PSL UNIVERSITÉ PARIS

Lab Session 10 — Solutions

Autograd: Building *micrograd* from Scratch

TAOCP2 — Week 10 — **Solutions**

1 Warm-up by Hand

1.1 Numerical derivative

(a) $f'(x) = 6x - 4$.
(b) $f'(2) = 8$. With $h = 10^{-3}$ the forward-difference estimate is

$$\frac{f(2.001) - f(2)}{10^{-3}} = \frac{(3 \cdot 4.004001 - 4 \cdot 2.001 + 5) - (12 - 8 + 5)}{10^{-3}} \approx 8.003,$$

i.e. the true value plus a $\mathcal{O}(h)$ truncation error.
With $h = 10^{-16}$ the truncation error is tiny but $f(x+h)$ and $f(x)$ agree on most of their leading digits, so the subtraction cancels almost all useful information. With `float64` (~ 15–17 significant digits), the result will be visibly noisy — typically 5–9 instead of 8.
(c) Floating-point cancellation: $f(x+h)$ and $f(x)$ agree to many digits, and the subtraction throws away precision. The relative error is roughly $\mathcal{O}(\varepsilon_{\text{mach}}/h) + \mathcal{O}(h)$, minimised around $h \sim \sqrt{\varepsilon_{\text{mach}}} \approx 10^{-8}$ for `float64`.

1.2 Manual backpropagation on a tiny graph

(a) Forward pass. $e = a \cdot b = -6, \quad d = e + c = 4, \quad L = d \cdot f = -8.$
(b) Backward pass.

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

(c) Sanity check. Bumping b by h should change L by $\approx h \cdot \partial L/\partial b = -4h$. Concretely with $h = 10^{-3}$: new $b = -2.999$, $e = -5.998$, $d = 4.002$, $L = -8.004$, so $\Delta L \approx -0.004 = -4h$. ✓

Common Mistakes

- Confusing $\partial L/\partial e$ (chain through d) with the local derivative $\partial d/\partial e = 1$. The chain rule *multiplies* them.

2 The `Value` Class — Forward Only

```python
def __add__(self, other):
    return Value(self.data + other.data, _children=(self, other), _op='+')

def __mul__(self, other):
    return Value(self.data * other.data, _children=(self, other), _op='*')
```

Running the warm-up snippet then prints `Value(data=-8.0, grad=0.0)` for `L`, and `L._prev == {d, f}`.

Explanation

`_prev` is a `set`, so duplicates are deduplicated. This is fine for the bookkeeping of which nodes belong to the graph, but be careful: the *closure* of `+` or `*` still adds two contributions even if `self is other`, because we use `+=` on `self.grad` and `other.grad` separately — they just happen to point at the same Python object.

3 Manual Backpropagation in Code

(a) The picture shows $L.\text{grad} = 1$, $d.\text{grad} = -2$, $f.\text{grad} = 4$, $c.\text{grad} = -2$, $e.\text{grad} = -2$, $a.\text{grad} = 6$, $b.\text{grad} = -4$ — exactly the table from §1.2.
(b) Numeric estimate of $\partial L/\partial b$:

```
numeric dL/db = -4.000000000026205
analytic db   = -4.0
```

The agreement is up to $\sim 10^{-11}$, dominated by the $\mathcal{O}(h)$ truncation error of the one-sided difference plus floating-point round-off.
(c) After the gradient *ascent* step `a.data += 0.01*a.grad` and similarly for b, c, f, the new values are roughly $a = 2.06$, $b = -3.04$, $c = 9.98$, $f = -1.96$. Recomputing the forward pass gives $L \approx -7.286$, which is greater than the previous -8.0 ("less negative" = larger). Walking with the gradient increases L, as expected from the geometric meaning of the gradient.

Common Mistakes

A surprisingly common mistake is to update parameters and forget to recompute the forward graph: students then read the *old* value of `L` and conclude nothing happened. The forward expressions `e = a*b ; d = e + c ; L = d*f` must be re-executed.

4 Automating the Backward Pass

4.1 Local closures

```python
def __add__(self, 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):
    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
```

4.2 The `backward` method

(a) The `build_topo` closure is a textbook DFS post-order:

```python
def backward(self):
    topo, visited = [], set()
    def build_topo(v):
        if v not in visited:
            visited.add(v)
            for child in v._prev:
                build_topo(child)
            topo.append(v)        # post-order
    build_topo(self)

    self.grad = 1.0
    for node in reversed(topo):
        node._backward()
```

(b) Calling `L.backward()` on a freshly built warm-up graph yields *exactly* the same gradients as the manual table: $a.grad = 6$, $b.grad = -4$, $c.grad = -2$, $f.grad = 4$.
(c) Without the seed `self.grad = 1.0`, every node has `grad = 0.0` when its closure runs, so all parents receive `0 * (...)` and stay at zero. The whole backward pass is silently a no-op. The seed encodes $\partial L/\partial L = 1$, which is the "identity" boundary condition of the chain rule.

Explanation

Note that `build_topo` traverses `v._prev` (*parents*, i.e. upstream nodes), not children. Because we then iterate in `reversed(topo)`, downstream nodes are popped first and have their gradients ready by the time their parents are processed.

Common Mistakes

- Building the topo list *before* appending `v` (pre-order) rather than after (post-order). With pre-order plus `reversed`, parents would be popped before children, breaking the chain.
- Calling `backward()` a second time on the same graph without zeroing gradients: contributions accumulate, giving $2\times$, $3\times$, … the correct values.

5 Extending the Operator Set

5.1 `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

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

5.2 The single neuron

(a) With $b = 6.8813735870195432$ chosen so that $x_1w_1 + x_2w_2 + b = -6 + 0 + 6.881\ldots \approx 0.881\ldots$, we get `o.data` $\approx$ `0.7071`. The bias was hand-picked so that `tanh(…)` equals exactly $1/\sqrt{2}$ — it makes the gradients themselves come out as round numbers, which is convenient for the cross-check.
(b) `o.backward()` produces:

```
x1.grad = -1.5    w1.grad =  1.0
x2.grad =  0.5    w2.grad =  0.0
```

The gradients are scaled by $1 - \tanh^2(n) = 1 - 0.5 = 0.5$. Inputs with `x = 0` (here x_2) contribute zero gradient on the corresponding weight: the network cannot learn anything about w_2 from a sample where x_2 is silent. Conversely, x_1 has the largest absolute gradient because it is multiplied by the relatively large weight $w_1 = -3$.
(c) PyTorch returns the same numbers to machine precision $(-1.5,\ 1.0,\ 0.5,\ 0.0)$. ✓

Explanation

The factor $1 - \tanh^2(n)$ is the *vanishing-gradient* story in miniature: when $|n|$ is large, tanh saturates and $1 - \tanh^2 \to 0$, so the gradient that flows back through the neuron is killed. This is one of the historical reasons people switched from `tanh` to `ReLU` for deep networks.

6 From a Scalar Engine to a Neural Network

```python
class Layer:

    def __init__(self, nin, nout):
        self.neurons = [Neuron(nin) for _ in range(nout)]

    def __call__(self, x):
        outs = [n(x) for n in self.neurons]
        return outs[0] if len(outs) == 1 else outs

    def parameters(self):
        return [p for n in self.neurons for p in n.parameters()]


class MLP:

    def __init__(self, nin, nouts):
        sz = [nin] + nouts
        self.layers = [Layer(sz[i], sz[i+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()]
```

For `MLP(3, [4, 4, 1])` the parameter count is

$$\underbrace{4 \cdot (3+1)}_{16} + \underbrace{4 \cdot (4+1)}_{20} + \underbrace{1 \cdot (4+1)}_{5} = 41.$$

`draw_dot(n([1.0, 2.0, 3.0]))` produces a graph with several hundred nodes already (every multiply, every sum, every `tanh`).

7 Application — Train an MLP on a Toy Dataset

7.1 Forward, loss, single backward

With `random.seed(1337)` and the architecture `MLP(3, [4, 4, 1])`, the raw predictions are roughly

$$\hat{y} \approx [0.50,\ 0.74,\ 0.20,\ 0.61]$$

(values vary slightly with the seed) and the initial loss is $L \approx 4.5$ — predictably awful, the targets being $[\pm 1]$.
After `loss.backward()`, every parameter has a non-zero gradient. For example the bias of the output neuron typically has a gradient of ~ 1–3 in absolute value; its sign tells us which way to push the output to lower the loss.

7.2 The complete training loop

The three missing lines are:

```python
# 2) ZERO GRADIENTS
for p in n.parameters(): p.grad = 0.0

# 3) BACKWARD
loss.backward()

# 4) UPDATE
for p in n.parameters(): p.data += -0.05 * p.grad
```

(a) With `random.seed(1337)`, `lr = 0.05` and 50 steps, the loss drops from ∼4.5 down to ∼0.02. Final predictions land very close to the targets:

```
Final predictions: [0.952, -0.928, -0.948, 0.939]
Targets          : [1.0,   -1.0,   -1.0,   1.0]
```

The network has learnt to interpolate the four labelled points perfectly (it has 41 parameters for 4 samples — this is heavy over-parametrisation, which is what makes the optimisation easy here).
(b) The log-scale loss curve is roughly linear over the first ∼30 steps (exponential convergence), then flattens out as the network approaches the fixed points ±1 of tanh where its own gradient vanishes.
(c) *Forgetting `zero_grad`.* Gradients accumulate across steps. After step k the effective gradient seen by the optimiser is roughly $k\,\nabla L$; with a fixed learning rate this is equivalent to a linearly growing step size. The loss either oscillates wildly or diverges within a handful of iterations.

7.3 Tweaking the experiment

(a) Typical observations on this toy problem:

`lr`	Behaviour
0.01	Slow but steady descent; loss still ∼1 at step 50.
0.05	Healthy convergence to ∼0.02.
0.5	Fast at first, but oscillates around a non-zero plateau.
2.0	Loss explodes within ∼10 steps.
10.0	Diverges immediately to `NaN` (overflow inside `tanh`).

(b) `MLP(3, [8, 1])` has $8 \cdot 4 + 1 \cdot 9 = 41$ parameters too, but a shallower computation graph, so each step is slightly cheaper. Convergence is comparable on a problem this trivial.
(c) Adding the fifth sample ($[-1, 0, 1]$, $+1$) adds one term to the loss. Convergence is essentially unaffected (still over-parametrised: $41 \gg 5$); the network simply interpolates one more point.

Explanation

The fact that we can perfectly fit any small dataset with an MLP that has "too many" parameters is called *interpolation*. Real machine learning is concerned with the harder question of *generalisation*: how does the network behave on points it has *not* seen? That requires held-out data, regularisation, and a much bigger course — but the engine that powers all of it is exactly the autograd we built today.

Common Mistakes

- Computing `loss = sum(...)` over a list of `Values` and forgetting that `sum` starts from the integer 0. It works thanks to our `__radd__`, but if a student omitted the reflected operator they will see a `TypeError`.
- Updating `p.data` *with the wrong sign*: writing `p.data += lr * p.grad` performs gradient *ascent* and the loss grows. Easy to spot from the printout.

- Re-using the same `n` across runs without reseeding — students then see different numbers and panic. `random.seed(1337)` before each `MLP(...)` call is the safe pattern.

Bonus — The Same Loop in PyTorch

The PyTorch snippet given in the handout converges to a loss of $\sim 10^{-3}$ within $\sim$200 steps (PyTorch's `nn.Linear` is initialised differently, so the constants change but the qualitative picture is identical). The mapping is one-to-one:

micrograd	PyTorch
`Value`	`torch.Tensor(requires_grad=True)`
`Neuron / Layer / MLP`	`nn.Module` subclasses
`ypred = [n(x) for x in xs]`	`ypred = net(xs_t)` (vectorised)
`(y - yhat)**2 summed`	`F.mse_loss(ypred, ys_t) * N`
`for p in n.parameters(): p.grad=0`	`net.zero_grad()`
`loss.backward()`	`loss.backward()` *(literally)*
`p.data += -lr * p.grad`	`p -= lr * p.grad` (under `torch.no_grad`)