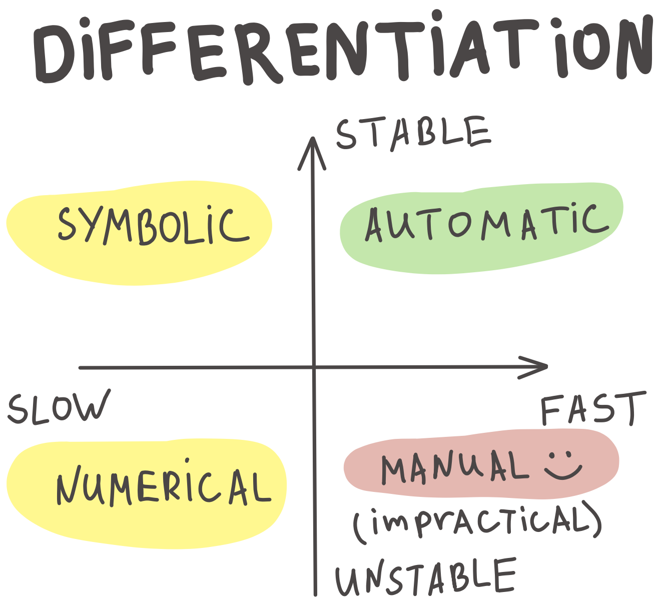
Automatic differentiation

Idea



Automatic differentiation is a scheme, that allows you to compute a value of the gradient of function with a cost of computing function itself only twice.

Chain rule

We will illustrate some important matrix calculus facts for specific cases

Univariate chain rule

Suppose, we have the following functions $R : \mathbb{R} \to \mathbb{R}, L : \mathbb{R} \to \mathbb{R}$ and $W \in \mathbb{R}$. Then

$$\frac{\partial R}{\partial W} = \frac{\partial R}{\partial L} \frac{\partial L}{\partial W}$$

Multivariate chain rule

The simplest example:

$$\frac{\partial}{\partial t} f(x_1(t), x_2(t)) = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t}$$

Now, we'll consider $f : \mathbb{R}^n \to \mathbb{R}$:

$$\frac{\partial}{\partial t} f(x_1(t), \ldots, x_n(t)) = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \ldots + \frac{\partial f}{\partial x_n}\frac{\partial x_n}{\partial t}$$

But if we will add another dimension $f : \mathbb{R}^n \to \mathbb{R}^m$, then the j-th output of f will be:

$$\frac{\partial}{\partial t} f_j(x_1(t), \ldots, x_n(t)) = \sum_{i=1}^{n} \frac{\partial f_j}{\partial x_i}\frac{\partial x_i}{\partial t} = \sum_{i=1}^{n} J_{ji}\frac{\partial x_i}{\partial t},$$

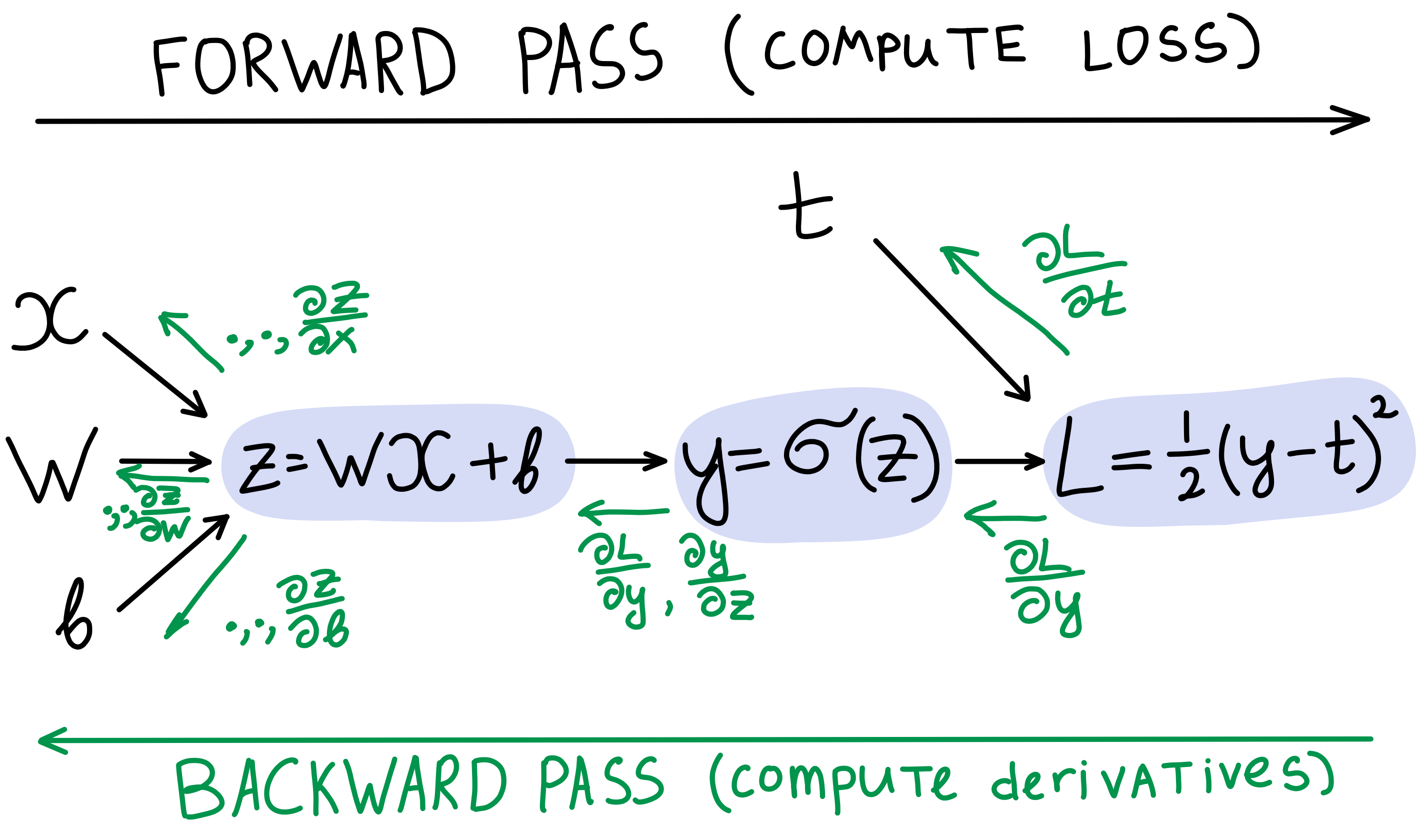
where matrix $J \in \mathbb{R}^{m \times n}$ is the jacobian of the f. Hence, we could write it in a vector way:

$$\frac{\partial f}{\partial t} = J^\top \frac{\partial x}{\partial t} \qquad \Longleftrightarrow \qquad \left(\frac{\partial f}{\partial t}\right)^\top = \left(\frac{\partial x}{\partial t}\right)^\top J$$

Backpropagation

The whole idea came from the applying chain rule to the computation graph of primitive operations

$$L = L\left(y\left(z(w, x, b)\right), t\right)$$



$$\begin{aligned}
&z = wx + b && \frac{\partial z}{\partial w} = x, \frac{\partial z}{\partial x} = w, \frac{\partial z}{\partial b} = 0 \\
&y = \sigma(z) && \frac{\partial y}{\partial z} = \sigma'(z) \\
&L = \frac{1}{2}(y-t)^2 && \frac{\partial L}{\partial y} = y - t, \frac{\partial L}{\partial t} = t - y
\end{aligned}$$

All frameworks for automatic differentiation construct (implicitly or explicitly) computation graph. In deep learning, we typically want to compute the derivatives of the loss function L w.r.t. each intermediate parameter to tune them via gradient descent. For this purpose it is convenient to use the following notation:

$$\overline{v_i} = \frac{\partial L}{\partial v_i}$$

Let $v_1, ..., v_N$ be a topological ordering of the computation graph (i.e. parents come before children). v_N denotes the variable we're trying to compute derivatives of (e.g. loss).

Forward pass:

- For $i = 1, \dots, N$:
    - Compute v_i as a function of its parents.

Backward pass:

- $$\overline{v_N} = 1$$
- For $i = N-1, \dots, 1$:
    - Compute derivatives $\overline{v_i} = \sum\limits_{j \in \text{Children}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$

Note, that $\overline{v_j}$ term is coming from the children of $\overline{v_i}$, while $\frac{\partial v_j}{\partial v_i}$ is already precomputed effectively.

Jacobian vector product

The reason why it works so fast in practice is that the Jacobian of the operations is already developed effectively in automatic differentiation frameworks. Typically, we even do not construct or store the full Jacobian, doing matvec directly instead.

Example: element-wise exponent

$$y = \exp(z) \qquad J = \text{diag}(\exp(z)) \qquad \overline{z} = \overline{y} J$$

See the examples of Vector-Jacobian Products from autodidact library:

```
defvjp(anp.add,          lambda g, ans, x, y : unbroadcast(x, g),
                         lambda g, ans, x, y : unbroadcast(y, g))
defvjp(anp.multiply,     lambda g, ans, x, y : unbroadcast(x, y * g),
                         lambda g, ans, x, y : unbroadcast(y, x * g))
defvjp(anp.subtract,     lambda g, ans, x, y : unbroadcast(x, g),
                         lambda g, ans, x, y : unbroadcast(y, -g))
defvjp(anp.divide,       lambda g, ans, x, y : unbroadcast(x,   g / y),
                         lambda g, ans, x, y : unbroadcast(y, - g * x / y**2))
defvjp(anp.true_divide,  lambda g, ans, x, y : unbroadcast(x,   g / y),
                         lambda g, ans, x, y : unbroadcast(y, - g * x / y**2))
```

Hessian vector product

Interesting, that a similar idea could be used to compute Hessian-vector products, which is essential for second order optimization or conjugate gradient methods. For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ with continuous second derivatives (so that the Hessian matrix is symmetric), the Hessian at a point $x \in \mathbb{R}^n$ is written as $\partial^2 f(x)$. A Hessian-vector product function is then able to evaluate

$$v \mapsto \partial^2 f(x) \cdot v$$

for any vector $v \in \mathbb{R}^n$.

The trick is not to instantiate the full Hessian matrix: if n is large, perhaps in the millions or billions in the context of neural networks, then that might be impossible to store. Luckily, `grad` (in the jax/autograd/pytorch/tensorflow) already gives us a way to write an efficient Hessian-vector product function. We just have to use the identity

$$\partial^2 f(x)v = \partial[x \mapsto \partial f(x) \cdot v] = \partial g(x),$$

where $g(x) = \partial f(x) \cdot v$ is a new scalar-valued function that dots the gradient of f at x with the vector v. Notice that we're only ever differentiating scalar-valued functions of vector-valued arguments, which is exactly where we know `grad` is efficient.

```
import jax.numpy as jnp

def hvp(f, x, v):
    return grad(lambda x: jnp.vdot(grad(f)(x), v))(x)
```

Code

Open in Colab

Materials

- Autodidact - a pedagogical implementation of Autograd

- CSC321 Lecture 6
- CSC321 Lecture 10
- Why you should understand backpropagation 😃
- JAX autodiff cookbook