Contents

Chapter 1

Introduction

Machine learning is used for problems that are difficult to solve with deterministic algorithms:

- Classification problems
- Image/Audio recognition
- Anomaly detection
- Generative AI
- ...

1.1 Approach

The general approach to solving problems with machine learning includes:

- Define a **model** specific to the problem to be solved that depends on the values of its **parameters**
- Define an **error** measure to evaluate the model
- Tune the model's parameters in order to minimize the error on the **training set**

Example 1.1.1 (Regression)
You have some points on a plane, the goal si to fit a line through them. Let's follow the steps defined above:

- First of all, we need to define the model. In this case, the model is the general equation of the line we want to create. For instance, if we choose a linear model the equation would be $y = ax + b$, where a, b are the parameters.
- To decide if a line is better than another, we need a loss function like the mean square error.
- Now we need to tune our model (line) by using the training data (points) and setting the parameters to minimize the error.

This way, if our model choice was correct and we had enough useful training data, we could be fairly sure that even other points not included in the training data will follow the line created by the model.

So Machine Learning is basically an **optimization problem**, where the solution isn't given in analytical form but aproximated by using iterative techniques.

1.2 Differences

There are different types of learning aproaches

- **Supervised**: needs inputs and associated outputs (labels) - classification, regression
- **Unsupervised**: only needs inputs - clustering, component analysis, autoencoding
- **Reinforcement**: works with actions and rewards - long term gains, model-free planning

There are also different types of ways to define models

- Decison trees
- Linear models
- Neural networks
- ...

And different types of error (loss) functions

- Mean Squared Errors
- Logistic
- Cross entropy
- ...

1.3 Features

Definition 1.3.1: Feature

Any information relative to a datum that describes one of its relevant properties. They're the input to the learning process.

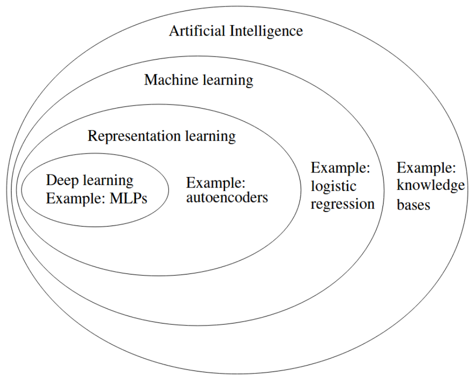
The coice of good features is extremely important for the learning process, and requires good domain knowledge.

Example 1.3.1 (Choosing features)
Medical diagnosis: symptoms, patient conditions, medical records, exam results
User profiling: demographic data, personal interests, social communities, life style
Weather forcasting: temperature, humidity, pressure, rain, wind

1.4 Deep learning

It's the modern approach. Raw data is supplied to the model, whose job it is to find good features itself.
Deep learning is implemented through **Neural Networks**, which are called *deep* when consisting of multiple hidden layers, each computing new features.

1.5 Overview of different types of AI



- **Knowledge-based systems**: take an expert, ask him how he solves a problem and try to mimic his approach by means of logical rules
- **Traditional Machine Learning**: the expert only tells us the important features, the machine learns the mapping
- **Deep Learning**: no more expert :(

Chapter 2

Decision Trees

Let's start with training set:

Definition 2.0.1: Training set

is defined training set a *set of examples*, where: $\langle x^{(i)}, y^{(i)} \rangle$ where:

- i is the instance of the example
- $x^{(i)} \in X$ is the set of *input*
- $y^{(i)} \in Y$ is the set of *output*

the problem of machine learning is to find a function $h : X \to Y$ that approximates the real function $f : X \to Y$. We have two types of problems:

- **Classification**: Y is a discrete set of values (e.g. $\{0, 1\}$)
- **Regression**: Y is a continuous set of values (e.g. $\mathbb{R}$)

2.1 Hypothesis space

In machine learning the *hypothesis space* H is defined as the set of all possible functions that can be used to approximate the real function $f : X \to Y$. Formally:

Definition 2.1.1: Hypothesis space

A hypothesis space H is defined as the set: $H = \{h | h : X \to Y\}$ where:

- h is a function (hypothesis) that maps input X to output Y
- X the input space (features, domain of data).
- Y the output space (labels, range of data).
- $|H|$ is the size of the hypothesis space (number of possible hypotheses)

this let us to define the model:

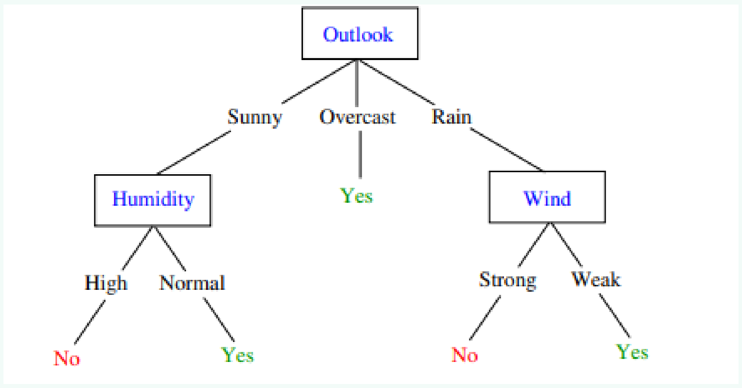
Definition 2.1.2: Model

A model is a way to compute a function $h \in H$ from the training set.

Example 2.1.1 (Decision tree)

A good day to play tennis? Our function F is:

$$\mathbb{F} : Outlook \times Humidity \times Wind \times Temp \longrightarrow PlayTennis?$$

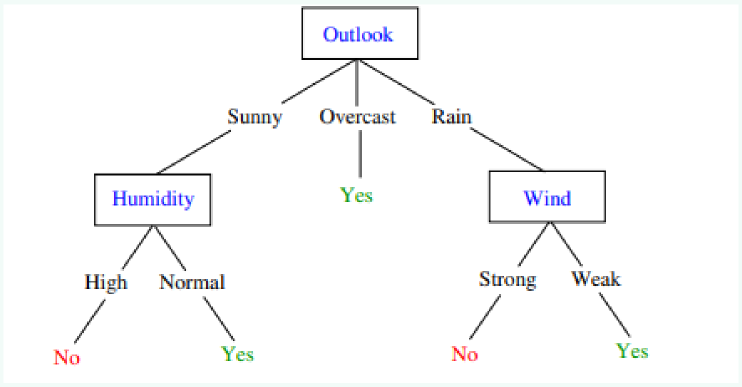


where:

- $Outlook \in \{Sunny, Overcast, Rain\}$
- $Humidity \in \{High, Normal\}$
- $Wind \in \{Weak, Strong\}$
- $PlayTennis? \in \{Yes, No\}$

Every node tests an attribute. Each branch corresponds to one of the possible values for that attribute. Each leaf node assigns a classification (Yes or No), in other words predicts the answer Y.
The prblem configurationg is the following:

- X is the set of all possible $x \in X$ that corresponds to a vector of attributes $(Outlook, Humidity, Wind, Temp)$
- Target function $f : X \longrightarrow Y$ is the function that maps the attributes to the target variable $PlayTennis?$ (booleans)
- Hypothesis space $H = \{h | h : X \longrightarrow Y\}$ is the set of all possible decision trees that can be constructed using the attributes in X to predict the target variable Y



2.1.1 Top-down inductive construction

Let $X = X_1 \times X_2 \cdots \times X_n$ where $X_i = \{\text{Truee}, \text{False}\}$
Can we represent, for instance, $Y = X_2 \wedge X_5$? or $Y = X_2 \wedge X_5 \vee (\neg X_3) \wedge X_4 \wedge X_1$?
and:

- do we have a decision tree for each h in the space hypothesis?
- if the tree exists, is it unique?
- if it is not unique, do we have a preference?

Theorem 2.1.1 Basta - Bonzo

Main loop:

- **Pick the "best" attribute X_i**: At the current node, choose which featue/attribute will bwst split the training data.

  Best means: the attribute that gives the most information gain
- **Create a child node for each possible value of X_i**: for instance if attribute is "weather" with values "sunny", "rainy", "overcast", create three child nodes.
- **check if all examples in the chils node are pure**: if all examples belong to the same class (e.g., all "yes" or all "no"), make that node a leaf node with that class label. If not repeat the process recursively for each child node.

2.2 Entropy

Definition 2.2.1: Entropy

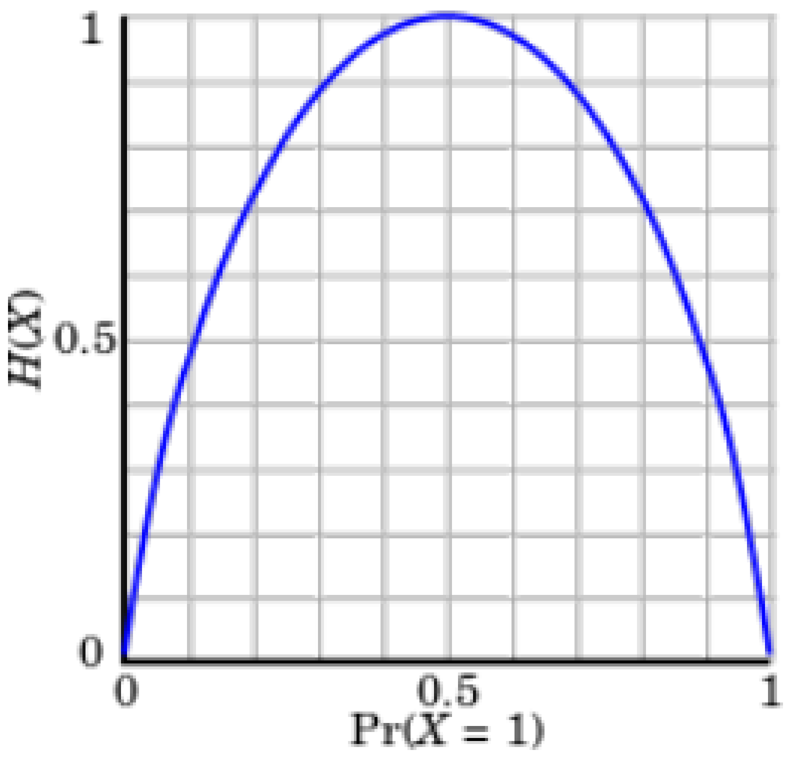
The entropy $H(S)$ of a set of examples S is defined as:

$$H(S) = -\sum_{i=1}^{n} P(X = i) \log_2 P(X = i)$$

where:

- $P(X = i)$ is the proportion of examples in S that belong to class i
- n is the number of classes (the number of possibile values of X)

In other words, Entropy meausers the *degree of uncertainty* of the information. It is maximal when X is uniformly distributed (all classes have the same probability) and minimal (zero) when all examples belong to the same class (pure set)



2.2.1 Information Theory (Shannon 1948)

The entropy is the average amount of information produced by a stochastic source of data. The *information* is associated to the *probability* of each datum (the surprise element):

- An event with probability 1 (certain event) provides no information (no surprise): $I(1) = 0$.
- An event with probability 0 (impossible event) provides infinite information (really surprising): $I(0) = \infty$.
- Given two independent events A and B, the information provided by both events is the sum of the information provided by each event:
$$I(A \cap B) = I(A) + I(B)$$

So is natoral defining
$$I(p) = -\log_2(p)$$

2.2.2 Code Theory (Shannon-Fano 1949, Huffman 1952)

The entropy is also related to the avarage number of bits required to transmit outcomes produces by a stochastic source process x.
Let suppose to have n events with same probability $p_i = \frac{1}{n}$. Home many bits do we need to encode these events? The answer is $\log_2(n)$ bits. For instance, if we have 4 events, we need 2 bits to encode them:.
In this case:
$$H(X) = -\sum_{i=1}^{n} P(X = i) \log_2 P(X = i) = -\sum_{i=1}^{n} \frac{1}{n} \log_2 \frac{1}{n} = \log_2(n)$$

2.3 Information Gain

In a decision tree, the goal is to maximize the information gain during the execution of the algorithm. In other words, the final split should result in the minimum possible impurity. Here are the main formulas:

Theorem 2.3.1 Entropy of X

$$H(X) = -\sum_{i=1}^{n} P(X = i) \log_2 P(X = i)$$

Theorem 2.3.2 Conditional Entropy of X given a specific $Y = v$

$$H(X \mid Y = v) = -\sum_{i=1}^{n} P(X = i \mid Y = v) \log_2 P(X = i \mid Y = v)$$

This measures the entropy of X restricted to the subgroup where $Y = v$.

Theorem 2.3.3 Conditional Entropy of X given Y

$$H(X \mid Y) = \sum_{v=1}^{m} P(Y = v) H(X \mid Y = v)$$

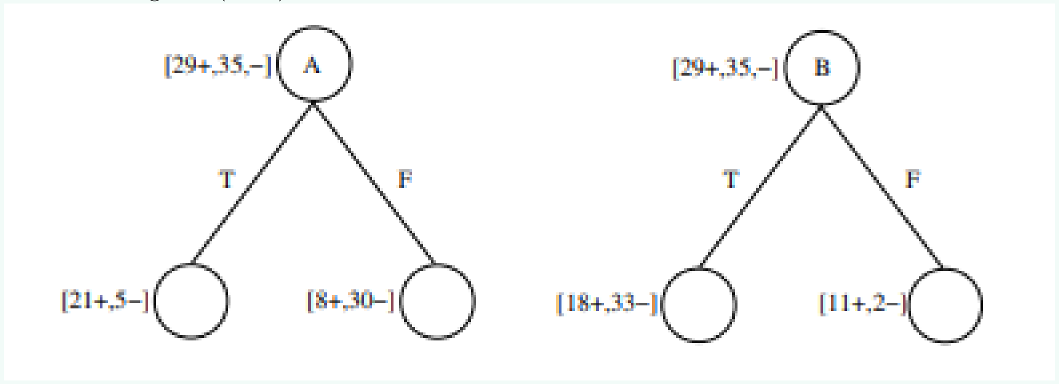
This is the generalization of 2.3, used to evaluate the utility of an attribute. It measures the average impurity that remains in X after splitting the data using all possible values of Y.

Theorem 2.3.4 Information Gain between X and Y

Here we are! $I(X, Y) = H(X) - H(X\|Y) = H(Y) - H(Y\|X)$

Example 2.3.1 (Information gain)

Let us measure the entropy reduction of the target variable Y due to some attribute X, that is the information gain $I(Y, X)$ between Y and X



$$H(Y) = -\frac{29}{64}\log_2\left(\tfrac{29}{64}\right) - \frac{35}{64}\log_2\left(\tfrac{35}{64}\right) = 0.994$$

$$H(Y \mid A = T) = -\frac{21}{26}\log_2\left(\tfrac{21}{26}\right) - \frac{5}{26}\log_2\left(\tfrac{5}{26}\right) = 0.706$$

$$H(Y \mid A = F) = -\frac{8}{38}\log_2\left(\tfrac{8}{38}\right) - \frac{30}{38}\log_2\left(\tfrac{30}{38}\right) = 0.742$$

$$H(Y \mid A) = 0.706 \cdot \tfrac{26}{64} \; + \; 0.742 \cdot \tfrac{38}{64} = 0.726$$

$$I(Y, A) = H(Y) - H(Y \mid A) = 0.994 - 0.726 = 0.288$$

$$H(Y \mid B) = 0.872$$

$$I(Y, B) = 0.122$$

2.4 Pros and Cons

Positive aspects:

- Explainable: decision trees can be easy to understand and can be visualized
- Inexpensive: low data preprocessing and low prediction constructed
- Both discrete and continuous

Negative aspects:

- Too descriptive: high risk of overfitting to training data
- Unbalanced: disproportionate training data (eg. medical diagnosis for rare disease) can be a problem

Let's see how we can fix some of these issues.

2.5 Random Forests

We use an **ensamble** of decision trees. This usually gives better results than a single component on its own. The various tree components have to be sufficiently uncorrelated, we can achieve this in various ways:

- **Bagging**: components are trained on different random subsets of the training data
- **Feature Randomness**: different components are built starting from a random subset of features

These techniques help reduce overfitting and improve the overall stability of the model.

Chapter 3

Overfitting

Let us consider the error of the hypothesis h

- on the training set, $error_{train}(h)$
- on the full data set $\mathcal{D}$, $error_{\mathcal{D}}(h)$

Definition 3.0.1: Overfitting

It's said taht h *overfits* the training set if there exixsts another hypotheses h' such that:

$$\text{error}_{\text{train}(h)} < \text{error}_{\text{train}(h')}$$

but

$$\text{error}_{\mathcal{D}(h)} > \text{error}_{\mathcal{D}(h')}$$

These models (h and h') represent two different situations. The first corresponds to a model that fits the training dataset very closely, including its uncertainty and noise. The second is simpler: it captures only the general trend of the training data and avoids fitting the noise. As a consequence, the error with respect to the true data distribution $\mathcal{D}$ is larger for the first model than for the second. The second one is better! Let's generalised.
But *We do not know* $\mathcal{D}$

3.1 Avoiding the over fitting

3.1.1 Detecting the Overfitting: validation set

For Detecting the Overfitting it's usefull divinding the dats aviables in two disjoint sets:

- **Training set**: set of datas that the model *use for learning*. The dtree is built by this datas
- **validation set**: This set is not shown during the training- It's used as "test" for evaluating the accuracy of the model

3.1.2 Early stopping

This is a proactive strategy. Instead of let the tree grows untill his major complexity, it's sopped first the possibility of Overfitting. The growing of a branch is stopped if these two conditions is verified:

- **The improvement is too small**: if a possible divsion of datas produces a gain of information below a certain threshold, it means that it's not usefull to continue
- **There are not enaugh datas**: if a node contains a number of examples too musch low, any decision taken would be statistically unreliable and probably based on noise. The tree stops to avoid creating rules based on coincidences.

3.1.3 Post - Pruning

This strategy is **reactive**. The decision tree is let grow completely on the training set, which may lead to overfitting, and then the useless or harmful branches are pruned.

Definition 3.1.1: Reduce-Error Post-Pruning

The *reduce-error post-pruning* technique works as follows:

- build the tree completely
- evaluate each branch using a validation set
- prune the branch whose removal improves accuracy the most
- repeat until no further pruning improves the accuracy

Chapter 4

Probabilistic approach

4.1 core idea

we have two main points of views:

- **traditional view**: we wanna to approximate a function $f : X \to X$
- **Probabilist view**: we wanna compute probablilities: $p : P(Y \mid X)$

4.1.1 Probs basics

Random variables

A random variables X represents an oyt come about which we're ncertain

Example 4.1.1 (Random variables)

- X =`true` if a randomly drawn stdent is male
- X = first name of the student
- X =`true` if a randomly drawn stdent have the same birthday

Formal def:

Definition 4.1.1: Probs variables

the set Ω of the possible outcomes is called the sample space. It is said random variable a measurable function over Ω:

- Discrete: $\Omega \to \{m, f\}$
- Continuos: $\Omega \to \mathbb{R}$

Definition 4.1.2: Probs def

it is defined $P(X)$ is the fraction of times X is true in repeated runs of the same experiment.

Note:

The definition requires that all samples

Pay attention:

Wrong Concept 4.1: bad examples

Sample space, let Ω be a space made the possibile sum:

$$\Omega = \{2, 3, 4, \ldots, 12\}$$

Problem: not all sums are equally likely! It should be:

$$P(sum = 2) = 1/11$$
$$P(sum = 7) = 1/11$$

but in reality:

- Sum = 2: can only happen one way: $(1, 1)$
- Sum = 7: can happen six ways:$(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)$

  so

$$P(sum = 2) \neq P(sum = 7)$$

A correct approach is

Claim 4.1.1 correct approach

Be $\Omega = (1, 1), (1, 2), (1, 3), ..., (6, 5), (6, 6)$, where $|\Omega| = 36$outcomes
each pair has equally probability $= \frac{1}{36}$
Now here is a correctly computing:

$$P(sum = 2) = \frac{|(1,1)|}{36} = \frac{1}{36}$$
$$P(sum = 7) = \frac{|(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)|}{36} = \frac{6}{36}$$

The Axioms of Probability Theory

These are the fundamental rules that make probability a "reasonable theory of uncertainty":

Axioms of probability theory

(1) Non-negativity: $0 \leqslant P(A) \leqslant 1$ for all events A. (4.1)

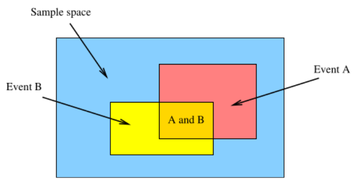
(2) Normalization: $P(\Omega) = 1.$ (4.2)

(3) Countable additivity: If $A_1, A_2, \ldots$ are disjoint, then $P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i).$ (4.3)

Then:

Corollary 4.1.1 consequences of the axioms

- Monotonicity: If $A \subseteq B$, then $P(A) \leq P(B)$
- Union rule (for two events): $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- $P(True) = 1$
- $P(False) = 0$



Derivied theorems

Corollary 4.1.2 Complement Rule

$$P(\neg A) = 1 - P(A)$$

Dm:

$$P(A \cup \neg A) = P(A) + P(\neg A) - P(A \cap \neg A)$$

But:

$$P(A \cup \neg A) = P(\text{True}) = 1 \quad \text{and} \quad P(A \cap \neg A) = P(\text{False}) = 0$$

Therefore:

$$1 = P(A) + P(\neg A) - 0 \quad \Longrightarrow \quad P(\neg A) = 1 - P(A)$$

Corollary 4.1.3 Partition Rule

$$P(A) = P(A \cap B) + P(A \cap \neg B)$$

Proof:

$$\begin{aligned} A &= A \cap (B \cup \neg B) && [\text{since } B \cup \neg B \text{ is always True}] \\ &= (A \cap B) \cup (A \cap \neg B) && [\text{distributive law}] \end{aligned}$$

Hence,

$$\begin{aligned} P(A) &= P((A \cap B) \cup (A \cap \neg B)) \\ &= P(A \cap B) + P(A \cap \neg B) - P((A \cap B) \cap (A \cap \neg B)) \\ &= P(A \cap B) + P(A \cap \neg B) - P(\text{False}) \\ &= P(A \cap B) + P(A \cap \neg B) \end{aligned}$$

Multivalued Discrete Random Variables

Definition 4.1.3: k-value Discrete Random Variables

A random variable A is *k-valued discrete* if it takes exactly one value from

$$\{v_1, v_2, \dots, v_k\}.$$

Proposition 4.1.1 Key proprieties

1. **Mutual exclusivity:** For $i \neq j$,
$$P(A = v_i \cap A = v_j) = 0$$
2. **Exhaustiveness:**
$$P(A = v_1 \cup A = v_2 \cup \cdots \cup A = v_k) = 1$$

Conditional Probability

Definition 4.1.4: Conditional probs

The Conditional probs of the event A *given* the event B is defined as the quantity

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

Corollary 4.1.4 Cahin roule

$$P(A \cap B) = P(B)P(A \mid B) = P(A)P(B \mid A)$$

Independent Events

Definition 4.1.5: Independent Events

Events A and B are independent when:

$$P(A \mid B) = P(A)$$

(Meaning: B provides no information about A.)

Corollary 4.1.5 consequences

- $P(A \cap b) = P(A)P(B)$ (from chail roule)
- $P(B|A) = P(B)$ (symmetry)

Bayes' Rule: The Heart of Probabilistic ML (ok chat... really?)

Theorem 4.1.1 Bayes's roule

Now we have Bayes roule

$$P(A \mid B) = \frac{P(A)P(B|A)}{P(B)}$$

Proof: It's true by the chain roule that: $P(A \cap B) = P(B)P(A \mid B)$. It's true also the reverse case $P(A \cap B) = P(B)P(A \mid B)$.
Since both expressions equal $P(A \mid B)$, they must equal each other:

$$P(A)P(B \mid A) = P(B)P(A \mid B)$$

that it's equal to

$$P(A \mid B) = \frac{[P(A)P(B \mid A)]}{P(B)}$$

Example 4.1.2 (The trousers problem)
Setup:

- 60% of students are boys, 40% are girls
- girls wear in the same number skirt and trousers
- boys only wear trousers

If we see a student wearing trousers, what is the probability that is a girl?

Solution: The probs a priori that a strudent is a girl is

$$P(G) = \frac{2}{5}$$

the probability that a student wears trousers is

$$P(T) = \frac{1}{5} + \frac{3}{5} = \frac{4}{5}$$

the probability that a student wear trousers, given that the student is a girl, is

$$P(T \mid G) = 1/2$$

So

$$P(G \mid T) = \frac{p(G)p(T \mid G)}{P(T)} = \frac{2/5 \cdot 1/2}{4/5} = 1/4$$

Machine Learning Form

Machine Learning Form For discrete Y with values $\{y_1, y_2, \ldots, y_m\}$ and X with values $\{x_1, x_2, \ldots, x_n\}$:

$$P(Y = y_i \mid X = x_j) = \frac{P(Y = y_i) \cdot P(X = x_j \mid Y = y_i)}{P(X = x_j)}$$

Expanding the denominator:

$$\begin{aligned} P(X = x_j) &= \sum_i P(X = x_j, Y = y_i) \quad \text{[sum over all } Y \text{ values]} \\ &= \sum_i P(Y = y_i) \cdot P(X = x_j \mid Y = y_i) \quad \text{[chain rule]} \end{aligned}$$

Complete Bayes' Rule:

$$P(Y = y_i \mid X = x_j) = \frac{P(Y = y_i) \cdot P(X = x_j \mid Y = y_i)}{\sum_i P(Y = y_i) \cdot P(X = x_j \mid Y = y_i)}$$

Terminology:

$$\underbrace{P(Y \mid X)}_{\text{posterior}} = \frac{\overbrace{P(X \mid Y)}^{\text{likelihood}} \cdot \overbrace{P(Y)}^{\text{prior}}}{\underbrace{P(X)}_{\text{marginal}}}$$

- **Posterior** $P(Y \mid X)$: What we want – probability of Y given observed X
- **Likelihood** $P(X \mid Y)$: How likely is X if Y is true?
- **Prior** $P(Y)$: What we believed before seeing X
- **Marginal** $P(X)$: Overall probability of observing X (normalization constant)

Alternative form:

$$\text{Posterior} = \frac{\text{Likelihood} \cdot \text{Prior}}{\text{Marginal Likelihood}}$$

where:

$$\text{Marginal} = \sum_Y P(X \mid Y) \cdot P(Y)$$

The term "marginal" means we've **marginalized** (integrated/summed) over Y.

4.2 The Joint Distribution

Definition 4.2.1: Joint Distribution

Let $X_1, X_2, \ldots, X_n$ be discrete random variables. The *joint probability distribution* (or *joint distribution*) of these variables is the function:

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n)$$

which assigns to every possible combination of values $(x_1, x_2, \ldots, x_n)$ the probability that the random variables simultaneously take those values.
Formally, for discrete variables, the joint distribution satisfies:

- $0 \leqslant P(x_1, x_2, \ldots, x_n) \leqslant 1$ for all $(x_1, \ldots, x_n)$
- $\sum_{x_1} \sum_{x_2} \cdots \sum_{x_n} P(x_1, x_2, \ldots, x_n) = 1$

Let's see an example

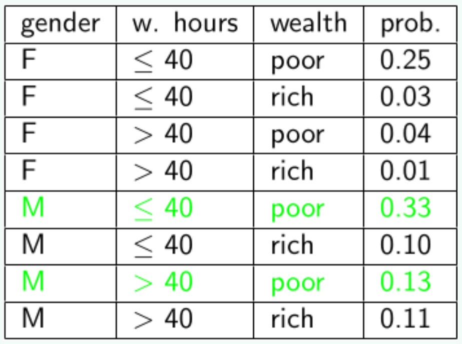
Example 4.2.1 (Joint distribution)

- build a table with all possible combinations of values of random variables (features)
- compute the probability for any different combination of values

This table is the "Joint distribution"!
Having that we may compute the probability of any event expressible as a logical combination of the features, with this formula

$$P(E) = \sum_{row \in E} (row)$$

in words for calculating an event we must add each row that is contained by the event. Let's provide an example (of an example)
Let us compute the probability $P(M, poor)$



gender	w. hours	wealth	prob.
F	≤ 40	poor	0.25
F	≤ 40	rich	0.03
F	> 40	poor	0.04
F	> 40	rich	0.01
M	≤ 40	poor	0.33
M	≤ 40	rich	0.10
M	> 40	poor	0.13
M	> 40	rich	0.11

we have: $P(M, poor) = 0.33 + 0.13 = 0.46$

4.2.1 Inference with the Joint distribution

Here are with the inference:

Definition 4.2.2: Contintional probability

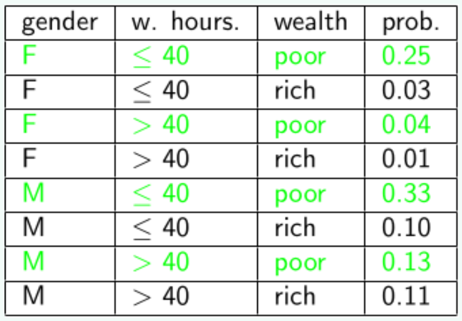
Let E_1 and E_2 be two events defined as logical conditions over subsets of the random variables (e.g., $E_1 : X_i = a, X_j = b$; $E_2 : Y = y$)
Then, *conditional probability* of E_1 given E_2 is:

$$P(E_1 \mid E_2) = \frac{P(E_1 \wedge E_2)}{P(E_2)} = \frac{\sum\limits_{row \in (E_1 \wedge E_2)} P(row)}{\sum\limits_{row \in (E_2)} P(row)}$$

for instance:

Example 4.2.2 (Conditional probability)

Let's compute $P(M|poor) = \frac{P(M \wedge poor)}{P(poor)}$. We know that $P(M, poor) = 0.46$. Let us compute $P(poor)$:



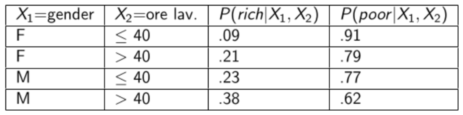
gender	w. hours.	wealth	prob.
F	≤ 40	poor	0.25
F	≤ 40	rich	0.03
F	> 40	poor	0.04
F	> 40	rich	0.01
M	≤ 40	poor	0.33
M	≤ 40	rich	0.10
M	> 40	poor	0.13
M	> 40	rich	0.11

Easy! $P(poor) = .75 \wedge P(M|poor) = 0.46/0.75 = 0.61$

4.2.2 Complexity issues

Let us build the joint table relative to

$$P(Y = wealth|X_1 = gender, X_2 = orelav.)$$



X_1=gender	X_2=ore lav.	$P(rich\|X_1, X_2)$	$P(poor\|X_1, X_2)$
F	≤ 40	.09	.91
F	> 40	.21	.79
M	≤ 40	.23	.77
M	> 40	.38	.62

To fill the table we need to compute $4 = 2^2$ parameters
If we have n random variable $X = X_1 \times X_2, \ldots, X_n$ where each X_i is boolean, we need to compute 2^n parameters.
These parameters are *probabilities*: to get reasonable value we would need a huge amount of data.
In particular the The Joint Distribution Requires *Exponential parameters*

Example 4.2.3 (features and params)

- With just 10 binary features, you need $2^{11} - 1 = 2047$ parameters
- With 20 features: over 1 million parameters
- With 100 features: 2^{101} a number larger than the estimated atoms in the observable universe.

This is computationally and statistically infeasible.

USing Bayes

for reducing complexity, we can rewrite the formula with the Bayes' rule:

$$P(Y = y_i \mid X = x_j) = \frac{P(Y = y_i) \cdot P(X = x_j \mid Y = y_i)}{\sum_i P(Y = y_i) \cdot P(X = x_j \mid Y = y_i)}$$

generalising:

$$P(Y \mid X_1, X_2, \ldots, X_n) = \frac{P(Y) \cdot P(X_1, X_2, \ldots, X_n \mid Y)}{P(X_1, X_2, \ldots, X_n)}$$

But... there is a problem, it's required to know

$$P(X_1, X_2, \ldots, X_n \mid Y)$$

that is the joint distribution of the features given Y, that requires, another time, 2^n params

4.2.3 Naive Bayes

For atteniung the complexity, it's possible assume an indipendencies conditional hypotesis, called "Naïve Bayes":

$$P(X_1, X_2, \ldots, X_n|Y) = \prod_i P(X_i|Y)$$

So given Y, X_i and X_j are independent from each other. In other therms:

$$P(X_i|X_j, Y) = P(X_i|Y)$$

Note:

This means: onece we know Y, the feature $X_i \forall i$ are independents between each others

Example 4.2.4 (example 1)

A box contains two coins: a regular coin and a fake two-headed coin ($P(H) = 1$).Choose a coin at random, toss it twice and consider the following events:

- A = First coin toss is H
- B = Second coin toss is H
- C = First coin is regular

Example 4.2.5 (example 2)

For individuals, height and vocabulary are not independent, but they are if age is given.

Giga formula with naive bayes

Theorem 4.2.1 Bayes rule

$$P(Y = y_i \mid X_1, \ldots, X_n) = \frac{P(Y = y_i) \cdot P(X_1, \ldots, X_n \mid Y = y_i)}{P(X_1, \ldots, X_n)}$$

Proof: Left to mesco as exercice

Theorem 4.2.2 Naïve Bayes

$$P(Y = y_i \mid X_1, \dots, X_n) = \frac{P(Y = y_i) \cdot \prod_j P(X_j \mid Y = y_i)}{P(X_1, \dots, X_n)}$$

Proof: Left to Bonzo as exercice

Theorem 4.2.3 Classification of a new sample $x^{\text{new}} = \langle x_1, \dots, x_n \rangle$

Given a new instance represented by the feature vector $x^{\text{new}} = (x_1, x_2, \dots, x_n)$, the predicted class is obtained as:

$$Y^{\text{new}} = \arg\max_{y_i} P(Y = y_i) \cdot \prod_j P(X_j = x_j \mid Y = y_i)$$

Proof: Seen as, using Bayes' formula, $\forall i$ the denominator used to calculate $P(Y = y_i|X_1, ..., X_n)$ remains the same,if we're only interested in maximizing the probability it's possible to only consider the numerator.
Given $P(X_1, ..., X_n) = C$,

$$\begin{aligned} P(Y = y_i) \cdot \prod_j P(X_j|Y = y_i) &= C \cdot \frac{P(Y = y_i) \cdot \prod_j P(X_j|Y = y_i)}{C} \\ &= C \cdot P(Y = y_i|X_1, ..., X_n) \end{aligned}$$

Because C is a positive constant for each y_i, $argmax_{y_i} P(Y = y_i|X_1, ..., X_n) = argmax_{y_i} C \cdot P(Y = y_i|X_1, ..., X_n)$.

Note:

Theorem 4.2.3 expresses the decision rule of the Naïve Bayes classifier. Given a new vector of features $x^{\text{new}} = (x_1, x_2, \dots, x_n)$, we estimate the most probable class y_i by maximizing the posterior probability $P(Y = y_i \mid X_1 = x_1, \dots, X_n = x_n)$, which—under the conditional independence assumption—reduces to the product of the prior $P(Y = y_i)$ and the individual likelihoods $P(X_j = x_j \mid Y = y_i)$.

4.3 Learning algorithm

Given discrete Random Variables X_i, Y, there are two phases

- **Training**: in this phases the maching learn from the data of training set, estimating two types of probs:
  - **Prior** (prob of the classes). For any possible value y_k of Y , estimate

    $$\pi_k = P(Y = y_k)$$

    example: if 9 out of 14 matches are "Play = Yes", then $\pi_{yes} = \frac{9}{14}, \quad \pi_{no} = \frac{5}{14}$
  - **Likelihoods**:(conditional probabilities of features). for any possible value x_{ij} of X_i estimate:

    $$\theta_{ijk} = P(X_i = x_{ij} \mid Y = y_k)$$

    It's the probability that a certain feature X_i assumes the value X_{ij}, given y_k.
    example: $P(Outlook = Sunny \mid Play = Yes) = \frac{2}{9}$
- **Classification of** $a^{new} = \langle a_1, \dots a_n \rangle$ (a vector with n observed values (one for each feature)). We want to establish which class it belong to

  decision-making formula:

  $$\begin{aligned} Y^{\text{new}} &= \arg\max_{y_k} P(Y = y_k) \cdot \prod_i P(X_i = a_i \mid Y = y_k) \\ &= \arg\max_k \pi_k \prod_j \theta_{ijk} \end{aligned}$$

  where:

- $P(Y = y_k)$: prior
- $P(X_i = a_i \mid Y = y_k)$: likelihood for each features
- the prod $\prod_i$ is given by Naive assumption

Example 4.3.1 (a good day to play tennis?)
we wanna build a model that, given certain weather conditions, predict whether it is a good day to play tennis or not
Our class variable is:

$$Y = Play \in \{Yes, No\}$$

and the features observed are:

$$X_1 = Outlook \quad X_2 = Temp \quad X_3 = Humidity \quad X_4 = Wind$$

Here we have the dataset:

Table 4.1: Dataset for the *Play Tennis* classification problem

Outlook	Temp	Humidity	Wind	Play
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

TODO: TABELLA FATTA FARE DA UN LLM NON È VENUTA BENISSIMO
Calculating the prior
From the dataset we can compute the prior probabilities of the class variable Y:

$$P(Y = \text{Yes}) = \frac{9}{14}, \qquad P(Y = \text{No}) = \frac{5}{14}.$$

These represent the empirical frequencies of the two possible outcomes of Y.

Calculating the likelihoods
For each feature X_i and each class $Y = y_k$, we estimate the conditional probabilities

$$P(X_i = x_{ij} \mid Y = y_k),$$

that is, the probability of observing a certain feature value x_{ij} given that the class is y_k.
For example:

$$P(\text{Outlook} = \text{Sunny} \mid Y = \text{Yes}) = \frac{2}{9}, \qquad P(\text{Outlook} = \text{Sunny} \mid Y = \text{No}) = \frac{3}{5}.$$

These values are computed as the relative frequencies in the dataset.

Classification of a new instance
Suppose we want to classify the new day

$$x^{\text{new}} = (\text{Outlook} = \text{Sunny},\ \text{Temp} = \text{Cool},\ \text{Humidity} = \text{High},\ \text{Wind} = \text{Strong}).$$

We apply the Naïve Bayes decision rule:

$$Y^{\text{new}} = \arg\max_{y_i} P(Y = y_i) \cdot \prod_j P(X_j = x_j \mid Y = y_i).$$

For Y = Yes:

$$P(\text{Yes}) \cdot P(\text{Sunny}|\text{Yes}) \cdot P(\text{Cool}|\text{Yes}) \cdot P(\text{High}|\text{Yes}) \cdot P(\text{Strong}|\text{Yes}) = \frac{9}{14} \cdot \frac{2}{9} \cdot \frac{3}{9} \cdot \frac{3}{9} \cdot \frac{3}{9} \approx 0.0053$$

For Y = No:

$$P(\text{No}) \cdot P(\text{Sunny}|\text{No}) \cdot P(\text{Cool}|\text{No}) \cdot P(\text{High}|\text{No}) \cdot P(\text{Strong}|\text{No}) = \frac{5}{14} \cdot \frac{3}{5} \cdot \frac{1}{5} \cdot \frac{4}{5} \cdot \frac{3}{5} \approx 0.0205$$

Decision:
Since

$$P(Y = \text{No} \mid x^{\text{new}}) > P(Y = \text{Yes} \mid x^{\text{new}}),$$

the predicted class is

$$Y^{\text{new}} = \text{No}.$$

Therefore, according to the Naïve Bayes model, it is **not a good day to play tennis**.

4.4 Generative techniques

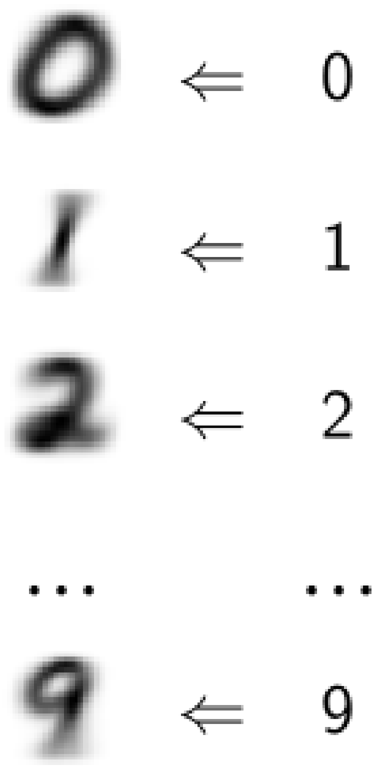
When we want to classify data (determine which category Y something belongs to given its features X), there is a fundamentally philosophical approaches, the **Generative Approach** that "learn how each class generates data", here is a sketch:

- Ask: "How does each class produce its characteristic data?"
- Model: $P(X|Y)$ (probability of features given category)
- Then use Bayes' Rule to reverse it and get $P(Y|X)$

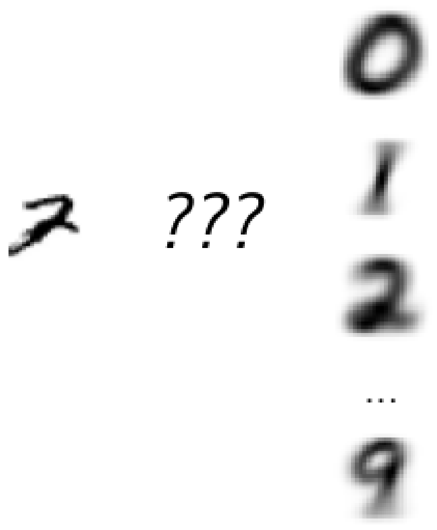
The term "Generative" come from the fact we're modeling the data generation process. We're essentially saying: "if I knew the category Y, I could generate/simulate typical data X from that category"

4.4.1 Big example: the visual intuition

We want to calssify images into categories $\{0, 1, \ldots, 9\}$ The generative approach says: "for each digit, learn waht imgs form that category typically look like. Then given a new img, see which category would most naturally produce such an img"



Okay, now we want to classify a new img:



Which of these distributions would most likely have generated this image?
Mathematically, for each category k:

$$\begin{aligned}\text{Score for category k} &= P(Y = k) \cdot P(X = \text{image}|Y = k)\\ &= \text{Prior} \cdot \text{Likelihood}\end{aligned}$$

You pick the category with the highest score. You're asking which generative mode (which category's distribution) best explains the observed data

Note:

The "score" is just the numerator of byes rule:

$$P(Y = k|X = \text{image}) = [P(Y = k) \cdot P(X = \text{image}|Y = k)]/P(X = \text{image})$$

Joint Distribution vs. Naïve Bayes

Ideally we'd want to model the Complete joint distribution:

$$P(X_1, X_2, \ldots, X_n|Y = y_i)$$

For MNIST, this would be the distribution of all 784 pixels (28×28 image) for images of each digit. This distribution would capture all the correlations between pixels - how pixel 1 relates to pixel 2, how groups of pixels form edges and curves, etc. Howver, modeling the full joint distribution for 784 dimensions is impossibly complex. We'd need:

- 2^{784} parameters just for binary pixels (more than atoms in the universe!)
- An astronomical amount of training data

So we play the card "Naïve Bayes" with 2500 atk and 1000 def:

$$P(X_1, \ldots, X_n \mid y = y_i) \approx \prod_j P(X_j \mid Y = y_i)$$

The assumption (effect) is: Given the category Y, all pixels are *independent*. This means we model each pixel separately:

$$P(X_1|Y = 0), P(X_2|Y = 0), \ldots, P(X_{784}|Y = 0)$$
$$P(X_1|Y = 1), P(X_2|Y = 1), \ldots, P(X_{784}|Y = 1)$$

This is computationally feasible, but we've lost all information about how pixels relate to each other!

4.4.2 Caution 1: The Zero Probability Problem

From a previous example if we have $P(Play = No|Outlook = Overcast)$ the result is $\theta_Overcast, No = 0$, This happened because in our training data, we never observed a "No" (don't play tennis) when the outlook was overcast

Note:

Remember the classification formula:

$$\text{Score} = \pi_k \cdot \prod_i \theta_{ijk}$$

If any single $\theta_{ijk} = 0$, the entire product becomes zero. So A single feature value you've never seen in training can completely eliminate a category from consideration, even if all other features strongly support it!

4.4.3 Caution 2: The Independence Assumption

Naïve Bayes assumes events are independent from each other (given Y). What if this is not the case?

The XOR Problem: A Fatal Limitation

Consider random binary images where pixels are either 0 or 1. We classify based on two pixels: p_1 and p_2.
Classification rule:

- **Category A:** if $p_1 = p_2$ (both same) — Images: $\{(0, 0), (1, 1)\}$
- **Category B:** if $p_1 \neq p_2$ (different) — Images: $\{(0, 1), (1, 0)\}$

This is an **XOR (exclusive OR)** relationship — a simple logical rule.
What Naïve Bayes learns:
For Category A (training: $(0, 0)$ and $(1, 1)$):

$$P(p_1 = 1 \mid A) = \frac{1}{2} \quad \text{[one out of two has } p_1 = 1\text{]}$$
$$P(p_2 = 1 \mid A) = \frac{1}{2} \quad \text{[one out of two has } p_2 = 1\text{]}$$

For Category B (training: $(0, 1)$ and $(1, 0)$):

$$P(p_1 = 1 \mid B) = \frac{1}{2} \quad \text{[one out of two has } p_1 = 1\text{]}$$
$$P(p_2 = 1 \mid B) = \frac{1}{2} \quad \text{[one out of two has } p_2 = 1\text{]}$$

Result: All probabilities are identical! For any test image (a, b):

$$\text{Score}_A = P(A) \cdot P(p_1 = a \mid A) \cdot P(p_2 = b \mid A) = 0.5 \cdot 0.5 \cdot 0.5 = 0.125$$
$$\text{Score}_B = P(B) \cdot P(p_1 = a \mid B) \cdot P(p_2 = b \mid B) = 0.5 \cdot 0.5 \cdot 0.5 = 0.125$$

Naïve Bayes cannot distinguish the categories! It achieves only 50% accuracy (random guessing) despite the trivially simple classification rule.
Why? The features p_1 and p_2 are **not independent** given the category — they're perfectly correlated:

- In Category A: if $p_1 = 0$ then $p_2 = 0$ (with certainty)
- In Category B: if $p_1 = 0$ then $p_2 = 1$ (with certainty)
- $P(p_2 = 1 \mid p_1 = 1, A) = 1 \neq P(p_2 = 1 \mid A) = 0.5$ — violates independence!

General lesson: Naïve Bayes cannot learn relationships between features. It only learns how common each individual feature value is within each class, not how features combine, interact, or correlate.

4.5 About Maximum Likelihood Estimation (MLE)

4.5.1 Problem definition

In words, the Maximum Likelihood Estimation (MLE) is the parameter value that maximizes the probability of observing the given data. For instance, if we have a model where all possible outcomes are 0 or 1, the parameter to estimate is computed using the *Bernoulli distribution*[1] $P(w) = \theta^w(1 - \theta)^{(1-w)}$ where $w \in \{0, 1\}$ represents all possible outcomes. However, normally we have a dataset (in this case a sequence of observations $D = \{w_1, w_2, \ldots, w_n\}$) and we don't know θ. The goal is to find the value of θ that makes our observed sequence most probable.
For independent observations, the likelihood is:

$$L(\theta|D) = P(D|\theta) = \prod_{i=1}^{n} P(w_i|\theta) = \prod_{i=1}^{n} \theta^{w_i}(1 - \theta)^{1-w_i}$$

Let $\alpha_0 = \sum_{i=1}^{n} w_i$ be the number of times we observed 1 (successes), and $\alpha_1 = n - \alpha_0$ be the number of times we observed 0 (failures). Then:

$$L(\theta|D) = \theta^{\alpha_0}(1 - \theta)^{\alpha_1}$$

The MLE is obtained by maximizing this likelihood (or equivalently its logarithm) with respect to θ.

[1]If you don't know this, please read the *Basta - Giolapalma notes for probability*

Definition 4.5.1: MLE

Given:

- A parametric probability model with parameter(s) θ
- Observed data $D = \{x_1, \ldots, x_n\}$
- A likelihood function $L(\theta|D) = P(D|\theta)$ (probability of data given parameters)

The *Maximum Likelihood Estimator* is defined as:

$$\hat{\theta}_{MLE} = \arg\max_\theta L(\theta \mid D) = \arg\max_\theta P(D|\theta)$$

In words: $\hat{\theta}_{MLE}$ is the parameter value that maximizes the probability of observing the given data.

4.5.2 Results for Bernoulli

Theorem 4.5.1 MLE for Bernoulli

Given a set of n i.i.d. (independent and identically distributed) observations $D = \{w_1, \ldots, w_n\}$ from a Bernoulli distribution with parameter θ. The Maximum Likelihood Estimate (MLE) for θ is the sample frequency of successes.
If α_0 is the number of successes (observations equal to 1), then the estimate is given by:

$$\hat{\theta}_{MLE} = \frac{\alpha_0}{n}$$

Derivation for Bernoulli: For independent Bernoulli trials with outcomes $w_i \in \{0, 1\}$, the likelihood is:

$$L(\theta|D) = \prod_{i=1}^{n} P(w_i|\theta) = \prod_{i=1}^{n} \theta^{w_i}(1-\theta)^{1-w_i}$$

Let $\alpha_0 = \sum_{i=1}^{n} w_i$ (number of 1's) and $\alpha_1 = n - \alpha_0$ (number of 0's). Then:

$$L(\theta|D) = \theta^{\alpha_0}(1-\theta)^{\alpha_1}$$

Taking the logarithm (which preserves the maximum since log is monotonically increasing):

$$l(\theta) = \log L(\theta|D) = \alpha_0 \log(\theta) + \alpha_1 \log(1-\theta)$$

To find the maximum, take the derivative and set to zero:

$$\frac{dl}{d\theta} = \frac{\alpha_0}{\theta} - \frac{\alpha_1}{1-\theta} = 0$$

Solving:

$$\begin{aligned} \frac{\alpha_0}{\theta} &= \frac{\alpha_1}{1-\theta} \\ \alpha_0(1-\theta) &= \alpha_1\theta \\ \alpha_0 &= \theta(\alpha_0 + \alpha_1) \\ \hat{\theta}_{MLE} &= \frac{\alpha_0}{\alpha_0 + \alpha_1} = \frac{\alpha_0}{n} \end{aligned}$$

To verify this is a maximum, check the second derivative:

$$\frac{d^2\ell}{d\theta^2} = -\frac{\alpha_0}{\theta^2} - \frac{\alpha_1}{(1-\theta)^2} < 0$$

Since the second derivative is negative for $\theta \in (0, 1)$, this confirms a maximum

Note:

This result can also be obtained by using the **binomial** distribution instead of the dataset likelyhood (which is basically the same thing without the constant factor). This is the approach seen in class, where we maximized $P(X^n = a_0|\theta)$ (where X^n is the number of 0 in D) as opposed to $L(\theta|D)$.

4.5.3 Multivalued case

Now I presented the formula just for two possible cases using the Bernoulli distribution, and for multivalued cases? WE HAVE THE *multinomial distribution*

Note:

Multinomial distribution is $P(X^1 = \alpha_1, X^2 = \alpha_2, ..., X^n = \alpha_n \mid \theta) = c \prod_i \theta_i^{\alpha_i}$. where α_i is the number of 'i' outcomes in the sequence and c is a combinatorial constant not depending on θ.

Theorem 4.5.2 MLE for Discrete Distributions

The Maximum Likelihood Estimate (MLE) for the probability θ_i of each outcome is its observed sample frequency.
If α_i is the number of times the i-th outcome has been observed in the n trials, then the estimate for its probability is:

$$\hat{\theta}_i = \frac{\alpha_i}{n}$$

Proof: We want to maximize $P(X^n = \alpha|\theta) = c \prod_i \theta_i^{\alpha_i}$ with respect to θ. We can ignore c as it's a positive factor and take the logarithm as before. So we end up with:

$$\theta_{\text{MLE}} = argmax_\theta \alpha_1 log(\theta_1) + \alpha_2 log(\theta_2) + ... + \alpha_n log(\theta_n)$$

Solving this seems kinda crazy, so I think the best approach is to consider $\theta_2 + ... + \theta_n = 1 - \theta_1$ and use the single variable case to prove for each θ_i.

Corollary 4.5.1 MLE for Naïve Bayes Parameters

As a direct consequence of the main theorem, the Maximum Likelihood Estimates for the parameters of a Naïve Bayes classifier are also given by their sample frequencies:

1. (π_k) The MLE for the prior probability of a class y_k is its relative frequency in the dataset.

$$\hat{\pi}_k = P(Y = y_k) = \frac{\#\mathcal{D}\{Y = y_k\}}{|\mathcal{D}|}$$

2. (θ_{ijk}) The MLE for the conditional probability of a feature X_i taking the value x_{ij} given a class y_k, is the relative frequency of that feature value within the subset of data belonging to class y_k.

$$\hat{\theta}_{ijk} = P(X_i = x_{ij}|Y = y_k) = \frac{\#\mathcal{D}\{X_i = x_{ij} \wedge Y = y_k\}}{\#\mathcal{D}\{Y = y_k\}}$$

4.6 Document classification with Bag of Words (BoW) approach

The Bag-of-Words (BoW) model is a technique for document classification, which involves assigning a document to a predefined category (like Sport, politics, Tech, ect...). The core idea is to treat a doc not like a sorted sequence of phrases but like a simple Bag where words have no order or grammar, in this technique the primary focus is on capturing the occurrence frequency of each word within the document (this is a very surface level method that doesn't consider things like sarcasm and negation).
So we assume that:

- words are the elementary value of events (X_i is the i-th word in the document): $\theta_{i,j,k} = P(X_i = x_j | Y = y_k)$, where x_j is the j-th distinct word contained somewhere in the document and y_k is the k-th category[2].
- events are independent (given the category [3]): $\forall i, j, m, n, k. i \neq m.\ P(X_i = x_j | X_m = x_n, Y = y_k) = \theta_{i,j,k}$
- distribution is independent from the position: $\forall i, m, j, k.\ \theta_{ijk} = \theta_{mjk}$

4.6.1 Training and classification

Event X_i = i-th word in the document: a discrete random variable assuming as many values as words in the language

$$\theta_{i,j,k} = P(X_i = x_j \mid Y = y_k)$$

In words: "the probability that in a document of the category y_k the word x_j appears at position i", but for the BoW technique we assume that we have a distribution independent from the position and that all event are independents so

$$\theta_{i,j,k} = \theta_{m,j,k} = \theta_{j,k}$$

For this reason, we can just consider the stochastic variable X = "word randomly extracted from document", with $\theta_{j,k} = P(X = x_j | Y = y_k)$. Then with discrete random variables X, Y let's define the training and classification:

- **Training**:
  - for any possible value y_k of Y, estimate the prior
    $$\pi_k = P(Y = y_k)$$
  - for each distinct word in the document x_j, estimate the likelyhood:
    $$\theta_{jk} = P(X = x_j \mid Y = y_k)$$
    (the condition prob that a certain word x_j appears in a doc, knowing that that doc belongs to category y_k)
- **Classification of $a^{\texttt{new}} = \langle a_1, \ldots, a_n \rangle$** (new sequence of words):
  $$\begin{aligned} Y^{\texttt{new}} &= \arg\max_{y_k} P(Y = y_k) \cdot \prod_{i=1}^{n} P(X = a_i | Y = y_k) \\ &= \arg\max_{k} \pi_k \cdot \prod_{i=1}^{n} \theta_{j(i),k} \end{aligned}$$
  where $j(i)$ is such that $x_{j(i)} = a_i$.

The probabilities needed for the classifier are estimated from the dataset using Maximum Likelihood Estimates (MLE's):

- The MLE for the prior probability of a class, π_k, is the fraction of the documents in the training set that belong to category y_k
- The MLE for the conditional probability of a word, $\theta_{\texttt{word},k}$, is the frequency of that word within all documents belonging to category y_k

The classification formula involves multiplying many small probabilities, which can lead to numerical instability (underflow). To solve this, we can maximize the logarithm of the likelihood instead, since the logarithm is a monotonically increasing function and will not change the location of the maximum.

[2]Words and categories aren't ordered in any particular way.

[3]Obviously, knowing a word in the document will give information regarding the type of the document (this is the notion that the model relies on), thus changing the distribution for all the other positions. For this reason we say that events are independent *only* if the category is considered known.

The classification rule becomes:

$$Y^{\texttt{new}} = \arg\max_{y_k} \left(\log(\pi_k) + \sum_i \log(\theta_{j(i)k}) \right)$$

This can be further simplified by grouping identical words (values of i with the same $j(i)$). If n_j is the number of occurrences of the unique word x_j in the new document, the sum becomes a weighted sum:

$$\sum_i \log(\theta_{j(i)k}) = \sum_j n_j \cdot \log(\theta_{jk})$$

This final expression can be elegantly interpreted as a dot product in a high-dimensional vector space where each word of the vocabulary is a dimension

4.6.2 Dot product, correlation, Cosine similarity

the core idea is manipulating texts in numerical vectors. For doing this, you can create a vectorial space at $n-$dimension where each dimension correspond to a word to one unique word of vocabulary. HEre is the training and classification:

- **Training** For any category k build a "spectral" vector

  $$s_k = \langle \log_{(\theta_{jk})} \rangle$$

  θ_{jk} = frequency of the word j in documents of the category k

  For each category (ex. "Sport") we'll create a spectral vector where each componet of this vector is the log of the probs ($\log(\theta_{jk})$) of a specific category. This vector represent the "profile" or carateristic direction of that category in the space of words

- **classification** Given a new doc, compute a vector

  $$d = \langle n_j \rangle_{j \in \texttt{words}}$$

  and classify the doc as

  $$\arg\max_k d \cdot s_k = \sum_j d_j \cdot s_{jk}$$

  so we're searching for the value of k that leads to the highest *correlation* between the two vectors.

geometrically and analytically

Definition 4.6.1: Dot product - Geometric def

We define dot product

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cdot |\mathbf{b}| \cos(\theta)$$

Definition 4.6.2: Dot product - Analytic definition

given $\mathbf{a} = (a_1, a_2, \ldots, a_n), \mathbf{b} = (b_1, b_2, \ldots, b_n)$

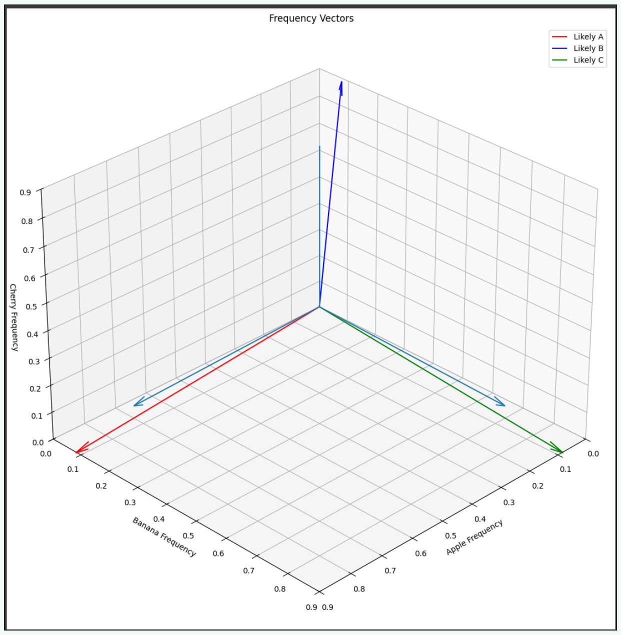
$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

The dot product can be influenced by the length of docs (a longer doc can have high frequencies). The cosine similarity resolve this problem normalizing the dot procd respect to the lenght of two vectors

$$S_c(\mathbf{a}, \mathbf{b}) = \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|}$$

the results is a value $\in [-1, 1]$ that measures the "similarity of direction" ignoring the length. A value of 1 means that the vectors points in the same direction

Example 4.6.1 (Vector correlation maximization)



This is an example in 3 dimensions showing how the BoW approach can be seen as minimizing the angle between vectors (or maximizing their correlation)

Theorem 4.6.1 Equivalence of Dot Product Definitions

The geometric definition of the dot product, $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$, is equivalent to the analytic definition, $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$.

Proof: The proof begins with the *Carnot's theorem*, which generalizes the Pythagorean theorem. Consider the triangle formed by the vectors $\vec{a}$, $\vec{b}$, and their difference, $\vec{a} - \vec{b}$.
The Law of Cosines states:

$$|\vec{a} - \vec{b}|^2 = |\vec{a}|^2 + |\vec{b}|^2 - 2|\vec{a}||\vec{b}| \cos(\theta)$$

Here, the term $|\vec{a}||\vec{b}| \cos(\theta)$ corresponds to the geometric definition of the dot product, $\vec{a} \cdot \vec{b}$. By rearranging the formula algebraically, we can express the dot product in terms of vector magnitudes:

$$\vec{a} \cdot \vec{b} = \frac{|\vec{a}|^2 + |\vec{b}|^2 - |\vec{a} - \vec{b}|^2}{2}$$

Next, we substitute the analytic definition of the squared magnitude of a vector. For simplicity, let's consider the two-dimensional case where $\vec{a} = [a_1, a_2]$ and $\vec{b} = [b_1, b_2]$. We have:

$$|\vec{a}|^2 = a_1^2 + a_2^2,$$
$$|\vec{b}|^2 = b_1^2 + b_2^2,$$
$$|\vec{a} - \vec{b}|^2 = (a_1 - b_1)^2 + (a_2 - b_2)^2.$$

Substituting these into the dot product expression yields:

$$\vec{a}\cdot\vec{b} = \frac{(a_1^2 + a_2^2) + (b_1^2 + b_2^2) - \left((a_1 - b_1)^2 + (a_2 - b_2)^2\right)}{2}.$$

Expanding the squared terms in the numerator:

$$\vec{a}\cdot\vec{b} = \frac{a_1^2 + a_2^2 + b_1^2 + b_2^2 - (a_1^2 - 2a_1b_1 + b_1^2 + a_2^2 - 2a_2b_2 + b_2^2)}{2}.$$

Simplifying by canceling out the squared terms:

$$\vec{a}\cdot\vec{b} = \frac{2a_1b_1 + 2a_2b_2}{2}.$$

This simplifies to the final result:

$$\vec{a}\cdot\vec{b} = a_1b_1 + a_2b_2.$$

This is the analytic definition of the dot product for the two-dimensional case. The proof can be generalized to n dimensions, demonstrating the equivalence of the geometric and analytic definitions of the dot product.

4.7 The linear nature of Naive Bayes (boolean case)

If both X_i and Y are boolean variables, it's possible to transform the Naive Bayes formula using certain properties of boolean functions to obtain a *linear* expression.

Theorem 4.7.1 Linarity of classifier Naive bayes for boolean feature

Given X_i, Y booleans, the classification of a new $\vec{x} = \langle x_1, \ldots, x_n\rangle$ happens comparing the probs that x belong to class 1 and 0:

$$\frac{P(Y=1 \mid X_1 \ldots X_n = \vec{x})}{P(Y=0 \mid X_1 \ldots X_n = \vec{x})} = \frac{P(Y=1)\prod_i P(X_i = x_i \mid Y = 1)}{P(Y = 10 \prod_i P(X_i = x_i \mid Y = 0} \geqslant 1$$

if the results is $\geqslant 1$ we choose the class one.
Passing to logarithms we have:

$$\log\frac{P(Y=1)}{P(Y=0)} + \sum_i \frac{X_i = x_i \mid Y = 1}{X_i = x_i \mid Y = 0} \geqslant 0$$

if the results is $\geqslant 0$ we choose the class one.
Now define the conditional probs loke $\theta_{ik} = P(X_i = 1 \mid Y = k)$. Of consequence, $P(X_i = 0 \mid Y = k) = 1 - \theta_{ik}$. The generic therm of the sum, $\log\frac{P(X_i=x_i|Y=1)}{P(X_i=x_i|Y=0)}$, using the fact that for a boolean function $f(x) = xf(1) + (1-x)f(0)$ we have:

$$\sum_i \left(x_i \cdot \log\frac{P(X_i = 1|Y=1)}{P(X_i = 1|Y=0)} + (1 - x_i)\cdot\log\frac{P(X_i = 0|Y=1)}{P(X_i = 0|Y=0)}\right)$$

with θ, the expression becomes:

$$\log\frac{P(Y=1)}{P(Y=0)} + \sum_i \left(x_i \cdot \log\frac{\theta_{i1}}{\theta_{i0}} + (1 - x_i)\cdot\log\frac{1-\theta_{i1}}{1-\theta_{i0}}\right) \geq 0$$

that is a linear expression in the set of features x_i in the form

$$w_0 + \sum_i w_i x_i \geqslant 0$$

Classification algorithms based on a linear combination of the features values. Every feature is evaluated independently from the others and contributes to the result in a linear way, with a suitable weight (that is a parameter of the model, to be estimated). For instance, if the features are pixels of some image, we may use linear methods only up some normalization (in position and dimension) of the object to be recognized

4.8 Gaussian Naive Bayes

The standard Naïve Bayes algorithm is designed for discrete features. However, in many real-world scenarios, features can be continuous, such as the height of an individual, the temperature, or the color intensity of a pixel. For continuous variables, the probability of observing a specific value is infinitesimally small, making the pointwise probability $P(X_i|Y)$ effectively null.
To address this challenge, Gaussian Naïve Bayes extends the classifier by making a crucial assumption: that the data for each continuous feature, conditioned on a specific class, follows a **Gaussian (or Normal) distribution** [4]. Instead of estimating a probability for each value, we estimate the parameters of this distribution (the mean and variance) from the training data.

Definition 4.8.1: Gaussian Distribution

The *Gaussian Distribution* is a continuous probability distribution characterized by its mean μ and variance σ^2. Its probability density function (PDF) is given by:

$$p(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where:

- The mean $E[X] = \mu$ defines the center of the distribution.
- The variance $\text{Var}[X] = \sigma^2$ defines the spread of the distribution.

4.8.1 Accuracy, Precision and Recall

Definition 4.8.2: Accuracy

$$\texttt{Accuracy} = \frac{TP + TN}{ALL}$$

number of correctly classified samples

Definition 4.8.3: Precision

$$\texttt{Precision} = \frac{TP}{TP + FP}$$

percentage of true prositives over predicted ones

Definition 4.8.4: Recall

$$\texttt{Recall} = \frac{TP}{TP + FN}$$

percentage of true positives over all positives

Definition 4.8.5

$$F1 = 2\frac{Precision \cdot Recall}{Precision + Recall}$$

harmonic mean of precision and recall

[4] The reason we make this assumption is because, given a mean and a variance, the Gaussian is the distribution with the highest entropy. This means that it makes the weakest assumptions, distributing the probability in the most even way possible for all possible results.

4.8.2 Descriptive parameters of the model

We assume (inductive bias) that for every value y_k of Y the random variable $P(X_i|Y = y_k)$ has a Gaussian distribution

$$\mathcal{N}(x \mid \mu_{ik}, \sigma_{ik}) = \frac{1}{\sigma_{ik}\sqrt{2\pi}} e^{-\frac{(x-u_{ik})^2}{2\sigma_{ik}^2}}$$

- **Learning**: estimate the values of the parameters μ_{ik}, σ_{ik} and $\pi_k = P(Y = y_k)$
- **Classification** of $x^{new} = \langle a_1, \ldots, a_n \rangle$

$$\begin{aligned} Y^{\texttt{new}} &= \arg\max_{y_k} P(Y = y_k) \cdot \prod_i p(X_i = a_i \mid Y = y_k) \\ &= \arg\max_k \pi_k \cdot \prod_i \mathcal{N}(a_i \mid \mu_{ik}, \sigma_{ik}) \end{aligned}$$

4.8.3 MLE for Gaussian Parameters

Given a set of training data, the Maximum Likelihood Estimates (MLE) for the parameters of the Gaussian distribution for each feature X_i and class y_k are the sample mean and sample variance, calculated from the subset of data belonging to class y_k.

- **Mean Estimate (μ_{ik}):** The mean value of X_i for samples with label $Y = y_k$. Formally:

$$\mu_{ik} = \frac{\sum_j X_i^j \delta(Y^j = y_k)}{\sum_j \delta(Y^j = y_k)}$$

- **Variance Estimate (σ_{ik}^2):** The variance of X_i for samples with label $Y = y_k$. Formally:

$$\sigma_{ik}^2 = \frac{\sum_j (X_i^j - \mu_{ik})^2 \delta(Y^j = y_k)}{\sum_j \delta(Y^j = y_k)}$$

where j ranges over all samples in the training set, and $\delta(Y^j = y_k)$ is the indicator function:

$$\delta(Y^j = y_k) = \begin{cases} 1 & \text{if } Y^j = y_k \\ 0 & \text{otherwise} \end{cases}$$

4.9 Logistic Regression

Logistic regression is a discriminative machine learning method used primarily for classification problems

4.9.1 Core idea

- Naive Bayes allows us to compute $P(Y|X)$ after having learned $P(Y)$ and $P(X|Y)$
- Why not directly learn $P(Y|X)$?

Theorem 4.9.1 The shape of $P(Y \mid X)$

hypotesis:

- Y boolean random variable
- X_i continuous random variable
- X_i independent from each other given Y
- $P(X_i \mid Y = k)$ have Gaussian distributions $N(\mu_{i_k}, \sigma_i)$ (Warning not $\sigma_i k$!) [a]

- Y has a Bernoulli distribution (π)

then

$$P(Y = 1 \mid X = \langle x_1 \dots x_n \rangle) = \frac{1}{1 + \exp\left(w_0 + \sum_i w_i x_i\right)} \tag{4.4}$$

[a]So we assume that features have the same gaussian "shape" for both values of Y, but they can be centered in different positions.

Proof: By hypothesis, the conditional probability of a feature X_i is given by a Gaussian PDF:

$$P(X_i|Y = k) = \mathcal{N}(x_i; \mu_{ik}, \sigma_i) = \frac{1}{\sigma_i\sqrt{2\pi}} e^{-\frac{(x_i-\mu_{ik})^2}{2\sigma_i^2}}$$

We want to find the shape of $P(Y = 1|\vec{x})$. Using the definition of conditional probability and the law of total probability, we have:

$$\begin{aligned} P(Y = 1|\vec{x}) &= \frac{P(Y = 1)P(\vec{x}|Y = 1)}{P(\vec{x})} \\ &= \frac{P(Y = 1)P(\vec{x}|Y = 1)}{P(Y = 1)P(\vec{x}|Y = 1) + P(Y = 0)P(\vec{x}|Y = 0)} \end{aligned}$$

Dividing the numerator and denominator by $P(Y = 1)P(\vec{x}|Y = 1)$ gives:

$$\begin{aligned} P(Y = 1|\vec{x}) &= \frac{1}{1 + \frac{P(Y=0)P(\vec{x}|Y=0)}{P(Y=1)P(\vec{x}|Y=1)}} \\ &= \frac{1}{1 + \exp\left(\ln\left(\frac{P(Y=0)P(\vec{x}|Y=0)}{P(Y=1)P(\vec{x}|Y=1)}\right)\right)} \end{aligned}$$

Using the Naïve Bayes assumption of conditional independence $P(\vec{x}|Y) = \prod_i P(x_i|Y)$ and substituting $P(Y = 1) = \pi$, we get:

$$P(Y = 1|\vec{x}) = \frac{1}{1 + \exp\left(\ln\frac{1-\pi}{\pi} + \sum_i \ln\frac{P(x_i|Y=0)}{P(x_i|Y=1)}\right)}$$

Substituting the Gaussian PDF and simplifying the term inside the summation leads to a linear function of x_i:

$$P(Y = 1|\vec{x}) = \frac{1}{1 + \exp\left(\ln\frac{1-\pi}{\pi} + \sum_i \left(\frac{\mu_{i0}-\mu_{i1}}{\sigma_i^2} x_i + \frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma_i^2}\right)\right)}$$

This expression has the form $\frac{1}{1+\exp(z)}$, where z is a linear combination of the features x_i. By defining w_0 and w_i as the appropriate combinations of the Gaussian parameters (μ, σ, π), we obtain the logistic form:

$$P(Y = 1|\vec{x}) = \frac{1}{1 + \exp(w_0 + \sum_i w_i x_i)}$$

In particular, we have:

- $w_i = \frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}$
- $w_0 = ln\left(\frac{1-\pi}{\pi}\right) + \sum_i \frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma_i^2}$

4.9.2 Training for logistic regression

So, logistic regression assumes

$$P(Y = 1 \mid X = \langle x_1 \dots x_n \rangle) = \frac{1}{1 + \exp\left(w_0 + \sum_i w_i x_i\right)}$$

and directly tries to estimate the parameters w_i (the change in sign is ininfluent).

Definition 4.9.1: Logistic function

It's a very important funcition and it's called logistic function:

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

With this def we know that $P(Y = 1 \mid x, w) = \sigma(w_0 + \sum_i x_i w_i)$

Training via Maximum Likelihood Estimation

Given independent training samples $\langle x^\ell, y^\ell \rangle$, we want to find the parameters w that **maximize** the probability of observing our training data.
The probability of all samples is:

$$\prod_\ell P(y^\ell | x^\ell, w) = \prod_\ell P(y^\ell = 1 | x^\ell, w)^{y^\ell} \cdot P(y^\ell = 0 | x^\ell, w)^{(1-y^\ell)}$$

Note:

This formula works because when $y^\ell = 1$, only the first term contributes (the second becomes 1), and vice versa when $y^\ell = 0$.
Since $P(Y = 0|x, w) = 1 - P(Y = 1|x, w)$, we can write:

$$\prod_\ell P(y^\ell | x^\ell, w) = \prod_\ell \sigma(w_0 + \sum_i w_i x_i^\ell)^{y^\ell} \cdot (1 - \sigma(w_0 + \sum_i w_i x_i^\ell))^{(1-y^\ell)}$$

Log-Likelihood

Instead of maximizing the product directly, we **maximize the logarithm** of this probability (which is equivalent since log is monotonic):

$$l(w) = \sum_\ell \log P(y^\ell | x^\ell, w) = \sum_\ell \left(y^\ell \cdot \log P(Y = 1 | x^\ell, w) + (1 - y^\ell) \cdot \log P(Y = 0 | x^\ell, w)\right)$$

Using the logistic function, we can set

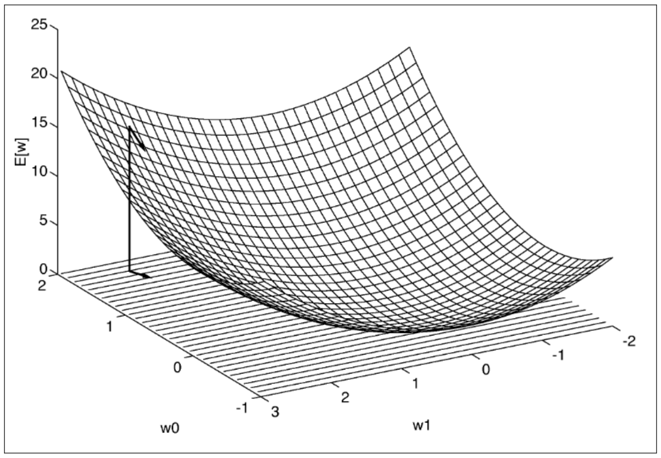
$$\alpha^\ell = \sigma\left(w_0 + \sum_i w_i x_i^\ell\right)$$

and rewrite the log-likelihood as

$$l(w) = \sum_\ell y^\ell \log(\alpha^\ell) + (1 - y^\ell) \log(1 - \alpha^\ell)$$

4.10 The Gradient Technique

The training set is fixed, so the loss function depends on the parameters of the model. Unfortunately, there is no analytic solution for the previous optimization problem. So, we use iterative optimization methods, like the **gradient technique**:



simple model with 2 parameters

Definition 4.10.1: Gradient

The *gradient* of a function $f(w_0, ..., w_n)$ is a vector composed of the partial derivatives of f with respect to each parameter w_i. It points in the direction of the steepest ascent on the function's surface.

$$\nabla_w[f] = \left[\frac{\partial f}{\partial w_0}, \frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_n}\right]$$

We can use the definition of the gradient in order to find the direction that lowers the loss function $E(w)$ the most (the direction of greatest descent) and to subsequently perform a "step" (determined by the learning rate) in that direction. This is a *general* technique that can be applied to any differentiable function.

Note:

This is also a *local* technique: it's not given that the minimum we find is global, because it could just be a local one. For logistic regression, we can be certain it's global (as we'll see later), but for more complicated models this isn't always true. For the latter, it's possible to balance out these fake minimals by having lots of parameters, because this way it's possible to simply ignore the outliers and trust the value found by the most parameters.

So the correct direction is calculated using the gradient (although tecnicaly in the opposite direction), and the parameters are updated by repeating these stepps:

Theorem 4.10.1 Training Rule (Gradient Descent)

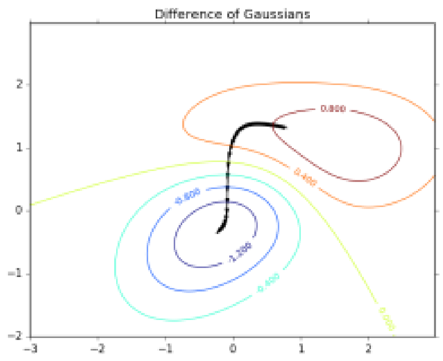
To find the parameters w that minimize $E(w)$, we use *Gradient Descent*. We start with an initial guess for w and iteratively update the parameters by taking small steps in the opposite direction of the gradient. The update rule for a single weight w_i is:

$$\Delta w_i = \eta \cdot \frac{\partial E}{\partial w_i}$$

$$w_i = w_i + \Delta w_i$$

where η (denoted μ in the slide) is the *learning rate*, a small constant that controls the size of each step. This process is repeated until the parameters converge.

The concept is simple. We can visualize the model's error function, $E(w)$, as a topographical map. On this map, the altitude represents the error (the vertical axis), and the goal is to find the lowest point (the minimum error). To achieve this, at iteration 0, we start from a random point. We then calculate the gradient ($\nabla E(w)$) at that position. The gradient, by definition, points in the direction of steepest ascent (the "maximum climb"). Therefore, to descend ('go down') and minimize the error, we must move in the opposite direction of the gradient



4.10.1 Error function for Logistic Regression

In ideal case, the error surface is convex, this is fundamentally 'cause it means that there is just a single global minimum. Fortunately we have THAT:

Theorem 4.10.2 Convexity of the Error Surface for Logistic Regression

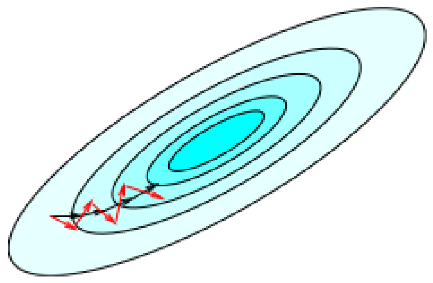
Let $E(w) = -l(w)$ be the error function for a logistic regression model, defined as the *negative log-likelihood* of the training data.
This error function $E(w)$ is a *convex function* with respect to the parameter vector w.

Note:
This is a critical property of the logistic regression model. Because the error surface is convex, it possesses a single global minimum and no local minima. This guarantees that iterative optimization methods, such as Gradient Descent, will converge to the unique optimal solution, regardless of the initial parameter values.

4.10.2 Different learning methods

How many points to use for measuring error and gradient? there are two ways:

- **Full Batch**: all the features in training set are used for calculating the gradient
- Online: one sample at a time. Gradient zig-zags around the direction of the steepest descent.
- *mini-batch*: random subset of training samples, a good compromise



SGD (Stochastic Gradient Descent): the direction calculated by the Mini-Batch isn't necessarily the best, but the hope is that the sum of the errors for all the considered parameters rounds out.

4.10.3 Calculating the Gradient for logistic regression

First a small review of the other formulas:

Definition 4.10.2: Probability that sample ℓ is in category $Y = 1$

$$P(Y = 1|x^\ell, w) = \sigma(w_0 + \sum_i w_i x_i^\ell) = \alpha^\ell$$

then:

Definition 4.10.3: Log-likelihood

$$l(w) = \sum_\ell \log P(Y = y^\ell|x^\ell, w) = \sum_\ell \left(y^\ell \log(\alpha^\ell) + (1 - y^\ell)\log(1 - \alpha^\ell)\right) \tag{4.5}$$

Theorem 4.10.3 Gradient of the Log-Likelihood for Logistic Regression

The partial derivative of the log-likelihood function $l(w)$ with respect to a single weight w_i is given by the sum over all samples:

$$\frac{\partial l(w)}{\partial w_i} = \sum_\ell x_i^\ell \cdot (y^\ell - \alpha^\ell) \tag{4.6}$$

Proof: We want to compute the partial derivative $\frac{\partial l(w)}{\partial w_i}$ of the log-likelihood function $l(w)$. First, let's define the components, where $\alpha^\ell = \sigma(z^\ell)$ and $z^\ell = w_0 + \sum_i w_i x_i^\ell$.

Step 1: Derivative of $\log(\alpha^\ell)$ *w.r.t.* z^ℓ

$$\frac{\partial \log(\alpha^\ell)}{\partial z^\ell} = \frac{1}{\alpha^\ell}\frac{\partial \alpha^\ell}{\partial z^\ell}$$

Using the property $\frac{\partial \sigma(z^\ell)}{\partial z^\ell} = \sigma(z^\ell)(1 - \sigma(z^\ell))$, which is equivalent to $\alpha^\ell(1 - \alpha^\ell)$, we get:

$$\frac{\partial \log(\alpha^\ell)}{\partial z^\ell} = \frac{1}{\alpha^\ell}(\alpha^\ell(1 - \alpha^\ell)) = 1 - \alpha^\ell$$

(Note: The slide shows an intermediate step $\frac{\exp(-z^\ell)}{1+\exp(-z^\ell)}$ which simplifies to $1 - \alpha^\ell$).

Step 2: Derivative of $\log(1 - \alpha^\ell)$ *w.r.t.* z^ℓ First, we simplify $\log(1 - \alpha^\ell)$:

$$\log(1 - \alpha^\ell) = \log\left(1 - \frac{1}{1 + \exp(-z^\ell)}\right) = \log\left(\frac{\exp(-z^\ell)}{1 + \exp(-z^\ell)}\right)$$

$$= \log(\exp(-z^\ell)) - \log(1 + \exp(-z^\ell)) = -z^\ell + \log(\alpha^\ell)$$

Now, we differentiate this expression with respect to z^ℓ, using the result from step 1:

$$\frac{\partial \log(1 - \alpha^\ell)}{\partial z^\ell} = \frac{\partial}{\partial z^\ell}(-z^\ell + \log(\alpha^\ell)) = -1 + (1 - \alpha^\ell) = -\alpha^\ell$$

Step 3: Final Gradient Calculation (using Chain Rule) We recall the log-likelihood function:

$$l(w) = \sum_\ell \left(y^\ell \log(\alpha^\ell) + (1 - y^\ell)\log(1 - \alpha^\ell)\right)$$

Using the derivative's properties, we can write:

$$\frac{\partial l(w)}{\partial w_i} = \sum_\ell \frac{\partial l^\ell(w)}{\partial w_i}$$

Where $l^\ell(w) = y^\ell \log(\alpha^\ell) + (1 - y^\ell)\log(1 - \alpha^\ell)$ is the log-likelihood for a single sample.

We can now apply the chain rule: $\frac{\partial l^\ell(w)}{\partial w_i} = \frac{\partial l^\ell(w)}{\partial z^\ell}\frac{\partial z^\ell}{\partial w_i}$.
First, $\frac{\partial z^\ell}{\partial w_i} = x_i^\ell$.
Second, we find $\frac{\partial l^\ell(w)}{\partial z^\ell}$ by substituting the results from steps 1 and 2:

$$\begin{aligned}\frac{\partial l(w)}{\partial z} &= \sum_\ell \left(y^\ell(1-\alpha^\ell) + (1-y^\ell)(-\alpha^\ell)\right)\\ &= \sum_\ell (y^\ell - y^\ell\alpha^\ell - \alpha^\ell + y^\ell\alpha^\ell)\\ &= \sum_\ell (y^\ell - \alpha^\ell)\end{aligned}$$

Finally, combining the parts with the chain rule:

$$\frac{\partial l(w)}{\partial w_i} = \sum_\ell \frac{\partial l^\ell(w)}{\partial z^\ell}\frac{\partial z^\ell}{\partial w_i} = \sum_\ell \left((y^\ell - \alpha^\ell)x_i^\ell\right)$$

Note: on the slide the prof makes the mistake of not parameterizing z over ℓ, leading to a result that doesn't make any sense (What is x_i^ℓ outside of the sum? We don't have a value for ℓ!). We can fix this by applying the chain rule inside the sum, leading to a valid value for z^ℓ.

Note:

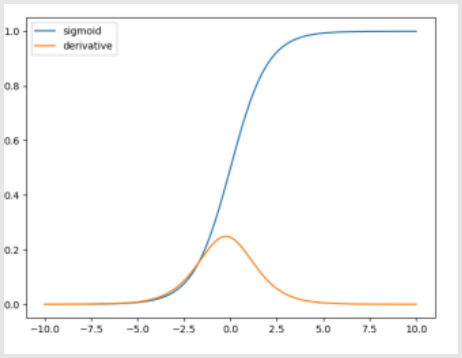
previsuly we proved that:

$$\frac{\partial \log \alpha^\ell}{\partial z^\ell} = \frac{1}{\sigma(z^\ell)}\frac{\partial \sigma(z^\ell)}{\partial z^\ell} = 1 - \sigma(z^\ell)$$

hence:

$$\frac{\partial \sigma(z^\ell)}{\partial z^\ell} = \sigma(z^\ell)(1-\sigma(z^\ell))$$

the slope (derivative) of the sigmoid at a point z^ℓ can be calculated simply using the value of the sigmoid itself at that point. But let's see the graph



we can see the the derivate of the logistic function is very flat! When the input z^ℓ is big (negatively or positively) the line of sigmoid is flat this, leading the derivative to be almost zero.
If the derivative is zero, the weight updates become zero ($w_i = w_i + 0$), and the model stops learning. This phenomenon, in which the gradients become very small, is known as the *Vanishing Gradient*

The learning process

The learning process for logistic regression is iterative. Since we cannot find an analytical solution that maximizes the log-likelihood $l(w)$, we must find the optimal parameters w using an iterative method like Gradient Ascent. The process involves starting with an initial guess for the weights (e.g., all zeros) and repeatedly applying an *update operation*. Each iteration, or step, moves the weights in the direction of the gradient, thereby increasing the log-likelihood.

Definition 4.10.4: Update Rule (Gradient Ascent)

The update rule for a single weight w_i at each iteration is given by:

$$w_i \leftarrow w_i + \mu \sum_{\ell} x_i^{\ell} \cdot (y^{\ell} - P(Y = y^{\ell} | x_i; w_i))$$

Where:

- w_i is the weight for the i-th feature.
- μ is the *learning rate*, a small constant that controls the size of each step.
- $\sum_{\ell}(\dots)$ is the gradient component $\frac{\partial l(w)}{\partial w_i}$, summed over all training samples ℓ.
- $(y^{\ell} - P(\dots))$ is the *error* for sample ℓ: the difference between the true label and the model's prediction.

This operation is iterated until the model converges, which can be defined as the point where the accuracy on a held-out testing dataset is satisfactory, or when the magnitude of the update (the increment) falls below a small threshold ϵ.

Regularization

To improve the model's ability to generalize to new data and prevent overfitting, a *regularizer* (also known as a prior) is frequently added to the update rule. This acts as a penalty on large weights.

Definition 4.10.5: Update Rule with Regularization

The modified update rule includes a decay term:

$$w_i \leftarrow w_i - \mu\lambda|w_i| + \mu \sum_{\ell} x_i^{\ell} \cdot (P(Y = y^{\ell} | x_i; w_i) - y^{\ell})$$

Note:

The slide shows a sign flip in the gradient term, which implies a switch from maximizing log-likelihood (Gradient Ascent) to minimizing a negative log-likelihood (Gradient Descent). The logic remains the same: the regularizer always pushes the weight towards zero.

The new term $-\mu\lambda|w_i|$ is a penalty proportional to the magnitude of the weight w_i, controlled by a new hyperparameter λ.
This regularization has two main benefits:

- It helps to keep the parameter values w_i small, or close to 0.
- By penalizing complex models (those with large weights), it tends to reduce overfitting and improve the model's performance on unseen data.

4.11 Generative vs Discriminative models

In short, the differences between the two machine learning models are as follows:

- **Discriminative**: used to learn the **decision boundary** that separates different classes in the data.
- **Generative**: used to model the **underlying data distribution**.

Both of these models can be used for data classification (estimating $f : X \to Y$ or $P(Y|X)$), but only generative models are used to generate new data or for data augmentation.

Example 4.11.1 (Generative vs Discriminative classifiers)

Discriminative (e.g. Logistic regression)

- Assume a distribution for $P(Y|X)$ or a shape for the discrimination function
- Estimate the parameters for $P(Y|X)$ (or for the discrimination function) from training data

Generative (e.g. Naive Bayes)

- Assume a distribution for $P(X|Y)$ and $P(X)$
- Estimate parameters for $P(X|Y)$ and $P(X)$ from training data
- Use Bayes' rule to infer $P(Y|X)$

4.11.1 Linear models

Both generative and discriminative models can be *linear*, meaning that features are considered independently from one another.
These models become usefull when the training data has a large number of (independent) features, seen as training points in high dimensional space presumably have a very sparse distribution, making it plausible to discriminate classes using hyperplanes.
Said models are usually faster to train relative to more complex non-linear ones. Linear models we've studied so far include:

- Logistic regression: although the function σ is not linear, the way its input varies given the features is.
- Naive Bayes: as a classifier, it's linear when considering multinomial or bernullian distributions and in some cases with Gaussian distribution.

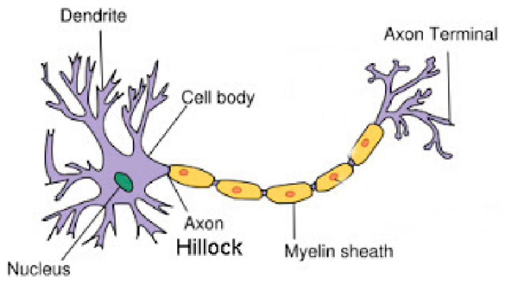
Chapter 5

Neural Networks

Only known way to implement deep learning.

5.1 The Artificial Neuron

A *Neural Network* is a structure that closly mimics the brain, as it's composed by interconnected layers of *artificial neurons* that work together to create an output.
An artificial neuron has a set of inputs that are summed with a bias, and the output is calculated using an activation function. The function mustn't be linear, because we want to sequencially compose neurons and the composition of linear functions is just another linear function and doesn't add any complexity (could be calculated with just one neuron).
There are different activation functions (threshold, logistic, hyperbolic, rectified linear (newest used in AlexNet, big jump)).
The name neuron comes from the biological counterpart (obviously) which works in a similar way:



- Dendritic tree: connections with other neurons (synapses)
- In the body the inputs are summed and passed through the axon hilock, which performs a sort of thresholding before being passed to other neurons

Comparing artificial neural networks to our brains, the number of neurons can be similar ($2 \cdot 10^{10}$for the biggest models), but the sheer size doesn't necessarily correlate to a more intelligent system (other animals have bigger brains), we actually don't really know what else there is. The switching time for real neurons is actually slower than artificial ones, seen as it's a chemical reaction and not electrical. Each neuron is connected to many other neurons in the brain (10^{4-5}), reaching reaction times < 100, so the brain isn't very deep (number of intermediate nodes) and it's very parallelised.

5.2 Topologies

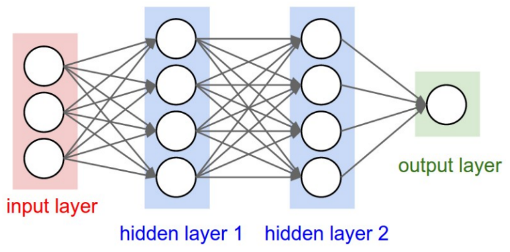
Definition 5.2.1: Feed-forward

Acyclic networks with unidirectional data flux.

Note:
Our brains are cyclic networks (also called *recurrent*), but we won't be studying this case.

5.2.1 Layers

The network is built by sets of structured neurons that are combined to build the whole.



Based on the number of layers, we have that a NN is:

- **Shallow**: if there is only one hidden layer.
- **Deep**: if there are multiple hidden layers.

Dense layers

Definition 5.2.2: Dense layer
A layer in a NN is called **dense** when neurons in adjacent layers are all interconnected.

For each dense layer, each neuron has an input I^n, weights W^n and a constant bias B^1 from which an output O^1 is calculated.

$$I^n \cdot W^n + B^1 = O^1$$

This operation can be parallelised using the whole layer with a matrix of weights $W^{n\times m}$ (one array of weights for each neuron) and a vector of biases, using the same inputs over all the neurons. The input is a vector in multiple dimensions (tensor), and after algebraic manupulations a new tensor is returned.

Note:
Dense layers usually work on *flat* (unstructured) inputs, where the order of the elements is irrelevant.

5.3 Features and deep features

TODO: manca sta parte qua che fa jumpscare alla fine della lezione del 24 ottobre ma che il 6 novembre non la fa TODO: boh ma sta roba qua non so dove sta, forse non l'ha fatta TODO: @bastini plz finiscie te, che altrimenti questi mi diventano ApuntiDelGiolaBambas Re: Yes yes ora mi sta mettendo a fare

5.4 Successful Applications

5.4.1 Image Processing

ImageNet

High resolution labeled image dataset covering 22k object classes.
A competition is held each year to classify given images.
Relevance
Classfication models are typically composed of two parts:

- **Front-end**: extracts features from the input image and convers them into a vector

- **Back-end**: uses the extracted features for classification

ImageNet is still importand because a front-end, **pre-trained over ImageNet**, can be reused for a lot of different applications (*transfer learning*).

Object Detection

YOLO (You Only Look Once) is a real-time object detection system. It's a smilar problem to image classification, but has the extra functionality of localizing the object using a bounding box (smallest possible rectangle containing the object) inside the image.
The model actually returns many thousands of possible bounding boxes from an image, each with its own weight. It's up to another piece of software to select which ones to keep.

Image Segmentation

Segmentation adds even more detail to object detection by classifying each pixel (so basically object detection but with the bounding box being the actual outline of the object).

Key-points detection

Finds specific areas of an object to track (e.g. eyes, joints, ecc. to analyse human position/movement).

Applications

- Medical imaging
- Autonomous driving
- Pose estimation
- Activity recognition
- Video surveillance
- ...

5.4.2 Generative Modeling

The objective of generative modeling is to learn the **distribution** $P(X)$ of training data - that is how points are distributed inside the feature space thay inhabit.
Typically, we aim to build a **generator** able to **sample** points according to the learned distribution.
But NN are **deterministic systems**, so how can we simulate a stochastic sampling procedure?

The generator

We know how to build **pseudo-random generators** for simple, known distributions (e.g. Gaussian).
So the problem reduces to learn a **transformation** mapping the known distribution to the actual distribution $P(X)$.
Morally, the generator learns $P(X|z)$, where z is the "**latent**" rapresentation of X.

Ancestral sampling

Generation is thus a two stage process:

- Sample z according to a known prior distribution
- Pass z as input to the generator, and process it to get a significant output.

Generative models (GANs, VAEs, Diffusion, ...) differ in the way the generatoris trained.

The latent space

The source space is the so called latent space.
Each latent point z contains **all information** needed to generate a complete sample, hence it can be seen as an **internal encoding** (latent representation) of the given sample.
Latent values must be disseminated with a **known, regular distribution** in their space (the so called prior distribution).
Important things to keep in mind:

- **Any** face is somewhere in the latent space
- Generation is a **continuous** process: small modifications of the encoding produce small modifications of the output

Note:
Latent space points are also often compressed versions of the explicit representation, meaning they take up less phisical space.

Conditional generation

The generator tries to model $P(X|z, c)$ where c is a condition integrating the latent encoding z. This condition can be:

- A label
- A segmentation
- A text prompt
- Another image
- A sequence of frames
- ...

5.4.3 Natural Language Processing

Main subfields in NLP

- **Natural language Understanding**: focus on interpreting and extracting meaning rom human language, includes
  - sentiment analysis
  - named entity recognition (NER)
  - question answering
- **Natural Language Modeling**: focus on producing human-like text, includes
  - text summarization
  - dialogue generation
  - story writing
  - retrieval augmented generation

Speech recognition/generation is an additional topic bridging text and audio.

Embeddings

NLU instruct models to understand context, ambiguity and nuances in language.
All previous tasks require a meaningful representation of words, provided by **embeddings**.

Definition 5.4.1: Embeddings

Mappings of words/sentences into a high-dimensional vector space. They help models understand relationships beween words, such as similarity, analogy and context.

These embeddings form the latent space of text representation, where the same transformations (e.g. going from singular to plural) have the same transformation vector.

Bridging text and images

CLIP (Contrastive Language-Image Pretraining) is a recent technology that allows the joint analysis of text and images.
It's trained on a dataset of text-image pairs, both mapped to a shared latent space where similar pairs have closer embeddings.

Language modeling

Images involve spatial relationships, often modeled **holistically** or in parallel.
NLP operates on **sequences** of tokens (words or subwords) where order is critical.
Given a sequence x of tokens $x_1x_2...x_n$, we are interested in modeling the mechanisms underlying their concatenation.
Generative models estimate the joint probability of a text sequence

$$P(x_1x_2...x_n)$$

This can be broken into **conditional probabilities** using the **chain rule**

$$P(x_1)P(x_2|x_1)P(x_3|x_1x_2)...P(x_n|x_1...x_{n-1})$$

In practice, you can train the model to guess the next token completing the given sequence, enabling **sequential generation**(each token depends on its predecessor.
Different possible approaches are:

- **N-grams**: estimate probabilities based on fixed window size (ex. Bigram model uses $P(x_n|x_{n-1})$). The main limitation is that it can't capture long-range dependencies.
- **Deep NN**:
  - **Recursive NN**: introduce sequence processing but strugle with long term dependencies
  - **Transformers**: with **self-attention** they can handle long-range context efficiently and are the foundation for modern NLP

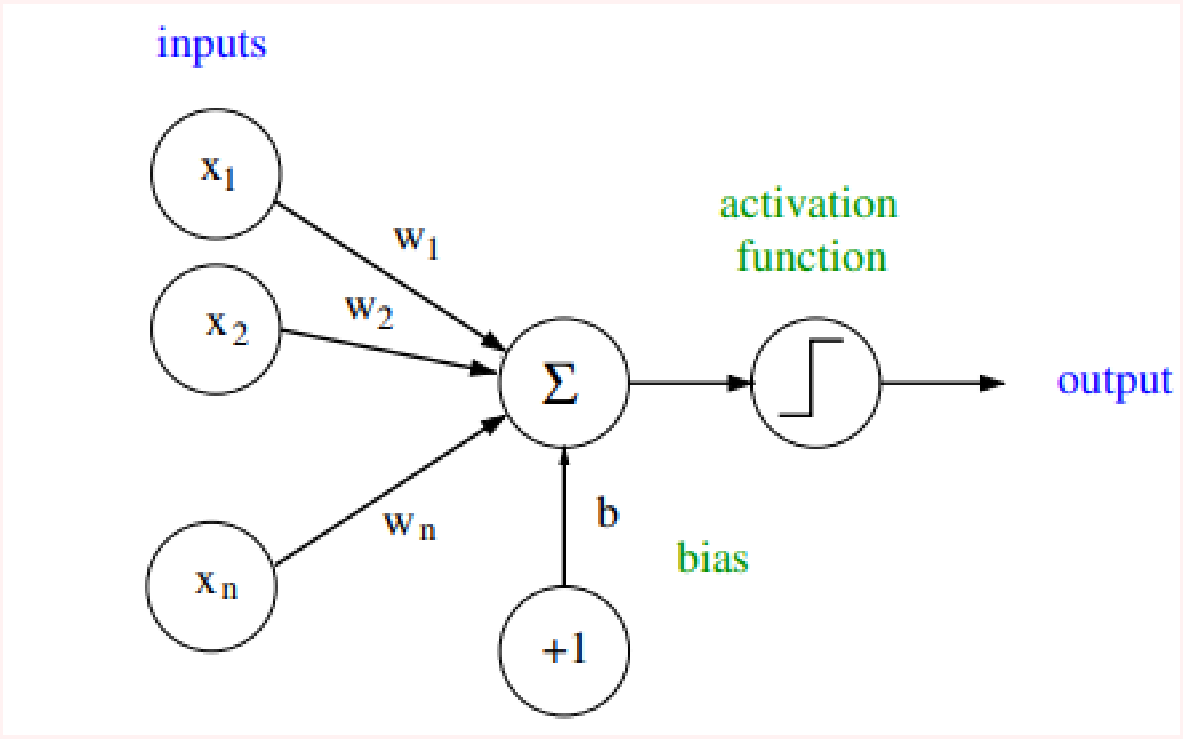
5.5 Expressiveness

This part begins with an important question: *Can we compute any function by means of a Neural Network?*. If so, every function must have a corresponding set of weights that, when applied to a Neural Network, induce the calculation of said function. But before that, we must first ask: *Can we compute any function with a single neuron?*
These questions are related to the **Expressiveness** of Neural Networks, that is the capacity of a Neural Network to approximate any function

5.5.1 Perceptron, the single layer case

A single neuron is called a **Perceptron**. It is the simplest form of a Neural Network.

Definition 5.5.1: Perceptron with binary threshold



A perceptron is a function $f : \mathbb{R}^n \to \{0, 1\}$ defined as:

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i \cdot x_i + b \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

where $x \in \mathbb{R}^n$ is the input vector, $w \in \mathbb{R}^n$ is the weight vector and $b \in \mathbb{R}$ is the bias.

Note:
It's easy to notice the the the bias b set the position of threshold

Hyperplanes

For major information about hyperplanes see the notes of "Ottimizzazione combinatoria di Alex Basta e Qua Qua dancer"

Definition 5.5.2: Hyperplane

We can define a hyperplane in $\mathbb{R}^n$ as the set of points $x \in \mathbb{R}^n$ that satisfy the equation:

$$\sum_{i=1}^{n} w_i \cdot x_i + b = 0$$

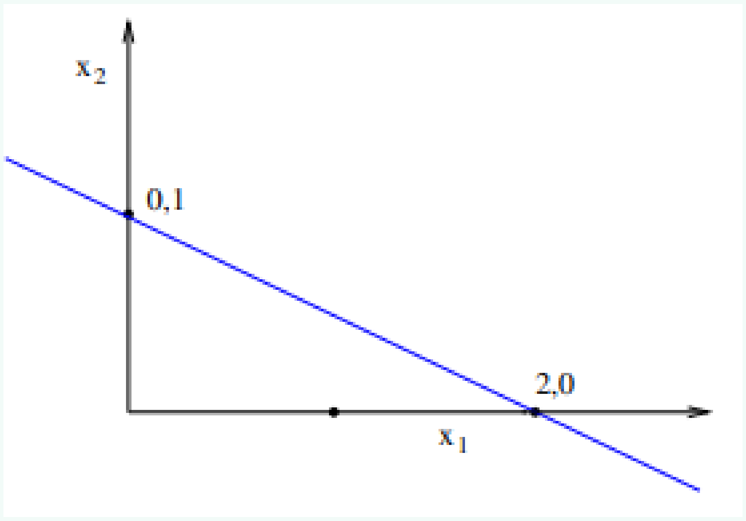
where $w \in \mathbb{R}^n$ is the normal vector to the hyperplane and $b \in \mathbb{R}$ is the bias

x_1	x_2	NAND(x_1, x_2)
0	0	1
0	1	1
1	0	1
1	1	0

Example 5.5.1 (Hyperplane in two dimensions)
We can consider this equation in $\mathbb{R}^2$:

$$-\frac{1}{2}x_1 + x_2 + 1 = 0$$

The graph is a line in the plane with normal vector $w = \left(-\frac{1}{2}, 1\right)$ and bias $b = 1$.



As we can see from the example, a general property of hyperplanes in $\mathbb{R}^n$, the hyperplans divedes the space in two half-spaces. By the definition of perceptron 5.5.1 ($\sum_i x_i x_i + b = 0$) this gives value 1 to all the points in one half-space and value 0 to all the points in the other half-space.

Note:
"above" and "below" can be inverted by just inverting the parameters:

$$\sum_{i=1}^{n} w_i \cdot x_i + b \geqslant 0 \iff \sum_{i=1}^{n} -w_i \cdot x_i + (-b) \leqslant 0$$

Computing logical connectives

For investingating the expressiveness of a single perceptron, a logical way is to see if it can compute logical connectives.

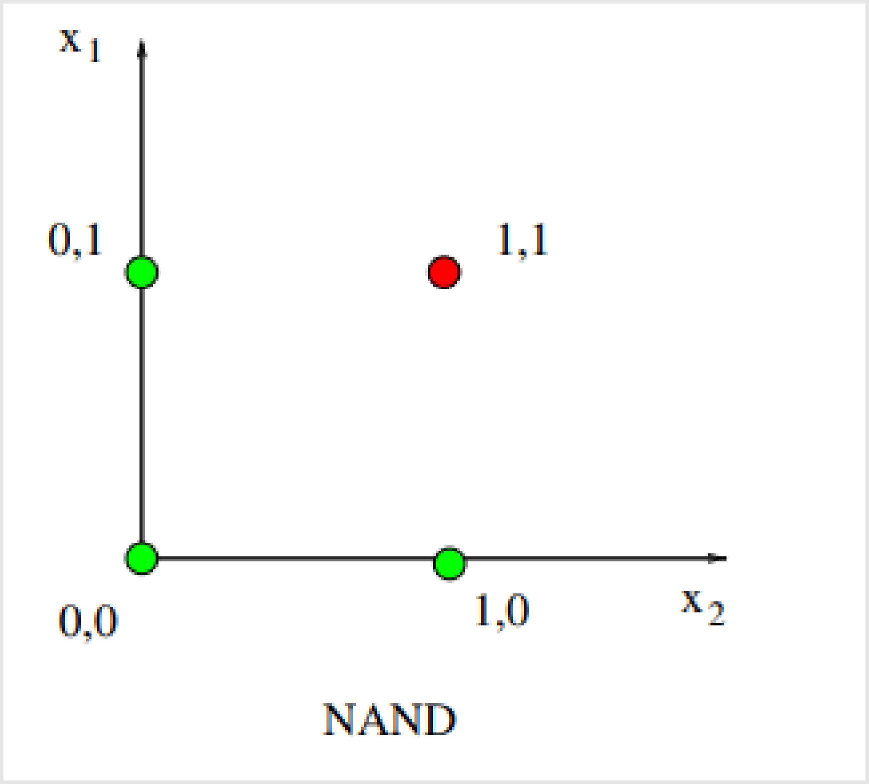
NAND case Ramarking NAND:

For computing NAND with a perceptron, we need to find weights w_1, w_2 and bias b such that:

$$\texttt{nand}(x_1, x_2) = \begin{cases} 1 & \text{if } w_1 \cdot x_1 + w_2 \cdot x_2 + b \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

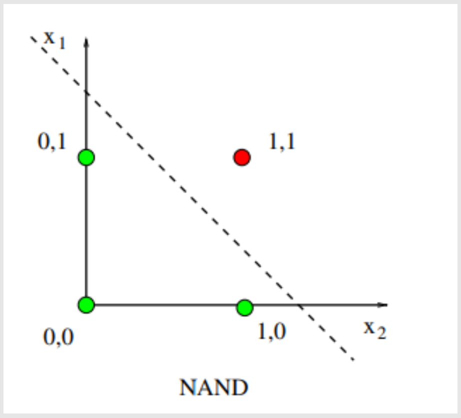
Note:

This is same as asking if we can a straight line to separate green and red points

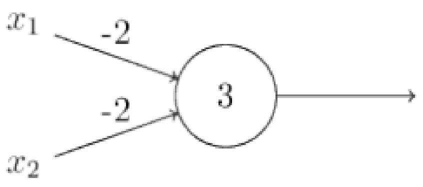


Remember: doing that means putting one to one side of the line all the green points and on the other side all the red points
Well, the answare is yes! We can choose for example:

$$w_1 = -2, \quad w_2 = -2, \quad b = 3 \Rightarrow 3 - 2x_1 - 2x_2 = 0$$



Watching the notes what we have is the NAND-perceptron:



x_1	x_2	$\text{XOR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

$$\texttt{nand}(x_1, x_2) = \begin{cases} 1 & \text{if } -2 \cdot x_1 - 2 \cdot x_2 + 3 \geqslant 0 \\ 0 & \text{otherwise} \end{cases}$$

Note:

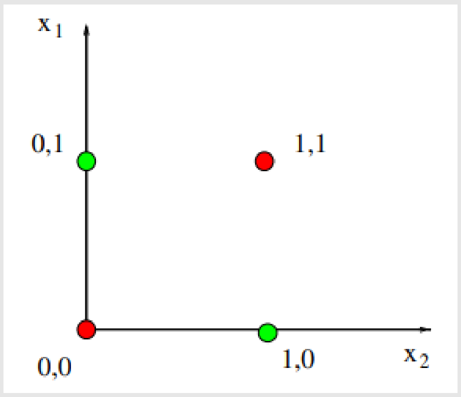
The set of logical operators that contains only the NAND gate is actually logically complete[a], but it needs to be composed to do so. Meaning a single layer being able to simulate such a function does not imply that the perceptron is logically complete, as we'll now see.

[a]A complete set of logical operators can calculate any boolean function.

XOR case XOR is defined as:

Note:

Same as asking: Can we draw a straight line separating red and green points?



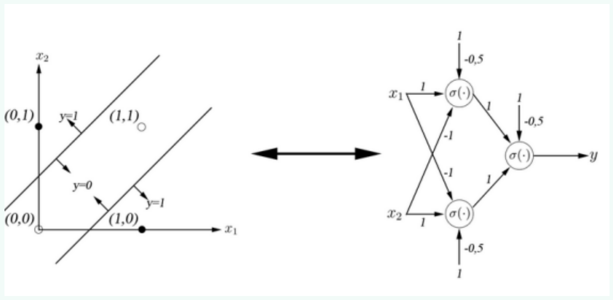
Solution? No solution :(

The XOR function is not linearly separable, so it cannot be computed by a single perceptron. This is a limitation of linear methods (like logistic regression), seen as it's impossible to compare features with one another.
In conclusion, single layer perceptrons are not logically complete.

5.5.2 Multi-layer perceptrons

A perpteptron, as it is demonstrated before, can compute NAND and it is known that NAND is logically complete, meaning that any logical function can be computed by a combination of NAND functions. So why it is not possible to compute XOR or others with a single perceptron? why why perceptrons are not *complete*? The answer is that a single perceptron can only compute *linearly separable functions* and they need to be combined in order to compute more complex functions.

Example 5.5.2 (Multi-layer perceptron for XOR)



The multi-layer perceptron above computes XOR using 3 perceptrons that compute NAND, OR and AND respectively.

So how deep does a neural network need to be to calculate a certain logic expression? We can look at the depth as the number of nested logical connectors, or as the depth of the tree generated by the expression. In conjunctive or disjunctive normal form, all trees have depth of 3, and each logic formula can be transformed in one of these forms. In reality, we can remove the negation layer and use only 2 layers by using more expressive connectors.

Note:
It's thanks to the activator function that the composition of layers adds expressivity, as a simple composition of linear functions is still linear and doesn't add complexity.

A significant theoretical result, known as the universal approximation theorem, states that even **shallow networks** (those with a single hidden layer) are **already complete**. This means that a shallow network can, in principle, approximate any continuous function to an arbitrary degree of accuracy, given a sufficient number of neurons in its hidden layer.
This naturally raises a critical architectural question: Why go for deep networks? If a shallow architecture is theoretically sufficient, what is the practical advantage of stacking multiple layers?

5.5.3 Deep Networks

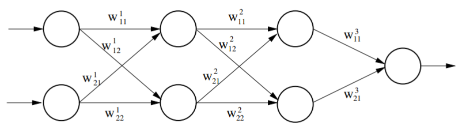
The answer lies not in *capability* but in *efficiency*. Research has demonstrated that with deep nets, the same function may be computed with less neural units Deep networks build a hierarchical representation of features—where each layer learns progressively more complex abstractions based on the previous—which is a far more parameter-efficient way to represent complex functions compared to the "brute force" approach of a single, massive hidden layer.
In fact, there are some cases where transforming a logical expression into normal form can create an exponentially longer expression, making it much less efficient.

The essential role of Activator Functions It is crucial to understand that this expressive power—whether in shallow or deep networks— originates from a single source. **Activation functions play an essential role**, as they are the **only source of nonlinearity** in the model.
A neural layer is composed of a linear transformation (the weighted sum and bias) followed by a non-linear activation. If this non-linear step were removed, the network would collapse. **Composing linear layers not separated by nonlinear activations makes no sense**, as the composition of any number of linear functions is, itself, just a single linear function. Therefore, it is the activation function that enables the network to warp and fold the input space, allowing it to learn the complex, non-linear relationships required to solve problems like the XOR example.

5.6 Training

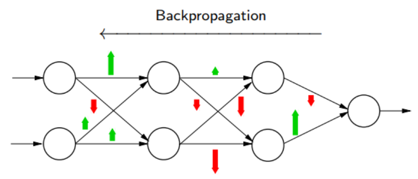
Training a neural network involves adjusting its weights and biases to minimize the difference between its predicted outputs and the actual target values. The training process can be described in three main steps:

- **Forward Pass**: Suppose to have a neural network with some configurations of the parameters θ (weights and biases)



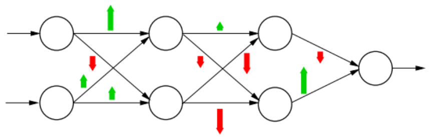
The foreard pass is defined as the process of passing the input data x through the network to obtain the output predictions then calculating the current loss relative to θ

- **Backward Pass**: The backward pass is the process of computing the gradients of the loss function with respect to each weight and bias in the network (next chapter will cover this in detail)



The algorithm for computing parameters updates is known as *backpropagation algorithm*

- **Parameters Update**: For decreasing the loss, the params need to be adjustent in different ways. The tool that allows us to establish in which way parameters should be updated is the gradient calculated during the backward pass



Note:
The cost of both the passes is relatively similar, but during the forward pass each tensor's output is memorised in order to calculate the gradients later on (as we'll see). Because of this there's a higher memory use during the training of a Neural Network.

5.6.1 The Backpropagation algorithm

For calculating this so called gradient (a vector of partial derivatives of the loss function with respect to each parameter in the network) it good remember that a neural network is a complex function resulting from the composition of many simpler functions (the layers). So a mathematical tool that is useful for calculating derivatives of composed functions is the **chain rule**

Chain roule

In order to calculate the gradients needed for the gradient descent algorithm, we need to use the **chain rule**:

Definition 5.6.1: Chain Roule

Given two functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$, the derivative of their composition $h(x) = f(g(x))$ is given by:

$$h'(x) = f'(g(x)) \cdot g'(x)$$

Eqyivalently, letting $y = g(x)$, we have:

$$h'(x) = f'(y) \cdot g'(x)$$

The derivative of a composition of a sequence of functions is the **product of the derivatives of the individual functions**. This function is iterable, so for each layer we multiply its derivative by the derivative of its forward input (which is why we need to memorise it).

Note:
Binary thresholding has null derivative everywhere (its a step function), so it would ruin the chain.

The Network as a Composite Function

Before applying we must formally define the function computed by the neural network.

Definition 5.6.2: Activation Vector at layer l

The activation vector at layer l is defined as the following function computed by the layer l:

$$a^l = \sigma(b^l + w^l \cdot x^l)$$

where:

- σ is the activation function
- $z^l = b^l + w^l \cdot x^l$ is the weighted input at layer l
- $x^{l+1} = a^l, x^1 = x$ (in fact the output of layer l is the input of layer $l + 1$)

So by the definition 5.6.1 the neural network with L layers computes the following function:

$$\sigma(b^L + w^L \cdot \sigma(b^{L-1} + w^{L-1} \cdot \sigma(\dots \sigma(b^1 + w^1 \cdot x^1))))$$

Note:
The dimension of w^l and b^l depend on the number of neurons at layer l (and $l - 1$)

Note:
All of them are *parameters* of the models

Backpropagation Rules in Vectorial Notation

The Backpropagation algorithm applies the chain rule to this composite function to efficiently compute the gradient of an error function E (e.g., Euclidean distance) with respect to all parameters. But first of all it's useful to define the **error at layer** l defined as the vector of partial derivatives of the error E with respect to the weighted input z^l. Formally:

Definition 5.6.3: Error derivative at layer l

The error derivative at layer l is defined as:

$$\delta^l = \frac{\partial E}{\partial z^l}$$

where $z^l = b^l + w^l \cdot x^l$ is the weighted input at layer l

This δ term represents the error signal at that layer. The algorithm is then defined by the following four equations:

1. **Error for the output layer (L):**

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

   This calculates the initial error signal by combining the gradient of the loss function ($\nabla_{a^L} E$) with the derivative of the final activation function ($\sigma'(z^L)$)

2. **Error backpropagation:**

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

   This is the core rule that *propagates* the error backward. The error from the next layer (δ^{l+1}) is passed back through the weights (W^{l+1}) and combined with the derivative of the current layer's activation function

3. **Gradient for the bias:**
$$\frac{\partial E}{\partial b_j^l} = \delta_j^l$$
The gradient for any bias is simply the error signal δ at that neuron.

4. **Gradient for the weight:**
$$\frac{\partial E}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$
The gradient for any weight is the activation from the source neuron (a_k^{l-1}) multiplied by the error signal at the destination neuron (δ_j^l)

The vanishing gradient problem

By the chain rule, the derivative is a long sequence of factor,where these factors are, alternately

- Derivatives of activation functions
- Derivate of linear functions (the weights)

Backpropagation alghorithm uses the chain rule to compute gradients efficiently, so the gradient used for an initial state is the result of a lot of factors. These factors include the activateion functions. For many years the activation function used was the sigmoid (continuous version of a binary threshold), whose derivative is always less than 1.

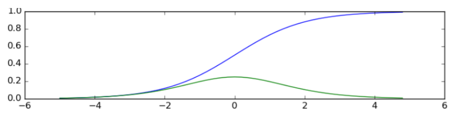


Figure 5.1: Blu line is logistic function (sigmoid), green line is its derivative

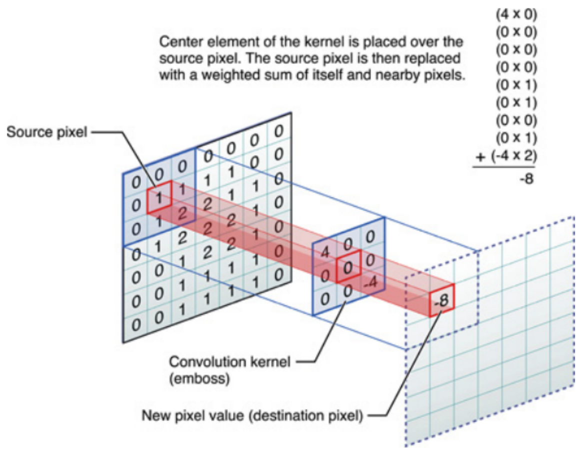
This graph is crucial for understanding the vanishing gradient problem. As we can see from the graph, the derivative of the sigmoid is always not greater than 0.25 and it's flat. The backpropagation multiplies a lot of these very small factors together, so the result is that the gradient becomes very small (ex. $gradiente = (\cdots \times 0.25 \times 0.23 \times 0.25 \times 0.21 \times \ldots)$) as it propagates backward through the layers. This means that the weights in the earlier layers receive very small updates during training, making it difficult for the network to learn effectively. If the gradient is close to zero, learning is impossible.
In contrast, the Relu function doesn't have this problem as its derivative is either 0 or 1.

5.7 Filter and convolution

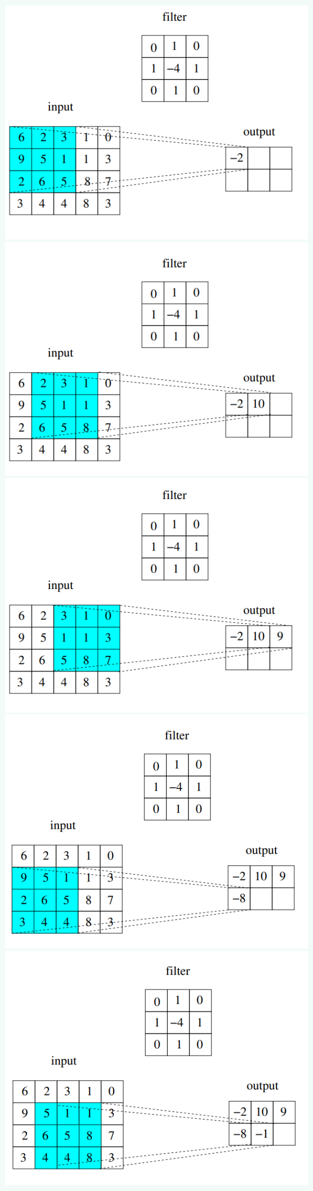
The convulution is an operation between two functions that produces a third function that expresses how the shape of one is modified by the other. In image processing, the convolution is used to apply filters to images, such as blurring, sharpening, edge detection, and more. In machine learning, convolutional neural networks (CNNs) use convolutional layers to automatically learn and extract features from images preserving spatial relationships. Mathematically

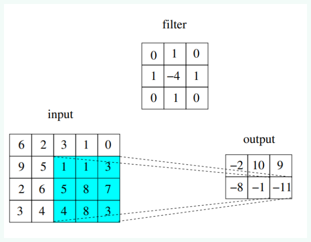
Definition 5.7.1: convulution operation

Fir a bidimensional input image I and a filter (or kernel) $K^{m\times n}$, the convolution operation is defined as:
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Example 5.7.1


filter
input
output



Note:
In a convolutional layer, each neuron produces a single value of the output feature map, which appears as one "pixel" in the resulting matrix.

In the neural networks, each neruon is linked to the first hidden layer. If we had a 1000×1000 pixel image ($1M$ inputs) and a hidden layer of 1000 neurons, we would need one billion weights (10^9 connections)— impossible to manage computationally. One of the main carateristics of convutional layers is the *local connectivity*, each neuron is connected only to a small region of the input image, called the receptive field. This drastically reduces the number of parameters and computations required, making it feasible to process high-dimensional inputs like images. Another important characteristic is *weight sharing*, where the same set of weights (the filter or kernel) is used across different spatial locations of the input. This means that the same feature can be detected regardless of its position in the image, enhancing the model's ability to generalize and reducing the number of unique parameters that need to be learned.
With a cascade of convolutional filters and pooling layers, the network can learn hierarchical features, from low-level edges and textures in the early layers to high-level object parts and entire objects in the deeper layers. This hierarchical feature learning is crucial for tasks like image classification, object detection, and segmentation.

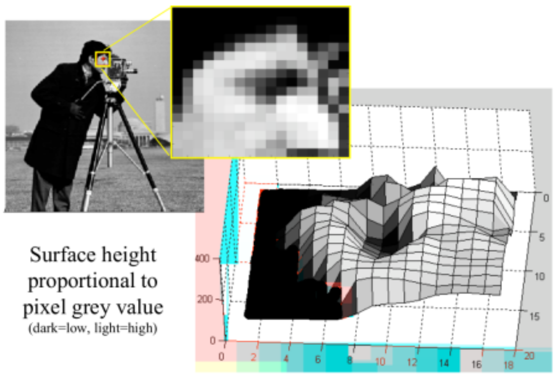
5.7.1 About the relevance of convolutions for image processing

An image is coded as a numerical matrix (array) grayscale (0-255) or rgb (triple 0-255)

$$\begin{bmatrix} 207 & 190 & 176 & 204 & 204 & 208 \\ 110 & 108 & 114 & 112 & 123 & 142 \\ 94 & 100 & 96 & 121 & 125 & 108 \\ 95 & 86 & 81 & 84 & 88 & 88 \\ 69 & 51 & 36 & 72 & 78 & 81 \\ 74 & 97 & 107 & 116 & 128 & 133 \end{bmatrix}$$

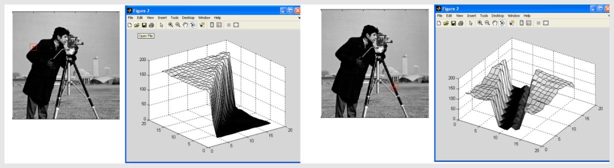
Img as surfaces

We can see an image as a surface in 3D space, where the x and y coordinates represent the pixel positions, and the z coordinate represents the intensity (brightness) of the pixel. In grayscale images, this intensity ranges from 0 (black) to 255 (white). In color images, each pixel has three intensity values corresponding to the red, green, and blue channels.



Note:

Edges, angles, ...: points where there is a discontinuity, i.e. a fast variation of the intensity



More generally, are interested to identify *patterns* inside the image. The key idea is that the kernel of the convolution expresses the pattern we are looking for.

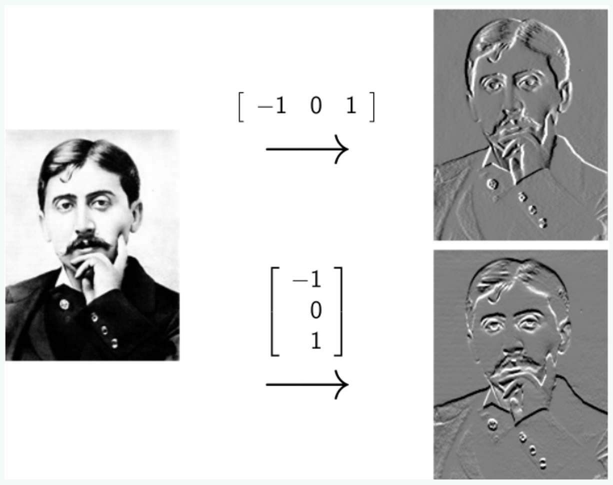
Example 5.7.2 (Finite Derivate)

Suppose we want to find the positions inside the image where there is a sudden horizontal passage from a dark region to a bright one. The pattern we are looking has that kernel:

$$\begin{bmatrix} -1 & -1 \end{bmatrix}$$

or, varying the distance between pixels:

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$


$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \longrightarrow$
$\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \longrightarrow$

5.7.2 Discovering patterns

instead of using human designed pre-defined patterns, let the net learn them. This is particularly useful in deep networks:

- stacking kernels we can learn more complex patterns
- adding non-linear activations we synthesize complex, non-linear kernels

Receptive field

Definition 5.7.2: receptive field

The receptive field of a neuron in a convolutional layer is the specific region of the input image that influences the neuron's output. It defines the spatial extent of the input data that the neuron "sees" and processes.

It is equal to the dimension of an input image producing (without padding) an output with dimension 1

Note:
A neuron cannot see anything outside its receptive field

5.7.3 Tensors and Dimensions

Multidimensional array of floats. The typical tensor for a 2D image has 4 dimensions:

$$\text{batchsize} \times \text{width} \times \text{height} \times \text{channels}$$

Dense processing

Unless stated differently, a filter operates on all channels in parallel. So, given an input layer of depth D, the size of a kernel $N \times M$ will be:

$$N \times M \times D$$

The kernel is tasked with simultaniously mapping cross-chanel correlations and spatial correlations.

Feature Maps

Each kernel produces a single feature map, which is stacked along the chanel dimension (the number of kernels is equal to che channel-depth of the next layer).
The spatial dimensions of the feature maps depends on:

- **Padding**: extra space added around the input
- **Stride**: how the kernel moves over the input

The dimension of the output is given by the following formula

$$\frac{W_{\text{in}} + 2P - K}{S} + 1 = W_{\text{out}}$$

Where

- W is the spatial dimension of the I/O
- P is the padding
- K is the kernel size
- S is the stride

By reverting this formula (without padding) we can find the receptive field

$$W_{\text{in}} = K + S \cdot (W_{\text{out}} - 1)$$

5.7.4 Pooling

Another way to reduce the spacial dimension of the input/increase the receptive range of neurons is to use pooling layers.
These layers simply take the max/avg of the values in a kernel region (with the default stride being the dimension of the kernel). Pooling layers have the advantages that:

- they reduce spatial dimensions
- they provide some tollerance to translations

5.8 CNNs and Transfer Learning

Transfer learning consists of using pretrained models as ”feature-extractors” and modifying only the classifier or some of the higher layers (fine tuning).

5.9 Autoencoders

An **autoencoder** is a network trained to reconstruct input data out of a learned internal representation. Usually, the internal representation has lower dimensionality w.r.t. the input data. This compression is made possible by exploiting **regularities** (correlatins) in the features describing the input data.

Note:
The higher the **entropy** of the structure of the input data, the lower the amount of possible compression is.

The type of compression is:

- **Data-specific**: only works well on data with strong correlations.
- **Lossy**: the output is degraded w.r.t. the input.
- **Directly trained**: on unlabeled data samples (**self-supervised** training).

Because of its lossy nature, autoencoders aren't great for compression. But they can be used for:

- Data denoising
- Anomaly detection
- Feature extraction
- Generative models

5.10 Generative Models

Definition 5.10.1: Generative Model

A model that tries to learn the actual distribution p_{data} of real data from available samples (training set).

We can either try to

- explicitly estimate the distribution
- build a generator able to sample according to p_{model}

The second approach is used the most.

5.10.1 Latent-variable Models

The generative process is modelled as

$$P(X) = \int P(X|z)P(z)dz$$

where

- $P(z)$ is the **prior distribution** over latent variables (typically Gaussian)
- $P(X|z)$ is the likelyhood function approximated by the deteministic generator $G(z)$.

Generators decode points in the latent space to produce output. Seen as they're continuous, we can identify **trajectories** for editing and manipulating the latent representation of the data directly.

5.10.2 Training Generators

We need to model $P(X|z)$, which associates an image X with a latent variable z. The problem is that we don't have labled data pairs (z, X):

- If we start with a real image, we need to determine its corresponding latent encoding in some way.
- If we start with an encoding, we must decide what image to generate and identify the factors that could drive this generation.

There are three main classes of generators based on training method:

- **Variational Autoencoders** (VAEs)
- **Generative Adversarial Networks** (GANs)
- **Diffusion Models**

5.10.3 Variational Autoencoders

The generator is coupled with an **encoder** producing a latent encoding z given X distributed according to an inference distribution $Q(z|X)$.
The loss function aims to:

- minimize the reconstruction error between X and $\hat{X}$
- bring the marginal inference distribution $Q(z)$ close to the prior $P(z)$

It's approach is reminiscent of the autoencoder idea.

5.10.4 Generative Adversarial Networks

The generator is coupled with a **discriminator** trying to tell apart **real** data from **fake** data produced by the generator.
The detector and generator are trained together, the loss function aims to:

- instruct the detector to spot the generator
- instruct the generator to fool the detector

This behaviour can be formalized as a **Min-Max game**, where the detector wants to maximize the amount of correct guesses and the generator wants to minimize this amount

$$Min_G Max_D V(D, G)$$

Where

$$V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

The two terms represent:

- The negative cross-entropy of the discriminator w.r.t. the true data distribution p_{data}
- The negative corss-entropy of the "false" discriminator w.r.t. the fake generator (where z is a random sample in the latent space)

Training

In practice it's better to train G to maximize

$$\log(D(G(Z)))$$

It's also possible to first train the discriminator (freezing the generator) and vice-versa

Algorithm 1: Training

```
1 repeat;
2 // Train the discriminator D freeze Generator's weights $\Theta_G$;
3 sample a batch of real data $x \sim P_{\text{data}}$;
4 sample a batch of noise $z \sim N(0,1)$;
5 update the Discriminator's weights $\Theta_D$ by stochastic ascent on:
      $\nabla_{\Theta_D}[V(D,G)]$
  // Train the generator G freeze Discriminator's weights $\Theta_D$;
6 sample a batch of noise $z \sim N(0,1)$;
7 update the Generator's weights $\Theta_G$ by stochastic ascent on:
      $\nabla_{\Theta_G}[E_{z \sim N(0,1)} \log D(G(z))]$
  until converged
```

5.11 Diffusion

Diffusion (or denoising) models work by using two basic processes:

- From left-to-right we have **forward diffuson**, where data is progressivly corrupted by adding noise at each step.
- From right-to-left we have **backward diffusion**, where the model learns to denoise the corrputed immage.

Each backward step involves two operations:

- **Denoising**: try to remove all the noise in the image.
- **Noise reinjection**: add back a small amount of noise

5.11.1 Denoising

It's the main and only trainable component of the model. It's used to predict the noise contained in an image.
It's unique for each step t, seen as there's a fixed amount of noise the image should have at each step (the signal rate).
So the network takes in an input:

- A noisy image x_t
- A signal rate α_t that expresses the amount of the original signal left in the data

And tries to predict the noise:

$$\epsilon_\theta(x_t, \alpha_t)$$

The predicted image is thus

$$\hat{x}_0^t = (x_t - \sqrt{1-\alpha_t}\epsilon_\theta(x_t, \alpha_t))/\sqrt{\alpha_t}$$

5.11.2 Noise reinjection

Can be **deterministc** (DDIM) or **probabilistic** (DDPM):

- DDPM reinjects a random amount of noise each time
- DDIM reinjects the same noise that was removed but at a lesser intensity, so it's deterministic given the starting noise.

5.11.3 Training

Consists of running these steps until converged:

- Take a sample (clean image)
- Randomly select a timestamp $t \in [1, ..., T]$
- Create random Guassian noise ϵ
- Corrupt the sample with the signal rate α_t and the gaussian noise
- Predict the error α_θ
- Use backpropagation to better the prediction

5.11.4 Sampling

To create a new image from a trained diffusion model:

- Generate random Gaussian noise
- Starting from timestamp $t = T$, compute the noise at the current level
- Denoise the image completely
- Reinject noise at rate α_{t-1}
- Repeat untill $t = 1$

5.11.5 Differences

Diffusion models operate in the full visible space, so it can generate complex textures without blurring or loosing randomness.
But, it's slow and memory intensive.

5.11.6 Stable Diffusion

Move the diffusion to a lower-dimensional latent-space using a pre-trained VAE