test-time scaling, the argument to the activation function at layer ℓ is systematically (i.e., not randomly) larger than the "typical" (mean) argument seen during dropout training. Therefore, the authors apply a compensating "test-time scale factor" at test time so that the deterministic *test-time pre-activations* match the *training-time pre-activations in expectation*, thereby avoiding a train–test scale mismatch in the inputs to the network's nonlinearities, $\Sigma^{(\ell)}$ between training and inference.

In short, comparing the two forward-pass formulations (with and without test-time scaling), the issue is that omitting the scaling induces a systematic *distribution shift* in the hidden activations that feed into subsequent layers: the network is then evaluated under activation (and hence logit) magnitudes that it was not trained to handle.

Accordingly, at layer ℓ, a common deterministic test-time forward-pass computation is

$$\mathbf{h}^{(\ell)} = \Sigma^{(\ell)}\Big(W^{(\ell)}\big(\underbrace{p^{(\ell-1)}}_{\text{test time scale factor}}\mathbf{h}^{(\ell-1)}\big) + \mathbf{b}^{(\ell)}\Big)$$

where the factor $p^{(\ell-1)}$ compensates for the expected fraction of activations kept during training-time dropout.

12. Feed forward Neural Network with Monte Carlo Dropout

13. Feed forward Neural Network with Residual

References

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.