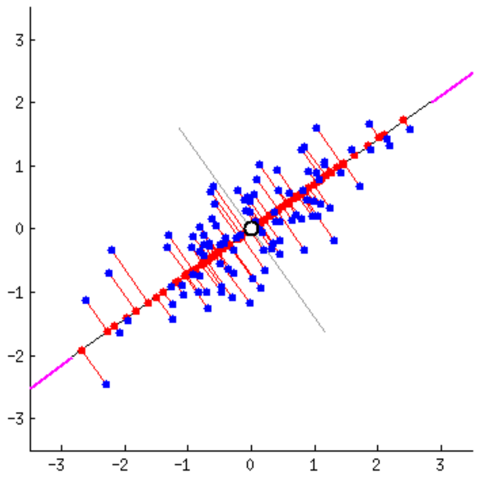
Intuition

Imagine, that you have a dataset of points. Your goal is to choose orthogonal axes, that describe your data the most informative way. To be precise, we choose first axis in such a way, that maximize the variance (expressiveness) of the projected data. All the following axes have to be orthogonal to the previously chosen ones, while satisfy largest possible variance of the projections.

Let's take a look at the simple 2d data. We have a set of blue points on the plane. We can easily see that the projections on the first axis (red dots) have maximum variance at the final position of the animation. The second (and the last) axis should be orthogonal to the previous one.



source

This idea could be used in a variety of ways. For example, it might happen, that projection of complex data on the principal plane (only 2 components) bring you enough intuition for clustering. The picture below plots projection of the labeled dataset onto the first to principal components (PCs), we can clearly see, that only two vectors (these PCs) would be enogh to differ Finnish people from Italian in particular dataset (celiac disease (Dubois et al. 2010)) source

Problem

The first component should be defined in order to maximize variance. Suppose, we've already normalized the data, i.e. $\sum\limits_i a_i = 0$, then sample variance will become the sum of all squared projections of data points to our vector $\mathbf{w}_{(1)}$, which implies the following optimization problem:

$$\mathbf{w}_{(1)} = \underset{\|\mathbf{w}\|=1}{\arg\max} \left\{ \sum_i \left(\mathbf{a}_{(i)}^\top \cdot \mathbf{w} \right)^2 \right\}$$

or

$$\mathbf{w}_{(1)} = \underset{\|\mathbf{w}\|=1}{\arg\max} \{ \|\mathbf{A}\mathbf{w}\|^2 \} = \underset{\|\mathbf{w}\|=1}{\arg\max} \left\{ \mathbf{w}^\top \mathbf{A}^\top \mathbf{A}\mathbf{w} \right\}$$

since we are looking for the unit vector, we can reformulate the problem:

$$\mathbf{w}_{(1)} = \arg\max \left\{ \frac{\mathbf{w}^\top \mathbf{A}^\top \mathbf{A}\mathbf{w}}{\mathbf{w}^\top \mathbf{w}} \right\}$$

It is known, that for positive semidefinite matrix $A^\top A$ such vector is nothing else, but eigenvector of $A^\top A$, which corresponds to the largest eigenvalue. The following components will give you the same results (eigenvectors).

So, we can conclude, that the following mapping:

$$\underset{n\times k}{\Pi} = \underset{n\times d}{A} \cdot \underset{d\times k}{W}$$

describes the projection of data onto the k principal components, where W contains first (by the size of eigenvalues) k eigenvectors of $A^\top A$.

Now we'll briefly derive how SVD decomposition could lead us to the PCA.

Firstly, we write down SVD decomposition of our matrix:

$$A = U\Sigma W^\top$$

and to its transpose:

$$\begin{aligned} A^\top &= (U\Sigma W^\top)^\top \\ &= (W^\top)^\top \Sigma^\top U^\top \\ &= W\Sigma^\top U^\top \\ &= W\Sigma U^\top \end{aligned}$$

Then, consider matrix $AA^\top$:

$$
\begin{aligned}
A^\top A &= (W\Sigma U^\top)(U\Sigma V^\top) \\
&= W\Sigma I \Sigma W^\top \\
&= W\Sigma\Sigma W^\top \\
&= W\Sigma^2 W^\top
\end{aligned}
$$

Which corresponds to the eigendecomposition of matrix $A^\top A$, where W stands for the matrix of eigenvectors of $A^\top A$, while Σ^2 contains eigenvalues of $A^\top A$.

At the end:

$$
\begin{aligned}
\Pi &= A \cdot W = \\
&= U\Sigma W^\top W = U\Sigma
\end{aligned}
$$

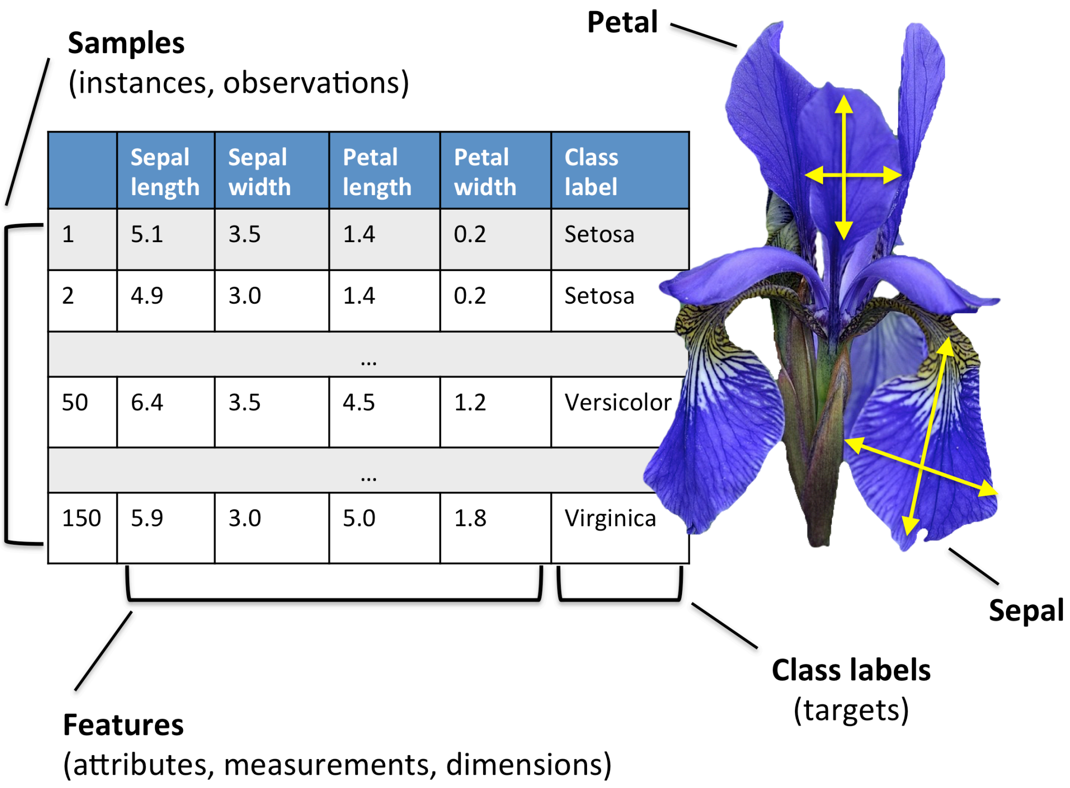
The latter formula provide us with easy way to compute PCA via SVD with any number of principal components:

$$
\Pi_r = U_r \Sigma_r
$$

Examples

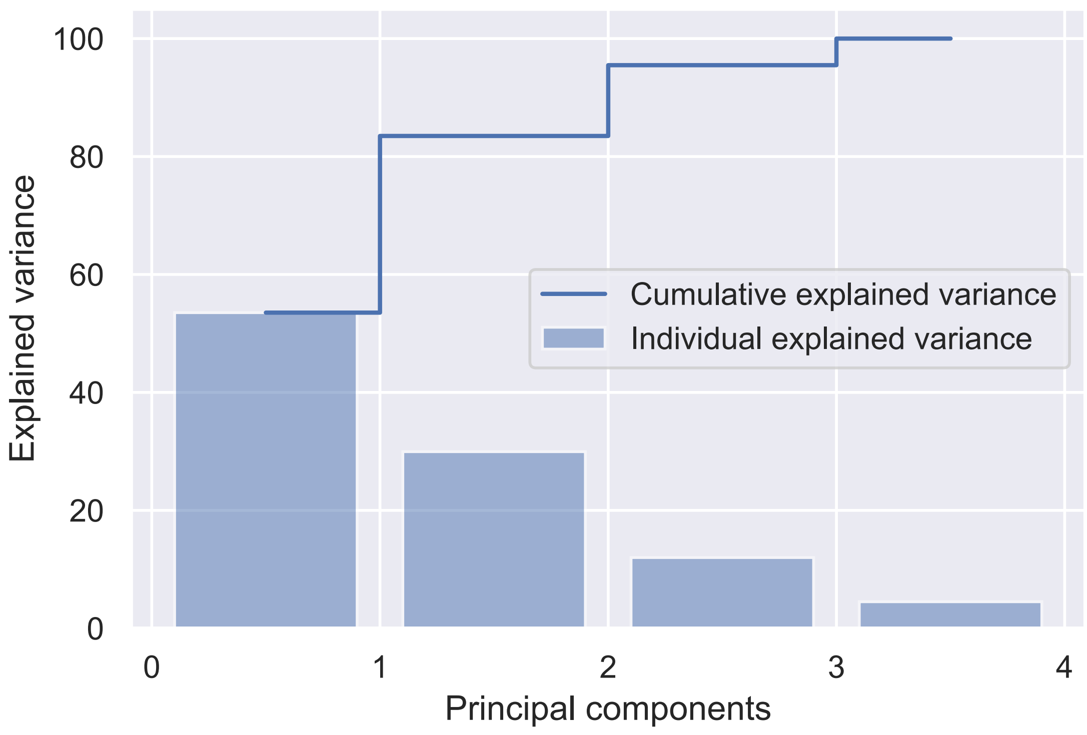
🌼 Iris dataset

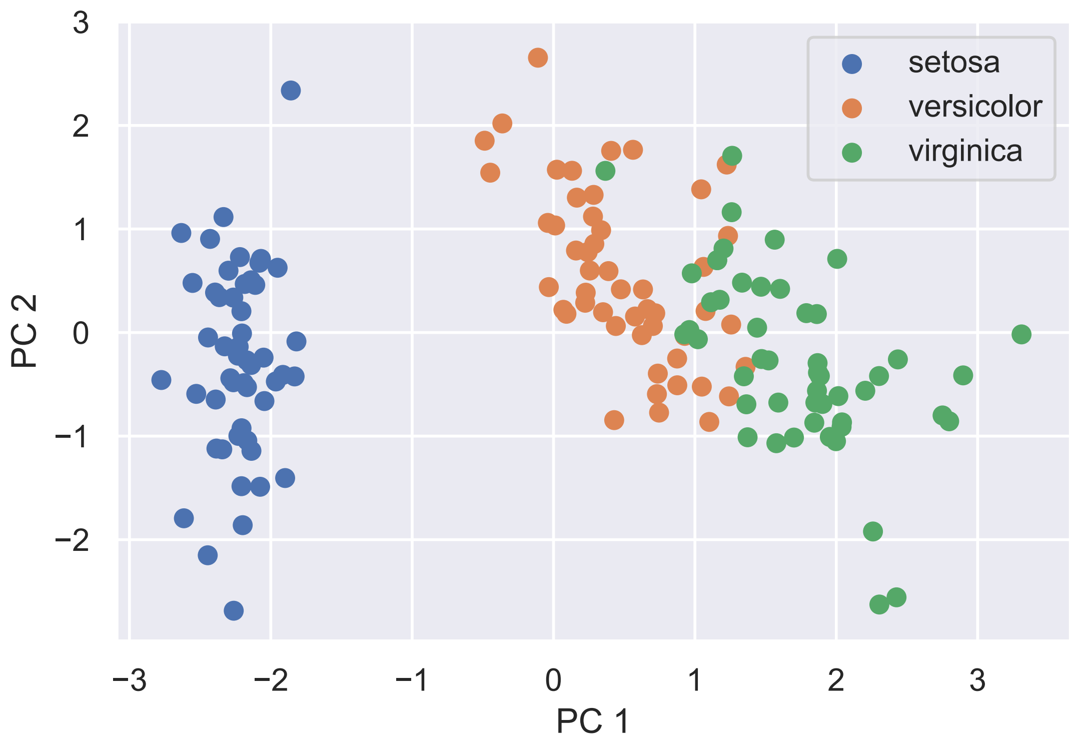
Consider the classical Iris dataset


Samples
(instances, observations)
Petal
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

source We have the dataset matrix $A \in \mathbb{R}^{150 \times 4}$





Code

Open in Colab

Related materials

- Wikipedia
- Blog post
- Blog post

Useful definitions and notations

We will treat all vectors as column vectors by default. The space of real vectors of length n is denoted by $\mathbb{R}^n$, while the space of real-valued $m \times n$ matrices is denoted by $\mathbb{R}^{m \times n}$.

Basic linear algebra background

The standard **inner product** between vectors x and y from $\mathbb{R}^n$ is given by

$$\langle x, y \rangle = x^\top y = \sum_{i=1}^n x_i y_i = y^\top x = \langle y, x \rangle$$

Here x_i and y_i are the scalar i-th components of corresponding vectors.

The standard **inner product** between matrices X and Y from $\mathbb{R}^{m \times n}$ is given by

$$\langle X, Y \rangle = \text{tr}(X^\top Y) = \sum_{i=1}^m \sum_{j=1}^n X_{ij} Y_{ij} = \text{tr}(Y^\top X) = \langle Y, X \rangle$$

The determinant and trace can be expressed in terms of the eigenvalues

$$\det A = \prod_{i=1}^n \lambda_i, \qquad \text{tr} A = \sum_{i=1}^n \lambda_i$$

Don't forget about the cyclic property of a trace for a square matrices A, B, C, D:

$$\text{tr}(ABCD) = \text{tr}(DABC) = \text{tr}(CDAB) = \text{tr}(BCDA)$$

The largest and smallest eigenvalues satisfy

$$\lambda_{\text{min}}(A) = \inf_{x \neq 0} \frac{x^\top A x}{x^\top x}, \qquad \lambda_{\text{max}}(A) = \sup_{x \neq 0} \frac{x^\top A x}{x^\top x}$$

and consequently $\forall x \in \mathbb{R}^n$ (Rayleigh quotient):

$$\lambda_{\text{min}}(A) x^\top x \leq x^\top A x \leq \lambda_{\text{max}}(A) x^\top x$$

A matrix $A \in \mathbb{S}^n$ (set of square symmetric matrices of dimension n) is called **positive (semi)definite** if for all $x \neq 0 (\text{for all } x) : x^\top A x > (\geq) 0$. We denote this as

$A \succ (\succeq) 0.$

The **condition number** of a nonsingular matrix is defined as

$$\kappa(A) = \|A\|\|A^{-1}\|$$

Matrix and vector multiplication

Let A be a matrix of size $m \times n$, and B be a matrix of size $n \times p$, and let the product AB be:

$$C = AB$$

then C is a $m \times p$ matrix, with element (i, j) given by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Let A be a matrix of shape $m \times n$, and x be $n \times 1$ vector, then the i-th component of the product:

$$z = Ax$$

is given by:

$$z_i = \sum_{k=1}^{n} a_{ik} x_k$$

Finally, just to remind:

- $C = AB \quad C^\top = B^\top A^\top$
- $AB \neq BA$
- $e^A = \sum\limits_{k=0}^{\infty} \frac{1}{k!} A^k$
- $e^{A+B} \neq e^A e^B$ (but if A and B are commuting matrices, which means that $AB = BA$, $e^{A+B} = e^A e^B$)
- $\langle x, Ay \rangle = \langle A^\top x, y \rangle$

Gradient

Let $f(x) : \mathbb{R}^n \to \mathbb{R}$, then vector, which contains all first order partial derivatives:

$$\nabla f(x) = \frac{df}{dx} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

named gradient of $f(x)$. This vector indicates the direction of steepest ascent. Thus, vector $-\nabla f(x)$ means the direction of the steepest descent of the function in the point. Moreover, the gradient vector is always orthogonal to the contour line in the point.

Hessian

Let $f(x) : \mathbb{R}^n \to \mathbb{R}$, then matrix, containing all the second order partial derivatives:

$$f''(x) = \frac{\partial^2 f}{\partial x_i \partial x_j} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

In fact, Hessian could be a tensor in such a way: $\left(f(x) : \mathbb{R}^n \to \mathbb{R}^m\right)$ is just 3d tensor, every slice is just hessian of corresponding scalar function $\left(H\left(f_1(x)\right), H\left(f_2(x)\right), \ldots, H\left(f_m(x)\right)\right)$.

Jacobian

The extension of the gradient of multidimensional $f(x) : \mathbb{R}^n \to \mathbb{R}^m$ is the following matrix:

$$f'(x) = \frac{df}{dx^T} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Summary

$$f(x) : X \to Y; \qquad \frac{\partial f(x)}{\partial x} \in G$$

X	Y	G	Name
$\mathbb{R}$	$\mathbb{R}$	$\mathbb{R}$	$f'(x)$ (derivative)
$\mathbb{R}^n$	$\mathbb{R}$	$\mathbb{R}^n$	$\dfrac{\partial f}{\partial x_i}$ (gradient)
$\mathbb{R}^n$	$\mathbb{R}^m$	$\mathbb{R}^{m \times n}$	$\dfrac{\partial f_i}{\partial x_j}$ (jacobian)
$\mathbb{R}^{m \times n}$	$\mathbb{R}$	$\mathbb{R}^{m \times n}$	$\dfrac{\partial f}{\partial x_{ij}}$

General concept

Naive approach

The basic idea of naive approach is to reduce matrix/vector derivatives to the well-known scalar derivatives.

$$\begin{tikzcd}
\text{Matrix notation of a function} & & \text{Matrix notation of a gradient} \\
f(x) = c^\top x \arrow[d] & & \nabla f(x) = c \\
\text{Scalar notation of a function} & & \\
f(x) = \sum\limits_{i=1}^{n} c_i x_i \arrow[dr] & & \dfrac{\partial f(x)}{\partial x_k} = c_k \arrow[uu] \\
& \text{Simple derivative} \arrow[ur] &
\end{tikzcd}$$

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial \left(\sum_{i=1}^{n} c_i x_i \right)}{\partial x_k}$$

One of the most important practical tricks here is to separate indices of sum (i) and partial derivatives (k). Ignoring this simple rule tends to produce mistakes.

Differential approach

The guru approach implies formulating a set of simple rules, which allows you to calculate derivatives just like in a scalar case. It might be convenient to use the differential notation here.

Differentials

After obtaining the differential notation of df we can retrieve the gradient using following formula:

$$df(x) = \langle \nabla f(x), dx \rangle$$

Then, if we have differential of the above form and we need to calculate the second derivative of the matrix/vector function, we treat "old" dx as the constant dx_1, then calculate $d(df) = d^2 f(x)$

$$d^2 f(x) = \langle \nabla^2 f(x) dx_1, dx \rangle = \langle H_f(x) dx_1, dx \rangle$$

Properties

Let A and B be the constant matrices, while X and Y are the variables (or matrix functions).

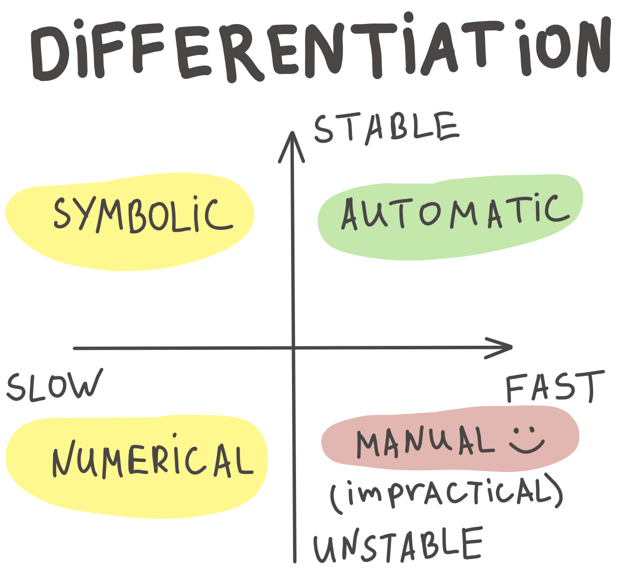
- $dA = 0$
- $d(\alpha X) = \alpha(dX)$
- $d(AXB) = A(dX)B$
- $d(X + Y) = dX + dY$
- $d(X^\top) = (dX)^\top$
- $d(XY) = (dX)Y + X(dY)$
- $d\langle X, Y \rangle = \langle dX, Y \rangle + \langle X, dY \rangle$
- $d\left(\dfrac{X}{\phi}\right) = \dfrac{\phi dX - (d\phi)X}{\phi^2}$
- $d\left(\det X\right) = \det X \langle X^{-\top}, dX \rangle$
- $d\left(\text{tr } X\right) = \langle I, dX \rangle$
- $df(g(x)) = \dfrac{df}{dg} \cdot dg(x)$
- $H = (J(\nabla f))^T$

- $d(X^{-1}) = -X^{-1}(dX)X^{-1}$

References

- Convex Optimization book by S. Boyd and L. Vandenberghe - Appendix A. Mathematical background.
- Numerical Optimization by J. Nocedal and S. J. Wright. - Background Material.
- Matrix decompositions Cheat Sheet.
- Good introduction
- The Matrix Cookbook
- MSU seminars (Rus.)
- Online tool for analytic expression of a derivative.
- Determinant derivative

Idea



Automatic differentiation is a scheme, that allows you to compute a value of 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 $f : \mathbb{R}^n \to \mathbb{R}^m$, than the j-th output of f will be:

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

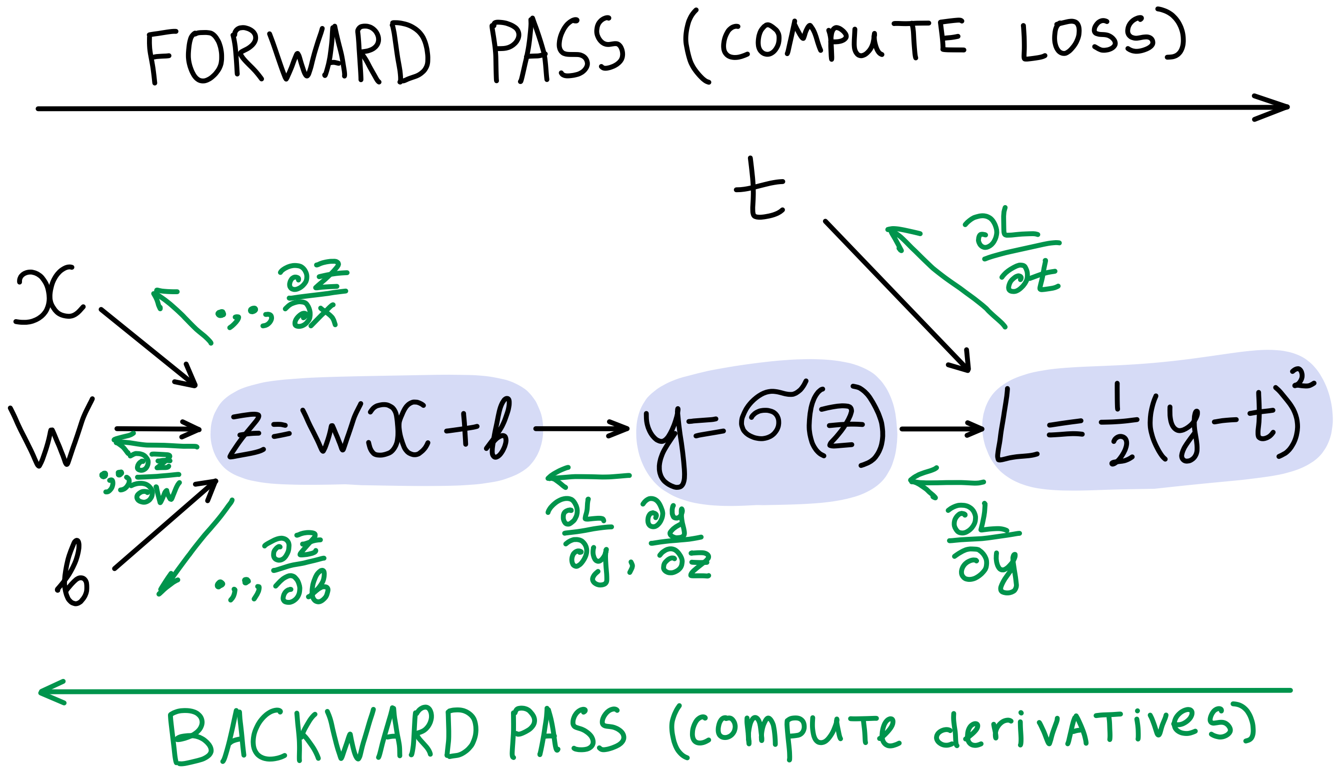
where matrix $J \in \mathbb{R}^{m \times n}$ is the jacobian of the f. Hence, we could write it in a vector way:

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

Backpropagation

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

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



FORWARD PASS (COMPUTE LOSS)
t
$\frac{\partial L}{\partial t}$
x
$\ldots, \frac{\partial z}{\partial x}$
W
$\ldots, \frac{\partial z}{\partial w}$
b
$\ldots, \frac{\partial z}{\partial b}$
$z = Wx + b$
$y = \sigma(z)$
$L = \frac{1}{2}(y-t)^2$
$\frac{\partial L}{\partial y}, \frac{\partial y}{\partial z}$
$\frac{\partial L}{\partial y}$
BACKWARD PASS (compute derivatives)

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

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 parameters in order 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 $v_1, \ldots, v_N$ 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 $i = 1, \ldots, N$:
  - Compute v_i as a function of its parents.

Backward pass:

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

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.

Univariate logistic least squares regression

$$\begin{tikzcd}
x \arrow[dr] & & t \arrow[dr] & & \\
b \arrow[r] & z = wx + b \arrow[r] & y = \sigma(z) \arrow[r] & L = \frac{1}{2}(y-t)^2 \arrow[r] & \mathcal{L} = L + \lambda R \\
w \arrow[ur] \arrow[rrr] & & & R = \frac{1}{2}w^2 \arrow[ur] &
\end{tikzcd}$$

Forward pass

$$z = wx + b$$
$$y = \sigma(z)$$
$$L = \frac{1}{2}(y-t)^2$$
$$R = \frac{1}{2}w^2$$
$$\mathcal{L} = L + \lambda R$$

Backward pass

$$\overline{\mathcal{L}} = 1$$
$$\overline{R} = \overline{\mathcal{L}}\frac{d\mathcal{L}}{dR} = \overline{\mathcal{L}}\lambda$$
$$\overline{L} = \overline{\mathcal{L}}\frac{d\mathcal{L}}{dL} = \overline{\mathcal{L}}$$
$$\overline{y} = \overline{L}\frac{dL}{dy} = \overline{L}(y-t)$$

$$\overline{z} = \overline{y}\frac{dy}{dz} = \overline{y}\sigma'(z)$$
$$\overline{w} = \overline{z}\frac{dz}{dw} + \overline{R}\frac{dR}{dw} = \overline{z}x + \overline{R}w$$
$$\overline{b} = \overline{z}\frac{dz}{db} = \overline{z}$$
$$\overline{x} = \overline{z}\frac{dz}{dx} = \overline{z}w$$

Jacobian vector product

The reason why it works so fast in practice is that the Jacobian of the operations are already developed in effective manner 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 the 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 vector-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
- Materials from CS207: Systems Development for Computational Science course with very intuitive explanation.