Lab Session 10

Autograd: Building *micrograd* from Scratch

The Art of Computer Programming 2 — Week 10 — May 15, 2026

Objectives

By the end of this lab, you will have re-built, step by step,the core of a modern automatic-differentiation engine, then used it to train a small neural network on a real (toy) classification problem. More precisely, you should be able to:

- Estimate a derivative numerically and explain its limits.
- Draw the computational graph of an arithmetic expression and backpropagate gradients on it *by hand*.
- Implement the `Value` class with operator overloading and local chain-rule closures.
- Use a topological sort (Course 6) to automate the backward pass.
- Compose `Neuron`, `Layer` and `MLP` on top of `Value`.
- Run a complete forward / zero-grad / backward / update training loop.

Why this matters

This lab walks you through Andrej Karpathy's celebrated micrograd engine (∼150 lines of pure Python). Every line we will write has a direct counterpart in PyTorch / JAX / TensorFlow — the *only* differences are tensors instead of scalars, and CUDA kernels instead of Python loops.

Hint

Setup. Open a fresh notebook `lab10.ipynb`. The only imports you will need for the whole lab are:

```
import math, random
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

For the very last visualisation you can also `import torch` (optional).

1 Warm-up by Hand (15 min)

Pen and paper — no computer needed yet.

1.1 Numerical derivative

Math reminder

For a differentiable $f : \mathbb{R} \to \mathbb{R}$,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \approx \frac{f(x+h) - f(x)}{h} \quad \text{for small } h > 0.$$

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

(a) Compute the analytic derivative $f'(x)$.

(b) Evaluate $f'(2)$ exactly. What value do you expect from the numerical estimate $(f(2+h)-f(2))/h$ with $h = 10^{-3}$? With $h = 10^{-16}$?

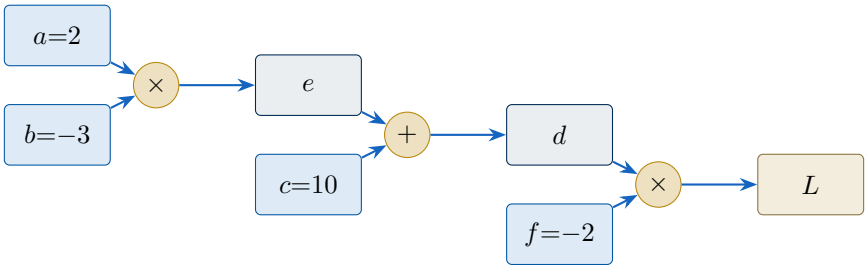
(c) In one sentence, why does taking h *too small* also fail?

1.2 Manual backpropagation on a tiny graph

Consider the expression

$$L = (a \cdot b + c) \cdot f, \qquad a = 2,\ b = -3,\ c = 10,\ f = -2.$$

Naming the intermediate nodes $e = a \cdot b$ and $d = e + c$, the computational graph looks like this:



Recall

Chain rule. If $z = g(y)$ and $y = h(x)$, then $\dfrac{dz}{dx} = \dfrac{dz}{dy} \cdot \dfrac{dy}{dx}$.

Local derivatives: $\partial(u+v)/\partial u = 1, \quad \partial(u \cdot v)/\partial u = v$.

(a) Forward pass: fill in e, d and L.

(b) Seed $\partial L/\partial L = 1$ and walk the graph *backwards*, completing the table:

Node v	Local rule (in symbols)	$\partial L/\partial v$
L	seed	1
d	$L = d \cdot f \ \Rightarrow\ \partial L/\partial d = f$	-2
f	...	...
c	$d = e + c \ \Rightarrow\ \partial L/\partial c = \partial L/\partial d$	...
e	...	...
a	...	...
b	...	...

(c) **Sanity check.** Bump b by $h = 10^{-3}$ in your head: by how much should L change? Compare with your $\partial L/\partial b$.

2 The `Value` Class — Forward Only (15 min)

We now mirror the graph above in Python. Each scalar in the computation will be wrapped in a `Value` object that remembers *how it was produced*.

Recall

A `Value` stores four pieces of information:

- `data` — the numerical value (a float).
- `grad` — $\partial L/\partial$self, initially 0.

- `_prev` — the set of parent `Values` in the graph.
- `_op` — a label (`'+'`, `'*'`, …) of the operation.

```python
class Value:

    def __init__(self, data, _children=(), _op='', label=''):
        self.data  = data
        self.grad  = 0.0
        self._prev = set(_children)
        self._op   = _op
        self.label = label

    def __repr__(self):
        return f"Value(data={self.data}, grad={self.grad})"

    def __add__(self, other):
        # YOUR CODE HERE
        # return a new Value whose data is self.data + other.data,
        # whose _children are (self, other), and whose _op is '+'.
        pass

    def __mul__(self, other):
        # YOUR CODE HERE  -- analogous to __add__
        pass
```

(a) Implement `__add__` and `__mul__`.

(b) Reproduce the warm-up graph in code and check the forward value:

```python
a = Value(2.0,  label='a')
b = Value(-3.0, label='b')
c = Value(10.0, label='c')
f = Value(-2.0, label='f')
e = a*b; e.label = 'e'
d = e + c; d.label = 'd'
L = d * f; L.label = 'L'
print(L)         # expected: data = -8.0
print(L._prev)   # expected: {d, f}
```

2.1 Visualising the graph (provided)

The function below is given — just copy it and run it on `L`. It uses graphviz (`pip install graphviz`) to render the DAG.

```python
from graphviz import Digraph

def trace(root):
    nodes, edges = set(), set()
    def build(v):
        if v not in nodes:
            nodes.add(v)
            for child in v._prev:
                edges.add((child, v))
                build(child)
    build(root)
    return nodes, edges
```

```
def draw_dot(root):
    dot = Digraph(format='svg', graph_attr={'rankdir': 'LR'})
    nodes, edges = trace(root)
    for n in nodes:
        uid = str(id(n))
        dot.node(name=uid,
                 label="{ %s | data %.4f | grad %.4f }"
                       % (n.label, n.data, n.grad),
                 shape='record')
        if n._op:
            dot.node(name=uid + n._op, label=n._op)
            dot.edge(uid + n._op, uid)
    for n1, n2 in edges:
        dot.edge(str(id(n1)), str(id(n2)) + n2._op)
    return dot

draw_dot(L)
```

Hint

You should see your warm-up DAG: leaves `a, b, c, f`, intermediates `e, d`, root `L`. All `grad` fields are still 0.0 — we will fill them in next.

3 Manual Backpropagation in Code ← (10 min)

Before automating anything, let's translate your warm-up table into Python. This makes the chain rule physically concrete.

```
# Seed
L.grad = 1.0

# L = d * f
d.grad = f.data * L.grad     # dL/dd = f
f.grad = d.data * L.grad     # dL/df = d

# d = e + c
e.grad = 1.0 * d.grad        # dL/de = dL/dd * 1
c.grad = 1.0 * d.grad        # dL/dc = dL/dd * 1

# e = a * b
a.grad = b.data * e.grad     # dL/da = dL/de * b
b.grad = a.data * e.grad     # dL/db = dL/de * a

draw_dot(L)
```

(a) Run the cell. Do the gradients in the picture match your hand computation from §1.2?

(b) **Empirical check.** Use finite differences to verify $\partial L/\partial b$:

```
def L_of(a_, b_, c_, f_):
    return (a_*b_ + c_) * f_

h = 1e-4
print("numeric dL/db =", (L_of(2.0, -3.0+h, 10.0, -2.0) - L_of(2.0,-3.0,10.0,-2.0)) / h)
print("analytic db   =", b.grad)
```

(c) **Tiny optimisation.** Suppose we want to *increase* L. Update each parameter by a small step *along* its gradient: `a.data += 0.01 * a.grad`, etc. Recompute the forward pass. Did L grow?

4 Automating the Backward Pass (25 min)

Doing it by hand was instructive but does not scale. We now teach each operation to know its own *local* gradient rule, and let a topological sort handle the global ordering.

4.1 Local closures attached to each op

Recall

Each operation creates an `out` node and attaches a tiny function `out._backward` that reads `out.grad` and **accumulates** into its parents (`self.grad`, `other.grad`). Local rules:

$$+: \partial(u+v)/\partial u = 1 \qquad \cdot: \partial(u \cdot v)/\partial u = v$$

```python
class Value:

    def __init__(self, data, _children=(), _op='', label=''):
        self.data  = data
        self.grad  = 0.0
        self._backward = lambda: None       # NEW: local backward closure
        self._prev = set(_children)
        self._op   = _op
        self.label = label

    def __repr__(self):
        return f"Value(data={self.data})"

    def __add__(self, other):
        out = Value(self.data + other.data, (self, other), '+')
        def _backward():
            # YOUR CODE HERE  (use += !)
            pass
        out._backward = _backward
        return out

    def __mul__(self, other):
        out = Value(self.data * other.data, (self, other), '*')
        def _backward():
            # YOUR CODE HERE
            pass
        out._backward = _backward
        return out
```

Pitfall

Always use `+=`, never `=`. A node can be re-used inside the same expression (e.g. `b = a + a`). The multivariable chain rule says contributions from each downstream path *sum* — overwriting would silently drop them.

4.2 The `backward` method (topological sort)

Recall

Course 6 reminder. A topological order of a DAG $G = (V, E)$ is a linear order such that every edge $u \to v$ has u before v. A standard DFS post-order produces a *reverse* topological order, which is exactly what we need: when we pop a node, all its children (downstream) have already contributed their gradient to it.

Add this method *inside* the `Value` class:

```
def backward(self):
    # 1. Topological sort of all nodes reachable from self
    topo = []
    visited = set()
    def build_topo(v):
        # YOUR CODE HERE  --  DFS that appends v to topo in post-order
        pass
    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()
```

(a) Implement `build_topo`. It should look almost exactly like the DFS we wrote for graph topo-sort in Course 6.

(b) Re-build the warm-up graph and call `L.backward()`. Compare the `grad` fields with your manual values from §3.

(c) **Common bug.** Comment out the `self.grad = 1.0` seed and re-run. What happens, and why?

Hint

The order matters. Because we pop nodes in *reverse* topological order, when `node._backward()` runs, `node.grad` has already been accumulated by every child below it.

5 Extending the Operator Set (20 min)

To express even a single neuron 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()`	$\tanh u$	$1 - \tanh^2 u$

5.1 Coercion and reflected operators (provided)

So that we can write `a + 1` or `2 * a` (where `1, 2` are plain Python numbers), every operator coerces its argument and we add the reflected versions. **Copy these into your `Value` class verbatim.**

```
def __add__(self, other):
    other = other if isinstance(other, Value) else Value(other)
    out = Value(self.data + other.data, (self, other), '+')
    def _backward():
        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():
        self.grad  += other.data * out.grad
        other.grad += self.data  * out.grad
    out._backward = _backward
    return out

def __pow__(self, other):
    assert isinstance(other, (int, float)), "only int/float powers"
    out = Value(self.data ** other, (self,), f'**{other}')
    def _backward():
        self.grad += other * (self.data ** (other - 1)) * out.grad
    out._backward = _backward
    return out

def __radd__(self, other): return self + other
def __rmul__(self, other): return self * other
def __neg__(self):         return self * -1
def __sub__(self, other):  return self + (-other)
def __truediv__(self, o):  return self * o**-1
```

5.2 Your turn: `tanh` and `exp`

X¹ Math reminder

$$\tanh(u) = \frac{e^{2u}-1}{e^{2u}+1}, \qquad \frac{d}{du}\tanh(u) = 1-\tanh^2(u), \qquad \frac{d}{du}\,e^u = e^u.$$

```
def tanh(self):
    x = self.data
    t = (math.exp(2*x) - 1) / (math.exp(2*x) + 1)
    out = Value(t, (self,), 'tanh')
    def _backward():
        # YOUR CODE HERE  -- use the formula above (and the trick that t == out.data)
        pass
    out._backward = _backward
    return out

def exp(self):
    out = Value(math.exp(self.data), (self,), 'exp')
    def _backward():
        # YOUR CODE HERE
        pass
    out._backward = _backward
    return out
```

5.3 A single neuron, end to end

We now build the artificial neuron from the slides: $o = \tanh(x_1 w_1 + x_2 w_2 + b)$.

```
# Inputs
x1 = Value(2.0, label='x1')
x2 = Value(0.0, label='x2')
# Weights
w1 = Value(-3.0, label='w1')
w2 = Value( 1.0, label='w2')
# Bias chosen so the output is a "nice" number
b  = Value(6.8813735870195432, label='b')

# Forward
n = x1*w1 + x2*w2 + b ; n.label = 'n'
o = n.tanh()          ; o.label = 'o'
print(o.data)

# Backward
o.backward()

draw_dot(o)
```

(a) What value does `o.data` take?

(b) Read off `x1.grad, w1.grad, x2.grad, w2.grad`. Which inputs matter most for the neuron's output here, and why?

(c) **Cross-check with PyTorch** (provided — just run it):

```
import torch
x1 = torch.tensor([2.0],  dtype=torch.double, requires_grad=True)
x2 = torch.tensor([0.0],  dtype=torch.double, requires_grad=True)
w1 = torch.tensor([-3.0], dtype=torch.double, requires_grad=True)
w2 = torch.tensor([ 1.0], dtype=torch.double, requires_grad=True)
b  = torch.tensor([6.8813735870195432], dtype=torch.double, requires_grad=True)
o = torch.tanh(x1*w1 + x2*w2 + b)
o.backward()
print(x1.grad.item(), w1.grad.item(), x2.grad.item(), w2.grad.item())
```

Your micrograd numbers should agree to machine precision.

6 From a Scalar Engine to a Neural Network 🕸 (25 min)

Why this matters

A neuron is just an expression in our `Values`. A layer is a list of neurons that share inputs. An MLP is a list of layers. *Every* parameter (each weight and each bias) is a `Value`, so backprop just works.

6.1 The `Neuron` class (provided)

Read this carefully — you'll mirror the same pattern for layers and the MLP.

```
class Neuron:
```

```
    def __init__(self, nin):
        self.w = [Value(random.uniform(-1, 1)) for _ in range(nin)]
        self.b = Value(random.uniform(-1, 1))

    def __call__(self, x):
        # x is a list of nin numbers (or Values).
        # act = w . x + b
        act = sum((wi*xi for wi, xi in zip(self.w, x)), self.b)
        return act.tanh()

    def parameters(self):
        return self.w + [self.b]
```

```
random.seed(1337)
n = Neuron(3)
print(n([1.0, 2.0, 3.0]))     # a Value, somewhere in (-1, 1)
print(len(n.parameters()))    # 4 = 3 weights + 1 bias
```

6.2 Your turn: `Layer` and `MLP`

Recall

Layer: given `nin` inputs and `nout` neurons, the layer returns a list of `nout` outputs (or a single `Value` when `nout == 1`).

MLP: given an input size `nin` and a list of layer widths `nouts = [h1, h2, ..., hL]`, build L layers with sizes $n_{in} \to h_1$, $h_1 \to h_2$, …, $h_{L-1} \to h_L$. The forward pass chains them.

```
class Layer:

    def __init__(self, nin, nout):
        # YOUR CODE HERE  -- create nout neurons of input size nin
        pass

    def __call__(self, x):
        # YOUR CODE HERE  -- run each neuron on x; unwrap if a single output
        pass

    def parameters(self):
        # YOUR CODE HERE  -- flatten parameters from every neuron
        pass


class MLP:

    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        # YOUR CODE HERE  -- create len(nouts) Layers chaining sz[i] -> sz[i+1]
        pass

    def __call__(self, x):
        # YOUR CODE HERE  -- forward through every layer in sequence
        pass

    def parameters(self):
        # YOUR CODE HERE
        pass
```

```
random.seed(1337)
n = MLP(3, [4, 4, 1])
x = [1.0, 2.0, 3.0]
y = n(x)
print(y)                          # one Value, output of the network
print(len(n.parameters()))        # 41 with this architecture
draw_dot(y)                       # have a look at the graph!
```

Hint

The full graph for `MLP(3, [4, 4, 1])` on a single input has dozens of nodes already. *This is exactly the graph backprop will traverse.*

7 Application: Train an MLP on a Toy Dataset (30 min)

Why this matters

Time to actually *learn*. We give the network four labelled examples in $\mathbb{R}^3$ and target outputs in $\{-1, +1\}$. It must adjust its 41 parameters so that the predictions match the targets.

7.1 The dataset and the loss

```
xs = [
    [ 2.0,  3.0, -1.0],
    [ 3.0, -1.0,  0.5],
    [ 0.5,  1.0,  1.0],
    [ 1.0,  1.0, -1.0],
]
ys = [1.0, -1.0, -1.0, 1.0]    # target labels
```

Math reminder

Mean-squared error (sum form, since N is fixed):

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

L is small when the predictions are close to the targets, exactly 0 when they match perfectly.

(a) Re-instantiate the network and inspect its raw predictions:

```
random.seed(1337)
n = MLP(3, [4, 4, 1])
ypred = [n(x) for x in xs]
ypred                                  # list of four Value objects
```

(b) Build the loss as a `Value`:

```
loss = sum((yout - ygt)**2 for ygt, yout in zip(ys, ypred))
print(loss)
```

(c) Call `loss.backward()`. Pick any parameter `p` and read `p.grad`. Sanity-check its sign against what would lower the loss.

7.2 The complete training loop

Recall

Gradient descent. To *minimise* L, step against the gradient:

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

where $\eta > 0$ is the learning rate. The four mandatory steps per iteration:

1. **Forward:** compute `ypred` and `loss`.
2. **Zero grads:** reset `p.grad = 0.0` for every parameter.
3. **Backward:** `loss.backward()`.
4. **Update:** `p.data += -lr * p.grad`.

```
random.seed(1337)
n = MLP(3, [4, 4, 1])

losses = []
for k in range(50):

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

    # 2) ZERO GRADIENTS
    # YOUR CODE HERE  -- one line, loop over n.parameters()

    # 3) BACKWARD
    # YOUR CODE HERE  -- one line

    # 4) UPDATE
    # YOUR CODE HERE  -- loop over n.parameters() with lr = 0.05

    losses.append(loss.data)
    if k % 5 == 0:
        print(f"step {k:3d}   loss = {loss.data:.6f}")

print("\nFinal predictions:", [round(yp.data, 3) for yp in ypred])
print("Targets          :", ys)
```

(a) Fill in the three missing lines and run the loop. Does the loss go down? How close are the final predictions to ± 1?

(b) Plot the loss curve:

```
plt.plot(losses); plt.xlabel("step"); plt.ylabel("loss")
plt.yscale("log"); plt.grid(True); plt.title("Training loss");
```

(c) **Forget the zero-grad.** Comment out step 2 and re-run from a fresh `n`. What happens to the loss after a few iterations? Why?

Pitfall

Two of the most common deep-learning bugs you can already encounter on this 4-sample toy:

- Forgetting `zero_grad()` ⇒ gradients accumulate across steps and the optimiser walks in the wrong direction.

- Learning rate too large ⇒ the loss explodes (try `lr = 10`!).

7.3 Tweaking the experiment

(a) Try `lr` values $\{0.01, 0.05, 0.5, 2.0, 10.0\}$. Plot the loss curves on the same figure. Identify a regime that is too slow, one that is healthy, and one that diverges.

(b) Change the architecture to `MLP(3, [8, 1])`. Does it train faster or slower? Same final loss?

(c) Add a fifth sample $(x, y) = ([-1, 0, 1], +1)$ and retrain. Anything change?

Bonus — The Same Loop in PyTorch

Bonus

PyTorch's autograd is a tensorised, GPU-accelerated version of what you just wrote. The training loop is structurally identical:

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

xs_t = torch.tensor(xs)
ys_t = torch.tensor(ys)

class MLP_torch(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = nn.Linear(3, 4)
        self.l2 = nn.Linear(4, 4)
        self.l3 = nn.Linear(4, 1)
    def forward(self, x):
        x = torch.tanh(self.l1(x))
        x = torch.tanh(self.l2(x))
        return self.l3(x)

torch.manual_seed(1337)
net = MLP_torch()
for k in range(200):
    ypred = net(xs_t).squeeze()
    loss  = F.mse_loss(ypred, ys_t)
    net.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in net.parameters():
            p -= 0.05 * p.grad
    if k % 20 == 0:
        print(k, loss.item())
```

Map every line back to the micrograd version you wrote in §7. The four steps are identical: forward, zero-grad, backward, update. The *only* substantive difference is the use of tensors and the built-in `nn.Linear` / `F.mse_loss` primitives.