Blaž Zupan

Machine Learning 1 for Data Science

2026

University of Ljubljana

LECTURE NOTES, UNIVERSITY OF LJUBLJANA

Draft, April 2026

Contents

Introduction

Needs some text here.

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:

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

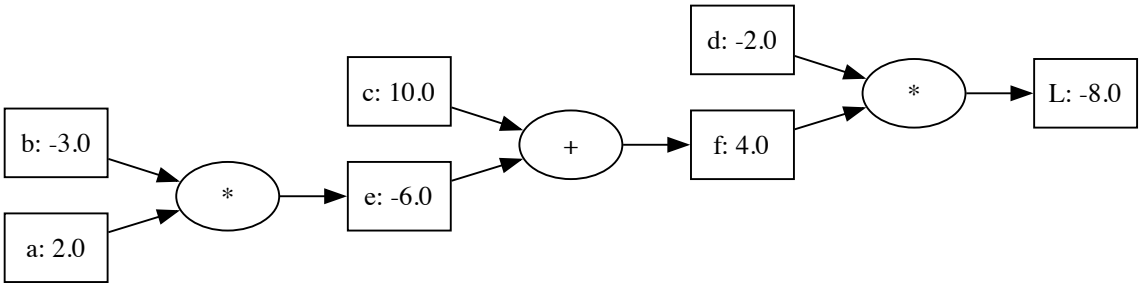
In other words, we want to compute the gradient of the function at the given parameter values:

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

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)}
```

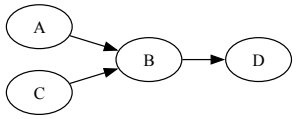
It works!

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
```

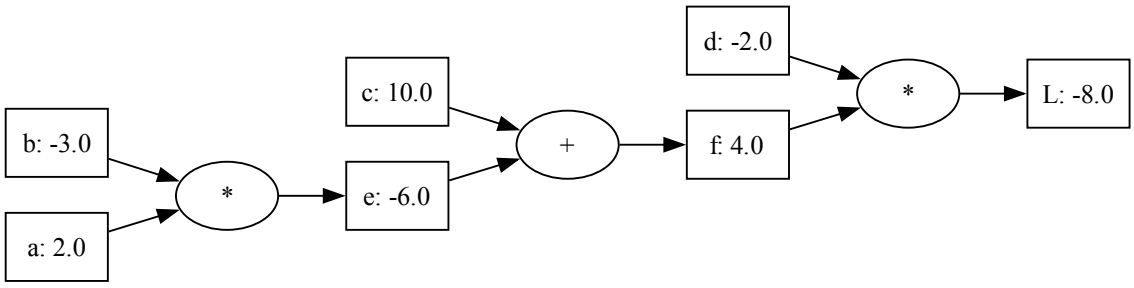
Now we can draw our computational graph:

```
>>> 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()
```

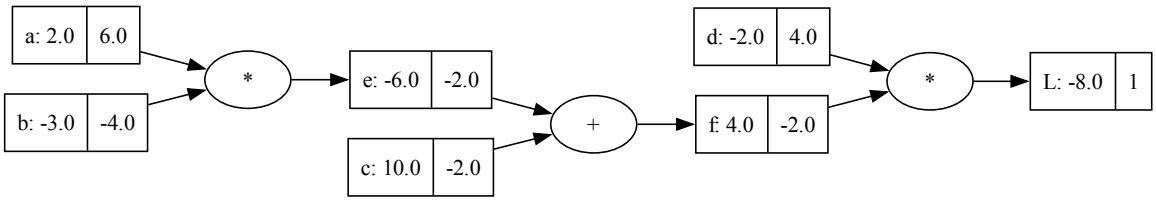
Let's test how it works on our simple function, that is, on the one for which we computed the gradients by hand in the previous chapter:

```
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.

Linear Regression the Autograd Way

This chapter isn't really entirely about linear regression. It's more about how we use automatic differentiation to build models from data. But along the way, we'll also think about linear regression, likelihood, and objective functions. We'll start with univariate linear regression, extend it to multivariate, try everything out on more serious data, and think about whether the discovered models can help us interpret the data.

Linear regression originates from the method of least squares, first published in 1805 by Adrien-Marie Legendre, and later, in 1809, further connected to the normal distribution by Carl Friedrich Gauss in the analysis of astronomical observations. The term *regression* was introduced later, in the 1880s, by Francis Galton while studying heredity. He observed that very tall parents often have tall children, but on average these children tend to be somewhat closer to the population mean; he called this phenomenon *regression toward the mean*.

Data and What We Actually Want

In *univariate regression*, we deal with data that contains one independent variable x and one dependent variable y. The goal is to describe or approximate the functional relationship between them with a linear model of the form

$$\hat{y}(x) = wx + b,$$

where w is the slope (weight) that determines the inclination of the line, and b is the intercept, i.e., the intersection with the vertical axis. So the model is completely determined by two parameters, w and b.

When training the model, we have a training dataset available, consisting of pairs of values (x_i, y_i), for example:

x_i	y_i
1.4	−2.0
−4.7	−18.1
−2.2	−1.9
−2.8	−4.4
2.4	6.5

For each training example, the model predicts a value $\hat{y}_i = wx_i + b$, which can differ from the actual value y_i. We can write the prediction error for an individual example as

$$\varepsilon_i = \hat{y}_i - y_i.$$

Since we don't care about the sign of the error , we square the

We introduced the idea of the sum of squared errors here somewhat informally. We'll get back to why this makes sense a bit later.

errors and compute their average. This gives us the cost function, or loss function, which for a given training set depends only on the model parameters:

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(wx_i + b - y_i)^2.$$

The cost function measures how well the chosen parameters w and b describe the given data. The goal of learning is to find such parameter values that minimize the cost function:

$$(w^*, b^*) = \arg\min_{w,b} J(w,b).$$

The pair (w^*, b^*) represents the optimal model parameters for the given training set and is obtained through model training, i.e., machine learning from data. Machine learning is then the process where, for a given training set and a chosen model structure, we find such model parameters that optimize the chosen objective function.

Univariate Linear Regression in Python

Let's implement linear regression in a Python class `LinReg`. In the implementation, we'll use the `Value` class as developed in the previous chapter and stored from here on in the `agrad` library:

```
from agrad import Value

class LinReg:
    def __init__(self):
        self.w = Value(random.uniform(-1,1), label='w')
        self.b = Value(0.0, label='b')

    def __call__(self, x):
        return self.w * x + self.b

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

    def loss(self, xs, ys):
        yhats = [self(x) for x in xs]
        return sum([(y - yhat)**2  for y, yhat in zip(ys, yhats)])

    def __repr__(self):
        return f"LinReg(w={self.w.data:.3f}, b={self.b.data:.3f})"
```

In the function `__init__()`, we define the initial values of the model parameters. The model call is implemented in the function `__call__()`. The class `LinReg` also returns a list of model parameters,

which we'll need when updating the parameters during gradient descent. Extremely important, however, is the `loss()` function, which computes the loss for given parameter values and a given training set.

For a first test of how `LinReg` works, we can try using our class to compute linear regression with specific model parameters:

```
>>> model = LinReg()
>>> model.w = Value(10); model.b = Value(3)
>>> model(10)
Value(: 103)
```

On a small training set, for example:

```
import random
random.seed(42)
n = 5
xs = [random.uniform(-5, 5) for _ in range(n)]
ys = [2*x -1 + random.gauss(0, 4) for x in xs]
```

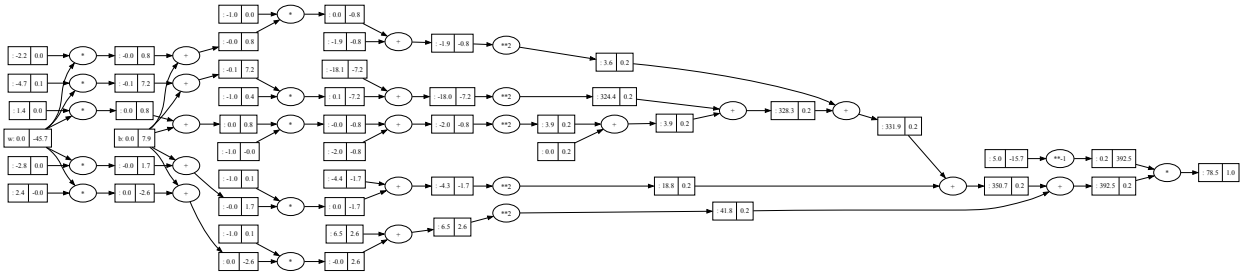
we can now check

```
>>> lr = LinReg()
>>> lr
LinReg(w=0.011, b=0.000)
>>> lv = lr.loss(xs, ys)
>>> lv
Value(: 78.49574710788079)
```

The loss (sum of squared errors) is of course large, since the above model is still untrained and random. What's interesting, though, is the computational graph for the loss. The code for drawing it is:

```
from agrad import draw_dot
lv.backward()
dot = draw_dot(lv)
dot.render('a', format='pdf', cleanup=True)
```

An attentive reader will of course notice that before drawing the computational graph, we computed the gradients. We hid the implementation of the drawing in the `agrad` library, but it's exactly the same as in the previous chapter, so we don't need to show it again here. The computational graph for our loss function is already quite complex here, even with a small training set, partly because it includes a sum over all five training examples:



Training

Ah, finally! Everything is ready to compute the *right* parameter values of our univariate model from the data. Recall: using the computational graph, we'll compute partial derivatives of the function with respect to the model parameters, adjust them a bit each time, and repeat the process until convergence or for some predefined number of iterations.

Let's now put together a training function that implements gradient descent. The optimization function below is actually quite general—we could use it for any model that, during training, uses labeled examples.

```
def train(model, xs, ys, learning_rate=0.001, n_epochs=1000):
    for k in range(n_epochs):
        # compute loss
        loss = model.loss(xs, ys)

        # compute gradients
        for p in model.parameters():
            p.grad = 0
        loss.backward()

        # update
        for p in model.parameters():
            p.data -= learning_rate * p.grad

        if k % 50 == 0:
            print(f"{k:3} Loss: {loss.data:5.3f} {model}")
    return model
```

In each iteration, we built the computational graph for our loss, set the gradients of the model parameters to zero (since we accumulate gradients into these values), then computed the gradients and updated the parameter values.

Time to train:

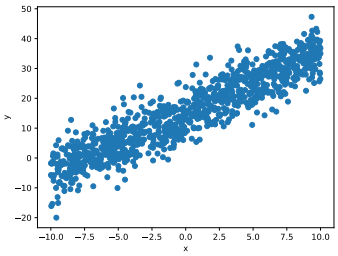
```
>>> model = train(LinReg(), xs, ys, learning_rate=0.01, n_epochs=500)
  0 Loss: 130.147 LinReg(w=-0.326, b=-0.102)
 50 Loss: 16.793 LinReg(w=2.578, b=-0.745)
100 Loss: 16.778 LinReg(w=2.566, b=-0.827)
150 Loss: 16.775 LinReg(w=2.560, b=-0.864)
200 Loss: 16.774 LinReg(w=2.558, b=-0.880)
250 Loss: 16.774 LinReg(w=2.557, b=-0.887)
300 Loss: 16.774 LinReg(w=2.556, b=-0.890)
350 Loss: 16.774 LinReg(w=2.556, b=-0.891)
400 Loss: 16.774 LinReg(w=2.556, b=-0.892)
450 Loss: 16.774 LinReg(w=2.556, b=-0.892)
```

So in a few hundred iterations, gradient descent found the (almost) correct model, i.e., the one with parameters $w = 2$ and $b = -1$. We leave it to the reader to experiment with different learning rates. Let's just say that for this data and this model, training can converge faster, but if we increase the learning rate too much, we may run into errors like `'OverflowError: (34, 'Result too large')`. Why?

Batch Learning

A reader of these notes might point out that the example in the previous section with only five samples was too simple, and that such linear regression could easily be done "by hand". Maybe—but even for such a small example, the amount of computation would already be quite large. So now is the right moment to test our implementation on a larger dataset:

```
random.seed(42)
n = 1000
xs = [random.uniform(-10, 10) for _ in range(n)]
ys = [2*x + 1 + random.gauss(0, 5) for x in xs]
```



With a large number of examples, the computational graph for the loss function grows significantly, and training can therefore become quite slow. There may be enough data to try a different approach, where we compute the loss only on a sample of the training set. This is called batch learning (or *batch learning*), and we can implement it simply with the function below, which we add to the `LinReg` class:

```
    def batch_loss(self, xs, ys, m=10):
        indices = random.sample(range(len(xs)), m)
        batch_xs = [xs[idx] for idx in indices]
        batch_ys = [ys[idx] for idx in indices]
        return self.loss(batch_xs, batch_ys)
```

For batch learning, we now only need to change the line that defines the loss function. It now becomes

```
loss = model.batch_loss(xs, ys, batch_size)
```

and assumes that we add another hyperparameter `batch_size` to the training method, with a default value, say 10.

With batch learning, the loss over the full training set will not decrease monotonically:

```
>>> model = train(LinReg(), xs, ys, learning_rate=0.01, n_epochs=500)
  0 Loss: 442.514 LinReg(w=0.620, b=0.266)
 50 Loss: 111.536 LinReg(w=1.792, b=9.632)
100 Loss: 12.292 LinReg(w=1.868, b=12.622)
150 Loss: 24.878 LinReg(w=2.042, b=13.802)
200 Loss: 46.238 LinReg(w=1.705, b=14.689)
250 Loss: 30.237 LinReg(w=1.685, b=14.732)
300 Loss: 23.126 LinReg(w=2.113, b=14.865)
350 Loss: 29.744 LinReg(w=1.662, b=14.941)
400 Loss: 25.119 LinReg(w=2.435, b=14.894)
450 Loss: 10.819 LinReg(w=1.884, b=14.988)
```

Our model isn't exactly perfect either, but the model parameters have gotten quite close to the ones we used to generate the data. Batch learning is much faster than gradient descent where we compute the loss over the entire training set each time, but we need to be careful when choosing the learning rate and the batch size.

All of a sudden, we've accumulated quite a few hyperparameters in our training procedure. These are parameters that influence the training process, including the learning rate, the number of epochs, and the batch size. We leave it to the reader to think about how to set these parameters—we'll deal with them more in the following chapters.

Multivariate Linear Regression

Let's extend our linear regression model to the multivariate case, i.e., to models that take multiple input, independent variables, which in machine learning we also call features.

```
random.seed(42)
n = 1000
X = [[random.uniform(-10, 10) for _ in range(3)] for _ in range(n)]
ys = [2*x[0] + 3*x[1] - x[2] + 1 + random.gauss(0, 5) for x in X]
```

There shouldn't be any major changes in the training code. In the `LinReg` class, the functions `loss()` and `batch_loss()` stay the same as before, while we slightly modify the others,

Our implementation could also be improved by storing the names of the variables. These become important whenever we try to interpret the model.

```
class LinReg:
    def __init__(self, n_inputs):
        self.weights = [Value(random.uniform(-1,1), label=f'w{i}')
                        for i in range(n_inputs)]
        self.b = Value(0.0, label='b')
```

```
    def __call__(self, x):
        # x is a list of input values
        return sum(w * xi for w, xi in zip(self.weights, x)) + self.b

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

    def __repr__(self):
        weights_str = ', '.join(f'w{i}={w.data:.3f}'
                                for i, w in enumerate(self.weights))
        return f"LinReg({weights_str}, b={self.b.data:.3f})"
```

Training, where the `train` function remains exactly as we developed it in the previous sections, successfully converges to the correct solution:

```
>>> lr = LinReg(n_inputs=3)
>>> model = train(lr, X, ys, n_epochs=10000, batch_size=50, learning_rate=0.01)
   0 Loss: 863.724 LinReg(w0=1.623, w1=2.440, w2=-0.938, b=-0.068)
 500 Loss: 26.504 LinReg(w0=2.054, w1=2.709, w2=-0.993, b=0.805)
1000 Loss: 32.210 LinReg(w0=2.174, w1=2.939, w2=-0.932, b=0.787)
...
4500 Loss: 18.590 LinReg(w0=1.745, w1=2.719, w2=-0.993, b=0.832)
5000 Loss: 32.224 LinReg(w0=2.269, w1=2.903, w2=-1.074, b=0.841)
```

It's actually quite surprising how easily we extended our implementation to learning from data with an arbitrary number of features. Still, just to remind you, we're using the automatic differentiation library we developed in the previous chapter, which is perfectly capable even for this last, not exactly small example.

Likelihood

Now it's time to take a closer look at why we defined the objective function as minimizing the sum of squared errors. Although this choice is intuitive, it also has a clear statistical explanation. In multivariate linear regression, we assume that observations are generated according to the linear model

$$y_i = \mathbf{w}^\top \mathbf{x}_i + b + \varepsilon_i,$$

where $\mathbf{x}_i = (x_{i1}, \ldots, x_{id})$ is the vector of input features, $\mathbf{w} = (w_1, \ldots, w_d)$ is the vector of weights, b is the intercept, and ε_i is the random error for the i-th example. We further assume that the errors ε_i are independent and identically distributed.

This assumption is often written as i.i.d., meaning the errors are *independent and identically distributed*.

Additionally, we assume that the errors are normally distributed with mean zero:

$$\varepsilon_i \sim \mathcal{N}(0, \sigma^2).$$

The assumption of a normal distribution is often reasonable. According to the central limit theorem, the sum of a large number of small, independent effects tends toward a normal distribution, so the normal distribution is often a good model for measurement noise.

Since

$$y_i = \mathbf{w}^\top \mathbf{x}_i + b + \varepsilon_i,$$

it follows that y_i is also normally distributed around the value predicted by the model:

$$y_i \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x}_i + b, \sigma^2).$$

The probability density of observing y_i is therefore

$$p(y_i \mid \mathbf{x}_i, \mathbf{w}, b) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\big(y_i - (\mathbf{w}^\top \mathbf{x}_i + b)\big)^2}{2\sigma^2}\right).$$

Now let's define the likelihood as a function of the model parameters, which tells us *how likely the observed data would be if they were actually generated by the model with given parameter values*. Since we assumed that the examples are independent, the likelihood for the entire training set is the product of the probabilities of all examples:

$$L(\mathbf{w}, b) = \prod_{i=1}^{n} p(y_i \mid \mathbf{x}_i, \mathbf{w}, b).$$

The concept of likelihood emerged in the early 20th century in statistics, when Ronald A. Fisher developed the method of maximum likelihood as a general approach for estimating parameters of statistical models. The idea is simple: we choose the model parameters for which the observed data are most likely. In machine learning, this concept became central in modern probabilistic machine learning, as it enables a systematic derivation of training criteria. It is important because it provides a statistical justification for learning models: instead of ad-hoc criteria, we optimize a function that directly measures how well the model explains the observed data.

We choose the model parameters so that the likelihood is as large as possible. This approach is called maximum likelihood (or *maximum likelihood estimation*). Since the product of many terms can be inconvenient to compute, we usually maximize the log-likelihood:

$$\log L(\mathbf{w}, b) = \sum_{i=1}^{n} \log p(y_i \mid \mathbf{x}_i, \mathbf{w}, b).$$

If we plug in the expression for the normal distribution, we get

$$\log L(\mathbf{w}, b) = -\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} \big(y_i - (\mathbf{w}^\top \mathbf{x}_i + b)\big)^2.$$

The first term is a constant that does not depend on the parameters $\mathbf{w}$ and b. The factor $1/(2\sigma^2)$ also does not affect the maximum. So maximizing the log-likelihood is equivalent to minimizing the expression

$$\sum_{i=1}^{n} \big(y_i - (\mathbf{w}^\top \mathbf{x}_i + b)\big)^2.$$

And that is exactly the sum of squared errors that we used as the objective function when training linear regression.

So it turns out that minimizing the squared error is not just an intuitive choice, but follows directly from the assumption that the

measurement errors in the data are normally distributed. In this framework, we estimate the parameters of the linear model using maximum likelihood.

Interpretation

The parameters of linear regression models are actually feature weights, and linear regression is a weighted sum of the inputs. The weights are related to the role of each feature, i.e., its influence on the output value of the linear function. Smaller weights correspond to less important features, while larger weights correspond to more important ones. The sign of the weight is also important, as it tells us whether the feature is negatively or positively related to the output. There is still room here for further thinking about interpretation, maybe even some interesting visualizations, but let's take it step by step and focus here only on the weights.

We would also like, for example, to determine the smallest set of features that still gives us a good model and ignore the rest. But that will be the topic of the next chapter.

Let's start with a practical example. We'll use a dataset with body measurements of men. For each example, we have the body fat percentage, which was computed using the Brozek formula (not important for us, but that's where the name of the target variable comes from, *Body Fat Brozek*) based on body density measured using underwater weighing. In addition, the data contains age, weight, height, and a number of easily measured body circumferences such as neck, chest, abdomen, and so on. Our goal will be to build a model that predicts body fat percentage from these simple measurements, and at the same time estimate which of the measurements plays the most important role. This is a typical regression problem: the examples are described by multiple input features and one continuous output variable.

The data comes from the *Body Fat Prediction Dataset*, which is publicly available on Kaggle and dates back to 1985. It contains 252 samples, 14 features, and a target variable representing body fat percentage.

Let's load the data and print the first five examples for a few selected features:

```
df = pd.read_csv("body-fat-brozek.csv")
feature_names = df.columns[1:].tolist()
ys = df.iloc[:, 0].tolist()
X = df.iloc[:, 1:].values.tolist()

preview = (
    df[["body fat brozek", "age", "weight", "ankle"]]
    .head()
)
print(preview.to_string(index=False))
```

Below is a table with the first five examples and selected features. The first column is the target, and the other three are some of the features. You can see that the feature scales are different—weight

values (clearly not in kg) are much larger than the values used for ankle circumference.

body fat brozek	age	weight	ankle
12.6	23	154.25	21.9
6.9	22	173.25	23.4
24.6	22	154.00	24.0
10.9	26	184.75	22.8
27.8	24	184.25	24.0

Table 1: First five examples from the dataset.

If we left the data in this form, the weights would depend on the scale of the feature values, and even if the weight for weight were smaller than that for ankle circumference, the two features could still be equally important. To remove the influence of feature scale, we normalize the data (features):

```
X_arr = np.array(X)
mean = X_arr.mean(axis=0)
std = X_arr.std(axis=0)
X_norm = ((X_arr - mean) / std).tolist()
```

Now everything is ready to build the model and analyze the resulting weights:

```
lr_norm = LinReg(n_inputs=len(feature_names))
model_norm = train(lr_norm, X_norm, ys, n_epochs=1000, batch_size=20, learning_rate=0.05)

def print_weights(model, feature_names):
    pairs = [(name, w.data) for name, w in zip(feature_names, model.weights)]
    pairs.sort(key=lambda p: abs(p[1]), reverse=True)
    for name, w in pairs:
        print(f"{w:6.3f} {name}")
```

The result,

```
>>> print_weights(model_norm, feature_names)
9.433 abdomen
-2.489 weight
-1.519 hip
-1.498 wrist
 1.367 tight
 1.330 forearm
-1.121 neck
 0.767 age
 0.347 ankle
 0.284 adiposity
 0.229 biceps
-0.078 chest
-0.041 height
```

```
-0.024 knee
```

is actually in line with expectations: to estimate body fat percentage, it's best to know the abdomen circumference. Next come weight, hip circumference, and wrist circumference, which have negative weights. A negative sign does not necessarily mean that higher weight or larger hips reduce body fat; it simply means that, when taking all other features into account, the model uses these variables as corrections to the prediction. Such situations are common when features are strongly correlated with each other.

We leave it to the reader to try what happens if the data is not normalized. Just a hint: the results would be very different.

Regularization and Feature Selection

In the previous chapter, we used computational graphs and gradient descent to build a linear regression model and then interpreted it through its weights. On the healthcare dataset, the most important attribute indeed received the largest weight, but the model also relied on many others. This raised two issues. First, some attributes were strongly correlated, which made interpretation harder. Second, all attributes remained in the model, since none of their weights was zero. It would be useful to construct a model that keeps only some attributes and ignores the rest. Such a model would be easier to use, easier to interpret, and could also tell us which attributes are not important for prediction.

We can implement the above idea of excluding certain input variables from the model using a procedure called *regularization*. But before introducing it, let's start with an example where additional attributes can strongly harm learning.

Polynomial Regression

We start with the data in Table 2. There is nothing particularly special about this data, except for its apparent unsuitability for linear regression, as also shown in the figure below.

x	y
0.19	0.28
0.09	0.24
0.16	0.48
0.34	0.60
0.59	0.71
0.87	0.51
0.89	0.37
0.68	0.62
0.36	0.51
0.82	0.48

Table 2: Training data.

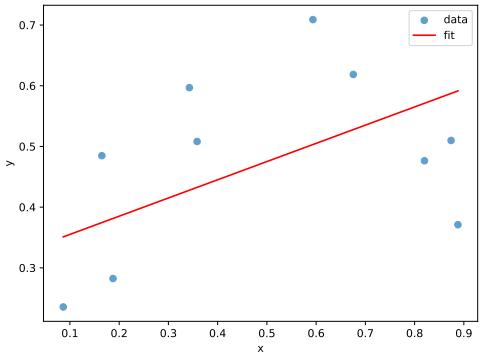


Figure 1: Linear regression does not model our initial data from Table 2 very well.

We could say that this data cannot be modeled with linear regression. Or can it? Let's add a new, derived attribute x^2 to the table and

use this extended table as input to linear regression. It will find the weights of the attributes for the model

$$y = w_0 + w_1 x + w_2 x^2.$$

We have thus added one more attribute to the linear model. As a function of x, the resulting model is quadratic rather than linear. But it is still linear in the parameters w_0, w_1, and w_2, so we can fit it with the same method as before.

We can again visualize the results of training such a model in the (x, y) plane, as shown in the figure below. Excellent! The model now fits the training data well. The value of the cost function that

x	x^2	y
0.19	0.04	0.28
0.09	0.01	0.24
0.16	0.03	0.48
0.34	0.12	0.60
0.59	0.35	0.71
0.87	0.76	0.51
0.89	0.79	0.37
0.68	0.46	0.62
0.36	0.13	0.51
0.82	0.67	0.48

Table 3: To the previous data table (Table 2) we added a new column, i.e., a new attribute computed from column x.

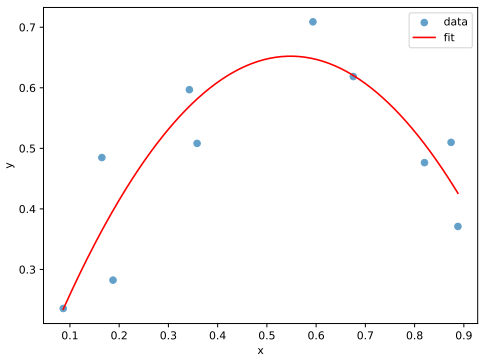


Figure 2: This is also linear regression.

we minimize in linear regression (mean squared error) is also much smaller than in the case where we did not add the new variable to the table. It decreased from 0.018 for the basic data to only 0.005.

Nothing really stops us from continuing to add more attributes. We could, for instance, add higher-order powers. Say x^3, x^4, all the way up to, say, x^7. We managed to reduce the cost function a bit further, this time to 0.003. Success, right!?

Such an extension of the training set is called *polynomial expansion*, and the trick of combining it with linear regression is called *polynomial regression*.

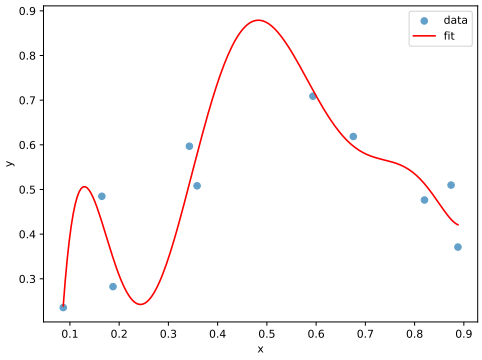


Figure 3: And this is also linear regression.

The desire to minimize the cost function has led us (almost) to the

What would happen if we added attributes up to a ninth-degree polynomial, i.e., including x^9? What would the value of the objective function be then? Why?

extreme. The model started to completely adapt to the training set and as such is no longer useful for prediction.

L2 Regularization

If we take a closer look at the model weights in the example above, we notice an interesting phenomenon. In the second-degree model, the weights are still relatively small. However, when we start adding new attributes, i.e., powers x^3, x^4, x^5, and so on, the weights quickly become very large. Some are positive, others negative, and their absolute values can become thousands or even tens of thousands of times larger than the values of the input variables.

This also explains why the fitted function becomes more complicated. Large weights make the prediction very sensitive to small changes in x. The model then starts to follow minor fluctuations in the training data, including noise. As a result, it can fit the training set very well while performing poorly on new examples.

Observing the values of the weights when adding attributes, as we did above, leads us to the following idea. In addition to wanting a good fit to the training data, when building a linear model we may also want the weights to be as small as possible, making the model simpler. We therefore need to include, in the cost function, not only the model error but also a penalty for large weights. Since we do not care whether a weight is positive or negative, we square the weights. We usually omit the weight w_0, as it only represents the intercept of the function. The cost function of linear regression thus becomes

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} w_j^2.$$

The first term of the cost function is the same as before, i.e., the mean squared prediction error. The second term penalizes large weights. Since this is a sum of two terms that are not in the same units and are conceptually different, we need to properly weight the second term. We do this using the parameter λ, which we call the *regularization strength*. In the implementation, we therefore only modify the computation of the loss function inside the `LinReg` class that implements linear regression:

```
def loss(self, xs, ys):
    yhats = [self(x) for x in xs]
    loss = sum([(y - yhat)**2  for y, yhat in zip(ys, yhats)]) \
           / Value(len(xs))
    loss += self.reg_strength * sum([w**2 for w in self.weights]) \
            / len(self.weights)
    return loss
```

The `loss` function assumes that we have properly initialized the variable `self.reg_strength` in the class. We additionally divide the regularization term by the number of model weights, noting that `self.b` is not included.

When the regularization strength $\lambda = 0$ (that is, when `self.reg_strength` equals 0.0), we obtain exactly the same model as in standard linear regression. As we increase the value of λ, the penalty for large weights becomes increasingly important. The model therefore prefers smaller weights, and as a result the function $y(x)$ becomes smoother and simpler. What happens if we increase the regularization strength to a very large value? In this case, the second term in the cost function dominates, and the model tends toward making all weights as close to zero as possible.

What will the value of w_0 be in this case? The cost function can then be written as

$$J(w_0) = \frac{1}{n}\sum_{i=1}^{n}(y_i - w_0)^2.$$

Let's find the minimum of this function. We differentiate with respect to w_0:

$$\frac{\partial J}{\partial w_0} = \frac{1}{n}\sum_{i=1}^{n} 2(w_0 - y_i).$$

At the minimum, the derivative must be zero:

$$\sum_{i=1}^{n}(w_0 - y_i) = 0.$$

From this it follows that

$$nw_0 = \sum_{i=1}^{n} y_i,$$

or

$$w_0 = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

With very strong regularization, the model therefore predicts only the mean of the training targets y. In other words, the simplest possible regression model is the one that ignores the input attributes and always predicts the average target value.

The procedure we just described is called *regularization*. Since we square the weights, we more specifically refer to it as *L2 regularization*. This approach is also commonly known in the literature as ridge regression.

The name comes from the shape of the optimization problem: the added term $(\lambda \sum_j w_j^2)$ changes the surface of the cost function so that it forms a pronounced ridge along directions where parameters are poorly determined due to strong correlations between attributes. The regularization term "rounds" this ridge, giving the problem a unique solution and keeping the weights bounded.

Evaluating Model Accuracy with R^2

Let's spend a bit more time on predicting with the mean value on the training set. That is, on the simplest regression model, which we

build without taking attribute values into account. We expect that models that do take these values into account will be more accurate, i.e., that their error will be smaller. It therefore makes sense to evaluate the ratio between the error obtained by an attribute-informed model and our baseline model, where we predict using the mean value:

$$\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

where $\hat{y}_i$ is the prediction of our model and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the mean value of the target variable. If the value of this ratio is close to 0, it means that our model is much better than predicting with the mean. If it is equal to 1, the model is as good as the baseline model, and if it is greater than 1, it is even worse.

Since we want a measure where higher values indicate a better model (ideally close to 1) and worse values are closer to 0, we subtract this ratio from 1 and obtain the R^2 measure:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}.$$

The R^2 measure is often convenient because it is dimensionless and compares the model directly with the baseline that always predicts the mean. RMSE, by contrast, is expressed in the units of the target variable. For this reason, R^2 is often easier to compare across different problems. It can also be interpreted as the proportion of variance explained by the model, although this interpretation should be used with some care. An R^2 value close to 1 indicates a good fit, a value near 0 means that the model performs similarly to the baseline, and negative values indicate that it performs even worse.

L1 Regularization

With L2 regularization, we penalized large weights by adding the sum of their squares to the cost function. As a result, the weights became smaller, but typically none of them became exactly zero. The model therefore still used all attributes, just with somewhat smaller weights. However, if our goal is for the model to completely eliminate some attributes, we need a slightly different approach. Instead of squaring the weights, we can include the sum of their absolute values in the cost function. We obtain the cost function

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} |w_j|.$$

What happens when we increase the regularization strength λ? Similar to before, the model starts to prefer smaller weights. But this time something else interesting happens: some weights become exactly zero. Such an attribute is effectively no longer used by the model. We could say that it has been removed from the model.

L1 regularization therefore has an additional useful property: it can perform *feature selection*. Some weights become exactly zero, which means that the corresponding input variables are removed from the model.

If we keep increasing the regularization strength, there will be fewer and fewer non-zero weights. For very large values of λ, only the weight w_0 will remain, and the model will again become a constant that predicts the mean value of the training data.

The procedure we just described is called *L1 regularization*. In the statistical literature, it is also known as *LASSO* (short for *Least Absolute Shrinkage and Selection Operator*). Its key feature is precisely that, in addition to regularization, it also enables automatic feature selection, which is often very useful in practice. In principle, implementing this regularization again only changes the computation of the objective function. Below is a function that implements both types of regularization discussed so far:

```
def loss(self, xs, ys):
    yhats = [self(x) for x in xs]

    loss = sum([(y - yhat)**2  for y, yhat in zip(ys, yhats)]) \
           / Value(len(xs))
    if self.reg == 'l1':
        loss += self.reg_strength * sum([abs(w) for w in self.weights]) \
                / len(self.weights)
    elif self.reg == 'l2':
        loss += self.reg_strength * sum([w**2 for w in self.weights]) \
                / len(self.weights)
    return loss
```

In many cases, this is enough. If we want to encourage exact zeros even more strongly, we can add an extra step after each gradient update and set very small weights to zero. In practice, this means choosing a threshold and replacing all weights below it in absolute value by zero. This kind of thresholding strengthens the tendency of L1 regularization to remove unimportant attributes.

```
if model.reg == 'l1':
    shrink = learning_rate * model.reg_strength
    for p in model.parameters():
        if abs(p.data) < shrink:
            p.data = 0
```

Graphical Comparison of Regularizations

We can also understand regularization in a slightly different way. Instead of adding the regularization term to the cost function, we can formulate the problem as a constrained optimization:

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \qquad \text{subject to} \qquad R(\mathbf{w}) \leq s.$$

Here, the function $R(\mathbf{w})$ determines the size of the weights, and the parameter s limits how large these weights are allowed to be.

If we only have two weights, w_1 and w_2, we can visualize this problem in the (w_1, w_2) plane. The error contours (without regularization) are ellipses. The solution is found at the point where the lowest ellipse first touches the feasible set of weights.

For L2 regularization, the constraint is

$$w_1^2 + w_2^2 \leq s,$$

which represents a circle in the plane. The point of contact with the ellipse therefore usually occurs somewhere along the smooth boundary of the circle, so the weights are smaller, but typically none of them is exactly zero.

For L1 regularization, the constraint becomes

$$|w_1| + |w_2| \leq s,$$

which forms a diamond shape in the (w_1, w_2) plane. Since the diamond has sharp corners along the coordinate axes, the ellipses often touch it exactly at one of these corners. At such a point, one of the weights is exactly zero.

Regularization and Accuracy on Training and Test Sets

So far, we judged the model only by how well it fit the training data. Regularization changes this, because it introduces a trade-off between fit and simplicity. As the regularization strength increases, training accuracy usually decreases. The more important question is what happens on the test set.

Let's observe these relationships through an experiment. We will take the body fat dataset that we already used in the previous chapter and now split it into training and test data. To make the effect of regularization more pronounced, we will sample a very small train-

Why would such a data split make the effect of regularization more visible?

ing set and leave all the remaining data for testing:

```
df = pd.read_csv("body-fat-brozek.csv")
feature_names = df.columns[1:].tolist()
ys = df.iloc[:, 0].tolist()
X = df.iloc[:, 1:].values.tolist()

X_arr = np.array(X)
mean = X_arr.mean(axis=0)
std = X_arr.std(axis=0)
X_norm = ((X_arr - mean) / std).tolist()

X_train, X_test, y_train, y_test =
   train_test_split(X_norm, ys, test_size=0.95, random_state=42)
```

Before training, we also standardized the data. This puts all features on a comparable scale and usually makes gradient descent more stable and faster. We now train the model with several regularization strengths and report accuracy on both the training and test sets.

```
reg_strengths = [0.001, 0.05, 0.01, 0.1, 0.5, 1, 5, 10, 20, 30]
for reg_strength in reg_strengths:
    lr = LinReg(n_inputs=len(feature_names), reg='l1',
                reg_strength=reg_strength)
    model = train(lr, X_train, y_train, n_epochs=1000,
                  batch_size=None, learning_rate=0.01)
    # print("Weights (most to least important):")
    # print_weights(model, feature_names)

    # evaluate the model on the training and testing sets
    y_pred_train = [model(xi).data for xi in X_train]
    r2_train = r2_score(y_train, y_pred_train)

    y_pred_test = [model(xi).data for xi in X_test]
    r2_test = r2_score(y_test, y_pred_test)

    non_zero_weights = len([w for w in model.weights if w.data != 0])
    print(f"lambda: {reg_strength:5.2f}, R2: {r2_train:.3f} & " \
          "{r2_test:.3f}, non-zero weights: {non_zero_weights}")
```

The results show a monotonic decrease in accuracy on the training set as the regularization strength increases:

```
lambda:  0.00, R2: 0.925 & 0.493, non-zero weights: 14
lambda:  0.05, R2: 0.927 & 0.490, non-zero weights: 13
lambda:  0.01, R2: 0.933 & 0.398, non-zero weights: 14
lambda:  0.10, R2: 0.929 & 0.472, non-zero weights: 14
lambda:  0.50, R2: 0.898 & 0.518, non-zero weights: 7
lambda:  1.00, R2: 0.889 & 0.540, non-zero weights: 8
```

```
lambda:  5.00, R2: 0.756 & 0.484, non-zero weights: 2
lambda: 10.00, R2: 0.725 & 0.592, non-zero weights: 3
lambda: 20.00, R2: 0.564 & 0.453, non-zero weights: 1
lambda: 30.00, R2: 0.496 & 0.414, non-zero weights: 1
```

However, the behavior on the test set is quite different, where the model is most accurate at regularization strength $\lambda = 10$. We would observe similar behavior with L2 regularization as well, but we leave that and further experiments to the reader. The question that arises, however, is how to find the regularization strength that leads to a model best suited for new data.

How We Find the Right Regularization Strength

Basically, it is hard, and we have to be very careful, especially because we want to report both the "right" regularization strength and an estimate of the accuracy of the model obtained in this way. Here we will assume that we search for the regularization strength by choosing it from some list and checking when our model achieves the best result on a separate set. Just as we did in the previous section. There, we got the best model at $\lambda = 10$, and its accuracy on the training set was $R^2 = 0.592$. So do we choose that model and predict that its accuracy on new examples will probably be $R^2 = 0.592$?

No. This estimate is biased, because we selected the regularization strength precisely based on the test set, which was therefore "used up" for learning. Such an accuracy estimate is therefore too optimistic and does not reflect the actual performance of the model on new data.

At this point, learning no longer consists only of fitting a linear model. It also includes choosing the regularization strength. We therefore have to evaluate the entire procedure on an independent test set.

So, to avoid bias in the accuracy estimate, we need to split the data into an additional test set. We will therefore divide the data into three disjoint sets:

1. **training set** (say 70% of the data), on which we train the model for a given regularization strength,

2. **validation set** (e.g., 15%), on which we observe the accuracy of the model for a given regularization strength and which allows us to select the appropriate strength, that is, the one for which the regularized model is most accurate on the validation set,

3. **test set** (e.g., 15%), on which we finally estimate the accuracy of the model.

Which model should we report? Not simply the one that happened to perform best on the validation set during the first split, because that model was trained on only part of the available data. Instead, after reserving the test set for the final evaluation, we repeat the model-selection procedure on the remaining data and then fit the selected model again.

We still need a training-validation split in the last step because model construction now includes hyperparameter selection. In other words, the final method is not just linear regression with fixed settings, but a complete procedure that also chooses the regularization strength.

Finally, let us describe the procedure proposed above in pseudocode:

```
# accuracy estimation
split data to train, validation, test
for reg in lambdas:
    model = fit(train, reg)
    score[reg] = evaluate(model, validation)
choose best reg
score = evaluate(fit(train, best_reg), test)

# derivation of final model
split data to train, validation
for reg in lambdas:
    model = fit(train, reg)
    score[reg] = evaluate(model, validation)
choose best reg
model = fit(train, best_reg)
```

Already quite complicated. But there is another problem here. The above is appropriate for really large datasets, but for smaller ones, the split into subsets itself can strongly affect the accuracy estimate. For that, we could use cross-validation and report an estimate as the average over several experiments. Would you know how to modify the above appropriately and include cross-validation?

Generalized Linear Models

The only predictive model we have considered so far is linear regression. Let us recall: at the input, we have features with continuous values x_i, which we weight with coefficients θ_i and add a bias term (intercept) θ_0, thereby obtaining a prediction of the target value $\hat{y}$. The model is obtained by estimating parameter values that minimize the sum of squared errors between the actual and predicted values of the target variable in the training set. Linear regression is one of the simplest regression models; an even simpler approach would be to predict using the mean value of the dependent variable in the training set, but such an approach does not take into account the values of the input features, and therefore can hardly be considered a true predictive model.

In this chapter, the vector and matrix notation of the above model will be useful, where we represent the data in a feature matrix X, combine the parameters into a vector $\boldsymbol{\theta}$, and express the predictions as $\hat{\mathbf{y}} = X\boldsymbol{\theta}$, which allows for a more compact representation and more efficient implementation of learning algorithms. We assume that the first column in the data matrix is a vector of ones, so that we avoid special treatment of the intercept, and that the indices for θ run from 0 to d, where d is the number of independent variables.

The question arises whether there exist other similarly simple models, perhaps for somewhat different predictive tasks, that is, models in which features are again combined into a weighted sum, and then the resulting value is transformed by an appropriate (nonlinear) link function, so that we can also model discrete or otherwise constrained target variables. In what follows, we begin with an example of such a model, which we will use for classification, and then ask whether such extensions are sufficiently general for a broader class of models, what assumptions they actually make, where their objective functions come from, and whether this represents an interesting class of models with shared properties.

Example

Let us begin with a (fictional) example. Table 4 contains swimmers at Bled whom we asked how many hours per week they exercise and how many hours they slept the previous night. We also recorded whether, on the day of the interview, they managed to swim to the island. The island is more than half a kilometer away from the nearest swimming area in one direction, so swimming to the island and back is quite a challenge and would not be suitable for weaker swimmers. The goal is to develop an application that would advise swimmers, based on their physical fitness and level of rest, whether they should attempt such an endeavor. The application, of course, requires a predictive model, which we can build from our data.

Since the data are two-dimensional, it is best to plot them in a scatter plot. We also mark the class. At first glance, the plot suggests that it might be possible to separate good swimmers from those who mostly just bathe with a line, i.e., a decision boundary. This boundary is linear, so we can write it as $X\theta = 0$. The plot also includes three new visitors: Sara, Martin, and Leon. For which of them will our application, or model, recommend that they can swim to the island and back?

Sara is on the swimmers' side. She is far from the decision boundary that separates the two classes. She can certainly swim to the island and back. Martin is far on the other side; he should definitely not move away from the shore. Leon is, with respect to the decision boundary, on the swimmers' side, but only barely. Advising him to try swimming to the island would be quite wrong. Our problem is indeed a classification one—we want to predict one of two possible classes—but it would be better to do this cautiously, using probabilities. Since Sara is very far from the decision boundary, she is certainly a good candidate for going to the island; Martin definitely is not, while Leon is somewhere in between, his probability of belonging to the "swimmer" class is around 50%.

This suggests that the distance from the decision boundary should be mapped to probabilities. The linear combination $z = X\theta$ is in fact proportional to the distance from the line determined by the parameters θ. For the transformation, we can use a function whose range is between 0 and 1. An example of such a function is the sigmoid $\sigma(z)$, and our probability is then

$$P(y = 1 \mid X) = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

The decision boundary in Figure has parameters $\theta_0 = -15.6$, $\theta_1 = 0.8$, and $\theta_2 = 1.6$, so we can write the decision equation for the

Figure 4: An example of classification data with a meta attribute, independent variables, and the class (Island).

Name	Exercise	Sleep	Island
Alenka	7	8	1
Ana	2	8	0
Andrej	5	5	0
Blaž	5	9	1
Boštjan	7	5	0
Goran	12	6	1
Gregor	10	9	1
Helena	4	9	1
Irena	9	3	0
Janez	5	6	0
Jure	8	4	0
Katarina	4	3	0
Klara	3	9	1
Luka	5	4	0
Maja	9	6	0
Marko	4	7	0
Matej	4	8	1
Miha	6	4	0
Mojca	11	5	1
Nika	2	5	0
Nina	8	6	1
Petra	3	6	0
Polona	9	8	1
Rok	10	6	1
Sara	6	7	1
Sašo	10	7	1
Sebastjan	12	5	1
Tatjana	12	9	1
Tjaša	11	3	0
Tomaz	12	5	1

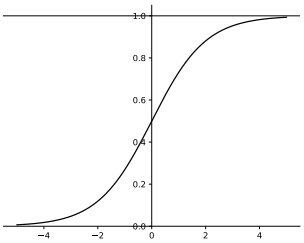


Figure 5: Sigmoid function.

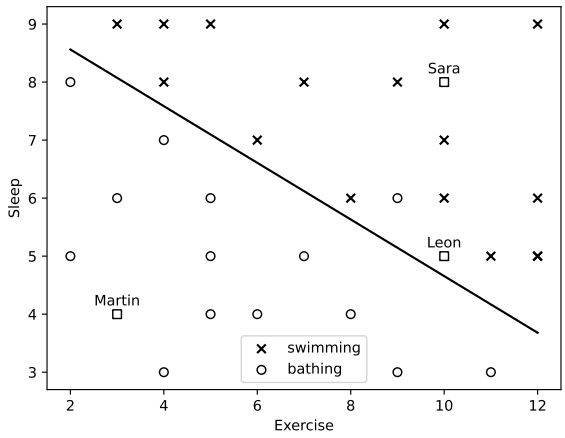


Figure 6: Training data, a possible decision boundary between the classes, and new (named) examples that we still need to classify.

distance from this boundary as

$$z = -15.6 + 0.8 \cdot \text{exercise} + 1.6 \cdot \text{sleep}.$$

For the new examples, we obtain: Sara has $z = 5.5$ and $P(\text{island} = 1) \approx 1.0$, so she is a very suitable candidate for swimming to the island; Martin has $z = -6.7$ and $P(\text{island} = 1) \approx 0.0$, so we advise against it; Leon has $z = 0.6$ and $P(\text{island} = 1) \approx 0.6$, which means he is close to the decision boundary and the decision is quite uncertain.

The model that we introduced rather quickly, and perhaps somewhat superficially, in our example is called logistic regression. It is important to note that, like linear regression, this model also uses a linear combination of independent variables, but this time the result is transformed using a sigmoid function, precisely so that we can output probabilities. However, several questions arise: how do we actually learn the "correct" parameters of our model, i.e., the vector θ? What objective function do we optimize for this purpose? From what assumptions is it derived? Besides linear and logistic regression, are there other models of this type? And finally, can we also use automatic differentiation and gradient descent to learn such a model?

It is time for a bit of theory.

Exponential Family of Distributions

Where do models such as linear and logistic regression come from? What assumptions do we actually make about the data? We take a similar approach as we already know from linear regression: instead of inventing an objective function (e.g., the sum of squared errors on the training set), we start from a probabilistic model, that is, a model

that generates the data in the training set, and from this assumption we derive the objective function.

A class of distributions that turns out to be particularly useful for our purposes is the *exponential family*. A distribution belongs to this family if its probability function (or density) for the variable y can be written in the form

$$p(y \mid \theta) = h(y) \exp\left(\eta(\theta)\, T(y) - A(\theta)\right),$$

where θ denotes the parameter of the distribution. Since in machine learning we are interested in the log-likelihood, it makes sense to take the logarithm of this expression:

$$\log p(y \mid \theta) = \eta(\theta) T(y) - A(\theta) + \log h(y).$$

The function $T(y)$ is called the *sufficient statistic* and in the simplest cases is equal to y. The function $\eta(\theta)$ is the so-called *natural parameter*, which represents a transformation of the original parameter θ. The function $A(\theta)$ ensures normalization of the distribution (this factor ensures that probabilities sum or integrate to 1), while $h(y)$ is the part independent of the parameter. We observe that the log-likelihood is linear in $T(y)$, which enables simple optimization and a connection with linear models.

In the exponential family of distributions, we find most of the distributions of interest in machine learning that are related to regression (predicting a continuous target), classification (predicting a discrete variable), and modeling counts. These distributions include, for example, the normal, Bernoulli, and Poisson distributions. Further details, of course, follow.

Natural parameter and link function

In supervised learning, we aim to model the conditional expected value of the target variable, that is, its mean value given input features x (in the case of classification, this corresponds to the probability of belonging to a particular class):

$$\mu(x) = \mathbb{E}[y \mid x].$$

In generalized linear models, we make the key assumption that the natural parameter is related to the features through a linear predictor

$$\eta = X\theta.$$

This is not a consequence of the exponential family, but rather a modeling assumption that extends the idea of linear regression to a broader class of distributions. For distributions from the exponential

family, however, it holds that that the expected value is determined by the natural parameter, $\mu = A'(\eta)$. In typical cases, we can invert this relationship and write

$$\eta = g(\mu).$$

This gives us the model

$$g(\mu(x)) = X\theta,$$

where the function g is called the link function.

If we choose the function g such that it coincides with the natural relationship between μ and η, we speak of the *canonical link function*. This choice leads to particularly simple expressions for the likelihood and its derivatives, and often to convex optimization problems.

Since the density must be normalized, we have

$$\int h(y) \exp\left(\eta T(y) - A(\eta)\right) dy = 1.$$

Differentiating with respect to η gives

$$\mathbb{E}[T(y)] - A'(\eta) = 0,$$

therefore

$$\mathbb{E}[T(y)] = A'(\eta).$$

Model learning

Here we only recall that, in order to determine the model parameters, we will need a criterion function, and for that we need the likelihood or its logarithm. In fact, we already have everything prepared for this. Assume that the training examples are independent of each other, and once we choose the target distribution of the classes, we can write the likelihood for the training data as

$$p(\mathbf{y} \mid X, \theta) = \prod_{i=1}^{n} p(y_i \mid x_i, \theta).$$

To learn the parameters, we maximize the log-likelihood

$$\ell(\theta) = \sum_{i=1}^{n} \log p(y_i \mid x_i, \theta),$$

which is equivalent to minimizing the negative log-likelihood, which can be interpreted as a criterion function.

This connects directly to standard machine learning practice: different choices of distributions lead to different criterion functions, while the optimization proceeds in the same way, e.g. using gradient descent.

Linear regression

Let us start with the simplest example, which we have already encountered, but now view it from a new perspective. Suppose that the target variable is continuous and follows a normal distribution

$$y \mid x \sim \mathcal{N}(\mu(x), \sigma^2).$$

The density can be written as

$$p(y \mid x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right).$$

If we rearrange the expression, we obtain

$$p(y \mid x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{\mu}{\sigma^2}y - \frac{\mu^2}{2\sigma^2} - \frac{y^2}{2\sigma^2}\right),$$

which is of the exponential family form

$$p(y \mid \theta) = h(y)\, \exp\big(\eta(\theta)\, T(y) - A(\theta)\big),$$

where we identify:

$$T(y) = y, \qquad \eta(\theta) = \frac{\mu}{\sigma^2}, \qquad A(\theta) = \frac{\mu^2}{2\sigma^2}, \qquad \log h(y) = -\frac{y^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma^2).$$

If we assume that the variance σ^2 is constant, then the natural parameter η is proportional to μ, so (with a slight abuse of notation) we can treat it simply as μ, which is equal to a linear combination of the features. In this case, the canonical link function is the identity,

$$g(\mu) = \mu = X\theta.$$

The log-likelihood for a single example is

$$\log p(y \mid x, \theta) = -\frac{1}{2\sigma^2}(y - X\theta)^2 + C,$$

where C does not depend on θ. Maximizing the likelihood is therefore equivalent to minimizing the sum of squared errors:

$$L(\theta) = \sum_{i=1}^{n} (y_i - x_i^T\theta)^2.$$

As we already know, the criterion function of linear regression directly follows from the assumption of normally distributed noise.

Logistic regression

In the case of binary classification, we assume that the target variable follows a Bernoulli distribution:

$$y \mid x \sim \text{Bernoulli}(p(x)),$$

where x is the vector of observed features for a given example, and $y \in \{0, 1\}$ is the class. The function $p(x)$ represents the probability that the class is equal to 1 given the features x, that is,

$$P(y = 1 \mid x) = p(x), \qquad P(y = 0 \mid x) = 1 - p(x).$$

The expected value of the target variable y given features x is therefore

$$\mu(x) = \mathbb{E}[y \mid x] = p(x).$$

The probability function of the target variable y given features x can be written in a single equation as

$$p(y \mid x) = p^y (1-p)^{1-y},$$

and its logarithm as

$$\log p(y \mid x) = y \log p + (1-y) \log(1-p).$$

This probability will be used to express the log-likelihood or the negative log-likelihood.

We rearrange this expression:

$$\log p(y \mid x) = y \log \frac{p}{1-p} + \log(1-p).$$

Under the assumption of independent training examples, this form is already suitable for constructing the likelihood, and then the criterion function. But that follows shortly. First, we must address how to relate p to $X\theta$.

Now we can compare the expression with the general form of the exponential family

$$\log p(y \mid \theta) = \eta(\theta)\, T(y) - A(\theta) + \log h(y),$$

and identify the components:

$$T(y) = y, \qquad \eta = \log \frac{p}{1-p}, \qquad A(\eta) = -\log(1-p), \qquad h(y) = 1.$$

The variable η, or the function $\eta(p)$ (the natural parameter), transforms the probability p into the logarithm of the odds ratio, i.e. $\log \frac{p}{1-p}$ (eng. *log-odds*). Since $\mu = p$, we obtain the link function, called the logit:

$$g(\mu) = \log \frac{p(x)}{1-p(x)} = X\theta,$$

which leads to the well-known sigmoid form with which we, somewhat intuitively, began this chapter:

$$p(x) = \frac{1}{1+e^{-X\theta}}.$$

Poisson regression

To model counts (e.g. the number of events in a given time interval), we assume a Poisson distribution:

$$y \mid x \sim \text{Poisson}(\lambda(x)).$$

This means that $y \in \{0, 1, 2, \dots\}$ and

$$p(y \mid x) = \frac{\lambda^y e^{-\lambda}}{y!}.$$

The expected value of this distribution is

$$\mu(x) = \mathbb{E}[y \mid x] = \lambda(x).$$

To see the connection with the exponential family, we take the logarithm:

$$\log p(y \mid x) = y \log \lambda - \lambda - \log(y!).$$

This expression is already almost in the form

$$\log p(y \mid x) = \eta\, y - A(\eta) + \log h(y),$$

from which we identify

$$\eta = \log \lambda.$$

Since $\mu = \lambda$, we obtain the link function

$$g(\mu) = \log \mu,$$

which is called the log link. The linear model can thus be written as

$$\log \lambda(x) = X\theta,$$

from which it follows that

$$\lambda(x) = e^{X\theta}.$$

The log-likelihood for a single example is

$$\log p(y \mid x, \theta) = yX\theta - e^{X\theta} - \log(y!),$$

and the negative log-likelihood leads to the criterion function

$$L(\theta) = \sum_{i=1}^{n} \left(e^{x_i^T \theta} - y_i x_i^T \theta \right),$$

where the term $\log(y!)$ can be omitted since it does not depend on the parameters.

Some coding

The above was a lot of theory and few examples. For a short break, let us check whether logistic regression really gives a solution similar to the one in our initial example. As with linear regression from the previous chapter, we begin with a class that implements the criterion function over the input data.

```
class LogReg:
    def __init__(self, n_inputs, reg=None, reg_strength=0.0):
        self.weights =
```

```
        [Value(random.uniform(-1, 1), label=f"w{i}")
         for i in range(n_inputs)]
        self.b = Value(0.0, label="b")
        self.reg = reg
        self.reg_strength = reg_strength

    def linear(self, x):
        return sum(w * xi for w, xi in zip(self.weights, x)) + self.b

    def __call__(self, x):
        return self.linear(x).sigmoid()

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

    def loss(self, xs, ys):
        eps = 1e-8
        losses = []
        for x, y in zip(xs, ys):
            yhat = self(x)
            y_val = Value(float(y))
            term = -(y_val * (yhat + eps).log() + \
              (1 - y_val) * (1 - yhat + eps).log())
            losses.append(term)
        data_loss = sum(losses) / Value(len(xs))

        if self.reg == "l2" and self.reg_strength > 0:
            l2_penalty = self.reg_strength * sum(w * w for w in self.weights)
            return data_loss + l2_penalty
        return data_loss

    def __repr__(self):
        weights_str = ", ".join(f"w{i}={w.data:.3f}" \
          for i, w in enumerate(self.weights))
        return f"LogReg({weights_str}, b={self.b.data:.3f})"
```

Upon initialization, `LogReg` creates a vector of (randomly initialized) weights and a bias term, optionally supporting L2 regularization controlled by `reg` and `reg_strength`. The method `linear` computes the affine combination of inputs and parameters, while the `__call__` method applies the sigmoid function to obtain a probabilistic prediction. The `parameters` method exposes all learnable quantities for optimization, which are used in the `train` function of autograd library (we defined this in our previous writings, and will not change it here). The key methods is the `loss` function, which constructs the computational graphs for the average binary cross-entropy over a traing set. A small numerical constant is added for numerical stability. Optionally, we add L2 penalty to the loss.

We invoke training with a call similar to one we have used in our

previous chapters for linear regression.

```
df_train = pd.read_excel("bled.xlsx")
feature_names = ["Exercise", "Sleep"]
xs = df_train[feature_names].values.tolist()
ys = df_train["Island"].astype(int).tolist()

model = LogReg(n_inputs=len(feature_names),
               reg="l2", reg_strength=0.01)
model = train(model, xs, ys, learning_rate=0.1,
              n_epochs=10000, batch_size=None)
```

The optimization converges relatively fast, and the weights and the end result is actually very similar to the one from Fig. . We could speed-up the process through data standardization, and, of course, by using faster optimization routines.

The changes to implement Poisson regression would be rather minimal. Instead of mapping the linear predictor through a sigmoid, we would use the exponential function to ensure that the predicted rate parameter remains positive, and the loss would correspond to the negative log-likelihood of the Poisson distribution. Concretely, the following parts of the implementation would change:

```
class PoissonReg(LogReg):
    def __call__(self, x):
        return self.linear(x).exp()

    def loss(self, xs, ys):
        losses = []
        for x, y in zip(xs, ys):
            yhat = self(x)
            y_val = Value(float(y))
            term = yhat - y_val * yhat.log()
            losses.append(term)
        return sum(losses) / Value(len(xs))
```

Multinomial logistic regression

The logistic regression we have considered so far is intended for binary classification. However, we often encounter tasks where there are more than two possible classes. For example, we may want to predict which mode of transport an individual will choose, or which category a document belongs to. Such problems can be addressed by an extension of logistic regression called *multinomial logistic regression*. This model is a special case of generalized linear models, where the target variable follows a categorical distribution.

Let the target variable be $y \in \{1, 2, \ldots, m\}$, where m is the number of classes. For each class j, we define a linear predictor

$$z_j = x^T \theta_j.$$

Since we want to obtain probabilities that are non-negative and sum to 1, we use the function

$$P(y = j \mid x) = \frac{e^{z_j}}{\sum_{l=1}^{m} e^{z_l}},$$

which is called the *softmax*. This function maps a vector of real values $(z_1, \ldots, z_m)$ to a probability vector.

The model is not uniquely determined, since we can add the same constant to all z_j without changing the probabilities. Therefore, we usually choose one class as the reference class and set its linear predictor to zero:

$$z_m = 0.$$

The model thus has $(m - 1)$ parameter vectors. Multinomial logistic regression is a generalized linear model where the target variable follows a categorical distribution, the linear predictor is $z_j = x^T \theta_j$, and the link function is the generalized logit, whose inverse is the softmax function.

The model can also be written as

$$y \mid x \sim \text{Categorical}(p_1(x), \ldots, p_m(x)),$$

where

$$p_j(x) = \frac{e^{x^T \theta_j}}{\sum_{l=1}^{m} e^{x^T \theta_l}}.$$

If we have only two classes ($m = 2$), the softmax simplifies to the sigmoid function, and we recover standard logistic regression.

The model parameters are estimated by maximizing the log-likelihood

$$\ell(\theta) = \sum_{i=1}^{n} \log P(y_i \mid x_i),$$

which leads to the criterion function known as multiclass cross-entropy. As in logistic regression, there is no closed-form solution, so the parameters are estimated numerically, e.g. using gradient descent.

Ordinal logistic regression

In some problems, the target variable is not only discrete, but its values also have a natural ordering. Such variables are called *ordinal*. Examples include survey responses (e.g. “disagree”, “neutral”,

"agree") or ratings (e.g. from 1 to 5). Such data could be handled using multinomial logistic regression, but that approach does not take into account the ordering between classes. Ordinal logistic regression explicitly models this ordering, and is therefore often more efficient and interpretable.

The basic idea of the model is that there exists a latent (unobserved) continuous variable z, which depends linearly on the features:

$$z = x^T\theta.$$

The observed ordinal variable y is determined by which interval the latent variable z falls into. These intervals are defined by thresholds (cutpoints) $t_1, t_2, \ldots, t_{m-1}$:

$$y = j \quad \text{if} \quad t_{j-1} < z \leq t_j,$$

where by convention we take $t_0 = -\infty$ and $t_m = \infty$.

To obtain probabilities, we assume that the latent variable is subject to logistic noise. This leads to a model where the probability that y is less than or equal to a given class is

$$P(y \leq j \mid x) = \frac{1}{1 + e^{-(t_j - x^T\theta)}}.$$

This expression represents a cumulative probability, which is why the model is often called the *cumulative logit model*. The probabilities of individual classes are obtained as differences between consecutive cumulative probabilities:

$$P(y = j \mid x) = P(y \leq j \mid x) - P(y \leq j - 1 \mid x).$$

Ordinal logistic regression is a generalized linear model where the target variable follows a categorical distribution with ordered classes, the linear predictor is $x^T\theta$, and the link function is the cumulative logit.

The model uses a single parameter vector θ, while the thresholds t_j determine the boundaries between classes. As a result, the model has fewer parameters than multinomial logistic regression.

The model parameters are estimated by maximizing the log-likelihood

$$\ell(\theta) = \sum_{i=1}^{n} \log P(y_i \mid x_i),$$

which again leads to an optimization problem without a closed-form solution, so the parameters are estimated numerically.