Politecnico di Milano
School of Industrial and Information Engineering
Academic Year 2024-2025

Numerical Analysis for Machine Learning

Professor: Edie Miglio

Last updated: December 2, 2024

This document is intended for educational purposes only.
These are unreviewed notes and may contain errors.

Made by Roberto Benatuil Valera

Contents

1 Numerical Linear Algebra tools **5**

1.1 Introduction: Recap of Linear Algebra . . . 5
1.1.1 Matrix-vector multiplication . . . 5
1.1.2 Column space of a matrix . . . 5
1.1.3 System of linear equations . . . 6
1.1.4 CR factorization . . . 6
1.1.5 Matrix-matrix multiplication . . . 6
1.1.6 Null space of a matrix . . . 7
1.1.7 Fundamental subspaces of a matrix . . . 7
1.1.8 Orthogonal matrices . . . 7
1.1.9 QR factorization . . . 8
1.1.10 Eigenvalues and eigenvectors . . . 8
1.1.11 Similar matrices . . . 9
1.2 Power method . . . 9
1.2.1 Rayleigh quotient . . . 10
1.2.2 Proof of convergence for the power method . . . 10
1.2.3 Inverse power method . . . 11
1.2.4 Shifted inverse power method . . . 11
1.2.5 Applications: Page Rank . . . 11
1.3 Symmetric matrices . . . 13
1.3.1 Symmetric positive definite matrices . . . 14
1.4 Singular Value Decomposition (SVD) . . . 14
1.4.1 Economy SVD . . . 15
1.4.2 Low-rank approximation . . . 15
1.4.3 Existance of the SVD . . . 17
1.4.4 Computation of the SVD . . . 18
1.4.5 Geometrical interpretation of the SVD . . . 19
1.4.6 Polar decomposition . . . 19
1.4.7 Eckart-Young theorem . . . 19
1.4.8 Principal Component Analysis (PCA) . . . 21
1.4.9 Data pre-processing: sensitivity to transformations . . . 22
1.4.10 Randomized SVD . . . 22
1.5 Matrix completion . . . 23
1.6 Regression Methods . . . 24

1.6.1 Least Squares . . . 24
1.6.2 Ridge regression . . . 26
1.6.3 LASSO regression . . . 28
1.6.4 Kernel methods . . . 28
1.6.5 Support Vector Regression (SVR) . . . 30

2 Automatic differentiation **31**
2.1 Introduction: differentiation methods . . . 31
2.2 Automatic differentiation: Wengert list . . . 33
2.3 Forward mode of AD . . . 34
2.3.1 Note: Forward mode and Jacobian-vector product (JVP) . . . 36
2.4 Backward mode of AD . . . 36
2.4.1 Note: Backward mode and vector-Jacobian product (VJP) . . . 38
2.5 Dual numbers . . . 38

3 Numerical Optimization and Training of Neural Networks **41**
3.1 Introduction: the Perceptron . . . 41
3.1.1 The XOR problem . . . 41
3.1.2 NAND gate . . . 42
3.1.3 The problem of the step function . . . 43
3.2 The Multi-Layer Perceptron . . . 43
3.3 Training a Neural Network . . . 44
3.3.1 Backpropagation equations . . . 45
3.4 Activation functions . . . 49

Chapter 1

Numerical Linear Algebra tools

1.1 Introduction: Recap of Linear Algebra

In this section we will review some basic concepts of Linear Algebra that will be useful for the rest of the course.

1.1.1 Matrix-vector multiplication

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^n$, the matrix-vector multiplication $y = Ax$ is defined as:

$$y_i = \sum_{j=1}^{n} A_{ij} x_j \tag{1.1}$$

A matrix-vector multiplication can be considered as a linear combination of the columns of the matrix A. Lets see an example:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} x_1 + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} x_2 \tag{1.2}$$

1.1.2 Column space of a matrix

The column space of a matrix $A \in \mathbb{R}^{m \times n}$ is the subspace of $\mathbb{R}^m$ spanned by the columns of A. In other words, it is the set of all possible linear combinations of the columns of A. The column space of a matrix is denoted as $C(A)$.

If the columns of A are linearly independent, then the column space of A is the entire $\mathbb{R}^m$. If the columns of A are linearly dependent, then the column space of A is a subspace of $\mathbb{R}^m$ with dimension equal to the rank of A.

The rank of a matrix A is is the size of the largest set of linearly independent columns of A. It is denoted as $rank(A)$. Note that $rank(A) = rank(A^T)$.

1.1.3 System of linear equations

A system of linear equations is a set of m equations with n unknowns of the form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned} \tag{1.3}$$

This system can be written in matrix form as $Ax = b$, where $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$.

The system $Ax = b$ has a solution if and only if $b \in C(A)$. If $b \in C(A)$, then the system has a unique solution if and only if $rank(A) = n$. If $rank(A) < n$, then the system has infinitely many solutions.

1.1.4 CR factorization

The CR factorization of a matrix $A \in \mathbb{R}^{m\times n}$, with $m \geq n$, is a factorization of A as $A = CR$, where $C \in \mathbb{R}^{m\times r}$ is a matrix with the linearly independent columns of A and $R \in \mathbb{R}^{r\times n}$ is obtained by determining the coefficients of the linear combination of the columns of C that give the columns of A. In this factorization, $r = rank(A)$.

Lets see an example:

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \end{bmatrix} = CR \tag{1.4}$$

The matrix C is also called the Row Reduced Echelon Form of A.

1.1.5 Matrix-matrix multiplication

Given two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, the matrix-matrix multiplication $C = AB$ is defined as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} \tag{1.5}$$

A matrix-matrix multiplication can be considered as the outer product of the columns of A and the rows of B. Lets see an example:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \begin{bmatrix} 3 & 4 \end{bmatrix} \tag{1.6}$$

Note that each outer product generates a matrix of the same size as the result matrix, but always with rank 1. So the matrix-matrix multiplication can be considered as a sum of rank 1 matrices, obtained by the outer products of the columns of A and the rows of B.

1.1.6 Null space of a matrix

The null space of a matrix $A \in \mathbb{R}^{m \times n}$ is the set of all vectors $x \in \mathbb{R}^n$ such that $Ax = 0$. The null space of a matrix is denoted as $N(A)$. It is also called the kernel of A, denoted as $ker(A)$.

Formally, we have that:

$$N(A) = \{x \in \mathbb{R}^n : Ax = 0\} \tag{1.7}$$

The null space of a matrix is a subspace of $\mathbb{R}^n$. The dimension of the null space of a matrix is called the nullity of the matrix.

1.1.7 Fundamental subspaces of a matrix

Given a matrix $A \in \mathbb{R}^{m \times n}$, we can define four fundamental subspaces:

- The column space of A, denoted as $C(A)$
- The row space of A, denoted as $C(A^T)$
- The null space of A, denoted as $N(A)$
- The left null space of A, denoted as $N(A^T)$

These subspaces are related by the following properties:

$$\begin{gathered} C(A) \perp N(A^T) \\ C(A^T) \perp N(A) \end{gathered} \tag{1.8}$$

They also satisfy the following dimensions properties:

$$\begin{gathered} dim(C(A)) + dim(N(A)) = n \\ dim(C(A^T)) + dim(N(A^T)) = m \end{gathered} \tag{1.9}$$

This is known as the Rank-Nullity Theorem.

1.1.8 Orthogonal matrices

An orthogonal matrix is a square matrix $Q \in \mathbb{R}^{n \times n}$ such that $Q^T Q = I$, where I is the identity matrix. This implies that $Q^T = Q^{-1}$.

Now, consider that Q is an orthogonal matrix, and set $w = Q^T x$. Then we have that:

$$\begin{aligned} \|w\|^2 = w^T w &= x^T Q Q^T x \\ &= x^T x = \|x\|^2 \end{aligned} \tag{1.10}$$

This means that the norm of a vector is preserved under an orthogonal transformation. This is called an isometry. It is a useful property for numerical algorithms, as it helps to avoid numerical instability.

There are two main types of orthogonal transformations that we are interested:

Rotation matrices

A rotation matrix is an orthogonal matrix that represents a rotation in $\mathbb{R}^2$ or $\mathbb{R}^3$. In $\mathbb{R}^2$, a rotation matrix is of the form:

$$Q(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{1.11}$$

Reflection matrices

A reflection matrix is an orthogonal matrix that represents a reflection with respect to a hyperplane. If n denotes the unit normal vector to the hyperplane, then the reflection matrix is of the form:

$$Q = I - 2nn^T \tag{1.12}$$

Note that the inverse of this matrix is itself, as $Q^T = Q^{-1}$ and in this case, Q is symmetric ($Q = Q^T$).

1.1.9 QR factorization

The QR factorization of a matrix $A \in \mathbb{R}^{m \times n}$, with $m \geq n$, is a factorization of A as $A = QR$, where $Q \in \mathbb{R}^{m \times n}$ is an orthogonal matrix and $R \in \mathbb{R}^{n \times n}$ is an upper triangular matrix.

Gram-Schmidt process

The Gram-Schmidt process is a method to compute the QR factorization of a matrix. Given a matrix $A \in \mathbb{R}^{m \times n}$, the Gram-Schmidt process computes an orthonormal basis for the column space of A, as follows:

$$\begin{aligned} q_1 &= \frac{a_1}{\|a_1\|} \\ q_i &= a_i - \sum_{j=1}^{i-1} (q_j^T a_i) q_j \quad \forall i = 2, \ldots, n \end{aligned} \tag{1.13}$$

where a_i denotes the i-th column of A. The matrix Q is obtained by stacking the vectors q_i as columns. The matrix R is obtained by computing the coefficients of the linear combination of the columns of Q that give the columns of A.

1.1.10 Eigenvalues and eigenvectors

Given a square matrix $A \in \mathbb{R}^{n \times n}$, a scalar λ is called an eigenvalue of A if there exists a vector $v \in \mathbb{R}^n$ such that:

$$Av = \lambda v \tag{1.14}$$

The vector v is called an eigenvector of A associated with the eigenvalue λ.

Let P be the matrix whose columns are the eigenvectors of A, and Λ be the diagonal matrix whose diagonal elements are the eigenvalues of A. Then we have that:

$$A = P\Lambda P^{-1} \tag{1.15}$$

This is called the eigendecomposition of A.

The eigenvalues of a matrix are the roots of the characteristic polynomial of A, which is defined as:

$$det(A - \lambda I) = 0 \tag{1.16}$$

1.1.11 Similar matrices

Two square matrices A and B are called similar if there exists a non-singular matrix M such that:

$$B = M^{-1}AM \tag{1.17}$$

Similar matrices have the same eigenvalues, but not necessarily the same eigenvectors. Let (λ, y) be an eigenpair of B, then we have:

$$By = \lambda y \Rightarrow M^{-1}AMy = \lambda y \Rightarrow A(My) = \lambda(My) \tag{1.18}$$

This means that My is an eigenvector of A associated with the eigenvalue λ. So, to obtain the eigenvectors of A from the eigenvectors of B, we need to multiply the eigenvectors of B by M.

This property can be useful. For example, if we want to compute the eigenvalues of a matrix A, we can find some transformation M such that $M^{-1}AM = B$ is a simpler matrix to work with, usually a lower triangular matrix. Then we can compute the eigenvalues of B and obtain the eigenvalues of A. M is obtained by the permutation matrices to get from A to B. The Givens and Householder transformations are examples of such method.

1.2 Power method

The power method is an iterative algorithm to compute the dominant eigenvalue of a matrix (i.e., the eigenvalue with the largest magnitude). The algorithm is as follows:

Algorithm 1 Power method

Choose a random vector $x^{(0)}$, s.t. $\|x^{(0)}\| = 1$
for $k = 1, 2, \ldots$ **do**
    $y^{(k)} = Ax^{(k-1)}$
    $x^{(k)} = \frac{y^{(k)}}{\|y^{(k)}\|}$
    $\lambda^{(k)} = x^{(k)T}Ax^{(k)}$
end for

The convergence rate is determined by the ratio of the largest eigenvalue to the second largest eigenvalue.

1.2.1 Rayleigh quotient

The Rayleigh quotient is the base of the power method algorithm. Given a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $x \in \mathbb{R}^n$, the Rayleigh quotient is defined as:

$$R(x) = \frac{x^T A x}{x^T x} \tag{1.19}$$

For every eigenpair (λ, v) of A, we have that:

$$R(v) = \frac{v^T A v}{v^T v} = \frac{v^T \lambda v}{v^T v} = \lambda \tag{1.20}$$

This means that the Rayleigh quotient is equal to the eigenvalue associated with the eigenvector v. This property is used in the power method to compute the dominant eigenvalue of a matrix.

1.2.2 Proof of convergence for the power method

The power method converges to the dominant eigenvalue of a matrix. The sketch proof is as follows:

Let $x^{(0)}$ be our initial vector, and let $\{v_1, \ldots, v_n\}$ be the eigenvectors of A. We can write $x^{(0)}$ as a linear combination of the eigenvectors of A (since the eigenvectors of A form a basis of $\mathbb{R}^n$):

$$x^{(0)} = \sum_{i=1}^{n} \alpha_i v_i \tag{1.21}$$

Then we have that:

$$A x^{(0)} = \sum_{i=1}^{n} \alpha_i A v_i = \sum_{i=1}^{n} \alpha_i \lambda_i v_i \tag{1.22}$$

Since in every iteration k of the power method we apply the matrix A to the vector $x^{(k-1)}$, we have that:

$$A x^{(k-1)} = A^k x^{(0)} = \sum_{i=1}^{n} \alpha_i \lambda_i^k v_i \tag{1.23}$$

Now, let us factorize the previous equation by the dominant eigenvalue $(\lambda_1)^k$:

$$A^k x^{(0)} = (\lambda_1)^k \left(\alpha_1 v_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1} \right)^k v_i \right) \tag{1.24}$$

Note that the term inside the parenthesis converges to zero as $k \to \infty$, since the ratio of the other eigenvalues to the dominant eigenvalue is less than 1. This means that $x^(k)$ converges to the direction of the dominant eigenvector v_1. When it is normalized, its Rayleigh quotient converges to the dominant eigenvalue λ_1.

1.2.3 Inverse power method

The inverse power method is an iterative algorithm to compute the eigenvalue with the smalles magnitude of a matrix. Note that the smallest eigenvalue of a matrix is the largest eigenvalue of its inverse, since:

$$Ax = \lambda x \Rightarrow A^{-1}x = \frac{1}{\lambda}x \tag{1.25}$$

The algorithm is similar to the power method, but instead of applying the matrix A to the vector x, we apply the inverse of the matrix A:

Algorithm 2 Inverse power method

1: Choose a random vector $x^{(0)}$, s.t. $\|x^{(0)}\| = 1$
2: **for** $k = 1, 2, \ldots$ **do**
3: $\quad y^{(k)} = A^{-1}x^{(k-1)}$
4: $\quad x^{(k)} = \frac{y^{(k)}}{\|y^{(k)}\|}$
5: $\quad \lambda^{(k)} = x^{(k)T}Ax^{(k)}$
6: **end for**

In practice, we normally don't compute the inverse of the matrix A, but instead solve the linear system $Ax = y^{(k)}$ in each iteration.

1.2.4 Shifted inverse power method

The shifted inverse power method is an iterative algorithm to compute the eigenvalue with the smalles magnitude of a matrix, but with a shift μ added to the matrix A. The algorithm is as follows:

Algorithm 3 Shifted inverse power method

1: Choose a random vector $x^{(0)}$, s.t. $\|x^{(0)}\| = 1$
2: **for** $k = 1, 2, \ldots$ **do**
3: $\quad y^{(k)} = (A - \mu I)^{-1}x^{(k-1)}$
4: $\quad x^{(k)} = \frac{y^{(k)}}{\|y^{(k)}\|}$
5: $\quad \lambda^{(k)} = x^{(k)T}Ax^{(k)}$
6: **end for**

Note that with this algorithm, we are computing the eigenvalue of A that is closest to the shift μ. This can be useful to compute the eigenvalues of a matrix that are close to a given value.

1.2.5 Applications: Page Rank

The Page Rank algorithm is an algorithm that is used to rank the importance of web pages. The Page Rank algorithm is based on the idea that the importance of a web page is determined by the number of links that point to the page. The Page Rank algorithm is used by search engines to rank web pages in search results.

The Page Rank algorithm is based on the idea of random walks on the web graph. The web graph is a directed graph where the nodes represent web pages and the edges represent links between web pages. The Page Rank algorithm computes the importance of a web page by simulating a random walk on the web graph.

Suppose that we have a web graph with n web pages. Let $A \in \{0,1\}^{n \times n}$ be the adjacency matrix of the web graph, where $A_{ij} = 1$ if there is a link from web page j to web page i, and $A_{ij} = 0$ otherwise. Now, suppose that from each web page j, it is equally likely to follow any of the links that point to another web page i. Then, the probability of following a link from web page j to web page i is given by:

$$P_{ij} = \frac{A_{ij}}{\sum_{k=1}^{n} A_{kj}} \tag{1.26}$$

The matrix $P \in \mathbb{R}^{n \times n}$ is called the transition matrix of the web graph. Now, suppose that we are on a certain page as an initial state, and we want to know the probability of being on each page after a certain number of steps. This can be computed by the following equation:

$$x^{(t+1)} = Px^{(t)} \tag{1.27}$$

where $x^{(t)} \in \mathbb{R}^n$ is the probability distribution of being on each page at time t. Notice that $x^{(0)}$ is the initial state, with $x_i^{(0)} = 1$ if we start on page i, and $x_i^{(0)} = 0$ otherwise.

Now, we define the steady state probability distribution as the probability distribution of being on each page after an infinite number of steps. We call this vector $\pi \in \mathbb{R}^n$. Then, we have that:

$$P\pi = \pi \tag{1.28}$$

Meaning that, the steady state probability distribution is the eigenvector of the transition matrix P with eigenvalue 1. But notice that it is not always true that the steady state probability distribution is unique. In fact, the transition matrix P is a stochastic matrix, and it is guaranteed to have an eigenvector with eigenvalue 1. However, this eigenvector may not be unique. It can even happen that this distribution does not exist.

We formulate the Perron-Frobenius theorem, which states that a stochastic matrix has a unique eigenvector with eigenvalue 1 if the matrix is in its irreducible form. Formally, a matrix is irreducible if it is not in the form of:

$$P = \begin{bmatrix} P_{11} & P_{12} \\ 0 & P_{22} \end{bmatrix} \tag{1.29}$$

The theorem even states that the dominant eigenvalue is 1, and the other eigenvalues are less than 1.

When we have an irreducible form, we can just use the power iteration method to find the eigenvector with eigenvalue 1. However, when P is reducible, we need to introduce the following convex combination:

$$\hat{P} = \alpha P + (1-\alpha)\frac{1}{n}\mathbb{1}\mathbb{1}^T \tag{1.30}$$

where α is a damping factor, and $\mathbb{1}$ is a vector of ones. Notice that now, $\hat{P}$ is irreducible. We can think of it as adding a teleportation to the random walk, meaning that with probability α, we follow a link, and with probability $1-\alpha$, we teleport to a random page.

The Page Rank algorithm is based on the idea of finding the steady state probability distribution of the web graph. Naturally, the page with the highest probability is the most important page.

1.3 Symmetric matrices

A matrix $A \in \mathbb{R}^{n\times n}$ is called symmetric if $A = A^T$. Symmetric matrices have important properties:

- The eigenvectors of a symmetric matrix form an orthonormal basis of $\mathbb{R}^n$.
- All the eigenvalues of a symmetric matrix are real.

Let us prove the first property:

Let A be a symmetric matrix, and let $\lambda_1, \dots, \lambda_n$ be its eigenvalues. Let $v_1, \dots, v_n$ be the eigenvectors associated with the eigenvalues $\lambda_1, \dots, \lambda_n$. Let us take two eigenvectors v_i and v_j, such that $i \neq j$. Then we have that:

$$Av_i = \lambda_i v_i \quad \text{and} \quad Av_j = \lambda_j v_j \tag{1.31}$$

Then we have that:

$$(A - \lambda_i I)v_i = 0 \quad \text{and} \quad (A - \lambda_i I)v_j = (\lambda_j - \lambda_i)v_j$$

$$\Rightarrow v_i \in N(A - \lambda_i I) \quad \text{and} \quad v_j \in C(A - \lambda_i I) \tag{1.32}$$

Since A is symmetric, we have that $A - \lambda_i I$ is also symmetric. Then we have that:

$$N(A - \lambda_i I) = N((A - \lambda_i)^T) \perp C(A - \lambda_i I) \tag{1.33}$$

Concluding that v_i and v_j are orthogonal. Since this holds for all pairs of eigenvectors, we have that the eigenvectors of a symmetric matrix form an orthonormal basis of $\mathbb{R}^n$.

Now, let us prove the second property:

Since A is symmetric, we have that $A = A^T$. Then we have that:

$$\begin{aligned} Ax &= \lambda x \\ A\bar{x} &= \bar{\lambda}\bar{x} \end{aligned} \tag{1.34}$$

Then we have that:

$$
\begin{aligned}
\bar{x}^T A x &= \lambda x^T x = \lambda \|x\|^2 \\
x^T A \bar{x} &= \bar{\lambda} x^T \bar{x} = \bar{\lambda} \|x\|^2
\end{aligned} \tag{1.35}
$$

Since $A = A^T$, we have that:

$$
\bar{x}^T A x = (Ax)^T \bar{x} = x^T A^T \bar{x} = x^T A \bar{x} \tag{1.36}
$$

Then we have that:

$$
\lambda \|x\|^2 = \bar{\lambda} \|x\|^2 \Rightarrow \lambda = \bar{\lambda} \tag{1.37}
$$

This means that the eigenvalues of a symmetric matrix are real.

1.3.1 Symmetric positive definite matrices

A matrix $A \in \mathbb{R}^{n \times n}$ is called symmetric positive definite if it is symmetric and if for every vector $x \in \mathbb{R}^n$ we have that:

$$
x^T A x > 0 \quad \forall x \neq 0 \tag{1.38}
$$

Symmetric positive definite matrices have important properties:

- All the eigenvalues of a symmetric positive definite matrix are positive.
- The Cholesky factorization of a symmetric positive definite matrix exists and is unique:

$$
A = LL^T \tag{1.39}
$$

In fact, a symmetric matrix is positive definite if and only if all its eigenvalues are positive, so:

$$
x^T A x > 0 \quad \forall x \neq 0 \quad \Leftrightarrow \quad \lambda_i > 0 \quad \forall i \tag{1.40}
$$

Note that the quantity $x^T A x$ is called the energy of the vector x with respect to the matrix A. This quantity is always positive for a symmetric positive definite matrix.

1.4 Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) is a factorization of a matrix $A \in \mathbb{R}^{m \times n}$ as:

$$
A = U \Sigma V^T \tag{1.41}
$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices, and $\Sigma \in \mathbb{R}^{m \times n}$ is a quasi-diagonal matrix with the singular values of A. The singular values of A are the square roots of the eigenvalues of $A^T A$.

Note that if $rank(A) = r$, then the matrix Σ has r non-zero singular values, and the remaining singular values are zero. Assume that the singular values of A are $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$. Then we have that:

$$
\begin{aligned}
Av_i &= \sigma_i u_i \quad \forall i = 1, \dots, r \\
Av_i &= 0 \quad \forall i = r+1, \dots, n
\end{aligned}
\tag{1.42}
$$

where u_i and v_i are the columns of U and V, respectively. The vectors u_i and v_i are called the left and right singular vectors of A, respectively.

The SVD can also be written as:

$$
A = \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{1.43}
$$

1.4.1 Economy SVD

The economy SVD is a factorization of a matrix $A \in \mathbb{R}^{m \times n}$ as:

$$
A = U_r \Sigma_r V_r^T \tag{1.44}
$$

where $U \in \mathbb{R}^{m \times r}$, $V \in \mathbb{R}^{n \times r}$ and $\Sigma \in \mathbb{R}^{r \times r}$, with $r = rank(A)$.

The economy SVD is useful when we are only interested in the first r singular values of A, which are the non-zero singular values.

1.4.2 Low-rank approximation

The SVD can be used to compute a low-rank approximation of a matrix $A \in \mathbb{R}^{m \times n}$. Given a rank k, with $k < r = rank(A)$, the low-rank approximation of A is given by:

$$
A_k = U_k \Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i u_i v_i^T \tag{1.45}
$$

where $U_k \in \mathbb{R}^{m \times k}$, $V_k \in \mathbb{R}^{n \times k}$ and $\Sigma_k \in \mathbb{R}^{k \times k}$, with $k < r = rank(A)$.

Because the singular values of A are sorted in decreasing order, the low-rank approximation only considers the first k singular values of A, as they are the components of A with the largest contribution.

The low-rank approximation of a matrix is useful for data compression, as it allows to represent a matrix with a smaller number of parameters. Note that the low-rank approximation of a matrix is the best rank-k approximation of the matrix in the Frobenius norm. This is called the Eckart-Young theorem.

Optimal thresholding for low-rank approximation

There are several methods to determine the optimal rank k for the low-rank approximation of a matrix. A common heuristic approach is to graph the singular values of the matrix and choose the rank k as the point where the singular values start to decay rapidly.

We can see the idea of this method in the following figure:

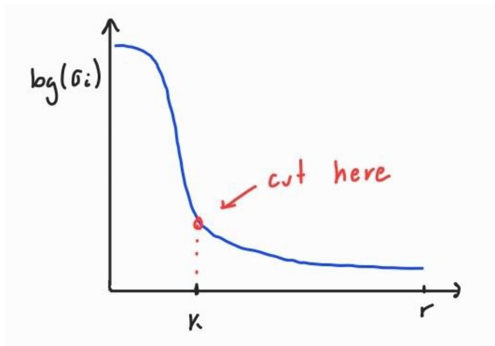

Figure 1.1: Heuristic method for choosing k

Another common method is to use the cumulative energy of the singular values. The cumulative energy is defined as:

$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{r} \sigma_i^2} \tag{1.46}$$

The cumulative energy measures the proportion of the total energy of the matrix that is captured by the first k singular values. A common heuristic is to choose the rank k as the smallest value such that the cumulative energy is above a certain threshold, e.g., 90%.

However, the optimal rank k for the low-rank approximation of a matrix is a complex problem that depends on the specific application. For example, we can use the following model:

$$A = A_{TRUE} + \gamma A_{NOISE} \tag{1.47}$$

where A_{TRUE} is the true low-rank matrix that we want to approximate, A_{NOISE} is the noise matrix, and γ is the noise level. In this case, the optimal rank k is the one that minimizes the error of the low-rank approximation of A_{TRUE}.

For this case, depending of the knowledge of the noise level, we can use the following methods to compute a threshold.

1. If the noise level is known, and $A \in \mathbb{R}^{n \times n}$ is a square matrix, then we can use the following threshold:

$$\tau = \frac{4}{\sqrt{3}} \sqrt{n} \gamma \tag{1.48}$$

2. If $n << m$:

$$\tau = \lambda \frac{n}{m} \sqrt{n} \tag{1.49}$$

Where $\lambda(\cdot)$ is a computable function (specific for this model, we don't care about it).

3. γ is unknown:

$$\tau = \omega(\frac{n}{m}) \sigma_{med} \tag{1.50}$$

Where σ_{med} is the median of the singular values of A, and $\omega(\cdot)$ is a computable function (specific for this model, we don't care about it).

The idea is to select the singular values that are above the threshold.

1.4.3 Existance of the SVD

The SVD of a matrix $A \in \mathbb{R}^{m \times n}$ always exists. The proof is as follows:

Let $A \in \mathbb{R}^{m \times n}$ be a matrix. Then we have that $A^T A \in \mathbb{R}^{n \times n}$. We have that $A^T A$ is symmetric and positive semidefinite. In fact:

- $A^T A$ is symmetric: $(A^T A)^T = A^T (A^T)^T = A^T A$
- $A^T A$ is positive semidefinite: $x^T A^T A x = (Ax)^T A x = \|Ax\|^2 \geq 0$

Therefore, $A^T A$ has an eigendecomposition:

$$A^T A = V \Lambda V^T \quad \text{s.t. } V^T V = I$$

Let us show now that $rank(A) = rank(A^T A)$. We can prove this by showing that the null space of A is the same as the null space of $A^T A$. Let $x \in N(A)$, then we have that:

$$Ax = 0 \Rightarrow A^T A x = 0$$

This means that $N(A) \subseteq N(A^T A)$. Now, let $x \in N(A^T A)$, then we have that:

$$A^T A x = 0 \Rightarrow x^T A^T A x = 0 \Rightarrow \|Ax\|^2 = 0 \Rightarrow Ax = 0$$

This means that $N(A^T A) \subseteq N(A)$. Therefore, $N(A) = N(A^T A)$, and we have that $rank(A) = rank(A^T A)$.

Now, let us consider the eigenpairs (λ_i, v_i) of $A^T A$. Then let us define u_i as:

$$u_i = \frac{1}{\sqrt{\lambda_i}} A v_i$$

We will prove that each u_i is a unitary vector, and that the vectors u_i are mutually orthogonal. Let us first show that each u_i is a unitary vector:

$$\|u_i\|^2 = \left\| \frac{1}{\sqrt{\lambda_i}} A v_i \right\|^2 = \frac{1}{\lambda_i} v_i^T A^T A v_i = \frac{1}{\lambda_i} \lambda_i = 1$$

Now, let us show that the vectors u_i are mutually orthogonal. Let $i \neq j$, then we have that:

$$u_i^T u_j = \frac{1}{\sqrt{\lambda_i}} v_i^T A^T \frac{1}{\sqrt{\lambda_j}} A v_j = \frac{1}{\sqrt{\lambda_i \lambda_j}} v_i^T A^T A v_j$$

$$= \frac{1}{\sqrt{\lambda_i \lambda_j}} \lambda_j v_i^T v_j = 0$$

Now, we can show that the vectors u_i are the eigenvectors of AA^T, with eigenvalues λ_i:

$$AA^T u_i = AA^T \frac{1}{\sqrt{\lambda_i}} A v_i =$$

$$\frac{1}{\sqrt{\lambda_i}} AA^T A v_i = \frac{1}{\sqrt{\lambda_i}} A \lambda_i v_i = \lambda_i u_i$$

Therefore, we have that $AA^T = U \Lambda U^T$, where U is the matrix whose columns are the vectors u_i. Now, let us define Σ as the diagonal matrix whose diagonal elements are the square roots of the eigenvalues of $A^T A$:

$$\Sigma = \begin{bmatrix} \sqrt{\lambda_1} & 0 & \dots & 0 \\ 0 & \sqrt{\lambda_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sqrt{\lambda_n} \end{bmatrix}$$

Then we have that:

$$A = U \Sigma V^T$$

where V is the matrix whose columns are the eigenvectors of $A^T A$. Therefore, the SVD of A exists.

1.4.4 Computation of the SVD

The SVD of a matrix can be computed using the following steps:

1. Compute the eigendecomposition of $A^T A$.
2. Compute the singular values of A as the square roots of the eigenvalues of $A^T A$.
3. Compute the right singular vectors of A as the eigenvectors of $A^T A$.
4. Compute the left singular vectors of A as $u_i = \frac{1}{\sqrt{\lambda_i}} A v_i$.

1.4.5 Geometrical interpretation of the SVD

The SVD of a matrix $A \in \mathbb{R}^{m \times n}$ can be interpreted geometrically as a composition of three transformations:

- The matrix V represents a rotation in $\mathbb{R}^n$.
- The matrix Σ represents a scaling in $\mathbb{R}^{m \times n}$.
- The matrix U represents a rotation in $\mathbb{R}^m$.

In $\mathbb{R}^2$, the SVD of a matrix A can be interpreted as a rotation of the unit circle, followed by a scaling along the axes, followed by another rotation. So this transformation can be represented with 4 parameters: two angles and two scaling factors. In $\mathbb{R}^3$, its the same idea, but with 6 parameters: three angles and three scaling factors.

1.4.6 Polar decomposition

Let $A \in \mathbb{R}^{n \times n}$ be a symmetric matrix. The polar decomposition of A is a factorization of A as:

$$A = QS \tag{1.51}$$

where Q is an orthogonal matrix and S is a symmetric positive definite matrix. The polar decomposition can be derived from the SVD of A as:

$$A = U\Sigma V^T = (UV^T)(V\Sigma V^T) = QS \tag{1.52}$$

where $Q = UV^T$ and $S = V\Sigma V^T$. So, the polar decomposition of a symmetric matrix always exists.

1.4.7 Eckart-Young theorem

The Eckart-Young theorem states that the best rank-k approximation of a matrix $A \in \mathbb{R}^{m \times n}$ in the Frobenius norm is given by the low-rank approximation of A, denoted as A_k:

$$\min_{rank(B)=k} \|A - B\|_F = \|A - A_k\|_F \tag{1.53}$$

This also applies to the spectral norm $\|\cdot\|_2$. To prove this, let us define the mentioned norms:

Frobenius norm

The Frobenius norm of a matrix $A \in \mathbb{R}^{m \times n}$ is defined as:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2} \tag{1.54}$$

The Frobenius norm is the Euclidean norm of the vector obtained by stacking the columns of A. It is also equal to:

$$\|A\|_F = \sqrt{tr(A^T A)} \tag{1.55}$$

It has 3 important properties:

1. $\|A\|_F = \|A^T\|_F$
2. It is invariant under orthogonal transformations: $\|QA\|_F = \|A\|_F$
3. It is equal to the square root of the sum of the squared singular values of A:

$$\|A\|_F = \sqrt{\sum_{i=1}^{r} \sigma_i^2} \tag{1.56}$$

Spectral norm

The spectral norm of a matrix $A \in \mathbb{R}^{n\times n}$ is defined as:

$$\|A\|_2 = \sup_{\|x\|_2=1} \|Ax\|_2 \tag{1.57}$$

It is equal to the largest singular value of A. This is denoted as:

$$\|A\|_2 = \sigma_1 \tag{1.58}$$

Proof of the Eckart-Young theorem

For both norms $\|\cdot\|_F$ and $\|\cdot\|_2$, we will prove that:

$$\|A - A_k\| \leq \|A - B\| \quad \forall B \in \mathbb{R}^{m\times n} \quad \text{s.t.} \quad rank(B) = k$$

Let $B \in \mathbb{R}^{m\times n}$ be a matrix such that $rank(B) = k$. Then we have that:

$$dim(N(B)) = n - k$$

Let us consider the matrix V_{k+1} whose columns are the vectors $v_1, \dots, v_{k+1}$ from the SVD of A. Then we have that:

$$rank(V_{k+1}) = k + 1 \Rightarrow dim(C(V_{k+1})) = k + 1$$
$$\Rightarrow dim(N(B)) + dim(C(V_{k+1})) = n - k + k + 1 = n + 1$$

This means that $N(B) \cap C(V_{k+1}) \neq \emptyset$. Let $w \in N(B) \cap C(V_{k+1})$, such that $\|w\| = 1$. Then we have that, for coefficients α_i:

$$w = \sum_{i=1}^{k} \alpha_i v_i \quad \wedge \quad Bw = 0$$

Since $\|w\| = 1$, then $\sum_{i=1}^{k} \alpha_i^2 = 1$. Now, we have that, for the spectral norm:

$$\|A - B\|_2^2 \geq \|(A - B)w\|_2^2 = \|Aw\|_2^2 = w^T A^T Aw$$

$$= w^T V \Sigma^2 V^T w = \sum_{i=1}^{k} \sigma_i^2 \alpha_i^2 \geq$$

$$\geq \sigma_{k+1}^2 \sum_{i=1}^{k} \alpha_i^2 = \sigma_{k+1}^2$$

$$= \|A - A_k\|_2^2$$

Therefore:

$$\|A - A_k\|_2 \leq \|A - B\|_2 \quad \forall B \in \mathbb{R}^{m \times n} \quad \text{s.t.} \quad rank(B) = k$$

For the Frobenius norm, we need to use the Weyl's inequality:

$$\sigma_{i+j-1}(X + Y) \leq \sigma_i(X) + \sigma_j(Y) \tag{1.59}$$

where $\sigma_i(X)$ denotes the i-th singular value of a matrix X. Using this inequality, let us define $X = A - B$ and $Y = B$. Then we have that:

$$\sigma_{i+k}(A) \leq \sigma_i(A - B) + \sigma_{k+1}(B)$$

Note that $\sigma_{k+1}(B) = 0$, since B has rank k. Therefore:

$$\sigma_{i+k}(A) \leq \sigma_i(A - B)$$

Then, we have that:

$$\|A - A_k\|_F^2 = \sum_{i=k+1}^{r} \sigma_i^2(A) = \sum_{i=1}^{r-k} \sigma_{i+k}^2(A) \leq$$

$$\leq \sum_{i=1}^{r-k} \sigma_i^2(A - B) \leq \sum_{i=1}^{n} \sigma_i^2(A - B) = \|A - B\|_F^2$$

Therefore:

$$\|A - A_k\|_F \leq \|A - B\|_F \quad \forall B \in \mathbb{R}^{m \times n} \quad \text{s.t.} \quad rank(B) = k$$

This completes the proof.

1.4.8 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a technique to reduce the dimensionality of a dataset. Given a dataset $X \in \mathbb{R}^{m \times n}$, PCA computes the SVD of the centered data matrix $\bar{B}$, where $\bar{B}$ is obtained by subtracting the mean of each column of X. Then, the principal components of the dataset are the right singular vectors of $\bar{B}$. The principal components are the directions of the dataset with the largest variance.

In detail, let $X \in \mathbb{R}^{m \times n}$ the dataset, with m samples and n features. Let $\bar{x}$ be mean vector of the data samples. Then, we define $\bar{X}$ as:

$$\bar{X} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \bar{x}^T \tag{1.60}$$

Then, we define the centered data matrix $\bar{B}$ as:

$$\bar{B} = X - \bar{X} \tag{1.61}$$

The covariance matrix is then defined as:

$$C = \frac{1}{m-1} \bar{B}^T \bar{B} \tag{1.62}$$

We observe that the eigendecomposition of the covariance matrix C is as follows:

$$C = V \Lambda V^T \tag{1.63}$$

We can obtain the principal components of the dataset as the columns of V. The principal components are the directions of the dataset with the largest variance. The first principal component is the direction with the largest variance, the second principal component is the direction with the second largest variance, and so on.

By doing the SVD of the centered data matrix $\bar{B}$, we can obtain the principal components of the dataset, such that:

$$\bar{B} = U \Sigma V^T \Rightarrow V = \text{principal components} \quad \wedge \quad \Lambda = \frac{1}{m-1} \Sigma^2 \tag{1.64}$$

1.4.9 Data pre-processing: sensitivity to transformations

The SVD is specially sensitive to scaling, rotating and translating the data. This is because the SVD is a geometrical transformation of the data, and these transformations change the geometry of the data. For that reason, if we want to apply the SVD to a dataset, we need to pre-process the data to remove or somehow filter the effects of these transformations.

For example, if we have a dataset that consist of face images, and we want to apply the SVD to this dataset, we need to pre-process the images to remove the effects of scaling, rotating and translating the faces. This can be done by aligning the faces to a common reference frame, and by normalizing the faces to have the same size and orientation.

In general, the pre-processing of the data is an important step in the application of the SVD to a dataset. The pre-processing step can have a significant impact on the results of the SVD, and it is important to carefully choose the pre-processing steps to obtain meaningful results.

1.4.10 Randomized SVD

The SVD of a matrix can be computed using the randomized SVD algorithm. The randomized SVD is an iterative algorithm that approximates the SVD of a matrix using random projections. The algorithm is as follows:

Algorithm 4 Randomized SVD

1: Choose a random matrix $Y \in \mathbb{R}^{n \times k}$, with $k \ll n$.
2: Compute the matrix $Z = AY$.
3: Compute the QR factorization of $Z = QR$.
4: Compute the matrix $B = Q^T A$.
5: Compute the SVD of $B = \hat{U}\Sigma V^T$.
6: Obtain U as $U = Q\hat{U}$.
7: The approximate SVD of A is given by $A \approx U\Sigma V^T$.

The randomized SVD algorithm is a fast and efficient way to compute the SVD of a matrix. The algorithm is based on the fact that the SVD of a matrix can be approximated using random projections. The algorithm is especially useful when the matrix is large and when only an approximate SVD is needed.

The idea behind this algorithm is a sampling of the column space of the matrix A using the matrix Y.

This method is most widely used in the context of large-scale data analysis, where the matrix A is too large to be stored in memory. In this case, the randomized SVD algorithm can be used to compute an approximate SVD of the matrix using only a small amount of memory.

1.5 Matrix completion

Let $X \in \mathbb{R}^{n \times p}$ be a matrix that is partially observed. That is, we only know X_{ij} for $(i, j) \in \Omega$, where Ω is the set of observed entries. Assume that $rank(X) = r << \min(n, p)$. Our goal is to estimate X_{ij} for every $(i, j) \notin \Omega$.

Formally, we express this as:

$$\hat{X} = \text{argmin}_Z \{rank(Z)\} \quad \text{s.t.} \quad Z_{ij} = X_{ij} \quad \forall (i, j) \in \Omega \tag{1.65}$$

However, this formulation is a non-convex optimization problem, and it is not solvable in practical cases. Instead, we can use the following convex optimization problem:

$$\hat{X} = \text{argmin}_Z \{\|Z\|_*\} \quad \text{s.t.} \quad Z_{ij} = X_{ij} \quad \forall (i, j) \in \Omega \tag{1.66}$$

where $\|Z\|_*$ is the nuclear norm of Z, defined as:

$$\|Z\|_* = \sum_{i=1}^{r} \sigma_i(Z) \tag{1.67}$$

It is possible to prove that minimizing the nuclear norm of Z is highly probable to be equivalent to minimizing the rank of Z. This is because the nuclear norm is the convex relaxation of the rank function.

There are many ways of solving this optimization problem. One of the most common methods is the Singular Value Thresholding (SVT) algorithm. The SVT algorithm is an iterative algorithm that approximates the solution of the optimization problem. The algorithm is as follows:

Algorithm 5 Singular Value Thresholding (SVT)

```
Initialize Z_0 = X.
for k = 1, 2, ... do
    Compute the SVD of Z_{k-1} = U Σ V^T.
    Compute the singular value thresholding of Σ as Σ_λ = max(Σ − λ, 0).
    Compute the matrix Z_k = U Σ_λ V^T.
    Make Z_k satisfy the constraints Z_k = X for (i, j) ∈ Ω.
    If ‖Z_k − Z_{k-1}‖ < ε, then stop.
end for
```

The SVT algorithm is a fast and efficient way to solve the matrix completion problem. The algorithm is based on the fact that the nuclear norm of a matrix is the sum of its singular values. The algorithm iteratively computes the singular value thresholding of the matrix, and then makes the matrix satisfy the constraints. The algorithm converges to the solution of the optimization problem, and it is guaranteed to converge to the global minimum.

1.6 Regression Methods

Regression methods are a class of machine learning algorithms that are used to predict the value of a continuous variable based on the value of one or more input variables. Regression methods are widely used in data analysis, and they are used to model the relationship between the input variables and the output variable.

1.6.1 Least Squares

The least squares problem is a problem to find the vector x that minimizes the residual of a linear system $Ax = b$. The least squares solution is given by:

$$x = (A^T A)^{-1} A^T b \tag{1.68}$$

Here, we are assuming that $A \in \mathbb{R}^{m \times n}$, with $m > n$ and $rank(A) = n$. The least squares solution is the solution that minimizes the residual $\|Ax - b\|_2$. We can prove this in 2 different ways:

- **Geometrical interpretation**:

  The least squares solution is the solution that minimizes the residual $\|Ax - b\|_2$. This means that the residual is orthogonal to the column space of A. Let $r = Ax - b$ be the residual, then we have that:

$$\begin{aligned} &r \perp C(A) \Rightarrow r \in N(A^T) \Rightarrow A^T r = 0 \\ &\Rightarrow A^T(Ax - b) = 0 \Rightarrow A^T A x = A^T b \end{aligned}$$

Note that A has full column rank, so A^TA is invertible (in fact, it is s.d.p). Therefore, we have that:

$$x = (A^TA)^{-1}A^Tb$$

- **Derivation**:

  The least squares solution is the solution that minimizes the residual $\|Ax - b\|_2$. This means that the least squares solution is the solution that minimizes the function:

  $$\mathcal{L}(x) = \|Ax - b\|_2^2 = (Ax - b)^T(Ax - b) = x^TA^TAx - 2b^TAx + b^Tb$$

  To find the minimum of this function, we need to find the critical points of the function. Let us take the derivative of the function with respect to x:

  $$\frac{\partial \mathcal{L}(x)}{\partial x} = 2A^TAx - 2A^Tb = 0$$
  $$\Rightarrow A^TAx = A^Tb$$

  By the same argument as before, we have that:

  $$x = (A^TA)^{-1}A^Tb$$

In the context of data analysis, we often have a matrix $X \in \mathbb{R}^{n\times p}$ that represents n samples of p features. In this case, the least squares solution is used to find the coefficients w of a linear model that best fits the data. The system of equations is given by:

$$Xw = y \tag{1.69}$$

where X is the matrix of features, w is the vector of coefficients, and y is the vector of labels. Note that an exact solution may not exist, so we need to find the least squares solution. The least squares solution, as we deduced before, would be given by:

$$w = (X^TX)^{-1}X^Ty \tag{1.70}$$

Note that the matrix $(X^TX)^{-1}X^T$ is called the pseudo-inverse of X, and it is denoted as $X^\dagger$.

Least squares and SVD

Computing the inverse of the matrix X^TX can be computationally expensive, especially when the matrix X is large. In this case, the least squares solution can be computed using the SVD of the matrix X. In fact, we have that:

$$w = V\Sigma^\dagger U^Ty \tag{1.71}$$

where U, Σ and V are the SVD of X, and $\Sigma^\dagger$ is the pseudo-inverse of Σ. The pseudo-inverse of Σ is obtained by taking the reciprocal of the non-zero singular values of Σ, and then taking the transpose of the resulting matrix. The pseudo-inverse of Σ is denoted as $\Sigma^\dagger$. Visually, the pseudo-inverse of Σ is as follows:

$$\Sigma^\dagger = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \dots & 0 & \dots & 0 \\ 0 & \frac{1}{\sigma_2} & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & \frac{1}{\sigma_p} & \dots & 0 \end{bmatrix} \tag{1.72}$$

Underdetermined system case (wide matrix)

In the previous case, we assumed that $n > p$, and $rank(X) = p$. However, in some cases, we have that $n < p$, and $rank(X) = n$. In this case, the system of equations $Xw = y$ is underdetermined, and there are infinitely many solutions. Still, the least square solution is still useful, as it gives the solution that minimizes the norm of the coefficients.

We can prove this as follows. Let w be a solution of the system of equations $Xw = y$ and let w_0 be the least squares solution. Then we have that:

$$\begin{aligned} \|w\|^2 = \|(w - w_0) + w_0\| &= [(w - w_0) + w_0]^T[(w - w_0) + w_0] \\ &= \|w - w_0\|^2 - 2(w - w_0)^T w_0 + \|w_0\|^2 \end{aligned}$$

Note that

$$\begin{aligned} (w - w_0)^T w_0 &= (w - w_0)^T X^T (XX^T)^{-1} y \\ &= (X(w - w_0))^T (XX^T)^{-1} y \end{aligned}$$

And since $Xw = y$ and $Xw_0 = y$, we have that $X(w - w_0) = 0$. Therefore, we have that:

$$\|w\|^2 = \|w - w_0\|^2 + \|w_0\|^2 \geq \|w_0\|^2$$

This means that the least squares solution is the solution that minimizes the norm of the coefficients. We often use this in practice to avoid error amplification in our calculations.

1.6.2 Ridge regression

Let $w_{LS} = (X^T X)^{-1} X^T y$ be the least squares solution of the system of equations $Xw = y$. Then, let us assume the following model:

$$y = Xw^* + \varepsilon \tag{1.73}$$

where w^* is the true coefficients of the model, and ε is the noise. Then:

$$w_{LS} = (X^T X)^{-1} X^T (Xw^* + \varepsilon) = w^* + (X^T X)^{-1} X^T \varepsilon \tag{1.74}$$

Note that $X = U\Sigma V^T$, so we have that $(X^T X)^{-1} X^T = V\Sigma^\dagger U^T$. Then:

$$w_{LS} = w^* + V\Sigma^\dagger U^T \varepsilon \tag{1.75}$$

The least squares solution is the true coefficients plus a term that depends on the noise. Let us suppose that for some p, we have that $\sigma_p \approx 0$. Then we have that $\Sigma^{\dagger}_{pp} = 1/\sigma_p >> 0$, and this will amplify the noise.

To avoid this, we can use the ridge regression. The ridge regression is a technique that adds a regularization term to the least squares solution. The optimization problem of the ridge regression is as follows:

$$w_R = argmin\ \{\|Xw - y\|_2^2 + \lambda\|w\|_2^2\} \tag{1.76}$$

where λ is the regularization parameter. The solution of the ridge regression is given by:

$$w_R = (X^T X + \lambda I)^{-1} X^T y \tag{1.77}$$

Let us analyse this solution with the SVD of X:

$$w_R = (V\Sigma^T U^T U\Sigma V^T + \lambda V V^T)^{-1} V\Sigma U^T y$$

$$(V(\Sigma^T\Sigma + \lambda I)V^T)^{-1} V\Sigma U^T y$$

$$V(\Sigma^T\Sigma + \lambda I)^{-1}\Sigma U^T y$$

Note that, if $S = (\Sigma^T\Sigma + \lambda I)^{-1}\Sigma^T$, then we have that:

$$S_{pp} = \frac{\sigma_p}{\sigma_p + \lambda}$$

So we have 2 cases:

- If $\sigma_p >> 0$, then $S_{pp} \approx 1/\sigma_p \approx 0$.
- If $0 < \sigma_p << 1$, then $S_{pp} \approx 0$.

In both cases, the ridge regression avoids the error amplification of the noise. However, notice that the ridge regression introduces a bias in the coefficients. This bias is controlled by the regularization parameter λ. The larger the value of λ, the larger the bias in the coefficients.

In general, when we know that $\varepsilon = 0$, the least squares solution is the best solution. However, when we have noise in the data, the ridge regression is a better solution, as it avoids the error amplification of the noise.

The ridge regression is a special case of the Tikhonov regularization, which is a general technique to regularize ill-posed problems. The Tikhonov regularization adds a regularization term to the least squares solution, which is controlled by the regularization parameter λ.

1.6.3 LASSO regression

The LASSO regression is a technique that adds a regularization term to the least squares solution. The optimization problem of the Lasso regression is as follows:

$$w_L = argmin \, \{\|Xw - y\|_2^2 + \lambda \|w\|_1\} \tag{1.78}$$

where λ is the regularization parameter. Notice that the Lasso regression adds a regularization term that is the L_1 norm of the coefficients. This approach is useful when we want to find a sparse solution, i.e., a solution with few non-zero coefficients.

LASSO stands for Least Absolute Shrinkage and Selection Operator. In general, we have that:

- When λ is large, we tend to have a higher sparsity in the solution.
- When λ is small, we tend to have a solution that is closer to the least squares solution.

We can mix both LASSO and Ridge regression in the Elastic Net regression. The Elastic Net regression is a technique that adds both L_1 and L_2 regularization terms to the least squares solution. The optimization problem of the Elastic Net regression is as follows:

$$w_E = argmin \, \{\|Xw - y\|_2^2 + \lambda_1 \|w\|_1 + \lambda_2 \|w\|_2^2\} \tag{1.79}$$

where λ_1 and λ_2 are the regularization parameters. The Elastic Net regression is useful when we want to find a solution that is both sparse and close to the least squares solution.

1.6.4 Kernel methods

Let us formulate the Ridge Regression solution in terms of the SVD of X. We have that:

$$\begin{aligned} w_R &= V\Sigma^T U^T U(\Sigma\Sigma^T + \lambda I)^{-1} U^T y \\ &= X^T \beta \end{aligned}$$

where $\beta = U(\Sigma\Sigma^T + \lambda I)^{-1}\Sigma U^T y = (XX^T + \lambda I)^{-1} y$. We call β the dual coefficients of the Ridge Regression.

In general, we can have a linear separator in the input space, given by:

$$y = f(x) = w^T x = x_{i1} w_1 + \ldots + x_{ip} w_p$$

But we could also have a linear separator in a new feature space $\phi(\cdot)$, given by:

$$y = f(x) = w^T \phi(x) = \phi(x)_{i1} w_1 + \ldots + \phi(x)_{id} w_d$$

where $\phi(x)$ is a non-linear transformation of the input space. This is the idea behind the kernel methods. Notice that the dimension of the feature space can be larger than the dimension of the input space. For example:

$$\phi(x) = (x_1, x_2, x_1^2, x_2^2, x_1 x_2)$$

In this case, the dimension of the feature space is 5, larger than the dimension of the input space, which is 2. Note that the feature space adds non-linear features to the input space.

In this sense, we can create the following matrix:

$$\Phi = \begin{bmatrix} \phi(x_1)^T \\ \vdots \\ \phi(x_n)^T \end{bmatrix}$$

Then, to find the coefficients w of the linear separator in the feature space, we can use the Ridge Regression solution:

$$w = \Phi^T \beta$$

where $\beta = (\Phi\Phi^T + \lambda I)^{-1} y$. To avoid the computation of the feature space, we can use kernel functions. A kernel function is a function that computes the inner product of two vectors in the feature space without computing the feature space. Formally:

$$K(x, x') = \phi(x)^T \phi(x') \tag{1.80}$$

The most common kernel functions are:

- Linear kernel: $K(x, x') = x^T x'$
- Polynomial kernel: $K(x, x') = (x^T x' + c)^d$
- Gaussian kernel: $K(x, x') = \exp(-\frac{\|x-x'\|^2}{2\sigma^2})$

In the previous case for $\phi(x) = (x_1, x_2, x_1^2, x_2^2, x_1 x_2)$, we have that:

$$K(x, x') = (x^T x')^2$$
$$\Rightarrow K(x, x') = (x_1 x_1' + x_2 x_2' + x_1^2 x_1'^2 + x_2^2 x_2' + x_1 x_2 x_1' x_2') = \phi(x)^T \phi(x')$$

Now, notice that:

$$[\Phi\Phi^T]_{ij} = \phi(x_i)^T \phi(x_j) = K(x_i, x_j)$$

Therefore, we have that:

$$\beta = (K + \lambda I)^{-1} y \tag{1.81}$$

where K is the kernel matrix, defined as $K_{ij} = K(x_i, x_j)$. Then, the coefficients w of the linear separator in the feature space are given by:

$$w = \Phi^T \beta \tag{1.82}$$

Suppose now that we have a new sample x and we want to predict its label y. Then, we have that:

$$y = w_R^T \cdot \phi(x) = \beta^T \phi(x) = \sum_{i=1}^{n} \beta_i K(x_i, x)$$

Note that the magnitude of the coefficients β_i is related to the importance of the sample x_i in the prediction of the label y. The larger the magnitude of β_i, the more important the sample x_i is in the prediction of y.

1.6.5 Support Vector Regression (SVR)

Until now, we have used the following loss function to obtain the coefficients of the linear separator:

$$\mathcal{L}(w) = \|Xw - y\|_2^2 + \lambda \|w\|_2^2 \tag{1.83}$$

With $\lambda = 0$, we have the least squares solution. With $\lambda > 0$, we have the Ridge Regression solution. Now, let us propose a new loss function:

$$\mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^{n} \max(0, |y_i - w^T x_i| - \varepsilon) + \lambda \|w\|_2^2 \tag{1.84}$$

We can even introduce a feature map $\phi(\cdot)$, as in the kernel methods:

$$\mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^{n} \max(0, |y_i - w^T \phi(x_i)| - \varepsilon) + \lambda \|w\|_2^2 \tag{1.85}$$

This loss function is the loss function of the Support Vector Regression (SVR). The SVR is a technique that is used to predict the value of a continuous variable based on the value of one or more input variables. The SVR is similar to the Support Vector Machine (SVM), but it is used for regression problems instead of classification problems.

We can reformulate the SVR in terms of the dual coefficients β:

$$\mathcal{L}(\beta) = \frac{1}{2} \beta^T K \beta - \beta^T y + \varepsilon \|\beta\|_1 \quad s.t \quad |\beta_i| \leq \frac{1}{2n\lambda} \tag{1.86}$$

We say that the support vectors are the samples x_i such that $\beta_i \neq 0$. With a bigger value of ε, we have a higher sparsity in the solution, meaning that we have fewer support vectors. Notice that when we don't have a feature map, the matrix K is equivalent to XX^T.

Chapter 2

Automatic differentiation

2.1 Introduction: differentiation methods

Suppose that we have a function $f : \mathbb{R}^n \to \mathbb{R}^m$ that we want to differentiate. We have many methods to do so:

1. **Manual differentiation**: we can differentiate the function by hand.
    - Pros: it is exact, fast, and good for theory
    - Cons: it is error-prone, time-consuming, and difficult for complex functions
2. **Numerical differentiation**: we can approximate the derivative by finite differences.
    - Pros: it is easy to implement
    - Cons: it is slow, inaccurate (round-off and truncation errors), and not suitable for complex functions

    Let us look at the example of finite differences in 1 dimension. We can formulate 3 ways to approximate the derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at a point x:

    - Forward difference: $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
    - Backward difference: $f'(x) \approx \frac{f(x)-f(x-h)}{h}$
    - Central difference: $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$

    The problem with the Finite Differences method (FDM) is that it presents two types of errors:

    - **Truncation error**: it is the error that arises due to the approximation of the derivative. It is a consequence of the fact that we are using a Taylor series to approximate the derivative. This error decreases as h decreases. For forward and backward differences, the error is $O(h)$, while for central differences, the error is $O(h^2)$.
    - **Round-off error**: it is the error that arises due to the finite precision of the computer. It is a consequence of adding two numbers of different magnitudes, or subtracting two numbers that are close to each other. This error increases as h decreases.

The following graph represents the trade-off between the truncation and round-off errors:

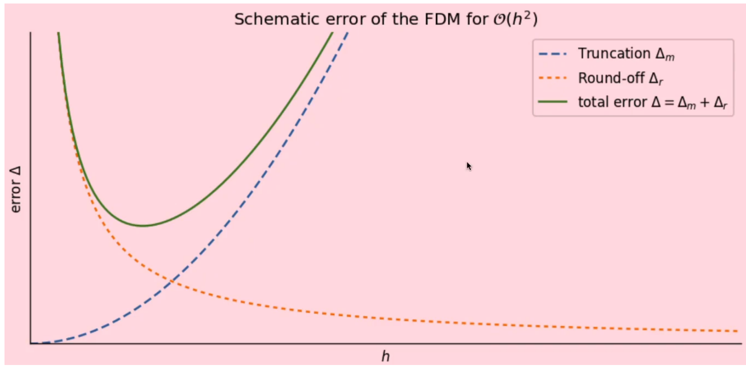

Figure 2.1: Trade-off between truncation and round-off errors

And we can actually see this in an example:

```
import numpy as np

def forward_diff(f, x, h):
    return (f(x + h) - f(x)) / h

def centered_diff(f, x, h):
    return (f(x + h) - f(x - h)) / (2 * h)

h = np.logspace(-16, 0, num=500, endpoint=True)
x = 1.0
D1f = forward_diff(np.sin, x, h)
D2f = centered_diff(np.sin, x, h)
err1 = np.abs(D1f - np.cos(x))
err2 = np.abs(D2f - np.cos(x))
```

Then, the plot of the errors is:

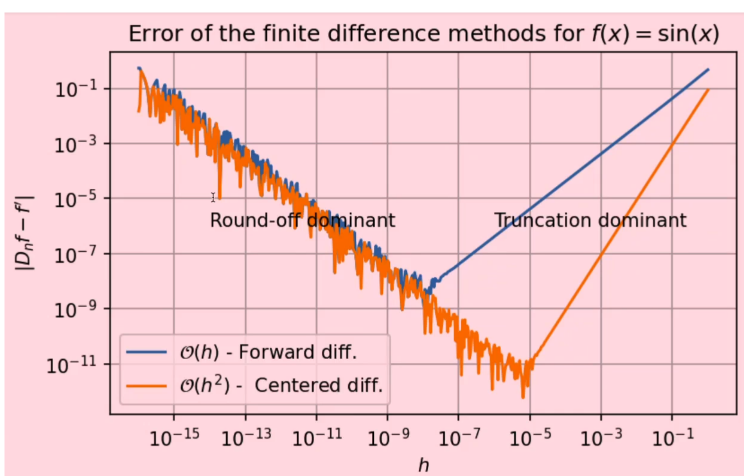

Figure 2.2: Errors of the finite differences method

3. **Symbolic differentiation**: we can use a computer algebra system (CAS) to compute the derivative symbolically.
    - Pros: it is exact, good for theory.
    - Cons: it is slow, memory-intensive, and can cause expression swell for complex functions.
4. **Automatic differentiation**: it is a technique that computes the derivative of a function by applying the chain rule.
    - Pros: it is exact, fast, and general (good for complex functions).
    - Cons: it is difficult to implement, and it requires a good understanding of the chain rule.

In this chapter, we will focus on automatic differentiation (AD). One of the main methods of AD is the backward propagation algorithm, which is used in deep learning to compute the gradients of the loss function with respect to the parameters of the model.

Automatic differentiation (AD) is a technique that it is based on the idea of splitting the computation of the derivative of a function into elementary operations:

- Unitary operations
- Exponentials, logarithms, trigonometric functions, etc.

2.2 Automatic differentiation: Wengert list

Automatic differentiation (AD) is a technique that computes the derivative of a function by applying the chain rule. It is based on the idea of splitting the computation of the derivative of a function into elementary operations. The main idea is to represent the function as a composition of elementary functions and then apply the chain rule to compute the derivative.

To implement AD, we take advantage of the evaluation trace of the function, also called the forward primal trace. This trace is a sequence of elementary operations that are applied to the input variables to compute the output of the function. It keeps track of the intermediate values of the variables and the operations that are applied to them. These variables are called primal variables, and are typically denoted by v_i for functions $f : \mathbb{R}^n \to \mathbb{R}^m$, following these rules:

- **Input variables**: $v_{i-n} = x_i, \quad i = 1, ..., n$
- **Intermediate variables**: $v_i = g_i(v_{j_1}, ..., v_{j_k}), \quad i = 1, ..., \ell, \quad j_i, ..., j_k \subseteq [1-n, \ell]$
- **Output variables**: $y_{m-i} = v_{\ell-i}, \quad i = m-1, ..., 0$

For example, consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined as $f(x, y) = \sin((x_1 + x_2)x_2^2)$. The evaluation trace of this function is:

- $v_{-1} = x_1$, $v_0 = x_2$
- $v_1 = v_{-1} + v_0$

- $v_2 = v_0^2$
- $v_3 = v_1 \cdot v_2$
- $v_4 = \sin(v_3)$
- $y_1 = v_4$

This trace is also called the Wengert list (as it was introduced by R. E. Wengert in 1964). We can represent the Wengert list as a directed acyclic graph (DAG), where nodes represent variables and edges describe the computational hierarchy of input to output transformations. This graph is also commonly referred to as the computational graph. For the previous example, we get:

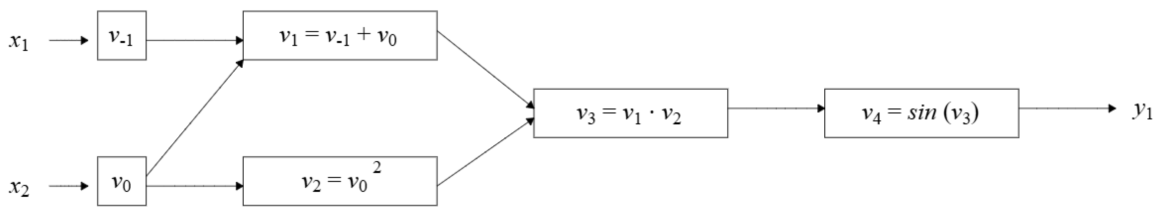

Figure 2.3: Comp. graph of the function $f(x, y) = \sin((x_1 + x_2)x_2^2)$

These tools are essential for the implementation of AD, in particular, its two modes:

- **Forward mode**
- **Backward mode**

2.3 Forward mode of AD

The forward mode of automatic differentiation (AD) adopts the idea of the evaluation trace, but introduces a new concept: the tangent variables, usually denoted by $\dot{v}_i$. These tangents carry the derivative information of the primal variables with respect to a single input variable of interest. For example, if we are interested in computing the derivative of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with respect to the variable x_i, we write:

$$\dot{v}_j = \frac{\partial v_j}{\partial x_i} \quad j = 1 - n, ..., \ell$$

The forward mode of AD computes the derivative of a function by applying the chain rule to the evaluation trace. It is based on the idea of propagating the derivative information from the input variables to the output variables. For this, we calculate what is called the forward tangent trace, by simply applying derivation rules to the evaluation trace. For example, consider the same function $f(x, y) = \sin((x_1 + x_2)x_2^2)$. The forward tangent trace of this function is:

- $\dot{v}_{-1} = \dot{x}_1$, $\dot{v}_0 = \dot{x}_2$
- $\dot{v}_1 = \dot{v}_{-1} + \dot{v}_0$

- $\dot{v}_2 = 2v_0\dot{v}_0$
- $\dot{v}_3 = v_2\dot{v}_1 + v_1\dot{v}_2$
- $\dot{v}_4 = \cos(v_3)\dot{v}_3$
- $\dot{y}_1 = \dot{v}_4$

In general, by following the computational graph, we can propose the following formula for the tangent variables:

$$\dot{v}_i = \sum_{j \in \{\text{predecesors of } i\}} \frac{\partial v_i}{\partial v_j} \dot{v}_j \tag{2.1}$$

In the case of the input variables (that don't have predecesors), we notice that the tangent variables, by definition, are always either 0 or 1. This is because the input variables are independent of each other, so their derivatives are 0, except for the variable of interest, which has a derivative of 1, as it is the variable with respect to which we are computing the derivative.

In fact, by setting the tangent variables of the input variables to 0 or 1, we can compute the derivative of the function with respect to any input variable, following the forward tangent trace. This is the main idea of the forward mode of AD.

For example, let us calculate the derivative of the previous function in the point $(x, y) = (1, 2)$ with respect to the variable x_1. We can do this by setting the tangent variables of the input variables to 0 or 1, as well as the primal input variables to their values at the point $(1, 2)$. Then, we can compute the forward primal and tangent traces to get the derivative of the function with respect to x_1:

Forward Primal Trace	**Output**	**Forward Tangent Trace**	**Output**
$v_{-1} = x_1$	1	$\dot{v}_{-1} = \dot{x}_1$	1
$v_0 = x_2$	2	$\dot{v}_0 = \dot{x}_2$	0
$v_1 = v_{-1} + v_0$	$1 + 2 = 3$	$\dot{v}_1 = \dot{v}_{-1} + \dot{v}_0$	$1 + 0 = 1$
$v_2 = v_0^2$	$2^2 = 4$	$\dot{v}_2 = 2v_0 \cdot \dot{v}_0$	$2 \cdot 2 \cdot 0 = 0$
$v_3 = v_1 \cdot v_2$	$3 \cdot 4 = 12$	$\dot{v}_3 = v_2\dot{v}_1 + \dot{v}_2 \cdot v_1$	$4 \cdot 1 + 0 \cdot 3 = 4$
$v_4 = \sin v_3$	$\sin(12) = -0.54$	$\dot{v}_4 = \cos(v_3) \cdot \dot{v}_3$	$\cos(12) \cdot 4 = 3.38$
$y_1 = v_4$	-0.54	$\dot{y}_1 = \dot{v}_4$	3.38

Table 2.1: Example of forward mode of AD

The same can be done, setting the tangent variable $\dot{v}_0 = 1$, to compute the derivative of the function with respect to x_2. The result is $\dot{y}_1 = -13.501$. So in the end, we have computed the derivative of the function with respect to x_1 and x_2 at the point $(1, 2)$, which is 3.38 and -13.501, respectively.

This process is the essence of forward mode AD. At every elementary operation for a given function, compute intermediate variables (primals) by applying basic arithmetic operations, and in synchrony, compute their derivatives (tangents).

We can generalize this procedure to compute Jacobians of functions $f : \mathbb{R}^n \to \mathbb{R}^m$ evaluated at a point $x_0 \in \mathbb{R}^n$. Remember that the Jacobian is a matrix that contains the partial derivatives of the function with respect to each input variable:

$$J_f(x_0) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2.2}$$

To compute the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ evaluated at a point $x_0 \in \mathbb{R}^n$, we can use the forward mode of AD. We set the input variables to x_0, and the tangent variables equal to the canonical basis vectors e_i of $\mathbb{R}^n$, for $i = 1, ..., n$. Every pass through the forward mode of AD for each e_i will give us the i-th column of the Jacobian.

Because the function $f : \mathbb{R}^n \to \mathbb{R}^m$ has n input variables, we need to perform n passes through the forward mode of AD to compute the Jacobian. So this procedure has a complexity of $O(n)$. This is the main drawback of the forward mode of AD: it is not efficient for functions with a large number of input variables, with respect to the output variables, i.e., $n \gg m$.

2.3.1 Note: Forward mode and Jacobian-vector product (JVP)

In practice, we are often interested in computing the product of the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ evaluated at a point $x_0 \in \mathbb{R}^n$ with a vector $r \in \mathbb{R}^n$. This operation is called the Jacobian-vector product, and it is denoted by $J_f(x_0) \cdot r$.

The Jacobian-vector product can be computed using the forward mode of AD. We set the input variables to x_0, and the tangent variables equal to r. Then, a single pass through the forward mode of AD will give us the product of the Jacobian with the vector r. This is a more efficient way to compute the Jacobian-vector product, and sometimes it is more convenient than computing the full Jacobian.

2.4 Backward mode of AD

Backward mode of automatic differentiation (AD), also known as reverse mode of AD, is a technique that computes the derivative of a function by applying the chain rule in reverse order. It is based on the idea of propagating the derivative information from the output variables to the input variables.

This method introduces a new concept: the adjoint variables, usually denoted by $\bar{v}_i$. These adjoints carry the derivative information of a single output variable with respect to a variable (intermediate or input) of interest. For example, if we are interested in computing the derivative of the output variable y_j with respect to the variable v_i, we write:

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i} \quad i = 1-n, ...\ell$$

As in forward mode, in reverse mode we also perform a pass through the evaluation trace of the function. But this time, the adjoint variables are not computed alongside the primal variables. Instead, we store any dependency information that is needed to compute the adjoints in the computational graph (Wengert list). Then we perform a backward pass through the computational graph to compute the adjoints.

Given the computational graph, the adjoint variables can be computed by the following formula:

$$\bar{v}_i = \sum_{j \in \{\text{successors of } i\}} \frac{\partial v_j}{\partial v_i} \bar{v}_j \tag{2.3}$$

In the case of the output variables (that don't have successors), we notice that the adjoint variables are always either 0 or 1. This is because, by definition, the adjoint of the output variables are calculated with respect to the output variables, so their derivatives are 0, except for the variable of interest, which has a derivative of 1, as it is the variable with respect to which we are computing the derivative.

Notice that when we reach to the input variables, we have computed the derivative of the function with respect to all the output variables. This is the main idea of the backward mode of AD. At every elementary operation for a given function, compute intermediate variables (primals) by applying basic arithmetic operations, and store any dependency information that is needed to compute the adjoints. Then, perform a backward pass through the computational graph to compute the adjoints.

For example, consider the same function $f(x, y) = \sin((x_1 + x_2)x_2^2)$. The computational graph of this function is shown in Figure 2.3. The adjoint trace of this function is:

- $\bar{y}_1 = 1$
- $\bar{v}_4 = 1 \cdot \bar{y}_1$
- $\bar{v}_3 = \cos(v_3) \cdot \bar{v}_4$
- $\bar{v}_2 = \bar{v}_3 \cdot v_1$
- $\bar{v}_1 = \bar{v}_3 \cdot v_2$
- $\bar{v}_0 = 2v_0 \cdot \bar{v}_2 + 1 \cdot \bar{v}_1$
- $\bar{v}_{-1} = 1 \cdot \bar{v}_1$

For the same example, let us calculate the derivative of the function in the point $(x, y) = (1, 2)$:

Forward Primal Trace	**Output**	**Backward Adjoint Trace**	**Output**
$v_{-1} = x_1$	1	$\bar{v}_{-1} = 1 \cdot \bar{v}_1$	3.375
$v_0 = x_2$	2	$\bar{v}_0 = 2v_0 \cdot \bar{v}_2 + 1 \cdot \bar{v}_1$	$4 \cdot 2.351 + 3.375 = 13.501$
$v_1 = v_{-1} + v_0$	$1 + 2 = 3$	$\bar{v}_1 = \bar{v}_3 \cdot v_2$	$0.843 \cdot 4 = 3.375$
$v_2 = v_0^2$	$2^2 = 4$	$\bar{v}_2 = \bar{v}_3 \cdot v_1$	$0.843 \cdot 3 = 2.531$
$v_3 = v_1 \cdot v_2$	$3 \cdot 4 = 12$	$\bar{v}_3 = \cos(v_3) \cdot \bar{v}_4$	$\cos(12) \cdot 1 = 0.843$
$v_4 = \sin v_3$	$\sin(12) = -0.54$	$\bar{v}_4 = 1 \cdot \bar{y}_1$	1
$y_1 = v_4$	-0.54	$\bar{y}_1 = 1$	1

Table 2.2: Example of backward mode of AD

Notice that the result of the derivative of the function with respect to x_1 and x_2 at the point $(1, 2)$ is the same as in the forward mode of AD. But in this case, we only needed to perform a single pass through the computational graph to compute the derivatives.

As well as in the forward mode of AD, we can generalize this procedure to compute Jacobians of functions $f : \mathbb{R}^n \to \mathbb{R}^m$ evaluated at a point $x_0 \in \mathbb{R}^n$. We just need to set the input variables to x_0 for the evaluation trace, and the adjoint output variables to the canonical basis vectors e_i of $\mathbb{R}^m$, for $j = 1, ..., m$. Then, perform a single pass through the computational graph to compute the Jacobian. For each e_i, we will get the i-th row of the Jacobian.

Because the function $f : \mathbb{R}^n \to \mathbb{R}^m$ has m output variables, we need to perform m passes through the backward mode of AD to compute the Jacobian. So this procedure has a complexity of $O(m)$. This is the main advantage of the backward mode of AD: it is more efficient for functions with a large number of output variables, with respect to the input variables, i.e., $n \gg m$, which is usually the case in deep learning.

2.4.1 Note: Backward mode and vector-Jacobian product (VJP)

In practice, we are often interested in computing the product of a vector $r \in \mathbb{R}^m$ with the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ evaluated at a point $x_0 \in \mathbb{R}^n$. This operation is called the vector-Jacobian product, and it is denoted by $r^T \cdot J_f(x_0) = J_f(x_0)^T \cdot r$.

In the same way as in the forward mode of AD, the vector-Jacobian product can be computed using the backward mode of AD. We set the input variables to x_0 for the evaluation trace, and the adjoint output variables to r. Then, a single pass through the computational graph will give us the product of the vector r^T with the Jacobian. This is a more efficient way to compute the vector-Jacobian product, and sometimes it is more convenient than computing the full Jacobian.

2.5 Dual numbers

We define a dual number as a pair of real numbers (a, b), where a is the real part, and b is the dual part. It is expressed as follows:

$$z = a + b\varepsilon \tag{2.4}$$

where ε is a number such that $\varepsilon \neq 0 \ \wedge \ \varepsilon^2 = 0$. We can define the operations of addition and multiplication as follows:

- Addition:
$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$

- Multiplication:
$$(a + b\varepsilon) \cdot (c + d\varepsilon) = ac + (ad + bc)\varepsilon$$

Now, let us consider a generic function $f(\cdot)$. We will evaluate the function in a dual number $z = x + \varepsilon$. We can expand the function in a Taylor series around x:

$$f(z) = f(x + \varepsilon) = f(x) + f'(x)\varepsilon + O(\varepsilon^2) \tag{2.5}$$

Because $\varepsilon^2 = 0$, we have that:

$$f(z) = f(x) + f'(x)\varepsilon \tag{2.6}$$

This means that the dual part of the function evaluated in a dual number is the derivative of the function evaluated at the real part of the dual number. This is the key idea behind the dual numbers: we can use them to compute the derivative of a function by evaluating the function in a dual number.

Let us expand the idea to a general dual number $a + b\varepsilon$. We can evaluate the function $f(\cdot)$ in the dual number as follows:

$$f(a + b\varepsilon) = f(a) + f'(a)b\varepsilon \tag{2.7}$$

With this property, we can also compute the derivative of a composite function. Suppose that we have two functions $f(\cdot)$ and $g(\cdot)$. We can compute the derivative of the composite function $h(\cdot) = f(g(\cdot))$ as follows:

$$h(x + \varepsilon) = f(g(x + \varepsilon)) = f(g(x) + g'(x)\varepsilon) = f(g(x)) + f'(g(x))g'(x)\varepsilon \tag{2.8}$$

This means that we can compute the derivative of a composite function by evaluating the functions in dual numbers.

We can also redefine the dual numbers to obtain the second derivative of a function. We just need to define $\varepsilon^2 \neq 0$ and $\varepsilon^3 = 0$.

Chapter 3

Numerical Optimization and Training of Neural Networks

3.1 Introduction: the Perceptron

The first model that introduced the idea of a neural network was the perceptron. The perceptron is a simple model that takes a set of inputs and produces an output. The model is based on the idea of a neuron in the brain, which receives signals from other neurons and produces an output signal.

The perceptron is a linear model that takes a set of binary inputs $x = (x_1, x_2, \ldots, x_n)$ and produces an output y in the following way:

$$y = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + \ldots + w_nx_n + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where $w = (w_1, w_2, \ldots, w_n)$ are the weights of the model and b is the bias. Note that this can be represented as a vector product:

$$y = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

The perceptron is a simple model that can be used to classify data into two classes. The model is trained by adjusting the weights and bias to minimize the error on the training data.

Note that the perceptron is a linear model, which means that it can only learn linearly separable functions. This means that the model can only learn functions that can be separated by a hyperplane. If the data is not linearly separable, the perceptron will not be able to learn the function.

This problem is illustrated by the XOR function, which is not linearly separable.

3.1.1 The XOR problem

The XOR logical function is a simple function that takes two binary inputs and produces an output, which is 1 if the inputs are different and 0 if the inputs are the same. The XOR function is defined as follows:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.1: XOR function

Note that the XOR function is not linearly separable, i.e., it is not possible to separate the data points with a hyperplane. For instance, the following plot shows the XOR function in the input space:

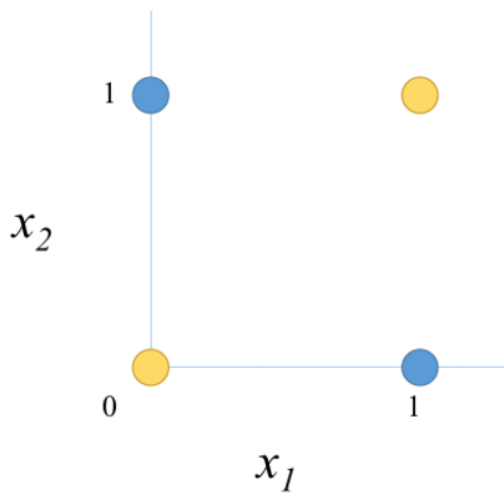

Figure 3.1: XOR function in the input space

As we can see, the data points are not linearly separable, which means that the perceptron alone cannot learn the XOR function. This is a limitation of the perceptron model. However, let us take a look at how we can overcome this limitation by using a more complex model: the multi-layer perceptron.

3.1.2 NAND gate

The NAND gate is a simple logical function that takes two binary inputs and produces an output, which is 0 if both inputs are 1 and 1 otherwise. The NAND gate is defined as follows:

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

Table 3.2: NAND gate

The NAND gate is an example of a function that is linearly separable, i.e., it can be separated by a hyperplane. This means that the perceptron can learn the NAND gate function. Notice that the NAND gate is an universal gate, which means that it can be used to implement any logical function. This is an important property of the NAND gate, as it allows us to build more complex functions using simple building blocks.

Because the NAND gate can represent any logical function by combining multiple gates, and the perceptron can learn the NAND gate, we can use the perceptron to learn any logical function, just by combining multiple perceptrons. This is the raw idea behind the multi-layer perceptron.

3.1.3 The problem of the step function

Notice that the perceptron uses a step function to produce the output. It takes the value 1 if the input is greater than 0 and 0 otherwise. This has some problems, as it implies that possible small changes in the input and weights can lead to a discontinuous change in the output, i.e., big jumps in the output. This can make the optimization process difficult.

To overcome this problem, we can use a different activation functions, such as the sigmoid function, which is a smooth function that takes values between 0 and 1. The idea of using a continuous activation function, combined with multiple layers of perceptrons, is the basis of the multi-layer perceptron.

3.2 The Multi-Layer Perceptron

The multi-layer perceptron (MLP) is a neural network model that consists of multiple layers of perceptrons. On the output layer, we use a different activation function than the step function, such as the sigmoid function. This allows the model to learn more complex functions, as it can approximate any continuous function.

We can represent the MLP model using a graph, where each node represents a perceptron and each edge represents a weight. The graph is organized in layers, where each layer is connected to the next layer. The input layer represents the input data, the output layer represents the output of the model, and the hidden layers represent the intermediate layers of the model.

Figure 3.2: Graph representation of the MLP model

For a neuron j in layer ℓ, we have the following:

$$z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell \tag{3.3}$$

where z_j^ℓ is the weighted sum of the inputs of neuron j in layer ℓ, w_{jk}^ℓ is the weight of the connection between neuron k in layer $\ell - 1$ and neuron j in layer ℓ, $a_k^{\ell-1}$ is the output after activation of neuron k in layer $\ell - 1$, and b_j^ℓ is the bias of neuron j in layer ℓ.

Notice that in each output layer, we apply an activation function to the weighted sum of the inputs. This activation function is a non-linear function that allows the model to learn complex functions. This is represented by:

$$a_j^\ell = \sigma(z_j^\ell) \tag{3.4}$$

where σ is the activation function. In general, we can represent the weights as a matrix W^ℓ and the biases as a vector b^ℓ. This allows us to write the equation that represents the output of a particular layer as follows:

$$a^\ell = \sigma(W^\ell a^{\ell-1} + b^\ell) \tag{3.5}$$

where a^ℓ is the output of layer ℓ, $a^{\ell-1}$ is the output of layer $\ell - 1$, and σ is the activation function. This equation represents the forward pass of the model, i.e., the process of computing the output of the model given an input.

Now, on the output layer, we need a function that can represent the distance between the output of the model and the true output. This function is called the loss function, and it measures the error of the model. The main goal of training a neural network is to minimize the loss function.

3.3 Training a Neural Network

To train a neural network, we need to define a loss function that measures the error of the model. This function takes the output of the model and the true output and produces a value that represents how well the model is performing. In general, a loss function J should have the following properties:

1. J can be written as an average of the loss over the training data.

$$J = \frac{1}{N} \sum_{i=1}^{N} J_s$$

   where N is the number of samples in the training data and J_s is the loss for sample s.

2. J is a function only of the output of the model and the true output.

Now, to train the model, we need to minimize the loss function. This is done by adjusting the weights and biases of the model to reduce the error. This process is called optimization. There are many optimization algorithms that can be used to train a neural network, but in general, all of them follow the same idea: compute the gradient of the loss function with respect to the weights and biases and update the weights and biases in the opposite direction of the gradient. This is called gradient descent.

Therefore, our aim is to compute the derivative of the loss function with respect to the weights and biases of the model, i.e., we need to compute:

$$\frac{\partial J}{\partial k_{jk}^{\ell}}, \frac{\partial J}{\partial b_j^{\ell}} \quad \forall \ell, j, k \tag{3.6}$$

where k_{jk}^{ℓ} is the weight of the connection between neuron k in layer $\ell - 1$ and neuron j in layer ℓ, and b_j^{ℓ} is the bias of neuron j in layer ℓ.

For this, let us define the vector of errors (or sensitivities) δ^{ℓ} for layer ℓ as follows:

$$\delta_j^{\ell} = \frac{\partial J}{\partial z_j^{\ell}} \quad \forall j \tag{3.7}$$

where z_j^{ℓ} is the weighted sum of the inputs of neuron j in layer ℓ. The vector of errors δ^{ℓ} represents how much the error changes with respect to the weighted sum of the inputs of neuron j in layer ℓ.

Now, we will deduce 4 funcamental equations that will allow us to compute the gradient of the loss function with respect to the weights and biases of the model. These equations are called the backpropagation equations, and they are the key to training a neural network.

3.3.1 Backpropagation equations

1. Compute the error of the output layer δ^L:

$$\delta_j^L = \frac{\partial J}{\partial z_j^L} = \sum_k \frac{\partial J}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_j^L}$$

Because a_k^L only depends on z_k^L if $k = j$, we have:

$$\delta_j^L = \frac{\partial J}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L}$$

Notice that:

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$$

where σ' is the derivative of the activation function. Then, we have:

$$\delta_j^L = \frac{\partial J}{\partial a_j^L} \cdot \sigma'(z_j^L)$$

Notice that the value of z_j^L has already been computed in the forward pass. Rewriting the equation in vector form, we have:

$$\delta^L = \nabla_{a^L} J \odot \sigma'(z^L) \tag{3.8}$$

where $\nabla_a J$ is the gradient of the loss function with respect to the output of the model and $\odot$ is the element-wise product.

2. Compute the error of the hidden layers δ^ℓ with respect to $\delta^{\ell+1}$:

$$\delta_j^\ell = \frac{\partial J}{\partial z_j^\ell} = \sum_k \frac{\partial J}{\partial z_k^{\ell+1}} \cdot \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell}$$

Notice that:

$$\frac{\partial J}{\partial z_k^{\ell+1}} = \delta_k^{\ell+1}$$

Then, we have:

$$\delta_j^\ell = \sum_k \delta_k^{\ell+1} \cdot \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell}$$

Now, we need to compute the derivative of $z_k^{\ell+1}$ with respect to z_j^ℓ. Notice that:

$$z_k^{\ell+1} = \sum_m w_{km}^{\ell+1} a_m^\ell + b_k^{\ell+1} = \sum_m w_{km}^{\ell+1} \sigma(z_m^\ell) + b_k^{\ell+1}$$

Then, we have:

$$\frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = w_{kj}^{\ell+1} \cdot \sigma'(z_j^\ell)$$

Finally, we have:

$$\delta_j^\ell = \sum_k \delta_k^{\ell+1} \cdot w_{kj}^{\ell+1} \cdot \sigma'(z_j^\ell)$$

Rewriting the equation in vector form, we have:

$$\delta^\ell = ((W^{\ell+1})^T \delta^{\ell+1}) \odot \sigma'(z^\ell) \tag{3.9}$$

3. Compute the gradient of the loss function with respect to b^ℓ:

$$\frac{\partial J}{\partial b_j^\ell} = \frac{\partial J}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial b_j^\ell}$$

Notice that:

$$z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell$$

Then, we have:

$$\frac{\partial z_j^\ell}{\partial b_j^\ell} = 1$$

Finally, we have:

$$\frac{\partial J}{\partial b_j^\ell} = \delta_j^\ell$$

Rewriting the equation in vector form, we have:

$$\nabla_{b^\ell} J = \delta^\ell \tag{3.10}$$

4. Compute the gradient of the loss function with respect to W^ℓ:

$$\frac{\partial J}{\partial w_{jk}^\ell} = \frac{\partial J}{\partial z_j^\ell} \cdot \frac{\partial z_j^\ell}{\partial w_{jk}^\ell}$$

Notice that:

$$z_j^\ell = \sum_k w_{jk}^\ell a_k^{\ell-1} + b_j^\ell$$

Then, we have:

$$\frac{\partial z_j^\ell}{\partial w_{jk}^\ell} = a_k^{\ell-1}$$

Finally, we have:

$$\frac{\partial J}{\partial w_{jk}^\ell} = \delta_j^\ell \cdot a_k^{\ell-1}$$

Rewriting the equation in matrix form, we have:

$$\nabla_{W^\ell} J = \delta^\ell (a^{\ell-1})^T \tag{3.11}$$

These are the backpropagation equations, which allow us to compute the gradient of the loss function with respect to the weights and biases of the model. This is the key to training a neural network. By computing the gradient of the loss function and updating the weights and biases in the opposite direction of the gradient, we can minimize the error of the model and learn complex functions.

Summarizing the equations:

1. Compute the error of the output layer:

$$\delta^L = \nabla_{a^L} J \odot \sigma'(z^L)$$

2. Compute the error of the hidden layers:

$$\delta^\ell = ((W^{\ell+1})^T \delta^{\ell+1}) \odot \sigma'(z^\ell)$$

3. Compute the gradient of the loss function with respect to b^ℓ:

$$\nabla_{b^\ell} J = \delta^\ell$$

4. Compute the gradient of the loss function with respect to W^ℓ:

$$\nabla_{W^\ell} J = \delta^\ell (a^{\ell-1})^T$$

Then, the backpropagation algorithm can be summarized as follows:

1. x is the input to the model. Compute a^1
2. For each layer ℓ from 2 to L:
   (a) Compute $z^\ell = W^\ell a^{\ell-1} + b^\ell$
   (b) Compute $a^\ell = \sigma(z^\ell)$
3. Compute the loss function J
4. Compute the error of the output layer δ^L
5. For each layer ℓ from $L-1$ to 1:
   (a) Compute the error of the hidden layers δ^ℓ
   (b) Compute the gradient of the loss function with respect to b^ℓ
   (c) Compute the gradient of the loss function with respect to W^ℓ
   (d) Update the weights and biases:
      - $W^\ell = W^\ell - \alpha \nabla_{W^\ell} J$
      - $b^\ell = b^\ell - \alpha \nabla_{b^\ell} J$

   where α is the learning rate.

3.4 Activation functions

The activation function of a neuron is a non-linear function that takes the weighted sum of the inputs of the neuron and produces an output. The activation function is a key component of a neural network, as it allows the model to learn complex functions. There are many activation functions that can be used in a neural network, but some of the most common are:

1. Sigmoid function:
$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{3.12}$$
The sigmoid function takes values between 0 and 1, which makes it useful for binary classification problems. However, the sigmoid function has some problems, such as the vanishing gradient problem, which can make training difficult.

2. Hyperbolic tangent function:
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.13}$$
The hyperbolic tangent function is similar to the sigmoid function, but it takes values between -1 and 1. This can make training easier, as the output is centered around 0. However, the hyperbolic tangent function also has the vanishing gradient problem.

3. Rectified Linear Unit (ReLU) function:
$$\mathrm{ReLU}(z) = \max(0, z) \tag{3.14}$$
The ReLU function is a simple function that takes the maximum between 0 and the input. The ReLU function is widely used in practice, as it is simple and efficient. However, the ReLU function has some problems, such as the dying ReLU problem, which can make some neurons inactive.

4. Leaky ReLU function:
$$\text{Leaky ReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{otherwise} \end{cases} \tag{3.15}$$
The Leaky ReLU function is a variant of the ReLU function that allows a small gradient when the input is negative. This can help to overcome the dying ReLU problem.

5. Swish function (Google):
$$\mathrm{swish}(z) = \frac{z}{1-e^{-z}} \tag{3.16}$$
The swish function is a new activation function that has been proposed recently. The swish function is similar to the sigmoid function, but it has a different shape that can make training easier. The swish function has been shown to perform well on a variety of tasks.

6. Softmax function:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{3.17}$$

The softmax function is a generalization of the sigmoid function that takes a vector of inputs and produces a vector of outputs that sum to 1. The softmax function is often used in the output layer of a neural network to produce probabilities for a classification problem. The softmax function is useful when the output of the model needs to be interpreted as a probability distribution.