The Art of Computer Programming 2

Week 10 — Autograd: Backpropagation from Scratch (*micrograd*)

Félix Chavelli felix.chavelli@inria.fr

May 13, 2026 · Semester 2

PSL UNIVERSITÉ PARIS

Today's Agenda

From Algorithms to Learning Machines

Differentiation: From Numerical to Automatic

Computational Graphs

The Chain Rule & Backpropagation

Automating the Backward Pass

Extending the Operator Set

From a Scalar to a Neural Network

Training: Gradient Descent in Action

Wrap-Up & What's Next

Part 1

From Algorithms to Learning Machines

Where We Stand

So far this semester:

- Sorting, hashing, search trees
- Dynamic programming, greedy
- Graphs: BFS, DFS, topo-sort, SCC
- MST, shortest paths, max-flow

All these algorithms share one trait: a human designed every step.

Today — a paradigm shift:

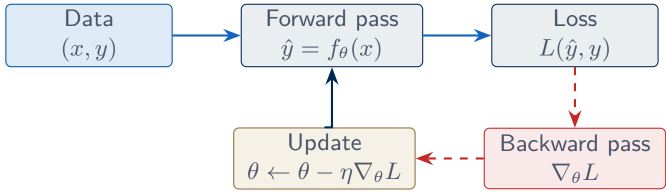
- We *don't* program the rules.
- We define a parameterised function (a neural network).
- We let an optimisation algorithm adjust those parameters from data.

The mathematical engine that makes this practical is automatic differentiation — a.k.a. *autograd*.

Key Idea

Modern deep learning = **enormous** computational graphs + **back-propagation** + **gradient descent**. All three are simple ideas. Today we build them from scratch.

The Training Pipeline of a Neural Network



- **Forward pass**: compute the output and the loss.
- **Backward pass**: compute $\partial L / \partial \theta_i$ for every parameter $\theta_i \Rightarrow$ this is what autograd automates.
- **Update**: nudge each parameter against its gradient.

Repeat... thousands or billions of times.

Why Build It From Scratch?

Production libraries (PyTorch, JAX, TensorFlow):

- operate on n-dimensional *tensors*;
- dispatch to fused CUDA kernels;
- hundreds of thousands of lines of C++.

micrograd:

- operates on *scalars* (one float at a time);
- pure Python, no dependencies;
- $\sim$150 lines total.

None of the math changes between micrograd and PyTorch.

Tensors are merely a vectorised *efficiency* trick over the very same scalar autograd we will write today.

Part 2

Differentiation: From Numerical to Automatic

Numerical Derivatives

Derivative

For a function $f : \mathbb{R} \to \mathbb{R}$, the derivative at x is

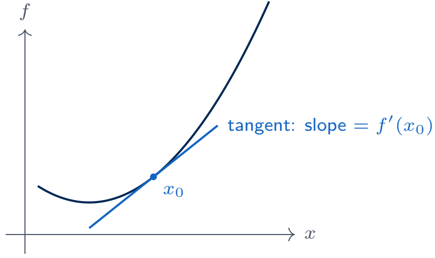
$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Operational meaning:

- The sign of $f'(x)$ tells us the direction f grows.
- The magnitude $|f'(x)|$ tells us *by how much*.

Numerical estimate (h small):

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$



A Single-Variable Warm-Up

Consider the parabola

$$f(x) \;=\; 3x^2 - 4x + 5.$$

Analytic derivative:

$$f'(x) \;=\; 6x - 4.$$

Sanity check at $x = 2/3$: $f'(2/3) = 0$ (vertex of the parabola).

Numerical derivative::

Python

```python
h = 1e-6
x = 2/3
slope = (f(x + h) - f(x)) / h
# slope ~ 0.0  (vertex of parabola)
```

From One Variable to Many: Partial Derivatives

Partial derivative

For $f : \mathbb{R}^n \to \mathbb{R}$, the partial derivative

$$\frac{\partial f}{\partial x_i}(x_1, \ldots, x_n) = \lim_{h \to 0} \frac{f(\ldots, x_i + h, \ldots) - f(\ldots, x_i, \ldots)}{h}$$

measures the slope of f *when only* x_i *varies.*

Three-input example: $d = a \cdot b + c, \quad (a, b, c) = (2, -3, 10).$

- $\partial d/\partial a = b = -3$ (numerical: nudge a by h, d drops by $\sim 3h$).
- $\partial d/\partial b = a = 2.$
- $\partial d/\partial c = 1.$

Key Idea

The vector of all partials is the gradient $\nabla f \in \mathbb{R}^n$.
For a neural network, $\nabla_\theta L$ is exactly what gradient descent needs.

Part 3

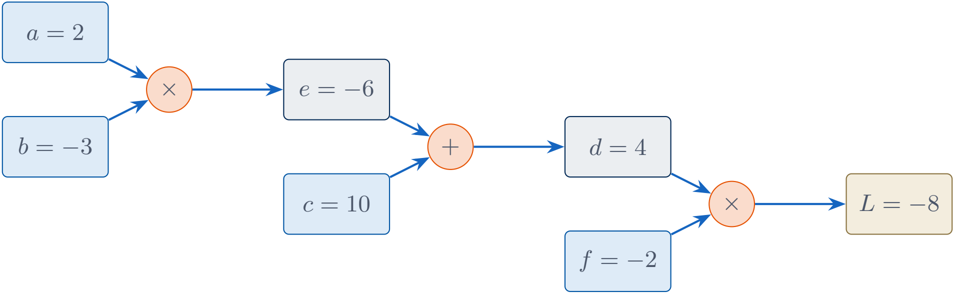
Computational Graphs

Expressions as Graphs

Key Idea

Any composite arithmetic expression can be drawn as a DAG whose *leaves* are inputs/parameters, *internal nodes* are operations, and the *root* is the output.

Running example: $L = (a \cdot b + c) \cdot f$ with $a = 2,\ b = -3,\ c = 10,\ f = -2.$



- **Forward pass**: traverse left-to-right; compute each node from its parents.
- **Backward pass** (next section): traverse right-to-left; compute $\partial L/\partial v$ for every node v.

The `Value` Wrapper

We need each scalar to remember *how it was produced* so we can later walk the graph backwards. Hence a thin Python class:

Python

```
class Value:
    """A single scalar with autograd bookkeeping."""

    def __init__(self, data, _children=(), _op=''):
        self.data      = data             # the numerical value
        self.grad      = 0.0              # dL/dself, filled in by backward()
        self._prev     = set(_children)   # parents in the comp. graph
        self._op       = _op              # which op produced this node
        self._backward = lambda: None     # local backward closure, set by ops
```

- `data`: the forward value.
- `grad`: the partial derivative of the (eventual) output w.r.t. this node.
- `_prev`, `_op`: structural metadata — enough to reconstruct the graph and visualise it.
- `_backward`: the local chain-rule closure attached when the operation runs.

Operator Overloading: Building the Graph

Python

```python
def __add__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data + other.data, (self, other), '+')
    def _backward():        # local rule: d(a+b)/da = d(a+b)/db = 1
        self.grad  += 1.0 * out.grad
        other.grad += 1.0 * out.grad
    out._backward = _backward
    return out

def __mul__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data * other.data, (self, other), '*')
    def _backward():        # local rule: d(a.b)/da = b,  d(a.b)/db = a
        self.grad  += other.data * out.grad
        other.grad += self.data  * out.grad
    out._backward = _backward
    return out
```

Each Python operator we overload *simultaneously*: (i) computes the forward value, (ii) records the parents, (iii) installs the corresponding chain-rule closure.

Part 4

The Chain Rule & Backpropagation

The Chain Rule

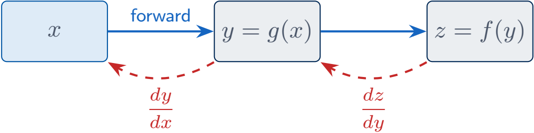
Chain Rule

If $z = f(y)$ and $y = g(x)$ (both differentiable), then

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Multivariable form. If z depends on x through several intermediates $y_1, \ldots, y_k$:

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{k} \frac{\partial z}{\partial y_i} \cdot \frac{\partial y_i}{\partial x}$$



Key Idea

Backprop: **the chain rule applied node by node**, propagated from the output of the graph back to every input.

Backpropagation by Hand — $L = (a \cdot b + c) \cdot f$

Initialise $\partial L/\partial L = 1$, then walk the graph backwards:

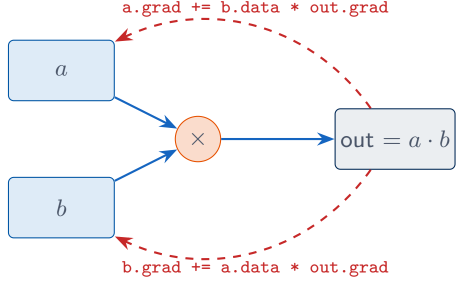
Node	Local rule	Value
L	seed	$\partial L/\partial L = 1$
d	$L = d \cdot f \Rightarrow \partial L/\partial d = f$	-2
f	$\partial L/\partial f = d$	4
c	$d = e + c \Rightarrow \partial L/\partial c = \partial L/\partial d \cdot 1$	-2
e	$\partial L/\partial e = \partial L/\partial d \cdot 1$	-2
a	$e = a \cdot b \Rightarrow \partial L/\partial a = \partial L/\partial e \cdot b$	$-2 \cdot (-3) = 6$
b	$\partial L/\partial b = \partial L/\partial e \cdot a$	$-2 \cdot 2 = -4$

Check with finite differences: bump b by $h{=}10^{-3}$, recompute L, divide by $h \Rightarrow$ slope ≈ -4.

The Local-Closure Trick

Each operation knows only its local derivatives.

When the global backward pass arrives at the output node `out`, we multiply by `out.grad` and *accumulate* into the parents:



Pitfall

Why += and not =? A node can appear as parent of *several* children (e.g. in `b = a + a` the node `a` is a parent twice). The multivariable chain rule says we must sum the contributions; overwriting would silently drop them.

Part 5

Automating the Backward Pass

The Ordering Problem

To call each `node._backward()` correctly, every node must already know its own `grad` before it propagates to its parents.

Constraint: process a node *only after* all of its *children* (downstream nodes) have been processed.

Key Idea

This is exactly a topological sort of the computational DAG — the very algorithm we studied in **Course 6 (Graphs — BFS, DFS, Topological Sort, SCC)**.

Recall: a topological order of a DAG $G = (V, E)$ is a linear ordering of V such that for every edge $(u \to v)$, u appears before v.

$\Rightarrow$ For backprop we want the reverse topological order.

Topological Sort via DFS (Course 6 reminder)

Standard DFS post-order produces a reverse topological order “for free”:

Python

```
def backward(self):
    # 1. Topological sort of all nodes reachable from self
    topo, visited = [], set()
    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:      # recurse on parents in the graph
                build_topo(child)
            topo.append(v)             # post-order: parents first, self last
    build_topo(self)

    # 2. Apply the chain rule in reverse topological order
    self.grad = 1.0                    # seed: dout/dout = 1
    for node in reversed(topo):
        node._backward()               # local closure installed by each op
```

- Time: $\mathcal{O}(V + E)$ in the size of the computational graph.
- Space: $\mathcal{O}(V)$ for `topo` and `visited`.

Part 6

Extending the Operator Set

Beyond + and *

To express a real neural network we need more primitives. Each one ships with its forward formula and its local derivative.

Operation	Forward	Local derivative w.r.t. `self`
`self + other`	$u + v$	1 (and 1 for `other`)
`self * other`	$u \cdot v$	v (and u for `other`)
`self ** k`	u^k, $k \in \mathbb{R}$	$k\,u^{k-1}$
`self.exp()`	e^u	$e^u =$ `out.data`
`self.tanh()`	$\tanh u$	$1 - \tanh^2 u$
`self.relu()`	$\max(0, u)$	$\mathbf{1}[u > 0]$

Once `+`, `*` and `**` exist we get `-`, `/`, unary `-` for free (`a - b = a + (-1)*b`, `a/b = a * b**(-1)`).

We must also implement the *reflected* operators (`__radd__`, `__rmul__`, ...) so that expressions like `2 * a` work.

Example: tanh and exp

Python

```
def tanh(self):
    x = self.data
    t = (math.exp(2*x) - 1) /
    ↪  (math.exp(2*x) + 1)
    out = Value(t, (self,), 'tanh')
    def _backward():
        self.grad += (1 - t**2) *
        ↪  out.grad
    out._backward = _backward
    return out
```

Python

```
def exp(self):
    out = Value(math.exp(self.data),
    ↪  (self,), 'exp')
    def _backward():
        # d/du exp(u) = exp(u) =
        ↪  out.data
        self.grad += out.data *
        ↪  out.grad
    out._backward = _backward
    return out
```

Key Idea

Granularity is a design choice. We can either expose tanh as one node, *or* decompose it as $\tanh u = (e^{2u} - 1)/(e^{2u} + 1)$ using exp, +, −, /.

Both give the exact same gradient — backprop doesn't care, as long as every atomic node knows its local derivative.

Part 7

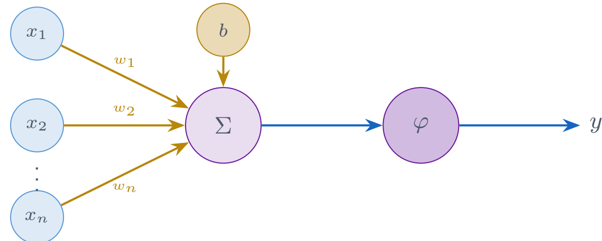
From a Scalar to a Neural Network

The Artificial Neuron

Artificial Neuron

Given inputs $x_1, \ldots, x_n$, weights $w_1, \ldots, w_n$, bias b, and a non-linear activation φ (e.g. $\tanh$, ReLU), an artificial neuron computes

$$y = \varphi\left(\sum_{i=1}^{n} w_i\, x_i + b\right).$$



In micrograd: w_i, b are `Value` objects with random initial data; the whole expression `act = sum(wi*xi for ...) + b` followed by `out = act.tanh()` is itself a Value whose `backward()` we already know how to call.

Implementing a Neuron

Python

```
class Module:
    def zero_grad(self):
        for p in self.parameters():
            p.grad = 0
    def parameters(self):
        return []

class Neuron(Module):
    def __init__(self, nin, nonlin=True):
        self.w = [Value(random.uniform(-1, 1)) for _ in range(nin)]
        self.b = Value(0)
        self.nonlin = nonlin
    def __call__(self, x):
        act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b)
        return act.relu() if self.nonlin else act
    def parameters(self):
        return self.w + [self.b]
```

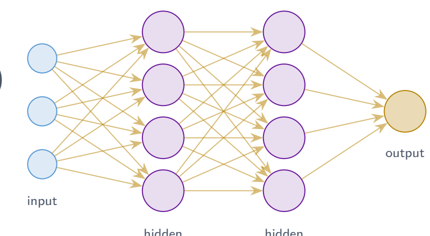
PyTorch-style API: `n(x)` runs the forward pass and *simultaneously builds the autograd graph*, exactly because every `w`, `b`, `x` is a `Value`.

Layer and Multi-Layer Perceptron (MLP)

Layer: a parallel array of independent neurons sharing the same input vector.

$$\text{Layer}_{n_{in}\to n_{out}}(x) = \left(\text{Neuron}_1(x), \ldots, \text{Neuron}_{n_{out}}(x)\right)$$

MLP: a sequence of layers $(n_{in} \to h_1 \to \ldots \to h_L)$. Forward = left-to-right composition.



`MLP(3, [4, 4, 1])`: 3-input, two hidden layers of width 4, one output.

Python

```
class Layer(Module):
    def __init__(self, nin, nout, **kw):
        self.neurons = [Neuron(nin, **kw) for _ in range(nout)]
    def __call__(self, x):
        out = [n(x) for n in self.neurons]
        return out[0] if len(out) == 1 else out
    def parameters(self):
        return [p for n in self.neurons for p in n.parameters()]
```

The MLP Wrapper

Python

```
class MLP(Module):
    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [
            Layer(sz[i], sz[i+1], nonlin=(i != len(nouts) - 1))
            for i in range(len(nouts))
        ]
    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
    def parameters(self):
        return [p for layer in self.layers for p in layer.parameters()]
```

- Last layer linear (`nonlin=False`) for regression-style outputs.
- `n.parameters()` flattens *every* `Value` that participates in training $\Rightarrow$ used by the optimiser.

Part 8

Training: Gradient Descent in Action

Loss Function

We need a scalar that measures *how wrong* our network is. For example the Mean-Squared Error (MSE) on a tiny dataset of four samples:

$$L(\theta) = \sum_{i=1}^{N} \left(\hat{y}_i - y_i\right)^2, \qquad \hat{y}_i = f_\theta(x_i).$$

- Each $\hat{y}_i$ is built from `Value` objects $\Rightarrow$ so is the sum L.
- Calling `L.backward()` fills `p.grad` for every parameter $p \in$ `n.parameters()`.
- The gradient points uphill: to decrease L, we step *against* it.

Gradient Descent — Geometric Intuition

Gradient Descent (Wikipedia)

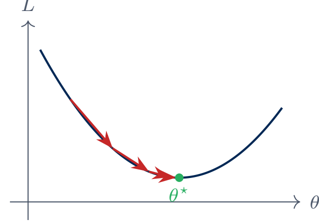
To minimise $L : \mathbb{R}^n \to \mathbb{R}$, repeatedly update

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta),$$

where $\eta > 0$ is the *learning rate* (step size).

Three things can go wrong:

- η too small $\Rightarrow$ painfully slow progress.
- η too large $\Rightarrow$ overshoot, diverge, loss explodes.
- Forget to `zero_grad()` between steps $\Rightarrow$ gradients accumulate over iterations and the optimiser walks in the wrong direction.

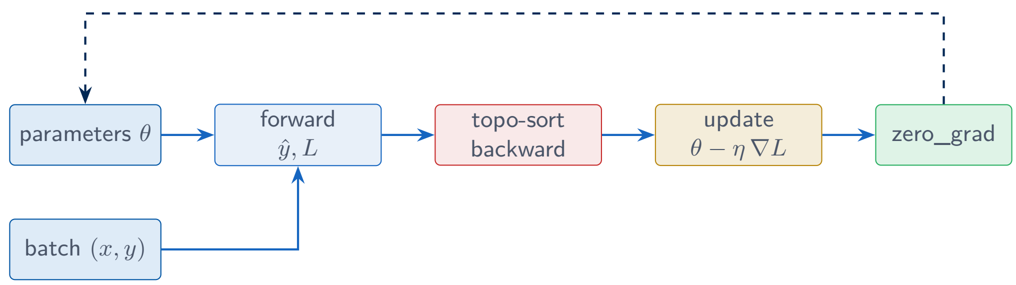


The Full Training Loop

Python

```
n = MLP(3, [4, 4, 1])                    # 3 inputs, 2 hidden layers, 1 output

for k in range(20):

    # 1) FORWARD PASS  <- compute predictions and loss
    ypred = [n(x) for x in xs]
    loss  = sum((yout - ygt)**2 for ygt, yout in zip(ys, ypred))

    # 2) ZERO GRADIENTS  <- reset gradients from previous step
    for p in n.parameters():
        p.grad = 0.0

    # 3) BACKWARD PASS  <- topologically sorted chain rule
    loss.backward()

    # 4) GRADIENT DESCENT   <- parameter update using gradients
    for p in n.parameters():
        p.data += -0.1 * p.grad

    print(k, loss.data)
```

Anatomy of One Training Step



- Every line of micrograd's training loop maps onto a step here.
- PyTorch's `optimizer.zero_grad()`, `loss.backward()`, `optimizer.step()` are exact analogues.
- Replacing scalars by tensors and Python loops by GPU kernels is the *only* difference between this and efficient deep learning frameworks.

Part 9

Wrap-Up & What's Next

Recap

We built, from scratch:

- A scalar `Value` with autograd metadata.
- Forward primitives `+`, `*`, `**`, `exp`, `tanh`, `ReLU` with their local derivatives.
- A `backward()` method that
  (i) topologically sorts the graph,
  (ii) propagates gradients in reverse via the chain rule.
- `Neuron`, `Layer`, `MLP` on top of `Value`.
- A complete forward / zero-grad / backward / update loop.

Key concepts to remember:

- Computational graph = DAG of operations.
- Backprop = chain rule + reverse topological order.
- Always `+=` when accumulating gradients.
- Always `zero_grad()` between steps.
- Tensors are an *efficiency* layer over the very same scalar machinery.

Key Idea

~ 150 **lines of Python** are enough to capture the core of modern deep-learning framework.

Looking Ahead — Next Week: NLP & LLMs

Next time we will use today's autograd intuitions inside a much larger story:

- Tokenisation: words $\rightarrow$ integers.
- Embeddings: integers $\rightarrow$ learned vectors (still trained by backprop !).
- Bigram / n-gram language models.
- A glimpse of the Transformer architecture and how GPT-style large language models are trained.

Key Idea

Every parameter of GPT-4 is updated by exactly the rule

$$\theta \leftarrow \theta - \eta \nabla_\theta L$$

we wrote today — just on *many* more parameters and *many* more samples, on *many* more GPUs.

References

- A. Karpathy. *The spelled-out intro to neural networks and backpropagation: building micrograd*.
  Video: youtube.com/watch?v=VMj-3S1tku0
  Code (MIT): github.com/karpathy/micrograd
- Course 6 of this semester — *Graphs: BFS, DFS, Topological Sort, SCC* (foundation for our `backward()` implementation).

End of Week 10

Thank you — questions?