# Machine learning for physicists [PHYS-467]

Lecture Notes

**Prof. Lenka Zdeborová**

This is the updated version of the lecture notes from 2025.
It is still in progress and there might be typos. If you notice any inconsistencies, ask a question on moodle or write to margarita.sagitova@epfl.ch.

First edition by the students of the 2021 course, with corrections and additions by the 2021 TAs Federica Gerace, Giovanni Piccioli, Alessandro Favero, and Prof. Lenka Zdeborová.
Updated 2023 by Luca Biggio, Christian Keup, in 2024 by Ekaterina Pankovets, in 2025 by Margarita Sagitova.

Institute of Physics – IPHYS
École polytechnique fédérale de Lausanne – EPFL
Last Update Fall 2025

# Contents

# 1 Introduction: ML & Physics

This course aims to introduce the foundations of Machine Learning (ML) to physicists. Students will learn to use ML as a tool for physics and sciences, think in a data-oriented way, and understand the foundations of ML methods[1].

The course will heavily utilize linear algebra notation, which students are encouraged to review if rusty, as it will be used throughout.

## 1.1 Machine Learning Problem types and Terminology

We will start by discussing types of problems typically solved with Machine Learning methods.

### 1.1.1 Supervised learning

This approach uses a training set of labeled data samples to train models that predict the labels of new samples.

**Examples:**

**A) Image Classification (Cats & Dogs).** Given labeled images (+1 for cat, -1 for dog), find a function that correctly classifies new, unseen images.

**B) Patient Survival Prediction,** Given past patient data (medical images, blood tests, surgery results, etc.) and their survival times after a new treatment, learn a function to predict the survival time of a new patient.

**Data:** A **training set** consisting of

- samples: $\vec{X}_\mu \in \mathbb{R}^d$ (e.g. pixels of the images or age, weight and blood pressure of the patient),
- corresponding labels: $y_\mu \in S$, where the set $S$ depends on the type of the problem:
  - **Regression.** $y_\mu \in \mathbb{R}$ (e.g.)
  - **Classification.** $y_\mu$ is in some finite set of classes, such as $\{-1, +1\}$ (cat label or dog label) or $\{1, 2, \ldots, k\}$ (distinguishing handwritten digits).

**Goal:** To find a function $f : \mathbb{R}^d \to S$ such that $f(X_\mu) = y_\mu$.

### 1.1.2 Unsupervised Learning

This approach uses a dataset with no predefined labels.

**Examples:**

**C) Protein sequence generation.** Given existing protein sequences data, generate new protein sequence with similar properties.

**D) Cell type classification from gene expression.** Based on gene expression patterns of various cells (without prior labels), group them into different cell types. This cell types can later be assigned meaningful labels, e.g. bone cell, brain cell, etc., without manually labeling all samples one by one.

**Types of unsuperivsed learning:**

- **Dimensionality Reduction / Representation Learning:** Aims to represent high-dimensional data in a more concise or useful shorter vector, preserving essential information. A type of representation learning is clustering, which aims to group data into types/clusters.
- **Generative Models:** Aims to generate new data points that share properties with the existing dataset, e.g. generating protein sequences.

[1] After this course you won't be able to build a model like ChatGPT. The aim is to provide a foundational understanding similar to learning Maxwell equations for building an iPhone, enabling students to understand the underlying principles of complex systems like ChatGPT

#### 1.1.3 Self-Supervised Learning:

This approach is predominant in modern ML. It utilizes pieces of training data as labels to solve unsupervised learning tasks using methods similar to those employed for supervised learning. For example, ChatGPT is trained this way by predicting the next word in a text, where the next word acts as the "label".

#### 1.1.4 Reinforcement Learning

**Examples:**

**E) Learning to play Go.** Teach a computer to play the complex board game Go, eventually beating human players.

**F) Teaching a robot to walk.** Develops an effective walking strategy for a robot no with no prior knowledge of how to coordinate its joints.

In reinforcement learning, the system learns through interaction with an environment. Instead of using training data in the usual sense, the model, called an agent, learns optimal behavior by taking actions in an environment, receiving rewards or penalties, and adjusting its strategy through trial and error to maximize total reward over time

### 1.2 Structure of the course

In this course, we will start with basic supervised learning methods (linear regression and classification) then proceed with basic unsupervised methods (principal component analysis, Bayesian inference, clustering). We will then go back to more advanced supervised learning (kernel regression, and learning with deep neural networks), and finish with advanced unsupervised methods involving some type of self-supervision (auto-encoders, next-token prediction, diffusion generative models). The course does not cover reinforcement learning.

## 2 Linear regression

Linear regression is a foundational model in Machine Learning. In this section, we discuss the model itself and introduce several important concepts that apply across all other Machine Learning settings.

### 2.1 Intuitive Introduction

A very common task while performing a physics experiment is to fit a straight line through measured points. For example, suppose we measured position $y$ of an object at time $z$ and got $n$ data pairs $\{(z_\mu, y_\mu)\}_{\mu=1}^n$. Now we can use this data and fit a line

$$y(z) = w_1 z + w_0 \tag{2.1}$$

through the points, motivated by one of the following goals:

A) **Prediction.** Use given data to make predictions about previously unseen samples (what will be the position at time $z_{\text{new}}$?)

B) **Estimation.** Find the parameter of the model, that has a meaningful interpretation (parameter $w_1$ in this model is the speed of an object).

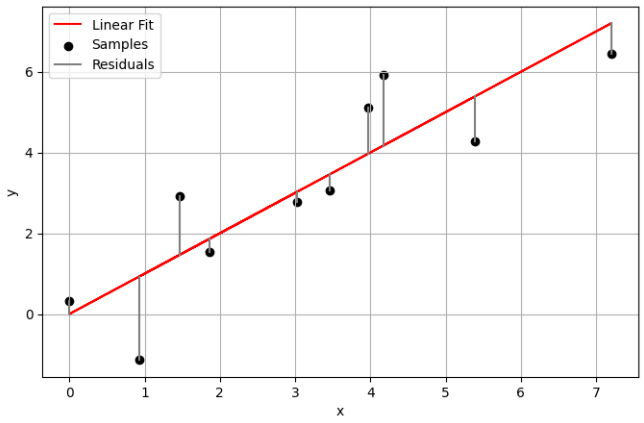



Figure 1: Example of linear regression showing sample data points, fitted regression line, and residuals.

In general, relationship (2.1) will not hold precisely, which is modeled through residuals $\{\varepsilon_\mu\}_{\mu=1}^n$. Thus, the model takes form:

$$y_\mu = w_1 z_\mu + w_0 + \varepsilon_\mu. \tag{2.2}$$

## 2.2 Least Squares Method

Now, the goal is to find the parameters of the model that would best describe the relationship between observed times and positions (intuitively, plot a line that is as close to all the measured points as possible, see Figure 1). Most commonly, this is done by defining a **loss function** that quantifies the error between the model's predicted output and the actual measured value.

One broadly used loss function is **Square loss** or **Mean Square Error** (MSE):

$$\mathscr{L}(w) = \frac{1}{n}\sum_{\mu=1}^{n} \varepsilon_\mu^2 = \frac{1}{n}\sum_{\mu=1}^{n} (y_\mu - w_1 z_\mu - w_0)^2. \tag{2.3}$$

**Ordinary Least Squares** (OLS) method consists of finding the estimated values $\hat{w}_0, \hat{w}_1$ for the parameters $w_0, w_1$, that minimize square loss. This method yields explicit expressions for $\hat{w}_0$ and $\hat{w}_1$.

The necessary conditions that $\mathscr{L}$ has a minimum[2] are:

$$\begin{aligned} 0 &\overset{!}{=} \frac{\partial \mathscr{L}(w)}{\partial w_1} = -2\frac{1}{n}\sum_{\mu=1}^{n} (y_\mu - w_1 z_\mu - w_0)\, z_\mu, \\ 0 &\overset{!}{=} \frac{\partial \mathscr{L}(w)}{\partial w_0} = -2\frac{1}{n}\sum_{\mu=1}^{n} (y_\mu - w_1 z_\mu - w_0). \end{aligned} \tag{2.4}$$

This system of equations can be solved by multiplying the first equation by $n$ and the second equation by $\sum_{\mu=1}^n z_\mu$ and then subtracting the two. After rewriting (2.4) we get for the first equation:

$$\begin{aligned} \sum_{\mu=1}^{n} z_\mu y_\mu &= \sum_{\mu=1}^{n} (w_1 z_\mu^2 + w_0 z_\mu), \\ n\sum_{\mu=1}^{n} z_\mu y_\mu &= n\sum_{\mu=1}^{n} (w_1 z_\mu^2) + n w_0 \sum_{\mu=1}^{n} z_\mu \end{aligned}$$

and for the second one:

$$\begin{aligned} \sum_{\mu=1}^{n} y_\mu &= \sum_{\mu=1}^{n} (w_1 z_\mu) + n w_0, \\ \sum_{\mu=1}^{n} (z_\mu) \sum_{\mu=1}^{n} (y_\mu) &= w_1 \left(\sum_{\mu=1}^{n} (z_\mu)\right)^2 + n w_0 \sum_{\mu=1}^{n} z_\mu \end{aligned}$$

[2]Here we rely on $\mathscr{L}$ being a quadratic function, so that the stationary point is the minimum

Now one equation can be subtracted from the other to get rid of $w_0$ and to find $w_1$ after rearranging. Finding $w_0$ is then straight forward.

As a result, we find the estimated values $\hat{w}_1$ and $\hat{w}_0$ which minimize $\mathscr{L}$:

$$\hat{w}_1 = \frac{\overline{zy} - \overline{z} \cdot \overline{y}}{\overline{z^2} - \overline{z}^2}, \quad \hat{w}_0 = \overline{y} - \overline{z} \cdot \frac{\overline{zy} - \overline{z} \cdot \overline{y}}{\overline{z^2} - \overline{z}^2}, \tag{2.5}$$

where we denote the mean of a generic variable $z = (z_1, \ldots, z_n)$ as:

$$\overline{z} := \frac{1}{n} \sum_{\mu=1}^{n} z_\mu.$$

## 2.3 Matrix Notation

In the example above, there were only two parameters. In general, the number of parameters can become large, and the simple notation we used before becomes difficult to work with. Thus, we introduce matrix notation, which allows for a much more manageable and compact form of equations:

| | | | |
|---|---|---|---|
| $X$ | input data | $\in \mathbb{R}^{n \times d}$ | matrix |
| $y$ | output labels | $\in \mathbb{R}^n$ | vector |
| $w$ | weights to learn | $\in \mathbb{R}^d$ | vector |

where

- $n$ is the number of samples,
- $d$ the dimension of the input samples.

Columns of the matrix $X$ represent particular attributes of the data samples (e.g. a pixel of an image on a specific position or age of the patient). We will use index $\mu$ to refer to a single sample $X_\mu \in \mathbb{R}^d$ and corresponding label $y_\mu$. This notation will be used during most of the lectures. In the example from before we have:

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} z_1 & 1 \\ \vdots & \vdots \\ z_n & 1 \end{pmatrix}, \quad w = \begin{pmatrix} w_1 \\ w_0 \end{pmatrix}.$$

Linear regression is then a linear ansatz (model) for a function $f : \mathbb{R}^d \to \mathbb{R}$ stating that a new label $y_{\text{new}} \in \mathbb{R}$ depends on the new sample $X_{\text{new}} \in \mathbb{R}^d$ linearly, that is

$$y_{\text{new}} = X_{\text{new}}^\top \hat{w}$$

where $\hat{w}$ is estimated from the training data $\{X_\mu, y_\mu\}_{\mu=1}^n$ by minimizing the square loss

$$\hat{w} = \operatorname{argmin} \mathscr{L}(w), \qquad \mathscr{L}(w) = \frac{1}{n} \sum_{\mu=1}^{n} (y_\mu - (Xw)_\mu)^2. \tag{2.6}$$

Setting $\frac{\partial \mathscr{L}(w)}{\partial w_k} \stackrel{!}{=} 0$ for $k \in \{1 \ldots d\}$ yields:

$$0 = \frac{\partial \mathscr{L}(w)}{\partial w_k} = -2 \frac{1}{n} \sum_{\mu=1}^{n} (y_\mu - (Xw)_\mu) \cdot X_{\mu k}. \tag{2.7}$$

From this we can write:

$$(X^T y)_k = \sum_{\mu=1}^{n} y_\mu X_{\mu k} = \sum_{\mu=1}^{n} (Xw)_\mu X_{\mu k} = \sum_{\mu=1}^{n} \sum_{j=1}^{d} w_j X_{\mu j} X_{\mu k} = (X^T X w)_k. \tag{2.8}$$

Since this holds for all $k \in \{1 \dots d\}$ we have[3]

$$X^T y = X^T X \hat{w}. \tag{2.12}$$

Now, assuming $X^T X$ is invertible, we derive that

$$\hat{w} = \left(X^T X\right)^{-1} X^T y. \tag{2.13}$$

The matrix $X^T X \in \mathbb{R}^{d \times d}$ is called the **covariance matrix** and $X^+ \coloneqq (X^T X)^{-1} X^T$ the pseudo-inverse of $X$.

After we obtained $\hat{w}$, we achieved our *estimation* goal, and we can use it for *prediction* as:

$$y_{\text{new}} = X_{\text{new}}^T \hat{w}.$$

## 2.4 Regularization

In general, the invertibility of $X^T X$ depends on the relationship between the number of samples $n$ and the dimension $d$ of the input samples. It is possible to distinguish two situations:

- $n \geq d$: $X^T X$ is typically invertible. Mathematically, this is only true if there are at least $d$ rows in $X$ that are linear independent. For real (typically noisy) data, this is usually the case.
- $n < d$: $X^T X$ is never invertible. This can be easily seen by examining the rank of the matrix:

  $$\text{rank}(X^T X) \leq \min\left(\text{rank}(X^T), \text{rank}(X)\right) = \text{rank}(X) \leq \min(d, n) = n.$$

  Since $X^T X$ is a $d \times d$ matrix with $\text{rank}(X^T X) \leq n < d$, $X^T X$ is not invertible. Note that the kernel of $X^T X$ has at least dimension $d - n$.[4]

Formally speaking, if $\text{rank}(X) = d$, namely, the predictors $X_{:,1}, ..., X_{:,d}$ (i.e. columns of $X$) are linearly independent, then $\hat{w}$, the minimizer of (2.6), is unique. In the case of $\text{rank}(X) < d$, which happens when $d > n$, $X^T X$ is singular, implying there are infinitely many vectors $\hat{w}$ for which the least-squares loss (2.6) is exactly zero, that is, for which $y = W\hat{w}$.

The case of $d > n$ is very common in modern Machine Learning problems, e.g. we might have the tissue images with $d = 10^6$ pixels of $n = 1000$ patients. But there is no unique minimizer of the loss which could be a priori chosen. Therefore, we must ask: which $w$ should we pick in the case $d > n$?

We might want to chose the weights vector with small $L_2$ norm to promote weights close to zero. To achieve that, we can add a regularization term (also called a penalty) $\|w\|_2^2 = \sum_k w_k^2$ to the loss function:

$$\mathscr{L}(w) = \frac{1}{n} \sum_{\mu=1}^{n} (y_\mu - (Xw)_\mu)^2 + \frac{\lambda}{n} \sum_{k=1}^{d} w_k^2 \tag{2.14}$$

where $\lambda > 0$ represents the regularization strength (the larger it is, the closer the solution weights are to 0). This method is called **$\mathbf{L_2}$ regularization** or **Ridge regression**.

[3] Let us demonstrate an even more compact way of deriving the same result utilizing multivariate calculus. For any real number $p \geq 1$, we define the $L^p$ norm of a vector $\boldsymbol{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$ as follows [1]:

$$||\boldsymbol{x}||_p = \left(|x_1|^p + \dots + |x_d|^p\right)^{\frac{1}{p}}. \tag{2.9}$$

For $p = 2$, we get the familiar Euclidean norm, which can be used to express Mean Square Error in matrix notation:

$$\begin{aligned} \mathscr{L}(w) &= \frac{1}{n} ||y - Xw||_2^2 \\ &= \frac{1}{n} (y - Xw)^T (y - Xw) \\ &= \frac{1}{n} (y^T y - y^T X w - w^T X^T y + w^T X^T X w). \end{aligned} \tag{2.10}$$

Setting $\nabla_w \mathscr{L} \stackrel{!}{=} \vec{0}$ yields:

$$\begin{aligned} \frac{1}{n} (-2X^T y + 2X^T X w) &= \vec{0} \\ \Rightarrow X^T y &= X^T X w. \end{aligned} \tag{2.11}$$

[4] I.e. $\{w \in \mathbb{R}^d : X^T X w = 0\}$ is a vector space with at least dimension $d - n$.

Now, if we minimize for $w$:

$$0 = n\frac{\partial \mathscr{L}(w)}{\partial w_k} = -2\sum_{\mu=1}^{n}(y_\mu - (Xw)_\mu)X_{\mu k} + 2\lambda w_k \tag{2.15}$$

Similar to the approach in the previous section, we derive $X^T y = X^T X w + \lambda w = \left(X^T X + \lambda \mathbb{I}\right) w$ and find $\hat{w}$ as:

$$\hat{w} = \left(X^T X + \lambda \mathbb{I}\right)^{-1} X^T y. \tag{2.16}$$

Notice that $\left(X^T X + \lambda \mathbb{I}\right)$ is always invertible for $\lambda > 0$.

Note that in cases the least-squares loss has many minimizers making it exactly zero the one that has the smallest $L_2$ norm can be selected by minimizing (2.14) for $\lambda \to 0^+$. Indeed, for a very small positive value of $\lambda$ the first term in (2.14) dominates as long as it is positive, leading to a minimizer where the first term is zero. At the same time, the second term chooses $\hat{w}$ of smallest norm among all the solutions making the first terms zero. When $\lambda > 0$ the minimizer seeks a tradeoff between the two terms and exactly zero first term will not be achieved.

An attentive reader should have several questions after reading the previous section: How do we choose the right value for $\lambda$? Is $\lambda \to 0^+$ the best choice? Why do we use square norm for the regularization and not other forms that promote small values of the components of $\hat{w}$? We will now develop tools to answer these questions.

## 2.5 Polynomial Regression

In *polynomial regression*, the relationship between the data $z$ and the label $y$ is given by a degree-$p$ polynomial

$$y = \sum_{k=1}^{p} w_k z^k + w_0. \tag{2.17}$$

We now demonstrate that polynomial regression can be naturally formulated in the matrix notation introduced previously; it can thus be seen as a special case of linear regression.

The vector $w$ will belong to the space $\mathbb{R}^{p+1}$, the vector $y$ to $\mathbb{R}^n$ and the matrix X to $\mathbb{R}^{n\times(p+1)}$ and we can write for the matrix notation:

$$w = \begin{pmatrix} w_p \\ w_{p-1} \\ \vdots \\ w_0 \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad X = \begin{pmatrix} z_1^p & z_1^{p-1} & \dots & z_1 & 1 \\ z_2^p & z_2^{p-1} & \dots & z_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ z_n^p & z_n^{p-1} & \dots & z_n & 1 \end{pmatrix}. \tag{2.18}$$

Let us examine the existence of solutions of the system of linear equations $y = Xw$ based on the relationship between $n$ and $p$. Three cases arise:

- if $p+1 < n$, the system typically will not have a solution, as it has more equations than variables. Notice, that in this case the matrix $X^T X$ is *invertible*;
- if $p + 1 = n$, the system typically admits a unique solution, since there is the same number of variables and equations;
- if $p + 1 > n$, the system typically has many solutions, as it has more variables than equations. In this case, $X^T X$ is *not invertible*.

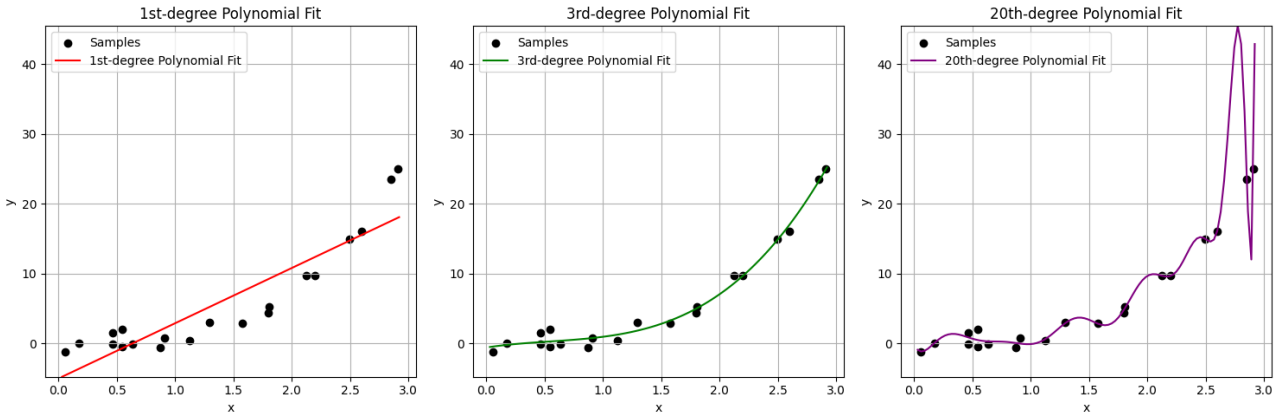



Figure 2: Different polynomial fits for data generated from the model $y = x^3 + \varepsilon$.

Notice that in the case $p + 1 \geq n$ there is at least one polynomial that passes exactly through all sample points. However, this is not always a desirable property, as can be seen in Figure 2. Even though the 20th-degree polynomial fits the samples perfectly, it fails to capture the underlying structure of the data. The model is too complex and fits the noise in the samples as well as the actual underlying process. This behavior is called **overfitting**.

On the other hand, linear model (1st-degree polynomial) is too simple and can not capture the underlying process. This behavior is called **underfitting**. Underfitting and overfitting are quantified through validation.

## 2.6 Validation

### 2.6.1 Validation Methodology

The degree of the polynomial is an example of what we call a **hyperparameter**—a parameter that controls the learning algorithm but is not directly learned from the training data (another example would be the regularization strength $\lambda$). Since hyperparameters are not learned automatically, we need a systematic way to choose their values. Simply selecting the hyperparameter that gives the best performance on the training data would be misleading, as we saw with the 20th-degree polynomial that fitted the given data perfectly but failed to generalize. Therefore, we need to set aside part of the data points, to evaluate the model performance on the new data.

Validation involves splitting the entire dataset into three subsets: training, validation, and test sets. The process of developing a model for the given problem then consists of

1. **Training:** We fit multiple models with different hyperparameter values. For each combination of hyperparameters (e.g., different values of $p$), we minimize the loss function over the training set to find the optimal parameters $\hat{w}$.

2. **Validation:** We evaluate each trained model (compute the loss function) on the validation set to estimate its generalization performance. The hyperparameter combination that yields the lowest validation error is selected as optimal.

3. **Testing:** Once we have selected the best model based on validation performance, we evaluate this final model exactly once on the test set to obtain an unbiased estimate of its true performance on unseen data.

The split between training, validation, and test sets must be done carefully to avoid introducing bias in the evaluation process. In general, the split should be completely random (e.g., if patient data were sorted by age and we trained only on younger patients while validating on older patients, the results would be inadequate). However, in some cases, such as time series forecasting, only historical data should be used for training and more recent data for evaluation, ensuring that the model does not "see the future" during training.

### 2.6.2 Bias-Variance Tradeoff

Let us consider the behavior of the validation error as a function of the model complexity, e.g. polynomial degree $p$ in polynomial regression (the model becomes more complex when $p$ is larger) or $\lambda$ in Ridge regression (the model becomes simpler when $\lambda$ is larger). The validation error usually follows a U-shaped curve conceptually similar to the one shown in the Figure 3. This phenomenon is known as **Bias-Variance Tradeoff**.

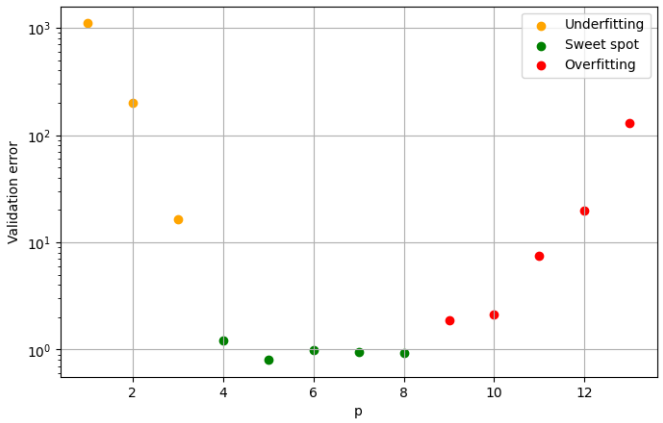



Figure 3: Validation error as a function of the polynomial degree.

Suppose we know the *ground truth* function generating the labels from the input data and we train a model that tries to approximate this ground truth function. Errors originate mainly from two different sources:

- **Bias:** the class of functions that the considered models represent is far away from the ground truth function. The bias is typically high for simple models and close to zero for complex models (e.g. when the number of parameters is small vs when the number of parameters is large)
- **Variance:** the fluctuations of the fitted model prediction due to noise in the dataset (the finite size of the training set, label noise etc). The variance is typically small when the model is simple and may grow when the model becomes complex and able to fit the noise.

The distinction between bias and variance is well illustrated in Figure 4.

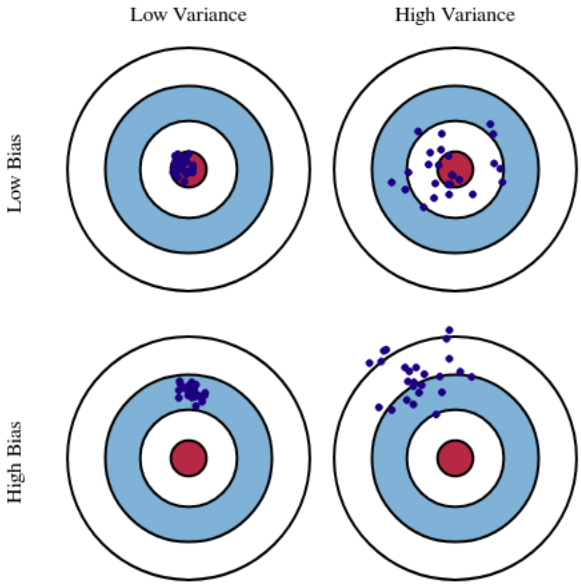



Figure 4: Difference between Bias and Variance. The figures can be interpreted as follows: The target is the ground truth of one component of the weight. Each point denotes an estimation of the the same component where the the measured values (input) for the training vary a little from point to point.

The high bias and low variance case corresponds to a underfitting, whereas the high variance and low bias case corresponds to overfitting. Our goal is to find a model with low bias and variance at the same time.

# 3 Introduction to statistical inference

## 3.1 Basic Probability Theory Refresher

Before discussing statistical inference, we will briefly review some basic probability theory concepts.

We use $P(X \text{ satisfies condition})$ to denote the probability of an event that a random variable $X$ satisfies some condition, and $P(x)$ to denote the probability of an event $X = x$.

**Discrete random variable** is a type of random variable that can only take on a countable number of distinct values (e.g. result of a dice roll). A random variable $X$ over the alphabet $\mathscr{A}_x = \{a_1, a_2, \ldots, a_n, \ldots\}$ is defined by the **Probability Mass Function**

$$P(X = a_i) = p_i, \text{ where } p_i \geq 0 \text{ and } \sum_i p_i = 1. \tag{3.1}$$

**Continuous random variable** is a type of random variable that can take any value on it's uncountable support $\mathscr{A}_x$, such as $\mathbb{R}$ or $[0, 1]$ (e.g. water temperature in lake Geneva). Continuous random variable $X$ is defined by the **Probability Density Function** $p(x)$, so that

$$p(x)dx = P(x < X < x + dx), \text{ where } p(x) \geq 0 \text{ and } \int_{\mathscr{A}} p(x)dx = 1. \tag{3.2}$$

**Joint probability** is used to describe the probability of two random variables $X$ and $Y$ falling into a particular set of values simultaneously. Two events are **independent** if

$$P(A \cap B) = P(A)P(B), \tag{3.3}$$

and random variables are **independent** if

$$P(X \in A, Y \in B) = P(X \in A)P(Y \in B). \tag{3.4}$$

Intuitively, $X$ and $Y$ are independent if knowing the value of one tells you nothing about the value of the other (e.g. if we flip 2 coins, there is no connection between the outcomes).

**Marginal probability** is the probability distribution of one variable obtained from a joint probability distribution of multiple variables. Given joint probability density function[5] $p(x, y)$, marginal densities can be computed as

$$\begin{aligned} p(x) &= \int_y p(x, y)dy, \\ p(y) &= \int_x p(x, y)dx. \end{aligned} \tag{3.5}$$

**Conditional probability** is a probability distribution that describes the probability of an outcome given the occurrence of a particular event (e.g., the probability of passing the exam given that you did the exercises during the semester is higher than the probability of passing without doing the exercises). Conditional probability of event $A$ conditioned on $B$ is expressed as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \tag{3.6}$$

and conditional density function of two random variables $X$ and $Y$ - as

$$p_{X|Y}(x|y) = \frac{p_{X,Y}(x, y)}{p_y(y)}. \tag{3.7}$$

Notice that by definition we get

$$\int_x p_{X|Y}(x|y)dx = 1 \tag{3.8}$$

and

$$P(A \cap B) = P(A|B)P(B) = P(B|A)P(A). \tag{3.9}$$

**Bayes' theorem**, also known as **Bayes' rule** or **Bayes' formula** is stated as the following equation:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \tag{3.10}$$

[5] The case of discrete random variables with probability mass functions follows by analogy, replacing integral with summation.

and is directly derived from (3.9).

Bayes' formula gives a mathematical rule for inverting conditional probabilities, allowing one to find the probability of a cause given its effect. For example, with Bayes' formula, one can calculate the probability that a patient has a disease given that they tested positive for that disease, using the probability that the test yields a positive result when the disease is present.

## 3.2 Statistical inference

We are concerned with the following problem. There is a ground truth process, parametrized by some parameters $\theta_*$, that generates the data $D$ which we can observe. We have some idea of the probability distribution $P(\theta)$ of the parameters $\theta$, as well as the conditional probability $P(D|\theta)$[6]. The goal of **statistical inference** is to estimate the underlying parameters $\theta_*$ of the process that generated the observed data $D$. To do so, we need an estimate of the probability of the parameter $\theta$ conditioned on observed data $D$. This is done using Bayes' formula (3.10):

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}, \tag{3.11}$$

where

- $P(\theta|D)$ is called the **posterior**, the probability of the parameters given the data;
- $P(D|\theta)$ is called the **likelihood**, the probability of the data given the parameters;
- $P(\theta)$ is called the **prior**, the prior belief about the parameters;
- $P(D)$ is the **evidence**, a normalization term that can be calculated as $P(D) = \int d\theta P(D|\theta)P(\theta)$, but usually doesn't need to be explicitly calculated if the goal is only to find the maximum of the posterior.

When these probabilities are known, there are two common methods for $\theta_*$ estimation:

- **Maximum likelihood (ML) estimation:** find the parameters $\theta$ that maximize the likelihood of the observed data: $\hat{\theta}_* = \arg\max_\theta P(D|\theta)$;
- **Maximum a posteriori (MAP) estimation:** find the parameters $\theta$ that maximize the posterior probability distribution: $\hat{\theta}_* = \arg\max_\theta P(\theta|D)$.

Let us use an example to demonstrate how to find the best estimate of the parameters $\theta_*$.

### An Example – Decay Constant

Consider a source of unstable particles, which are emitted through a collimator and decay at a distance $x > 0$ from the source. The random variable $x$ follows the exponential distribution:

$$P(x) = \frac{1}{\lambda}\exp\Big(-\frac{x}{\lambda}\Big). \tag{3.12}$$

Suppose we wish to infer $\lambda^*$, the "ground truth" value of $\lambda$. We set up a detector in front of the source which can measure the value $x$ for a given decay. Unfortunately, the detector only covers a finite range of values, for example, it can only measure decays in the range $1 \le x \le 20$ (see Figure 5).

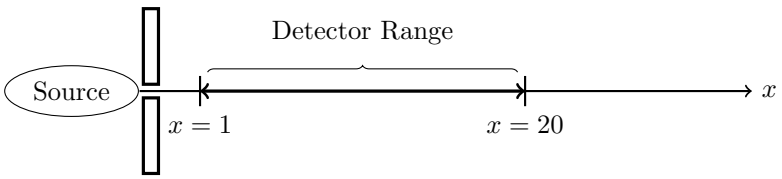



Figure 5: Experimental setup: the source of unstable particles, which travel in a straight line and decay at distance $x$ from the source. The detector can only measure decays in the "Detector Range" from $x = 1$ to $x = 20$.

[6]Here we use the notation $P(\theta) \overset{!}{=} P(\theta_* = \theta)$ and $P(D|\theta) = P(D_{\text{observed}} = D|\theta_* = \theta)$

We observe $n$ measurements of $x$, $X = \{x_1, x_2, ..., x_n\}$ and assume that all these measurements are independent. Now we want to estimate $\lambda^*$.

One might notice that in the suggested model

$$\bar{x} = \int_0^\infty xP(x)dx = \lambda.$$

And therefore, one could use $\lambda^{\text{naive}} = 1/n \sum_{i=1}^n x_i$ as an estimate for $\lambda^*$.

However, **this method is flawed** as a result of the experiment setup. Suppose $\lambda^* = 0.5$. Even though this value is lower than the start of the detector measurement range, there will still be particles reaching the detector before decaying, and we will compute $\hat{\lambda}^{\text{naive}} > 1$, which is in this case far from the true value.

A better method is to use the known probability $P(X|\lambda)$ to infer the most likely value of the parameter $\lambda$.

We don't have any extra information to specify the prior distribution $P(\lambda)$, which can take any value in the interval $[0, +\infty)$, and we can't define a uniform distribution on this interval, so we will focus on finding **maximum likelihood estimator**, where the knowledge of prior distribution is not required.

The likelihood, $P(x_i|\lambda)$, or the probability of getting a measurement $x_i \in X$ for $i = 1, 2, ..., n$ for a given $\lambda$ is:

$$P(x_i|\lambda) = \frac{1}{\lambda Z(\lambda)} e^{-x_i/\lambda}, \tag{3.13}$$

where $Z(\lambda)$ is a normalization factor used to normalize the probability, since in our experiment we can only detect particles that decay in the range $[1, 20]$, rather than at all possible decay positions. To compute $Z(\lambda)$, we integrate the probability of decay over the detector range $[1, 20]$ :

$$Z(\lambda) = \int_1^{20} \frac{1}{\lambda} e^{-x_i/\lambda} dx_i = \Big(e^{-1/\lambda} - e^{-20/\lambda}\Big).$$

The joint probability to obtain all $n$ measurements in $X$ is the product of all the individual likelihoods, because all measurements are independent:

$$\begin{aligned} P(X|\lambda) = \prod_{i=1}^n P(x_i|\lambda) &= \Big(\frac{1}{\lambda Z(\lambda)}\Big)^n \exp\Big(-\frac{1}{\lambda}\sum_{i=1}^n x_i\Big) \\ &= \left[\frac{1}{\lambda Z(\lambda)} \exp\left(-\frac{1}{\lambda}\bar{x}\right)\right]^n. \end{aligned} \tag{3.14}$$

One may find the value of $\hat{\lambda}$ numerically, by maximizing $\frac{1}{\lambda Z(\lambda)} \exp\left(-\frac{1}{\lambda}\bar{x}\right)$ with respect to $\lambda$. The result for $\hat{\lambda}^*$ does not have to lie within the detector range, and is a better estimate of $\lambda^*$ than simply the mean $\bar{x}$.

## 3.3 Probabilistic view of Linear regression

So far, we have treated linear regression as a purely deterministic optimization problem—finding parameters that minimize a loss function. Now, let us apply the principles of statistical inference to this same problem.

We will assume, that the labels are generated by a "ground truth" process, that takes a vector $X_\mu \in \mathbb{R}^d$ for $\mu = 1, \ldots, n$ and generates a label $y_\mu \in \mathbb{R}$ as follows:

$$y_\mu = X_\mu^T w^* + \xi_\mu, \tag{3.15}$$

where $w^* \in \mathbb{R}^d$ are weights sampled from a Gaussian distribution $\mathcal{N}(0, \sigma I_d)$:

$$P(w^*) = \prod_{i=1}^d \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2\sigma}(w_i^*)^2}. \tag{3.16}$$

And the noise terms $\xi_\mu$ for $\mu = 1, 2, \ldots, n$ are independent identically distributed (i.i.d.) Gaussian variables[7]:

$$P(\xi_\mu) = \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(\xi_\mu)^2}. \tag{3.17}$$

[7]Note that noise and weights can be sampled from other distributions, which we will discuss later. However, the Gaussian distribution provides a direct connection to the least squares and $L_2$ regularization methods we encountered earlier.

Given this model, we are able to write the distribution of label $y$ conditioned on the weights $w$ and input to the model $X$:

$$\begin{aligned} P(y|w,X) &= P(\xi_\mu = y_\mu - X_\mu^T w,\ \forall \mu = 1,\dots,n|w,X) \\ &= \prod_{\mu=1}^{n} P(\xi_\mu = y_\mu - X_\mu^T w|w,X) \\ &= \prod_{\mu=1}^{n} \frac{1}{\sqrt{2\pi\Delta}} \exp\left\{\left[-\frac{1}{2\Delta}\left(\underbrace{y_\mu - \sum_{i=1}^{d} X_{\mu,i} w_i}_{\xi_\mu^2}\right)^2\right]\right\} \\ &= \frac{1}{(\sqrt{2\pi\Delta})^n} \exp\left\{\left[-\frac{1}{2\Delta}\sum_{\mu=1}^{n}\left(y_\mu - \sum_{i=1}^{d} X_{\mu,i} w_i\right)^2\right]\right\}. \end{aligned} \tag{3.18}$$

Here we notice that the randomness of the labels in the model (3.15) conditioned on $w$ comes only from the additive noise and then used the fact that the noise $\xi_\mu$ is independent in all samples and distributed according to (3.17).

Now we are able to find the **maximum likelihood estimator** (MLE) $\hat{w}_{\text{ML}} = \arg\max_w P(y|X,w)$. After careful examination of the likelihood (3.18), we can notice that it has the form $c_1 e^{-c_2 f(X,y,w)}$, where $c_1, c_2 > 0$ are some constants. Taking the logarithm of such functions often simplifies the problem:

$$\begin{aligned} \arg\max_w c_1 e^{-c_2 f(X,y,w)} &= \arg\max_w \log\left(c_1 e^{-c_2 f(X,y,w)}\right) \\ &= \arg\max_w -c_2 f(X,y,w) \\ &= \arg\min_w f(X,y,w). \end{aligned}$$

Thus, we obtain

$$\hat{w}_{\text{ML}} = \arg\min_w \sum_{\mu=1}^{n}\left(y_\mu - \sum_{i=1}^{d} X_{\mu,i} w_i\right)^2. \tag{3.19}$$

Notice that the MLE (3.19) is simultaneously the minimizer of the square loss (2.6). This equivalence arises from our assumptions about the noise structure. *Therefore, the ordinary least squares method finds the MLE under the assumption of i.i.d. additive Gaussian noise.*

The **maximum a posteriori estimator** (MAP) is given by $\hat{w}_{\text{MAP}} = \arg\max_w P(w|X,y)$. Using Bayes' theorem (3.10), we can express the posterior distribution $P(w|X,y)$ as

$$\begin{aligned} P(w|X,y) &= \frac{P(y|w,X)P(w|X)}{P(y|X)} \\ &= \frac{1}{P(y|X)} \cdot \frac{1}{(2\pi\Delta)^{\frac{n}{2}}} \exp\left(-\frac{1}{2\Delta}\sum_{\mu=1}^{n}\left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i\right)^2\right) \cdot \frac{1}{(2\pi\sigma)^{\frac{d}{2}}} \exp\left(-\frac{1}{2\sigma}\sum_{i=1}^{d} w_i^2\right). \end{aligned} \tag{3.20}$$

One useful modification of the optimization problem is to maximize the logarithm of the function instead of the function itself. By taking the natural logarithm of (3.20) and ignoring all constants outside the exponential terms, the optimization problem becomes:

$$\begin{aligned} \hat{w}_{\text{MAP}} &= \arg\max_w \left[P(w|X,y)\right] = \arg\max_w \left[\log\left(P(w|X,y)\right)\right] \\ &= \arg\min_w \left[\sum_{\mu=1}^{n}\left(y_\mu - \sum_{i=1}^{d} X_{\mu i} w_i\right)^2 + \frac{\Delta}{\sigma}\sum_{i=1}^{d} w_i^2\right]. \end{aligned} \tag{3.21}$$

Here, we recognize the ridge regression loss (2.14) with regularization strength $\lambda = \frac{\Delta}{\sigma}$. *Therefore, ridge regression finds the MAP estimator under the assumption of Gaussian prior and i.i.d. additive Gaussian noise.*

### 3.3.1 Examples – Inverse Problems in Signal Processing

Let us consider some examples of linear regression problems where the weights $w^*$ have concrete physical meaning. Given an indirect measurements, we want to estimate the "ground truth" parameters.

**X-ray Computed Tomography (CT Scans).** In X-ray Computed Tomography X-ray source and detector rotate around a sample to take 2D projections from many angles, from which a 3D image of an object is reconstructed. To render the image, first, we should infer the absorption coefficients at each "voxel" (3D pixel) in the sample.

Suppose, there are $d$ voxels in total and we wish to know the absorption coefficients $w^* = (w_1^*, \dots, w_d^*)^T \in \mathbb{R}^d$. For each position of the X-ray source and detector and for each pixel in the obtained 2D projection there is a specific X-ray path, that is characterized by the voxels it passed through. For the path $\mu$, we can specify a vector $X_\mu \in \mathbb{R}^d$, such that

$$X_{\mu,i} = \begin{cases} 1, & \text{if X-ray } \mu \text{ passes through voxel } i, \\ 0, & \text{otherwise.} \end{cases}$$

The CT Scan detector measures the intensities $y_\mu$, representing the total absorption along each path, which is the sum of absorption coefficients $w_i$ for all voxels $i$ that X-ray $\mu$ passes through, plus some noise. In matrix notation, we can write

$$y_\mu = X_\mu^T w^* + \varepsilon_\mu.$$

Now, we are able to estimate the absorption coefficients $\hat{w}^*$ by solving linear or ridge regression[8].

**Nuclear Magnetic Resonance (NMR) Imaging.** Similar to CT, NMR Imaging reconstructs a 3D image of molecular composition of the sample from indirect measurements. The algorithm using Fourier transform of the observed measurements[9] is based on the nuclear magnetic precision measurements[10], and again leads to a linear model formulation $y = Xw + \varepsilon$, where $y$ represents the measurements and $X$ specifies the configuration of the machine that takes the measurements.

### 3.3.2 Generalizing Priors and Losses

So far, we have assumed a simple Gaussian prior for our parameters and Gaussian noise. However, selecting different priors leads to different models, each suited for different types of problems. It will be useful to provide an explicit link between an arbitrary choice of prior and likelihood and a corresponding loss minimization problem.

We will use the following general assumptions:

- the labels $y_\mu$ depend on the linear combination of features $X\mu$, so that $P(y|X, w) = P(y|X \cdot w)$;
- the samples are independent, and therefore we can write $P(y|X, w) = \prod_{\mu=1}^{n} P(y_\mu|X_\mu^T w)$;
- the prior is separable, i.e. the components of the vector $w$ are independent, so we can write $P(w) = \prod_{i=1}^{d} P(w_i)$.

We can now derive a new form for the optimization problem for finding the MAP estimator. Taking the logarithm of the posterior probability (3.11), disregarding the constant $-\log(P(y|X))$, which appears from the normalization term, and dividing by constant $\frac{1}{n}$, we get:

$$\max_w P(w|X, y) \Leftrightarrow \min_w \mathscr{L}(w) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{\mu=1}^{n} -\log\big(P(y_\mu|X_\mu^T w)\big) + \frac{1}{n} \sum_{i=1}^{d} -\log(P(w_i)). \tag{3.22}$$

First term of this new minimization objective is called **negative log-likelihood**.

In machine learning, in general, we will mostly encounter optimization problems of similar form:

$$\mathscr{L}(w) = \frac{1}{n} \sum_{\mu=1}^{n} \underbrace{l(y_\mu, f_w(X_\mu))}_{\text{loss function}} + \frac{\lambda}{n} \sum_{i=1}^{d} \underbrace{r(w_i)}_{\text{regularization}}, \tag{3.23}$$

were $f_w(\cdot)$ is the model.

Previously, we have seen the example of ridge regression with linear model $f_w(X) = X^T w$, loss $l(y, z) = (y - z)^2$ and regularization $r(w) = w^2$.

[8] $L_2$ regularization for linear regression was independently developed in signal processing (where it is often called Tikhonov regularization) and statistics

[9] for which Richard Ernst was awarded the Nobel Prize in Chemistry in 1991

[10] for which Felix Bloch and Edward M. Purcell were awarded the Nobel Prize in Physics in 1952

### 3.3.3 Robust Regression

Least squares regression (Gaussian noise assumption) is sensitive to outliers because the squared differences between the label and the predicted value of such outliers dominate the loss function. From the statistical point of view, it happens because large noise values are highly improbable under a Gaussian distribution.

To counteract this, we can assume a different noise distribution with heavier tails (i.e. with larger probabilities of large noise values), in particular, the Laplace distribution (see Figure 6)

$$P(x) \sim \exp(-\gamma|x-\mu|). \tag{3.24}$$

We assume noise distribution to have zero mean $\mu = 0$.

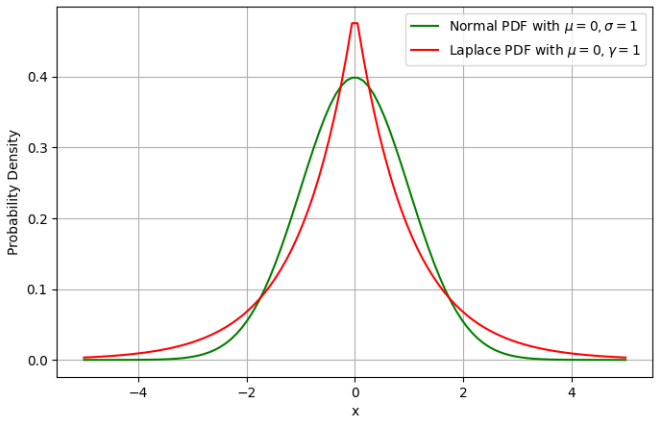



Figure 6: Gaussian vs Laplace Distribution PDFs.

This leads to **Mean Absolute Error (MAE)** loss:

$$l(y,z) = |y-z|, \tag{3.25}$$

which is more robust to outliers than MSE because large errors contribute linearly rather than quadratically, making them less dominant in the total loss.

### 3.3.4 Sparse Linear Regression (Lasso)

Sometimes, we want to select only a few important features from the input data, and for that we need to promote sparsity in the components of the estimated weights $\hat{w}$, i.e. find the weights $\hat{w}$ such that most of its components are 0. This can be done by assuming the weights are distributed according to a Laplace prior (3.24) with zero mean.

This leads to $L_1$ regularization:

$$r(w) = \|w\|_1 = \sum_{i=1}^{d} |w_i|. \tag{3.26}$$

Linear regression with square loss and $L_1$ regularization is often called Lasso regression.

$$\mathscr{L}(w) = \frac{1}{n}\sum_{\mu=1}^{n}(y_\mu - X_\mu \cdot w)^2 + \lambda\frac{1}{n}\sum_{i=1}^{d}|w_i|\,. \tag{3.27}$$

To better understand the difference between $L_1$ and $L_2$ regularization, we can turn to the geometry of the "loss landscape". Figure 7 illustrates this concept. The green dot represents the minimizer $w \in \mathbb{R}^2$ of the square loss, with green level curves showing constant square loss values. The red dot represents the minimizer of the regularization term (which is $w = \vec{0}$ in both cases), with black curves showing constant regularization values.

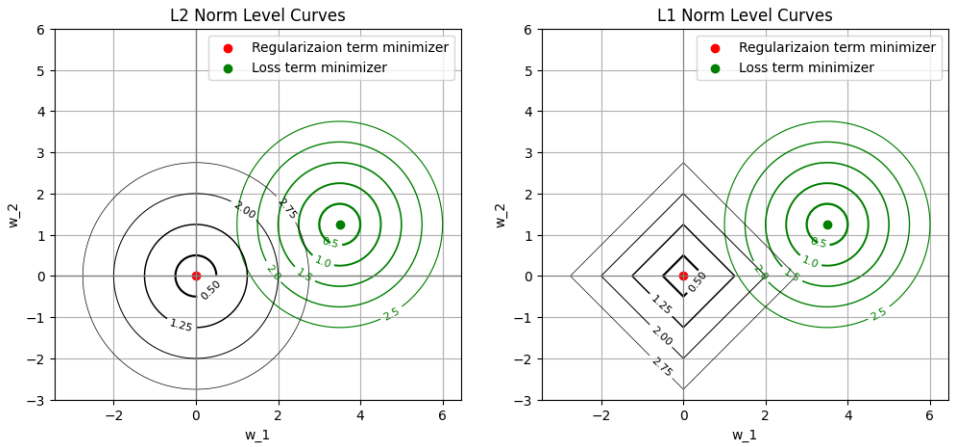



Figure 7: Square loss and regularization term level curves.

Imagine solving a constrained optimization problem of finding the parameters $w$ with the smallest square loss under the constraint $\|w\|_p \leq r$ for $p = 1, 2$ for some $r > 0$. Graphically, the solution is the point of tangency between the black curve of level $r$ and one of the green curves. Notice that the minimizer of regularized loss should also be one of such tangent points. Now, it's easy to see that for any level $r$, the point of tangency with the $L_1$ level curve lies close to the "corner", where one of the coordinates is almost zero, while the point of tangency with the $L_2$ level curve does not necessarily have this property.

Sparsity of the parameters produced by this model is useful in many cases. For example, when predicting a patient's survival time, out of millions of features, only a small subset, like certain medical conditions, might be relevant, while others, like their address or eye color, might not be.

Another important application is efficient Image Reconstruction. It's known that most images are sparse in Haar wavelets basis, so that $s^* = Gw^*$, where $w^*$ is a sparse ground truth vector in wavelets basis and $G \in \mathbb{R}^{d\times d}$ is a transition matrix. Thus, we can reformulate our initial problem

$$y_\mu = X_\mu s + \varepsilon_\mu$$

to

$$y_\mu = \tilde{X}_\mu w + \varepsilon_\mu, \tag{3.28}$$

where $\tilde{X}_\mu = X_\mu G$ and we have the prior knowledge that $w$ is sparse. This allows us to use less samples $X_\mu$, since we don't need to have more samples $n$ than dimensions $d$, as we add the regularization and the problem becomes well defined. Such reduction in the number of measurements is particularly beneficial in applications like medical imaging, where it leads to reduced patient scan time.

Even though minimizing the loss (3.27) produces parameters with desirable properties, no closed-form solution exists for this minimization problem. In the next section, we will describe a popular iterative algorithm for solving such continuous optimization problems.

# 4 Gradient Descent

## 4.1 Basic algorithm

In many Machine Learning problems, there is no explicit formula for the minimizer of the loss function. In such cases, iterative optimization algorithms are applied. Gradient descent (GD) is one of the most popular optimization algorithms that is used when the optimized function is differentiable[11]. In fact, most of the modern state-of the art models, such as ChatGPT, are trained by minimizing a loss function with a variant of gradient descend.

The algorithm is based on the observation that if the multivariate function $f$ is defined and differentiable in the neighborhood of a point $x_0$, then the direction of the negative gradient $-\nabla f(x_0)$ is the direction of the fastest decrease of the function $f$.

Let us discuss the steps of the algorithm (the pseudo-code is presented in the listing 1):

[11] or when the subgradient can be defined in the points where the function is non-differentiable, which will be discussed later in this section

**Algorithm 1** Gradient descent algorithm

**Require:** $w_0 \in \mathbb{R}^d, \mathscr{L}(\cdot) : \mathbb{R}^d \to \mathbb{R}, \gamma > 0, \delta > 0, t_{\max} \in \mathbb{N}$
  $w^{(0)} \leftarrow w_0$
  $t \leftarrow 0$
  **repeat**
    $t \leftarrow t + 1$
    $w^{(t)} \leftarrow w^{(t-1)} - \gamma \nabla_w \mathscr{L}(w^{(t-1)})$
  **until** $|\mathscr{L}(w^{(t)}) - \mathscr{L}(w^{(t-1)})| < \delta$ or $t > t_{\max}$

1. **Initialization.** The algorithm requires an initial point from which to begin iterating. The initialization method can be considered a hyperparameter, as it sometimes affects the final solution (as we will see later in this section).

2. **Iteration.** At each time step the parameters $w_i, i = 1, \ldots, d$ are updated according to

$$w_i^{t+1} = w_i^t - \gamma \frac{\partial \mathscr{L}(\vec{w})}{\partial w_i}\bigg|_{\vec{w}^t}.$$

   Here, $\gamma$ is a **learning rate** – a hyperparameter that controls the "step size" in the direction of the steepest decrease of the function. The learning rate can be constant or time dependent. We will discuss the role of $\gamma$ in the next paragraph.

3. **Stopping Criteria.** The algorithm stops when the increment between successive steps is very small, which implies the derivatives are close to zero. This could indicate a local minimum or a saddle point. The convergence threshold is controlled by hyperparameter $\delta$.

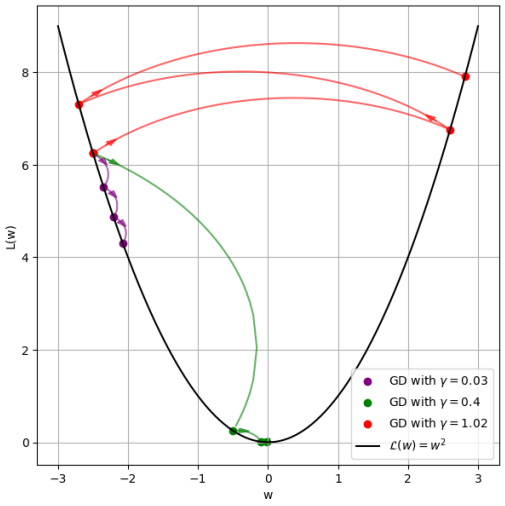
   However, the algorithm can be stopped before full convergence, specifically when the validation error stops decreasing or starts increasing. This method, called **Early Stopping**, treats the maximum iteration time $t_{\max}$ as a hyperparameter, chosen through validation. This is particularly important because, while training loss monotonically decreases, validation loss can increase due to overfitting.

Choosing the right learning rate $\gamma$ is crucial for the convergence of GD. As can be seen in the example in Figure 8a, choosing overly small learning rate leads to slow convergence, while overly high learning rate leads to instability by "jumping over" the optimum.

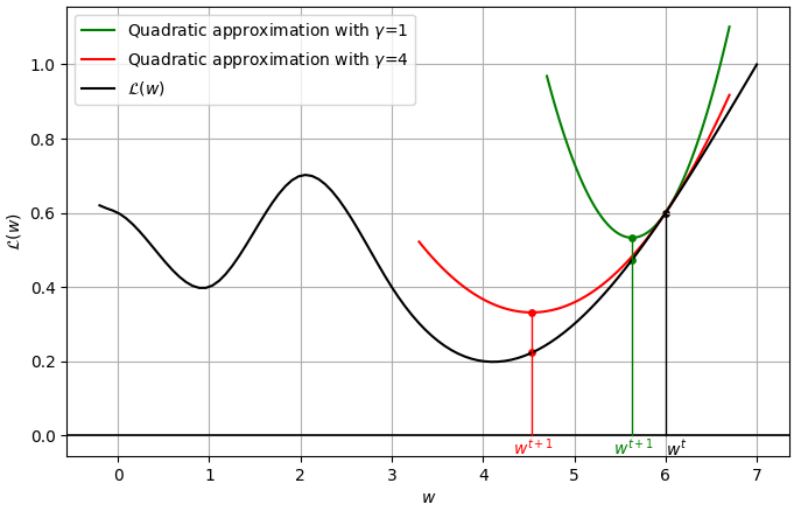
To better understand the role of learning rate, consider the following interpretation. Gradient descent can be viewed as approximating the loss function locally with a parabola that shares the same value and first derivative at the current point $\mathbf{w}^t$. The second derivative of this approximating parabola is set to $\frac{1}{2\gamma}$:

$$\mathscr{L}(w) \approx \mathscr{L}(w^t) + \frac{\partial \mathscr{L}(w)}{\partial w}\bigg|_{w^t}(w - w^t) + \frac{1}{2}\frac{1}{\gamma}(w - w^t)^2 \,. \tag{4.1}$$

Minimizing this quadratic approximation leads directly to the gradient descent update rule. Notice, that $\gamma$ corresponds to the "width" of the approximating parabola. A small $\gamma$ results in a "narrow" parabola and small steps, while a large $\gamma$ results in a "wide" parabola and larger steps, potentially overshooting the true minimum (see Figure 8b).




(a) Iterations of gradient descent with different values of learning rate on the loss function $\mathscr{L}(w) = w^2$.




(b) Approximation of the loss with parabolas of different width.

Figure 8: Role of learning rate $\gamma$ in gradient descent.

## 4.2 Implicit Regularization of Gradient Descent

Generally, in the high-dimensional regime, when dimension of the problem is larger than the number of samples $d > n$ (this is particularly relevant in modern machine learning), the OLS regression has many global minimizers where the loss is exactly zero. We will call these solutions interpolants. Geometrically, this creates a "canyon" of solutions, a linear subspace where the loss is zero. A profound property of gradient descent is its implicit regularization: with appropriately chosen hyperparameters, the algorithm converges to the minimum $L_2$ norm interpolant, i.e. finds the solution with the smallest $L_2$ norm among all the minimizers of the loss.

**Proposition 1** (Implicit Bias of Gradient Descent)**.** *If gradient descent with small enough learning rate is applied to find parameters $w$ of linear regression with squares loss, parameters are initialized with zero $w^0 = \vec{0}$, and it runs until convergence, then it will converge to the minimum norm interpolant. That is, GD finds the $w$ that minimizes the loss and has the smallest $L_2$ norm.*

Let us give a brief justification for this proposition without a formal proof. It comes from three facts:

- All global minimizers can be written as the specific solution which lies in the span of the input data $X$ (which is unique due to the strict convexity of the problem) plus a vector orthogonal to the span of $X$. The value of the loss does not depend on this orthogonal component. Thus, among all global minimizers, the one with the smallest norm is precisely the one that lies entirely within the span of $X$.
- For OLS regression, the gradient is always a linear combination of the data points (see equation (2.7)), meaning it lies within the span of $X$. And since the algorithm starts at $w^0 = \vec{0}$ and all gradient updates are in the span of $X$, the parameter vector $w^t$ at any time $t$ will also be a linear combination of the data points.
- Due to the convexity of the problem, gradient descend converges to the global minimizer. Combining it with the fact, that this minimizer lies in the span of $X$, we notice, that the solution must be the smallest norm interpolant.

Even without explicitly adding a regularization term to the loss function, gradient descent, by virtue of its initialization at zero and convergence, naturally selects the solution that minimizes the $L_2$ norm. This means that in many practical scenarios the need for explicit $L_2$ regularization might be implicitly handled by the optimization algorithm itself. This concept is believed to extend to more complex loss functions and neural networks, suggesting that gradient descent inherently guides the model towards "useful" solutions.

## 4.3 Variants of Gradient Descent

To address practical challenges like computational cost and convergence speed, several variants of gradient descent have been developed.

### Subgradient Descent

In cases where loss function is not differentiable everywhere, but still has left and right derivatives in the points where it's not differentiable

$$a = \lim_{w \to w_0^-} \frac{\mathscr{L}(w) - \mathscr{L}(w_0)}{w - w_0}, \qquad b = \lim_{w \to w_0^+} \frac{\mathscr{L}(w) - \mathscr{L}(w_0)}{w - w_0}, \tag{4.2}$$

we can use any value between $a$ and $b$ instead of a partial derivative in the gradient descent iteration. Such values are called subgradients, and using them allows to extend application of gradient descent to optimizing such functions as $|x|$.

For example, in Lasso regression (3.27), when for some $i$ the weight $w_i = 0$, the derivative $\frac{\partial \mathscr{L}}{\partial w_i}$ is not defined, because of theregularization term $|w_i|$. However, $|w_i|$ has both left and right derivatives in this point, equal to -1 and +1 respectively. Therefore, we can use any value in $[-1, +1]$ instead of the undefined $\frac{\partial |w_i|}{\partial w_i}$ [12].

### Stochastic Gradient Descent (SGD)

Usual machine learning problem formulation (3.23) implies derivatives of the form

$$\frac{\partial \mathscr{L}(w, X, y)}{\partial w_i} = \frac{1}{n} \sum_{\mu=1}^{n} \frac{\partial l(y_\mu, f_w(X_\mu))}{\partial w_i} + \frac{\lambda}{n} \frac{\partial r(w_i)}{\partial w_i}. \tag{4.3}$$

Computing such expressions at every iteration of GD becomes expensive with growing dataset size $n$. Stochastic gradient descent approximates the gradient by using only a mini-batch (a small subset of the training set) $B_t \in \{1, \ldots, n\}$ at each time step. First, the dataset is randomly partitioned into disjoint mini-batches of equal size $B_1 \sqcup \ldots \sqcup B_{t_{\max}} = \{1, \ldots, n\}$. Then, the weight updates become:

$$w_i^{t+1} = w_i^t - \gamma \left( \frac{1}{n} \sum_{\mu \in B_t} \frac{\partial l(y_\mu, f_w(X_\mu))}{\partial w_i} + \frac{\lambda}{n} \frac{\partial r(w_i)}{\partial w_i} \right). \tag{4.4}$$

This process can be repeated for several "epochs", reshuffling mini-batches at each new pass. The size of mini-batches $|B_t|$ and number of epochs are hyperparameters that should be chosen through validation.

SGD iterations are significantly faster then GD iterations (by a factor $n/|B_t|$). Moreover, "noise" introduced by this procedure (compared to the gradient computed over all points in the dataset) is often beneficial for the generalization of the model.

### Momentum and Adaptive Learning Rate

There are several methods to address the practical challenges of optimization when the loss landscape has a complicated shape with flat regions or narrow local minima. One such method is momentum. It is designed to help navigate flat regions of the loss landscape and to stabilize training with SGD (when the gradient follows a "zig-zag" pattern because of the mini-batch noise). Inspired by classical mechanics, momentum-based variants of gradient descent incorporate information from previous gradient updates to maintain "momentum" in a particular direction. This approach helps accelerate convergence and enables the optimizer to navigate through flat regions and reduce oscillations. The gradient step with momentum can be written as

$$\begin{aligned} m^{(t)} &= \beta m^{(t-1)} + (1 - \beta) \nabla_w \mathscr{L}(w^{(t-1)}), \\ w^{(t)} &= w^{(t-1)} - \gamma m^{(t)}, \end{aligned} \tag{4.5}$$

where term $m^{(t)}$ accumulates the gradient from the previous steps by computing exponential moving average.

Adaptive learning rate is another method, that counteracts the problem of different scales of gradient w.r.t. different parameters or in different regions of the loss landscape. Instead of a constant learning rate $\gamma$, the learning rate is dynamically adjusted at each step, either by a predefined schedule or by adapting based on the gradients observed (e.g., larger steps in flat regions, smaller steps near minima). This allows us to quickly converge to the

[12] typically, it's set to 0 in this case

neighborhood of the minimum with larger learning rate, while reducing the learning rate when needed to obtain a more precise updates. One such algorithm is called RMSProp and it uses the following update step:

$$\begin{aligned} g^{(t)} &= \nabla_w \mathscr{L}(w^{(t-1)}), \\ v^{(t)} &= \beta v^{(t-1)} + (1-\beta)(g^{(t)})^2, \\ w^{(t)} &= w^{(t-1)} - \gamma \frac{g^{(t)}}{\sqrt{v^{(t)}} + \varepsilon}, \end{aligned} \tag{4.6}$$

where $0 < \varepsilon \ll 1$ is a small constant for numerical stability. In this algorithm $v^{(t)} \in \mathbb{R}^d$ accumulate the norm of the gradient of the loss w.r.t. all parameters, and adjusts the step accordingly.

A very popular algorithm combining this two ideas is called ADAM. In this algorithm, running averages with exponential forgetting of both the gradients ($m^{(t)}$ – the momentum part) and the second moments of the gradients ($v^{(t)}$ – the adaptive learning rate part) are used. The full pseudocode is given in listing 2.

**Algorithm 2** ADAM algorithm

**Require:** $w_0 \in \mathbb{R}^d, \mathscr{L}(\cdot) : \mathbb{R}^d \to \mathbb{R}, \alpha > 0, \beta_1 \in [0,1), \beta_2 \in [0,1), \delta > 0, \varepsilon > 0, t_{\max} \in \mathbb{N}$
  $w^{(0)} \leftarrow w_0$
  $m^{(0)} \leftarrow 0$
  $v^{(0)} \leftarrow 0$
  $t \leftarrow 0$
  **repeat**
    $t \leftarrow t + 1$
    $g^{(t)} \leftarrow \nabla_w \mathscr{L}(w^{(t-1)})$
    $m^{(t)} \leftarrow \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)}$
    $v^{(t)} \leftarrow \beta_2 v^{(t-1)} + (1-\beta_2)(g^{(t)})^2$
    $\hat{m}^{(t)} \leftarrow \frac{m^{(t)}}{1-\beta_1^t}$
    $\hat{v}^{(t)} \leftarrow \frac{v^{(t)}}{1-\beta_2^t}$
    $w^{(t)} \leftarrow w^{(t-1)} - \alpha \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \varepsilon}$
  **until** $|\mathscr{L}(w^{(t)}) - \mathscr{L}(w^{(t-1)})| < \delta$ or $t > t_{\max}$

Notice that momentum and step-size scaling factor get adjusted: $\frac{m^{(t)}}{1-\beta_1^t}$, $\frac{v^{(t)}}{1-\beta_2^t}$. Since $m$ and $v$ are initialized at 0, it leads to the first several estimates $m^{(t)}$ and $v^{(t)}$ being biased towards zero as well. The adjustment helps to correct for this bias in the early steps, and at further time steps it doesn't affect the terms as much, because $\beta_1^t$ and $\beta_2^t$ exponentially decay to 0.

This algorithm combines the benefits from several techniques, however, when L2 regularization is added to the loss, it affects the adaptive learning rate which leads to inconsistent regularization. AdamW is a version of Adam algorithm that decouples weight decay (regularization) from the gradient update step. To understand the motivation, first let us take a look at the gradient descent step for a loss function of the form $\mathscr{L}(w) + \lambda \|w\|_2^2$:

$$w_i^{t+1} = w_i^t - \gamma \left( \frac{\partial \mathscr{L}(\vec{w})}{\partial w_i}\bigg|_{\vec{w}^t} + \lambda w_i^t \right).$$

We notice that there are two terms affecting the weight at each step: the gradient of the loss, that "pushes" the weight toward the direction of smaller loss and the *weight decay* that comes from the regularization term and "pushes" the weight towards zero.

So, in AdamW, instead of including the regularization directly into the loss, we add the *weight decay* term and the weight update is rewritten as:

$$w^{(t)} = w^{(t-1)} - \gamma \left( \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \varepsilon} + \lambda w^{(t-1)} \right). \tag{4.7}$$

This allows us to control the regularization during training more precisely. AdamW algorithm is very popular in practice and is often used to train Large Language Models and other state-of-the-art architectures.

# 5 Linear Classification

Now we turn to classification, where the task is to predict discrete categories or classes. Linear classification provides a simple yet powerful approach to this problem by finding linear decision boundaries that separate different classes.

## 5.1 Binary Classification

Consider the following example. We are given a (very famous, see [2]) dataset with records of sepal length and sepal width from Iris flowers of two species: setosa and versicolor. After plotting this data in Figure 9, we notice that points belonging to different classes (species) are **linearly separable** – there exists a line (one-dimensional hyperplane) such that all points of one class lie on one side of this line and all points of the other class lie on the opposite side.

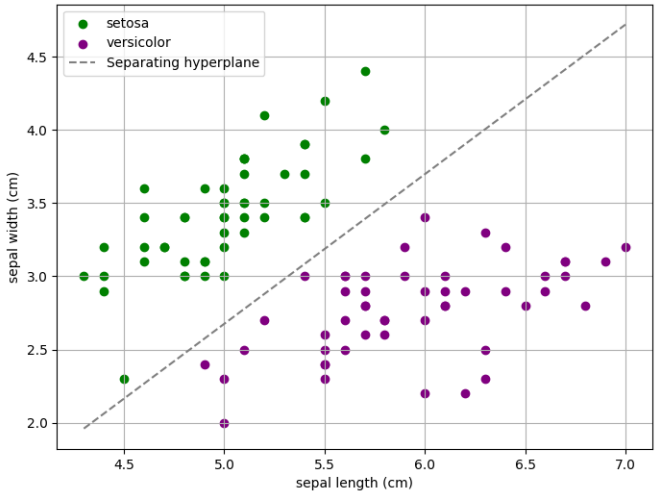



Figure 9: Sepal length vs sepal width in two species of Iris.

Therefore, to classify new samples we can just check, if the new point is above or below the separating line. We can formalize this problem as follows: given the data samples $X^\mu \in \mathbb{R}^p$ in $p$ - dimensional space, and labels $y^\mu \in \{\pm 1\}$, we want to find parameters $w \in \mathbb{R}^{p+1}$, such that a hyperplane

$$w_0 + \sum_{i=1}^{p} w_i X_i^\mu = 0 \tag{5.1}$$

is a separating hyperplane, i.e.

$$w_0 + \sum_{i=1}^{p} w_i X_i^\mu > 0 \iff y^\mu = +1. \tag{5.2}$$

In matrix notation, we would append a constant feature equal 1 to each sample and write the separating hyperplane in the form

$$Xw = 0, \tag{5.3}$$

where

$$w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_p \end{pmatrix}, \quad X = \begin{pmatrix} 1 & X_1^1 & \vdots & X_p^1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & X_1^n & & X_p^n \end{pmatrix}.$$

Notice that we can write a classifying model as

$$y^\mu = \text{sign}(w^T X^\mu). \tag{5.4}$$

Now, that we have our model (5.4), we can define the loss function of the form $\mathscr{L}(w) = \sum_{\mu=1}^{n} l(y_\mu, w^T X^\mu)$ and optimize it to find the parameters $w$ of the separating hyperplane. Let us explore the choices for element-wise loss $l(y, z)$ illustrated in the Figure 10:

- **Zero-one loss**:
$$l_{01}(y, z) = \frac{1}{2}(1 - y \cdot \text{sign}(z)), \tag{5.5}$$
which corresponds to counting the number of mismatches (i.e., the terms of the sum are $+1$ for misclassified points and 0 for well-classified points). This loss is not suitable for gradient descent minimization as $l_{01}$ has zero derivative almost everywhere, and therefore is not used in practice.

- **Least-square loss** :
$$l_{\text{LS}}(y, z) = (y - z)^2, \tag{5.6}$$
which corresponds to treating the problem exactly as a regression that we have seen before and trying to predict a label as $y^\mu = w^T X^\mu$. However, this loss function penalizes correctly classified points, as soon as the absolute value of prediction is not exactly 1. For instance, even if $X_\mu \cdot w$ is correctly positive but much larger than $+1$, there will be a penalization (which is surprisingly not leading to a bad classifier in practice).

- **Hinge loss** :
$$l_{\text{hinge}}(y, z) = \max(0, 1 - yz), \tag{5.7}$$
which penalizes the incorrectly classified points and also encourages a margin, i.e., not only misclassified points are penalized but also data points that are close to the linear separator (hyperplane). This loss thus often leads to more robust classification.

- **Logistic loss**
$$l_{\text{logistic}}(y, z) = \log\left(1 + e^{-yz}\right). \tag{5.8}$$
The logistic loss has the same behavior as the hinge loss at $\pm\infty$, but it is smooth at $yz = 1$. Linear classification with logistic loss is called **logistic regression**. It is one of the most widely used classical machine learning models.

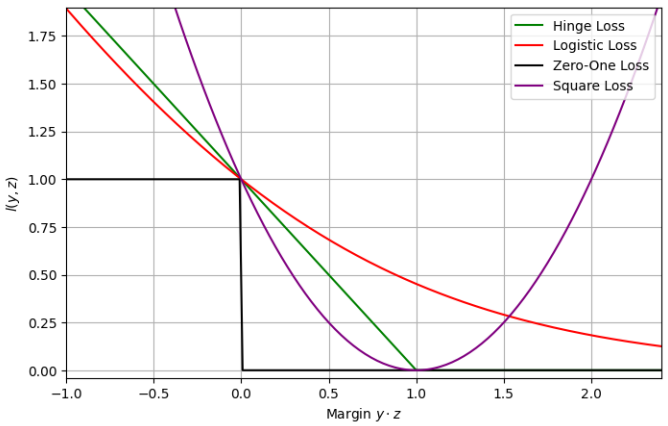



Figure 10: Loss $l(y, z)$ as a function of margin $yz$.

### Probabilistic interpretation of Logistic Regression

Logistic regression can be seen as a maximum likelihood estimator for the following probabilistic model. We assume, there is a "propensity" to belong to a positive class ($y = +1$), which is described by a linear model with ground truth parameters $w^*$:
$$z^\mu = \sum_{i=1}^{d} X_i^\mu w_i^*,$$
So that the larger $z^\mu$, the more likely the point belongs to a positive class (i.e. the probability to pass the exam should hopefully be larger with more hours of studying). To describe this dependency between the propensity and

the probability of belonging to a certain class, we will use the following distribution:

$$P_{\text{out}}(y_\mu|(w^*)^T X_\mu) = \frac{\exp\left(y_\mu \sum_{i=1}^d X_{\mu i} w_i^*\right)}{\exp\left(\sum_{i=1}^d X_{\mu i} w_i^*\right) + \exp\left(-\sum_{i=1}^d X_{\mu i} w_i^*\right)} = \frac{1}{1+\exp\left(-2y_\mu \sum_{i=1}^d X_{\mu i} w_i^*\right)} . \tag{5.9}$$

Notice that we normalized the probability using the fact that $y_\mu$ has only two possible values, $+1$ and $-1$.

To get the MLE for this model, we take the product of $P_{\text{out}}(y_\mu|w^T X_\mu)$ over all the students and maximize its logarithm:

$$\begin{aligned} \hat{w}_{\text{ML}} &= \arg\max_w \left[\log \prod_{\mu=1}^n P_{\text{out}}(y_\mu|w^T X_\mu)\right] \\ &= \arg\max_w \left[-\sum_{\mu=1}^n \log\left(1+\exp\left(-2y_\mu \sum_{i=1}^d X_{\mu i} w_i\right)\right)\right] \\ &= \arg\min_w \left[\sum_{\mu=1}^n \log\left(1+\exp\left(-2y_\mu \sum_{i=1}^d X_{\mu i} w_i\right)\right)\right] . \end{aligned} \tag{5.10}$$

We recognize the logistic loss function (5.8) with an additional constant in the exponential. This factor of 2 can, however, be included in the weights and does not matter. The logistic loss is therefore retrieved from the initial assumption on $P_{\text{out}}$ (5.9). As shown in Figure 11, it implies that the larger $w^T X_\mu$, the larger $P(y_\mu = 1)$.

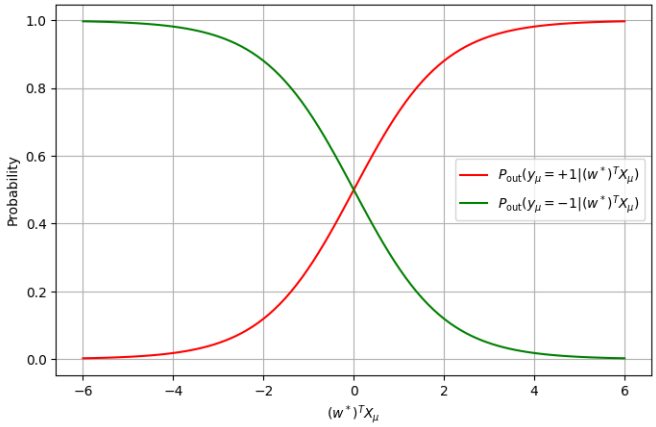



Figure 11: Probabilistic model $P_{\text{out}}(y_\mu|(w^*)^T X_\mu)$ leading to the logistic regression.

## 5.2 Multiclass Classification

If the classification problem has $k > 2$ target classes (e.g., classification of handwritten digits), one natural extension of binary classification would be to train $k$ models to separate one class from the rest. This approach is valid but has several drawbacks: it requires training multiple models, introduces data imbalance (there might be far fewer examples of one class than examples from all other classes combined), and creates ambiguity when multiple classifiers claim the same sample belongs to their respective classes.

Another approach is **multiclass logistic regression**. First, instead of using scalar labels, we encode the classes using so-called **one-hot-encoding**:

$$y_\mu \in \mathbb{R}^k : \quad y_\mu = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \quad \text{with 1 in the } a\text{-th row if } y \text{ belong to class } a\,. \tag{5.11}$$

The weights are combined in a matrix $w^* \in \mathbb{R}^{d\times k}$, where $w^*_{:a}$ corresponds to a weights vector used to calculate the "propensity" for class $a$, so that the probabilistic model becomes:

$$p_{\mu a} = P_{\text{out}}\left(y_{\mu a} = 1 | X_\mu, w^* \in \mathbb{R}^{d\times k}\right) = \frac{\exp\{\sum_{i=1}^{d} X_{\mu i} w^*_{ia}\}}{\sum_{b=1}^{k} \exp\{\sum_{i=1}^{d} X_{\mu i} w^*_{ib}\}}\,, \tag{5.12}$$

where the numerator represents how probable it is that the sample belongs to class $a$ and the denominator is the normalization term. The likelihood then reads:

$$P(y|X,w) = \prod_{\mu=1}^{n}\left[\prod_{a=1}^{k} p_{\mu a}^{y_{\mu a}}\right]\,. \tag{5.13}$$

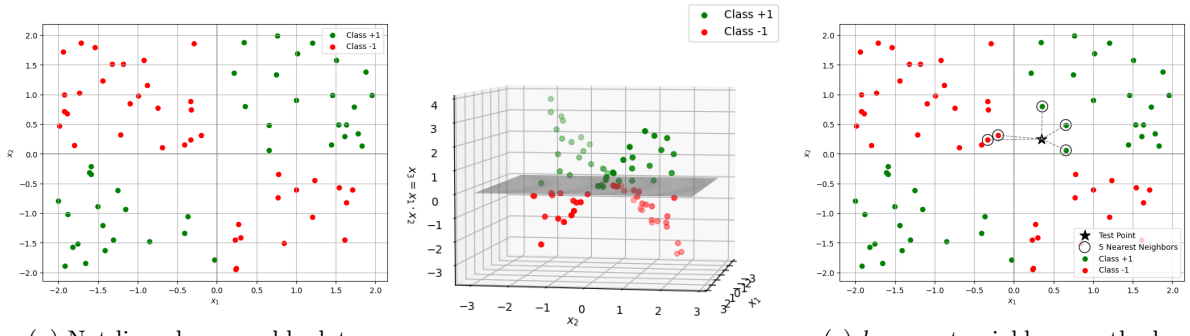
And the negative log-likelihood (i.e. the loss, that we want to minimize) is:

$$\mathscr{L}(w) = -\frac{1}{n}\sum_{\mu=1}^{n}\sum_{a=1}^{k} y_{\mu a}\log(p_{\mu a}) = -\frac{1}{n}\sum_{\mu=1}^{n}\sum_{a=1}^{k} y_{\mu a}\log\left(\frac{e^{X_\mu^T w_a}}{\sum_b e^{X_\mu^T w_b}}\right) \tag{5.14}$$

which is called the **cross-entropy loss**. The function $f_a(\vec{z}) = \frac{e^{z_a}}{\sum_b e^{z_b}}$ is called **softmax**, and we will see other applications of this function later in the course.

The minimization over $w$ leads to an optimal $\hat{w}$. We then calculate $p_{\text{new,a}}$ and predict the class as $\arg\max_a p_{\text{new,a}}$.

## 5.3 How to classify data that are not linearly separable?




(a) Not linearly separable data.

(b) Adding new feature.

(c) $k$-nearest neighbors method.

Figure 12: Not linearly separable data and ways to overcome this problem.

Sometimes, the data points can not be linearly separated, as can be seen in the Figure 12a.

The first way to solve this problem is to make the data separable by adding extra dimensions or changing coordinates, see example of creating a dimension by adding new feature $x_3 = x_1 \cdot x_2$ to the samples in Figure 12b.

Another way would be to use the **k-nearest neighbors method** (KNN).

$$\hat{y}_{\text{new}}(X_{\text{new}}) = \frac{1}{k}\sum_{\mu: X_\mu \in N_k(X_{\text{new}})} y_\mu \tag{5.15}$$

with $N_k(X_{\text{new}})$ the set with the $k$ closest points to $X_{\text{new}}$ from the training set. The choice of $k$ should be made through validation.

The $k$-nearest neighbors method is simple and intuitive, but it relies on our low-dimensional intuition about geometry and distance. However, in high dimensions we face the **curse of dimensionality**: when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. Since no point is close to a sample point anymore, all the points in the dataset are likely almost equidistant to the new point, so that the $k$ closest neighbors are essentially "random". Consequently, their labels are often not indicative of the new point's actual class, making the resulting prediction unreliable.

**Geometry in high dimensions or why high-dimensional watermelons are useless**

Let us imagine a cubic watermelon in dimension $d$ with a side of length $L = 1$, so that the overall volume of a watermelon is also $L^d = 1$. Now, let us assume, that the "crust" of a watermelon (i.e. the part closest to the boundary of the cube) takes up 0.02 of the total side length, so that the "interior" of a watermelon is a sub-cube with a side of length $l = 0.98$.

Now, in $d = 3$ we would have a very nice watermelon, with the volume of edible part $l^d = 0.94$. But in dimension $d = 100$, the volume of interior part becomes $l^d = 0.13$, so most of the volume "concentrates in the crust", leaving us with a very unsatisfying watermelon.

Furthermore, to illustrate how higher dimensionality leads to sparser points, let us consider the following question: if $N$ points are thrown randomly into a sphere of radius one, what is the smallest distance from the origin to one of the random points? We can find the radius $r$ of the sphere with the center in origin, such that the probability that there are no points inside this sphere is equal to $\frac{1}{2}$. The probability of one point to fall further than radius $r$ from the origin is $1 - r^d$, so for $N$ independent points, the probability is

$$(1 - r^d)^N \sim \frac{1}{2}.$$

We can estimate the radius $r$ from the origin to the closest point as

$$r \sim \left(1 - \frac{1}{2}^{1/N}\right)^{1/d}.$$

If we take $N = 500$ points, then

$$\begin{array}{ll} \text{when } d = 2 & r \simeq 0.04, \\ \text{when } d = 100 & r \simeq 0.94. \end{array}$$

In high dimension, the closest to origin point turns out to be almost on the boundary of the sphere.

# 6 Unsupervised learning: Dimensionality reduction

To escape the curse of dimensionality, the data should be transformed to low-dimensional representation. Dimensionality reduction belongs to **unsupervised learning** methods, since there are no labels we wish to predict, only data vectors we want to transform.

In this section, we will focus on one particular method of dimensionality reduction: **Principal Component Analysis** (PCA). The idea of PCA is to apply an orthogonal linear transformation to the data such that the first coordinate in the new space is the direction along which the input data varies the most (first greatest variance), the second coordinate the second greatest variance, etc. This method can be used to reduce the dimensionality of the data in such a way that the data structure is preserved.

## 6.1 Examples

Mapping human genome. Novembre, Johnson, Bryc, *et al.* published a paper about the link between the geographic location of people and their genetic information [3]. They sequence the genome of 3129 people in Europe and encoded this data using 500'568 loci (specific locations on a chromosome) using SNP chips (Single Nucleotide Polymorphism). After cleaning the data (i.e. removing people with uncertain ancestry or mixed/non-European origin) and keeping only the loci with high quality and variability, they end up with 1387 people and 197146 loci. Next, a matrix was constructed where entries were 1 if a person's DNA at a specific loci differed from the population average, and 0 otherwise. By applying the Principal Component Analysis (PCA) on this matrix, they found the following plot:

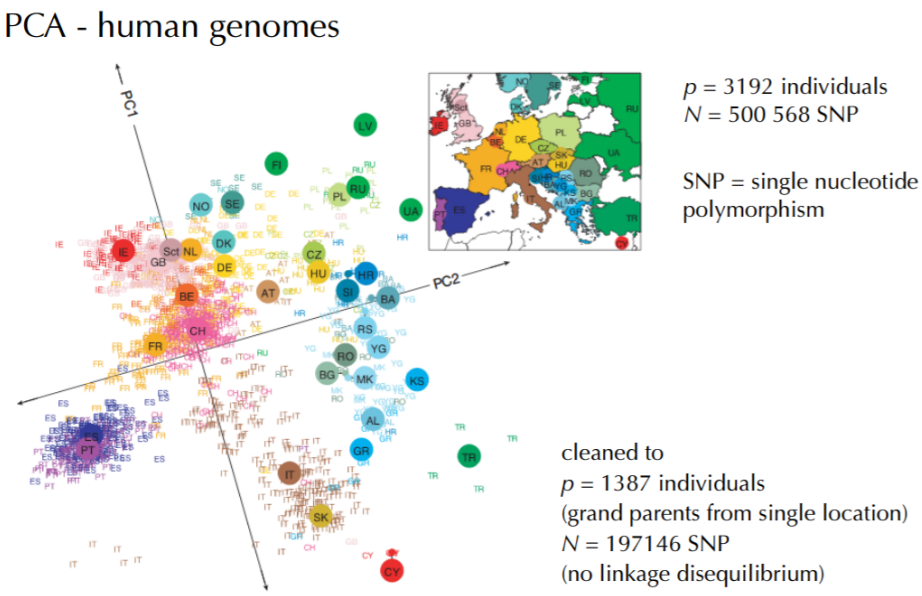



Figure 13: Result of the study [3] showing the correlation between geographic and genetic information.

The two axes of this plot are the two principal components detected trough PCA. The colored circles represent the geographic position of different European states. The individual points, whose position is obtained with respect to the two principal components (PC1,PC2), are colored according to the location of the grandparents of the subject and denoted by the first letters of the country. We can observe that geographical information got encoded into the genetic information and could be uncovered with PCA.

Movie recommendations. In order to recommend movies, companies such as Netflix used to ask their clients to rate the movies they watched and then build a matrix $X$ where the rows represent the users, the columns – the movies available, and the element $X_{ij}$ represents the rating of user $i$ for the film $j$ (from 1 to 5 for example). The matrix $X \in \mathbb{R}^{n\times d}$, where $n$ is the number of users and $d$ the number of movies, will only be partially filled and might look like:

$$X = \begin{array}{|c|c|c|c|c|} \hline & 1 & 5 & 4 & \\ \hline 3 & & 2 & & \\ \hline & 1 & & 4 & \\ \hline & & 5 & 3 & \\ \hline 2 & & 3 & & \\ \hline \end{array} . \tag{6.1}$$

To predict the rating of the movies the user hasn't seen yet, one should fill in the values in the matrix – perform matrix completion. With PCA one can obtain a low-rank approximation of this matrix $X$ and thus predict the unknown ratings.

## 6.2 Singular value decomposition

For any real matrix $X \in \mathbb{R}^{n\times d}$, there exists a factorization of the form

$$X = U\Sigma V^T, \tag{6.2}$$

where

$$\begin{array}{ll} U \in \mathbb{R}^{n\times n} & \text{is orthonormal, } UU^T = \mathbb{I}_n, \\ V \in \mathbb{R}^{d\times d} & \text{is orthonormal, } VV^T = \mathbb{I}_d, \\ \Sigma \in \mathbb{R}^{n\times d} & \text{is diagonal, such that } \sigma_\alpha = \Sigma_{\alpha\alpha} \text{ are real and non-negative.} \end{array}$$

It is called **singular value decomposition** (SVD). Diagonal elements of the matrix $\Sigma$ are called **singular values** and, by convention, are ordered so that $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min\{n,d\}} \geq 0$. Columns of matrix $U$ are called **left singular vectors**, and rows of matrix $V^T$ (columns of matrix $V$) are called **right singular vectors**.

Another way of defining singular values and singular vectors is to say that $\sigma$ is a singular value of $X \in \mathbb{R}^{n\times d}$ iff there exist unit vectors $v \in \mathbb{R}^d, u \in \mathbb{R}^n$, such that

$$\begin{aligned} Xv &= \sigma u, \\ X^T u &= \sigma v. \end{aligned} \tag{6.3}$$

Notice that we can rewrite singular value decomposition in a way similar to the eigen-decomposition, to highlight that the definitions (6.2) and (6.3) are the same:

$$X = \sum_{\alpha=1}^{\min\{n,d\}} \sigma_\alpha u_\alpha v_\alpha^T. \tag{6.4}$$

**How to compute the SVD**

If you know how to compute eigenvalues and eigenvectors of symmetric matrices, then it's easy to find singular values of any given matrix $X$. Notice that left singular vectors $u_\alpha$ of matrix $X$ are actually the eigenvectors of $XX^T$ and right singular vectors $v_\alpha$ of matrix $X$ are actually the eigenvectors of $X^TX$, with corresponding eigenvalues $\lambda_\alpha = \sigma_\alpha^2$:

$$\begin{aligned} X^TXv = X^T(\sigma u) = \sigma(X^T u) = \sigma^2 v, \\ XX^T u = X(\sigma v) = \sigma(Xv) = \sigma^2 u. \end{aligned}$$

Therefore, to find singular values and vectors of a matrix, one can find eigen-decomposition of $XX^T$ or $X^TX$ and take $\sigma_\alpha = \sqrt{\lambda_\alpha}$.

**Low Rank Approximation**

First, we notice that the number of non-zero singular values is equal to the rank of matrix $X$.

Without loss of generality, assume that $d < n$ and let $r$ be the number of non-zero singular values of the matrix $X$. After rotating matrix $X$ by $V$, we get:

$$XV = U\Sigma V^T V = U\Sigma, \tag{6.5}$$

where $\Sigma$ is a diagonal matrix with only $r$ non-zero components, hence the resulting matrix $U\Sigma$ will only have $r$ non-zero columns. As the rotation leaves the original subspace of the data unchanged, this means that the data lies in $r$ dimensional subspace of $\mathbb{R}^d$. In the Figure 14 below, one can see the principle illustrated for a matrix of rank 3.

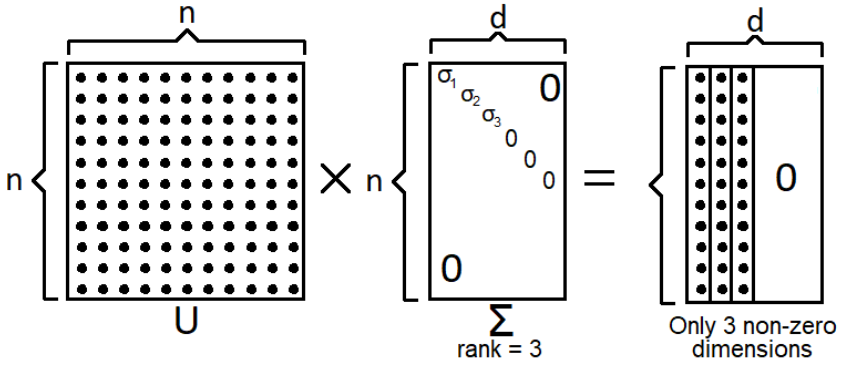



Figure 14: Illustration of the data dimensionality for $X$ of rank 3. After rotation, it is clear that the data is 3D

This idea of zeroing out columns using the SVD factorization properties leads to the method of **low rank approximation** of the matrix $X$, which is stated in the following theorem.

**Theorem 1** (Young-Eckart)**.** *For a real-valued matrix $X \in \mathbb{R}^{n\times d}$ with singular value decomposition $X = U\Sigma V^T$, rank-k approximation $X^{(k)}$ that minimizes the Frobenius norm of the difference*

$$\|X - X^{(k)}\|_F^2 = \sum_{\mu,i} (X_{\mu i} - X_{\mu i}^{(k)})^2 \tag{6.6}$$

*is given by*

$$X^{(k)}_{\mu i} = \sum_{\alpha=1}^{k} U_{\mu\alpha}\sigma_\alpha V_{i\alpha}\,, \tag{6.7}$$

*where the sum goes over the first k largest singular values.*

This theorem states that to best approximate the data with only $k$ dimensions (thus reducing its dimension to $k$) it is best to use the first $k$ singular vector corresponding to the $k$ largest singular values.

To obtain an intuition, why it is true, note that the square of the Frobenius norm of a matrix is equal to the sum of the squares of its singular values. Indeed, the square of the Frobenius norm of $X$ is equal to the trace of $XX^T$:

$$Tr(XX^T) = \sum_{\mu=1}^{n} X^T_\mu X_\mu = \sum_{\mu i} X^2_{\mu i}.$$

Then, using the properties of trace we get

$$Tr(XX^T) = Tr(U\Sigma V^T V\Sigma^T U^T) = Tr(U\Sigma\Sigma^T U^T) = Tr(\Sigma\Sigma^T U^T U) = Tr(\Sigma\Sigma^T) = \sum_{\alpha=1}^{r}\sigma^2.$$

Therefore, to minimize the Frobenius norm of the difference between the matrix and it's approximation we need to "remove" the largest possible singular components.

## 6.3 Low-rank matrix completion

Now, let us return to the movie recommendations example to see how this property of SVD can be used in matrix completion task.

Assume, each user can be represented as a vector $U_\mu \in \mathbb{R}^k$, that describes relevant features for their movies preferences (e.g. some encoding of age, gender, etc.), and each movie can be represented as a vector $V_i \in \mathbb{R}^k$ that somehow evaluates the alignment of the movie with the user's vector representation (e.g. encoding the genre and the language of the movie). Then, we would like to find such matrices $U^{(k)}$ and $\tilde{V}^{(k)}$, that best explain the known ratings. This objective can be formalized as a minimization problem:

$$\min_{U^{(k)}\in\mathbb{R}^{n\times k}, \tilde{V}^{(k)}\in\mathbb{R}^{d\times k}} \sum_{\mu,i}\left(X_{\mu i} - \sum_{\alpha=1}^{k} U^{(k)}_{\mu\alpha}\tilde{V}^{(k)}_{i\alpha}\right)^2, \tag{6.8}$$

where we can recognize the Frobenius norm. Therefore, according to Young-Eckart theorem, taking the first $k$ singular vectors from SVD, should give us the optimal $U^{(k)}$ and $\tilde{V}^{(k)}$.

To perform the SVD, first the blank entries must be filled to complete the matrix $X$. A common pre-processing technique involves filling the blanks with row means (e.g., the average rating given by that specific user), column means (the average rating received by that specific movie), or zeros.

Next, the SVD $X = U\Sigma V^T$ is computed and the required matrices are $U^{(k)} = U_{:,1:k}$ and $\tilde{V}^{(k)} = V_{:,1:k}\Sigma_{1:k,1:k}$. Then the prediction on a new user-movie pair of the ratings is given by

$$\hat{X}_{\mu i} = \sum_{\alpha=1}^{k} U_{\mu\alpha}\sigma_\alpha V_{i\alpha}. \tag{6.9}$$

It is important to note that the optimal $k$ is not necessarily the full rank, as using a lower-rank approximation helps to generalize better than using the matrix filled artificially with means. The choice of the rank $k$ is treated as a hyperparameter. Since recommendation can be considered as a prediction problem, $k$ is determined using validation:

1. a hold-out set (e.g., 10% of the actual observed ratings) is initially kept aside and the hold-out positions are treated as if the ratings are unknown for this positions;
2. he method is run for various values of $K$ (and possibly different data pre-processing variants, like using column versus row means);
3. the results are compared against the hold-out set to measure prediction accuracy;
4. the value of $k$ (and the method variant) that minimizes the validation error is chosen.

## 6.4 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is an SVD-based dimensionality reduction technique often used in data visualization and data preprocessing. It aims at finding a $k$-dimensional representation of the data $X \in \mathbb{R}^{n \times d}$, such that the structure of the data is preserved, in a sense that the directions of the largest variation in the data are selected.

It includes three steps:

1. Center and normalize the raw data $\tilde{X}_{\mu i}$ along each direction $i$ as :

$$X_{\mu i} = \frac{\tilde{X}_{\mu i} - \frac{1}{n}\sum_{\tilde{\mu}=1}^{n} \tilde{X}_{\tilde{\mu} i}}{\sqrt{\frac{1}{n}\sum_{\tilde{\mu}=1}^{n} \tilde{X}_{\tilde{\mu} i}^2 - \left(\frac{1}{n}\sum_{\tilde{\mu}=1}^{n} \tilde{X}_{\tilde{\mu} i}\right)^2}}, \forall i = 1, ..., d \tag{6.10}$$

   This normalization is introduced because the features of the data could have been measured with different units and scales. This centring and normalizing process yields features that are comparable in the sense that they all have no units and the same scale.

2. Do SVD on $X = U\Sigma V^T$ by computing the eigen-decomposition of the covariance $X^T X$. The first $k$ right-singular vectors $v_i \in \mathbb{R}^d$, $i = 1, \ldots, k$ are the principal components.

3. Get the coordinates of the projection on the principal components as:

$$X_\mu^T v_i = \sigma_i U_{\mu i}$$

Applying PCA in the genome mapping example with dimension $k = 2$ allowed us to plot the data in 2 dimensions uncovering the connection between genome variation and geography of Europe. Besides visualization, PCA can be used for:

- Projecting new data points onto the established principal components. In the genome example, this leads to identifying the new person's location;
- Generating typical data from the coordinates $(c_1, \ldots, c_k)$ in low dimension: $X_{\text{new}} = \sum_{i=1}^{k} c_i v_i$. Constructing a "typical genome from any place" by combining the principal components and based on desired coordinates in the low-dimensional space.

## 6.5 How to choose $k$ in PCA?

Since PCA is an unsupervised learning technique and we don't have any labels to perform validation on, to choose the rank $k$ during our exploratory data analysis, we can rely on an heuristic.

If we compute the eigenvalues of $XX^T$ (squared singular values of $X$) and plot them on a histogram, it is often the case that the histogram will have the behavior seen in Figure 15.

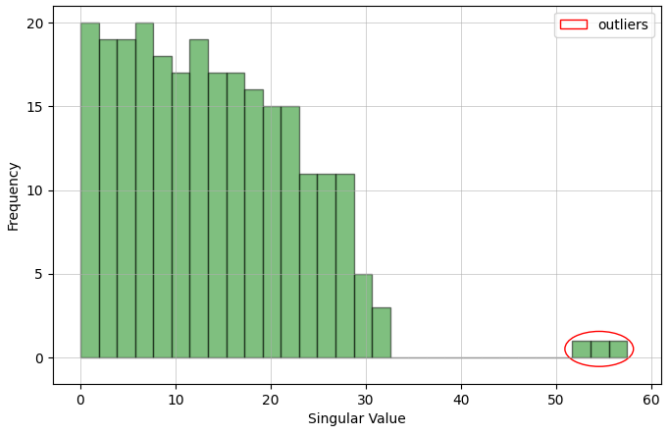



Figure 15: Typical histogram of the eigenvalues $\lambda$ of $XX^T$ (or $X^T X$)

If there are singular values that appear as outliers—meaning they are large and detached from the main bulk of smaller singular values—it is a good heuristic to choose $k$ as the number of these outliers. This method assumes

that the largest, outlier singular values capture the most important structure in the data, while the bulk of small singular values represents noise or less significant variations. In section 7.2.2 we will discuss a case, when this outlier behavior is guaranteed theoretically.

# 7 Inference, Statistical Physics & Monte Carlo Sampling

## 7.1 Spin Glass Card Game

In this chapter we will use a running example inspired by a problem in statistical physics, which we can frame as a simple card game. Imagine a room with $N$ people. Each person is secretly dealt a card with either a +1 or a -1 written on it. This set of $N$ values is the "ground truth" configuration, which we denote as the vector $S^*$, where $S_i^* \in \{\pm 1\}$ represents the value of a card of $i$-th person. We, as observers, do not know this ground truth.

The only data we can observe is a symmetric matrix $Y \in \mathbb{R}^{N\times N}$ constructed as:

$$Y_{ij} = \frac{1}{\sqrt{N}}(S_i^* S_j^*) + W_{ij}, \tag{7.1}$$

where $W_{ij} \sim \mathcal{N}(0, \Delta)$.

The task is to recover the original, hidden card assignments $S^*$ up to a sign change, i.e. split the $N$ people in two groups such that all people with the same sign card are in the same group, using only the noisy observation matrix $Y$. This problem is an example of **clustering**: we want to split the data into separate groups based on some structure in the observations.

## 7.2 The Bayesian Framework for Estimation

Let us apply Bayesian framework that we have already discussed in section 3.2 to this problem. To derive the posterior distribution as in (3.11), we first need to write the prior, the likelihood and the evidence.

**Prior.** Since every spin is equally likely to be +1 or -1, the distribution for the spin vector is given by

$$P_S(S) = \frac{1}{2^N}\prod_{i=1}^{N}(\delta_{S_i,-1} + \delta_{S_i,+1}). \tag{7.2}$$

**Likelihood.** The distribution of the matrix $Y$ elements is determined by the gaussian noise that was added to the signal $\frac{S_i^* * S_j^*}{\sqrt{N}}$:

$$P_{\text{out}}(Y|S) = \prod_{i\le j}\frac{1}{\sqrt{2\pi\Delta}}e^{-\frac{1}{2\Delta}\left(Y_{ij}-\frac{S_iS_j}{\sqrt{N}}\right)^2} = \frac{1}{\sqrt{2\pi\Delta}^{\binom{N}{2}+N}}e^{-\frac{1}{2\Delta}\sum_{i\le j}\left(Y_{ij}^2+\frac{1}{N}\right)}\prod_{i\le j}e^{\frac{1}{\Delta}\left(Y_{ij}\frac{S_iS_j}{\sqrt{N}}\right)}, \tag{7.3}$$

where the last equation is obtained by writing the square of the sum explicitly and taking advantage of the fact that $(S_iS_j)^2 = 1$ for all possible values of $S_i, S_j \in \{\pm 1\}$.

**Evidence.** This quantity $\tilde{Z}$ normalizes the posterior distribution and is given by:

$$\tilde{Z} = P(Y) = \sum_{S_1,\dots,S_N} P(Y,S) = \sum_{S_1,\dots,S_N} P_{\text{out}}(Y|S)P_S(S). \tag{7.4}$$

Notice, how we can factor out the normalization terms that do not depend on $S$ from prior and likelihood, so that

$$\tilde{Z} = \left(\frac{1}{2^N}\frac{1}{\sqrt{2\pi\Delta}^{\binom{N}{2}+N}}e^{-\frac{1}{2\Delta}\sum_{i\le j}\left(Y_{ij}^2+\frac{1}{N}\right)}\right)Z, \tag{7.5}$$

where $Z$ is the sum of the terms that depend on $S$ over all possible configurations.

**Posterior.** Plugging those equations into the formula for posterior distribution (3.11), we get the posterior:

$$\underbrace{P(S|Y)}_{\text{Posterior}} = \underbrace{\frac{1}{Z}}_{\text{Evidence}}\underbrace{\prod_{i=1}^{N}(\delta_{S_i,1}+\delta_{S_i,-1})}_{\text{Prior}}\underbrace{\prod_{i<j}e^{\frac{1}{\Delta\sqrt{N}}S_iS_jY_{ij}}}_{\text{Likelihood}}. \tag{7.6}$$

### 7.2.1 Connection to Statistical Physics

Recalling the formalism of statistical physics, we see that the posterior probability distribution takes the form of a Boltzmann distribution. The normalization $Z$ takes the place of the partition function. The noise variance $\Delta$ plays the role of the temperature and the corresponding Hamiltonian is:

$$\mathscr{H}(S) = -\frac{1}{\sqrt{N}} \sum_{i<j} Y_{ij} S_i S_j \tag{7.7}$$

which reminds us of the **Ising model**. Note that the interactions $Y_{ij}$ are different for every pair of spins. An Ising model with such interactions is called a spin glass.

The Table 1 summarizes the Bayesian inference and Statistical Physics correspondence. This analogy will help us interpret and understand some of the properties and behavior of statistical inference and machine learning problems as we can view them through the lens of statistical physics.

| **Statistical Physics** | **Inference** |
|---|---|
| Ising spins $S_i$ | Prior $S_i$ |
| Boltzmann measure $\frac{1}{Z} e^{\mathscr{H}}$ | posterior $P(S\|Y)$ |
| Hamiltonian $\mathscr{H}$ | Negative log-likelihood |
| Partition function $Z$ | Evidence $Z$ |
| Temperature $T$ (or $\beta^{-1}$) | Noise variance $\Delta$ |
| Interactions between spins $J_{ij}$ | Observations $Y_{ij}$ |

Table 1: Analogy between Statistical Physics and Inference when looking at the spin glass card game.

### 7.2.2 Estimators

**Maximum Likelihood Estimator.** Maximum likelihood estimator is a vector $S$ that maximizes probability $P_{\text{out}}(Y|S)$ given by (7.3) or minimizes negative log-likelihood:

$$\hat{S}^{\text{ML}} = \arg\min_{S \in \mathbb{R}^N} \sum_{i \le j} \left( \frac{S_i S_j}{\sqrt{N}} - Y_{ij} \right)^2 = \arg\min_{S \in \mathbb{R}^N} \sum_{i,j} \left( \frac{S_i S_j}{\sqrt{N}} - Y_{ij} \right)^2 . \tag{7.8}$$

Here, we used the fact, that the values on the diagonal are constant w.r.t. $S$ and that summing over all $i, j$ is the same as multiplying the sum over $i \le j$ by 2 and subtracting the sum of the diagonal values.

This problem is equivalent to finding the rank-one approximation of the matrix $Y$ that minimizes the Frobenius norm of the difference, which, according to Young-Eckart theorem (see theorem 1 in the previous section), is given by the first principal component (i.e. the leading singular vector). Therefore, the MLE for this problem can be found by applying PCA to the matrix $Y$.

Notice, that to find this estimator we ignored the prior constraint that $S_i$ must be $\pm 1$. The final estimate is obtained by taking the sign of the elements in the leading singular vector. The primary drawback of the MLE is that it discards the information that the variables are binary, which can lead to suboptimal results.

**Maximum A Posteriori Estimator.** The MAP estimator improves on the MLE by maximizing the full posterior probability $P(S|Y)$ given by (7.6).This incorporates the prior constraint that the spins must be $\pm 1$. Maximizing the posterior probability or minimizing negative logarithm of the posterior is equivalent to minimizing the Hamiltonian (7.7):

$$\hat{S}^{\text{MAP}} = \arg\min_{S \in \{\pm 1\}^N} -\frac{1}{\sqrt{N}} \sum_{i<j} Y_{ij} S_i S_j . \tag{7.9}$$

Therefore, the MAP estimator corresponds to finding the ground state (the lowest energy configuration) of the equivalent spin glass system.

The main challenge of the MAP approach is its computational complexity, which makes finding the exact minimum intractable for large systems.

**Bayes Optimal Estimators.** Rather than maximizing a probability, these estimators are designed to be "optimal" by directly minimizing a specific loss function that measures the error between the estimate and the true value, while given only the observations $Y$ and the posterior distribution $S^* \sim P(S|Y)$.

- The **Minimum Mean Square Error** (MMSE) estimator seeks to minimize the mean square error (MSE) between the "ground truth" spins $S^*$ and the estimations $\hat{S}$:

$$\mathscr{L}_{\text{MSE}}(S^*, \hat{S}) = \frac{1}{N}\sum_{j=1}^{N}(S_j^* - \hat{S}_j)^2. \tag{7.10}$$

  However, since the true values of the spins are unknown, we only can evaluate the expected MSE over the posterior distribution $S^* \sim P(S|Y)$:

$$\hat{S}^{\text{MSE}}(Y) = \arg\min_{\hat{S}} \mathbb{E}_{P(S|Y)}\mathscr{L}_{\text{MSE}}(S, \hat{S}) = \arg\min_{\hat{S}} \sum_{\{S_i\}_{i=1}^N} P(S|Y)\frac{1}{N}\sum_{j=1}^{N}(S_j - \hat{S}_j)^2. \tag{7.11}$$

  To find the minimum, we set the partial derivative w.r.t. $\hat{S}_j$ to 0:

$$\begin{aligned} &\frac{\partial \mathbb{E}_{P(S|Y)}\mathscr{L}_{\text{MSE}}(S, \hat{S})}{\partial \hat{S}_j} = \frac{2}{N}\sum_{\{S_i\}_{i=1}^N} P(S|Y)(S_j - \hat{S}_j(Y)) = 0 \\ \Rightarrow\ &\hat{S}_j^{\text{MSE}}(Y)\underbrace{\left(\sum_{\{S_i\}_{i=1}^N} P(S|Y)\right)}_{=1} = \sum_{\{S_i\}_{i=1}^N} P(S|Y)S_j = \mathbb{E}_{P(S|Y)}S_j \end{aligned} \tag{7.12}$$

  This posterior expectation is precisely the definition of the local magnetization $m_j$ of spin $j$ in the corresponding Ising model, calculated at a finite "temperature" equal to the noise variance $\Delta$. Notice, that this estimator again produces values not necesseraly equal to $\pm 1$.

  The MAP estimator, by seeking the single lowest-energy state, is equivalent to a zero-temperature analysis. At the same time, the MMSE estimator matches the "temperature" of the analysis to the level of noise in the observed data.

- The **Maximum Overlap** (MO) estimator aims to find a $\pm 1$ valued estimate that maximizes the number of correctly identified spins (i.e., maximizes the overlap with the true configuration):

$$\hat{S}^{\text{MO}}(Y) = \arg\max_{\hat{S}\in\{\pm 1\}^N} \mathbb{E}_{P(S|Y)}\sum_{j=1}^{N}\delta_{S_j,\hat{S}_j}. \tag{7.13}$$

  The resulting estimator is the sign of the local magnetization:

$$\hat{S}_i^{\text{MO}} = \text{sign}(m_i). \tag{7.14}$$

Both of these Bayes optimal estimators can be expressed through the more general concept of the marginal distribution:

$$\mu(S_j) = \sum_{\{S_i\}_{i\neq j}} P(S|Y), \tag{7.15}$$

which gives the probability distribution of $S_j$ alone. Then we can state the optimal estimators in a more general form:

- the MMSE estimator: $\hat{S}_j^{\text{MSE}} = \mathbb{E}_{\mu_j} S_j$;
- the MO estimator: $\hat{S}_j^{\text{MO}} = \arg\max_{S_j} \mu_j(S_j)$.

The optimal estimators do not depend on finding a single "best" configuration (like the ground state). Instead, they require computing averages over the entire ensemble of possible configurations, weighted by their posterior probability. This pose a severe computational challenge. Calculating the required averages (the magnetizations) requires summing over all $2^N$ possible configurations This computational bottleneck forces us to seek approximation methods, that will be discussed in the next section.

## 7.3 Monte Carlo Markov Chains Sampling

Since the direct computation of posterior averages is impossible for large $N$, we must turn to approximation methods. Monte Carlo Markov Chains (MCMC) are a class of algorithms for sampling from complex, high-dimensional probability distributions like our posterior.

MCMC is used to generate a sequence of configurations $(S^1, S^2, \ldots)$ that are effectively random draws from the target probability distribution. Once we have a sufficiently large collection of these samples, any desired average (like the local magnetization $m_i$) can be accurately approximated by calculating the average of that quantity over our collected samples.

### 7.3.1 The Metropolis-Hastings Algorithm

We will now introduce an example of the Metropolis-Hastings algorithm for the Spin Glass Card Game (SGCG). The Metropolis-Hastings algorithms are a class of MCMC algorithms for generating a sequence of values from a distribution $P$, which are especially convenient, when we know some function $f$ proportional to the density $P$, without the need to compute the normalization factor $Z = \sum_S f(S)$ for $P(S) = \frac{1}{Z} f(S)$. For example, in case of the posterior distribution in the SGCG: $P(S) = \frac{1}{Z} e^{-\frac{1}{\Delta}\mathscr{H}(S)}$, it is enough to know the Hamiltonian and compute $f(S) = e^{-\frac{1}{\Delta}\mathscr{H}(S)}$.

Before giving the general algorithm, let us discuss the steps of the algorithm in the SGCG example, when we are given $Y$, $\Delta$, and $N$:

1. Start from a random configurations $S \in \{\pm 1\}^N$;
2. Pick an element $j$ uniformly from $\{1, \ldots, N\}$, and compute the local field $h_j = \frac{1}{\sqrt{N}} \sum_{k \neq j} Y_{kj} S_k$:
   - if $h_j S_j \geq 0$, i.e. the considered spin aligns with the local field, then flip the spin $S_j \to -S_j$ with probability $\sim e^{-\frac{2h_j S_j}{\Delta}}$,
   - if $h_j S_j < 0$, then flip the spin $S_j \to -S_j$ with probability 1;
3. Repeat the previous step $T$ times.

This algorithm has intuitive physical interpretation. When the candidate new state with the flipped spin has lower energy (when the spin doesn't align with local magnetization), we certainly flip it. But when current configuration has lower energy that the candidate new configuration, we flip the spin with probability proportional to the ratio of Boltzmann probabilities of these two states, i.e. the lower the energy difference, the higher the probability to flip the spin.

After running this algorithm, we are left with the sequence of states $(S^t)_{t=1}^T$. We will only consider samples from this sequence after the system achieved stability, in the sense that the distribution of the states is not changing from one timestep to the next one. We achieve that by setting some time $T_{\text{eq}}$ and only taking samples from $t > T_{\text{eq}}$. And to obtain "almost" independent samples from the posterior, we are going to take remote enough samples by setting some $T_{\text{decorr}}$ and only taking samples from every $T_{\text{decorr}}$ steps.

After selecting the samples, we can use them to compute the magnetization of each spin $j$ and obtain a Bayes optimal estimator.

While the algorithm is clear, determining the equilibration and decorrelation times $T_{\text{eq}}$ and $T_{\text{decorr}}$ remains a significant practical challenge. In general, these times can be exponential in the dimension $N$, but in practice they can be much smaller. Finding them is often a matter of trial and error.

### 7.3.2 The mathematical foundations of MCMC

**Markov chain** is a stochastic process describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. A discrete-time Markov chain is a sequence of random variables $(S^1, S^2, \ldots, S^t, \ldots)$ from a countable alphabet $S^t \in \mathscr{A}$ satisfying the **Markov property**: the probability of transitioning to the next state, $S^{t+1}$, depends only on the current state $S^t$, and not on any prior history. Therefore, a Markov chain is fully described by its initial state distribution $P^0(S)$ and transition matrix $T$, which contains the probabilities $T(S'|S)$ of moving from any state $S$ to any other state $S'$ in a single step:

$$T(S'|S) = P(S^{t+1} = S' | S^t = S). \tag{7.16}$$

Consider that at time $t$ we have the probability distribution over the states $P^t(S)$, then at time $t+1$ we can compute the probability distribution $P^{t+1}$ using the following relation:

$$P^{t+1}(S') = \sum_{S \in \mathscr{A}} T(S'|S) P^t(S). \tag{7.17}$$

Notice, that it can be rewritten in the matrix form, if we represent the probability distribution at timestep $t$ in the form of a vector $P^t \in [0,1]^{|\mathscr{A}|}$, such that $P^t_j = P^t(S=j)$, and transition matrix $T$, such that $T_{ij} = T(S'=i|S=j)$:

$$P^{t+1} = TP^t. \tag{7.18}$$

An attentive reader might notice, that such multiplication of the vector $P^t$ by the same matrix $T$ over and over is similar to the **power iteration** algorithm for finding the leading eigenvector of the matrix $T$. Therefore, under some assumptions on the initial distribution and the transition matrix, it is going to converge to the equilibrium.

**Equilibrium distribution** (or stationary distribution) $P_{\text{eq}}(S)$ is a special probability distribution that remains unchanged when the transition matrix T is applied to it:

$$P_{\text{eq}}(S') = \sum_{S \in \mathscr{A}} T(S'|S) P_{\text{eq}}(S). \tag{7.19}$$

If the chain is irreducible (there is non-zero probability to get to all the possible states of the chain from some initial state after some number of steps) and aperiodic (when starting from some state, we can not be certain that we return to this state after some fixed number of steps), the state probability distribution of a Markov chain converges to the stationary distribution as time goes to infinity. One very widely used sufficient condition for convergence is called the *Detailed Balance Condition*.

The transition matrix $T$ satisfies the Detailed Balance Condition with respect to the distribution $\tilde{P}$ if:

$$\tilde{P}(S)T(S'|S) = \tilde{P}(S')T(S|S'). \tag{7.20}$$

Notice, that this condition can also be used to find the transition probability matrix for a given equilibrium distribution.

Now, we can describe general **Metropolis-Hastings algorithm**, which is based on the fact, that after the Markov chain converged to the equilibrium, each state of the chain is effectively sampled from that distribution.

Given a current configuration $S$ and a target distribution $P_{\text{eq}}$, the algorithm proceeds as follows (the pseudo-code is given in the listing 3):

1. Propose a "test" next configuration $S'$;
2. The test configuration is accepted with probability $p = \min(1, \frac{P_{\text{eq}}(S')}{P_{\text{eq}}(S)})$

**Algorithm 3** Metropolis-Hastings algorithm

**Require:** $P_{\text{eq}}$, $S_0$, $T$
  $S^{(0)} \leftarrow S_0$
  $t \leftarrow 0$
  **repeat**
    $t \leftarrow t+1$
    $S' \sim \text{Uniform}(\mathscr{A})$
    **if** $P_{\text{eq}}(S') > P_{\text{eq}}(S^{t-1})$ **then**
      $S^t \leftarrow S'$
    **else**
      $S^t \leftarrow \begin{cases} S' \text{ with probability } \frac{P_{\text{eq}}(S')}{P_{\text{eq}}(S^{t-1})} \\ S^{t-1} \text{ with probability } 1 - \frac{P_{\text{eq}}(S')}{P_{\text{eq}}(S^{t-1})} \end{cases}$
    **end if**
  **until** $t > T$

This acceptance rule is constructed specifically to satisfy the detailed balance condition. To verify this, without loss of generality assume $P_{\text{eq}}(S') > P_{\text{eq}}(S)$. Per the algorithm's rule, the transition probability $T(s'|s) = 1$, while the reverse transition probability is $T(S|S') = P_{\text{eq}}(S)/P_{\text{eq}}(S')$. We now check if the detailed balance equation 7.20 holds.

- The left-hand side is $P_{\text{eq}}(S) \cdot 1 = P_{\text{eq}}(S)$
- The right-hand side is $P_{\text{eq}}(S') \cdot \frac{P_{\text{eq}}(S)}{P_{\text{eq}}(S')} = P_{\text{eq}}(S)$

Since both sides are equal, the condition is satisfied.

**Reduction to the Spin Glass Algorithm.** If we set the "target" distribution to be the posterior of the SGCG model, the ratio $\frac{P_{\text{eq}}(S')}{P_{\text{eq}}(S)}$ simplifies:

$$\frac{P_{\text{eq}}(S')}{P_{\text{eq}}(S)} = \exp\left[\frac{1}{\Delta}(\mathscr{H}(S) - \mathscr{H}(S'))\right] = \exp\left[-\frac{2h_j S_j}{\Delta}\right], \tag{7.21}$$

and we obtain exactly the update probability we saw in the example earlier.

### 7.3.3 Gibbs Sampling

The Gibbs sampling algorithm, or heat bath in statistical physics, is another common example of a Monte Carlo Markov Chain algorithm. It is used for sampling from a specified multivariate probability distribution when direct sampling from the joint distribution is difficult, but sampling from the conditional distribution is more practical. It is particularly useful in high-dimensional configurations setting, when variables are continuous, common in machine learning.

Given the multivariate configuration $S \in \mathscr{A}^N$ where $\mathscr{A}$ can be both a set of discreet or continuous variables, the iteration of the algorithm proceeds as follows:

1. Select one component $k \in \{1, \dots, N\}$ at random;
2. Sample $S_k$ from the conditional distribution $P(S_k|\{S_i\}_{i\neq k})$, when all the components except from $k$ are fixed to the current values.

In contrast to the Metropolis-Hastings algorithm, Gibbs Sampling directly draws a new value for the selected variable from its conditional distribution, completely forgetting the variable's previous state.

This procedure is easier to implement than sampling from a full, high-dimensional distribution. For a single scalar variable, if the conditional probability density is known, one can compute the cumulative distribution function (CDF), draw a random number uniformly from $[0, 1]$, and find the corresponding variable value by inverting the CDF, which is much harder to do in high dimension.

In the SGCG example, the conditional probability of a single spin can be written as:

$$\begin{aligned} P(S_k|\{S_i\}_{i\neq k}) &= \frac{\frac{1}{Z}\exp\left(\frac{1}{\Delta\sqrt{N}}\sum_{i,j\neq k} Y_{ij}S_iS_j\right)\exp\left(\frac{1}{\Delta\sqrt{N}}S_k\sum_{i\neq k} Y_{ik}S_i\right)}{\frac{1}{Z}\exp\left(\frac{1}{\Delta\sqrt{N}}\sum_{i,j\neq k} Y_{ij}S_iS_j\right)\sum_{S_k=\pm 1}\exp\left(\frac{1}{\Delta\sqrt{N}}S_k\sum_{i\neq k} Y_{ik}S_i\right)} \\ &= \frac{e^{\frac{1}{\Delta}h_kS_k}}{e^{\frac{1}{\Delta}h_kS_k} + e^{-\frac{1}{\Delta}h_kS_k}}, \end{aligned} \tag{7.22}$$

where we used the definition of a local field that we have seen before $h_j = \frac{1}{\sqrt{N}}\sum_{k\neq j} Y_{kj}S_k$. It can be proven that Gibbs sampling transition matrix satisfies the detailed balance condition, and therefore, the Markov chain generated with this algorithm converges to the stationary distribution that is the posterior of the SGCG.

### 7.3.4 Simulated Annealing

Markov Chain Monte Carlo methods are designed to sample from a probability distribution, a task required for the Bayes-optimal estimator. However, if we want to find a minimum of a loss function, which is the objective of the MAP estimator, we should use a different technique called Simulated Annealing.

Simulated Annealing provides a generic optimization framework inspired by the physical process of annealing metals.

The loss function $\mathscr{L}(S)$ is transformed into a Boltzmann-like probability distribution

$$P_\beta(S) = \frac{1}{Z}e^{-\beta\mathscr{L}(S)}.$$

Here, the loss function $\mathscr{L}(S)$ plays the role of energy, and $\beta$ is an inverse temperature parameter.

Then some MCMC algorithm is used to draw samples from this probability distribution at a given inverse temperature $\beta$. The algorithm starts at a very high temperature or small $\beta$ (notice that $\beta = 0$ produces a uniform distribution) and gradually increases $\beta$ towards infinity (driving the temperature T to zero).

As $\beta$ approaches infinity, the probability distribution $P_\beta$ becomes sharply peaked over the states $S^*$ that minimize the loss function $\mathscr{L}(S)$. At zero temperature, the equilibrium state of the system is its lowest-energy configuration. The algorithm thereby finds the optimal solution.

A key advantage of Simulated Annealing is its versatility. Unlike gradient descent, which requires continuous variables to compute derivatives, Simulated Annealing can handle both discrete and continuous variables. This makes it a highly general optimization tool[13].

## 7.4 Langevin Sampling

Another popular sampling method for the probability distributions of continuous random variables is by using the properties of Langevin dynamics equation. Given a distribution $P(w)$ of a continuous $w \in \mathbb{R}^d$, let us define a "potential" $U(w) = -\log P(w) + \text{const}$. The distribution can be represented as $P(w) = \frac{e^{-U(w)}}{Z}$, where $Z$ is normalization constant.

We can write the overdamped Langevin equation:

$$\frac{dw}{dt} = -\nabla_w U(w(t)) + \sqrt{2}\xi(t), \tag{7.23}$$

where $\xi(t)$ is a standard Brownian motion. It describes a stochastic process, where $w(t)$ at a fixed point $t$ can be viewed as a random variable that we want to sample. It can be proven that the stationary distribution of the random variable $w(t)$ as $t \to \infty$ is exactly $P(w) = \frac{e^{-U(w)}}{Z}$. We want to use this fact to design an algorithm to sample from $P(w)$.

In order to do that, we discretize the dynamics using some small step size $\eta$:

$$w^{(t+1)} = w^{(t)} - \eta \nabla_w U(w^{(t)}) - \sqrt{2\eta}\varepsilon_t, \tag{7.24}$$

where $\varepsilon_t \sim \mathcal{N}(0, 1)$ is gaussian noise and we took $\sqrt{2\eta}$ step to ensure correct scaling of the noise. This way, if $\eta$ goes to 0 we retrieve exactly the differential equation 7.23. One can notice the similarity between the equation 7.24 and the gradient descent step with learning rate $\eta$ to minimize the potential $U(w)$. Here, as before, there is a trade-off: when parameter $\eta$ is very small, the algorithm takes long time to converge, but when $\eta$ is too big, we risk to not converge to the desired stationary distribution.

We have already encountered a variant of gradient descent "with noise" earlier in the course, when we discussed stochastic gradient descent (SGD) algorithm. There, the noise comes from the stochastic procedure of choosing mini-batches to estimate the full gradient. It is not exactly the same as adding gaussian noise and currently it is actively discussed in the field of theoretical machine learning, how SGD and Langevin dynamics are related. It is suggested that one of the benefits of SGD, i.e. adding noise to the GD dynamics, is that this noise can help escape "bad" local minima of energy similarly to the thermal noise when sampling using Langevin equation.

## 7.5 Learning Hyperparameters via Bayesian Inference

Consider a slightly different Spin glass card game: in the deck of cards, we only have fraction $\rho$ of cards with $\pm 1$ and fraction $1 - \rho$ of cards with value 0, so that the prior becomes:

$$P_S(S_i) = \frac{\rho}{2}(\delta_{S_i,-1} + \delta_{S_i,+1}) + (1 - \rho)\delta_{S_i,0}. \tag{7.25}$$

Now, the "signal" part of $Y_{ij}$, i.e. $S_i S_j$ will only be non-zero when both spins are non-zero. It is unclear, how to estimate the value of $\rho$ directly from such data.

Bayesian inference provides a principled framework for learning such hyperparameters. The core idea is to estimate the hyperparameter by calculating its probability given the observed data. For this example, we want to find $P(\rho|Y)$:

$$P(\rho|Y) = \frac{P(Y|\rho)P(\rho)}{P(Y)} = \frac{P(\rho)}{P(Y)} \sum_{\{S_i\}_{i=1}^N} P(Y, \{S_i\}|\rho) = \frac{P(\rho)}{P(Y)} \sum_{\{S_i\}_{i=1}^N} P(Y|\{S_i\}, \rho)P(\{S_i\}|\rho), \tag{7.26}$$

[13] Quoting one of its inventors: "Simulated annealing is always the second best optimization algorithm for any problem."

where we have used Bayes theorem (3.10) and the law of total probability. If we assume that we know the model $Y_{ij} = \frac{S_i^* S_j^*}{\sqrt{N}} + W_{ij}$, where $W_{ij}$ is a symmetric matrix with noise sampled from a normal distribution with variance $\Delta$, we can write:

$$P(\rho|Y) = \frac{P(\rho)}{P(Y)} \underbrace{\sum_{\{S_i\}_{i=1}^N} \prod_{i<j} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(Y_{ij} - \frac{S_i S_j}{\sqrt{N}})^2} \prod_{i=1}^N [\frac{\rho}{2}(\delta_{S_i,-1} + \delta_{S_i,+1}) + (1-\rho)\delta_{S_i,0}]}_{\text{evidence } Z(\rho)} \tag{7.27}$$

The term that we denote $Z(\rho)$ corresponds to the evidence or normalization in the standard Bayesian inference problem. This $Z(\rho)$ will be exponential in $N$, but $P(\rho)$ will be of order 1. Thus, we can disregard $P(\rho)$ for large $N$ and optimize directly to find $\rho$ that maximizes $Z(\rho)$.

Making a comparison with physics, we can define the "free energy" $f(\rho) = -\frac{1}{N}\log(Z(\rho))$. The maximum likelihood estimator will be the maximum of the evidence over $\rho$, or, equivalently, the minimum of the free energy $f(\rho)$. In particular, this will be a good estimator if $N$ is large. This method can be used in general to learn hyperparameters.

### 7.5.1 Expectation Maximization algorithm

To find the optimal value of $\rho$ let us start by setting the derivative of the free energy $f(\rho)$ to zero:

$$\begin{aligned}
0 = \frac{\partial \log Z(\rho)}{\partial \rho} = \\
&\frac{1}{Z(\rho)} \sum_{\{S_i\}_{i=1}^N} \prod_{i<j} e^{-\frac{1}{2\Delta}\left(Y_{ij} - \frac{S_i S_j}{\sqrt{N}}\right)^2} \sum_{j=1}^N \left[\frac{1}{2}\delta_{S_j,-1} + \frac{1}{2}\delta_{S_j,+1} - \delta_{S_j,0}\right] \\
&\times \prod_{i\neq j} [\frac{\rho}{2}\delta_{S_i,-1} + \frac{\rho}{2}\delta_{S_i,+1} + (1-\rho)\delta_{S_i,0}] = \\
&\sum_{\{S_i\}_{i=1}^N} \frac{1}{Z(\rho)} \prod_{i<j} e^{-\frac{1}{2\Delta}\left(Y_{ij} - \frac{S_i S_j}{\sqrt{N}}\right)^2} \times \prod_{i=1}^N [\frac{\rho}{2}\delta_{S_i,-1} + \frac{\rho}{2}\delta_{S_i,+1} + (1-\rho)\delta_{S_i,0}] \\
&\times \sum_{j=1}^N \frac{\frac{1}{2}\delta_{S_j,-1} + \frac{1}{2}\delta_{S_j,+1} - \delta_{S_j,0}}{\frac{\rho}{2}\delta_{S_j,-1} + \frac{\rho}{2}\delta_{S_j,+1} + (1-\rho)\delta_{S_j,0}} = \\
&\sum_{\{S_i\}_{i=1}^N} P(S|Y) \times \sum_{j=1}^N \frac{\frac{1}{2}\delta_{S_j,-1} + \frac{1}{2}\delta_{S_j,+1} - \delta_{S_j,0}}{\frac{\rho}{2}\delta_{S_j,-1} + \frac{\rho}{2}\delta_{S_j,+1} + (1-\rho)\delta_{S_j,0}},
\end{aligned} \tag{7.28}$$

Where we recognized the posterior distribution $P(S|Y)$ for given $\rho$. Recalling the definition of the marginal distribution (7.15) and changing the order od summation to $\sum_{j=1}^N \sum_{S_j} \sum_{\{S_i\}_{i\neq j}}$ we get:

$$0 = \sum_{j=1}^N \sum_{S_j} \mu(S_j) \frac{\frac{1}{2}\delta(S_j-1) + \frac{1}{2}\delta(S_j+1) - \delta(S_j)}{\frac{\rho}{2}\delta(S_i-1) + \frac{\rho}{2}\delta(S_i+1) + (1-\rho)\delta(S_i)} = \sum_{j=1}^N \left[\mu(S_j=+1)\frac{1}{\rho} + \mu(S_j=-1)\frac{1}{\rho} + \mu(S_j=0)\frac{-1}{1-\rho}\right]. \tag{7.29}$$

Now, using the fact that $\mu(S_i=+1) + \mu(S_i=-1) + \mu(S_i=0) = 1$:

$$\sum_{j=1}^N \left[\mu(S_j=+1)\frac{1}{\rho} + \mu(S_j=-1)\frac{1}{\rho} + (1-\mu(S_i=+1)-\mu(S_i=-1))\frac{-1}{1-\rho}\right] = 0. \tag{7.30}$$

By rearranging the terms, it is easy to verify that the above condition corresponds to:

$$\rho = \frac{1}{N}\sum_{j=1}^N \left(\mu(S_j=1) + \mu(S_j=-1)\right). \tag{7.31}$$

So $\rho$ can be estimated just as the average of the sum of the marginals with $S_i = \pm 1$, i.e. the expected number of spins not equal to 0.

However, we actually need $\rho$ to calculate marginals $\mu(S_i)$. We could solve this recurrence issue by iteratively updating our estimate of $\rho$:

$$\rho^{t+1} = \frac{1}{N}\sum_{j=1}^{N}\left(\mu(S_j = 1|\rho^{(t)}) + \mu(S_j = -1|\rho^{(t)})\right). \tag{7.32}$$

To obtain the marginals, we can use the current estimate of the hyperparameter $\rho^t$ and run a Monte Carlo sampling.

The algorithm iterates enforcing the consistency condition until it converges. This framework allows us to learn hyperparameters directly from data. However, there is a significant caveat. The Bayesian framework is mathematically beautiful and principled, but it relies on a core assumption: that we know the correct generative model for the data. In practice, the model we postulate is never exactly right. This is the fundamental bottleneck of all model-based inference methods and we should keep it in mind when we apply them.

# 8 Unsupervised learning: Clustering

General goal of clustering is to partition a set of unlabeled data points into distinct groups, or clusters. The guiding principle is that points within a given cluster should be more similar to each other than to points in other clusters. Spin glass card game that we discussed in the previous chapter is an example of this unsupervised learning problem.

Formally, the problem can be formulated as follows. We are given a set of $n$ samples in dimension $d$ $X \in \mathbb{R}^{n\times d}$, and we want to assign each point to one of $k$ clusters $\{C_1, \ldots, C_k\}$. The number of clusters $k$ is normally treated as another hyperparameter, but for the sake of simplicity we will discuss the case of fixed given $k$. We will use one-hot encodings $v_\mu \in \{0,1\}^k$ of the cluster labels:

$$v_{\mu,a} = \begin{cases} 1, \text{ if } \mu \in C_a \\ 0, \text{ otherwise} \end{cases}$$

and denote the centers of the clusters $u_a \in \mathbb{R}^d$:

$$u_a = \frac{1}{|C_a|}\sum_{\mu\in C_a} X_\mu. \tag{8.1}$$

### 8.0.1 $k$-Means Algorithm

One way of approaching clustering problem is to define a loss function to evaluate how good is the clustering and minimize this loss function. For example, we can measure, how close (in terms of the Euclidean norm) the points of the cluster are to the center of this cluster:

$$\mathscr{L}(\{C_1, \ldots, C_k\}) = \sum_{a=1}^{k}\sum_{\mu\in C_a}\|X_\mu - u_a\|_2^2 = \sum_{\mu=1}^{d}\sum_{i=1}^{d}(X_{\mu i} - \sum_{a=1}^{k} u_{ai}v_{\mu a})^2. \tag{8.2}$$

Now, to minimize this loss, we can use the $k$-Means algorithm:

1. Initialization: Randomly select $k$ data points from the dataset to serve as the initial cluster centers;
2. Assignment Step: Assign each data point to the cluster whose center is nearest to it;
3. Update Step: After all points are assigned, recalculate the centers for each cluster by taking the mean of all the points newly assigned to it;
4. Repetition: Repeat the Assignment and Update steps iteratively until a stopping criterion is met (e.g., the cluster assignments no longer change)

The algorithm steps results are illustrated in Figure 16.

Convergence is in general not guaranteed and in case the algorithm converges the result is not necessarily a clustering that minimizes $\mathscr{L}(S)$. The reason for this is that minimizing the loss $\mathscr{L}(S)$ is computationally hard. However, the $k$-Means algorithm is a simple heuristic one that often works well. Another thing to notice is that the algorithm depends on the choice of cluster centers for initialization. It can be seen, for instance, if we select the first cluster center as one of the data points and the other centers very far away from the actual data. Then the algorithm will not find a good clustering, it would suggest, that all the points belong to the first cluster, while all other clusters will be empty. Therefore, one should choose the appropriate initialization.

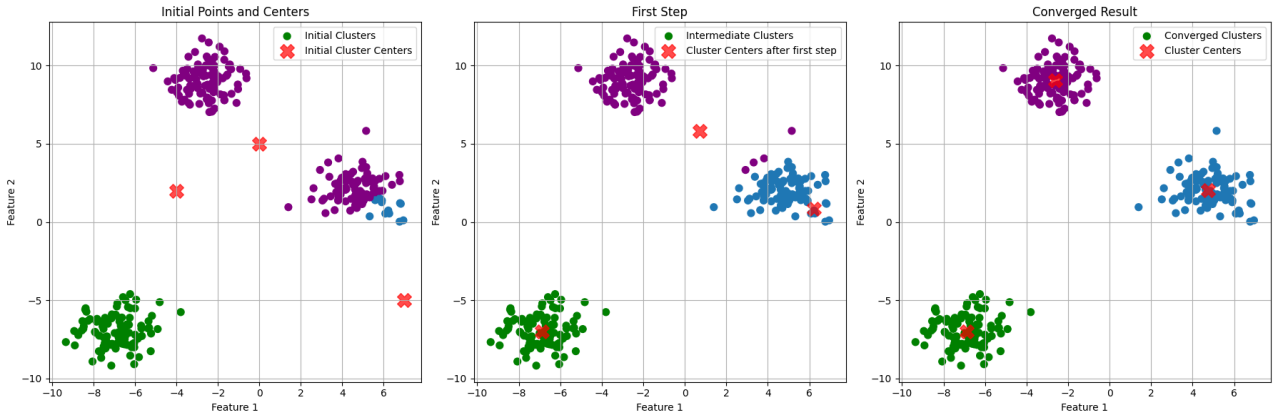



Figure 16: $k$-Means algorithm steps in clustering problem with $d = 2$ and $k = 3$. The different colors refer to different cluster membership.

### 8.0.2 Probabilistic Foundation: Gaussian Mixture Model

Minimization of the loss function (8.2) can be viewed as maximizing the likelihood when the data is generated by the following probabilistic model, called Gaussian Mixture Model (GMM). We fix ground truth centers $u_1^*, \dots, u_k^* \in \mathbb{R}^d$. The prior probability of sample $\mu$ to belong to cluster $a$ is

$$P(v_\mu^*) = \sum_{a=1}^{k} \rho_a \delta(e_a - v_\mu^*), \quad \text{where } \rho_a \geq 0 \text{ and } \sum_{a=1}^{k} \rho_a = 1, \tag{8.3}$$

here $\rho_a$ is the probability to belong to cluster $a$ and $e_a$ – basis vectors in $k$ - dimensional canonical basis, so that $v_\mu^*$ are one-hot encodings of the clusters.

Now, for given $u$ and $v$ the probability of generating the data matrix $X$ is

$$P(X|u, v) = \prod_{\mu=1}^{n} \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\Delta}} e^{-\frac{1}{2\Delta}(X_{\mu i} - \sum_{a=1}^{k} v_{\mu a}^* u_{ai}^*)^2}. \tag{8.4}$$

Therefore, the whole GMM data generation process is: first, sample a cluster index with probability $\rho_a$, and then, generate a gaussian vector with the mean in the center of this cluster and variance $\Delta$.

Notice, that maximizing the likelihood (8.4) is exactly the same as minimizing the loss (8.2). However, the MLE doesn't take into account the prior distribution and ignores the fact that $v$ is one-hot encoded. If the prior is taken into the account (as can be seen from writing the posterior), we still need to minimize the same loss function, at the same time satisfying the constraints on $v$. This leads to an algorithm that does alternate minimization of $\mathscr{L}(u, v)$:

- Fixing centers $u$, minimize over assignments $v$: When we hold the cluster centers fixed and seek the best assignment for each data point, the loss is minimized by assigning each point to its closest center. This recovers the $k$-Means Assignment Step.
- Fixing assignments $v$, minimize over centers $u$: When we hold the cluster assignments fixed and seek the best location for the centers, we can set the derivative of the loss function with respect to the centers to zero. Doing so reveals that the optimal center is simply the mean of all points assigned to that cluster. This recovers the $k$-Means Update Step.

# 9 Features, Kernel methods

## 9.1 Representer theorem

**Theorem 2** (Representer theorem)**.** *Consider the linear model, learning the vector $w \in \mathbb{R}^d$ given the data $X \in \mathbb{R}^{n \times d}$, $y \in \mathbb{R}^n$ by minimizing the loss function of the form*

$$\mathscr{L}(w) = \sum_{\mu=1}^{n} l(y_\mu, w^T X_\mu) + \lambda \sum_{i=1}^{d} w_i^2. \tag{9.1}$$

*Any minimizer of the loss (9.1) can be written as*

$$\hat{w} = \sum_{\nu=1}^{n} \alpha_\nu X_\nu. \tag{9.2}$$

*Proof.* Notice that any vector $w \in \mathbb{R}$ can be written as a sum of the vector that lies in the span of $X$ and the vector orthogonal to it:

$$w = \sum_{\nu=1}^{n} \alpha_\nu X_\nu + w^\perp, \quad \text{where } X_\nu^T w^\perp = 0 \ \forall \nu = 1, \dots, n.$$

Then, we can write the loss as

$$\begin{aligned}
\mathscr{L}(w) &= \sum_{\mu=1}^{n} l\left(y_\mu, \sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i}\right)X_{\mu,i} + \sum_{i=1}^{d} w_i^\perp X_{\mu,i}\right) + \lambda \sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i} + w_i^\perp\right)^2 \\
&= \sum_{\mu=1}^{n} l\left(y_\mu, \sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i}\right)X_{\mu,i}\right) + \lambda\left[\sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i}\right)^2 + 2\sum_{i=1}^{d}\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i} w_i^\perp + \sum_{i=1}^{d}(w_i^\perp)^2\right] \\
&= \left[\sum_{\mu=1}^{n} l\left(y_\mu, \sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i}\right)X_{\mu,i}\right) + \lambda\sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu,i}\right)^2\right] + \lambda\sum_{i=1}^{d}(w_i^\perp)^2.
\end{aligned} \tag{9.3}$$

Assume, $\hat{w} = \sum_{\nu=1}^{n}\alpha_\nu X_\nu + \hat{w}^\perp$ is a minimizer of the loss (9.1). But then $\hat{w}^\parallel = \sum_{\nu=1}^{n}\alpha_\nu X_\nu$ is also a minimizer of the loss, since $\mathscr{L}(\hat{w}^\parallel) = \mathscr{L}(\hat{w}) - \lambda\sum_{i=1}^{d}(w_i^\perp)^2 \le \mathscr{L}(\hat{w})$.

Therefore, there always exists a minimizer of the loss (9.1) of the form

$$\hat{w} = \sum_{\nu=1}^{n} \alpha_\nu X_\nu.$$

□

Now, using this theorem, we can rewrite the linear model:

$$\hat{y}_\mu = \sum_{i=1}^{d} \hat{w}_i X_{\mu,i} = \sum_{i=1}^{d}\sum_{\nu=1}^{n} \alpha_\nu X_{\nu,i} X_{\mu,i} = \sum_{\nu=1}^{n}\alpha_\nu\left(\sum_{i=1}^{d} X_{\nu,i}X_{\mu,i}\right). \tag{9.4}$$

And the loss function becomes

$$\begin{aligned}
\mathscr{L}(\alpha) &= \sum_{\mu=1}^{n} l\left(y_\mu, \sum_{\nu=1}^{n}\alpha_\nu\left(\sum_{i=1}^{d} X_{\nu,i}X_{\mu,i}\right)\right) + \lambda\sum_{i=1}^{d}\left(\sum_{\nu=1}^{n}\alpha_\nu X_{\nu i}\right)^2 \\
&= \sum_{\mu=1}^{n} l\left(y_\mu, \sum_{\nu=1}^{n}\alpha_\nu\left(\sum_{i=1}^{d} X_{\nu,i}X_{\mu,i}\right)\right) + \lambda\sum_{\nu,\mu}\alpha_\nu\alpha_\mu\left(\sum_{i=1}^{d} X_{\nu,i}X_{\mu,i}\right).
\end{aligned} \tag{9.5}$$

We can choose to optimize over the coefficients $\alpha$ instead of $w$ to find the minimizer of the loss. One advantage of this approach is, that it reduces the computational costs, when $n < d$, since we only need to optimize over $n$ parameters $\alpha_\nu$ instead of $d$ parameters $w_i$.

## 9.2 Kernel method

Notice, that for linear model in this form we don't use the feature vectors $X_\mu$ themselves, we only use scalar products between them. This naturally leads to the definition of the Gramm matrix:

$$K_{\mu\nu} = \sum_{i=1}^{d} X_{\nu,i}X_{\mu,i} = X_\mu^T X_\nu. \tag{9.6}$$

The loss can be rewritten as

$$\mathscr{L}(\alpha) = \sum_{\mu=1}^{n} l\left(y_\mu, \sum_{\nu=1}^{n} \alpha_\nu K_{\nu\mu}\right) + \lambda \sum_{\nu,\mu} \alpha_\nu \alpha_\mu K_{\nu\mu}\,, \tag{9.7}$$

and the estimate on the new sample

$$\hat{y}_{\text{new}} = \sum_{i=1}^{d} w_i X_{\text{new},i} = \sum_{\nu=1}^{n} \alpha_\nu \sum_{i=1}^{d} X_{\nu,i} X_{\mu,i} = \sum_{\nu=1}^{n} \alpha_\nu K_{\nu,\text{new}}. \tag{9.8}$$

We can view the scalar product $K_{\mu\nu}$ as a measure of similarity between the samples $X_\mu$ and $X_\nu$, and linear regression as a way to sum the labels of the samples in the training set with weights proportional to this similarity. But then, we also can consider a similarity measure different from the scalar product, such as

- Exponential kernel: $K(X_\nu, X_\mu) = e^{X_\nu^T X_\mu}$;
- Radial basis kernel: $K(X_\nu, X_\mu) = e^{\|X_\nu - X_\mu\|^2}$.

This model with general kernel not necessarily equal to the scalar product is called kernel regression.

### 9.2.1 Feature Maps

In Section 5.3, we introduced the problem of non-linearly separable data and potential solutions. Now, we will develop the idea of modifying the existing data to create new useful features. In the examples in the Figure 17, the transformations that allow to linearly separate the data are

1. $\Phi(x_1, x_2) = x_1 x_2$;
2. $\Phi(x_1, x_2) = (x_1^2, x_2^2)$.

Another example of adding useful features to solve the problem is polynomial regression, that we discussed in Section 2.5. There, from the given $x$ we created new features $\Phi(x) = (1, x, x^2, x^3, \ldots, x^p)$.

We can formalize this concept of adding useful features by defining the feature map: a function $\Phi(x) : \mathbb{R}^d \to \mathbb{R}^p$, that maps from original features to the new ones.

After we apply the map, we perform the linear regression with the new features:

$$\hat{y}_\mu = w^T \Phi(X_\mu).$$

Therefore, analogously to the ordinary linear regression, we can rewrite the loss function using a kernel $K(X_\nu, X_\mu) = K(\Phi(X_\nu), \Phi(X_\mu)) = \sum_{a=1}^{p} \Phi_a(X_\nu)\Phi_a(X_\mu)$:

$$\mathscr{L}(\alpha) = \sum_{\mu=1}^{n} l\left(y_\mu, \sum_{\nu=1}^{n} \alpha_\nu K(X_\nu, X_\mu)\right) + \lambda \sum_{\nu,\mu} \alpha_\nu \alpha_\mu K(X_\nu, X_\mu)\,. \tag{9.9}$$

Now, we will consider the following questions: Can we always find a feature map that helps us learn a desired function? And can we find an appropriate feature map to express any kernel?

### 9.2.2 Feature Map expressivity

Consider the case of supervised regression in dimension $d = 1$. We believe, that the labels $y_\mu$ can be expressed as some function of samples $x_\mu$: $y_\mu = f(x_\mu)$. If the function $f$ is "reasonable", we can do a Tailor expansion of this function, and therefore, we can write the function in the form

$$f(x) = \sum_{a=0}^{\infty} \alpha_a x^a. \tag{9.10}$$

Now let's define a "infinite-dimensional" feature map $\Phi(x) = (1, x, x^2, \ldots, x^k, \ldots)$, which allows us to express the true label function. Assuming the values of $X_\mu$ and $X_\nu$ to be normalized so that $|X_\mu X_\nu| < 1$, we can compute the kernel

$$K(X_\mu, X_\nu) = \sum_{p=0}^{\infty} X_\mu^p X_\nu^p = \frac{1}{1 - X_\mu X_\nu} \tag{9.11}$$

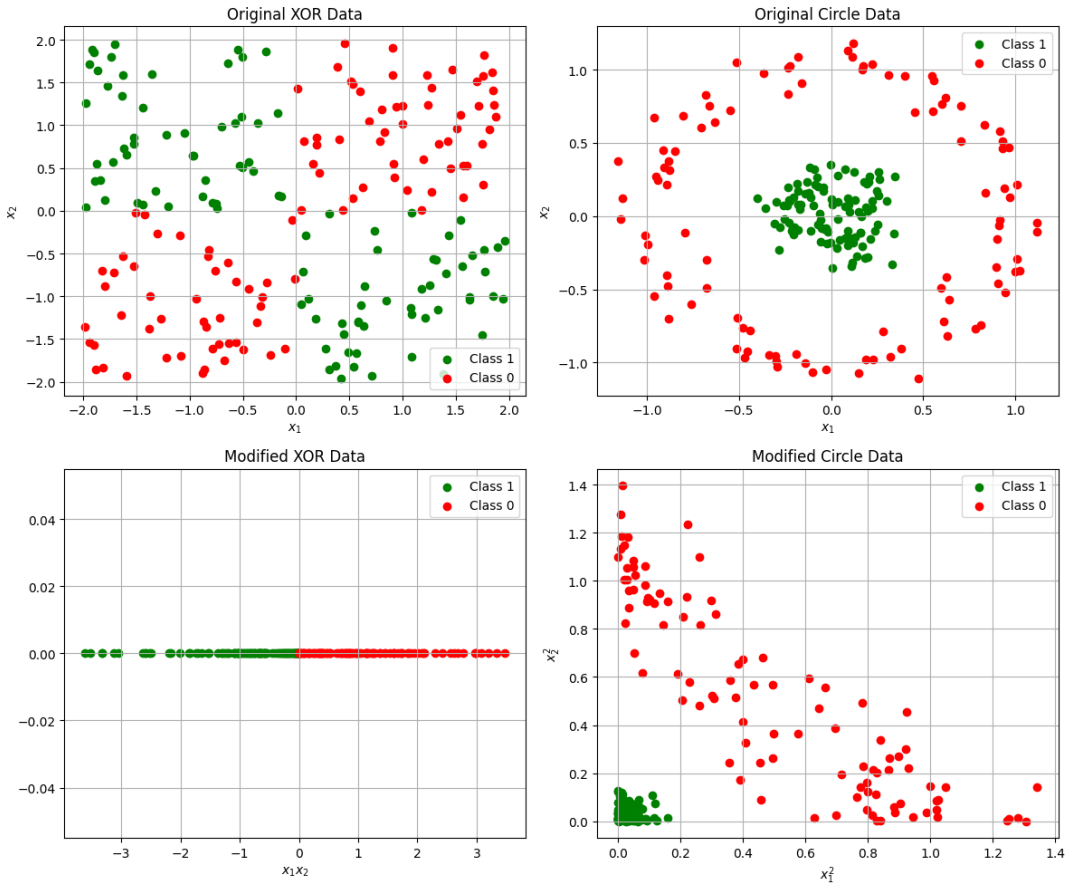



Figure 17: Examples of non-linearly separable data and their linearly separable representations.

by using the geometric progression formula.

Notice that the implicit infinite-dimensional representation of the data reduced to easily computable kernel depending only on 2 variables.

Transitioning to the case of general $d > 1$, we can find a feature map that leads to the same kernel

$$K(X_\mu, X_\nu) = \frac{1}{1 - X_\mu^T X_\nu} = \sum_{p=0}^{\infty} (X_\mu^T X_\nu)^p. \tag{9.12}$$

Expanding the multinomial, we find

$$\sum_{p=0}^{\infty} (X_\mu \cdot X_\nu)^p = \sum_{p=0}^{\infty} \sum_{\beta_1+\beta_2+\cdots+\beta_d=p} \frac{p!}{\prod_{i=1}^d \beta_i!} \prod_{i=1}^d (X_{\mu i} X_{\nu i})^{\beta_i}. \tag{9.13}$$

The coefficient $\frac{p!}{\prod_{i=1}^d \beta_i!}$ accounts for the number of times the term $\prod_{i=1}^d x_i^{\beta_i}$ appears in the expanded form of $(\sum_{i=1}^d x_i)^p$[14]. From this kernel, we find that the feature space can be constructed as:

$$\Phi_{\{\beta_1+\beta_2+\cdots+\beta_d=p\}_{p=0}^{\infty}}(X_\mu) = \sqrt{\frac{p!}{\prod_{i=1}^d \beta_i!}} \prod_{i=1}^d X_{\mu i}^{\beta_i}. \tag{9.14}$$

However, the expansion was done assuming that the infinite sum is converging, which is not always the case with generic data. The data should be carefully normalized, which sometimes is impractical.

[14] which is the same as the number of ways to place $p$ objects in $d$ boxes such that there are $\beta_1$ elements in box 1, $\beta_2$ elements in box 2 and so on

We can now treat the exponential kernel, because the exponent expansion doesn't require the argument to be bounded. Proceeding as before we have:

$$K(X_\mu, X_\nu) = e^{X_\mu X_\nu} = \sum_{p=0}^{+\infty} \frac{(X_\mu \cdot X_\nu)^p}{p!}\,. \tag{9.15}$$

For this kernel, exploiting the multinomial expansion, the feature space can be constructed as:

$$\Phi_{\{\beta_1+\beta_2+\cdots+\beta_d=p\}_{p=0}^{\infty}}(X_\mu) = \sqrt{\frac{1}{\prod_{i=1}^{d}\beta_i!}} \prod_{i=1}^{d} X_{\mu i}^{\beta_i}. \tag{9.16}$$

So we find that the features are the same as in Eq.(9.14), with only one coefficient changed.

The final kernel we have introduced in Eq.(**??**), the radial basis kernel, is the most commonly used. Its feature map is given by

$$\Phi_{\beta_1,\beta_2,\cdots,\beta_d}(X_\mu) = \sqrt{\frac{1}{\prod_{i=1}^{d}\beta_i!}}\sqrt{2^p}\exp\left\{-\sum_{i=1}^{d} {X_{\mu i}}^2\right\}\prod_{i=1}^{d} X_{\mu i}^{\beta_i} \tag{9.17}$$

and is also called the *radial basis function* (RBF).

### 9.2.3 Informal introduction to Support Vector Machine

Let us now consider the binary classification task. It is possible to show, that with large enough number of arbitrary features, we can always separate the samples in two classes. Moreover, there will be infinitely many separating hyperplanes.

In that case, one could use hinge loss (5.7), that was introduced in Section 5.1. It encourages a margin between the separating hyperplane and the sample points and therefore helps to find a better generalizing model.

Applying "kernel trick", i.e. using the representer theorem to switch from parameters $w$ to parameters $\alpha$, with this loss is called Support Vector Machine. It is a very popular classical method, that allows to perform non-linear classification. The name comes from the fact, that the optimal $\alpha_\nu$ values are non-zero only for the examples in the training set that are the closest to the decision boundary (they all are exactly at a distance equal to the margin between the hyperplane and the sample closest to the hyperplane).

### 9.2.4 Kernel method drawbacks

To interpret what kernel regression does, consider a one-dimensional example. Assume we have a function to estimate the label:

$$f(X) = \sum_{\nu=1}^{n} \hat{\alpha}_\nu K(X_\nu, X), \tag{9.18}$$

where coefficients $\hat{\alpha}_\nu$ are learned by optimizing the loss (9.7) and $K(X_\nu, X)$ measures the similarity between the new point and the training sample $\nu$.

We can depict functions $f_\nu(x) = \hat{\alpha}_\nu K(X_\nu, x)$ separately on a graph (see Figure 18).

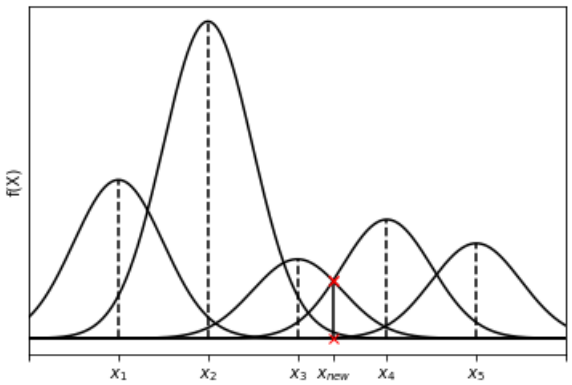



Figure 18: Single terms $f_\nu(x)$ around five different training samples $X_\nu$.

Now, let's place a new sample $X_{\text{new}}$ as shown in Figure **??** and compute the $f(X)$ corresponding to it. We notice that contributions from $X_1$ or $X_5$ have little weight at $X_{\text{new}}$, but we sum the contributions from $X_3$, $X_4$, and $X_2$ because their close distance to $X_{\text{new}}$ adds more weight to them.

This is similar to the $k$-nearest neighbors method – the kernel method computes a *weighted* average of the training points, so that similar points in terms of $K(X_\nu, X_\mu)$ get more weight, while further points almost don't contribute to the result. Unfortunately, this means the method may suffer from the curse of dimensionality in the same way as the $k$-nearest neighbors method did.

Another possible drawback is computational. Optimizing the loss of kernel regression requires to store the matrix $K(X_\mu, X_\nu) \in \mathbb{R}^{n\times n}$ and do computations that require quadratic number of steps in terms of $n$. With the modern dataset sizes, this is infeasible. Ideally, we would like our methods to take linear number of steps in terms of $n$, as for example, in gradient descent, and we will discuss how to achieve this in the next chapter.

# 10 Introduction to Neural Networks

## 10.1 Random Features Regression

In the previous chapter we discussed the feature mappings that allowed us to express a very wide class of functions, which, unfortunately, is computationally infeasible for large dataset size $n$. Now we introduce a way to mitigate this issue.

Consider the following feature mapping:

$$\Phi_a(X) = \sigma(F_a^T X), \quad \text{for } a = 1, \dots, p, \tag{10.1}$$

where $F_a \in \mathbb{R}^d$ are chosen at random, e.g. $F_{ai} \sim P(F) = \mathcal{N}(0, 1)$, and $\sigma$ is a non-linear function, e.g. $\sigma(x) = \tanh(x)$ or $\sigma(x) = \text{sign}(x)$.

It was shown in [4] that for many popular $K(X_\mu, X_\nu)$, we can choose a non-linear transformation $\sigma$ and distribution $P(F)$, so that

$$K(X_\mu, X_\nu) = \sum_{a=1}^{\infty} \sigma(F_a^T X_\mu)\sigma(F_a^T X_\nu). \tag{10.2}$$

Now, to resolve the computational drawback of the kernel method, which is equivalent to having infinite-dimensional features, we can truncate the sum (10.3), using only $D$ features:

$$K(X_\mu, X_\nu) \simeq \sum_{a=1}^{D} \sigma(F_a^T X_\mu)\sigma(F_a^T X_\nu). \tag{10.3}$$

The parameter $D$ should be sufficiently large, so that the approximation of the kernel is good, but as long as $D < n$ we are saving compute by performing linear regression on this random features mapping instead of kernel regression on original samples. The model becomes

$$\hat{y} = \sum_{a=1}^{D} w_a \Phi_a(X) = \sum_{a=1}^{D} w_a \sigma(F_a^T X). \tag{10.4}$$

and we minimize the loss w.r.t.$w$.

## 10.2 Neural Networks

To transition from random features regression to a neural network, we only need to do one step: make $F_a$ learnable instead of sampling and fixing them. The prediction will take the form

$$\hat{y}_{\text{new}} = \hat{\sigma}\left(\sum_{a=1}^{p} w_a \sigma\left(\sum_{i=1}^{d} F_{ai} X_{\text{new},i}\right)\right), \tag{10.5}$$

where $\hat{\sigma}$ can be identity (e.g. for regression task), or some different function (e.g. sigmoid for binary classification task). And the optimization problem we would like to solve is

$$\hat{w}, \hat{F} = \arg\min_{w\in\mathbb{R}^p, F\in\mathbb{R}^{p\times d}} \left\{ \mathscr{L}(w, F) = \frac{1}{n}\sum_{\mu=1}^{n} l\left(y_\mu, \sum_{a=1}^{p} w_a \sigma\left(\sum_{i=1}^{d} F_{ai} X_{\mu i}\right)\right)\right\}. \tag{10.6}$$

To find the optimal parameters we can still use gradient descent, as long as we can compute the gradient of the loss function.

### Terminology

First, let's consider a visual representation of a neural network defined by (10.5), as depicted in Figure 19. The nodes represent the values, while the edges represent the coefficients with which this values are recombined. Non-linear function $\sigma$ is applied "to each node" of the hidden layer before passing the values further to the output layer.

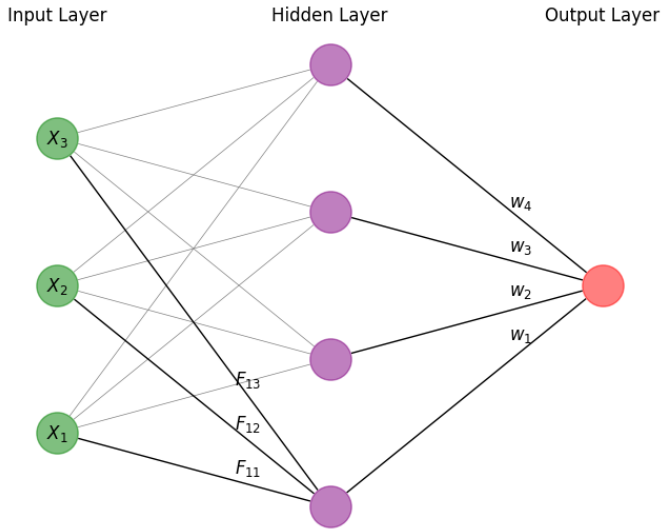



Figure 19: Example of a two layer neural network with $d = 3$ and $p = 4$.

Let us introduce the terminology, which is usually used to discuss neural networks:

**Weights.** Learnable parameters $w$ and $F$ are called weights of the neural network.

**Layers.** A layer is a collection of nodes (neurons) that represent the transformation of data coming from the previous layer (except for input layer, which represents the unmodified data sample).

**Depth ($L$).** The number of layers with learnable weights. In this example, we have two layers of learnable weights, and therefore, depth $L = 2$ neural network. Alternatively, we can call it a *1 hidden layer* neural network, because it has only one layer of "hidden" nodes between the input and the ouput layers.

**Width ($p$).** The number of neurons in the hidden layer (in our example $p = 4$).

**Activation Function.** A non-linear function $\sigma$ that is applied to the output of each hidden neuron. We will discuss common choices of $\sigma$ later.

**Pre-activation.** The term, to which the activation function is applied, i.e. $h_{a\mu} = F_a^T X_\mu$.

**Post-activation** The term after the activation $t_{a\mu} = \sigma(F_a^T X_\mu)$.

This kind of neural networks is called *fully-connected* or *feed-forward* neural network, because in the graphical representation we can see, that all the neurons from the previous layer are connected to all the neurons of the next layer.

Notice, that one-layer neural network represents exactly the same model as linear regression.

### Choice of activation function

Non-linear activation function is what makes a deep neural network different from a (reparametrized) linear regression. Indeed, if we take $\sigma(h) = h$, the estimator that we obtain is just a linear combination of the input features.

The most commonly used activation function is called **rectified linear unit** (ReLU):

$$\sigma(h) = \max(0, h). \tag{10.7}$$

Other popular choises are:

- **hyperbolic tangent**: $\sigma(h) = \tanh(h)$;

- **sigmoid function**: $\sigma(h) = 1/(1 + e^{-h})$.

To draw an analogy with biological neural network, one can consider an activation function $\sigma(h) = \text{sign}(h)$, which ensures that the neurons have binary states (they fire or they don't), and the weights represent the synaptic strength of the connection between the neurons. However, this approach is not suitable for optimization with gradient descent, because the function $\text{sign}(h)$ has 0 gradient almost everywhere.

## 10.3 Universal Approximation theorem

First, we will state a more general form of estimator function, including biases $b$ in addition to weights $w$ and $F$:

$$f(X) = \sum_{a=1}^{p} w_a \sigma \left( \sum_{i=1}^{d} F_{ai} X_i + b_a \right) + b. \tag{10.8}$$

Then, we can state the result that establishes the expressive capacity of neural networks:

**Theorem 3** (Universal Approximation theorem)**.** *Any continuous function $f^* : \mathbb{R}^d \to \mathbb{R}$ can be approximated by a two-layer neural network of the form 10.8, so that*

$$\forall \epsilon > 0 \ \exists w, F, b \quad \sup_{X \in \mathbb{R}^d} \|f(X) - f^*(X)\| < \epsilon,$$

*as long as activation function $\sigma$ is not a polynomial and the width of the network $p$ is large enough.*

This result guarantees that neural networks are, in principle, powerful enough to represent almost any function of interest. The proof, however, only establishes the existence of the required weights, not how to find them. The training process involves minimizing a high-dimensional, non-convex loss function. It was proven that finding the minimizer in general is an NP-complete problem. This means that in the worst-case scenario, the time required to find the global minimum of the loss function can be exponential in the problem size.

However, it was empirically discovered that gradient descent, while offering no guarantees of finding the true minimizer, finds the solution in many practically interesting cases. A central open scientific question remains: how to characterize the data structure that make the training tractable?

## 10.4 Deep Neural Networks

In theory, two layer should be enough to approximate any continuous function. In practice, usually more than two layers are used, because empirically they produce better results (smaller test error). An example of a deep ($L = 4$) feed-forward neural network is depicted in Figure 20.

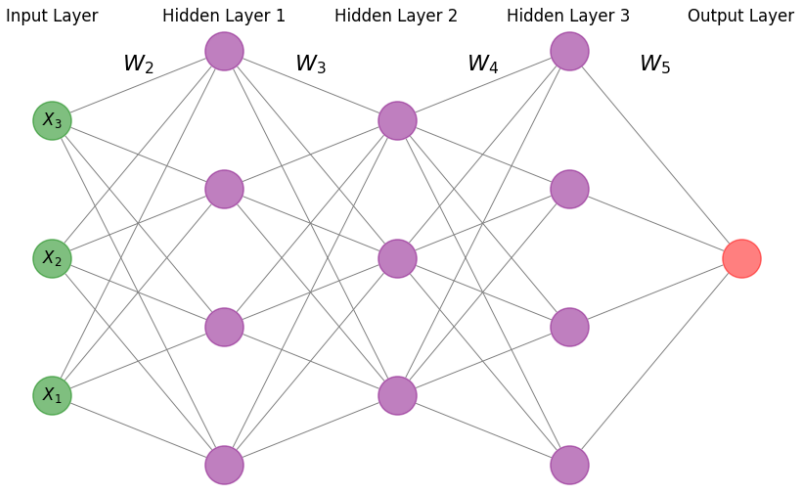



Figure 20: Example of a 4 layer neural network.

The function that this network represents is

$$f_W(X_\mu) = \phi^{(L)} \left( \sum_{a_L=1}^{p_L} W_{a_L}^{(L)} \phi^{(L-1)} \left( \cdots \phi^{(2)} \left( \sum_{a_2=1}^{p_2} W_{a_3 a_2}^{(2)} \phi^{(1)} \left( \sum_{a_1=1}^{p_1} W_{a_2 a_1}^{(1)} X_{\mu a_1} \right) \right) \right) \right), \tag{10.9}$$

where $W = (W^{(1)}, \dots, W^{(L)})$ is the collection of trainable weights, $W^{(l)} \in \mathbb{R}^{p_{l+1}, p_l}$, $p_1 = d$, $p_{L+1} = 1$ and $\phi^{(l)}$ are activation functions at each layer.

## 10.5 Training Neural Networks

We have already discussed all the components of the training procedure in this course before:

**Model** — a parametric function that maps the data samples to the labels (e.g. linear regression);

**Loss function** — a function that measures the quality of our model, i.e. how far the predictions are from the true labels. One common loss for regression problems is the quadratic loss, we have already discussed in the context of linear regression (2.3):

$$l(y, z) = (y - z)^2,$$

where the target $y$ and the model output $z$ are scalars. And for multi-class classification, cross-entropy loss is often used (5.14):

$$l(y, p) = -\sum_{a=1}^{k} y_a \ln(p_a),$$

where the target $y$ is one-hot encoding of the class, i.e. $y_a = \delta_{a,\text{true class label}}$ and $p \in \mathbb{R}^k$ represents the probability predicted by the model, using softmax transformation of the model outputs $z \in \mathbb{R}^k$: $p_a = \frac{e^{z_a}}{\sum_{b=1}^{k} e^{z_b}}$;

**Optimization procedure** — a way to find the parameters of the model that minimize the loss function. Most commonly it is a variant of stochastic gradient descent, discussed in Chapter 4:

$$W_{ab}^{(l)}(t+1) = W_{ab}^{(l)}(t) - \gamma \sum_{\mu \in B_t} \left. \frac{\partial l(y_\mu, f_W(X_\mu))}{\partial W_{ab}^{(l)}} \right|_{W(t)}, \tag{10.10}$$

where $\gamma$ is the learning rate and at each iteration we take the batch of samples $B_t$. The pass through all the samples in the training set is usually called an *epoch*, and the batches are usually resampled at each epoch;

**Validation** — a process of evaluating the performance of the model to find the best hyperparameters, which we discussed in detail in Section 2.6. Notice, that we can measure the prediction quality with a metric not necessarily equal to the loss, such as accuracy (the fraction of correctly predicted samples) for classification:

$$A = \frac{1}{n} \sum_{\mu=1}^{n} \delta_{y_\mu, \hat{y}_\mu}, \quad \text{where } \hat{y}_\mu = \arg\max_a (f_W(X_\mu))_a.$$

Notice that compared to previously considered models, deep neural networks introduce additional hyperparameters: the number of layers $L$, the width $p_l$ and activation function $\phi^{(l)}$ for each layer $l = 1, \dots, L$ and the way to initialize each layer.

While gradient calculation for a deep network is more difficult than in the previously discussed models, we can compute them efficiently using the **back-propagation algorithm**.

1. **Initialization.** The algorithm starts by randomly initializing the weights, usually using normal distribution $\mathcal{N}(0, \sigma_{\text{init}})$.

2. **Forward pass.** For each sample in the batch, we compute and save pre-activations and post-activations of each layer (we start from initial post-activation $t_\mu^{(0)} = X_\mu$, which is the input sample itself):

$$\begin{aligned} h_{\mu a}^{(l)} &= \sum_b W_{ab}^{(l)} t_{\mu b}^{(l-1)}, \\ t_{\mu a}^{(l)} &= \phi^{(l)}(h_{\mu a}^{(l)}), \end{aligned} \tag{10.11}$$

   until we obtain the final $t_\mu^{(L)} = f_W(X_\mu)$.

3. **Backward pass.** This is the step that allows us to efficiently compute the gradients using the chain rule. We start by expressing the derivatives of the loss with respect to the input at the last layer (the post-activation of the previous layer):

$$\delta_a^{(L)} = \frac{\partial l(y, f_W(X))}{\partial h_a^{(L)}} = \frac{\partial l(y, t^{(L)})}{\partial t^{(L)}} \frac{\partial t^{(L)}}{\partial h_a^{(L)}} = \frac{\partial l(y, t^{(L)})}{\partial t^{(L)}} (\phi^{(L)})'(h_a^{(L)}) \tag{10.12}$$

Now, if we want to express the derivative with respect to the input of the previous layer $l-1$ given the derivatives of the layer $l$ we can write:

$$\delta_a^{(l-1)} = \frac{\partial l(y, f_W(X))}{\partial h_a^{(l-1)}} = \sum_b \delta_b^{(l)} \frac{\partial h_b^{(l)}}{\partial h_a^{(l-1)}} = \sum_b \delta_b^{(l)} W_{ba}^{(l)} (\phi^{(l-1)})'(h_a^{(l-1)}). \tag{10.13}$$

This way we can recursively compute the derivatives for all layers from $L$ to 1 using the derivatives computed on the previous steps.

The derivatives with respect to the weight $W_{ab}^{(l)}$ are written as

$$\frac{\partial l(y, f_W(X))}{\partial W_{ab}^{(l)}} = \frac{\partial l(y, f_W(X))}{\partial h_b^{(l)}} \frac{\partial h_b^{(l)}}{\partial W_{ab}^{(l)}} = \delta_b^{(l)} \cdot t_a^{(l-1)}. \tag{10.14}$$

4. **Weights updates.** After the derivative with respect to the weight $W_{ab}^{(l)}$ is computed for each sample in the batch $B_t$, we can update the weight following the gradient descent iteration procedure (10.10).

## 10.6 What contributed to the Deep Neural Networks success

Deep Neural Networks were known since the second half of the 20th century with back-propagation algorithm being applied since the 70s. However, they became popular only recently due to the following reasons:

1. GPUs enabled parallel execution of many simple arithmetic operations, dramatically accelerating matrix multiplication and element-wise operations compared to CPUs. This parallelization significantly reduced deep network training times.

2. Clean and large datasets of labelled images were collected only recently:

   - MNIST, the handwritten digits database with $\approx$ 70k digits, developed in 1988.
   - CIFAR-10 (CIFAR-100), from the Canadian Institute For Advanced Research developed in 2009: The dataset contains $\approx$ 60k colored images in 10 (100) different classes (dogs, cats, etc.).
   - ImageNet — a dataset of 14 million images for 1000 classes was collected in 2010 using Amazon Mechanical Turk, a crowd-sourcing platform where people around the world are paid to label images.

In 2012, Deep Convolutional Neural Network (this architecture will be discussed in the next chapter) achieved 85% accuracy of classification on ImageNet, with the runner up architecture achieving $\approx$ 74 %. This was a jump in quality never observed before in Machine learning for image classification. It is often seen as the official start of the "Deep Learning revolution".

# 11 Convolutional networks

## 11.1 Motivation

Suppose we trained a fully-connected neural network to classify cat and dog images. All the weights in one layer are "symmetric" in the sense, that if we permute the "neurons" together with the connections, the output of the modified network will be the same. That leads to not taking into consideration local properties of the input. Therefore, to utilize the same useful features that humans use to recognize object in the pictures (e.g. pointy cat ears), we need to introduce some notion of **locality** in the network, so that the network is not permutationally invariant.

Another important aspect of the way humans view images is that the objects in the picture are recognizable even when the image is shifted (i.e. if the cat is in the top left corner or at the center of the image, we can still recognize

it as a cat). Therefore, we want to introduce **shift invariance** to the network, which fully-connected layers don't have.

Moreover, training a fully connected network can be very computationally costly. For an image of 100 by 100 pixels, the network input dimension is already $3 \times 10^4$, and for higher resolution images training a feed-forward network can be infeasible because of the large input dimension size.

Each neural network layer can be considered from an abstract perspective as a mapping from $\mathbb{R}^{p_{l-1}}$ to $\mathbb{R}^{p_l}$. So far, we have only seen fully connected layers, which are linear maps. But we can think of other types of layers, that will help us resolve the issues we stated before.

## 11.2 Building blocks of Convolutional Networks

### 11.2.1 Convolutional layer

A convolutional layer is a primary building block of a convolutional neural network (CNN) that applies a convolution operation to its input. The convolution operation involves sliding a small window (called a kernel or filter) across the input data and computing the dot product between the values in the kernel and the input at each position. This process creates a feature map that represents detected features in the input (i.e. lines or angles in the image).

**Convolutional layer in 1D** (illustrated in Figure 21). First, let's consider a convolution operation on 1-dimensional input in $\mathbb{R}^I$. First, we fix a filter, which is a vector in $\mathbb{R}^F$. Then we take a sliding window of size $F$ and slide it across the input vector, shifting the *receptive field* by $S$ positions at each step (the number of positions to shift is called *stride*). At each step the dot product between the filter and the receptive field is computed and stored as an element in the output vector.

Additionally, we can add zeroes $P_{\text{start}}$ and $P_{\text{end}}$ to each side of the boundaries of the input, which is called *zero-padding*. It is done, so that each item of the input vector appears on each position in receptive field and the filter "sees" the input end-to-end.

The output vector dimension can be computed as

$$O = \frac{I - F + P_{\text{start}} + P_{\text{end}}}{S}$$

Applying the kernel to all patches of the input helps the network to keep track of the *locality* in the data. It's important to notice that the values of the filter weights do not depend on which receptive filter we take. This property is known as *weight sharing* and it induces the *shift invariance* of the convolutional layer.

Notice, that convolutional layer is a special case of a fully-connected layer with special structure in the weights. In addition to previously discussed benefits, this structure helps to reduce the number of learned parameters and therefore makes the learning more efficient.

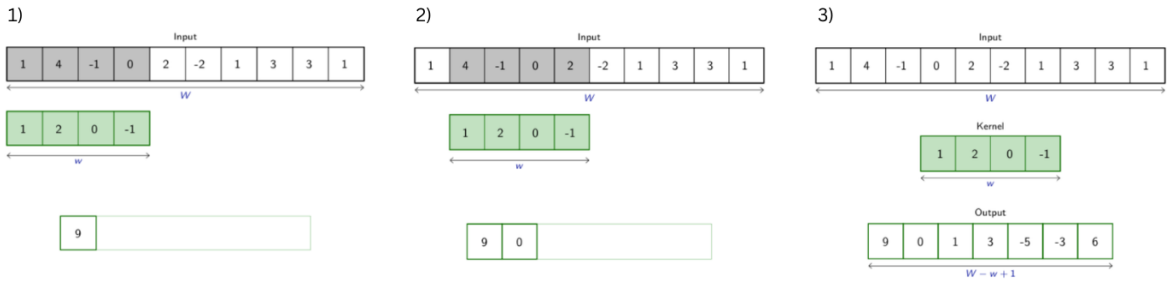



Figure 21: Convolution on 1D input.

**Convolutional layer in 2D** (illustrated in Figure 22). The input image is usually represented by a matrix in $\mathbb{R}^{I \times I}$ or a set of $C$ matrices corresponding to $C$ different color channels (typically for an image that can be displayed on the screen $C = 3$). This motivates a natural generalization of the convolution operation to higher dimension. We can define a filter, as a tensor in $\mathbb{R}^{H \times W \times C}$ and take the sliding window that moves both along horizontal and vertical dimensions of the input. The values in the receptive field get multiplied by the values of the filter and then all resulting values are summed up and stored in the output matrix.

Moreover, we can define several filters, that are applied to the same image, to create multiple output channels.

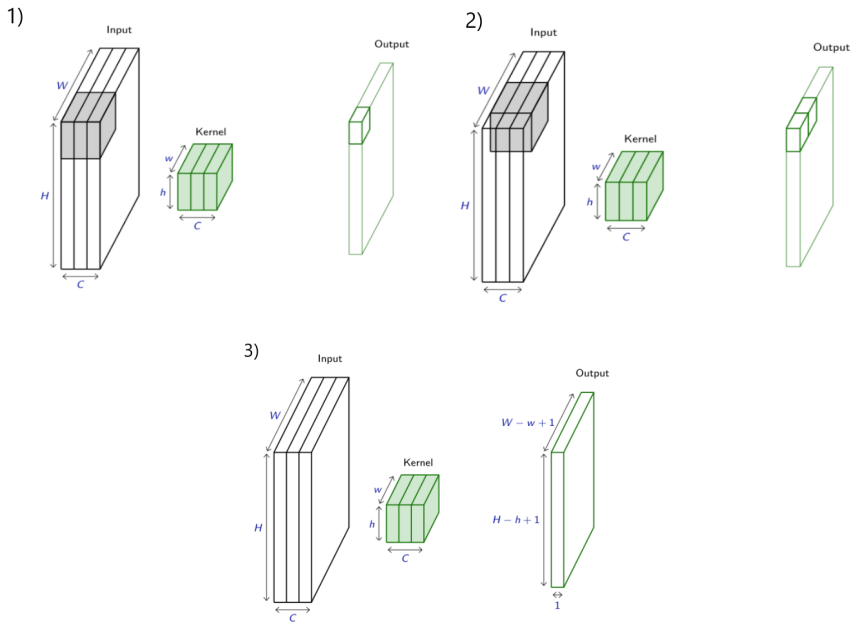



Figure 22: Convolution on 2D input with 3 channels

Mathematically, a convolution operation for a 2D input with $C_{\text{in}}$ channels can be expressed as

$$h_{ij}^k = \sum_{c=1}^{C_{\text{in}}} \sum_{m=i}^{i+F_h-1} \sum_{n=j}^{j+F_w-1} W_{c,mn}^k t_{mn}^c, \tag{11.1}$$

where $t_{mn}^c$ is the element of the input at position $(m, n)$ in channel $c$, $h_{ij}$ is the output at position $(i, j)$ in channel $k$, and a filter $W^k \in \mathbb{R}^{C_{\text{in}} \times F_h \times F_w}$ is applied to the input to create the output channel $k$.

### 11.2.2 Pooling Layer

Pooling is a down-sampling operation that summarizes the nearby inputs and helps to make the network *shift invariant*. The input is divided into equal size patches and the pooling operation is applied to each of the patches, as illustrated in the Figure 23. If the input has several channels, pooling is performed separately for each channel.

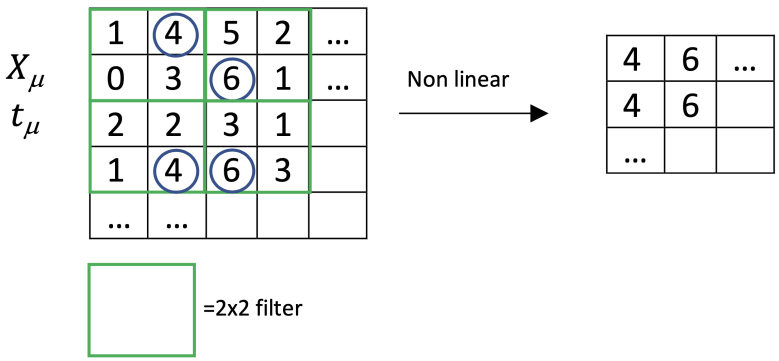
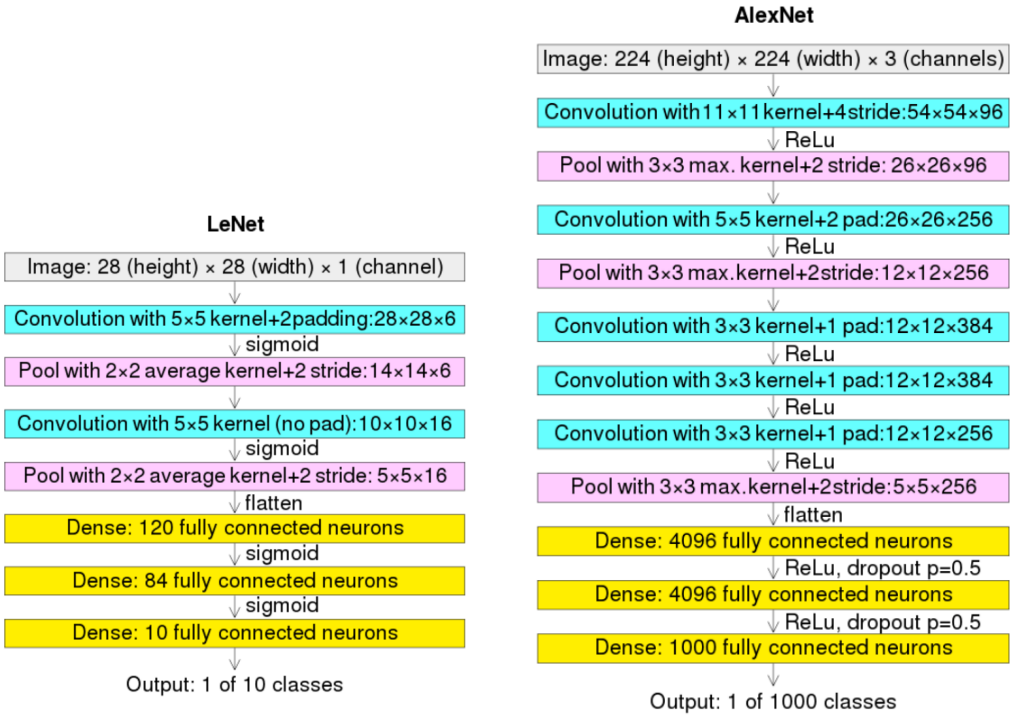



Figure 23: Example of max pooling with 2x2 filter applied to a matrix input.

The most popular types of pooling are Max pooling and Average pooling that take respectively the maximum and the average over each patch in the input.

## 11.3 Examples of CNN architectures

In a convolutional neural network, the hidden layers include one or more layers that perform convolutions. Typically, for image classification task, first several hidden layers interchange between convolutions and pooling and applying non-linear activation function in between them. Then the outputs get flattened and several fully connected layers are applied. To predict the class of the image, softmax function is applied to the last layer outputs, and the results are interpreted as probabilities of the image to belong to a particular class. In Figure 24, we show the architectures of two historically important networks: LeNet and AlexNet.




(a) LeNet CNN architecture.

(b) AlexNet CNN architecture

Figure 24: Examples of convolutional networks.

**LeNet** was one of the earliest convolutional neural networks and it was designed for reading small grayscale images of handwritten digits and letters. One of the factors in the success of this network for the digits recognition tasks was the use of a large (especially for that time period) dataset of hadwritten digits MNIST. The MNIST dataset is composed of handwritten digits represented in images of $28 \times 28$ pixels; the images are black-and-white and in this case we only have one channel. The database is made of 60'000 training and 10'000 testing images with labels from 0 to 9.

**AlexNet** was proposed in 2012 and became a catalyst for the following rapid development of the deep learning field. It was trained to classify images from the ImageNet dataset, made of higher resolution, and colored images ($224 \times 224$ pixels and 3 channels) which could represent objects of 1000 different classes.

In the subsequent years deeper models were able to achieve even better performance on the ImageNet classification task (see Figure 25). However, when the number of layers in the network increases, we may encounter a **vanishing gradient** problem during training. When the derivatives with respect to each layers input are small in absolute value, multiplying them according to the chain rule leads to a gradient extremely close to zero. Then gradient descent algorithm is unable to update the weights of the network.

An important advance occurred in 2015 with ResNet which had 152 layers. This architecture partially overcame the vanishing gradient problem by adding **residual connections** to the network. The idea of the residual connection will be discussed in more details in section 12.3.2.

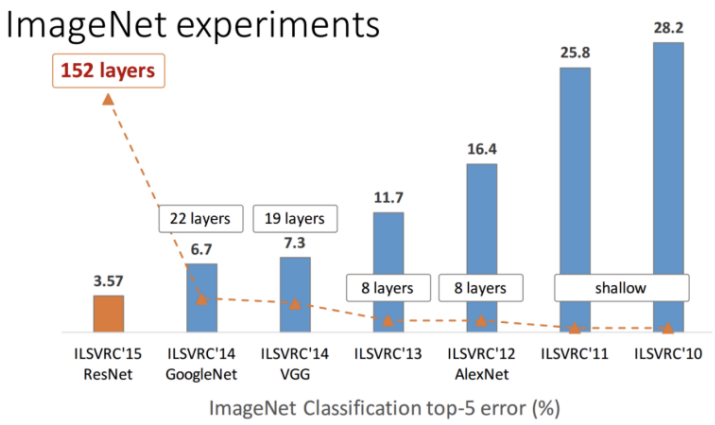



Figure 25: Classification top-5 error in percentage for the ImageNet experiment

# 12 Transformers

Some of the most famous recent advancements in the field of Machine Learning, namely Large Language Models (LLMs) such as ChatGPT, DeepSeek, Claude, etc. as well as some of the recent computer vision models are based on the transformer architecture introduced in the paper "Attention is all you need" (Vaswani, Shazeer, Parmar, *et al.* [5]).

So far, we discussed architectures where the input sample was treated as a single object (a vector or a tensor, that was modified as a whole in each layer). In the fully-connected layer the input data was presented as vectors in $\mathbb{R}^d$ and each layer $l$ transformed it into a vector of different dimension $\mathbb{R}^{p_l}$. In the convolutional networks the data and the output of each convolution layer were grids in the image resolution dimension times the number of channels $\mathbb{R}^{C\times H\times W}$.

However, sometimes it is useful to split the sample into parts. Let's take as an example the following task: we are given a sentence, e.g. "I like statistical ..." and we want to predict the next word in this sentence. As a human, you know, that the sentence as a whole consist of several words, i.e. parts, each of which has its own meaning, plus some meaning added from the combination of these parts[15]. Therefore, when we construct a model to work with such data, we want to conserve this essential structure of having several parts. The process of converting the input (e.g. the block of text or a whole image) into a sequence of parts called tokens is called tokenization and will be discussed in detail later.

## 12.1 Dot-product self-attention layer

Now we will focus on the neural network layer that allows us to look at tokens individually and combine them between each other to produce new representations.

We are given an input $X^\mu \in \mathbb{R}^{T\times d}$, where $T$ is the length of the sequence of tokens (e.g. number of words in the sentence) that we call *context length*. We call the vector representation $X^\mu_a \in \mathbb{R}^d$ of the token on position $a$ in the sequence an **embedding** of this token.

Attention layer[16] output $t^{(l+1)}_{ai}$ can be written as

$$t^{\mu,(l+1)}_{ai} = \sum_{b=1}^{T} A^{(l)}_{ab}(t^{\mu,(l)}) \sum_{j=1}^{d} t^{\mu,(l)}_{bj} V^{(l)}_{ji}, \tag{12.1}$$

where $V^{(l)} \in \mathbb{R}^{d\times d'}$ is called the value matrix, $A^{(l)} \in \mathbb{R}^{T\times T}$ is the attention matrix. Notice that attention layer takes the embeddings of each token, projects them to some new $d'$ dimensional space using the value matrix and then "mixes" this projections according to the weights from the attention matrix.

[15] sometimes you can even say that parts of words themselves have meaning, for example "statist" and "ical", where the first part gives semantic meaning and the second gives grammatical sense to the word.

[16] This layer is sometimes called a self-attention layer, because the attention scores are computed between the embeddings of the sequence and itself, while attention can also be computed between embeddings of two different sequences.

Attention matrix represents the "attention" a model should pay to each of the tokens while constructing the output embeddings, i.e. if the attention score between tokens $a$ and $b$ is large, then the projection of the embedding $t_b^{\mu,(l)}$ will be added in the next embedding $t_a^{\mu,(l+1)}$ with a large weight.

The most popular variant of attention computation is a softmax dot-product attention defined as follows:

$$A_{ab} = \frac{\exp\left\{\frac{1}{\sqrt{r}}\sum_{k=1}^{r}(\sum_{j=1}^{d} t_{aj}Q_{jk})(\sum_{j=1}^{d} t_{bj}K_{jk})\right\}}{\sum_{c=1}^{T}\exp\left\{\frac{1}{\sqrt{r}}\sum_{k=1}^{r}(\sum_{j=1}^{d} t_{aj}Q_{jk})(\sum_{j=1}^{d} t_{cj}K_{jk})\right\}}, \tag{12.2}$$

where $Q$ is a query matrix, $K$ is a key matrix in $\mathbb{R}^{d\times r}$. Let us take a look at this computation step by step:

1. For tokens at each position $a = 1, \dots, T$ we construct query and key representations: $Q^T t_a^{\mu,(l)}$ and $K^T t_a^{\mu,(l)}$
2. For each token $a$ in the sequence, we take its query projection and compute the dot-product with the key projections of all tokens to get the scores $(Q^T t_a^{\mu,(l)})^T (K^T t_b^{\mu,(l)})$ for $b = 1, \dots, T$
3. We normalize the scores using the softmax function. Notice that this operation "focuses" the attention on the tokens which key projections have larger dot-product with the query projection of the current token.

This computation can be written more concisely as

$$A = \text{softmax}(\frac{1}{\sqrt{r}}(tQ)(tK)^T) \tag{12.3}$$

### 12.1.1 Multi-head attention

Usually, transformer layers use more than one head of attention. First, the output of each head is computed:

$$t_{ai}^{\mu,(l+1),\gamma} = \sum_{b=1}^{L} A_{ab}^{\mu,(l),\gamma} \sum_{j=1}^{d} t_{bj}^{\mu,(l)} V_{ji}^{(l),\gamma} \tag{12.4}$$

and then we concatenate the outputs of different heads to get the final output $t_a^{\mu,(l+1)} = (t_a^{\mu,(l+1),1}, \dots, t_a^{\mu,(l+1),h})^T$. Usually, the dimension of each head is set to $\frac{d}{h}$ (both $r$ and $d'$), so that after concatenation the final output has the same dimension as the input to the layer.

The architecture is depicted in the Figure 26 from a course [6].

### 12.1.2 Number of trainable parameters and time complexity

Time complexity of the dot-product attention layer is different from the feed-forward or convolutional layer that are linearly dependent on all dimensions of the input. The attention layer requires to compute a $T \times T$ attention scores matrix, i.e. this computation time grows quadratically w.r.t. the context length, which might become the computational bottleneck when the sequence length grows.

At the same time, the number of parameters in each layer is $d \times d' \times h + 2d \times r \times h = 3d^2$, which doesn't depend on the context length $T$. While the fully-connected layer applied to the same sequence would require $d \times T$ parameters, which might require much more memory when the sequence length grows larger than the embedding dimension.

The predecessor of most modern LLM architectures GPT-2 had $L = 12$ layers with dimensions $d = 768, h = 12, r = 64$ of each layer and worked with sequences of length $T = 1024$. It is considered small compared to the SOTA models, but the dimension is still quite large.

## 12.2 Embeddings

The first step of training the transformer is to pre-process the data: separate it into a sequence of parts called tokens and convert this tokens into vectors in $\mathbb{R}^d$ called embeddings.

Let us start with discussion of tokenization for images. We are usually given a square image $W \times W$, which we split into patches of size $k \times k$. Each patch gets processed by a "backbone". After performing this operation we have $T = (\frac{W}{k})^2$ token embeddings. The backbone can be just taking all the pixels in all the channels as a vector and applying linear map on them or applying a small convolutional neural network on the patch. The backbone is usually trained together with the network.

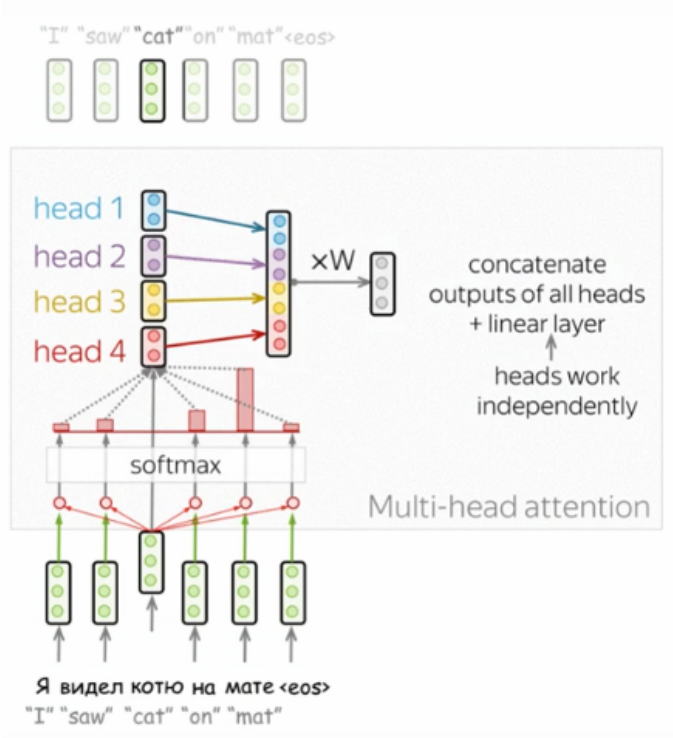



Figure 26: Multi-head attention layer schema.

Tokenization for language is different, because tokens come from a discrete set. A natural way to separate strings into parts is to use each word as a token, but this approach leads to a very large vocabulary and it is not clear, how to treat new words that we never encountered in training. Moreover, in some languages, e.g. German, it is common to concatenate several words together to create a new word, that in our approach will have to be treated as an entirely different new token and we will lose the information we could get from knowing the parts of the word separately. Another way to tokenize is to consider each letter to be a separate token. However, this approach leads to the increased sequence length that slows down the computations.

The most common approach is to use some algorithm that extracts most commonly encountered sequences of several characters to be the tokens, so that some popular words are considered as a single token, while words that do not often appear in the training set are separated into several tokens, e.g. in the sentence "I am learning","I" and "am" would be one token each, and "learning" gets separated into "learn" and "ing". In modern architectures, number of tokens used is usually $d_{\text{token}} \sim 5 \times 10^4$.

After tokenization, we can apply one hot encoding to the sequence of tokens to get vectors $R_a \in \mathbb{R}^{d_{\text{token}}}$, where $R_{aj} = 1$ if token on the position $a$ is the $j$-th token in the dictionary and 0 otherwise. And then we apply a trainable linear map $W \in \mathbb{R}^{d \times d_{\text{token}}}$ to the one-hot encoded token vectors to get the embeddings $X_{ai} = \sum_{j=1}^{V} W_{ij} R_{aj} + b_i$.

### 12.2.1 Positional encoding

So far, the architecture we discussed is invariant to the permutation of the tokens. However, in natural language and in computer vision positional information from the input data plays important role (e.g. sentences "The dog chases the cat" and "The cat chases the dog" have different meaning). Therefore, we introduce positional encodings to the network in order to keep the positional information.

We add the embeddings of the position to the token embedding:

$$t_{ai}^{\mu,(0)} = X_{ai}^{\mu} + P_{ai}, \tag{12.5}$$

where $P_a \in \mathbb{R}^d$ is encoding the position of the token, it doesn't depend on the sample itself.

There are different options for positional encodings (PE):

- Trainable PE, where $P \in \mathbb{R}^{T \times d}$ is trained together with the network.

- Sinusoidal PE (was originally used by Vaswani, Shazeer, Parmar, *et al.*).

$$\begin{cases} P_{a,2k} = \sin\left(\frac{a}{10000^{2k/d}}\right) \\ P_{a,2k+1} = \cos\left(\frac{a}{10000^{2k/d}}\right). \end{cases} \tag{12.6}$$

  The form of the encoding was motivated by the fact that if we multiply them the result $P_a^T P_b$ will only depend on the difference between $a$ and $b$, i.e. on the distance between the token positions in the sequence.

- Rotary PE - a positional encoding that is done by rotating the embedding vectors depending on their position in the sequence after they are projected by $K$ and $Q$ matrices. This approach captures the relative distance between tokens even for larger context lengths.

## 12.3 Vanishing and Exploding gradients

Before turning to the discussion of the tasks that can be solved by transformers, we will discuss some problems that can be encountered during training. The problems we want to discuss are general for any deep network and they arise from the fact that we stack several layers one after the other, so we will consider an example of the fully connected network.

The error in the backward pass of a fully connected network is defined recursively:

$$e_{a_{l-1}} = (\phi^{(l-1)})'(h^{\mu,(l-1)}_{a_{l-1}}) \sum_{a_l=1}^{p_l} W^{(l)}_{a_l a_{l-1}} e^{\mu,(l)}_{a_l}. \tag{12.7}$$

Therefore, if the values of $e_{a_l}$ for all layers are larger than 1, the gradient "explodes", i.e. the error grows exponentially fast with the number of layers. And if the values of $e_{a_l}$ for all layers are smaller than 1, the gradient "vanishes", i.e. the error goes to 0 exponentially fast with the number of layers.

### 12.3.1 Layer Normalization

To fix exploding gradient problem, we use a data-adaptive rescaling of the activation. We define:

$$\begin{aligned} \mu^{\mu,(l)} &= \frac{1}{p_l} \sum_{a_l=1}^{p_l} t^{\mu,(l)}_{a_l}, \\ \sigma^2 &= \frac{1}{p_l} \sum_{a_l=1}^{p_l} (t^{\mu,(l)}_{a_l} - \mu)^2. \end{aligned} \tag{12.8}$$

Then we can normalize the activation:

$$t^{\mu,(l)}_{a_l} \leftarrow \frac{t^{\mu,(l)}_{a_l} - \mu}{\sigma}. \tag{12.9}$$

This operation guarantees that input to the layers always has the same mean and variance, which stops outputs of the layers from blowing out and can additionally help the stability of the network when it receives outlier sample as input.

### 12.3.2 Residual connection

Another method to fix the vanishing gradient problem is residual connection. The idea is to take the output of some hidden layer $l_{\text{start}}$ and add it to the output of the sub-network from layer $l_{\text{start}} + 1$ to $l_{\text{end}}$, before passing it to the next layer as illustrated in Figure 27. This method helps to propagate the gradient and stabilizes the training of deep networks.

## 12.4 Transformer architecture

Previously in the course we discussed supervised learning, where the data consisted of samples and labels that we wanted to predict, or unsupervised learning, where we wanted to extract some properties of the data or learn useful representations without having explicit labels. Now we will discuss how transformers are trained to solve next token prediction task — a self-supervised task, i.e. when part of the sample itself becomes the label for training, which is discussed in more details in Chapter 14.

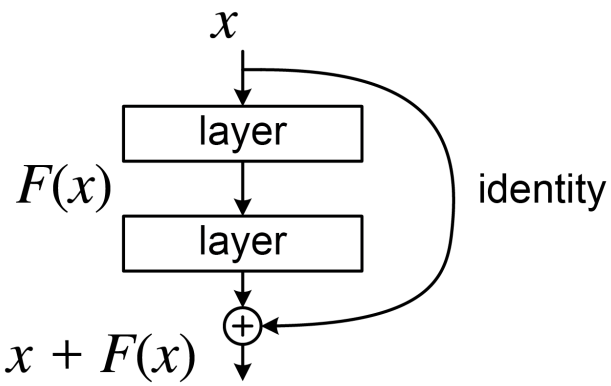



Figure 27: A residual block in a deep residual network. Here, the residual connection skips two layers.

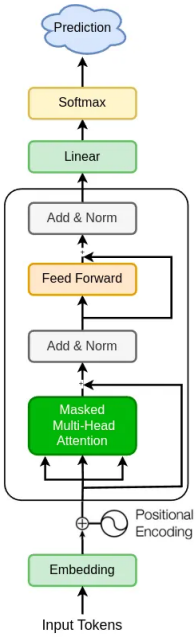



Figure 28: GPT2 architecture schema

The ability of LLMs to predict the next token given some input sequence is enabling them to answers questions and chat with users. Given an input written by a human, the LLM predicts the next token that should follow in the text and by continuing to do so until a special end-of-sequence token is generated it constructs the final answer.

To train a model, we take the tokenized sequence, remove the last token and give it to the model as input. Then we take the output embedding of the last token, and use it to solve a classification problem, that is to predict the "class" of the hidden token. To do that we us an "unembedding" layer (which usually applies a transpose of embedding matrix), then we get a vector in $\mathbb{R}^{d_{\text{token}}}$ and apply soft-max to it to get the "probability" of the hidden token to be equal to each of the tokens in the dictionary. Therefore, the model can be trained with cross-entropy loss as a supervised classification problem would be trained.

The GPT2 model for next token prediction architecture is schematically depicted in the figure 28. It consists of

- Learned token embeddings ($d_{=}50k$ - vocabulary size, $d = 768$);
- Absolute learned positional encoding;
- $L = 12$ transformer layers:
  - $h = 12$ head self-attention
  - Layer norm + skip connection

  - MLP with 1 hidden layer, applied token-wise. Usually, the activation taken is ReLU or GeLU: $gelu(x) = xC(x)$, where $C$ is the CDF of normal distribution;
  - Layer norm + skip connection;
- Unembedding layer and softmax.

# 13 Deep learning modus operandi

In this chapter we will discuss a collection of practical techniques and surprising observations that form what we can call the modus operandi of modern deep learning.

## 13.1 Techniques for Practical Deep Learning

### 13.1.1 Data Augmentation

The success of deep learning models is heavily dependent on the availability of massive datasets like ImageNet. However, in many domains, such as medicine, acquiring millions of samples is impossible. Data augmentation is a surprisingly effective strategy designed to overcome this limitation by artificially creating more training samples from the ones you already have. Consider image classification task. The core idea of data augmentation is to apply a series of transformations to an existing image, creating new variations that are still recognizable to a human as the original class (see example in Figure 29). This helps the model become more robust and less sensitive to irrelevant variations.

Figure 29: Example of transformation for data augmentation

Common techniques for image data include:

- Flipping an image horizontally;
- Randomly cropping different sections of the image;
- Changing contrast or colors to create slight variations in lighting and hue.

More advanced methods of deformation can also be used. While it may seem counter-intuitive that creating slightly modified copies of data would significantly improve performance, in practice, this technique is a cornerstone for training robust models on limited data.

### 13.1.2 Transfer Learning

Another technique for working on problems where large datasets aren't available uses the observation, that deep networks tend to learn useful features in the first layers, that can be useful for multiple "downstream" tasks.

This innate ability to learn a hierarchy of features can be illustrated by visualizing features that maximize the activation scores of the consecutive layers in a convolutional network as illustrated in figure 30. We can observe that early layers learn simple, small-scale features such as edges, corners, and color gradients. Intermediate layers compose these simple features into more complex and meaningful patterns, such as an eye or a nose. Finally, later layers assemble these larger components into complete objects, like the face of a person. This hierarchical composition, from simple to complex patterns, is a key reason why the features learned by these networks can be utilized across different tasks.

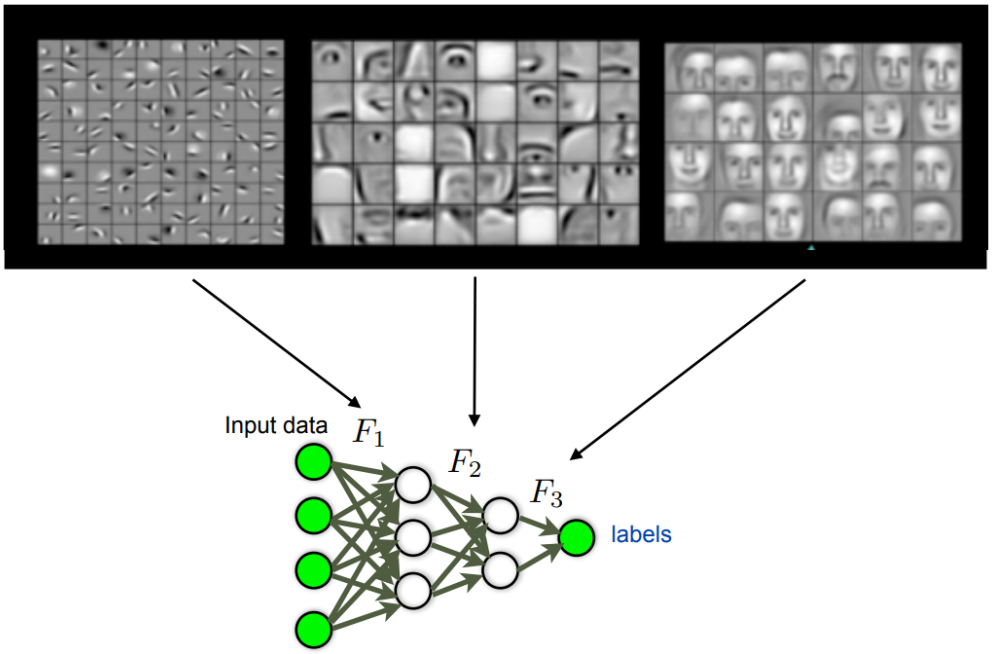



Figure 30: Features learned by a CNN.

The core principle of transfer learning is to take a network and its learned weights, which have been pre-trained on a very large and general dataset (like ImageNet for images or the wikipedia dump for text), and use it as a starting point for a new, more specific task that has a much smaller dataset. This leverages the general features learned from the large dataset, significantly reducing the need for new data. Two primary approaches are used:

- **Feature Extraction**. The weights of the early layers from the pre-trained model are "frozen" and treated as a fixed feature extractor. Only the final one or two layers are trained from scratch on the new, smaller dataset.
- **Fine-Tuning**. The entire network is initialized with the pre-trained weights, but all of the weights are updated during training on the new dataset, typically with a lower learning rate.
- **Low-rank Adaptation**. In cases, when finetuning all the weights is infeasible, one might add a low-rank perturbation to a weight matrix and then only update the elements of that perturbation, adjusting the learned function while updating only a small number of parameters.

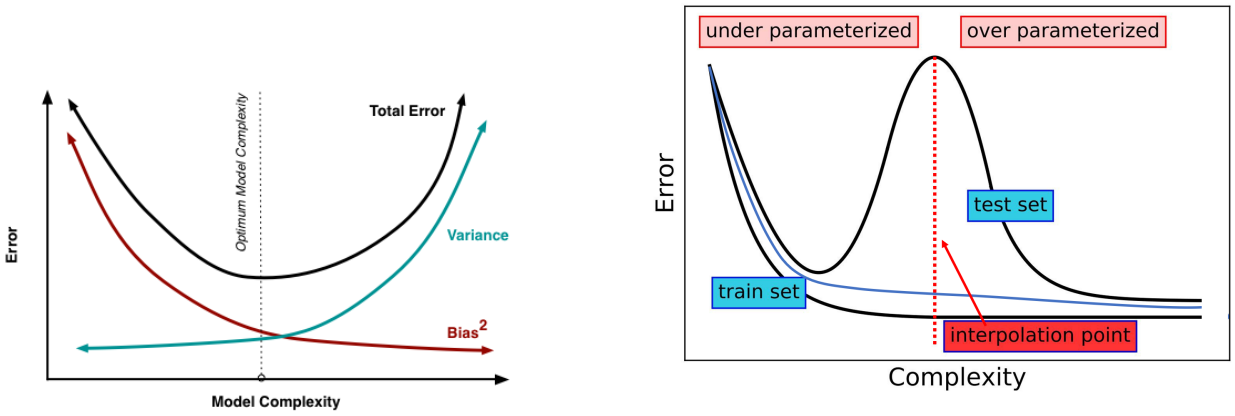
## 13.2 The "Bigger is Better" Paradigm

A defining characteristic of modern deep learning is the trend towards building increasingly large models, a practice supported by strong empirical evidence. Contrary to the classical learning theory, where over-parametrization leads to degraded test performance, deep models with vastly more parameters than training samples achieve lower error rates.

Classical machine learning assumed there is some optimal model complexity corresponding to some optimal number of model parameters, such that when the number of parameters becomes larger than that, the model overfits: it memorizes the training data but can't generalize to the unseen examples as illustrated in the Figure 31a.

However, with deep neural networks we observe a double descent curve (Figure 31b). In the beggining, we observe the same behavior of the test error, there is some number of parameters, after which the model begins to overfit. But the behavior changes, after the model reaches the *interpolation point*, where the training error reaches 0. Starting from this point, the training set can be perfectly interpolated by the model. The test error at this point is called the *interpolation peak*, and after this peak we observe the decrease in the test error when increasing model complexity.

Figure 32 showcase the 'double-descent behavior' of a neural network. The last panel, in particular, constitutes an example of neural network that works in the overparametrized (interpolation) regime.




(a) in classical machine learning theory.

(b) in deep neural networks.

Figure 31: Test error as a function of model complexity (number of parameters).

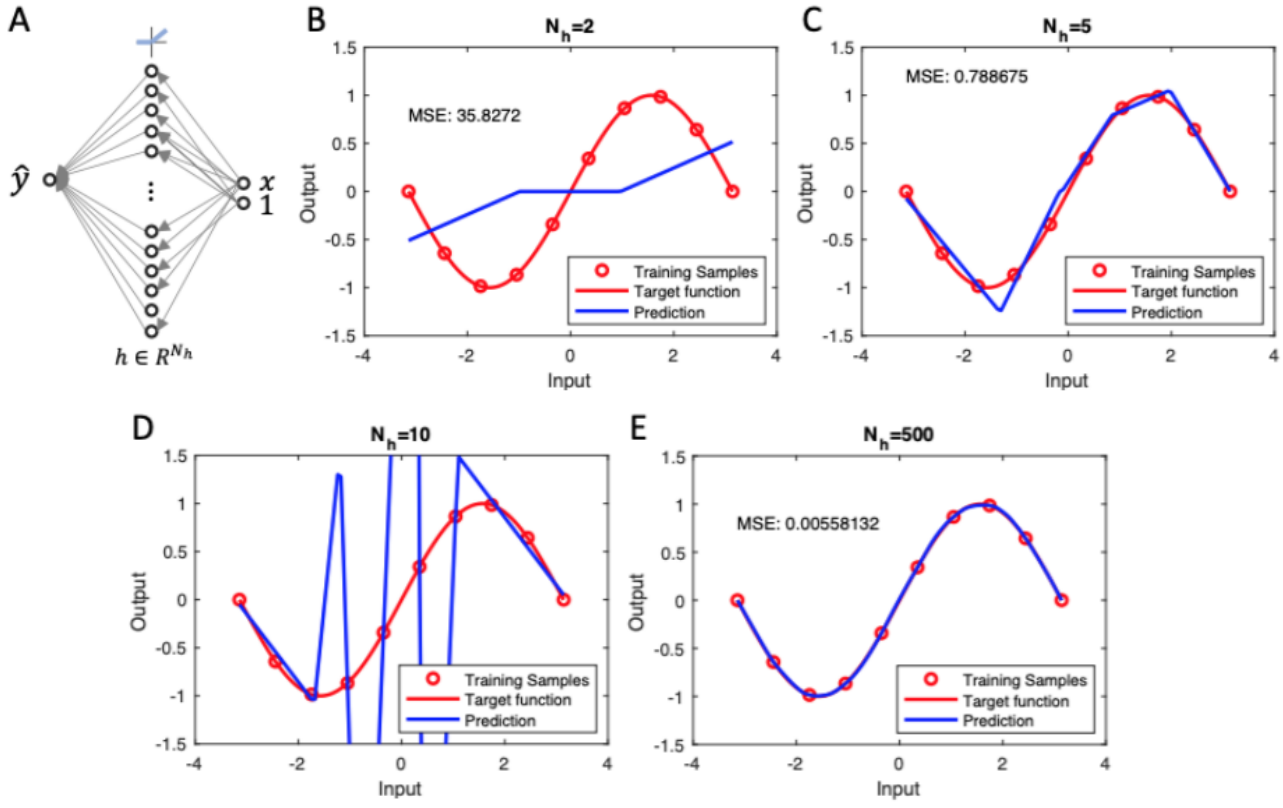



Figure 32: overparameterization and generalization (from a talk by Andrew Saxe). Sub-figure **(A)** illustrates a simple ReLU network with random first layer weights and trained second layer weights. The network receives a scalar input and bias term and is trained to minimise squared error on ten points (red circles) from a target sinusoid (red curve). In sub-figures **(B)** to **(E)**, the blue curves show example functions learned by networks with different numbers of hidden units $\mathbf{N_h} \in \{2, 5, 10, 500\}$. Networks in panel **(B)**, **(C)** and **(D)** show the standard progression from underfitting the training data to overfitting it in the underparameterized regime of the double-descent curve. However, the large network panel **(E)** generalizes the best. This network has 50 times more parameters than training examples but generalizes well. It illustrates the overparametrized regime of the double-descent curve.

### 13.2.1 Implicit regularization

A leading hypothesis for why overparameterized models generalize well lies in the concept of implicit regularization. When a model is overparameterized, there is no longer a single unique solution that minimizes the training loss. In fact, for tasks like linear regression, there exists an infinite number of parameter settings that achieve zero training loss. Then the quality of the predictions on the new samples depends on the particular solution found by the minimization algorithm that was used.

We already discussed in section 4.2 that in the simple case of linear regression, gradient descent with small weight initialization converges to the solution with the smallest norm, i.e. the model is implicitly regularized by the training procedure.

Extrapolating this to deep learning gives us the modern understanding of the problem. The loss landscape is not a bumpy surface with many isolated local minima. Instead, it is better pictured as a high-dimensional space containing vast, interconnected "canyons" of global minima. Many of this minima are not generalizing to the test set and can sometimes be reached as was shown in [7], but most of the time gradient descent algorithm leads the training to a "good" one. Researchers still don't have a full theoretical explanation for this behavior.

### 13.2.2 Adversarial examples

The landscape of infinite solutions is also the direct cause of a troubling consequence: the model's fragility to adversarial examples. Because the model has such high capacity to interpolate the training data perfectly, the decision boundaries it learns is complex and brittle. This property can be exploited to construct an adversarial example that will lead the model to predict incorrect labels.

An attack consists of adding a small, often human-imperceptible amount of specially crafted noise to an input, with the goal of completely changing the model's output (while a human would still give the same label to the sample). For example, a real-world stop sign, when a few specific stickers are applied, can be misinterpreted by a self-driving system as a "max speed 100" sign.

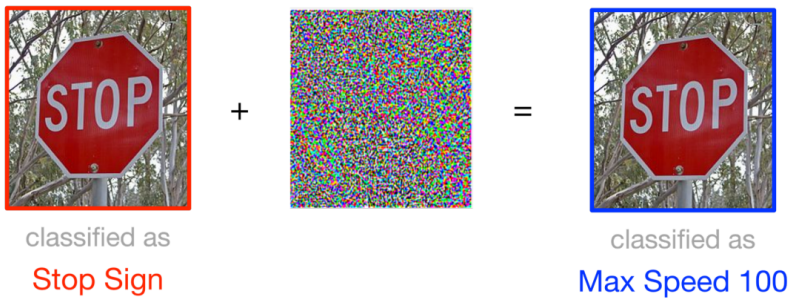



Figure 33: Adversarial example over a stop sign. In this particular case, even a small noise added to the sign picture may lead to a wrong classification from the recognition software of the car, leading to undesirable consequences such as a car crash.

This phenomenon reveals a fundamental lack of robustness in deep models. The very same overparameterization that allows the model to navigate a complex landscape to find a good solution also makes it vulnerable to tiny perturbations that push an input across a nearby, but incorrect, decision boundary.

## 13.3 Fighting overconfidence

When we train a model for classification, the outputs of the model usually can be interpreted as probabilities of the sample to belong to different classes. It should be noted that these numbers are just scores of the model and they might not reflect the real confidence of prediction. This leads to models sometimes being overconfident in the incorrect predictions, e.g. in the case of LLMs it leads to *hallucinations*, when a model confidently generates factually wrong answers.

To counteract this, the model should be calibrated, i.e. the confidence should be corresponding to actual quality of the predictions. One way to calibrate is by using a temperature parameter $T$. A model for classification outputs

scores $z_a$ for each class $a = 1, \dots, K$, that are then converted to probabilities using the softmax function

$$p_a(X) = \frac{e^{z_a(X)}}{\sum_b e^{z_b(X)}}.$$

Now we can introduce temperature $T$, to make this distribution more peaked for low temperature or more uniform for high temperature:

$$p_a(X, T) = \frac{e^{\frac{1}{T} z_a(X)}}{\sum_b \frac{1}{T} e^{z_b(X)}}.$$

And choose the value of $T$ that minimizes the *validation* loss:

$$\mathscr{L}(\{X_\mu, y_\mu\}_{\mu=1}^{n_{\text{val}}}) = -\frac{1}{n_{\text{val}}} \sum_{\mu=1}^{n_{\text{val}}} \sum_{a=1}^{K} y_{\mu,a} \log p_a(X_\mu, T).$$

This way we can adjust model's confidence after the training is already done.

# 14 Unsupervised Learning: Generative models

## 14.1 Auto-Encoders

### 14.1.1 Another view of PCA

Self-supervised learning, that we briefly discussed in the section 12.4, is a powerful technique, that allows us to use algorithms from supervised learning such as gradient descent in the settings where data has no explicit labels. The idea of self-supervision is to use parts of the input sample itself or some input augmentation as a label. By creating a supervised task from the unlabeled data, we can train a deep network to learn meaningful representations without human-provided annotations.

We already encountered this idea when discussing PCA in Chapter 6. Finding principal components can be formulated as a self-supervised task. By Young-Erkart theorem (Theorem 1), the matrices of $p$ first singular vectors are the solution to the following optimization problem:

$$\min_W \left[ \mathscr{L}(W) = \frac{1}{n} \sum_{\mu=1}^{n} \sum_{i=1}^{d} (X_{\mu i} - \sum_{a=1}^{p} w_{ia}^{(2)} \sum_j w_{aj}^{(1)} X_{\mu j})^2 \right]. \tag{14.1}$$

However, the same task can be considered as training a single hidden layer neural network with linear activation, hidden layer of width $p$ and the output dimension the same as the input dimension to reconstruct the inputs, i.e. given the sample $X_\mu$, we want to predict $X_\mu$, using minimization of the square loss.

### 14.1.2 Deep Auto-Encoders

We can generalize this concept by adding non-linear activation functions and multiple hidden layers to create a full autoencoder, depicted in Figure 34. This network has a symmetric structure. The first half, the Encoder, takes the high-dimensional input and maps it through one or more hidden layers to a low-dimensional central layer, often called the bottleneck. The second half, the Decoder, takes the low-dimensional representation from the bottleneck and attempts to reconstruct the original high-dimensional input.

The Encoder's job is to compress the input into a meaningful, low-dimensional representation. The Decoder's job is to decompress that representation back into the original data format. The entire network is trained to minimize the difference between the input and the reconstructed output:

$$\mathscr{L}(W) = \sum_{\mu=1}^{n} l(X_\mu; f_W(X_\mu)) = \sum_{\mu=1}^{n} \sum_{i=1}^{d} (X_{\mu i} - [f_W(X_\mu)]_i)^2 \tag{14.2}$$

where the input data $X_\mu$ play the role of the labels and $f_W(X_\mu)$ is the output of a multi-layer neural network.

This architecture has a dual utility:

- **Representation Learner**. The activations in the central bottleneck layer provide a learned, compressed representation of the input data. This representation can be used for downstream tasks.

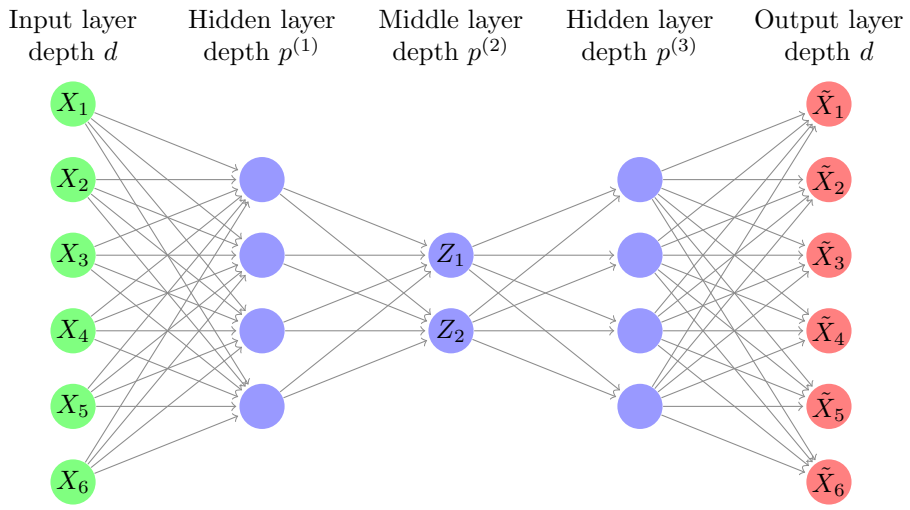



Figure 34: Example of an auto-encoder with the input dimension $d = 6$ and the middle layer dimension $p = p^{(2)} = 2$. The layers from the input layer up to the middle layer (included) form the Encoder, and the Decoder consists of all subsequent layers.

- **Generative Model** After training, we can detach the decoder. By feeding random noise (e.g., from a Gaussian distribution) into the bottleneck layer, the decoder can be used to generate new, synthetic data samples that resemble the original training data.

Moreover, auto-encoders can be used to remove the noise in an image or a signal. Using the corrupted images as inputs and original images as labels, we can train a model that learns to reconstruct the original data, and after training it can be used to denoise new samples. An example of the denoising auto-encoder outputs can be observed in Figure 35.

Figure 35: Effect of a denoising auto-encoder on old pictures.

## 14.2 Boltzmann Machine

General goal of generative modeling is to generate samples from the same distribution as the given training set. For example, text and image generation are generative task, when a model is given samples of text or images and is trained to produce similar outputs. In this section we are going to consider a special case of a generative model, called Boltzmann machine.

The training dataset for this model consists of $n$ binary strings $X_\mu \in \{\pm 1\}^d$. And our goal is to find such distribution $p(x)$, that the empirical mean and covariance of the given data is equal to the mean and covariance of this distribution:

$$\begin{aligned} \langle x_i \rangle_{p(x)} &= \frac{1}{n} \sum_{\mu=1}^{n} X_{\mu i}, \\ \langle x_i x_j \rangle_{p(x)} &= \frac{1}{n} \sum_{\mu=1}^{n} X_{\mu i} X_{\mu j} \end{aligned} \tag{14.3}$$

Then we consider strings sampled from this distribution to be similar to the real data, i.e. if at some position $k$ in the training dataset we only encountered $+1$, then in our generated examples we will also only have $+1$, since we need to enforce $\langle x_k \rangle_{p(x)} = \frac{1}{n}\sum_{\mu=1}^{n} X_{\mu k} = +1$.

Boltzmann machine is defined by a probability distribution:

$$p_{h,J}(x) = \frac{1}{Z} e^{\sum\limits_{i<j} J_{ij}x_i x_j + \sum\limits_{i=1}^{d} h_i x_i}, \tag{14.4}$$

with $x \in \{\pm 1\}^d$. We can recognize Boltzmann distribution for the Ising model with $J_{ij}$ being the interactions and $h_i$ being the fields at each spin.

When the correct couplings and fields are inferred, we can sample from this distribution by running Markov Chain Monte Carlo algorithm. Let us now discuss, why do we chose this type of probability distribution, and how to infer the values of $J_{ij}$ and $h_i$.

### 14.2.1 Maximum Entropy Principle

To justify the choice of a Boltzmann probability distribution among all the distributions that have required means $\langle x_i \rangle_{p(x)}$ and covariances $\langle x_i x_j \rangle_{p(x)}$, we utilize the Maximum Entropy Principle, which states that we should pick the distribution that maximizes the Shannon entropy.

The Shannon entropy quantifies the average level of uncertainty associated to the possible outcomes of a given random variable $x$ and given the probability distribution $p(x)$ it is defined as:

$$S = -\sum_{x} p(x) \log p(x)\,, \tag{14.5}$$

where the sum runs over all possible configurations of $x$ (i.e. over all $2^d$ possible strings of $\pm 1$ in our case).

Assume we have a set of observed quantities $f_a(x)$, with $a = 1, \ldots, m$ (the index $a$ here runs over the number of constraints we want to impose). These observables are the quantities that we want to conserve in the system:

$$\frac{1}{n}\sum_{\mu=1}^{n} f_a(X_\mu) = \langle f_a \rangle_{\text{emp}} = \langle f_a \rangle_{p(x)} = \sum_{x} f_a(x) p(x)\,. \tag{14.6}$$

Given this set of constraints, we then would like to find a function $p(x)$ that solves the following constrained maximization problem:

$$\begin{aligned} &\max_{p(x)} -\sum_{x} p(x)\log p(x), \\ &\text{s.t.} \\ &\langle f_a \rangle_{p(x)} = \langle f_a \rangle_{\text{emp}} \ \forall a = 1, \ldots, m; \\ &\sum_{x} p(x) = 1. \end{aligned} \tag{14.7}$$

The last constraint ensures that $p(x)$ is normalized to one, i.e. it is a the probability distribution.

To solve this problem, we use the method of Lagrange multipliers. First, we define the Lagrangian functional as the sum of the quantity that we aim to maximize (the Shannon Entropy in this specific case) and the set of constraints we aim to satisfy:

$$\mathscr{L}(p, \lambda, \gamma) = -\sum_{x} p(x)\log p(x) + \sum_{a=1}^{m} \lambda_a \Big( \langle f_a \rangle_{\text{emp}} - \sum_{x} f_a(x) p(x) \Big) + \gamma \big(1 - \sum_{x} p(x)\big)\,. \tag{14.8}$$

Having defined the Lagrangian, we take its derivatives to find the maximum:

$$\frac{\delta \mathscr{L}}{\delta p(y)} = 0 = -\log p(y) - 1 - \sum_{a=1}^{m} \lambda_a f_a(y) - \gamma \tag{14.9}$$

This will give rise to a Boltzmann-like type of measure:

$$p(x) = e^{-\gamma - 1} e^{-\sum\limits_{a=1}^{m} \lambda_a f_a(x)}. \tag{14.10}$$

Note that, Boltzmann distribution is obtained when we have constraint on the energy of the system: $E(X) = \text{const}$. We rename $\frac{1}{Z} = e^{-\gamma-1}$ as $\gamma$ is the multiplier that ensures the normalization of the probability distribution and denote $\beta = \lambda$.

$$p(X) = \frac{1}{Z} e^{-\beta E(X)} \,. \tag{14.11}$$

As we can see, the exponential form of the Boltzmann measure arises from the maximization of the Shannon entropy while keeping the energy fixed.

In the specific case of the Boltzmann Machine, we have more than one constraint, and instead of $f_a(x)$ we can write:

$$\begin{aligned} f_a(x) &= x_i, \qquad a = i = 1, \dots, d \,, \\ f_a(x) &= x_i x_j, \qquad a = ij = d+1, \dots, d(d+1)/2 \,, \end{aligned} \tag{14.12}$$

If we plug it in the solution of the maximization problem (14.10), and denote the Lagrange multipliers $\lambda_a = -h_a$ for $a = 1, \dots, d$ and $\lambda_a = -J_{ij}$ for $a = d+1, \dots, d(d+1)/2$, we retrieve the expression for the Boltzmann machine distribution (14.4).

### 14.2.2 How to train the Boltzmann Machine

Now we turn to the question of how to find the values of $h_i$ and $J_{ij}$.

If we relax the requirement $X \in \{\pm 1\}^{n \times d}$ and consider $X_{\mu i}$ to belong to real numbers, then $p(x)$ takes the form of a multivariate Gaussian distribution:

$$p(x) = \frac{1}{Z} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)} \tag{14.13}$$

where $Z$ is the normalization factor, namely $Z = (2\pi)^{\frac{d}{2}} (\det \Sigma)^{\frac{1}{2}}$, while the mean $\mu$ and the covariance $\Sigma$ of the multivariate Gaussian distribution have the following relationship with the Boltzmann machine parameters:

$$h = -\Sigma^{-1}\mu, \;\; J = -\Sigma^{-1} \,. \tag{14.14}$$

Then, for this distribution to satisfy the constraints (14.6), we should set

$$\begin{aligned} \mu_i &= \langle x_i \rangle_{p(x)} = \frac{1}{n} \sum_{\mu=1}^{n} X_{\mu i}, \\ \Sigma_{ij} &= \langle x_i x_j \rangle_{p(x)} - \langle x_i \rangle_{p(x)} \langle x_j \rangle_{p(x)} = \frac{1}{n} \sum_{\mu=1}^{n} X_{\mu i} X_{\mu j} - (\frac{1}{n} \sum_{\mu=1}^{n} X_{\mu i})(\frac{1}{n} \sum_{\mu=1}^{n} X_{\mu j}). \end{aligned} \tag{14.15}$$

Although this method seems simple and naive, we can still obtain quite a good approximation for both the couplings $J$ and the fields $h$. This method is often used in the direct coupling analysis of protein sequences (see e.g. the work of De Los Rios and Goloubinoff at EPFL [8]).

Another training strategy is utilizing the maximum likelihood principle that we have seen earlier in the course. In this case, the Boltzmann measure quantifies the probability of having observed a certain configuration, given the couplings $J$ and the magnetic fields $h$ and we can write the likelihood:

$$\mathscr{L}(X|\theta) = \prod_{\mu=1}^{n} p_{h,J}(X_\mu) = \frac{1}{Z(\theta)^n} e^{\sum_{\mu=1}^{n} E_\theta(X_\mu)} \,, \tag{14.16}$$

where $\theta$ represents the set of parameters $(h, J)$ we aim to infer, and the energy is given by $E_\theta(x) = \sum_{i<j} J_{ij} x_i x_j + \sum_i h_i x_i$.

We can then maximize the logarithm of the likelihood:

$$\max_\theta \left( \log \mathscr{L}(X|\theta) \right) = \max_\theta \left( \frac{1}{n} \sum_{\mu=1}^{n} E_\theta(X_\mu) - \log Z(\theta) \right). \tag{14.17}$$

At the maximum we have:

$$\frac{\partial \log(\mathscr{L}(X|\theta))}{\partial \theta_a} = 0 = \left\langle \frac{\partial E_\theta(X)}{\partial \theta_a} \right\rangle_{\text{emp}} - \frac{1}{Z(\theta)} \sum_x e^{E_\theta(x)} \frac{\partial E_\theta(x)}{\partial \theta_a} = \left\langle \frac{\partial E_\theta(X)}{\partial \theta_a} \right\rangle_{\text{emp}} - \langle \frac{\partial E_\theta(x)}{\partial \theta_a} \rangle_{p_{h,J}(x)}. \tag{14.18}$$

Given the dependency of the energy function $E_\theta$ on the couplings $J_{ij}$ and the magnetic field $h_i$, we have

$$\frac{\partial E_\theta(X)}{\partial J_{ij}} = x_i x_j,$$
$$\frac{\partial E_\theta(X)}{\partial h_i} = x_i,$$

and the maximum conditions can be rewritten as:

- $\langle X_i \rangle_{\text{emp}} = \langle x_i \rangle_{p_{h,J}(x)}$ ,
- $\langle X_i X_j \rangle_{\text{emp}} = \langle x_i x_j \rangle_{p_{h,J}(x)}$ ,

which are exactly the constraints we aim to satisfy.

To achieve maximum log-likelihood algorithmically, we can use gradient descent:

$$\theta^{t+1} = \theta^t - \gamma \frac{\partial(-\log(\mathscr{L}(X|\theta)))}{\partial \theta}\Big|_{\theta^t} \tag{14.19}$$

where $\gamma$ represents the learning rate. If we rewrite the gradient descent update in terms of $h$ and $J$, we then get the following update rules:

- $h_i^{t+1} = h_i^t + \gamma[\langle X_i \rangle_{\text{emp}} - \langle x_i \rangle_{p_{h^t,J^t}(x)}]$ ,
- $J_{ij}^{t+1} = J_{ij}^t + \gamma[\langle X_i X_j \rangle_{\text{emp}} - \langle x_i x_j \rangle_{p_{h^t,J^t}(x)}]$.

The values of empirical means can be easily estimated directly from the data. However, to compute the expectations $\langle x \rangle_{p(x)}$ over the model at fixed $J$ and $h$, we need to run Monte Carlo Markov Chain to estimate the expectations over the Boltzmann measure. So, we see that the process of training Boltzmann Machine is combining several methods we learned before to be able to infer the parameters of the Boltzmann distribution, which we can then use to produce new samples distributed according to the same Boltzmann measure.

## 14.3 Adding hidden variables to the Boltzmann Machine

Let's define *visible* and *hidden variables* respectively, as:

$$x_i \in \{\pm 1\}, \quad i = 1, ..., d \tag{14.20}$$
$$u_i \in \{\pm 1\}, \quad i = 1, ..., p. \tag{14.21}$$

We can now postulate the following probability distribution:

$$P_\theta(x) = \frac{1}{Z} \sum_{\{u_i\}_{i=1}^p} e^{E_\theta(\{x_i\}_{i=1}^d, \{u_i\}_{i=1}^p)}, \tag{14.22}$$

where

$$E_\theta(x,u) = \sum_{i<j} J_{ij} x_i x_j + \sum_{i=1}^d h_i x_i + \sum_{i,j} \tilde{J}_{ij} x_i u_j + \sum_{i<j} \tilde{\tilde{J}}_{ij} u_i u_j + \sum_{i=i}^p \tilde{h}_i u_i, \tag{14.23}$$

and $\theta = \{J_{ij}, \tilde{J}_{ij}, \tilde{\tilde{J}}_{ij}, h_i, \tilde{h}_i\}$. Since equation (14.22) depends on more parameters than the regular Boltzmann machine, it is more general and can represent more complex probability distributions.

To train the Boltzmann machine with hidden variables, we follow the same steps as before. We begin by writing the maximum likelihood:

$$\begin{aligned} \mathscr{L}(\theta) &= \prod_{\mu=1}^n P_\theta(X_\mu) \\ &= \prod_{\mu=1}^n \left[ \frac{1}{Z(\theta)} \sum_u e^{E_\theta(X_\mu, u)} \right] \\ \frac{1}{n} \log \mathscr{L}(\theta) &= \frac{1}{n} \sum_{\mu=1}^n \log \left( \sum_u e^{E_\theta(X_\mu, u)} \right) - \log Z(\theta). \end{aligned} \tag{14.24}$$

Taking the gradient with respect to a single parameter $\theta$, e.g., $\theta$ could be $J_{ij}$ or $\tilde{h}_i$, we obtain:

$$\begin{aligned}\frac{1}{n}\frac{\partial \log \mathscr{L}(\theta)}{\partial \theta} &= \frac{1}{n}\sum_{\mu=1}^{n}\frac{\sum_u \frac{\partial E_\theta}{\partial \theta} e^{E_\theta(X_\mu,u)}}{\sum_u e^{E_\theta(X_\mu,u)}} - \frac{\partial}{\partial\theta}\log Z_\theta \\ &= \underbrace{\frac{1}{n}\sum_{\mu=1}^{n}\left\langle \frac{\partial E_\theta(X_\mu,u)}{\partial\theta}\right\rangle_{P_\theta(X=X_\mu,u)}}_{\text{fix } X_\mu\text{, sample } u} - \underbrace{\left\langle \frac{\partial E_\theta(x,u)}{\partial\theta}\right\rangle_{P_\theta(x,u)}}_{\text{sample both } x \text{ and } u}.\end{aligned} \tag{14.25}$$

The average where we fix $X_\mu$ is solely with respect to the hidden variables $u_i$, and is often referred to as the *clamped average*. The update equation for $\theta$ becomes:

$$\theta^{t+1} = \theta^t + \gamma\left[\left\langle \frac{\partial E_\theta}{\partial\theta}\right\rangle_{\text{clamped}} - \left\langle \frac{\partial E_\theta}{\partial\theta}\right\rangle_{P_\theta(x,u)}\right]. \tag{14.26}$$

In practice, however, training these models is quite cumbersome, and therefore they are not very used in practice.

### 14.3.1 Restricted Boltzmann Machine (RBM)

The most widely used version of the Boltzmann Machine with hidden units is the *Restricted Boltzmann Machine* (RBM). In this special case, the energy takes into account only interactions between the visible and hidden units:

$$E_\theta(x,u) = \sum_{ij} J_{ij}x_i u_j + \sum_{i=1}^{d} h_i x_i + \sum_{i=1}^{p}\tilde{h}_i u_i. \tag{14.27}$$

In particular, there are no visible-visible nor hidden-hidden interactions.

The clamped average for the RBM can be written explicitly in closed form (there is no need of MCMC). For instance, if $\theta = J_{ij}$

$$\langle X_{\mu j}u_i\rangle_{\text{clamped}} = X_{\mu j}\tanh\left(\sum_{l=1}^{n} X_{\mu l}J_{li} + h_i\right) \tag{14.28}$$

which corresponds to a model of non-interacting spins. The clamped average is often called *positive gradient*. The model average $\langle x_i u_j\rangle$, or *negative gradient*, can be efficiently computed by using MCMC sampling:

1. $u$ is sampled with $x$ fixed.
2. $x$ is sampled with $u$ fixed.
3. Repeat.

All in all,

$$J_{ij}^{t+1} = J_{ij}^t + \gamma[\text{Positive Gradient} - \text{Negative Gradient}]. \tag{14.29}$$

This model works nicely and is very used in practice.

### 14.3.2 Deep RBM

We can also add layers of hidden units to the RBM. In the three-layers case, the energy reads

$$E_\theta(x,u^{(1)},u^{(2)}) = \sum_{ij} J_{ij}^{(1)}x_i u_i^{(1)} + \sum_{ij} J_{ij}^{(2)}u_i^{(1)}u_j^{(2)} + \sum_{i}^{d} h_i x_i + \sum_i h_i^{(1)}u_i^{(1)} + \sum_i h_i^{(2)}u_i^{(2)} \tag{14.30}$$

where the interactions are between the first and second layer, and the second and third layer. However, these deeper models see limited use nowadays.

## 14.4 Autoregressive models

We call a model autoregressive when we use a sequence of values to predict the next element in this sequence. For example, if we try to predict the weather based on the data from the previous week or the next word based on the previous words in a sentence.

To perform autoregressive modeling, we augment the data using causal masking. Having a sample $X_{\mu,i}$ for $i = 1, \dots, T$, we introduce a causal mask:

$$\tilde{X}_\mu^{(t)} = (X_{\mu,1}, \dots, X_{\mu,t-1}, 0, \dots, 0).$$

After this data augmentation, we aim to predict $\hat{X}_{\mu,t} = f_W(\tilde{X}_\mu^{(t)})$. Then the loss is defined as

$$\mathscr{L}(W) = \frac{1}{n} \sum_{\mu=1}^{n} \sum_{t=1}^{T} l(X_{\mu,t}, f_W(\tilde{X}_\mu^{(t)})),$$

where $l$ is the sample-wise loss function, e.g. square loss $l(X_{\mu,t}, \hat{X}_{\mu,t}) = (X_{\mu,t} - \hat{X}_{\mu,t})^2$

The architecture we choose to solve this task can be arbitrary, here we will consider a simple fully-connected network with one hidden layer:

$$\mathscr{L}(W) = \frac{1}{n} \sum_{\mu=1}^{n} \sum_{t=1}^{T} (X_{\mu,t} - \sum_{a=1}^{p} w_a^{(2)} \phi(\sum_{i=1}^{t-1} X_{\mu i} w_{ai}^{(1)}))^2.$$

Compare it to a classical autoencoder with one hidden layer:

$$\mathscr{L}(W) = \frac{1}{n} \sum_{\mu=1}^{n} \sum_{t=1}^{T} (X_{\mu,t} - \sum_{a=1}^{p} w_{ta}^{(2)} \phi(\sum_{i=1}^{T} X_{\mu i} w_{ai}^{(1)}))^2.$$

In the autoregressive model, the meaning of the sum over $t$ from 1 to $T$ changes: now it represents different samples, while in the autoencoder architecture it represented different components of one input sample. Additionally, $p$ is not required to be smaller than $T$, because in the case of autoregressive model we can not "cheat" by using identity to perfectly reconstruct the input.

### 14.4.1 How to generate sequences?

When we trained our model, we can use it to generate new sequences. To do this, we first postulate the conditional probability of the next token:

$$P_w(X_t | X_{<t}) = \frac{1}{Z} e^{-l(X_t, f_W(\tilde{X}^{(t)}))}. \tag{14.31}$$

Notice that the probability of the full sequence can be written as the product of this conditional probabilities:

$$p(X_1, \dots, X_T) = \prod_{t=1}^{T} P(X_t | X_{<t}).$$

E.g. for square loss, we retrieve Gaussian distribution $P(X_t | X_{<t}) = \frac{1}{\sqrt{2\pi}} e^{-1/2(X_t - f_W(X_{<t}))^2}$.

Now, to generate a new sequence we do the following:

1. Start with an empty vector;
2. For $t = 1, \dots, T$, sample $X_t \sim P(\cdot | X_{<t})$;
3. Append $X_t$ to the sequence;

When solving the **next token prediction** task, e.g. when an LLM is used, the same idea is applied, but elements of the sequence $X_t$ become embedding vectors instead of scalars. And since the tokens come from a discrete set, training is done for classification, so cross-entropy loss $l$ is used.

## 14.5 Diffusion models

Diffusion models are another type of generative models. They use a similar idea of slightly changing the input data and trying to predict the original sample. We compare the two paradigms in the Table 2

| Model | Data augmentation | Training | Generation |
|---|---|---|---|
| Autoregressive | mask | predict the masked token | sequentially unmask from empty vector |
| Diffusion | add noise | remove the noise | sequentially denoise from pure noise |

Table 2: Comparison between autoregressive models and diffusion models.

We can apply the diffusion modeling principle to any architecture of the model $f_W$. Usually, the architecture of a diffusion model for image generation such as DALL-E or Midjourney involves using a convolutional network, but it can also be used with a simple fully-connected network. We have a sample $X_\mu \in \mathbb{R}^d$ which we augment with noise $\tilde{X}_\mu = X_\mu + \Delta\epsilon$, where $\epsilon \sim \mathcal{N}(0, I_d)$. Then the network is trained to restore the original sample:

$$\mathscr{L}(W) = \mathbb{E}_\epsilon \frac{1}{n} \sum_{\mu=1}^{n} \sum_{i=1}^{d} (X_{\mu i} - [f_W(\tilde{X}_\mu)]_i)^2. \tag{14.32}$$

Notice, that we take the expected value over the noise, because we do not want to fit the model to the particular noise that was used to augment the data, but rather to denoise from any possible realization of noisy sample. To approximate this loss during training with SGD, we should generate new noise to add to the sample at every step when we use the sample $X_\mu$ during SGD training.

### 14.5.1 How to generate new samples?

The generation procedure is inspired by the following process. We have a sample $X_0 \sim P_0$ from the target distribution at time 0, and a random gaussian noise $Z$ at time 1. And there is a process that interpolates between them at times from $t \in (0, 1)$:

$$y(t) = \alpha(t)X_0 + \beta(t)Z, \tag{14.33}$$

such that $\alpha(0) = \beta(1) = 1$ and $\alpha(1) = \beta(0) = 0$. For example $\alpha = 1 - t$, $\beta(t) = t$.

Now, if we don't know the target distribution, but we know the expected value $\mathbb{E}(X_0|X_t)$, we can use it to reverse this process from $t = 1$ to $t = 0$ following the procedure listed in the listing 4.

**Algorithm 4** Diffusion reverse process

**Require:** $\alpha(t) : [0, 1] \to \mathbb{R}, \beta(t) : [0, 1] \to \mathbb{R}, \delta > 0$
  $X_{t=1} = Z \sim \mathcal{N}(0, I_d)$
  $l \leftarrow 1$
  $t \leftarrow 1$
  **repeat**
    $\hat{X}_0 \leftarrow \mathbb{E}(X_0|X_t) = f_W(X_t)$
    $X_{t-\delta} \leftarrow X_t(1 - \delta\frac{\dot{\beta}(t)}{\beta(t)}) - \hat{X}_0\delta(\dot{\alpha} - \frac{\dot{\alpha}(t)}{\beta(t)})$
    $t \leftarrow 1 - l\delta$
    $l \leftarrow l + 1$
  **until** $t > 0$
  **return** $X_{t=0}$

Notice, that we use a neural network $f_W(X_t)$ to estimate the value of $\mathbb{E}_{X_0}(X_0|X_t)$. We augment the data using the forward process:

$$\tilde{X}_\mu^{(t)} = \alpha(t)X_\mu + \beta(t)Z.$$

And then minimize the loss:

$$\mathscr{L}(W) = \mathbb{E}_{t\sim\text{Unif}[0;1]}\mathbb{E}_Z \frac{1}{n} \sum_{\mu=1}^{n} \sum_{i=1}^{d} (X_{\mu i} - [f_W(\tilde{X}_\mu^{(t)})]_i)^2. \tag{14.34}$$

### 14.5.2 The differential equation

The formula for the update in the algorithm 4 comes from the following observation. The forward process can be written as

$$\rho(y,t) = \mathbb{E}_{X_0,Z}\delta(y - (\alpha(t)X_0 + \beta(t)Z)). \tag{14.35}$$

And it can be shown that $\rho$ satisfies the followinf PDE

$$\partial_t \rho(y,t) + \nabla_y(b(y,t)\rho(y,t)) = 0, \tag{14.36}$$

where $b(y,t) = \mathbb{E}_{X_0,Z}(\partial_t y(t)|y(t) = y)$ is a velocity field.

If we know $b(y,t)$, then the solution of $\frac{d\tilde{y}}{dt} = b(y,t)$ with the initial conditions $\tilde{y}_{t=1} \sim \mathcal{N}(0, I_d)$ has the same distribution as $y(t)$. So, we are interested in retrieving the value of $\tilde{y}(0)$ which will be a sample from the target distribution.

The discretization of this process is given by

$$\tilde{y}_{t-\delta} = \tilde{y}_t - \delta b(\tilde{y}_t, t). \tag{14.37}$$

Notice that the velocity field can be expressed as

$$b(y,t) = \mathbb{E}_{X_0,Z}(\dot{\alpha}(t)X_0 + \dot{\beta}(t)Z|y(t)) = \mathbb{E}_{X_0,Z}(\dot{\alpha}(t)X_0 + \dot{\beta}(t)(\frac{y(t) - \alpha(t)X_0}{\beta(t)})|y(t)) = \frac{\dot{\beta}}{\beta}y(t) + (\frac{\dot{\beta}}{\beta}\alpha + \dot{\alpha})\mathbb{E}(X_0|y(t)). \tag{14.38}$$

By plugging it in the equation (14.37), we obtain exactly the expression used by the algorithm 4.

The most complicated part to estimate in this equation is the expected value $\mathbb{E}(X_0|y(t))$ and we use a neural network to estimate it.

## 14.6 Generative adversarial networks

The idea behind *Generative Adversarial Networks* (GANs) is slightly different from previously discussed approaches. It uses the idea, that we can use a neural network, called the discriminator, to distinguish between the generated samples and the real data and we can train a neural network, called the generator, to generate artificial data in such a way that discriminator can not distinguish true data from the generated.

**G : Generator neural network** This neural network $f_G$ with weights $W_G$ takes a random input $Z \in \mathbb{R}^d$, and generates the output $\widetilde{X}$:

$$\widetilde{X} = f_G(W_G, Z)\,. \tag{14.39}$$

**D : Discriminator neural network** This neural network $f_D$ with weights $W_D$ takes inputs that can be either a real sample $X_\mu$ or a sample $\widetilde{X}_\mu$ generated by the generator networks. Labels are assigned depending on the origin of the samples, i.e., $y_\mu = +1$ for the true data and $y_\mu = -1$ for the generated data. The discriminator returns the output:

$$f_D(W_D, (X, \widetilde{X})) \tag{14.40}$$

with $(X, \widetilde{X})$ the concatenated real and generated data.

**Training a GAN** In order to train a GAN, we introduce the loss function :

$$\mathscr{L}(W_D, W_G) = \frac{1}{2n}\sum_{\mu=1}^{2n} l(y_\mu;\ f_D(W_D, (X, \widetilde{X}))) = \frac{1}{2n}\sum_{\mu=1}^{2n} l(y_\mu;\ f_D(W_D;\ (X, f_G(W_G, z_\mu))))\,. \tag{14.41}$$

In order to optimize the performance of the discriminator, we minimize the loss function with respect to its weights $W_D$. Moreover, since the generator is trying to make the task harder for the discriminator, we maximize the loss function with respect to $W_G$, obtaining the following *max-min problem* formulation:

$$\max_{W_G}\min_{W_D} \mathscr{L}(W_D, W_G). \tag{14.42}$$

The details of this optimization procedure can be daunting, and often there are convergence problems. Nevertheless, there are many successful examples of the use of GANs. For instance, these models can generate impressive artificial pictures of people faces; with only a visual inspection, we are not able to tell that they were generated by an algorithm. Figures 36 and 37 represents two non-existing people artificially created by a GAN.

Figure 36: Picture generated by GANs - Example 1

Figure 37: Picture generated by GANs - Example 2

## Acknowledgements

A big thanks both to the TA's of the initial 2021 lecture, Frederica Gerace, Giovanni Piccioli and Alessandro Favero; and to the numerous students of this lecture who wrote the backbone of these lecture notes:
Tibke, Toffenetti, Torre, Verdier, Sinibaldi Alessandro, Nikunj Sangwan, Lana Maria Smith, Nolan James Sandgathe, Alessandro Salvatore, Soutter Jacques Bernard Jean, Peng Kanlai, Pérez Ortiz Rodrigo, Pohle Paul Timo, Rey Anthony Bernard, Rodriguez Mateos Borja, Roy Camille, René Charles, Mitais Charles Edmond Alfred, Moawad Alix Marie Gabrielle, Monnet Nathan Jean, Niggli Loïs, Norberg Astrid Mona Joséphine, Parusa Gian, Michalski Elisa Magdalena, Mejean Maïlys Joséphine, Méan Baptiste Maurice, Mauron Linda, Massard Fanny, Martres Delphine Renée Evelyne, Marchand Charles Edouard Eliott Nils, Malanyuk Oleg, Maier Aude, Maier Antoine, Linteau David, Koller Cédric Xavier, Kuang Luochen, Leontsinis Matthew James, Libardi Gabriele, Linder Louis, Kashko Pavlo, Karlsson Hannes Georg, Kaloyannis Panagiotis Stilianos, Jeandupeux Zoé, Jaubert Timothée Benjamin Antoine, He Shengyu, Haji Mirsaeedi Sayed Mohammad, Golomer Bastien Olivier Yves Francis, Gollerthan Tim, Griffon Amaury Brice Marie, Girod Alexis Théo Allan, Giriens Matthieu Philippe, Galletti Chiara, Forster Benjamin John Morcombe, Ferrari Filippo, Farchy Tobias Thagi Mckee, Ernst Samuel, Dufour Charles François Paul Pierre, Håvard Falch, De Schoulepnikoff Paulin Jean, Delévaux David, De Lucca Brenno Jason Sanzio Peter, Dardano Thomas, Cucurachi Daniele, Claudon Baptiste Paul François, Carranza I Botey Ester, Brown Steven Sinclair, Briand Guillaume Miro André, Dardano Thomas, Cucurachi Daniele, Claudon Baptiste Paul François, Carranza I Botey Ester, Brown Steven Sinclair, Briand Guillaume Miro André, Brenna Greta Yin-Sang, Boucard Manon Nathalie Laura, Bolognini Gaia Stella, Bianchi Yannick, Bergerioux Hugo Georges Henri, Azzoug Nour, Astruc Lopez Alejandro, Arbez Hugo Paul, Aragon Antoine Louis Marie, and Abdallah Mohamed Samy Mohamed Tawfik.

# References


[1] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, pp. 1–124, May 2019.

[2] (). "Iris flower dataset wikipedia page," [Online]. Available: https://en.wikipedia.org/wiki/Iris_flower_data_set.

[3] J. Novembre, T. Johnson, K. Bryc, Z. Kutalik, A. R. Boyko, A. Auton, A. Indap, K. S. King, S. Bergmann, M. R. Nelson, *et al.*, "Genes mirror geography within europe," *Nature*, vol. 456, no. 7218, pp. 98–101, 2008.

[4] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Advances in Neural Information Processing Systems*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., vol. 20, Curran Associates, Inc., 2007. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2007/file/013a006f03dbc5392effeb8f18fda755-Paper.pdf.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2017. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[6] E. Voita, *NLP Course For You*, Sep. 2020. [Online]. Available: https://lena-voita.github.io/nlp_course.html.

[7] S. Liu, D. S. Papailiopoulos, and D. Achlioptas, "Bad global minima exist and SGD can reach them," *CoRR*, vol. abs/1906.02613, 2019. arXiv: 1906.02613. [Online]. Available: http://arxiv.org/abs/1906.02613.

[8] P. De Los Rios and P. Goloubinoff, "Chaperoning protein evolution," *Nature Chemical Biology*, vol. 8, 2012.