Automatic Differentiation of Tensor Operations

Contents

1 Introduction

An overview of automatic differentiation (autograd), computational graphs, and the use of the chain rule to propagate gradients backwards through a network of tensor operations.

2 Prerequisites

This document requires a working knowledge of several foundational topics in mathematics and computer science. The key prerequisites are itemized below.

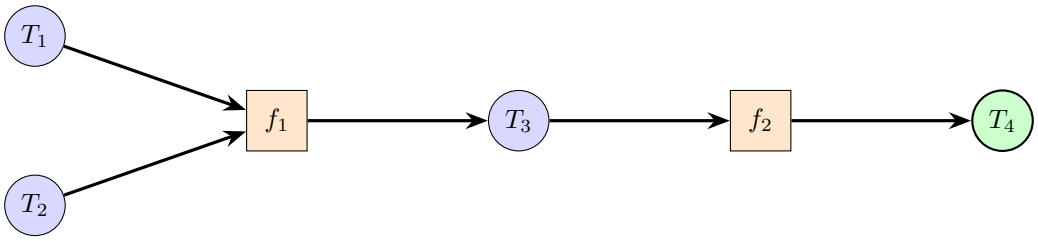
- **Calculus:** Proficiency in single and multi-variable calculus is assumed. A firm grasp of differentiation, partial derivatives, and the chain rule is essential, as the chain rule is the mathematical basis for backpropagation.
- **Linear Algebra:** The core data structures in neural networks are tensors. Familiarity with vectors, matrices, and their fundamental operations—including matrix multiplication, transposition, and element-wise products—is required.
- **Programming:** All code is implemented in Python with the NumPy library. Practical knowledge of Python syntax and experience with NumPy for creating, indexing, and transforming arrays is necessary. An understanding of NumPy's broadcasting mechanism is also expected.
- **Machine Learning Concepts:** A conceptual understanding of the machine learning training process provides essential context. This includes the definitions of a model, its parameters, a loss function, and the role of gradient descent in minimizing that loss.

3 Computation Graphs

3.1 What is a Computation Graph?

A *computation graph* is a directed acyclic graph (DAG) whose circle nodes represent *tensors* T_i (stored numeric values) and whose square nodes represent the elementary *operations* f_j that transform one or more tensors into a new tensor. Inputs appear as leaf nodes, intermediate tensors as interior nodes, and the final output (often a loss) is the unique sink. Decomposing a complex function into such primitive steps enables efficient forward evaluation and, crucially, the application of the chain rule for automatic differentiation on the backward pass.

3.2 Abstract Computation Graph and the Chain Rule



Let $T_3 = f_1(T_1, T_2)$ and $T_4 = f_2(T_3)$. By the multivariable chain rule the gradient of the result with respect to each input factorises into local derivatives along the path:

$$\frac{\partial T_4}{\partial T_1} = \frac{\partial T_4}{\partial T_3}\,\frac{\partial T_3}{\partial T_1}, \qquad \frac{\partial T_4}{\partial T_2} = \frac{\partial T_4}{\partial T_3}\,\frac{\partial T_3}{\partial T_2}.$$

More generally, for a chain of tensors $T_1 \to T_2 \to \ldots \to T_n$ we have

$$\frac{\partial T_n}{\partial T_1} = \prod_{k=1}^{n-1} \frac{\partial T_{k+1}}{\partial T_k}.$$

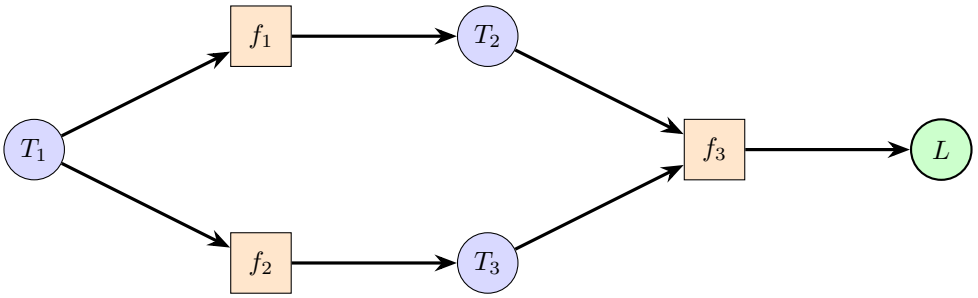
Automatic differentiation exploits this property by traversing the graph in reverse (*back-propagation*), multiplying the local Jacobians encountered along every computational path.

3.3 Multiple Paths and Gradient Accumulation

Real networks often *reuse* the same tensor in several branches (skip connections, residual blocks, etc.). A single source tensor may therefore reach the loss through multiple independent paths. The chain rule is linear, so the overall gradient with respect to that tensor is the *sum* of the pathwise contributions.

More formally, let $\mathcal{P}(s, L)$ be the set of directed paths from a tensor T_s to the loss L. Then

$$\frac{\partial L}{\partial T_s} = \sum_{p \in \mathcal{P}(s,L)} \prod_{(u \to v) \in p} \frac{\partial T_v}{\partial T_u}.$$



With $T_2 = f_1(T_1),\ T_3 = f_2(T_1),\ L = f_3(T_2, T_3)$ the gradient w.r.t. T_1 splits into two terms:

$$\frac{\partial L}{\partial T_1} = \underbrace{\frac{\partial L}{\partial T_2}\,\frac{\partial T_2}{\partial T_1}}_{\text{upper path}} + \underbrace{\frac{\partial L}{\partial T_3}\,\frac{\partial T_3}{\partial T_1}}_{\text{lower path}}.$$

Each term is a product of local derivatives along its path; the final gradient is their sum. In matrix form, modern autodiff systems accumulate this sum by adding the partial gradient coming back along every incoming edge of a node during the backward sweep.

3.4 Dynamic Computation Graphs

Traditional "static" frameworks (e.g. early TensorFlow, Theano) require users to build the *entire* computation graph symbolically before any data can flow through it. By contrast, **dynamic** computation graphs are *defined on–the–fly*: the graph is constructed node–by–node as the forward pass executes native Python code. Control–flow statements such as 'if', 'for', and 'while' therefore influence the *structure* of the graph that is recorded for that specific input.

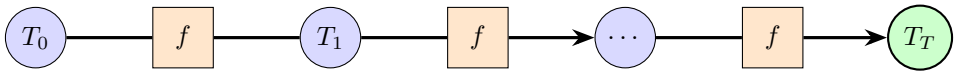
Why dynamic graphs matter

- **Variable–length sequences.** Recurrent networks naturally loop over the time dimension; sequence length T is data–dependent.
- **Conditional computation.** Mixture–of–experts and sparsely activated layers execute only a subset of branches based on the input.
- **Ease of debugging.** Because graph construction follows normal execution, developers can insert breakpoints or print statements anywhere in the model code.

Abstract example Suppose we unroll an RNN over a sequence of length T. At step k we create an output tensor T_k using the same operation f but *different data* (each step has its own node):

$$T_k = f\big(T_{k-1},\, X_k\big), \qquad k = 1, \ldots, T.$$

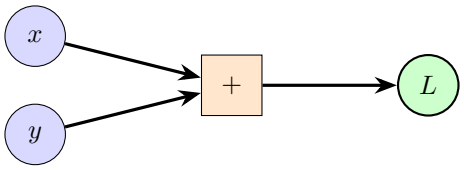
The dynamic graph for a single input sequence is therefore a chain of length T. Gradient back–propagation follows the chain in reverse (*Backprop through Time*) and automatically stops at the earliest time step that was actually executed.



Because the graph can differ for every forward pass, the autodiff engine keeps only the nodes actually executed, leading to memory savings for models with conditional branches.

3.5 A simple example

1. $L = x + y$

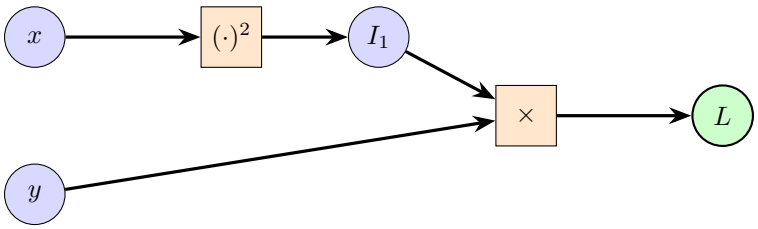


Tensor Breakdown

Tensor	Inputs	Operation
x	—	—
y	—	—
L	x, y	$+$

3.6 A more complex example

2. $L = x^2 \cdot y$

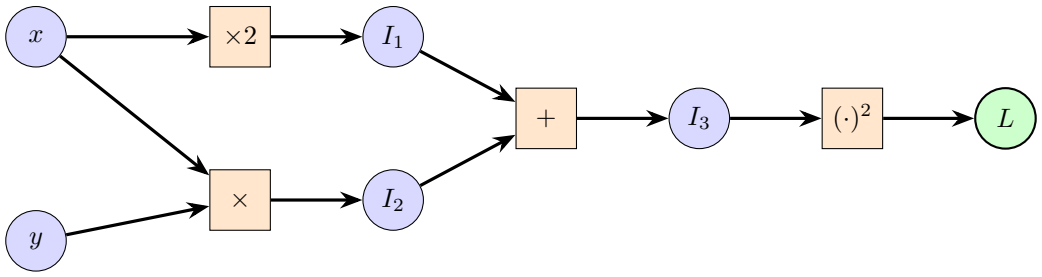


Tensor Breakdown

Tensor	Inputs	Operation
x	—	—
y	—	—
I_1	x	x^2
L	I_1, y	$\times$

3.7 An example with tensor re-use

3. $L = (2x + xy)^2$



Tensor Breakdown

Tensor	Inputs	Operation
x	—	—
y	—	—
I_1	x	$\times 2$
I_2	x, y	$\times$
I_3	I_1, I_2	$+$
L	I_3	x^2

4 Gradient Derivations for Tensor Operations

In what follows, let R denote the result of an operation and X denote an input tensor. We derive $\partial R/\partial X$ for each primitive operation implemented in `tensor.py`.

4.1 Addition & Subtraction

Let $R = A + B$ with $A, B \in \mathbb{R}^{d_1 \times \cdots \times d_k}$. During back-propagation we receive an upstream tensor $G = \partial L/\partial R$ of identical shape.

Local derivatives Because addition acts elementwise,

$$\frac{\partial R_{i_1 \ldots i_k}}{\partial A_{i_1 \ldots i_k}} = 1, \qquad \frac{\partial R_{i_1 \ldots i_k}}{\partial B_{i_1 \ldots i_k}} = 1.$$

Gradient rules Applying the chain rule elementwise gives

$$\boxed{\frac{\partial L}{\partial A} = G}, \qquad \boxed{\frac{\partial L}{\partial B} = G}.$$

If instead $R = A - B$, the derivative w.r.t. A is unchanged while B picks up a minus sign:

$$\boxed{\frac{\partial L}{\partial A} = G}, \qquad \boxed{\frac{\partial L}{\partial B} = -G}.$$

Simplified implementation

```
# Upstream gradient: G (same shape as A and B)
dA = G                # for addition and subtraction
dB =  G      # addition
dB = -G      # subtraction
```

Broadcasting: if A or B were broadcast in the forward pass, sum G along the broadcast axes when accumulating into that operand.

4.2 Element–wise (Hadamard) Multiplication

Let $R = A \odot B$ where "$\odot$" denotes elementwise (Hadamard) multiplication. Components satisfy $R_{i\ldots} = A_{i\ldots} B_{i\ldots}$.

Local derivatives

$$\frac{\partial R_{i\ldots}}{\partial A_{i\ldots}} = B_{i\ldots}, \qquad \frac{\partial R_{i\ldots}}{\partial B_{i\ldots}} = A_{i\ldots}.$$

Gradient rules Given upstream G:

$$\boxed{\frac{\partial L}{\partial A} = G \odot B}, \qquad \boxed{\frac{\partial L}{\partial B} = G \odot A}.$$

Simplified implementation

```
dA = G * B   # elementwise product
dB = G * A
```

4.3 Sigmoid

The sigmoid (logistic) activation is applied elementwise:

$$R = \sigma(X) = \frac{1}{1 + e^{-X}}.$$

Let $G = \partial L / \partial R$ be the incoming gradient.

Local derivative A convenient identity is

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big).$$

Thus elementwise

$$\frac{\partial R}{\partial X} = R\,(1 - R).$$

Gradient rule

$$\boxed{\frac{\partial L}{\partial X} = G \odot R \odot (1 - R)}.$$

Simplified implementation

```
# R is the forward-pass output sigma(X)
dX = G * R * (1 - R)
```

Numerical stability tip: clip X to a reasonable range (e.g. $[-500, 500]$) before calling e^{-X}—see the safeguard in `sigmoid_tensor`.

4.4 Scalar Multiplication

Consider scaling a tensor by a constant scalar: $R = c\,A$ where $c \in \mathbb{R}$ and $A \in \mathbb{R}^{d_1 \times \cdots \times d_k}$. In most deep-learning codebases c is a literal or a hyper-parameter and therefore *does not* require a gradient; nevertheless we derive it for completeness.

Local derivatives Elementwise we have $R_{i\ldots} = c\,A_{i\ldots}$, so

$$\frac{\partial R_{i\ldots}}{\partial A_{i\ldots}} = c, \qquad \frac{\partial R_{i\ldots}}{\partial c} = A_{i\ldots}.$$

Gradient w.r.t. the tensor A Let $G = \partial L / \partial R$.

$$\boxed{\frac{\partial L}{\partial A} = c\,G}.$$

This is simply a rescaling of the upstream gradient.

Gradient w.r.t. the scalar c (optional) Flattening all indices, the total derivative accumulates over every element of A:

$$\frac{\partial L}{\partial c} = \sum_{i_1,\ldots,i_k} G_{i_1\ldots i_k}\; A_{i_1\ldots i_k} = \langle G, A \rangle,$$

where $\langle \cdot, \cdot \rangle$ denotes the Frobenius inner product.

Simplified implementation

```
# Forward: R = c * A
# Backward inputs: G (same shape as A)

dA = c * G              # gradient for tensor
if requires_grad_c:
    dc = np.sum(G * A)  # gradient for scalar
```

The function `scalar_multiply_tensor` in `tensor.py` follows this rule, propagating gradients only to the tensor argument because the scalar is treated as a constant.

4.5 Division

Let $R = A/B$ denote elementwise division with $A, B \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ and B containing no zeros. During the backward pass we receive an upstream gradient $G = \partial L / \partial R$ of matching shape.

Local derivatives For each entry i (multi–index suppressed):

$$\frac{\partial R_i}{\partial A_i} = \frac{1}{B_i}, \qquad \frac{\partial R_i}{\partial B_i} = -\frac{A_i}{B_i^2}.$$

Gradient rules

$$\boxed{\frac{\partial L}{\partial A} = \frac{G}{B}}, \qquad \boxed{\frac{\partial L}{\partial B} = -G \odot \frac{A}{B^2}}.$$

Simplified implementation

```
dA = G / B
dB = -G * A / (B ** 2)
```

If broadcasting occurred in the forward pass, sum the resulting gradients over broadcast axes before adding them to the stored gradients, mirroring `tensor_divide` in `tensor.py`.

4.6 Power

Consider raising each element of a tensor to a *scalar* exponent p: $R = A^p$ with $A \geq 0$ elementwise (to keep derivatives real). The exponent $p \in \mathbb{R}$ is typically a constant.

Local derivative For each element i:

$$\frac{\partial R_i}{\partial A_i} = p\, A_i^{p-1}.$$

Gradient rule Given upstream G:

$$\boxed{\frac{\partial L}{\partial A} = G \odot \left(p\, A^{p-1}\right)}.$$

Optional gradient w.r.t. the exponent p If p were a learnable parameter we would also obtain

$$\frac{\partial L}{\partial p} = \sum_i G_i\, A_i^p\, \ln A_i.$$

This is seldom required in practice, so `tensor_pow` only propagates to A.

Simplified implementation

```
dA = G * (p * (A ** (p - 1)))
# If p requires grad:
dp = np.sum(G * (A ** p) * np.log(A))
```

4.7 Sum (Reduction)

Let $R = \mathrm{sum}(A)$ denote the sum of all elements in $A \in \mathbb{R}^{d_1 \times \cdots \times d_k}$. The forward pass returns a scalar; during back-propagation we receive an upstream scalar $g = \partial L / \partial R$.

Local derivative Because every entry of A contributes linearly,

$$\frac{\partial R}{\partial A_{i_1 \ldots i_k}} = 1 \quad \text{for all indices.}$$

Gradient rule

$$\boxed{\frac{\partial L}{\partial A} = g\, \mathbf{1}},$$

where $\mathbf{1}$ is a tensor of ones with the same shape as A.

Simplified implementation

```
# Forward: R = A.sum()
# Backward: upstream scalar g
dA = g * np.ones_like(A)
```

This matches `tensor_sum` in `tensor.py`.

4.8 Mean (Reduction)

The mean divides the sum by the number of elements $N = d_1 \cdots d_k$:

$$R = \text{mean}(A) = \frac{1}{N} \sum_{i_1,\ldots,i_k} A_{i_1 \ldots i_k}.$$

The upstream gradient is again a scalar $g = \partial L / \partial R$.

Local derivative

$$\frac{\partial R}{\partial A_{i_1 \ldots i_k}} = \frac{1}{N}.$$

Gradient rule

$$\boxed{\frac{\partial L}{\partial A} = \frac{g}{N}\,\mathbf{1}}.$$

Simplified implementation

```
N = A.size

dA = g / N * np.ones_like(A)
```

`tensor_mean` in `tensor.py` uses this exact scaling factor.

4.9 Reshape

Reshaping changes the view of the data without altering its values. Let

$$R = \text{reshape}(A,\ \text{shape}_{\text{new}}), \qquad A \in \mathbb{R}^{d_1 \times \cdots \times d_k},\ R \in \mathbb{R}^{s_1 \times \cdots \times s_m},$$

where the total number of elements $N = d_1 \cdots d_k = s_1 \cdots s_m$ is preserved. During back-propagation we receive an upstream gradient $G = \partial L / \partial R$ having shape $(s_1, \ldots, s_m)$.

Local derivative Reshape is a *data-layout* operation: every output element corresponds one-to-one with an input element. Hence

$$\frac{\partial R_j}{\partial A_i} = \delta_{ij},$$

with i and j the flattened indices. The Jacobian is therefore the $N \times N$ identity matrix.

Gradient rule Because the Jacobian is the identity, back-prop simply reinterprets the storage back into the original shape:

$$\boxed{\frac{\partial L}{\partial A} = \text{reshape}(G,\ d_1, \ldots, d_k)}.$$

Simplified implementation

```
dA = G.reshape(A.shape)
```

This mirrors the behaviour of `tensor_reshape` in `tensor.py`.

4.10 Transpose

For a matrix input let

$$R = A^\top, \qquad A \in \mathbb{R}^{m\times n},\ R \in \mathbb{R}^{n\times m}.$$

Define $G = \partial L/\partial R$ with shape $n \times m$.

Local derivatives Elementwise $R_{ij} = A_{ji}$, so

$$\frac{\partial R_{ij}}{\partial A_{pq}} = \delta_{ip}\,\delta_{jq}.$$

Gradient rule Applying the chain rule yields

$$\boxed{\frac{\partial L}{\partial A} = G^\top}.$$

Illustrative example Let

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}, \quad R = A^\top.$$

If the upstream gradient is

$$G = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \end{bmatrix},$$

then

$$\frac{\partial L}{\partial A} = G^\top = \begin{bmatrix} g_{11} & g_{21} & g_{31} \\ g_{12} & g_{22} & g_{32} \end{bmatrix},$$

matching the original shape of A.

Simplified implementation

```
dA = G.T
```

The function `tensor_transpose` in `tensor.py` implements this exact rule.

4.11 Concatenation

Let $R = \text{concat}(A,\, B;\, \text{axis} = k)$ join two tensors of identical rank along axis k. Suppose

$$A \in \mathbb{R}^{d_1\times\cdots\times d_k^{(A)}\times\cdots\times d_m}, \quad B \in \mathbb{R}^{d_1\times\cdots\times d_k^{(B)}\times\cdots\times d_m}$$

so the result has size $d_k = d_k^{(A)} + d_k^{(B)}$ along that axis. During back-propagation we receive an upstream gradient $G = \partial L/\partial R$ of the same shape as R.

Local derivatives Concatenation merely copies elements; the Jacobian is therefore a block-diagonal matrix that routes each slice of G back to the corresponding input tensor. Concretely,

$$\frac{\partial R_{\boldsymbol{i}}}{\partial A_{\boldsymbol{i}}} = 1 \quad \text{if the index lies in the } A \text{ slice}, \qquad \frac{\partial R_{\boldsymbol{i}}}{\partial B_{\boldsymbol{i}}} = 1 \quad \text{if the index lies in the } B \text{ slice},$$

and zero otherwise.

Gradient rule Split G along axis k using the original sizes of A and B:

$$G^{(A)},\ G^{(B)} = \text{split}(G,\ [d_k^{(A)}],\ \text{axis} = k).$$

Then

$$\boxed{\frac{\partial L}{\partial A} = G^{(A)}}, \qquad \boxed{\frac{\partial L}{\partial B} = G^{(B)}}.$$

Illustrative example Consider two 2×2 matrices concatenated *row-wise* (axis 0):

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

The result is a 4×2 matrix

$$R = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

If the upstream gradient is

$$G = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \\ g_{41} & g_{42} \end{bmatrix},$$

we simply slice it:

$$\frac{\partial L}{\partial A} = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix}, \qquad \frac{\partial L}{\partial B} = \begin{bmatrix} g_{31} & g_{32} \\ g_{41} & g_{42} \end{bmatrix}.$$

Simplified implementation

```
# axis = k, size_a = A.shape[k]
G_a, G_b = np.split(G, [size_a], axis=k)

dA = G_a

dB = G_b
```

The autograd function `tensor_concat` in `tensor.py` follows exactly this logic, splitting the upstream gradient along the concatenation axis before accumulating it into the input tensors.

4.12 Matrix Multiplication

Let $R = AB$ where $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$. Throughout this section L denotes the final scalar loss and $G = \partial L / \partial R$ is the upstream gradient supplied by later nodes in the computational graph.

Forward pass

$$R = AB, \quad R_{ij} = \sum_{s=1}^{k} a_{is} b_{sj}.$$

Element–wise partial derivatives For a single entry r_{ij} we have

$$\frac{\partial r_{ij}}{\partial a_{pq}} = b_{qj}\, \delta_{ip}, \qquad \frac{\partial r_{ij}}{\partial b_{pq}} = a_{ip}\, \delta_{qj},$$

where δ is the Kronecker delta.

Kronecker delta The Kronecker delta δ_{ij} is a discrete analogue of the identity function:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

It 'picks out' the terms where the indices coincide, ensuring that a partial derivative with respect to a_{pq} (or b_{pq}) only receives contributions from entries r_{ij} that actually depend on that element.

Illustrative example (2×2 case) Let $m = n = k = 2$ and

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Then

$$R = AB = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}.$$

One can directly verify the earlier formulas: for instance,

$$\frac{\partial R_{11}}{\partial a_{12}} = b_{21}, \qquad \frac{\partial R_{11}}{\partial b_{21}} = a_{12},$$

while derivatives with mismatched indices vanish because of the Kronecker delta.

Transpose as a shortcut for the full gradient Computing every partial derivative individually quickly becomes tedious. Notice, however, that the summation $\sum_j G_{pj} b_{qj}$ that appears in $\partial L/\partial A$ is precisely the (p, q) entry of the matrix product $GB^\top$. The transpose operator swaps the indices of B so that its first index lines up with the second index of G:

$$(GB^\top)_{pq} = \sum_j G_{pj} B^\top_{jq} = \sum_j G_{pj} b_{qj}.$$

The same index–swapping idea explains $A^\top$ in $\partial L/\partial B = A^\top G$.

For the 2×2 case let the upstream gradient be

$$G = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix}.$$

Form the transpose of B and multiply:

$$B^\top = \begin{bmatrix} b_{11} & b_{21} \\ b_{12} & b_{22} \end{bmatrix}, \qquad GB^\top = \begin{bmatrix} g_{11}b_{11} + g_{12}b_{12} & g_{11}b_{21} + g_{12}b_{22} \\ g_{21}b_{11} + g_{22}b_{12} & g_{21}b_{21} + g_{22}b_{22} \end{bmatrix}.$$

Compare the $(1, 2)$ entry to the manual derivative $\partial L/\partial a_{12} = \sum_j G_{1j} b_{2j} = g_{11}b_{21} + g_{12}b_{22}$, which matches exactly. The transpose thus provides a *structural* shortcut: by swapping indices, it aligns the correct rows and columns so that ordinary matrix multiplication performs all the required summations in one shot.

Gradient with respect to A Applying the chain rule

$$\frac{\partial L}{\partial a_{pq}} = \sum_{i,j} \frac{\partial L}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial a_{pq}} = \sum_j G_{pj} b_{qj}.$$

Recognising the right–hand side as a row of G times a column of $B^\top$ leads to the compact matrix form

$$\boxed{\frac{\partial L}{\partial A} = GB^\top}$$

with shape $m \times k$ matching A.

Gradient with respect to B An analogous argument yields

$$\boxed{\frac{\partial L}{\partial B} = A^\top G}$$

with shape $k \times n$ matching B.

Simplified implementation (Python–like)

```
# G: upstream gradient dL/dR  (m x n)
# A: left operand (m x k)
# B: right operand (k x n)

dA = G @ B.T      # (m x n) @ (n x k) -> (m x k)
dB = A.T @ G      # (k x m) @ (m x n) -> (k x n)
```

This matches the logic found in `matmul_tensor`'s `_backward` closure in `tensor.py`, but omits runtime checks for clarity.

4.13 ReLU

The Rectified Linear Unit (ReLU) is applied elementwise:

$$R = \mathrm{ReLU}(X) = \max(0, X).$$

Let $G = \partial L/\partial R$ be the upstream gradient with the same shape as X.

Local derivative

$$\frac{\partial R}{\partial X} = \begin{cases} 1 & X > 0, \\ 0 & X \leq 0. \end{cases}$$

Gradient rule Define the indicator tensor $I = \mathbf{1}_{\{X>0\}}$. Then

$$\boxed{\frac{\partial L}{\partial X} = G \odot I}.$$

Simplified implementation

```
I = (X > 0).astype(X.dtype)
dX = G * I
```

This matches the behaviour of `relu_tensor` in `tensor.py`.

4.14 Leaky ReLU

Leaky ReLU introduces a small slope α ($0 < \alpha \ll 1$) for negative inputs:

$$R = \mathrm{LReLU}(X; \alpha) = \begin{cases} X & X > 0, \\ \alpha X & X \leq 0. \end{cases}$$

Upstream gradient G again has the same shape as X.

Local derivative

$$\frac{\partial R}{\partial X} = \begin{cases} 1 & X > 0, \\ \alpha & X \leq 0. \end{cases}$$

Gradient rule Let $I_+ = \mathbf{1}_{\{X>0\}}$ and $I_- = 1 - I_+$. Then

$$\boxed{\frac{\partial L}{\partial X} = G \odot \big(I_+ + \alpha I_-\big)}.$$

Simplified implementation

```
I_pos = (X > 0).astype(X.dtype)
I_neg = 1 - I_pos

dX = G * (I_pos + alpha * I_neg)
```

The function `leaky_relu_tensor` in `tensor.py` follows the same pattern, defaulting to $\alpha = 0.01$.

5 Loss Functions

Loss functions quantify how well a model's predictions $\hat{y}$ match the true target values y. During training the network parameters are optimised to minimise the chosen loss. Because they are expressed solely in terms of primitive tensor operations (addition, multiplication, power, logarithm *etc.*), defining a new loss in our framework is no different from writing an ordinary forward pass.

5.1 Defining a new loss is just composing tensors

Let us recall that every call to a tensor operation immediately creates both

1. a new *tensor* that stores the numerical result, and
2. a corresponding *operation node* that links it to its parents in the computation graph.

Consequently a one-line Python expression such as

```
loss = ((y_true - y_pred) ** 2).mean()
```

instantiates four graph nodes (`sub`, `pow`, `mean`, and the final L) and three intermediate tensors. No additional boiler-plate is required—the graph is built dynamically while the user writes idiomatic NumPy-like code.

5.2 Gradients via automatic differentiation

During the backward pass the library walks the graph in reverse order, propagating partial derivatives from the loss back to each parameter according to

$$\delta_T \;=\; \sum_{k \in \,\mathrm{out}(T)} \delta_k \, \frac{\partial T_k}{\partial T},$$

where $\delta_T = \partial L / \partial T$ and $\mathrm{out}(T)$ denotes all descendants that take T as input. Because every primitive operation already knows its own local Jacobian, the global gradient emerges by repeated application of the chain rule—exactly as described in Section 3.4.

5.3 Example: Mean–Squared Error (MSE)

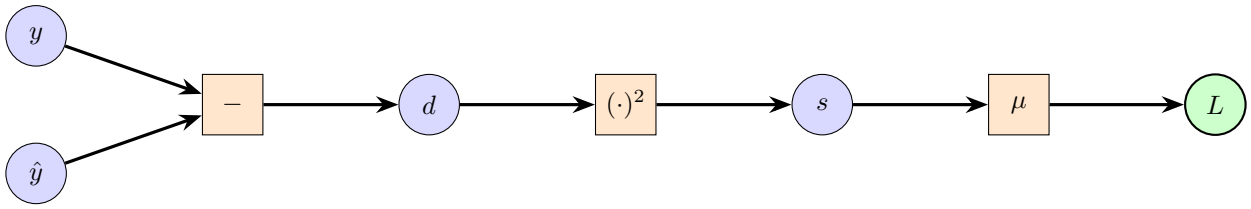
For concreteness consider the Mean–Squared Error

$$L_{\mathrm{MSE}} \;=\; \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.$$

The forward pass can be written in three lines:

```
diff   = y_true - y_pred
sq_err = diff ** 2
loss   = sq_err.mean()
```

Below we depict the resulting computation graph.



The gradient with respect to the prediction is recovered automatically:

$$\frac{\partial L}{\partial \hat{y}} = \frac{2}{n}(\hat{y} - y),$$

which is exactly the hand-derived formula; the autodiff engine has taken care of the algebra.

5.4 Beyond MSE

Other popular losses—Mean Absolute Error, Binary Cross-Entropy, Hinge loss, *etc.*—are all definable with the same three-step recipe:

1. Express the loss as a composition of primitive operations.
2. Execute the forward pass to build the dynamic graph.
3. Call `backward()` on the scalar loss tensor to obtain every required gradient.

Because the machinery is identical, we omit the individual DAGs for brevity; the interested reader is encouraged to draw them by substituting the corresponding operations (`abs`, `log`, `max`...) into the template above.

6 Layer Implementations

Future sections derive gradients for *operations*. It is equally instructive to see how complete **layers**—the building blocks of neural networks—can be assembled entirely from those primitives. This section provides a high-level map that connects each layer in `autograd/layers` to the underlying tensor operations that power it. Detailed mathematical derivations will follow in future revisions; for now we list the core ingredients.

- **Design principles**
  - Each concrete layer inherits from `BaseLayer` which unifies parameter handling, training/eval mode switching, and basic SGD.
  - The `forward` method composes primitive tensor operations (see the previous section) and therefore obtains automatic gradients for free.

6.1 Dense (Fully Connected)

File: ***`dense.py`***

- *Parameters*: weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and bias vector $b \in \mathbb{R}^{d_{\text{out}}}$. Both are `Tensors` set to `requires_grad=True`.
- *Forward*: $Y = XW^\top + b$
  - Matrix multiplication: `matmul_tensor`
  - Broadcasting addition: `tensor_add`
- Xavier/Glorot initialisation sets W using `numpy.random.uniform`.

6.2 Activation Layers

- Common trait: they apply an element–wise non-linearity implemented as a single primitive tensor op.

ReLU *File: activations.py*

- Operation: $Y = \max(0, X)$ via `relu_tensor`.

Sigmoid *File: activations.py*

- Operation: $Y = \sigma(X)$ via `sigmoid_tensor`.

Leaky ReLU *File: activations.py*

- Operation: $Y = \begin{cases} X & X > 0 \\ \alpha X & X \leq 0 \end{cases}$ via `leaky_relu_tensor` with slope α (default: $\alpha = 0.01$).

6.3 Dropout

File: dropout.py

- Training mode: sample Bernoulli mask $M \sim \text{Bernoulli}(1 - p)$.
- Forward: $Y = \frac{1}{1-p} X \odot M$ using element-wise multiplication (`tensor_mul`) and scalar scaling.
- Evaluation mode: identity map ($Y = X$).

6.4 Skip Connection (Residual)

File: skip_connection.py

- Wraps an inner layer f and returns $Y = f(X) + X$.
- Uses `tensor_add` to combine the original input with the wrapped layer's output. Requires matching shapes.

7 Loss Functions

Loss functions quantify how well a model's predictions $\hat{y}$ match the true target values y. During training the network parameters are optimised to minimise the chosen loss. Because they are expressed solely in terms of primitive tensor operations (addition, multiplication, power, logarithm *etc.*), defining a new loss in our framework is no different from writing an ordinary forward pass.

7.1 Defining a new loss is just composing tensors

Let us recall that every call to a tensor operation immediately creates both

1. a new *tensor* that stores the numerical result, and
2. a corresponding *operation node* that links it to its parents in the computation graph.

Consequently a one-line Python expression such as

```
loss = ((y_true - y_pred) ** 2).mean()
```

instantiates four graph nodes (`sub`, `pow`, `mean`, and the final L) and three intermediate tensors. No additional boiler-plate is required—the graph is built dynamically while the user writes idiomatic NumPy-like code.

7.2 Gradients via automatic differentiation

During the backward pass the library walks the graph in reverse order, propagating partial derivatives from the loss back to each parameter according to

$$\delta_T \;=\; \sum_{k \in \,\mathrm{out}(T)} \delta_k \,\frac{\partial T_k}{\partial T},$$

where $\delta_T = \partial L / \partial T$ and $\mathrm{out}(T)$ denotes all descendants that take T as input. Because every primitive operation already knows its own local Jacobian, the global gradient emerges by repeated application of the chain rule—exactly as described in Section 3.4.

7.3 Example: Mean–Squared Error (MSE)

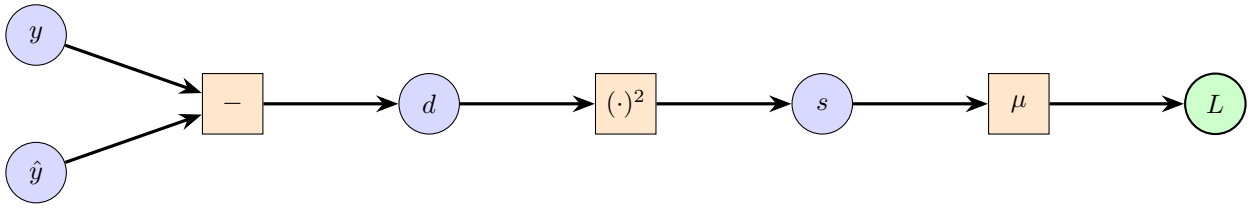
For concreteness consider the Mean–Squared Error

$$L_{\mathrm{MSE}} \;=\; \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

The forward pass can be written in three lines:

```
diff   = y_true - y_pred
sq_err = diff ** 2
loss   = sq_err.mean()
```

Below we depict the resulting computation graph.



The gradient with respect to the prediction is recovered automatically:

$$\frac{\partial L}{\partial \hat{y}} \;=\; \frac{2}{n}\,(\hat{y} - y),$$

which is exactly the hand-derived formula; the autodiff engine has taken care of the algebra.

7.4 Beyond MSE

Other popular losses—Mean Absolute Error, Binary Cross-Entropy, Hinge loss, *etc.*—are all definable with the same three-step recipe:

1. Express the loss as a composition of primitive operations.
2. Execute the forward pass to build the dynamic graph.
3. Call `backward()` on the scalar loss tensor to obtain every required gradient.

Because the machinery is identical, we omit the individual DAGs for brevity; the interested reader is encouraged to draw them by substituting the corresponding operations (`abs`, `log`, `max`...) into the template above.