D. Autodiff - Hands-On Machine Learning with Scikit-Learn and TensorFlow

Appendix D. Autodiff

This appendix explains how TensorFlow's autodiff feature works, and how it compares to other solutions.

Suppose you define a function $f(x,y) = x^2y + y + 2$, and you need its partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are manual differentiation, symbolic differentiation, numerical differentiation, forward-mode autodiff, and finally reverse-mode autodiff. TensorFlow implements this last option. Let's go through each of these options.

Manual Differentiation

The first approach is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the partial derivatives manually. For the function $f(x,y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda - 1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions' derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive Equation D-1:

Equation D-1. Partial derivatives of *f*(*x*,*y*)

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y\frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

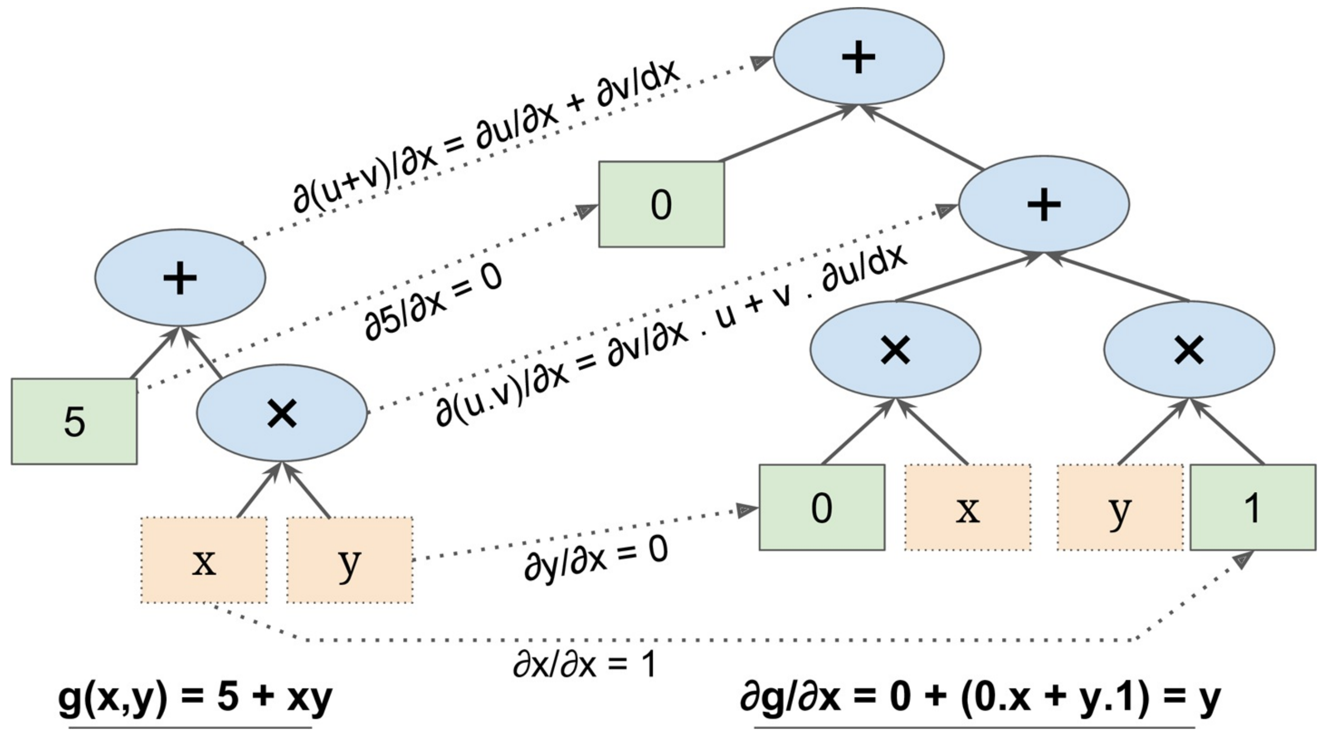
$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. The good news is that deriving the mathematical equations for the partial derivatives like we just did can be automated, through a process called *symbolic differentiation*.

Symbolic Differentiation

Figure D-1 shows how symbolic differentiation works on an even simpler function, *g*(*x*,*y*) = 5 + *xy*. The graph for that function is represented on the left. After symbolic differentiation, we get the graph on the right, which represents the partial derivative (we could similarly obtain the partial derivative with regards to *y*).

$$\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$$



∂(u+v)/∂x = ∂u/∂x + ∂v/dx
∂5/∂x = 0
∂(u.v)/∂x = ∂v/∂x . u + v . ∂u/dx
∂y/∂x = 0
∂x/∂x = 1
+
+
+
×
×
×
5
0
0
1
x
y
x
y
g(x,y) = 5 + xy
∂g/∂x = 0 + (0.x + y.1) = y

Figure D-1. Symbolic differentiation

The algorithm starts by getting the partial derivative of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable *x* returns the constant 1 since $\frac{\partial x}{\partial x} = 1$, and the variable *y* returns the constant 0 since $\frac{\partial y}{\partial x} = 0$ (if we were looking for the partial derivative with regards to *y*, it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function *g*. Calculus tells us that the derivative of the product of two functions *u* and *v* is . We can therefore construct a large part of the graph on the right, representing 0 × *x* + *y* × 1.

$$\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + \frac{\partial u}{\partial x} \times u$$

Finally, we can go up to the addition node in function *g*. As mentioned, the derivative of a sum of functions is the sum of these functions' derivatives. So we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: .

However, it can be simplified (a lot). A few trivial pruning steps can be applied to this graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\frac{\partial g}{\partial x} = y$.

$$\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$$

In this case, simplification is fairly easy, but for a more complex function, symbolic differentiation can produce a huge graph that may be tough to simplify and lead to suboptimal performance. Most importantly, symbolic differentiation cannot deal with functions defined with arbitrary code—for example, the following function discussed in Chapter 9:

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i
)
    return z
```

Numerical Differentiation

The simplest solution is to compute an approximation of the derivatives, numerically. Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point, or more precisely Equation D-2.

Equation D-2. Derivative of a function $h(x)$ at point x_0

$$h'(x) = \lim_{x \to x_0} \frac{h(x) - h(x_0)}{x - x_0} = \lim_{\epsilon \to 0} \frac{h(x_0 + \epsilon) - h(x_0)}{\epsilon}$$

So if we want to calculate the partial derivative of $f(x,y)$ with regards to x, at $x = 3$ and $y = 4$, we can simply compute $f(3 + □, 4) – f(3, 4)$ and divide the result by □, using a very small value for □. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps
)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complex functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call `f()` at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When you are dealing with large neural networks, this makes numerical differentiation way too inefficient.

However, numerical differentiation is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

Forward-Mode Autodiff

Forward-mode autodiff is neither numerical differentiation nor symbolic differentiation, but in some ways it is their love child. It relies on *dual numbers*, which are (weird but fascinating) numbers of the form $a + b\epsilon$ where a and b are real numbers and ϵ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$). You can think of the dual number $42 + 24\epsilon$ as something akin to 42.0000⋯000024 with an infinite number of 0s (but of course this is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, $42 + 24\epsilon$ is represented by the pair (42.0, 24.0).

Dual numbers can be added, multiplied, and so on, as shown in Equation D-3.

Equation D-3. A few operations with dual numbers

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$
$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$
$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Most importantly, it can be shown that $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, so computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. Figure D-2 shows how forward-mode autodiff computes the partial derivative of $f(x,y)$ with regards to x at $x = 3$ and $y = 4$. All we need to do is compute $f(3 + \epsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\frac{\partial f}{\partial x}(3, 4)$.

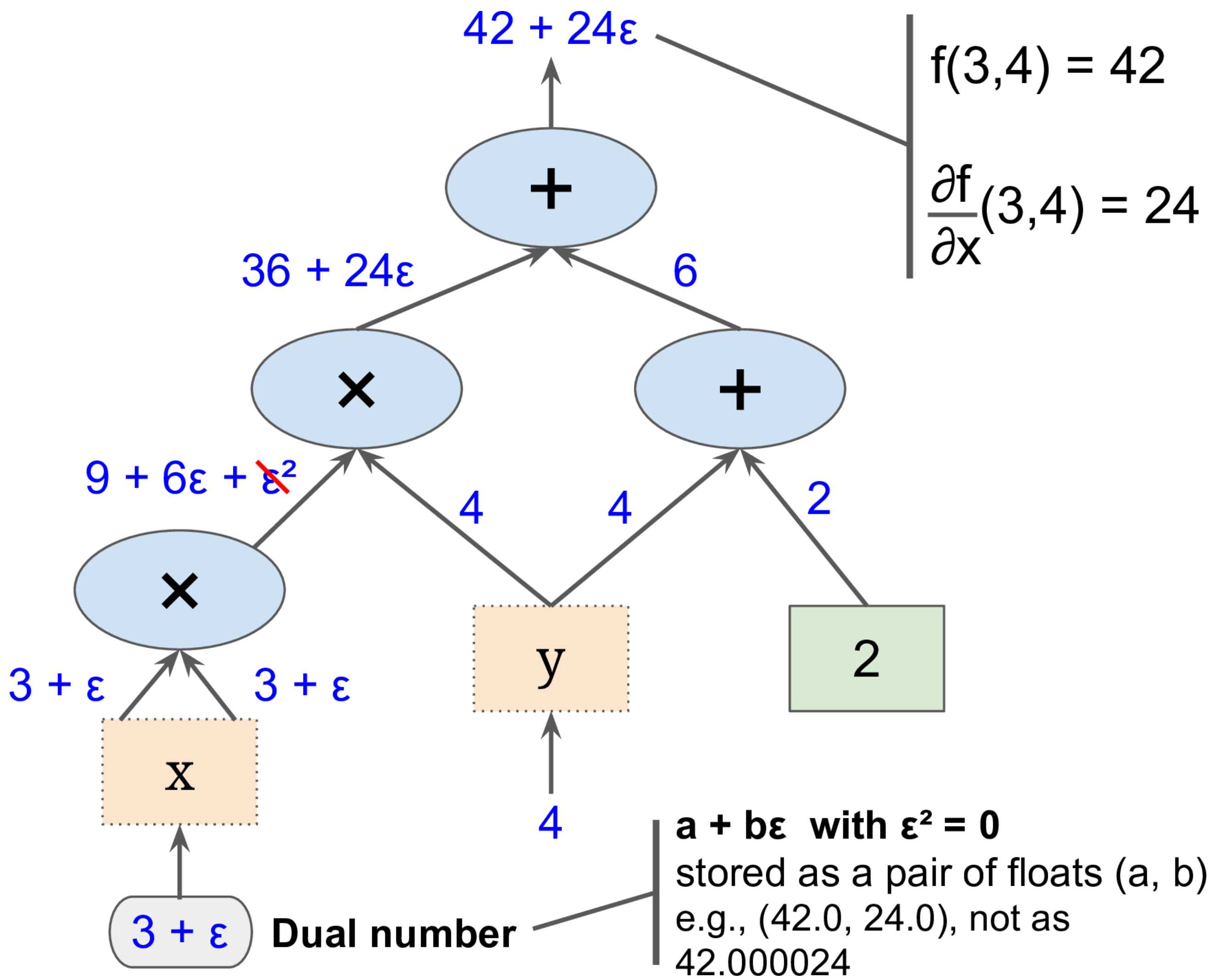


Figure D-2. Forward-mode autodiff

To compute $\frac{\partial f}{\partial y}(3, 4)$ we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + $ □.

So forward-mode autodiff is much more accurate than numerical differentiation, but it suffers from the same major flaw: if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs) to compute all the partial derivatives. Figure D-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7: $f(3,4) = n_7 = 42$.

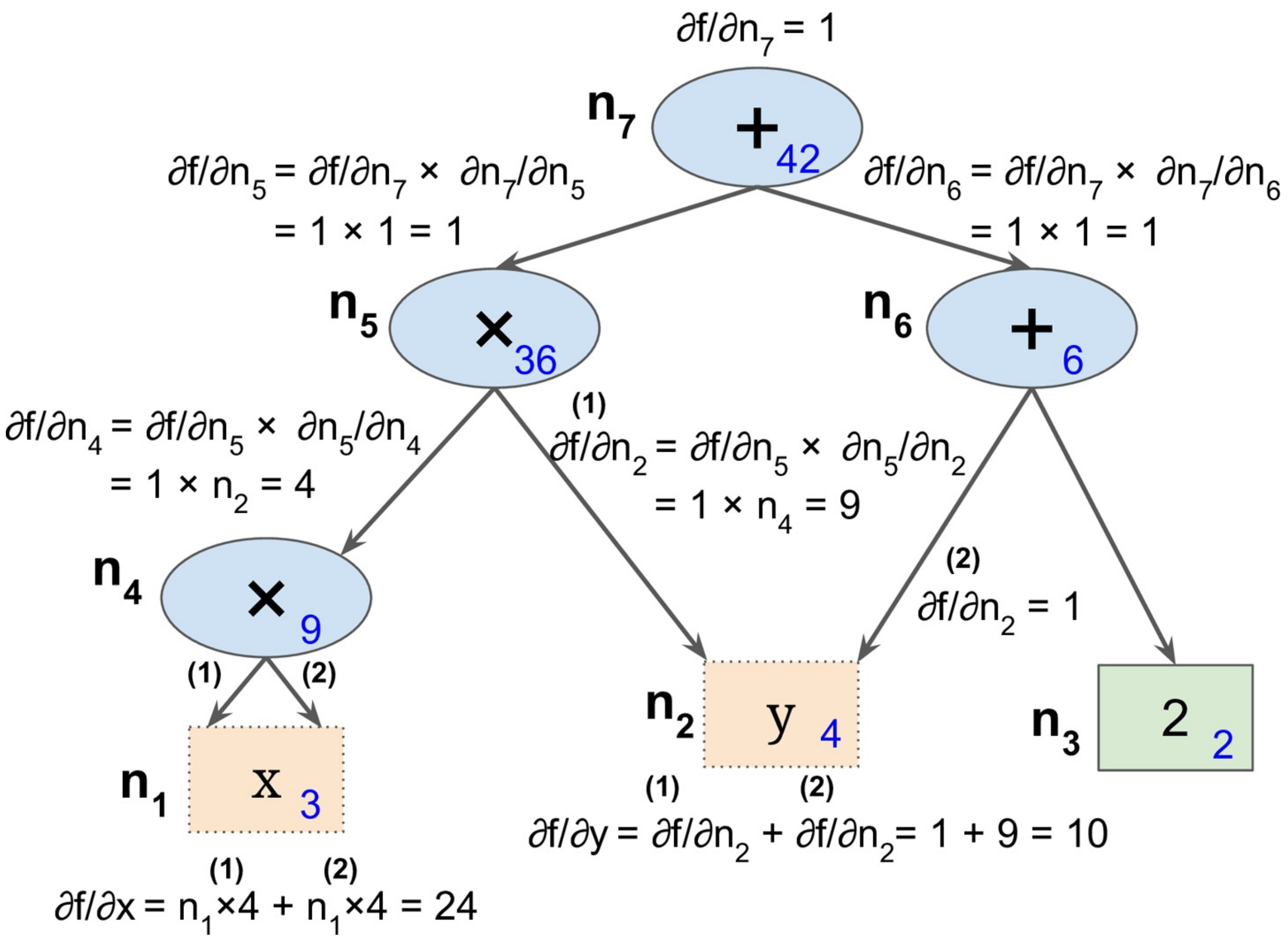


Figure D-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of *f*(*x*,*y*) with regards to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown in Equation D-4.

Equation D-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$ so trivially $\frac{\partial f}{\partial n_7} = 1$.

Let's continue down the graph to n_5: how much does *f* vary when n_5 varies? The answer is . $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$
We already know that $\frac{\partial f}{\partial n_7} = 1$, so all we need is $\frac{\partial n_7}{\partial n_5}$. Since n_7 simply performs the sum n_5

+ n_6, we find that $\frac{\partial n_7}{\partial n_5} = 1$, so .

$$\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$$

Now we can proceed to node n_4: how much does f vary when n_4 varies? The answer is .

Since $n_5 = n_4 \times n_2$, we find that $\frac{\partial n_5}{\partial n_4} = n_2$, so .

$$\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$$

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x,y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\frac{\partial f}{\partial x} = 24$ and $\frac{\partial f}{\partial y} = 10$. Sounds about right!

$$\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$$

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regards to all the inputs. Most importantly, it can deal with functions defined by arbitrary code. It can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that are differentiable.

Tip

If you implement a new type of operation in TensorFlow and you want to make it compatible with autodiff, then you need to provide a function that builds a subgraph to compute its partial derivatives with regards to its inputs. For example, suppose you implement a function that computes the square of its input $f(x) = x^2$. In that case you would need to provide the corresponding derivative function $f'(x) = 2x$. Note that this function does not compute a numerical result, but instead builds a subgraph that will (later) compute the result. This is very useful because it means that you can compute gradients of gradients (to compute second-order derivatives, or even higher-order derivatives).