UNIVERSITÀ DI PISA

Intelligent System for Pattern Recognition

Notes by Gennaro Daniele Acciaro

AA: 2021/2022

Index

this page was intentionally left blank

(use it to give vent to your frustrations)

Fundamentals

Time-series

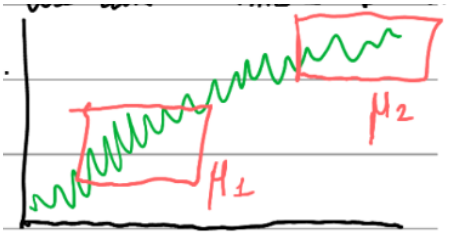
A time-series X is a sequence of measurements.

$$X = x_0 .. x_N$$

The time between two different measurements could be different from the time between another two different measurements (irregular time intervals).

Time series are assumed **weakly stationary** which means that the expectation is equal to a certain value called μ for each measurement:

$$\mathbb{E}[x_t] = \mu$$



If we don't have a weakly stationary we can normalize data. An example of non-weakly stationary time-serie is the one on the right, where $\mu_1 \neq \mu_2$.

Moreover, we need fixed covariance for fixed length:

$$Cov(x_{t+\tau}, x_t) = y_\tau$$

where τ is called "lag" — it's a sliding window.

Key methods

We can study a signal from two different points of view.

1) **Time domain analysis**
   which is based on **how a signal changes over time**

2) **Spectral domain analysis**
   which uses the **frequency domain** (spectrum).
   In particular, it uses **the Fourier transformation** to convert the time-domain signal into a spectrum, then when the work is done, it uses the **inverse Fourier transformation** to return to the time-domain

Time Domain Analysis

Autocovariance

In order to introduce the most simple process in the time-domain analysis (the autocorrelation) we need to introduce the concept of **autocovariance**.
Since we are using just one time series, we can use the "auto" word.

$$\hat{y}_X(\tau) = \frac{1}{N}\sum_{t=1}^{N-|Y|}(x_{t+|\tau|} - \hat{\mu})(x_t - \hat{\mu})$$

where τ is the lag an $\hat{\mu}$ is the simple meaning.

Autocorrelation

For "correlation" we mean how much a signal is similar to another signal.
So for **auto**correlation we mean a value that tells us how much the signal is correlated with itself.

The autocorrelation analysis can help us to reveal patterns and it's performed using the auto-covariance:

$$\hat{p}_x(\tau) = \frac{\hat{y}_x(\tau)}{\hat{y}_x(0)}$$

Cross-correlation

The cross-correlation is a measure between two different signals x^1 and x^2.
As the autocorrelation, it's based on a time lag τ.

$$\phi_{x^1x^2}(\tau) = \sum_{t=\max\{0,\tau\}}^{\min\{(T^1-1+\tau),(T^2-1)\}} x^1(t-\tau)\cdot x^2(t)$$

2 SEGNALI

This measure can be normalized using:

$$\bar{\phi}_{x^1x^2}(\tau) = \frac{\phi_{x^1x^2}}{\sqrt{\sum_{t=0}^{T^1-1}\left(x^1(t)\right)^2 \sum_{t=0}^{T^2-1}\left(x^2(t)\right)^2}} \in [-1,+1]$$

this value is useful because, if:

- $= 1$, the two signals have the same shape
- $= -1$, the two signals have the same shape but in opposite sign
- $= 0$, the two signals aren't **linearly** similar but this doesn't mean that they could be correlated in higher dimensions.

Convolution

The first formula in the cross-correlation is very similar to the formula of **convolution**.
Recall that the convolution between two functions is an operation that produces a third function that expresses how the shape of one is modified by the other.

$$(f * g)[n] = \sum_{t=-M}^{M} f(n-t)g(t)$$

The operation is commutative, and we can see it as a smoothing operation.

AutoRegressive Process

A linear system with this form is called **AutoRegressive process (AR)** of order k:

$$x_\tau = \sum_{k=1}^{K} \alpha_k x_{\tau-l} + \epsilon_\tau$$

"AutoRegressive" means that the time series x_t regress on itself.

α_k are linear coefficient and they are all less than one for stability reasons. They are a set of hyperparameters, so we can select them using model selection.

ϵ_τ are errors, and in an AR we assume them as i.i.d.

ARMA

The fact that the errors ϵ_t are i.i.d. could be not always true because in a time series, usually, error tends to be compounded.
Based on this idea, we can introduce the Autoregressive with Moving Average process (ARMA) which compounds through time.

$$x_t = \sum_{k=1}^{K} \alpha_k x_{t-k} + \sum_{q=1}^{Q} \beta_q \epsilon_{t-q} + \epsilon_t$$

The current value is the result of a regression on its past values plus a term that depends on a combination of uncorrelated information.
As far α values, the βs can be selected using model selection.

Comparing time series using AR

A first way to compare time series is using AR, indeed we can

$$d(x^1, x^2) = \|\alpha^1 - \alpha^2\|_M^2$$

A use case could be the **anomaly detection**, indeed if we use the AR process to predict a value $\hat{x}_t$ and we know the real value x_t we can check anomalies.
Since α and β parameters identify a specific signal, we can use these vector as input for a ML model.

Spectral Analysis

Recall that the spectral analysis is based on decomposing time series into linear combinations of sinusoidal and using these sinusoidal to make regression.

Using the Discrete Fourier Transformation (DFT) we can transform a time series from the time domain into the frequency domain, and we can easily execute this process and vice versa. This allows us to find periodicity in time series.

Representing functions

Given an orthonormal system of bases $E = \{e_1 .. \ e_n\}$ we can represent any function $f \in E$ as as a linear combination of bases:

$$\sum_{k=1}^{\infty} \langle f, e_k \rangle e_k$$

where $\langle f, \ e_k \rangle$ can be continuous (integrals) or discrete (summatory) between f and the basis e.

Given the following orthonormal system:

$$\{\frac{1}{\sqrt{2}}, \sin(x), \cos(x), \sin(2x), \cos(2x), \dots\}$$

we can describe the Fourier Series as:

$$\frac{a_0}{2} + \sum_{k=1}^{\infty} [a_k \cos(kx) + b_k \sin(kx)]$$

where α, β are coefficient calculated integrating $f(x)$ with the sin and the cos functions.

Representing function in complex space

If we move into the complex space, we can use the equation:

$$e^{-i\,k\,x} = cos(k_x) - sin(k_x)$$

to rewrite the Fourier Series as:

$$\sum_{k=-\infty}^{\infty} c_k e^{ikx}$$

on the orthonormal system :

$$\{1, e^{ix}, e^{-ix}, e^{2ix}, e^{-2ix}, \dots\}$$

where c integrates $f(x)$ using $e^{-i\,k\,x}$.

The complex space allows us to plot the function in a 2D euclidean space.

Image processing

We want to extract useful information from images.
The properties needed for a good representation of visual information are:

1) the informations have to be **informative**
2) they need to be **invariant** to some transformations like zoom, rotations, scaling and light.
3) the representation must be efficient for indexing and querying

In an image, we can define:

- Edges (lines)
- Blob (homogeneous pixels)
- Segments (larger blobs)

Image histograms

The simpler way to get information from an image are the **image histograms**; a color histogram is a graphical representation of the number of pixels.
The same idea can be applied to edges and corners.
Any information about shapes and positions is lost, so histograms are invariant.
We can use histograms to compare and index images.

SIFT

The **Scale-Invariant Feature Transform** is a method for extracting relevant features which are invariant to the image scale, rotation, noise and change in luminous intensity.

This is performed using a Gaussian Filter, which is convoluted with a portion of the image (called patch)

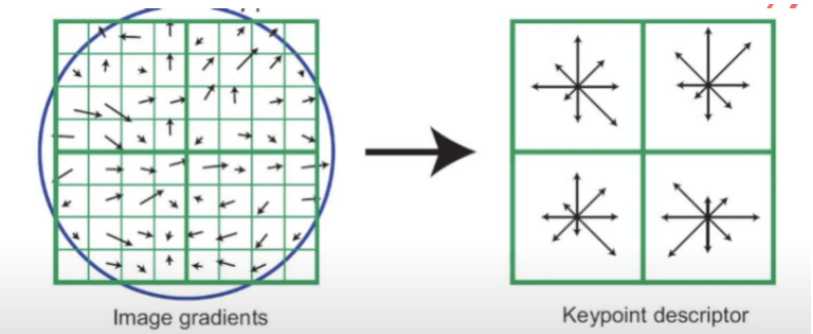
SIFT: orientation assignment

For each candidate feature, we define a "patch" as a group of pixels near the feature.
For each pixel in a patch, we compare the pixel above with the pixel below, and the pixel on the right with the one on the left using a function that is the convolution of a Gaussian filter with the image itself.
This value is called **magnitude** and if we apply the $tanh^{-1}$ to the ratio between the two differences we obtain the orientation.

$$m_\sigma(x,y) = \sqrt{\left(L_\sigma(x+1,y) - L_\sigma(x-1,y)\right)^2 + \left(L_\sigma(x,y+1) - L_\sigma(x,y-1)\right)^2}$$

$$\theta_\sigma(x,y) = \tan^{-1}\left(\frac{\left(L_\sigma(x,y+1) - L_\sigma(x,y-1)\right)}{\left(L_\sigma(x+1,y) - L_\sigma(x-1,y)\right)}\right)$$

The magnitude is larger when we found an edge in a pixel within the patch (smaller in blobs)
So, we have a set of vectors, hence we can sum up every vector in order to obtain the **key point description.**



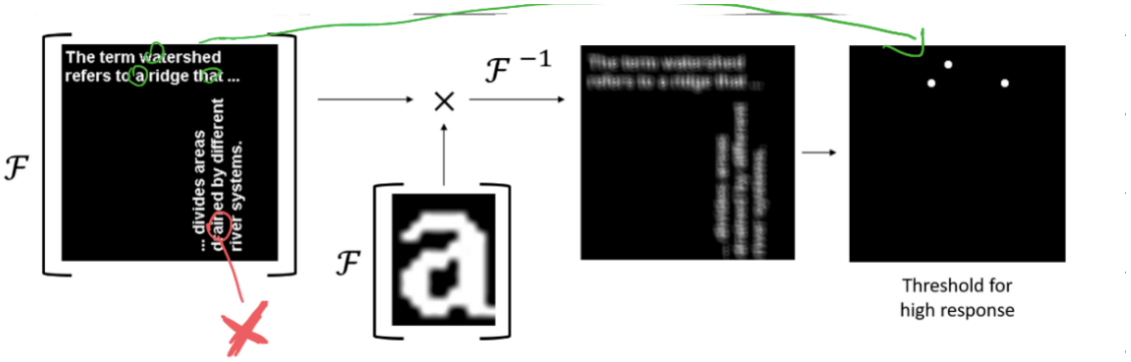
Fourier Analysis

Images can be seen as function that return an intensity value on a given pixel, indexed by an x and a y: $I(x, y)$

So we can use the **Fourier analysis** to write the image as a sum of sine and cosine waves:

$$H(k_x, k_y) = \sum_{x=1}^{N-1} \sum_{y=1}^{M-1} I(x, y) e^{-2\pi i \left(\frac{xk_x}{N} + \frac{yk_y}{M}\right)}$$

The Discrete Fourier Transformation can be useful for image pattern recognition.



Convolution theorem

The Fourier transform of the convolution of two function is the product between their single Fourier transform

$$\mathcal{F}(f * g) = \mathcal{F}(f)\mathcal{F}(g)$$

In image processing, this is so useful, since we can see an image I as a function and a filter g as another function.

So, we can easily obtain the result of the convolution $I * g$ applying efficiently the Fourier on I and g, and then use the inverse Fourier transformation to get the convolution.

$$I * g = (F)^{-1}(\mathcal{F}(I)\mathcal{F}(g))$$

Edge Detector

An edge is a short region with a big intensity change and the direction of change (so, how the edge is) is given by the gradient.
For this reason, we convoluted our image with two filters (one for horizontal lines and one for vertical lines).

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Blob Detector

A blob is a portion of the image with the same color.
For this reason the best way to recognize them is to recognize big slacks in gradient, indeed, inside the blob the gradient doesn't change too much.
We can recognize blob using 2° order derivatives: **Laplacian of Gaussian (LoG)**

But in order to compare responses, it's useful to normalize responses using scale.

Deeping on LoGs

We convolve the image with the LoG using different signals because we want to find the maximum extension for each blob.

But using a LoG filter can be inefficient, so for this reason we can approximate it using the **Difference of Gaussian (DoG)** where we can reuse Gaussian without calculating them every time.

Pro:
Detectors based on Laplace gradients are invariant to scale...

Cons:
... but not to transformation (for this reason we use MSER).

Maximally stable extremal region (MSER)

The Maximally stable extremal region (MSER) is another way for blob detection.
"Stable" is referred with respect to the intensity (the light)

The idea is to use a huge number of thresholds after the image is binarized, this idea gives us several benefits:

1) It's invariant to affine transformations
2) It's stable (thanks to multiple thresholds)
3) Multiscale, since blobs are identified by intensity and not by scale

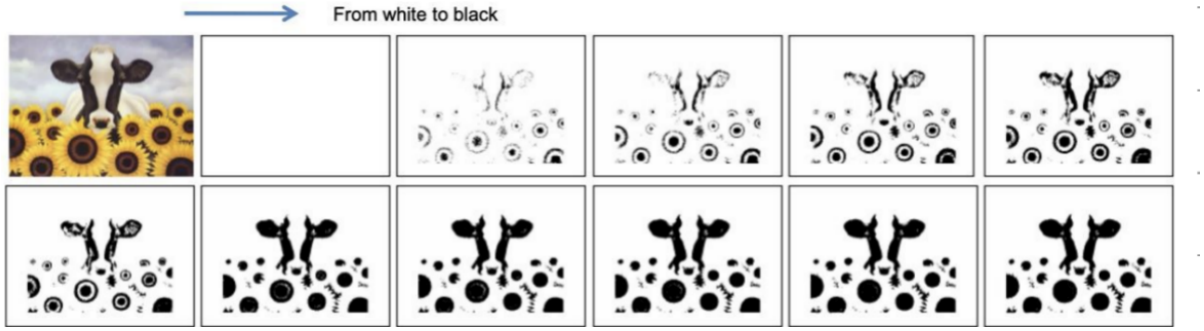


Image segmentation

The process of partitioning an image into a set of homogeneous pixels is called **image segmentation**.
A first approach would be the applying of the K-Means model to pixels, but it's not a fantastic method due to bad results.

A modern way is called NCut.

Normalized Cut

A modern approach to image segmentation is called **Normalized Cut (NCUT)**.
We can see an image as a graph where nodes are pixels.
We can label groups of adjacent nodes in order to obtain image segmentation. Furthermore, we can separate groups using some affinity (a such metric chosen by us). The separation is called **cut**.

The normalized cut could be NP-hard, so we can find an approximate solution using eigenvalues in the spectral domain.

In particular, since pixels in an image could be a lot, **Ncut could take ages to complete**.
For this reason, we introduce **Superpixels**, groups of similar pixels (important: a pixel can belong only to one superpixel, so superpixels don't overlap).
Once we have superpixels, we can apply Ncut (or every kind of Image Segmentation algorithm).

Wavelets : limitation of DFT

DFT can be applied to a whole sequence but sometimes we might need to localize frequencies. For this reason, we introduce **wavelets**.

A wavelet is a little wave that we use like a filter, indeed, we convolve it to the signal getting a local match; then we slice the wavelet.

A wavelet is regulated by a function $\Psi()$ that has two parameters: k and j, the first one regulates the scale, the second the shift.

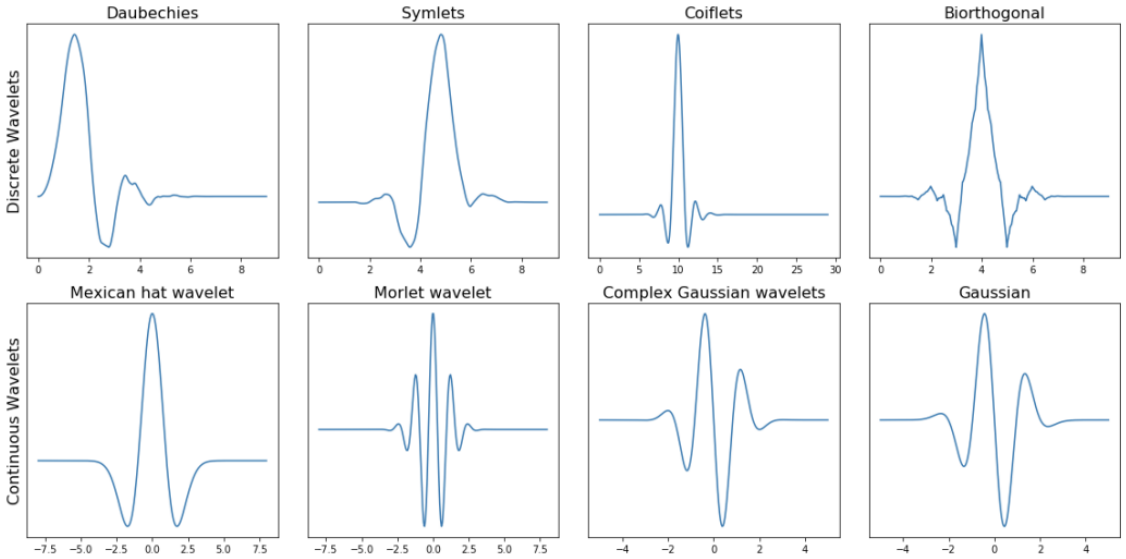
$$\sum_t \boldsymbol{x}(t)\, \Psi_{j,k}(t)$$

$$\Psi_{t,k}(\boldsymbol{x}) = 2^{\frac{k}{2}}\Psi((t - j2^k)/2^k)$$

In particular, if $k > 1$, the wavelet compress the signal, and $k < 1$ allows the wavelet to dilate the signal.

Wavelets can be more effective on discontinuous data.

Examples:



Generative Learning

Generative Learning theory supports models that represent knowledge as probabilities; we like this approach because knowing the distribution probability of data allows us to generate new data.
Furthermore, we can apply even some **prior knowledge** on data.
Using generative learning models, we are able to deal with many variables.

Graphical Model Framework

This framework allows us to describe:

1) How a generative model can be represented.
2) How to execute **inference**
3) How to learn model parameters

Graphical Model Representation

Graphical models are a compact way to represent exponentially large probability distributions encoding **conditional independence assumptions**.
We will use a graph whose nodes are random variables and edges are probabilistic relationships between variables.
We can have three different classes of graphs:

1) **directional models,** which explicit causes and effects (a very specific form of probability) using directional arcs. (Eg. Bayesian Networks)
2) **unidirectional models**, which are useful for soft constraints using undirectional arcs (Markov Networks)
3) **dynamic models**, which are graphs with no fixed structure which changes with data (HMM)

Probability Refresh

Random Variables

A random variable is a function which describes the result of a random process.
Uppercase letters denote a RV, and lowercase denotes observations.

Probability Function

Probability functions $P(X = x) \in [0, 1]$ is a discrete function that measures the probability of a RV X to have a certain value x.

The sum over all the possible values is 1: $\sum_{x} P(X = x) = 1$

The same concept can be extended to continuous functions, but in that case we use a density function $p(t)$ that describes the relative Likelihood of a RV to take on a value t.

Joint and conditional probability

Given a set of RV $x_1 .. x_n$, the joint probability is the probabilities to have at the same time the values $x_1 ... x_n$

$$P(X_1 = x_1, X_2 = x_2 ....)$$

If we want to measure the effect of the realization of an event y we can use the conditional probability:

$$P(x_1 ... x_n | y)$$

Chain Rule

$$P(x_1, .. , x_i, ..., x_n | y) = \prod_{i=1}^{N} P(x_i | x_1 .... x_{i-1}, y)$$

Marginalization

$$P(X) = \sum_{Y} P(X|Y)P(Y)$$

Expectation

Let X be a random variable, we define the **expected value** as:

$$\mathbb{E}[X] = \Sigma_{x \in domain\,(X)} x * P(X = x)$$

Conditional Independence

Two random variables X, Y are **marginally independent** if knowing X don't influence the estimation of Y and vice versa.

$$X, Y \; Marg. Indep \Leftrightarrow P(X, Y) = P(X)P(Y)$$

Instead, if we introduce a third RV Z, we can say that X, Y are **conditional independent** from Z if:

$$X \perp Y \mid Z \Leftrightarrow P(X, Y|Z) = P(X|Z)P(Y|Z)$$

Prior Distribution

In Bayesian inference, a prior distribution that expresses the beliefs one **previously** has about this quantity before some evidence is considered.

Posterior Distribution

Instead, the posterior distribution is a probability distribution that represents your updated beliefs about the parameter after having seen the data.

$$Posterior = Priori + New\ Evidence \qquad (Likelihood)$$

Inference and Learning with probability

Inference means how to find the distribution given observations.
With this definition, we can see the **learning** concepts from Machine Learning as an instance of inference, indeed, it uses a dataset (observations) to predict the distribution of RVs (output).

3 Approaches to inference

1) Bayesian

In the Bayesian approach, we consider **all the hypotheses** weighted by their probabilities.

$$P(X|d) = \Sigma_i P(X|h_i) P(h_i|d)$$

where X is an unknown RV, d is the dataset and i iterates over all the hypotheses $h_1 .. h_n$

This is an optimal approach, but has computational issues.

2) Maximum A-Posteriori

If we have all the probabilities of hypothesis, we could pick the most likely one, given the dataset.

$$h_{MAP} = \arg \max_{h \in H} P(h|d)$$
$$= \arg \max_{h \in H} P(d|h)P(h)$$

3) Maximum Likelihood

Assuming a **uniform prior** to MAP, which means that the $P(h)$ is equal for each $h \in H$, we can rewrite h_{MAP} into the Maximum Likelihood.

$$h_{ML} = \arg \max_{h \in H} P(d|h)$$

MAP and ML are point estimators of Bayesian, since they infer using only one hypothesis (instead of all).

Notice that MAP and ML become closer as more data gets available because the prior becomes smaller with respect to the posterior because in practice the likelihood is typically a multiplication of probability and longer is the data then longer will be the multiplication and the effect of this multiplication will be huge with respect to the simple prior

MAP is a regularization of ML

Recall:

$$h_{MAP} = \arg \max_{h \in H} P(d|h)P(h)$$

$$h_{ML} = \arg \max_{h \in H} P(d|h)$$

The $P(h)$ term can be seen as a trade-off between complexity and fitting, introducing preference across hypotheses.
It penalizes complex hypotheses, since they have a lower prior probability.

Graphical Models

Joint probability and exponential cost

Given a set of RV $X_1 \dots X_N$ describing the whole joint distribution as a table may be infeasible due to the exponential cost of enumerating all the possible values.
For this reason we will split this tale, as example, in several conditional probabilities tables (as the Bayesian Network does)

Graphical models

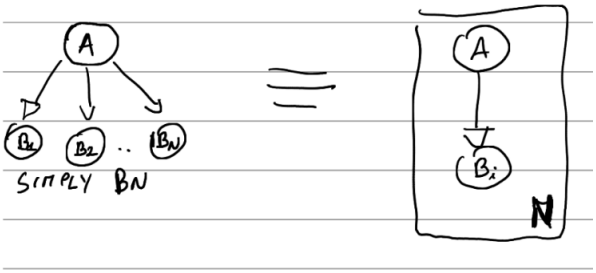
Graphical Models are compact representations of large joint distributions.
They simplify marginalization and inference algorithms, allowing us to add prior knowledge and relationship between RV. They can be directed (BN) or undirected (MRF).

Bayesian Networks

A **Bayesian Network** is a DAG which nodes represent RVs.
If we know the probabilities of a node (aka, we observe that node), we define the node as shaded, otherwise it's called empty.
Edges describe the conditional independence relationship and every node has a **Conditional Probability Table** which describes the probability distribution given its parent.

Plate Notation

Plate notation simplifies graphically a BN. If the same causal relationship is replicated for several variables, we can compactly represent it using this notation.

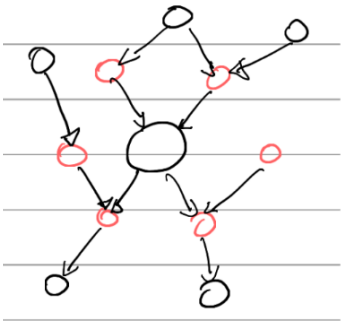


Conditional independence and Causality

Local Markov property

A node is **conditionally independent from all its non-discentents** given a joint state of its parent. In other words, a node only depends on its parents.

Markov Blanket



The **Markov Blanket** $MB(A)$ of a node A is the minimal set of nodes that shields the node from the rest of the BN. The behavior of a node is completely determined and predicted knowing its Markov blanket.

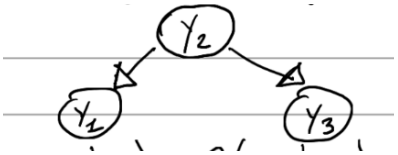
This MB contains the parents, the children and the other children of the parents of a fixed node.

Joint Probability Factorization

Using the chain rule and the Markov assumption, we can simplify the joint probability given the structure of the underlying Bayesian Network.

Fundamental BN Structure

1) Tail - to - tail (Causa in comune)



$$P(Y_1, Y_2, Y_3) = P(Y_1|Y_2)P(Y_3|Y_2)P(Y_2)$$

The dependency between Y_1 and Y_3 **changes when we observe** Y_2, indeed if Y_2 is unobserved then Y_1 and Y_3 are **marginally dependent**:

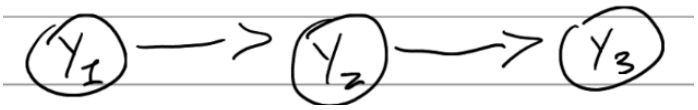
$$Y_1 \not\perp Y_3 \equiv P(Y_1|Y_3) \neq P(Y_1)$$

but if we observe Y_2, the two nodes Y_1 and Y_2 became **conditionally dependent** given Y_2

$$Y_1 \perp Y_3|Y_2 \Leftrightarrow P(Y_1, Y_3|Y_2) = P(Y_1|Y_2)\,P(Y_3|Y_2)$$

When we observe Y_2 we say that it blocks the path between Y_1 and Y_3

2) Head- to - tail (Causa - Effetto)

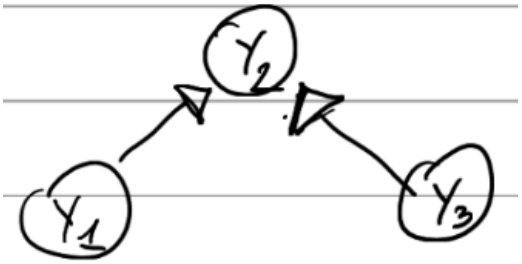


$$P(Y_1, Y_2, Y_3) = P(Y_1)\, P(Y_2|Y_1)\, P(Y_3|Y_2)$$

Since we can observe that Y_2 blocks the path between Y_1 and Y_3, it's the same as the previous case.

In other words, if we know the state of Y_2, we know everything we can about Y_3 without knowing the state of Y_1 .

3) Head- to - head (Effetto in comune)



$$P(Y_1, Y_2, Y_3) = P(Y_1)\, P(Y_3)\, P(Y_2|Y_1, Y_3)$$

Here we have the opposite case: if we observe Y_2 then Y_1 and Y_3 are conditionally dependent and if we don't observe Y_2, the nodes Y_1 and Y_3 are marginally independent.

- If Y_2 is observed then Y_1 and Y_3 are conditionally dependent

$$Y_1 \not\perp Y_3 | Y_2$$

- If Y_2 is unobserved then Y_1 and Y_3 are marginally independent

$$Y_1 \perp Y_3$$

But if any Y_2 descendants are observed, it unlocks the path.

Head-to-tail === tail-to-tail proof

Tail-to-Tail and Head-to-Tail imply the same factorization of the joint distribution.
The last observation that we can do is that if we reverse the order of Y1, Y2, Y3 and we get Y3, Y2, Y1 then we get exactly the same thing so it is still the same.

Recall:

Head-to-tail : $P(Y_1, Y_2, Y_3) = P(Y_1)\,P(Y_2|Y_1)\,P(Y_3|Y_2)$

Tail-to-tail : $P(Y_1, Y_2, Y_3) = P(Y_1|Y_2)P(Y_3|Y_2)P(Y_2)$

Bayes: $P(A|B) = P(A, B) \,/\, P(B)$

Bayes: $P(A, B) = P(A|B)\,P(B)$

Let's start from the Head-to-tail

$$P(Y_1, Y_2, Y_3) = P(Y_1)\,P(Y_2|Y_1)\,P(Y_3|Y_2)$$

The first two terms are equal to the joint probability of Y_1, Y_2 due to the Bayes Theorem.

$$P(Y_1)\,P(Y_2|Y_1) = P(Y_1, Y_2)$$

$$= P(Y_1, Y_2)\,P(Y_3|Y_2)$$

Using Bayes again, we can rewrite the last term:

$$= P(Y_1, Y_2)\frac{P(Y_3, Y_2)}{P(Y_2)} \qquad = \frac{P(Y_1, Y_2)}{P(Y_2)}\,P(Y_3, Y_2)$$

We can apply Bayes on both terms:

$$= \frac{P(Y_1, Y_2)}{P(Y_2)}\,P(Y_3|Y_2)P(Y_2) \qquad \text{(right)}$$

$$= P(Y_1|Y_2)\,\frac{P(Y_2)}{P(Y_2)}\,P(Y_3|Y_2)P(Y_2) \qquad \text{(left)}$$

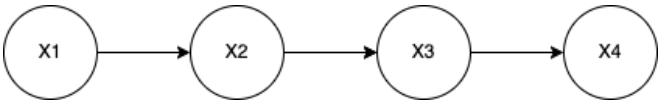
And we conclude with the simplification:

$$= P(Y_1|Y_2)\,P(Y_3|Y_2)P(Y_2)$$

Derived Relationships

In addition to the three basic structures, a BN can also represent **derived relationships**; this means that relationships between nodes are extendable through the paths of a BN.

Example 1

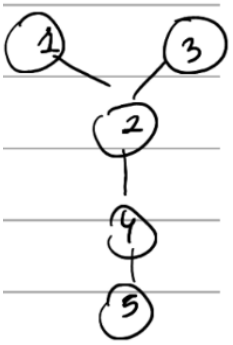


X1 and X3 are conditionally independent given X2. X4 and X1,X2 are conditional independent given X3.

$$X1 \perp X3 \mid X2 \quad ; \quad X4 \perp X1 \mid X2 \mid X3$$

Hence, we can infer that X1 is conditional independent by X4 given X2

$$X1 \perp X4 \mid X2$$



Example 2

$2 \perp 3 \mid 5$ is false
$2 \perp 5 \mid 4$ is true
$1 \perp 5 \mid 4$ is true

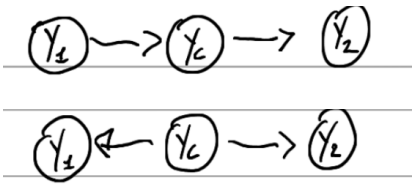
D-Separation

Let Y_1 and Y_2 be two RVs and let r the path that connects those two variables.

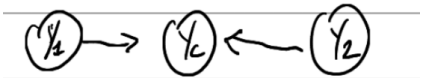
We define as **d-separated** the path r by a subset of nodes z if there exists at least one node $Y_c \in Z$ for which path r is blocked.

In other word, we can say that the path is d-separated if:

1) r contains a head-to-tail and $Y_c \in Z$
2) r contains a tail-to-tail and $Y_c \in Z$



3) r contains a head-to-head and neither Y_c nor its descendents are $\in Z$

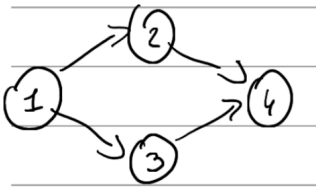


Markov Blanket and d-separation

Two nodes Y_i and Y_j in a Markov Blanket are d-separated by a set Z if and only if every undirected path between Y_i and Y_j are d-separated by z.

Undirected Representation (Why we need Markov Fields)

Bayesian Network cannot be used to model **symmetrical dependencies,** which need unidirectional graphs



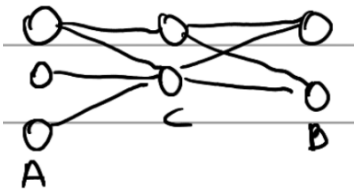
We can express with a BN both

- $1 \perp 3 \mid 2, 4$
- $2 \perp 4 \mid 1, 3$

For this reason, we need to introduce Markov Random Fields

Markov Fields (and D-separation)

A **Markov Random Field (MRF)** is a set of random variables that use the Markov property in order to describe an undirected graph.
We can extend the d-separation concept even to MRF, indeed suppose to have the following MRF:



We can say that A and B are d-separated by C because it blocks all the paths, which means that A and B are conditionally independent given C.

Joint Probability Factorization in Markov Fields

Recall: in directed models, we had the conditional probability of a node that is given with respect to its parents:

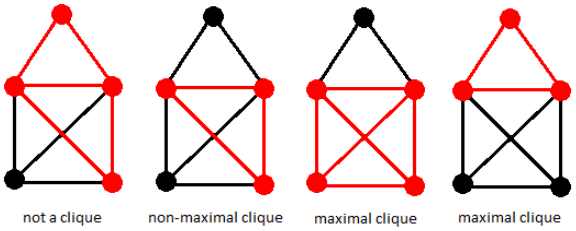
$$P(Y_1 \dots Y_n) = \prod_i^n P(Y_i \mid parent(Y_i))$$

What is the equivalent for undirected graphs?
According to the Markov Blanket idea, nodes that are not neighbors are conditionally independent given the remainder of the nodes.

$$P(X_v, X_i \mid X_{V \setminus \{v,i\}}) = P(X_v \mid X_{V \setminus \{v,i\}}) P(X_i \mid X_{V \setminus \{v,i\}})$$

$V \setminus \{v, i\}$ means all the nodes different from v and i.
Hence, we will use cliques that allow us to split in two different factor X_v and X_i nodes, if they are not neighbors.

Cliques



A clique is a subset of nodes C of an undirected graph G such that the graph contains an edge between all pairs of nodes in C.
In other words, we define a clique of an undirected graph as a subset of nodes such that for each couple of nodes, there exists an edge between them.
Then, we can define the concept of **maximal clique** such that a clique that cannot be extended by including one more adjacent vertex.

Maximal Cliques Factorization

Let X be a set $x_1 \dots x_n$ of random variable associated to the N nodes in the graph.
A theorem says that **we can obtain a factorization of a joint distribution as a product over maximal cliques** given the potential function Ψ which takes in input a partition $X_C \subset X$of the nodes.

Since we are multiplying functions I need to do an additional operation in order to be sure that the whole product is still a probability (which means that the sum-to-1 constraint holds).
For this reason we divide everything by a **partition function** $Z()$ which ensures normalization.

$$P(X) = 1/Z \prod_C \psi(X_C)$$

Potential Functions

A potential function expresses which configuration of local variables are preferred.
An example could be:

$$\psi(x_1, x_2) = \begin{cases} 1 & \text{if } x_1 = x_2 \\ 4 & \text{if } x_1 = 2x_2 \\ 0 & \text{otherwise} \end{cases}$$

It's worth noting that potential function $\Psi(X_c)$ are not probabilities.

If we restrict to only positive potential functions, a theorem provides guarantees that the **distribution can be represented by clique factorization**.
An easy and widely used strictly positive potential function is the following:

$$\Psi(X_C) = \exp\{-E(X_C)\}$$

Where E is called **energy function**

Moralization

Moralization is a way to create a coherent undirected graph from a directed one.
Indeed, if we don't perform the operation of connecting the parents, these nodes will belong to different cliques.

Structure Learning Problem

With Bayesian networks we could resolve another kind of problem: suppose that you know the nodes but you don't know the structure of the Bayesian network itself (the connection between the nodes). How can we create this structure?
We can have three approaches:

1) Search and Score

It's a model selection approach, a search into the space of all the possible graphs looking for the highest score.
The score is given by a function that have to respect two properties:

- Same score for equivalent graph
- Can be computed locally

The search is performed maximizing the score.

We have two approaches:

1) Start from a given structure, modifying it iteratively. Each modification has a cost, so it's a **cost optimization problem**
2) Knowing the order of the nodes we can reduce the search space, adding constraints

2) Constraint Based

Another approach to create the network is based on testing the conditional independence of two nodes X_i , X_j given their neighbors Z.

The choice of the testing order has just the aim of avoiding super-exponential complexity calculus.
We can use two ways to do this choice:

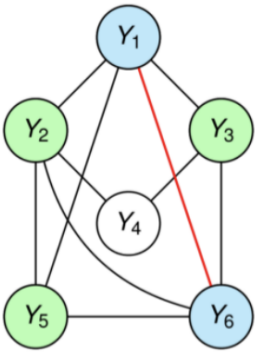
1) Lower-wise testing: starting from an empty Z, we test nodes in order to increase the size of Z.
2) Node-wise testing : tests are performed on a single edge at the time, checking on all the variables.

3) Hybrid

This is a mix of previous methods.
It relies on multi-stage algorithms that use constraints (based on independence tests) to create a skeleton, which is updated by a search-and-score algorithm.

PC Algorithm

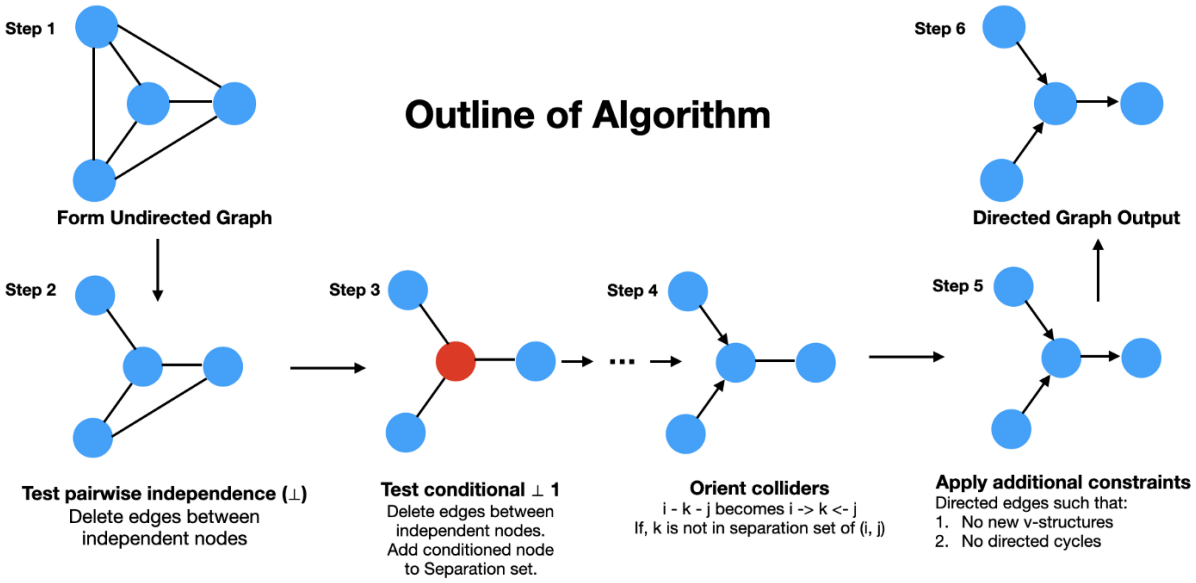
The PC algorithm is an algorithm used to discover causal relationships among nodes.
We start from a fully connected undirected graph.
Then, we delete an edge between two nodes if they are independent.
Then, we try to delete each edge that connect two nodes that are conditionally independent given a set of order $1 \dots |Z|$ (we try all these orders) of nodes.



Initialize a fully connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
for each edge $(Y_i, Y_j) \in \mathcal{V}$
- **if** $I(Y_i, Y_j)$ **then** prune (Y_i, Y_j)

$K \leftarrow 1$
for each test of order $K = |Z|$
- **for each** edge $(Y_i, Y_j) \in \mathcal{V}$
  - $Z \leftarrow$ set of conditioning sets of K-th order for Y_i, Y_j
  - **if** $I(Y_i, Y_j|z)$ **for any** $z \in Z$ **then** prune (Y_i, Y_j)
- $K \leftarrow K + 1$

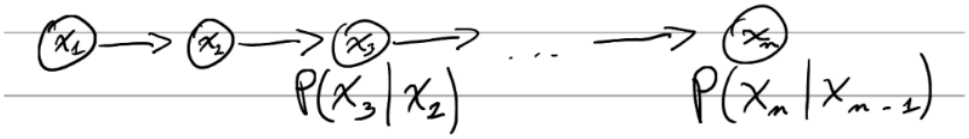
return $\mathcal{G}$



HMM

Markov Chain

A sequence $X = x_1 \dots x_n$ can be seen as a simple Bayesian Network like:



This is a **Markov Chain**: a directed graphical model for sequences such that each element depends only on its predecessors.

The joint probability can be factorized as:

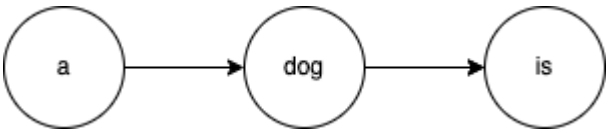
$$P(X) = P(X_1 \dots X_n) = P(X_1) \prod_{t=2}^{T} P(X_t | X_{t-1})$$

where:

- $P(X_1)$ is the **prior** distribution
- $\prod_{t=2}^{T} P(X_t | X_{t-1})$ is the **transition** distribution

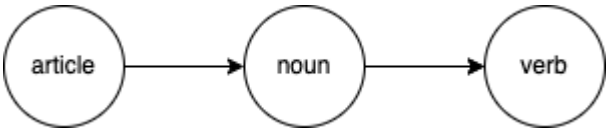
Deeping: Why do we need HMM?

We could use a Markov chain to model the relationship between elements in a sequence. Suppose to have the sequence ("a", "dog", "is"), we could create the following Markov Chain:



But, for instance, we would be interested in find $P(is|dog)$ but it's the probability of a specific verb ("is") given a very specific noun ("dog").

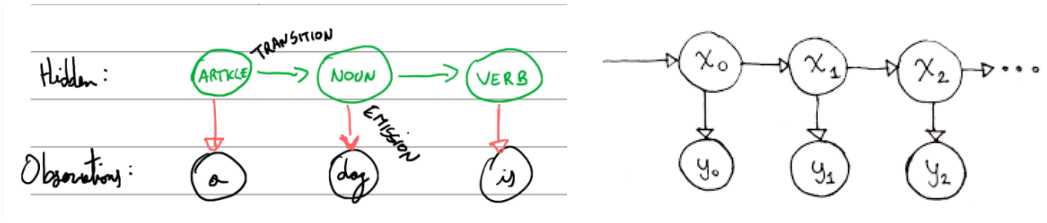
It would be more useful to generalize concepts from a sequence using some kind of category, for example:



Hidden Markov Models allow us to do exactly this.

Hidden Markov Models

An **Hidden Markov Model** is a particular Markov chain and it's a stochastic process where the transition between two nodes is dissociated from observations



It' composed of hidden states and observable states.
There are C hidden states and the state transition is an unobservable process

We store in a matrix A all the transition probabilities, in particular using indices we say that an item $a_{ij} \in A$ is the probability of moving from the state i to the state j.

Moreover, fixed an i, the sum of all the possible states must be 1:

$$\Sigma_j \; a_{ij} = 1 \qquad \forall i$$

The prior (initial) probability distribution is a vector π, where each item π_i is the probability that the Markov chain (behind the HMM) will start in the state i.

Then, there is the **emission distribution** which express the probability of an observation Y_t being generated from a state i.

$$b_i(y_t) = P(Y_t = y_t, S_t = i)$$

All the b_i can be pulled in a single matrix B.

So, the whole HMM is parametrized by $\theta = (\pi, A, B)$.

$$P(Y = y) = \sum_s P(Y = y, S = s)$$

$$= \sum_s P(S_1)P(Y_1|S_1) \prod_{t=2}^{T} P(S_t|S_{t-1}) \, P(Y_t|S_t)$$

Where

- Y are all the observation, so $P(Y = y)$ is the joint probability over all the observations
- $P(S_1)P(Y_1|S_1)$ is the **prior** distribution
- $P(S_t|S_{t-1})$ is the **transition** distribution
- $P(Y_t|S_t)$ is the **emission** distribution

Hidden Markov Models : 3 inference problems

Smoothing (Forward-Backward Algorithm)

Given a model $\theta = (\pi, A, B)$ and an observed sequence y, determinate the distribution of the hidden state at the time t:

$$P(S_t | Y_t = y, \theta)$$

Learning (EM Algorithm)

Given a dataset of N observed sequences $D = \{Y_1, ..., Y_N\}$ we want to find the parameters $\theta = (\pi, A, B)$ that maximize the probability of an HMM which has generated that sequence D.

Decoding : Optimal state assignment (Viterbi Algorithm)

Given a model $\theta = (\pi, A, B)$ and an observed sequence y find an optimal hidden state assignment $S = s^*_1, \dots, s^*_t$ for the HMM.

Smoothing | Forward-Backward Algorithm

Given a model $\theta = (\pi, A, B)$ and an observed sequence y, determinate the distribution of the hidden state at the time t.

$$P(S_t | Y_t = y, \theta)$$

"determinate the distribution of the hidden state at the time t " can be read as "hi, HMM, I give you this sequence y, tell me in which hidden state you will arrive".

We could resolve this task by calculating the sum of all the possible probabilities of the set of hidden variables, but this approach has an extremely exponential cost.
For this reason we introduce a dynamical programming algorithm called **Forward-Backward**.
"Dynamic programming" means that there is a table to store intermediate values during the execution.

We want to calculate the posterior distribution $P(S_t = i | Y_t = y, \theta)$ which is **proportional** to the joint probability of the $S_t = i$ and the sequence y.

(A/N: From now on, the parameters θ will be omitted for simplicity)

$$P(S_t = i \mid y) \propto P(S_t = i, y)$$

The sequence y goes from 1 to a big T: $y = \{y_1, \dots, y_T\}$ but we have a **fixed time point** $t \in [1, T]$ hence we could split the joint probability by this fixed time point:

$$\begin{aligned} P(S_t = i, y) &= P(S_t = i, y_1, \dots, y_t, \dots, y_T) \\ &= P(S_t = i, y_1 \dots, y_t)\ P(S_t = i, y_{t+1}, \dots, y_T) \\ \text{(product rule)} \quad &= P(S_t = i, y_{1:t})\ P(y_{t+1:T} | S_t = i) \end{aligned}$$

The first term $1 \rightarrow t$ can be calculated with the **forward** pass and we will refer to it as $\alpha_t(i)$.

The second term $t + 1 \leftarrow t$ can be calculated with the **backward** pass and we will refer to it as $\beta_t(i)$.

$$P(S_t = i \mid y) \propto P(S_t = i, y) = \alpha_t(i)\beta_t(i)$$

Forward pass : How to calculate α

Let's start from the definition of α:

$$\alpha_t(i) = P(S_t = i, y_{1:t})$$

where S_t is the t-th hidden state.

Now we can marginalize on all the values of the previous hidden state S_{t-1}:

$$\alpha_t(i) = P(S_t = i, y_{1:t}) \quad \text{(def)}$$

$$= \sum_{j=1}^{C} P(S_t = i, S_{t-1} = j, y_{1:t}) \quad \text{(marginalization)}$$

Using the Bayesian product rule we can isolate y_t:

$$= \sum_{j=1}^{C} P(y_t | S_t = i, S_{t-1} = j, y_{1:t-1}) \; P(S_t = i, S_{t-1} = j, y_{1:t-1})$$

Now, since we observed S_t, the $S_{t-1} = j, y_{1:t-1}$ term is no longer needed because it is d-separated.

$$= \sum_{j=1}^{C} P(y_t | S_t = i) \; P(S_t = i, S_{t-1} = j, y_{1:t-1})$$

Using the usual product rule on the second term, we can isolate $S_t = i$

$$= \sum_{j=1}^{C} P(y_t | S_t = i) \; P(S_t = i | S_{t-1} = j, y_{1:t-1}) P(S_{t-1} = j, y_{1:t-1})$$

And applying the d-separable idea on the $y_{1:t-1}$ we can rewrite the second term

$$= \sum_{j=1}^{C} P(y_t | S_t = i) \; P(S_t = i | S_{t-1} = j) \, P(S_{t-1} = j, y_{1:t-1})$$

Now we can notice that:

1) The first term is the **emission**
2) The second term is the **transition**
3) The third term is a **recursive** calculus on the previous $\alpha_{t-1}(i)$ by definition of the α itself.

So we can conclude the forward pass rewriting using the same notation of HMM:

$$\alpha_t(i) = \sum_{j=1}^{C} b_i(y_t) \quad A_{ij} \quad \alpha_{t-1}(j)$$

$$\alpha_1(i) = P(Y_1 | S_1 = j) \, P(S_1 = j)$$

Backward pass : How to calculate β

For the backward pass, the calculus are similar. We have to compute

$$\beta_t(i) = P(y_{t+1:T} | S_t = i)$$

By applying marginalization on all the possible value for the future state S_{t+1} we can get

$$\beta_t(i) = P(y_{t+1:T} | S_t = i) \qquad \text{(def)}$$

$$= \sum_{j=1}^{C} P(y_{t+1:T},\ S_{t+1} = j \mid S_t = i) \qquad \text{(marginalization)}$$

Again we can apply the product rule in order to isolate y_{t+1}

$$= \sum_{j=1}^{C} P(y_{t+1} \mid S_t = i,\ S_{t+1} = j, y_{t+2:T})\ P(S_t = i,\ S_{t+1} = j, y_{t+2:T})$$

Once we observed S_{t+1} we can elute the other two terms because they are d-separated

$$= \sum_{j=1}^{C} P(y_{t+1} \mid S_{t+1} = j)\ P(S_t = i,\ S_{t+1} = j, y_{t+2:T})$$

Again, we can isolate $S_{t+1} = j$ on the second term in order to obtain a transition distribution

$$= \sum_{j=1}^{C} P(y_{t+1} \mid S_{t+1} = j)\ P(S_{t+1} = j \mid S_t = i,\ y_{t+2:T})\ P(y_{t+2:T} \mid S_t = i)$$

Guess what we will do now:

spoiler: $y_{t+2:T}$ is d-separated

$$= \sum_{j=1}^{C} P(y_{t+1} \mid S_{t+1} = j)\ P(S_{t+1} = j \mid S_t = i)\ P(y_{t+2:T} \mid S_t = i)$$

Now we can notice that:

1) The first term is the **emission**
2) The second term is the **transition**
3) The third term is a **recursive** calculus on the future $\beta_{t+1}(i)$ by definition of the β itself.

So we can conclude the forward pass rewriting using the same notation of HMM:

$$\beta_T(i) = 1$$

$$\beta_t(i) = \sum_{j=1}^{C} b_i(y_{t+1})\ A_{ij}\ \beta_{t+1}(j)$$

Sum-Product message passing

The forward-backward algorithm is an example of a **sum-product message passing** algorithm where, there, the two messages have a generalized form for the forward message and the backward message.

Learning | Expectation-Maximization Algorithm

Given a dataset of N observed sequences $D = \{Y_1, ..., Y_N\}$ we want to find the parameters $\theta = (\pi, A, B)$ that maximizes the probability of an HMM which has generated that sequence D.

Another common task is to learn the parameters $\theta = (\pi, A, B)$ which are the one used by the **maximum likelihood** given a dataset of i.i.d. sequences.

The likelihood we want to maximize is the following:

$$\mathcal{L}(\theta) = log \prod_{n=1}^{n} P(Y^n \mid \theta) = to\ be\ continued...$$

It's a log-likelihood because we will work with exponential terms, but nothing changes.
Moreover, it's worth noting that **the likelihood of the model is the product of the likelihood of each single sequence** $Y_i \in D$, assuming that the dataset is i.i.d.

Recall how we marginalized over all the possible states in order to factorize the joint probability of a single sequence into the priori, emission and transition probabilities.

$$P(Y = y) = \sum_{s} P(Y = y, S = s)$$

$$= \sum_{s} P(S_1)P(Y_1|S_1) \prod_{t=2}^{T} P(S_t|S_{t-1})\, P(Y_t|S_t)$$

Now we can do the same within the productory $\prod_{n=1}^{n}$

$$\mathcal{L}(\theta) = log \prod_{n=1}^{n} P(Y^n \mid \theta) =$$

$$= log \prod_{n=1}^{n} \{ \sum_{S^n_1 \dots S^n_{T^n}} P(S^n_1)P(Y^n_1|S^n_1) \prod_{t=2}^{T} P(S^n_t|S^n_{t-1})\, P(Y^n_t|S^n_t) \}$$

That summary is a **huge** problem because it's a <u>combinatorial sum</u> inside an logarithm, since we are marginalizing on unobservable states.
<u>Maximizing this log-likelihood is **unfeasible!**</u>

So, instead of maximize this log-likelihood, we **<u>APPROXIMATE</u>** it using the **Expectation-Maximization Algorithm** which uses the **indicator variables** z^n_{ti}

An indicator variable is a binary variable so defined:

$$z^{n}_{t\,i} = \begin{cases} 1 & if\ S_t = i \\ 0 & otherwise \end{cases}$$

Complete HMM Likelihood

In particular, we will work maximizing the **COMPLETE Likelihood** $\mathcal{L}_C(\theta)$.

We can see the complete Likelihood as the Likelihood with the information that we are missing (the assignment of hidden states).

The EM algorithm works iterating the two steps and it stops until the convergence (between $\mathcal{L}_C(\theta)^{t}$ and $\mathcal{L}_C(\theta)^{t-1}$) was achieved or until a max number of iterations.

The complete Likelihood is so defined:

$$\mathcal{L}_C(\theta) = log \prod_{n=1} \{ \prod_{i=1}^{C} [P(S_1 = i)\ P(Y^n{}_1 | S_1 = i)]^{z^n{}_{1\,i}} \prod_{t=2}^{T_n} \prod_{i,j=1}^{C} \{ [P(S_t = i | S_{t-1} = i]^{z^n{}_{t\,i}\ z^n{}_{(t-1)\,j}} P(Y^n{}_t | S_t = i)^{z^n{}_{t\,i}} \}$$

PRIOR — FIRST TERM OF THE EMISSION — TRANSITION — REMAINING TERM OF EMISSION (STARTS FROM 2)

If you are still alive, you could notice that just a log of nested productories, so we could apply the properties of logarithms:

$$log(AB) = log(A) + log(B) \qquad log(A^n) = n\, log(A)$$

$$\mathcal{L}_C(\theta) = \sum_{n=1}^{n} \{ \sum_{i=1}^{C} z^n{}_{1\,i}\ log\ \pi_i + \sum_{t=2}^{T} \sum_{i,j=1}^{C} z^n{}_{t\,i}\ z^n{}_{(t-1)\,j}\ log\ A_{ij} + \sum_{t=1}^{T} \sum_{i,j=1}^{C} z^n{}_{1\,i}\ log\ b_j(y^n{}_t) \}$$

(prior) (transition) (emission)

That's nice because each term depends on only one "piece" [prior, transition, emission] so when we will differentiate it over one single piece the other will be detected (for example if we differentiate $\mathcal{L}_c(\theta)$ for the prior, the transition, and the emission parts will be constants so they will become zero.

The problem is that we don't know the value of z

For this reason we will use the Expectation, and since the **Expectation is a linear operator** we will apply on the three terms indepently on the z (due to the fact that they are constant within the expectation).

E-Step

We want to perform the Expectation over the z variables and we call Q the result of the Expectation of the complete likelihood. Spoiler: we will maximize this expectation in the M-Step.

$$Q^{t+1}(\theta|\theta^t) = \mathbb{E}_{z|y}[\mathcal{L}_c(\theta)]$$

In order to perform this expectation the non-trivial terms that we have are just the indicator variables, so let's focus on these.

In general, the expectation of a discrete Random Variable X is defined as

$$\mathbb{E}_X[X] = \Sigma_{x \in Domain(X)} x \cdot P(X = x)$$

In our case, the variables z are binary, so their domain is $\{0, 1\}$, so performing the expectations over $z|y$, is simply the probability of $P(z|y)$ because:

$$\mathbb{E}_{z|y}[z] = 0 \cdot P(z = 0|y) + 1 \cdot P(z = 1|y) = P(z|y)$$

Now we can apply the trick of the indicator variables, indeed, by the definition of indicator variables we know that $z_{ti} = 1 \Leftrightarrow S_t = i$

So:

$$\mathbb{E}_{z|y}[z_{ti}] = P(S_t = i|y)$$
$$\mathbb{E}_{z|y}[z_{ti} z_{(t-1)j}] = P(S_t = i, S_{t-1} = j|y)$$

But here we can connect the dots in our brain and observe that these **two probabilities are posteriors** and this means that **we compute them with the Forward-Backward algorithm !**
The results I will never remember are:

$$\gamma_t(i) = P(S_t = i|\boldsymbol{Y}) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{C} \alpha_t(j)\beta_t(j)}$$

$$\gamma_{t,t-1}(i,j) = P(S_t = i, S_{t-1} = j|\boldsymbol{Y}) = \frac{\alpha_{t-1}(j)A_{ij}b_i(y_t)\beta_t(i)}{\sum_{m,l=1}^{C} \alpha_{t-1}(m)A_{lm}b_l(y_t)\beta_t(l)}$$

M-Step

Once we obtained $Q^{t+1}(\theta|\theta^t)$ we can get the next parameter solving the optimization problem

$$\theta^{t+1} = \arg\max_{\theta} Q^{t+1}(\theta \mid \theta^t)$$

We can solve this problem by the gradient ascent, differentiation for each component of $\theta = (\pi, A, B)$

$$\frac{\delta Q^{t+1}(\theta|\theta^t)}{\theta} = \frac{\delta Q^{t+1}(\theta|\theta^t)}{\pi_i} \frac{\delta Q^{t+1}(\theta|\theta^t)}{A_{ij}} \frac{\delta Q^{t+1}(\theta|\theta^t)}{B_{k\,i}}$$

π, A and B are now variables and performing these derivatives we will obtain them.

But they are still distributions, so we need to force the values to respect the probabilities axioms.
For this reason we have to introduce the Lagrange multipliers, and the results are:

$$A_{ij} = \frac{\sum_{n=1}^{N}\sum_{t=2}^{T^n}\gamma_{t,t-1}^{n}(i,j)}{\sum_{n=1}^{N}\sum_{t=2}^{T^n}\gamma_{t-1}^{n}(j)} \quad \text{and} \quad \pi_i = \frac{\sum_{n=1}^{N}\gamma_1^n(i)}{N}$$

$$B_{ki} = \frac{\sum_{n=1}^{N}\sum_{t=1}^{T_n}\gamma_t^n(i)\delta(y_t = k)}{\sum_{n=1}^{N}\sum_{t=1}^{T_n}\gamma_t^n(i)}$$

EM: Conclusion

The main idea of these two steps is to find an approximation of the complete Likelihood, finding the expectation first and use that expectation to get the parameters.

Decoding | Viterbi Algorithm

Given a model $\theta = (\pi, A, B)$ and an observed sequence y find an optimal hidden state assignment $S = s^*_1, \dots, s^*_t$ for the HMM.

Let $y = y_1 \dots y_t$ the observations and $s = s_1 \dots s_t$ the hidden states. We want to solve

$$\max_s P(y, s) =$$

We can rewrite this problem using the usual prior-transition-emission factorization of HMMs. Note: for simplicity, the prior is included into the transition probability.

$$\max_s P(y, s) = \max_s \prod_{t=1}^{T} P(Y_t | S_t) P(S_t | S_{t-1})$$

So, since $s = s_1 \dots s_t$ we can focus only on the last item : s_t

$$[s_t]: \quad \max_{s_t} P(y, s) = \max_{s_t} \prod_{t=1}^{T} P(Y_t | S_t) P(S_t | S_{t-1})$$

We can see this product as the product from 1 to $N - 1$ multiplied for the max value of the last item s_t (note the $N - 1$ in the productory):

$$[s_t]: \quad = \prod_{t=1}^{T-1} P(Y_t | S_t) P(S_t | S_{t-1}) \max_{s_t} P(Y_t | S_t) P(S_t | S_{t-1})$$

Backward pass

After we did split the product, we have to perform the $\max_{s_t}$ term for each term of s_{t-1}.

In practice, we can store these values in a vector of C elements and we denote it as $\epsilon(S_{t-1})$.

$$[s_t]: \quad = \prod_{t=1}^{T-1} P(Y_t | S_t) P(S_t | S_{t-1}) \epsilon(S_{t-1})$$

We started this calculus from the **last** item in the sequence s. Now we can do the same maximizing s_{t-1}:

$$[s_{t-1}]: \quad \max_{s_{t-1}} P(y, s) = \prod_{t=1}^{T-1} P(Y_t | S_t) P(S_t | S_{t-1}) \epsilon(S_{t-2})$$

This means that we can solve the original maximization problem $\max_s P(y, s)$ starting from backwards, compute the vector ϵ and pass it to the predecessor, so we just need to calculate ϵ_{ACTUAL} using ϵ_{PREV} and going backward, using recursion.

$$\epsilon(S_{t-1}) = \max_{s_t} P(Y_t | S_t) P(S_t | S_{t-1}) \epsilon(S_t)$$

Forward pass

The backward recursion concludes in the root, and there we can finally find our s^* applying the forward pass, where at each iteration t we find the optimal state S_t using $\epsilon_T(s)$ found from the backward pass:

$$s^*_t = \arg\max_s P(Y_t|S_t)\, P(S_t|S_{t-1} = s_{t-1})\, \epsilon(s)$$

Dynamical Bayesian Networks

HMMs are a specific class of directed models that represent dynamical processes which means graphical models whose structure changes with respect to the time of different samples.

Markov Random Fields

A **Markov Random Field** is an **undirected graph** whose nodes are random variables and they can be observed or unobserved.
The joint probability of the X is defined as:

$$P(X) = 1/Z \prod_c \Psi_c(X_c)$$

where:

- X_c is the set of the nodes that belong to the **maximal clique C**
- $\Psi_c(X_c)$ is the **potential function**
- Z is the term for normalization (**partition function**)

$$Z = \sum_x \prod_c \Psi_c(X_c)$$

Potential Functions

In general, potential functions are not probabilities, they **just express a preference over variables.**
If we assume to use only strictly positive potential functions, there is a theorem which assures us that the output distribution can be represented by the clique factorization.
So, **the choice of the potential function is up to us** and we use the (widely used) Boltzmann distribution that is strictly positive.

$$\Psi_c(X_c) = \exp(-E(X_c))$$

where $E(\cdot)$ is the **energy** function.

Factorizing Potential Functions

Now we can move to the MRF scenario where we use the following potential function:

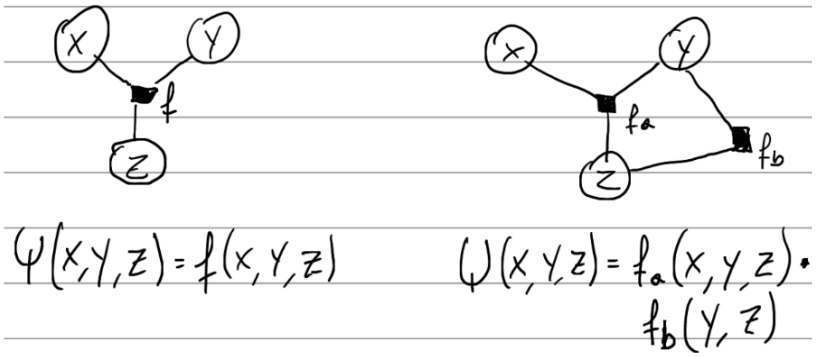
$$\Psi_c(X_c) = \exp(-\sum_{k=1}^{K} \theta_{c_k} * f_{c_k}(X_c))$$

where:

- $f_{c_k}(X_c)$ are the **feature functions**: **functions that represent constraints between RV.**
- K is the number of the feature function
- $\theta_{c_k}(X_c) \in \Re$ are parameters
- $X_c \in$ *to the clique C*

Factor Graphs

We can factorize the potential function of MRF using **feature functions**.
Classical undirected graphical models cannot express these feature functions (which are constraints among RV).
For this reason we introduce **Factor graph**



The feature functions are represented as squares.

Sum Product Inference

Factor graph is an **unified formalism** for both directed and undirected models because they can express feature functions even in directed models.
Inference is feasible for each graphical model that has a chain or a tree structure.

Now we will see how to constrain a MRF to obtain tractable classes of undirected models.

CRF: Restricting to Conditional Probabilities

So far, we were always calculating joint probabilities $P(X, Y)$ but in real case scenarios we are interested in learning hidden RV from a given and always visible inputs (the usual dataset). So we want to restrict our model to $P(Y|X)$.

We can define:

- X_k as the **observable** inputs for the factor k
- Y_k as the **hidden**[1] Random Variable for the factor k
- $f_k(X_k, Y_K)$ as the factor feature

Under this assumption we can write the conditional distribution as:

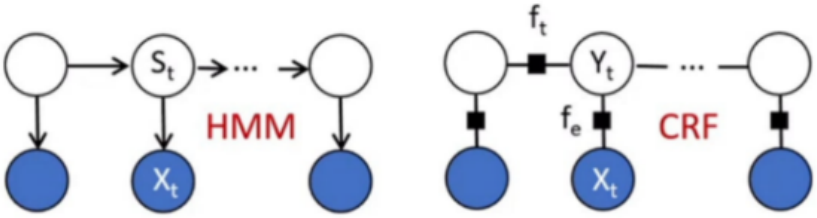
$$P(Y|X,\theta) = \frac{1}{Z(X)} \prod_k \exp\{\theta_k \cdot f_k(X_k, Y_k)\}$$

where $Z(X)$ is needed for normalization (and here we marginalize only over y)

$$Z(X) = \Sigma_y \Pi_k \exp\{\theta_k f_k(X_k, Y_k = y_k)\}$$

[1] Di solito, le Y sono le osservazioni e X gli hidden, qui è il contrario :)

CRF as HMM generalized

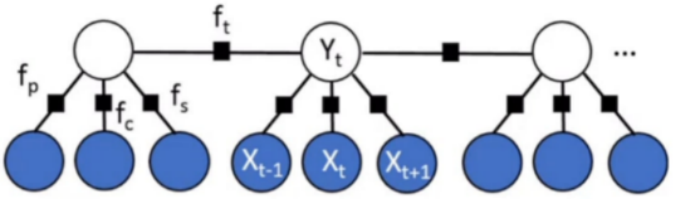


(Linear) Conditional Random Field can be seen as the generalized version of a HMM.
Indeed, a generic factor graph f_t which connect y_{t-1} with y_t could be compared with $P(S_t|S_{t-1})$ in HMMs.

Even the emission functions $P(X_t|S_t)$ could be compared to $f_e(X_t, Y_t)$ in the figure.

But, **the two models are intrinsically different** because HMMs must use a **single** probability among two RVs, instead of (L)CRF could have several factor graphs among two RVs, because **factor graphs are just functions and so we can combine them linearly**.

Another main difference between these two models is that **HMMs**, using probability, **cannot recognize any intrinsic feature of an observed input.**
For example if X_t has a value that the HMM has never seen, it will have a probability of zero, even if it's a valid value. **Instead, CRF can recognize unknown values by the intrinsic features.**



Moreover, **CRFs are able to use information from previous and future steps** in order to better understand the actual context.

LCRF Likelihood

In general the likelihood of a LCRF is

$$P(Y|X,\theta) = \frac{1}{Z(X)} \prod_t \exp\{\theta_k \cdot f_k(Y_t, Y_{t-1}, X_t)\}$$

Our model takes into account what is the current hidden state Y_t, the past hidden state Y_{t-1} and some observation X_t.

Posterior Inference in LCRF (Smoothing in LCRF)

Recall that we mean for smoothing the task of calculating the probability to be in a state S_t given a full sequence y, so we want to calculate the posterior $P(S_t|y)$

We can extend this task of HMMs to LCRF, calculating the posterior $P(Y_t, Y_{t-1}|X)$.

Similarly to HMM,we solve this problem with the Forward-Backward algorithm but we extend the sum-product message with the LCRF factor graph Ψ, so:

$$P(Y_t, Y_{t-1}|X) \propto \alpha_{t-1}(Y_{t-1})\, \Psi_t(Y_t, Y_{t-1}, X_t)\, \beta_t(Y_t)$$

Where

- Forward message:
$$\alpha_{t-1}(Y_{t-1}) = \Sigma_j \Psi_t(i, j, X_j)\, \alpha_{t-1}(j) \qquad \text{(we marginalize over the past)}$$
- Clique Weighting: (bilancia tra α e β)
$$\Psi_t(Y_t, Y_{t-1}, X_t) = \exp\{\theta_e f_e(X_t, Y_t) + \theta_t f_t(Y_{t-1}, Y_t)\}$$
- Backward message:
$$\beta_t(j) = \Sigma_j \Psi_{t+1}(i, j, X_{t+1})\, \beta_{t+1}(j) \qquad \text{(we marginalize over the future)}$$

Viterbi in LCRF

After the smoothing inference problem, we can apply the Viterbi algorithm to LCRF.
Compute the max-product inference (Viterbi) is problematic because now we have a more articulated posterior (not just two RV as before in HMMs) and this create problems with the **partition function Z(X)**, which is the sum over every RV which is not observable.

If the CRF does not have a chain structure we cannot perform the exact inference because it's computationally impracticable, so we just need to approximate this inference and we can do it with Monte-Carlo methods (for example).

Training LCRF

The last inference problem is about learning the parameters, as usual we are looking for parameters θ that maximize the likelihood $\mathcal{L}(\theta)$.
The maximization is made by differentiation:

$$\max_{\theta} \mathcal{L}(\theta) = \max_{\theta} \sum_{n=1}^{N} log\, P(Y^n|X^n, \theta)$$

where

$$\mathcal{L}(\theta) = \sum_n \sum_t \sum_k \theta_k f_k(Y_t^n, Y_{t+1}^n, X_t^n) - \sum_n log\, Z(X^n) - \sum_k \frac{\theta_k^2}{2\sigma^2}$$

where:

$$Z(X^n) = \sum_{y_t, y_{t-1}} \sum_t \exp\{ \sum_k \theta_k f_k (Y_t^n, Y_{t+1}^n, X_t^n) \}$$

Don't worry now about the last regularization term and focus on the partial derivative:

$$\frac{\delta \mathcal{L}(\theta)}{\delta \theta_k} = \sum_n \sum_t \theta_k f_k(Y_t^n, Y_{t+1}^n, X_t^n) \sum_n \sum_t \sum_{y, y'} f_k(y, y', X_t^n) P(y, y'|X_t^n) - \frac{\theta_k}{\sigma^2}$$

[This paragraph may be wrong] this result is achieved because we are deriving with respect to θ_k so every summatory over k simply disappears and the $P(Y, Y'|X^n)$ appears due to the derivation of $Z(\cdot)$:

$$\frac{\delta\, log\, Z(X)}{\delta \theta_k} = \frac{1}{Z(X)} \cdot \frac{\delta\, Z(X)}{\delta \theta_k}$$

$$= \frac{1}{Z(X)} \cdot \sum_{y_t, y_{t-1}} \sum_t \exp\{ \sum_k \theta_k f_k (y_t, y_{t-1}, X_t^n)\}\, f_k (y_t, y_{t-1}, X_t^n)$$

But $\frac{1}{Z(X)} \cdot \sum_{y_t, y_{t-1}} \sum_t \exp\{ \sum_k \theta_k f_k (y_t, y_{t-1}, X_t^n)\}$ can be seen as the $P(y, y'|X_t^n)$, in this way we achieved that value for $\frac{\delta \mathcal{L}(\theta)}{\delta \theta_k}$.

We can apply the definition of $\mathbb{E}$ on the last term, so we can rewrite:

$$\frac{\delta \mathcal{L}(\theta)}{\delta \theta_k} = \sum_n \sum_t \theta_k f_k(Y_t^n, Y_{t+1}^n, X_t^n)\, \mathbb{E}_{P(Y|X,\theta)}[\, f_k(y, y', X_t^n)\,] - \frac{\theta_k}{\sigma^2}$$

Even the first term is a very dump expectation : the one when y and y' are taken from the dataset (the empirical distribution), so:

$$\frac{\delta \mathcal{L}(\theta)}{\delta \theta_k} = \mathbb{E}_{y,y' \sim D} f_k(y, y', X_t^n)] \; \mathbb{E}_{P(Y|X,\theta)} [\; f_k(y, y', X_t^n) \;] - \frac{\theta_k}{\sigma^2}$$

Important comments

This means that in order to maximize the parameters we have to match these two expectations: the first one is given from the dataset and the second one is how the model "thinks" the world. So, if we match these two observations we obtain a model that has a consistent internal view of the world.

...but we forgot the regularization term!

Let's return to our original likelihood:

$$\mathcal{L}(\theta) = \sum_n \sum_t \sum_k \theta_k f_k(Y_t^n, Y_{t+1}^n, X_t^n) - \sum_n log\, Z(X^n) - \sum_k \frac{\theta_k^2}{2\sigma^2}$$

The last term penalizes the model when it uses too much the θ parameter.
It derives from the fact that we can use the Maximum A-Posteriori (MAP) to regularize likelihoods; in this case the value is obtained by deriving a Gaussian distribution with mean zero and variance as parameter.
So, to conclude, θ parameters are learned using SGD.

Bayesian Learning

Latent Variable Models

Let X be a set of observations $\{x_1, .. , x_D\}$ with D very large, so my data are a lot and then we cannot perform the joint probability of all these observations (because there are too many).

Hence, we split the complex joint probability into a product of simpler joint probabilities, using **Latent Variables Models**, adding (through marginalization) latent variables and then we can work with simpler stuff.

We can define latent variable models as a class of generative models for which variational or approximation methods are needed to explain complex relations between hidden variables.

Latent Variables

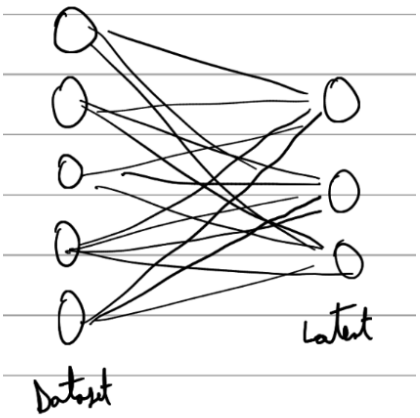
Latent variables are unobserved random variables that define an hidden generative process of observed data (e.g., hidden variables in HMM/CRF)

The latent variables can be continuous and so the likelihood is

$$P(X) = \int_z \prod_{i=1}^{D} P(x_i|z)\, P(z)\ dz$$

Latent Space

Given a large dataset, we can use the **latent space** to reduce the high-dimensionality of the dataset.



The assumption is that the latent variables conditional and marginal distribution are more tractable than the joint distribution $P(X)$
Now we can explain the reason of this chapter: **we want to approximate the likelihood.**

KL Divergence

We need a measure to approximate probabilities.
A possible metric is the **KL Divergence** and **it measures how much a distribution q is close to a distribution p**, and it's defined as:

$$KL(q||p) = \mathbb{E}_q[log \frac{q(z)}{p(z|x)}]$$

Since, it's a difference in expectations we can rewrite it:

$$KL(q||p) = \langle log\, q(z) \rangle_q - \langle log\, P(z|x) \rangle_q$$

where $\langle \cdot \rangle_q$ is just another way to denote $\mathbb{E}_q$.

We are interested to this value only when q is "high" and if even p is high we are happy, if p is low we are sad because we have a large divergence.
If q is low, we don't care.

Jansen Inequality

It's a property of linear operators on convex or concave functions:

$$f(\mathbb{E}[X]) \geq \mathbb{E}[f(X)]$$

Bounding Log-Likelihood with Jensen

We want to maximize the log-likelihood and typically we cannot manipulate the probability unless we introduce an hidden variable, so we have:

$$log\, P(x|\theta) = log \int_z P(x,z|\theta)\, dz$$

we can add, without loss of generality, a distribution $Q(z|\phi)$

$$= log \int_z \frac{Q(z|\phi)}{Q(z|\phi)} P(x,z|\theta)\, dz$$

Note that we have two set of parameters: θ and ϕ.
Adding the distribution $Q(z|\phi)$, now we can apply the definition of expectation

$$= log\ \mathbb{E}_q[\frac{P(x,z|\theta)}{Q(z|\phi)}]$$

Now, we can apply the Jensen inequality:

$$log\ \mathbb{E}_q[\frac{P(x,z|\theta)}{Q(z|\phi)}] \geq \mathbb{E}_q[log \frac{P(x,z|\theta)}{Q(z|\phi)}]$$

Let's focus on the second term:

$$\mathbb{E}_q[log \frac{P(x,z|\theta)}{Q(z|\phi)}] = \mathbb{E}_q[log\, P(x,z|\theta) - log\, Q(z|\phi)]$$
$$= \mathbb{E}_q[log\, P(x,z|\theta)] - \mathbb{E}_q[log\, Q(z|\phi)]$$

expectation of joint distribution *entropy*

We call this difference of expectations **ELBO (EVIDENCE LOWER BOUND)** which is an useful lower bound on the log-likelihood of some observed data, and it's denoted as

$$= \mathcal{L}(X, \theta, \phi)$$

So, to conclude:

$$log\, P(x|\theta) \geq \mathcal{L}(X, \theta, \phi)$$

ELBO: How good is this lower bound?

We want to be sure about the fact that the ELBO $\mathcal{L}(X, \theta, \phi)$ is a good approximation of the log-likelihood $log\, P(x|\theta)$.

Recall :

$$\mathcal{L}(X, \theta, \phi) = \mathbb{E}_q[\, log \frac{P(x, z|\theta)}{Q(z|\phi)}\,]$$

we can use exploit the definition of expectation, so:

$$\mathcal{L}(X, \theta, \phi) = \mathbb{E}_q[\, log \frac{P(x, z|\theta)}{Q(z|\phi)}\,] = \int_z Q(z|\phi)\, log \frac{P(x, z|\theta)}{Q(z|\phi)} dz$$

The question "how good is the lower bound means" that we want to calculate the following difference:

$$log\, P(x|\theta) - \mathcal{L}(X, \theta, \phi) = ???$$

Applying the exploitation of the definition of expectation as seen in the recall part:

$$log\, P(x|\theta) - \mathcal{L}(X, \theta, \phi) = log\, P(x|\theta) - \int_z Q(z|\phi)\ log \frac{P(x, z|\theta)}{Q(z|\phi)} dz$$

Now we can add by marginalization $Q(z|\phi)$ on the left term

$$= \int_z Q(z|\phi)\, log\, P(x|\theta)\, dz - \int_z Q(z|\phi)\, log \frac{P(x, z|\theta)}{Q(z|\phi)} dz$$

Since $log\, A - log\, B = log\, A/B$:

$$= \int_z Q(z|\phi)\, log \frac{P(x|\theta)\, Q(z|\phi)}{P(x, z|\,\theta)} dz$$

Reapplying the def. of expectation:

$$= \mathbb{E}_q[\, \frac{P(x|\theta)\, Q(z|\phi)}{P(x, z|\,\theta)}]$$

Since $\frac{P(x|\theta)}{P(x, z|\,\theta)} = \frac{1}{P(z|x)}$

$$= \mathbb{E}_q[\, \frac{Q(z|\phi)}{P(z|\,x, \theta)}]$$

We conclude applying the def. of KL:

$$log\ P(x|\theta)\ -\ \mathcal{L}\ (X,\theta,\phi)\ \ =\ \ KL[\ Q(z|\phi)\ ||\ P(z|\ x,\theta\ )\ ]$$

This means that the goodness of the ELBO to approximate the likelihood $P(x|\theta)$ depends only by the choice of the Q distribution.

Moreover, we can learn our model in two ways:

1) Maximizing the lower bound
2) Minimizing the KL, indeed a perfect Q will have $KL(Q||P)\ =\ 0$

If the posterior itself $P(z|x,\theta)$ is tractable, we can use it as Q, in that case we have the **optimal ELBO**.
Otherwise, we have to choose a tractable Q maximizing the $KL(Q||P)$ or finding the parameters ϕ which maximize the Evidence Lower BOund $\mathcal{L}\ (X,\theta,\phi)$.

EM Learning reformulated

All the previous ideas can be used to rethink the EM because, in order to obtain the better parameters in the learning inference task, we can maximize the ELBO.

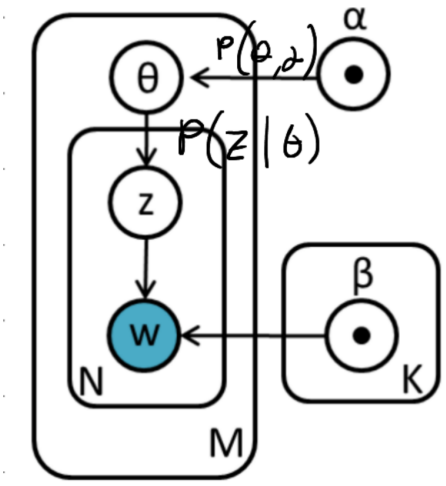
Bag of words

Let's introduce some concepts needed to explain an application of the Latent variables.
The **bag of words (BOW)** representations are a classical example of **multinomial data.**
We can see a BOW as a document matrix $N\ \times\ M$
where

N is the number of vocabulary items
M is the number of document

Often we have a huge number of documents and we want to summarize these documents with a smaller representation using a **mixture of topics**, where for "topic" we mean a pattern into the items that belong to that document.
Applying these topics to documents can be performed using Latent Variables.

Latent Dirichlet Allocation



This model assigns a topic z to each item w with a **multinomial item topic distribution**

$$P(w|z,\beta)$$

In particular, it chooses the topic for the whole document with a probability

$$P(z|\theta)$$

Each document has a topic parameter θ (which can be a RV) sampled from a distribution.

$P(\theta|\alpha)$ is given by the **Dirichlet distribution** with α as hyperparameter.

The beta hyperparameter controls the distribution of words per topic. Turn it down, and the topics will likely have less words. Turn it up, and the topics will likely have more words.

Dirichlet Distribution

We use this particular distribution because we can interpret it as a way to introduce a prior distribution to a multinomial one.
If the likelihood is multinomial, it's guaranteed that using a Dirichlet Prior, the posterior will be Dirichlet itself.

This distribution uses a hyperparameter α to set the sparsity level.

LDA Generative Process

LDA can be even used to generate words with respect to a topic and the process is similar to the "classification" one.

For each document into the BOW, we choose the θ parameter from the Dirichlet distribution (using α as parameter).
Then, for each word within the fixed document we choose a topic z from a multinomial distribution using the θ parameter found before and now we can assign a probability $P(w|z,\beta)$ to the topic.

Learning in LDA

The likelihood of a document d is given by the multiplication of the probability of each word w_i given α and β:

$$P(w|\alpha,\ \beta)\ = \int \sum_{z} P(\theta, z, w|\alpha, \beta)\ d\theta$$

Expanding, we obtain:

$$= \int P(\theta|\alpha) \prod_{j=1}^{N} \sum_{z_j=1}^{k} P(z_j|\theta) P(w_j|z_j, \beta) \quad d\theta$$

Given the set of words $w_1 \ldots\ w_M$ we want to find α and β that maximizes the formula above.

But we have a huge problem: we cannot infer latent posterior variables because the optimal ELBO is achieved when $Q(z)$ is equal to the latent variable posterior and this posterior is not tractable due to a combinatorial sum over all the possible couples between β and θ.

So, we can pick a different Q function which helps us to approximate it.

Approximating parameters

Since the exact inference of parameters in LDA is not tractable, we have to approximate it.
We can perform this with two ways:

1) **Variational Inference**
   This technique maximize the variational bound without the optimal posterior solution, finding a $Q(z, \phi)$ that is **similar to the posterior, but tractable.**
   We will fit the ϕ parameter such that $Q(z, \phi)$ will be close to $P(w|\alpha, \beta)$ using KL as metric.
   It's fast but it's an approximation.

   A/N: look back to "optimal ELBO" to connect dots within your brain

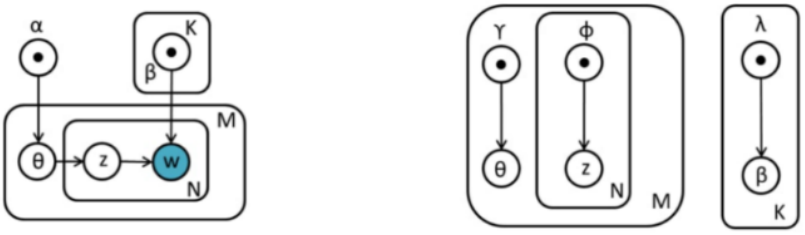
2) **Sampling Approach**

Variational Inference (Variational EM)

This is the first method to approximate LD parameters.
We assume that the $Q(z, \phi)$ distribution is tractable: this is the **mean-field assumption** and it means that we assume that the joint probability of our hidden variables can be expressed as independent products:

$$Q(z, \phi) = Q(z_1 \dots z_k | \phi) = \prod_{k=1}^{K} Q(z_k | \phi_k)$$

This technique can be made more general using groups of hidden variables (instead of single z_i).
The variational inference uses three parameters $\{\gamma, \phi, \lambda\}$ and we use the posterior to approximate the LDA parameters.



The model for variational inference, so, uses these three parameters $\{\gamma, \phi, \lambda\}$ to estimate respectively $\{\theta, z, \beta\}$.
As we can see from the figure on the right, the model approximates $\{\theta, z, \beta\}$ independently because there is no link between them.

Note: β are the parameter of the multinomial distribution which can be excluded from the variational approximation (because they are no hidden variables)

$$Q(z, \phi, \beta) = Q(\theta | \gamma) \prod_{n=1}^{N} Q(z | \phi) \prod_{k=1}^{K} Q(b_k | \lambda_k)$$

A/N: in order to understand and remember this formula look at the Markov blankets of the nodes in the figure on the right.

Hence we have:

- $\Psi = \{\alpha\, \beta\}$ (model parameter)
- $\Phi = \{\gamma, \phi, \lambda\}$ (variational approximation parameters)

we would find the values for Φ,Ψ that maximize the ELBO

$$\mathcal{L}(w) = \mathbb{E}_Q[\, log\, P(\theta, z, w | \Psi)\,] - \mathbb{E}_Q[\, log\, Q(\theta, z, \psi | \Phi)\,]$$

The maximization is performed by the alternate maximization technique (such as the EM algorithm): we fix a parameter and update the other parameters, alternatively:

1) [E-Step] Fixing the model parameter Ψ, estimate an approximation of the posterior Φ
2) [M-Step] Fixing the posterior Φ, update the model parameter Ψ

Unlike EM, variational EM has no guarantee to reach a local maximizer of the likelihood.

Sampling

The sampling is a technique which consists of drawing a set of realizations of a random variable X with a distribution $P(X)$
The set of realization is denoted with $X = \{x^1 \dots x^L\}$

Example: if we roll a dice five times we could have this set $\{5, 3, 2, 3, 1\}$

Sampling is important because we can approximate the distribution $p(x)$ using the empirical distribution.

$$P(X) \approx 1/L \, \Sigma^{L}_{l=1} \mathbf{1}[x^l = i]$$

with sampling, we can approximate even the expectation of a function:

$$\mathbb{E}[\, f(x) \,] \approx 1/L \, \Sigma^{L}_{l=1} f(x^l)$$

We use sampling when $P(X)$ is intractable.

Sampling for learning

In a Bayesian model, parameters are random variables.
Since their posterior is usually intractable, this is a perfect use case for sampling.
Hence, we can use one sampling technique to obtain the model parameters.

For example, we can obtain model parameters of a LDA, sampling the intractable posterior.

Sampling procedures

A Sampler S is a procedure that generates a sample set X from a generic distribution $P(X)$
The sample set X, since it contains the samples, has a probability distribution $\widetilde{P}_s(X)$, and that distribution defines the whole sampler.

Recall that $P(X)$ is intractable and usually $\widetilde{P}_s(X) \neq P(X)$

Deeping : Unbiased estimator

An unbiased estimator of a parameter θ is an estimator whose expected value is equal to the parameter θ itself.
Let S be an estimator and θ the parameter to estimate.
We define S as **unbiased** if

$$\mathbb{E}[\, S \,] = \theta$$

If S is unbiased, if draw a lot of samples from S and then we perform an average, we simply obtain θ; so we would have unbiased estimators.

If S is unbiased we call it **VALID** since it draws samples from the desired distribution.

Properties of a sampler: unbiased

Let's focus on the sampling approximation of expectation:

$$\mathbb{E}[\, f(x)\,] \approx 1/L\, \Sigma^{L}_{l=1} f(x^{l}) = \hat{f}(x)$$

$\hat{f}(x)$ estimates values, we could ask if it is an unbiased estimator for the parameter $\mathbb{E}_{p(X)}[f(x)\,]$, so let $\tilde{p}(x)$ be the distribution over sampling realization (as usual) and by the definition of unbiased estimator we have:

$$\mathbb{E}_{\tilde{p}(X)}[\hat{f}(x)\,] = \mathbb{E}_{p(X)}[f(x)\,]$$

this is true if $\tilde{p}(x^{l}) = p(x)$

Properties of a sampler: variance

The variance of f(x) tells us how much we can trust the approximation computed using X as sampling set.

If the variance is low, $\hat{f}(x)$ is close to its expected value (a good scenario) and so, if the variance is low, a small number of samples is required to have a good estimation of the expectation.

The variance of f(x) is defined as:

$$\mathbb{E}_{\tilde{p}(x)}[\,(\hat{f}(x) - \mathbb{E}_{\tilde{p}(x)}[\,\hat{f}(x)\,])^{2}\,]$$

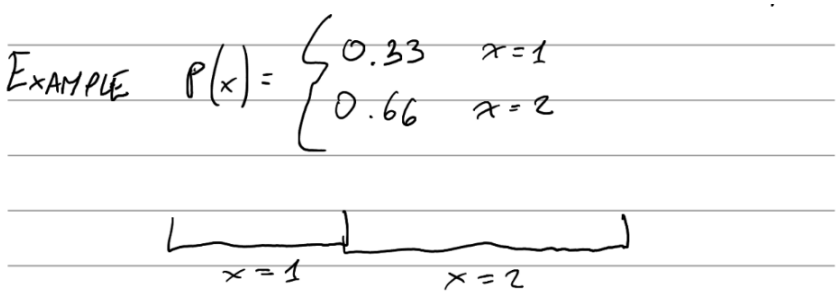
From this definition, thought some calculus, we can obtain that is equal to

$$\mathbb{E}_{\tilde{p}(x)}[\,(\hat{f}(x) - \mathbb{E}_{\tilde{p}(x)}[\,\hat{f}(x)\,])^{2}\,] = 1/L\, Var(f(x))$$

This means that if we sample a lot of times we reduce the variance (due to the $1/L$)

Univariate Sampling

Drawing samples from an univariate distribution is easy, we just need to get a random uniform number between 0 and 1 and the probability distribution $P(X)$.
$P(X)$ defines ranges and the random number select a single value looking the ranges
Example:




```
if (0 < random number < 0.33){
        return 1
}else{
        return 2
}
```

Multivariate Sampling

In the multivariate case, $P(X)$ represents the joint distribution of a set of variables.
We can sample from $P(X)$ building an univariate distribution $P(S)$ where S is the set of all the possible combinations of the states of a Random Variable.
Then we can sample from $P(S)$ using the univariate idea.

If a random variable has C states and we have n RV, $P(S)$ have to consider C^n states and this could become easily infeasible.

Naive Multivariate Sampling

So, we could use the chain rule to rewrite the joint distribution as:

$$P(S_1 \dots S_n) = P(S_1)P(S_2|S_1)P(S_3|S_2, S_1)$$

Then we could sample from $P(X)$ in the multivariate scenario, following the chain rule.

It would be easy because we just have a univariate distribution $P(S_n|\tilde{S}_1 \dots \tilde{S}_{n-1})$ where $\tilde{S}_i$ is obtained backwards starting from $\tilde{S}_1 \sim P(S_1)$.

But here we have an exponential number of states that need to be summed up.

Ancestral Sampling

If the distribution $P(S_1 \dots S_n)$ is represented by a Bayesian Network we can use the Bayesian Ancestral Order as a sampling order.
We will start from the roots, then we will pick the children and so on.

This technique is a valid sampler because each value is drawn from $P(X)$ and it has low variance because samples are independent.

Gibbs Sampling: why we need it

Suppose to have a set of RV S and only a subset $S_\epsilon \subset S$ of them are visible.

In order to use ancestral sampling we could change the structure of the BN, but it would require too much time or we could discard non-visible items, but in this latter case we lost a lot of samples.

For this reason we need an efficient method to sample under evidence: **Gibbs Sampling**

Gibbs Sampling

The main idea behind the Gibbs sampling is to start from a sample $X = \{S_1^1 .. S_n^1\}$ and update only one variable at the time

Sample	s_1	s_2	s_3	s_4	s_5
x^1	1	1	2	4	5
x^2	3	1	2	4	5
x^3	3	4	2	4	5
x^4	3	4	2	1	5
x^5	3	4	6	1	5
⋮	⋮	⋮	⋮	⋮	⋮

Given the $(l+1)$-th iteration, we select a variable then we sample its value according to its Markov Blanket

$$s_j^{p+1} \sim P(s_j | s_{\setminus j})$$

The distribution $P(s_j | s_{\setminus j})$ is simple to compute, because depends only on the Markov Blanket of s_j

Gibbs Sampling for LDA

We saw that LDA parameters are difficult to obtain due to the intractable probability.
For this reason we could use the Gibbs Sampling to obtain the parameters: starting from an initial guess $\{z^1, \theta^1, \alpha^1, \beta^1\}$ and then we can update each parameter looking to the Markov Blanket of that value (e.g. θ depends only on z and α)

p(x) of Gibbs Sampling

Let $X = \{x^1 \dots x^l\}$ be a set of samples obtained using Gibbs. **Each sample is obtained from the previous sample.**
We can define a probability distribution $q(x^{l+1}|x^l)$ which tell us the probability to obtain the sample x^{l+1} given x^l.
Then, we can this q to compute $\tilde{p}(x)$

$$\tilde{p}(x) = \prod_{l=1}^{L} q(x^{l+1} \mid x^l)$$

Properties of Gibbs Sampling (Problems)

1) **Gibbs Sampling is not a valid sampler**, which means that it could have some bias, but however if we compute the Gibbs a sufficient number of times, the algorithm converges to samples taken from $P(X)$, so it became valid (and we can discard first samples)

2) The main problem is that a **Gibbs sampler has no low variance** because each sample depends on the previous.

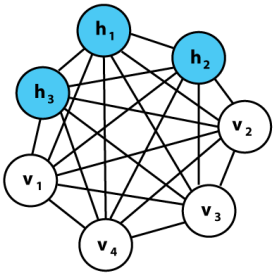
MCMC Sampling Framework

The Gibbs sampling is a specialization of the Markov Chain Monte Carlo (MCMC) sampling framework which rely on the idea of building a Markov Chain whose stationary distribution is $p(x)$.
This Markov Chain need to respect two properties:

1) **Irreducible**, meaning it's possible to reach any state from anywhere
2) **Aperiodic**, meaning at each time-step we can be anywhere

So, the state transition of the chain, $q(x^{l+1}|x^l)$, converges to $p(x)$.

Boltzmann Machines



Boltzmann machines are an example of **a Markov Random Field** (so an undirected graph) with **binary nodes**.
The variables can be **hidden** $h \in \{0, 1\}$ or **visible** $v \in \{0, 1\}$.
The set of all the nodes is called $s = [hv]$
Being a MRF, we define a linear **energy function** as

$$E(s) = -\frac{1}{2}\sum_{ij} M_{ij} s_i s_j - \sum_j b_j s_j = -\frac{1}{2} s^T M s - b^T s$$

BIAS
VETTORIAL POV
WEIGHTS
FEATURE FUNCTION → CORRELATION BETWEEN TWO RV (PRODUCT OF BINARIES)

So, our model parameters θ are $\{M, b\}$, weights and bias.
They encode the iterations between variables.

BM as NN

Boltzmann machines can be seen as a NN; in particular a NN whose activation is determined by a stochastic function.
For "stochastic" function think about a threshold function like:

if input ≤ *threshold*
    return 0
else
    return 1

but the threshold is **randomly picked**, and the network learns these thresholds (which are a probability distribution) from training data.

Stochastic Binary Neurons

At a given time t, we have for each input a probability p_j^t which tells us if the unit is fired (1) or not (0). These are the thresholds we talked about before. These probabilities replace the activation functions of the neurons.

$$s_j^{(t)} = \begin{cases} 1, \text{with probability} & p_j^{(t)} \\ 0, \text{with probability } 1 - p_j^{(t)} \end{cases}$$

So, a unit x_j^t is a weighted sum, as usual:

$$x_j^{t+1} = \sum_{i=1}^{N} M_{ij} S_i^t + b_j$$

where: M is the weight matrix and b is the bias matrix

The probability/threshold p_j^{t+1} is calculated using the stochastic sigmoid

$$p_j^{t+1} = \frac{1}{1+e^{-x_j(t+1)}}$$

Parallel Dynamics

Once we know how to update a single neuron, we could ask how to update every neuron in parallel. This technique is called **Parallel dynamics**.
Since, every node is independent, we can factorize the joint distribution of the next time step as:

$$P(s^{t+1}|s^t) = \prod_{j=1}^{N} P(s^{t+1}|s^t) = T(s^{t+1}|s^t) \quad \Leftarrow transition$$

This is done for all the neurons;
and since this approach is a Markov Chain, we can say that the probability of being in a state S' at the time $t+1$ is equal to the probability of being in the state s at the previous time t multiplied for the transition T from S' to S (for each possible state)

$$P(s^{t+1} = s') = \sum_s T(s'|s)\, P(s^t = s)$$

Glauber Dynamics

Another way to update a BM is the so-called **Glauber dynamics**.
Here we don't update all the neurons together but we update **a single neuron at a time**; the neuron is selected randomly.
A theorem tells us that if we use Glauber dynamics in a BM we achieve a stationary distribution.
In particular this distribution is the **Boltzman-Gibbs** one that at the time of infinity tells how the neurons are fired.

$$P_\infty(s) = \frac{e^{-E(s)}}{Z}$$

where $E(s)$ is the energy function and Z is the partition

Learning in BM

A theorem says that a Boltzmann machine is able to represent any distribution of binary variables.

The parameters M and b can be learned.

Let's do some simplifications:

1) the bias b is included into the matrix M
2) we consider only visible nodes, so $s = v$. (next we will include hidden nodes).

Since it's a MRF, we can express our objective function as a probability and this means that we will maximize our log-likelihood.

$$\mathcal{L}(M) = \frac{1}{L}\sum_{l=1}^{L} log\, P(v^l|M)$$

The likelihood $P(v^l|M)$ can be obtained using the Boltzmann-Gibbs distribution and we can maximize it differentiating over the parameters:

$$\frac{\delta \mathcal{L}(M)}{\delta M_{ij}} = \frac{\delta \frac{1}{L}\sum_{l=1}^{L} log\, P(v^l|M)}{\delta\, M_{ij}}$$

Recall:

- $v^l = v_1^l \dots v_n^l$
- $P(v^l|M) = \frac{\exp\{-E(v)\}}{Z}$
- $E(v) = -1/2 v^T M^T v = -\Sigma_{ij} M_{ij} v_i v_j$

Let's perform the calculus starting from a single pattern : $log\, P(v^l|M)$

$$\frac{\delta\, log\ P(v^l|M)}{\delta\, M_{ij}} = \frac{\delta(-E(v) - log\, z)}{\delta\, M_{ij}}$$

in this step we simplified log and exp and applied a property of logarithms.

Now, apply the definition of Energy function:

$$= \frac{\delta(-(-\Sigma_{ij} M_{ij} v_i v_j) - log\, z)}{\delta\, M_{ij}} = \frac{\delta(\,(\Sigma_{ij} M_{ij} v_i v_j) - log\, z)}{\delta\, M_{ij}}$$

prop of derivatives:

$$= \frac{\delta(\,(\Sigma_{ij} M_{ij} v_i v_j)}{\delta\, M_{ij}} - \frac{\delta\, log\, z}{\delta\, M_{ij}}$$

the result for the first term is just $v_i v_j$

$$= v_i v_j - \frac{\delta \log z}{\delta M_{ij}}$$

applying the definition of partition function on the right we obtain:

$$= v_i v_j - \frac{\delta \Sigma_v P(v|M) v_i v_j}{\delta M_{ij}}$$

and now we can notice the definition of expectation (on the right)

$$= v_i v_j - \mathbb{E}_{v_i, v_j \sim P(V|M)}[v_i v_j]$$

$$= v_i v_j - \langle v_i v_j \rangle_M = \frac{\delta \log P(v^l|M)}{\delta M_{ij}}$$

We obtained this result from just one single pattern, if **we perform it over all the patterns** we obtain:

$$\frac{\delta \mathcal{L}}{\delta M_{ij}} = \frac{1}{L} \sum_l^L (v_i^l v_j^l) - \langle v_i v_j \rangle_M$$

The summary can be seen as the expectation over the empirical distribution, so we can rewrite everything in this way:

$$= \langle v_i v_j \rangle_C - \langle v_i v_j \rangle_M$$

We can note $\langle v_i v_j \rangle_C$ as "**wake**" and $\langle v_i v_j \rangle_M$ as "**dream**".

Maximizing the log-likelihood, **we are making the wake and the dream part identical.**

This is an Hebbian Learning case: the wake part is the Hebbian term and it means that the two neurons v_i, v_j **are following the empirical distribution from the dataset**, while the dream part is the anti-Hebbian and **it means what the model thinks about the world.**

In both cases, since the two neurons are binary, both of them must be activated in order to obtain a non-zero value in the product $v_i v_j$ for this reason we talk about the "**coactivation** of the neurons".

Learning in BM with hidden nodes

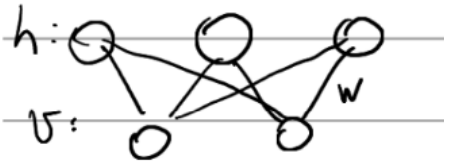
The whole previous calculus was made under the assumption of having only visible nodes.
If we want to add even hidden nodes h, the log-likelihood gradient ascent doesn't change so much:

$$= \langle s_i s_j \rangle_C - \langle s_i s_j \rangle_M \qquad \text{where } s = [hv]$$

But here there is a big problem: expectations became intractable due to the partition Z. So, we have to **restrict** the model.

Restricted Boltzmann Machines

A Restricted BM is a special BM: a **bipartite graph** with connections only from hidden and visible units. RBM is a probabilistic graphical model for unsupervised learning that is used to discover hidden structure in data.



Note that each visible node is connected to all and only the hidden ones and vice versa (no connection among the same "components").

The graph bipartition itself allows us to **perform learning and inference in a tractable way** because the bipartition factorizes distribution.

The energy function is so defined:

$$E(v,h) = -v^T M v + b^T v - c^T h$$ (c and b are biases)

Hidden units are conditionally independent given visible units and vice versa, so they can be updated in batch.

$$P(h_i|v) = \sigma(\Sigma_i M_{ij} v_i + c_j)$$
$$P(v_i|h) = \sigma(\Sigma_i M_{ij} h_i + b_j)$$

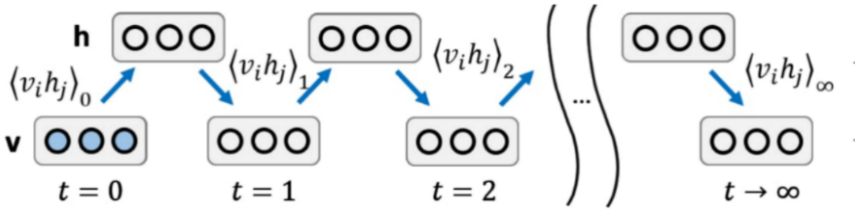
Train a RBM using Gibbs

Training can be performed by likelihood maximization as usual.

$$\frac{\delta \mathcal{L}}{\delta M_{ij}} = \langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_\infty$$

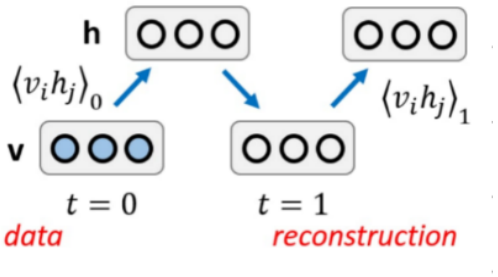
The first expectation $\langle v_i h_j \rangle_0$ represent the **data** while the second one $\langle v_i h_j \rangle_\infty$ represent the **model**, and both can be approximated using a **Gibbs sampling approach**.

Given an input data v^l (so, a binary array) in order to approximate $\langle v_i h_j \rangle_0$ we get a simple input value and we sample from $P(h|v)$ the weights for the hidden nodes.
~~Then, we can perform the same sampling but with respect to visible nodes, given hidden one: $P(v|h)$~~. To obtain $\langle v_i h_j \rangle_0$ we do this just once, but we iterate many times. Indeed, to obtain $\langle v_i h_j \rangle_\infty$ we should perform this process infinite time, but obviously this is not possible, so we stop after a certain value k



Train a RBM using Contrastive Divergence

Using the Gibbs approach can require too much time, for this reason, we can use the **Contrastive Divergence** approach which rely on just four steps:



1) Starting from data v^l we put them in the visible units of the model.
2) We update all the hidden units, in parallel, given v.
   Here can obtain the data correlation $\langle v_i h_j \rangle_0$
3) We reconstruct the visible units once, given the hidden units
4) We update the hidden units once again, obtaining $\langle v_i h_j \rangle_1$

Now, our approximation of the log-likelihood would become just:

$$\langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_1$$

This method is a very crude approximation of the gradient of the log-likelihood but we use it just because it works well.

CNN

CNNs are simply NNs which use the convolutional operation instead of matrix manipulation in at least one layer.

Initially, CNNs were created for sequences, and the introduced the **parameter (weights) sharing** into a NN trained with the backpropagation.
Then, they were applied to images.

Dense Vector Multiplication

Suppose to have a $32x32x3$ RGB image (very small) and a NN with 100 neurons.
If we would process the image using a "naive" approach w would have $32x32x3 = 3072$ features and so we would have $100 * 32x32x3$ different weights for handling a simple image.

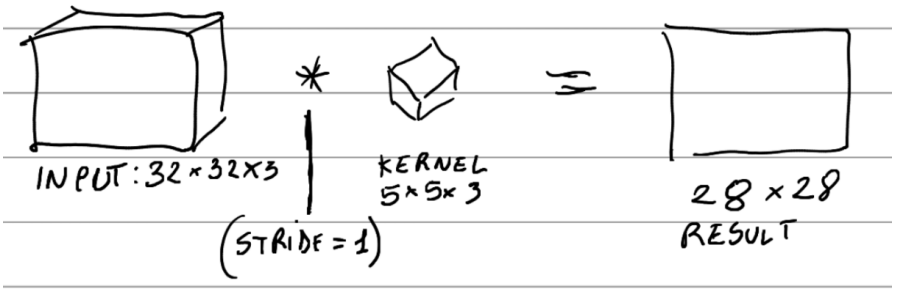
This can become infeasible for larger images. For this reason we introduce convolution.

Adaptive convolution

For "adaptive" convolution we mean the convolution operation executed using a kernel on adaptive weights. We have a larger input matrix, a kernel which contains the weights and the output matrix (called **feature map**).

Multichannel convolution

An image, usually, has 3 dimensions, hence, our convolution needs to be performed in three dimensions.
All channels are typically convolved together, and they are then summed up in the convolution.
The result is a bidimensional matrix.



Stride

The stride is a hyperparameter of a convolutional layer. It refers to the spacing at which we apply the filter to the image and so it reduces the number of multiplications.

Activation Map Size

The size of the image after the application of a kernel given the stride and size is given by:

$$W' = \frac{W-K}{S} + 1$$

$$H' = \frac{H-K}{S} + 1$$

Zero Padding

Suppose to have an input of $7x7$ pixels, a $3x3$ filter and a stride$= 3$.
The output image simply doesn't fit with these parameters and we wouldn't scan the whole image. So, we can use the **zero padding** strategy which adds columns and rows of zeros to the borders of the image in order to fit each combination of shape of filer and stride value.

The necessary value for the padding is given by:

$$P = \frac{K-1}{2}$$

and the output image will have a dimension by

$$W' = \frac{W-K+2P}{S} + 1$$

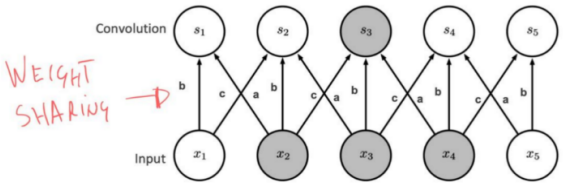
If we don't use the zero padding, the CNN **shrinks** the number of units, layer by layer. This is called **Zero Padding Effect**.
So by adding the zero padding, we prevent the unit shrinking.

Pooling

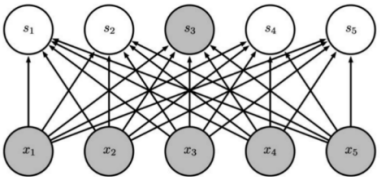
Pooling is an operation executed on the feature map, used to subsample it.
There are different kinds of pooling (Average, Max) and each of them has different aims.
The pooling operation helps to make the hidden representation invariant to small transformations of the input.

CNN as Sparse NN

CNNs can be seen as a **sparse NN**, indeed using the convolution we have a sparse connectivity among the neurons and using the **weight sharing** we even reduce the parameter number. The connectivity sparsing is dependent also from stride (higher the stride, higher the sparsity)



On the contrary this is an example of dense NN:



Pooling and spatial invariance

The pooling operation helps to make the hidden representation invariant to small translations of the input.
In particular, the max-pooling achieves this behavior because it takes the max-value from its input but it doesn't care the position of the maximum value (within the input).

So, since each feature map is a way to represent the same object (ex. rotations) using the max pooling we will have spatial invariance.

Hierarchical Feature Organization

If the hierarchy of the CNN is deep enough, every unit in the output layer is undirected connected to the input layer.

CNN training

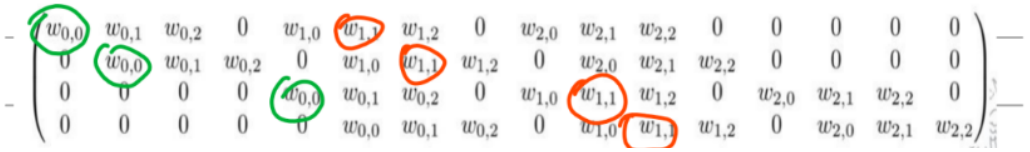
CNNS are trained with a slither modified version of the backpropagation.
The main difference is that CNNs backpropagation needs to take into account the **weight-sharing**.

The gradient Δw_i is obtained by **summing the contribution from all connections**. Moreover we need to deconvolve and consider the zero padding.

Backpropagation on CNN

After the convolution, the output is smaller than the input. So when we apply the backpropagation, we need a way to get the same input dimensions.

We solve this problem **with a matrix,** writing the convolution as a dense operation (see CNN as Sparse NN figure) **filled with zeros** other items in the matrix.



Note that the weights are repeated.
Now, we can finally apply the backpropagation on this weight matrix.

Deconvolution

The **deconvolution** operation can be seen as the inverse operation of the convolution, even if the image after a convolution-deconvolution is not the exact same input, as for the "mathematical" deconvolution operation.

AlexNet

This is the main model for computer vision.
It works on RGB images with $227x227x3$ images and there are 5 convolutional layers and 3 dense layers.
For each layer, the path is splitted into two parts in order to parallelize the whole work on different GPUs.

The main innovations of AlexNet were:

1) **Data augmentation**: the use of scaling and rotation to make the model more robust in terms of invariance.
2) **ReLU**
3) **Dropout**

AlexNet has millions of parameters, most of them are in the dense layer.

ReLU

The sigmoid activation function satures its input easily, so we use it only for the last layer.
For hidden layers, we prefer the ReLU because it helps to fight the gradient vanish and it's easy to compute and favorites sparsity

GoogleNet

Another main architecture is called **GoogleNet** which uses different kernel sizes for the same layer in order to capture details at various scales.

GoogleNet's Inception Module

In the GoogleNet architecture we can find the **inception module**, this architecture performs multiple operations (convolutions, pooling) with multiple kernel-size in parallel, in order to have no bias.

1x1 Convolutions

It might seems that 1x1 convolution can be an useless operation, but, this is not true because it can be useful in order to reduce dimensionality (and saving operations)

Batch Normalization

In a very deep network, we could have the **internal covariate shift**, which is the change in the distribution of the network activations due to the change of parameters during the training.

This problem can be solved by applying the batch normalization, which is the normalization with respect to mean and variance for each minibatch.

ResNet

ResNet is another architecture with 152 layers; since it's huge it has a mechanism for fighting the gradient vanish.

ResNet Trick (Residual Blocks)

In traditional NNs, each layer feeds into the next one.
ResNet introduced the residual blocks, where each layer feeds into the next one and the layer two steps ahead.
In this way, the input bypasses the convolution and it's combined with the last part.

Deconvolutional Network

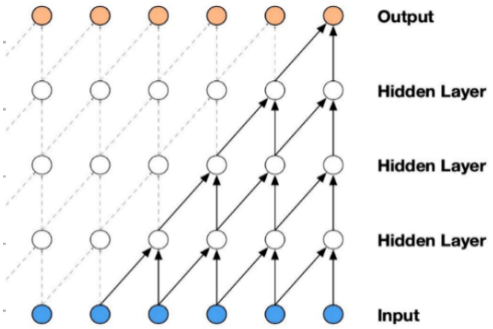
If we want to see graphically the result of a convolution we can plug a **Deconvolution Network** after a Convolutional one.
Note that the pooling operator has its "inverse" operation called **unpooling**.

Occlusions

Another technique to understand how a CNN works inside is called **Occlusion**.
This strategy applies a gray patch on the initial image and projects back the response of the best filters using deconvolution.

In this way we can generate a heatmap of useful and useless parts of the image.

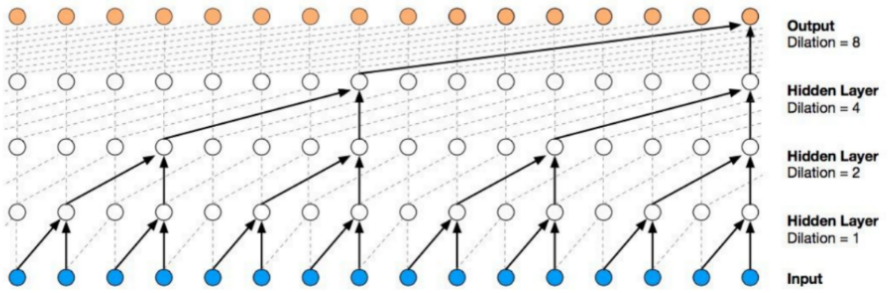
Causal Convolutions



If we want to use a Deep NN for temporal data we need to be sure that the model doesn't violate the order behind the data. In other words, the prediction for x_{t+1} must depend only on from $x_1 \dots x_t$.

This can work only for **short** sequences.

Dilated Causal Convolution



If we want to extend the causal convolution to longer sequences without increasing the complexity cost, we can use the **dilatation**.
Here, the filter is applied over an area larger than its length, skipping inputs with a certain step: it's similar to stride but here the output has the same size as the input.

Semantic Segmentation

Semantic Segmentation is the task of clustering part of an image applying the correct label to each cluster.
Traditional CNN architectures cannot be used for this task because their output is downsampled (with respect to the input) due to the stride and the pooling operation.

Fully convolutional networks for Semantic Segmentation

The very first approach to semantic segmentation was the **Fully Convolutional Network** approach which didn't use a single array for the output but a matrix with the same size of the input.

The convolutional part extracts features with different sales and all the information (from a different scale) are merged.
Then we can learn the upsampling function which is the one that segments the image.

Deconvolution architecture for Semantic Segmentation

Another approach to Semantic Segmentation is the **Deconvolutional architecture**. We can perform a number of convolutional operations and then a number of deconvolutional operations to achieve the Segmentation task.
This is an example of encoder-decoder architecture.
At the end the results are given by a softmax operation

Autoencoders

An autoencoder is a model that reconstructs the input passing through an hidden layer h.

The size K of the hidden layer usually is much less than the size of data input (or output) layer D, but we could have even $K = D$ or $K >> D$.

An autoencoder can be seen as a composite function $g(f(x))$ where x is the input, f() is the encoder and g() is the decoder.
The training of this NN is made by loss minimization.

We can distinguish between 3 kind of regularized AE:

- Sparse
- Denoising
- Contractive

Plus we also have autoencoders with dropout layers.
Regularized Autoencoders are usually overcomplete $K > D$

Sparse Autoencoder

In a SAE we have a huge number of units in the hidden layer.
Let $\tilde{x}$ be the reconstructed input (the result of g(x)) and the loss function $L(x, \tilde{x})$.

In a SAE, we add a regularization term to the hidden layer because we want a small number of active inputs; this extra cost is parametrized by a hyperparameter.

$$J_{SAE}(\theta) = \Sigma_x(L(x, \tilde{x}) + \lambda \Omega(h))$$

Where typically:

$$\Omega(h) = \Sigma_j \; | h_j(x) |$$

Note 1: the activation of hidden layer units depends only on from inputs
Note 2: $\Omega(h)$ can be seen as a L1 Regularization but L1 is applied to weights, we here apply $\Omega(h)$ only to inputs.

Sparse Autoencoder : Probabilistic Interpretation

Training with regularization is a problem of Maximum A-Posteriori Inference (MAP).
So we want to learn by maximizing the log of the joint probability of h and x (hidden and observable units)

$$max\,[\,log\;\;P\,(h,x)\,]$$

This formula is equivalent to (by the def. of MAP as Likelihood + Prior):

$$max\,[\,log\,P(x|h)\;\;+\;\;log\,P(h)\,]$$

where:

- $log\,P(x|h)$ is the likelihood
- $log\,P(h)$ is the prior

The likelihood part is connected to the ability of the AE to reconstruct the input, the prior part is chosen by us. So we pick the Laplace prior which gives us the norm-1 penalization we want

$$P(h)\;\;=\;\;\frac{\lambda}{2}exp(-\frac{\lambda}{2}|h|_1)$$

Denoising Autoencoder

In this kind of autoencoders, we train them to minimize the following the Loss function:

$$L(x,\;g(f(\hat{x}))$$

where $\hat{x}$ is the original input x corrupted by some noise. The noise is drawn from a Gaussian with mean = 0 and variance = 1.

We want, in the training step, to reconstruct x from its noisy version, in order to increase the robustness of the DAE, with respect to noise.
DAE can be used to reduce dimensionality, finding for each group of features a concept.

Denoising Autoencoder : Probabilistic Interpretation

A DAE is able to learn the denoising distribution $P(x|\hat{x})$ by minimizing the log probability of reconstructing the input x given the encoded version of the corrupted input.

$$-\;log\;\;P(x\,|\,f(\hat{x})\,)$$

Manifold Learning : How DAE increase the robustness

We can imagine that data belongs to a high-dimensional space called **manifold**.
When we add noise to input x, we are pushing out the x from the manifold and when this corrupted input $\hat{x}$ is reconstructed by DAE, it is not exactly equal to the original x.

In particular, the reconstructed input is close to the manifold, so by iterating this process we can learn every and only relevant inputs (the ones that remain in the manifold).
Indeed, if adding noise to an input, this one is pushed out from the manifold, it is not a relevant one.
So, we are learning:

$$g(h) - x \propto \frac{\delta \log P(x)}{\delta x}$$

This works relying on the manifold assumption.

The manifold assumption

The manifold assumption assumes that data lies on a lower dimensional non-linear manifold.
A denoising AE learns the lower-dimensional manifold of its inputs, adding noise, and trying to reconstruct the input.

Contractive Autoencoder

Contractive AE is similar to an DAE but CAE is robust to the infinitesimal variations of inputs.
A DAE which introduces really small noise gives us a result similar to the CAE.

But, CAE has a different penalty term, defined as the Frobenius norm:

$$\Omega(h) = \left|\left| \frac{\delta f(x)}{\delta x} \right|\right|_F^2$$

this term became high when $f(x)$ changes a lot, but x don't change so much.

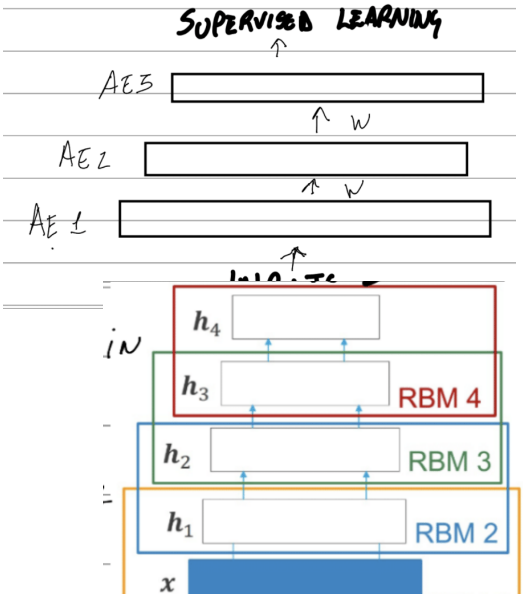
this Ω is plugged into J.

Deep Autoencoder

We can stack several autoencoders in a way that they reduce the dimensionality of the input.

AEs can be used also for data visualization and or for simplifying tasks for supervised learning models.

In a Deep AE, between layers we have weights and we update them using backpropagation and so we could have gradient vanishing problems.



G.D. Acciaro

A very stupid solution to fix this is to stack autoencoders, reconstructing layer-by-layer the previous hidden representations, this technique is called **Unsupervised Layerwise Pretraining**.

Deep Belief Networks

In unsupervised layerwise pretraining technique, each layer h_i is trained to transform h_{i-1} into $\tilde{h}_{i-1}$.
So, we can split the stack two by two and see these couples as a **RBM**; indeed, we can use a RBM to learn the weights between two layers.

Recall: This is a deep AE, not a deep Boltzmann machine because the edges are directed.

This model is called **Deep Belief Network**.

The trick of DBNs

The train of a DBN starts from training the first RBM (first two layers: inputs and h^1) in order to obtain the weights W^1.

$$P(x|\theta) = \Sigma_{h^1} \; P(h^1|W^1) \; P(x \mid h^1, W^1)$$

from this, we can extract the probability of h_1 given W^1:

$$P(h^1 \mid W^1) = \Sigma_x P(h^1, x \mid W^1)$$

Independently, we can train the second RBM in the same way, obtaining:

$$P(h^1 \mid W^2) = \Sigma_x P(h^1, h^2 \mid W^2)$$

Then we have two weights for the same h^1, so we can apply the trick: we simply average the contribution of W^1 and W^2. In this way we don't count the contribution of x twice, due to h^2 depends on x as well.

A/N: Ho qualche dubbio su questo paragrafo, ti consiglio vivamente di confrontarlo con altre fonti.

DBN: Discriminative Fine-Tuning

Once we perform the trick for each RBM within the model, we can use matrices to initialize a Deep AutoEncoder, fine-tuning the whole structure with backpropagation.

Gated RNN

Dealing with sequences

We want to deal with sequential data, which are variable-size data that describe some sequentially-dependent information.

Our aim is to capture the dynamic context c_t which is a fixed vector for identity piece of sequences in order to perform predictions.

We can use two main models:

1) RNNs
2) Gated RNN

RNN design

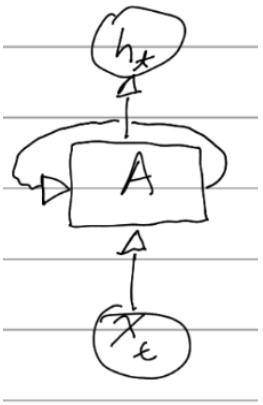
A RNN structure influences the processing of sequential data **(inductive bias)**, regarding the training it's quite easy to do, but it depends a lot on the architecture.

Supervised recurrent tasks

Using sequences we could have 4 kinds of task:

1) Element-to-Element (real time analysis)
2) Sequence-to-Element (sentiment analysis)
3) Element-to-Sequence (generative models)
4) Sequence-to-Sequence (machine translation)

Vanilla RNN



The simpler RNN is called **Vanilla** and it's made up by a pre-activation function called $g_t(\dots)$ which is a weighted sum of the previous output, the actual input and the bias.

$$g_t(h_{t-1}, x_t) = W_h h_{t-1} + W_{in} x_t + b_h$$

where usually, the activation function used is the tanh

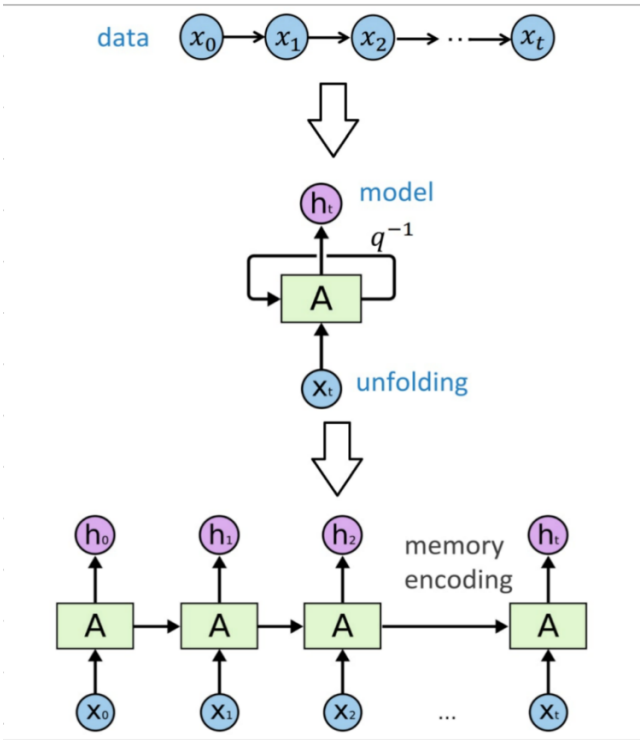
$$h_t = tanh(g_t)$$

so, the output is given by:

$$y_t = f(W_{out} h_t + b_{out})$$

Unfolding RNN (Forward)

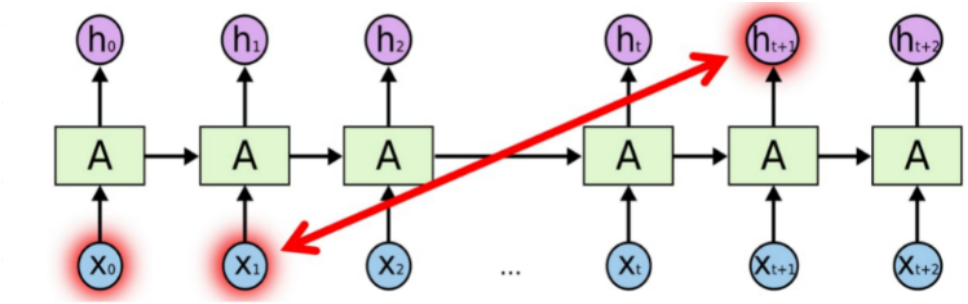
We want to map a sequence $x_0 \ldots x_t$ to a fixed length encoding h_t.



Long term dependency

The hidden state h_t summarizes information on the history of the input signal, up to the time t.

But when the time gap between the observations and the output grows, there is only a **little** influence on that input to the current output.



The cause of the long-term problem depends on the fact that a gradient that is propagated over too many layers tends to vanish or to explode.

In a RNN, weights are shared between time, so summing the gradient contributions through time, it can easily generate vanishing or exploding gradients.

RNN Backprop (Backward)

The gradient in a RNN is made up by three terms:

1) the actual gradient (at time t)
2) the gradient t with respect to all the previous $1 .. t-1$
3) the parameter matrix (Jacobian) W

$$\frac{\delta L_t}{\delta W} = \sum_{k=1}^{t} \frac{\delta L_t}{\delta h_t} \frac{\delta h_t}{\delta h_w} \frac{\delta h_w}{\delta W}$$

$$= \sum_{k=1}^{t} \frac{\delta L_t}{\delta h_t} \left(\prod_{l=k}^{t-1} \frac{\delta h_{l+1}}{\delta h_w}\right) \frac{\delta h_w}{\delta W}$$

The first two terms are recursive over time, so the **gradient is a recursive product of hidden activation gradients.**

RNN: Finding the magnitude

$$\frac{\delta L_t}{\delta W} = \sum_{k=1}^{t} \frac{\delta L_t}{\delta h_t} \frac{\delta h_t}{\delta h_w} \frac{\delta h_w}{\delta W}$$

$$= \sum_{k=1}^{t} \frac{\delta L_t}{\delta h_t} \left(\prod_{l=k}^{t-1} \frac{\delta h_{l+1}}{\delta h_w}\right) \frac{\delta h_w}{\delta W}$$

Let's focus only on the recursion part: we are interested in finding the magnitude of the gradient since it tells us if the gradient will vanish or explode.

$$\frac{\delta h_{l+1}}{\delta h_w} = D_{l+1} W^T_{h\,l}$$

where D_{l+1} is the diagonal matrix with tanh.

So:

$$\frac{\delta L_t}{\delta h_k} = \frac{\delta L_t}{\delta h_t} \left(\prod_{l=k}^{t-1} \frac{\delta h_{l+1}}{\delta h_w}\right) = \frac{\delta L_t}{\delta h_t} \left(\prod_{l=k}^{t-1} D_{l+1} W^T_{h\,l}\right)$$

and we are interested in the following magnitude:

$$\left\| \frac{\delta L_t}{\delta h_k} \right\| = \left\| \frac{\delta L_t}{\delta h_t} \left(\prod_{l=k}^{t-1} D_{l+1} W^T_{h\,l}\right) \right\|$$

Given a property of norms we can put an upper bound

$$\left\| \frac{\delta L_t}{\delta h_t} \left(\prod_{l=k}^{t-1} D_{l+1} W^T_{h\,l}\right) \right\| \leq \left\| \frac{\delta L_t}{\delta h_t} \right\| \left(\prod_{l=k}^{t-1} \left\| D_{l+1} W^T_{h\,l} \right\|\right)$$

$$\left\| D_{l+1} W^T_{h\,l} \right\| = \sigma(D_{l+1}) \sigma(W^T_{h\,l})$$

where σ is the **spectral radius** which tells us if the gradient vanish ($\sigma < 1$) or explode ($\sigma > 1$).

Gradient clipping

The easiest way to fix the gradient problem is to just limit the gradient itself using a threshold θ for the maximum value of the gradient.
So if the norm of the gradient is $> \theta$ then we limit the gradient.

$$if \ \ || g || \ > \ \theta \ \ then$$
$$g \ = \ g \ * \ \frac{\theta}{|| g ||}$$

Constant error propagation

If we would pick the identity activation function and the identity weight matrix, we can notice that we would have a nice spectrum property ($\sigma = 1$).
Unfortunately, this cannot work in practice because it saturates memory so fast since it replicates inputs.

Gated units

In order to obtain derivatives which neither vanish nor explode, we could use gated units, which are units that have connection weights that may change at each timestep.
We want to accumulate information, scrolling a sequence, up until we need.

Once we use that information, it might be useful to forget the state and we want to do it (**so we can control the forget**) automatically. This can be performed by a Gated RNN (a RNN with gated units).
A single gated unit is a unit with a mechanism to control the output, allowing or not, some part of the unit itself.

The gates allow us to control the forgetting.

Long Short Term Memory

An approach to solve the vanish/explode problem is the LSTM architecture, which is based on cells which have 3 kinds of gates: **input, forget** and **output**.
Each cell has an internal state c_t.

- the **input gate** controls how inputs contribute to the internal state
- the **forget gate** controls how past internal states c_{t-1} contributes to the actual internal state c_t.
- the **output gate** controls what part of the internal state is propagated out of the cell.

All of these three gates are implemented with the sigmoid function (look equations) and it's worth noting that the connection between the past internal state and the current internal state, which is handled by the forget gate, is essentially a self-loop, the idea of weight this self-loop was revolutionary.

Long Short Term Memory : Equations

LSTM works in this way:

1) First we compute the input and the forget gates, using the actual input x_t and the previous output state h_{t-1}.

$$I_t = \sigma(W_{Ih} h_{t-1} + W_{input} x_t + b_{input})$$
$$F_t = \sigma(W_{Fh} h_{t-1} + W_F x_t + b_{forgot})$$

The input gates tells us how much we have to consider the input
the forget gates tells us how much we have to consider the previous state

2) Once we have I_t and F_t, we can compute the potential g_t and then we can use that to calculate the internal state c_t

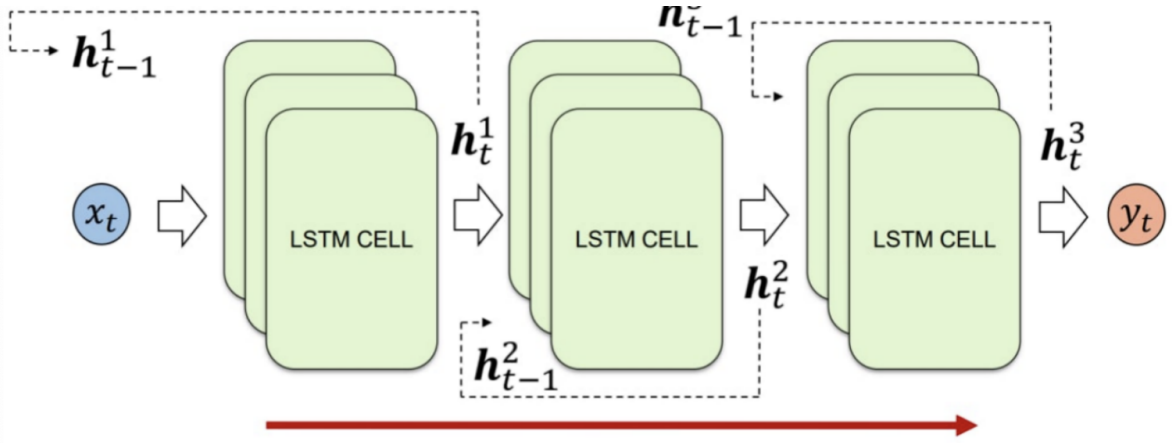
$$g_t = tanh(W_h h_{t-1} + W_{in} x_t + b_n)$$
$$c_t = F_t \odot c_{t-1} + I_t \odot g_t$$

where $\odot$ is the element-wise multiplication

3) At the end, we can finally compute the output gate o_t and the output state h_t.

$$o_t = \sigma(W_{O\,h} h_{t-1} + W_{O\,in} x_t + b_o)$$
$$h_t = o_t \odot tanh(c_t)$$

Deep LSTM



Once we have one single LSTM, we could stack several LSTMs in order to extract information (features) at an increased level of abstraction.

Long Short Term Memory : Training

The original paper of LSTM describes a training algorithm which was a custom version of backpropagation through time (BPTT).
Modern LSTM versions use the original BPTT algorithm.

Regularized LSTM: Dropout

Dropout is a technique that randomly disconnects units from the network during training.
It has a lot of benefits, first of all it prevents the LSTM to co-adapt units, and so it regularizes the LTSM.
It works very well with models with many units, and we have a "committee" effect (it's like having more than once LSTM which work together) but it's required to adapt the prediction phase (because in that case we don't need to have the dropout).
Another disadvantage is that it may drop too many units.

Minibatch LSTM and Truncated Backprop

As each NN, if we have a huge dataset, we could use the minibatch approach.
Another way to deal with huge sequences is the **truncated gradient propagation** which blocks the propagation after a number of iterations.

Gated Recurrent Unit

A **lighter** version of a LSTM is called GRU, indeed, GRU uses only two gates, instead of three (in the LSTM case).
If the dataset is small then GRU is preferred, otherwise LSTM is preferred.

In a GRU there is no internal state.
These two gates are:

- **Reset gate**: it acts directly on the output. It controls how much of the previous state we might still want to remember.
- **Update gate**: it allows us to control how much of the new state is just a copy of the old state.

Gated Recurrent Unit: Equations

- **Reset gate**: it acts directly on the output. It controls how much of the previous state h_{t-1} we might still want to remember in h_t. This control is made using a parameter z_t.

  $$h_{new} = 1 - z_t \odot h_{t-1} + z_t \odot h_t$$

  where

  $$h_t = tanh(W_{hh}(r_t \odot h_{t-1}) + W_{h\,in} x_t + b_n)$$

- **Update gate**: it allows us to control how much of the new state is just a copy of the old state. Here we can calculate z_t and r_t.

  $$z_t = \sigma(W_{zh} h_{t-1} + W_{z\,in} x_t + b_z)$$
  $$r_t = \sigma(W_{rh} h_{t-1} + W_{r\,in} x_t + b_r)$$

Attention

Language Modeling

Language modeling is an **unsupervised** task whose aim is to **determine the probability of a given sequence of words occurring in a sentence.**

Gated RNN Limitations

Gated RNN are powerful for handling inputs with different sizes, **but they are not so useful to manage outputs with different sizes.**
Furthermore, **Gated RNN don't have a way to focus on important parts of a sequence** (this problem will be solved using an Attention Mechanism).
At the end, **GRNNs still have troubles with very long dependencies** (this problem will be fixed with an external memory).

Sequence Transduction

Inputs and outputs are sequences but in some tasks they may have different lengths.
For example, machine translation:

- "The dog is on the table" has 6 words
- "Il cane è sul tavolo" has 5 words

RNNs for Language Models

A first approach to learn the length of the output sequence automatically was realized using an unfolded RNN where we ignore outputs while we are putting the input sequence and we leave blank the input when we are reading the output.
This approach doesn't work well because of the distance between from the first item x_1 and the last output y_m is very large.

This could be a reason for vanishing and so we could have a forgetting problem.
For this reason we introduce the Encoder/Decoder architecture.

Encoder/Decoder RNN

The encoder/decoder model for RNN was developed to address the seq-to-seq nature of **machine translation** tasks, where inputs may have different lengths with respect to the outputs.
From an high-level the model has two subparts:

1) Encoder: it creates a compressed and fixed length representation c (called context vector) of all the input sequence $x_1 \dots x_n$
2) Decoder: it executes the inverse operation, starting from the context vector c , it generates the output sequence.

Deeping on the decoder part

The decoder, as well as the encoder, is made up of a set of LSTM or GRU cells.
A first approach consists of passing the context vector c only to the very first cell and passing the output of this first cell to the second and so on.
But in this way we forgot the context c soon because of the vanish off the gradient within the decoder.
So, we can fix this, passing c to every cell in addition to the result of the previous one.

Hence, the update function will be:

$$s_i = f(c, s_{i-1}, y_{i-1})$$

where:

- s_i is the current state
- c is the context
- s_{i-1} is the previous state
- y_{i-1} is the previous output

Encoder/Decoder Learning

The Encoder/Decoder model can share parameters, but it's not a common case.

The learning can be performed in two ways:

1) **end-to-end** : the backprop algorithm is performed from y_m to x_1

2) **independently** : we split the learning task in two parts. It's useful when we have more combinations of encoders and decoders to be trained (e.g., an encoder/decoder for each language and we want to recreate Google Translate).

Attention Mechanism

The **Attention mechanism** selects which part of the sequence we need to focus on to obtain a good context c.
This mechanism is used to fix the assumption that the Encoder/Decoder has: it assumes that the last input summarizes sufficient information to generate the output, but in most scenarios, the last part of a sequence is not the most important part.
This mechanism is performed by an attention module that is placed after each cell, in the encoder part.

How attention works

The attention model (the component which performs the attention mechanism) takes in input a fixed number of hidden states ($h_1 \dots h_n$).

In addition, it takes the current context s.
Inside the module, gates give us the solution for choosing a good context vector c.

How attention works: Equations

This module, first of all, merges each hidden state h_i with the current context s.

Here we calculate $e_1 \dots e_n$: the coefficient of relevance for each hidden state i

$$RELEVANCE: e_i = a(s, h_i)$$

where:

- a is usually a tanh
- s is the context
- h_i is the hidden state

Then, we have to calculate the normalization term, applying the (differentiable(!!)) softmax function to $e_1 \dots e_n$ getting $\alpha_1 \dots \alpha_n$

$$NORMALIZATION: \alpha_i = \frac{exp(e_i)}{\Sigma_j exp(e_j)}$$

At the end, we aggregate $\alpha_1 \dots \alpha_n$ with an attention voting, getting our new context c

$$AGGREGATION: c = \Sigma_i \alpha_i h_i$$

Attention: Final considerations

(!) If h_i is a relevant information given the actual context s, the relevance parameter e_i will product a relevant value for α_i.

Advanced Attention

The calculus for finding the e_i values can be replaced with a neural network.

Hard Attention

A variant of the final combination of all the α_i at the end of the module, would be the use of choose a single α_i using probabilities.
So, we sample the $\alpha_1 \dots \alpha_n$ set and so we cannot use the backpropagation anymore, for this reason we will use an averaged result using the Monte Carlo method.

Transformers

A transformer is a model that uses the mechanism of self-attention, for the usual task of transforming a sequence into another sequence.
The main difference between this method and other architectures is that a Transformer doesn't use an RNN.
A famous one is BERT.

In the paper "Attention is all you need", transformers are still defined as an encoder-decoder architecture, but instead of using RNNs it uses the Multi-Head Attention method.

Since they use a lot of large matrix multiplications, transformers are optimized for GPUs, and in addition, since there are no recurrences, they can be executed in parallel.

Self attention

The self-attention idea is based on three different vectors which are calculated for each element of the input.

1) **Key** : a vector which represents all inputs
2) **Query** : a vector which represents what the model needs to respond to
3) **Values**: a vector which represents the output

Once we have these three vectors, we can compute the attention (over all other vectors):

$$SA(Q_i, K, V) = \Sigma_j Softmax(\frac{Q_i * K^T}{\sqrt{d_k}})v_j$$

where $\sqrt{d_k}$ is needed for normalization

Multi-Head Attention

The multihead approach is a generalization of the attention, which uses more than one attention module concatenating a number of heads, where a single head is the result of the attention given the query, key and values vectors (all these vectors have a weight matrix).

$$MultiHead(Q, K, V) = concat(head_1, \dots, head_n) W^{OUTPUT}$$

Masked Multi-Head Attention

The very first layer of the decoder part of a transformer is a slight version of multihead attention that is called masked multi-head attention where the mask is needed to not look at the future.

Positional encoding

Using RNNs the sequential order (the order within the sequence) is implicitly defined by the input. In Transformers we have no RNN and furthermore the multi-head layer reads all the input at once.
For these reasons we introduce the positional encoder layer that is a tensor with the same shape as the input which encodes the temporal order of the input.

RNN and Memory

Gated RNN was created to fix the problem of learning long-range dependency but in practice it is still difficult to learn long ranges.
For this reason, there are some alternatives that optimize the memory usage – which means ways to skip in a clever way the updates

1) Clockwork RNN
2) Skip RNN
3) Multiscale RNN
4) Zone-Out

We want to skip updates because in this way we mitigate the vanishing gradient effect.

Zone-out

The zone-out approach consists of, for each timestamp, forcing some unit to be the same.
These units are chosen stochastically; it's a very simple approach to avoid updates, in this way we improve the gradient propagation.
It's very similar to the Dropout idea but in that case we force the unit to be zero, in this case we force the unit to be simply the same.

Clockwork RNN

Zoneout is a simple static way that works for a single unit; Clockwork RNN is still a static way but it works for fixed length blocks of units.

The idea is that each block can be update T_i times at most

Note: The i start from 1, so earlier blocks can be updated only a few times.

It's useful for tasks that require information with several resolutions, the model generates RNNs for each resolution (e.g., Hz in a music file)

Skip RNN

Another clever way to skip the updates in a RNN, in order to manage the long dependency problem is called Skip RNN.
It's an **adaptive** method which means that it learns when it's needed to update the hidden state.

Hierarchical Multiscale RNN

Many sequences have an hierarchical structure and we want to express this structure.
If we could know the boundaries of the sequence, we could create layers of abstraction, but usually this is not possible.
For this reason we introduce hierarchical multiscale RNN which is an adaptive approach to fix the long-distance issue.
In particular, it tries to learn the underlying hierarchical structure in the sequence without explicit boundary information.

This approach is made up by 3 fundamental operations:

1) Update - update LSTM cells according to the "boundary detection"
2) Copy - copy LSTM cells and states to the current timestamp (from the previous one)
3) Flush - Send a report to the next layer reinitializing the current one

Neural Reasoning

NNs can be used for classical AI algorithms that provide some kind of reasoning.
We prefer to use NN because classical algorithms require proper inputs and are not always possible.
In order to solve this task, we have to memorize facts, questions and answers, so , a lot of information.
We introduce a **memory slot** which is used to contain all the data the NN needs.
This kind of architecture is called **Memory Network**.

Memory Network

Memory Networks are a kind of architecture that uses a memory slot to maintain data for a NN.
It's made up by four elements:

1) Input Feature Map which encodes the input
2) Generalization which decides what input needs to be written into the memory
3) Output Feature Map which reads the relevant memory slots
4) Response

We could have a stack of multiple memory layers in order to obtain better answers.

Neural Turing Machines

An example of memory network is the **Neural Turing Machines** which is a model that contains two main components:

- NN (controller)
- Memory Bank

The NN interacts with the world using input and output vectors, as usual, and with the memory using write and read operations.
It's worth mentioning that each component is differentiable so we can train everything with gradient descent.

The network is a RNN which uses soft-attention to determine how much to read and write from each point.

Unsupervised Generative Learning

In this section we will describe what are the deep learning models for unsupervised and generative learning.

We prefer unsupervised learning because applying labels to data is **expensive**.
Moreover if we focus only on classifying data, rather than learn how data are characterized, we could have some problems.

For this reason, we introduce **Generative Learning**, because it tries to characterize data distribution.

Explicit VS Implicit Generative Learning

Our aim is: given training data, we want to learn a NN that can generate **new** examples on approximation of the data distribution.

We can define two **main** approaches:

1) **Explicit:** it learns a model density $P_\theta(x)$. If it's tractable, we can use all the visible data in a **Sampling RNN** model.
   Instead, if the density is not tractable, we can introduce a latent space in a variational way (used in Variational AutoEncoder models)

2) **Implicit:** It learns a process that sample data $P_\theta(x) \approx P(x)$. It can be done in a direct way (GANs) or stochastically.

Learning distributions with fully visible informations

If we can assume that all data are visible, we can compute the joint probability using the chain rule factorization.
For example, for a generic input x we could have:

$$P(x) = \prod_i^n P(x_i \mid x_1 \dots x_{i-1})$$

If x is an image, a single $P(x_i \mid x_1 \dots x_{i-1})$ is the probability of a pixel to have a certain intensity value, knowing the intensity of the predecessors.

This idea has **two main problems:**

1) It requires to define an order among inputs (and it's not always possible)
2) The conditional distribution is difficult to compute.

Approximating the conditional probability

Since the two main problems in calculating the conditional probabilities, we can use a RNN which scans the inputs and encodes the dependency from the previous pixel in the states of the RNN itself.

It's worth noting that generating pixels from an image (even a small one 64x64) can be difficult due to the gradient **vanishing problem** (since there is a long term dependency from the first pixel and the last pixel).

Variational AE: From visible to latent information

The previous models can work only under the assumption that we have a **fully visible informations scenario**, only in this way we can learn the parameters θ.
We can generalize the parameter learning process into a **latent process** regulated by unobserved variables z.

$$P_\theta(x) = \int P_\theta(x|z)\, P_\theta(z) \quad dz$$

The calculation of the integral is **intractable** for non-trivial models (which means that we cannot compute all the z assignments).
For this reason we introduce **Variational AutoEncoders.**

Variational Autoencoders : Intro

Let's recall that an Autoencoder is a NN which transform input data x into output data $\tilde{x}$ in a fully deterministic way, passing through an encoder layer, which transform x into a latent space z, and a decoder layer, which transform z into the output data $\tilde{x}$.

In a probabilistic way, we can say that an AE generates $\tilde{x}$ sampling z (we assumed that), during the decoder phase, hence, as output we have value sampled from $P(\tilde{x}|z)$.
But, since we cannot have access to the true distribution we have to introduce **Variational AEs**.

Variational Autoencoders

The main idea behind a VAE used to estimate the parameters θ of $P_\theta(x)$ is the following cycle:

- We sample z conditioned by x, and we train the decoder with z in order to obtain x again.
- We sample new values for z, sampling the Gaussian distribution of mean and variance of the original x
- We reconstruct $\tilde{x}$ from z using only the decoder
- Now, we can represent $P(\tilde{x}|z)$ with a NN.

Variational Autoencoders : Problems

The VAE training is not so easy because ideally we would maximize the Likelihood on the dataset D.

$$\mathcal{L}(D) = \Pi^{N}_{i=1} P(x_i)$$

$$= \Pi^{N}_{i=1} \int P(x_i|z)\, P(z)\, dz$$

We have two problems finding the parameters which maximize this likelihood:

1) $\Pi^{N}_{i=1} P(x_i)$ is **intractable**, we can solve this problem with **Variational Approximation**.
2) $\int P(x_i|z)\, P(z)\, dz$ is **not differentiable**, we can solve it with the **Reparameterization trick**.

Variational Autoencoders : Reparameterization trick

Sampling z from the Gaussian distribution $N(\mu(x), \sigma(x))$ is **not** a differentiable operation. For this reason we sample a variable ϵ from a Gaussian $N(0, 1)$ and pass μ, σ and ϵ independently, to the "latent layer"of the decoder.

In this way we can use the backpropagation.

Variational Autoencoders : Variational Approximation

This technique is needed because $\Pi^{N}_{i=1} P(x_i)$ is **intractable.**

In practice, we can maximize the **ELBO** (Evidence Lower BOund) since:

$$log\, P(x|\theta) \geq \mathbb{E}_Q [\, log\, Q(x, z)\,] - \mathbb{E}[log\, P(x)] = \mathcal{L}(x, \theta, \phi)$$

So, maximizing the ELBO we can approximate the $log\, P(x|\theta)$.

We can rewrite (expanding) the $\mathcal{L}(x, \theta, \phi)$ in this way:

$$\mathcal{L}(x, \theta, \phi) = \mathbb{E}_Q [\, log\, P(x|z)\,] + \mathbb{E}_Q[log\, P(z)] - \mathbb{E}_Q[log\, Q(z)]$$

Note that the last part is equal to $KL(\, Q(z|\phi) \,||\, P(z|\theta)\,)$.

From that we can conclude that we are looking for a Q (a generic distribution) which is able to approximate P, maximizing the ELBO.

Variational Autoencoders : Training

The training of a VAE is performed by backpropagation on θ and ϕ, in order to optimize the ELBO.

$$\mathcal{L}(x, \theta, \phi) = \mathbb{E}_{Q}[\, log\, P(x|z)\,] + KL(Q(z|x, \phi)\,||\,P(z|\theta))$$

where $Q(z|x, \phi)$ represents the **encode** network and $P(\tilde{x}|z, \theta)$ describes the **decoder** network.

The first term $\mathbb{E}_{Q}[...]$ represents the ability of the VAE to **reconstruct** x given z and θ.

The KL part can be seen as a **regularized** term which force z to be "more simple as possible" and it can be computed easily when both $Q(z)$ and $P(z)$ are Gaussian.

Variational Autoencoders : Training Loss

Our final Loss is applied to each item $x \in D$ (dataset):

$$\mathbb{E}_{x\sim D}[\,\mathbb{E}_{z\sim Q}[\, log\, P(x|z)\,] - KL(Q(z|x, \phi)\,||\,P(z))\,]$$

But here we still have the non-differentiable operation of drawing z from the Gaussian which doesn't allow us to apply the backpropagation.

So, <u>we can reuse the parametrization trick in order to obtain a fully deterministic loss usable in a standard backpropagation algorithm.</u>

$$\mathbb{E}_{x\sim D}[\,\mathbb{E}_{\epsilon\sim N(0,1)}[\, log\, P(x|z, \theta)\,] - KL(Q(z|x, \phi)\,||\,P(z))\,]$$

Where $z = \mu(x) + \sigma(x) * \epsilon$ and spot how $z \sim Q$ has become $\epsilon \sim N(0, 1)$.

Note: after the reparameterization trick, the expectations don't depend on distributions so we can perform SDG.

Variational Autoencoders : Information theoretic interpretation

$$\mathbb{E}_{x\sim D}[\,\mathbb{E}_{\epsilon\sim N(0,1)}[\, log\, P(x|z, \theta)\,] - KL(Q(z|x, \phi)\,||\,P(z))\,]$$

where $z = \mu(x) + \sigma(x) * \epsilon$

The **first part** of the equation indicates the **number of bits required to reconstruct x** from z under the ideal encoding (i.e. Q(z|x) is generally suboptimal) meanwhile the **second term is the number of bits required to convert an uninformative sample from P(z) into a sample from Q(z|x).**

The information gain is the amount of extra information that we get about x when z comes from Q(z|x) instead of from P(z)

Variational Autoencoders : Testing

After the training time, we can test the VAE detaching the encoder and sample from a random encoding (eg. a Gaussian distribution $N(0, 1)$), generating a sample that is the reconstruction.

VAE vs DAE

In Denoising (or Contractive) AE, given an input data, we create a more robust version of that input data with respect to a lot of very small variations, generating noisy data which are mapped all on the same input.

The Variational AE, instead, tells us that given an input data x, we can obtain the distribution of the latent space.This distribution can be sampled into the original space in order to obtain the reconstructed input.

Conditional Generation (CVAE)

A variant of VAE called **Conditional VAE** is able to control the generation process, adding, in the training phase of the VAE an additional visible y (both in the decoder and the encoder) which let the model to be able to learn the conditional distribution $P(x|y)$.

At inference time, this y is used into the decoder part only, in addition to the usual random samples $z \sim N(0, 1)$ to obtain new datas $\tilde{x}$ conditioned by y_i.
In this way we can even control the generating process.

GAN

VAE learns how to approximate the intractable distribution:

$$P_\theta(x) = \int P_\theta(x|z)\, P_\theta(z)\, dz$$

Instead, GAN (**Generative Adversarial Networks**) learns how to generate samples from a complex, high-dimensional distribution. It starts sampling from random noise, training a NN to transform this random noise into a training distribution.

GAN: Components

A GAN is made up by two components:

- **Generator** : which try to generate samples that can **fool** the discriminator
- **Discriminator** : which tries to **distinguish fake** data **from original** data.

GAN: Loss

The discriminator and the generator compete to win a min-max game because the generator wants to learn how to generate better and better samples in order to fool the discriminator.
This means that **the generator wants to <u>minimize</u> the distance between 1** (the value that discriminator uses to label real samples) **and the output of the discriminator on generated data** $D(G(z))$. This is performed using a gradient <u>descent</u>.

On the contrary, **the discriminator doesn't want to be fooled**, so it wants to maximize the same distance; so it uses a gradient <u>ascent</u> method.

$$C_{GEN} = \min_{\theta_G} \mathbb{E}_z[\, log\,(\,1 - D(G(z))\,)]$$

$$C_{DISCR} = \max_{\theta_D} \mathbb{E}_x[log\, D(x)] - \mathbb{E}_Z[log(1 - D(G(z))]$$

Actually, optimizing the generator formula does not work due to a flat gradient.
For this reason, we rewrite the loss function of the generator.

$$C_{GEN} = \max_{\theta_G} \mathbb{E}_z[\, log\,(\,D(G(z))\,)]$$

Wasserstein distance

The optimal solution of a min-max problem is a saddle point.
We could have better results using the **Wasserstein Distance** as loss which is a constrained distance between the generator and the empirical distribution. The constraint is applied on the discriminator weights.

$$G^* = argmin_G \; sup_{||\,D\,|| \le 1} \mathbb{E}_{x\sim\mu}[D(x)] - \mathbb{E}_{x\sim\mu_G}[D(x)]$$

Adversarial AE

An **Adversarial AE** is a mix between GANs and VAEs. Indeed, it uses the main idea of a VAE but applying an adversarial loss of a GAN.
Recall that the "theoretical" Loss of a VAE (before reparametrization) is:

$$\mathcal{L}(x) = \mathbb{E}_Q[log\, P(x|z)] - KL(\, Q(z|x) \,||\, P(z)\,)$$

The KL term is replaced with the adversarial loss in a AAE.
Here, the **<u>reconstruction</u> phase** updates the encoder and the decoder to minimize the reconstruction error, while the **<u>regularization</u> phase** updates the discriminator to distinguish true samples from generated samples and updates the generator to fool the discriminator.

AAE are useful to impose prior to a VAE.

Reinforcement learning : Intro

Reinforcement learning is a general-purpose **framework for decision making**.

Reinforcement learning means how to decide what we have to do in order to maximize a numerical **reward**. There is no supervisor, just rewards.

A RL agent has explicit goals and it can perform a set of actions (depending on the state it is in at that moment). Actions influence the environment.

Formerly, RL is based on the **reward hypothesis**:

> Every goal can be seen as the maximization of the expected value of the cumulative sum of rewards.

Rewards

A **reward** $R_t \in R$ **is** simply **a scalar feedback** signal **and** it **indicates how well the agent is doing** at the step t; in particular, at each timestep the environment sends to the agent the reward.
The aim of the agent is to maximize the total reward.

At each step t, the agent executes an action A_t and receives, from the environment, the observation O_t and the reward R_t

Sequential Decision Making (our aim)

Our problem is about finding a sequence of actions that maximize the reward.
It's worth mentioning that we want to maximize the reward "on the long run", so we could have some situations where it's better to sacrifice the immediate reward, in order to obtain a bigger reward later.

History and State

The **History** H_t is the sequence of observations, actions and rewards up to the time t.

$$H_t = O_1, R_1, A_1, \ldots\ldots, O_t, R_t$$

It's important because what happened in the next timestep depends on the history, this means that the agent chooses the action relying on the history and the environment selects observations and rewards relying on the history too.
The **State** S_t is the information used to determinate what will happen in the very next timestep and it takes the history as input.

$$S_t = f(H_t)$$

Environment State

The environment is the manifestation of the problem being solved. It could be real or virtual.

The environment state S_t^e is the representation of the environment at the time t.

Usually, the environment state is not visible to the agent, and if it's visible it could contain irrelevant information for the agent.

Agent State

The agent S_t^a is the representation of the point of view of the agent. It contains all the information the agent uses to select its next action. Also, this state can be seen as a function of the history:

$$S_t^a = f(H_t)$$

Information (Markov) State

An **information** (or **Markov**) **state contains all the information from the history**.

Formerly, it's defined using the first-order Markov assumption:

A state is Markov if and only if

$$P(S_{t+1}|S_1 \dots S_t) = P(S_{t+1}|S_t)$$

Some facts:

1) the future is independent from the past, given the present (d-separation)
2) the actual state is all we need for the future
3) the environment S_t^e is Markov
4) the history H_t is Markov

Environment Observability

We can distinguish two kinds of environment observability:

1) Full Observability

In this case, the agent can observe directly the environment state, so it has all the information it might need.

$$O_t = S_t^a = S_t^e$$

This is defined formerly as a **Markov Decision Process (MDP)**

2) Partial Observability

In this other case, the agent indirectly observes the environment: $S_t^a \neq S_t^e$.

Moreover, the agent needs to create its own state representation S_t^a.

RL Agent Components: Policy

The first component of a RL Agent is the **policy π. It's a function which describes the agent behavior**, so it's a map from the state s to the action a.

We could have two types of policies:

1) Deterministic policy $a = \pi(s)$
2) Stochastic policy $\pi(a|s) = P(A_t = a \mid S_t = s)$

RL Agent Components: Value Function

The **value function v is a predictor of future rewards**. It's used to <u>evaluate the goodness</u> of state and so it's used to select actions.
It's defined as the expectation on the discounted sum of future rewards given the actual state:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s]$$

where γ is a **discount** value.

RL Agent Components: Model

A **model** predicts what the environment will do next, predicting the next state s' and the next reward R_s^a.
Mathematically speaking, **the model calculates the probability of being in a certain next state** s' (at the next time $t+1$) **given the actual state** s at time t **and the action** a performed at the time t.

$$P^a_{ss'} = P(S_{t+1} = s' \mid S_t = s, A_t = a)$$

Similarly, **it predicts the next reward given the actual state s and the actual action a.**

$$R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

We can distinguish different RL scenarios depending on the fact if the model is known or not: in particular we can have model-free RL where we ignore the model and we have to use some kind of sampling or simulation in order to estimate rewards, or we can have model-based RL where we can predict the future rewards (and we will pick the actions in way to maximize that reward).

RL Problems: Learning VS Planning

We can differentiate Reinforcement Learning from Planning.
In a RL scenario, the agent doesn't know a-priori the environment: the agent has to interact with environment in order to improve its own policy π.

Instead, **in a Planning scenario, the model** ($P^a_{ss'}$) **is known** and the agent performs computation based on that model, in order to improve its own policy π.

RL Problems: Exploration VS Exploitation

The RL scenario follows the trial-and-error process to obtain a good policy, using the experience of the environment performed by the agent.

Here we have two possible stages that the agents can perform:

1) **Exploration:** how the agent can find more informations about the environment
2) **Exploitation**: how the agent can better use the information it already have (in order to maximize the reward)

There is a tradeoff between them.

RL Problems: Predict VS Control

A RL agent can perform two main tasks:

1) **Predict: the expected total reward given a fixed state**
2) **Control: optimize the policy** in order to maximize the expected total reward from any given state with that optimal policy

Markov Decision Process

Markov Decision Process formally describes a fully observable environment (so they work under the assumption of full observability).
Almost every RL problem can be formalized as a MDP.

A MDP is a **Markov chain** with rewards and actions, and where every state is Markov.

Formally, a MDP is a tuple (S,A,P,R,γ),
where:

- S is a finite set of **states**
- A is a finite set of **actions**
- P is the **state transition matrix** which tells us, given that we are in state s, what the probability the next state s′ will occur if we perform the action a at time t.

  $$P^{a}_{s\,s'} = P(S_{t+1} = s' | S_t = s,\ A_t = a)$$

  Note: the future state depends only on the current one (Markov First-Order Ass)
- R is the **reward function** such that

  $$R^{a}_{s} = \mathbb{E}[R_{t+1} | S_t = s,\ A_t = a]$$
- γ is a **discount factor**

Return G_t

The return G_t is the **total future discounted reward starting from the time** t.

$$G_t = R_{t+1} + \gamma^1 R_{t+2} + \ldots = \Sigma^{\infty}_{k=0} \gamma^K R_{t+k+1}$$

γ regulate our choice between an immediate reward ($\gamma \simeq 0$) or a delayed reward ($\gamma \simeq 1$).

Why should we use the discount term? It is mathematically convenient to discount rewards and it avoids infinite returns in cyclic Markov processes. Since we have uncertainty about the future may not be fully represented.

Policy

A policy π is a **distribution of execute a particular action given a state**:

$$\pi(a|s) = P(A_t = a | S_t = s)$$

The policy defines the behavior of an agent and it depends only on the current state (so the policy is defined as Markov).
It's worth noting that policies are **stationary** which means that they are **time-independent**.

In other words, a policy is simply a mapping from states to probabilities of selecting each possible action.

State-value function

Once we defined G_t we can rewrite the state-value function as:

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s]$$

The value function $v_\pi(s)$ of a state s under a policy π is the **EXPECTED** return when we start acting from the state s and following π

Action-value function

After the state-value function, we can define the **action-value** function $q_\pi(s,a)$ which is the **EXPECTED** return when we start acting from the state s and following π and **taking the action** a.

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s]$$
$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a]$$

Bellman Equations : Decomposing value functions

The value function uses the return $G_t = R_{t+1} + \gamma^1 R_{t+2} + \dots$.

This implies that we can separate the value function into two parts:

1) The **immediate reward:** R_{t+1}
2) The (recursive) **value function of successor state** in the future discounted by γ : $\gamma v(S_{t+1})$

Let's explicit every passage:

$$v(s) = \mathbb{E}[G_t|S_t = s]$$
$$\text{(def of } G_t\text{)} \quad = \mathbb{E}[\Sigma^{\infty}_{k=0}\gamma^K R_{t+k+1}|S_t = s]$$
$$\text{(take out } R_{t+1}\text{)} = \mathbb{E}[R_{t+1} + \Sigma^{\infty}_{k=1}\gamma^K R_{t+k+1}|S_t = s]$$
$$\text{(take out } R_{t+2}\text{)} = \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \Sigma^{\infty}_{k=2}\gamma^K R_{t+k+1})|S_t = s]$$
$$\text{(def of } G_{t+1}\text{)} = \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s]$$
$$\text{(def of } v_\pi\text{)} = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$
$$\text{(prop. of } \mathbb{E}\text{)} = \mathbb{E}[R_{t+1}|S_t = s] + \gamma\,\mathbb{E}[v_\pi(S_{t+1})|S_t = s]$$

Note: the first term is the expected value of the **immediate reward** R_s and the second term is the expected value of the future rewards, which can be seen as the sum over all the value functions of all the possible future states s', multiplied by the probability of pass from the actual state s into that specific s': $\gamma\Sigma_{s'} P_{ss'} v(s')$

$$v(s) = R_s + \gamma\Sigma_{s'} P_{ss'} v(s') = \mathbb{E}[R_{t+1} + \gamma v_\pi(s_{t+1})|S_t = s]$$ [2]

[2] Quest'uguaglianza dovrebbe derivare dalla def. stessa di Expectation, ma potrebbe essere un errore.

Decomposing v with q and vice versa

Recall:

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}[R_{t+1} + \gamma v_\pi(s_{t+1}) \mid S_t = s]$$
$$q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a]$$
$$\pi(a|s) = P(A_t = a|S_t = s)$$

Once we defined the action-value q, we can use it to decompose the state-value v.

The state-value $v_\pi(s)$ can be seen as the sum of the possible action values performed on all the possible actions weighted by the policy:

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(s_{t+1}) \mid S_t = s] = \Sigma_{a\in A}\pi(a|s)\, q_\pi(s,a)$$

In this way we can see $v_\pi(s)$ depending by $q_\pi(s)$.

We can do the same vice versa, using v to decompose q.

$$q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$
$$= R^a_s + \gamma \Sigma_{s'} P^a_{ss'}\, v_\pi(s')$$

This is true because $q_\pi(s,a)$ defined as the expected return when we start acting from the state s and following π and taking the action a, which gives us an immediate reward R^a_s (due to being in the state s) plus all the expected return of being the future states s' (aka the value function) weighted by the probability of achieve that state s' starting from s and performing the action a.

One more step of nesting

Recall: we can write q using v and vice versa

$$v_\pi(s) = \Sigma_{a\in A}\pi(a|s)\, q_\pi(s,a)$$
$$q_\pi(s,a) = R^a_s + \gamma \Sigma_{s'} P^a_{ss'}\, v_\pi(s')$$

we can obtain:

1) the expected return of being in a state reachable from s through action a and continue following the policy π (aka we can plug $q_\pi(s,a)$ into $v_\pi(s)$)

$$v_\pi(s) = \Sigma_{a\in A}\pi(a|s)\,(R^a_s + \gamma \Sigma_{s'} P^a_{ss'}\, v_\pi(s'))$$

2) the expected return of any action a' taken from states reachable from s through action a (and then follow policy)

$$q_\pi(s,a) = R^a_s + \gamma \Sigma_{s'} P^a_{ss'}\, (\Sigma_{a'\in A}\pi(a'|s')\, q_\pi(s',a'))$$

Bellman Equations direct solution (closed form)

Bellman equations can be described as a linear system with a direct solution.
System:

$$v_\pi = R^\pi + \gamma P^\pi v_\pi$$

Solution:

$$v_\pi = (I - \gamma P^\pi)^{-1} R^\pi$$

Definition of optimal value functions

Using the previous equations for v_π and v_π, we can calculate which state and which action gives us the best reward (we still are under the assumption of the full observability of the environment).
Hence, we can define the **optimal** state-value and the **optimal** action value as:

- The **optimal state-value** $v_*(s)$ is the maximum state-value function over all the policies

$$v_*(s) = \max_\pi v_\pi(s)$$

- The **optimal action-value** $q_*(s,a)$ is the maximum action-value function over all the policies

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

If we have q_* and v_* we have the best performance for the MDP.

Definition of optimal Policy

For the same MDP we could have different policies and we can sort them using the value function. The partial order is defined in this way:

$$\pi \geq \pi' \quad if\ v_\pi(s) \geq v_{\pi'}(s) \quad \forall s$$

The optimal policy π_* is always better or equals to the other and it will achieve the optimal state value v_* and an optimal action-value q_*.

We can find the optimal policy **maximizing over** $q_*(s,a)$, so if we know q_* we can easily calculate π_* in this way

$$\pi_*(a|s) = \begin{cases} 1 & if\ a = arg\max_{a \in A} q_*(s,a) \\ 0 & otherwise \end{cases}$$

Bellman Optimality Equations

Now, we can define the **Bellman Optimality Equations** as the optimal value functions defined recursively:

$$v_*(s) = \max_{a \in A} q_*(s,a) = \max_{a \in A} R^a_s + \gamma \Sigma_{s' \in S} P^a_{ss'} v_*(s')$$

$$q_*(s,a) = R^a_s + \gamma \Sigma_{s' \in S} P^a_{ss'} v_*(s') = R^a_s + \gamma \Sigma_{s' \in S} P^a_{ss'} \max_{a' \in A} q_*(s',a')$$

The optimality equation that we obtained is **non-linear** so we do not have a closed form solution. For this reason we use an iterative method like Value Iteration, Policy Iteration, Q-Learning or SARSA (the last two get rid of the model of the environment).

MDP Extensions

Infinite MDPs Countably infinite state and/or action spaces.
Continuous state and/or action spaces: closed form for linear quadratic model (LQR).
Continuout time: requires partial differential equations.

POMDP Partially Observable MDPs are MDPs with hidden states: a Hidden Markov Model with actions.
A POMDP is a tuple $(S, A, O, P, R, Z, \gamma)$

S is a finite set of states

A is a finite set of actions a

O is a finite set of observations

P is a transition matrix such that $P^a_{ss'} = P(S_{t+1} = s' \mid S_t = s, A_t = a)$

R is a reward function such that $R^a_s = E[R_{t+1} \mid S_t = s, A_t = a]$

Z is an observation function

$\gamma \in [0, 1]$ is a discount factor

Belief State A history H_t is a sequence of actions, observations and rewards

$$H_t = A_0 O_1 R_1, \ldots, A_{t-1} O_t R_t$$

A belief state $b(h)$ is a distribution over states conditioned on the history h

$$b(h) = [P(S_t = s_1 \mid H_t = h), \ldots, P(S_t = s_n \mid H_t = h)$$

Model-based Planning

Recall: in a **Planning scenario, the model** ($P^{a}_{ss'}$) **is known** and the agent performs computation based on that model, in order to improve its own policy π.

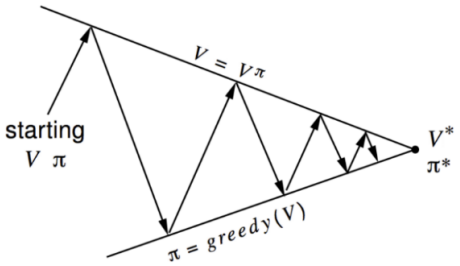
Backup Diagram

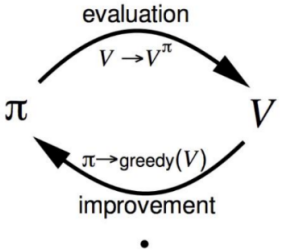
https://towardsdatascience.com/all-about-backup-diagram-fefb25aaf804

Deeping: "Acting Greedly"

In RL, "**acting greedly**" is usually the short for "acting greedly with respect to the value function". It **refers to executing the action that would give us the highest immediate rewards** (ignoring possible future major rewards).

Policy Iteration





Given a policy π, we can iterate two steps in order to improve our policy:

1) **Policy Evaluation:**
   we evaluate the policy, updating every state $v(s)$

2) **Policy Improving:**
   we update the policy acting greedly with respect to v_{π}

$$\pi' = greedly\,(\pi)$$

This process will always converge to the optimal policy π_*

(Iterative) Policy Evaluation

The task we want to solve is to **evaluate a policy** π and we could perform this by applying iteratively the Bellman Expectation, calculating the state-value function iteratively until we arrived to v_{π} for each state.

For each iteration k and for each state $s \in S$, we update every state-value function $v(s)$ getting $v(s')$

$$v_k(s) \rightarrow v_{k+1}(s')$$

we start from an initialization of every v and then we update them using the Bellman expectation.

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \ (R^{a}_{s} + \gamma \sum_{s' \in S} P^{a}_{s s'} v_{k}(s'))$$

Since it's exactly the Bellman expectation, we can rewrite it in a closed form:

$$v_{k+1} = R^{\pi} + \gamma P^{\pi} v_{\pi}$$

This means that we can update each value function evaluating actions and each successor states.

This Iterative Policy Evaluation is clearly model-based because for every state, I am exploring all the possible actions and all the consecutive states. In order to do that I need the reward function and the transition function to be able to know with which probability I am going to get in the next state.

Policy Improving

Given a starting policy $\pi(s)$, **we can improve it by acting greedly**.

$$\pi'(s) = \arg\max_{a \in A} q_{\pi}(s, a)$$

Then, we can improve the values of each state, since we can update the state-value function with the action-value function and vice versa.

$$q_{\pi}(s, \pi'(s)) \geq q_{\pi}(s, \pi(s))$$

$$v_{\pi'}(s) \geq v_{\pi}(s)$$

when we achieve $q_{\pi}(s, \pi'(s)) = v_{\pi}(s)$ we can stop since we have reached the Bellman optimality, so $v_{\pi}(s) = v_{*}(s), \forall s$.

Value Iteration

One drawback of policy iteration is that it requires policy evaluation, which, if it's done iteratively converges exactly to v_{*}, but we might not have to wait for the exact convergence.

Indeed, we could stop the policy evaluation after k step without losing the convergence guarantees of policy iteration.

A particular case is when we stop after just one step (so, we update each state only once). This algorithm is called **value iteration**.

Since we now don't have to evaluate a policy, **we could have no a starting policy and we could act greedly**. In this way we move from an algorithm that **evaluates** a given policy to an algorithm that **finds** a policy, because the value function in the middle may not correspond to any policy.

Optimality Principle

Any optimal policy can be subdivided into two components:

- An **optimal first action** a^*
- An **optimal policy for successor states** s'

This means that the only thing we have to care about is the choice of the best possible action **now** and we can assume that in the future we will do the same thing.

(Deterministic) Value Iteration

Knowing $v_*(s')$ we can calculate $v_*(s)$ in just one step:

$$v_*(s) = \max_{a \in A} q_*(s,a) = \max_{a \in A} R^a_s + \gamma \Sigma_{s' \in S} P^a_{ss'} v_*(s')$$

So, starting from the final rewards and working backwards, we can update every state-value iteratively, applying the **Bellman optimality** (Note: it's the same formula)
We perform this for each iteration.

Policy VS Value Iteration : Recap

Policy Iteration:

- Used to evaluate a policy
- It uses the Bellman expectation
- Updates the state-value function v after that the policy evaluation is finished.

Value Iteration:

- Used to find a policy
- It uses the Bellman optimality
- Updates the state-value function v after each iteration
- There is no explicit policy (because I act greedly)

Model-free RL

So far, we assumed to know the whole Markov Decision Process, so we knew states, transitions, rewards and actions.
Now, we delete this assumption and so we have a **model-free** scenario where we don't know the model, and we don't know the environment and rewards.

Hence, **we have to adapt** the concepts of **prediction** and **control**:

- **Model-free prediction** : estimate the value function of an unknown MDP
- **Model-free control** : optimize the value function of an unknown MDP

Deeping: Episodes

An episode is a sequence of states, actions and rewards which ends with a terminal state.
Every episode is independent from the previous ones.

Deeping: Bootstrapping

Bootstrapping is when we estimate something based on another estimation.
An example of bootstrapping is the update of the v function, indeed we update v for the state s looking ahead to the $v(s')$.

$$v_{\pi}(s) = \Sigma_{a \in A} \pi(a|s)\, q_{\pi}(s,a)$$

$$q_{\pi}(s,a) = R^{a}_{s} + \gamma \Sigma_{s'} P^{a}_{s\,s'}\, v_{\pi}(s')$$

Monte-Carlo RL

A Monte-Carlo method learns directly from episodes of experience, it's a model-free method so we have no knowledge about the rewards and the transitions.

It learns from **complete** episodes, so we can use it only when we know that every episode terminates, and since it learns from complete episodes it's not a bootstrapping method.

Monte-Carlo methods are a way to solve RL problems based on an average sample, so they sample and return an average reward for each state.

Monte-Carlo Policy Evaluation

Given a policy π, we can use MC methods to learn v_{π} from a set of episodes.

Let $G_t = R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \dots$ be the total discounted reward and recall that the value function is just the expected return:

$$v_{\pi}(s) = \mathbb{E}[\, G_t \mid S_t = s \,]$$

MC policy evaluation updates the array of estimation of the value function $V(S)$ applying an **incremental mean after each episode**.

Note: $V(S)$ should be the Random Variable that estimate the value function $v(S)$.

The pseudo-algorithm of a MC policy evaluation is:

- For each state S_t
  - We increment a counter $N(S_t)$
  - We use the incremental mean to update $V(S_t)$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

In non-stationary problems track a running mean.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

Temporal Difference Methods

Another strategy to learn the v_π function from episodes in a model-free scenario is the **Temporal Difference Learning**; the main difference with Monte-Carlo is that TD learns from **incomplete** episodes with a **bootstrapping** technique.

Indeed, the TD(0) method updates $V(S_t)$ toward the **estimated return** (called **TD TARGET**).

$$V(S_t) \leftarrow V(S_t) + \alpha(\underbrace{R_t + \gamma V(S_{t+1})}_{\text{TD target}} - V(S_t))$$

$$\text{TD error } \delta_t$$

This is a bootstrapping method because TD target is an estimate of the true value of $V(S_t)$, so we are using a part of $V(\cdot)$ to update another part of $V(\cdot)$.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

MC vs TD (pt.1) : waiting time

TD can learn before knowing the final outcome, learning step-by-step. Instead, **MC must wait until the end of the episode.**
TD can learn even when there is no final outcome (for example: incomplete sequence) and it can work in non-terminating environments, while MC only works for complete sequences and terminating environments.

Bias-Variance Tradeoff

Recall that :

Return: $G_t = R_{t+1} + \gamma^1 R_{t+2} + \dots$.

True TD target: $R_{t+1} + \gamma^1 v_\pi(S_{t+1})$

Both return and the true TD target are **unbiased estimate** of $v_\pi(S_t)$

The "usual" TD target $R_{t+1} + \gamma^1 V(S_{t+1})$, instead, is a **BIASED estimate** of $v_\pi(S_t)$
The TD Target has a lower variance because it depends only on a single action, reward and transition, while the return depends on many actions, transitions, and rewards.

MC vs TD (pt.2): Bias-Variance

MC has high-variance and no bias, it's not sensible to initial values and it has good convergence performances.
TD has low-variance with some bias, usually it's more efficient than MC but it's sensible to initial values, and TD(0) converges to $v_\pi(s)$

Batch MC and TD

If we perform an infinite number of experience, both MC and TD converge to $v_\pi(s)$

$$V(s) \rightarrow v_\pi(s)$$

but if we limit the number of experiences, MC and TD give us different results.
In particular, MC converges to the **minimum mean-square error** [A/N: that is an estimation method which minimizes the mean square error (MSE)]

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} \left(G_t^k - V(s_t^k) \right)^2$$

Instead TD(0) converges to the **maximum likelihood Markov model** that is the MDP $< S, A, P, R, \gamma >$ that best fit the data

$$\hat{P}^a_{s\,s'} = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \mathbf{1}(s_t^k = s,\, a_t^k = a,\, s_{t+1}^k = s')$$

$$\hat{R}^a_s = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \mathbf{1}(s_t^k = s,\, a_t^k = a)\, r_t^k$$

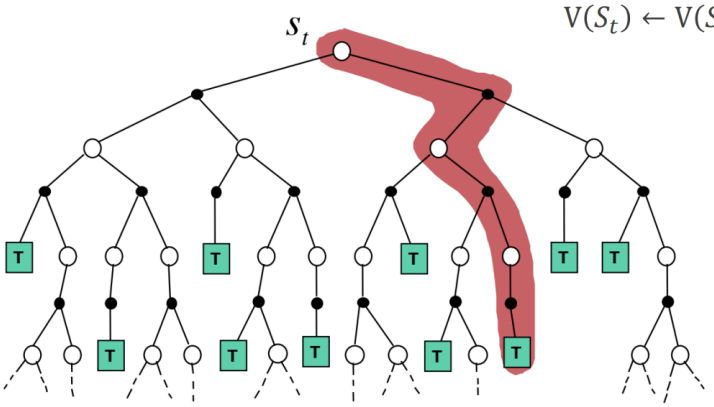
MC vs TD : Markov property

TD exploits Markov property (and so it's usually more efficient in Markov environments) while **MC does not exploit Markov property** (and so it's usually more effective in non-Markov environments)

Graphical representation

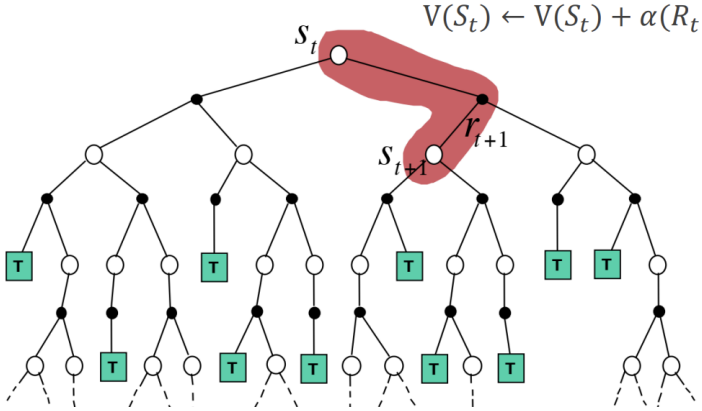
MC Update

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$



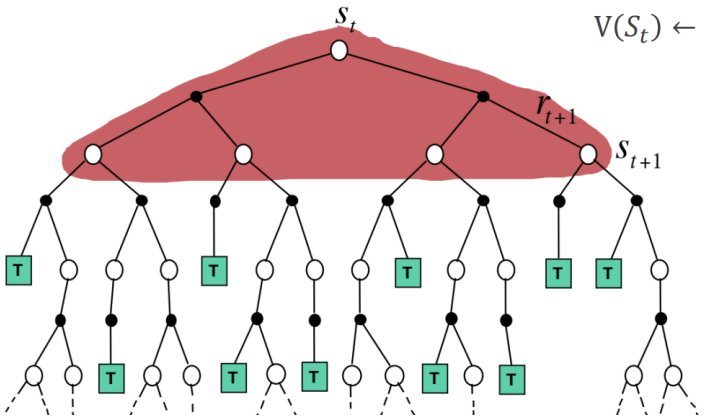
TD Update

$$V(S_t) \leftarrow V(S_t) + \alpha(R_t + \gamma V(S_{t+1}) - V(S_t))$$



Dynamic Programming

$$V(S_t) \leftarrow \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})]$$



N-Steps

Recap:

> Consider estimating v_π from sample episodes generated using π.
>
> MC performs an update for each state based on the rewards from that state until the end of the episode.
> Instead, TD(0) updates only the value of the current state using the value of the next state as proxy [3] of the next reward (bootstrapping)

But, we could perform the update of $V(S_t)$ after n steps.

So, the return will became:

$$G_t^{\ n} = R_{t+1} + \gamma R_{t+2} + \ .. \ + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

and we update $V(S_t)$ at the same way as before in TD(0)

$$V(S_t) \leftarrow V(S_t) + \alpha(G^n_{\ t} - V(S_t))$$

Intro of Eligibility traces

Eligibility traces are a popular mechanism in RL, the " λ" in the TD(λ) refers to the use of an eligibility trace in the algorithm.
Eligibility traces unify and generalize TD and MC methods with good computational costs.
The mechanism is a **short-term memory vector with a parallel weight memory**.

The rough idea is: <u>when a component of w_t is used to estimate a value then the corresponding component z_t starts to **fade** away.</u>

<u>The fade is given by the parameter</u> $\lambda \in [0, 1]$ which controls how fast the exponential weighting falls and so how far into the future the λ-return algorithm looks (see next paragraph).

It can incorporate two main heuristics together:

- **Frequency heuristic**: assign credit to most frequent states
- **Recency heuristic:** assign credit to most recent states

[3] Per "proxy" si intende quello che si usa per approssimare

λ-Return (Forward)

Given a fixed number of iteration (or even infinite) we can repeat the n-step, having several G_t^n.

The λ-return so it's a compounded sum of all these G_t, weighted using $(1-\lambda)\lambda^{n-1}$.

So our new return will be:

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G^n_t \qquad \lambda \in [0,1]$$

Note the λ as apex.

Then we could use G_t^{λ} as usual to update the array of estimation of the state functions v

$$V(S_t) \leftarrow V(S_t) + \alpha(G^{\lambda}_t - V(S_t))$$

Problems with forward

The λ-return algorithm (or simply forward) provides the theory that the idea to update $V(S_t)$ after a fixed number of step can be done, but it's not feasible because it needs to wait until the whole compounded error G^{λ}_t is calculated and this means also **the possibility to wait until infinity.**

For this reason we introduce the backward strategy, also called $TD(\lambda)$, which updates the $V(S_t)$ at every step from incomplete sequences using an eligibility trace.

TD(λ) (Backward)

The forward view provides the theory but it's unfeasible, the TD(λ) backward view provides the practical mechanism. In TD(λ) algorithm the eligibility trace vector is initialized to zero at the begin of the episode and it's updated state-by-state, with a fade part given by γλ

$$E_0(s) = 0$$
$$E_t(s) = \gamma\lambda E_{t-1}(S) + \mathbf{1}(S_t = s)$$

A/N: Note the recursive backward term E_{t-1}

TD(λ) algorithm updates the value $V(S_t)$ for every state s in proportion to the TD error δ_t and the eligibility trace $E_t(s)$

TD Error: $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$

Update : $V(s) = V(s) + \alpha\delta_t E_t(s)$

Model Free RL Control : On-Policy VS Off-Policy Learning

Acting greedy in a model free world

The policy improvement needs to be executed carefully, because it a model free scenario we have no idea about the $P^{a}_{s\,s'}$ factor.

So, we need to maximize the action to the $Q(s,a)$ function, choosing the action with maximize the Q

$$\pi'(s) = \arg\max_{a \in A} Q(s,a)$$

ε-greedy exploration

Acting greedy is not smart because with an all-zero initialization, the agent will always choose one single action because after the first reward, we have all zeros except for the action we just made, so the agent will re-execute the same action forever.

In order to solve this problem we can rewrite the definition of $\pi(a|s)$ using a ϵ parameter.

Indeed, now, with a probability of ϵ we take a random action, and wit a probability of $1 - \epsilon$ we take the greedy action.

We don't lose any theoretical convergence using ε-greedy instead of simple greedy.

On-Policy and Off-Policy Learning

Recall:

- A policy π describes the distribution of perform a certain action given the state
- In the RL framework, we can learn two things:
    - Prediction: evaluate the $Q(s,a)$ function, predicting the sum of future discounted rewards.
    - Control: update our policy in order to achieve the maximum reward

On-policy and off-policy are two ways to perform the control, so we will update our policy.

In a nutshell, we can say that:

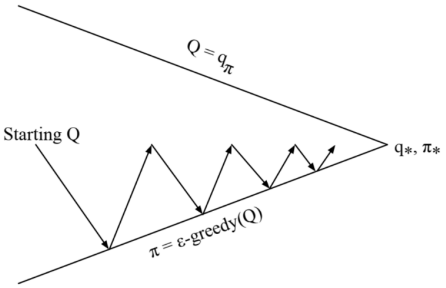
- On-policy learning algorithms learn $Q(s,a)$ using action from the **current policy** $\pi(a|s)$ → ***learn on the job***
    - SARSA algorithm
- Off-policy learning algorithms learn $Q(s,a)$ using a **different policy** (even a random one) → ***learn over someone's shoulder***
    - here we don't need to know a-priori $\pi(a|s)$
    - Q-Learning algorithm

In both cases, once we found the Q function, we can act ε-greedy (it will be defined shortly) in order to get the policy:

$$\pi = \text{ε-greedly}(Q)$$

We need to update Q, instead of V because we are in the model free scenario.

On-Policy Learning



As seen before, we can iterate two steps in order to obtain V^* and π^*

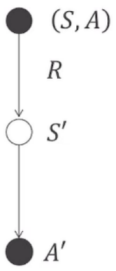
1) **Policy evaluation:** in order to estimate v_π, using MC or TD, using the policy π
2) **Policy improvement:** generating a better policy π' acting ε-greedly, using the values V

SARSA

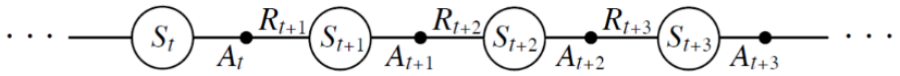
SARSA is an on-policy algorithm which uses TD Learning for policy evaluation.
For each time-step we perform

1) Policy evaluation using TD
2) Policy improvement using ε-greedly

Then, for each time-step we update **ALL** the states.

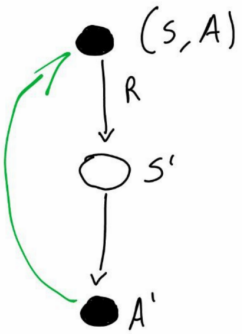


SARSA means : State → Action → Reward → State → Action



How SARSA updates Q(s,a)

Given a state s and an action a, the $Q(s, a)$ is updated adding to the old value the difference between the TD error and the actual $Q(s, a)$.
We need to update Q, instead of V because we are in the model free scenario.



$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$$

Note that A' will be picked from the ε-greedly policy that we have, so if we know that in the state S, we will perform the action A, we know that we will arrive into S' and there acting ε-greedly we know that we will perform A' .

Then A' will became the new A, and we go ahead in this way.

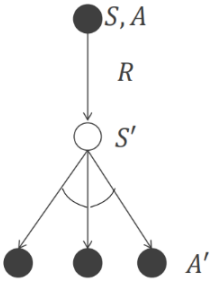
Off-Policy Learning

An off-policy learning algorithm is able to learn $Q(s,a)$ of a target policy $\pi(a|s)$ using a different policy, called behaviour policy $\mu(a|s)$ from where we sample states, actions and rewards.

$$S_1, A_1, R_1, S_2, \dots, \sim \mu$$

This is important because we learn from imitation and we can re-use experience generated from old policies.

Q-Learning



The Q-Learning is a TD learning algorithm that uses the off-policy strategy to evaluate $Q(a,s)$ in a model free world. It's a TD learning algorithm because the target contains the estimate of the next step (so, it's a bootstrapping method).
The next action is chosen using the behaviour policy:

$$A_{t+1} \sim \mu(\cdot, S_t)$$

But we update the actual $Q(S_t, A_t)$ sampling it from the target policy

$$A' \sim \pi(\cdot, S_t)$$

The action we will execute is always that ones we sample from the behaviour policy, we use A' just to evaluate $Q(S,A)$.

Both behaviour and target policies need to be improved:
Q-Learning act greedly while improve the target policy maximizing the Q function selecting the best action.

$$\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$$

Instead, Q-Learning acts ε-greedly with respect to $Q(s,a)$.

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R_{t+1} + \max_{a'} \gamma\, Q(S',a') - Q(S,A))$$

There is a nice theorem which tells us that Q-Learning converges to the optimal policy.
Note: since in the Q update function there is a maximization over the next actions (that will be used only for selecting the target, all of them will not be executed) then this is an implementation of the **Bellman optimality.**

Value-Function Approximation

In non-trivial scenarios, the number of states is really huge and so far $V()$ and $Q()$ were just lookup tables, if we have a lot of states we cannot store all of them in memory because it would require too much memory and furthermore we have slow learning if we want to learn each state individually.

In order to deal to all of these problems we need a new approach which allows us to **approximate** value functions:

$$\hat{v}(s; w) \approx v_\pi(s)$$

$$\hat{q}(s, a; w) \approx q_\pi(s, a)$$

where w are parameters we found (using MC or TD learning)

The approximation can be made in different ways but usually we use a NN.

SGD in Value Approximation

Our goal is to find parameters w that minimize the error between the approximated value $\hat{v}(s; w)$ and the true value function $v_\pi(s)$.

We don't know $v_\pi(s)$, hence, we want a sequential and incremental algorithm to approximate it, and why not **SDG**?

SDG is an incremental method, where we update our values as soon as we perform an action within the environment.

$$J(w) = \mathbb{E}_\pi[\,(v_\pi(S) - \hat{v}(S; w))^2\,]$$

The gradient solution is given from:

$$\Delta w = -1/2\,\alpha\,\nabla_w J(w)$$

The expectation of $J(w)$ runs under the policy, which means that the error is computed on sample generated by the policy, but since we have just one sample (because the batch_size of SDG is 1) the expectation disappears when we calculate the Δw:

$$\Delta w = \alpha(v_\pi(S) - \hat{v}(S; w)) * \nabla w\ \hat{v}(S; w)$$

where :

- α is the step size
- $\nabla w\ \hat{v}(s; w)$ is the first derivative with respect to weights

But we still don't know v_π!

Feature Vector State

A state S can be represented by a feature vector

$$x(S) = [x_1(S), \ldots, x_n(S)]$$

Using this strategy we can include the old lookup tables into this new approach using the indicator function.

Linear Value Function Approximation

One of the most important special cases of function approximation is when we just use a **linear function** with weights.
In this case, linear methods approximate the **state-value function** using the inner product between w and $x(S)$, where the latter term is the feature vector for the state S.

$$\hat{v}(s; w) = w^T \cdot x(S)$$

In this way the objective function is quadratic in w and the SDG is able to reach the global optimum.
Now that we have the model of the parameter we can rewrite the Δw:

$$\nabla w \, \hat{v}(S; w) = x(S)$$
$$\Delta w = \alpha(v_\pi(S) - \hat{v}(S; w)) * x(S)$$

We are shifting from reinforcement learning to supervised learning

Incremental Prediction

We still have a problem in the Δw equation: we don't know the true $v_\pi(S)$.

In practice, we substitute the target (we use these methods as oracle) using:

- **MC -** the target is the return G_t

  $$\Delta w = \alpha(v_\pi(S) - \hat{v}(S; w)) * x(S)$$
  $$= \alpha(G_t - \hat{v}(S; w)) * x(S)$$

  Recall that G_t is an **unbiased** and noisy sample of the true value $v_\pi(S)$.

  This evaluation converges to a **local** optimum.

- **TD(0) -** the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}; w)$

  $$\Delta w = \alpha(v_\pi(S) - \hat{v}(S; w)) * x(S)$$
  $$= \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}; w) - \hat{v}(S; w)) * x(S)$$
  $$= \alpha \, \delta \, x(S)$$

Recall that G_t is a **biased** sample of the true value $v_\pi(S)$.

This evaluation converges to a **global** optimum.

- **TD(λ) -** <u>the target is the λ-return</u> G_t^λ

  the λ-return Gt is also a biased sample of the true value $v_\pi(S)$.

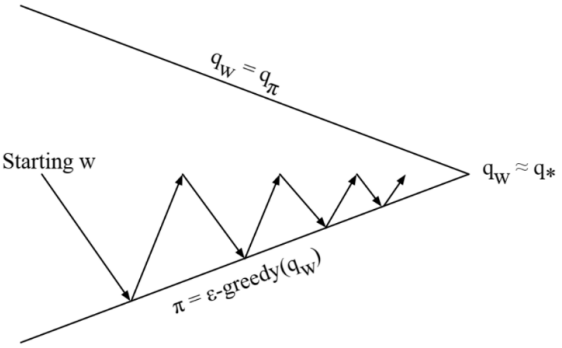
  The forward view linear TD(λ) is

  $$\Delta w = \alpha(v_\pi(S) - \hat{v}(S;w)) * x(S)$$
  $$= \alpha(G_t^\lambda - \hat{v}(S;w)) * x(S)$$

  but the forward view is just a theoretical view, so we have to implement the backward view linear TD(λ)

  $$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1};w) - \hat{v}(S_t;w)$$
  $$E_t = \lambda\gamma E_{t-1} + x(S_t)$$
  $$\Delta_w = \alpha\delta_t E_t$$

Incremental Control



We need to adapt the control task in order to use the value approximation.

Recall that we are still in the model free scenario and then we have to approximate q function (using MC, TD(0), TD(λ)) in order to perform the policy evaluation.

Note: since we are in model-free scenario, we have no clues about $P^a_{ss'}$ so we cannot evaluate the v function.

Then, as usual we have the two steps:

- **Policy evaluation**

  which is an **approximation** of the evaluation of the policy $\hat{q} \simeq q$
- **Policy improvement**

  which is simple an ϵ-greedy policy improvement.

So, we want approximate q.

The good news is that everything is pretty the same of the prediction task, we just need to change $\hat{v}$ with $\hat{q}$, and we can still use the SGD in order to approximate $q_\pi(S,A)$.

In formulae:

MSE to minimize: $J(w) = \mathbb{E}[\,(q_\pi(S,A) - q(S,A,w)\,)^2]$

In general the SGD would will find

$$\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A;w))\,\nabla_w \hat{q}(S,A,w)$$

Using feature action-states (a feature representation of the couple (state S, action A)) that we denote as $x(S,A)$ and using the linear approximation we will have:

$$\nabla_w \hat{q}(S,A,w) = x(S,A)^T w$$

$$\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A;w))\,x(S,A)^T w$$

We still have the same problem as before: we don't know the true $q_\pi(S,A)$.

So, again, we can use MC/TD(0)/TD(λ) [assuming the linear approximation] :

- **MC -** <u>the target is the return</u> G_t

$$\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A;w)) * x(S)$$
$$= \alpha(G_t - \hat{q}(S,A;w)) * x(S)$$

- **TD(0) -** <u>the target is the TD target</u> $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}; w)$

$$\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A;w)) * x(S)$$
$$= \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}; w) - \hat{q}(S,A;w)) * x(S)$$

- **Forward TD(λ) -** <u>the target is the action-value λ-return</u> q_t^λ

$$\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A;w)) * x(S)$$
$$= \alpha(q_t^\lambda - \hat{q}(S,A;w)) * x(S)$$

- **Backward TD(λ) -** <u>the target is the action-value λ-return</u> q_t^λ

$$\delta_t = R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}; w) - \hat{q}(S_t, A_t, w)$$
$$E_t = \lambda\gamma E_{t-1} + x(S)$$
$$\Delta w = \alpha\,\delta_t E_t$$

Deadly Triad

- Function Approximation
- Bootstrapping
- Off Policy Training

As soon as we have all three of them we are basically dead because we lose guarantees on the convergence.

In other words, using TD Learning we could have no convergence: in the case of On-Policy we have no convergence when we approximate with a nonlinear function (and TD)
Instead, in Off-policy we have no convergence even using the linear function.

MC Learning instead, always converges.

Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Batch Reinforcement Learning

SGD is useful but we can achieve better performance on value approximation with batch mode. Moreover, we have the problem that, using SGD, the agent lives an experience from which it chooses the action and after that it forgets the experience because it needs to focus only on the next one.

SGD with experience replay

We want to perform approximations over more than one experience. In order to do this, we have to store experiences into a replay memory.
For this reason we can construct a dataset of ⟨ *state*, *value* ⟩ pairs and we can use this dataset into a Gradient Descent method with a batch size more than one.

Deep Q-Networks

Deep Q-Network is a model that uses the **experience replay** technique, it uses a neural network to approximate the Q function.
The actions are chosen with an ϵ-greedy policy and we store into the replay memory the tuple $(s_t, a_t, r_{t+1}, s_{t+1})$.
The minibatch is feeded with some random samples from the replay memory.

Linear Least Squares

Using the linear value function approximation and the experience replay led us to find the least squares solution, but this requires several iterations.
Why do we need to perform these iterations if we can apply the least squares solution directly?

Linear Least Squares prediction

We can rewrite the problem using the least squares method but again we don't know (in the prediction task) the value for the true v_t.

Guess!
Yes, we will use MC/TD, in particular:

- Least Squares Monte-Carlo **(LSMC)**
- Least Squares TD **(LSTD)**
- Least Squares TD(λ) **(LSTD(λ))**

In any case we can get a closed form and we can solve it directly.
Using LS instead of linear we gain the convergence for TD learning in the off policy learning.

Linear Least Squares control

As usual for the control task we have two phases:

- Policy evaluation
- Policy improvement

In the case of LS Control, the policy improvement is performed by acting with a greedy policy, while the policy evaluation is performed by the **least squares Q-Learning**.

The idea of LS Q-Learning is the same as Q-Learning, so the action that is executed is different from the action used to assess.

Policy-based RL

So far we approximate value and action-value function using parameters and then we generate policy from the value function (acting ϵ-greedly).

Now we can try a different approach parametrizing the policy.

$$\pi_\theta(s, a) = P(a|s, \theta)$$

Recall that a policy is a distribution of actions given states. So, we will add a parametrization term into the policy, hence we add a parametrization term to a distribution.
(We are still in a model-free scenario).

Instead of value-based RL, **policy-based learning has better convergence properties**. Theoretically.
In practice, this kind of learning has a lot of **variance**, so we typically converge to a local minima instead of the global optimum.

Another advantage is that learning the policy is effective in a high-dimensional or continuous action space and using this technique we can learn stochastic policies.

Policy objective functions

So, our new task is the following: we start from a given policy $\pi_\theta(s, a)$ and we want to fine tune the parameters θ in order to improve the policy itself.
Since we are in a learning scenario, **we do need a metric that tells us if the policy is a good one (we will optimize this value)**.

In order to measure the quality of a policy $\pi_\theta(s, a)$ we have two ways, depending on whether one is in an episodic or continuous environment.

Episodic Environment

This is a simple case because when the episode terminates, we can compute the actual return for each state (MC-style).
But if we want to assess how good a policy is, we don't need to check every return from every state, **we can assume that a good policy has a good value for just the very first state S_1.**
This means that if the value of being in the starting state is equal to the expected return.
In other words, we want to maximize:

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1].$$

Continuing Environment

In this case we have two approaches:

Average value

We can measure the quality of the policy summing up all the expectation of all the states multiplied by a stationary distribution $d^{\pi_\theta}(s)$ which is the frequency of visit (how often I visit a state)

$$J_{\bar{V}}(\theta) = \Sigma_s d^{\pi_\theta}(s)\, V^{\pi_\theta}(s)$$

Average reward

In average value strategy we need to wait until the value is calculated, and it can require a lot of time.
We can shift from values to rewards, using the sum of all the reward weighted by the probability of being in a state s after the state action a [this is the policy $\pi(s,a)$].
Then the sum is weighted by the same stationary distribution as before

$$J_{\bar{R}}(\theta) = \Sigma_s d^{\pi_\theta}(s)\ \Sigma_s \pi_\theta(s,a) R^a{}_s$$

Policy gradient (theorem)

Given a differentiable policy $\pi_\theta(s,a)$ any policy objective function $J_1(\theta)$, $J_{\bar{V}}(\theta)$ or $J_{\bar{R}}(\theta)$ the gradient is the expectation of the gradient of the logarithm of the policy (this last value is called "score") multiplied by the Q function.

$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\ \nabla_\theta log\ \pi_\theta(s,a)\ Q^{\pi_\theta}(s,a)\ ]$$

Policy gradient algorithms search for a local maximum in $J(\theta)$ using the gradient ascending with respect to θ and $\nabla_\theta J\ (\theta)$ is called **policy gradient**.

Score function

The score function is the logarithm of the policy and we can use it to rewrite the gradient of the policy with respect to θ

$$\nabla_\theta \pi(s,a) = \pi_\theta(s,a)\, \nabla_\theta log\ \pi_\theta(s,a)$$

Let's focus on how to calculate $\nabla_\theta log\, \pi_\theta(s,a)$

Softmax Policy (episodic world)

Now, we have a way to measure how good a policy is and a way to optimize the policy, using the policy gradient theorem; now we have to choose how to parametrize the policy.

If we assume an **episodic world** we can parametrize the π distribution using a linear combination of features $\phi(s,a)^T\theta$, so, given a state we have a set of actions to perform and for all these actions we have a parameter ϕ
The probability of an action is proportional to the exponential weight

$$\pi_\theta(s,a) = exp\{\, \phi(s,a)^T\theta \,\}$$

Since it's exponential, it's easy to obtain the score function:

$$\nabla_\theta log\, \pi_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\, \phi(s,\cdot) \,]$$

This is the difference about how much the current action a is different from the average action in the future representation.

Gaussian Policy (continuous world)

In the continuous world we need a way to emit continuous actions.
We continuously draw action from a Gaussian with these parameters:

- Mean : $\mu(s) = \phi(s)^T\theta$
- Variance : σ^2 which may be fixed or parameterized

So:

$$a \sim N(\mu(s), \sigma^2)$$

$$\nabla_\theta log\, \pi_\theta(s,a) = \frac{a-\mu(s)\,\phi(s)}{\sigma^2}$$

REINFORCE

REINFORCE is a Policy Gradient algorithm.

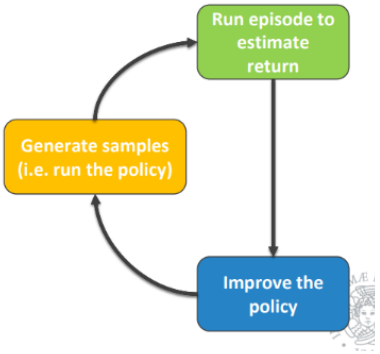
From the policy gradient theorem, we know that the policy gradient can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\, \nabla_\theta log\, \pi_\theta(s,a)\, Q^{\pi_\theta}(s,a) \,]$$

The first term $\nabla_\theta log\, \pi_\theta(s,a)$ can be obtained by a Gaussian or Softmax policy, depending on the scenario.

The second term $Q^{\pi_\theta}(s,a)$ can be estimated using a Monte-Carlo approach.

Since it's a MC strategy, we wait until the episode terminates, then, we update the V function for every state that I have encountered in the episode.



The episodes are drawn from the policy π_θ and we update the same policy (so, it's an on-policy strategy). After an initialization of the parameters θ, we update them using a stochastic gradient ascent method every time an episode terminates.

The updating of the parameters θ is performed for each item of the episode.

Note that we use $v_1 \ldots v_t$ because it's a proxy for the Q-function (because, using MC, v_t is an unbiased sample for $Q^{\pi_\theta}(s_t, a_t)$).

function REINFORCE
Initialise θ arbitrarily
for each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
for $t = 1$ to $T - 1$ **do**
$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
end for
end for
return θ
end function

The value for the gradient is calculated generating samples (yellow part) and for each sample we estimate the return (green part).

Then we use the gradient to update parameters (blu part).

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(s_t, a_t) \right) \left(\sum_{t=1}^{T} v_t^i \right)$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Policy gradient VS Max. Likelihood

Since we are parametrizing a distribution we could compare the policy gradient method with the maximum likelihood, and we can see that the only difference is the return term we have in the policy gradient.

- Policy Gradient

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta log\, \pi_\theta(s_t, a_t) \right) \sum_{t=1}^{T} v^i_t$$

- Maximum Likelihood

$$\nabla_\theta J_{ML}(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta log\, \pi_\theta(s_t, a_t) \right)$$

Assuming a continual scenario, we can see **the return term** $\sum_{t=1}^{T} v^i_t$ as a way to **module the Maximum Likelihood**, this means that we can regularize the actions in a way that gives us better results. In this way, we made "good things" more likely and "bad things" less likely, because the v function is high when we find a good path.

So, we just formalized the trial-and-error idea.

We saw a way to parametrize the policy in a on-policy strategy, the main problem is that it's inefficient due to the **high variance.**

Actor-Critic

Policy Gradient methods have high variance and we want to deal with this problem introducing bias.

The component of a RL agent that learns the Q or the V function is called **critic**.
We use a critic in order to reduce the variance.

Instead, we define as **actor** the policy learner (the component which updates the policy parameters θ).

In an actor–critic algorithm we have two kind of parameters:

1) w managed by the critic which estimate the value function (action-value Q or value-function V).
2) θ managed by the actor which update it in the direction suggested by the critic

This new kind of family of algorithms (composed by the actor and the critic), use an approximated policy gradient : Q_w (no more Q^{π_θ})

$$\nabla_\theta J(\theta) \simeq \mathbb{E}_{\pi_\theta}[\ \nabla_\theta log\ \pi_\theta(s,a)\ Q_w\ (s,a)\ ]$$

$$\Delta\theta = \alpha\, \nabla_\theta\ log\ \pi_\theta(s,a)\ Q_w(s,a)$$

The critic is just a policy evaluator, so we could use MC,TD(0) or TD(λ).

Action-Value Actor-Critic

A simple actor-critic algorithm is the so-called **action-value**.

function QAC
  Initialise s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = \mathcal{R}^a_s$; sample transition $s' \sim \mathcal{P}^a_{s,\cdot}$
    Sample action $a' \sim \pi_\theta(s', a')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w \leftarrow w + \beta \delta \phi(s, a)$
    $a \leftarrow a', s \leftarrow s'$
  end for
end function

Here we use the **linear approximation**: $Q_w(s,a) = \phi(s,a)^T w$

The critic updates w using $TD(0)$ and the actor updates θ using policy gradient.

Note: instead of REINFORCE, which works only when the episode is terminated, here we can work step-by-step so we can use TD.

Approximating the policy gradient, we introduce bias even if a biased policy gradient may not find the right solution but we could choose carefully the value function approximation, in a way to follow the exact policy gradient.
Using a baseline we can reduce the variance without adding a bias.

Reduce variance using a Baseline

So, now we want to **reduce variance** but we don't want to add a bias in the approximation of the policy gradient.
To achieve this, we can subtract a baseline function $B(s)$ to the policy gradient; this will reduce the variance without affecting the expectation value.
The choice of the baseline $B(s)$ is up to us and a good choice may be the value function itself:

$$B(s) = V^{\pi_\theta(s)}$$

So, we can rewrite the policy gradient using the **advantage function** $A^{\pi_\theta}(s,a)$ which is the Q function subtracted the V one.

Estimating the advantage function

The advantage function can significantly reduce the variance of the policy gradient, so, the critic should estimate it, by estimating both $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s,a)$ which are approximated respectively by $V_v(s)$ and $Q_w(s,a)$.
Then we can update both V_v and Q_w using TD learning.

Recall the definition of unbiased estimator:

Let S be an estimator of θ, if $\mathbb{E}[S] = \theta$ then S is an unbiased estimator.

Given the **true** value function $V^{\pi_\theta}(s)$, the TD error is:

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

Note: there is no Q here!

This TD error is an unbiased estimator for the Advantage function (non verificato: che a sua volta è un unbiased estimator della funzione Q).

So approximating the TD error we can use it to compute the policy gradient:

$$\nabla_\theta J(\theta) \simeq \mathbb{E}_{\pi_\theta}[\nabla_\theta log\, \pi_\theta(s,a)\, \delta^{\pi_\theta}]$$

The approximation of the TD error can be performed by:

$$\delta^{\pi_\theta} = r + \gamma V_v(s') - V_v(s)$$

Since in the δ^{π_θ} there is no Q, we don't need two sets of parameters, we just need the parameters v for the V function.

Natural Policy Gradient

The classical vanilla gradient ascent approach has a big problem it can follow any ascent direction, and so a policy could be updated (updating its parameters θ) in several ways, depending on the ascent direction.

Some of these directions can significantly change the probabilities contained in a policy, e.g., if we have a policy with two actions: the first is 80% probable and the second 20% probable, we do not want the situation to reverse after one step of the gradient ascent (i.e., the second action to become more probable than the first).

For this reason we can regularize the ascending direction, while we are optimizing $J(\theta)$.
We can do this, using the $KL(\pi_{old} || \pi_{new})$ as parametrization term, so we will have:

$$J(\theta) - \epsilon KL(\pi_{old} || \pi_{new})$$

A good ascent direction can significantly improve the speed convergence.

Trust Region Policy Optimization

In practice people use **Trust Region Policy Optimization** that generalizes the natural policy gradient. It uses second-order informations.
This strategy bound the difference between the old policy and the new policy (the KL divergence between them) with a threshold value. In this way we limit steps too long.

$$KL(\pi_{old} || \pi_{new}) \leq \delta$$

A variant of the Trust Region is the **Proximal Policy Optimization** which is an TRPO method but it uses only first-order informations.
Instead of KL, it uses a penalty term in the loss.

Deep Policy Networks

A **Deep Policy Network** is a deep NN used to estimate the action distribution which can be implemented by a (stochastic or deterministic) policy.

All these things take us to the A3C model that is an actor critic algorithm currently used.
It is just the classical actor critic scheme computed with neural networks and it uses TD Learning.

Model-based RL

In model planning (dynamic programming) we knew the model. In model free RL we have no model and we learned the value function and/or the policy from experience.
In this section we will build the model from the environment, integrating both planning and learning.

Model-Based RL – Pros and Cons

Advantages:

- Can efficiently learn model by supervised learning methods (so, very effective)
- Can reason about model uncertainty

Disadvantages

- First learn a model, then construct a value function (so, this strategy of first creating the "world" and then calculating the value function could introduce some errors).

Definition of model in Model-Based RL

A model M_η is a representation of an MDP $\langle S, A, P, R \rangle$ parametrized by η.
In a Model-Based RL we **assume** that **the state space S and the action space A are known.**
So, the model needs only to approximate the state transition $P_\eta \simeq P$ and the rewards $R_\eta \simeq R$.
Hence our model can be simplified as $M_\eta = \langle P_\eta, R_\eta \rangle$

Model Learning

We have an experience : $\{S_1, A_1, R_2, \dots S_T\}$ and we want estimate the model, in particular we have to learn how we can get a next state s'(approximating the state transition) and how much we can obtain as reward r (approximating the reward distribution).

Both of these tasks are computed given the actual state and the actual action.

The task of predicting the reward is a **regression** problem while the task of approximating the next state is a **density estimation problem**.

So, we need to choose a loss function (e.g. the KL divergence) and find the parameter η that minimizes that empirical loss.

Model Learning in a discrete world (table lookup)

If everything is discrete, we can perform model learning with a table lookup : given a certain state and a certain action, the table returns the probability of being in a specific next state.

We can create it by storing the maximum likelihood of the frequencies (the times I have been in a specific state with an action, it is just a frequency counting), for each pair state-action.
This frequencies are, so, counted and stored in a function $N(s,a)$.

So, the approximated value for P_η and R_η could be:

$$P_\eta = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t = s, A_t = a; S_{t+1} = s')$$

$$R_\eta = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t = s, A_t = a) R_t$$

Note: we are constructing the table lookup, used to approximate the model, from experience.

Once we obtained our model $M_\eta = \langle P_\eta, R_\eta \rangle$, since S and A are know we obtain the full MDP and we can use any planning algorithm as value/policy iteration (e.g. Bellman).

In reality we don't prefer Bellman because it requires expectation so I need to consider all the possible states and actions and this costs a lot of time. For this reason we introduce Sample-Based planning.

Sample-based planning

The **Sample-Based Planning** is simple but powerful approach to planning and it uses the **model only to generate samples**, we sample experience from the model, in particular it obtain S_{t+1} sampling from the approximated state-transition distribution P_η and the reward R_{t+1} sampling from the approximated reward distribution R_η .

This method doesn't converge but it works well because it's an incremental method: we build the model starting from scratch and each time we refine some details of it (it will become better and better through time).

Planning with an Inaccurate Model

Suppose that the model we found is not perfect: $\langle P_\eta, R_\eta \rangle \neq \langle P, R \rangle$.

When the model is inaccurate, planning process will compute a suboptimal policy.
We can solve it in two ways:

1) Using a model-free RL
2) Reason explicitly about model uncertainty

Dyna : Integrating Learning and Planning

We consider two sources of experience.

- **Real** Experience — sampled from the environment (the true MDP)

$$S' \sim P^{a}_{s\,s'}$$

$$R = \mathcal{R}^{a}_{s}$$

- **Simulated** experience — sample from the model (the approximated MDP)

$$S' \sim P_{\eta}(S'|S, A)$$

$$R = \mathcal{R}_{\eta}(R|S, A)$$

Recall that in Model-Free RL we have no model and we learn the value function (and/or the policy) from real experience, while in Model-based RL (using sample-based planning) we learn the model from real experience and plan the value function (and/or the policy) from simulated experience.

We can mix these two approaches into the so-called **Dyna**.

This hybrid approach:

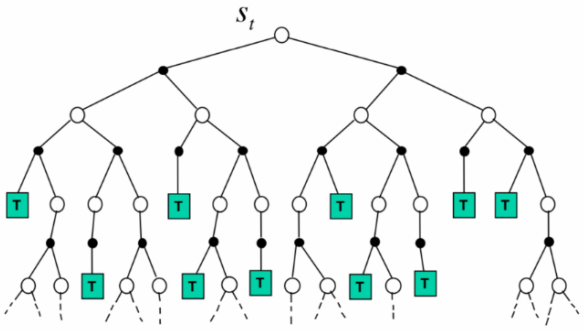
- Learn a model from real experience (as Model-based)
- Learn and plan value function (and/or policy) from both real and simulated experience (it mixes Model-free and Model-based).

Learning with simulation

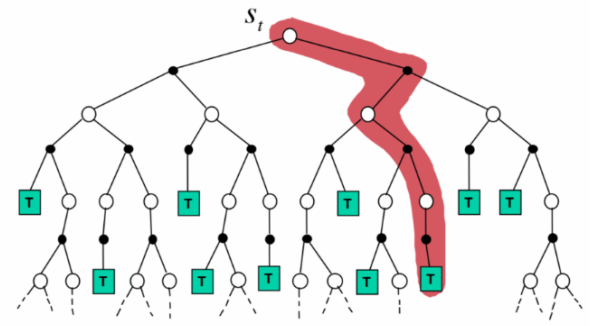
We want to efficiently find the best action , so we can introduce **trees** that help us to perform the forward search among the actions, locking ahead.

In white we have the states and in black we have the actions.

The root of this tree is the actual state S_t, this means that we can forget about the post because taking in account the past is costly. Building the tree, so, means building (on the fly) all the possible futures, but we don't like it because it's a Bellman approach.



We prefer to choose one single path and go down through it.
In practice, we use the Sample-based planning: **we simulate episodes of experience with the approximated model and then we apply model-free methods to the simulated episodes.**



If we use Monte-Carlo control on simulated episodes we talk about **Monte-Carlo search**, instead if we use SARSA on simulated episodes we talk about **TD search**.

Monte-Carlo Search

Simple Monte-Carlo Search

A simple method of using MC is using a model and a simulation policy π.
Starting from a current and real state s_t, for each possible action, we **simulate** K different episodes. Then, we select just one single action evaluating all of them by the mean return G_t calculated for all the K different simulated episodes (we select the one that has the maximum mean return).

The problem is that this method is quite stupid: we only learn about the root, but we are visiting many states below!

Monte-Carlo Tree Search

In MCTS, we have no a simulation policy π (we have just the model) and we learn and evaluate the policy "at run-time" using as much as possible generated data.
Indeed, given the model we simulate K episodes from the current and real state, as before, but then we build a tree with states and actions visited.
Then, we evaluate state by the mean return of all the episodes from starting s, a.

After the evaluation of the states, we improve the tree policy acting ϵ-greedly.
This method converges on the optimal search tree.

This method have a lot of advantages:

- Computationally efficient, anytime, parallelizable
- Uses sampling to break curse of dimensionality
- Evaluates states dynamically

But since it uses a MC approach, it hereditage its main disadvantage: the high variance.

TD Search

For the high-variance reason, we could try to use the **SARSA** method to sub-MDP in order to estimate the action-value function, for each step of simulation (bootstrapping).
This method reduce the variance but it increase the bias and usually, it's more efficient than MC search

Wrap up: 3 types of RL

Value-based	Policy-based	Model-based
○ Learn the state or state-action value	○ Learn the stochastic policy function that maps state to action	○ Learn the model of the world, then plan using the model
○ Act by choosing best action in state	○ Act by sampling policy	○ Update model often
○ Exploration is a necessary add-on	○ Exploration is baked in (trial-and-error)	○ Re-plan often