Under the Hood: Computation Graphs, Autograd, and Gradient Descent

Most of the procedures we encounter today in the field of atrificial intelligence are based on modern machine learning approaches. In simple terms, these build models from training data by, given a certain predefined model structure, searching for its parameters such that the conditions of a loss function defined over the training data are satisfied. Satisfying these conditions typically means minimizing the loss function and thus using optimization algorithms. From the constructed models we can identify patterns in the data, and if these are exposed in an understandable way or clearly visualized, we can may interpret the models and then, possibly, discover new knowledge.

The paragraph above might be quite an overload. We'll have to work our way toward fully understanding it. But we won't be doing that in this chapter. The previous paragraph was written only to introduce the concept of a "loss function" and to hint that for some function—possibly simple or very complex—we are looking for its "minimum". For now, we stop here and take a look at everything through a simple example.

Example

Let's start with a simple function:

$$f(a) = a^2 - 10a + 28$$

We want to find such a value of the parameter a at which this function attains its minimum.

Analytical solution. Since the given function is simple, quadratic, we can find its minimum analytically. The function has an extremum at the point where the first derivative equals zero. Let's differentiate our function:

$$\frac{d}{da} f(a) = 2a - 10$$

and set the derivative equal to zero:

$$2a - 10 = 0$$

Solve for a:

$$a = 5$$

The value of the function at this point is:

$$f(5) = 5^2 - 10(5) + 28 = 25 - 50 + 28 = 3$$

So the function reaches its minimum at $a = 5$, where $f(5) = 3$.

Numerical solution. Let's now find this solution numerically. We start the process of finding the minimum at some chosen value of the parameter a, compute the derivative there, and then update the parameter value in the direction of the negative derivative—so we increase a if the derivative is negative, or decrease a if the derivative is positive.

Let's think about it: if the derivative is positive at a given value of a, then increasing a increases the function value. The derivative tells us in which direction and how fast the function value changes for a small change in the parameter a. So if we are looking for a minimum, we need to decrease a. Of course, we also need to decide by how much to decrease it, and intuitively this should depend on the magnitude of the derivative: if the function increases quickly at a given value of a, i.e., the derivative is large, we can adjust a more than in the case where the increase is very slow and we might already be close to the extremum. Similarly, but with the opposite sign, we proceed when the derivative is negative. The update of the parameter a can be written as:

$$a_{t+1} = a_t - \eta \frac{d}{da} f(a_t)$$

where a_{t+1} is the parameter value after the t-th update, and where η is the step size, which in machine learning we also call the *learning rate*. The meta-parameter η therefore determines how large the steps are when updating the parameter value. We call it *meta* because it is a parameter of our numerical method used to find the solution, and not a parameter of the function we are optimizing (those are simply called parameters, without the "meta" prefix).

We of course start with some *initial value* of the parameter, a_0. For example, let's start with $a_0 = 6$ and choose a learning rate $\eta = 0.1$, and see where this leads:

1. Compute the derivative of the function at $a_0 = 6$:

$$\frac{d}{da} f(6) = 2(6) - 10 = 12 - 10 = 2$$

and update the parameter value,

$$a_1 = 6 - 0.1 \times 2 = 6 - 0.2 = 5.8$$

2. Compute the derivative at the new parameter value, $a_1 = 5.8$:

$$\frac{d}{da} f(5.8) = 2(5.8) - 10 = 11.6 - 10 = 1.6$$

and update it again,

$$a_2 = 5.8 - 0.1 \times 1.6 = 5.8 - 0.16 = 5.64$$

3. Compute the gradient at $a_2 = 5.64$:

$$\frac{d}{da} f(5.64) = 2(5.64) - 10 = 11.28 - 10 = 1.28$$

and update the parameter value a,

$$a_3 = 5.64 - 0.1 \times 1.28 = 5.64 - 0.128 = 5.512$$

Did you notice that the value of the derivative is decreasing, and with it also the updates of the parameter a? Of course, we would need to continue the process, and if everything goes as expected, the value of the parameter a should approach 5. At that point, the derivative of the function $f(a)$ goes to zero, and with it also the updates of the parameter vanish.

It's time to implement this kind of function minimization in code:

```
def f(a):
    return a**2 - 10*a + 28

def df(a):
    return 2*a - 10

a = 6
eta = 0.1
for _ in range(20):
    grad = df(a)
    a = a - eta * grad
    print(f"a = {a:.6f}, f(a) = {f(a):.6f}")
```

We used the analytical derivative of the function and implemented it in the function `df`. When we run the program, it quickly approaches the correct solution:

```
a = 5.800000, f(a) = 3.640000
a = 5.640000, f(a) = 3.409600
a = 5.512000, f(a) = 3.262144
...
a = 5.014412, f(a) = 3.000208
a = 5.011529, f(a) = 3.000133
```

Instead of using the analytical form of the derivative, we can also compute it numerically using the finite difference method, as shown in the function below.

```
def df(a, h=0.0001):
    return (f(a + h) - f(a)) / h
```

Typically, using analytically computed derivatives (or gradients) is faster and more accurate, and it does not depend on the choice of the parameter `h`, whose value we would otherwise need to determine (above we simply chose a default value of 0.0001), and which actually affects the accuracy of the results. So from now on, we'll stick to analytically computed derivatives!

Gradient Descent

Before we continue with (analytical) differentiation, let's just note and name this: the procedure we used above to compute the minimum of a given function is called *gradient descent*. At each step, it evaluates how steep the function is at the current parameter value and then updates the parameter in the direction of the negative derivative. It is important to choose an appropriate step size η. If it is too large, we might "overshoot" the minimum; if it is too small, the process will be very slow.

To perform gradient descent, we need to know the derivative of the function with respect to the given parameter. In the case of a single parameter, the gradient is equal to the standard derivative, while for a multi-parameter function we need derivatives with respect to all parameters. The vector of these derivatives is called the gradient and is denoted by $\nabla f(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the vector of parameters:

$$\nabla f(\boldsymbol{\theta}) = \left(\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_n} \right)^T$$

The gradient of a function can also be computed analytically or numerically using the finite difference method, but for the same reasons as with derivatives, we will also use the analytical solution here. Since manually computing derivatives for complex functions (read: complex models), such as neural networks, is tedious, we need to rely on a better solution: using a program for differentiation. We will develop a program for automatic differentiation in the next chapter, but here we first take a look at how to approach the problem of differentiation in general—starting manually.

Computational Graph

Let's start with an example and a simple function of four parameters:

$$L(a, b, c, d) = (ab + c)d$$

For this function, we want to compute the gradient at the parameter values $a = 2$, $b = -3$, $c = 10$, $d = -2$. So we want to compute the following partial derivatives:

$$\frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial c}, \frac{\partial L}{\partial d}.$$

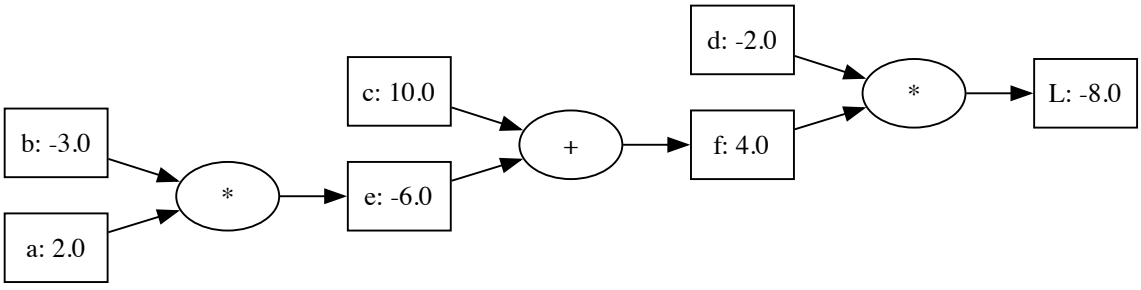
In other words, we want to compute the gradient of the function at the given parameter values:

$$\nabla L = \left(\frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}, \frac{\partial L}{\partial c}, \frac{\partial L}{\partial d}\right)$$

First, let's try to make our derivative computations easier by breaking down the function and rewriting it using intermediate variables, whose values we compute using some basic operations (multiplication or addition):

$$\begin{aligned} e &= ab \\ f &= e + c \\ L &= fd \end{aligned}$$

Written this way, we can now represent the function graphically using the *computational graph* below. In the graphical representation, we also included the (current) values of all derived variables, that is, variables e and f.



Let's start taking derivatives. Notice that our function L directly depends on variables d and f, so it's easiest to start differentiating with respect to those. First for variable d. We are interested in how the value of function L changes if we change the value of d:

$$\frac{\partial L}{\partial d} = \lim_{h \to 0} \frac{f(d+h) - fd}{h} = \lim_{h \to 0} \frac{fd + fh - fd}{h} = \lim_{h \to 0} \frac{fh}{h} = f$$

Since the value of $f = 4$, the gradient of function L with respect to variable (or parameter) d is therefore:

$$\frac{\partial L}{\partial d} = f = 4$$

In a similar way, let's compute the derivative of function L with respect to variable f:

$$\frac{\partial L}{\partial f} = \lim_{h \to 0} \frac{(f+h)d - fd}{h} = \lim_{h \to 0} \frac{fd + hd - fd}{h} = \lim_{h \to 0} \frac{hd}{h} = d$$

Since we know that $d = -2$, the derivative with respect to variable f is:

$$\frac{\partial L}{\partial f} = d = -2$$

To compute the derivative with respect to parameter c, we need to use the chain rule:

$$\frac{\partial L}{\partial c} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial c}$$

where $\frac{\partial f}{\partial c}$ is:

$$\frac{\partial f}{\partial c} = \frac{\partial (e+c)}{\partial c} = \lim_{h \to 0} \frac{(e+c+h) - (e+c)}{h} = \lim_{h \to 0} \frac{e+c+h-e-c}{h} = \lim_{h \to 0} \frac{h}{h} = 1$$

Since we already computed $\frac{\partial L}{\partial f} = d = -2$, we get:

$$\frac{\partial L}{\partial c} = d \times 1 = -2$$

Similarly, we compute the partial derivative with respect to variable e, which is also equal to -2. Now we just need to compute the partial derivatives with respect to variables a and b. Let's start with variable a:

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial a}$$

We already know $\frac{\partial L}{\partial e}$, which is -2, and let's compute $\frac{\partial e}{\partial a}$:

$$\frac{\partial e}{\partial a} = \frac{\partial (ab)}{\partial a} = \lim_{h \to 0} \frac{(a+h)b - ab}{h} = \lim_{h \to 0} \frac{ab + hb - ab}{h} = \lim_{h \to 0} \frac{hb}{h} = b$$

Since $b = -3$ and $\frac{\partial L}{\partial e} = -2$, we get $\frac{\partial L}{\partial a} = 6$. Similarly, we could compute that $\frac{\partial L}{\partial b} = -4$.

Doing all this "by hand" is a bit tedious, and actually, this is the last time here we do it. We here just wanted to highlight that when computing partial derivatives, it can be really helpful to represent a function—no matter how complex—as a computational graph, which typically uses simple and easily differentiable operations, and then compute derivatives backward using the chain rule.

We also saw that when dealing with sums in computational graphs, we "copy" the gradient value from the parent node, while for multiplication we multiply that gradient by the value of the neighboring (sibling) node. We do need to be careful with functions where a node has multiple parents: in that case, we have to sum the contributions to that node. All of this—the process of differentiation using a computational graph—will come in very handy in the next chapter, where we will build code for automatic differentiation.

Autograd

In the previous sections, we saw that when differentiating functions, it helps to write them in the form of a computational graph and then, during differentiation, walk along the graph from the end back to the beginning. This backward "walk" (engl. *back-propagation*) corresponds to the chain rule of differentiation, while all intermediate results are remembered and computed exactly once.

When developing the procedure and code for automatic differentiation, we drew heavily on Andrej Karpathy's outstanding lecture *The spelled-out intro to neural networks and backpropagation: building micrograd*.

Based on this backward walk, we will build an algorithm for automatic differentiation. We will start with its skeleton – the computational graph. First, we will implement a graph node, add information about its predecessors, and check whether we can use the computational graph to compute function values (engl. *forward computation*). Once we are happy with that, we will also implement the computation of gradients. We will develop the procedure gradually and finish it with a demonstration of its usefulness.

Nodes in a computational graph

It's time to also write some code for automatic differentiation, this time in the Python programming language. Let's start by introducing a node in a computational graph, which we implement with the `Value` class, since it will store the value of some variable:

```
class Value:
    def __init__(self, data):
        self.data = data

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

The node thus stores the `data` and can also print its value in a meaningful way. Using the class is straightforward:

```
>>> a = Value(2.0)
>>> a
Value(2.0)
```

Computational graph

We would like to perform computational operations on the nodes, or rather on the variables represented by the nodes. For example, we would like the `Value` class to support the following usage:

```
>>> a = Value(2.0)
>>> b = Value(-3.0)
>>> e = a + b
>>> g = e * b
```

We therefore extend the class for a node in a computational graph with the implementation of addition and multiplication. Since we are building a graph, we will store the connections in such a way that each node remembers its predecessors. For the purpose of visualizing the graph, we will also store the label of the mathematical operation and the name of the variable held by the node. Note that in serious machine learning applications (for example in neural networks), we typically do not need these labels and names; however, when learning about automatic differentiation, this functionality doesn't hurt.

```
class Value:
    def __init__(self, data, _children=(), _op='', label=''):
        self.data = data
        self.label = label
        self._prev = set(_children)
        self._op = _op

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

    def __add__(self, other):
        out = Value(self.data + other.data, (self, other), '+')
        return out

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

Let's build a computational graph for a simple function that we explored in the previous chapter:

```
a = Value(2.0, label='a')
b = Value(-3.0, label='b')
c = Value(10.0, label='c')
d = Value(-2.0, label='d')

e = a * b
e.label = 'e'
f = e + c
f.label = 'f'
L = f * d
L.label = 'L'
```

Let's check that it works:

```
>>> L
Value(L: -8.0)
>>> L._prev
{Value(d: -2.0), Value(f: 4.0)}
```

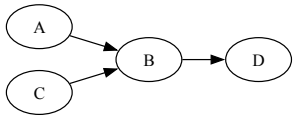
It works!

Drawing the computational graph

It would also make sense to draw the computational graph. For that, we'll use the `graphviz` library, an open-source tool for drawing graphs that is especially suitable for visualizing directed and undirected graphs, where nodes and edges represent data or relationships. To describe a graph, Graphviz uses the DOT language, a simple textual notation for describing nodes and edges. In Python, we replace this description with function calls.

To start, let's look at a simple example of how to use it:

Graphviz was developed by Stephen North and Ellie Gansner at AT&T Labs – Research in the early 1990s (1991). The project emerged as part of research in graph visualization and automatic diagram drawing, and was later released as an open-source project.




```
import graphviz
from IPython.display import display
dot = graphviz.Digraph()
dot.node('A')
dot.node('B')
dot.edge('A', 'B')
dot.node('C')
dot.edge('C', 'B')
dot.node('D')
dot.edge('B', 'D')
display(dot)
```

The code for drawing our computational graph is a bit more complex (here too we follow Andrej Karpathy's approach). Especially

important is the `trace` function, which recursively walks through the graph, collects the nodes and edges, and arranges them topologically. This ordering will also be useful later when computing gradients.

```
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, format='svg', rankdir='LR'):
    """
    format: png | svg | ...
    rankdir: TB (top to bottom graph) | LR (left to right)
    """
    assert rankdir in ['LR', 'TB']
    nodes, edges = trace(root)
    dot = Digraph(format=format, graph_attr={'rankdir': rankdir})

    for n in nodes:
        dot.node(name=str(id(n)), label = "{ %s: %.1f }" % \
         (n.label, n.data), shape='record')
        if n._op:
            dot.node(name=str(id(n)) + n._op, label=n._op)
            dot.edge(str(id(n)) + n._op, str(id(n)))

    for n1, n2 in edges:
        dot.edge(str(id(n1)), str(id(n2)) + n2._op)

    return dot
```

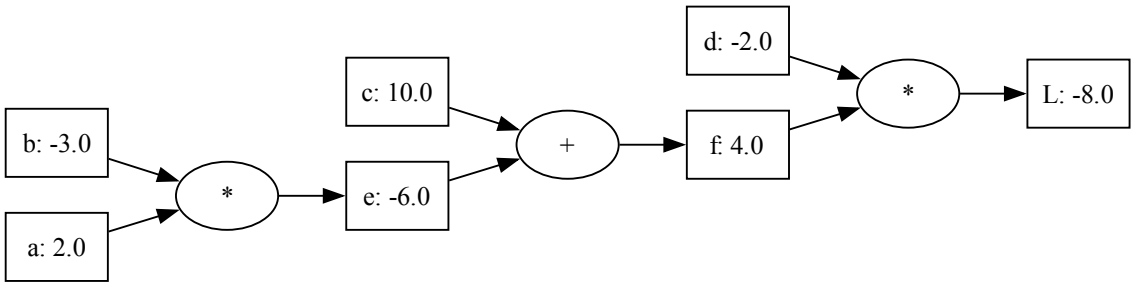
Now we can draw our computational graph:

```
>>> draw_dot(L)
```

Everything is computed correctly. The only thing missing now is derivatives.

Computing gradients

So far, things have been going nicely, but everything we've done up to this point has only been about computing function values. Simple ones, with sums and products. What we're still missing is (at least) the computation of gradients. We compute those using the chain rule,



from back to front. So, from the final nodes with the final function value back to the initial nodes, or input parameters.

We will therefore compute gradients automatically in such a way that each node's gradients will be computed by its successors, namely by appropriately adding the value of their own gradient to them (sum), or in addition multiplying it by the value of the neighboring node (product). All of this will be implemented in the `_backward()` function, which will belong to each of the implemented operations (for now we only have addition and multiplication here, later we'll add some other function as well). For the final computation of the gradient, we need to walk through the computational graph. This will be done by the `backward()` function, which will first topologically order the nodes, and then walk through them, calling `_backward()` on each one.

A few more thoughts before we reveal the final extension of our code. Successors add to the gradients, they do not set them. So at the beginning, the gradients must be initialized to the value 0. This addition is useful in the case where a node has more than one successor. For example, in the function $b = a + a$, each a contributes one 1 to the derivative. In neural networks, for example, nodes in the computational graph will typically have many successors, and each of them will contribute its own value to the node's gradient.

We also take into account that the derivative of the last node, that is, the derivative of the final variable L with respect to L, is equal to 1. So we start with this derivative value and propagate it from that final node back toward the initial nodes. Here is the code, where we implement the `_backward()` function for each operation:

```
class Value:
    def __init__(self, data, _children=(), _op='', label=''):
        self.data = data
```

```
        self.label = label
        self.grad = 0.0
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op

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

    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
```

The only thing left now is to actually use the `_backward()` functions. Let's remember (maybe we're repeating ourselves too much, but it's important) that for a given node, the `_backward()` function records the influence of that node's immediate predecessors on the gradient of the node. If we want to compute the gradients for all variables in the computational graph, we have to start at the back, and in topological order from back to front run `_backward()` for each node. We add the implementation of such backpropagation of gradients to the `Value` class:

```
    def backward(self):
        # topological ordering of the nodes
        topo = []
        visited = set()
        def build_topo(v):
            v.grad = 0
            if v not in visited:
                visited.add(v)
                for child in v._prev:
                    build_topo(child)
                topo.append(v)
        build_topo(self)
```

```
        # application of chain rule
        self.grad = 1
        for v in reversed(topo):
            v._backward()
```

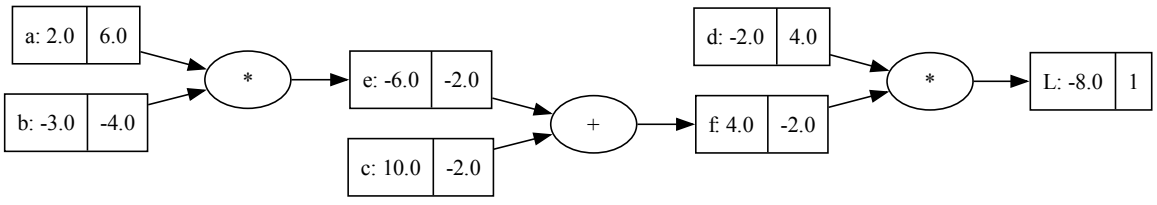
Let's test how it works on our simple function, that is, on the one for which we computed the gradients by hand in the previous chapter:

```
a = Value(2.0, label='a')
b = Value(-3.0, label='b')
c = Value(10.0, label='c')
d = Value(-2.0, label='d')

e = a * b
e.label = 'e'
f = e + c
f.label = 'f'
L = f * d
L.label = 'L'

L.backward()
```

Let's also print the gradient values:

```
>>> a.grad, b.grad, c.grad, d.grad
(6.0, -4.0, -2.0, 4.0)
```

Yay! It works. Or rather: it correctly computes the gradient values for our simple function. Simple because it only adds and multiplies. Below is also the drawing of the computational graph with derivatives:



Constants, negation, exponentiation

For slightly more complex functions, we'd also like to handle constants in our computational graphs, compute differences, powers, and

so on. In other words, we'd like to work with expressions such as the ones below:

```
>>> a = Value(3, 'a')
>>> b = Value(42, 'b')
>>> a + 10
>>> -13 + a
>>> a - b
>>> (a + b) ** 3
```

For all of these, our current implementation returns an error. So we need to extend it. For operations with constants, we will automatically add a new node; in order for them to also be able to appear on the left-hand side in our operations, we will implement functions such as `__radd__`; we will implement subtraction as addition with the negative value of the right operand; and we will also write a new operation for exponentiation. Below is the extended implementation of the `Value` class:

```
class Value:
    def __init__(self, data, _children=(), _op='', label=''):
        self.data = data
        self.label = label
        self.grad = 0.0
        self._backward = lambda: None
        self._prev = set(_children)
        self._op = _op

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

    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 += out.grad
            other.grad += 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):
        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):  # other + self
        return self + other

    def __neg__(self): # - self
        return self * -1

    def __sub__(self, other):
        return self + (-other)

    def __rsub__(self, other):  # other - self
        return other + (-self)

    def __rmul__(self, other):  # other * self
        return self * other
```

From the code above, we've left out the `backward()` function, which stays the same as before. With the code as written above, we can now implement gradient descent for the function from the previous chapter:

```
a = Value(0, label='a')
for _ in range(50):
    L = a**2 - 10 * a + 28
    backward(L)
    a.data -= 0.1 * a.grad
    print(L.data, a.data)
```

Now it's time for the reader to test our code, maybe turn it into a library, and add computation and differentiation for some additional functions. As for us, we'll move on to new adventures in the next chapter, where we'll use this code for automatic differentiation on more serious examples, where the (loss) functions will also use external data and will, for example, implement the search for a model for linear regression or its regularized version. With the extensions above, and maybe a few other small ones, we are now ready for the "serious" use of our little automatic differentiation library.