Continuation of Task 2: Backpropagation with Three Parameters

Group 2 – Ferschl Martin, Reiter Roman, Zenkic Mirza

December 12, 2025

1 Task specification

We generalize the setting of Task 2 to a function

$$f = f(x; a, b, c),$$

i.e. a function of an input variable x and three parameters a, b, c. The first part asks us to explain how the chain rule approach from Task 2 extends to this case.

In a concrete example we choose

$$f(x; a, b, c) = ax^2 + b + c^2$$

and

$$F(x) = f\big(f(f(x + a) + b) + c\big),$$

where the same parameters a, b, c are used in each occurrence of f. The partial derivatives of F with respect to a, b, c can be evaluated either

- directly (by symbolic differentiation), or
- via backpropagation, using the chain rule layer by layer.

Both approaches must give the same result. We implement and test this numerically for specific values of x, a, b, c.

2 Generalization of the chain rule to three parameters

In Task 2 we had a function $f(x; a, b)$ and a composite

$$F(x) = f\big(f(x + a; a, b) + b\,;\ a, b\big),$$

and we computed F_a and F_b via the chain rule. The key ideas were:

- represent the computation as a sequence of simple steps (a *computational graph*),
- apply the chain rule at each step,
- reuse intermediate derivatives (sensitivities) when computing gradients with respect to different parameters — this is the idea of *backpropagation*.

If we extend f to three parameters,

$$f = f(x; a, b, c),$$

then we simply get one more partial derivative,

$$f_x(x; a, b, c) = \frac{\partial f}{\partial x}, \quad f_a(x; a, b, c) = \frac{\partial f}{\partial a}, \quad f_b(x; a, b, c) = \frac{\partial f}{\partial b}, \quad f_c(x; a, b, c) = \frac{\partial f}{\partial c}.$$

In the computational graph, f may appear several times (multiple layers), and each occurrence contributes to the total derivatives F_a, F_b, F_c. Backpropagation works exactly as before:

1. Perform a *forward pass*: compute all intermediate values in the graph (the "layers").
2. Perform a *backward pass*: starting from the output, propagate derivatives $\frac{\partial F}{\partial(\text{node})}$ backwards using the chain rule. At each occurrence of f we add contributions involving f_a, f_b, f_c, and at each simple operation such as additions we apply the corresponding derivative rule.

The only difference to the two-parameter case is that we now track three parameter gradients F_a, F_b, F_c instead of two.

3 Concrete example: $f(x; a, b, c) = ax^2 + b + c^2$

3.1 Definition of f and its partial derivatives

We consider

$$f(x; a, b, c) = ax^2 + b + c^2.$$

Its partial derivatives are straightforward:

$$\begin{aligned}
f_x(x; a, b, c) &= \frac{\partial}{\partial x}(ax^2 + b + c^2) = 2ax, \\
f_a(x; a, b, c) &= \frac{\partial}{\partial a}(ax^2 + b + c^2) = x^2, \\
f_b(x; a, b, c) &= \frac{\partial}{\partial b}(ax^2 + b + c^2) = 1, \\
f_c(x; a, b, c) &= \frac{\partial}{\partial c}(ax^2 + b + c^2) = 2c.
\end{aligned}$$

We define the composite function

$$F(x) = f\big(f(f(x + a) + b) + c\big),$$

where each $f(\cdot)$ uses the same parameters a, b, c. For clarity, we write the computation as a sequence of intermediate variables:

$$\begin{aligned}
z_0 &= x + a, \\
z_1 &= f(z_0; a, b, c), \\
z_2 &= z_1 + b, \\
z_3 &= f(z_2; a, b, c), \\
z_4 &= z_3 + c, \\
F(x) &= f(z_4; a, b, c).
\end{aligned}$$

3.2 Backpropagation: computing F_a, F_b, F_c

We now compute the gradients F_a, F_b, F_c via backpropagation. We keep track of:

- the "upstream" derivatives $g_k = \frac{\partial F}{\partial z_k}$,
- the parameter derivatives F_a, F_b, F_c accumulated along the way.

We use the shorthand

$$f_x(u) := f_x(u; a, b, c) = 2au, \quad f_a(u) := f_a(u; a, b, c) = u^2, \quad f_b(u) := 1, \quad f_c(u) := 2c.$$

Step 1: Node $F = f(z_4; a, b, c)$. We start with

$$\frac{\partial F}{\partial F} = 1.$$

For the final application of f,

$$F = f(z_4; a, b, c),$$

the chain rule gives

$$g_4 := \frac{\partial F}{\partial z_4} = f_x(z_4),$$

and contributions to the parameter derivatives

$$F_a \mathrel{+}= f_a(z_4), \quad F_b \mathrel{+}= f_b(z_4), \quad F_c \mathrel{+}= f_c(z_4).$$

Step 2: Node $z_4 = z_3 + c$. Here,

$$\frac{\partial z_4}{\partial z_3} = 1, \qquad \frac{\partial z_4}{\partial c} = 1.$$

Thus

$$g_3 := \frac{\partial F}{\partial z_3} = g_4 \cdot 1 = g_4,$$

and

$$F_c \mathrel{+}= g_4 \cdot 1.$$

Step 3: Node $z_3 = f(z_2; a, b, c)$. For this application of f,

$$\frac{\partial z_3}{\partial z_2} = f_x(z_2), \quad \frac{\partial z_3}{\partial a} = f_a(z_2), \quad \frac{\partial z_3}{\partial b} = f_b(z_2), \quad \frac{\partial z_3}{\partial c} = f_c(z_2).$$

So

$$g_2 := \frac{\partial F}{\partial z_2} = g_3\, f_x(z_2),$$

and

$$F_a \mathrel{+}= g_3 f_a(z_2), \quad F_b \mathrel{+}= g_3 f_b(z_2), \quad F_c \mathrel{+}= g_3 f_c(z_2).$$

Step 4: Node $z_2 = z_1 + b$. Here,

$$\frac{\partial z_2}{\partial z_1} = 1, \qquad \frac{\partial z_2}{\partial b} = 1.$$

Thus

$$g_1 := \frac{\partial F}{\partial z_1} = g_2 \cdot 1 = g_2,$$

and

$$F_b \mathrel{+}= g_2 \cdot 1.$$

Step 5: Node $z_1 = f(z_0; a, b, c)$. Again applying the derivatives of f,

$$\frac{\partial z_1}{\partial z_0} = f_x(z_0), \quad \frac{\partial z_1}{\partial a} = f_a(z_0), \quad \frac{\partial z_1}{\partial b} = f_b(z_0), \quad \frac{\partial z_1}{\partial c} = f_c(z_0).$$

Hence

$$g_0 := \frac{\partial F}{\partial z_0} = g_1 f_x(z_0),$$

and

$$F_a \mathrel{+}= g_1 f_a(z_0), \quad F_b \mathrel{+}= g_1 f_b(z_0), \quad F_c \mathrel{+}= g_1 f_c(z_0).$$

Step 6: Node $z_0 = x + a$. Finally,

$$\frac{\partial z_0}{\partial a} = 1,$$

so

$$F_a \mathrel{+}= g_0 \cdot 1.$$

At the end of this backward sweep, the accumulated values F_a, F_b, F_c are exactly the partial derivatives $\frac{\partial F}{\partial a}$, $\frac{\partial F}{\partial b}$, and $\frac{\partial F}{\partial c}$. This is backpropagation in a small scalar computational graph.

4 Numerical verification by code

We now implement this procedure in Python and compare the backpropagation results with derivatives obtained from a computer algebra system (SymPy). We define two functions:

- `f_scalar(x,a,b,c)` evaluating $f(x; a, b, c) = ax^2 + b + c^2$,
- `f_partials(x,a,b,c)` returning f_x, f_a, f_b, f_c at a given x.

Then we implement:

- `F_backprop(x,a,b,c)`: evaluation of F and its partial derivatives via backpropagation,
- `F_direct(x,a,b,c)`: evaluation of F and its partial derivatives using SymPy's symbolic differentiation.

For chosen numerical values of x, a, b, c, both methods give the same results (up to floating-point rounding).

```python
import sympy as sp
import numpy as np

# ----- Define symbols for the direct (symbolic) approach -----
x_sym, a_sym, b_sym, c_sym = sp.symbols("x a b c", real=True)

def f_sym(expr):
    return a_sym*expr**2 + b_sym + c_sym**2

# F(x) = f(f(f(x+a) + b) + c)
z0_sym = x_sym + a_sym
z1_sym = f_sym(z0_sym)
z2_sym = z1_sym + b_sym
z3_sym = f_sym(z2_sym)
z4_sym = z3_sym + c_sym
F_sym = f_sym(z4_sym)

Fa_sym = sp.diff(F_sym, a_sym)
Fb_sym = sp.diff(F_sym, b_sym)
Fc_sym = sp.diff(F_sym, c_sym)

F_direct_func = sp.lambdify((x_sym, a_sym, b_sym, c_sym),
                            (F_sym, Fa_sym, Fb_sym, Fc_sym),
                            "numpy")

# ----- Scalar function f and its partials -----
def f_scalar(x, a, b, c):
    return a*x**2 + b + c**2

def f_partials(x, a, b, c):
```

```python
    fx = 2*a*x
    fa = x**2
    fb = 1.0
    fc = 2*c
    return fx, fa, fb, fc

# ----- Backpropagation for F -----
def F_backprop(x, a, b, c):
    # Forward pass
    z0 = x + a
    z1 = f_scalar(z0, a, b, c)
    z2 = z1 + b
    z3 = f_scalar(z2, a, b, c)
    z4 = z3 + c
    F_val = f_scalar(z4, a, b, c)

    # Backward pass
    # Initialize parameter gradients
    Fa = 0.0
    Fb = 0.0
    Fc = 0.0

    # Node F = f(z4)
    fx4, fa4, fb4, fc4 = f_partials(z4, a, b, c)
    g4 = fx4
    Fa += fa4
    Fb += fb4
    Fc += fc4

    # Node z4 = z3 + c
    g3 = g4
    Fc += g4  # derivative wrt c

    # Node z3 = f(z2)
    fx2, fa2, fb2, fc2 = f_partials(z2, a, b, c)
    g2 = g3 * fx2
    Fa += g3 * fa2
    Fb += g3 * fb2
    Fc += g3 * fc2

    # Node z2 = z1 + b
    g1 = g2
    Fb += g2  # derivative wrt b

    # Node z1 = f(z0)
    fx0, fa0, fb0, fc0 = f_partials(z0, a, b, c)
    g0 = g1 * fx0
    Fa += g1 * fa0
    Fb += g1 * fb0
    Fc += g1 * fc0

    # Node z0 = x + a
    Fa += g0  # derivative wrt a

    return F_val, Fa, Fb, Fc

# ----- Numerical test -----
x_val = 0.7
a_val = 1.3
```

```
b_val = -0.5
c_val = 0.9

F_dir, Fa_dir, Fb_dir, Fc_dir = F_direct_func(x_val, a_val, b_val, c_val)
F_bp,  Fa_bp,  Fb_bp,  Fc_bp  = F_backprop(x_val, a_val, b_val, c_val)

print("Direct    F, Fa, Fb, Fc:")
print(F_dir, Fa_dir, Fb_dir, Fc_dir)
print("Backprop  F, Fa, Fb, Fc:")
print(F_bp,  Fa_bp,  Fb_bp,  Fc_bp)

print("\nAbsolute differences:")
print(" |Fa_dir - Fa_bp| =", abs(Fa_dir - Fa_bp))
print(" |Fb_dir - Fb_bp| =", abs(Fb_dir - Fb_bp))
print(" |Fc_dir - Fc_bp| =", abs(Fc_dir - Fc_bp))
```

Running this script yields the same values for F_a, F_b, F_c for the direct and the backpropagation method (up to tiny numerical differences on the order of machine precision), confirming that both approaches are mathematically equivalent.

5 Conclusion

The chain rule approach from Task 2 generalizes directly to functions with more parameters. The only change is that we track more partial derivatives f_a, f_b, f_c and corresponding parameter gradients F_a, F_b, F_c, but the structure of the backpropagation algorithm remains the same.

In the concrete example

$$f(x; a, b, c) = ax^2 + b + c^2, \qquad F(x) = f(f(f(x + a) + b) + c),$$

we implemented backpropagation on the scalar computational graph and verified numerically that the resulting partial derivatives F_a, F_b, F_c coincide with those obtained from direct symbolic differentiation. This small example illustrates the core idea of backpropagation as it is used in training neural networks: efficiently reusing intermediate derivatives across many parameters and layers.