Machine Learning Basics

Zichao Yang

Zhongnan University of Economics & Law

Date: September 6, 2024

Textbooks

1. Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

2. Géron, Aurélien. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2022.

3. Nielsen, Michael. *Neural Networks and Deep Learning*. Online Book, 2019.

4. Stevens, Eli, Luca Antiga, and Thomas Viehmann. *Deep learning with PyTorch*. Manning Publications, 2020.

5. Zhang, Aston, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into deep learning*. English Version, Chinese Version, 2021.

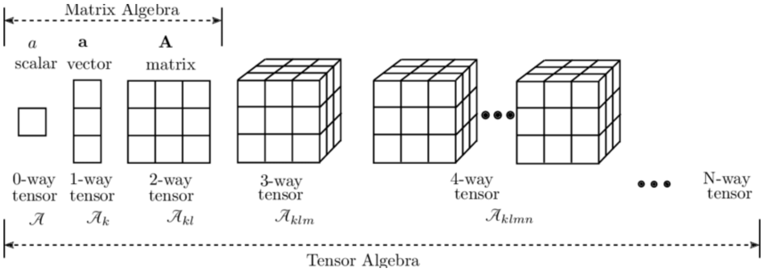
Roadmap

- Linear Algebra
- Probability and Information Theory
- Numerical Computation
- Machine Learning Basics

Part I. Linear Algebra

Scalars, Vectors, Matrices and Tensors

Scalars: A scalar is just a single number.
Vectors: A vector is an array of numbers.
Matrices: A matrix is a 2-D array of numbers.
Tensors: A tensor is a multi-dimensions array of numbers.


Matrix Algebra
a scalar
$\mathbf{a}$ vector
$\mathbf{A}$ matrix
0-way tensor $\mathcal{A}$
1-way tensor $\mathcal{A}_k$
2-way tensor $\mathcal{A}_{kl}$
3-way tensor $\mathcal{A}_{klm}$
4-way tensor $\mathcal{A}_{klmn}$
N-way tensor
Tensor Algebra

Source: Shulga, Dmytro, et al. "Tensor b-spline numerical methods for pdes: a high-performance alternative to fem." arXiv preprint arXiv:1904.03057 (2019).

Adding Matrices

We can add matrices to each other, as long as they have the same shape:

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$$

We can also add a scalar to a matrix or multiply a matrix by a scalar:

$$\boldsymbol{D} = a \cdot \boldsymbol{B} + c$$

In machine learning, we also allow the addition of a matrix and a vector, this operation is called **broadcasting**.

$$\boldsymbol{E} = \boldsymbol{A} + \boldsymbol{b}$$

Try $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 \\ 6 \end{bmatrix}$ in python to see what will happen?

Multiplying Matrices

Matrix Product (Matrix Multiplication): this is what you learned in your math class.

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$$

Element-Wise Product (Hadamard Product):

$$\boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B}$$

Matrix multiplication is:
(1) distributive: $\boldsymbol{A}(\boldsymbol{B}+\boldsymbol{C}) = \boldsymbol{A}\boldsymbol{B}+\boldsymbol{A}\boldsymbol{C}$
(2) associative: $\boldsymbol{A}(\boldsymbol{B}\boldsymbol{C}) = (\boldsymbol{A}\boldsymbol{B})\boldsymbol{C}$
(3) **not** commutative: $\boldsymbol{A}\boldsymbol{B} \neq \boldsymbol{B}\boldsymbol{A}$

Matrix transpose: $(\boldsymbol{A}\boldsymbol{B})^\intercal = \boldsymbol{B}^\intercal \boldsymbol{A}^\intercal$

Let's try them out in python!

Identity and Inverse Matrices

An **identity matrix**, $\boldsymbol{I}_n \in \mathbb{R}^{n \times n}$, is a matrix that does not change any vector when we multiply that vector by this matrix.

$$\boldsymbol{I}_n \boldsymbol{x} = \boldsymbol{x}, \forall \boldsymbol{x} \in \mathbb{R}^n$$

Identity matrix: all the entries along the main diagonal are 1, and all the other entries are 0.

$$\boldsymbol{I}_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Python code: identity_matrix = np.eye(n)

Identity and Inverse Matrices

The **matrix inverse** of $\boldsymbol{A}$ is denoted as $\boldsymbol{A}^{-1}$, and it is defined as the matrix such that

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n$$

We now know enough linear algebra notation to write down a system of linear equations:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

where $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is a known matrix (m obs with n features), $\boldsymbol{b} \in \mathbb{R}^m$ is a known vector (dependent variable), and $\boldsymbol{x} \in \mathbb{R}^n$ is a vector of unknown variables we would like to solve (coefficients). Solve $\boldsymbol{x}$ as:

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

Obvious, this process depends on the existence of $\boldsymbol{A}^{-1}$.

Linear Dependence and Span

For $\boldsymbol{A}^{-1}$ to exist, $\boldsymbol{Ax} = \boldsymbol{b}$ must have exactly **one** solution for every value of $\boldsymbol{b}$.

However, it is possible for this system of equations to have no solutions or infinitely many solutions.

Q: can this system of equations have many but not infinite solutions?

Linear Dependence and Span

Q: can this system of equations have many but not infinite solutions?

A: it is not possible. if both $\boldsymbol{x}$ and $\boldsymbol{y}$ are solutions, then

$$\boldsymbol{z} = \alpha \boldsymbol{x} + (1 - \alpha)\boldsymbol{y}$$

is also a solution for any real α.

Linear Dependence and Span

We can think each column inside $\boldsymbol{A}$ as a direction we can travel in from the origin point to reach $\boldsymbol{b}$, and each element of $\boldsymbol{x}$ specifies how far we should travel in each of these directions:

$$\boldsymbol{A}\boldsymbol{x} = \sum_i^n x_i \boldsymbol{A}_{:,i}$$

The **span** of a set of vectors (i.e., $\{\boldsymbol{A}_{:,i}\}, i = 1, 2, ..., n$) is the set of all points obtainable by linear combination fo the vectors.

Determining whether $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ has a solution thus equals to testing whether $\boldsymbol{b}$ is in the span of the columns of $\boldsymbol{A}$. This particular span is known as the **column space**, or the **range**, of $\boldsymbol{A}$.

Linear Dependence and Span

Condition 1: we know that $\boldsymbol{b} \in \mathbb{R}^m$. If we want to at least have a solution, we need to require that the column space of $\boldsymbol{A}$ be all of $\mathbb{R}^m$, implying that $\boldsymbol{A}$ must have at least m columns, that is, $n \geq m$.

Condition 1 is a **necessary but not sufficient** condition. It is possible for some of the columns to be redundant, then the column space of $\boldsymbol{A}$ would not be able to cover the whole value space of $\boldsymbol{b} \in \mathbb{R}^m$. This kind of redundancy is known as **linear dependence**.

Condition 2: for the column space of $\boldsymbol{A}$ to encompass all of $\mathbb{R}^m$, the matrix must contain at least one set of m linearly **independent** columns.

Condition 2 is a **necessary and sufficient** condition for $\boldsymbol{Ax} = \boldsymbol{b}$ to have **at least one** solution for every value of $\boldsymbol{b}$.

Linear Dependence and Span

Condition for having an inverse:

For the matrix $\boldsymbol{A}$ to have an inverse, we additionally need to ensure that $\boldsymbol{Ax} = \boldsymbol{b}$ has **at most** one solution for each value of $\boldsymbol{b}$. To do so, we need to make sure that $\boldsymbol{A}$ has at most m columns. Otherwise, there is more than one way of parameterizing each solution.

Hence, the additional condition means that $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ must be **square** (i.e., $n = m$) and all the columns should be linear independent.

A square matrix with linearly dependent columns is known as **singular**.

Note: $\boldsymbol{A}^{-1}$ is primarily useful as a theoretical tool, and is not widely used in practice due to limited precision on a digital computer.

Norms

In machine learning, we usually measure the size of vectors using a function called a **norm**. Formally, the L^p norm is given by:

$$||\boldsymbol{x}||_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}$$

for $p \in \mathbb{R}, p \geq 1$.

A norm is a function that satisfies the following properties:

- $f(\boldsymbol{x}) = 0 \rightarrow \boldsymbol{x} = \boldsymbol{0}$
- $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$
- $f(\alpha \boldsymbol{x}) = |\alpha| f(\boldsymbol{x})$

L^2, L^1 and L^∞ norms

The L^2 norm, $||\boldsymbol{x}||_2$, is known as the **Euclidean norm**, this norm is frequently used in machine learning and is often denoted simply as $||\boldsymbol{x}||$. It is also common to use the **squared** L^2 norm, $||\boldsymbol{x}||_2^2$, which can be calculated simply as:

$$||\boldsymbol{x}||_2^2 = \boldsymbol{x}^\intercal \boldsymbol{x}$$

In many contexts, $||\boldsymbol{x}||_2^2$ may be undesirable because it increases very slowly near the origin. In these cases, we turn to a function that grows at the same rate in all locations, the L^1 norm:

$$||\boldsymbol{x}||_1 = \sum_i |x_i|$$

L^2, L^1 and L^∞ norms

Another norm commonly used in machine learning is the L^∞ norm, also known as the **max norm**. This norm simplifies to the absolute value of the element with the largest magnitude in the vector $\boldsymbol{x}$:

$$||\boldsymbol{x}||_\infty = max(|x_1|, ..., |x_n|)$$

Sometimes, we may also want to measure the size of a matrix, and the common way to do this is using **Frobenius norm**:

$$||\boldsymbol{A}||_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

The dot product of two vectors can be written in terms of norms:

$$\boldsymbol{x}^\intercal \boldsymbol{y} = ||\boldsymbol{x}||_2 ||\boldsymbol{y}||_2 \cos\theta$$

where θ is the angle between $\boldsymbol{x}$ and $\boldsymbol{y}$.

Special Kinds of Matrices and Vectors

Diagonal Matrices

A matrix $\boldsymbol{D}$ is a diagonal matrix iff $D_{i,j} = 0$ for all $i \neq j$. We usually write $\mathrm{diag}(\boldsymbol{v})$ to denote a square diagonal matrix whose diagonal entries are given by a vector $\boldsymbol{v}$.

$$\mathrm{diag}(\boldsymbol{v})\boldsymbol{x} = \boldsymbol{v} \odot \boldsymbol{x}$$

Inverting a square diagonal matrix is also efficient. The inverse exists only if every diagonal entry is nonzero, and in that case we have:

$$\mathrm{diag}(\boldsymbol{v})^{-1} = \mathrm{diag}([1/v_1, ..., 1/v_n]^\mathsf{T})$$

In machine learning, we may want to obtain a less expensive algorithm by restricting some matrices to be diagonal.

Special Kinds of Matrices and Vectors

Symmetric Matrices

A symmetric matrix is any matrix that is equal to its own transpose:

$$\boldsymbol{A} = \boldsymbol{A}^\intercal$$

Symmetric matrices are used when the entries are generated by some function of two arguments that does not depend on the order of the arguments. For example, a symmetric matrix is used to describe an undirected network.

Special Kinds of Matrices and Vectors

Orthogonal Matrices

A **unit vector** is a vector with **unit norm**: $||\boldsymbol{x}||_2 = 1$
A vector $\boldsymbol{x}$ and a vector $\boldsymbol{y}$ are **orthogonal** to each other if: $\boldsymbol{x}^\intercal \boldsymbol{y} = 0$
If the vectors not only are orthogonal but also have unit norm, we call them **orthonormal**.

An **orthogonal matrix** is a **square** matrix whose rows are mutually **orthonormal** and whose columns are mutually **orthonormal**:

$$\boldsymbol{A}^\intercal \boldsymbol{A} = \boldsymbol{A}\boldsymbol{A}^\intercal = \boldsymbol{I}$$

And this implies that:

$$\boldsymbol{A}^{-1} = \boldsymbol{A}^\intercal$$

Orthogonal matrices are of interest because their inverse is very cheap to compute. **Note:** there is no special term for a matrix whose rows (columns) are **orthogonal** but not **orthonormal**.

Eigendecomposition

Why do we want to do eigendecomposition?

Sometimes, we want to decompose matrices to show us information about their functional properties that is not obvious from the representation of the matrix as an array of elements.

One of the most widely used matrix decomposition methods is called **eigendecomposition**, in which we decompose a matrix into a set of eigenvectors and eigenvalues.

Eigendecomposition

An **eigenvector** of a square matrix $\boldsymbol{A}$ is a nonzero vector $\boldsymbol{v}$ such that multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$:

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v}$$

where λ is known as the **eigenvalue** corresponding to the eigenvector.

Meanwhile, if $\boldsymbol{v}$ is an eigenvector of $\boldsymbol{A}$, so is any re-scaled vector $s\boldsymbol{v}$ for $s \in \mathbb{R}, s \neq 0$ and $s\boldsymbol{v}$ still has the same eigenvalue. For this reason, we usually look only for unit eigenvectors.

Eigendecomposition

Suppose that a matrix $\boldsymbol{A}$ has n linearly independent eigenvectors $\{\boldsymbol{v}^{(1)}, ..., \boldsymbol{v}^{(n)}\}$ with corresponding eigenvalue $\{\lambda_1, ..., \lambda_n\}$. We can concatenate them to form a matrix $\boldsymbol{V}$ with one eigenvector per columns: $\boldsymbol{V} = [\boldsymbol{v}^{(1)}, ..., \boldsymbol{v}^{(n)}]$, and a vector $\boldsymbol{\lambda} = [\lambda_1, ..., \lambda_n]^\intercal$. The **eigendecomposition** of $\boldsymbol{A}$ is then given by:

$$\boldsymbol{A} = \boldsymbol{V} \operatorname{diag}(\boldsymbol{\lambda}) \boldsymbol{V}^{-1}$$

If any two or more eigenvectors share the same eigenvalue, then any set of orthogonal vectors lying in their span are also eigenvectors with that eigenvalue. Hence, the eigendecomposition is unique only if all the eigenvalues are unique.

Eigendecomposition

The matrix is **singular** iff any of the eigenvalues are zero.

A matrix whose eigenvalues are all positive (negative) is called **positive (negative) definite**.

A matrix whose eigenvalues are all positive (negative) or zero valued is called **positive (negative) semidefinite**.

Positive semidefinite matrices are interesting because they guarantee that $\forall \boldsymbol{x}, \boldsymbol{x}^\intercal \boldsymbol{A}\boldsymbol{x} \geq 0$.

Positive definite matrices additionally guarantee that $\boldsymbol{x}^\intercal \boldsymbol{A}\boldsymbol{x} = 0 \Rightarrow \boldsymbol{x} = \boldsymbol{0}$.

Singular Value Decomposition

Every real **symmetric matrix** can be decomposed into an expression using only real-valued eigenvectors and eigenvalues. How to decompose matrix that we cannot find real-value eigenvectors?

The **singular value decomposition**(SVD) provides another way to factorize a matrix, into **singular vectors** and **singular values**.

Every real matrix (no need to be a square matrix !) has a singular value decomposition.

Singular Value Decomposition

The singular value decomposition can be written as:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{V}^\intercal$$

Suppose that $\boldsymbol{A}$ is an $m \times n$ matrix. Then $\boldsymbol{U}$ is defined to be an $m \times m$ matrix, $\boldsymbol{D}$ to be an $m \times n$ matrix, and $\boldsymbol{V}$ to be an $n \times n$ matrix. The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are both defined to be orthogonal matrices. The matrix $\boldsymbol{D}$ is defined to be a diagonal matrix.

The elements along the diagonal of $\boldsymbol{D}$ are called **singular values** of the matrix $\boldsymbol{A}$. The columns of $\boldsymbol{U}$ ($\boldsymbol{V}$) are known as the **left(right)-singular vectors**.

The most useful feature of the SVD is that we can use it to partially generalize matrix inversion to non-square matrices.

The Moore-Penrose Pseudoinverse

Matrix inversion is not defined for matrices that are not square. The **Moore-Penrose Pseudoinverse** enables us to extract a pseudoinverse of a non-square matrix.

The pseudoinverse of $\boldsymbol{A}$ is defined as a matrix:

$$\boldsymbol{A}^{+} = \lim_{\alpha \searrow 0} (\boldsymbol{A}^{\intercal}\boldsymbol{A} + \alpha \boldsymbol{I})^{-1}\boldsymbol{A}^{\intercal}$$

However, practical algorithm for computing the pseudoinverse are based not on this definition, bu rather on the formula:

$$\boldsymbol{A}^{+} = \boldsymbol{U}\boldsymbol{D}^{+}\boldsymbol{U}^{\intercal}$$

The Moore-Penrose Pseudoinverse

When $\boldsymbol{A}$ has more columns than rows, then solving a linear equation using the pseudoinverse provides one of the many possible solutions. Specifically, it provides the solution $\boldsymbol{x} = \boldsymbol{A}^{+}\boldsymbol{y}$ with minimal Euclidean norm $||\boldsymbol{x}||_2$ among all possible solutions.

When $\boldsymbol{A}$ has more rows than columns, it is possible for there to be no solution. In this case, using the pseudoinverse gives us the $\boldsymbol{x}$ for which $\boldsymbol{A}\boldsymbol{x}$ is as close as possible to y in terms of Euclidean norm $||\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}||_2$.

The Trace Operator

The **trace operator** gives the sum of all the diagonal entries of a matrix:

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_i \boldsymbol{A}_{i,i}$$

Why is the trace operator useful?

Some operations that are difficult to specify without resorting to summation notation can be specified using matrix products and the trace operator. For example, we can rewrite the Frobenius norm of a matrix using the trace operator:

$$||\boldsymbol{A}||_F = \sqrt{\sum_{i,j} A_{i,j}^2} = \sqrt{\mathrm{Tr}(\boldsymbol{A}\boldsymbol{A}^\intercal)}$$

The Trace Operator

The trace operator has some properties:

(1) The trace operator is invariant to the transpose operator:

$$\mathrm{Tr}(\boldsymbol{A}) = \mathrm{Tr}(\boldsymbol{A}^{\mathsf{T}})$$

(2) The trace of a square matrix composed of many factors is invariant to moving the last factor into the first position, as long as the resulting product is defined:

$$\mathrm{Tr}(\boldsymbol{ABC}) = \mathrm{Tr}(\boldsymbol{CAB}) = \mathrm{Tr}(\boldsymbol{BCA})$$

(3) The invariance to cyclic permutation holds even if the resulting product has a different shape. For example, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{n \times m}$, we have:

$$\mathrm{Tr}(\boldsymbol{AB}) = \mathrm{Tr}(\boldsymbol{BA})$$

Part II. Probability and Information Theory

Why Probability?

Machine Learning must always deal with uncertain quantities and sometimes stochastic quantities. There are three possible sources of uncertainty:

- Inherent stochasticity in the system being modeled.
- Incomplete observability.
  Even deterministic systems can appear stochastic when we cannot observe all the variables that drive the behavior of the system.
- Incomplete modeling.
  When we use a model that must discard some of the information we have observed, the discarded information results in uncertainty in the model's prediction.

Why Probability?

In many cases, it is more practical to use a simple but uncertain rule rather than a complex and certain one, because it is expensive to develop and maintain systems that can accommodate complex rules.

There are two kinds of interpretation of probability:

- **Frequentist Probability**: interprets probability as the rates at which events occur. e.g., If we toss a coin, the probability of getting the tail is 0.5. Here we use probability to describe the results of repeatable events.
- **Bayesian Probability**: interprets probability as the qualitative levels of certainty. e.g., The patient has a 90 percent chance of fully recovering from the flu. Here the concept of probability denotes a degree of belief.

Random Variable

A **random variable** is a variable that can take on different values randomly. For example, x_1 and x_2 are both possible values that the random variable x can take on.

On its own, a random variable is just a description of the states that are possible; it must be coupled with a probability distribution that specifies how likely each of these states are.

Probability Distributions

A **probability distribution** is a description of how likely a **random variable** or set of random variables is to take on each of its possible states.

The way we describe probability distributions depends on whether the variables are discrete or continuous.

(1) Discrete Variable : Probability Mass Function

(2) Continuous Variable : Probability Density Function

Discrete Variable and Probability Mass Function

A probability distribution over a **discrete** random variable, x, may be described using a **probability mass function**(PMF), $P(\mathrm{x})$. This function maps a state of a random variable to the probability of that random variable taking on that state.

PMF can act on many variables at the same time. Such a probability distribution over many variables is known as a **joint probability distribution**, $P(\mathrm{x} = x, \mathrm{y} = y)$.

Discrete Variable and Probability Mass Function

To be a PMF on a random veriable x, a function P must satisfy the following properties:

- The domain of P must be the set of all possible states of x.
- $\forall x \in \mathrm{x}$, $0 \leq P(x) \leq 1$. An impossible event has probability 0, and no state can be less probable than that. Likewise, an event that is guaranteed to happen has probability 1, and no state can have a greater chance of occurring.
- $\sum_{x \in \mathrm{x}} P(x) = 1$. We refer to this property as being **normalized**.

Continuous Variables and Probability Density Functions

A probability distribution over a **continuous** random variable, x, may be described using a **probability density function**(PDF), $p(\mathrm{x})$.

Similar to PMF, to be a PDF, the function p must satisfy the following properties:

- The domain of p must be the set of all possible states of x.
- $\forall x \in \mathrm{x}$, $p(x) \geq 0$. **Note**: here we do **NOT** require $p(x) \leq 1$.
- $\int p(x)dx = 1$.

A PDF does not give the probability of a specific state directly. Instead, the probability of landing inside an infinitesimal (aka, extremely small) region with volume δx is given by $p(x)\delta x$.

Marginal Probability

Sometimes we know the probability distribution over a set of variables and we want to know the **probability distribution** (NOT a single probability) over a subset of them. Then we need to turn to the concept of **marginal probability distribution**.

CASE 1: suppose we know the PMF for discrete random variables x and y, $P(\mathrm{x}, \mathrm{y})$. Then we can find $P(\mathrm{x})$ using the sum rule:

$$P(\mathrm{x} = x) = \sum_{y} P(\mathrm{x} = x, \mathrm{y} = y), \qquad \forall x \in \mathrm{x}$$

CASE 2: suppose we know the PDF for continuous random variables, we need to use integration instead of summation:

$$p(x) = \int p(x, y) dy$$

Conditional Probability

In some cases, we are interested in the probability of some events, given that some other events have happened. This is called a **conditional probability**.

We denote the conditional probability that $\mathrm{y} = y$ given $\mathrm{x} = x$ as $P(\mathrm{y} = y | \mathrm{x} = x)$.

The conditional probability can be computed with the formula:

$$P(\mathrm{y} = y | \mathrm{x} = x) = \frac{P(\mathrm{y} = y, \mathrm{x} = x)}{P(\mathrm{x} = x)}$$

Note: the conditional probability is only defined when $P(\mathrm{x} = x) > 0$. We cannot compute the conditional probability conditioned on an event that never happens.

Bayes' Rule

We often find ourselves in a situation where we know $P(\mathrm{y}|\mathrm{x})$ and need to know $P(\mathrm{x}|\mathrm{y})$. If we happen to know $P(\mathrm{x})$, we can compute the desired quantity using **Bayes' rule**:

$$P(\mathrm{x}|\mathrm{y}) = \frac{P(\mathrm{x})P(\mathrm{y}|\mathrm{x})}{P(\mathrm{y})}$$

Note: It is usually feasible to compute $P(\mathrm{y}) = \sum_x P(\mathrm{y}|x)P(x)$, so we do not need to begin with knowledge of $P(\mathrm{y})$.

The Chain Rule of Conditional Probabilities

Any joint probability distribution over many random variables may be decomposed into conditional distributions over only one variable:

$$P(\mathrm{x}^{(1)}, ..., \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)}) \prod_{i=2}^{n} P(\mathrm{x}^{(i)} | \mathrm{x}^{(1)}, ..., \mathrm{x}^{(i-1)})$$

This observation is known as the **chain rule**, or **product rule**.

After imposing the assumption of **Markov process**, the chain rule can be significantly simplified, and it is the foundation of some **natural language processing** (NLP) models.

Independence and Conditional independence

Two random variables x and y are **independent** if:

$$p(\mathrm{x}=x,\mathrm{y}=y)=p(\mathrm{x}=x)p(\mathrm{y}=y), \qquad \forall x\in\mathrm{x}, y\in\mathrm{y}$$

Two random variables x and y are **conditionally independent** given a random variable z if:

$$p(\mathrm{x}=x,\mathrm{y}=y|\mathrm{z}=z)=p(\mathrm{x}=x|\mathrm{z}=z)p(\mathrm{y}=y|\mathrm{z}=z), \forall x\in\mathrm{x}, y\in\mathrm{y}, z\in\mathrm{z}$$

$\mathrm{x}\perp\mathrm{y}$ denotes that x and y are independent.

$\mathrm{x}\perp\mathrm{y}|\mathrm{z}$ denotes that x and y are conditional independent given z.

Expectation, Variance and Covariance

The **expectation** of some function $f(x)$ with respect to a probability distribution $P(\mathrm{x})$ is the average, or mean value, that f takes on when x is drawn from P.

$$\mathbb{E}_{\mathrm{x}\sim P}[f(x)] = \sum_{x} P(x) f(x)$$

$$\mathbb{E}_{\mathrm{x}\sim p}[f(x)] = \int p(x) f(x) dx$$

When the distribution and the random variable are both clear, we can omit the subscript, and write the expectation as $\mathbb{E}[f(x)]$.

Expectation is linear: $\mathbb{E}[\alpha f(x) + \beta g(x)] = \alpha\mathbb{E}[f(x)] + \beta\mathbb{E}[g(x)]$ when α and β are not dependent on x.

Expectation, Variance and Covariance

The **Variance** gives a measure of how much the values of a function of a random variable x vary as we sample different values of x from its probability distribution:

$$\mathrm{Var}(f(x)) = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right]$$

The square root of the variance is known as the **standard deviation**.

The **covariance** gives some sense of how much two values are linearly related to each other, as well as the scale of these variables:

$$\mathrm{Cov}(f(x), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]$$

High absolute values of the covariance mean that the values change very much and are both far from their respective means at the same time.

Expectation, Variance and Covariance

If we only care about how much two values are linearly related and do not care about the scale, we can normalize the covariance and get the **correlation**:

$$\rho_{f(x),g(y)} = \frac{\mathrm{Cov}(f(x), g(y))}{\mathrm{Var}(f(x)) \cdot \mathrm{Var}(g(y))}$$

Note: the notions of **covariance (correlation)** and **dependence** are related but distinct concepts. For two variables to have zero covariance, there must be no **linear dependence** between them. Independence is a stronger requirement that also excludes nonlinear relationships.

Covariance Matrix

In machine learning, computations are usually conducted on matrices, if not even higher dimension arrays, to boost up efficiency.

The **covariance matrix** of a random vector ($\mathbf{x} \in \mathbb{R}^n$ is an $n \times n$ matrix, such that:

$$\mathrm{Cov}(\mathbf{x})_{i,j} = \mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j)$$

The diagonal elements of the covariance give the variance:

$$\mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_i) = \mathrm{Var}(\mathrm{x}_i)$$

Common Probability Distributions

- Bernoulli Distribution
- Multinoulli Distribution
- Normal Distribution (aka, Gaussian Distribution)
- Exponential and Laplace Distribution
- Dirac Distribution and Empirical Distribution

Bernoulli Distribution

The **Bernoulli distribution** is a distribution over a single binary random variable. It is controlled by a single parameter $\phi \in [0, 1]$, which gives the probability of the random variable being equal to 1.

Bernoulli distribution has the following properties:

$$\left.\begin{aligned} P(\mathrm{x}=1) &= \phi \\ P(\mathrm{x}=0) &= 1-\phi \end{aligned}\right\} \Rightarrow P(\mathrm{x}=x) = \phi^x (1-\phi)^{1-x}, \mathrm{x} \in \{0, 1\}$$

$$\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = \phi$$

$$\mathrm{Var}_{\mathrm{x}}[\mathrm{x}] = \phi(1-\phi)$$

Now we can say the result of tossing a coin should satisfy the Bernoulli distribution with $\phi = 0.5$.

Bernoulli Distribution VS. Binominal Distribution

Bernoulli Distribution:

- **Single Trial:** The Bernoulli distribution describes a single random experiment (or trial) with two possible outcomes/categories: success (usually denoted as 1) or failure (usually denoted as 0).
- **Parameter:** It has one parameter, ϕ, which represents the probability of success in a single trial.

The probability mass function (PMF) of the Bernoulli distribution is given by:

$$P(\mathrm{x} = x) = \phi^x (1 - \phi)^{1-x}$$

The mean (μ) and variance (σ^2) of the Bernoulli distribution are:

$$\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = \phi$$

$$\mathrm{Var}_{\mathrm{x}}[\mathrm{x}] = \phi(1 - \phi)$$

Bernoulli Distribution VS. Binominal Distribution

Binomial Distribution:

- **Multiple Trials:** The binomial distribution is an extension of the Bernoulli distribution and describes the number of successes in a fixed number of independent Bernoulli trials.
- **Parameters:** It has two parameters, n (the number of trials) and ϕ (the probability of success in a single trial).

The probability mass function (PMF) of the binomial distribution is given by:

$$P(\mathrm{x} = x) = \binom{n}{x} \cdot \phi^x \cdot (1 - \phi)^{n-x}$$

The mean (μ) and variance (σ^2) of the binomial distribution are:

$$\mathbb{E}_{\mathrm{x}}[\mathrm{x}] = n\phi$$
$$\mathrm{Var}_{\mathrm{x}}[\mathrm{x}] = n\phi(1 - \phi)$$

Multinoulli Distribution VS. Multinomial Distribution

The **multinoulli**, or **categorical distribution** is a distribution over a single discrete variable with k different states, where k is finite.

The **multinoulli distribution** is an extension of the **bernoulli distribution** with $k > 2$ different categories. Both of them describe a **single** random trail.

The **multinomial distribution** represents how many times each of the k categories is visited when n samples are drawn from a **multinoulli distribution**.

The **multinomial distribution** is an extension of the **binominal distribution** with $k > 2$ different categories. Both of them describe **multiple** random trails.

Normal Distribution (aka, Gaussian Distribution)

The most commonly used distribution over real numbers is the **normal distribution**:

$$\mathcal{N}(x;\mu,\sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

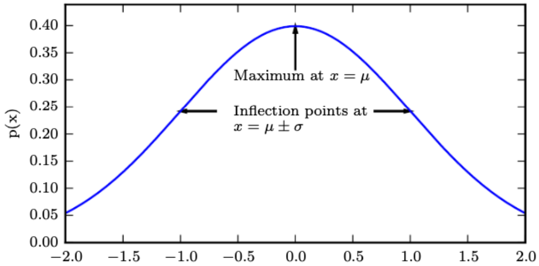


Figure 3.1: **The normal distribution**: The normal distribution $\mathcal{N}(x;\mu,\sigma^2)$ exhibits a classic "bell curve" shape, with the x coordinate of its central peak given by μ, and the width of its peak controlled by σ. In this example, we depict the **standard normal distribution**, with $\mu = 0$ and $\sigma = 1$.

Source: Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

Normal Distribution (aka, Gaussian Distribution)

In the absence of prior knowledge, the normal distribution is a good default choice for the distribution of our data for two major reasons:

(1) Many distributions we wish to model are truly close to being normal distribution. The **central limit theorem** shows that the sum of many independent random variables is approximately normally distributed.

(2) Out of all possible probability distributions with the same variance, the normal distribution encodes the maximum amount of uncertainty over the data. **We can think of the normal distribution as being the one that inserts the least amount of prior knowledge into a model.**

Exponential and Laplace Distribution

In the context of machine learning, we often want to have a probability distribution with **a sharp point at $\boldsymbol{x = 0}$**. To accomplish this, we use the **exponential distribution**:

$$p(x;\lambda) = \lambda \mathbf{1}_{x\geq 0} \exp(-\lambda x)$$

where $\mathbf{1}_{x\geq 0}$ assigns probability 0 to all negative values of x.

If we want to place a sharp point at an arbitrary place μ, we can use **Laplace distribution**:

$$\text{Laplace}(x;\mu,\gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x-\mu|}{\gamma}\right)$$

Dirac Distribution and Empirical Distribution

In some cases, we want to specify a probability distribution that all the mass is clustered around a single point. So we define the **Dirac delta function**, $\delta(x)$:

$$\delta(x) \simeq \begin{cases} +\infty, & x = 0 \\ 0, & x \neq 0 \end{cases}$$

A common use of the Dirac delta distribution is as a component of an **empirical distribution**, which is a distribution that describes the observed frequencies of a set of data points.

Unlike theoretical probability distributions, which are based on mathematical models, the empirical distribution is derived directly from the data itself.

Useful Properties of Common Functions

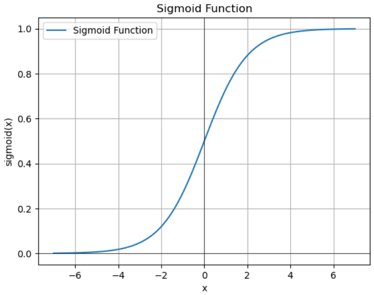
Here we discuss some widely used functions in machine learning. These functions are usually called as activation functions in deep learning models, but more on that in later chapters.

- Sigmoid Function
- Tanh Function
- ReLU Function
- Leaky ReLU Function
- Softplus Function
- Softmax Function

Sigmoid Function

One commonly used function in machine learning is the Sigmoid function. It is commonly used to produce the ϕ parameter of a Bernoulli distribution because its range is $(0, 1)$.

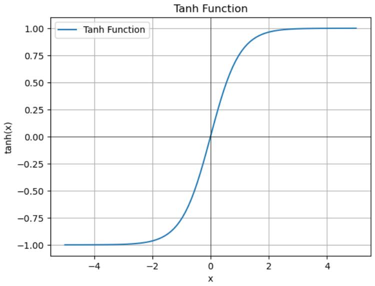
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



Tanh Function

Tanh function is also widely used in machine learning. Its output value is between -1 and 1.

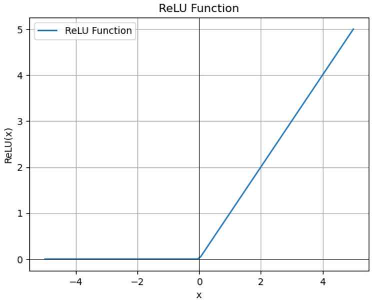
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$



ReLU Function

ReLU function returns 0 if it receives any negative input, but for any positive value x it returns that value back.

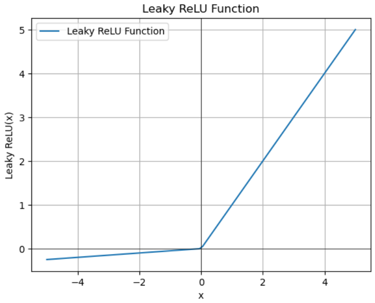
$$f(x) = \max(0, x)$$



Leaky ReLU Function

Leaky ReLU function is based on ReLU, but it has a small slope for negative values instead of a flat slope.

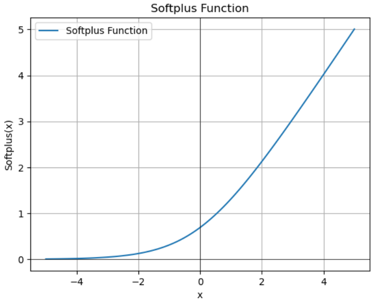
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$



Softplus Function

Softplus is a smooth approximation to the ReLU function and can be used to constrain the output of a machine to always be positive.

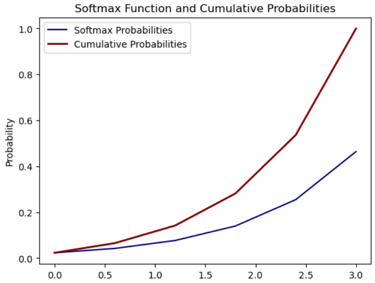
$$f(x) = \ln(1 + e^x)$$



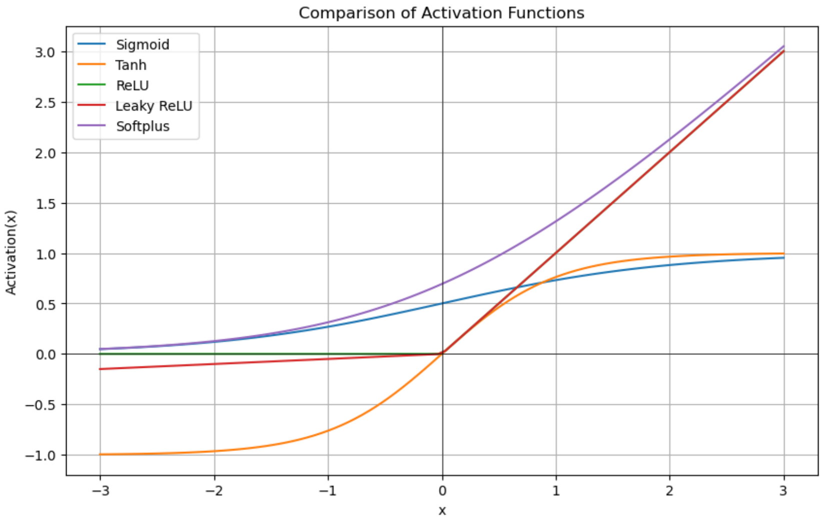
Softmax Function

Softmax function onverts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. And the probabilities should sum to 1.

$$f(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$



Comparison of Different Activation Functions



Information Theory

The basic intuition behind information theory is that **learning that an unlikely event has occurred is more informative than learning that a likely event has occurred**.

We would like to quantify information in a way that formalizes this intuition:

- Likely events should have low information content. Events that are guaranteed to happen have no information content.
- Less likely events should have higher information content.
- Independent events should have additive information.

Information Theory

To satisfy the above three properties, we define the **self-information** of an event $\mathrm{x} = x$ to be:

$$I(x) = -\log P(x)$$

When x is continuous, we use the same definition of information by analogy, but some of the properties from the discrete case are lost. For example, an event with unit density still has zero information.

Information Theory

Self-information deals only with a single outcome. We can quantify the amount of uncertainty in an entire probability distribution using the **Shannon entropy**:

$$H(\mathrm{x}) = \mathbb{E}_{\mathrm{x}\sim P}[I(x)] = -\mathbb{E}_{\mathrm{x}\sim P}[\log P(x)]$$

Distributions that are nearly deterministic have low entropy; distributions that are closer to uniform have high entropy.

When x is continuous, the Shannon entropy is known as the **differential entropy**.

Information Theory

If we have two separate probability distribution $P(\mathrm{x})$ and $Q(\mathrm{x})$ over the same random variable x, we can measure how different thest two distribution are using the **Kullback-Leibler (KL) divergence**:

$$D_{KL}(P||Q) = \mathbb{E}_{\mathrm{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathrm{x}\sim P}[\log P(x) - \log Q(x)]$$

D_{KL} is the extra amount of information needed to send a message containing symbols drawn from probability distribution P, when we use a code that was designed to minimize the length of messages drawn from probability distribution Q.

Information Theory

The KL divergence (D_{KL}) has one notably property: $D_{KL} \geq 0$. (Can you prove it?)

The KL divergence is often conceptualized as measuring some sort of distance between two distributions.

However, the KL divergence is not a true distance measure because it is asymmetric: $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ for some P and Q.

A concept that is closely related to the KL divergence is the **cross-entropy**:

$$H(P, Q) = H(P) + D_{KL}(P||Q) = -\mathbb{E}_{\mathrm{x} \sim P} \log Q(x)$$

Structured Probabilistic Models

Machine learning algorithms often involve probability distributions over a very large number of random variables. Using a single function to describe the entire joint probability distribution can be very inefficient.

We can split a probability distribution into many factors that we multiply together.

For example we can represent the following probability distribution over three variables as a product of probability distributions over two variables:

$$p(a, b, c) = p(a)p(b|a)p(c|b)$$

These factorizations can greatly reduce the number of parameters needed to describe the distribution.

Structured Probabilistic Models

The above factorization can also be described using graphs. When we represent the factorization of a probability distribution with a graph, we call it a **structured probabilistic model**, or **graphical model**.

There are two main kinds of structured probabilistic models: **directed** and **undirected**. Any probability distribution may be described in either model.

Both graphical models use a graph $\mathcal{G}$ in which each **node** in the graph corresponds to a random variable, and an **edge** connecting two random variables means that the probability distribution is able to represent direct interactions between those two random variables.

Directed Model

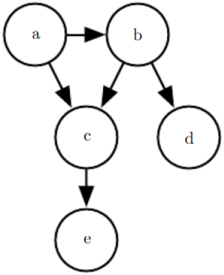


Figure 3.7: A directed graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as

$$p(\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}) = p(\mathrm{a})p(\mathrm{b} \mid \mathrm{a})p(\mathrm{c} \mid \mathrm{a}, \mathrm{b})p(\mathrm{d} \mid \mathrm{b})p(\mathrm{e} \mid \mathrm{c}). \tag{3.54}$$

Undirected Model

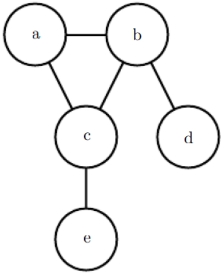


Figure 3.8: An undirected graphical model over random variables a, b, c, d and e. This graph corresponds to probability distributions that can be factored as

$$p(\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e}) = \frac{1}{Z}\phi^{(1)}(\mathrm{a},\mathrm{b},\mathrm{c})\phi^{(2)}(\mathrm{b},\mathrm{d})\phi^{(3)}(\mathrm{c},\mathrm{e}). \quad (3.56)$$

Part III. Numerical Computation

Overflow and Underflow

The fundamental difficulty in performing continuous math on a digital computer is that we need to represent infinitely many real numbers with a finite number of bit patterns.

This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer.

One form of rounding error that is particularly devastating is **underflow**, which occurs when numbers near 0 are rounded to 0.

Another highly damaging form of numerical error is **overflow**, which occurs when numbers with large magnitude are approximated as ∞ or $-\infty$.

One Example: Softmax Function

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

Consider what happens when all the x_i are equal to some constant c. Analytically, we should see the outputs be $\frac{1}{n}$.

This may not happen when c is very large (**overflow**), or c is very negative (**underflow**).

Q: Can you find a way to address this issue?

One Example: Softmax Function

Q: Can you find a way to address this issue?

Both of these overflow and underflow difficulties can be resolved by instead evaluating $\mathrm{softmax}(\boldsymbol{z})$ where $\boldsymbol{z} = \boldsymbol{x} - \max(\boldsymbol{x})$.

Simple algebra shows that the value of the softmax function is not changed analytically by adding or subtracting a scalar from the input vector.

Let's try it out in python!

Overflow and Underflow

For the most part, we do not explicitly detail all the numerical considerations when we implement our own machine learning models. Normally, we can simply rely on low-level libraries that provide stable implementations.

However, developers of low-level libraries should keep numerical issues in mind when implementing machine learning algorithms.

Poor Conditioning

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs.

Functions that change rapidly when their inputs are slightly changed can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

Consider the function $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. When $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has an eigenvalue decomposition, its **condition number** is:

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$$

The condition number is the ratio of the largest and smallest eigenvalues.

Poor Conditioning

When the **condition number** is large, matrix inversion is particularly sensitive to error in the input.

This sensitivity is an intrinsic property of the matrix itself, not the result of rounding error during matrix inversion.

Poorly conditioned matrices amplify pre-existing errors when we multiply by the true matrix inverse. In practice, the error will be compounded further by numerical errors in the inversion process itself.

Gradient-Based Optimization

Most machine learning algorithm involve optimization, which referees to the task of either minimizing or maximizing some function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$.

The function, $f(\boldsymbol{x})$, we want to minimize or maximize is called the **objective function**, or **criterion**. When we are minimizing it, we may also call it the **cost function**, **loss function**, or **error function**.

We usually denote the value that minimizes or maximizes a function with a superscript $*$. For example, we might say $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$.

Gradient-Based Optimization

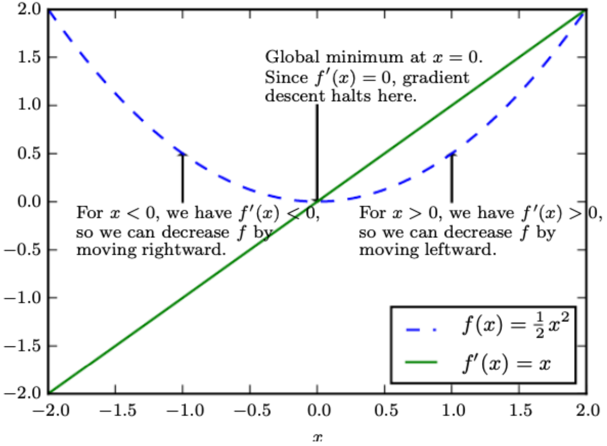
The **derivative** of a function with a single input, $y = f(x)$, is denoted as $f'(x)$ or as $\frac{dy}{dx}$.

The derivative gives the slope of $f(x)$ at the point x, which measures the corresponding change in the output caused by a small change in the input: $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$.

The derivative is useful for minimizing a function because it tells us how to change x to make y even smaller.

Gradient-Based Optimization

From the picture, we can see that $f(x - \epsilon \cdot \text{sign}(f'(x)))$ is less than $f(x)$. So we can reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative. This technique is called **gradient descent**.

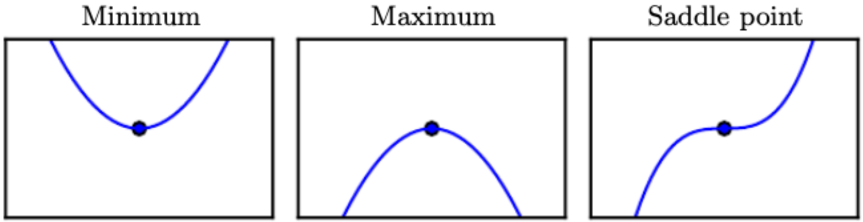


Gradient-Based Optimization

Critical points, or **stationary points**: points where $f'(x) = 0$.

Local minimum (maximum) : a critical point where $f(x)$ is lower (higher) than all neighboring points.

Saddle point: a critical point that is neither the maximum nor the minimum.

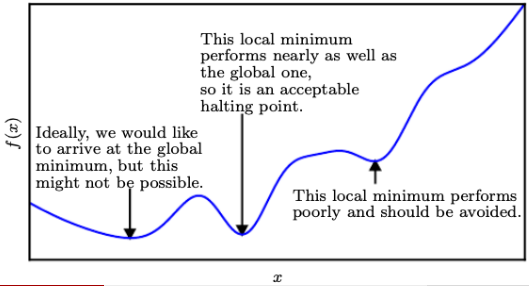


Gradient-Based Optimization

Global minimum (maximum): a point that obtains the absolute lowest (highest) value of $f(x)$.

In machine learning, optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present.

We generally accept solutions that are not truly minimal, as long as they correspond to significantly low values of the cost function.



Gradient-Based Optimization

For functions with multiple inputs, $f(\boldsymbol{x})$, we need to use the concept of **partial derivatives**, $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$.

The **gradient** of $f(\boldsymbol{x})$ is the vector containing all the partial derivatives, denoted $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

The **directional derivative** measures how a function $f(\boldsymbol{x})$ changes at a particular point $\boldsymbol{x}$ in the direction of a specified vector $\boldsymbol{u}$ (a unit vector). And we can denote it as:

$$D_{\boldsymbol{u}} f(\boldsymbol{x}) = \lim_{\alpha \to 0} \frac{f(\boldsymbol{x} + \alpha \boldsymbol{u}) - f(\boldsymbol{x})}{\alpha} = \boldsymbol{u}^{\intercal} \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

Gradient-Based Optimization

To minimize $f(\boldsymbol{x})$, we would like to find the direction $\boldsymbol{u}$ in which f decreases the fastest. We can do this using the directional derivative:

$$\min_{\boldsymbol{u},\boldsymbol{u}^\intercal\boldsymbol{u}=1} \boldsymbol{u}^\intercal \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \min_{\boldsymbol{u},\boldsymbol{u}^\intercal\boldsymbol{u}=1} ||\boldsymbol{u}||_2 ||\nabla_{\boldsymbol{x}} f(\boldsymbol{x})||_2 \cos\theta$$

where θ is the angle between $\boldsymbol{u}$ and the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

Substituting in $||\boldsymbol{u}||_2 = 1$ and ignoring the factors that do not depend on $\boldsymbol{u}$, the above question can be simplified to $\min_{\boldsymbol{u}} cos\theta$.

Obviously, since $\cos\pi = -1$, the value is minimized when $\boldsymbol{u}$ points in the **opposite direction** as the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

Gradient-Based Optimization

Gradient descent: the method that we decrease $f(\boldsymbol{x})$ by moving in the direction of the negative gradient, $-\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

Gradient descent method proposes a new point $\boldsymbol{x}'$ based on:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

where ϵ is called as **learning rate**, a positive scalar determining the size of the step.

Learning rate is an important hyperparameter that needs to be fine-tuned in the model training phase. We will come back to this concept in later chapters.

Let’t try it out in python!

Beyond the Gradient: Jacobin and Hessian Matrices

If the function whose input and output are both vectors, we need a matrix to describe all partial derivatives. This matrix is called **Jacobian matrix**.

For function $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n$, then the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ is defined as:

$$J_{ij} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$$

Furthermore, if we want to describe all the second derivatives (if they exist), the matrix is called **Hessian matrix**:

$$\boldsymbol{H}(f)(\boldsymbol{x})_{ij} = \frac{\partial^2}{\partial_{x_i} \partial_{x_j}} f(\boldsymbol{x}).$$

1 We can think the Hessian as the Jacobian of the gradient.

Beyond the Gradient: Jacobin and Hessian Matrices

Recall the **second derivative test**, for a simple function $f(x)$:

(1) When $f'(x) = 0$ and $f''(x) > 0$, x is a local minimum.

(2) When $f'(x) = 0$ and $f''(x) < 0$, x is a local maximum.

(3) When $f'(x) = 0$ and $f''(x) = 0$, the test is inconclusive. In this case, x may be a saddle point or a part of a flat region.

Now we want to extent this test to multiple dimensions relying on the Hessian matrix.

Beyond the Gradient: Jacobin and Hessian Matrices

At a critical point ($\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \mathbf{0}$), we examine the eigenvalues of the Hessian to determine local maximum/minimum, or saddle point.

(1) **Positive-Definite** Hessian (all eigenvalues are negative) indicates a local minimum.

(2) **Negative-Definite** Hessian (all eigenvalues are negative) indicates a local maximum.

(3) **Indefinite** Hessian (both positive and negative eigenvalues) indicates a saddle point.

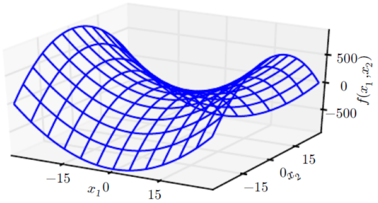
(4) The result is inconclusive if there are zero eigenvalues and the rest are either positive or negative. Then, we need to use higher order terms to decide if we have a saddle point or local minimum/maximum.

Saddle Points

We may encounter saddle points when: (1) its Hessian matrix has both positive and negative eigenvalues; (2) its Hessian matrix has at least one zero eigenvalue.

This is different from the simple function case where a saddle point only occurs when $f''(x) = 0$.

Here is a function $f(\boldsymbol{x}) = x_1^2 - x_2^2$ with a saddle point. You can see the corresponding Hessian has both positive and negative eigenvalues. (try it out in python!)



Newton's method

The **condition number** of the Hessian at a point measures how much the second derivative differ from each other. When the Hessian has a **poor** condition number, gradient descent performs **poorly** and makes it **difficult** to choose a good **step size**.

A solution is known as **Newton's method**, which uses a second-order Taylor series expansion to approximate $f(\boldsymbol{x})$ near some point $\boldsymbol{x}^{(0)}$:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})(\boldsymbol{x} - \boldsymbol{x}^{(0)})$$

The critical point of the above function can be written as:

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})^{-1} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)})$$

Note: Newton's method is only appropriate when the nearby critical point is a minimum. (try it out in python!)

Part IV. Machine Learning Basics

Machine Learning Basics

Machine learning is essentially a form of **applied statistics** with increased emphasis on the use of computers to statistically estimate **complicated functions** and a decreased emphasis on **proving confidence intervals** around these functions.

Learning Algorithms

When we talk about machine learning, what do we mean by learning?

Mitchell(1997) provides a succinct definition:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Next, we will discuss what are T, P and E.

The Task, T

Machine Learning tasks are usually described in terms of how the machine learning system should process an **example** (aka, **observation** or **data point**).

An example is a collection of **features** that have been quantitatively measured from some object or event that we want the machine learning system to process.

Here we list some of the most common machine learning tasks.

The Task, T

- Classification: Binary, Multi-class (each input is mapped to one of multiple outputs), Multi-label (each input can be mapped to multiple outputs).
- Regression
- Transcription
- Machine Translation
- Structured Output
- Synthesis and Sampling
- Imputation of missing values
- Denoising

The Performance Measure, P: Classification

- **Accuracy:** The ratio of correctly predicted instances to the total instances.
- **Precision:** The ratio of true positive predictions to the total predicted positives.
- **Recall (Sensitivity):** The ratio of true positive predictions to the total actual positives.
- **F1 Score:** The harmonic mean of precision and recall, providing a balance between the two. $F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$
- **Specificity:** The ratio of true negative predictions to the total actual negatives.
- **Area Under the Receiver Operating Characteristic (ROC-AUC):** Measures the trade-off between sensitivity and specificity across different thresholds. (try it out in python!)

The Performance Measure, P: Classification

- **Confusion Matrix:** A table shows the numbers of True Positive, True Negative, False Positive (Type I error) and False Negative (Type II error) predictions.

Table 1: Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

For Multi-label Classification:

- **Hamming Loss:** Measures the fraction of labels incorrectly predicted.
- **Jaccard Similarity Score:** Measures the similarity between predicted and true label sets.

The Performance Measure, P: Regression

- **Mean Absolute Error (MAE):** The average absolute differences between predicted and actual values.
- **Mean Squared Error (MSE):** The average of the squared differences between predicted and actual values.
- **Root Mean Squared Error (RMSE):** The square root of the MSE, providing an interpretable scale.
- $\boldsymbol{R^2}$**:** Measures the proportion of the variance in the dependent variable explained by the model.

The Performance Measure, P

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system.

In some case, this is because it is difficult to decide what should be measured.

in other cases, we know what quantity we would ideally like to measure, but measuring it is impractical.

The Experience, E

Machine learning algorithm can be broadly categorized as **unsupervised** or **supervised** by what kind of experience they are allowed to have during the learning process. And most of the learning algorithms are allowed to experience an entire **dataset**.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a **label** or **target**.

The Experience, E

The lines between supervised and unsupervised learning sometimes are blurred. For example, in semi-supervised learning, some examples include a supervision target but others do not.

Some machine learning algorithms do not just experience a fixed dataset. For example, in reinforcement learning, there is a feedback loop between the learning system and its experiences.

Sometimes, the label in supervised learning can be more than just a single number. For example, if we train a speech recognition system to transcribe entire sentences, then the label for each example sentence is a sequence of words.

Training Set, Validation Set, Test Set

In machine learning, we usually divide our whole dataset into three sets:

Training Set: a subset of the dataset used to train a machine learning model. The model learns from the patterns and relationships within this set to make predictions on new, unseen data.

Validation Set: a subset that is not used for training the model. Instead, it is employed to fine-tune the model's **hyperparameters** and assess its performance during training.

Test Set: a subset that is not used during the training or validation phases. It serves as an independent evaluation to assess how well the model generalizes to new, unseen data.

Hyperparameters and Validation Set

Hyperparameters are settings that we can use to control the algorithm's behavior. The values of hyperparameters are not learned by the learning algorithm itself.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not choose to learn because the setting is difficult to optimize. More frequently, it may be not appropriate to learn that hyperparameter on the training set.

If we have multiple candidates for one hyperparameter, we would like to pick the one that demonstrates the best performance on the **validation set**, this process is called **hyperparameter fine-tuning**.

Capacity, Overfitting and Underfitting

Generalization: The ability of a model that can perform well on previously unobserved inputs.

Typically, when training a machine learning model, we can compute some error measure on the **training set**, called the **training error**. Reducing the training error is simply an **optimization problem**.

Obviously, optimizing our model based on the training set is our first goal. However, our final goal is to measure the model performance on the **test set** and make the **generalization error**, also called the **test error**, to be low as well.

Capacity, Overfitting and Underfitting

Q: How can we affect performance on the test set when we can only train our model on the training set?

Because we impose the **i.i.d. assumptions** on our dataset. We assume that the examples in each dataset are **independent** from each other, and the training set and test set are **identically distributed**.

These i.i.d. assumptions give us some guidance on how to construct a valid training set. Can you think of a situation where these assumptions are violated?

Capacity, Overfitting and Underfitting

The factors determining how well a machine learning algorithm will perform are its ability to:

(1) Make the training error small.

(2) Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: **underfitting** and **overfitting**.

Capacity, Overfitting and Underfitting

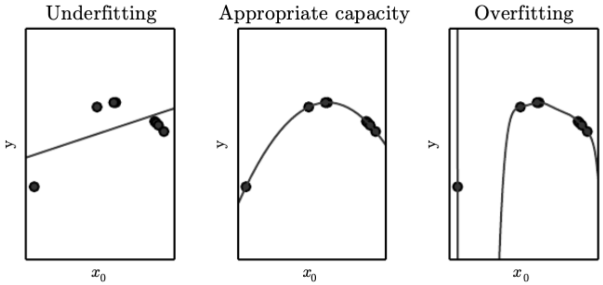
Underfitting occurs when the model is not able to obtain a sufficiently low error value on the **training set**.

Overfitting occurs when the gap between the **training error** and **test error** is too large.

We can control whether a model is more likely to overfit or underfit by altering its **capacity**, which is the model's ability to fit a wide variety of functions.

Capacity, Overfitting and Underfitting

Model with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed, they may overfit.



Statistical theory tells us that the discrepancy between training error and test error grows as the **model capacity** grows, but shrinks as the **number of training examples** increases.

Point Estimation

Point Estimation is the attempt to provide the single “best” prediction of some quantity of interest, which can be a single parameter, or an array of parameters.

To distinguish estimates of parameters from their true value, our convention will be to denote a point estimate of a parameter $\boldsymbol{\theta}$ by $\hat{\boldsymbol{\theta}}$.

Let $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ be a set of m i.i.d. data points. A point estimator is any function of the data:

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)})$$

While $g(\cdot)$ can be any function, a good estimator is a function whose output is close to the true $\boldsymbol{\theta}$.

Bias

The bias of an estimator is defined as :

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}$$

An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **unbiased** if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies that $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **asymptotically unbiased** if $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies that $\lim_{m\to\infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Variance

The first estimator of the true variance, σ^2, is known as the **sample variance**:

$$\hat{\sigma}_m^2 = \frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)} - \hat{\mu}_m \right)^2$$

The **unbiased sample variance** estimator:

$$\tilde{\sigma}_m^2 = \frac{1}{m-1} \sum_{i=1}^{m} \left(x^{(i)} - \hat{\mu}_m \right)^2$$

Trading Off Bias and Variance

Bias and variance measure two different sources of error in an estimator. **Bias** measures the expected deviation from the true value. **Variance** provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

One common way to negotiate the trade-off is to use cross-validation. Alternatively, we can also compare the **mean squared error** (MSE) of the estimates:

$$\mathrm{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] = \mathrm{Bias}(\hat{\theta}_m)^2 + \mathrm{Var}(\hat{\theta}_m)$$

When test error is measured by the MSE, increasing model capacity tends to increase variance and decrease bias.

Consistency

We usually wish that, as the number of data points m in our dataset increases, our point estimates converge to the true value of the corresponding parameters:

$$\text{plim}_{m\to\infty}\hat{\theta}_m = \theta$$

This condition described above is known as **consistency**.

Consistency ensures that the bias induced by the estimator diminishes as the number of data points grows.

However, the reverse is not true — asymptotic unbiasedness does not imply consistency.

Maximum Likelihood Estimation

Consider a set of m examples $\mathbb{X} = \boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}$ drawn independently from the true but unknown data-generating distribution $p_{data}(\mathbf{x})$.

Let $p_{model}(\mathbf{x}; \boldsymbol{\theta})$ be a parametric family of probability distributions over the same space indexed by $\boldsymbol{\theta}$.

The maximum likelihood estimator for $\boldsymbol{\theta}$ is then defined as:

$$\begin{aligned}\boldsymbol{\theta}_{ML} &= \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{X}; \boldsymbol{\theta}),\\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\end{aligned}$$

Maximum Likelihood Estimation

In the above equation, the product over many probabilities is prone to numerical underflow. We usually transform the above equation by taking the logarithm of the likelihood:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

Because the argmax does not change when we rescale the cost function, we can divide by m to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution $\hat{p}_{data}$ defined by the training data:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} \log p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$$

Maximum Likelihood Estimation

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{data}$, defined by the training set and the model distribution.

The degree of dissimilarity between these two distributions can also be measured by the KL divergence:

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}}[\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})]$$

When we train the model to minimize the KL divergence, we only need to minimize:

$$-\mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{x})$$

which is the same as: $\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{x};\boldsymbol{\theta})$.

Maximum Likelihood Estimation

Minimizing this KL divergence corresponds exactly to minimizing the cross-entropy between the distributions.

We can thus see maximum likelihood as an attempt to make the model distribution, p_{model}, match the empirical distribution, $\hat{p}_{data}$.

While the optimal $\boldsymbol{\theta}$ is the same regardless of whether we are maximizing the likelihood or minimizing the KL divergence, we prefer to minimize the cost function in software.

Hence, maximum likelihood becomes **minimization of the KL divergence or the cross-entropy**. Another advantage of using KL divergence is that it has a known **minimum value of zero**.

Maximum Likelihood Estimation

Under appropriate conditions, the maximum likelihood estimator has the property of consistency. These conditions are:

(1) The true distribution p_{data} must lie within the model family $p_{model}(\cdot; \boldsymbol{\theta})$. Otherwise, no estimator can recover p_{data}.

(2) The true distribution p_{data} must correspond to exactly one value of $\boldsymbol{\theta}$. Otherwise, maximum likelihood can recover the correct p_{data} but will not be able to determine which value of $\boldsymbol{\theta}$ was used by the data-generating process.