Notes for "Machine Learning for NLP 1" course

Work in Progress

2023-2024

Marie Candito
Computational Linguistics - M1 - Université Paris Cité

Last modification: September 14, 2023

Contents

1 Introduction

Machine learning is everywhere in NLP, as in other fields such as computer vision, bioinformatics, finance etc...
In this course, we will introduce machine learning concepts for the task of automatic classification. This generic task can be used to perform various more specific NLP tasks, such as spam detection, part-of-speech tagging, sentiment analysis etc...
Mathematical details are only provided for the essential notions.
Vocabulary in **orange** background is **essential** to know (I also sometimes provide the French equivalent within brackets).

1.1 Recommended readings

- Very pedagogic online book (unfinished though) by Hal Daumé III : An introduction to Machine Learning, http://ciml.info/
- Intro to deep learning for NLP:
  - Yoav Goldberg, "A primer on neural network models for natural language processing" 2016 http://u.cs.biu.ac.il/~yogo/nnlp.pdf
  - which ended up as a book (**?**)
  - general NLP book by Jurafsky and Martin https://web.stanford.edu/~jurafsky/slp3/
    - broad coverage, some of the chapters are a bit outdated, but many were recently revised
    - relevant for this course :
      - chapter 5: logistic regression = log-linear classification, intro to learning by gradient descent, sigmoid, softmax, cross-entropy loss
      - chapter 6: representing words as vectors ("word embeddings"), cosine, word2vec learning, evaluating word vectors
      - chapter 7: neural networks, prediction (forward propagation), computing gradients using backpropagation, neural language models
  - Jacob Eisenstein's nlp notes, "Natural Language Processing"
  - https://github.com/jacobeisenstein/gt-nlp-class/tree/master/notes

If you want to go (much) further in machine learning and deep learning:

- Ethem Alpaydin : Introduction to Machine Learning, second edition, MIT Press, 2010
- Schwartz & Ben David : Understanding Machine Learning, from Therory to Algorithms, Cambridge University Press, 2014
- Goodfellow, Bengio & Courville "Deep Learning", MIT Press, 2016 http://www.deeplearningbook.org/

2 Reminders about vectors and matrices

2.1 Vectors

- Let n be a strictly positive integer. The set of all n-tuples of real numbers forms the **$\mathbb{R}^n$ vector space**. Each of these n-tuples is called a **vector**, and is simply an ordered sequence of n real numbers.
- Each of these real numbers is called an **element** or **component ("composante")** of the vector.
- These components belong to $\mathbb{R}$, which is a field ("corps") over which the vectorial space $\mathbb{R}^n$ is defined. Other fields (such as $\mathbb{Q}$) may define other kinds of vector spaces but we will stick here to the $\mathbb{R}$ field and the $\mathbb{R}^n$ vector space.

- For a given vector space, a **scalar ("scalaire")** is simply a member of the field over which the vector space is defined. Hence for us, **scalar = real number**.
- **Notations:** In general, we will use bold face lower case letters for vectors, e.g. $\mathbf{v}$. Normal small letters for scalars. We will use simple subscripts for components : the jth component of a vector $\mathbf{v}$ will be noted v_j or $v_{[j]}$.
- Each **position** or **rank** of the components of a vector is associated to a certain semantics, either explicitely or implicitly.
  - For instance, colors can be defined using a 3-dimensional space with each position corresponding to the quantity of the three primary colors

2.1.1 Geometric interpretation

When considering the case D=2 or D=3, vectors can have a geometric interpretation.

- The $\mathbb{R}^2$ vector space, with standard basis $\mathbf{e}_1 = (0, 1)$ and $\mathbf{e}_2 = (1, 0)$ can serve to represent geometry in a plane. In an orthonormed system with origin O and basis $\mathbf{e}_1 = (0, 1)$ and $\mathbf{e}_2 = (1, 0)$, a vector $\mathbf{u} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2$ can be interpreted as the move from origin O to the extremity point with coordinates (x_1, x_2).
- The couple of points $O = (0, 0)$ and $X = (x_1, x_2)$ are a representative bipoint of the vector $\mathbf{u} = (x_1, x_2)$. Other representative bipoints (A,B) are all the pairs of points such that $x_1 = a_1 - b_1$ and $x_2 = a_2 - b_2$

2.1.2 Example of vectorial representation of a document: Bag-of-word (BOW) vectors

A central step needed in practically any current NLP system is to represent linguistic symbols by vectors, in particular words as vectors, and sequences of words (be it sentences, paragraphs, full documents, user queries ...). We talk here of document representation, but it holds for any sequence of words.

Before the advent of neural methods in NLP, documents were typically represented as **bag-of-words** vectors (**BOW** vectors).

BOW vectors for documents suppose a **vocabulary**, i.e. set of words, has been selected: for instance the set of word forms in a given corpus, or a selection among these (e.g. one can filter out the words appearing too rarely, or word forms of function words etc...).

Let $|V| = n$ be the size of the set the word vocabulary. Words are typically associated to ids from 1 to n : $V = w_1, w_2, ..., w_n$. Word of id i is noted w_i (very much used later on).

BOW vectors can be defined in $\mathbb{R}^n$, in which each position i corresponds to the word w_i in the vocabulary. For a given vector $\mathbf{v}$ representing a document d, the i-th component is the number of occurrences in d of the word w_j in d (or a function of that number of occurrences).

Since V is usually large, these vectors are said to be **high-dimensional**.

Since only a small fraction of the vocabulary is actually present in a single document, these vectors have a lot of null components: they are **sparse** .

NB: to represent a document d as a BOW vector over vocabulary V, the words in d that are not in the vocabulary are just ignored.

Rem: in neural NLP, word vector spaces associate a low-dimensional **dense** vector to each word (cf. sections 11.2 and 12), but the exact semantics of each component is unknown (it results from a machine learning process).

2.2 Matrices and tensors

Matrices are a generalization over vectors, in which a component (or cell) is identified using a position along two axes instead of one, usually called the rows and the columns.

- $\mathbf{A} \in \mathbb{R}^{r \times c}$ defines a matrix of real numbers with r rows and c columns
- we will note $A[i, j]$ the value at row i and column j (which is a scalar)
- we will note $\mathbf{A}[\mathbf{i}, *]$ the ith row, $\mathbf{A}[*, \mathbf{j}]$ the jth column (which are vectors)

Each axis has its own semantics, and each position in a given axis has its own semantics. Examples:

- For instance an adjacency matrix can be used to represent a directed graph
  - if the graph has n nodes, then a matrix in $\mathbb{R}^{n \times n}$ can be defined to represent the graph, with ones to represent edges
  - the row number (resp. column number) can correspond to nodes as source node (resp. as target nodes)
  - then $A[i, j] = 1$ if an edge from node i to node j exists, and 0 otherwise
- A set of n documents, each represented as a BOW vector using vocabulary V can be organized into a matrix $T \in \mathbb{R}^{n \times |V|}$. E.g. $T[3, 17]$ is the number of times word w_{17} occurs in document number 3.

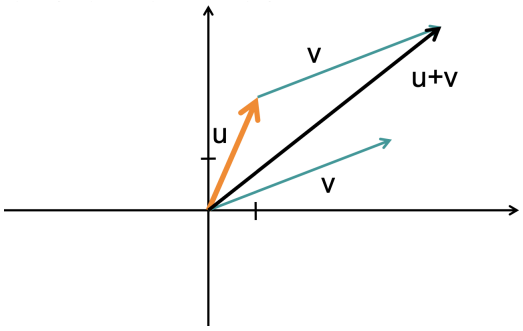
Transposition reverses rows into columns : if $\mathbf{A} \in \mathbb{R}^{r \times c}$, then $\mathbf{A}^T \in \mathbb{R}^{r \times c}$ with $\mathbf{A}[\mathbf{i}, *] = \mathbf{A}^T[*i]$
Tensors are a generalization over vectors and matrices, with a finite number of axes. For instance a Rubik's cube can be viewed as a tensor with 3 axes.
The **shape** of a matrix or tensor is the tuple providing the size of each axis. E.g. a Rubik's cube has shape $(3, 3, 3)$.

Notations: In the following

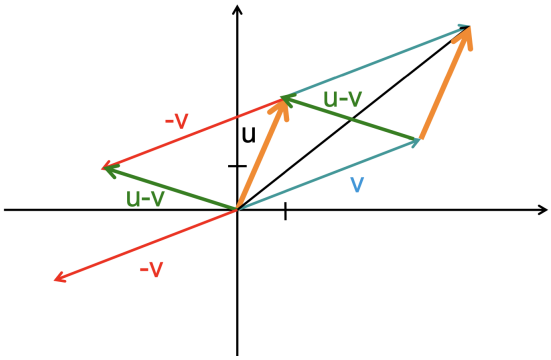
- we will use lower case bold face for vectors : $\mathbf{x}, \mathbf{w}$, in $\mathbb{R}^d$
- and upper case bold face for matrices : $\mathbf{W}, \mathbf{W^1}, \mathbf{E}$ ...

2.3 Operations on vectors

- Sum of two vectors $\mathbf{z} = \mathbf{u} + \mathbf{v} = (u_1 + v_1, u_2 + v_2, ..., u_d + v_d)$



- Substraction of two vectors $\mathbf{z} = \mathbf{u} - \mathbf{v}$: each component is $z_i = u_i - v_i$

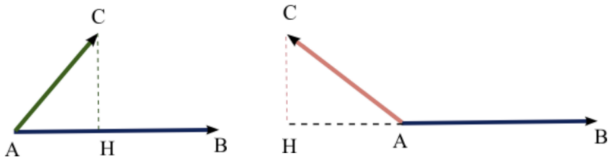


- Multiplication by a scalar :
  - $\mathbf{z} = a\mathbf{u}$: each component is $z_i = au_i$
  - $-\mathbf{z} = (-z_1, -z_2, ..., -z_d)$
- Two vectors $\mathbf{u}$ and $\mathbf{v}$ are colinear iff $\exists a \in \mathbb{R}, a \neq 0, /\mathbf{u} = a\mathbf{v}$
  - If $a > 0$, both vectors have same **direction**
  - If $a < 0$, both vectors have opposite **direction**

2.3.1 Dot product, orthogonality, norm

- In $\mathbb{R}^d$, **Dot product = inner product = scalar product ("produit scalaire")** is an operation taking two vectors and **outputing a scalar** defined as $\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{d} u_i v_i$
- Two vectors $\mathbf{u}, \mathbf{v}$ are **orthogonal** iff $\mathbf{u} \cdot \mathbf{v} = 0$
  - Geometrical interpretation: the directions of the vectors are orthogonal (at right angles)
- **Norm** of a vector : $\|u\| = \sqrt{\sum_{i=1}^{d} u_i^2}$
  - Geometrical interpretation (d=2 or d=3) : the norm of vector $\mathbf{u}$ is the **distance** between the two points of a representative bipoint of vector $\mathbf{u}$
  - Cf. Pythagore in d=2
  - Hence also called **length** of a vector, or **magnitude**
- A particular basis in $\mathbb{R}^n$ is the **canonical basis** $(\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_n)$ where each $\mathbf{e}_i$ has a component equal to 1 at position i and all other components are 0. This basis is orthogonal (each pair of e_i, e_j are orthogonal) and each of its member is of length (=norm) 1. For any vector $\mathbf{v}$, the linear combination in the canonical basis has the v_i as coefficients: $\mathbf{v} = v_1\mathbf{e}_1 + v_2\mathbf{e}_2 + ... + v_d\mathbf{e}_d$
- **Normalized vector** : for a vector $\mathbf{u}$, the normalized vector is $\frac{\mathbf{u}}{\|\mathbf{u}\|}$ has same direction as u, but its norm is 1
  - indeed $\left\| \frac{\mathbf{u}}{\|\mathbf{u}\|} \right\| = \sqrt{\sum_{i=1}^{d} \frac{u_i^2}{\|\mathbf{u}\|^2}} = \sqrt{\frac{1}{\|\mathbf{u}\|^2} \sum_{i=1}^{d} u_i^2} = \sqrt{\frac{1}{\|\mathbf{u}\|^2} \|\mathbf{u}\|^2} = 1$
  - A vector is fully defined by its norm (length) and its direction, which is given by its normalized vector $\frac{\mathbf{u}}{\|\mathbf{u}\|}$
- **Analytical interpretation: which components matter for a dot product?**:
  - Only positions having non-nul components for both vectors will count
  - Big dot product iff components having same sign are big, components having opposite sign are small
  - Dot product is sensitive to magnitude: $(2\mathbf{u}) \cdot \mathbf{v} = 2(\mathbf{u} \cdot \mathbf{v})$
- **Geometrical interpretation**:

- Orthogonal projection of $\mathbf{u}$ on $\mathbf{v}$ and vice-versa:
- let $\mathbf{z}$ be a vector orthogonal to $\mathbf{v}$. Then $\mathbf{u}$ can be expressed as a linear sum of the normalized vectors of $\mathbf{v}$ and $\mathbf{z}$: i.e. $\exists \alpha, \beta \in \mathbb{R}$ such as $\mathbf{u} = \alpha \frac{\mathbf{v}}{\|\mathbf{v}\|} + \beta \frac{\mathbf{z}}{\|\mathbf{z}\|}$.
  - **Exercise:** express the value of α (apply the dot product with $\mathbf{v}$ to the two sides of the above equality)
- Two geometrical situations:



- $\overrightarrow{AC} \cdot \overrightarrow{AB}$ equals either $AHAB$ if H on same side as B with respect to A, or $-AHAB$ otherwise
- **Question**: let $\mathbf{u}$ a vector in $\mathbb{R}^2$, let V bet a set of vectors of length l, graphically search the vector $\mathbf{v} \in V$ that maximizes the dot product with u

2.3.2 Cosine

- Usually interpreted as a **similarity measure** between two vectors
- $cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$
- values between -1 and 1
- 0 if orthogonality
- $cos(-\mathbf{u}, \mathbf{v}) = -cos(\mathbf{u}, \mathbf{v})$
- It is equal to the inner product of the two normalized versions of $\mathbf{u}$ and $\mathbf{b}$: $cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}}{\|\mathbf{u}\|} \cdot \frac{\mathbf{v}}{\|\mathbf{v}\|}$
- hence **the cosine of two vectors depends on their direction only, not on their lengths**
- We can say that cosine is normalized for length
- Hence, insensitive to magnitude : $cos(2\mathbf{u}, \mathbf{v}) = cos(\mathbf{u}, \mathbf{v})$

Geometrical interpretation with d=2

- TODO figure: trigonometric circle
- cosine sensitive to the angle between the two vectors only

Cosine as a similarity measure Cosine is very often used as a **similarity measure** when the magnitude of the vectors do not matter: when only the proportions between the components matter. This is very different from using euclidian distances (see lab session on K-NN, empirical study of difference when using cosine versus distances).

2.3.3 Distance

- dist$(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|$
- **Geometrical interpretation**: distance between extremities of two representative bipoints with same origin.
- TODO: figure

Exercise: Graphical comparison of measures Suppose we are in $\mathbb{R}^2$, in an orthonormed system with origin O. Let us consider two vectors $\mathbf{u}$ and $\mathbf{v}$, and two points A and B such that $\mathbf{u} = \overrightarrow{OA}$ and $\mathbf{v} = \overrightarrow{OB}$.

- Draw / characterize the set of X points such as $\text{dist}(\overrightarrow{OA}, \overrightarrow{OB}) = \text{dist}(\overrightarrow{OA}, \overrightarrow{OX})$
  - $\longrightarrow$ the set of points on the circle having A as center and passing through B
- Draw / characterize the set of X points such as $\overrightarrow{OA} \cdot \overrightarrow{OB} = \overrightarrow{OA} \cdot \overrightarrow{OX}$
  - $\longrightarrow$ the set of points X on the line passing through B, perpendicular to (OA)
- Draw / characterize the set of X points such as $\cos(\overrightarrow{OA}, \overrightarrow{OB}) = \cos(\overrightarrow{OA}, \overrightarrow{OX})$
  - $\longrightarrow$ the set of points X on the semi-line [OB)

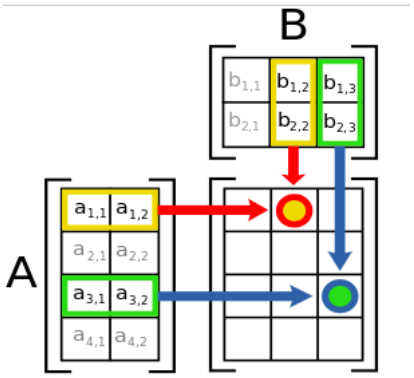
Note that these 3 sets of points are quite different.

2.3.4 Linear combination

- In linear algebra, **linear combination ("combinaison linéaire")** of a set of k vectors $\mathbf{v_1}, \mathbf{v_2}, ...\mathbf{v_k}$ is any weighted sum of these vectors: $a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + ... + a_k\mathbf{v}_k \longrightarrow$ it is thus a vector.
- **BUT** in machine learning, the term **linear combination** is rather used for a weighted sum of the n components of a vector in $\mathbb{R}^n \longrightarrow$ it is a scalar (a real number).
  - We will (improperly) say that $c = \sum_{i=1}^{n} w_i x_i$ is a linear combination of the components of $\mathbf{x}$, with each component x_i being multiplied by a weight w_i before summing, i.e. actually combining the components using a **linear function**.
  - Moreover, in ML, we will often use affine functions instead of linear functions, namely adding a real-numbered **bias**, e.g. $c = \sum_{i=1}^{n} w_i x_i + b$
  - Finally, we will pack the weights into a vector $\mathbf{w} \in \mathbb{R}^n$, so this so called linear combination can be written using dot product : $\mathbf{w}.\mathbf{x} + b$
- The **dimension** of the vector space is the number of "independent directions" in this space. In $\mathbb{R}^n$, the dimension equals the **size** of the vectors, i.e. n.
  - In the most general case, the size of the vectors does not necessarily equal the dimension of the vector space E: if one position i can systematically be defined as a linear combination of other positions in the vector, the directions are not independent. E.g. the vector space consisting of the triples $(a, b, 3a - b)\ \forall (a, b) \in \mathbb{R}^2$ is a vector space of dimension 2, even though the vectors have size 3.
  - A set $B = \{\mathbf{b}_1, ..., \mathbf{b}_n\}$ of vectors is called a **basis** of a vector space E if any vector in E can be written as a unique linear combination of elements of B
  - A vector space has an infinite number of basis, but each of them has the same number of elements, which is the dimension of E
  - In a basis, the elements are linearly independent: no b_i can be defined as a linear combination of the other elements in B
- In the following we will use variable d to refer to the size of the vectors.

2.4 Matrix product

Let $\mathbf{A} \in \mathbb{R}^{r \times c}$ a matrix of real numbers with r rows and c columns. Let another matrix $\mathbf{B} \in \mathbb{R}^{c \times d}$. The matrix product of $\mathbf{A}$ and $\mathbf{B}$ is noted $\mathbf{AB}$, and is in $\mathbb{R}^{r \times d}$. Its cell at row i and column j is the dot product of A's ith row with B's jth column : $\mathbf{A}[i, *].\mathbf{B}[*, j]$.



source: wikipedia

This matrix multiplication is also improperly called inner product or dot product of matrices.

NB: the nb of rows of $\mathbf{A}$ has to match the nb of columns in $\mathbf{B}$ otherwise $\mathbf{AB}$ is **undefined**. The matrix product is not symetric. If $\mathbf{A} \in \mathbb{R}^{r \times c}$ and $\mathbf{B} \in \mathbb{R}^{c \times d}$, then to switch the order $A.B$ requires to transpose:

$$AB = (B^T A^T)^T$$

NB: Matrix product can be used e.g. if we have two sets of vectors, organized into matrices X and W, and want to compute the dot product of all vectors in X with all vectors in W. **This will be abundantly used in linear and neural models.**

NB: The **Hadamard product** of matrices $\mathbf{A} \odot \mathbf{B}$ or **element-wise matrix multiplication** is a very different operation: $\mathbf{A}$ and $\mathbf{B}$ have to have same shape, and each resulting cell [i,j] is the multiplication $A[i,j]B[i,j]$.

2.4.1 Product of a vector and a matrix, row vectors versus column vectors

To be multiplied with a matrix, a vector $\mathbf{x} \in \mathbb{R}^d$ needs to be interpreted either as a matrix with a single row ("row vector", in $\mathbb{R}^{1 \times d}$) or as a matrix with a single column ("column vector" , in $\mathbb{R}^{d \times 1}$).

The notation is variable, with or without a dot ($\mathbf{xW}$ or $\mathbf{x}.\mathbf{W}$), the dot reminds the dot product between vectors.

If $\mathbf{x}$ is taken as a row vector, the corresponding column vector is supposed to be written as its transpose $\mathbf{x}^T$. **But the interpretation of a vector as row or column vector is usually implicit**.

Matrix multiplication being non-symetric, the notation $\mathbf{x}.\mathbf{W}$ or $\mathbf{W}.\mathbf{x}$ will implicitely correspond to taking $\mathbf{x}$ as a row or column vector, and will define the order of the axis in $\mathbf{W}$: if $x \in \mathbb{R}^d$, and $\mathbf{W}$ is a matrix which is neither a row nor a column vector, then

- the notation $\mathbf{x}.\mathbf{W}$ necessarily means that $\mathbf{x}$ is taken as a row vector, that $\mathbf{W}$ has d rows, and that the result is a row vector
- the notation $\mathbf{W}.\mathbf{x}$ necessarily means that $\mathbf{x}$ is taken as a column vector, and $\mathbf{W}$ has d columns, and that the result is a column vector

It is essential to learn how to infer the shapes of matrices from the mathematical notation (cf. exercises in lab session).

In all the following we will take vectors as row vectors (cf. Goldberg's notation). Let $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{W} \in \mathbb{R}^{d \times c}$

- $\mathbf{xW}$ is a (row) vector in $\mathbb{R}^c$

2.4.2 Product of a one-hot vector and a matrix

Let $\mathbf{x} \in \mathbb{R}^d$ is a one-hot vector, in which i is the position for the non-null component $x_i = 1$. Then the product $\mathbf{xW}$ **amounts to selecting the ith row of W. this will be abundantly used in the maths formulation, but in practice, in a program we'll rather implement it as a row selection, and not as a matrix product.**

3 A first simple classification algorithm: K-NN ("k nearest neighbors")

Recommended reading: Hal Daumé III, CIML section 3.2.

The K-NN is a classification algorithm: the set of possible classes is known in advance. NB: the classes are often called the **labels**.

- It is **supervised ("supervisé")**, meaning it uses a set of examples for which the correct class is known. We will call it the **training set** although there is no training phase in the K-NN algorithm.
- In all this document, when considering a single label classification task, a classification **example** is a pair (x, y)
  - with $x \in \mathbb{R}^d$ being the vector representating an object to classify
  - and $y \in \mathcal{Y}$ is the known class for this object, among a set of C classes ($C = |\mathcal{Y}|$)
- We will note the training set of T examples, $X = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), ..., (\mathbf{x}^{(T)}, y^{(T)})\}$
- the known class in an example is called the **gold class**. More generally, **gold data** is when the data comprises some inputs, along with the desired output for each of them.

For a given input object o with vector representation x, the assigned class is obtained using a **majority vote** on the set of classes of the K nearest neighbors of x, among the training set:

- input = vector $\mathbf{x} \in \mathbb{R}^d$, number k of neighbors to consider
- compute the distance between x and each training example
- get the "k nearest neighbors" to $\mathbf{x}$, namely the k training examples that have shortest distance to $\mathbf{x}$
- get C_k = the gold classes of the k neighbors
- output the most frequent class in C_k
  - A heuristic has to be applied in case of ties (e.g. most frequent class in training set)

Illustration with d=2 TODO

The number k is a **hyperparameter** : its value is set in advance, before training time (if there is a training phase) and prediction time. See below for more examples of hyperparameters.

3.1 Variations

There are several variations of the basic algorithm

- concerning the way the K-neighbors are identified:
  - instead of using euclidian distance, one can use vector similarity, in particular cosine similarity
- concerning the majority vote:

  - instead of counting all neighbors in the same way, one can weight them according to their distance / similarity to x: a closer example should count more than a further one
  - weight can be the inverse of the distance
  - or the cosine when using the cosine similarity
- These two booleans form two other possible hyperparameters of the K-NN algo
- Which variation to use has to be determined empirically, and may vary from one dataset to another.

Computing the distances (or similarities) of x with all the training examples can be quite long.

- tricks can be used to fasten distance computation
- if the training set is too big, clusters of training examples can be used instead of the examples themselves:
- within examples having the same gold class, a clustering algorithm can be used to form clusters (namely subsets). Each cluster can be represented using the centroid of the vectors of its members.
- instead of identifying neighbor examples, one can identify neighbor clusters

3.2 Scaling the features

- The **features** used for classification correspond to each position in the vector space of the vectors representing the objects to classify.
- The same word "feature" is used for the position itself (and its semantics), and for the value at this position. If needed we will insist by using **feature value** in the latter case.
- The K-NN algo **does not use any weighting of the feature values** when computing distances (or similarities).
- We will see that linear classification consists in automatically learning the relative weight of the features (cf. ML1 course, perceptron).
- Hence, choosing and scaling features is very important when using the K-NN
- Example:
  - When classifying emails into spam / non spam, using a binary feature "this email is in English" will be masked if other features like "number of recipients" can have large values.

4 General concepts in ML

4.1 What is machine learning?

- Machine learning is "the field of study that gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, cited by Andrew Ng)
- Machine learning is programming computers to optimize a performance criterion using example data or past experience. (Alpeydin, 2010)
- Tom Mitchell, 1997 :
  - we can say a computer **learns** if its performance measured on a task T using the metric P augments with experience E.

Use cases

- when there is no known direct symbolic method (=that can be made explicit by a human)
  - e.g. : speech recognition : impossible to make the competence of speakers fully explicit
- when the direct method is too costly
  - for instance, writing a symbolic electronic grammar with sufficient coverage is extremely costly
  - moreover, disambiguating between the possible parses requires to model lexical / world knowledge, a task that is not resolved as to now!
    - I am observing the man with a hat / with binoculars
    - modelling the information needed to "understand" that one cannot use a hat to observe anything remains beyond our reach;
    - $\longrightarrow$ statistical models work better than symbolic models for this kind of tasks;
    - $\longrightarrow$ integration of frequency information;
    - but lack of control...

4.2 Typology of ML tasks

Types of ML tasks are distinguished according to the structure of their input / output:

- clustering
- regression
- classification
- ranking

NB: a given NLP task may give rise to formalizations using several of these task types. For instance, coreference resolution can use binary classification or ranking.

Clustering (essential but not covered here) : Objective = to group objects

- without using a predefined set of categories
- but using instead a predefined notion of similarity between objects
  - Guiding principle = maximize similarity within clusters / minimize similarity between clusters

Ranking (not covered here)

- input = a set of objects
- output = an ordering over these objects

4.2.1 Classification

- Input = feature vector representing an object
- Output = one or several classes (or "categories")
  - among a **predefined** set of classes

Examples of NLP tasks typically solved using classification:

- language identification
- document classification
- "Sentiment analysis"
- Speech recognition
- Word sense disambiguation (WSD)
- Semantic role labeling
- POS tagging

Other tasks can be cast into a classification problem :

- For instance transition-based syntactic parsing
- input object = a parser "configuration", e.g. a buffer of yet-to-read tokens, a stack of not-fully-processed tokens, a set of already built dependency arcs
- classes = a set of possible actions to transform the configuration
  - move next buffer token to the top of the stack
  - build a dependency arc between first buffer token and top stack token
  - etc...
- another example is the task of predicting a word given a context (in particular, predicting a word given its left context), which can be viewed as a classification problem input = context, possible output classes = the vocabulary of words. **This is central to current neural NLP.**

Subtypes of classification problems

- **binary classification** : only two possible classes $\longrightarrow$ this often gives rise to simplified algorithms, in which the score of only one of the two classes is computed, and a threshold is used to decide whether to predict that class or the other one.
- **multiclass** : more than 2 classes
- **multilabel** : the number of classes for an object is not necessarily one, and is not known in advance
- **structured prediction**
  - objects and/or classes have a internal structure
  - typically : **sequences**, **trees** or more general **graphs**
  - structured problems can be further classified as:
  - **one to one**, e.g. WSD
  - **many to one**, e.g input = sequence, output= 1 class, e.g. document classification, sentiment analysis
  - **one to many**, input = 1 object, output = a sequence, in particular a sequence of words. E.g. predicting a legend for an input image
  - **many to many** : input = a sequence of objects, output = a sequence of classes
    - if sequences may have different sizes : **sequence-to-sequence** models ("seq2seq")
    - otherwise the term is rather **sequential classification**
      - input = sequence of n objects, output = one class for each object
      - e.g. POS tagging

· finding the optimal sequence of n classes (**global optimization**) does not always correspond to choosing the best class locally for each object (**local optimization**)

* classes can also have a more complex structure. E.g. parsing can be cast a the problem of associating a (syntactic) tree to a sequence (of words): the set of possible trees given a sequence of words is considered as the set of potential classes

Strategies for multiclass classification

- Let C be the number of possible classes ($C = |\mathcal{Y}|$)
- The multiclass trait can either be taken in charge by the classification algorithm itself
- or binary classifiers can be used to perform multiclass classification
  - **one-versus-all ("OVA")** strategy (or "one versus rest") : learning of C binary classifiers, one for each class versus the other $C - 1$ classes. And prediction of the class having best score against all the other ones.
  - **one-versus-one ("OVO")** strategy: learning of one binary classifier per pair of classes ($C(C-1)/2$ classifiers) and majority vote: the predicted class is the one winning the highest number of duels

Selecting between n candidates as a classification task Some problems can be reduced to the selection of one element among n candidates.

- Example: anaphora resolution
  - Input = a pronoun + a set of antecedent candidates
  - Output = the closest antecedent
- dependency parsing, which can amount to
  - input : a word token (and its context) + a set of governor candidates (generally : all the other tokens)
  - output : picking one governor
  - plus well-formedness constraints (no cycles in output)

Several strategies can be used for this sort of problem:

- via binary classification, input = pair [object + one candidate], output = "yes this is a correct candidate"
- $\longrightarrow$ and outputing the candidate with best binary score
- or directly via a **ranking** task, over the n candidates
- technically in this case, the set of n candidates can be viewed as a dynamically defined set of output classes

4.2.2 Regression

- Input = input features characterizing an object (= a feature vector)
- Output = a number, either corresponding to another characteristic of the object, or corresponding to the probability of the object to have a given characteristic

In **statistics**,

- the object = an individual from a population, or from a population sample
- the features are values of statistical variables defined on this population

- regression is used to **model and analyze the relationship** between
  - a **dependent** statistical variable Y
  - and a certain number of **independent** or **explanatory ("explicatives")** variables
- by means of a mathematical function $Y_i = f(X_{i1}, X_{i2}, ..., X_{in})$, with i the id of an individual, having values Y_i for the dependent variable, and X_{ij} for the jth dependent variable
- the type of mathematical relation between the dependent variable and the explanatory variables defines the type of regression
- most famous regression models are
  - linear regression : the relation is supposed to be linear $Y_i = \beta_1 X_{i1} + \beta_2 X_{i2} + ... \beta_n X_{in} + \epsilon_i$
  - binary logistic regression (see below section XXX) considers
    - a binary categorical variable Y, taking values 1 or 0,
    - it considers the probability $p = P(Y = 1)$ for an individual to have 1 as value for Y
    - and supposes that the log of odds $ln(\frac{p_i}{1-p_i})$ is a linear combination of the explanatory variables $\beta_1 X_{i1} + \beta_2 X_{i2} + ... \beta_n X_{in} + \epsilon_i$

Example:

- object = a word token (or word occurrence)
- input features : frequency, number of syllables (numerical features), plus maybe categorical features turned into one-hot vectors : syntactic position, part-of-speech, ...
- output : reading time
- interpretation : to which extent the input features allow to "explain" the reading time of a word
- linear regression (see below): the weights of each explanatory variable can be studied

In machine learning, the "individuals" are objects, the explanatory variables are features

- a linear regression model can be learnt (i.e. the weights β_j are learnt) and used to **predict** a numerical value : e.g. predict the number of accidents a person will have in the next 5 years
- **logistic regression can be used ... for classification problems** : e.g in binary logistic regression, what is computed is the probability of an object belonging to one of the two classes. Dependeing on whether this probability is > 0.5 then we can say the model predicts the object belongs to this class or the other one (see below section ???)

4.3 "Learning"/"training" versus "prediction"/"decoding"/"inference" phase

For all the previous task types except clustering, one can distinguish between

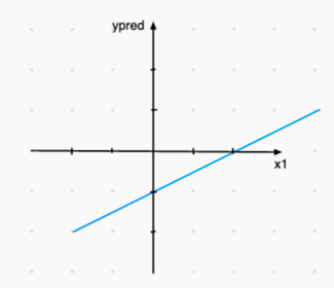
1. a preliminary **learning** phase (or **training** phase), meant to adjust some **parameters**
2. the **prediction** phase (also called **inference** or **decoding**, for historical reasons): a mathematical function, the **predictive function** is used to predict a numerical value for an input object, given
   - the parameters adjusted during the training phase, often noted θ (for all the parameters)
   - and the vector representation $\mathbf{x}$ of an object o (we suppose that the features to represent o have been chosen). The transformation of o into its vector representation is usually noted ϕ : $\phi(o) = \mathbf{x}$
   - the predicted value is usually noted with a hat $\hat{y}$ to distinguish it from the "true" value:

$$\hat{y} = f(\mathbf{x}, \theta)$$

4.3.1 Example : linear regression

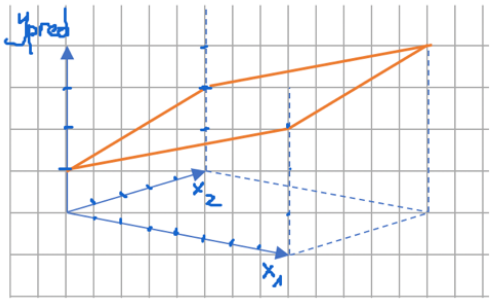
- In linear regression, the predictive function is a linear combination of the components in $\mathbf{x}$, actually an affine one : $\hat{y} = f(\mathbf{x}) = \mathbf{x}.\mathbf{w} + b$
- $\hat{y}$ is a numerical value
- the parameters that need to have been learnt beforehand are $\mathbf{w}$ and b.

If d = 1, $\hat{y} = x_1 w_1 + b$, $\longrightarrow$ equation of a **line** : the predicted values will form a line



NB: to be distinguished from representing classification data in d=2, with 1 axis for x_1 and one axis for x_2 (as we did in the K-NN section). Here the second coordinate is the predicted value $\hat{y}$.

If d = 2, $\hat{y} = x_1 w_1 + x_2 w_2 + b$, $\longrightarrow$ equation of a **plane**



Exercise: find the equations of the line and the plane above.

4.3.2 Parameters versus hyperparameters

A key distinction needs to be made between parameters (automatically learnt at training time) versus hyperparameters, that are needed to compute the prediction, but that are not learnt (they are set by some other means). Hyperparameters can be used either at training time, prediction time or both. The latter can be divided into

- hyperparameters used at prediction time only, e.g. K in K-NN;
- hyperparameters used at training time only, e.g. learning rate, number of epochs for online algorithms (see section 6)
- hyperparameters used at both times, e.g. size of hidden layers in multi-layer perceptrons (cf. below section 5.5.2).

4.4 Supervised versus unsupervised learning

Training phase = learning of the parameters, to be used in the predictive function at prediction time.
Classical distinction in machine learning: **supervised** versus **unsupervised** learning, which concerns the "past experience" E in Mitchell's definition of machine learning, namely the input to the learning phase:

- Input of learning phase =
  - general form of the prediction function, with some of the variables yet to be set
    - e.g.: in multiclass linear classification, score of class y has the form $w_y.x + b$, with $w_y \in \mathbb{R}^d$ and $b \in \mathbb{R}$.
  - a **training set**
    - in **supervised learning**: training set = set of examples = set of pairs [input, expected output]
    - in **unsupervised learning**: training set = set of inputs + some knowledge about the expected output
      - e.g. for POS tagging: knowledge of the possible POS for each word form
      - e.g. for clustering: knowledge of a similarity metric between objects, and general objective that similar (resp. disimilar) objects should be (resp. should not be) in same cluster.
- Output of learning phase = parameters' values

4.5 Generalization in supervised learning

How is a learning algorithm evaluated?

- performance metric for the task at hand (e.g. accuracy)
- what about more general criteria?
- Parallel to evaluating whether students have learnt something when attending a course
- $\longrightarrow$ exam should not be too close nor too far from what has been talked about in the lectures
- students are expected to be able to generalize

In machine learning, **generalization** is a central notion. In supervised learning,

- the training set should follow the same distribution as the inputs that we expect to use the classifier on in the future
- and we expect a learning algorithm to provide classifiers that can generalize what was learnt on the training set to new inputs, unseen at training time.

5 Prediction phase for the classification task

5.1 General prediction algorithm : return highest scoring class

For multiclass classification tasks, let us note $\mathcal{Y}$ the set of predefined classes. The prediction for object o represented by feature vector x is performed as follows:

- compute a **score** for each class
- and simply output the highest scoring class

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}}\ \text{score}(x, y, \theta)$$

- if computing the scores is not computationally too expensive, the prediction algorithm can run quickly
- the complexity is more in learning the parameters θ beforehand, during a learning phase.
- The score may be probabilistic $score(x, y, \theta) = P_\theta(class = y | object = x)$
- In structured prediction: the output is not a single local class, but a sequence or a graph (e.g. a dependency tree)

- the total number of possible output class sequences is too high to compute the score of each of them
- specific inference algorithms are needed (such as Viterbi algorithm for efficiently computing the class sequence with highest global score)

5.1.1 Simplification in binary classification

When there are only two possible classes, though the previous procedure can be used, a simplification is usually made:

- the two classes are arbitrarily mapped to 1 and 0, or to 1 and -1
- a single score is computed, interpreted as the score of class 1: score(x, θ)
- a threshold is used, usually 0, to predict one of the two classes:

$$\hat{y} = \begin{cases} +1 & \text{if score}(x) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

5.2 Families of scoring functions : linear, log-linear, non-linear (neural)

In machine learning for classification, one first chooses a family of predictive functions, and then learns the parameters to get the fully instantiated predictive function. We will cover here

- linear scores
- log-linear scores
- neural scores : obtained using a neural network

5.3 Linear scores

5.3.1 Binary linear classification

- 2 possible output classes
- generally arbitrarily mapped to +1 and -1 (or to +1 and 0 for probabilistic models)
- the computed score is that of class +1, and is linear :

$$\text{score}(x) = \mathbf{w} \cdot \mathbf{x} + b$$

- the parameters are the weight vector $\mathbf{w} \in \mathbb{R}^d$ and the **bias** $b \in \mathbb{R}$

Classification decision: **predict class +1 if $\mathbf{x}.\mathbf{w} + b \geq 0$ and -1 otherwise.**

This can be mathematically formulated using the sign function:

$$\text{sign}(u) = \begin{cases} +1 & \text{if } u \geq 0 \\ -1 & \text{otherwise.} \end{cases}$$

When applied to the linear combination $sgn(\mathbf{x}.\mathbf{w} + b)$, it directly the predicted class, namely either +1 or -1

NB: a binary linear classifier is totally defined by the pair $\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$

NB: all the pairs $\alpha\mathbf{w} \in \mathbb{R}^d, \alpha b \in \mathbb{R}$ with $\alpha > 0$ define the same classifier, whereas when $\alpha < 0$, the classifier $\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$ makes opposite predictions.

Interpretation of the bias The bias will directly define a **threshold** that the linear combination needs to exceed in order to classify the input as +1 class.

More precisely, the threshold is -b : $\mathbf{x}.\mathbf{w} \geq -b \Leftrightarrow \mathbf{x}.\mathbf{w} + b \geq 0$.

Interpretation of the weights Example : spam detection (= predict whether an email is spam or not). Suppose we represent the input email using a BOW vector. Suppose we map "spam" to class 1, and "not spam" to class -1. Then, learnt weights can be interpreted as follows:

- highly positive weights : words frequently used in spams
- highly negative weights : words frequently used in non-spams
- close to zero weights : words that are neither characteristic of spam nor ham

Comparison of linear classifiers Let α be a non null real number.

- comparing $(\alpha\mathbf{w}, \alpha b)$ and $(\mathbf{w}, b)$ linear classifiers:
  - same hyperplane cf. $\mathbf{x}.\mathbf{w} + b = 0 \Leftrightarrow \mathbf{x}.\alpha\mathbf{w} + \alpha b = 0$
  - if $\alpha > 0$: same classification decisions
  - if $\alpha < 0$: opposite classification decisions
- comparing $(\mathbf{w}, \alpha b)$ and $(\mathbf{w}, b)$ linear classifiers:
  - define two parallel but distinct hyperplanes:
  - plane with equation $x_d = -\frac{1}{w_d}\sum_{i=1}^{d-1} w_i x_i - b$
  - versus equation $x_d = -\frac{1}{w_d}\sum_{i=1}^{d-1} w_i x_i - \alpha b$
- comparing $(\alpha\mathbf{w}, b)$ and $(\mathbf{w}, b)$ linear classifiers:
  - same situation
  - cf. $(\alpha\mathbf{w}, b)$ is equivalent to $(\mathbf{w}, b/\alpha)$

Trick: bias integration To simplify notations and computations, we can switch to $\mathbb{R}^{d+1}$ and "integrate" the bias by setting:

- $\mathbf{x}' = (1, x_1, x_2, ..., x_d)$
- $\mathbf{w}' = (b, w_1, w_2, ..., w_d)$,
- and then work with the $\mathbf{x}'$ vectors and the $\mathbf{w}'$ vector
- cf. we have $\mathbf{x}.\mathbf{w} + b = \mathbf{x}'.\mathbf{w}'$

Note that then b can be called w_0, and the x_0 component of $\mathbf{x}'$ is 1.
The bias corresponds to one additional component. In practice, when D is big, the bias has limited impact.

5.3.2 Multiclass linear classification

When switching to multiclass classification, for a given input object vector x, we have to compute one score per class.

In linear classification this is done by **having one weight vector per class and one bias per class**.

Let us note $\mathcal{Y}$ the set of classes, and suppose the $C = |\mathcal{Y}|$ classes are arbitrarily mapped to integers from 1 to C, hence $\mathcal{Y} = \{1, 2, ..., C\}$.

Then the weight vectors for each class can be organized as a **weight matrix $\mathbf{W} \in \mathbb{R}^{d \times C}$ in which the ith column corresponds to the weights for class i.**. The biases for each class can form a **bias vector $\mathbf{b} \in \mathbb{R}^C$**,

For an input object represented by vector $\mathbf{x} \in \mathbb{R}^d$:

- the vector of scores of the C classes is $\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{b} \in \mathbb{R}^C$
- the score of class i is: $y_i = \mathbf{x}\mathbf{W}_{[*,i]} + b_i$

The **predicted class** is thus simply the highest scoring class, namely

$$\hat{y} = \underset{i \in 1,\ldots,C}{\text{argmax}}\ y_i = \underset{i \in 1,\ldots,C}{\text{argmax}}\ \mathbf{x}\mathbf{W}_{[*,i]} + b_i$$

Interpretation of the weights Now that there is a weight vector for each class, for a given input feature, the weights can vary from class to class. E.g. for document classification into themes, with a document represented as a BOW vector: for a given class (a theme), high weights will go to words that are representative of this theme. E.g. the weight for the feature-word "hand-ball" will be high for the sports class, but low for the society class.

5.3.3 Prediction for a batch of inputs

In practice, the computation $\mathbf{xW}$ can be made in parallel for a bunch of input vectors (in particular when using a GPU instead of a CPU).

Let $X \in \mathbb{R}^{b \times d}$ be a matrix in which each row is a vector representation of an input object. This is usually informally called a **batch** of b inputs, with b being the **batch size**.

Then $\mathbf{XW}$ belongs to $\mathbb{R}^{b \times c}$, and has as r-th row the vector of class scores for the r-th input element in the batch.

5.4 Log-linear scores (logistic regression)

Log-linear models are probabilistic models in the sense that a score $\text{score}(\mathbf{x}, y)$ is interpreted as the probability that $\mathbf{x}$ belongs to class y : $P(\text{class} = y | \text{object vector} = \mathbf{x})$.

This can be done by applying a sigmoid or softmax function to the linear scores.

5.4.1 Getting probability distributions in the binary case: sigmoid

Usually, in binary probabilistic classifiers, the two classes are +1 and 0 (instead of +1 and -1), but this is just a convenient convention.

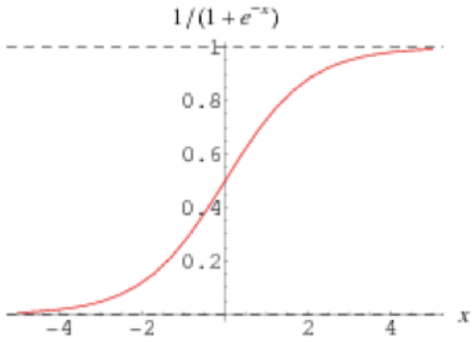
In binary classification, the linear score $\mathbf{x}.\mathbf{w} + b$ can be mapped to the $]0, 1[$ interval using the **sigmoid** function, and the result can be interpreted as the probability of belonging to class +1 : $P(\textbf{class} = +1 | \textbf{object vector} = \mathbf{x}) = \textbf{sigmoid}(\mathbf{x}.\mathbf{w} + b)$.

The probability of the other class (usually class 0) can be obtained as :

$$P(\text{class} = 0 | \text{object vector} = \mathbf{x}) = 1 - P(\text{class} = +1 | \text{object vector} = \mathbf{x})$$

Sigmoid is often noted σ and is a function from $\mathbb{R}$ to $]0, 1[$:

$$\text{sigmoid}(u) = \sigma(u) = \frac{1}{1 + e^{-u}} = \frac{e^u}{e^u + 1}$$



Properties:

- symmetric : $\sigma(-u) = 1 - \sigma(u)$
- derivative : $\sigma'(u) = \sigma(u)(1 - \sigma(u))$
- corresponds to an infinitely derivable approximation of the **Heaviside function**
  - which returns 0 if u ¡0 and +1 if u*geq*0
- it is the inverse function ("fonction réciproque") of the **logit** function:
  - logit(sigmoid(u)) = u and sigmoid(logit(p)) = p
  - with logit(p) = $ln(\frac{p}{1-p})$
  - **NB:** "ln" = log_e = "natural logarithm" ("logarithme népérien")
  - Logit corresponds to the "log of odds": If p is a probability, then $\frac{p}{1-p}$ are the **odds** ("ratio de chances") (cf. the odds of a bet ("côte d'un pari")). Then logit(p) = ln(odds of p)
  - In log-linear classification, we take the linear combination $u = \mathbf{x}.\mathbf{w} + b$ to be the logit (= the log of odds for the +1 class)
  - $u = ln(\frac{p}{1-p})$
  - $\Leftrightarrow e^u = \frac{p}{1-p}$
  - $\Leftrightarrow e^u - pe^u = p$
  - $\Leftrightarrow p = \frac{e^u}{e^u+1}$
  - $\Leftrightarrow p = \frac{1}{1+e^{-u}}$

5.4.2 Getting probability distributions in the multiclass case: softmax

Recall that in linear classification, the vector of the C class scores is $\mathbf{u} = \mathbf{x}.\mathbf{W} + \mathbf{b}$.

The **softmax** function can map any vector of real numbers to a probability distribution, namely a vector of values between 0 and 1, summing to 1. Crucially, **softmax preserves the order of the original vector**.

NB: Softmax takes as input a vector and output a vector of same size.

If $\mathbf{u} \in \mathbb{R}^C$, then so does softmax($\mathbf{u}$), and the component at position i is defined as:

$$(\text{softmax}(\mathbf{u}))_i = \frac{e^{u_i}}{\sum_{j=1}^{C} e^{u_j}}$$

NB: The denominator is constant for all classes, and serves to normalize the exponentials.

Due to the use of the exponential function, softmax accentuates the differences in the input vector, giving a very high probability for the maximum value. A few examples:

```
>>> l = [-10, -1, 2, 10]
>>> softmax(l)
```

```
[2.0604280089451095e-09, 1.6695821083209265e-05, 0.00033534453082973483, 0.9996479575876591]
>>> l = [-10, -1, 2, 100]
>>> softmax(l)
[1.6889118802245324e-48, 1.368539471173853e-44, 2.7487850079102147e-43, 1.0]
```

This is true even if the difference to the second most value is small:

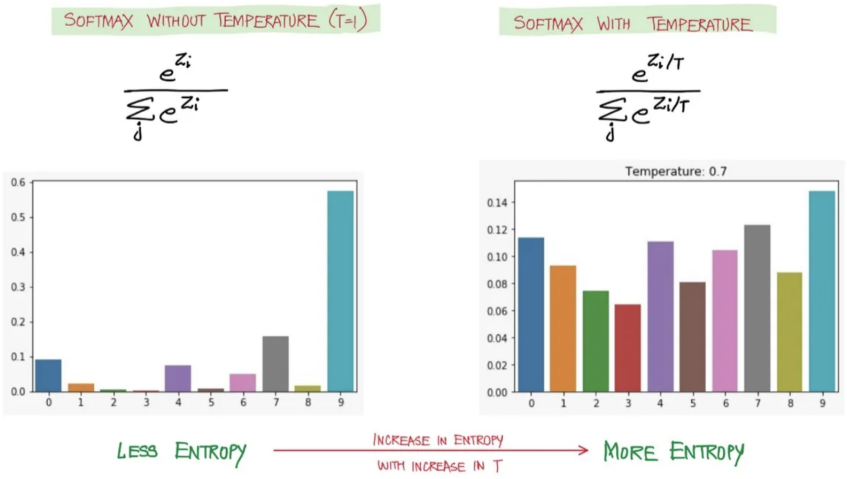
```
>>> l = [-2, -1, 5, 6]
>>> softmax(l)
[0.00024501940316294137, 0.0006660317912377041, 0.2686964019668322, 0.7303925468387672]
>>> l = [-10, -1, 5, 6]
>>> softmax(l)
[8.221499038237082e-08, 0.0006661949671731538, 0.2687622318375977, 0.7305714909802389]
>>> l = [-2, -1, 10, 11]
>>> softmax(l)
[1.6524230506650507e-06, 4.491751551549667e-06, 0.26893976894694444, 0.7310540868784533]
>>> l = [-2, -1, 100, 101]
>>> softmax(l)
[1.3540058529806076e-45, 3.680569505784375e-45, 0.2689414213699951, 0.7310585786300048]
```

Rem: the term "soft max" actually refers to a smoothed version ("soft") of argmax. Indeed, argmax can be viewed as mapping the input vector to an output one-hot vector, with 1 at the position of the maximum. Softmax provides a smoothed continuous mapping instead.

TODO: temperature: a "temperature" T hyperparameter can be used to smoothen the softmax :

$$(\text{softmax}(\mathbf{u}))_i = \frac{e^{u_i/T}}{\sum_{j=1}^{C} e^{u_j/T}}$$

The higher T is, the more it will diminish the differences between the components of the softmax-ed vector. In probabilistic terms, this means that the probability distribution represented by the softmax-ed vector has a higher entropy, as illustrated below, with $T = 0.7$. A high value of T (e.g. 100) will simply uniformize the distribution, hence is useless.



(source: https://medium.com/mlearning-ai/softmax-temperature-5492e4007f71)

5.4.3 Binary logistic regression for binary log-linear classification

Taking the sigmoid of a linear combination of a vector $\mathbf{x}$ can be viewed as what is called in statistics a **binary logistic regression**.
The d components of the input vector representation are interpreted as the values of d explanatory variables $X_1, ..., X_d$, which are used to output the probability of a binary dependent variable Y of taking value 1.

More precisely, if Y is a binary random variable taking values +1 or 0.
A model of binary logistic regression outputs $P(Y = 1|X_1 = x_1, X_2 = x_2, ...X_d = x_d)$ using a linear combination $\mathbf{x}.\mathbf{w} + b$:

$$P(Y = 1|X_1 = x_1, X_2 = x_2, ...X_d = x_d) = \sigma(\mathbf{x}.\mathbf{w} + b)$$

The term **log-linear** is used, given the output of the model (here $\sigma(u)$), taking the log will give a result close to u (actually $u - ln(1 + e^u)$).

To do **binary classification** with a binary regression model, one will predict class 1 iff
$P(Y = 1|X_1 = x_1, X_2 = x_2, ...X_d = x_d) \geq 0.5$
$\Leftrightarrow \frac{e^u}{e^u+1} \geq 1/2$
$\Leftrightarrow 2e^u \geq e^u + 1$
$\Leftrightarrow e^u \geq 1$
$\Leftrightarrow u \geq 0$

Hence the classification decisions are the same in a linear model and in a log-linear model.

The difference lies in the probabilistic interpretation of the output, and will be used at learning time.

5.4.4 Multinomial logistic regression for multiclass log-linear classification

In a similar way, computing the probability distribution over classes using linear combinations can be viewed as a using a multinomial logistic regression model.

$$P(Y = .|X_1 = x_1, X_2 = x_2, ..., X_d = x_d) = \text{softmax}(\mathbf{x}\mathbf{W} + \mathbf{b})$$

To do **multiclass classification** with a multinomial regression model, one can simply predict the class with highest probability:

$$\begin{aligned}\hat{y} &= \underset{i \in \mathcal{Y}}{\text{argmax}}\ P(Y = i|X_1 = x_1, X_2 = x_2, ...X_d = x_d) \\ &= \underset{i \in \mathcal{Y}}{\text{argmax}}\ \frac{e^{\mathbf{x}\mathbf{W}_{[*,i]}+b_i}}{Z} \\ &= \underset{i \in \mathcal{Y}}{\text{argmax}}\ \mathbf{x}\mathbf{W}_{[*,i]} + b_i\end{aligned}$$

Hence, as for the binary case, the classification decisions are the same in a linear model and in a log-linear model.

5.4.5 Note on vocabulary

In NLP, logistic regression (whether binary or multinomial) is also called the **MaxEnt** model, for "maximum entropy model".

The score of one class is e^u/Z, with $u =$ the linear combination of components of $\mathbf{x}$. When applying a log, the score is $u - ln(Z)$, hence the term **log-linear**: the score is linear after applying log to it.

5.5 Neural networks

In this section, we will now turn to neural networks. We will first see how (log)linear classifiers as a network, introduce the notion of artificial neuron, and of multi-layer networks (hidden layers and non-linearities).

A network is basically a way to view a multivariate scoring function, taking as input the vector representation of the input (e.g. in classification, vector representing the object to classify).

- in linear classification, we've seen that
  - binary linear classification :
    - score of the +1 class = $\mathbf{w}.\mathbf{x} + b$
    - predicted class = sgn(score)
  - multiclass
    - vector of class scores $y = \mathbf{x}\mathbf{W} + \mathbf{b}$
    - predicted class = $\hat{y} = \underset{c \in 1,\ldots,C}{\text{argmax}}\ y_c$

So for instance in multiclass classification with input vectors of size d, and C classes, the scoring function is $\mathbb{R}^d \to \mathbb{R}^C$.

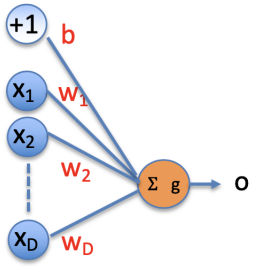
A neural network is a standardized representation of a more complex scoring function:

- can be viewed as a computation graph (input nodes = input vector, internal nodes for elementary operations)
- makes use of **building blocks**, in particular
  - artificial neuron
  - layers

An **artificial neuron** is a fancy name for representing a function $\mathbb{R}^d \to \mathbb{R}$ consisting in a linear function plus an "activation function":

- input to the neuron = a vector in $\mathbb{R}^d$
- output obtained by applying
  - a linear combination $z = \mathbf{w}.\mathbf{x} + b$ = the **preactivation** value
  - plus an **activation** function, that we will note g
  - $o = g(z) = g(\mathbf{w}.\mathbf{x} + b)$
  - see below for biological inspiration
- g is a hyperparameter of the artificial neuron
- whereas $\mathbf{w}$ and b are in general learnt, hence are parameters

Usual graphical representation:



A binary (log)linear classifier can be viewed as a neural network ... with a single neuron:

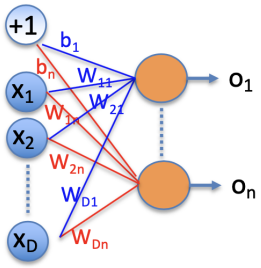
- choosing g = sgn corresponds to the binary linear classifier
- choosing g = sigmoid corresponds to the binary logistic regression classifier

Neurons are traditionally organized in layers: a **layer** is a list of n neurons, which receive the same input:

- input = a vector in $\mathbf{x} \in \mathbb{R}^d$

- parameters of the layer = a matrix $\mathbf{W} \in \mathbb{R}^{d \times n}$, and a bias vector $b \in \mathbb{R}^n$
- hyperparameters of the layer = the size n, and the activation function g which must be defined from $\mathbb{R}^n$ to $\mathbb{R}^n$
- output is computed with n linear combinations of $\mathbf{x}$, outputing a vector in $\mathbb{R}^n$ = a vector of preactivation values: $z = \mathbf{xW} + \mathbf{b}$
- and application of the activation function, $o = g(z) \in \mathbb{R}^n$

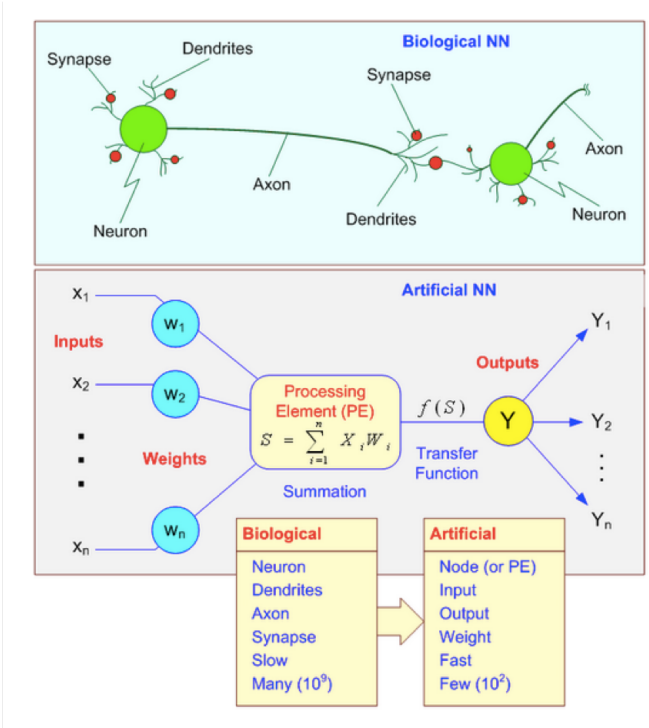
Usual graphical representation:



A multiclass (log)linear classifier can be viewed as a neural network with a single output layer of size C, outputing the scores for the C classes:

- one output neuron per class
- choosing g = identity function corresponds to the multiclass linear classifier
- choosing g = softmax corresponds to the multiclass logistic regression classifier

5.5.1 Biological inspiration



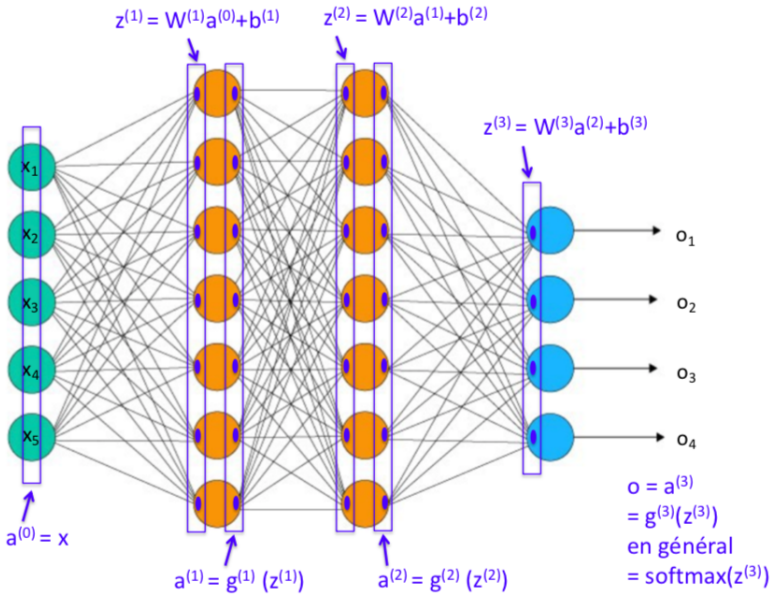
(by Vipin Tyagi)

- When using e.g. sigmoid as activation function, the output value of a neuron is close to 1 or 0 depending on the pre-activation value (cf. the S-shape of the sigmoid). Hence the term "activation" function: one says a neuron "**fires**" or not, i.e. outputing ≈1 or ≈0.
- The tanh function (see below) has a similar effect, but between -1 and 1 instead of 0 and 1.
- This is somehow similar to real neurons: the quantity of input nerve impulses needs to surpass a certain threshold for the neuron to conduct them to other neurons.

5.5.2 Hidden layers and multi-layer perceptrons (MLP)

- "Deep" learning is obtained by using additional **hidden layers**
- in particular to get a **fully-connected forward neural network**
- synonym of **multilayer perceptron (MLP)**
- obtained by adding one or more **hidden layers**, with **non-linear activation functions**
- the architecture of an MLP is thus
  - layer 0 : input vector
  - n hidden layers : layers 1 to n
  - output layer : layer n+1
  - the parameters are n+1 matrices $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(n+1)}$
  - and n+1 bias vectors $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, ..., \mathbf{b}^{(n+1)}$
  - $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ are the weight matrix and bias vector to go from layer i-1 to layer i
  - if n_i is the nb of neurons of layer i
  - then $\mathbf{W}^{(i)}$ has n_{i-1} rows and n_i columns (or vice-versa depending on the notations)

In the illustration below, we can distinguish for layer $i \neq 0$ between the pre-activation vector $z^{(i)}$ and the activation vector $a^{(i)}$:



Forward propagation : the **forward propagation** means actually computing the output vector given an input vector, and a network. It is "forward" because values are computed sequentially, from layer 1 to layer n and finally output layer:

- input : $\mathbf{a}^{(0)} = \mathbf{x}$
- recurrence : for each layer $i > 0$:
  - input to layer i = $\mathbf{a}^{(i-1)}$ activation vector at preceding layer
  - linear combination $\mathbf{z}^{(i)} = \mathbf{a}^{(i-1)}\mathbf{W}^{(i)} + \mathbf{b}^{(i)}$
  - activation $\mathbf{a}^{(i)} = g^{(i)}(\mathbf{z}^{(i)})$
  - output vector = $\mathbf{o} = \mathbf{a}^{(n+1)}$ = vector of class scores

Exercise: Give the mathematical expression for $\mathbf{a}_3^{(2)}$ (provide the intermediary variables), and for the score of class 2.

A MLP is a "forward" network because outputs at given layers are not re-used for preceding layers (contrary to recurrent networks).

A MLP is "fully-connected" meaning all neurons in layer i-1 are connected to all neurons in layer i.

Prediction with a MLP classifier :

- in binary classification: as usual, we compute the score for the +1 class only $\longrightarrow$ meaning here that the MLP has a 1-neuron output layer, outputing a single real-valued output o
  - last activation function can be sgn or more often the **sigmoid** function $\longrightarrow$ o is interpreted as $P(class = +1|object = \mathbf{x})$
- in multiclass classification:
  - output layer = one neuron per class
  - last activation function can be identity function
  - or more often the **softmax** function : $\mathbf{o}_j$ is interpreted as $P(class = j|object = \mathbf{x})$

5.5.3 Non linearities: custom activation functions

A key aspect of neural networks is that activation functions should be non linear.

Actually it can be easily proved that given a MLP M with a single hidden layer, with either no activation or a linear activation function, an equivalent (= representing the exact same function from $\mathbb{R}^d \rightarrow \mathbb{R}^C$) network with no hidden layer can be easily computed (its single weight matrix is a linear combination of the matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$).

Moreover, activation functions should possibly have nice mathematical properties. In particular, for gradient-based optimization techniques, the functions should be derivable, possibly multiple times.

Custom non-linear activation functions for hidden layers are show below, as detailed in **?**:

Sigmoid The sigmoid activation function $\sigma(x) = 1/(1+e^{-x})$, also called the logistic function, is an S-shaped function, transforming each value x into the range $[0, 1]$. The sigmoid was the canonical non-linearity for neural networks since their inception, but is currently considered to be deprecated for use in internal layers of neural networks, as the choices listed below prove to work much better empirically.

Hyperbolic tangent (tanh) The hyperbolic tangent $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$ activation function is an S-shaped function, transforming the values x into the range $[-1, 1]$.

Hard tanh The hard-tanh activation function is an approximation of the tanh function which is faster to compute and take derivatives of:

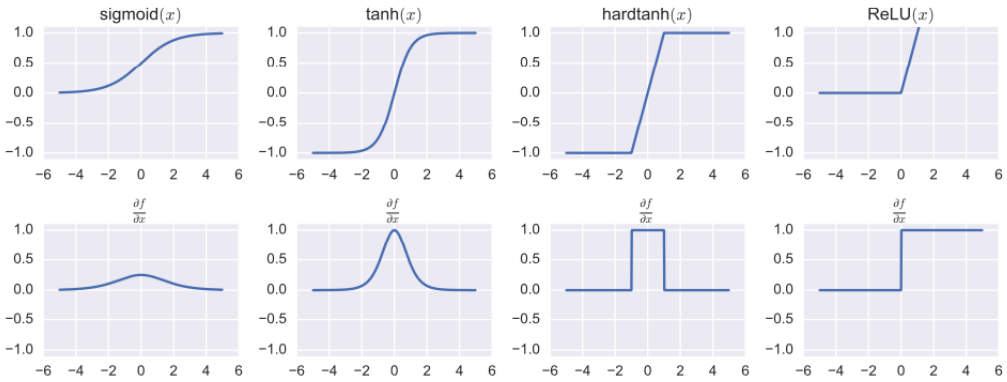
$$\text{hardtanh}(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{otherwise} \end{cases} \tag{4.5}$$

Rectifier (ReLU) The Rectifier activation function [Glorot et al., 2011], also known as the rectified linear unit is a very simple activation function that is easy to work with and was shown many times to produce excellent results.[6] The ReLU unit clips each value $x < 0$ at 0. Despite its simplicity, it performs well for many tasks, especially when combined with the dropout regularization technique (see Section 4.6).

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases} \tag{4.6}$$

As a rule of thumb, ReLU units work better than tanh, and tanh works better than sigmoid.

Here are the corresponding shapes of these functions, and of their derivatives (still from Goldberg 2017):



6 A famous learning algorithm for linear classifiers : the perceptron

In order to get a sense of a concrete learning algorithm, we provide now the famous perceptron algorithm. Note we will come back to its properties (convergence and speed of convergence) in section 8.3, once we've covered linear separability (section 8.2).
Reminder: learning a binary linear classifier means learning a pair $\mathbf{w} \in \mathbb{R}^d, b$, while learning a multiclass linear classifier means learning $\mathbf{W} \in \mathbb{R}^{d \times C}, b \in \mathbb{R}^C$.
Recommended reading: Hal Daumé III chapter 4 (note: in the CIML book, "activation" is used to refer to the affine combination $\mathbf{x}.\mathbf{w} + b$).

6.1 Perceptron for binary classification

Dates back to work by Rosenblatt, 1957 (Cornell University).
Already called "neural network" at that time.

- actually a linear classifier for the binary case can be viewed as a neural network ... with a single "neuron" (see below section **??**).
- original motivation : modeling human cognition
- part of a scientific stream called "cognitivism"
  - in which thinking is viewed as information processing
  - logical analysis of human cognitive tasks, and reproduction via a formal model
- and of "connexionism" : use of networks to model cognitive tasks
- practical motivations: simply obtain good performance for artificial intelligence systems

The perceptron algorithm is an **online** algorithm:

- $\mathbf{w} \in \mathbb{R}^d, b$ are initialized with zeros
- and are iteratively updated, considering one training example at a time
- as opposed to using all the training examples at once (such as in SVMs)

Perceptron algo for the binary case:

INPUT:
- X = training set of T examples : $\{(\mathbf{x}^{(1)}, y^{(1)}), ...\}$ with each $\mathbf{x}^{(t)} \in \mathbb{R}^d$ and $y^{(t)} \in \{+1, -1\}$
- **hyperparameter** E = positive integer = number of **epochs**

INITIALIZE $\mathbf{w}$ = null vector in $\mathbb{R}^d$ and a bias $b = 0$

LOOP: for each epoch in 1, ..., E:

  LOOP: for each t in 1, ..., T:

  - PREDICT class for t-th example given current $\mathbf{w}, b$: $\hat{y^{(t)}} = sgn(\mathbf{x}^{(t)}.\mathbf{w} + b)$
  - if prediction is wrong, namely if $\hat{y^{(t)}} \neq y^{(t)}$

    UPDATE the parameters:

    - $\mathbf{w} \longleftarrow \mathbf{w} + y^{(t)}\mathbf{x}^{(t)}$
    - $b \longleftarrow b + y^{(t)}$

OUTPUT: $\mathbf{w}, b$

- the perceptron is a **passive** online algorithm, meaning that when the current version of the parameters makes the right prediction for example t, then no update is performed.
- An **epoch** is a loop over all the training set.
- The number of epochs is a hyperparameter of the algorithm.

Analysis of the update step The two loops correspond to $T \times E$ iterations. Let us note w^0, b^0 the null initialization of the parameters, and $\mathbf{w}^i, b^i$ for the values of $\mathbf{w}, b$ at the end of iteration i.

Let $(\mathbf{x}^{(t)}, y^{(t)})$ be the example used at iteration i.

- if the class is correctly predicted using w^{i-1}, b^{i-1}, then no update is done
- otherwise
  - $w^i = w^{i-1} + y^{(t)}\mathbf{x}^{(t)}$
  - $b^i = b^{i-1} + y^{(t)}$

Exercise: analyze what would happen if for the same example $(\mathbf{x}^{(t)}, y^{(t)})$, we now use the parameters after the update, namely if we use w^i, b^i instead of w^{i-1}, b^{i-1} ?
$\longrightarrow$ Prove that the update "is beneficial" for the t-th example.

Sensitivity to the order of the examples:

- Suppose that the last training example is non typical $\longrightarrow$ it is likely to trigger an update of the parameters,
- which might be beneficial for this particular example
- but detrimental for the whole dataset
- bf Solutions:
  - shuffling of the training examples before each epoch
  - **averaged perceptron**
  - in practise, the training set is usually not linearly separable, or the perceptron is stopped before convergence (see infra)
  - **shuffling and averaging provide substantial performance gains**

Averaged perceptron:

- instead of returning the last version of the parameters $\mathbf{w}, b$, we return the average of their values at each iteration
- NB: average over all the iterations, whether or not there was an update
- The complexity of the algorithm is almost unchanged
  - while a naive computation of the average of the $\mathbf{w}^i$ is costly,
  - there exists a simple algorithm to do it efficiently (see lab session)

6.2 Perceptron algorithm in the multiclass case

We now need to learn a matrix $\mathbf{W} \in \mathbb{R}^{d \times C}$ and a vector of bias $\mathbf{b} \in \mathbb{R}^C$.
The only modification with respect to the binary case stands in the update step.

Perceptron algo for the multiclass case:

INPUT:
- X = training set of T examples : $\{(\mathbf{x}^{(1)}, y^{(1)}), ...\}$ with each $\mathbf{x}^{(t)} \in \mathbb{R}^d$ and $y^{(t)} \in \{1, 2, ..., C\}$
- **hyperparameter** E = positive integer = number of **epochs**

INITIALIZE $\mathbf{W}$ = null matrix in $\mathbb{R}^{d \times C}$ and a bias null vector $\mathbf{b}$ in in $\mathbb{R}^C$

LOOP: for each epoch in 1, ..., E:

LOOP: for each t in 1, ..., T:

- PREDICT class for t-th example given current $\mathbf{w}, b$: $\hat{y^{(t)}} = \underset{i \in \mathcal{Y}}{\text{argmax}}\ \mathbf{x}\mathbf{W}_{[*,i]} + b_i$
- if prediction is wrong, namely if $\hat{y^{(t)}} \neq y^{(t)}$

  Let's note the gold class $g = y^{(t)}$ for short

  and $s = \hat{y^{(t)}}$ the incorrect predicted class

  UPDATE the parameters:

  - add $\mathbf{x}^{(t)}$ to the weight vector of the gold class: $\mathbf{W}_{[*,g]} \longleftarrow \mathbf{W}_{[*,g]} + \mathbf{x}^{(t)}$
  - substract $\mathbf{x}^{(t)}$ to the weight vector of the incorrectly predicted class: $\mathbf{W}_{[*,s]} \longleftarrow \mathbf{W}_{[*,s]} - \mathbf{x}^{(t)}$
  - add 1 to b_g : $b_g \longleftarrow b_g + 1$
  - substract 1 to b_s : $b_s \longleftarrow b_s - 1$

OUTPUT: $\mathbf{W}, \mathbf{b}$

Analysis of the update step

Exercise: as for the binary case, analyze what would happen if for an example $(\mathbf{x}^{(t)}, y^{(t)})$ giving rise to an update at iteration i, we now use the parameters after the update, namely if we use $\mathbf{W}^i, \mathbf{b}^i$ instead of $\mathbf{W}^{i-1}, \mathbf{b}^{i-1}$ to predict the class of $\mathbf{x}^{(t)}$? **Hint:** recompute the new score of the gold class, and the new score of the incorrectly predicted class.
$\longrightarrow$ Prove that the update "beneficial" for the t-th example.

7 Methodology in ML

7.1 Evaluation metrics

When designing a classifier or a regressor, one needs to be able to assess its quality. This can be done via an intrinsic or an extrinsic evaluation:

- in **intrinsic evaluation**, the task is evaluated independently of its use in other (more sophisticated) tasks.
  - For instance, a syntactic parser can be evaluated by comparing its predicted parse trees to "gold" parse trees (namely manually-validated parse trees), independently of what these predicted syntactic trees could be used for.
- in **extrinsic evaluation**, the task is evaluated by measuring to which extent it helps another task.
  - For instance, suppose one designs a machine translation system which uses predicted parse trees of the source sentences. Two parsers A and B can be compared by comparing the machine translation quality (e.g. using the BLEU metric) when using parser A versus parser B.

In intrinsic evaluation, the general evaluation principle is to compare, for a given set of examples:

- the output predicted by the system
- and the reference output (the "gold" output) for these examples

7.1.1 Metrics for the regression task

To evaluate a regression model on a set of T examples $(x^{(1)}, y^{(1)}), ..., (x^{(T)}, y^{(T)})$, each gold value $y^{(t)}$ is compared to the predicted value $\hat{y}^{(t)}$.
Usual metrics are:

- **absolute error** $= \sum_{t=1}^{T} |y^{(t)} - \hat{y}^{(t)}|$
- **squared error** $= \sum_{t=1}^{T} (y^{(t)} - \hat{y}^{(t)})^2$: this gives a higher penalty to large errors.
- **mean squared error (MSE)** = the average squarred error (squarred error divided by T).

See other metrics e.g. in sklearn module : https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics

7.1.2 Intrinsic evaluation of a mono-label classifier

In this case, $y^{(t)}$ and $\hat{y}^{(t)}$ are categorical labels, either equal or different. The metric is the **accuracy**, namely the number of examples well classified by the system divided by the total number of test examples.

7.1.3 Intrinsic evaluation of a multi-label classifier

Accuracy cannot be used in this case, or more generally, accuracy cannot be used when the task doesn't inform the system concerning the gold **number** of expected answers for a given input.
Indeed, when there is a variable number of answers for a given input, the system thus may output a variable number of answers. Simply counting how many of the gold answers are actually output by the system is insufficient: it does not penalize wrongly predicted answers. Hence the use of two metrics recall and precision, penalizing silence and noise respectively.
Let us consider a multi-label classifier, for which we consider that one answer is one predicted class for a given input object.

- let G be the set of gold class instances, for the whole test set of T examples
- let S be the set of predicted class instances, for the whole test set
- the **noise** are the wrongly predicted classes $= S - (S \cap G)$
- the **silence** are the unpredicted gold classes $= S - (G \cap G)$
- **recall** = proportion of gold classes that are predicted $= \frac{|G \cap S|}{|G|}$
- **precision** = proportion of predicted classes that are correct $= \frac{|G \cap S|}{|S|}$
- recall penalizes silence, while precision penalizes noise

We can check that if the system knows in advance the nb of classes to predict for each object (e.g. a single class), then $|G| = |S|$ hence recall equals precision and both correspond to an accuracy.

Partial match versus exact match evaluation The evaluation we defined above corresponds to a **partial match** evaluation: for instance for an object x with gold classes c1 and c2, if the system predicts classes c1 and c3, the overall answer is not perfect, but partial credit is given for having predicted class c1.
An alternative is to evaluate using an **exact match** criterion: in the case of a multi-label classifier, for a given input object, exact match amounts to considering the set of gold classes as a whole, and count a prediction as correct if the set of predicted classes exactly matches the set of gold classes.
In such case, the system answers only one answer per input object (one answer = the set of predicted classes), and the evaluation can be a plain accuracy.

Example Provide the various metrics for a multi-label document classifier evaluated on the following data:

- document 1 : gold classes = c1, c3, predicted classes = c1, c3
- document 2 : gold classes = c2, predicted classes = c2, c3
- document 3 : gold classes = c3, c4, c5 predicted classes = c1, c4, c5

7.1.4 Fscore

To get a single score blending recall and precision, use the **Fscore** (or Fmeasure) metric, which is the harmonic mean of precision and recall (inverse of the mean of the inverses):

$$Fscore = \frac{1}{\frac{\frac{1}{R}+\frac{1}{P}}{2}} = \frac{2PR}{P+R}$$

Example:

- if R=0.99 and P=0.05, Fscore $\approx$ 0.01
- if R=0.5 and P=0.55, Fscore $\approx$ 0.5

The fscore is between 0 and 1. It will be low whenever recall and/OR precision is low. It will be high iff both are high.

7.1.5 Binary case with a minority class

- A frequent case is that of a binary classification task, for which one class is preponderant.
  - e.g. find out whether a patient has a certain disease or not
  - in NLP: for each word in a sentence, find out whether the word is a predicate or not
- usual vocabulary:
  - **positive** class = minority class
  - two booleans for each prediction
    - **positive / negative** class (for the predicted class)
    - **true / false**, for whether the prediction is correct or not
    - $\Longrightarrow$ 4 types of predictions: "true positive (tp)", "true negative (tn)", "false positive (fp)", "false negative (fn)"
- Example: 10000 word occurrences, among which 140 are (gold) predicates. Suppose a system predicts 160 predicate occurrences, 100 of which are correct.
  - draw and categorize the predictions according to the 4 types
  - compute the accuracy
- $\longrightarrow$ the accuracy will be very high, and uninformative
- an alternative is to evaluate predictions for the minority class only
- $\longrightarrow$ view the task as identifying which of the input objects belong to the minority class, and use recall and precision

7.1.6 Confusion matrix

A single accuracy score can be completed by an per-class analysis, in particular for the multi-class case: a **confusion matrix** shows the number of instances for each class, with gold classes in rows and predicted classes in columns (or vice-versa). TODO figure.

7.2 Using unseen-in-train evaluation examples : frozen split and cross-validation

In supervised learning, learning means be able to **generalize** over the training set. Hence the final evaluation should be performed on **examples that were not used at training time**.

Indeed, evaluation on the training examples can be inform us on how the training went, but it is insufficient to inform us on the generalization power of the trained system.

A common practise is to split the set of available examples into a test set and a remaining training set (see below for further splitting, in case of hyperparameter tuning).

It is common to use 10% of the examples as test examples. But if 10% correspond to less to ≈500 examples, evaluation on the test set will not be very precise. If the proportion of examples to reach 500 test examples is high (e.g. 1500 examples in total, keeping 500 for testing leaves only 2 thirds of the examples for training), then the evaluation procedure will evaluate a system trained on significantly less examples than if training on the full set of available examples. This means the evaluation will underestimate the achievable quality.

A common solution for this case is to use **cross-validation** ("validation croisée" en français):

- evaluation is performed on ALL the available examples
- but respecting the golden rule of not evaluating an example seen at training time
- $\longrightarrow$ **k-fold cross-validation** ("validation croiséepar rotation" en français)=
  - split the set of T examples in k parts (named "folds")
  - for i from 1 to k,
    - learn a system on all folds except the i-th fold
    - and use such system to predict outputs for the i-th fold
  - this provides a set of T predictions (made by k distinct systems though), which can be evaluated using standard metrics

Cross-validation is costly cf. it requires k trainings. Given that current neural systems require many trainings to tune the hyperparameters, cross-validation is rarely used, unless the amount of examples is really small.
K-fold cross-validation is a special case of the more general **leave-p-out** evaluation: evaluating p examples using a system trained on all but these p examples.
An extreme case if the **leave-one-out** case, corresponding to k-fold cross-validation with k=T, meaning that each fold contains a single example only. This is the most precise evaluation (the evaluated systems are trained on almost the full amount of available examples), but also the most costly (cf. it requires T trainings).
Note that along with numeric evaluation, **Error analysis** is crucial to have a better sense of the behavior of a learnt system: this means manually inspecting wrong predictions / correct predictions. This can give hints to characteristics of the inputs leading to wrong / correct predictions, and hence provide ideas to modify the system accordingly.

7.3 Tuning the hyperparameters: "development" or "held-out" data

Supervised systems often work with hyperparameters, namely variables that are given as input to the learning process (e.g. the number of epochs in perceptron learning, C and the kernel function in soft-SVM learning...).
Searching for good hyperparameter values is called **hyperparameter tuning**. This is done at the cost of several learning phases, using various hyperparameter values.
A **grid-search** of hyperparameter values means actually testing each combination, for a given set of values for each hyperparameter. E.g. each combination in soft-SVM:

- using feature f1 or not
- using feature f2 or not
- using kernel linear or RBF or polynomial
- using C = 0.0001 or 0.001 or 0.01 or 0.1 or 1 or 10 or 100
- $\longrightarrow$ this gives rise to $2 \times 2 \times 3 \times 7 = 84$ possible combinations.
- Note that as the number of hyperparameters increases, the number of possible combinations can become intractable.

Now to choose a single hyperparameter combination, one needs to train n systems (one per combination) and evaluate each system on a set of unseen-in-train examples. Hence in case of hyperparameter tuning, the standard practice is to split the available examples in three parts:

- **training set** : 80% for training
- **development set** (or held-out set, or validation set) : 10% for hyperparameter tuning (= to evaluate each hyperparameter combination). The chosen hyperparameters are optimal for this set. Choosing these particular hyperparameter values was done using this dev set, hence dev examples are actually "seen-in-train".
- **test set** : 10% for final testing, in order to test with really unseen-in-train examples.

7.4 Comparing systems' performance

An absolute performance value is rarely meaningful, unless compared to a **baseline**. E.g. what does it mean for a tagger to reach 90% of accuracy?
A baseline performance is that reached by a system which is either

- basic, easy to produce
  - in single-label classification, a baseline can be the accuracy reached when assigning the most frequent class in the training set ("**most frequent class baseline**");
  - a more informed baseline may use the classes seen in the training set for particular inputs: the most frequent class per input can be a better baseline. E.g. in part-of-speech tagging, suppose the most frequent tag overall is "noun". If "firm" was seen in the training set 5 times as an adjective, once as a verb, and once as a noun, then a baseline applied to a dev or test set will tag any occurrence of "firm" with tag "adjective". Words not seen in the training set will receive the overall most frequent tag (noun).
- or a historically well-established technique (and one wants to check whether this well-known technique is surpassed or not)

More generally, choosing a given classifying method (including feature representation, learning algorithm, network architecture, hyperparameter values etc...) implies to compare the performance of various systems against the same set of examples. Note that **statistical significance** of performance differences should be carefully evaluated.

- e.g. suppose two taggers A and B obtain an accuracy of 96.9 and 97.1 respectively, when evaluated on a dev set of 2000 word occurrences. Can this 0.2 point difference be interpreted as tagger B being better, or is it simply due to chance?

TODO: methods for statistical significance of classification results[1]. Classic significance test is:

- for binary classification: McNemar's test see e.g. https://en.wikipedia.org/wiki/McNemar%27s_test, and the chi-square distribution (loi du χ^2) https://en.wikipedia.org/wiki/Chi-squared_distribution
- for multi-class classification:
  - either consider only "correct" versus "incorrect" output and use McNemar's test
  - or use the "McNemar-Bowker Test of Symmetry" https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/PASS/Tests_for_Multiple_Correlated_Proportions-McNemar-Bowke Test_of_Symmetry.pdf

7.5 Learning curves, underfitting, overfitting and early stopping

A learning curve is a plot of the performance reached on a certain dataset, as a function of the number of training examples or as a function of the number of training epochs (one epoch being an online use of the full set of training examples).

TODO Figure

Plotting the performance (accuracy or fscore or learning loss) achieved on the TRAINING SET allows to check that some learning does take place.

- in general, performance on the training set should increase following a kind of log-like shape

[1] See in particular T. G. Dietterich, "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms," in Neural Computation, vol. 10, no. 7, pp. 1895-1923, 1998 https://ieeexplore.ieee.org/document/6790639

- if performance on the training set does not become asymptotic when adding data: adding more data should help (but additional data may be expensive to get)
- if performance on the training set is chaotic: something is wrong! The model might be **underfitting** ("sous-ajustement"), namely the vector representation + architecture of the classifier are not enough powerful to achieve the task, even on the seen-in-train examples
  - the vector representation should be modified, or the number of training examples (often unavailable though)
- but a good performance on the training set may be due to irrelevant characteristics of the training examples, which will not be useful to classify new unseen inputs: this situation is called **overfitting** ("sur-ajustement" or "surapprentissage"). The model tries to fit all the characteristics of the training examples (see image below). The resulting model performs very well on the training set, but fails to generalize.
  - In order to fit the characteristics of the training examples, the model gets too complicated.

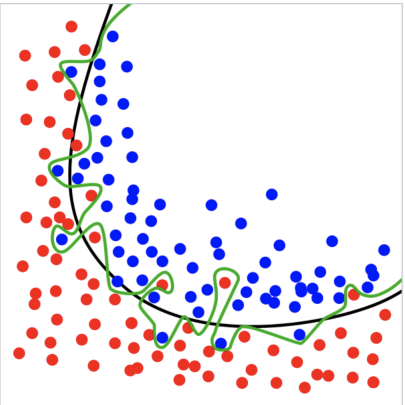


Figure 1. The green line represents an overfitted model and the black line represents a regularized model. While the green line best follows the training data, it is too dependent on that data and it is likely to have a higher error rate on new unseen data, compared to the black line.

(from Wikipedia)

Several techniques are used to combat overfitting:

- in linear models, regularization terms are added to the loss function to minimize (see below). Regularization terms aim at keeping the weights $\mathbf{W}$ low, enforcing that learning perform only small modifications of the parameters. Indeed, when overfitting, a model has to adjust to details in the training set. Small differences in two training examples having different gold classes may have to result in big variations in the model's prediction scores. Enforcing that weights remain low will thus tend to limit overfitting.
- in neural models, dropout techniques can be used (see below)
  - train the model while randomly ignoring certain neurons
  - will enforce that a given neuron cannot rely on the use of other neurons,
  - dropout is thus said to prevent complex co-adaptations of neurons (neurons firing only in the context of other ones)
  - and experimentally proves very efficient at preventing overfitting
- in both cases, stopping the learning before overfitting can prove beneficial. This is done via the technique of **early stopping**: the number of training epochs is considered as a hyperparameter, which is optimized using the dev set (as other hyperparameters).
  - After each epoch, loss (see below) or performance on the dev set is computed, using the current values of the learnt parameters $\longrightarrow$ learning can be stopped when performance (resp. loss) decreases (resp. increases) on the dev set.
  - Fancier rules can be used, such as having "patience": wait for a certain nb of epochs with decreased performance before stopping the learning.

8 Capacity of linear, log-linear, neural models

8.1 Capacity of a linear regressor

Limitations due to the linear hypothesis class :

- among an infinity of mathematical functions $\mathbb{R}^d \longrightarrow \mathbb{R}$
- we limit ourselves to the set of affine functions
- the **hypothesis class** is that f takes the form $\mathbf{x}.\mathbf{w} + b$
- one particular hypothesis is when choosing particular values for $\mathbf{w}$ and b
- this introduces what is called an **inductive bias ("biais inductif")** : prediction is constrained by the family of predictive functions.

The vector representation of the object is also part of the inductive bias.

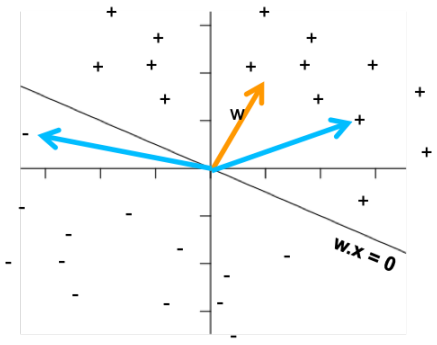
8.2 Capacity of a linear or log-linear classifier

8.2.1 Decision boundary (separating hyperplane)

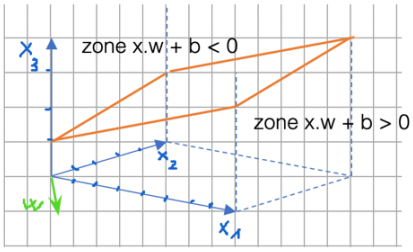
The **decision boundary** is the set of points in $\mathbb{R}^d$ that delineate a frontier between regions of $\mathbb{R}^d$ for which the predicted class differs: crossing the boundary will result in a different prediction.

In binary linear classification :

- the decision boundary is the set of points satisfying $\mathbf{x}.\mathbf{w} + b = 0$: **cf. these are the points for which the decision is most uncertain.**
- This equation defines a **hyperplane**: a subspace of $\mathbb{R}^d$ whose dimension is $d - 1$
- hence a synonym of decision boundary in case of linear classification is **separating hyperplane**.
- The separating hyperplane divides $\mathbb{R}^d$ into two regions : one for which $\mathbf{x}.\mathbf{w} + b > 0$ and one for which it is < 0
- if d=1, decision boundary
  - = points (x_1) for which $x_1 w_1 + b = 0$
  - = **single point** (hyperplane with dimension d-1=0) $x_1 = -b/w_1$
- if d=2, decision boundary
  - = points (x_1, x_2) satisfying $x_1 w_1 + x_2 w_2 + b = 0$
  - = **line** with equation $x_2 = -(w_1/w_2)x_1 - b/w_2$
  - the separating hyperplane has dimension d-1 = 1



- if d=3, decision boundary
  - = points (x_1, x_2, x_3) satisfying $x_1 w_1 + x_2 w_2 + x_3 w_3 b = 0$
  - = **plane** with equation $x_3 = -(w_1/w_3)x_1 - (w_2/w_3)x_2 - b/w_3$
  - the separating hyperplane has dimension d-1 = 2



- plan d'équation x3=1/4 x1 + 1/5 x2 + 1
- qui peut s'écrire 1/4 x1 + 1/5 x2 - x3 + 1 = 0
- donc avec w = (1/4, 1/5, -1) et b = 1 on a bien x.w + b = 0 pour tous les x sur ce plan

NB: in the figures, be careful to distinguish between having all axes corresponding to components of $\mathbf{x}$, versus having one axis corresponding to y (in regression).
NB: in NLP classifiers, d is very big, the separating hyperplane is a quite complex vector space, cf. d-1 is still very big.

8.2.2 Linear separability

Let X be a set of T examples $\{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(T)}, y^{(T)})\}$, with each $\mathbf{x}^{(t)} \in \mathbb{R}^d$ and $y^{(t)} \in \{+1, -1\}$.

Definition : a linear classifier defined by $\mathbf{w}, b$ **separates** the set X iff $\forall t y^{(t)}(\mathbf{x}^{(t)}.\mathbf{w} + b) \geq 0$

Definition : the set X is said **linearly separable** iff $\exists \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$ that separates X.

Rem: If one there is one linear classifier separating X, then there are an infinity of other classifiers that separate it (multiplied by a positive scalar).

TODO Figure : separable case, non separable case.

A linear classifier will never be able to classify X perfectly if X is not linearly separable.
The same goes for a log-linear model: as seen above in section ???, switching to a log-linear model (i.e. adding a sigmoid or a softmax after the linear combination) does not change the classification algorithm, hence log-linear models have the same capacity as linear models: they are limited to linear decision boundaries, hence:
A log-linear classifier will never be able to classify X perfectly if X is not linearly separable.

How do we know whether X is linearly separable? : there is no general algorithm to compute linear separability for any d and X. Now, some algorithms, such as the perceptron (cf. section 5.5), are guaranteed to find a separating classifier if there exists one.

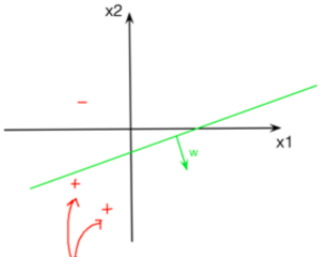
In practice : linear classifiers are applied to sets, **even though we don't know about the linear separability of these sets.**

8.3 Properties of the perceptron algorithm

8.3.1 Margin and optimal classifier (in the binary linear classification case)

To look into some properties of the perceptron, we need to introduce the notion of **margin**, which captures

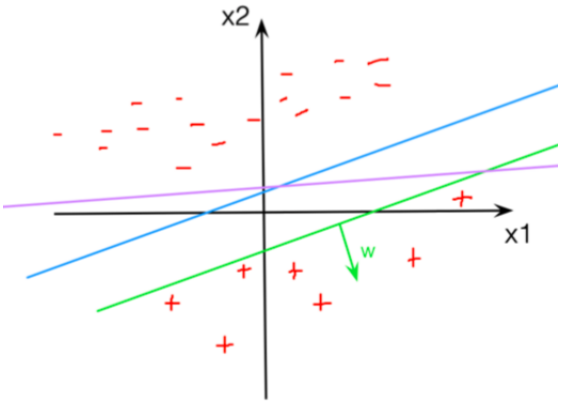
- how easy/confident a given classifier is to classify an example, or a set of examples,
- and more generally how easy it is to find a classifier to separate a given set of examples X
- **Margin of a pair (classifier, example)**



The classifier w,b, does classify well the two red examples, but with a different level of confidence (here the distance to the decision boundary).

- the margin of a pair ((w,b), (x,y)) = the level of confidence in the classification decision of x
- Geometrical interpretation: whether x is on the right side of the decision boundary, and how far it is from it (distance from x and the decision boundary
- Analytical interpretation: do x.w+b and y have same sign, and how big is $|x.w + b|$ (how far from 0)
- hence the **definition of margin for a classifier (w,b) and an example (x,y) : margin(x,y,w,b) = y(x.w + b)**
- the margin is ≥ 0 iff x is well classified by (w,b)
- When considering the case with integrated bias : margin(x,y,w) = y(x.w)
  - It can be shown that $|\mathbf{x}.\mathbf{w}| = ||\mathbf{w}|| \times$ distance from x to the hyperplane$\mathbf{x}.\mathbf{w} = 0$
  - Hence the **margin is proportional to the norm of w** and can grow artificially if we take another classifier $\alpha\mathbf{w}$
  - To remove this dependency to the norm of w, we can consider classifiers that have norm 1 only
    - for a given classifier $\mathbf{w}$, we consider instead $\mathbf{w}' = \mathbf{w}/||\mathbf{w}$, which has norm 1
    - in such case margin(x,y,w') is directly equal to the distance (or -distance) to the hyperplane

- **Margin of a classifier w,b and a whole set of examples X**
  - defined iff w,b separates X
  - corresponds to the margin for the example in X for which the classifier is **least confident**
  - **margin(X, w,b) = $\mathbf{min}_{(x,y)\in X}$ margin(x,y,w,b)**
  - this can be used to **compare** how well classifiers do for a given set X
  - in the figure below, the classifier in blue is better than the green one, **it has a greater margin : margin(X, blue classifier) > margin(X, green classifier)**.



- **Margin of a set of examples X** is the best possible margin for any classifier of norm 1
  - defined iff X is linearly separable
  - margin$(X) = max_{w/||w||=1}$margin(X, w)
  - (the norm of w has to be bound, otherwise the margin would artificially grow)
- the **optimal classifier** is the one with maximal margin
  - optimal classifier w^* = argmax$_{w/||w||=1}$margin(X, w)

8.3.2 Convergence of the perceptron

We can now turn to the properties of the perceptron algorithm:
It has been proved if the training set X is linearly separable, then the perceptron converges (no more updates) after a finite number of iterations. The algorithm is said to be **complete**.

Note that when the perceptron converges, its solution $\mathbf{w}, b$ separates X (cf. no more updates mean every example is well classified) : the perceptron algo is said to be **correct**.

Convergence speed : the convergence speed is expressed as an upper bound to the number of updates that are necessary to separate the training set.

- **Theorem** (Rosenblatt, 57)
  - Let X be a training set with **margin** γ (**which implies it is separable**), such as $\forall (x, y) \in X, ||x|| \leq R$
  - the perceptron using X as training set will converge after at most R^2/γ^2 updates
- **Analysis:** the max nb of updates depends on how the margin compares to the maximal norm found in X

The classifier output by the perceptron algorithm is not optimal : if X is linearly separable, the perceptron will converge to a solution $\mathbf{w}, b$ that separates X, **but this solution is not the optimal classifier for X**, namely, it does not reach the maximal reachable margin.

On the contrary, the support vector machines do aim at finding the optimal classifier.

8.4 A glimpse at support vector machines (SVMs)

The objective of this section is to provide the general ideas behind SVMs, and be able to understand the hyperparameters typically defined when using packages implementing SVMs.

SVMs = support vector machines were initially defined for the binary classification case (Cortes and Vapnik, 1995).

The extension to the multiclass case is either performed

- using an OVO or OVA strategy
- or using a intrinsically multiclass formulation of the SVM algorithm (Crammer and Singer 2001)

In all this section, and we stick to the binary classification case, and we forget the bias (we can suppose it is integrated in the vector space, cf. above).

Maximize margin versus minimize $||w||$ We've seen that the perceptron is not meant to find a classifier that maximizes the margin of X. On the contrary SVM do. More precisely:

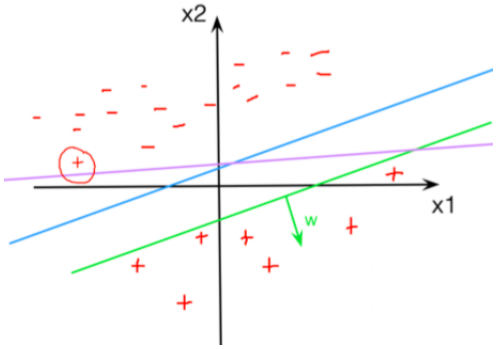
- instead of finding a w,b maximizing the margin, under the constraint that $||w|| = 1$ (cf. maximizing the margin can only be an objective if $||w||$ is bound, otherwise the margin can grow indefinitely)
- SVMs arbitrarily set the value of the margin to 1, and seek to find the w with lowest norm
  - for instance, let (x, y) be an example, and 2 vectors $\mathbf{w}$ and $\mathbf{w}'$ such as the margin is 1 for both classifiers for this example: $y(\mathbf{x}.\mathbf{w} = y(\mathbf{x}.\mathbf{w}' = 1$
  - the distances between x and the hyperplanes defined by $\mathbf{w}$ and $\mathbf{w}'$ are $d = 1/||w||$ and $d' = 1/||w'||$ respectively.
  - hence, for the same margin (here 1), the classifier with lowest norm will correspond to a higher distance to the separating hyperplane
- **SVMs use the equivalence that, for a given training set X, finding a w that maximizes the margin under the constraint $||w|| = 1$ is equivalent to find a try to find a w with minimal norm $||w||$ under the constraint that all examples are classified with margin ≥ 1.**

Hard margin SVM In the original formulation, SVMs search for the minimal w so that all examples are classified with margin at least 1:

$$\underset{w\in\mathbb{R}^d}{\text{argmin}}||w|| \text{ under the constraint } \forall t, y^{(t)}(x^{(t)}.w) \geq 1$$

This can only be found if X is linearly separable. In such case, all the examples are well classified by the found w, and the examples classified with least confidence are those having margin 1. These are called the **support vectors**.

Soft margin SVM But if X is not linearly separable, finding such a classifier is not possible. And even if X is linearly separable, it may have outliers, which implies that the best w for X might work less well for many points in X, just because it tries to accomodate to the outlier: compare the purple and the blue hyperplanes below. Ignoring the (circled) outlier, allows the blue classifier to be much more confident than the purple one for most data points.



Hence the idea to introduce one positive **slack variable** ζ_t per training example $(x^{(t)}, y^{(t)})$. The mathematical formulation below allows the margin for a given example to be not strictly 1, but less than 1 (1-ζ_t), meaning that $x^{(t)}$ can in the end be less well classified, or even misclassified (if the margin gets negative). The algorithm uses a hyperparameter C, and looks for:

$$w^*, \zeta^* = \underset{w\in\mathbb{R}^d, \zeta\in\mathbb{R}^T}{\text{argmin}} ||w||^2 + C\sum_t \zeta_t \text{under the constraint that } \forall t, \zeta_t \geq 0 \text{ and } y^{(t)}(x^{(t)}.w) \geq 1 - \zeta_t$$

The **hyperparameter C** controls how much the resulting classifier will tolerate less-well or wrongly classified examples.

- the bigger C is, the more letting the ζ_t grow will contradict the objective of having all examples well classified with at least margin 1
- hence the bigger C, the closer we are to the hard margin case
- the lower is C, the more we allow mis-classified examples
- $\longrightarrow$ the soft SVM algo can be used for non linearly separable cases, and/or cases with outliers
- in practise, C can be set empirically (see tuning of hyperparameter, in methodology in ML below). Tested values are typically 100, 10, 1, 0.1, 0.001, 0.0001 etc...

Learning of the parameters using the (soft or hard) SVM formulation above corresponds to solving a convex optimization problem, meaning there exists a unique solution (=values of the parameters) of the constrained minimization problem. Training algorithms to find this solution is beyond the scope of this course[2].

8.5 What if X is not linearly separable?

The training set may very well not be linearly separable. Moreover, we don't have any general algorithm to check whether it is. Here are four general solutions to that problem:

1. Solution 1 is simply to use a linear method even if X is not linearly separable. E.g. using a perceptron with a certain number of epochs → the obtained classifier won't separate X, but might work rather well on subsequent test data.
2. Solution 2: SVM with soft margin (see below): the soft SVM learning algorithm introduces some parameters that will allow to better cope with outliers / difficult data points.
3. Solution 3 is using kernels. The idea is to transform the vector space in which the objects are represented, into a vector space having higher dimension. The higher the dimension, the more likely it is for the data to be linearly separable (because the hyperplane has dimension d-1, so if d is bigger, then d-1 is still a very rich vector space).

[2]See e.g. https://shiliangsun.github.io/pubs/ROMSVM.pdf

- if an object to classify is initially represented by a vector $\mathbf{x} \in \mathbb{R}^d$, then a **transformation function** f can be applied, so that $f(\mathbf{x}) \in \mathbb{R}^{d'}$ with $d' >> d$. In particular, in the higher vector space, combination of features can be added. E.g. turning vector (x_1, x_2) into $(x_1, x_2, x_1^2 + x_2^2)$. The target representation is said to lie in a "redescription vector space". The intuition is that data will more likely be linearly separable in the redescription vector space.
- In practise, thanks to the **kernel trick**, one can avoid to actually transform the input vectors, but only transform the dot product instead. Using functions having specific properties, called **kernel functions**, one can replace the dot product by a kernel function everywhere in linear classifiers.
- Kernel functions are often used with SVMs. Before the neural era, soft SVMs with kernels were often the most powerful classification methods.

4. Solution 4: the current solution is to use neural networks with non-linearities, which can have non-linear decision boundaries (see below 8.6.3).

8.6 Capacity of a MLP

8.6.1 The XOR problem

We've seen that (log)linear binary classifiers define scoring functions $f(x, \theta)$ (with $x \in \mathbb{R}^d$ and θ the parameters) such as the decision frontier, namely the x points satisfying $f(x, \theta) = 0$ is an hyperplane : this restricts the possible learnt scoring functions.[3]
In contrast, we don't have general results restricting the class of scoring functions that MLPs can represent. But we have a much more powerful **universal approximation theorem** (see below).
The XOR problem is a very famous illustration of limitations of (log)linear classifiers:

- suppose we see logical operators as binary classifiers outputing +1 or -1
- there are 4 possible inputs in $\mathbb{R}^2$: (1,1), (-1,1), (1,-1), (-1,-1)
- e.g. for AND, the gold output classes are +1, -1, -1, -1 respectively
- for XOR, the gold output classes are -1, 1, 1, -1 respectively

Exercise: draw the AND dataset and possible separating hyperplanes. Same question for XOR.

8.6.2 Adding dimensions can solve the problem

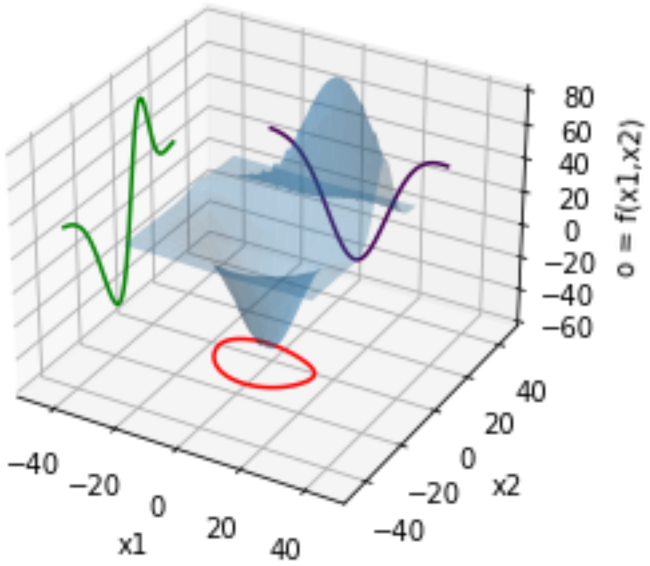
Suppose we transform out input vectors in $\mathbb{R}^2$ into vectors in $\mathbb{R}^3$ using the transformation : $\phi(x_1, x_2) = (x_1, x_2, x_1 x_2)$. Draw the resulting dataset and possible separating hyperplane.
To linearly separate the 4 resulting data points, e.g. $\mathbf{w} = (0, 0, 1)$ and b = 0 works!

This is handy but if d is high, then the transformation can lead to a very high dimensional space, and slow down computations. This is at the core of **kernel methods** (cf. above section 8.5): the "kernel trick" refers to the trick of not explicitely computing transformed vectors in the higher-dimensional vector space $\phi(\mathbf{x})$ but to use a "kernel function" K instead of dot products, **in the original vector space**: functions so that $K(\mathbf{x^1}, \mathbf{x^2}) = \phi(\mathbf{x^1}) \cdot \phi(\mathbf{x^2})$.
So you can do linear classification replacing the dot product everywhere by a kernel function, resulting in a linear classification in the high-dimenstional "redescription" space, but corresponding to a non-linear classification in the original space.

8.6.3 Universal approximation theorem

With a MLP with at least one non linear activation function in internal layer(s), the decision frontier is no longer necessarily linear, as shown in the illustration below: suppose we work with d=2, and a MLP for binary classification outputs the score $f(x_1, x_2)$ for class 1. The decision frontier is made of the points satisfying $f(x_1, x_2) = 0$), shown in red in the illustration below:

[3] Note that adding a sigmoid (or softmax) function at the end of the scoring does not modify the decision frontier, that's why this is true of log-linear models (logistic regression).



(generated using a modified version of code at
https://matplotlib.org/stable/gallery/mplot3d/contour3d_3.html)

On the contrary, With d=2, a linear decision frontier would be a line.

Now, suppose

- we have a very complicated set of examples $X = \{(x^{(1)}, y^{(1)}), ...\}$, for the binary classification case, with d=2, in which some positive examples and negative examples are difficult to separate.
- And suppose we draw X on a horizontal plane, and have a function $f(x_1, x_2)$ on the vertical axis.
- We can imagine a complicated continuous function from $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$, working like a "**blanket**", which forms a hill around any positive example, with $f(x_1, x_2) > 0$ and a basin around any negative example ($f(x_1, x_2) < 0$).
- $\longrightarrow$ Such a function f would be a perfect classifier, separating X perfectly (non linearly though).

The good news with MLPs is that no matter how complicated such a f function is, if f is continuous and bound, then there exists a MLP that **approximates** f !!

More precisely, the **universal approximation theorem** (Hornik et al., 89; Cybenko, 89) states that

- for any function f continuous and defined from a compact subset[4] of $\mathbb{R}^D$ to $\mathbb{R}$
- there exists a binary MLP with
  - a single hidden layer, with a finite nb of neurons
  - and sigmoid activation
  - a single neuron in the output layer, with no activation
- that approximates f

Results were then extended to cover

- functions outputing several values, hence usable for multiclass classification
- and other activation functions: any bound non-constant activation function (Hornik et al., 91) (hence sigmoid, tanh, hard tanh), and even ReLU (Leshno et al., 93)

This means that functions defined using MLPs, namely precise building blocks (linear combinations + activation functions) are enough to separate any finite training set whose vector components are bound.

Limitations of the theorem: unfortunately, the theorem does not provide results concerning

- the required size of the hidden layer
- nor whether such an approximating MLP is learnable using the set X

[4]Providing the precise definition of a compact subset is beyond the scope of this course, but we can give a simple particular case of compact subset: if I is an interval in $\mathbb{R}$, then I^D is a compact subset of $\mathbb{R}^D$. Hence continuous functions whose input vectors have bound values are covered by the theorem.

- the non-linear activation function is crucial
- **NB:** for any MLP with a single hidden layer with no activation function or a linear activation function, there is an equivalent network with 0 hidden layer (hence a linear function). Equivalent means that it defines the exact same function.
- **From the theoretical point of view**, having more than 1 hidden layer does not change the descriptive capacity of MLPs
- **From the practical point of view**:
  - it might be empirically found beneficial to use 2 or more hidden layers (with non linear activation functions)
  - depending on the task, it is custom to use hidden layers with a few dozens or hundreds of neurons

9 Learning as minimizing a loss function

Suggested readings:

- quick summary within the logistic regression chapter in Jurafsy and Martin : https://web.stanford.edu/~jurafsky/slp3/5.pdf : cross-entropy, gradient descent, regularization
- Goldberg Y. 2017 sections 2.7 and 2.8
- more advanced readings: Hal Daumé 7.1 to 7.5, or Bengio et al. 2016 : https://www.deeplearningbook.org/

The general principle for supervised machine learning algorithms is to use:

- a set $X = (x^{(1)}, y^{(1)}), ..., (x^{(T)}, y^{(T)})$ of training examples
- a system's scoring function f, which scores outputs given an input x and some **parameters** θ
  - in regression, $f(\mathbf{x}, \theta)$ is a real number
  - in binary classification, $f(\mathbf{x}, \theta)$ is a score before applying the sign function to actually predict class +1 or the other class (e.g. the linear combination $w.x + b$ in linear binary classification);
  - in multiclass classification, $f(\mathbf{x}, \theta)$ is the vector of the scores for the C classes
- an **objective function** J defined for X (or a subset of X), using f and θ

The basic principle is then to search for the parameter values that minimize or maximize the value of the objective function, when computed using the set of training examples. The search can be exact (finding an analytic solution) or approximate (in general using iterative refinements of the parameter values).

- If J increases (resp. descreases) when the performance of the system increases, then the goal will be to maximize J (resp. minimize J).
- It is more usual though to define J as decreasing when performance increases: this is typical of functions measuring the **error** of a system. In such cases, the objective of the learning algorithm will be to find parameters **minimizing** the objective function. The usual name for such objective functions is **cost** or **error** or **loss** function (we will use loss). That's why in the following we will use L instead of J for the name of the function to minimize.

We will use $l(x, y, \theta)$ for the loss computed for one example, and $L(X, \theta)$ for the loss computed for a set X of examples, which is in general simply the sum or the average of the losses for each example.
The goal of the learning is find an exact or approximate solution to the minimizing problem:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\ L(X, \theta) = \underset{\theta}{\operatorname{argmin}} \sum_{t=1}^{T} L(X, \theta)$$

NB: actually, the real objective is to find parameter values that will provide minimum loss for examples **unseen** in the training set. Indeed, trying to minimize loss (=maximize performance) for the training set may lead to overfitting. Hence the technique of early stopping mentioned in section 7.5, as a way to combat overfitting.

9.1 Usual loss functions

For regression tasks, the typical loss function is the "mean squared error" (MSE):

$$L(X, f, \theta) = \sum_{t=1}^{T} \sum_{t=1}^{T} (f(\mathbf{x^{(t)}}, \theta) - y^{(t)})^2$$

MSE is minimal and =0 when all predicted values are exactly the gold values.
Learning searches for $\theta^* = \underset{\theta}{\text{argmin}} \sum_{t=1}^{T} (f(x^{(t)}, \theta) - y^{(t)})^2$
For example, if the regression model is linear, then the parameters θ consists in a vector of weights $\mathbf{w}$ and a bias b. The MSE can be rewritten as $L(X, f, \theta) = \sum_{t=1}^{T} (\mathbf{w}.\mathbf{x^{(t)}} + b - y^{(t)})^2$. There exists an **analytical solution** to the minimizing problem (see e.g. section 7.6 in CIML Hal Daumé III, or 5.1.4 in Bengio et al. 2016 https://www.deeplearningbook.org/contents/ml.html.

For classification tasks, we can distinguish margin-based losses, and cross-entropy loss, the latter being defined when scores correspond to probabilities of belonging to a certain class.

9.1.1 Margin-based losses for classification tasks

The simplest loss function is the "zero-one loss", which simply counts the number of misclassified examples.

Note that if f is linear, and X is linearly separable, the perceptron algorithm is guaranteed to find parameter values that minimize the 0-1 loss, reaching a loss of zero.

But in the non-linearly separable case, the problem of minimizing the 0-1 loss is NP-hard (no known algorithm in polynomial time).

That's why other loss functions are used, having good mathematical properties for the minimizing problem. In particular, **convex** loss functions are ideal to minimize, because by definition they have a single minimum.
Margin-based losses are defined using the classification margin of each example. Recall that margin is defined to measure the confidence with which the prediction algorithm makes the right prediction. Hence:

- for binary classification $\text{margin}(x, y, f(x, \theta)) = yf(x, \theta)$
- for multiclass classification, the margin is the difference between the score of the gold class and the maximum score among non gold class scores. Mathematically, we can write $\hat{y} = f(x, \theta)$, and $\hat{y}_y$ is the component of $\hat{y}$ for the gold class, and s is the highest scoring non-gold class. Then $\text{margin}(x, y, f(x, \theta)) = \hat{y}_y - \hat{y}_s$. The margin is positive if the system correctly ranks the gold class y first, and is negative otherwise.

Famous margin-based losses are:

- **Hinge loss** : $l^{hinge}(margin) = max(0, 1 - margin) = \begin{cases} 0 & \text{if } margin \geq 1 \\ 1 - margin & \text{otherwise} \end{cases}$
  - Hinge loss is null iff margin is sufficiently high (arbitrary threshold of 1)
  - equals 1 when margin is 0
  - penalizes cases for which the margin is < 1, hence not only the misclassified ones, but the classified ones with insufficient margin.
- **Logarithmic loss** : $l^{log}(margin) = \frac{log(1+e^{-margin})}{log(2)}$
  - this is a smoothed version of the hinge loss
  - cf. equals 1 when margin is 0, tends towards 0 when margin tends towards +inf

TODO figure loss as a function of the margin

The important traits of loss functions are:

- whether or not there is a penalty (= a non-null loss) even for well classified examples
- and how the loss behaves for very badly classified examples: if the loss is too high in such case, this can render the training too sensitive to outliers.

9.1.2 Cross-entropy loss for probabilistic classification tasks

In case of classifiers outputing probabilistic scores (e.g. using sigmoid for binary classification or using softmax for multiclass classification), the most famous loss is the **cross-entropy loss**. Minimizing such a loss corresponds exactly to trying to maximize the probability of the training set, as computed by the system, namely this corresponds to **maximum-likelihood estimation**.

Indeed, let $P(X|\theta)$ denote the joint probability of the training examples, as computed using the probability distribution of the inputs (the $x^{(t)}$s) and the probability of classes for each example, computed using the parameters θ (note that viewing $P(X|\theta)$ as a function of θ, with the set X given corresponds to the **likelihood** of the parameters). Then :

$$
\begin{aligned}
\theta^* &= \underset{\theta}{\text{argmax}}\ P(X|\theta) \\
&= \underset{\theta}{\text{argmax}}\ \textstyle\prod_{t=1}^{T} P(obj = x^{(t)}, class = y^{(t)}|\theta) && \text{\#supposing the examples are all independent} \\
&= \underset{\theta}{\text{argmax}}\ \textstyle\prod_{t=1}^{T} P(obj = x^{(t)})P(class = y^{(t)}|object = x^{(t)}, \theta) && \text{\#by applying the chain rule} \\
&= \underset{\theta}{\text{argmax}}\ \textstyle\prod_{t=1}^{T} P(class = y^{(t)}|obj = x^{(t)}, \theta) && \text{\#because the } P(obj = x^{(t)}) \text{ are constant} \\
&= \underset{\theta}{\text{argmax}}\ \textstyle\sum_{t=1}^{T} log(P(class = y^{(t)}|obj = x^{(t)}, \theta)) && \text{\#because log is an increasing function} \\
&= \underset{\theta}{\text{argmin}}\ \textstyle\sum_{t=1}^{T} \text{-}log(P(class = y^{(t)}|obj = x^{(t)}, \theta)) && \text{\#minus to get a loss}
\end{aligned}
$$

Hence **maximizing the likelihood ⇔ minimizing the negative log-likelihood**.

Hence the definition of the **negative log-likelihood** for an example $(x^{(t)}, y^{(t)})$:

= **negative log-likelihood (NLL)**

= also called **cross-entropy loss** (entropie croisée)

= - log of the probability of the gold class

= $-log(P(class = y^{(t)}|obj = x^{(t)}, \theta))$

- corresponds to the objective of maximizing the probability of the gold class
- we search for the parameters that maximize the probability of the observed classes of the training examples

NB: Note that the NLL loss only uses the score of the gold class: trying to maximize the probability of the gold class will mechanically decrease the probability of all other classes.

NB: The NLL loss for an example will be 0 if the probability of the gold class is 1 and tends towards -inf when proba of the gold class tends towards 0.
TODO figure -log for prob values

Notational trick in binary probabilistic case :

- In such a case, it is custom to consider that the classes are either +1 or 0 (instead of +1 and -1). So the $y^{(t)}$ are either 1 or 0.
- And in the binary case, $\hat{y}^{(t)} = f(x^{(t)}, \theta) = P(class = +1|x^{(t)}, \theta)$
- NLL loss is $\begin{cases} -log(\hat{y}^{(t)}) \text{ if } y^{(t)} = 1 \\ or \quad -log(1 - \hat{y}^{(t)}) \text{ if } y^{(t)} = 0 \end{cases}$
- hence the notational trick : $NLL(y^{(t)}, \hat{y}^{(t)}) = -y^{(t)}log(\hat{y}^{(t)}) - (1 - y^{(t)})log(1 - \hat{y}^{(t)})$

10 Gradient-based optimization and stochastic gradient descent

We've just seen that learning parameters can be formulated as a minimization problem.

$$\theta^* = \underset{\theta}{\text{argmin}}\ Loss(X, \theta)$$

More generally, the objective function to minimize can be a loss plus a regularizer term. Hence we will use J for the function to minimize, the simplest case is J = Loss.

Minimizing or maximizing problems are the same (cf. maximizing function g ⇔ minimizing $-g$, and are called **optimization** problems (in the sense of finding the optimum of a function). In all the following

we consider the minimization problem.

A family of optimization algorithms uses the **gradient** (see below section 10.3) of the objective function to minimize. We will sketch first the most famous optimization algo: gradient descent and stochastic gradient descent (SGD), and then provide some mathematical backgroung on derivatives and gradients to understand the update step at the heart of SGD.

10.1 Gradient descent algorithm

Input:

- a set X of training examples $X = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), ..., (\mathbf{x}^{(T)}, y^{(T)})\}$
- the objective function J, taking X and $\theta \in \mathbb{R}^n$ as arguments. X is constant, and the optimization searches for $\theta^* = \underset{\theta}{\operatorname{argmin}}\ J(X, \theta)$
  - J is typically a loss function L
  - or a loss function plus a regularizer term (intended to prevent overfitting, in general by preventing parameters to grow too much)
- $\eta \in \mathbb{R}^+$: the learning rate
- a stopping condition (see below)

Algorithm:

- initialize n variables θ_{val} with random values
- while stopping condition not met
  - compute J at point X, θ_{val} (generally needed to compute the gradient)
  - $\mathbf{g} \leftarrow$ the gradient vector of J with respect to variables θ, at point X, θ_{val}
  - (see below mathematical reminders on gradient : $\nabla_\theta L(X, \theta_{val})$)
  - update parameters: $\theta_{val} \leftarrow \theta_{val} - \eta \mathbf{g}$
- **We will see below why this update of the parameters is likely to be beneficial, namely is likely to move the parameters so that J actually decreases.**
- to summarize, if J is a convex function (= having a unique minimum), then GD converges to find the actual argmin, provided η is small enough

10.2 Stochastic gradient descent

Typically, $J(X, \theta)$ is a sum over all the examples in X: $J(X, \theta) = \sum_{t=1}^{T} j(x^{(t)}, y^{(t)}, \theta)$.

j is a loss function for one example, sometimes plus a regularizer term.

GD computes the loss and the gradient summing over the full set X. Each update is hence quite **long to compute** if the number of examples is large. Alternatives are to consider

- one example at a time $\longrightarrow$ stochastic gradient descent.
- or rather subsets of examples (which used to be called **mini-batches** but tend now to be simply **batches**) $\longrightarrow$ (mini-)batch gradient descent

Stochastic Gradient Descent:

- initialize n variables θ_{val} with random values
- while stopping condition not met
  - sample one example (x, y) from X
  - compute j at point x, y, θ_{val}
    - (in general through a forward propagation,
    - in general needed for the gradient calculation)
  - $\mathbf{g} \leftarrow \nabla_\theta j(x, y, \theta_{val})$ the gradient vector of j with respect to variables θ, at point x, y, θ_{val}
    - in general via the backpropagation algorithm (for an efficient computation)
  - update parameters: $\theta_{val} \leftarrow \theta_{val} - \eta \mathbf{g}$

(mini-)Batch Stochastic Gradient Descent: Additional input : mini-batch size k

- initialize n variables θ_{val} with random values, for the n parameters to optimize
- while stopping condition not met
  - sample a (mini-)batch B of k examples (x, y) from X
  - compute $J(B, \theta_{val}) = \sum_{(x,y)\in B} j(x, y, \theta_{val})$
  - $\mathbf{g} \leftarrow \nabla_\theta J(B, \theta_{val})$ (the gradient vector of J with respect to variables θ, at point B, θ_{val})
  - update parameters: $\theta_{val} \leftarrow \theta_{val} - \eta\mathbf{g}$

In practise, in order to take advantage of the full training set, several epochs are used, each epoch processing all of the training examples:

- initialize n variables θ_{val} with random values
- while stopping condition not met (eg. stop after a fixed number of epochs, or when the loss on dev set increases)
  - shuffle the training examples
  - for each (mini-)batch B of k examples (x, y) in X
    - compute $J(B, \theta_{val}) = \sum_{(x,y)\in B} j(x, y, \theta_{val})$
    - $\mathbf{g} \leftarrow \nabla_\theta J(B, \theta_{val})$
    - update parameters: $\theta_{val} \leftarrow \theta_{val} - \eta\mathbf{g}$

Advantages of using (mini-)batches: The mini-batch approach is a tradeoff between the other too: the gradient, hence the update, is meaningful, cf. computed over a batch of examples and not just one example. And the computation is parallelizable, hence more efficient than when the gradient is computed for the loss of the full training set : matrix operations can be used on a (mini-)batch but not on the full dataset. Moreover, gradients over small batches are less precise but lead to more robust convergence (said better to avoid local minima).

10.3 Mathematical reminders: derivatives, partial derivatives, gradients

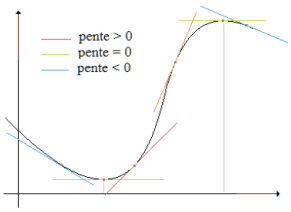
(This section has an accompanying exercise sheet)
Recommended reading: section 4.3 in Bengio et al. 2016.

10.3.1 Derivative of a function with a single variable

Reminder: defined as the limit of the increase of f, when the increase of x tends towards 0

$$f'(x) = lim_{\epsilon\to 0}\frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Hence, graphically, the derivative at a point x is the slope ("pente") of the tangent to the curve of f at point x

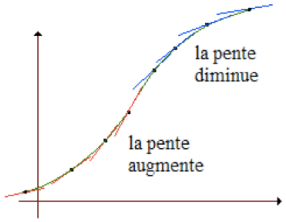


10.3.2 Direction to decrease a univariate function

Given the definition of the derivative, **the derivative of f at a point x allows to compute an approximate value of f at a point close to x**: if a is small then

- $f(x+\epsilon) \approx f(x) + \epsilon f'(x)$
- $f(x+\epsilon) - f(x) \approx \epsilon f'(x)$

Now suppose we look for a modification a of point x (namely going from x to x+a) so that f decreases, i.e. so that $f(x+a) < f(x)$, then **we need to add to x a value a with sign opposite to that of $f'(x)$**. This will be true if a is sufficiently small given the evolution of the derivative itself.



10.3.3 Partial derivatives of a multivariate function

For a multivariate function f (taking n variables $x_1, x_2 ... x_n$), there is no single derivative, but one **partial derivatives** per variable x_i:

- if f is a differentiable function $\mathbb{R}^n \to \mathbb{R}$, then the partial derivative of f with respect to one of its variables x_i is noted $\frac{\partial f}{\partial x_i}$
- it is computed by deriving f considering all other variables as constants
- NB: $\frac{\partial f}{\partial x_i}$ remains a function $\mathbb{R}^n \to \mathbb{R}$, although some of the other variables may disappear when deriving.

Gradient vector: For a differentiable function from $\mathbb{R}^n \to \mathbb{R}$, the n partial derivatives computed at point $a = (a_1, ..., a_n)$ is called the **gradient of f at point a**

- NB: it's a vector in $\mathbb{R}^n$
- it is noted $\nabla f(a)$
- it is possible to derive for part of the variables only : in gradient descent for optimizing parameters θ, we consider partial derivatives of $L(X, \theta)$ wrt θ only, the training examples X are considered as constants. We will sometimes note $\nabla_\theta L(X, \theta)$ in such case.

NB: the term "gradient of f" is ambiguous:

- a set of n mathematical functions, each being one partial derivative function ∇f
- a vector of n real numbers, which are the values of the partial derivatives computed for actual values of the variables a : $\nabla f(a)$

10.3.4 Direction to decrease a multivariate function

It can be shown[5] that for a multivariate function $f(x_1, x_2, ...x_n)$, modify each of the variables $x_1 + \epsilon_1, x_2 + \epsilon_2, ...$ in the opposite direction wrt each partial derivative provides the most rapide decrease : it is the direction of the **steepest descent**.

Using vector notation, given a vector x^i of values for the variables of f, and updated values $x^{i+1} = x^i - \eta \nabla f(x^i)$, if η is small enough, $f(x^{i+1}) < f(x^i)$.

10.4 Convergence of gradient descent?

Now the question is whether decreasing the value of the objective function at each iteration is sufficient to find the argmin.

10.4.1 Critical points, convexity and minima for a univariate function

A critical point of a function f (or stationary point) is a value for which the derivative function f' is null.

Depending on the properties of f,

- there can be no critical point (e.g. a strictly increasing function)
- or f has 1 or several critical points
- a single critical point can correspond to
  - a maximum, whether local or global
  - a minimum, whether local or global
  - or an inflection point with horizontal tangent (f' = 0 and f" = 0)

[5] see e.g. https://www.whitman.edu/mathematics/calculus_online/section14.05.html

∗ cf. inflection point = point for which the second derivative is null f"

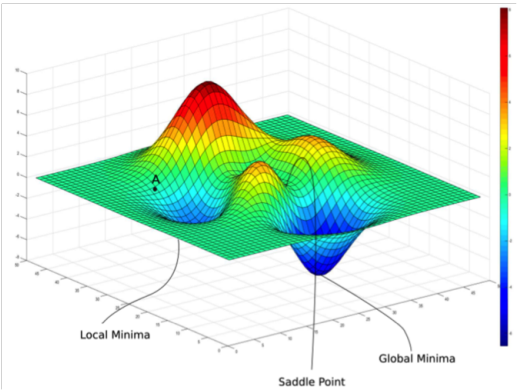
TODO: illustration
If a function if strictly **convex** derivable function (⇔ f' is strictly increasing), there is a unique value x^* nullifying the derivative f', and this value is a global minimum of f. Hence, for a convex derivable function, minimizing f ⇔ solves the equation $f'(x) = 0$. Two general techniques:

- either solve the equation analytically
- or use an iterative algo, cf. **iteratively changing the variables so that each iteration decreases f is guaranteed to converge to the global minimum,** **whatever the starting values are**.
  - with gradient descent applied on a convex objective function J, the update step is guaranteed to have J decrease if the learning rate is small enough (the upper bound on the learning rate is inversely proportional to the maximum of the second derivative).

Moreover, unfortunately, if f is not convex, then iteratively decreasing f will find a local minimum, which is not necessarily the global minimum. **The local minimum found after one training will depend on the initial values. Given that these are initialized at random, several runs may end up with distinct solutions for the parameters, leading to potentially varying classification performance.**.

TODO illustration

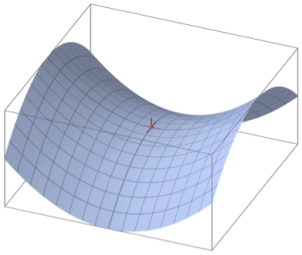
10.4.2 For a multivariate function



(source: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/)

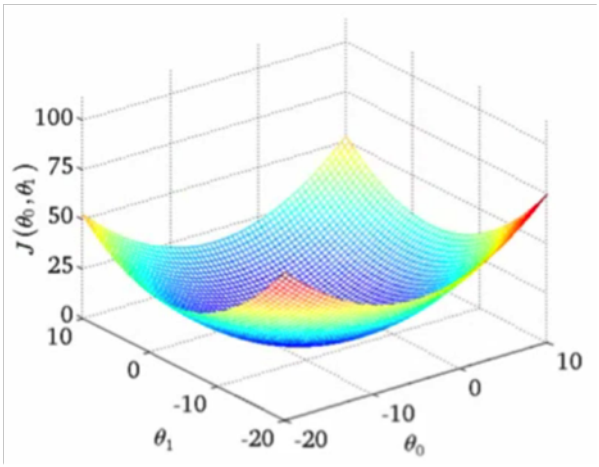
In the multivariate case, a critical point is a point for which all partial derivatives are null (gradient is the null vector). Such a point may correspond to

- as for the univariate case, a minimum or maximum of f, whether local or global
- but also to a **saddle point** ("point selle" ou "point col"), namely for which the partial derivatives are null, but do not evolve in the same way (some are growing / some are decreasing). Hence the saddle point is a local minimum/maximum on one of the dimensions, but neither a maximum nor a minimum overall. Cf. illustration in the d=2 case below.



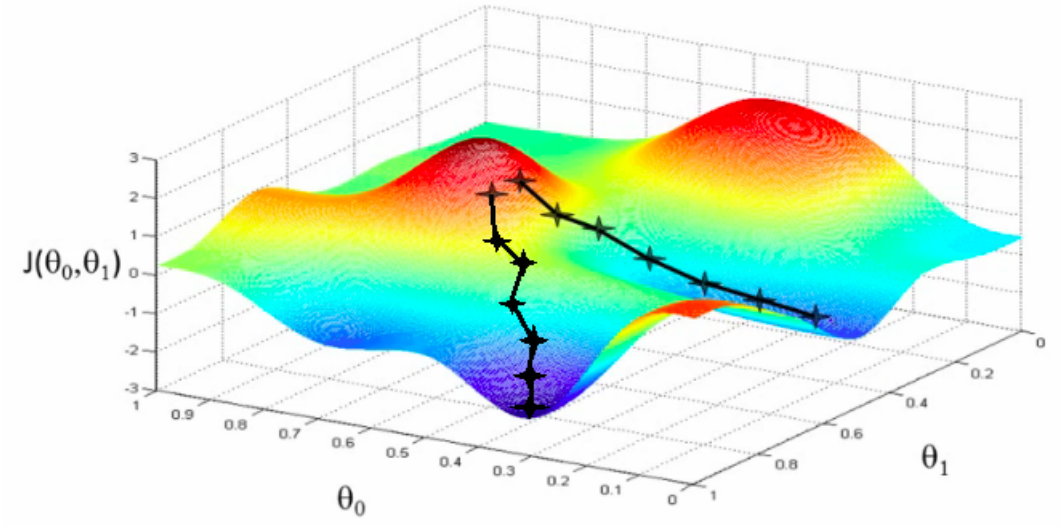
(source: saddle point, Wikipedia)

As for the multivariate case, a **convex** function has a unique global minimum (convexity definition is more complex in the multivariate case, not detailed here).



(source: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/)

For a **non-convex** function, there may be several local minima, and the result of gradient descent can vary depending on the initialization values of the parameters!!. Cf. illustration below, where two starting points (initialization values) lead to different local minima.



(source: https://julienharbulot.com/gradient-descent.html)

10.4.3 Convergence of gradient descent in the case of a MLP?

Hinge loss on a linear classifier is convex, hence any initial values should lead to the global minimum.

This is also the case for logistic regression with NLL loss.

Yet for a MLP with NLL loss, the overall loss function is not convex: there are several local minima. One simple justification for this is that neurons of the hidden layer could be shuffled, resulting in several values of the parameters leading to the exact same loss value.
Yet, there are empirical results showing that (from Choromanka et al. 2015 https://arxiv.org/pdf/1412.0233v3.pdf) (more precisely, proved for a MLP with ReLU activation):

- "For large-size networks, most local minima are equivalent and yield similar performance on a test set"
- "The probability of finding a "bad" (high value) local minimum is non-zero for small-size networks and decreases quickly with network size"

Yet, empirically, a substantial variance in performance can be observed: several training runs achieve varied performance. This can result from unappropriate learning rates, unappropriate input features to represent the objects to classify, or the number of parameters to learn being too high with respect to the number of examples.

For these reasons, in scientific experiments, it is custom to launch several trainings, and report the performance variance. Small variance indicates that the training set is appropriate for the number of parameters.

10.5 Connection with the perceptron

see Exercice sheet

10.6 Efficient gradient computation: backpropagation

see Exercice sheet

10.7 Variants of SGD

(not treated in detail here, see for instance this blog http://sebastianruder.com/optimizing-gradient-descent/index.html

- "momentum" : update of the parameters using a history of the gradients on several iterations
- variation of the learning rate during learning
  - SCHEDULING : variation as a function of the iteration or the epoch
  - e.g. $\eta_i = \eta_0/e$
  - ADAPTIVE LEARNING RATES : method to set distinct learning rates for each parameter, changing over time
  - famous algorithms: ADAGRAD, ADAM
  - $\longrightarrow$ in general work better, but longer training

10.8 Dropout

Dropout is the classic current technique to combat overfitting in MLPs and other neural networks (on top of early stopping): it consists in modifying the learning procedure by randomly setting to 0 the output of certain neurons (in certain layers or all the network). This is applied at learning but not at testing time.

As mentioned in section 7.5, dropout:

- will enforce that a given neuron cannot rely on the use of other neurons,
- dropout is thus said to prevent complex co-adaptations of neurons (neurons firing only in the context of other ones)
- and experimentally proves very efficient at preventing overfitting

10.9 Wrapping up example

Take the problem of sentiment analysis, in which a verbatim is to be categorized into positive, negative or neutral. With a bag-of-word vector representation of the input, a one-hidden layer MLP with ReLU and NLL loss, give the mathematical expression for the loss function (use intermediary results), and give the overall procedure to learn the parameters of the model.

11 Vectorial representations of the objects to classify: sparse versus dense features

All the work done so far on the automatic classification task assumes that an object to be classified is represented by a vector. We talk about "feature" vectors: each component of the vector corresponds to a characteristic of the objects to be classified. For example, for an email, a feature can be the language used, the number of recipients...
The aim is to transform the characteristics of the objects into numerical values.

Ambiguity in the terminology:

- **feature** = a component of the vector representing the object
  - (= a numerical value)
  - (for Goldberg = "input vector entry")

  - =¿ in the following I will use "component".
- **feature** = a characteristic of the objects to classify, possibly coded on several components
  - (Goldberg = "feature")
  - Example: "the number of recipients of a mail". coded on one component, or several if bins are used
  - Example: for tagging: "the word preceding the word to be tagged" $\longrightarrow$ coded on $|V|$ components, with V = the vocabulary
  - =¿ I will use this meaning for "feature"

There has been a lot of changes in the way features are coded: between linear classifiers vs. deep networks, we went from "sparse" representations (lots of zeros) to dense representations.

11.1 Sparse features in linear classifiers

To illustrate all the following, we will take 2 classical examples:

- Classification of a "document" (or portion of text) into a class
  - sentiment analysis (e.g. movie review, positive/negative)
  - mail classified as spam / not spam
  - The document can contain meta-data or not (for the mail: sender, recipients etc...)
- tagging of a word in a sentence, for example
  - morpho-syntactic category
  - semantic category (human, object, state, event ...)

Types of features:

- **Boolean feature**: 0 / 1 or more -1 / 1
  - e.g. "does the mail have an attachment".
  - codable on a specific component, with values 0 / 1, or values -1 / 1
- **Numerical trait**: possible values: integer or real values, in a range [X;Y]
  - can be coded on a single component
  - the value can be numerical value itself, or the value after a change of scale (e.g. logarithmic scale)
  - or can be "binned":
    - we consider n ranges of values
    - one-hot encoding on n components with 0/1 values (see below "one-hot" encoding)
    - example : for the number of recipients of an e-mail: 4 ranges (4 "bins") between 1 and 2 / between 3 and 10 / between 11 and 100 / more than 100 recipients
    - for a mail with 4 recipients =¿ code 0 1 0 0
    - for an email with 110 recipients =¿ code 0 0 0 1
- Categorical trait ("categorical"): a symbol among a vocabulary of symbols
  - We talk about "nominal" values (= a name as opposed to a number), for example
    - spam / non-spam classification: the country of origin of the mail among a list of countries
    - tagging : the word preceding the word to be tagged, among a Vf vocabulary of inflected forms
    - semantic tagging : the morpho-syntactic category of the word to be tagged, among a set of categories Vt
    - semantic tagging: the lemma of the word to be tagged, among a vocabulary Vl of lemmas
    - etc...
  - $\longrightarrow$ **one-hot encoding**: if V is the vocabulary of possible values, of size $|V|$
    - $|V|$ binary features = we have one component per possible value
    - e.g. by possible country
    - it is like making $|V|$ boolean features "country=XXX" = 0 or 1
    - rem: the mutual exclusion is represented by the fact that we have a single non-null component
    - hence the term "one-hot": only one component is active

11.1.1 Example for part-of-speech tagging: representation of the word to be tagged

Suppose we are tagging a sentence $w_1 w_2 ... w_n$, from left to right, in a **greedy** fashion: when we process a position i, we choose a tag, without trying to optimize the tag choices over the whole sentence.

Suppose we have already tagged the words $w_1 ... w_{i-1}$. So for the features, we have access to :

- the words of the whole sentence, in particular those in a "window" around w_i
- the tags already predicted for the positions $w_1 ... w_{i-1}$

To tag w_i, we can for example use the following features: (cf. Ratnaparkhi 1996 (maxent model = logistic regression))

- the current word (w_i)
  - $\longrightarrow$ one-hot encoding, on $|Vf|$ boolean components
  - by noting Vf the vocabulary of all the inflected forms of the training set (modulo treatment of unknown words, see below)
- the next word (w_{i+1}) $\longrightarrow$ idem but for the next word
- the previous word (w_{i-1}) $\longrightarrow$ same but for the previous word
- the suffix of size 2 of w_i $\longrightarrow$ one-hot encoding on $|S2|$ boolean components, with S2 the set of suffixes of 2 letters of the words of the training corpus
- the suffix of size 3 of w_i $\longrightarrow$ idem but with the suffixes of 3 letters
- the tag t_{i-1} $\longrightarrow$ $|Vt|$ boolean components, with Vt the set of tags
- the tag pair $t_{i-2} t_{i-1}$ $\longrightarrow$ 1 boolean component per tag pair
- the word starts with a capital letter? $\longrightarrow$ 1 boolean component

With such features, the size of the vector representing a word to be tagged is thus: $D = 3|Vf| + |S2| + |S3| + |Vt| + |Vt| * |Vt| + 1$
All components are boolean (0 or 1), and for a given w_i within a sentence, most components are zero: the vector is sparse.
Example:
The little cat plays with the ball .
DET ADJ N ?
The only non-zero components of the vector representing "plays" are:

- w_i=play?
- w_{i-1}=cat?
- w_{i+1}=with?
- s2=ys?
- s3=ays?
- t_{i-1}=N?
- $t_{i-2} t_{i-1}$= ADJ_N?

11.1.2 Coping with "unknown" symbols

The different vocabularies mentioned above (Vf, S2, S3, Vt) must be fixed at learning time, because they condition the size of the input vector (and thus the size of some parameter matrices). So these vocabularies cannot vary when tagging words in new sentences.

- $\longrightarrow$ hence the different vocabularies are defined according to what is observed in the training data
- $\longrightarrow$ but when using the tagger on new sentences, one may encounter "unknown words" and possibly unknown suffixes. "Unknown" meaning "not present in the training data". The problem is obvious in the case of lexical features (inflected forms and lemmas), a little less so for suffixes (using suffixes is precisely for that).

Many techniques have been proposed to handle unknown symbols. Note that using suffix features is one of them. A classic simple technique is:

- Before learning, replace some words in the train with a dummy word "UNK" (which will be included in Vf)
  - it is better to replace by UNK rare words, cf. it is the rare words which resemble most the unknown words

  - in particular, an unknown word is much more likely to be a noun than a determinant for example.
  - we can therefore make a random replacement of some of the hapaxes of the train (= the words of the train having only one occurrence)
  - we fix a hyperparameter α, for the probability for a hapax in the training set to be replaced by UNK. We go through the training set, randomly draw a number r between 0 and 1, and if $r \leq \alpha$, we replace the word by UNK

- learn the classifier $\longrightarrow$ the parameters (weights) concerning UNK will be learned as for all other words
- in the test phase, replace all the words in the test that are not in Vt with "UNK"

11.1.3 Document representation (word sequence)

already seen in lab sessions:

- BOW representation: sum of the one-hot of each word
- possible normalization of the numbers of occurrences (divide by sum of occurrences, or max nb of occurrences)
- TF.IDF weighting is often beneficial (cf. lab session)

We can combine the BOW model with other features, like the number of recipients of the mail etc...

11.2 Dense features in neural networks: "embeddings"

We will start by considering dense vectors for words, and then we will see how actually any type of vocabulary can be represented as dense vectors.

Representing each word as a one-hot vector does not capture any form of similarity between words. Indeed, let's imagine one-hot vectors for the following words:

pear	1 0 0 0 0 0 0 0 0 0 0 0 ...
apple	0 1 0 0 0 0 0 0 0 0 0 0
car	0 0 1 0 0 0 0 0 0 0 0 0
dazzle	0 0 0 1 0 0 0 0 0 0 0
blind	0 0 0 0 0 1 0 0 0 0 0 0 0

The scalar product, or cosine between any pair of one-hots will be zero. So for example, suppose we have a training example $(x^{(1)}, y^{(1)})$ where the input $x^{(1)}$ contains the one-hot of "pear". Suppose we have to classify a new entry, $x^{(2)}$, similar to $x^{(1)}$ but where we replace "pear" by "apple", and a new entry $x^{(3)}$, where we replace "pear" by "car" .

If the words are coded as one-hots, $x^{(2)}$ will be as different from $x^{(1)}$ as $x^{(3)}$. If on the contrary we represent "pear" by a vector close to that of "apple", then we will obtain that the input $x^{(2)}$ is close to the input $x^{(1)}$. **This allows a generalization effect of the training set: although $x^{(2)}$ was not seen at training, being close to $x^{(1)}$, we can hope that having $(x^{(1)}, y^{(1)})$ in the training set will help to classify $x^{(2)}$.**

There is a long tradition in NLP of representing words as vectors, computed on the basis of the **distributional hypothesis**: words with close meanings have close distributions (in the sense of the set of their contexts) ($\longrightarrow$ see section 12). With word vectors, called "distributional vectors", we can capture a form of similarity between words via a simple vector similarity, in particular a cosine between vectors. Hence the idea of using for lexical features not one-hot vectors, but these distributional vectors. Because such distributional vectors are integrated into neural networks, the term **word embeddings** has emerged. The term "distributed representations" is also used, to indicate that the representation of a word is distributed along the various components of a vector.

Integration of embeddings in a neural network Embeddings are typically used in a neural network, with 2 distinct modes:

1. vectors learned upstream, and used in a **frozen** way in a network defined for another task
2. or vectors being part of the **parameters** to be learned
   - 2.a either randomly initialized (like the other parameters)
   - 2.b or initialized with vectors that were previously learnt, on some other task (like in [1])

Let's review these 3 cases.

11.2.1 Integration of frozen embeddings

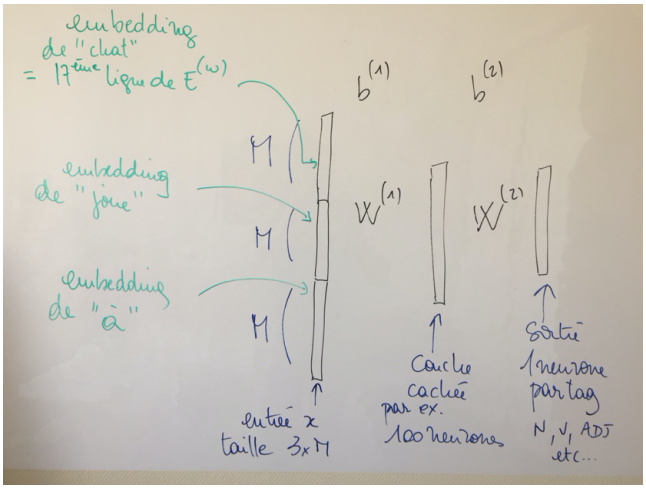
Let us suppose that we already have word vectors: a vector of size M for each word of a vocabulary V.

- $\longrightarrow$ these vectors form a matrix $E^{(w)} \in \mathbb{R}^{|V| \times M}$
- $\longrightarrow$ any lexical feature can be replaced by this vector

Example: tagging

- Take the features defined for tagging previously, and replace the one-hot features involving inflected forms by dense vectors.
- Question: What will be the size of the vector representing a word to be tagged?
- Let's simplify: suppose we only keep the features "previous word", "current word", "next word" $\longrightarrow$ the input of the network, for a word to be tagged, is the concatenation of 3 word embeddings

Example, for the tagging of the word "joue" in "le petit chat joue à la balle":



Example : representation of a document (or a portion of text).
We can switch from a BOW representation, to a CBOW representation (for "**continuous bag of words**")

- instead of a BOW representation
- we can use the sum of the dense vectors of words appearing in the document
  - as for a BOW, we can weight by the number of occurrences of the words, normalized or not
  - as for a BOW, taking the sum of the dense vectors totally ignores the linear order of the words
  - the size of the CBOW vector is the size of the word vectors, typically a lot less than the size of the vocabulary

11.2.2 Integration of embeddings as network parameters

But the strength of embeddings is that **they can themselves constitute parameters of a network**, and that they can be learned, or "tuned" (see below) to optimize the task targeted by the network: one hopes that the learned vectors capture the similarities that are relevant for the task. For example, the word vectors useful for tagging are not necessarily the same as for semantic labeling.

- $\longrightarrow$ the embedding matrix can be integrated as parameters of the network, as a first layer (we talk about an "**embedding layer**").
- $\longrightarrow$ this means that the $E^{(w)}$ matrix is considered as parameters of the loss function. The gradient of the loss function will include partial derivatives for each of the parameters in $E^{(w)}$, which will be updated upon learning.

Example in part-of-speech tagging: Let us take the tagging task in a set of T=20 categories, using as features the concatenation "previous word, current word, next word".
This time we distinguish the notation t_i = the i-th token of a sentence, versus w_j the jth word of the vocabulary, $1 \leq j \leq |V|$.
We can use for example a network defined with:

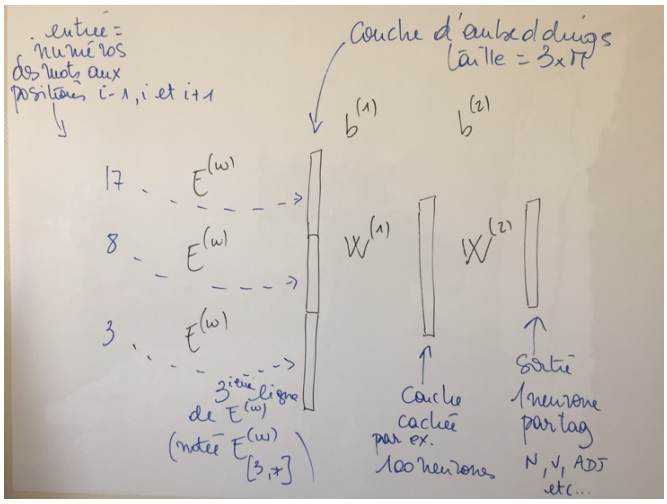
- an embedding layer for these 3 words, with parameters $E^{(w)} \in \mathbb{R}^{|V| \times M}$
- the jth row of $E^{(w)}$ corresponds to the embedding (=the dense vector) of the word w_j of the vocabulary

  - so for a given input word to tag t_i, we have 3 word vectors, for the tokens t_{i-1}, t_i, t_{i+1}
  - the most common ways of combining several embeddings are:
    * either to sum the embeddings
    * or to concatenate them (here this would lead to a vector of size 3*M)
    * in the following, we suppose we concatenate the 3 embeddings. Indeed, for the tagging task, we want to distinguish the information relative to the 3 positions (previous, current, next), which would be lost with a sum

- the reminder of the network, after the embedding layer, is a MLP with 1 hidden layer of size 100, with parameters $W^{(1)} \in \mathbb{R}^{(3*M)\times 100}, b^{(1)} \in \mathbb{R}^{100}, W^{(2)} \in \mathbb{R}^{100\times 20}$ and $b^{(2)} \in \mathbb{R}^{20}$.

If p, c, s are the word indices of the tokens t_{i-1}, t_i, t_{i+1} respectively then, the embedding layer concatenates the rows p, c and s of the matrix $E^{(w)}$.
For example: *The little cat plays with the ball.*

- suppose these tokens correspond to the words w2(the) w30(little) w17(cat) w8(plays) w3(with) ...
- and that we are to tag the toke "plays"



NB: The integration of the embedding layer implies that the resulting network is not fully-connected network, and the embedding matrix is shared accross the 3 positions.

Rem: mathematically, "take the jth row of $E^{(w)}$" can be written by means of a scalar product with the one-hot vector j:

- let's note ohw_j is the one-hot vector of the word w_j
- then **the scalar product $ohw_j.E^{(w)}$ is the j-th row of $E^{(w)}$**
- note that this is convenient in mathematical formula, but this is not used in actual implementation, in which one rather "take the j-th row" of the matrix.

Forward propagation of the network i.e. here calculation of the score of the 20 possible tags for the current token t_i:

- the input of the network contains the 3 indices of the words for the tokens t_{i-1}, t_i and t_{i+1} (in our example, the indices 17, 8 and 3)
- passing these to the embedding layer = simply means concatenating rows 17, 3 and 8 of $E^{(w)}$: $e = [E^{(w)}_{[p,*]}; E^{(w)}_{[c,*]}; E^{(w)}_{[s,*]}]$
  - (there is no activation function, the input and output of the embedding layer is the same)
  - rem: the concatenation is sometimes noted $\oplus$
- then we can apply the MLP as usual: the score vector of each category is o = MLP(e)

Learning of the parameters The matrix $E^{(w)}$ is a parameter matrix, updated during the gradient descent. The gradient is quite simple:

- the partial derivatives $\frac{\partial e}{\partial E^{(w)}_{(j,k)}}$ are zero for all indices j other than c, p or s
- For the 3 indices, c, p and s, $\forall$ k from 1 to M $\frac{\partial e}{\partial E^{(w)}_{(p,k)}} = 1$, $\frac{\partial e}{\partial E^{(w)}_{(c,k)}} = 1$, $\frac{\partial e}{\partial E^{(w)}_{(s,k)}} = 1$.

At the end of the training, **we hope that the word embeddings are optimized for the tagging task**.

11.2.3 Initialization of word embeddings

2 modes:

- Either $E^{(w)}$ is treated as any parameter matrix, with random initialization. But be careful, for a vocabulary of e.g. 20 000 words (which is quite small), and for embeddings of size e.g. 300, $\longrightarrow$ we get 6 million parameters
  - $\longrightarrow$ if the order of magnitude for training data is 100 000 or 1 million tokens
  - $\longrightarrow$ may be insufficient to correctly learn so many parameters
  - rem: one sign of insufficient data is a variance in performance between several training runs: if the performance varies from one run to another, it shows that the random initial values did not converge well
- or we retrieve a matrix of so-called **pretrained embeddings**, i.e. already computed on a large raw corpus (see section 12 for a famous algorithm to learn word vectors: word2vec): there are many algorithms learning word vectors simply by using a raw corpus, which can be very large since there is no annotation cost $\longrightarrow$ one can pre-train vectors on corpora of hundreds of millions or even billions of words, and then use them as initial values for other more sophisticated tasks
  - $\longrightarrow$ pre-trained embeddings form already consistent initial values (as opposed to random values). When learning the tagger, they will be slightly modified, so as (hopefully) to specifically serve the tagging task. This is called **fine-tuning** of parameters (here fine-tuning of the embeddings)
  - in practice, this version is technically very close to the use of "frozen" lexical embeddings: to fine-tune the embeddings or on the contrary to freeze them, it is sufficient to use a boolean controlling if these parameters are to be updated or not during the learning process (and thus if it is appropriate or not to compute the partial derivatives with respect to these parameters).

11.2.4 Special embeddings: unknown word, beginning/end of sentence

As seen previously, we must be able to manage the case of a word to be tagged, t_i, where one of the words taken into account to represent the context (in the previous example: t_{i-1}, t_i, t_{i+1}) is absent from the learning corpus.

- $\longrightarrow$ We can reuse the replacement of some hapaxes in the training set by UNK
- $\longrightarrow$ UNK is then part of the vocabulary, corresponds to a row of $E^{(w)}$, and a vector for UNK is learnt, just as for the real words.

It is also necessary to integrate to the vocabulary (and thus to $E^{(w)}$) a false word for the "word preceding the first word of the sentence" (t_{-1}) and another for the "word following the last word of the sentence".

NB: if we initialize the lexical embedding matrix with pre-trained embeddings, introducing new, non-pre-trained vectors for special tokens can be problematic, as these new vectors will be initialized randomly, and potentially "far" from the pre-trained vectors. A solution sometimes used is to initialize with averages of pre-trained vectors.

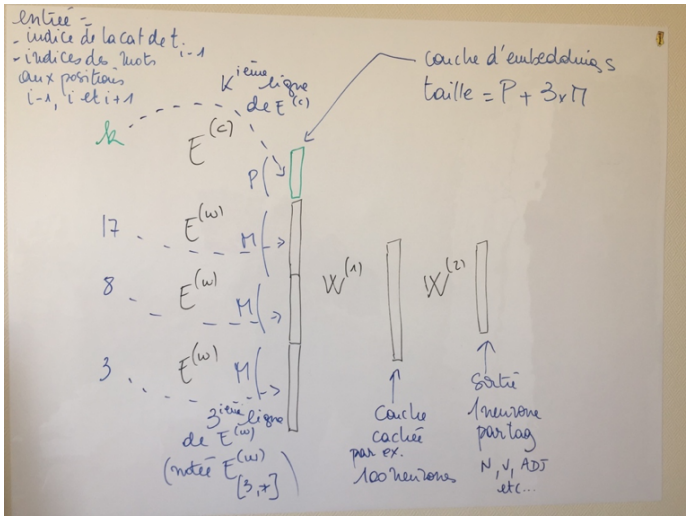
11.2.5 Generalization to any type of symbolic feature

In the same way as for words, representing any set of symbols as embeddings instead of one-hot vectors can be interesting for capturing similarities between these symbols. For example, within the vocabulary of morpho-syntactic categories, it can be interesting to capture that the category "past participle" shares behaviors with adjectives and with conjugated verbs, but not with prepositions.

$\longrightarrow$ to do this, we can introduce as parameters of the network a matrix of embeddings for each feature having as domain a vocabulary of symbols: we can use lemma embeddings, morpho-syntactic category embeddings, grammatical function embeddings etc....

For example: for tagging a token t_i, we can add as input feature the category predicted to the previous token, represented by a vector of size P, which we can call a "category embedding". If we have a vocabulary of 20 categories (20 tags) numbered from 1 to 20, we define $E^{(c)}$ with 20 rows and P columns, a matrix where the j-th row represents the embedding of the j-th category.

Remark: as there are much less categories than words, P is rather of the order of 20 or 50, versus a few hundred for word embeddings.



Learning: For symbols other than inflected forms there are usually no pre-trained embeddings. So these embeddings are randomly initialized and learned for a specific task.

11.3 Check your understanding

- What is the point of using word embeddings?
- What does it mean to use "frozen" word embeddings?
- What are the 2 modes for initializing word embeddings in a MLP?
- What technique was mentioned above to obtain vectors representing morpho-syntactic categories (be precise)?

12 How to get word embeddings

In the previous section, we have seen the notion of word embedding, their utility, and how word embeddings can be integrated in a neural network: either as frozen vectors, being part of the input of the network, or as parameters of the network.

This section is now focused on the methods to obtain lexical embeddings (we will simply use the term "embeddings" in the following).
We have seen that we can integrate the embeddings as a matrix of parameters of a neural network, a network defined for example for a tagging task. The embeddings after training the tagger are thus adapted for the tagging task.

We focus now on learning generic embeddings, independently of a specific task like part-of-speech tagging. We aim at learning a **vector space of words, which reflects as much as possible the semantic similarity or proximity of words**.

12.1 The distributional hypothesis

The central assumption for obtaining word embeddings is that semantically close words will have similar distributions.

- light version:
  - Words with "similar" meaning will tend to appear in the same contexts.
  - So conversely, words with similar contexts are probably similar words, and therefore should have close vectors
- Stronger version: it is the distribution of a word that defines the meaning of that word.
- cf. the famous quote by Firth (1957): "You shall know a word by the company it keeps".

To illustrate this, we note that we can get an idea of the meaning of a word that we do not know, by studying the contexts in which it appears. **?** quotes the simple example (borrowed from Nida (75)):

- *A bottle of tezguino is on the table*

- *Everybody likes tezguino*
- *Tezguino makes you drunk*
- *We make tezguino out of corn*

The basic principle for constructing a vector space of words $\mathcal{E}$ is therefore to ensure that the more two words have similar contexts observed in corpora, the closer they should be in $\mathcal{E}$.

We thus speak of "distributional (vector) space models of words" (DSM).

12.1.1 Types of distributional similarity

Depending on the way the word contexts are defined, different types of similarity can be obtained. Schematically we distinguish:

- **"topical" similarity** : similarity of the domain (for example : lawyer / trial / court / judge / etc...
  - $\Longrightarrow$ obtained rather by considering a large context around a word (for example 10 words before and 10 words after)
- **"functional" similarity**: similarity of words that can be exchanged one for another (synonyms, but also antonyms ...)
  - $\Longrightarrow$ obtained rather with context tightened around the word

NB: we are talking here about vectors associated with inflected forms:

- which are easy to obtain from raw corpora (simply tokenized corpora)
- it is rather simple to build vectors of lemmas: it is enough to lemmatize the learning corpus
- on the other hand, it is much more complicated to obtain meaning vectors (i.e. for example a vector for "lawyer"-profession and a vector for "lawyer"-fruit)
  - in the latter case, we speak of **sense embeddings**

Note that a major advance in recent years (since 2018) has been the use of vectors representing a word in context, i.e. an occurrence of a word, and not a word in absolute terms. These vectors are mainly obtained thanks to pre-trained language models, based on biLSTM recurrent networks (e.g. **?**), or since 2018, based on transformers blocks (cf. the famous BERT model **?** $\Longrightarrow$ see M2 course).

12.2 "Old school" technique: counting co-occurrences

Historically, the idea of representing words as vectors (i.e. distributed representations on several dimensions) is quite old (article by Hinton et al., 86 often quoted, see (**?**)). The distributional approach, constructing vectors representing the meaning of words, dates from the 90's, with in particular (**?**).

12.2.1 Construction of sparse distributional vectors

Method (naively described, without computational efficiency):

- We have a corpus C, generally segmented into sentences and tokenized.
- We browse the corpus in order to retrieve the number of occurrences of each pair (word, context)
- The different contexts encountered for each word form the dimensions of the final vector space (one component corresponds to one context) (see below for the contexts typically used)
- So a word w is represented by a sparse vector, let us call it $v(w)$, in which:
  - a component corresponding to a context not encountered for w in C is null
  - a component corresponding to a context c encountered x times for w will have for value either the nb of occurrences x, or a certain weight function for the couple (w,c) see infra)

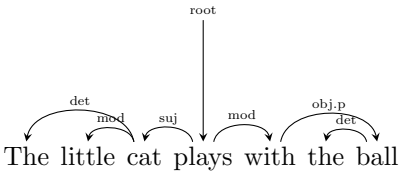
As the set of contexts is huge, the obtained vectors are very sparse. We give below briefly the types of context classically used, and the most frequently used weight function: the PPMI.

Linear contexts A context c for a word w is usually modeled as a pair (r, w'), where w' is a word in the context of w, and r is a "relationship" between w and w': it describes how w and w' are located relative to each other.
Relationships are usually "linear": the term linear has nothing to do with the mathematical term used for classification or linear regression, i.e. they only consider the order of the words. For example, in the sentence ***The little cat plays with the ball***, for the occurrence of **w=*plays***, we can consider the words at a certain distance to the left and to the right of w. This is called a **window** around w. For example a window of size 2 is usually 2 words on the left and 2 words on the right (we rarely say the total size). We can then use more or less precise relations:

- case where one considers only the membership to the window around w: relation **r=**"appears at a distance < 3" : one would extract the contexts
  - $(r, cat), (r, small), (r, with), (r, la)$
- case where we distinguish r_1 ="appears on the left at a distance < 3", and r_2="appears on the right at a distance < 3": we would extract the contexts
  - $(r_1, cat), (r_1, small), (r_2, with), (r_2, la)$
- case where we specify in the relation the precise distance:
  - r_{-2} = "appears on the left at a distance=2", r_1 = "appears on the right at a distance =1" etc.
  - we would extract $(r_{-1}, cat), (r_{-2}, small), (r_1, with), (r_2, la)$ (and so on depending on the size of the window)

The size of the window and the types of relationship are hyperparameters to be defined upstream.
The same context will be seen for occurrences of different words. For example the context $(r_{-1}, small)$ will be incremented by 1 each time a word is preceded by "small".

Syntactic Contexts To target more informative contexts than just appearing in a window, one can use more sophisticated relationships, in particular syntactic relationships, in the form of paths in a dependency tree. The simplest way is to first consider paths of length 1: for a word w, we consider its syntactic dependents, and its governor.
For our example above, let's assume that the dependency tree is:



We can extract for the occurrence *plays* the contexts *(has-subject, cat), (has-mod, with)*. For the occurrence *cat* one could extract *(is-subject-of, play), (has-det, the), (has-mod, small)*.
The advantage over linear contexts is that words in direct syntactic relation with w are a priori very informative about the meaning of w, but can be linearly distant. For example for the sentence *The little cat on the 3rd floor plays with the ball*, a window of 2 does not reach the subject *cat*. For example **?** show that vectors obtained by combining linear contexts and syntactic contexts are of better quality than those obtained with each of these types of contexts separately (see below for evaluation techniques).

Vocabulary and context filtering Computational difficulty: Corpus sizes range from a few million to over a billion tokens. Thus, the size of the vocabulary (= the number of distinct words) can grow to tens of thousands (proper nouns tend to increase the vocabulary), and depending on the relations used, the number of distinct contexts is even larger. It is therefore necessary to set up filtering (in general, only keep the words that appear more than a certain number of times, and the same for the contexts.

Depending on the type of vectors targeted, we can ignore the tool words to keep only the semantically full words. For example, for the previous example, we would carry out the context extraction on a filtered sentence: little cat plays with a ball, we thus reach with a window of 2 the full word all not reached previously.
This type of filtering is rather adapted to obtain a topic similarity rather than a functional one, cf. the tool words are essential to capture the grammatical properties of words.

12.2.2 Context weighting

Some contexts will be uninformative because they appear for many distinct words. Others will be very specific to a word. The raw number of occurrences of a context is not sufficiently informative: if a word w is seen twice with the context $(r_{-1}, grievement)$ (i.e. if w is preceded 10 times by "grièvement"), and 20 times by "très", 10 and 20 must be compared by taking into account the frequencies of "grièvement" and "très", the latter being probably much more frequent.
Hence the idea of not keeping the raw numbers of occurrences of the contexts, but of weighting them.
A classic technique is to use pointwise mutual information (PMI), used since the 90's (Hindle, 90) to measure association scores between words, in order to identify collocations: pairs of words which appear together more frequently than expected. In the context weighting of a word, we consider not a pair of words, but a word/context pair (w, c), where the context c is itself a pair (r, w').
But let's take the general case of the PMI calculation: we consider a probabilized space, we have two events A and B which can be realized together or each separately (we will use this model to consider 2 linguistic phenomena which can appear together in a sentence, or appear separately). The PMI is the ratio between the probability of joint realization of the 2 events $P(AB)$, and their probability of joint realization calculated as if these events were independent $P(A)P(B)$:

$$PMI(A, B) = log_2 \frac{P(AB)}{P(A)P(B)} = log_2 \frac{P(A|B)}{P(A)} = log_2 \frac{P(B|A)}{P(B)}$$

If the numerator is greater than the denominator, it means that the realization of A (resp. B) makes the realization of B (resp. A) more probable: the more the ratio is greater than 1, the more the PMI will be positive (cf. log), the more we can interpret that A and B are dependent, i.e. linked, i.e. it is not a coincidence that they are realized at the same time.
The opposite case where the PMI is negative is more difficult to interpret: the realization of A makes the realization of B less likely (and vice versa). In practice, these cases occur in particular when AB is rare, and the estimation of P(AB) is not very reliable. For this reason, we generally consider the positive pointwise mutual information (PPMI), with $PPMI(A, B) = max(0, PMI(A, B))$.
In the case of distributional similarity, we can use PPMI to represent the link between a word w and a context c: the higher the PMI, the more the word and the context are linked, the more we can say that their co-occurrence is not a coincidence, and thus that the context c will be particularly representative of the word w. Hence the idea to use the PMI(w,c) value in the sparse vector representation of w, for the component corresponding to the context c.
To compute the PMI in this case, we consider events of type "the word w appears in the context c", and more general events "the word w appears (no matter the context)" and "the context c appears (no matter the word w with which c appears). . We can consider a random variable W for the word, and a v.a. for the context C. The event "the word w appears in the context c" corresponds to the conjunction of W=w and C=c. The PMI can be written:

$$PMI(W = w, C = c) = log_2 \frac{P(W = w, C = c)}{P(W = w)P(C = c)}$$

We can estimate the 3 probabilities by relative frequency, on an observed set of word / context pairs. If we note occ(w,c) the number of occurrences of the pair (w,c), and we use $*$ for "sum over all values":

$$P(W = w, C = c) = \frac{occ(w, c)}{occ(*, *)}$$

$$P(W = w) = \frac{occ(w, *)}{occ(*, *)}$$

$$P(C = c) = \frac{occ(*, c)}{occ(*, *)}$$

Hence

$$PPMI(W = w, C = c) = max(0, log_2 \frac{occ(w, c)occ(*, *)}{occ(w, *)occ(*, c)})$$

We can further refine by using the fact that the context c is structured: it is a couple (relation, word w'). **?** proposes to use a ratio between $P(w, r, w')$ and the realization as if w and w' were independent knowing r: applying the multiplication rule to $P(w, r, w')$ gives $P(w, r, w') = P(r)P(w|r)P(w'|r, w)$, which is, if w and w' are independent knowing r: $P(r)P(w|r)P(w'|r)$. Hence the association score in Lin (called I(w,r,w') for "quantity of information given by the fact that (w,r,w') appear x times"):

$$I(w, r, w') = log_2 P(w, r, w')P(r)P(w|r)P(w'|r)$$

To summarize, we can obtain the sparse vectors by

- extracting couples (word, context) and their number of occurrences (linear contexts, or more sophisticated)
  - in general a context is a couple (relation, other word) (r, w').
- contexts form the dimensions of a vector space
  - the component corresponding to (r, w'), for word vector $v(w)$, can be for example $I(w, r, w')$, or $PPMI(w, rw')$

12.2.3 Dimensionality reduction (not treated here)

The sparse vectors obtained by the previous method can be difficult to handle because they are too large. Techniques in linear algebra exist to "reduce the dimensionality" of a vector space. The underlying intuition is to remove unnecessary dimensions, and combine dimensions to obtain a smaller number, which are less redundant.

The techniques used are for example the PCA (Principal Component Analysis), which searches for the dimensions with the most variance, or the Singular Value Decomposition (SVD).

12.3 Approach via the word prediction task: detour by language models

The advent of word embeddings within neural networks has transformed the way such vectors can be computed. The original idea stands in the use of word embeddings as parameters of a network designed for language modeling.

We will first quickly remind what language models are, and what problems arise with the famous n-gram language model (Markov model), before moving to neural language models.

12.3.1 Problems with n-gram language models

A **language model** [6] is a probabilistic model which associates a probability to a sequence of words $w_1, ..., w_n$ (which we will note here $w_{1:n}$). Language models are used in particular in speech recognition, and also in machine translation before the arrival of neural translation models. For example, for two possible translations of a sentence, we choose the most probable one.

In general, the event 'the sentence is $w_{1:n}$' is decomposed into the conjunction of 'the 1st word of the sentence is w_1' AND 'the 2nd word is w_2' AND etc.

Then, we apply the chain rule of probability[7], in general from left to right (any order is possible):

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{1:3})...P(w_n|w_{1:n-1})$$

Reminder: Markov hypothesis The problem of this decomposition is the impossibility to reliably estimate $P(w_i|w_{1:i-1})$ as soon as i exceeds 3 or 4, cf. to estimate by relative frequency, for example

$$P(ball|the\ little\ cat\ plays\ with\ it)$$

, one counts $\frac{occ(the\ little\ cat\ plays\ with\ the\ ball)}{occ(the\ little\ cat\ plays\ with\ *)}$, and the counts are too low.

Hence we make an approximation, considering that the occurrence of a word depends only on the k words which precede it (k-order Markov hypothesis). Of course this is a totally false hypothesis here, but it is used for lack of anything better. Thus, for example with k=2, we obtain:

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})P(w_4|w_{2:3})...P(w_n|w_{n-2:n-1})$$

The parameters to be estimated for such a model are the $P(w_i|w_{i-k:i-1})$. This can be done using relative frequency on a large corpus: the corpus does not have to be manually annotated, but is simply automatically segmented into sentences and tokens, so that one can use corpora of hundreds of millions or even billions of tokens. A specific "smoothing" treatment must be introduced to manage the unseen occurrences in the estimation corpus.

Problem 1: tension between having a small k and a large k:

[6] Recommended reading: chapter 9, Goldberg's book

[7] cf. probas course, not to be confused with the "chain rule of derivation" seen in the context of back-propagation. The chain rule of probability applied to the conjunction of 3 events A, B and C is written $P(ABC) = P(A)P(B|A)P(C|AB)$. All permutations are possible e.g. $P(ABC) = P(B)P(C|B)P(A|BC)$.

- On the one hand, if k is too small, the approximation is too coarse. For example, if k is 2, to compute the probability of occurrence of a word after "a pineapple is very", we will only consider "is very", and we will not be able to favor for example "sweet" at the expense of "happy".
- On the other hand, if k is too large:
  - even with a huge corpus, estimating $P(w_i|w_{i-k:i-1})$ is not sufficiently reliable, and in practice, reaching k=5 is already very difficult and requires huge corpora
  - the number of parameters to estimate gets huge: increasing k increases the number of parameters by a factor $|V|$: for k=2, for a vocabulary V, we have $|V|^3$ possible trigrams, more generally we have about $|V|^{k+1}$ parameters to estimate.

Problem 2: similarity between words not taken into account: In addition, the parameters $P(w_i|w_{i-k:i-1})$ use the words as completely distinct from each other. For example, the estimation of $P(automobile|drive\ a)$ is totally independent of that of $P(car|drive\ a)$: the distributional hypothesis discussed above is not used at all: the fact that "automobile" and "car" have globally similar contexts is not exploited to estimate $P(automobile|x\ y)$ and $P(car|x\ y)$.

12.3.2 Neural language model

? popularize the concept of neural language model, with **two ideas fundamental for current NLP**[8]:

- the first central idea is to view the problem of estimating $P(w_i|left\ context)$ as a classification problem, whose classes are all the words of the vocabulary: the proba $P(w_i|w_{i-k:i-1})$ is obtained as output of a probabilistic neural network (i.e. with softmax as output). This network takes as input the ids of the words of the context $w_{i-k:i-1}$, and as output, the possible "classes" are the different words of the vocabulary (we thus have $|V|$ neurons as output). Once the network is learned (see below), for any context of k words of the vocabulary $c_1, ...c_k$, applying the forward propagation with this context as input will provide a probability distribution $P(.|c_1, ..., c_k)$.
- the second idea is to integrate an embedding matrix as parameters of the network (as seen in the previous section), thus allowing to benefit from a representation of words where similar words are vectorially close

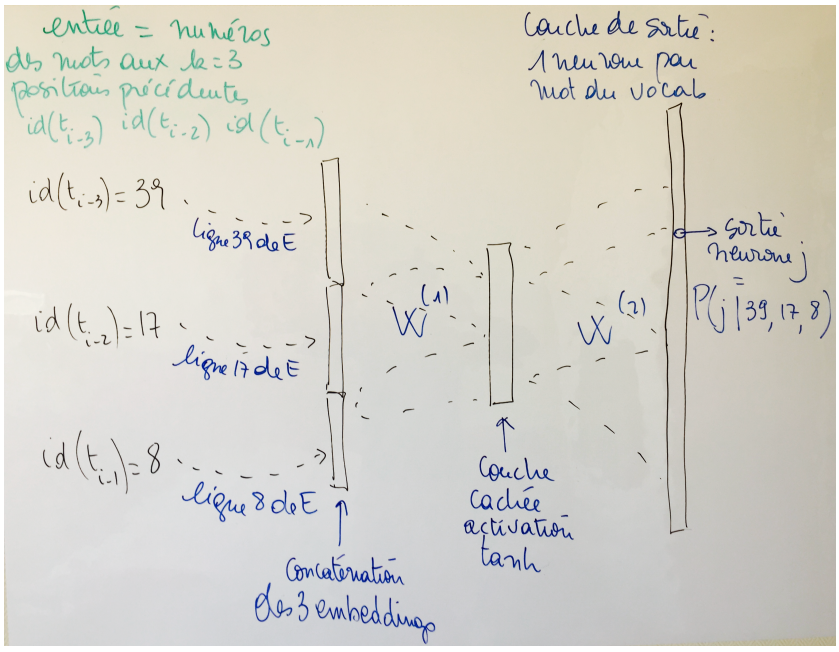


Figure 1: Simplified version of the neural network of the language model of **?** (we removed the direct connections of the embeddings to the output layer).

[8]These authors cite various pioneering works concerning the use of neural networks for a language model, and the use of distributed word representations, but the fundamental idea here is to learn the lexical embedding matrix in the language model network, and that this matrix is shared for all words

Forward Propagation: The network consists of an embedding layer (= the concatenation of the k embeddings of the k context words, a hidden layer with tanh activation, and an output layer of the size of the vocabulary, with softmax activation[9]. Let us detail the network-model by taking k=3. We consider the sentence *The little cat plays ball*, and the position i=5 : the network gives as output the proba of the different words of the vocabulary at the position i=5, considering as context the k=3 previous words, so here $P(W = . \mid t_2\ t_3\ t_4)$. The tokens belong to a vocabulary of V words numbered from 1 to $|V|$, and we note $id(t_j)$ the word id of t_j. In the figure, we have assumed that $id(t_2) = 39, id(t_3) = 17, id(t_4) = 8$. The jth neuron in output has as value $P(W = j|39, 17, 8)$: the probability that the word is the word number j, knowing that the 3 previous words are (in order) the words number 39, 17 and 8.

Learning : Learning is done in a supervised way, **by trivially creating examples (context, word gold) from a tokenized corpus**. The usable loss is the cross-entropy i.e. -log of the probability of the gold word. It is important to remember that we are technically using supervised learning, but on a training set obtained trivially from raw corpora: as we do not need to manually annotate any data, the size of the training corpus can be very large. This is sometimes referred to as **self-supervised learning**. **NB: Learning using a word prediction type objective is the basis of current advances in NLP.**
The resulting learning provides as a by-product a randomly initialized matrix E of word embeddings, whose learning is guided by the objective that **the vectors of the previous k words give a high probability to the current word actually observed**. We can see empirically that this objective allows us to have "close" vectors for words having close distributions.
The advantage is enormous, because we solve the 2 problems mentioned in the previous section:

- On the one hand, increasing k will be possible, without drastically increasing the number of parameters. Indeed, the parameters of the network are the matrix E : $|V|$ rows and M columns (M=size of the lexical embeddings), matrix W^1 : k rows and h columns (h = size of the hidden layer) and $W^{(2)}$: h rows and $|V|$ columns. So if k increases by 1, we only add $M \times h$ parameters (in W^1).
- and moreover we benefit from a vector space of words (the matrix of embeddings), and thus the similarities between words are taken into account

Note however that having a softmax on a vector of $|V|$ values is computationally expensive: for a large corpus one can have a vocabulary of 50,000 or even 100,000 distinct words, when training, to obtain a cross-entropy type loss, the computation of the denominator is expensive and the training is too slow. Many techniques have been proposed, such as hierarchical softmax or negative sampling (see below), but the improvement of the computational capabilities has more or less made this problem disappear.

12.4 Approach via the word prediction task: Word2vec

Among the many proposals for computing word vectors, we choose to detail the very famous word2vec model, described in (**??**). This model has had a resounding echo (its authors work at Google), and is commonly associated with the idea of deep learning, but it is comical to note that in fact word2vec is precisely a simplification of the above idea of integrating lexical embeddings into a neural network: word2vec proposes two models (CBOW and Skip-gram) which are in fact non-deep models: log-linear (!) models, i.e. without any non-linear activation function (except for the final softmax, which does not count). The objective was to speed up the learning process, using very simple models (hence the title of the article "Efficient Estimation of Word Representations in Vector Space").

Word2vec builds on the integration of word vectors within a language model network. But it focuses on learning the word embeddings matrix, **dropping the constraint of having to compute the probabilities of a whole sentence, so that the context of a word can be constituted by its preceding words but also by its following words.**

Word2vec (i.e. the two publications (**??**))

- defines two models, CBOW and Skip-Gram, as two different prediction goals,
- and proposes the technique of negative sampling to circumvent the computational problem of computing the softmax on a vector having the size of an entire vocabulary.

12.4.1 CBOW Model

Let the example sentence *The little cat played ball for 2 hours*, and the position i=4 (the word "played").

[9] In the original paper we also have a direct connection from the embeddings to the output layer, but we will ignore this point here for simplicity

The CBOW model (for continuous bag of words) considers the task of predicting a word w, from a window of k words around w. Let us take the case of a window of size 2 (i.e. we take k=2*2 context words). The sentence above is used to create the learning examples:

- (x = [The, little, played, ball], y = cat)
- (x = [little, cat, ball, for], y = play)
- (x = [cat, played, for, 2], y = ball)
- etc...
- as usual, you have to consider false words for the positions of beginning of sentence and end of sentence, to derive for example (x = [*d*, *d*, little, cat], y = The)

The network defined for CBOW has the following characteristics:

- parameters: a matrix of embeddings of words as context $E^{(c)} \in \mathbb{R}^{|V| \times m}$ (one row per word), and a matrix of target words $E^{(w)} \in \mathbb{R}^{m \times |V|}$ (one column per word)
- forward propagation for a single example (x, y) , with x = the ids of the input context words, and y the target word id:
  - input = x = (the id of the) context words (here 4 words, e.g. "little", "chat", "to", "the")
  - first operation is to compute e = the **sum** of the embeddings $E^{(c)}$ for these 4 words (but **without any activation function**)
  - then linear combination towards the output vocabulary, using the matrix $E^{(w)} \in \mathbb{R}^{m \times |V|}$ (as parameters)
    - the output probability distribution is $o = \text{softmax}(e.E^{(w)})$
  - $\longrightarrow$ output = one neuron per word of the vocabulary (but see below the different output, in the case of negative sampling).
- loss is NLL i.e. -log of probability of the gold target word y (loss = $-log(o_y)$)
- note the network is not used to predict anything, but just to learn the matrices

The model is called CBOW for "continuous bag of words" because to compute the content of the hidden layer, we sum the lexical embedding of each of the k context words: the positions are interchangeable, hence the term "bag of words". The "continuous" term is because we sum real-valued vectors instead of one-hot vectors.
Note that both matrices $E^{(c)}$ and $E^{(w)}$ are learnt, and both can be viewed as matrices of word vectors. In general, only $E^{(s)}$ is used as output word embeddings.

12.4.2 Skip-Gram Model

The Skip-gram model reverses the task, thus considering the even more artificial task of predicting, from a starting target word, a word from its context. So still for a window of size 2*2, for our previous example, we create the learning examples:

- with target word = "chat" : (x=chat, y=The), (x=chat, y=little), (x=chat, y=play), (x=chat, y=to)
- with target word = "play" : (x=play, y=little), (x=play, y=cat), (x=play, y=to), (x=play, y=la)
- etc...

The architecture of the network is even simpler:

- parameters: same as CBOW but transposed: $E^{(w)} \in \mathbb{R}^{|V| \times m}$, $E^{(c)} \in \mathbb{R}^{m \times |V|}$
- input = (the id of the) target word t
- get the corresponding embedding in $E^{(w)}$ (its tth row)
- linear combination to get one score per word of the vocab $E^{(w)}_t.E^{(c)}$
- softmax to get probabilities over the vocabulary

Again, skip-gram is used for learning only (not for prediction), the loss is -log(probability of the gold context word for this example).

12.4.3 Negative sampling

Negative sampling was introduced to overcome the computational problem of applying a softmax over a large vocabulary.
We will take here the case of the skip-gram model (but it is the same principle for CBOW)[10].
Instead of producing, for a given target word w_I, a complete probability distribution for all words in the vocabulary (representing the probability that another word is in the context of w_I), we consider a probabilistic binary classification task, which takes as input a pair of words w_I, w_O, and provides as output the probability that w_O is a word in the context of w_I.
A tokenized corpus gives us positive examples (y=1):

- with target word = "cat" : (x=[cat,The], y=1), (x=[cat,small],y=1) (x=[cat,play],y=1), (x=[cat,to],y=1)
- etc...

To learn correctly, you also need negative examples[11]: for each positive example $[w_I, w_O]$, we randomly draw k words $w_1, ...w_k$ which are not in the context of w_I. More precisely, we draw randomly following a certain probability distribution related to the frequency of the words (frequent words are more likely to be drawn).[12], which is called "sampling", hence the term negative sampling.
So for example, next to the positive example (x=[cat,small],y=1) seen in the corpus, we use k negative examples like for example "can", "eating", "sky", which gives 3 negative examples (x=[cat,can],y=0), (x=[cat,eating],y=0), (x=[chat,sky],y=0) ... Note that the drawing of negative words can very well give examples which in fact exist in the corpus, which will thus be sometimes positive and sometimes negative, but it doesn't matter.
Mathematically, the parameter matrices are the same as in plain skip-gram $E^{(w)}$ and $E^{(c)}$

- let's note $v_{w_I}^{(w)}$ the vector in $E^{(c)}$ associated to the word w_I, when w_I plays the role of target word
- and $v_w^{(c)}$ the vector in $E^{(w)}$ associated to a word w, when w plays the role of context word

To compute the proba that w_I, w_O is a positive example (i.e. that w_O is likely to be the context of w_I), we simply use the dot product of the target word and the context word, and then a sigmoid function to obtain a probability:

$$P(Y = 1|w_I, w_O) = \sigma(v_{w_I}^{(w)}.v_{w_O}^{(c)})$$

To compute the proba that w_I, w_n is a negative example (i.e. that w_n is not likely to be the context of w_I):

$$P(Y = 0|w_I, w_n) = 1 - P(Y = 1|w_I, w_n) = 1 - \sigma(v_{w_I}^{(w)}.v_{w_n}^{(c)}) = \sigma(-v_{w_I}^{(w)}.v_{w_n}^{(c)})$$

The objective function to be maximized for a positive example (x=w_I, w_O, y=1) and k negative examples obtained by replacing w_O by $w_{n1}, w_{n2}..., w_{nk}$ is:

$$log\ \sigma(v_{w_I}^{(w)}.v_{w_O}^{(c)}) + \sum_{i=1}^{k}[log\ \sigma(-v_{w_I}^{(w)}.v_{w_{ni}}^{(c)})]$$

On which we can apply the usual stochastic optimization techniques (such as a gradient "ascent", cf. objective to maximize instead of minimize).

12.4.4 Tricks and comparison to the counting method

Word2vec has very quickly become extremely popular. **?** show however two important points:

- On the one hand there is a certain mathematical equivalence between word2vec and the classical technique (i.e. by counting, PPMI weighting and SVD dimensionality reduction): word2vec is therefore not new from a theoretical point of view

[10] We call it SGNS for skip-gram with negative sampling. We follow the way of presenting **?**, clearly introducing a binary classification, which is not the case of the original article of word2vec.

[11] In the model without negative sampling, the examples are of type (x=target word, y=context word), so the gold "class" varies over the whole vocabulary. On the contrary, in negative sampling, we extract from the corpus examples of the positive class (y=1) only.

[12] Technically, let us take an example to illustrate how we can sample following a certain finite discrete probability distribution. Suppose we have only 4 vocabulary words, and we want to randomly draw one of the 4 words following the distribution $(0.1, 0.3, 0.15, 0.45)$. We pass to the cumulative probabilities: $(0.1, 0.3 + 0.1, 0.3 + 0.1 + 0.15, 1) = (0.1, 0.4, 0.55, 1)$, which gives 4 intervals $x \leq 0.1$, $0.1 < x \leq 0.4$, $0.4 < x \leq 0.55$, $0.55 < x \leq 1$. We randomly draw a nb between 0 and 1, if it belongs to the interval i, we return the word i. In word2vec, the proba of drawing a word w_i as a negative example is $P_n(w_i)$ = a slight change in the relative frequency of w_i (i.e. $nbocc(wi)/nbocc(w_i)$): word2vec uses $P_n(w_i) = \frac{nbocc(wi)^{\frac{3}{4}}}{\sum_j nbocc(w_j)^{\frac{3}{4}}}$

- On the other hand, the performance gains reported for example by **?** on different lexical semantic tasks, using word2vec vs. "classical" vectors are in fact due to details of the word2vec implementation, which once transposed into the classical technique, allow to have comparable performances. These "details" are in particular:
  - **subsampling of frequent words**: for any occurrence of a frequent word (whose frequency f is higher than a threshold t), one chooses randomly to ignore the occurrence In practice, choosing randomly "yes" with a probability p means to draw at random a nb x between 0 and 1, and to choose "yes" if $x \leq p.$, with probability $p = 1 - \sqrt{\frac{t}{f}}$. We see that the more frequent the word is, the more likely it will be ignored.
  - **smoothed distribution for sampling negative words**: as indicated above, the proba distribution used to draw the words to create the negative examples is not directly the relative frequency of the words: word2vec actually uses $P_n(w_i) = \frac{nbocc(wi)^{\frac{3}{4}}}{\sum_{w_j \in V} nbocc(w_j)^{\frac{3}{4}}}$.
  - **sampling of the window size**: finally, when constructing the training examples, the size of the (half)window is not actually fixed L, but sampled between 1 and L (with a uniform distribution): this means that context words at distance 1 from the target word (half-window size = 1) will always be used, but for example if L=5, words at distance 5 will only be used on average once out of 5 times[13].

The lesson to learn is that the devil is in the details. Note that some of these details were not explicit in the 2 word2vec publications, people had to check the code to actually understand what was done. The fact remains that word2vec has opened the way to a very efficient (and easily parallelizable) computation of word embeddings.

12.5 Evaluation of word embeddings

The evaluation of the quality of word embeddings can be extrinsic or intrinsic.
With extrinsic evaluation, we study the impact of the embeddings when they are used for a task other than constructing the embeddings themselves. We can measure the contribution of pre-trained embeddings within a neural network for a tagging or parsing task, etc...

In the intrinsic case, the aim is to evaluate the word embeddings in a more direct way. We come back to the first generic objective of word embeddings: to capture the semantics of words. Different lexical semantics tasks have been proposed to evaluate this objective, in particular semantic similarity, and analogy tasks.

12.5.1 Evaluation via semantic similarity task

We compare for a pair of words:

- the predicted similarity according to the word vector space (a simple cosine between the vectors of the 2 words)
- and similarity evaluated by humans: we ask people to "rate" on a scale (e.g. from 0 to 4) the semantic similarity of 2 words, averaged over several human judgments

On this task, the usable metric is a measure of the correlation between predicted similarities and human similarities for the same word pairs: the Spearman correlation coefficient. We are looking for the degree to which the correlation is "positive", i.e. the degree to which when one of the scores increases, the other also increases.
We consider a sample containing n pairs of words and 2 statistical variables on the word pairs:

- X = the human similarity score
- Y = the predicted similarity score (cosine)

We define X' and Y' as the s.v. giving the rank in the X series and the rank in the Y series. Example:

- 4 pairs of words P1, P2, P3 and P4
- X = human scores: 2.7 3.2 0.3 1.8
- Y = system scores: 0.83 0.8 0.01 0.3
- X' = rank of human scores: 2 1 4 3
- Y' = rank of the system scores : 1 2 4 3

[13] It can be calculated that for a single target occurrence, words at distance i will have a probability of $\frac{L-i+1}{L}$ to be used.

The Spearman coefficient between X and Y is the Pearson coefficient between the ranks X' and Y' (= the linear correlation coefficient between X' and Y') = $r = \frac{covariance(X',Y')}{\sigma(X')\sigma(Y')}$.
The Spearman coefficient is between -1 and 1, the closer it is to 1, the more there is a positive linear correlation between the ranks X' and Y': the rank Y' can be written as an affine combination of the rank X'. Switching to ranks (i.e. not using Pearson directly on X and Y) captures a correlation that may not be linear. We simply want to measure how much when one of the scores increases, the other also increases.
Famous word pair games:

- RG65 and its FR version http://www.site.uottawa.ca/~mjoub063/wordsims.htm
- English WordSim-353 (Finkelstein et al., 2001), split by (Agirre et al. 09) into two parts:
  - WordSim-353-S : for the topic similarity (for example *cup / coffee* or *lion / zoo* (called "relatedness" in English)
  - WordSim-353-F : for functional similarity (for example *cup / bowl* or *lion / panther*) (called "similarity" in English)
  - http://alfonseca.org/eng/research/wordsim353.html
- SimLex 999 : functional similarity https://fh295.github.io/simlex.html
- Multilingual datasets: http://lcl.uniroma1.it/similarity-datasets/

12.5.2 Evaluation using an analogy task

On top of negative sampling, the (**?**) paper comprises a new method to evaluate word embeddings, which became very popular: the analogy task.

We construct quadruplets of words A, B, C, D, such as A : B :: C : D, which is interpreted as "the relation between the words A and B is the same as the relation between the words C and D". For example we can consider the relation "to be the capital of", and the quadruplet Paris : France :: Rome : Italy, or the popular example "to be the masculine of", and the quadruplet queen : king :: woman : man. See the relations initially proposed in (?), and the data sets in different languages, including English and French https://fasttext.cc/docs/en/crawl-vectors.html#evaluation-datasets.

Prediction of a word by analogy : In vector terms, if we denote xA the vector of A (and same for the others), the analogy A : B :: C : D is supposed to be translated as $xA - xB = xC - xD$. The assessable task can then be to **guess the word D given a triplet A : B :: C : ?**. Concretely, given a vector space of words, for a triplet A, B, C, the predicted word is the word Z whose vector xZ is closest to -xA + xB + xC (ignoring the words A, B and C themselves). We can then compute an accuracy on a whole set of gold quadruplets.

12.6 To go further: improvements and issues

Research on word embeddings is extremely active, and there are advances in various directions:

- Models using subwords: initially proposed by **?** (fasttext). Are calculated embeddings of words, but also of character ngrams. A word has its own embedding, plus an embedding obtained as the sum of the embeddings of the ngram of characters it contains.
- Work on contexts: in particular use syntactic contexts instead of linear contexts, cf. dep2vecgoldberg-word2vec-14, but the improvements are modest.
- Sense embeddings
- Multilingual embeddings
- Resource exploitation (retrofitting): this involves modifying pre-entered embeddings, so that they are more consistent with a pre-existing resource that can be transformed into a lexical network. If two words are close in the lexical network, we try to make sure that they are also close in the vector space of lexical embeddings. See among others (**?**)
- and finally, the new state of the art concerning word embeddings: one computes an embedding for a given word occurrence, a "contextual embedding", instead of a static word embedding. Cf. in particular the BERT model (**?**) and its variants.