Intelligent Systems for Pattern Recognition Notes

Andrea Lepori

February 2025

1 Introduction to the ISPR course

Provide you with more advanced knowledge about ML models, more recent development of the field, including Deep Learning, Generative Models and probabilistic approaches and other paradigms. Cast whatever we introduce as a pattern recognition problem, very specific set of problems that entails mainly images, videos, signals in general, including time series. Touching slightly Reinforcement Learning. Focus on challenging and complex data. Machine Vision: problems that have to deal with how do I interpret visual information and structured data: complex/relational type of data which is compound, made of multiples pieces that are brought together by relationships. The simplest case is a time series: different measurements in time, not a single measurement, is multiple, compound information and there is a relationship between them, because there is an order, one measurement comes before the other, before the other, before the other... Time series is a simple case but as complex as you wish, until graphs and hyper-graphs. Expected outcome of the course: gain knowledge of ML models that are more than classical Kernel-base methods, SVMs, shallow NNs you might have seen so far. Focus on understanding the underlying theory, models and paradigms in this world change everyday, it does not make any sense to fall in love with a model, but understand the idea/theory/paradigm behind and you'll understand all the thousands of models that will come in the future, otherwise your knowledge will be outdated in a year. The course is organized in 5 slots:

- Introduction to Pattern Recognition: classical historical approach of what is a pattern recognition problem;
- Probabilistic (Generative) Models;
- Deep Learning;
- Generative Deep Learning;
- Advanced models and applications.

Incremental approach: from old school pattern recognition to state-of-the-art deep learning.

Introduction to Patter Recognition(PR): old school stuff, basic approaches to how to deal with PR problem until the early 2000. First by working with time series (signal processing: how do I do pattern recognition with time series) and then image processing: how to use this old school models to classify images, find interesting patterns within images, understand what's inside an image. Completely artificial separation cause images are signals: time series are 1-dim signals and images are 2-dim signals. They're something you perceive through some sensors that has an analogical representation and you want infer things about interesting parts of this signals. Shared methodologies, that you can apply to both 1-dim and 2-dim signals and very specialized methodologies.
Probabilistic (Generative) Models

- Graphical models: formalism to represent models that use the language of probabilities to describe both the world and the learning process. Graphical model is a formalism that connects with Bayesian network, a way to represent through graphs how a complex probability distribution factorizes in simpler probability distributions, making learning and inference efficient and scalable and also interpretable by humans.
- Bayesian networks and causality: how can we represent cause-effect relationship in our model, different from representing correlation aspects in our data.
- Hidden Markov Models: interesting dynamic model that unfolds in time
- Markov Random Fields: undirected graphical model interesting using machine vision
- Bayesian learning and variational inference: techniques mathematically involved but helpful because are the basis of how you train DL models nowdays. Approximation techniques derived from statistic and physics that we use to perform approximated training of very complex model;
- Sampling: very relevant when you want to generate things, when you generate from a probability distribution you're sampling from a probability distribution, when you generate an image that does not exist by taking it out from a diffusion model you're sampling from a model.
- Boltzmann machines

Deep Learning (DL) fundamentals We'll see a certain number of models that are representative of classes of models in DL:

- Deep autoencoders;
- Convolutional architectures for images and time series;
- RNN with gates;
- Transformers and encoder-decoder architectures;

- DL toolset: dropout, bacth normalization, residual connections, attention
- Neural memories

Generative deep learning How to train a NN whose job is to approximate a distribution or to approximate a sampling process

- Variational AutoEncoders
- Generative adversarial networks (GAN)
- Normalizing flow
- Diffusion model

Advanced topics and applications

- Reservoir computing: design paradigm for dynamical system inspired NN
- Deep Learning for graphs: how can you have NN which instead of ingesting vectors/sequences/images ingest graphs and predicts on graphs.
- Reinforcement Learning

The origins of Pattern recognition Duda and Hart 1973: Machine recognition of meaningful regularites in noisy or complex data. Certain number of approaches through which PR has been pursued, ML is one of the last. PR has been attacked for several years by other means than ML and NN especially. People were looking in rule based system, signal processing derived from engineering, even logic and reasoning, so formal methods like grammars applied to PR. In the end who win the prize was Deep NNs who put the name of PR out of order. **The Viola-Jones Algorithm** For many years was considered the de-facto standard for PR problems on faces. Before the advent of DL how do I recognize the presence of faces into an image. People were looking to a certain number of images and aligned many images and decided where to put these filters, whose job was to recognize relevant feature in an image that represents a face. VJ1 is a filter that is made of a orizontal stripe of white pixels sorrounded by a stripe of black pixel. Why? Because if you scan your image, that filter will tend to respond highly when you encounter the eyes. VJ2 same but vertical, will respond to the presence of nose, same hand-designed filters exist for mouth, ...hand engineering of the features to gain a presentation of the information you were processing clean enough to put a simple predictor on the top it. Pipeline for any PR application in the past: identifying distinguishing attributes in to my data.
Feature detector $\rightarrow$ extract relevant feature automatically $\rightarrow$represent a complex image in a simpler vector$\rightarrow$ match this rep. with stored info (pattern matching).
AlexNet $\rightarrow$ Convolutional NN brought at a competion in object detection 2012. Pattern recognition closed field. Long Short Term Memory $\rightarrow$ gated RNN before the advent of transformers were the de-facto standard for language modeling,

still remain a model for time series or sequential info, because transformer are NOT model for sequences but for multisets and become model for sequences because you hammer inside of them positional encoding that tells that things are ordererd according to some complete ordering but inductive bias of that model is not for sequential data. Compositionality of neural modules works becuase they speak the same language of differentiation. Generative Adversarial networks : probably the model that created all the wave of generative deep learning, first model that could create new data that really look like real data, in particular for images. Variational Deep Learning $\rightarrow$ is a old paradigm, we won't speak about Variational Autoencoder for their ability of generating high quality images because GAN or diffusion models are superior in these, but because it is interesting for the methodologies used by other models and esemplify clearly what DL is about, is about not having super deep NN it's about representation learning, key thing about DL, learning to represent complex information into possibly semantically organized Neural representation spaces in such a way I can use those representations to simplify any kind of task I want to solve, learn to represent nicely (connected to the task you wanna solve) whatever type of data you have, no matter the complexity or the noise of the data. An this model is very interesting cause it allows you for instance to play with how you can embed additional info into you neural repr. space. You can fed your model with MNIST digit images, not get projected in a random vectorial space in which 0 are separated from 1, but in which images of 0 get project nearby, and the next nearest thing is a 1, and the next nearest thing is 2, and then 3,... so ther's an ordering so that you can use operations in the latent space that makes sense algebraically, take two vector that represent a 0 and a 3, sum the 2 vectors and then I decode i get an image of a 3, cause $0 + 3 = 3$, there is a semantic organization underlying the neural representation space, enviewing the neural representation space with relational info. When we talk about generating data, nowdays, we have Diffusion models or whatever language model driven approach used to generate image. Diffusion process is essentially the idea to train a model to transform an image into pure noise and then to revert this process from pure noise to an image. An then I can sample pure noise and decode it to an actual image, and that's how i generate stuff. Graph Neural Network : NN whose job is to ingest a graph and make prediction on the top of it, a molecule is a graph for instance. Process or generate graph, e.g. generate molecules. An interesting thing of Graph Neural Networks is that is an interesting way to encode into a neural representation any structured knowledge you may have. Throughout the history of AI we've been chasing this dream of creating a knowledge graph that represents everything (concepts) a human knows, including the relationships between those concepts. With Language model we kind of bypass the thing, knowledge graphs represents consolidated knowledge we want to align the Language Model with the knowledge in the knowledge graph, so that the LM does not generate things that are false. The ability to process a knowledge graph (which is a discrete combinatorial structure) and transform it in a dense representation (vector) you can put in a NN (analogical machine) $\rightarrow$ graph neural network. Neural algorithmic reasoning $\rightarrow$ teach a NN to solve classical program-

ming exercises on graphs : shortest path, minimum cut, traveling salesman,... if you train a NN to solve step by step these algorithms the NN gets better to solve other problems on graphs. Understand how inferencial statistics could be used as a procedure to learning.

2 Introduction to Signal Processing

1-dim signal → time series. An image is a signal rather than to be defined over time is defined over space. time series → sequence of measurements in time, multiple info which are not independent one another, relaxing i.i.d assumption. Within a single sample itself it is made of observations which are not i.i.d. between different time series they could be i.i.d. In order to evaluate a specific point in time, I can't consider only the present but i Need to consider also the past otherwise I'm not accurate enough in predicting. X = x_0 x_1 ...x_N where each element x(t) at time t could be a vector → monodimensional VS multivariate time series. Observations may not be equally spaced in time. Irregular time series is what you typically do not want in a RNN, the RNN makes the assumption that data are equispaced, otherwise you have to inform the NN that there's a varying time step. Assumptions: weak stationarity is the and of 2 conditions: the expectation/average of the time series does not change with time (exclude time series that are monotonically increasing/decreasing along time). For time series that does not respect this condition: detrending: as you normalize dataset, or rescaling into an interval (0-1 or -1/1) normalization, or z-score normalization (0 mean). Same here, compute the running average of my observations in a period and subtract by a point in that period the local average (detrending), for the second period we will have a different average and we subtract it to all the points in that period. We remove the slope, the fact that the average value of my observations is varying in time. 2° condition: the covariance between 2 measurements in time separated by a time interval tau is a certain value that depends only on tau, not depends on time t. It depends on how far they're, not from the the specific time i compute the covariance. Goals : - descriptive statistics of the time series, to summarize in a single value some interesting trend of my data. - analyzing time series : finding any dependencies, recurrent pattern...e.g. time series of traffic any recurrent trend?? - prediction - control : classical robotic problem: adjust some parameters in such a way that I can pursue in time a specific trajectory. see from 2 diff perspectives: time domain - looking how a signal changes in time: correlation (covariance) and autoregressive models. spectral domain - analyzing a time series in the domain of the frequencies, look at what are the characterizing frequencies in a specific signal: fourier analysis and Wavelets. Classical descriptive statistics: time series average (and variance) : the average of all the observations. Two time series may look different but if I compute the respective average and i subtract it, they're similar. Preprocessing to consider if there is some effect that should be canceled out. Understand when should be canceled and when should not! E.g. time series of ECG from different subjects if I take the average overall my

dataset, the mean of all the time series put together i mix people with different basal frequencies. Autocovariance - tipycal covariance is between two vectors of two different signals, in autocovariance is between two differents measurements in time of the same signal. I compute the covariance at a specific time gap tau, I take my measurement at time t and subtract the mean, i take my measurement at time t + tau and subtract the sample mean and multiply the two. If you normalize the autocovariance at time gap tau with the autovocariance at 0 distance you get autocorrelation, of the signal with itself at a specific time distance tau. The term at the denominator gets away the magnitude of the signal (sum of the squares), it's a measure that allows you to compare correlations across different time series/signals. We can use it to study recurrent patterns in a time series, because I' m correlating a time series with itself at different taus, i see if the signal kind of match a shifted (by tau) version of itself, the autocorrelation will spike when I find a tau after what the time series repeats itself. Autocorrelogram : plot of the rho for different taus, on the x axis you have the tau, it's simmetric, there is a pick of correlation in 0 (trivial, correlating the signal with itself), near the origin dumb correlation, identify picks in correlation, looks if it follows a sinusoidal behavior (identifify the period tau in which the signal repeats). cross correlation $\rightarrow$ correlation between 2 different time series. there is a displacement/shift, look if two time series is alignibile by shifting at a certain time shift tau. tau where the correlation is maxima is the displcaement between the two time series, if you cancel that gap (pull one of the time series back) you have the highest chance of alignement. Normalized cross correlation - correlation normalized by the magnitude of the signal. value between [-1,1]. if this value is 0 there is no linear correlation but that could be that exists a non linear correlation. Cross-correlation remembers convolution. Convolution is a smoothing operator, weight the measurements of a signal by another signal, and it's symmetrical, the convolution between f and g is the same as the convolution between g and f, this not holds for correlation. Convolution is friendly to differentiation:

$$\frac{d}{dt}(f * g) = \frac{df}{dt} * g = f * \frac{dg}{dt}$$

the derivative of the convolution is equal to taking the derivative of f and then convolving, or taking the derivative of g and then convolving. Helpful when f is an image and g is a simple filter, taking the derivative of an image can be costly and approximated, g filter can be a simple continuos function with an analytical derivative, we take the derivative, we convolve with the image, we automatically have a smoothed approximation of the derivative of the image. Convolution: you keep one signal fixed f, and let another signal g sligth thorugh time by changing tau. When you start crossign path with the other signal you are computing the integral of f multiplied by the values of g, you're taking the integral w.r.t. to time weighted by the values of the g function. Principle used when we filter images, we take an image and a simple filter, we sort of rewrite the pixel values of the images with a smoothed version of themselves, depending by the filter that i choose i can make the differences between nearby pixels

stronger to highligth the details or I can make the differences in luminance between pixels lower reducing the resolution.
Equivalent of linear regression for time series → autoregressive process: linear model. I regress a signal on itself. the observation at time t can be obtained by a linear combination of the past k observation, the coefficients of the combination alpha is going to be learned. While doing so i commit an error epsilon_t, instantaneous error at time_t. k is the order of the autoregressive model = number of degrees of freedom of the model/num. of params. epsilon_t is a sequence of errors i.i.d.... but error compounds... is a too strong assumption. ARMA : autoregressive model with moving average, there is still a part of the error that I cannot predict based on the history of my errors, but I add a moving average of the errors in the past. two sets of param: coefficient of autoregression alpha , beta coefficients of the moving avarage of the errors, two order to decide k and q. now epsilon_t accounts for the amount of the errors of the past that are not linear. let's assume that with the term of the error I'm removing all the dependencies from the past errors, than epsilon_t become more truly i.i.d. to estimate K and Q use model selection by putting some penalty term on model complexity to penalize model with too many parameters !!! compare time series even of different length by comparing the alpha coefficients (a compressed representations), compare two time series w.r.t. to the autoregressive coefficients under the assumption that this coefficients compress the time variant info that allows me to predict on the time series. Represent time series in a fixed dim vector. clustering → run kmeans on the alpha embedding

spectral method → decompose a signal in a set of frequecies (spectral domain). We use function good to represent diff. freqs. → sinusoids, I just change the freq. param of a sinusoids, I decompose a function that varies in time as a weighted combination of sinusoids oscillating at diff. frequences.

DFT → tool to compute the coefficients of the weighted combination of sinusoids. One transform that goes from time to frequency and exists the inverse.

$\langle f, e_k \rangle$ dot product between the two functions or the integral between f and e w.r.t to its domain. orthonormal basis for us is sin and cos at different frequencies. the summation in digital world does not go to infinite but I cannot have in my time series an higher number of frequencies than the number of observations in my signal.
As many vectors in my orthonormal basis as the number of elem in time series each e_k has N element, same length of time series to compute the scalar product between e_k and x to obtain X_k. X_k how much of the vector e_k I'm using to approximate my function.

$$X_k = \langle f, e^{-ikn} \rangle$$

Result of the DFT is a vector of coefficient of N elems with the importance of each frequence in the signal. coefficient in the spectral / frequencies domain.
Formally, given the orthonormal system:

$$\left\{ \frac{1}{\sqrt{2\pi}}, \frac{1}{\sqrt{\pi}} \sin x, \frac{1}{\sqrt{\pi}} \cos x, \frac{1}{\sqrt{\pi}} \sin 2x, \frac{1}{\sqrt{\pi}} \cos 2x, \dots \right\},$$

the Fourier series of a function $f(x)$ is given by:

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

where

$$a_k = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(kx)\, dx, \quad b_k = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(kx)\, dx,$$

and

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)\, dx.$$

Representing Functions in Complex Space:

Using Euler's formula:

$$\cos(kx) - i \sin(kx) = e^{-ikx}$$

we can express a periodic function $f(x)$ defined on the interval $[-\pi, \pi]$ as a sum over complex exponentials:

$$f(x) = \sum_{k=-\infty}^{\infty} X_k e^{-ikx}$$

Here:

- $\{e^{-ikx}\}_{k \in \mathbb{Z}}$ forms an orthonormal basis for complex-valued functions on $[-\pi, \pi]$.
- The Fourier coefficients X_k are computed by:

$$X_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-ikx}\, dx$$

This representation is known as the **complex form of the Fourier series**. I use those X_k as informative points, i compute the amplitude (modulo) a number that tells me how strong it's a specific frequencies k in my signal.
power $\rightarrow$ the modulo squared of my fourier coefficients divided by the num of time points sort of approximation of the density: something that resemble the probability of the k frequency. plot the power spectrum of the time series: on the x axis is k(Hz), on the y is the power of that frequency. power spectrum tell me the distribution of the frequencies of my signal. in the picture Bunch of low frequency components that are very relevant and some high frequencies component that are not very relevant, go back in the time domain but does not use all the frequencies use only the relevant one. You've a smoothed fitered version of the original signal, you've removed the little important high frequencies component that make your signal noise. In the freq. domain you can pick more some frequencies, zero some, dump some, decide which to keep, and then reconstruct in time domain. Peaks in power spectrum can reveal some recurring

(repeating) patterns and so their period.
Spectral centroid compute the weighted average frequency, weigth each frequency with the fourier coefficient. used to discriminate between sounds, different instruments have different characterizing frequencies → take the average centroid.
Kurtosis measures flatness or non-Gaussianity of the spectrum around the centroid mu, Kurtosis is a statistical measure that describes the "tailedness" or how heavy the tails of a distribution are compared to a normal (Gaussian) distribution. Monitor the kurtosis across frequencies and in general if there is not something strange the kurtosis will be in a thin little band, when something non gaussian happen there is a peak in kurtosis, you know that something interesting happens, unexpected. kurtosis gives hint on where information is, sudden changes. spectral entropy measure of disorder compute on power spectrum sort of approximation of a density/distribution you compute its entropy. Different sounds have different distribution of entropy, can be used to differentiate between sounds of different natures.

3 Image Processing I- Descriptors

I do you represent visual content in an effective way. Images → matrix of pixel intensity (graylevel), color images multi dimensional matrix (RGB), alpha level transparency level. CIE more smooth, sligth changes in the representation correspond to slight changes in human perception. object identification/recognition : identify all the objects in an image where they're located and their label. Bounding box of object. pixel level tasks → Image Segmentation: unsupervised , group pixels in perceptually consistent groups/segments. semantic segmentation → supervised, this pixel is part of sky , this is part of a car , this part of a tree.... How do we represent visual information?? How do we extract part of the image that are informative ? Today answer is NN, but let's see old school PR answer. Representation must be robust to transformation/variations in the content and representation of an image. Invariant to transformation that are common when you take a picture, luminance and geometric transformation of the visual content, ideally your representation should stay the same with light on/off or if i shoot the object from 1/10 meters, the objects will occupy a bigger portion of the image but it's the same object the rep must be the same. I move point of view (aside/above), not affine transformation that change substantially object shape in the image but the object is still that one and the change in the representation should be smooth not disruptive in order to setup on the top of that representation a learning process. How to identify informative parts? I need to separate object from the background for object recognition task.
image hitograms → counts of relevant info in the image e.g. colors. grayscale 0 ... 255 bins how many pixel has this coulor? or how many edges oriented in 30°, 45°, 15°... with histrogram you destroy all the info about geometry. correlogram to be a little more position preserving, binning both color info and position info, how many pixels of that color in a specific position/area of the

image. You can take an image, scrumble completely the pixel ordering, it will have the same representation as the original image but certainly not the same semantic content.
First approaches: similarity matching between histrograms. Looking at an image at a global level, with this distributional repr. does not work→ local descriptor : rather than represent in a single piece info about all image represent info on portions of image. Discrete based → describe distribution of pixels into a sub image. Dense based → you fit a gaussian on a portion of image, not fill an histogram but rather find the params of a gaussian that spans an image patch. To describe a full image many local descriptors in different locations but not only : these descriptor must be robust considering also different scale. Convolution of an image with some form of filter. Simplest form of localized descriptor → the region/piece itself. I linearize the image into a vector and this is my representation, I can normalize by subtracting the mean luminance and make it illumination invariant. But if I translate the squirrel? If I rotate by 90° degrees? the descriptor is completely different but it's the same object! We need something less strictly anchored to the position in a specific pixel sub-window. Distribution-based descriptors to little portion of the image, the disruption of local geometry is now an advantage, robustness to transformation, local patches are represented by histogram of features, histrogram of pixel intensities of the subwindow, but it's not invariant enough! The descriptor must be invariant to scale, if I change the distance w.r.t. to which I shoot my picture the descriptor should be resistent and at least I need rotation (affine transformation) invariance, not panning (if you change the point of view w.r.t. you look to the object - pan change completely the geometry of the object (nested transformation difficult to be invariant to).
SIFT (Scale Invariant Feature Transform): reference model to represent visual content into an image, essential building block for image retrieval search engines, you represent your image with a collection of SIFT descriptors, feeding them to a learning model trained to connect sift descriptor with classes. SIFT are localized descriptor, centered into a center pixel x,y (key point) with a certain fixed dim neighborhood surrounding the key point. Since this descriptor should be resistent to different scale I compute it at different scales, not a single photo of an object but the same photo taken at different distances. We emulate this. sigma to change how close we look to an image, how we convolve the image with a gaussian filter with std (standard deviation) sigma. Convolution is a smoothing operator, weight a function with another function, my leading function is the image, the filter is a gaussian filter, we take the intensity of the pixels within the neighborhood, summing them weighted by the response of a gaussian filter and rewrite the center position of the convolution, the intensity value we have in the key point with the result of this convolution. Rewrite the intensity value of the key point with the weighed average of all the intensity values inside the area spanned by the gaussian filter weighed by the response of the filter. By varying the size of sigma, I'm varying the intensity of the weighing by the neighbors, if sigma small, pixel is gonna be influenced by a small number of neighbors → sharpening effect, large sigma → influenced by broader neighborhood → pixels

in same neighborhood will have same intensity level (blurring effect). You emulate the blurring effect of the distance.
Convolution of a gaussian filter with an image, gaussian filter is a gaussian without the scale component as you do not need in this contest. By varying sigma I vary the amount of smoothing. Convolution slides through all the pixels, I obtain a new image, the intensity pixels of my image will be rewritten in a smoothed way (smooth according to sigma). The closer to the center point the more you contribute to the weighthed average. Moving sigma you enlarge/make thin the gaussian. sigma = 0.05 image is sharpened. sigma = 5 image is blurred.

$$L_\sigma(x, y) = G(x, y, \sigma) * I(x, y)$$

$$G(x, y, \sigma) = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

Now i take an image and I can represent that image at different scale sigma. For each image I can represent in 24 scales and compute descriptors on 24 scales $\rightarrow$ robust to distance changes.
Now we have a way to take an image and find a multiscale/multiresolution representation of it by convolving it with gaussian filters with different sigmas.
Problem of representing the information? Instead of representing the distribution of pixel intensities I'm gonna be representing the distribution of pixels gradient. How to take the gradient (as in analysis/calculus) of an image? An image is a function that assign luminance to position, the intensity associated to a specific x and y, function of pixels coordinates, to find the gradient i differentiate with respect to the coordinate. I do not have the analytical expression of the image of course, but there is your friend numerical approximation of the gradient, CENTRAL DIFFERENCES that are very easily applied into the setting of an image: I can compute the gradient in a specific pixel by taking the intensity in the nearby pixels and subtracting them. Compute magnitude and orientation of the gradient, the gradient is a vector with direction and a length. The higher is the variation in a direction the longer is the vector/the higher the magnitude. Compute the gradient of intensity of all the pixels in my patch, for each pixel I will have a gradient vector, one magnitude and one direction.
The magnitude of the gradient of the intensity of an image at point x,y is the square root of the sum of square of x and y gradient. One step on the left and one step to the right w.r.t. the center point to compute the gradient along x direction, one step at the bottom and one step on the top to compute the vertical gradient. ARCTAN to get the orientation.

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

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

More scalable to apply to all pixels of an image using convolution, to obtain the x direction of the gradient I just convolve my image with a filter that gives

the horizontal central difference. Same for the y component by convolving my image with a filter that gives vertical central difference. I compute an histogram of gradient in a single patch. In a single patch I've multiple pixels and I'll count the orientation of those pixels. 4x4 gradient window for the sake of computing the histogram, centered on a key point, 4x4 set of pixels surrounding and i compute the gradient histogram counting the orientation in those pixels, I'm gonna be binning those samples in 8 orientation bins. Scaling the counts based on the magnitude of the gradients. Pixels will contribute to counts in a bin based on their magnitude. 1 counts as 0.1 if its magnitude is 0.1. Another scaling dependent on a gaussian with the size of the scale to which the image has been rescaled. I'll get a localized descriptor 4x4x8 (4x4 neighborhood) and for each neighborhood I'll have an 8 dim histogram = 128 size. I have for the time being a gradient computed only on that pixel, very noise, I want a more robust gradient, distributed computation of the gradient, the final gradient of a center point depends on its own gradient and on the gradient of the neighbors. This is the part of orientation assignement. Collect all the gradients of my sorrounding into an histogram with 36 positions (every position indexes 10°) weighing by their magnitude and by gaussian weigthing (1 * m_i * G_i) where m_i is the magnitude of the gradient and G_i the response of a gaussian filter for that pixel, pixels very far away from me will contribute little. Then pick up the leading direction. If any other orientation falls in the 80% of the top orientation pick also them. I create a copy of myself with that one, doubling the number of key points. A key point has a denoised version of a gradient. 4 x 4 neighborhood of a key point, point (1,1), is made of 4x4 pixels, neighborhood is hence 16 x 16.
For each of the pixel in there I'll have an orientation, this will correspond to an histogram in which I count how many of the 16 pixels in region 1,1 are in each of the eight leading orientation. 128 size representation by having one histogram for each of the 16 subregions.
The **Scale-Invariant Feature Transform (SIFT)** is a technique to detect and describe local features in images. It provides descriptors that are robust to scale, rotation, and small distortions.

Steps to Build the SIFT Descriptor:

Step 1. Keypoint Detection: Identify interesting points (keypoints) in the image, such as corners or blobs. These are detected at multiple scales using the Difference of Gaussian (DoG).

Step 2. Gradient Computation: For each pixel in the image, compute the gradient magnitude and orientation:

$$G_x = I(x+1, y) - I(x-1, y), \quad G_y = I(x, y+1) - I(x, y-1)$$

$$\text{Magnitude: } M(x,y) = \sqrt{G_x^2 + G_y^2}, \quad \text{Orientation: } \theta(x,y) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Step 3. Orientation Assignment:

- Extract a 16×16 region around the keypoint.
- Compute a histogram of gradient orientations (36 bins, 10° each), weighted by gradient magnitude and a Gaussian window centered on the keypoint.
- The dominant orientation becomes the keypoint's orientation.
- If another orientation is within 80% of the maximum, a new keypoint is created at the same location but with that orientation.

Step 4. Descriptor Construction:

- Rotate the 16×16 region to align with the keypoint's orientation.
- Divide it into 4×4 blocks of 4×4 pixels (16 blocks total).
- For each block, compute an 8-bin orientation histogram.
- Each pixel contributes to the histogram based on:
  - Its gradient magnitude m_i.
  - A Gaussian weighting G_i.
  - Total weight per pixel: $w_i = m_i \cdot G_i$.
- Total descriptor size: $16 \times 8 = 128$ values.

To be rotationally invariant instead of representing absolute angles represent relative angles, compute the leading direction of the gradients among all the pixels, and instead of representing the absolute direction of the gradient represent the local difference between the local gradient and the leading direction of the gradient. If the same patches is rotated also the leading direction rotates but the relative position w.r.t. the leading direction do not change.
Image are 2-dim signal, i can do Fourier analysis on images, in seq. we had time and spectral dimension, with images we have space and spectral dimension. There exist basis functions through which you can approximate every function by appropriately linearly combining them, where the params of the linear combinations, the coefficients are the results of a vector to vector dot product between the base function and the original function. Images as a combination of wave like sinusoidal and cosinusoidal images. X(kx,ky) on time series a single parameter for the k-th frequence, here 2 param for the k-th freq. one of them for the freq. over x and freq. over y axis. Fourier density plot will be bidimensional.
The Convolution Theorem: to obtain the Fourier transform of the convolution of 2 signals you can fourier transform the first signal, fourier transform the second signal and then multiply (element-wise) the 2 representations that you obtain. Convolution in the spatial domain become multiplication in the spectral domain. When you want to compute the convolution of an image with a filter, you get your filter in the fourier domain, you get you image in the fourier domain, you multiply them and you use the inverse fourier transform to go back to the spatial domain. Template matching: Filter to detect the presence of letter 'a' in an image, I take my image, I fourier transform it, same for the image of the

letter, they're gonna be projected in the same space of the fourier frequencies, than multiply them, that will be equivalent to convolution, then invert and generate the response image. Pixel intensity depends on the matching between the template of the letter 'a' and what's in each area of the image, then i threshold the image and obtain areas where 'a' letters are located.
Use convolutional theorem to implement convolution in CNN as multiplication in Fourier Domain rather than actual convolution, or implements your CNN directly in fourier domains.

The Discrete Fourier Transform (DFT) is symmetric in both spatial directions. To better visualize its power spectrum, it is common to shift the zero-frequency component $(0, 0)$, also known as the DC component, to the center of the plot. This component typically has a much higher magnitude compared to other frequencies, which can dominate the spectrum visualization.

The DC component is calculated as:

$$X(0,0) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} I(x,y)e^0$$

To make the spectrum more interpretable and suppress the dominance of high-magnitude values (especially the DC component), we usually apply a logarithmic transformation:

$$\log(\text{abs}(H))$$

where H is the Fourier-transformed image. This transformation enhances the visibility of smaller frequency components in the spectrum.

4 Image Processing 2: Detectors

What we required for visual descriptors. Repeatability : the same interesting point can be found despite some form of perturbations. Edge detectors: edges define borders of objects, good area of the image on which to focus. Edge detector→ thresholds the image to decide where is more likely to be an edge. An edge is characterized by sudden change in luminance (gray level). Recognize sharpe changes in luminance, the luminance/greylevel is a function of the pixel→ use the gradient to identify in which region of the image there is an edge. Gradient along x and along y, an approximated method to compute the gradient of an image is the finite differences method. I center myself on a pixel, one step on the left and take the lumincance on the pixel on the left, same for the pixel on the rigth, subtract the two to take the difference. This is very rough, you want something smoother. Use Prewitt operators, specialized form of a finite difference methods for the gradient, the gradient over the x is no longer computed with a vector but is computed with a matrix. That matrix is convolved with an image. element by element multiplication and you sum to obtain the gradient of the central point. By the application of the two filters you get two images, the image filtered through the x-gradient and the image filtered through the y-gradient. You can compute the magnitude of the gradient vector

(square root of the sum of the square of the 2 components), take the intensity value of the gradient then thresholding to obtain the good candidates for the presence of an edge. Sobel operator an additional level of smoothing on a central component.
Prewitt and Sobel Operators

Edge detection in image processing can be performed using gradient-based methods. Two commonly used methods are the **Prewitt** and **Sobel** operators. Both approximate the gradient of the image intensity function.

Prewitt Operator

The Prewitt operator uses two 3x3 kernels to estimate the gradient in the horizontal (G_x) and vertical (G_y) directions:

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

Sobel Operator

The Sobel operator gives more weight to the center pixels. Its kernels are:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Gradient Magnitude

For both operators, the gradient magnitude is typically calculated as:

$$G = \sqrt{G_x^2 + G_y^2} \quad \text{or} \quad G \approx |G_x| + |G_y|$$

These operators are useful for detecting edges in images by highlighting regions with high spatial derivatives.

Edge detectors are often gradient based and let us to compute interest point, then I can compute there a SIFT descriptor. I do not compute SIFT descriptor for every nearby points, because they're gonna be very close, a lot of descriptors very redundant, space-level, scale-level suppression of non maxima. If 2 descriptors are very close in the space, I'm gonna keep only the descriptor with the highest response with the filter (highest magnitude w.r.t. the edge detector).
BLOB $\rightarrow$ luminance stays constant in a neighborhood, regular and small. You want to identify to place an interest point in the center of your circular blobs.
Laplacian of a gaussian tends to respond highly when it is centered in a blob that is proportional to its sigma value.
Laplacian is the sum of the second order partial derivative of a function, is telling when there is an abrupt change in the gradient of a function, you're summing the single component of the second partial derivative of function. Laplacian of a gaussian its known to have a maximum response when it's centered on a circle of radius sqrt(2)*sigma. Sigma is std, the bandwidth of the gaussian. If you position the Laplacian of a gaussian centered on a blob with radius similar to sqrt(2)*sigma you get the maximum response. Blob detector, take a set of laplacian of a gaussian with different sigma, convolve with the image, find the points

of maximum response, there are blobs, whose radius is close to sqrt(2)*sigma. different laplacian → non maxima suppression → take the highest response. high pass filter → only things that are high pass → clean noise.
Scale normalized response: not take the actual Laplacian i take the laplacian and multiply by sigma square. If you peak a gaussian it become taller, or viceversa more fat, it changes the amount of response that you have there, smooths your estimate and makes it comparable to compare response of gaussian with different scale, a normalization factor.
Blob Detection Blobs are regions in an image where the pixel intensity is relatively uniform or varies slowly, meaning they exhibit **low gradient variability**. These regions often correspond to areas of interest such as spots, circles, or blobs.

Laplacian of Gaussian (LoG)

The **Laplacian of Gaussian** (LoG) is a second-order derivative operator used to detect blobs in an image. It involves applying the Laplacian operator (which computes the second spatial derivative) to a Gaussian-smoothed image.

LoG Response

The LoG filter, denoted as $\nabla^2 g_\sigma(x, y)$, has a **maximum response** when centered on a circular blob of radius approximately 2σ. This is because the filter captures regions where the intensity changes in a circular pattern.

Mathematical Expression

The LoG operator is defined as:

$$\nabla^2 g_\sigma(x, y) = \frac{\partial^2 g_\sigma}{\partial x^2} + \frac{\partial^2 g_\sigma}{\partial y^2}$$

Here, $g_\sigma(x, y)$ is the 2D Gaussian function with standard deviation σ.

Scale-Normalized LoG

To detect blobs at multiple scales, a **scale-normalized Laplacian** is used, which compensates for the fact that larger structures require stronger filter responses. The scale-normalized Laplacian is:

$$\nabla^2_{\text{norm}} g_\sigma(x, y) = \sigma^2 \left(\frac{\partial^2 g_\sigma}{\partial x^2} + \frac{\partial^2 g_\sigma}{\partial y^2} \right)$$

This normalization ensures that the filter responds equally to the same structure, regardless of its size. The detector used by SIFT descriptor → LoG Blob detector.

$$\sigma = 2^k \cdot \sigma_0$$

image → 1 response image for each laplacian of a gaussian. Threshold image , put zeros when the response is very low. find maxima on a space scale, I take one of the slice and i focus on maxima, if 2 pixels are nearby and highly active/very high in response I keep only one. Do for all the portions of the image. Keep only the higher of its neighbor, this for every of the images (feature map), then look across the scale direction, two interest points nearby but on 2 different scales, keep only the maxima. Then obtain only a filtered image with only those pixels that are maxima in their local neighborhood (space level) and on a scale level. At the end i obtain circles centered on interest points

with a radius sigma. Detecting interest point at different resolution/scale. For each of the detected interest point compute a SIFT. The LoG can be efficiently computed/approximated by difference of Gaussians. The LoG at a certain scale (k-1)sigma_0 can be approximated by the difference between two gaussians the gaussian ksigma_0 - the gaussian sigma_0. Such detectors are not invariant to affine transormations. Gaussian is circular blob, scale transform ok, just a different LoG responding with an higher/lower sigma. But affine transformation stretches stuff, you can get ellypses LoG does not work.
MSER → Maximally Stable Extremal Regions (MSER) de-facto standard for detection. I look at region that are stable in an image, region of connected pixels, (convex hull that encloses your set of pixels), this pixels connected regions does not have to change much if I alter more or less slightly the condition of my image. Connected components that if the images is altered slightly the connected region does not change too much, but i only shoot 1 image I can't ask people to shoot hundreds of images and see if the condition of the image changes, the connected regions stay the same. I can be the one altering the image. I can take the original greylevel image and start thresholding with diff threshold. Below the threshold white 0, above 1 black and use different thresholds, series of images of black and white pixels and look to areas that remain constant under different thresholding. No circular assumption, robust to affine transformation. What if I change substantially the distribution of color light, some shadows may appear in a certain region, changing the shape of the connected components, robust to affine transformations but not to strong changes in luminance that project shadows.
Image segmentation: take an image and split in regions of homogeneous intensity of color, I don't look on tiny little blobs, that describe local content of an image, I'm rather looking to the largest possible homogeneous segment in an image which possibles describe parts of an object. Having 3 segment one for the background, one for the squirrel, one for the bottle. Totally unsupervised way. Without knowing anything about the semantic of the visual content. We only base on color/intensity information. Naive approach: image collections on pixels, every pixel is 3-dim vector RGB, run K-means. Will assign pixel to cluster according to similarity in color. Problems? set k, very disconnected segments, segment a tiger or a zibra :)
You have to add information about local consistency. Add 2 dim to the vector with x,y now 5-dim vector RGBXY color info and if the are nearby in the space. To be more robust run clustering in a more perceptually consistent space like L*a*b (another color space).
De-facto standard for image segmentation: Normalized Cuts (Ncut), a clustering algorithm, like k-means, only smarter cause it segments by looking at pixels as if they were a graph made of nodes, each node is a pixel, each pixel has associated info about the color and an edge between two pixels, how to create an edge? they're neighbors or if they are affine, they have the same visual appearance i.e. similar color. So I create a graph that reflects the color of the pixels and an edge between two pixels if they are sort of neighbors and their color is similar. Gonna be measuring the similarity with a weight a_{ij}, the strength be-

tween pixel i and j. Weighted graph. Try to find a minimum cut in this graph, i.e: a set of edges with their weight/cost which i can cut, a minimum cut that disconnects a portion of the graph w.r.t. the rest of the graph. I disconnect a strongly connected component of pixels, they are similar, a segment of the image, those pixels are part of the same segment which is radically different from the rest of the image/graph. Minimum cut: is the set of edges that you can cut that cost less you to cut/to disconnect. Pay less price possible: cut fewer amount of edges with smallest affinity values. The cost of the cut is the sum of the affinity values of the edges that you cut. Problem? you will cut one node at time (dumb), normalized cut, normalize the cost of the cut w.r.t. the sort of overall distribution of the cost in a graph, in such a way you don't make the dumb cut of one node, you look for cutting a good connected component in which the relative cost of the edges you are cutting is not that high. Nomalized cut is NP-hard. Solve a simplified problem. Spectral dimension. The normalize cut solution is linked to the second smallest eigenvector of the affinity matrix. The second smallest eigenvalue is an approximated solution to the normalized cut problem, you disconnect that component, than you iterate on the rest of the graph until nothing more to cut.
Instead of doing for every pixels, do it on set of consistent pixels, obtained with k means. Partition an image e.g. in thousands of sub-images (superpixels). Each of of those regions become a node in my graph, i compute the affinity between those regions and i run N-cut on these. Or smaller k, bigger regions. Group pixels in group using k-means. Superpixels = cluster of pixels. I can compute N-cut at different level of granularity, with superpixel at different level of granularity. Multi-resolution segmentation of an image. You can have detectors that respond to corner rather than edges→ harris corner detector.
Image processing is very much in convolution, convolving an image with a filter, and use the result filtered image either to compute distributions of gradients or representing visual content in visual descriptors or finding interesting pixels in an image by suppressing those pixels that does not respond maxima to my filter, where the filter response to edges, corner, blob,...Gaussian come at the rescue to obtain descriptor and detector at different scale, gradient operator is very important to identify geometrically relevant info to describe, and identify interesting point in an image that are those where gradient change (edge) or does not change (blob). Computational efficiency is a key factor: pixels are many especially in high resolution images, superpixel concept, convolution in fourier domain that saves us some computation thanks to convolutional theorem, to be super fast: random sampling pixels from you images.

5 Wavelets

In time series we've seen that we can analyze info in spatial domain or frequency domain. The last looks for the characterizing/relevant frequencies in my signal/time series. What are the frequencies that have the higher likelihood on the whole time series? But may not be the most accurate description. Signals where

the characterizing frequencies in the first quadrant are very different from the characterizing frequencies in the last quadrant. Can I get this kind of analysis with a Fourier tool? No, cause I'm studying the time series as a global object, as a monolith, we're not localizing our frequency analysis in time.
Wavelets provide localized frequency analysis, localized in time for time series, in space for images. I analyze the spectral properties separately for each areas of the signal, different spectral analysis for different regions of my signal. With Wavelets I'm gonna be looking at my signal at different degrees of resolution in frequency domain and in time domain, localizing my analysis.
Define a different set of basis function, for Fourier the basis are sin and cos which has infinite support, they are almost non zero in all domain, wavelets have finite support, the energy is concentrated in a region of the space, this function is gonna be 0 almost everywhere except the place where the wavelet is centered. Wavelet has again an oscillating behavior, cause I wanna do again a frequency analysis. What characterizes the wavelet is the point in which it has support (the center) that i can shift, first DoF it's deciding where to center the wavelet, decide on which area of the time domain my wavelets is not zero, 2° dOf connected to the oscillatory behavior of the wavelet, allows to strech/dilate the wavelet or compress. The degree of streching of the wavelet (scale) is connected to a reduction of the characterizing frequencies (oscillation slower), if I compress the oscillation become faster, connected to an increase of the characterizing frequencies.
Localized function with 2 DoFs that allow me to shift it in time or space, to position wherever i want in my domain and to regulate the characterizing frequencies to which the wavelets respond.
Determining at which scale and at which position in space I'm using that wavelets, computing the convolution between my signal and my wavelets and look how much my signal responds to my wavelet. If my signal responds highly to a wavelets in a certain point, in that specific point in time there is a characterizing frequency, consistent with the scale of the wavelets that I'm using.
j: point in time where the wavelets is centered. k: is the scale. Depending on how I define psi different families of wavelets.
Useful if we fall outside the assumption of stationarity, frequencies can change in time.
Mother wavelets: a parametric family of functions, can obtain different wavelets by changing j and k.
Wavelets operate like Fourier transform, you can use it to understand what are the characterizing frequencies of a signal of course, but also you can take a signal that is noisy, project it into the frequency domain, identify the non relevant freq., zeros their coefficients and thanks to the inverse transform you reconstruct a cleaned version of the signal. Wavelets being a more general things than Fourier, Wavelets transform may not be invertible, there are types of wavelets for analysis, they provide fine grained details but cannot be inverted and other that lead to invertible transformations, more adequate for compression and signal cleaning. Wavelet transform is essentially akin to fourier transform, someone give us a signal, we're gonna make a choice of our wavelets, selecting the scale k and

the shift in time j, and then we're gonna be convolving it with the signal. We re gonna be convolving with multiple wavelets, at different scale and shift, as in Fourier analysis we obtain coefficients by convolving the signal with sinusoids or exponential at different frequencies.
Shifting operates on the j param, and scale operates on the k parameter, we're gonna be choosing the k param from a certain set of scales, and the same for j. j can be every point in time of my signal, in that case you are computing the continuous wavelet transform or non decimated wavelets transform, but there are other approaches that bind j, they make it proportional to the scale (discrete /decimated wavelet transform). You do this for 2 reasons: reducing computational cost of doing it and you want to gain invertibility. Continuous wavelets transform is to obtain a coefficient plot not different from power density plot in fourier transform. For each frequency k in Hz we have the power that was an approximation of the density of the distribution of how likely is that frequency in the signal, given a frequency the power spectrum tell me how probable is that frequency in the signal, coefficient plot does the same but we have a second dimension cause we cannot say that 10 Hz is a popular frequency in the signal, but 10 Hz is a popular frequency in a certain position in time of the signal, on the x-axis you've time. For a specific point in time what are the characterizing frequency, how strong is each of the scale in that particular time, scale close to 0 higher frequency, larger you go is lower freqeuncy.
Continuos wavelets transform select a discrete range of scale generated with a nice power of two and then interpolate between consecutive points. Continuos wavelets transform for exploratory time series analysis provides a fine graned analysis → coefficient plot.
Discrete wavelet transform use a discrete fixed set of scale, no need for all these in between scale computation, fixed numbers with no high resolution info in between them. I can decide to compute for each element of the time space and I get an undecimated wavelet transform. Or I can decide to make the number of points in time in which i compute my wavelet transform proportional to the scale, imagine you have a large scale wavelets so low frequency do not have sense to compute something that you expect to be low frequency for every point in time, no make sense the signal is changing very slowly so measure convolution at a very relaxed space, very compressed signal, high frequency changes you have to compute for much more many time steps. decimation → picking up j proportional to the scale.
DWT good because it leads to an orthonormal basis that you can invert without losing anything. signal →bring to scale space→ kill scale near to 0 → reconstruct back without the transformation being lossy → cleaned signal. It tipically leads to sparse representation → good for compression.
e.g. in human activity recognition with 9 time series from different sensors use the CWT to obtain a "fingerprint" of an activity and then use a PR algorithm that can match a new activity with a known activity pattern you can easily have a classifier e.g. use CNN cause Coefficient plot are images. CNN trained to recognize activity based on coefficient plot.

6 Generative Graphical Models

Generative highlights the fact that we deal with probabilistic models, before generative AI was associated uniquely to the use of deep learning models for generating text or images. Generative Machine Learning is historically associated with ML models that learn a distribution rather than try to solve a discriminative tasks. Dichotomy between being generative and discriminative. Discriminative models, e.g. a classifier, try to learn a separating hyperplane from more complex boundary function to create a separation between the classes that you want to divide/partion your samples of. Similar for a regression problem, you try to draw a very complex function in space in such a way it approximates nicely my future examples.
Generative learning problem of trying to learn a probability distribution that spans over you data, either directly or implicitly, you may not actually have an actual probability distribution over your data, but you rely on the assumption that you can get some sort of approximation of it, or a process that is supported by learning a distribution. Major difference: in a case you're trying to estimate a density, in another you're drawing bounds within classes, with a properly fit distribution you can classify, with a classifier you cannot compute probability, Generative learning is much more general and powerful than discriminative learning, but it is more complex, no free lunch.
How to learn a distribution? We'll learn a ML model that is expressed in terms of probabilities, the language used to define the ML model and its objective is probability. Graphical because most of the early approaches, in an attempt to simplify the complexity of the model, need to take simplifying assumption about the probability and to represent those simplifying assumptions we use a graphical formalism, not just syntactic sugar but a computational engine to speed up inference, on the top of the graphical formalism you can define inferential procedure, there is a way to transform probabilistic inference into an algorithm that operates on a graph, from math to computer science.
Once you have a distribution, we can sample from a distribution!! Every time you query from chatgpt you sample from a distribution, in order to generate highly realistic data, resembling the nature of the data it was trained on, but novel.
Having solved a generative learning task you have solved also all the supervised, unsupervised tasks...
In generative probabilistic model, you can craft into the model your prior knowledge, we can imbue the model with an inductive bias, NN do it only partially e.g. by putting symmetry in the neural architecture, I'm analyzing the nature of the problem understanding that there are invariances that I transform in weight sharing, but very limited.
Reality of world is complex, data is not easy and typically highly dimensional, in order to make it work we need to make assumptions/approximations tools. Techniques and tools to approximate stuff. Ways to approximate distribution to make our inferential problem computationally feasible.
Graphical model is a language to represent exponentially large probability dis-

tribution. Imagine a distribution over discrete values, if it is defined over a large enough set of random variables, computation explodes. Graphical formalism allow you to explicitly denote the fact that you're simplifying and factorizing it as a product of simpler distributions and also gives you tools to perform inference of that.
Concept of conditional independence: will help us to simplify this complex articulated joint distribution.
Different families of this graphical formalism/graphs: oriented graphs, undirected graphs, multigraphs, hypergraphs, ... get to different families of probability distribution that you can represent.
In NN inference is when you predict, inference is much more than that prediction, it is a thing that we use to do everything in a probabilistic model, when you're learning a parameter you're solving an inferential problem, when you're predicting an output you're solving an inferential problem, also model selection is an inferential problem... Graphs are made of nodes associated to random variables, a link/an edge between 2 nodes represents a probabilistic relationship, the meaning of the probabilistic relationship depends from the nature of the model we're looking at, or from the nature of the graphs: e.g. directed graphs: the existence of edges between 2 nodes represents sort of a causal relationship, actually not really causal, it can be causal but it can also be non-causal.
Let's assume there is causality. The existence of an oriented edge between two nodes (A→B) implies that A is the cause to B, A is sort of driving B, A can be used to characterize completely the behavior of B.
A graphical model tells you how much you can restrict your context when you're trying to assest things about a random variable, if I want to characterize the probability distribution of B, even if I'm in a graph with 3 other nodes, the only thing I need to characterize B is A.
Simplification, simple world described by four nodes/random variables A,B,C and D, if I reason in terms of probability the behavior of this world is gonna be defined if I know the joint probability P(A,B,C,D).
The graph tells me that if I want to infer something about B I do not need to reason in terms of C and D, I just need to fix A and this allows me to simplify to P(A,B,C,D) = P(B | A) * small dim. probabilities.
Undirected models can't encode causality, there is no a clear cause and an effect. Allow to describe symmetric relationships, the fact that 2 random var. varies together according to a law. This edge represents a soft contraint between random variables, these random variables gonna change but they do not change freely there is a path between them.
Dynamic models, graphical models that are allowed to change in time, the model itself is based on a number of random variables that changes over time, probabilistic equivalent of a RNN, RNN unfolds in time over the sequence length, we're gonna be looking at probabilistic models that unfold in time over a sequence length.
In machine vision you use an undirected graphical model to refine the predictions of a CNN or a visual transformer, to be sure that whatever is predicted in output, what are the pixels of the roads, what of the vehicle , e.g semantic

segmentation tasks, the CNN predicts for area of pixels a class: I know that is a machine vision problem applied to an autonomous car problem, if I know the setting I know there are constraints to the problem: unless something go wrong the sky must be above the car, and the road below the car...you cannot encode this kind of knowledge in a NN, you can encode this facts in constraints that you place in the undirected graphical model, you take the output of the CNN, and you check if it is consistent with the prior knowledge, Markov Random Field output layer for a CNN.
Provide mathematical machinery to train generative deep learning model, they try to solve the same problem using a very powerful and flexible approximator to infer the distribution $\rightarrow$ a DNN.
Try to do this with parametric models from probability.
Building blocks: In a probabilistic world we describe the world in terms of random variables, a function that associates values with outcomes of a stochastic process. E.g: coin toss, you toss a coin, two possible outcomes: head or tail, the random variable associates 0 to head and 1 to tail.
Random variables can be used to describe events/processes that lead to discrete values, but also to describe continuous stuff.
For the purpose of ML, a random variable when it's in input it describes a dimension of interest. E.g: Classifier for subject of a clinical study, a random variable will describe an aspect of my patient, if a patient is described by 100 different measurements they're gonna be one hundred of random variables describing those measurement, the equivalent of the features. There will also be the equivalent of hidden neurons, cause at some point I'll have random variables associated with stuff that I cannot measure in the dataset. Exactly what happens in NN, the input layer of a NN it's data that comes from the dataset, the output layer is data that at training time come from the dataset, what you have in the middle is stuff that you made up, doesn't exist in the dataset, just a bunch of numbers processed by the NN. The equivalent are the unobservable or latent random variables, random variables not associated with the observable data, but are there because otherwise the model is too simple, e.g. cannot solve the XOR problem.
Probability function : measure the probability of an outcome/event, or in another sense they measure areas.
We do not work with a single random variable, that would be equivalent to work with a model with a single input neuron, we work with models that need to be described by multiple random variables, then we need the glue to keep all the random variables together? In NN we have the links between neurons, the synapses, weights, keep the NN together, neurons do not float in space they're connected. What connects random variables? Other probabilities, the joint probability looks at the all VS all connectivity of random variables.
The joint probability is our master probability, if we know the joint probability of all the variables in the world we can do everything, we can get every probability out of the joint probability and we can make any inference out of the joint probability. Problem: you typically do not have it, and even if you have it, any inferential problem becomes a nightmare, so we postulate the existence

of that thing and then we find ways to break it into pieces and those pieces are not gonna be involving single random variables but groups of random variables. Conditional probability measures how the realization of an event y, knowing that y is occurred, is not anymore in the probability world, how does the knowledge of the fact that y is occurred influence the probability of other events, it measures how the realization of an event influences the probabilites of other events.
$P(x_1, x_2, \ldots, x_n \mid y)$ where on the left of | is the conditioned side, y is the conditioning side. $P(x \mid y)$ is a family of probabilities, cause I have a probabilities for each different value of y. If i change value of y i change how I asses x. $P(x \mid y)$ entails $P(x \mid y = 0)$ and $P(x \mid y = 1)$ are 2 separates and independent distribution, This means that

$$\sum_x P(x \mid y = 0) = 1$$

$$\sum_x P(x \mid y = 1) = 1$$

If we have a joint probability distribution we can extract the marginal, and aside from the joint and the marginal we have conditional distributions, more properly exists a family of conditional distributions, a different one for each choice of the conditioning variables.
Now we have both the building blocks to define a model: random variables and the glue to keep them together: probability distributions. We need tools to manipulate stuff, Chain rule of probability: a conditional joint distribution can be rewritten as a product of distribution over a single random variable. We need first to pick up an ordering,

$$P(x_1, \ldots, x_n \mid y) = P(x_1 \mid y) \cdot P(x_2 \mid x_1, y) \cdot P(x_3 \mid x_2, x_1, y) \cdots$$

So we're taking a joint distribution across multiple random variables and we transform into a product of distribution each on a single variable, but conditioned possibly on many many variables e.g.

$$P(x_n \mid x_{n-1}, x_{n-2}, \ldots, x_1, y)$$

a lot of variables in the conditioning side. possible gain?? If I can make elaborations about the dependencies between random variables, and simplify this formulation, if I have a Bayesian network with a structure of dependencies e.g. x2→x3, if I know x2 I have everything is needed to characterize x3, x1 is not relevant in characterizing x3, instead of writing $P(x_3 \mid x_2, x_1, y)$ i write only $P(x_3 \mid x_2)$. Allow to simplify my conditional probability, use notion of conditional independence between random variables and use them to simplify this conditional distribution, factorize this conditional joint distribution in a product of much simpler distributions. Simpler how much is the amount of conditional independence I decide to put in the model

Marginalization: derives from the fact that in my distribution I have a sum to 1 constraint

$$P(x_1) = \sum_{x_2} P(x_1, x_2)$$

$$\sum_{x_2} P(x_2) = 1$$

We start from the fundamental axiom that a joint probability distribution over discrete variables x_1 and x_2 must sum to one:

$$\sum_{x_1} \sum_{x_2} P(x_1, x_2) = 1.$$

By rearranging the order of summation, we can write:

$$\sum_{x_1} \left(\sum_{x_2} P(x_1, x_2) \right) = 1.$$

We define the marginal distribution $P(x_1)$ as:

$$P(x_1) = \sum_{x_2} P(x_1, x_2).$$

Substituting this into the equation above, we obtain:

$$\sum_{x_1} P(x_1) = 1,$$

which confirms that $P(x_1)$ is a valid probability distribution. Therefore, the marginalization rule $P(x_1) = \sum_{x_2} P(x_1, x_2)$ is a direct consequence of the sum-to-one constraint on the joint distribution $P(x_1, x_2)$.
Why you pass from a simpler to a more complex distribution with marginalization. You might not know P(x1) but you may know $P(x_1 \mid x_2)$. This typically happens in probabilistic ML models, where x1 is tipically the equivalent of data and you do not know the probability of data because if you know the probability of data you don't need ML, but with a ML model you can compute the probability of data given something else, that is to say some non observable random variables which are the equivalent of the hidden neurons, if you have some hidden explenatory variables that you do not know how to characterize but you know that they exist, just like hidden neurons, then you can explain better your observable data.
Bayes rule: d is the data, a set of observations, h are hypothesis, a ML model. $P(h_i \mid d)$ probability of an hypothesis given the data and flip it:

$$P(h_i \mid d) = \frac{P(d \mid h_i)\, P(h_i)}{P(d)}$$

used to do inference, $P(d \mid h_i)$ = likelihood, how you interpret your data, under

specific hypothesis, you assume h_i is true and interpret any data you see under that hypothesis.
When we're training models out of MLE (Maximul Likelihood estimation) we're sort of fitting data to our view of the world. P(hi)→ prior : a priori how much I trust a specific hypothesis, its probability. P(d)→ is marginal distribution over the data, you do not have it. P(hi|d) → posterior, a posteriori reassesment of my hypothesis, how much I value my hypothesis after having seen the data
To resume, let:

- h_i be a hypothesis (e.g., a model or a parameter setting),
- d be observed data (your dataset).

Bayes' rule states:

$$P(h_i \mid d) = \frac{P(d \mid h_i) \cdot P(h_i)}{P(d)}$$

Where:

- $P(h_i \mid d)$ is the **posterior** — the probability of the hypothesis after seeing the data,
- $P(d \mid h_i)$ is the **likelihood** — how well the data is explained assuming the hypothesis is true,
- $P(h_i)$ is the **prior** — your belief about the hypothesis before observing the data,
- $P(d)$ is the **evidence** — a normalizing constant ensuring the posterior is a valid probability distribution.

Maximum Likelihood Estimation (MLE) ignores the prior and maximizes only the likelihood:

$$\hat{h}_{\text{MLE}} = \arg\max_h P(d \mid h)$$

This corresponds to assuming all hypotheses are equally likely a priori and choosing the one that best explains the observed data. In this view, we are fitting the model to the data, assuming no prior preference over hypotheses — effectively bending the model to match the world.
The marginal probability can be rewritten using the marginalization trick.
Conditional indipendence: $P(X, Y \mid Z) = P(X \mid Y, Z) \cdot P(Y \mid Z) = P(X \mid Z) \cdot P(Y \mid Z)$
If $X \perp Y \mid Z$ (i.e., X is conditionally independent of Y given Z), then:

$$P(X, Y \mid Z) = P(X \mid Z) \cdot P(Y \mid Z)$$

It means that two things that might not be independent can become independent if a third event is observed.

What does learning mean in this context? Discovering the values of the joint distribution of the random variables that describe my world. Difficult to estimate, might not know how to write it, might not know the family of the distribution, might be too complex, gonna introduce assumptions and approximations to make it possible to estimate this distributions and since we are ML guys we'll have a parametrized view of this distribution, this distribution must have some parameters in order to learn.
How to link these concepts of statistics/probability with concept of ML? Inferential problem is how to determine the distribution of the values of one random variable knowing the values of other random variables.
From a ML view, this inferential problem becomes the problem of: what if I'm given a set of hypothesis and some data, how can I use this known things to infer things about a new random variable, where a new random variable is a new sample, how can i make a new prediction X after that somebody has given me data and some hypothesis.
Learning becomes an inferential problem, if your hypothesis are random variables you parametrize your hypothesis with a set of parameter theta, learning become the problem of finding theta that maximize a probability distribution.
Given a set of observations, a probabilistic model of a given structure, how do I find the parameters theta of its distribution?
Find the best hypothesis h_θ given the data.
3 approaches of inference in ML: 3 possible views of declining optimality and increasing feasibility: no free lunch in ML to enhance feasibility AND scalability you lose optimality.
The optimal way to make inference in ML is the Bayesian approach: $p(X \mid d) = \sum_i P(X \mid h_i)\, P(h_i \mid d)$
We want to compute the posterior predictive distribution:

$$P(X \mid d) \tag{1}$$

Step 1: Marginalize over hypotheses h_i

$$P(X \mid d) = \sum_i P(X, h_i \mid d) \tag{2}$$

Step 2: Apply the chain rule

$$P(X \mid d) = \sum_i P(X \mid h_i, d)\, P(h_i \mid d) \tag{3}$$

Step 3: Assume conditional independence $(X \perp d \mid h_i)$

$$P(X \mid h_i, d) = P(X \mid h_i) \tag{4}$$

Final result: Posterior predictive distribution

$$P(X \mid d) = \sum_i P(X \mid h_i)\, P(h_i \mid d) \tag{5}$$

$P(X \mid d)$ is a prediction i want to do on some data I was given. I do not have the distribution $P(X \mid d)$ between the observable data and the random variable I want to predict on, but I have hypothesis, as in ML terms, and I introduce them via marginalization. Since i cannot estimate $P(X \mid d)$ directly, I can use my hypothesis $\sum_i P(X, h_i \mid d)$ and then apply the product rule $P(X, h_i \mid d) = P(X \mid h_i, d) \cdot P(h_i \mid d)$ by applying the chain rule.
$P(X \mid h_i, d)$ is the probability of the unknown given a specific hypothesis (that is true) and the data, the ML assumption comes into play, if I'm in a world in which my hypothesis can explain my data and future data, if I know that a specific hypothesis is true, d does not influence any more the prediction about X, in other term I have a trained ML model, I do not need to keep my data. I can just use the trained model to make predictions, in probabilistic terms : $P(X \mid h_i, d)$ simplifies in $P(X \mid h_i)$
So in order to make an inferential decision about X, with a ML mindset, I assume that exists a set of hypothesis, possible fitted ML models, I use marginalization to introduce the hypothesis, factorize with chain rule and apply the above simplification on the conditional distribution.
In order to make an inferential step on X, given the data, I can take all my hypothesis (sum overall i), I pick the first hypothesis, h1, I make a prediction with P(X|h1), how much I trust this prediction ? $P(h_1 \mid d)$ the posterior, how much I trust this specific hp having seen the data, how much i trust this ML models after the training on your data, I compute a weighing voting, compute an inferential step with all your hp, weighed by how much you trust each specific hypothesis.
The optimal theoretical classifier predicts according to this rules, you cannot beat a classifier that marginalizes overall the possible hp. Hp can be parametrized by θ, rather than be discrete hp if you take a NN, a MLP as architecture is a single model but multiple hp, it is as many hp as the choice of the weights, the summation become an integral overall possible choices of your param. This is a theoretical optimum but unfeasible.
Take a step of simplification, the problem is summation or integral, I cannot integrate overall the possible things, it does not have an analytical solution, instead of visiting all the possible hp, I choose a single hp and ask that hp to make a vote for me, I pick the one that i trust more, the maximum a posteriori hp, the one that maximize the posterior probability, predict X using h_{MAP}.

$$h_{\text{MAP}} = \arg\max_h P(h \mid d) = \arg\max_h P(d \mid h)\, P(h) \tag{6}$$

Using Bayes I rewrite as the product likelihood * prior, P(d) at the denominator disappear because I maximize w.r.t. h and I remove a constant that does not depend on h.
Picking up an hp means selecting a set of param of the NN, if I train my NN to maximize the posterior distribution, if I find θ that maximizes the posterior distribution than if I use those params to predict X then I'll be doing maximum a-posteriori inference. MAP (Maximum A-Posteriori) is the closest approximation to the optimal Bayesian classifier.

However we need an additional step of simplification to make things feasible: the problem stands in the multiplication of $P(d \mid h)P(h)$, I can sacrifice the one without the data, I make the assumption that apriori all hp are equivalent/equiprobable so the maximation does not depend on the prior. The problem simplify in finding the h that maximize only the likelihood: $\arg\max_h P(d \mid h)$
I'll look to the params that maximize the likelihood, in order to train a probabilistic model I need to find a way to write the likelihood in function of the params, then become a function of the params that I can maximize, with the derivative, looking for stationary points, ... The more data you have the less it matters the approximation, because in principle if you have infinite data all the approaches converge to the same solution under the law of large numbers
MAP is a regularization of ML estimation, because in ML you have a preference for the simplest possible model for the job.
The MLE does not have any term to keep the complexity of the model under control. In principle if I have a Look-up table, will give the MLE of your data, but really bad generalization. The model that remembers better the training data, MAP has the term P(h) to measure your hypothesis, penalizes complex models. If i take the logarithm and flip the sign i notice that map is $\min_h \big(-\log(P(d \mid h)) - \log(P(h))\big)$ the second term is the num.of bit to store the hp h, a measure of complexity of my hp, a look-table need many bits to store all of the data, too complex : $-\log(P(d \mid h))$ is the num. of bit to store the part of the data that does not fit the hp, assuming I have an hp where i can store stuff, how many additional bit I need to store the info on my data that are not in the hp. Look up table first term small, but high second term.
MLE I'll pay the price of favoring storage over compression, instead MAP trades between storage complexity and compression.
MAP costs computationally but in principle is the regularized solution.
The reason why in NN we use MLE and then dropout to simplify.
MLE learning is looking for those params θ of the model using as optimization principle the maximization of the probability $P(d \mid \theta)$ i.e. the likelihood, i need to write $P(d \mid \theta)$ as a function of parameters, and then it's easy to optimize a function w.r.t. to some params. I'll look to stationary point, take the derivative of the likelihood function w.r.t. to the params and put it to 0, and find the learning rule.
This is typically straightforward when the random variable X or whatever I'm maximizing in $P(d \mid \theta)$ is expressed only in terms of visible stuff, e.g. random variables for which I've examples in my dataset, I'm happy. Distribution that describes the results of tossing a coin, I can train a model to infer the param of this model, single parameter θ which is p = probability of heads, 1-p is probability of tails. The data are the results on 10 coin tosses, 6 heads, 4 tails. $\theta = 6/10$. MLE of param theta.
You decide the probability distribution for the ML model e.g binomial, identify the param theta, plugged in there with a logarithm, differentiate and solve w.r.t. to theta. Works nicely when everything is observable, but ML works with non observable stuff, like with NN, you must introduce unobservable random variables.

ML with hidden-latent variables, the model will contains both observed random variables X and Z unobserved random variables, assumptions I make within the model, not associated with observable stuff.

$$\mathcal{L}(\theta \mid X, Z) = P(X, Z \mid \theta) = P(Z \mid X, \theta) \cdot P(X \mid \theta)$$

Hidden variables tries to explain something that you can't see, with basing on something you can't really say but make it sense if it was there.
P(X|theta): true, real likelihood only defined in terms of observable data
P(Z|X,theta) probability of unobservable information given the observable data. It's a posterior probability, probability of something I cannot observe based on something i can observe and this is gonna weight my likelihood.
EM algorithm it's a 2 step iterative algorithm, used when you want to do MLE estimation of your params and your model involves Z, non observable latent variables. For each iteration 2 step: compute your current estimation of posterior P(Z|X,theta) using the current value of theta (you held theta fixed), then you plug the value of that distribution, the posterior, in the equation P(Z|X,theta) * P(X|theta) and find new params theta' that maximize it. Then repeat until your likelihood stops increasing.
The expression

$$\mathcal{L}(\theta \mid X, Z) = P(X, Z \mid \theta) = P(Z \mid X, \theta) \cdot P(X \mid \theta)$$

represents the likelihood of the model parameters θ given both the observed data X and latent (unobserved) variables Z. This form appears in probabilistic models that include hidden structure, such as Gaussian Mixture Models or in the Expectation-Maximization (EM) algorithm.

Terms:

- X: observed data.
- Z: latent variables — unobserved, hidden random variables (e.g., cluster assignments).
- θ: parameters of the model.
- $\mathcal{L}(\theta \mid X, Z)$: the likelihood of the parameters given the data and latent variables (also called the complete-data likelihood).

Decomposition via Chain Rule: The joint probability of X and Z given θ can be decomposed using the chain rule:

$$P(X, Z \mid \theta) = P(Z \mid X, \theta) \cdot P(X \mid \theta)$$

This expresses the joint distribution as a product of:

- $P(Z \mid X, \theta)$: the posterior distribution over the latent variables given the data and model parameters.
- $P(X \mid \theta)$: the marginal likelihood of the observed data under the model.

Use in EM Algorithm: This decomposition is fundamental in the Expectation-Maximization (EM) algorithm:

- In the **E-step**, the algorithm estimates the posterior distribution $P(Z \mid X, \theta)$ of the latent variables.
- In the **M-step**, it maximizes the complete-data likelihood $\mathcal{L}(\theta \mid X, Z)$ with respect to θ.

This approach allows the model to learn parameters even when part of the data (i.e., Z) is hidden.

$$\theta^{(k+1)} = \arg\max_{\theta} \sum_{\mathbf{z}} P(\mathbf{Z} = \mathbf{z} \mid \mathbf{X}, \theta^{(k)}) \log \mathcal{L}_c(\theta \mid \mathbf{X}, \mathbf{Z} = \mathbf{z})$$

Explanation of terms:

- $\theta^{(k)}$: Parameter estimate at iteration k
- $\theta^{(k+1)}$: Updated parameter estimate at iteration $k+1$
- $\arg\max_\theta$: Finds the value of θ that maximizes the expression
- $\mathbf{X}$: Observed data
- $\mathbf{Z}$: Latent (hidden) variables
- $P(\mathbf{Z} = \mathbf{z} \mid \mathbf{X}, \theta^{(k)})$: Posterior probability of latent variables given observed data and current parameter estimate (this is the **E-step**)
- $\mathcal{L}_c(\theta \mid \mathbf{X}, \mathbf{Z})$: Complete-data likelihood (i.e., the likelihood assuming we had access to the full data including latent variables)
- $\log \mathcal{L}_c(\theta \mid \mathbf{X}, \mathbf{Z} = \mathbf{z})$: Log-likelihood of the complete data (this is used in the **M-step**)

This expression computes the expected complete-data log-likelihood under the current parameter estimate, then maximizes it with respect to θ to update the parameters.

EM alg. estimating the values of params theta at k+1 iteration by maximizing an expression with the posterior estimated with the value of theta at iteration k times the loglikelihood function, there is a sum over k because I do not know the value of Z, in order to introduce the value of Z in my MLE i need to sum over all the possible value of Z effectively multiplying by one.
EM is the equivalent of backprop, needed in NN as soon we introduce hidden neurons, we need EM as soon as we introduce hidden variable.
In a regime with scarse data MLE will give estimations that highly depends by the sampling, not only a MLE will give me completely out of the target or very biased estimation, but overconfident, very small standard deviation.
The effect ot the prior in the MAP estimation is to fattening your gaussian and to make the model not to become overly optimistic.

Working with an explicit representation of all the possible instantions of the random variables become soon to complex if i have to do all the combinations, graphical models avoid to have a big enormous look-up table but represents that as a set of smaller dimensional tables that are conditional probabilities table, find a compact graphical based rep. of those joint distributions broken down into simpler once.
Bayesian network has rand. var on the nodes and links between the nodes that denotes form of dipendencies: P(Y1, Y2, Y3, Y4, Y5) can be rewritten as P(Y1) * P(Y2) * P(Y3| Y1, Y2) * P(Y4|Y3) * P(Y5|Y3).

7 Conditional Independence: Representation and Learning

Represent joint distribution, model uncertainty in the world. The goal is to represent the joint distribution of a set of variables, we want a way to represent all the possible interactions between random variables.
Represent joint distribution is important for different applications: sampling new instances, in generative model you're able to represent how variables interact and you want to sample new instances as they are taken from the world, that follow the same distribution. Probabilistic model to infer the value of hidden variables, not everything is observable at the same time, classification tasks: you've a lot of variables that contain some sort of features and you want to classify, obtain a value of something you do not observe, at inference time.
Estimating the likelihood of a configuration, think of the params as variables, by estimating the likelihood of the params as you do when you want to train a model you use implicitly a probabilistic formulation, you want to see which one is the more likely configuration of a set of variables, this way you're able to formalize the problem of learning in probabilistic models.
N distinct random variables, each one can take 1 out of k different values. How many params to represent this joint prob. distribution? No assumption about independence of the variables, consider all possible configurations. k^N - 1 configuration. You can define just all of them except one and the last one is 1 - the sum of all the others.
Probabilistic models are used for text, images ... you cannot represent the joint distribution for all the config. We need to reduce the number of params.
Instead of considering all of them at the same time, we start by considering one variable at the time, we use chain rule, you start by giving the probability of one variable and then you condition on the variable you already have to get the new variable. You can use any ordering you want in principle.
(num.var)! possible ordering
How to use graphs to represent probability distributions. Directed graphs: the edge means to get this variable I'm conditioning on what's before.
Jointly representing random variables is complicated and requires a lot of params,

so we exploit the chain rule to treat each variable one at the time and we found a nice graphical representation. Does this solve the problem? This reduce the params ?? Before we had k^N-1 param: P(Y1, Y2, Y3) = P(Y1)P(Y2|Y1)P(Y3|Y1,Y2), for the first object we need 1 param, for the second 2, for the third 4 (4 possible configurations of Y1 and Y2, 00,01,10,11), ... 1+2+4 = k^N- 1, simply decomposing using the chain rule does not give any advantage, we get the exact same number as param.
Marginal and conditional independence, independence: using the chain rule we were not saving anything, we were still conditioning on everything that comes before, we want to remove some stuff in the conditioning parts, to reduce the number of params we must reduce the size of the set on which we're conditioning. We can do this with the notion of independence.
Bayesian interpretation of probability theory, Random Variables are a measure of how much you know about something, when two variables are (marginal) independent, knowing something about X does not change your knowledge/uncertainty about Y. Best scenario ever: everything is indipendent, we only need k*N where k is the number of possible realizations of a random variable. P(Y1,Y2,Y3) = P(Y1)*P(Y2)*P(Y3) really compact representation but not reasonable, too strong assumption, in real world phenomena interacts with each other.
Conditional independence:

$$\begin{aligned} P(X, Y \mid Z) &= P(X \mid Y, Z) \cdot P(Y \mid Z) \\ &= P(Y \mid X, Z) \cdot P(X \mid Z) \\ &= P(X \mid Z) \cdot P(Y \mid Z) \quad (\text{if } X \perp Y \mid Z) \end{aligned}$$

X e Y are not anymore marginal dependent, they're not independent by default let'say, but they're independent when conditioning on Z. Bayesian interpretation: knowing Z make X and Y independent, there's nothing that X can give to Y that Z already gave and viceversa.
Y1 is independent of Y3 given Y2: P(Y1,Y2,Y3) = P(Y1)*P(Y2|Y1)*P(Y3|Y1,Y2) = P(Y1)*P(Y2|Y1)*P(Y3|Y2)
We removed Y1 from the conditioning side, because all the information that Y1 could give us about Y3 is already there from Y2.
The information I can get from Y1 is already there from Y2. 1 param from the first term, 2 for the second and 2 for the third.

Bayesian network: is a **directed acyclic graph** (DAG), where nodes represent random variables, edges represent the conditional independence relation, if Y3 depends on Y1 and Y2 there's gonna be an edge, from Y1 to Y3 and from Y2 to Y3.
Color: shaded nodes to represent observed variables, empty nodes unobserved variables, with a probabilistic model we cannot observe everything anytime, e.g. classification problem we're gonna have the covariates in blue and the label in white, to state I'm not gonna observe this variable at inference time, that is the task, what I'm trying to recover from the other variables.

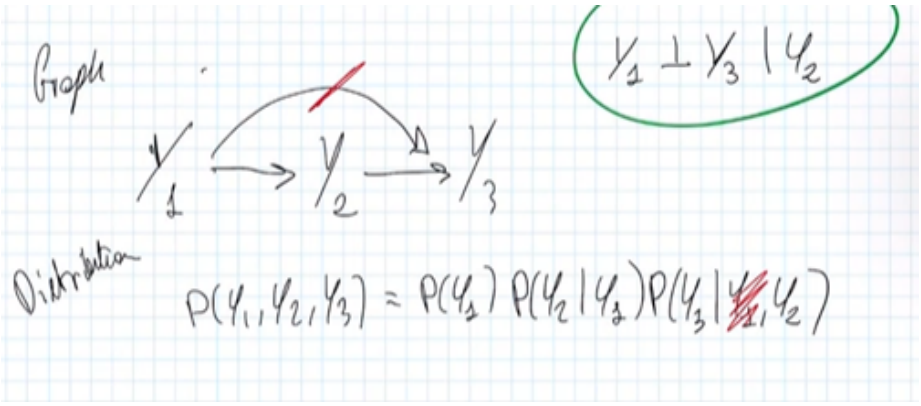


Figure 1:

In the discrete case we're gonna represent this Bayesian network by using a conditional probability table, for each node a small table that represents the distribution of that particular node given the parents as we did before in the decomposition. In the more general scenario you can think that you're gonna have your function approximator, like a NN, that takes in input the values of the parents, and is gonna give you out the density of that particular node.
If we assume that L is the max number of ingoing edges, (ingoing nodes' parent, outgoing nodes' children), is the maximum number that a variable can have, determine the size of the conditional table of that particular node, and so the number of parameters is at most $N(k-1)^L$. For each node with r parents k^r, different combinations of parents value, for each combination you need k-1 indip. params to define a valid prob. distribution.Hence the number of params need for a node is $(k-1)*k^r$. Considering r ¡= L, there are N nodes $\rightarrow$ $N*(k-1)*k^L$. This is just an upper bound, the sparser the network, the less edges we have, the less complex is gonna be the model, we reduce the number of parameters, as an analogy in the more general continuous case, the sparser the network, the easier will be the function approximator.
Do you have an intuition about the meaning of the edges? Try to image to model something, the probability of observing rain and the probability of observing an open umbrella, how would you put that in a Bayesian net? Suppose they're both observable. COMMON MISTAKE in Bayesian network is to assume that the relationships are causal, in general this is not true. There is a causal interpretation of Bayesian network, but in general it is not true, they're just representing statistical dependence.
How to represent Bayesian network? Some statistical dependencies might be replicated in the same distribution, you might have different objects with the same conditioning sets in the distribution, this means in the graph having the same parents. The plate notation is a way to represent the network in a more compact way.

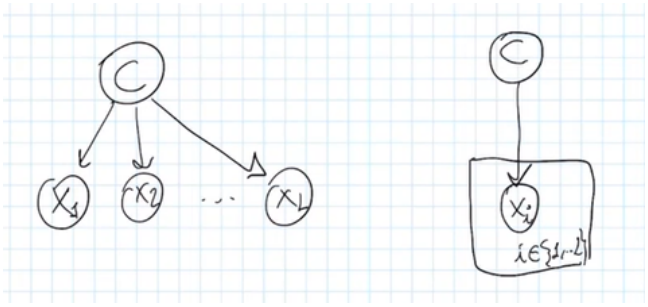


Figure 2: Plate Notation

You can replicate also for samples, you can say that there are L different features, X variables, covariates and you can replicate that as well as the samples, N different realizations of each variables.
Boxes denote the replication of the same variables.

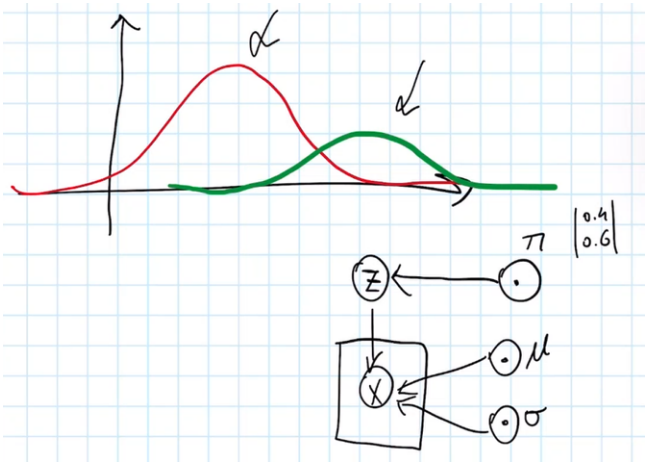


Figure 3: Gaussian Mixture models

Local Markov Property is fundamental property of Bayesian network, if the graph represents a given distribution, than from graphical properties we can get conditional independent relations. A random variable is conditionally independent of all its non-descendants (set of all the variables less the descendants) given a joint state of its parents.
Y_v is independent of the variables without the children given the parents and this holds for all possible variables.

When you have a graph, you've a way to get back all the conditionally indepen-

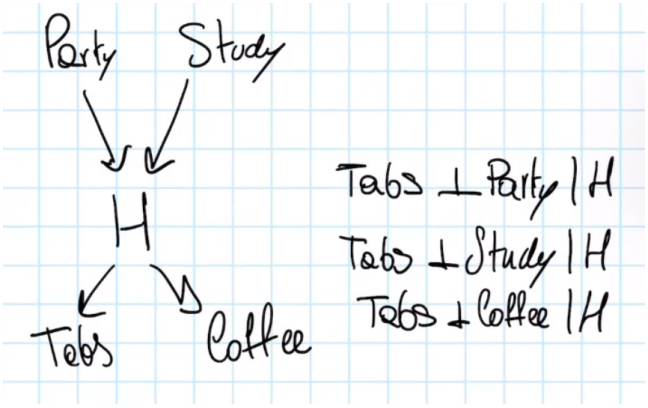


Figure 4: Local Markov property

dence relations. We'll see a generalization → Global Markov Property.
Topological ordering: is an ordering of the nodes that respects the directed edges in the graph, it is not unique. Then apply the chain rule. This depends on the topological ordering. Use the local Markov property to derive conditionally independence relations which are represented by the graphs and start cutting the conditioning sets.
However, with different topological ordering we would have got to the same ordering.
Once we have a factorization, we can use it to get new samples, new instances (Ancestral sampling): use the factorization we've seen for computing the density of an instance, to actually sample by following the same order and by following the same relations. The same with different topolog. ordering.
3 fundamental substructures: **Tail-to-Tail** or **Fork** or if you have a causal interpretation of Bays. net is common cause. **Head-to-Tail** or **chain** structure or causal effect. The same for the opposite direction. **Head-to-Head** or **Collider** structure or **Common Effect** or inverted fork.
Important: Local Markov property is not telling you that you've to condition on 1 parent or on N parents, you have to condition on **ALL** the parents.

We're conditioning on Y2 to render Y1 and Y3 independent. To condition on something, I need to have that value to condition on, I need to observe that value. The graphical intuition is that you're blocking the path. You have a path between Y1 and Y3 and you're blocking that by observing a variable in the middle.

Do we expect a different probability distribution if we completely invert the chain? In a sense the last two structures are equivalent because they entail the same set of conditional independence relations. This structures are equivalent, Markov equivalence class, two graphs are in the same equivalence class if

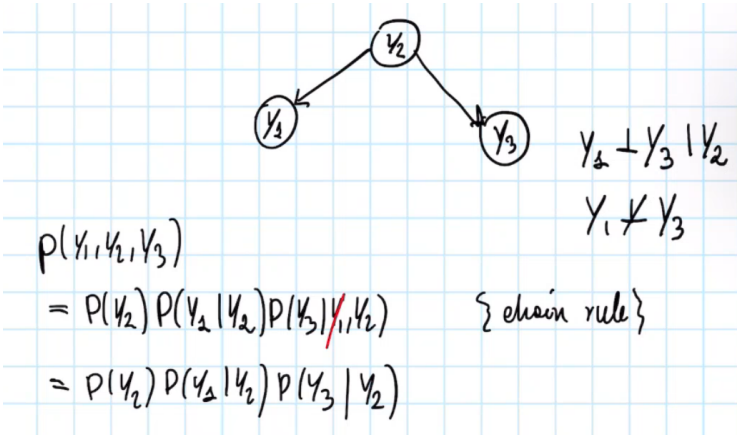


Figure 5: Fork

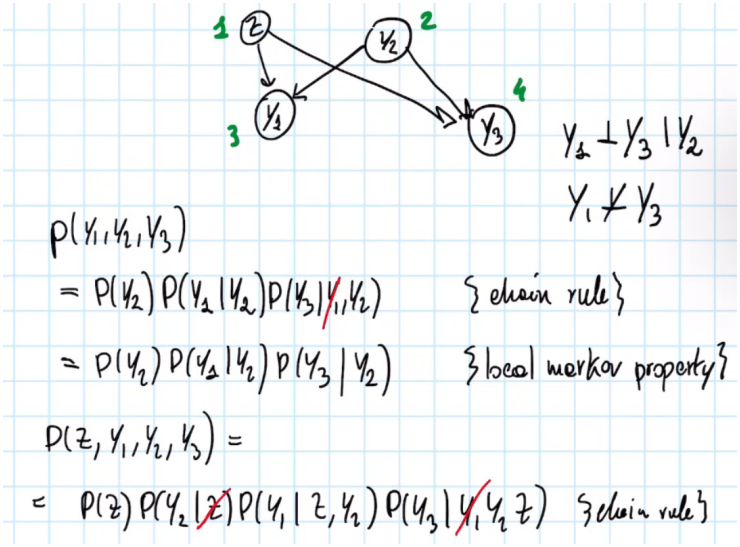


Figure 6:

they represent the same set of conditionally independence relations. Graphs do not provide an unique way to represent conditional independence relations, you could have different design choices to represent the same information.
The last fundamental structure that is not equivalent to the previuous is the collider or head to head connection.

The problem here (with the collider) is that Y1 and Y3 are independent (marginally)

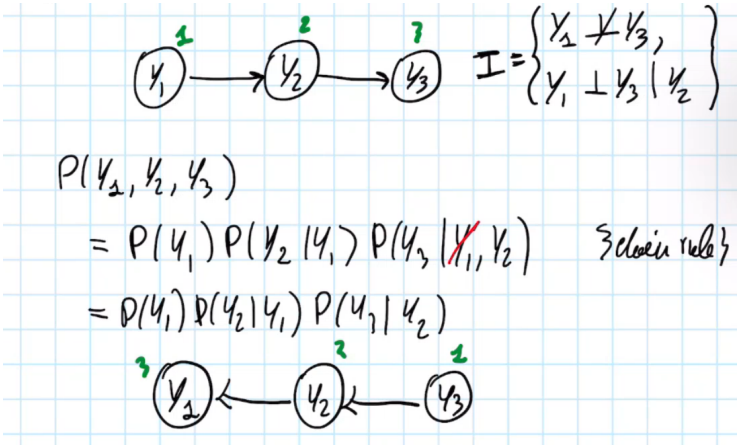


Figure 7: Chain

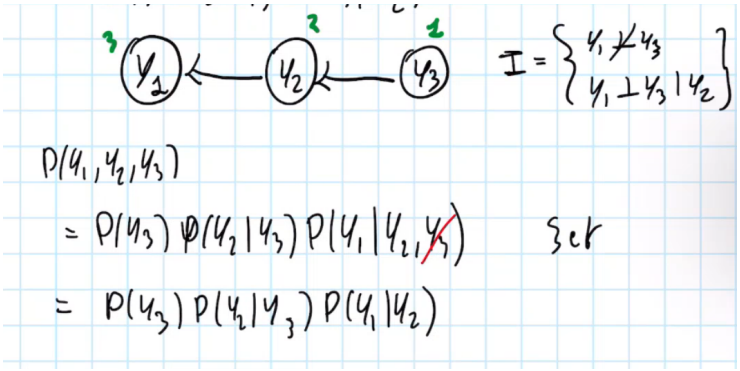


Figure 8: Reversed chain

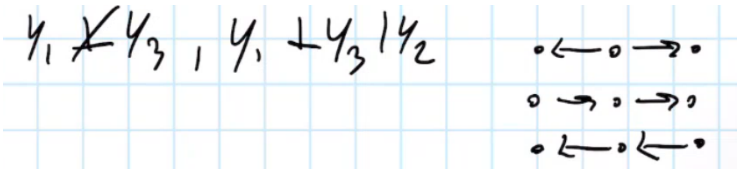


Figure 9:

but if you condition on Y2, you have some info on Y2, they are not independent anymore. There is information flowing through the observation of Y2. Previous structures conditioning makes 2 variables conditionally independent, here conditioning makes two variables conditionally dependent.
We're using directed graphs, we have directions for the edges, we're going to

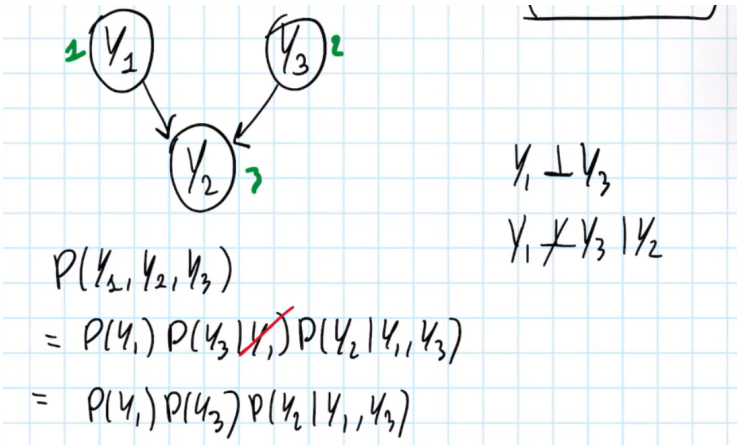


Figure 10: Collider

define paths on the undirected edges.
The path r is blocked by a set Z (possibly empty) if one of the following holds:

- the path r contains a fork (tail-to-tail) Yi ← Yc → Yj such that Yc is in Z.
- r contains a chain (head-to-tail) Yi→Yc→Yj such that Yc is in Z.
- r contains a collider (head-to-head) Yi→Yc← Yj when neither Yc neither its descendandts are in Z

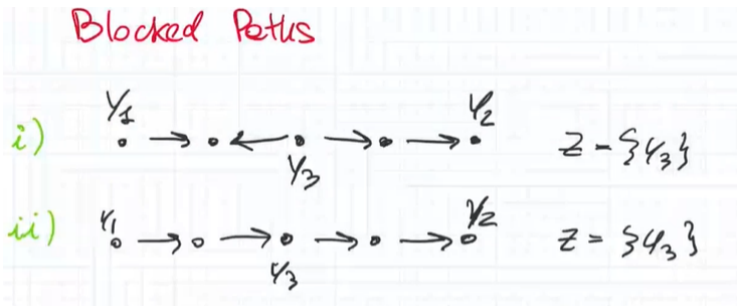


Figure 11:

d-Separation, where d stands for dependence: Given an undirected path r from Y1 to Y2 the path is d-separated by Z if there exists at least one node in Z for which path r is blocked.
d-separation between nodes: two nodes Yi and Yj are d-separated by Z, whenever all the undirected paths between Yi and Yj are d-separated/blocked by Z.
D-separation is a graphical property, $Y_1 \perp_G Y_2 \mid Z$ means they're only in a

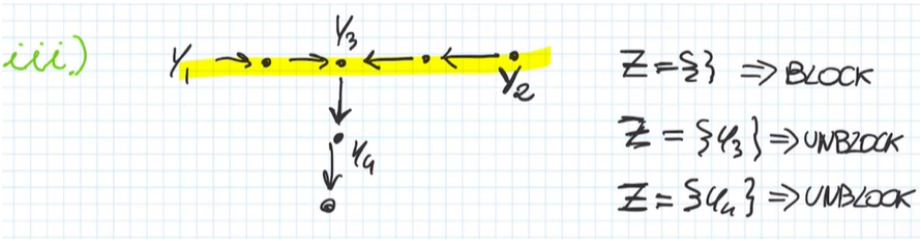


Figure 12:

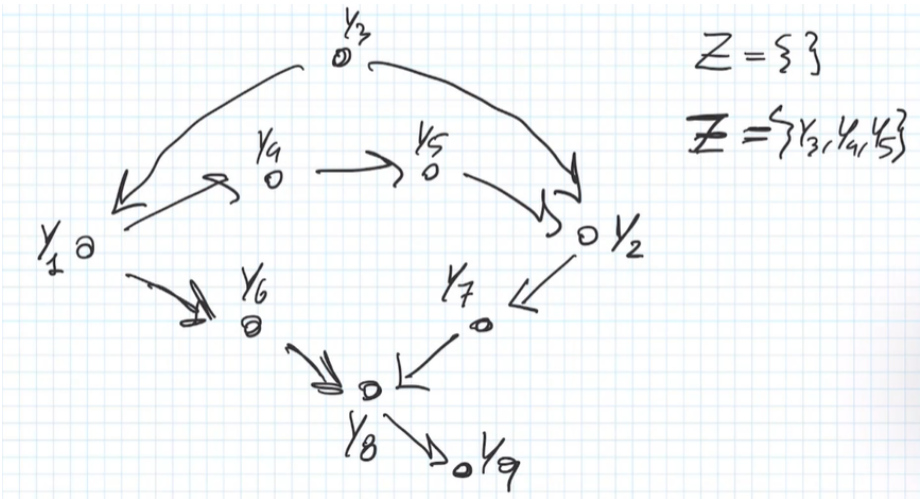


Figure 13:

graph and we can use a set Z to provide d-separation. What is the relations between d-separation and conditional independence?
Global Markov Property: whenever two nodes are d-separated by Z, then the corresponding variables are conditionally independent given Z. Whenever d-separation holds conditional independence holds.

Markov blanket: you've a Bayesian network with a lot of variables, but you're interested only in one variable Y, that somehow is your target. By using this idea of conditional independence you know that you don't really need the values of all possible variables, you only need a subset of variables. D-separation is telling you which are the variables you need, cause there is a set of variables that captures the dependencies from all the other. The Markov Blanket of the node Y, Mb(Y), *is the minimal set of vertices that shield the node from the rest of the Bayesian Network*. In a DAG, the Markov Blanket of Y contains the parents of Y, the children of Y and the parents of the children of Y.
Given these variables (in the blanket) you know that Y is d-separated from all

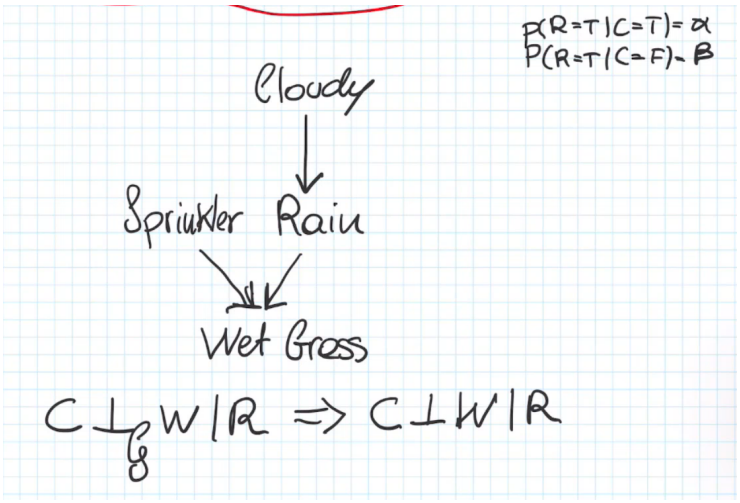


Figure 14:

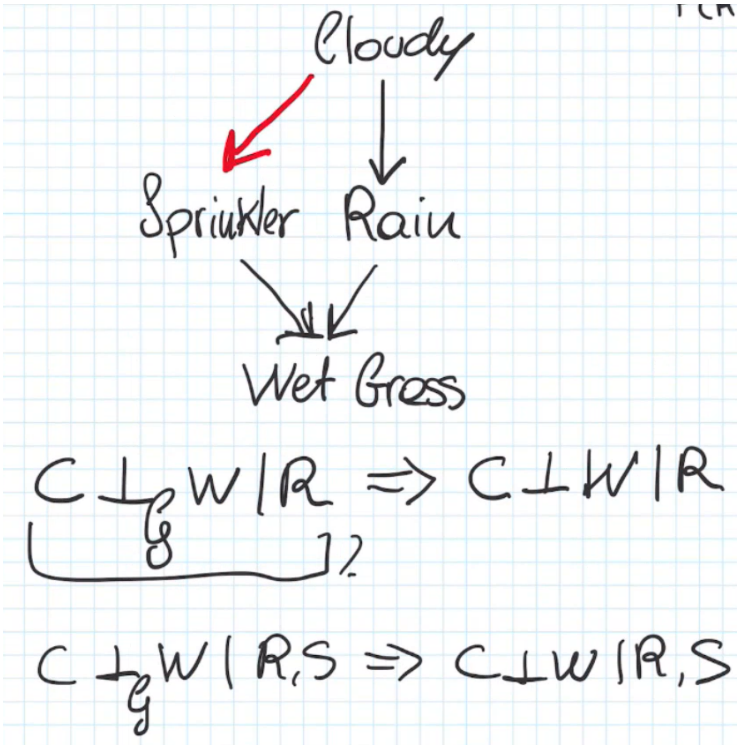


Figure 15:

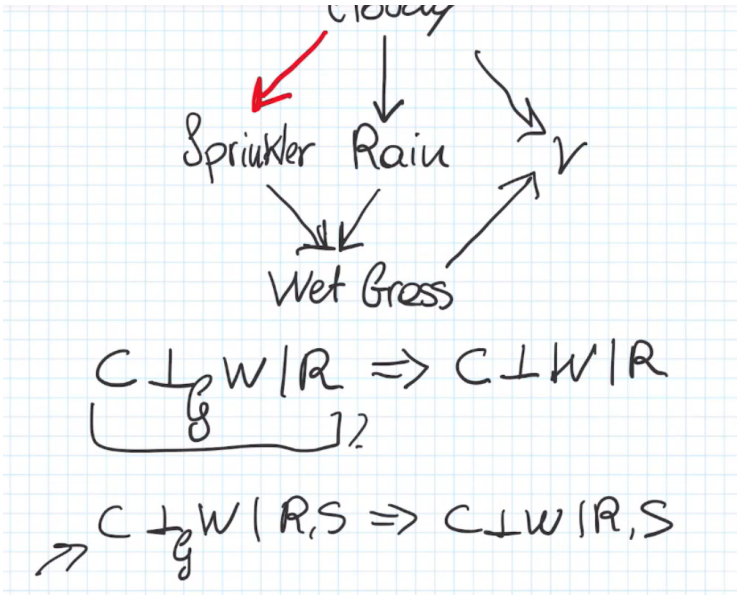


Figure 16:

the other variables.
You need the parents to be d-separated from the ancestors, you need your children to be d-separated from all the descendants, but then you've to condition on the parents of the children because if there is a collider you've to shield again from all the ancestor of the children. The behavior of a node can be completely determined and predicted from the knowledge of its Markov Blanket. The probability of Y given the Markov Blanket and any other set Z, not in the Markov blanket, is gonna be equal to the probability of Y given the Markov Blanket. If you want to predict Y you do not need any other variables apart from the ones in the Markov blanket.

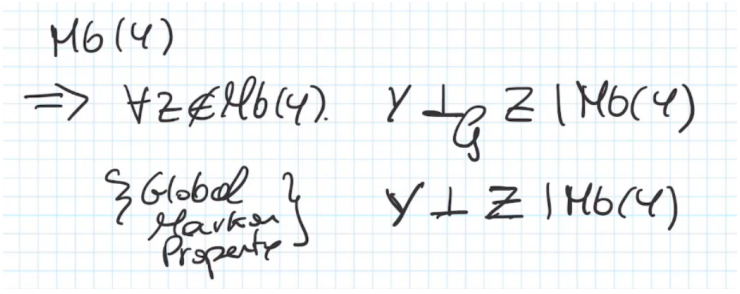


Figure 17: Proof

Am I representing all the possible conditional independence? Until now I've

said, I have a graph that is representing some distribution, I start from the graph, I look at d-separation and I know that some variables are gonna be conditional independent, I'm not telling you that I'm representing all the possible conditionally independences. **Faithfulness Property** is exactly the other direction, the global Markov condition is telling you that you can represent only conditional independences, the faithfulness condition is gonna tell you that you need to represent all of them, each conditional independence needs to correspond d-separation in the graph.

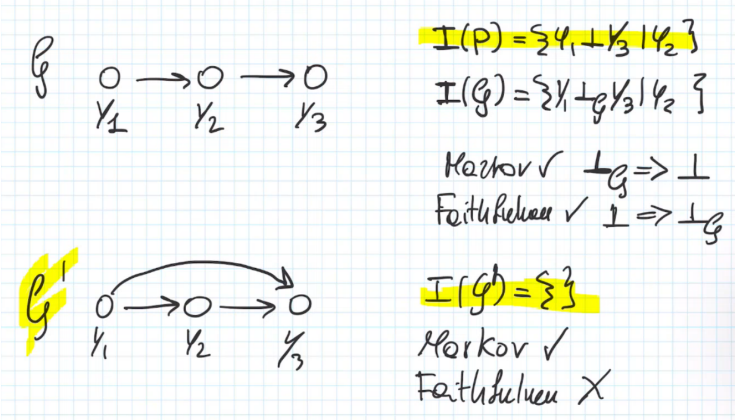


Figure 18:

Faithfulness is fundamental to represent joint distr. in a coincise way, you wanna use few params, only the edges you really need. The more conditional independences you represent the less parameters needed to store the model. The more conditional independences you have the more elems you can remove in the conditioning set. So you want to represent them all
Bayesian network to model *asymmetric dependencies*, children depends on its parents but not viceversa. To model symmetric dependencies? undirected graphs.
Markov Random Fields: we need an equivalent of d-separation for undirected models. d-separation was based on undirected paths and the main problem was the presence of colliders: nasty objects; we have to not condition on to block paths. Now we do not have directions. **Markov Random Fields** is a Bayesian network with undirected edges, nodes represent variables and undirected edges to state Y1 is depending from Y2 and viceversa.
d-separation is still based on node separation but simpler: A and B are conditional independent given C if all the paths from one node in A to one node in B pass through at least one of the nodes in C.
The Markov blanket of a node here includes all and only its neighbors. Given the Markov blanket that variable is conditionally independent (shielding) from all the others, and if you want to reconstruct that variable you only need its

neighbors.
Markov blankets: nodes that are not neighbors are conditionally independent given the neighbors. We want a way to factorize, how to put together nodes to decompose the joint probabilities, by putting together nodes that share the same neighbors, they're gonna have the same Markov Blanket.
Graph structure that only includes nodes that are pairwise connected, i.e. they have the same neighbors? Clique, a subset of nodes in a graph is a clique if you have an edge for every possible pair of nodes. Maximal Clique: if you add another node it stops to be a clique.
A good way to decompose the joint probability is to use maximal clique, X is a set of RV, and we want to define joint probabilities. To do that in an undirected graph, iterates over all the possible maximal cliques, and given the so-called potential function you can represent the joint probability. Xc is the set of all the variables in the clique, the potential function is just a number and to ensure that everything is gonna behave like a probability function, is having a normalization term to have every possible configuration to sum to 1. In the discrete case, the probability of a given configuration X is the product over the potential on all the maximal cliques, divided by the sum over all possible configurations, not only the one you're computing the probability of the potential over all the maximal cliques. This way you've a representation for joint probability of X, whenever you have undirected edges. The problem is computing the partition function, k^N, k distinct values, N distinc variables.

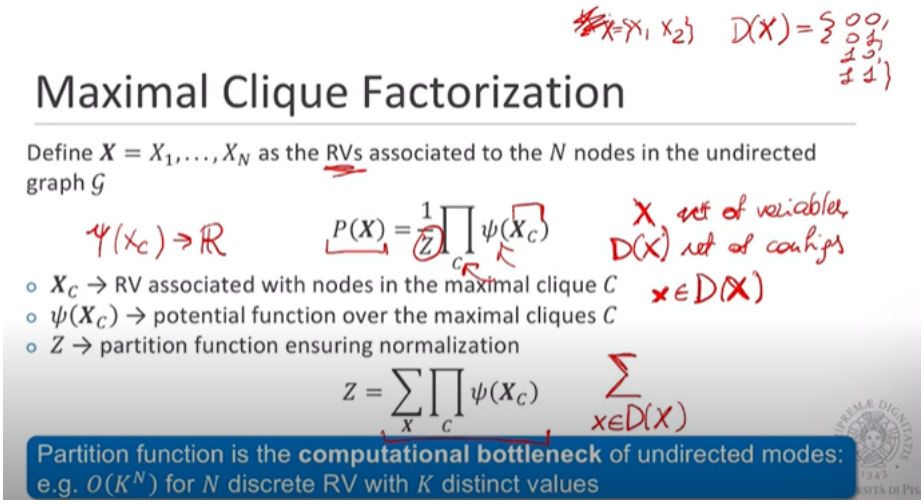


Figure 19:

Undirected models are in a sense a generalization of directed model, you can easily pass from directed to undirected but you cannot do the viceversa. **Moralization algorithm**, you take a Bayesian network and you want to compute

the corresponding undirected version. First, connect with an undirected edges all the nodes that have the same child but do not have already an edge. (moralization: if you have a child than marry). Next step after marriage of the parents is just replace directed edges with undirected ones.

8 (Graphical) Causal Models: representation and Learning

Bayesian networks are used to represent probabilistic info using directed models, this directionality could be interpreted as causal but it is not always that. Distinguish between probabilistic and causal model.
Causation: a random variable is causing another variable if performing a manipulation on the former we alter the distribution of the latter.
Probabilistic models we only observe the world, what is the probability of observing this variable in this particular value if we know that this other variables are in a given configuration. In causal models we can manipulate variables, e.g. healthcare scenario: a vaccine/drug is good or not? You're not just observing you want to predict what happens if you act on the world. Different causal structures can entail the same set of conditional dependencies/independencies. We've just saw it, the set of conditional independences is the same for the chain in the 2 directions and the fork. We cannot distinguish which is the right causal structure.
We know which conditional dependency relations a causal relations entails, Reichenbach's principle (Common Cause): when 2 variables X and Y are statistically dependent, then it holds:

- i) X is indirectly (there is a huge chain of events in the middle) causing Y or
- Y is indirectly causing X or
- There is a possibly unobserved common cause Z that indirectly causes both X and Y.

Generalization of the chain, reversed chain and the fork.
Causal Bayesian Network: a Bayesian network where an edge Y1 $\rightarrow$ Y2 represents not only conditional dependence relations but the fact thah Y1 directly causes Y2.
Y1$\rightarrow$Y2$\rightarrow$Y3 and Y1$\leftarrow$Y2$\leftarrow$Y3, are equivalent Bayesian network they represent the same set of conditional dependencies but distinct causal bayesian networks.
A causal model is actually a collection of different probabilistic models, Py$\rightarrow$i$\rightarrow$PY_i I have a given distribution Py and after i perform an intervation i I 've a new distribution PY_i. For each manipulation I can express a different probability distribution.
Manipulation of a variable: we can change the distribution of the variable, ideal interventions: replace the mechanism of a given variable, in the discrete case we

had the conditional probability table, for each variable we have the table with the probability of a given configuration given the parent. Ideal intervention: we replace a variable Y with a value k. do(Y:=k) we're assigning the value k to Y. P(Ya, Yb, Yc | do(Yb:=k)) used as in conditioning but different, in this way Yb does not depend anymore from Ya, there is no an edge anymore.
If we had another variable Yd entering into Yb, after intervention we cut this edge off.
Assigning the value is something like forcing someone to stop smoking, to take a given drug, the distribution of a variable given conditioning on a given value is different from intervening with the same value.
Fundamental concept : Observing that someone stop smoking is not giving the same distribution on lung cancer than forcing someone to stop smoking. It's a bit counterintuitive.
How to represent the joint distribution of an interviened model? Let V be a set of variables and K a set of values, we want to intervein at the same time on different variables. More in general we can intervein on a set of variables, and assign to each of the variables a different value. Then we can decompose the joint interventional distribution as this: let us start by looking at what happens if V is the empty set: if we are not intervening on anything we're in the observation scenario and this is simply the joint factorization that we saw for Bayesian network, the joint distribution of all the variables is the probability of each variable conditioned on the parents. If I intervene on a given variable: do(Y1:=k) then P(y1 = k', Y2, ...,yk)= 0 when k'!= k. We need a condition that checks if the intervened variables are respecting the intervened values, all the probabilities go to 0 if the values are different from the interviened ones, otherwise We're considering the intervening variables only in the right part of the products, and we are looking for the conditional probabilities for all the others, we do not need to determine the probabilities of the intervened variables but we use to look at the distribution of the childs. Look at the example if you intervene on Y2 you want all the ancestors of Y2 to not depend on the interventions but all the descendants must still depend on Y2.
Average treatment effect: how can I measure if an intervention is beneficial or not, get the effect that we desire or not. You have two specific variables: the treatment variables Y1 (smoking, vaccine, education level) and the outcome (lung cancer, infection, average salary), given Y1 we want to measure the effect on Y2, when you have a binary treatment you measure the expected value of Y2 under a given treatment and the expected value of the same outcome under a different treatment, suppose that you want to maximize Y2, if Y2 is increasing →the treatment is what you expect, 0 treatment no effect on the outcome/unrelated, < 0 negative effect.
Given a binary treatment variable Y1 and an outcome variable Y2, the average treatment effect of Y1 on Y2 is

$$\text{ATE}(Y_1, Y_2) = \mathbb{E}[Y_2 \mid \text{do}(Y_1 := 1)] - \mathbb{E}[Y_2 \mid \text{do}(Y_1 := 0)] \neq \mathbb{E}[Y_2 \mid Y_1 = 1] - \mathbb{E}[Y_2 \mid Y_1 = 0]$$

In general, conditioning on the treatment is different than acting on the treatment. Why the expected value? Fundamental problem of causality, if you

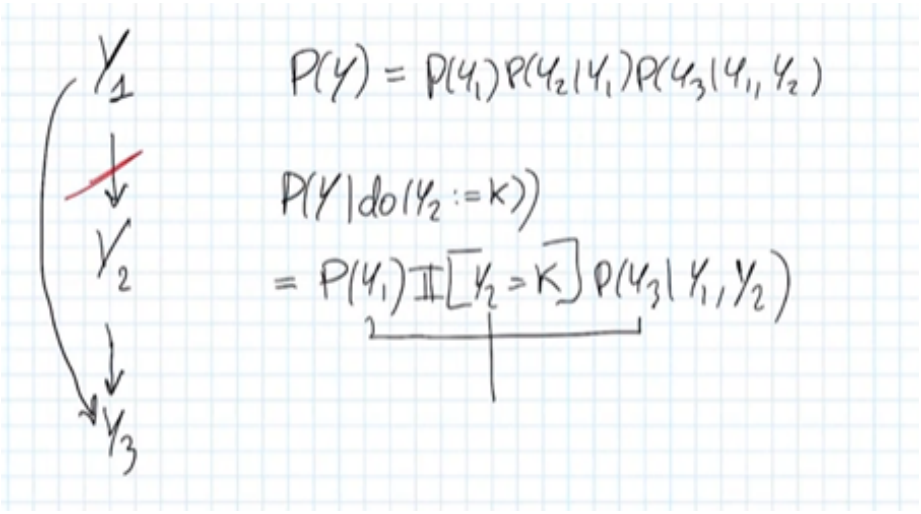


Figure 20: Joint factorization

perform an action, that's it, you cannot perform in the other way, if you select a patient, give a drug to a patient, you cannot observe the counterfactual.
Why we measure using interventions and not conditioning: Observing $\leftrightarrow$ conditioning, we're just looking at the data. Conditional dependence works by undirected paths, we've an undirected path from arm pain to infection but causation works by directed paths, there is no a directed path from arm pain to infection, intervening on arm pain is not gonna have any effect on that particular variable.
Does observing a vaccine increase the probability of observing an infection? Counterintuitive but it's actually true (Sympson's paradox) $\rightarrow$ looking at conditional probabilities. Does taking a vaccine increase the probability of being infected? To answer this we need to compute the probability of getting an infection after an intervention.

Causal effect identifiability: the problem is how do I compute the probability of the outcome given the treatment: main intuition $\rightarrow$ there's must be another set of variables Z, such that if I condition both on Z and Y1 is gonna be equal to intervening on Y1. P(Y2|Y1)!=P(Y2|do(Y1))=P(Y2|Y1,Z)
How do we find an adjustement set Z? **do-calculus**, system able to find adjustments but it's quite convoluted but there are 2 scenarios we can prove using do-calculus we're gonna see, common in real world application.
Back door criterion to handle observable confounders, and front door adjustment to handle latent confounders.
Back-Door Adjustment: suppose we want to compute the causal effect e.g. of var Y6 to var. Y8, we say that a set of variables Z satisfies the back-door criterion for the causal effect of Y6 on Y8 if: no node in Z is descendant from

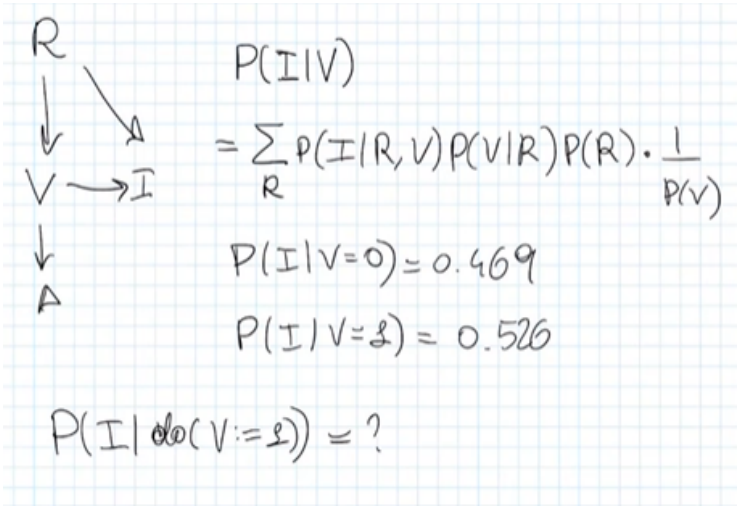


Figure 21: Factorization: $P(I,V) = \sum_R P(I,V,R) = \sum_R P(I \mid V,R)P(V \mid R)P(R)$;
$P(I \mid V) = \frac{P(I,V)}{P(V)} = \sum_R P(I \mid V,R)P(V \mid R)P(R) \cdot \frac{1}{P(V)}$

the treatment Y6 and the conditioning set Z blocks all paths from the treatment Y6 to the outcome Y8 that contain an edge entering on the treatment Y6. Remember we use the notation of undirected paths. Given this conditioning set you can estimate the probability of Y8 given the intervention on Y6 by looking at the possible configuration of the adjustment set

$$P(Y_8 \mid \text{do}(Y_6)) = \sum_z P(Y_8 \mid Y_6, Z = z)\, P(Z = z)$$

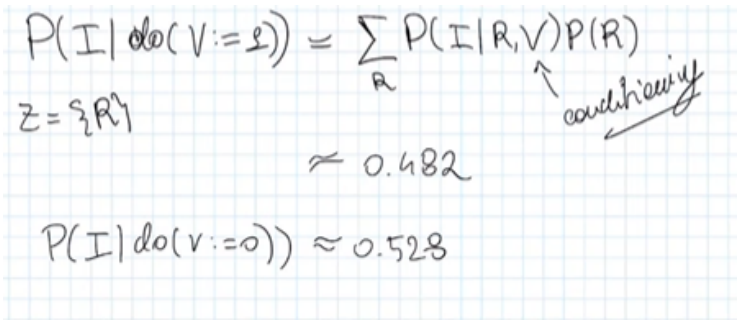


Figure 22:

The problem with BD adjustment is that requires to condition on a confounder, to do so that variables must be observable, but variables are not observable

all the time, and this is when the Front-Door adjustment comes into play: in the FD adjustment we assume that the confounder is not observable (latent). **Front-Door adjustment**: the adjustment set must intercept all the directed paths from the treatment to the outcome, there must not be an unblocked back-door path from Y1 to Z, if you had for instance another variable Y5 that causes both Y1 and Y2, you cannot use the front-Door adj. anymore, cause you're introducing some confounding between the intermediate variables, you want the intermediate variable to depend only on the treatment, the connection (back-door path) between the mediator Z and the outcome must be blocked by the treatment.

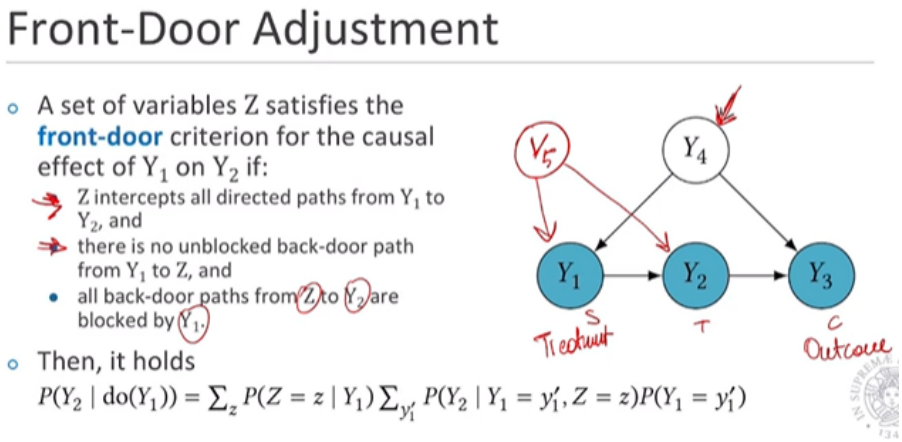


Figure 23:

Whenever you have observable confounders and you can condition on them, and you want to estimate the causal effect, you use the BD adjustment, if you have latent confounding, and you're lucky enough to have some sort of mediator which is also not confounded you can use the FE adj. for all the rest do-calculus. **Counterfactual queries** we reason on what would be an alternative outcome, we cannot observe counterfactual, but can we estimate a counterfactual?? If the patients that took the vaccine had received the placebo, would they recover? Causal Bayesian Networks cannot answer counterfactual queries, similarly to how the same distribution can be represented by different Bayesian network, the same counterfactual can be represented by different causal Bayesian network.
We need a different model **Structural Causal Models** alternative way to represent causal relations. In Bayesian networks you define relations between variables using conditional probability table or in general an explicit rep. of a probability of a variable given its parents (P(Y |Pa(Y). In a structural causal model you define the mechanism between variables as deterministic functions, you had some sort of exogenus variables, that you assume that are always unobserved, to model the uncertainty and then you compute the value of the

endogenus value, using a function.

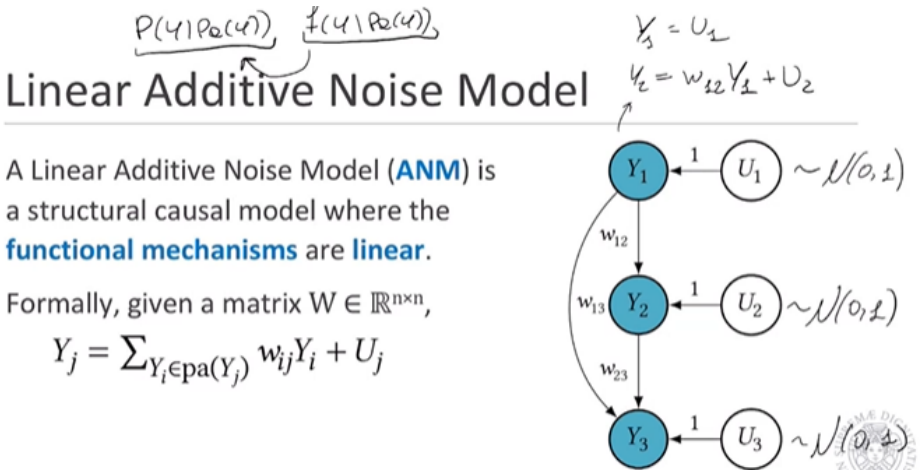


Figure 24:

U1 $\sim \mathcal{N}(0,1)$ and Y1 = U1 then Y1 $\sim$ N(0,1)
U2 $\sim$ N(0,1) and Y2 = w_{12} Y1+U2 then Y2 $\sim$ N(0,2)
The idea is that you can pass from the deterministic mechanism to the probability distribution.
Now you're able to compute counterfactuals. Suppose that you observe some variables Y and you ask yourself what would happen if I perform an intervention on Y1, you can simply reverse the distribution from the observations to exogenus, P(U|Y) what is the most likely configuration of the exogenus, which you do not observe given the observation, then you perform the action and you perform the prediction using the updated exogenus distribution. Abduction step means recover the latent exogenus variables, then you perform the action, the intervention. Suppose you are interest in the causal effect of Y2 on Y3, so you perform an intervention, to do this you remove the dependence of Y2 on Y1. And finally you can predict the probability of Y given the intervention and the new exogenus term.

9 Causal Models: Representation and Learning

How to learn structure from data? You need probabilistic models to perform predictions and you need causal models to perform predictions on the actions/manipulations. How to actually learn from data this structures. Two different kinds of problems:

- Parameter Learning: we assume to have the structure, enough knowledge from a domain expert and you already know how the graph is considered,

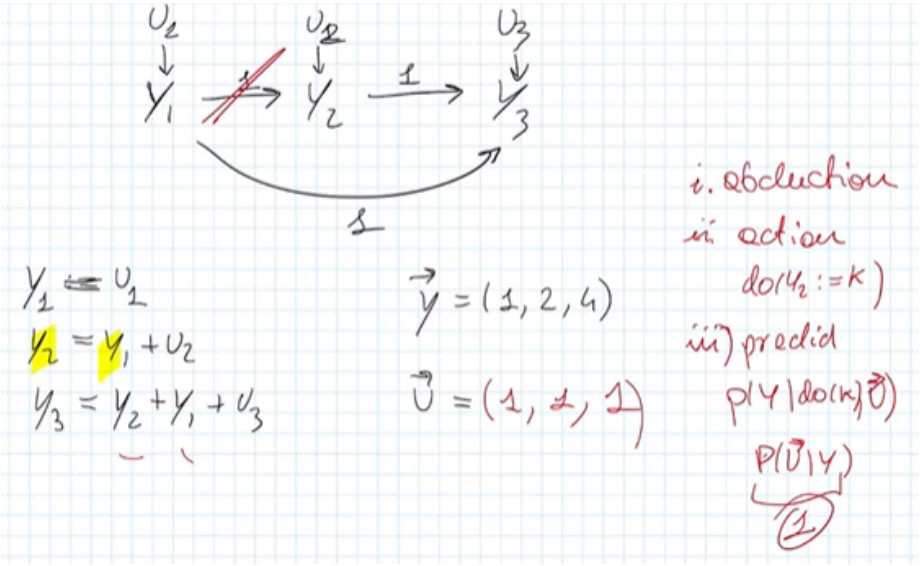


Figure 25:

what's causing what, in a causal model, or the conditional independence relations in a probabilistic model. The task is simply to get the parameters of the model. So you have your network, there is an edge from X to Y and you know the distributions of Y given X, P(Y|X) if you're training a probabilistic model, or if you are in a structural causal model the function of X that render Y. Best scenario you have the structure and all the variables, if the data is incomplete, more complex, you've also to consider latent variables.

- structure learning on the other hand adds another degree of complexity, you do not know which is the structure of the graph, you've to solve both problems: first you have to discover the structure of the graph and then you've to fit the parameters.

Recovering causal graph for a causal model or a Bayesian network for probabilistic model by simply using data that we assume to be complete.
We assume to have a dataset composed of observations of a fixed number of random variables. Understand which are the arcs that exist in the network: the edges determine independence relations in a probabilistic model or a causal relation in a causal model. First thing, we've to determine whether the edge exists or not. Second problem is to direct the edge. Then you have to learn the params, and this is a difference for Bayesian Network and SCM, in Bayesian network you assume there is some sort of distribution, you specify the family of the distribution and you want to learn the params of that family, in a SCM you have to learn the function.

3 main strategies: first **CONSTRAINT BASED STRATEGY** where we use the conditional independence testing to constrain the network. We know some kind of conditional independence implies a given structure, with Bayesian network we say that the fork, the chain and the reversed chain have the same conditional independence set. We will use this knowledge to remove some edges from the graph.
SEARCH AND SCORE STRATEGY: another approach where we start from a graph and we explore the space of all the possible graphs and assign a score to each graph, We return the best score graph.
PARAMETRIC IDENTIFIABILITY: which SCM can be identified from data and which cannot.
Markov Equivalence class: is a set of DAG that encodes the same set of conditional independences. Two dags are Markov equivalent $\leftrightarrow$ have the same skeleton and the same set of colliders (v-structure). The fork, the chain and the reversed chain are in the same Markov equivalence class because they entail the same set of conditional independence. The skeleton of a directed graph is the undirected graph obtained by removing the arrow so they have the same skeleton and they have the same (empty) set of V-structure. We can represent a Markov equivalence class by using this sort of mixed graph with both directed and undirected edges. You can represent a Markov equivalence class with the skeleton and the V-structure. (CPDAG- CompletePartial DAG).
CONSTRAINT BASED METHODS require some assumptions: **faithfulness** $\rightarrow$ all conditional independences are represented in the graph. **causal sufficiency**: a model is causal sufficient if all the confounders are observed and there is no selection Bias. Why do we need this? Suppose that you have a collider, and Y1 is independent from Y2 but they are not anymore independent once you condition on the collider. Suppose that you have some form of selection bias where you're implicitly conditioning on Y3, you're observing that Y1 and Y2 are dependent when they actually are independent.

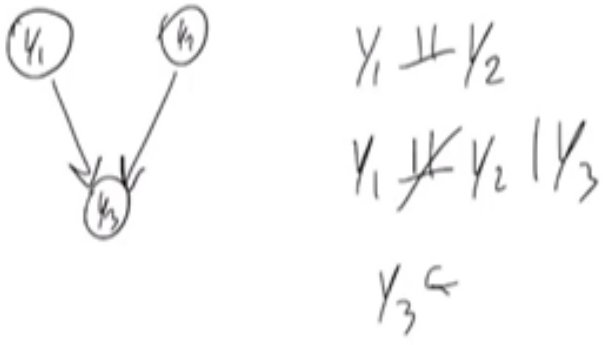


Figure 26:

The same scenario might happen if you have latent confounders. You may have a Y3 that is not observed, you might get a dependence between Y1 and Y2 and you do not know if there is a latent confounders or if there is an edge between the two.

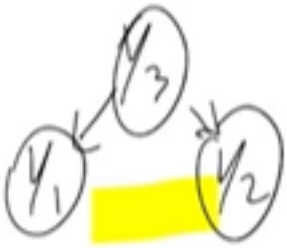


Figure 27:

Let's take these two assumptions and let's try to build a strategy to recover the graph using conditional independence testing: several strategies to perform conditional independence testing, let's assume to have an oracle.
SGS algorithm and **PC algorithm** are the most common algorithms used for structure learning. The idea is that we want to recover the Markov equivalence class, and we hope that, if we want to recover a causal graph, that the size of the Markov equivalence class is small, cause in the reality you have some variables and only one causal graph to describe them. With this method you can retrieve a Markov Equivalence class but if you want a causal model you have to choose one inside the class and you need more knowledge. // **SGS algorithm**: the first part of the algorithm determines the skeleton, you want to understand whether there is an edge between two variables or not. The skeleton is going to be the undirected graph, two variables are adjacent in the skeleton, so there is an edge between them, if they are always conditionally dependent, so there is no separating set without X and Y.
Is there a separating set between X and Y? Yes, it is a fork if you condition on Z, the center of the fork, they become independent. But if there was an edge between X and Y you cannot really separate X and Y, without conditioning on one of them.
First we create a CPDAG where everything is connected with everything and then we take one pair of node (X,Z) and we ask ourselves, is there an edge between X and Z? And, to determine that, we look for possible sets on which we can try to separate X and Z? We have only three variables, if we took apart X and Z we have only the empty set and the set with only Y to condition. We do not see the ground truth, but produce the dataset and we can answer some queries on the dataset through conditional independence testing, is X independent from Z marginally? In the example we know the ground truth, we can answer with d-separation, is X independent from Z? no , cause they're connected, is X independent of Z given Y? NO! There is not a conditioning set that

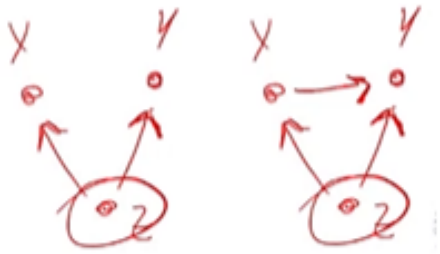


Figure 28:

separates X and Z so there must be an edge between X and Z.

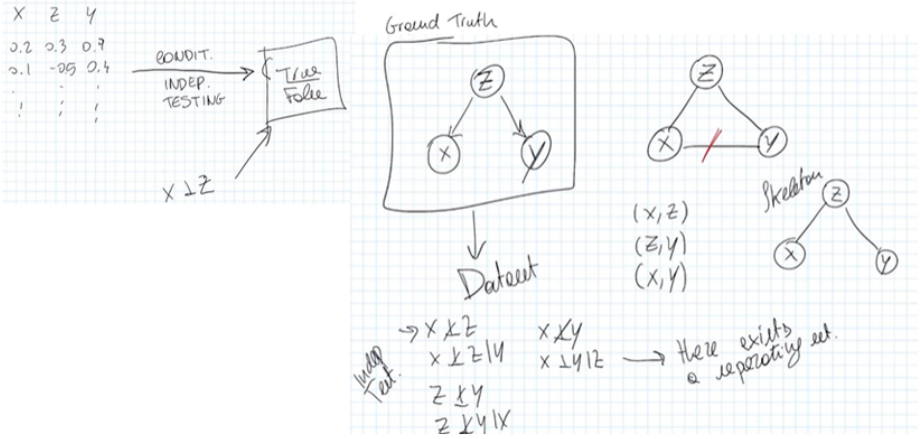


Figure 29:

Suppose that you have your skeleton, if two variables X and Y are not adjacent in the skeleton but they are connected to a third variable W and W is not a member of any separating set we found a collider. Why? if X was Independent of Y given W, there we know that we were in a situation where we have either a fork, or a reversed fork or a confounder, but if X is dependent of Y given W then the only possible solution is the collider, so we can start orienting something.
Conditioning on the collider opens the path, so they're dependent now. Once you have found all the v-structures, you have a representation of the Markov Equivalence class. (skeleton + v-structure). You cannot orient the remaining edges randomly, you have a risk of introducing cycles.
Rules for entering additional orientations. This not change the Markov equivalence class, because it is just constraining some edges that can be only oriented in a particular way. First is due to the fact that otherwise you create a v-

Figure 30:

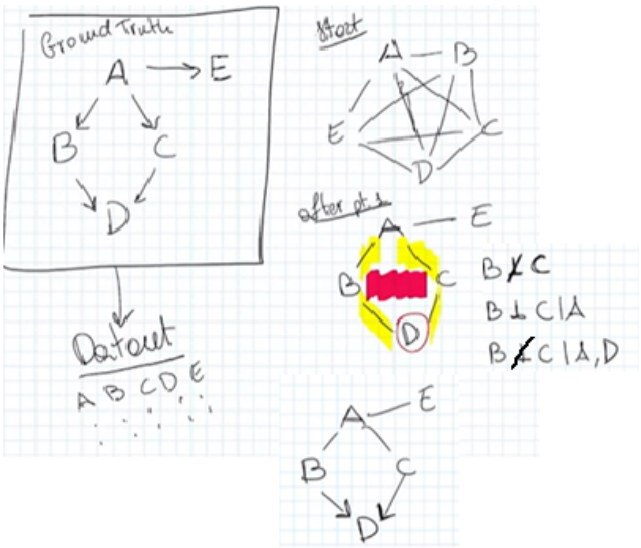


Figure 31:

structure but you found all the v-structure in the previous step. The second one is to avoid introduction of cycles.

Figure 32:

Meek rules tells you how to avoid to indirectly introduce new v-structure. Fundamental for SGS algorithm and PC algorithm, all the collider have to be found in the previous pace. You can derive in the similar way for rule R4.
These are all rules that ensure that you are mantaining the Markov equivalence class, so same skeleton and same set of v-structure. They still don't guarantee a DAG, the Markov equivalence class has the same size but you'll have less edge to orient.
The choice of the testing order is crucial to avoid superexponential complexity

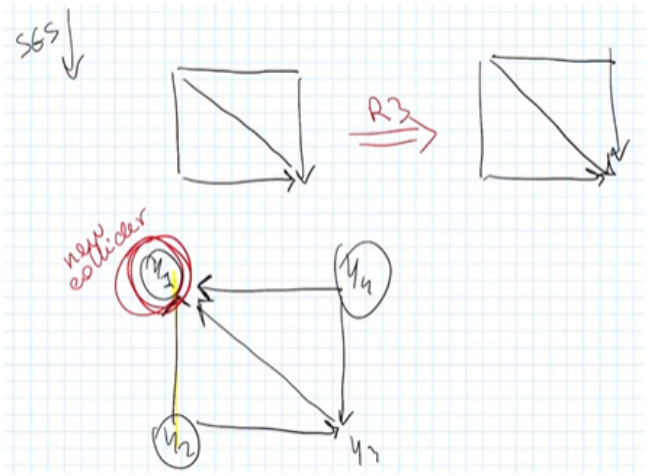


Figure 33:

because if you have to test all the possible pairs with all the possible separating sets it's not feasible. The main strategy is the one from the PC algorithm that proposes to test nodes on conditioning sets of increasing size, the intuition here is that you start by looking for instance which variables are marginally independent, then you look for variables that are separated by only one, only two and so on. The worst case is gonna be the same as SGS, you still have to look to all the pairs, but on average it works better computationally speaking.
The only difference of the PC algortithm w.r.t. the SGS algorithm is in the definition of the skeleton. Euristhic: you start from k equal to 0 and you select pairs of nodes that are connected in the graph, so you still start with an undirected graph where everything is connected, but then you look at conditioning set that has size K and are in the adjacent set of X. When you start you still have to look into all possible subsets because the adjacency contains everything, but once you prune you start reducing the size of the adjacency, so you look to a smaller number of subsets. The more you go with the algorithm, the more you reduce edges, the more you increase the size K but you 'll end up in scenario where k is larger than the number of possible adjacency. An then you perform conditional independence testing and you prune edges as you did in the SGS, then you increase the maximum size of the conditioning set. You can procede in looking the v-structure and orienting edges as before.
Is the CPDAG produced by constraint-based methods enough? If you just want to perform probabilistic queries ok just take one, they're equivalent in terms of conditional probabilities if your goal is to build a Bayesian Network you can just orient stuff without introducing cycle or new v-structure (use Meek rules), you can take one, learn the params and perform inference. But if you want to perform interventional queries or counterfactual queries you need further knowledge to orient the edges (domain expert or perform experiments) otherwise no.

Anyway given the graph, so any graph for a Bayesian network or the magical graph you get from an oracle for the causal Network, you need to choose the distribution families, whether a continuous variable is: exponential/ gaussian whatever, or the mechanism for a SCM linear, non linear is a NN and you have to learn the params. You're gonna have some dataset from which you build your graph, from the graph and the dataset you get the params.
How to get directly a graph instead of a Markov Equivalence class: search and score strategy, we're gonna explore all the space of the DAG and gonna assign a score to each structure. We would like that the graphs very different from the ground truth have a really low score and a graph similar to the ground truth has a nice score, and the exact graph has the best score ever. We want a function that given the data, takes a graph and return a number that quantifies how good is that graph given the data.
Scoring function: how do we define the score and how do we search the space of DAG, cannot enumerate all DAG we need some heuristics.
When defining a scoring function that must respect 2 property: consistency: graphs in the same Markov equivalence class must give the same score. Decomposability: computing the score on the whole graph might be difficult and could be useful to compute the score on the nodes and have the score of the graph as an aggregation function of the score of each node. The intuition here is that you want to compute the probability of observing the data given a particular structure, and you get it by marginalizing over the variables and the possible datasets and computing the probability of observing that variable given the parents in the graph you're testing. The graph is telling you: okay if this is the ground truth graph this is gonna be the set of parents, different parents will give different parental set, and you want to find the graph that maximize the likelihood of the data given that graph regularized by the prior info on the graph itself.
The second part is on how we explore the space of DAG, start with a given graph and try to explore the alternatives by adding and removing edges. Instead of looking to completely unrelated graph you look to similar graph, e.g. you add an edge, or you remove an edge.
Assuming to know the topological order of the graph can help you to constraint the search space. And you can combine search and score with constraint based strategy. You can start by selecting the skeleton even if it is not the complete skeleton of the model, if you remember in SGS and PC you start with a graph with everything is connected to everything, and the more you go on the more you remove edges, ideally you want to get in the situation where you remove all the edges that are not there, you could stop a bit earlier with some edges that are already there and use it as initialization for a search and score strategy, starting from that you have to refine the skeleton and orient edges and you can do that with a search a score strategy by adding or removing edges.
Linear Additive Noise Model: the idea in this model is instead of defining probability distribution of the variables we're defining a functional mechanism. We need SCM from counterfactual queries.
Here an instance of a SCM where the connections are linear. Y2 is gonna be a

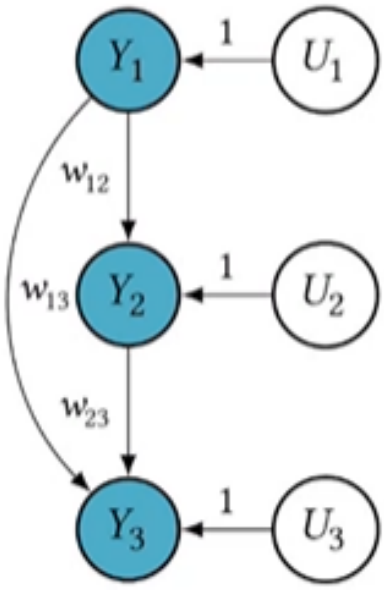


Figure 34:

deterministic function of is parent Y1 added to the noise term U2. Y3 is gonna be a function of Y2 and Y1 plus the noise term U3. In a Bayesian Network or in a Causal BN you define the probability/distribution of a variable given its parents: P(Yj|Pa(Yj) while in a SCM you define the function that reconstruct the value of a node given the parents.
How can we learn this models from data? The problem is that the identifiability from data of linear Additive Noide model and in general of SCM depends on the distribution of the noise terms. The exogenus variables assumed to be latent and unobservable, these variables correspond to the noise term of the model P(U) is some probability/distribution to represent the likelihood to get a given noise value. Depending how this object is distributed you have the identifiability results: i.e. the possibility of recovering the ground truth graph. For instance if the noise terms are Gaussian and all have the same variance: $P(U_1) = \mathcal{N}(0, \sigma^2)$ and $P(U_2) = \mathcal{N}(0, \sigma^2)$, if the model is linear you are able to recover from data whether it is Y1 causing Y2 or viceversa. This is not so common in practice. Suppose that you simply rescale your dataset you change the empirical variance of the variance you observe so you lose this property. Not robust enough to go through a data normalization does not seem a good choice. If you do not have equal variances you cannot anymore identify the model. From data you cannot distinguish in a linear additivive Noise model if it is Y1 causing Y2 or viceversa. You can recover it if you assume that all the noise term are non-Gaussian then you're able to recover the graph.
What are the identifiability results not only for linear ANMs but for more gen-

eral structure, the bad news: for general SCM we cannot always identify the graph, in constraint based method the most we can do is to recover the Markov Equivalence class, by DAG identifiability it means that we try to recover an unique object the ground truth graph from data. If you have an additive noise model NON LINEAR, the function f is non linear you can recover the graph from data without further info. CAM Structure: the contribution of each variable to Y is not linear but then you're just summing them so you have linearity in the aggregation function, the linear gaussian is not identifiable unless equal variance or non gaussian variance.

10 Hidden Markov Model

How Bayesian Network become less static, we can design Bayesian network whose structure is not fixed once and for all, but can change according with the nature of the problem we're tackling.
Sequential data: info with a temporal behavior, you have to take into account that data are not i.i.d. in your ML model. I cannot interpret a single observation in the T.S. without considering the past, info are not independent from one another.
In a probabilistic model we map this assumption in a conditional dependence assumption.
Sequence is more general than Time series: it does not assume that time is orienting the direction of compound information. e.g. character sequence: each elem takes value in the alphabet. Model a world of strings. The probability of observing a letter 'a' at time t is not 1/25 (the i.i.d probability, a character at random), but this probs. depends on the history of the characters seen so far. The probability of observing a specific character at a specific point in a structured language is not i.i.d. but depends on the context.
P(Y_t = 'a' | y1,...,yt-1) the complexity is $25^{|seq.length|}$.
We need some simplifying assumption: put a-priori knowledge into the model, these dependencies are to much, I need simplest thing that takes into consideration dependencies but has least complexity: use a single conditioning variable: the preceding element.
P(yt|yt-1) for t=1...T P(y2|y1) P(y3|y2) are these 2 prob., identical given the same assignement of letters. For us yes, but only under the stationarity assumption, our dependency model is stationary/constant w.r.t. to time, the params are gonna be reused across time. stationary w.r.t. to the position of a seq. Very useful because you can reuse the same model across a sequence, if i have two seq. i can reuse the same model with the same params even if the seqs. are of different lengths. Same assumptions as in RNN. Weights sharing, stationarity of state transition functions
seq. made of observations in time: realization of a random variable, nodes in a Bayesian network. But they are still floating in space: how do I connect them? Draw arrows that go from left to right, stationarity property → replicate

the arrow for all the elems → Markov Chain: take a seq. make a simplifying assumption on the structure of dependencies, I depend only by the precedent, stationarity to replicate the same prob across all the time istances → firs order markov chain. All of them are observable. Now i have a Bayesian network and i could try to use it to factorize a joint distribution, i.e. the prob of all data points in the seq. → apply chain rule of calculus. → simplify using the assumption, a point is completely determined by the precedent, if I observe/conditioning on Y2 the path from Y1 to Y3 is blocked, no flow of info/exchange, whatever Y1 does, no info flow because I'm observing Y2, I can get rid of Y1.
I obtain a factorization of the joint distr. as the product between each random variable involved, given the parents of the random variable. The predecessor in this case.
A joint distribution over seq. can be reduced to the product of two distribution. P(y1) (π): prior probability of observing a specific letter of the alphabet in the initial pos. of the seq. π is a vector with the probability of Y1 to be equal to each letter, sum to 1, same size as the alphabet. P(yt|yt-1) = is the prob. of observing a specific letter if previously i've observed another specific letter: 25x25. This probability look like a 25x25 matrix.
It's a conditional probability, not a single but a family of distribution, a different prob. for each different choice of the conditioning variables/preceding letter. The elements of each columns sum to 1.
At some points this cells will become params, when we train Markov model we train an hidden markov model, we are training the numbers in this tables, we will take seq of sentences from text and we will try to induce the values of this matrices and vectors from the observations in my TR set.
Perform inference of a single point in time. By marginalizing, we need marginalization because we do not now how to solve the problem, but if we introduce some additional info, then I can solve the problem. Marginalize w.r.t. to all possible values of yt-1. Apply the same thing and marginalize w.r.t. to Yt-2. Unfold again until you arrive To Y1 you have the probability cause it's the prior.

$$P(y_t = a) = \sum_{y_{t-1} \in \{a_1, \ldots, a_z\}} P(y_t = a, y_{t-1})$$

$$P(y_t = a, y_{t-1}) = P(y_t = a \mid y_{t-1}) P(y_{t-1})$$

$$P(y_{t-1}) = \sum_{y_{t-2}} P(y_{t-1} \mid y_{t-2}) P(y_{t-2})$$

$$P(y_t) = \sum_{y_1, \ldots, y_{t-2}} \left(\prod_{s=2}^{t} P(y_s \mid y_{s-1}) \right) P(y_1)$$

$$P(y_t) = A^{t-1} P(y_1)$$

$P(y_t) = A^{t-1}P(y1)$: restricted form of random walks requires invertibility, if you think about to be in a graph p(y1) is the probability to wake up in any

node of the graph, A is telling you the probability given you're in a node of the graph of jumping in one of your neighbors. P(yt) is the probability to ends up in any node of the graph after t jumps, starting from any node in the graph, mediated by A, state transition distribution, probability of transiting from one letter to another.
First Order Markov Chain: is a Markov chain in which i'm assuming that the conditional probability that regulates everything only depends by the predecessor, first order refers to the size of the conditioning set. I can make a second order markov chain by making t+1 dependent not just on t but also on t-1, transition probability.
P(Xt+1|Xt, Xt-1): second order, A now is a 3D matrix, the higher the order of the Markov Model, the more tensorial distribution becomes. Higher order of the Markov Model make it a more complex model for the perspective of the param. More expressive but not only more params, (tendent to overfitting), also more multiplication, no free lunch.
How can we use this model? Model the relations between characters, how much is likely to observe an alphabet letter after having seen other alphabet letter? I can use to model the dependency between words, each word is one element of a vocabulary. Model the fact that there is a direct dependency between the verb is and the word cat. How do we model formal grammar in computer science? Or the word 'the' is an article, 'cat' and 'table' are nouns and nouns are preceded by article, here emerges the dependency P(yt|yt-1).Rather than words model Depends the model between part of the speech and given the part of the speech decide which word they actually are instantiating →NECESSITY OF HIDDEN MARKOV MODEL: model where the structural dependencies are between part of the speech, dependency is between article and noun and between noun and verb and once you fix the noun, you sample a specific noun like dog, cat or whatever. We assume that while words can be observed, categories like article, noun and verb are not, there will be random variables that are not observed (latent, hidden).
I do not know apriori which S2 will emit a cat, at some point by observing my data, I ll learn that some S2 will need to be used to represent cat and certain S2 will be related closed to some S3 to model the part of speech dependency and this is gonna be inferred from data. We're postulating that exists a certain structure but params not fixed, the params are those, when we're gonna be learning the params, those params will tell us what is the association between article and specific words, between the nouns and specific words and which is the structure of the grammar. In the HMM setting we disentagle the transition dynamics from the observations.
The fact that St in $\{1...C\}$ is not observable, means that I'm not gonna be knowing a priori the association of a specific elem of this alphabet to a specific time, but i will have to infer by manipulating my probability. In Markov Model terminology St are hidden states. Hidden because i cannot measure with on my data which is the actual i associated with a specific st, states because i'm assuming they are the generator of the data I'm observing. 3 bins, the first 1 green balls, the second red ball and the third blue ball. Imagine that a certain

point somebody starts pulling out from behind a curtain a red ball followed, by a green ball , RGBRGB you're observing the ball that you can see are the observable, what you cannot observe is that you're observing this sequence of colored balls because the person behind the curtain is taking first from red, then transit to blue, then to green etc etc. There is an hidden process you cannot observe, but if you assume that there exists 3 bins and those 3 bins have the probability of giving out a certain proportion of colored balls that is exacts your HMM, the bins are the state, there is a certain probability to move from one bin to another one, so to transition from a state to another one, and once you're in a state there is a certain probability of pulling out a colored ball, an observation. If you are in a specific state say the blue bin you always transit in a green bin, equivalent if there was a 1 probability of going from blue to green and 0 of going from blue to red, infact i can say that there is a probability of 0.9 to go from B to G and a residual probability of 0.1 from B to R. I can say also that there is a certain probability to stay in the same state. Instead of taking out only blue balls with probability 1, from the blue bin i take out blue ball with probability 0.7, G with 0.2 and R with probability 0.1.
3 probabilities now: the usual prior probability (π), the usual transition probability A, and the probability that returns balls, B the emission probability, the probability if you are in a specific state to observe a specific balls of a given color.
Under what conditions are yt and yt-1 independent, if nothing is observed, are the paths between yt-1 and yt blocked?? There is a single path between them: y_t-1 $\rightarrow$ s_t-1 $\rightarrow$ s_t $\rightarrow$ y_t that's a chain path, this path is locked when at least one variable on the path is observed, s_t or s_t-1 are not observed now so the conditional indipendence between y_t and y_t-1 is not supported by the bayesian network. What the bayesian network support instead is that y_t is conditional indipendent of y_t-1 given s_t. No information transfer between yt-1 and yt, observations are not marginally independent they will become independent if i can observe the hidden states: Can I observe the hidden states? Not from the dataset, I will introduce them by marginalization, as soon as I introduce them by marginalization the marginal dependency existing between the observation will vanish because I can now introduce the conditioning on the hidden states. And the joint distribution of the observations will factorize into a simpler distribution.
The first ugly summation comes from the fact i have to introduce all the latent variables for each time in order to be able to factorize my joint distribution, as soon I've introduced them i can apply my conditional probabilities in my model and break down that huge ugly joint distribution into a product of simpler distributions, a prior distribution, an emission distribution, and the product across time of the transition distribution and the emission distribution.

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

$$P(\boldsymbol{Y} = \boldsymbol{y}) = \sum_{s_1,\ldots,s_T} P(S_1 = s_1)P(Y_1 = y_1 \mid S_1 = s_1) \prod_{t=2}^{T} P(S_t = s_t \mid S_{t-1} = s_{t-1})P(Y_t = y_t \mid S_t = s_t)$$

(π) is a vector with the probability of values between 1 and C.
A is the usual matrix CxC, b is the emission distribution that relates all possible C states with all possible output/observations. If my observations are letter of the alphabet B becomes a 25 X C matrix, given a state I can emit the 25 letter of the alphabet.
You can make this model more general, the output labels can be continuous gaussian observations, B is not anymore a table but a gaussian, how do I learn a Gaussian: find mean and the std of the gaussian, the mean of a gaussian is a vector, the std is tipically a matrix with just a diagonal.
HMM model for sequential data, Yi are the observation in time, but rather than expressing directly the dependencies between observations in time in HMM we asssume that exists an hidden unobservable process, that has a time dependency St, on a specific St the observation is generated independently, the time dependency is projected onto the latent variable St and the observation became independent because of my assumption. The direct relations between observables may be very articulated, even NN do not model directly observations/the relationships between features, they use hidden neurons. Latent state $\leftrightarrow$ hidden unit. The only distribution that we compute is over the observables, but we need the hidden states otherwise the observable remain dependent one another, we introduce the latent variables through Marginalization, introducing all the random variables one for each time step. Then apply the conditional independence relationship described by that graphical model Bayesian network, we rewrite that joint distribution as a product of simpler distribution. Prior distribution, apriori in which state we're in the first item of my sequence, my emission distribution b which occurrs in two difference place, probability a specific observable at time t given I'm in a specific state in time t and the transition distribution A that is the prob to move from a specific state a time t-1 to another state at time t. Pi, A and B are model params. If your observation are discrete, b is a table if it is continuous is a gaussian whose params are its mean and its variance.
Hidden Markov models are not so incredibly different from NN, see Recursive Model repr, they're different in how you implement/parametrize them but not different in the assumption they make about you process to recursion. NN represents an input seq. in a continuous space (the activation of hidden neurons), HMM represents seq. in a discrete space, the space of the values that St can take, they are discrete.
HMM has an interpretation as automata/state machines.
see the examples of the automata with 2 states St can take values in 1,2 we find again al the params pi, A, b . Special class of automata weighted automata. Transducer generalization of an automata, they process an input sequence and output an output sequence. HMM can be extended to be transducer.
Inferential problems in HMM: **smoothing**: computing the probability to be

in a specific state at time t having observed the full seq. Posterior estimation problem cause I'm assessing the probability of something that is unknown using something that I can observe, data (a-posteriori re-evalution of my belief about St). $P(S_t \mid \mathbf{Y} = \mathbf{y}, \theta)$
Learning: given a dataset and a set of hidden States C, can i find the parameters π, A and B. In the smoothing problem θ the params are in the conditioning side, we assume them to be fixed, in the second problem, learning problem, i will be estimating thetas.
Optimal state assignment: looking for the optimal assignment of each hidden state for each time step given a sequence. Optimality depends on how you formulate. Will be formulated as the joint of the observations and the state. To solve 1 and 3 we use sum product and max product algorithms.
Forward-Backward Algorithm solves the smoothing problem and then parameters estimation.
Forward-Backward Algorithm: Smoothing We are interested in computing the smoothed posterior:

$$P(S_t = i \mid \boldsymbol{Y} = \boldsymbol{y})$$

Using Bayes' rule, we exploit the factorization:

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

Decompose the observation sequence into past and future and application of the chain rule of probability:

$$P(S_t = i, \boldsymbol{y}) = P(S_t = i, \boldsymbol{Y}_{1:t}, \boldsymbol{Y}_{t+1:T}) = P(S_t = i, \boldsymbol{Y}_{1:t}) \cdot P(\boldsymbol{Y}_{t+1:T} \mid S_t = i)$$

** We now apply the **chain rule of probability** to decompose this joint probability:

$$P(S_t = i, \boldsymbol{Y}_{1:t}, \boldsymbol{Y}_{t+1:T}) = P(S_t = i, \boldsymbol{Y}_{1:t}) \cdot P(\boldsymbol{Y}_{t+1:T} \mid S_t = i, \boldsymbol{Y}_{1:t})$$

At this point, we apply the **Markov property** of the hidden Markov model (HMM), which states that the future observations $\boldsymbol{Y}_{t+1:T}$ are conditionally independent of the past observations $\boldsymbol{Y}_{1:t}$, given the current hidden state S_t. That is:

$$P(\boldsymbol{Y}_{t+1:T} \mid S_t = i, \boldsymbol{Y}_{1:t}) = P(\boldsymbol{Y}_{t+1:T} \mid S_t = i)$$

Substituting this into our previous equation gives:

$$P(S_t = i, \boldsymbol{y}) = P(S_t = i, \boldsymbol{Y}_{1:t}) \cdot P(\boldsymbol{Y}_{t+1:T} \mid S_t = i)$$

Define: - $\alpha_t(i) = P(S_t = i, \boldsymbol{Y}_{1:t})$ — Forward variable - $\beta_t(i) = P(\boldsymbol{Y}_{t+1:T} \mid S_t = i)$ — Backward variable
So:

$$P(S_t = i \mid \boldsymbol{y}) \propto \alpha_t(i) \cdot \beta_t(i)$$

Forward Recursion (Computing $\alpha_t(i)$)

Base case:

$$\alpha_1(i) = \pi_i \cdot b_i(y_1)$$

Where: - π_i is the initial state probability - $b_i(y_1)$ is the emission probability of observing y_1 in state i

Recursive case:

$$\alpha_t(i) = b_i(y_t) \sum_{j=1}^{N} A_{ji} \cdot \alpha_{t-1}(j)$$

Where: - A_{ji} is the transition probability from state j to state i *** **Derivation of the Forward Variable** $\alpha_t(i)$

We define the forward variable as:

$$\alpha_t(i) = P(S_t = i, \mathbf{Y}_{1:t})$$

This is the joint probability of being in state i at time t, and having observed the first t observations $\mathbf{Y}_{1:t} = (Y_1, Y_2, \ldots, Y_t)$.

Base Case: $t = 1$ At time $t = 1$, we initialize the forward variable as:

$$\alpha_1(i) = P(S_1 = i, Y_1 = y_1) = \pi_i \cdot b_i(y_1)$$

Where:

- π_i is the initial probability of being in state i
- $b_i(y_1)$ is the emission probability of observing y_1 from state i

Recursive Step: $t \geq 2$

We compute $\alpha_t(i) = P(S_t = i, \mathbf{Y}_{1:t})$ by marginalizing over all possible previous states $S_{t-1} = j$:

$$\alpha_t(i) = \sum_{j=1}^{N} P(S_t = i, S_{t-1} = j, \mathbf{Y}_{1:t})$$

Using the chain rule:

$$\alpha_t(i) = \sum_{j=1}^{N} P(S_t = i \mid S_{t-1} = j) \cdot P(Y_t = y_t \mid S_t = i) \cdot P(S_{t-1} = j, \mathbf{Y}_{1:t-1})$$

Recognizing that:

$$P(S_t = i \mid S_{t-1} = j) = A_{ji}, \quad P(Y_t = y_t \mid S_t = i) = b_i(y_t), \quad P(S_{t-1} = j, \mathbf{Y}_{1:t-1}) = \alpha_{t-1}(j)$$

We obtain the recursive formula:

$$\alpha_t(i) = b_i(y_t) \sum_{j=1}^{N} A_{ji} \cdot \alpha_{t-1}(j)$$

Backward Recursion (Computing $\beta_t(i)$)

Base case:

$$\beta_T(i) = 1 \quad \forall i$$

Recursive case:

$$\beta_t(j) = \sum_{i=1}^{N} A_{ji} \cdot b_i(y_{t+1}) \cdot \beta_{t+1}(i)$$

*** **Backward Recursion: Derivation of $\beta_t(j)$**

We define the backward variable as:

$$\beta_t(j) = P(Y_{t+1:T} \mid S_t = j)$$

This is the probability of observing the sequence $Y_{t+1}, Y_{t+2}, \ldots, Y_T$, given that the system is in state j at time t.

Base Case: $t = T$ At the final time step, there are no future observations, so we initialize:

$$\beta_T(i) = 1 \quad \text{for all } i$$

Recursive Step: $t = T-1, \ldots, 1$

We use the law of total probability over all possible next states $i \in \{1, \ldots, N\}$:

$$\beta_t(j) = \sum_{i=1}^{N} P(Y_{t+1:T}, S_{t+1} = i \mid S_t = j)$$

Apply the chain rule:

$$\beta_t(j) = \sum_{i=1}^{N} P(Y_{t+1:T} \mid S_{t+1} = i, S_t = j) \cdot P(S_{t+1} = i \mid S_t = j)$$

Using the conditional independence of future observations given $S_{t+1} = i$, we have:

$$P(Y_{t+1:T} \mid S_{t+1} = i, S_t = j) = P(Y_{t+1} \mid S_{t+1} = i) \cdot P(Y_{t+2:T} \mid S_{t+1} = i)$$

So:

$$\beta_t(j) = \sum_{i=1}^{N} A_{ji} \cdot b_i(y_{t+1}) \cdot \beta_{t+1}(i)$$

Final Recursion Formula

- **Base case:**

$$\beta_T(i) = 1$$

- **Recursive step:**

$$\beta_t(j) = \sum_{i=1}^{N} A_{ji} \cdot b_i(y_{t+1}) \cdot \beta_{t+1}(i)$$

Smoothed Posterior (Final Result)

$$P(S_t = i \mid \boldsymbol{y}) = \frac{\alpha_t(i) \cdot \beta_t(i)}{\sum_{k=1}^{N} \alpha_t(k) \cdot \beta_t(k)}$$

This normalization ensures the result is a valid probability distribution over states at time t.

Sum-Product Message Passing Interpretation of the Forward-Backward Algorithm

The Forward-Backward algorithm can be interpreted as a special case of the **sum-product algorithm**, which is a general method for exact inference in graphical models. In this interpretation:

- Forward messages are denoted $\mu_\alpha(X_n)$
- Backward messages are denoted $\mu_\beta(X_n)$

Forward Message Update Rule

The forward message $\mu_\alpha(X_n)$ sent from variable X_{n-1} to X_n can be written as:

$$\mu_\alpha(X_n = i) = \sum_{j=1}^{N} \mu_\alpha(X_{n-1} = j) \cdot \psi(X_{n-1} = j, X_n = i) \cdot \phi(Y_n \mid X_n = i)$$

For a Hidden Markov Model, this corresponds to:

- $\mu_\alpha(X_n = i) = \alpha_n(i)$: forward variable
- $\psi(X_{n-1}, X_n) = A_{ji}$: transition probability
- $\phi(Y_n \mid X_n = i) = b_i(y_n)$: emission probability

Therefore, the forward message (forward recursion) becomes:

$$\alpha_t(i) = b_i(y_t) \sum_{j=1}^{N} A_{ji} \cdot \alpha_{t-1}(j)$$

Where:

- $A_{ji} = P(S_t = i \mid S_{t-1} = j)$
- $b_i(y_t) = P(Y_t = y_t \mid S_t = i)$

Connection to Sum-Product Algorithm

This is a specific instance of the sum-product rule:

$$\mu_{\text{to node}}(x) = \sum_{x'} \mu_{\text{from node}}(x') \cdot \psi(x', x) \cdot \phi(x)$$

Backward Message in Sum-Product Form (Forward-Backward Algorithm)

In the **sum-product message passing** framework, the backward message sent from variable X_{n+1} to X_n is denoted:

$$\mu_\beta(X_n = j) = \sum_{i=1}^{N} \psi(X_n = j, X_{n+1} = i) \cdot \phi(Y_{n+1} \mid X_{n+1} = i) \cdot \mu_\beta(X_{n+1} = i)$$

Interpretation in HMM Terms

Mapping this to HMMs:

- $\mu_\beta(X_n = j) = \beta_n(j)$: the backward variable
- $\psi(X_n = j, X_{n+1} = i) = A_{ji}$: transition probability
- $\phi(Y_{n+1} \mid X_{n+1} = i) = b_i(y_{n+1})$: emission probability

So the backward recursion becomes:

$$\beta_t(j) = \sum_{i=1}^{N} A_{ji} \cdot b_i(y_{t+1}) \cdot \beta_{t+1}(i)$$

This corresponds exactly to the backward step of the Forward-Backward algorithm.

Base Case

At the final time step:

$$\beta_T(i) = 1 \quad \text{for all } i$$

Summary of Backward Message

The general backward message passing form in sum-product terms:

$$\mu_\beta(X_n = j) = \sum_{X_{n+1}} \psi(X_n = j, X_{n+1}) \cdot \phi(Y_{n+1} \mid X_{n+1}) \cdot \mu_\beta(X_{n+1})$$

Specialized to HMMs:

$$\beta_t(j) = \sum_{i=1}^{N} A_{ji} \cdot b_i(y_{t+1}) \cdot \beta_{t+1}(i)$$

Given a certain time t, I can identify the present and the future. The present d-separates the past from the future, st seats in between the present and the future, all the paths from the past to the future, pass to the present. In conditional independence term if I am observing the present the past does not influence the future.

Recursion that starts at time t=1 and computes alpha1, one for each choice of i: (1,...,C). Then **message passing to the future**, sends this vector to time t = 2, to compute alpha2 using the recursive formulation. Forward to the future

until I come to a generic time t, in which I compute alphat for all choice of i. Example of an algorithm that computes a vector alpha by leveraging whatever is being precomputed at the previous time step. I receive a message from the past and I sum overall states of the past in order to compute my message at time t that I will send to the future. Exactly the same thing we will do for beta. This time the recursion comes from the future. The base case is for t = T. Beta at time T is a vector of 1, which I'm gonna be sending back to time T-1 and I'll compute beta(T-1) with the message coming from time T and so on until i get to beta(t+1). Since I've to compute for all the time I just need to take alpha and say alpha go to the future until T, beta starts from T and go to time 0 and every time step whenever you meet multiply yourself and you get your posterior for each time of the sequence.
Forward-Backward algorithm is an example of a broader family of algorithm which are called the sum-product message passing algorithm, sending a message to the past to the future and a message from the future to the past, the forward message mu_alpha computed on a specific random variables, computed by summing over all the states from the past, from my left-adjacent neighbor Xn-1 i will receive a message mu_alpha_xn-1, I will combine this messages from the past with this function PSI which is a function that depends on the params of the model that joint together the random variables involved in a locality (Xn and Xn-1)→ local structure parameters that describe this dependence, those parameters are gonna get multiplied in the psi term, i will receive the messages from my neighbor combine with PSI term, and come up with my messages, can be generalized to multiple neighbors with more articulated PSI function, but it's the kind of algorithm that you use when you want to compute the posteriors of latent variables in your graphical model. This work if I can find a partial ordering in the random variables involved because there is a recursion, works only in lattices, graphical models of a certain structure, but it is combinatorial we need approximation to make it computationally feasable.
After posterior estimation we've a way to asses the probability of a single state at each time step of my sequence. Smoothing problem, we assume θ as given, at the contrary in the learning problem we ask what are the θs? Inferencial problem of finding θ. The joint probability of the observable given the parameters is called the likelihood, the simplest way we can do inference is maximum likelihood, so we're gonna be finding the parameters θs that maximize the likelihood. Log-likelihood is the logarithm of the likelihood, we gonna maximize it, because there are a lot of products and exponential and having the logarithm helps us a lot. Product over the dataset, joint distribution P(y_vec1, y_vec2, ..., y_vecN) factorization of the joint distribution into a product of simpler distribution one for each sample in the dataset, usual i.i.d. assumption. Usual assumption in ML, the sample of my Dataset are i.i.d. while the elements inside the sample instead are not i.i.d. The logarithm works nicely with the product: Log(a*b) = log (a) + log(b). Then factorize the joint distribution using the assumptions of my HMM, I need to introduce the hidden state in order to do that with marginalization and you obtain that little huge summation. Then there is a function you need to maximize, but there are non observables for the specific

sequence, i don't know about them. How do we do learning when some of the random variables are non-observable.
We estimate the HMM parameters $\theta = (\pi, A, B)$ via Maximum Likelihood Estimation (MLE):

$$\theta^* = \arg\max_\theta \prod_{n=1}^{N} P(Y^n \mid \theta)$$

Taking the logarithm of the likelihood:

$$\log P_\theta(Y^1, \ldots, Y^N) = \sum_{n=1}^{N} \log P_\theta(Y^n)$$

We marginalize over the hidden states:

$$P_\theta(Y^n) = \sum_{s_1^n, \ldots, s_T^n} P_\theta(Y^n, S^n)$$

Using the HMM factorization of the joint distribution:

$$P_\theta(Y^n, S^n) = P(s_1^n) \prod_{t=2}^{T} P(s_t^n \mid s_{t-1}^n) \prod_{t=1}^{T} P(y_t^n \mid s_t^n)$$

Thus, the log-likelihood of the sequence becomes:

$$\log P_\theta(Y^n) = \log \sum_{s_1^n, \ldots, s_T^n} P(s_1^n) \prod_{t=2}^{T} P(s_t^n \mid s_{t-1}^n) \prod_{t=1}^{T} P(y_t^n \mid s_t^n)$$

Take a lip of faith, there is this magic indicator variables that gives you all the knowledge that you need: $z_{t,i}^n$ I have 1 if I take the n-th seq. of my dataset and at time t I'm at state i and 0 otherwise, essentially tells you what is the hidden state at time t for sequence n and, tells you for which i it is true. It's supplying the missing knowledge to assign responsability to your hidden state, because you cannot observe. Then plug into the model. Rewrite the things you've previously using the indicator variable. We introduce indicator variables $z_{t,i}^n = \mathbb{I}(S_t^n = i)$ to express the complete-data log-likelihood with parameters $\theta = (\pi, A, B)$:

$$\log P(Y, Z \mid \theta) = \sum_{n=1}^{N} \log P(Y^n, S^n \mid \theta)$$

This expands to:

$$\log P(Y, Z \mid \theta) = \sum_{n=1}^{N} \Bigg[\sum_{i=1}^{K} z_{1,i}^{n} \log \pi_i + \sum_{t=2}^{T_n} \sum_{i=1}^{K} \sum_{j=1}^{K} z_{t-1,j}^{n} z_{t,i}^{n} \log A_{ji} + \sum_{t=1}^{T_n} \sum_{i=1}^{K} z_{t,i}^{n} \log b_i(y_t^n) \Bigg]$$

The first production stays equal, then happens some magic, there was a summation, now there is a multiplication of the hidden states of the prior $[P(S1 = 1)P(Y1^n|S1 = 1)]*[P(S1 = 2)P(Y1^n|S1 = 2)]*...*[P(S1 = C)P(Y1^n|S1 = C)]$ but is not only a multiplication, is on the power of the indicator variable for the n-seq. those are gonna be all zero, except from a single indicator variable that tells you what is the specific state i that i'm assigning to the first time step in the n-seq, 1*1*1*1* $P(s1 = i)P(y1|s1 = 1)$*1*1, exactly the same summation as before. Summation is identical to multiplication as well as we've this indicator variables. And repeat the same thing for all the probability involved in my factorization, rewrite summation in a product and making to the power of z. The only complex thing is that whenever there is the joint distribution involved I'll have two indicator variables involved. One for the previous time and one for the future time, and this big product is gonna be 1 for all the element except when both the first indicator variable and the second indicator variable are one so when I'm in a specific right hidden state and i'm landing in a specific right hidden state.
Now having all products the thing get easier, the logarithm starts entering the production and transform in summation. Use also the rule for the logarithm of an exponential, I managed to write a complex thing where I've isolated params (π, A, b) in different elems of the summation, nice because I'm gonna be doing Maximum likelihood I'm gonna be differentiating these things, If i differentiate with respect to π, i consider only the first term because term 2 and 3 does not depend from π. To derive the learning equation for π i will differentiate only the first term, etc etc only caveat I do not have this indicators variables , I've cheated.

Expectation-Maximization: 2 step algorithm, formulating hypothesis in the first step, and using them to update params in the second step. In the first step i will make hypothesis on what are the likely values for those z indicator variables which i do not know, in the second step I'm gonna be using those hypothesized values for z to compute an update of the params and iterate again until convergence.
E-Step makes hypothesis: compute a value for this magic Q function which is a function of the parameters θ. **M-Step** : maximize the Q function to find the parameters of the model θ. But it is an iterative algorithm, at the E-step I find an estimate of the Q function at the step k+1, using the values of the params

at time k. Here I'm finding new values of the params at time k+1 using my current estimate of Q at time k+1, those params will be used to estimate Q at the next iteration, circular dependence.
First the computation of some Q function which is gonna be based on me doing some hypothesis on the indicator variables, then using the Q function to optimize my params, go on until my loglikelihood start not increasing. E-STEP, you do not need the values of the z, but their expectation, their expected value. z takes values in 0 and 1, so we make an expectation over a discrete random variable, so the expectation is a linear operator, that makes the weighted summation of the values of the Z weighted by the probability of Z. We've a slightly more complex expectation but essentially what we are doing, essentially Z is gonna be the subject of the expectation, is gonna be marginalized out by the expectation operator, given y and θ means consider them fixed, do not apply marginalization, y and θ stay fixed. I'm assessing f for a specific y and θ, given fixed, for all possible values of z. But z has only two values. Whenever I'm gonna applying the expectation operator, all variables that are not subject to the expectation are gonna stay as they're, z is gonna be marginalize out and whatever it remains is only probability of Z, if you need to compute a function for a variable for which you do not have values the sensible way of computing that function is computing for all the possible values of that variable you do not have values and make an average, the variables we do not know nothing about is an indicator variable, only have 0 and 1, we only need to compute this thing for only P(z =1). Whenever we do learning, expectation step, we compute the expectation of the log-likelihood w.r.t. to my z variables, which is gonna be transforming all the z, the rest remains fixed. I will be applying the expectation operation to the log-likelihood $logP(Y, Z|theta)$. Expectation passes through the sum and brings out the constant. We arrive at a guy that we can estimate with the forward backward, $\mathbb{E}_Z[z^n_{1,i}]$ will transform in the posterior of the first hidden state of my sequence $P(S1 = i|Y^n, theta)$
Log-likelihood of HMM which splits into a sum of terms with each terms that depends only on a single distribution of our model, nice thing we can optimize them independently. Issue we're assuming that we have an oracle knowledge of which hidden state I'm in, in a specific sequence at a specific time.
Expectation-Maximization: you don't really have to know those indicator variables, you only need to be able to compute the expectations of that function w.r.t. to those indicator variables.
Through the E-step we create an estimated function Q, which is the Expectation of the completed log-likelihood, in the second step we can maximize it changing the parameters essentially, now this change of params will reflect on the first step because i'm gonna be changing my Q function using these new values of the params and go on alternating E and M step until log-likelihood does not change too much.
We're running an expectation over a binary variable, it's zero except when the binary variable is equal to one, and when it's equal to one, i compute my function considering θ and y fixed, with Z fixed to one, the only surviving term of the expectation, multiplied by this probability of z. My function in the specific case

of the expectation step is the log-likelihood. I'm computing the log-likelihood for the specific case of z fixed to 1.
The first term of the expectation over f(z, theta, y) is 0 because z = 0 is plugged inside the function and it appears in all the terms zero them.
The net effect of applying the expectation operator is that whatever before was an indicator variable becomes a posterior. I know the parameters because I assume i can use the current values of the params π, A,b. Alpha-Beta recursion as an algorithm to compute the posterior knowing the params, essentially the expectation step is computing the posterior given the current estimate of the params, next stage I'll change the param, I'll have to revaluate evaluate the posterior. Expectation step is essentially computing the posterior, maximization step is using the value of the posterior i'll find to compute the new values for the params. I need some params update equation, i need to maximize the Q function, derivation/differentiation of the Q function.
The elems of π are not free params they need to be values of a distribution, positive and sum to one.
I cannot use free optimization to find them, i cannot simply differentiate that equation but i need to do constraint optimization. $\rightarrow$ Lagrange multipliers
The optimization problem is not altered because I've altered i nice square zero,

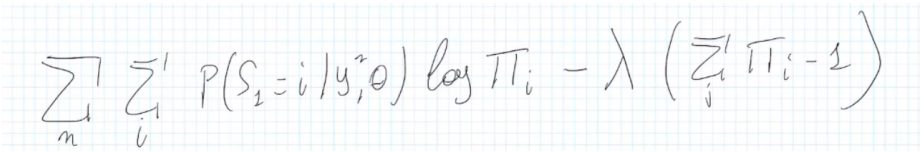


Figure 35:

because the sum over all j of pij is 1, and 1-1 is 0 so I've added 0. Just a trick to keep things in the simplex.
If b, emission distribution, is a gaussian rather than a multinomial what you get is an update rule for the mean mu and the variance sigma.
Remember is a maximization, we're maximizing the function loglikelihood. Whenever we're gonna be rewriting the eq. to update pi_k , this eq. is function of the posterior, in the E-step you're gonna compute the posterior given the current value of the params, because forward-backward algorithm use the value of the params of the model to compute the posterior, in M-step we use the values of those posterior to update the params, that's why we need two stages.
Let's reason on the update rule of pi_i : in order to know what is the probability of being in a specific state i at the beginning, what I need to do is observe how many times i've been in that state divided by the total number of sequence. But I cannot observe how many times I've been in that i state at the beginning of the seq. because it's a non observable thing, but the posterior will tell me a probabilistic count/estimation of that number.
Same thing for the transition: if you look to the transition matrix you count through the posterior how many times i transited from a specific state j at time

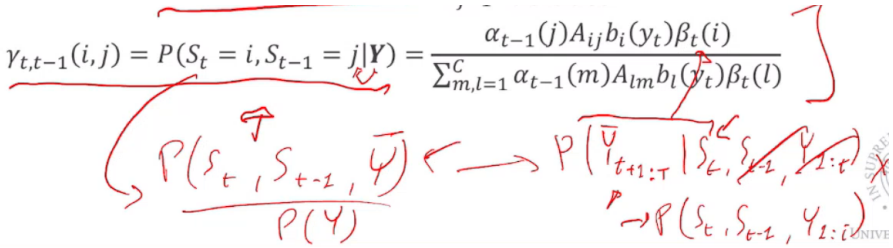


Figure 36:

t-1 to a specific state i at time t, w.r.t. to all the transits from a state j to any possible arrival state. EM is a general framework for learning: we have a

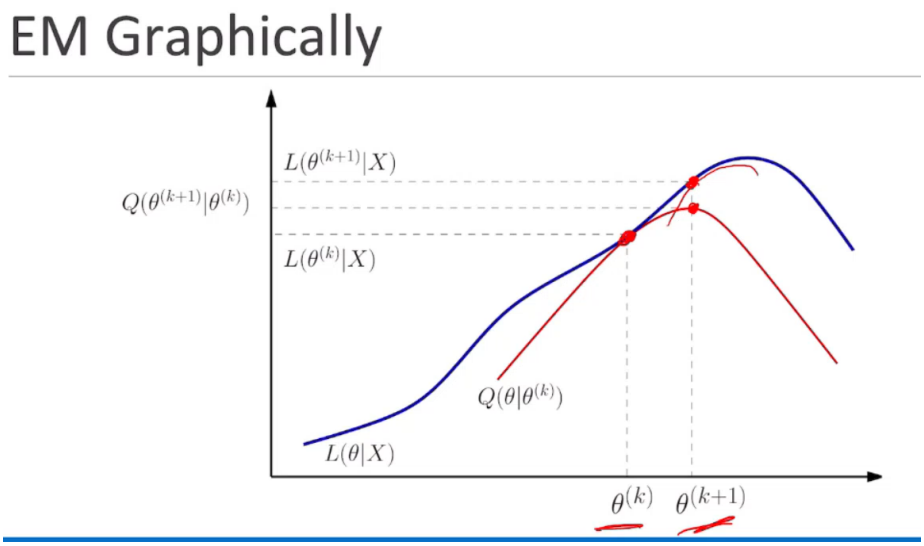


Figure 37:

problem to solve in probalistic models that is to maximize the likelihood, your likelihood is nasty, you do not know how to handle in general, EM tells you if you want to optimize the likelihood you can focus on optimizing a slightly simpler problem, optimize the Q function, and you have some theoretical assurances that if you maximize the q function is gonna be easier and gonna be approximating the maximization of the complete likelihood.
In the blue line is the log-likelihood, whenever you say in the E-step let's fix theta_k, i'm positioning is some point of the likelihood function, then what does the expectation step gives me this beautiful little concave function that has the property to bound from below the likelihood.
The kind of assurance you were given is that if you maximize (easily) the Q function you'll find a point which at the next step you'll project on the actual

likelihood, then you're gonna be building another q function, since the q fun. is guaranteed to bound from below the log-likelihood for certainly it does not decrease the likelihood.
The q-function instead of being touching the likelihood can be slightly lower, depending on how much approximation you take, sometimes we need to solve very difficult problem, that even computing the posterior is very difficult, instead of computing the true posterior we compute approximation of those posteriors, that still guarantees to not decrease the likelihood, though you are not guaranteed that you're gonna be staying touching the likelihood, you lose the equality, but you still keep the boundness from below. How close depends on the quality of the approximation
Guess what? Too difficult problem appears when you use NN to approximate probability distribution, that happens when you use diffusion model, variational autoencoder etc ...
HMM to find regime to describe the volatility (change in price through days) of stocks, HMM fit to this observations to assign hidden states to observations, imagine I choose to fit an HMM with only 2 hidden states, for each of the time point in the time series will assign one hidden stases. In black the approximation of the HMM fitted with 2 hidden states of the volatility, with 2 hidden states two level of volatility, with 2 hidden states you can only model 2 level of volatility low and medium, if you increase the number of hidden states you get better and better in approximating the time series, and you start capturing different degrees or regimes of volatility. Hidden states are just clustering of observations, all of this is unsupervised. HMM is an unsupervised model to cluster observations in a time series.
How do I assign the most likely hidden states to each point of the TS? I can do it with the posterior, but it will assign one hidden state independently for each time step, i want to solve the problem of assigning it optimally all togheter.
Decoding problem = find the joint assignment of all the hidden states all together.
Forward backward is a sum product alg., Hidden state assignment requires a max over product. Sort of dynamic programming problem.
It is combinatorial if I approach by brute force, it goes all through the C possible assignments of T terms.
epsilon_t is a vector with C values, one for each possible choice of the hidden state of my predecessor which I don't know yet, I'll know on my way back, I'll be solving this maximization problem for each choice of sT-1 and write it inside epsilon_t and keep track of which of the C values is associated to the solution of the maximization problem. Now I will pass this epsilon_t to time t-1, that computes epsilon_t-1 pass it backward... until i arrive to epsilon1 that i can solve directly. I can find the best state s1* that I send to time 2, plug into epsilon2 vector, choose my S2* accordingly, cause I've already solved this problem, send to S3... and so on, backward recursion I compute all this vector epsilon and solve intermediate problem of maximization, when I reach the time 0 i can finally solve one full optimization problem for state 0, which can then send this choice to time 1, which can use to choose its own state, that sends

to time 2, dynamic programming problem, quadratic instead of being t to the power of C.
The graphical model by its own nature let also to design efficient algorithm.
The Viterbi algorithm computes the most likely sequence of hidden states in a Hidden Markov Model (HMM), given a sequence of observations over T time steps. The algorithm proceeds as follows:

Step 1: Initialization

At time $t = 1$, for each state s, compute the initial score:

$$\delta_1(s) = P(S_1 = s) \cdot P(Y_1 \mid S_1 = s)$$

This represents the probability of starting in state s and generating the first observation. Also, store the state that gives the highest score for future backtracking.

Step 2: Recursion (Forward Pass)

For each time step $t = 2$ to T, and for each possible current state s_t, compute:

$$\delta_t(s_t) = \max_{s_{t-1}} [\delta_{t-1}(s_{t-1}) \cdot P(s_t \mid s_{t-1}) \cdot P(Y_t \mid s_t)]$$

This equation finds the best previous state s_{t-1} that leads to the current state s_t, considering both the transition probability and the emission likelihood. At each step, store the *backpointer*, i.e., the s_{t-1} that gave the maximum value.

Step 3: Termination

At the final time step T, select the ending state s_T^* with the highest accumulated score:

$$s_T^* = \arg\max_s \delta_T(s)$$

Step 4: Backtrace (Backward Pass)

Starting from s_T^*, use the stored backpointers to iteratively recover the optimal state sequence by tracing backward through the selected states:

$$s_{t-1}^* = \text{backpointer}(s_t^*), \quad \text{for } t = T, T-1, \ldots, 2$$

This yields the most probable path $s_1^*, s_2^*, \ldots, s_T^*$ that explains the observation sequence.
Model the conditional generation of data, imagine you have a stream of a sequence in input, and you want to transform in a sequence in output, classifying a seq. for each element of an input seq. you want to output a classification.
I/O HMM: your hidden state not only emits an observation but the choice of the hidden state itself is influenced by additional observable input information (exogenus). This is completely a RNN. Only represents input information into a discrete alphabet.
Bidirectional I/O Input Driven Models: two directions to look at your series, no sense in time series, but genomic data, couple of HMM one that read the info from left to right and another that read the info from right to left, the output depends on both the hidden state from left to right and the one from right to left.

HMM for multivariate time series, **Coupled HMM**, healthcare application: EEG + respiratory signal: 2 covariates quite different in dynamics, if you fit them in a single state you're gonna be mixing 2 signals with radically different dynamics into a single hidden states, fuse them EEG info in an hidden state chain and respiratory signal in another hidden state chain and the hidden state in which you end up will depend on the hidden state of the EEG and the hidden state of the respiratory signal. The hidden state operates as a sort of filter to fuse substantially different info and based on those hidden states predict something
HMM generalized to more articulated Bayesian Network that have not a line-based structure. → Dynamic Bayesian Network only requirement are DAG, allowed to change/adapat not necessarily w.r.t. to time. Genealogical trees. Dataset of genealogical tree, if i want a single probabilistic model to model all of them, must be adaptive w.r.t. to tree of different sizes. Use HMM for trees, in HMM is robust to sequence of different length, (stationarity assumption, same transition distribution for all time step), for tree i need a transition distribution I can reuse across the nodes.
$P(GAdam|GAbel, GSeth)$ this needs to factorize into something $w1P(GAdam|GAbel)+$ $w2P(GAdam|GSeth)$ this will work if one has 2,3,4 children provided that this probability here can be reused for all the children.
Similar to the assumption you made with NN to process trees or graphs, you reuse the same set of params, and make sure that the aggregation mechanism you use is independent of the number of children.
Weight sharing , reusing of params and ability to be aggregated over a variable number of neighbors.
Hierarchical HMM parallelism with hierarchical clustering, general argument for text → specialized into a subtopic → further specialized and the Y are the words i generate consistent with the subtopic.

11 Markov Random Fields

Undirected graphs. Undirected edges change the nature of the relationship you can represent, directionality implies an asymmetric relation source/destination or cause/effect. Undirected edges represent more general form of associations or dependencies. Symmetric dependencies: mutual constraint, the values can't change independently but are somehow linked by some form of functions.
In directed models the association between two random variables has a probabilistic interpretation, in Bayesian network if there is an oriented edge between two nodes means there is a conditional probability associated to that (P(targed node | (family of)source nodes)). Here a function attached to edges, the overall model will have the probability but this won't factorize in a product of probabilities, what BN does, here a joint distr. for the full graph won't factorize in a product of distributions but in a product of functions.
These models take the form of the structure of the data they're modeling, if they model sequences they will be line, if you model a tree they'll look like a

tree, a picture they seem a grid.
Models the fact that your data is made by simple pieces that do not change independently, in a picture if pixels are allowed to change independently you do not obtain a semantically valid image, even if you sample randomly pixels from a uniform distribution you have a very low prob. to obtain an actual/realistic image. A node for each pixel and nearby pixels should have similar values, an edge between them, smoothness constraints on non-monolitic data. Relations between pixels are expressed with constraints between pixels by functions.
Model made of nodes, associated to random variables and so I'll have a joint distribution over those random variables involved in the model. Factorized as a product of potential functions, I can obtain a prob. distr. over random variables by taking the product of functions and dividing by a magical factor Z. This thing works if I constraint the form of the function PSI.
$P(X) = \frac{1}{Z} \prod_C \psi_C(X_C)$ and $Z = \sum_X \prod_C \psi_C(X_C)$
The product do not run from 1 to N, the number of random variables, but from 1 to C, the number of maximal cliques (connected components: sub graphs where every member is connected with every other members, maximal) of the model, Potential functions are defined over the cliques of the graph, will apply over a subset of the nodes X that are part of the clique C. ψ must be positive definite. Under this assumption is a valid factorization of the joint distribution of the random variables involved.
I need that all things sum to 1 to be a distribution $\rightarrow$ Z is a normalization factor.
Partition function simplest case: combinatorial term, worst case continuous integral.
Potential functions are not probabilities but are sort of functions that tell us what are the preferred configurations, physics terminologies, potential of the configuration of the random variables. I want to reach a certain config. of the random variables and the potential indicates the road to achieve this ideal configuration. Crafting into the model prior knowledge into the form on how we write those potential functions.
One way to get easily a nice potential function with all the properties that we like is by choosing potential function to be exponential of some function, called energy function. Spoiler: exponential over minus an energy, that is a linear function.
$\psi_C(X_C) = \exp\{-E(X_C)\}$ and $\psi_C(X_C) = \exp\left(\sum_k \theta_{c,k} f_{c,k}(X_C)\right)$
Potential function is an exponential over a linear combination of things: an exponential over a sum over k. feature functions are defined over the same random variables of the cliques, not need to include all random variables of the clique, can be a subset. The summation does not span the random variables in the cliques, summation over a dictionary of feature functions, k feature functions, a vector of feature functions, each of them operates on a subset of the random variables of the cliques with no assumptions on which one you're taking. You can decide to have a partition, overlapping, ... You have a separate page of the dictionary for each different clique (indexed by C), you can also share between cliques. Feature functions are usually defined by designer, there is not learning.

So where is learning? In the param theta. Not incredibly complex, number of param equal to the number of feature functions, while in HMM the number of params is (num. of values you can have in the latent variables) to the power of 2 becuase the transition distribution is a matrix + the vector of priors + the matrix of the emission distribution. In MRF is more parameter efficient w.r.t. to Bayesian Network (BN), no need of representing distribution with params, with params represent combination of functions, distribution will arise naturally, no free lunch, you pay this parametric efficiency with the complexity of the partition function.
feature functions implicitly define the potential function, theta are the params. The factorization of the MRF into cliques is evident from the graph, but I don't see the feature functions, that graphical formalism is not sufficient.
Factor graphs are graphs that really represent a computational perspective over how inference is done in graphical model (directed/undirected). RV are again rounded circles, shaded/not shaded, edges, but a different type of node as well. Representation that highlights the fact that psi1 factorizes into feature functions. Explicitly writing the features, with a specific type of node, black square factor node, represents the fact that the random variables attached to it are all used by a feature function. There is not going to be edge between circle nodes/random variables, they go through black square. Given a clique I obtain

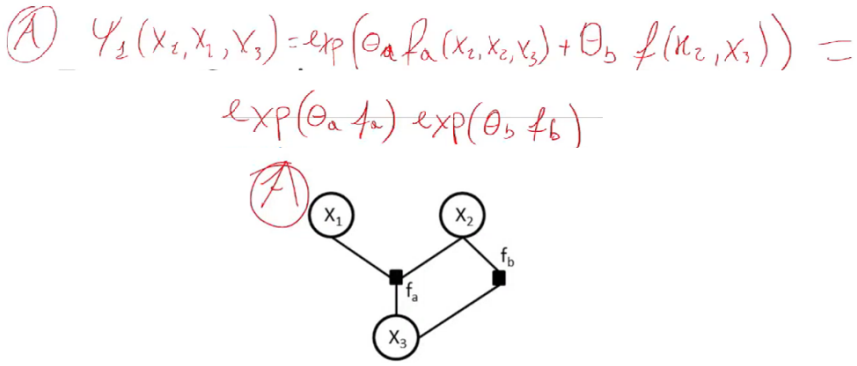


Figure 38:

different factorizations depending on how I choose to pick up my feature functions and the factor graph represents exactly to what factorization i refer. The factor graphs represents also a factorization of a directed model, e.g. a HMM.
Factor graphs represent any graphical model whether they're directed or undirected, all the inference procedures, message passing, alpha beta recursion ..are defined in the general case as message passing algorithms over the structure of the factor graphs.
Sum-Product Inference or alpha-beta recursion these algorithms are defined over factor graphs, single way to compute whatever you need to compute for your model independently from the orientations of the edges, compile your model

into a factor graph, ready-made inferential procedures that operates by message passing on the structure of the graph.
Message passing to be well defined and convergent, the factor graph should define a partial order and the more it looks like a chain, a sequence, or a tree the more is efficient.
Approximated inference procedures defined over factor graphs when they are not chain-like or tree-like.
Variational-sampling based approximation. When the problem is too difficult over this kind of structures, you have to find ways of solving the problem either you approximate the structure or you approximate the inferential procedure.
junction tree $\rightarrow$ I take something that is not a tree, a graph that is complex, i try to extract a tree from it and run exact inference on this tree.
Alternative keep the structure as is, and focus on making inference approximated. The first works mostly on probabilistic model, the last both on probabilistic model as well as NN.
Look to a constrained version of the MRF, which leverages a restriction based on the nature of the random variables typically involved in a ML model. In a ML model there are some random variables that are always observed: those corresponding to the input, the X of a ML model are always observable, even at test/prediction time. Any ML model can be split in x_{bold}, the input to the model, always observed, and y is the prediction of the model, non observable. Non putting into the game latent variables/hidden states.
Even in HMM, we model the joint prob P_theta(X,Y), when we write the likelihood we write the joint likelihood of all the joint variables involved in my model, and we are learning , parametrized distributions theta in HMM are the params of transition, emission and prior distribution, we'are learning those theta params for the joint distribution.
My ML problem is the usually predictive task of returning the most likely Y given X: P_theta(X,Y) = P_theta(Y|X) * P_theta(X), only the second distribution is really relevant for this task, but when we're training a model by MLE on the joint distribution I'm learning the first distribution, so inside the params of this distribution are not only the info to solve the task of prediction but is also leaking info on the distribution of the data which is by definition a nightmare, solving the problem of learning the joint probability of X and Y is a much more complex problem than solving and learning this thing alone.The first (the joint), learns how X and Y move together in the space where data lives, the second one only learn how Y should behave in the point in which X make sense.
MRF instead of learning the joint distribution of X and Y, learn the conditional probability of Y given X. $\rightarrow$ Conditional Markov Random field (CRF).
$P(\mathbf{Y}|\mathbf{X}) = \frac{1}{Z(\mathbf{X})} \prod_k \exp\{\theta_k f_k(\mathbf{X}_k, \mathbf{Y}_k)\}$ and $Z(\mathbf{X}) = \sum_{\mathbf{y}} \prod_k \exp\{\theta_k f_k(\mathbf{X}_k, \mathbf{Y}_k = \mathbf{y}_k)\}$
My MRF is made of a subset of a variable Xk that are always visible because they're inputed from outside and the rest of the variables, whatever they are, hidden. Whenever I do the production over k, the Yk are unknown but the Xk are known as they come from outside, the partition term is no longer Z, no longer a partition term that marginalized over all the random variables in the MRF,

marginalizes over all the random variables that are non observed, infact observed ones are passed as arguments, they're always observable, I don't need to marginalize among all the possible values that X can take because I do not care about it, I'm modeling the probability for x given, not the likelihood. Simplify the tractability of the partition term, reduces the amount of marginalization, less difficult model, the params will need to capture less info. Through conditional MRF you can solve discriminative problems in a probabilistic approach which is typically generative. Probabilistic approaches go through learning the joint distribution (generative approach), instead discriminative approach learns just a separating function (what NN do whenever you train to classify things, assume input always available).
Grid like models, grid of non visible Y and observable blues, not connected with themselves, would mean I'm trying to find correlation among the input, you need to go through some hidden knowledge. X_i the observable value of the pixel i of an image, e.g . black/white Xi = {+1,-1}. This is a noisy image, obtain a picture of something acquired with noise, some pixels are flipped, I want to do image denoising.
Constraining feature functions in such a way they do image denoising, some pixels have a not right observed values, X_i is the observed value, Y_i is the actual true value possibly correcting the error in the observed value. Y_i = {+1,-1} non observable, with the corrected version of the noise version of the pixel whenever necessary. f1 is a feature function between the observed and the unobserved version of the pxel, what could possible be a good rule to embed in the feature function, the constraint typically assumed is that on averaged it going to be the same value, in most of the picture there is consistency between the two, unconsistency will be allowed if X_i is a noisy pixel.
f1 = X_i * Y_i, sign concordance, they need to have the same values because they're binary values, the role of f1 is constraining or better suggesting strongly that they should have the same value. This is not denoising anything however, need to operate on Y_i and adding another constraint into f2 that have a sort of cleaning effect w.r.t. to noise, I can smooth over the neighborhood, f2 = YiYj constraint the neighbor to have the same sign, these 2 little rules applied to all nodes in the graph, I can get a model that when trained denoise the image, because it will find a solution to this problem/a trade off between the first and the second feature function, that ensures that if there is a discrepancy between X_i and Y_i this is motivated by the fact that Y_i needs to flip X_i because is sorrounded by Y_j pixels with a different sign, feature functions are really simple, giving the model some general indication on how the things that make up the data should behave, here we are imposing consistency (between hidden interp. & observation) and averaging (consistency between neighbors). You can reuse/replicate the same two feature functions for all the models, only two params, very parameter-efficient, strong inductive bias if the structure of your problem don't match this constraint anything does not work but if it match very efficient, complexity control.
Discriminative learning through this approach but through probabilistic model which by nature if you don't focus on the conditional distribution would be a

fully generative approach. 2 separate paradigms: SVM, classification of a NN, discriminative learning, focus on the conditional prob. of the output given the input. No info on where the real data resides. When you train a NN, to map input to output in a discriminative way, the NN learns to project a certain type of input in a certain prediction, couldn't tell if the input you're supplying falls outside of the data distribution, if it's not within P(X), because conditional model does not have P(X) inside its param, e.g. a NN trained in a discriminative does not have any info within its param that model where input data resides with high likelihood → behaves in over-confident data. Generative models are much more complex, they have a model of the data distribution, but you can reason on certainty and uncertainty of the data.
Naive Bayes classifier is trained generatively, to model the joint probability of the class and the feature that describe your individual P(c,k1,...,kk) and you use it to predict P(c|X) but it is trained in a generative way. It is an example of generative classifier, can be turned in a discriminative one if the parameters are trained to learn the conditional distribution P(C|k1,...kk). Still using this simple factorization that tells you that given the class the features of your input becomes independent.
The discriminative version of the Naive Bayes classifier, looks like a MRF, is an exponential over a linear combination of the input features → Logistic regression. It's trained in a fully discriminative way by maximizing the posterior of the class given the data, it learns a conditionally probability for a classifier and it is a MRF. Confront our generative model HMM with is discriminative

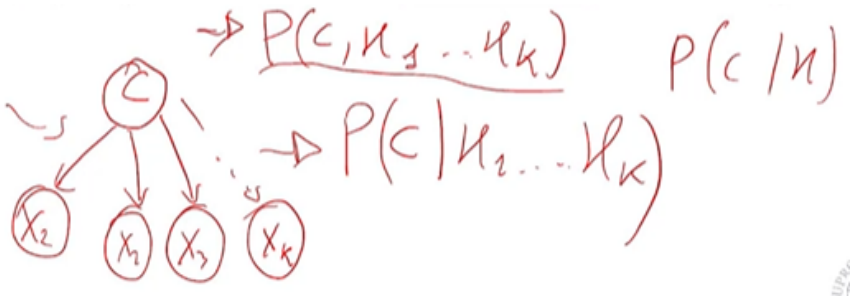


Figure 39: Naive Bayes

counterpart Linear conditional random field, linear because of the structure of the conditional random field.
They look alike if I have drawn the HMM using its factor graph they would be indistinguishable, but they are not the same thing: they are parametrized differently, HMM is parametrized in term of distributions → a lot of param, the CRF in term of params of linear combination of feature functions, much less params, HMM are trained in a generative sense, (St, X) together, CRF is trained in a conditional fashion you maximize (St—X) simpler problem.
The fact that I can relax the need of having distributions to factorize my joint as in HMM, allows me to have feature functions that are more creative and i

put together more info, in HMM typically i have an emission distribution that links the current state with the current observation, in CRF the hidden state at time t can be linked to observations in 3 time steps, if i put them in the same feature function I'll pay as one param.

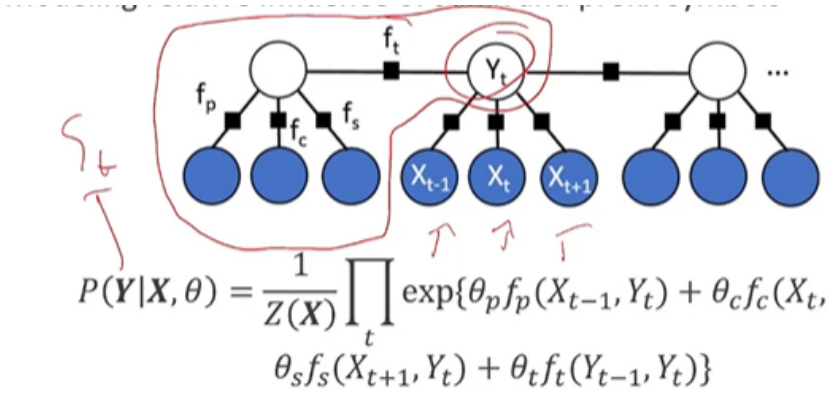


Figure 40:

$$P(\mathbf{Y} \mid \mathbf{X}, \theta) = \frac{1}{Z(\mathbf{X})} \prod_t \exp\left\{\theta_p f_p(X_{t-1}, Y_t) + \theta_c f_c(X_t, Y_t) + \theta_s f_s(X_{t+1}, Y_t) + \theta_t f_t(Y_{t-1}, Y_t)\right\}$$

Conditional probability because it's a CRF, where X, the observable elems of my sequences are given, so fixed, the model expects to receive them from outside, Y are the counterparts of the hidden states, they are non- observable, I've my partition function and a product over the cliques, it's a linear model: big circle in red is a full clique and the feature fucntions are decomposing that joint clique into a summation or a product, depends where you position, of 4 feature functions, fp links the current hidden state with the previous observable observation, fc links the current hidden state with the current observation, fs links the current hidden state with the future observation + ft which links the current hidden state with the future hidden state. One clique for each time, PSI is replicated, multiplication over time , at each time you have exactly the same clique and for each clique you reuse the same params. Only 4 params.
CRF like an HMM but lost all directionality in the arrows, factor graph. squares = factor nodes, make very explicit what is the function that connects random variable, gives them names. General enough to represent not only probability, f_t in HMM would be the transition probability, they become functions. In MRF functions could be anything provided that they end up into an exponential → consistent with Hammersley–Clifford theorem.
We can express more articulated relationships between random variables without the fear of having exploding computational complexity.
P(Y|X) simpler when you learn because it does not contain the marginal of the data P(X), very difficult distribution to capture. Simplify the learning prob-

lem, the solution is less general, because we don't have the joint of Y and X, the Y we obtain are only valid within the scope of reasonable variability of X. P(Y|X,theta) in MRF rewrites as a product of potential functions PSI, all are an exponential of the linear combination of the feature functions, the parameters of the model are the params of that linear combination. The model can be more complex but in general when you use it is because of its simplicity, the factorization/produttoria needs to run over the cliques of the graph, anyway are the feature functions that define the complexity of the interaction between the random variables.
Feature functions describe some sort of smoothness constraint or how random variable cochanges. If I am to compute conditional probability of Ybold, conditional probability of a full sequence of a certain length, given X bold, input seq. of a certain length, Y is the output seq., X is the input seq, Seq. classification problem predict an element in output for each element of X
Product for all the time, I'm reusing the feature functions for all the time steps, it's a design choice, limit the number of params and let us to reuse with seq. of the different length. First product is unfolding over time, the second product is the sum over k inside the exponential, when you take the summation out become a product.
Used to do Part-Of-Speech tagging, you parse a seq. of words and you want to attribute each word its role in the sentence, you choose feature function to achieve this objective. If a word is the noun of a person, the first name, the

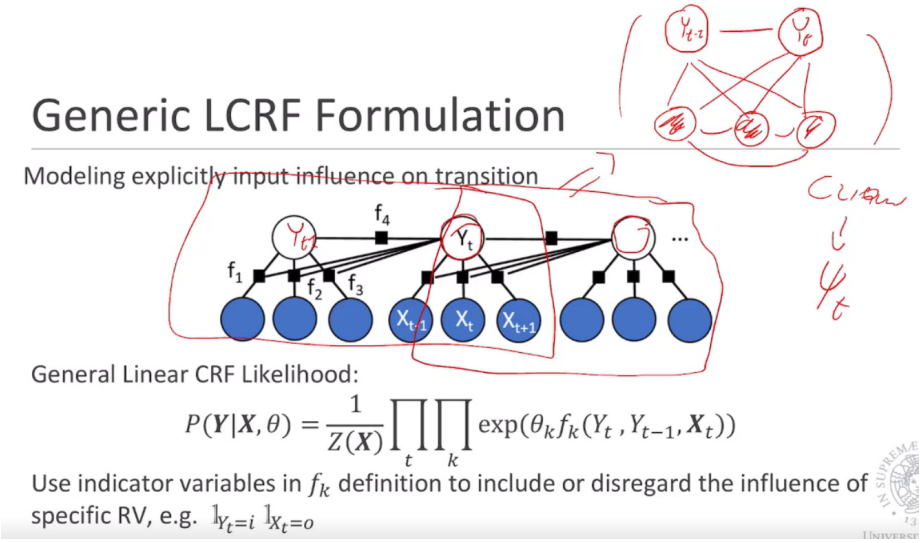


Figure 41:

feature function is a list of names, a dictionary with all the valid names in a language, the feature function will give an high value whenever the current observed X_t is inside the list.

It's a look up table, useless feature function. New name are not in the list. High values if the observable that you have start with a capital letter. This for the observable, when we come to non-observable they are often things that make sure that non observable are consistent one another, non observable at time t is very similar to the one at time t-1. Or in case of part of speech tagging, if a time t-1 i have observed a noun, i would be very happy to observe afterward a verb and then an article... You can think as rule based system where the way you combine the rules it's given by theta params, fk dictionary of rules, thetak are the weights learned to decide the relevance of each rule.
Indicator variable: e.g. the rule applies only if 1 if the hidden variable is in a certain state and the output variable is another thing.
e.g. if Yt is a noun Yt+1 should not be an article. Paradigm interesting for learning and reasoning integration.
If we have to do learning at some point we have to do posterior estimation. Estimating the posterior in this particular case is estimating a distribution over the non-observed variables which i want to predict, the Y_i.
X is observed, my input seq, Y_t and Y_t-1 are my (non always) observed random variables. It is solved again in the linear conditional random field case by the forward-backward propagation, cause it's a line, use again forward-backward recursion. Forward-Backward inference is an example of sum products alg. and MRF and in general factor graphs are a formalism that defines a straightforward way of running max product and sum product, in this case we need sum product cause it is a posterior estimation.
alpha_t-1 is a message that comes from the past, and goes to Y_t-1, and a message beta_t that comes from the future. As in HMM I get the two messages at the two entrances of Y_t, i need to combine them. In the HMM I multiplied the alphas with the betas, here in the CRF, I multiply the alpha with the beta, mediated by the potential function. The potential function is defined over the clique.
In the posterior estimation I assume the parameters are fixed and known. The psi function can be computed, the alpha and beta needs to be computed again by some form of recursive message passing.
The alpha_t(i) = alpha_t(Y_t=i) is computed by taking a message that comes from my previous time step which is alpha, summing over all j values, a_t-1 is a vector and j is running over the vector. You're combining the values of the vector from alpha with again the potential function of the clique that combines the state at times t, the state at time t-1 and the other elem of the clique. This recursion starts at time 0, (base case) and will move forward. Something similar for the backward messages, the backward messages at time t for a specific state j will be the sum overall states i for all messages in the future Beta_t+1 combined by the potential function between time t and t+1. The potential functions act as mediators of the message that i'm receiving. Whenever the message enter a clique, that message is taken, summed up and combined, used as a weighing the clique potential. This is a general concept, whenever a message enters a potential, that message gets summed in all its components and each element of the sum gets weighted by the potential of the clique.

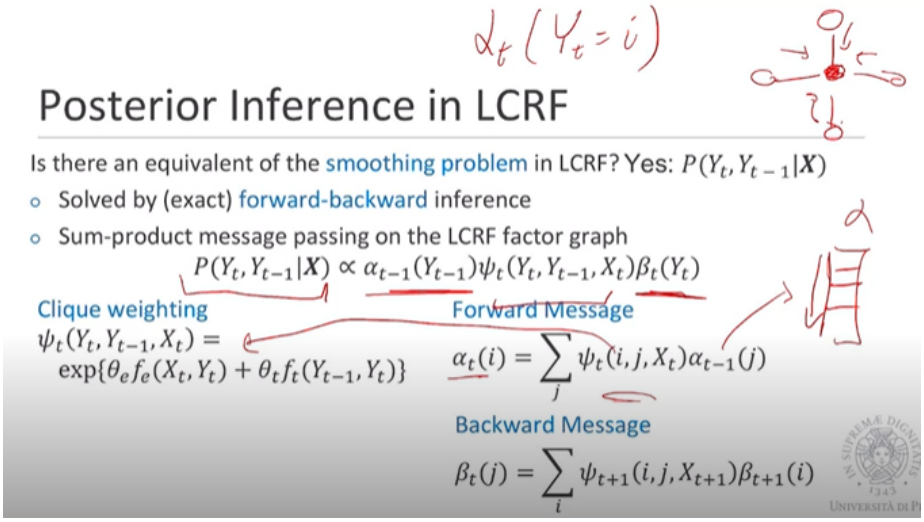


Figure 42:

Learning: in CRF we do again maximization of a conditional probability, find parameters theta, that maximize the conditional likelihood, conditional likelihood, not a generative one, because Xn̂ is on the conditioning side, the problem is the usual, find the params theta that maximize that conditional probability that factorizes according to the MRF assumptions, theta params. are inside because when you factorize a MRF is an exponential of a linear combination of features functions where theta are the params of the linear combination.
We use the logarithm, the logarithm pass through the multiplication, over all the sequence in the dataset and tranform it in a summation.
Log(1/Z(X)) = - Log(Z(X)) and logaritm of an exponential cancels out.
That's way you get a nice formulation for the log-likelihood, the first term is a simple linear combination of the feature functions, to maximixe that function i take the derivative. dL(theta)/dtheta_l where l is one specific index of theta.
The derivative pass thorugh the sum over n and t, Sum over k the only one interest for the derivation is for k=l, when you derive theta_l * f_l remain only f_l, cause it's a linear function of theta.
$\frac{\partial \mathcal{L}(\theta)}{\partial \theta_k} = \sum_{n,t} f_k(Y_t^n, Y_{t-1}^n, X_t^n) - \sum_{n,t} \sum_{y,y'} f_k(y, y', X_t^n) P(y, y' \mid X^n) - \frac{\theta_k}{\sigma^2}$
→The second term is an expectation of the expected value of the feature function under the model distribution. $P(Yt, Yt+1|X^n) \rightarrow$ distribution that tells how much a specific couple of y and y' is likely/probable. Estimating the value of the feature function, doing a weighted average of the values of the feature function when the configuration of y and y' is weighed by how probable they are according to the model distribution.
The first term is an expectation w.r.t. the empirical distribution, the distribution of the dataset, you're summing over the dataset, you're taking values from the dataset, the Ys are from the dataset, the labels/target during training: an

expectation w.r.t. data distribution, expectation with y and y' that comes from data. When I'm taking the derivative i look for stationary points, where the gradient is equal to 0, what you really do is trying to 0 the difference between those two expectations. Can you change the expectation w.r.t to the dataset? No, no way to change the data generating distribution, but we have way to change the model based observations. When you're training you're pushing the model distribution p(y,y'—X) to be non distinguishable in expectation by the data generating distribution. They have to match the single predictions?NO, in expectation, in average values. General property of MLE, operating on exponential family distribution, e.g. Gaussian. Whenever you assume that your model is somehow following the exponential distribution, if you do MLE learning, from a theoretical perspective you are pushing your model belief to be consistent with your data in expectation. LEARNING MUST WORK, does not depart in expectation from reality.

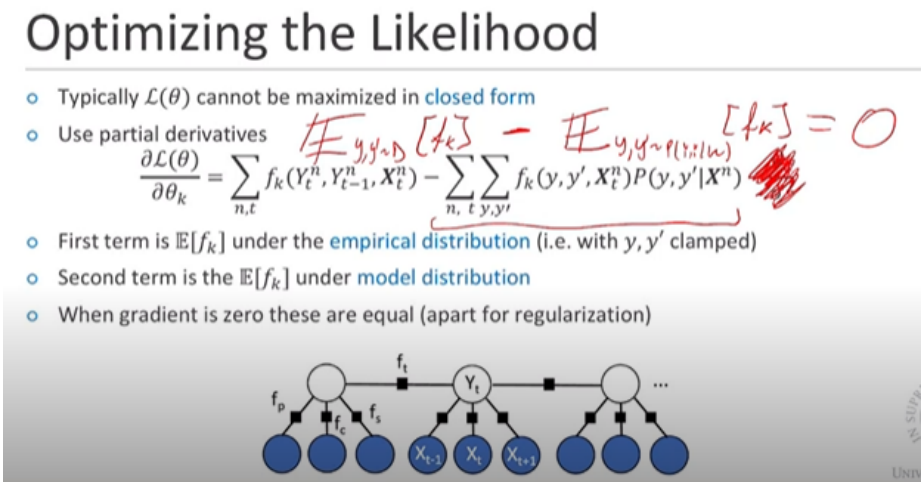


Figure 43:

Add regularization to MLE learning, add a penalization term with the sum of the squares of your parameter (hyperparam. lambda = $1/(2sigma^2)$). Probabilistic interpretation, you're solving a maximization problem on the likelihood * prior, so it's posterior maximization, you need to have a prior distribution of theta, params are vectors on continuous space, a good distribution is a gaussian with zero center and fixed variance sigma square.
Maximum a posteriori is regularization of MLE.
When i take the derivative the penalty term alters the results shown before, the prior creates a perturbation in MLE that pushes the model towards the data generating distribution, model less attached to the data. Good the model less attached to the data.
In practice you typically learn this model by SGD.
Classical example in vision: Background removal tool in office its implemented

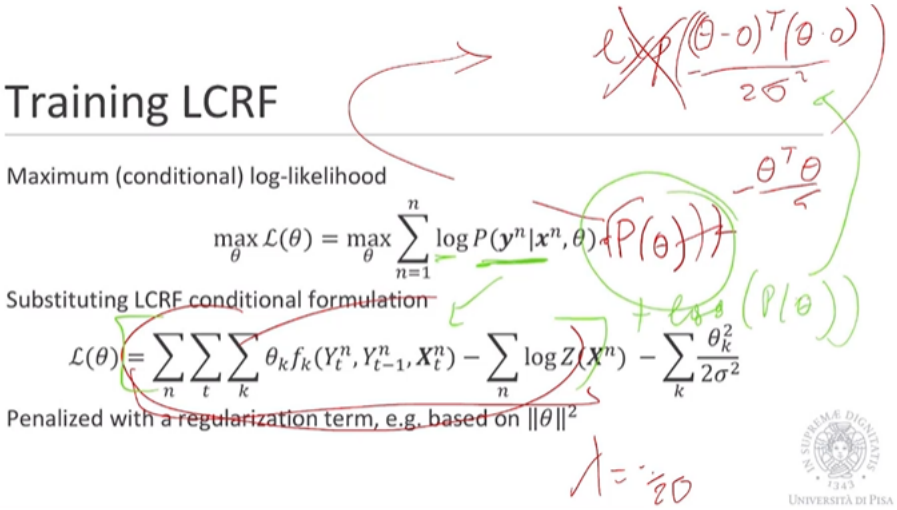


Figure 44:

with conditional random field, problem of finding the hidden Y, 0 whenever its background and 1 whenever its foreground, the observables are the pixels values X.
The feature function fs relates specific colors with the likelihood of being a background/foreground: black/blue (sky), green(grass) is likely to be in the background. fh the one that connects two nearby hidden rep. of pixels is an affinity function, if my neighbors is foreground probably i should be foreground as well.
Belief propagation I plug some observations and assign these values to only some of the nn observables and propagates them to estimate the posterior.

12 Bayesian Learning and Variational Inference

Approximation of our inferential process, we will use a specific approximation tool → variational inference and we are gonna be using it to solve learning into a specific model Latent Dirichlet Allocation (LDA), then we're gonna introduce another tool → sampling.
Variational Inference is a general tool to learn in the presence of approximated posterior, what happens in Generative deep learning, when the tool to approximate a distribution is a Neural Network. Models such that Variational Autoencoders, diffusion models fit in this framework.
Concept of Evidence/variational Lower BOund (ELBO) gives in a straightforward way a different objective function for our learning process, an approximated one but that we can tackle with learning.
Latent variable models: HMM are latent variable models cause there are non-

observable random variables, that we introduce for the purpose of simplifying the relations between the observable ones.
When we generalize latent variables are no longer discrete, they might be continuous, in that case there are integrals that bind together pieces of your distribution, things are more difficult to treat computationally.
Simplest non trivial model in this class is the Latent Dirichlet Allocation: which is a latent variable model which uses concept from Bayesian Learning (relaxing the idea that our random variables are simple realization of events, rather my random variables become something that can describe entire distributions). In our Bayesian Network, some random variables, typically the non-observed ones, whose values are distributions, they take values and those values are istantions of distributions. Latent Dirichlet Allocation (LDA) looks like a multinomial mixture model, where the params of the multinomial distribution are sampled from a random variable rather than being learned.
Whenever your random variables can become a distribution, what happens is that instead of learning the params of distribution, you are gonna be sampling them from a random variable which has itself another distribution.
Latent variable model: in my model there are at least some non-observed random variables, for which i cannot have ground truth.
We introduce latent variables cause expressing relationships between the observable ones could be difficult. X=[X1,...,XN] multiple features with complex interrelationships, we cannot model direct the probability of X, too complex, we introduce unobserved random variable Z, replication for all N input features. Factorize the joint distribution of the X, with my N features, thanks to the introduction of my Z latent variables, can be factorized into a product of each of the single Xi but only when i can observe/put in the conditioning side Z, but Z is latent, its not observable by definition, the only way to make Z appears is marginalizing out, introducing them by considering all the possible way with I can pick up Z. Integral or summation if Z is discrete. Generally I cannot make any assumptions about the nature of the joint distribution of by observables, but typically when i factorize this model the probabilities that are involved are typically from the exponential family, they're reasonably treatable, I know what happens when i multiply one with another one, the multiplication of two distributions is not necessarily a distribution, multiplication is not close out of specific assumption, in exponential families I've rules to compose probabilities. So the joint of the observable is quite untractable, but whenever I introduce Z I can make reasonable assumptions about the distribution and I know how to deal with.
M samples in your data, original input space X is N-dim.
When I introduce a latent variables model, there exist a Latent space in the middle, that allows me to represent my samples in a lower dim representation, latent space is where my Z random variables take values, e.g. latent space with a single random values that takes 3 distinct discrete values, latent variables act as mediator to project/represent high dim data in lower dim data. Imagine one of your sample of your dataset, originally existed in a N-dim space I say he can also be represented in a smaller dim space given by my Z repr. that can

associate to the sample.

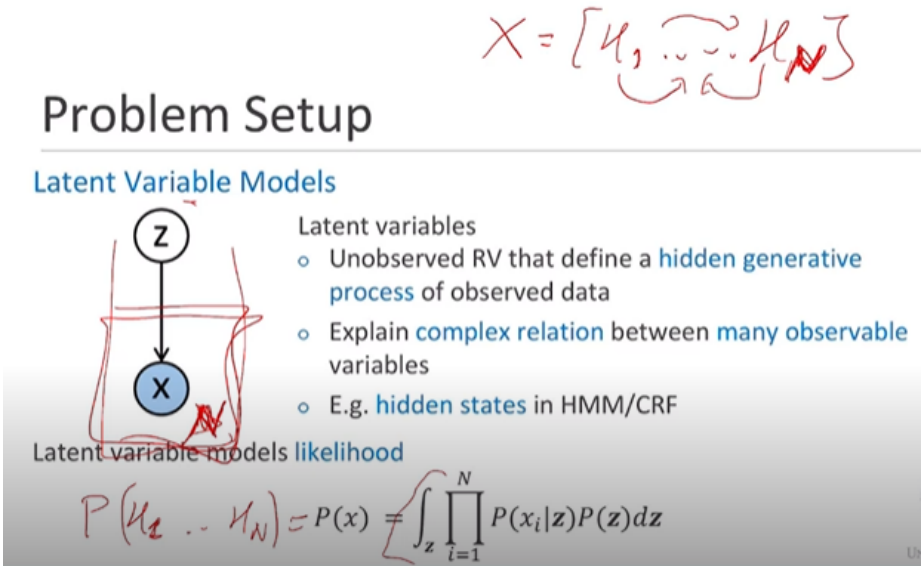


Figure 45:

Now it's more tractable because I can name the distributions involved, but I have integral, I need to marginalize out over the non available Z, makes it the estimation of the posterior is not tractable. **Posterior**: $P(Z|X)$, probability of something you cannot observe given evidence. I need to make the posterior tractable because it is used everywhere, posterior inference is needed for learning for sure at some point in order to learn the param of my model, the update rule is expressed in term of the posterior of the model, i need a way to estimate it, even for prediction most of the time i need to go thorugh the posterior. Need of posterior approximation.
Calculus of variation to approximate the posterior, allows to maximize a functional (a function that I am optimizing that itself includes inside of it another function). I'm gonna be moving a function that changes another function.
We want to approximate the posterior, the posterior is a probability, i need to measure how close I'm to the real posterior, I need something that measures the discrepancy between distributions. **KL Divergence**:
$\mathrm{KL}(q\|p) = \mathbb{E}_q\left[\log q(z) - \log p(z \mid x)\right]$
Expectation w.r.t to q, $Kl(q||p)! = Kl(p||q)$, it is not symmetrical. Both q and p are defined over Z, the expectation is over Z taken from Q(Z) of a function f which is the logarithm of $q(z)/p(z|x)$, In the general case on the denominator you can have whatever distribution over z, since we're interesting in approximating the posterior i write the KL-divergence already specialized with the posterior. Use the probability of the log of the division and the expectation is

a linear operator that goes through sum and difference.
KL-div is the difference between two expectation, the log of q w.r.t. to q and the expectation of the log. of our posterior w.r.t. to q. Measure the discrepancy between this two distributions in expectation.
Whenever q is high, in the areas in which q is high, also p has to be high, otherwise the first term is gonna be high and the second term is gonna be low, high - low = high. KL-div, will push for distributions in which whenever q is high then p is high. q high and p high low KL-div $\rightarrow$ happy, q high and p low, high KL-div $\rightarrow$ unhappy , when q is low $\rightarrow$ I do not care, because if q is low, everything is multiplied by q here, those are expectations w.r.t. to q, it will zero whatever happens both on the first term and on the second term, those parts of the q distr. where q is low do not contribute to the KL divergence substantially, irrespectively of what Z does. That's why it's asymmetric, there is one distribution that pushes more than the other the values of the KL due to the expectation.
Jensen inequality: the linear combination of f apply to your 2 points is lower bounded (is greater or equal than) by combining two points and then applying f. This is for convex functions. Concave is flipped, you flip the directionality of the inequality. This generalizes to linear combination of whatever number of points you take.
The expectation is a linear operator, that actually does convex combination using as a weighing term the probabilities.
This thing is gonna be applying in our specific case in which the linear combination is given by the expectation, the f function is the logarithm (concave function).
Why relevant? Due to the integrals we're introducing for latent variables models we often find ourselves dealing with logarithm of an integral of a probability, very bad because you cannot separate the terms in the integral because of the logarithm before, logarithm works nicely with the product not with the summation. The Jensen inequality says that if we're ready to lose something (it's a lower bound) we can bring the logarithm inside of the integral, but we're losing something and we're gonna be seeing how much we are losing.
We're dealing with the problem of maximizing the log-likelihood of our model. $logP(x|theta)$ $\rightarrow$ distribution that describes our model, contains the model param theta, we would like to maximize it. In the original setting is not tractable, it becomes so if I introduce my latent variables through marginalization $\rightarrow$ factorize in something simpler. The objective is to do approximation, in the end I'm gonna be approximating the posterior, we've seen the KL div. that there is this magic q function that i use to match my posterior, I'm gonna be introducing a q function (variational function). Multiply and divide by the variational function, reasonable in general, aside when that q function is zero. Q(z) is not just any function but a parametrized function, something i have control on, which I can tweak, which I can learn, I introduce a $Q(z|phi)$ that depends on params ϕ. 2 set of params: theta $\rightarrow$ params associated with the model, inside the model distribution, ϕ are the variational params, introduced for the purpose of the approximation. 2 distributions separately parametrized, I

need to be smart in the choice of Q, simple enough so that its params are gonna be some sort of disentangled, not entangled by all the marginalization operation, so i can optimize them more easily than whatever i would be doing estimating a posterior. Q is simple enough and has params i can tweak to approximate my posterior.
Appears the definition of expectation computed w.r.t. to Q, Q is the probability $Q(z|phi)$ is a f(z), (the values of z are taken from the Q distribution. A gaussian is a function of x but also the probability of x. The values that you observe of your Z behave according to Q, the freq. with which you observe stuff, depends on the distribution that generates it. Magic distribution q, which exists and has its own params, it can generates my Z, i can take the expectation w.r.t. to that. With all the manipulations the initial log-likelihood is rewritten as a logarithm of an expectation of those two terms taken w.r.t. to parametric variational distribution Q.

$$\begin{aligned}
\log P(x \mid \theta) &= \log \mathbb{E}_{Q(z)} \left[\frac{P(x,z)}{Q(z)} \right] \\
&\geq \mathbb{E}_{Q(z)} \left[\log \frac{P(x,z)}{Q(z)} \right] \quad \text{(by Jensen's inequality)} \\
&= \mathbb{E}_{Q(z)}[\log P(x,z)] - \mathbb{E}_{Q(z)}[\log Q(z)] \\
&= \mathcal{L}(x, \theta, \phi)
\end{aligned}$$

We've started from the logarithm of the observables, the likelihood, that I want to maximize, but I do not know how to maximize, because I don't know that distribution, I need to rewrite in terms of other distributions that includes Z, through manipulations I found a way to write a lower bound of that term made of two terms which separates the parameters. Whenever I'm gonna be maximizing w.r.t. to θ only the first term depends on θ, when I'm gonna be maximizing this term w.r.t. to ϕ only the second depends on ϕ, I can maximize them independently.
I can even maximize them into different steps, the part number 1 during the maximization step of the EM so the EM step is gonna be operating only on the part number 1, the expectation step is gonna be giving me the params ϕ, of the variational distribution that allow me to estimate this overall thing. EM is something that arises from the fact that when I'm doing EM I first find params ϕ of a variational distribution which I then use to maximize the term number 1. Because i need q to compute that.
The two sets of params are nicely isolated into different things which allow me to do this alternated optimization in EM, term number 1 includes the logarithm of the joint of the visible and latent variables, which is good because I can factorize it as a product. $\prod_i P(X_i \mid Z)\, P(Z)$
This now is nice, before it was very bad, because it was the product factorized inside of an integral with the logarithm outside, now I still have the integral hiding in the Expectation but I have the logarithm first, apply the logarithm to that big list of multiplication, transform them into summation, that go out also

of that expectation (linear operator). By this lower bound I managed to obtain that I can manipulate my original intractable log likelihood, and rewrite with a different term in which the logarithm linearizes all the terms that are involved and so everything I'm gonna be doing here is gonna be essentially linear combination/operation, and the logarithm will directly apply to probabilities P(Xi|Z) and P(Z) which are of the exponential family, logarithm of an exponential, the two disappear and we only left with the argument of the exponential. All these manipulations will make the all things tractable at the cost of solving an approximated problem, cause it is a lower bound.
Evidence/variational Lower BOund (ELBO), is a function that is lower bounding my likelihood, in a maximization setting, if I want to maximize my likelihood, I can as well try to maximize my lower bound, it can't happen that if i maximize the lower bound i minimize the original function, of course I end up with a sub optimal result. Need to asses how good is this lower bound.
However this lower bound simplify a lot my optimization problem, everything becomes a summation.
Interpretation: Term 1 is an energy term, optimized whenever the model distribution matches the empirical distribution, a property of all exponential families models whenever we're learning by maximum likelihood, the first term is a likelihood, is assuming that P(Z) is somehow joint with X. Optimization of the first term will match my target whenever the model distribution matches the empirical distribution, the empirical distribution: is the distribution that places a sort of Dirac function, a 1 over my examples from the dataset, very dumb distribution, 0 everywhere except in one point which is one, i take my model distribution which is in principle some nicer, smoother function and asking my model distribution to behave deterministically, like a Dirac, the Dirac is fully deterministic distribution, zero stochasticity cause it's one exactly for one value and zero everywhere, the first term makes my model behaves as the original data, which is what I want to obtain in a proper ML model, but has also another unpleasant effect, it pushes the model to become a Dirac of my trainig set, a more mathematical involved way to say that my model become a look-up table, because it stores all the data. The first term, encourages learning but also memorization, encourages the model distribution to behave like a fully deterministic Dirac.
The second term is the entropy, a measure of disorder in my distribution, high entropy $\rightarrow$ my distribution is very unpredictable. The Dirac is perfectly predictable, it's deterministic. Fully deterministic distribution $\rightarrow$ 0 entropy, fully stochastic very unpredictable distribution is e.g. the uniform distribution in [0,1] $\rightarrow$ maximum entropy.
The first term is satisfied whenever my model possibly becomes the Dirac of the data, the second term privileges something unpredictable (unpredictability term) pushes the model to be less deterministic. The first term does learning and overfitting the second make sure that you don't overfit, entropy term that pushes your model to be a bit more stochastic, naturally regularize. In physical intepretation first term is Energy term second term Entropy term.
ELBO, evidence lower bound, to my likelihood L(X,theta,phi): z is marginal-

ized is not an input to the model, it's a lower bound it can be different from the likelihood, how different? how good it is ?
We begin with the difference between the log marginal likelihood and the Evidence Lower Bound (ELBO):

$$\log P(x \mid \theta) - \mathcal{L}(x, \theta, \phi)$$

Recall the ELBO is defined as:

$$\mathcal{L}(x, \theta, \phi) = \mathbb{E}_{Q(z)} \left[\log P(x, z) - \log Q(z) \right]$$

Substituting this into the expression gives:

$$\log P(x \mid \theta) - \mathbb{E}_{Q(z)} \left[\log P(x, z) - \log Q(z) \right]$$

We rewrite the expectation as an integral:

$$= \int Q(z) \log \left(\frac{P(x \mid \theta)}{P(x, z)/Q(z)} \right) dz = \int Q(z) \log \left(\frac{Q(z) \cdot P(x \mid \theta)}{P(x, z)} \right) dz$$

Now, applying Bayes' Rule: $P(z \mid x) = \frac{P(x,z)}{P(x)}$, we find that:

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

Substituting this in:

$$= \int Q(z) \log \left(\frac{Q(z)}{P(z \mid x)} \right) dz = \mathrm{KL}(Q(z; \phi) \,\|\, P(z \mid x; \theta))$$

Therefore, the difference between the log-evidence and the ELBO is the KL divergence between the variational distribution and the true posterior.

Maximizing my expectation lower bound will be a close approximation to maximizing the likelihood, if the variational distribution that I introduce Q behaves like the posterior P(Z|X), and would be an exact maximization whenever Q is exactly P, the posterior. Which is exactly what happens with HMM, when optimizing the HMM we're actually computing the posterior, my Q function is actually the posterior, Kl-div is 0, so whatever EM I'm doing is an actual exact maximization cause I'm using the posterior, if I'm not using the posterior the approximation is as close as the difference between q and the posterior. Whenever I'm learning either i focus on maximizing the ELBO or i make sure that the Q function matches the posterior by minimizing the KL.
The first is a maximization problem, the second is a sampling problem cause there is an expectation involved, expectation can be approximated by sampling, if sampling it's easy enough i can approximate the KL divergence using sampling, and solve my learning problem using sampling, if it's too difficult to sample I can choose the Expectation lower bound way and this solve a maximization problem. Two exchangeable ways, of computing the posterior or find the params of

my approximated posterior which are totally interchangeable. The likelihood is the lower bound + the KL, either minimizing the KL or maximizing the lower bound are identical.
$X \sim P(X \mid \theta)$: when you ask Dalli to generate an image, you're drawing a sample X which is an image from a distribution of the Xs, you need a way to fit a neural network to that likelihood which is a nightmare if you try to do it directly, you hypothesize that your NN is made of latent variables, all the latent representations that you can find in the hidden neurons and you use the principle of variational approximations to find a way to approximate this thing with the lower bound we just derive.
Provided that I can write a Q, variational function, approximating function with its own params, simple enough (i have to either run the KL expectation on it or maximizing it w.r.t. to its params) but then I know that i have this ELBO available to find a simpler maximization problem to the problem of optimizing my log-likelihood.
EM reformulation:
$\max_{\theta,\phi} \sum_{n=1}^{N} \mathcal{L}(x_n, \theta, \phi)$
problem of maximizing w.r.t. to θ (model params.) and ϕ (params. of the variational approximation q) of the sum over the samples of the variational lower bound. Can be an exact maximization of the loglikelihood if my q function is exactly the model posterior, otherwise it is an approximation as good as the difference between my model posterior and the variational approximation.
The structure of the lower bound tells me that operations on the model params can be performed separately from the operations on the phi params of the variational distribution, I can fix the variational param ϕ and optimize w.r.t. to the model param θ and viceversa. EM step, when i fix ϕ and optimize θ is maximization step, when I fix θ and optimize ϕ is the expectation step: You're estimating a Q function that matches in expectation the posterior.
When you use SGD on this objective you lose the guarantees of finding maximum but still increasing.
Solving this maximization problem by any optimization mechanism including SGD which allows to introduce NN. Must be smart in picking up q, that factorizes in a form where params get nicely isolated. Because everything pass through the logarithm, that multiplication become a summation, all diff params. separated in diff terms, when maximizing a lot of zeros and only one term that survives.

Figure 46:

To solve EM: if I can compute posterior in close form use it, it's the best solution you can find. If it's not, choose a Q that is tractable, and either find the params

ϕ that minimize KL divergence, which entails sampling or perfectly equivalent find ϕ params that maximize the ELBO. When you find ϕ, fix them to optimize the ELBO w.r.t. to θ. Once you have the θ, fix them, pass them either to the KL minimization or the ELBO maximization and find the new params ϕ and so on.
Latent variables model: specific model used to do modeling of documents (text, images, video) by leveraging the fact that modeling the joint distribution of pixels is too difficult but easier if i model the marginal distribution of the single pixel given some latent variable Z. Bag of Word or equivalents for images and videos, is the high dim rep. of your document, I want to find a smaller rep. of my document that instead of being represented by a vector sized the number of word in the Vocabulary is represented by a vector sized the number of possible topics. Model a document as a mixture of topics, each topic is a random variable 0/1 whether is present or not, or single multinomial variable that can take as possible values as the number of values of the topic inside of the documents. Document as mixtures of latent variables (Topic), the occurence of actual words in the document is connected, I do not know the direct link , I hypothesize that exists a latent topic that explains why some word are occurring in a document. For each document BoW takes the words and color them according to the latent topic they are associated with. From a set of words occurring to a set of topics occurring. Clustering the words in topic cluster. I don't know how to partition them, because I do not have the ground truth, it is automatically inferred from data. Unsupervised model that learns how to cluster information and to explain documents in terms of these mixtures of latent variables.
Latent Dirichlet Allocation (LDA) - Bayesian Network replicated in plate notation to make compact the replication of dependencies, N words in my vocabulary assign to each of them a different meaning/latent topic, once for each of the M document. M docs, each composed of N words, words are the only observables here, I look to the words and try to assign a meaning to the words according to the distribution of the words, specified by the model: $P(w|z, \beta)$.
z is the topic(meaning) associated to the word, β is the params of the model (black seed), words are multinomially distributed, β is the table (N x K) with the probability of the topic for each word, when I fix a topic I can then assign the words according to the probability of a word to be part of a topic. I'm gonna be learning β. For each word I assign a latent meaning by this assignment z, the word is generated by this latent meaning. You can consider topic(meaning) to be clusters. z is a multinomial as well as w. z is the latent variable that defines the meaning of each word, coming from θ. z is a multinomial, can be taken in 1 to K, the different meanings. P(Z) is a vector with the probability of each topic. In this way i take the assumption that all topics assume the same proportion of meanings, I would like to generate different proportions of meaning for each document $\rightarrow$ job of θ. θ generates the proportion of meaning for the specific documents, the values of P(Z) distributions, the values of the multinomial, θ is a random variable that is actually a distribution because it is generating the value of P(Z) which is a multinomial. θ generates vector with k-dim with each elem of the vector can be an instance of the multinomial probability. θ is a

random variable that generates continuously valued vector, k-dim summing to 1. θ has to be drawn from a distribution $P(\theta|\alpha)$ which is a distribution capable of generating this vector size k that sum to 1. $\rightarrow \theta$ is a Dirichlet distributed random variable.
When we start working with interest enough learning models, my random variables first of all, they do not need to be all observables, but my random variables can also be something as complex as generating sampling themselves, or distribution, not just instances of discrete processes. They can generate distribution, you can sample distribution.
Here I use this trick to generate for each document, a personalized, randomly drawn, from a Dirichlet, proportion of topics.
This model here is telling you that whenever you're learning a model, some of the distribution in your model can be learned, e.g. β params, other can be sampled from an appropriate distribution. Different way of looking at learning as the problem of sampling from an appropiate distribution, the params of your model are sampled from a distribution, instead of being fixed and learning.
α is an hyperparameter for the time being, it's a concentration parameter that regulates the behavior of the Dirichlet in terms of the vector it provides to you. Probability $p(theta|alpha)$, when we write the likelihood is gonna be multiplied by ¶$(z|theta)$ and $P(w|z,\beta)$, the last two are two multinomials, when you multiply two multinomials you obtain a multinomial. When you multiply a multinomial by another distribution, it doesn't always make sense, it has sense when the two distributions are conjugate and Dirichlet is the conjugate of the multinomial and their product is a Dirichlet again. With Dirichlet distribution we have a method to sample the params. of the multinomial and whenever we're gonna be writing the joint distribution this gonna make sense and we know what distribution we get. In the formula of the Dirichlet you can see the k θs you sample and α hyperparameter, that regulates the concentration of the vector that I can pull out from my Dirichlet. [1,0,0,...,0] very sparse vector or very uniform vector [1/k,1/k,...1/k] very dense, or you can generate everything that is in between, choice of alpha that are big $\rightarrow$ very sparse vector, low values of alpha 0.1 e.g. your vector will be very dense/uniform. Alpha = 1, generates with same prob dense and sparse vectors. Sparse vectors and uniform vectors (uninformative priors). Alpha introduces in your model prior knowledge about what you would expect from the distribution of your data.
Create vectors with distribution of topics inside of documents very picked on 1,2 topics only. Or alpha 0.001 very general document, relative proportion of topic within a document, quite uniform. Alpha regolates this trade off, use prior info. Can be seen also as regularization. Dirichlet distribution used also in NN activations to regulate the sparsity of the activations of neurons.
Observations as we receive them from the world may be high dimensional, but meaningful observations/data doesn't really usually live in this high dimensional space, but in a smaller mathematical structure, a manifold which defines how observations can be changed and still remains something semantically consistent. In the big word simplex, plenty of words, but only some specific configuration of words will make sense, with the topic simplex i'm trying to capture that

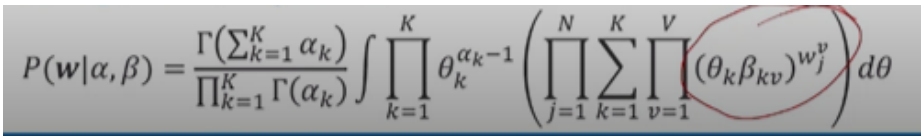
sub-space, part of configuration of the word space, smaller than the full area where meaningful documents live and it is lower dimensional. Here is discrete with multinomials, in NN with the activations of the neurons in hidden layers but similar concept. A document can be expressed as a combination of words in the words simplex or as a combination of topic in the topics/latent simplex. Alpha big implies topic simplex populated in the neighborhood vertices, sparse theta vectors, alpha very small very dense theta vectors, every topic as more or less the same probabilities, representation of document in the center of the simplex, with non-informative prior I'm asking to fill with more or less the same probability all the topic space.
Whenever we're taking a text, imagine to have my Latent Dirichlet Allocation trained, by means of θ, given a document, I'm going to generate an histogram with θ k values, the proportion of topic/meaning that i'm gonna be using for this specific document, for the document below i will have another set of param. Start going through each word in a vocabulary and assign each of those words a meaning, taking into consideration that I expect that a certain proportion of word will have a certain meaning, another topic will be present in another proportion of words, the words are colored based on their meaning. I can do more than that, I can have a global assesment of what is the proportion of certain word in certain topic. β is a multinomial distribution that assigns words to topic, what I can do is given a topic, given a column vector, a vertical slice of that matrix i can order the words from the most probable to the lowest probable. If i take the top 3 word for each topic i can figure out what this topic is about, topics are not supervised, they are attached to a latent variable, no supervision, they are cluster, inspect the matrix to give interpretation. Look to the most probable words for each topic that i can extract from the learned beta params. There is another distribution that puts into communication words and topic, it's the posterior, than assign topic to words, the posterior is computed on observed data: $P(Z, \theta|w, \beta, \alpha)$. Z non observable, words are observable, there is another random variable that is latent here θ. The full posterior will give me this assignment of topic specialized for the specific word for the specific document. Two dimensions of probabilities here, the global one at dataset level, that does emerge after i fitted the model params and the personalized one or local one that are the posterior that I obtain when I compute the probability of the non observable things, given a specific observation, which is a full document.
Wrap up of what are the distributions involved in this model and their params: this models here are described in terms of what is the process that generates the observables, generative view of your data: given a document the way in which you generate the observations in your document, which are the words' proportions, is that you first sample for a document what is your expected proportion of meanings/topics inside it. Then using that, you pick up a topic with a certain probability that depends on the overall topics distribution of the specific document and then given that meaning, once you have chosen that meaning, you decide to concretize that meaning in one of the N possible words, according to the distribution of the words given the topic that you have in β.
β is matrix K x N, only params matrix we are gonna be considering, multi-

nomial matrix, it's in one-to-one relationships with the probability of the j-th word equal to one while that the k-th topic is equal to one.
Let's look at the factorize distribution: ¶$(\theta, z, w|\alpha, \beta)$ is the complete likelihood of the model, it contains both the observables and the latent variables, if I can observe also the latent variables, i.e. I have θ and Z along with w, I can decompose this big joint distribution into its factorized version by following the conditional independence relationship I have in the model.
If i observe α, α totally determines θ, so I can take θ out of there and i have $P(\theta|\alpha)$. Once that I've observed θ, Z is completely determined by θ so that's why i have $P(z_j|\theta)$ where z_j is the meaning of the j-th word. Once I have observed Z the words are independent, once I observe z_j and β of course, β is the parameter set.
That will allow me to factorize this joint distribution w_{bold} (full document, all the words) in a product of simpler distribution, one element for each word in the document. Simply factorize in three distributions: the first is a Dirichlet, the second and the third are multinomial. I can plugin the equation of those distributions when will be the time of doing some learning.
Problem: Z and θ are not observables, the actual likelihood is the probability of the observables, How do I get a θ and Z in here? By marginalization, the usual sum overall Z, but θ is continuous so integral over the theta, sum over Z of the complete likelihood.
The problem of learning: problem of given a dataset of M documents find α and β or only β with α hyperparameter.
Find those that maximize the likelihood, maximize the logarithm of the likelihood, assumption of i.i.d between each sample of the dataset so that the probability of the dataset factorizes in the probability of each document.
There are latent variables so I'm gonna be using EM, in order to do learning I need to be able to deal with the posterior, i need the posterior to update my params.

$$P(\theta, \mathbf{z}|\mathbf{w}, \alpha, \beta) = \frac{P(\theta, \mathbf{z}, \mathbf{w}|\alpha, \beta)}{P(\mathbf{w}|\alpha, \beta)}$$

Figure 47: Posterior of theta and Z given the observable

When I plug in the definition of the distributions that are involved in P(w) what you get:
The problem is circled in red: that thing is coupling two terms which is gonna be making computation of this thing, in analytical term, impossible, and this term is at the denominator of your posterior so this thing won't allow me to compute the posterior in analytical form. EM, at least in its exact form is gonna fail, **need a way to approximate my posterior**. How? We need to find another function q, which can approximate the posterior, but it'is enough



$$P(\boldsymbol{w}|\alpha,\beta) = \frac{\Gamma\left(\sum_{k=1}^{K}\alpha_k\right)}{\prod_{k=1}^{K}\Gamma(\alpha_k)} \int \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \left(\prod_{j=1}^{N} \sum_{k=1}^{K} \prod_{v=1}^{V} (\theta_k \beta_{kv})^{w_j^v} \right) d\theta$$

Figure 48:

simplified that i can compute it decently in closed form.
Whatever function I'm gonna be taking, it needs to remove this coupling. I'm gonna be applying variational approximation with a q function that is simpler than this.
Two ways to approach this problem by now: **Variational Inference**, I'm gonna be looking for this q function that allows me to compute the posterior, inside an optimization problem, gonna be writing a q function that is simple enough, and then gonna be optimizing the ELBO. When I have the ELBO i have both the params of the model and the params of the variational distribution q: so the optimization problem that I'm gonna be solving here, we'll be a **two 2 stage optimization** (Expectation and maximization), one stage will estimate the params of the q function, which is the distribution that simplifies the posterior, second stage I'm gonna be updating the params of the model (LDA) such as the βs.
In one step I'm gonna be updating the params ϕ of my variational distribution Q, in the other step I'm gonna be updating the params β of my model.
You can instead take a sampling approach, in which rather than building an approximating function q, you construct a sampling process that can generate samples from the posterior without using the posterior that is too complex, you need a sampling process that you know does not look very different from the posterior if you go to infinity. To the limit it will produce stuff that looks like stuff that you produce from the posterior but that comes from a more samplable thing. You'll generate samples from this sampler and you'll use to approximate with empirical data the KL div that contains an expectation.
The first one is a full approximation, you 're simply saying I don't konw how to compute the posterior, I need to simplify, you're cutting the posterior, you get an approximation and you leave with the approximation that you have, no way to make it better, it remains an approximation but is fast!
Second one: if you need to make your approximation close to reality, Law of the large number just tell you just need to sample more data: more accurate approximation more data, at the cost of time and energy.
Variational Inference; my problem is that posterior, in which things are coupled, I need a q function in which things are not coupled which is essentially building on the concept inherited from statistical physics of mean-field approximation, you approximate a multi body system with a single body. **Here you approximate a joint system of multiple interacting random variables, with a product of non interacting random variables, non interacting params.** Not capturing all the correlations that full posterior captures, loses complexity.

$$Q(\boldsymbol{z}|\boldsymbol{\phi}) = Q(z_1, \ldots, z_K|\boldsymbol{\phi}) = \prod_{k=1}^{K} Q(z_k|\boldsymbol{\phi}_k)$$

Figure 49:

My Q function, defined over Z, here Z is the generic set of all the latent variables, this is an approximation of all the random variables involved in the posterior, I factorize entirely the posterior, each latent variable is considered independent of the others one. Each random variable is independent one another, not only that but parameters factorize, this is the key thing. Instead of having a big joint distribution with mixed params across all the random variables, I have k different indep. distributions, each one with its own tiny little set of params specific for that random variable.
Given that now we have a factorizable distribution, we can now apply the ELBO principle, plug q in the ELBO equation and $P(theta, z, w|alpha, beta)$ into the other equation, the last will be plugged in the expectation over the joint, which is the energy term of the ELBO, so the expectation over this thing will be the first ELBO term, and the second term will be the expectation over the Q, the entropy term
At the left the LDA perfectly nice in all its dependencies, on the right the variational distribution, you grasp even visually that they look different. You can visualize the factorization: that you pick up θ based on parameter γ, you pick up Z indep. N times based on params ϕ_k one for each Z indep. We're gonna be using the left one to learn the params and the right one to estimate the posterior essentially. Gonna be computing the expectation on the left one for the energy term, and the expectation over the right one for the entropy term. The things into play in the learning part are the β params of the model distribution learned separately from the γ and ϕ params of the variational distr. whose job is to approximate the posterior.
$\mathcal{L}(\mathbf{w}, \Phi, \beta) = \mathbb{E}_Q[\log P(\theta, \mathbf{z}, \mathbf{w} \mid \beta)] - \mathbb{E}_Q[\log Q(\theta, \mathbf{z} \mid \Phi)]$
EM: maximize ELBO with a 2-step iterative process: first generate at random my params of the model β and parameter of the variational distribution ϕ and γ. Then i start my repeated cycle, in the first step i fix β the model params and I update the variational params ϕ to obtain ϕ*. I will have found the variational param. that maximizes the ELBO or minimizes the KL div. Once i have those ϕ*, i will plug and fix them into the ELBO and update the model params β by maximizing the ELBO w.r.t. to β with inserted inside the current value of the variational params.
The lower bound allows the logarithm to play with the product and when the logarithm encounters the products its breaks down the products into separate sums. For the energy term the multiplications of the different distribution into

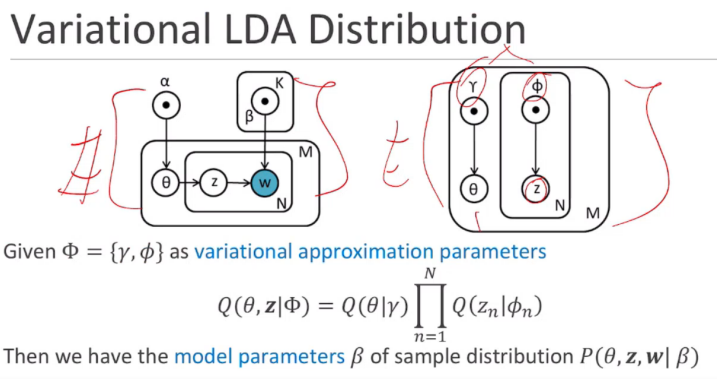


Figure 50:

the sum of different distributions. Why nice? When I'm gonna be optimizing this ELBO w.r.t. with the beta params a lot of zero, because i'm gonna be taking the derivative w.r.t. to β, all the rest is constant.
Something similar obtained for the variational distribution Q cause i've taken the simplifying assumption that things factorize, so that the distribution over θ separates from the distribution over Z and the distribution over Z can be separated in each of the different Z.
I've to derive the update equations: take that ELBO differentiates w.r.t. β in order to have the update rule for M-step, then i need to derive the update rule for the step number 2 (E-step): I can either choose (to maximize) the lower bound w.r.t. to the params ϕ and γ of the variational distribution, it's one possibility, in the original article the author choose the KL divergence way, they derive the update eq. by writing the KL-div between the variational distribution Q, the untractable posterior with all those integrals and find an approximated solution using Taylor.
Update rule for the variational params. obtained by KL minimization:

$$\gamma_i = \alpha_i + \sum_n \phi_{ni}$$

$$\phi_{ni} \propto \beta_{iw_n} \exp\left(\Psi(\gamma_i) - \Psi\left(\sum_{j=1}^{k} \gamma_j\right)\right)$$

Figure 51:

γ is connected with $P(\theta|\alpha)$, is connected with the posterior of the proportion of topics into a document, you know what takes your prior evaluation of how much you expect a specific i-th topic to be present in a document alpha, and

add evidential knowledge about the i-th topic by summing over all the words. That ϕ is essentially giving you a measure of how present is the i-th topic in each of the N words, you're summing over all the words and keeps your measure that tells you how much a topic is present evidentially, on the data. I add it to my prior assesment, which is the hyperparam. of the Dirichlet and I get a statistic of the strength of the i-th topic in the document.
The other variational param. is the proportion of the i-th topic in the n-word of the vocaboulary, it is proportional to the model parameter β and that is a model params that tells how much connected is the i-th topic to the n word according to the model, according to the knowledge of the model, a topic 'i' is associated with word 'n' with a certain strength that is captured by that param. and that is fixed because I'm in the step in which I fix the model params and use it to estimate the variational params ϕ. And this thing is modulated by the other term which is just the result of an expectation, an expectation of the prevalence of a given topic w.r.t. to all the other topics. It's a sort of a measure of a prevalence of the i-th topic when you compare with all the other topics. Adjusting my estimate by saying this topic is in expectation more present than all other topics, it will reinforce or dampen my current para. Psi is just the first derivative of the log Gamma, Gamma is this function we have in the Dirichlet. Unsupervised model, given a dataset of unstructured info it gives us some structure, tells us that documents are sort of organized into topics, topics are characterized by certain regularities/certain meanings and this can be used to organize collections of data, to cluster them. For textual docs but the same idea can be applied to images and videos. How can we apply latent topic analysis to visual documents? Need to represent images as Bag Of Words (BoW) (discrete elems from a vocabulary). Use localized descriptor, tells me what is the informative content of certain portion of the image, combinining visual detectors and visual descriptors I can get an image as represented as a collection of sift descriptors: good but they are not discrete!. How can I come up with a vocabulary, N images, extract the SIFT descriptor and run K-means to cluster the descriptors in e.g 500 cluster. Each Sift descriptor across all the images is assigned to one cluster. If I take an image I take the descriptors of those image, assign the cluster to each of them and transform an image into a vector of cluster counts.
Run LDA : LDA colors words with the topic assigned to each of them, now words are visual parts of an image. LDA has understood that all the visual words that pertain to the central part of the image, gets associated with the same meaning, cause they more or less cover the same object. Everything here is unsupervised, no label provided, they organize that, find structure into data. Nice it learns to identify which visual patches are of some color and what of another color, but there is a lot of holes in the middle, I want to color every single pixel, do I have a mechanism that allows me to propagate a labeling inferred by the LDA on nearby pixel to obtain an assignment of a topic to each of the pixel rather than to each of the visual patches $\rightarrow$ MRF, put Markov Random field on the top of a LDA, takes an image, label the visual patches with topics using LDA, then for those pixels that are overlapping the patches that have been labeled by LDA, on the top of them use MRF propagation to propagate

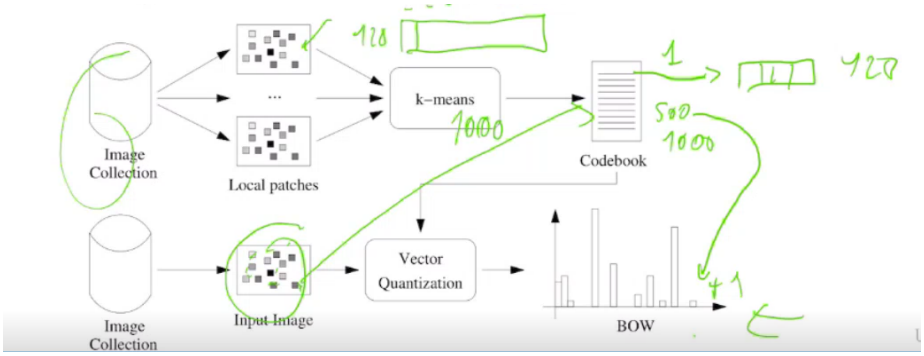


Figure 52:

info across all the pixels in the image with a classical grid-like MRF, and i get an automatic, completely unsupervised, segmentation of my image into semantic classes.
LDA and its variants has found application in modeling the evolution of documents across time, look into how, docs produced in different moments of history evolve, through LDA with a proper dynamical evolution, same thing with video, video are images that evolve in time. Time extension of LDA. Dynamic Topic Models: on the top we have an LDA, on the bottom part we have parameters that change over time, β_i are allowed to shift over time. Adding an arrow from the previous Z to the next Z, you can also denote the evolution in time of topics, how topics move from one to another in a different time step.
Once you have fitted your model, and your params. for each time slice, you can check how the leading words change through time, since these models are essentially unsupervised models that assign a discrete value to words, to the items of your document, essentially is a clustering tool. You can use them for instance, even on graphs, to do community detection, nodes in a graph can be seen as document, you can represent a node based on the neighbors of a node. You can run your LDA and cluster these nodes/documents into topic, and those topics are communities.
You can devise more articulated Latent Topic Models with Z articulated into layers, to identify hierarchical structure in topics (topic/subtopic).
In the LDA model Z takes values from 1 to k, the number of cluster/topic k is a parameter. Parametric models $\rightarrow$ needs of model selection. There are extensions of this Bayesian models: non-parametric Bayesian models that allow to get rid of this decision, to make it completely data driven, the number of topics present in a document collection is learned from data. Wrap up: variational inference $\rightarrow$ gives us a way to handle the problem of learning whenever the likelihood is not analytically tractable, it happens when there are couplings between the params of my model which I cannot decouple when i'm writing the integral and that requires approximating things. What typically requires to be approximated is the posterior, approximating the posterior by introducing

a variational distribution q, as similar as possible as the true posterior, this is achieved indifferently by the principle of maximization of the ELBO that is equivalent to the principle of minimization of the KL divergence between my variational distribution q and the true posterior.
Latent Dirichlet Allocation, simple model where this problem of the coupling arises which forces us to use variational approximation. How we solve the same problem we've seen in LDA, too complex posterior to estimate, but with a different approach that is sampling based.

13 Sampling Methods

Sampling is the process of creating realizations of random variables, coming up with a set X, made of my number of realizations of the random variables.
My process is a dice, i want to sample a dice: i roll it again, again and again and you observe realizations, what happens when you run the process. To analyze property of a process either I work analytically on an analytical definition of my process or I try to find way of measuring that property on realizations that I have and then estimate the property on average.
Most of the time we face problems that have to do with approximating expectation, compute the expectation of f(x), a function of my process, applied to data of a stochastic process, according to a well known distribution that defines exactly the process, if you can write analytically that equation, you are done, fine! It's a summation or an integral of p(x) * f(x), if you can close it analytically, ok but most of the time you do not have p(x) or if you have it is analytically untractable.
If somebody gives me a set l of examples taken from p(x), this $x \sim P(x)$, x_l are drawn from p(x), are realization of the actual process that I do not know. I can approximate the expectation of f(x) over p(x), i can apply my function f to all the samples of my dataset, compute all the f and take the average, if i push l to the limit (to infinity) this thing will converge to the actual expectation.
We've seen the posterior of the LDA is untractable, if somebody gives us examples of θ and Z, we can approximate that posterior in expectation. When you multiply two distributions, if they are not conjugate you do not know what is the resulting distribution, you may work in some of those cases, you cannot work with the actual distribution but if you have sample you can estimate in expectation.
When p(x) is not accessible, but samples x are, you can use the empirical distribution to approximate the analytical one.
Problem: Most of the time my x1...xl does not come from p(x), they come from another distribution which is not exactly the p(x) from which I am sampling.
p(x) usually is too complex, we can't sample from it.
After seeing LDA is not a surprise that params of a model can be random variables/distributions
θ in LDA is a distribution and contains the parameters of the multinomial dis-

tribution. The realization of θ for the specific document are drawn from a Dirichlet distribution. Generalizing this concept: whenever doing learning, one of the possible ways for doing learning, it's to draw the params from distributions, rather than learning them by an optimization mechanism, provided that I have the right distribution. In LDA model params, as well as the unobserved latent variables, they can all be drawn from the joint distribution of the unobservable. And here I put also β, the params. of the multinomials, nothing prevent us from saying that provided that I can write and execute this distribution, I can generate the params of my model by sampling them from the distribution. *Learning can be casted as a sampling problem.* Sampling from the posterior is useful whenever you want to classify something: that's sampling with evidence, I'm outputting a prediction for the mixture of the topics present in a document $\theta*$ and the assignment of topic to words z*, by sampling them from the posterior distribution of θ and z given a specific w*, it's a prediction, a stochastic prediction, because it involves sampling rather than a deterministic one, but provided that this things are fitted appropriately, it's a perfectly fine way of generating predictions, even a strong way because generate a prediction together with a measure of confidence around the prediction, if you want your model to be able to estimate the uncertainty about its prediction.
Works under the assumption that observations come from the actual distribution. The quality of the approximation that I get of my expectation depends from the fact that I'm not under this assumption, I'm gonna be sampling my examples from a different distribution. What are the conditions under which I get good estimates? The approximation that I obtain needs to be **unbiased** for being a good approximation, it's not completely off the target, I look also for that estimator that is so called **low variance**. Bias is connected to your ability to shoot on the target, on a small neighborhood of the target, your estimator gives correct estimates, the second sentence means that gives responses that are close one another, the first guarantees accuracy of estimation, the second ensures that the estimate that you get you get with few examples, if you shoot in a too large area you need more samples to converge to a proper estimation. With low variance you need less. $\tilde{p}(x)$ is the distribution from which i can sample $\tilde{p}(x)$!=p(x). Unbiased estimate: the expectation that I obtain w.r.t $\tilde{p}$ of my function coincides with the expectation which i obtain with the true distribution. Not require that the two distributions are equal point-wise, if you integrate out on the realization of X and you measure it through $\tilde{p}$ is the same measure you obtain with p, on expectation, on average. Provided that you can demonstrate that your sampling distribution $\tilde{p}$ on the samples (dataset) has the same distribution of the original distribution, it's ok cause you gonna be computing this expectation on a finite number of samples.
If the sample distribution and the true distribution has the same marginal, we have an unbiased estimator, and we have samples independence (two samples that I'm observing are marginally independent, no information sharing between the two). We have a low variance estimator: the variance complexity does not depend on the size of x, low cost estimates of your function even if your problem is high dimensional.

The quality of our estimator is connected with these 2 properties we've seen, in our case will depend not only on the properties of $\tilde{p}$ as a distribution but intimately with how we sample from that distribution. If we sample in a certain way from that distribution will happen that my samples are not independent and the distribution has nothing to do with it, but depends only on how i sample. This is not a problem if I'm working on the true distribution, i don't care about the sampling process.
Sampling all from Univariate distributions, if you apply the chain rule but you make it semplable, if you reduce their size leveraging conditional independence, and if you reduce interlocking between random variables, if you sample only sequentially that is the problem, in sampling the problem typically comes when you have a lot of random variables and sampling becomes sequential, to sample a random variable in the middle you have to sample before all of the others one. difficult to scale.
Look for reduction of the number of distributions involved, governed by the number of random variables, reduce the space they occupy, reduction of the interlinking between these distributions to sample in parallel. Bayesian Network gives all of this, easy to identify an order, sampling order given by the topological partial ordering, partial ordering is very nice, whatever is in the same class can be sampled in parallel.
But this is sampling from the actual distribution, this way of sampling lead to unbiased and low variance sample because all sampling are independent. What could possibly go wrong?
Typically I'm required to solve estimation of a posterior $P(Z|O)$, where Z are my non observable and O are the observable, let's say that O is S3, i want to sample a sample from my Bayesian network, for all the random variables but with S3 fixed, I sample from S1, then s2 ops!!! that's an unvalid sample, when you observe the common effect (collider) S1 and S2 becomes dependent one another, having observation in the presence of colliders leads to your distribution changing for all the colliders that you plug in, and understanding in which order you need to rearrange your things in order to sample from an appropriate distribution has exponential complexity.
Solution: I sample no matter what, and if I sample a sample not adequate for the distribution I reject it and resample again: very inefficient. Whenever there is evidence, sampling from evidence with an ancestor sampling is highly unadvisable from a computational perspective.
We need something alternative, cause sampling under evidence gonna be fundamental for us, there is a lot of posterior we would like to estimate→ use **Gibbs Sampling**: very simple, randomly initialize your variables with an assignment for my variables, then pick up one, flip it according to the proper distribution, sample a change, choose another one, sample it, choose another one, sample it, again again.. is it unbiased? $P(S_i|Markovblanket)$ I'm sampling from a marginal that is the true distribution, for the specific S_i is a valid sample, it will support the unbiased estimator, low variance property requires that two samples are independent, x2 is x1 with one thing flipped, even x5 is x1 with a couple of things flipped. All procedure is actually a Markov Chain $P(x_l|x_{l-1})$

they are not independent at all.
Start with a sample, let your Markov Chain run for a little while, and then you take this sample after that burn out time. You reduce number of samples, they are more independent but you waste a lot of time.
Hint of a more general class of sampler, Markov Chain Montecarlo Method that uses a fact that if you are smart enough you can design a Markov Chain whose limiting distribution, if you bring time to infinity converges to a stationary distribution, under some conditions, and the distribution you're converging to is consistent with the distribution you should be sampling from and you not have access to it.
We can flip more things at a time, we can update multiple random variables at the same time, provided that they share the Markov Blanket, if they have the same Markov blanket and the Markov Blanket of them is all set for all of them, all those that are sharing the same Markov blanket or a subset of it, they can be sampled in parallel.
The LDA exactly uses the Gibbs sampler, it works by deciding an ordering between the things you want to sample from, in this model we want to sample the latent topic association, the random variables that associates a specific topic to a specific word in a document, θ_i the parameter of the multinomial distribution corresponding to the document's topic mixing probabilities and also β. This is a model in which you don't learn, your parameters are sampled entirely from distributions. Which distributions? These 3 marginal distributions which are the Gibbs sampling distributions, there is an ordering between them, that ensure sampling that is consistent and unbiased. There is a time index in there, if you need to sample the association of the i-th topic and the j-th word at time l+1 of the Giggs sampler, you can draw from the marginal, having fixed the observations that are fine with Gibbs, observations are clamped, you never change them during the Gibbs sampling, so under certain observation you fix in that distribution the values of the Z to the values of the previous assignment, which is exactly the Gibbs sampling. Zi is gonna be a vector, for each topic which words has been assigned to it, I take the assignment from the previous time step l, and you keep all fixed and you are gonna be flipping one of them, the j-th here, and theta and beta are fixed, because also them are random variables but i'm not changing them. Completely in accordance with the Gibbs sampling, then once i have sampled all of them, i will plug them in the distributuon that i use to sample theta, and i will flip one theta at time or multiple theta at a time then in the same way i'll be sampling beta.
Gives you both the topic assignment as well as the params, and this are parametrized distributions, you're gonna be sampling from them, and their params are gonna be set by sampling, at the beginning you are gonna be sampling from distributions where params are set at random, and you're gonna be generating new values of parts of those params by sampling from this sort of random distribution the new values and putting them inside and this process will converge at some point.
Why sampling from something that starts from random will somehow converge at some point? What makes those distributions more consistent with reality?

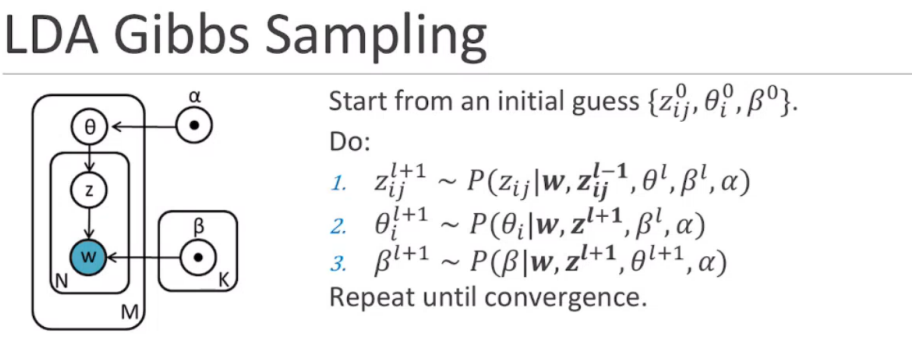


Figure 53:

The fact that there is w, everytime you're not just changing the params. of the model, that distributions depends from observations, every time you're including a piece of information that comes from the observations, that's what makes them learning. not the pure bootstrapping that you do by changing yourself based on the past version of yourself, you change yourself based on the past version of yourself and additional input info that comes from the data. That's what makes it learning.
It's a valid sampler provide we can proof that this thing converges to the actual distribution, at stationarity, with infinite time, provided that we can demonstrate that to the limit converges to sample taken from p(x) that's true. And if you're sampling from marginals that are the true marginals that's granted, because you keep flipping things, and if you take the samples at infinite time, after you have flipped infinite time according to the marginals, then the sample with almost certainty will be a sample from the original distribution.
If you sample from something that is marginally your distribution you cannot forever generate things inconsitent with the overall distribution.
Gibbs sampler is an example of a family of models which are the Markov chain montecarlo in which this state transition distribution to get a valid sampler needs to converge at infinite to the stationary distribution, which is the one from which we should be sampling. If you manage to demonstate that your q, transition function $q(x^{l+1}|x^l)$, follows the datailed balance condition (2 properties: **irreducibility** and **aperiodicity**) is a valid sampler. Then you can get a certain number of different models depending on how you pick up q, in Gibbs sampling q uses the marginals from the Bayesian Network and that will give me a valid sampler. Metropolis-Hasting uses proposal distribution that needs to have certain properties, particle filtering is something used in recursive model such as HMM, in the Kalman filter which is the continuous version of the HMM, Hybrid Montecarlo, you can get Langevin in which you move in a space in the direction of a gradient that is given by the so called score function which is the logarithm of the likelihood, certain number of sampling algorithm.
Sampling can give to you empirical estimates of expectations, a way for sampling

procedures of drawing even new complex data from a distribution in an efficient way, all things needed in a ML model, whenever you are generating new data from a generative deep learning model you are sampling from it. Sometime you use sampling to train a model, we've seen that one of the use of sampling is to generate the params of a learning model, we can use sampling to approximate the expectation in the ELBO, you have expectations some of those might not close, and you may decide to use sampling to approximate them, sometime you have a KL divergence between two distributions that you need to minimize, the KL div is a nasty beast, it does not have a close form solution in general, if you do not have it, well, the KL div is an expectation of the logarithm of two distributions, the function you're gonna be approximating is the expectation of the f, where the f is the logarithm of these two distributions and sampling gives you a tool to do that. Ancestral sampling while can look very interesting have issues whenever you're doing with observables and in ML you're almost certainly dealing with observables, you need to throw away a good quantity of the samples that you are generating cause they are not consistent or u need to use some member of the family of the MCMC typically the Gibbs sampling, often Langivin ultimately when you work with NN because it works on gradients and we works nicely with gradients in NN. Boltzmann machines which are interesting for many reasons: probabilistic models but also the first NN we're gonna be seen in the Deep learning module: they are both a Markov Random Field and a RNN, but a stochastic one, one whose activations are driven by a probability distribution, and that will make things slightly complex from the learning perspective and that will require some approximations: we are gonna be using Gibbs sampling to generate samples from which to update parameters of my model, Restricted Boltzmann machines in their deep version has been the original model that has created the old deep learning fads.

14 Boltzmann Machines

Fully hybrid model: probabilistic model (MRF), RNN, energy-based model. An excuse to see Gibbs sampling used to train a model.
Boltzmann Machines are first of all MRF, they need to be undirected. Nodes correspond to Random Variables, some of them will be obsederved/visibile, undirect full connectivity (undirected links), they are MRF with a very little bias, everything is connected with everything.
h $\rightarrow$ hidden, non visible, latent variables, v $\rightarrow$ visible units. s includes both the visible and the hidden units.
It is also a NN, I need to be able to find a sort of NN interpretation: what could possibly be the params, usually params are associated with the connectivity of the NN, if each of those nodes (RV) is actually a neuron, I would expect the params associated to the link between them. Link between two random variables interpreted as the existence of a synaptic connection between the 2 neurons, and the param M_{ij} is the synaptic weight between i and j.
The model of the neuron S_i: S_i aside for being a RV, is gonna have an inter-

pretation in term of activation of a neuron, but it's a RV, my activation needs to be stochastic, cause it's the value that I assign to a RV. Model of a neuron network defined of stochastic neurons that are binary. The output is 1 with prob. p_i and 0 with prob. 1-p_i. I can interpret the assignment of a binary value to my random variable as the output of a stochastic neuron. A neuron that fires 1 with a certain prob. p_i, stochastic because I will have a prob. of firing, and in order to determine if a neuron is firing for real, I toss a coin, a throw a dice. I will generate a random number between 0 and 1, if the random number falls between p_i and 1 the neuron fires 1, otherwise it fires 0. Stochastic because they have a probability of being active, and the fact that they're gonna be active or not depends on the interaction between this probability and randomness. Randomness is concretized by the fact that I'm pulling out a random number between 0 and 1, I may have in principle a neuron with 0.99 probability of firing, but i can be very unlucky in my drawing of the random number and the neuron is gonna be firing 0 in that particular case.
Haven't yet defined P_i, that's where input and params gonna play a role, I define probability P_i to depend on the input and the params.
$P_i = \sigma\left(\sum_j M_{ij} S_j\right)$
Looks like the local potential of a neuron, weighted summation of the input to a neuron, j are all the neighbors of my current neuron i, s_j is the output of all the neighbors, multiplied by the corresponding synaptic weight.
The actual activation of the neuron is obtained by passing this weighted summation through an activation function σ, that needs to return a probability $\rightarrow$ sigmoid activation. Has issues with the gradients but defines well a probability distribution over binary variables. M_{ij} are elements from a matrix M with all the params $M = NxN$ with N number of neurons, cause fully connected but symmetrical due to the undirectedness.
Why is a RNN? because I have undirected connectivity, which is the exact definition of having a loop between any two connected neurons. As soon as I have a RNN I should realize that my equations are missing something, whenever you have a feedback, you have a RNN, your activation at this current time, depends on the activation at the previous time step, in my equations is missing the indication of time, the activation S_i^t will be the the activation of my i-th neuron at time t, and the probability is gonna be the probability at time t, and the probability at time t will be computed by integrating the activation of all my surrounding neurons s_j at time t-1.
S_i^{t-1} this term is actually taking the state of the network at the previous time step, the activation of all the neurons at he previous time, and pushing it in input to each of the neuron to compute the current new activation. In M there is no diagonal, there are no self loops, the summation run on all the neurons but myself.
You treat the visible units as input neurons, at a given time you attach the input to v, then you use this update rule to compute the probability of firing of all the neurons in the network.
Each neuron in the network will take the value either 0 or 1, depending on their

probability and how this probability interact with the coin that you toss.
Be aware You toss a different coin for each different neuron in the network.
$E(\mathbf{s}) = -\frac{1}{2}\sum_{i \neq j} M_{ij} s_i s_j - \sum_j b_j s_j$
As it is a MRF over binary variables, we have a single clique (all the graph), the feature function or the energy function is particularly dumb, cause it is a sum over all the nodes in the clique which is all the graph. What this energy function is showing is that you are simply correlating linearly the activation of the different neurons mediated by the weight of the link existing between them, minus a term which is the equivalent of a bias. b_j are other params of my model, used to weigth the correlation between sj and the fixed input 1, which is the definition of a bias.
This M is a symmetric matrix, with no self-recurrent connectivity, so with 0s on the diagonal, my model params are M and b, encode the interactions between the variables and has this interpretation as synaptic weights.
To compute that probability, I plug the inputs and then compute the activations of the neuron, wordly way to say that I'm computing the probability $P(h|v)$ i.e. the posterior of the non-visible given the visible, I'm attaching the inputs and computing the activation of the other neurons
Let's look at the Neural dynamic from a Markovian perspective: Pj probability of firing of the j-th neuron, Pj (at time t+1) = P(Xj) at time t+1 = sigma(Xj(at time t+1)), xj at time t+1, is the net, the local potential, that depends on time t. So the current potential of the neurons in my network which is what that gives me, after I throw the coin, the new state, if the network S_{t+1} depends on S_t, since all of these are probabilities, there exists in the network a transition probability, probability of j-th neuron firing at time t+1, is the probability of S_{t+1} being equal to 1 given what I know, what i used to compute this Sj at time t+1 , i.e. the probability that depends on S_t, the evolution in time of the activations of my neurons, follow/is regulated by a Markovian dynamics, it's a fully first order Markovian dynamics.
How does the model state evolves in time? According to this Parallel dynamics, if you think that you are updating all the neurons. You are computing the probabilities of all the neurons in the network given the previous output of all the neurons, which you can obtain by computing independently the transition of each single neuron.
State transition function in classical Markovian Chain sense and this yields to this Markov Process: if I want to compute the probability of the full network, all the neurons being in a specific state s', where both things are vector, this is the assignment of the outputs for each of the neurons in my network at time t+1, this probability here I can estimate by introducing by Marginalization s which is the activation of the neurons at the previous time step and then applying the definition of conditional independence.
Highlight the classical thing that we know from Markov Chains, if I want to estimate the state of the MC at the specific time t+1, I marginalize on the assignment of the previous time, I compute the probability of the previous time and then with the transition probability i can estimate the current value.
Remember what we said for Gibbs sampling and more in general for Markov

Chain Monte Carlo methods, I can sample from some Markov Chain whose limiting distribution, when i push it to infinity, is consistent with the actual distribution from which I cannot sample, remember we are doing sampling cause there is a bad distribution, which is the true model distribution, but impossible to sample from it or do anything because there is a damn posterior that i do not know how to handle, with MonteCarlo Markov Chain methods such as Gibbs sampling, instead of drawing my samples from that distribution, in order to approximate things, I'm gonna be drawing samples from a Markov Chain, this thing works as long as I can demonstrate that the markov chain to the limit(when i push t to infinite) converge to the distribution I cannot sample. Things that have this particular Markovian structure they converge to a known distribution, which is the Boltzmann-Gibbs distribution, which is the distribution underline the Boltzmann machine whenever T, transition function, has some properties, related with the detailed-balance condition, that condition requires that your connectivity matrix is symmetric and you do not have self-loops. As long as this is true, we can sample from this Markov Chain to estimate stuff of my Boltzmann machine, and this is how we are gonna be doing learning, by sampling samples from this chain, having guarantees in theory that this thing converge where it should be converging
Substantially the detailed balance condition says that you can invert, going from s to s' with the same cost/probability. That guarantees that exists this equilibrium distribution, which is the Boltzmann distribution that guess what? Is the distribution that my Boltzmann machine has. Whenever you are doing sampling with Montecarlo you need to be able to show that you are sampling from a chain that converges at infinity, to the same distribution of your model. This gives you the constraint you need to apply to your model to be easily sampled: the conditions in this particular case for the Boltzmann machine are symmetric connectivity and no self loop.
Theorem: **Ackley, Hinton and Sejnowsky**: Boltzmann machines are capable of approximating any distribution defined over binary variables. Boltzmann machines can learn any distribution of binary vectors.
This entails learning the values of the model params as usual, these are synaptic weights or the strength of connectivity between the random variables that are enclosed in the weight matrix M, I can always include the bias into it.
First reasonable approximation to do some learning is to assume that all of the random variables are visibles (simplest case) in your MRF (then you introduce the hidden afterwards), learning is simple and require to maximize the log-likelihood.
Still a log-likelihood maximization problem, maximize the log-likelihood L(M) w.r.t. the model params, which are M_{ij}. L is it the number of dataset samples, and usually I had the logarithm before it, this was a multiplication, it becomes a summation, it is the usual log-likelihood defined over i.i.d. samples in my dataset, that allows to transform the multiplication into a summation and then I'm left with the logarithm of my visibles given model params.
How to find the update equation? It's a maximization problem w.r.t. to params

M_{ij}, I take the derivative of L(M) w.r.t. a specific elem of my matrix, $\frac{\partial \mathcal{L}(M)}{\partial M_{ij}}$.
I need to plug into the equation for P, but P is actually the model distribution, it's a MRF. $P(V \mid M) = \frac{\exp(-E(v))}{Z}$ where E is the energy function and Z is the usual normalization factor, partition function that makes sure that all things sum to 1.
If P is expressed like this, and I plug in the logarithm in front of it, log (exp(-E(v))/Z), logarithm plays nicely with the division, log (exp(-E(v))) - log (Z) and plays even better with the exponential, the two things annihilate one another and I'm only left with -E(v) -log(Z), so this is the thing on which i then take the derivative, reasonably easy becuase the energy is a linear function $-E(v) = -\sum_{i,j} M_{ij} v_i v_j$, taking the derivative of that w.r.t. to a specific M_{ij}, out of all the possible M_{ij} present in that summation only one will survive, when those ij are the one w.r.t. I'm differentiating. So when taking the derivative remains also $v_i * v_j$, all the rest disappears, is constant.
The derivative of $-logZ$ seems a nasty thing, but not that nasty, it's the derivative of the logarithm of a sum of an exponential. The derivative of the logarithm is 1/(argument of the logarithm), then the derivative of the argument is the derivative of a sum of exponentials, it remains as is, then the derivative of the argument of the exponential, but the argument of the exponential is the E(V) i've differentiated before, it's $vi * vj$, the only one survives. If we re-arrange things, we see that there is a sum over all possible values, you can give to any possible neurons in you network, it's a vectorial summation, it's a sum overall the values you can assign to all the variables in your network and v_i and v_j, which are gonna be indexed by this summation among the others. By noticing that $exp(-E(v))/Z) = P(V|M)$ and substituting we get that it is equal to the expectation of the co-activation of the i-th neuron and j-th neuron when the activation of these neurons does not come from outside, but it is generated by drawing v_i and v_j from the model distribution, it is the expected co-activation of neuron i and j under the model distribution, these are visible units, I have data that gives me these observations, what I'm saying here since I'm marginalizing out and weighting by the model distribution, is I do not care about the data, I' m caring about what the model thinks it should be the correlation between those units. $\frac{d \log P(V|M)}{dM_{ij}} \rightarrow$ this is the gradient for M_{ij} the thing I'm gonna be using to update the params M_{ij}, is equal to $v_i v_j - \langle v_i v_j \rangle_M$, where that notation is to indicate vi*vj under the model distribution.
Semantic: the way in which I change the weights existing between unit i and unit j, will look at how the two correlates when i take an input sample v_i and an input sample v_j, from the dataset and, I confront them with what the model thinks should be the co-activation. Sample of correlation or co-activation that comes from the evidential distribution, from observation and the other is what the model beliefs to be the correlation between the two, because the samples are taken from the model distribution and this thing is supposed to be equal to 0 because this is how you take the minimization, again what I'm doing is pushing the model expectation, cause there is subtraction between the two, to be consistent with data. This is the contribution to the update of the weight between

i and j, of a single sample. It is possible to generalize to L samples, then it is more evident that the first term is an expectation, the empirical expectation, the dumb distribution that returns 1 whenever a sample comes from the dataset. vi*vj expectations under data. When you are maximizing the log-likelihood, you are trying to push the difference to 0, i.e. pushing model expectations to match what you are observing from data.
What is the problem here? That updating requires to be able to estimate this expectation, and while the former might be doable when all the random variables are visible, also the former become a nightmare when part of the RV in my Boltzmann Machine are non-visible, in order to make this model usable I need to make some simplifying assumptions, to reduce the complexity. It's a MRF, the only way to reduce the complexity of a MRF, it's by constraining its structure, there are less interaction/synchronize computation in this expectations. The key problem in the Boltzmann machine is the fact that there is a lot of synchronization, because everything depends on everything, there is full connectivity, does not work if you use it like that, you need to restrict. You need to restrict your connectivity, **Restricted Boltzmann Machine** comes into play, you cannot afford full connectivity, I think a much simpler graph structure, a bipartite graph, a set of neurons (visible ones) that are disconnected between each other but connected to all the hidden units and viceversa that will allow me to be more efficient.
BM has both a fully probabilistic interpretation and also a Neural intepreptation, the two terms derived before, the data induced and the model based, they have two names **wake** and **dream** and connect to two well known concepts in NN learning, hebbian and anti-hebbian learning, the first term is an hebbian term because it will reinforce the weigth between i and j, whenever i and j are coactive, they're firing one together, it will reinforce them when the data is suggesting they should be active, the dream term, dream because it is not based on data, it's based on the model expectation, it is the model that is dreaming about the visible units, the second term is an anti-hebbian term, if the two units are co active will decrease the synaptic weight, if the expectation from reality and those from the model/dream coincide the term will be 0 you do not change the synpatic weights, if the reality tells you that 2 neurons are highly associated because they should be firing together and the second term says no, you have to change the dream part to be compliant with reality, so you'll push the param existing between i and j to be higher, so next time you'll see the samples you'll be more likely to be coactive together, pure hebbian term, if instead life tells you that these two should not be coactive together, the first term is 0, and the second term is saying oh yer this two are active together, you have again a disconnection between reality and dream part, anti-hebbian term to reduce coupling between neuron i and j so the next time, they won't fire together if supplied with the stimuli/input information.
In reality I'm hardly working with only visible units, I'm typically working with both visible and hidden units, so what I'm gonna be getting is this expectation and in this expectation there are non-observable units in both cases, in the second term everything is marginalized out, both the visible and the hidden, and

even in the empirical part i will need to marginalize w.r.t. to all the hidden units, so there is a lot of marginalization going on, this thing as such cannot be used, this expectation cannot be used in close form, but I have a tool to learn whenever I have expectations which is **sampling** so rather than learning using the actual expectations in closed form, I'm gonna be doing learning by using an estimate of these two expectations computed on sample that i can drawn from a Markov Chain, that i know will converge to the Boltzmann-Gibbs distribution cause i've done my homework, and my Boltzmann Machine has symmetric connectivity and no self loops.
I need first to restrict my Boltzmann machine, it cannot work in full connectivity, I need something that makes things computationally more efficient, remember we are gonna using Gibbs Sampling, efficiency measured in terms of how efficient is the Gibbs sampler defined over this method, Gibbs sampler is as efficient as I'm able to sample in parallel, based on the Markov blanket of the random variables involved, perfect! Because the Markov blanket of all the visible units, is all the hidden units, I can sample all the hidden units in parallel and update them in parallel while keeping the visible fixed, and viceversa. There is a lot of parallel sampling going on, which is highly efficient, all visible units can be sampled in parallel, and all hidden units can be sampled in parallel, you need to alternate between the units that are in two separate Markov blankets, so visible cannot be sampled together with hidden, you are gonna be alternating between sampling the visible keeping the hidden fixed, and viceversa. And all the visible and all the hidden (separatelly) can be sampled in parallel.
Energy function which has a particular form because it is a bipartite graph, there is only connection between v and h.
$-v^{\mathrm{T}}Mh$, no need to have v on the right side, no need to have h on the left side and the bias are separated. Graph bipartition makes my expectation tractable now, the all things factorize nicely, we are gonna be using Gibbs sampling, we are gonna be sampling hidden variables and visible variables to replace them. Remember how Gibbs sampling works, you start with a certain number of visible units, and a certain number of hidden units $[v_1, ..., v_{Lv}, h_1, ..., h_{Lh}]$, in principle i fix all and I change only one, but since there is this parallel possibility i fix/copy the values of the hiddens and then i change the values of the visibles to obtain a new sample, how do I change the values of the visibles? I sample that from the distribution, Is it possible to write the distribution of the visibles? Of course, we've written it before, it is the sigmoid of the sum over the input, which are the hidden units, weighted by the params, and when at the next step, you need to fix the visible and change the hidden you sample from the hidden distribution which is again a sigmoid of the sum of the inputs to each hidden neurons which are the visibles.
Now the 2 objects that I need to compute to update the weights between unit i and unit j, becomes something estimated by Gibbs sampling, so I clamp data on v, I fix v, estimate h, get a sample, these used to estimate the first distribution (data term), then to estimate the second one, I don't clamp anything in principle, I start sampling from the model distribution, until i get to infinity, and those will be samples from the model distribution. I have two expectations

to estimate, expectation D from the data and expectation M from the model, so i need to estimate two expectations, I need samples from the Gibbs sampler for the first expectation and samples from the Gibbs sampler for the second expectation, what kind of expectation I'm computing, $vi * hj$: this is the function I'm trying to estimate in expectation, need a sample of v and a sample of h, to multiply them, and another one, another one, and take the average of this multiplications and using as an estimate of my data and model expectation. How do I solve the problem of getting samples for the data expectation? I take my dataset that contains variables for the visibles, not for the hidden, I take one sample from my dataset and I plug into my visible units, I clamp it there, $v^0 = [010]$ the visible units at time 0 their activation are equal to 010, assuming that my network has 3 visible units, this means the first visible neuron is set to 0, the second visible unit is set to 1, and the third to 0, and these numbers comes from one sample of the dataset. Now Gibbs sampling: i fix my V and the previous are my V fixed and I compute my H at time 0, i use $p(h_j^0|v)$ where v at time 0 is [0,1,0] because I fix it, I plug into my equation there,i 'll get a number which is the probability of activation of the unit h_j let's say 0.9, then I randomly sample a number, if the number is larger than 0.9 so h_j at time 0 is equal to 1, and a repeat this process for all the j neuron in the hidden layer, and I'll get an activation for all of them at time 0, e.g. in this toy scenario we have 2 hidden neurons h^0= [1,0], the full sample at the end of this first iteration at time 0 is [v^0 = [0,1,0], h^0 = [1,0]], this is a sample that I can use to compute the correlation: the correlation between v_i and h_j when i = 0 and j = 0 it's 0*1 = 0, that i can compute for all the couples between v^0 and h^0.
I can compute this thing for one sample 1, I can compute for another sample 2, cause i'm taking the second element from the dataset, plug it here, use it to infer h^0, compute this correlations,..., compute this for a certain number of sample s, so this correlation i can compute for a certain number of samples, and I can approximate my exact approximation D as the average of this co-activation in my s samples, this will allow me to estimate the first term, but how do i get a good sample to estimate the model, this one is easy, the visible I obtain that from the dataset, and corresponding hiddens I can obtain them reasonably by inference based on the visible, but samples where both v and h come from the model distribution, means I can invent some initial v here, infer h^0 than use this to sample v^1 use this to infer:
$h^1 \rightarrow v^2 \rightarrow h^2 \rightarrow ....v^{\text{inf}} \rightarrow h^{\text{inf}}$ I need to get at infinity and this would get me one sample, because at infinity you'll be reasonably independent from the initial condition, of course you do not have time to go to infinity to estimate only one sample when you need s, you surrender to the bias of Gibbs sampling, you accept something this is not ideal and you truncate this to a finite number of time, instead of going to infinity you stop after a reasonable number of time steps and use that as a sample to estimate empirically the model distribution.
You are sampling from the Markov Chain: $P(S^{(T+1)}) = \sum_s T(s' \mid s)P(s')$. The summation on is due to marginalization over all possible previous states s. This is the Markov Chain you are sampling, you are sampling in a very peculiar way, given by the fact that you have a Gibbs sampler and a bipartite Graph, the

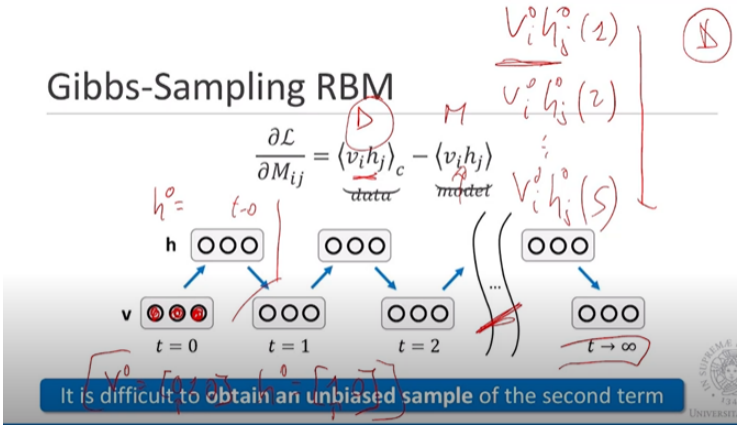


Figure 54:

transition from state S_t to state S_{t+1} is done into steps, because thanks to the bipartition you can sample all the visibles and all the hiddens all together, but separately one another, once you have that thing you can move to the next time step.
The point is that: it's reasonably easy to sample good samples from the first expectation, but it is difficult to sample good samples, unbiased samples for the second expectation, the model one. We are gonna be needing to approximate the approximation, in order to make it efficient, and the way to approximate the approximation, is not to wait until infinity to get my examples at stationary time, because at infinity I'm guaranteed that those samples are from the same distribution as the Boltzmann-Gibbs. I can't wait till infinite, so I wait for a finite number of step, how many??? For each of those time step I can estimate the correlation $< v_i, h_j >_1$, this is the correlation that I estimate using samples taken at time 1, ..., $< v_i, h_j >_{inf}$ is at infinity, whatever was before: an expectation from data minus an expectation from model becomes an expectation from data estimated using samples at time 0 - an empirical expectation using samples taken from time infinity which is the model expectation, in principle I have my ideal expectation data and model, now I'm using Gibbs sampling to estimate those 2 ideal expectations using empirical estimates, using samples taken from that Markov chain, the data expectation is an empirical expectation using sample that comes from time 0 from the Markov chain, the model distribution becomes an empirical expectation based on samples that come from infinity from my Markov chain. If you compute the correlation between v_i and h_j at time 0 for example number 1, and if you compute the correlation between v_i and h_j at time 0 for sample number 2, and so on and you averaged i, you are exactly computing $< v_i, h_j >_0$, empirical expectation using samples drawn from time 0, if you do it with samples coming from infinity you get an estimator of the model distribution, this thing allows me to rewrite the 2 ideal expectations as 2 empirical expectations, this still does not solve the problem because the

first one I can compute efficiently, the second one requires go to infinity, change infinity with a finite fixed time k, cutting those Markov chains.
This is particularly problematic with Gibbs sampling as it is high variance and it requires a lot of samples in order to converge to something decent and I cannot afford to go to infinity for a lot of samples, what I'm gonna be doing is computing the first term, the expectation w.r.t. to the data as usual using samples from time 0, and compute the approximation of the model distribution using sample from time 1 as a rough approximation of infinity.
That's your delta with which you update the params, the difference between the empirical expectation from samples drawn at time 0 minus the one of the samples drawn at time 1. **Constrantive-Divergence** learning , CD(1) cut/clamp at 1, CD(k) in general
This approximation of an approximation does really converge somewhere in theory? Nowhere, the contrastive divergence loss is something that exists, it's a difference between two Kullback Liebers, the KL between your model distribution and stationary distribution and the model distribution and the empirical distribution from which you are sampling, in principle it recalls that is minimizing that divergence, in practice the way you are sampling is inconsistent with that objective, cause that objective entails one term that is really hard to compute, you hope to be going in that direction but it is a very very crude approximation, there is actually a theorem that shows that is not following the gradient of any function.

It is an iterative algorithm goes through your data multiple times (epochs), you need to compute the wake part, the part about data expectation, I take my data, *dataOr* is the original dataset packaged into the matrix, and since i want to be extremely stochastic I randomized also the dataset, I confront my data which are 0 and 1, with randomly drawn thresholds between 0 and 1, or may be my data is not binary i want to randomized the binarization of my data, once i have the visibles I need to compute the probability of activating the hidden, *poshidP* it is the sigmoid of the weighted summation, this is the probability of h^0, the visibles are v^0 , to get the actual h^0 I can thresholding with some random values, or i can use directly the probability rather than their concretization, it is a smoothed version. Then I compute the correlation between the visible and the hidden so here I get something that is the estimation of the correlation between the visibles and the hiddens at time 0, now i need to compute the correlation between the hidden and the visibles at time 1, How? I take the stochastic hidden activation h^0, take the probability of h^0 and thresholding with some random thresholds, then I compute the probability of activating v at time 1 $P(v^1)$, sigmoid of the linear combination of the weights and hidden inputs, then I get the reconstruction of the actual v^1 but taking the probability, thresholding it by random thresholds, then I use this one to generate $P(h^1)$ that use the v taken from before and again I get my sample correlation and then I can update my weights. Of course I'm gonna be go through my dataset a certain number of times.
Used for compression → Mnist digit, visible units are pixels of Mnist images,

RBM-CD in Code

```
for epoch = 1: maxepoch

%--- Compute wake part
data = dataOr > rand ( size ( data ) ) ; %Stochastic clamped input
poshidP = 1 . / ( 1 + exp (-data*W - bh ) ) ; %Hidden activation probability
wake = data ' * poshidP ;
% Alternatively : wake = data ' * ( poshidP > rand (size(poshidP) ) ) ;

%---Compute dream part
poshidS = poshidP > rand ( size ( poshidP ) ) ; %Stochastic hidden activation
reconDataP = 1 . / ( 1 + exp (-poshidS*W' - bv ) ) ; %Data reconstruction probability
reconData = reconDataP > rand ( size ( data ) ) ; % Stochastic reconstructed data
neghidP = 1 . / ( 1 + exp (-reconData*W - bh ) ) ;
dream = reconData '* neghidP ;
% Alternatively : dream = reconData '*( neghidP > rand (size( neghidP))) ;

% Reconstruction error
err = sum (sum (  ( data-negdata ) .^ 2 ) ) ;

%---CD_1 Update
deltaW = (wake - dream ) / numcases ;
deltaBh = (sum ( poshidP ) - sum ( neghidP ) ) / numcases ;
deltaB v = (sum ( data ) - sum ( reconData ) ) / numcases ;
. . .
end
```

Figure 55:

the hidden units are features, 50 h, you are taking an high dim rep 16*16 and represent as binary vectors with 50 bit, find a compressed representation of higher dim images, it's also a latent space/variable model, smaller dimensional non observable representation of higher dimensional visible info.
When you start training a BM, on picture of a 2 for example, you can see slowly sign of a 2 emerging in the weight space, each of these picture is a set of weights associated with one hidden unit, and you can visualize as an image cause each hidden unit is connected with each visible unit, you have one visible unit for every pixel you can rearrange as an image and visualize.
When you have an input image you plug it there in the visibles, you can then sample h^0 you create a vector with 50 elems of 0 and 1. $h^0 = [0101...0010]$ when you reconstruct i.e. computing v1, you take this vector of bits and decoding back to an image, v $\rightarrow$ h projecting an image in a space of binary vector, viceversa h

$\rightarrow$ v you are decoding that binary vector in a reconstructed image. The weights of an hidden neurons are filters from which you can then reconstruct your image, you are learning a generative model anyways, you can combine those weights to reconstruct the original image, you pick-up all the weights corresponding to all the one, discard all corresponding to the 0, you sum them, and you get a reconstructed image, each hidden neuron will learn to focus only on relevant aspect of the original image that deserve to be represented in order for that image to be reconstructed, is an information compression mechanism which also leads to denoising, to capturing only the relevant features of the image, because are the only one that will allow with high likelihood to reconstruct a realistic image according to your data, since you are pushing your model expectation towards the empirical distribution.
Deep learning $\rightarrow$ a particular DL architecture is neural autoencoder, and the first Deep neural autoencoder to be produced use the RBM to train the layers.

15 Deep Learning - Autoencoder Models

Introduction on unsupervised learning, in Deep Learning connects nicely with Boltzmann machine.
Relevant foundational model for vectorial data: images and sequential data. Deep autoencoders and their connection with RBM. Convolutional NN relevant model and framework for dealing with images, GRU and LSTM for sequences. And more advanced models.
Autoencoders, the model that has introduced all DL, including the term, autoencoder architecture has inspired also diffusion model.
3 families: **sparse**, **denoising** and **contractive**. Autoencoders, aside from their historical value, are important because what they really give to us, is a deeper understanding on what DL is about, DL is not about having super deep neural network, that's not the point, the point of having deep layers is that of **learning good representation**, Deep Learning is all about learning appropriate representation, learning representation from *complex articulated data*, *highly noisy*, making sure that this representation that we learn, and we learn them using the hidden layers of a NN, are informative, robust and generalize.
Neural autoencoders, help because they provide us with a view of what really means finding good representations, irrespectively of the predicting task, cause neural autoencoders are unsupervised.
Deep generative-based autoencoders, that essentially uses Boltzmann machine to be trained, and that would be an excuse to introduce a trick that has let DL to start: *greedy layer-wise pretraining.*
Autoencoder: NN whose job is reconstructing in the output, the input that receives. Receive in input x and should reconstruct in output x, you're learning the identity function, in order to reconstruct an input in output, you make sure that your input information passes through an information bottleneck, restrict the capacity of the model, in term of representing info, to force the model to represent only those aspects of the input data that are extremely relevant to

reconstruct it in output. *(Relevant factors of variations)*. Assumption to make it work: *manifold assumption*. Input layer x, gets projected in a latent space h, made of hidden neurons, and from those hidden neurons, x is reconstructed back. If the size of the input is much larger than the size of latent space or we can constraint our hidden layers, there must be an information bottleneck: architecturally less hidden neurons than input neurons, simplest thing you can do and no much control on the property of what you are doing. The 3 models we will see are ways to introduce this information bottleneck in other ways than purely constraining the number of hidden neurons.
In principle you would like very powerful NN, highly parametrized, but with incentives to not use that power all the time. SGD makes sure that even if the NN is highly parametrized, *trained with SGD is naturally regularized*.
Use the largest model possible without using it all, all the flexibility but only if you need it. H is a latent representation of the input x, latent cause obtained by hidden neurons, no ground truth to attach to hidden neuron activation. h is akin to repr. Z in latent variable models. h is continuous, in probabilistic models can also be continuous but are often discrete. The first part that projects the input rep. in the latent is the encoder; the second is the decoder. This scheme generalizes to transformer, that are encoder-decoder architecture.
First proposed as neural compressor, train the model, the sender use the encoder, the receiver use the decoder, cable in between them, you want to maximize the amount of info send without overloading the communication channel. They used linear encoder, not very dissimilar than PCA, not much more powerful than that. Linear autoencoder trained with MSE has been demonstrated equivalent to PCA.
Having a lot of hidden neurons, but constraining/penalizing the model to have the h sparsely active.
Hidden representation of the neurons should be set to a value different from or not closed to zero, as few neurons as possible.
Trained by minimization of a discrepancy loss/reconstruction loss which compares the reconstruction with the original, so MSE essentially. Then we use modification of this Loss, regularized appropriately also to obtain the information bottleneck, one possible way is by sparsifying the activation, the other by penalizing the loss for complexity.
Regularized autoencoders build a neural autoencoder, not imposing a bottleneck on the architecture but rather imposing the bottleneck using some form of regularization, **sparse AE**: *sparsifying the activation*, **denoising autoencoders**: *use noise to regularize your model*, and **contractive AE**: *adding an explicit regularization term to the loss*.
Sparse AE adds a penalty term to the cost function to regularize for what? Till now, in ML course, we have seen regularization applied to weights, norm 2 or norm 1 penalization applied to the params, here we have a different objective, use all the params if needed, but we want to regularize the usage of the hidden representation, we want to impose the computational bottleneck on how much of it can be used, penalizing not the params but the activation of the hidden layer, with the same mechanism.

The loss of the sparse autoencoders: $J_{\text{SAE}}(\theta) = \sum_{\mathbf{x} \in S} \left(L(\mathbf{x}, \tilde{\mathbf{x}}) + \lambda \, \Omega(\mathbf{h}) \right)$, where $\Omega(\mathbf{h}) = \Omega(f(\mathbf{x})) = \sum_j |h_j(\mathbf{x})|$
The first component is the reconstruction error, the second term is a penalization with hyperparam. lambda, trade off param selected with model selection, penalization function omega of my hidden layer activations.
One way in which this activation is typically taken according to what we do, whenever we use for instance, Lasso regression, is to pick up the norm one of the vector of the neural activation, the norm one of the encoder applied to the input information. Penalized for the sum of the absolute values of each of the hidden neuron activations in response to a specific input each, classical norm 1 penalization. This thing has a probabilistic interpretation, MLE is more or less akin to Mean Square minimization, actually MS minimization comes from taking a maximum likelihood approach on a regression problem with a given error, normally distributed with 0 mean and unitary variance, but we also know that MAP Maximum A-Posteriori is a regularization of maximum likelihood, it works by adding a prior distribution over the params, here we can interpret what we have seen before as a Maximum likelihood problem where we have introduced a prior distribution rather than on the parameters on the distribution of the activations of the hidden neurons.
P(h) is not really a prior,a prior is your a-priori assessment of your hypothesis/parameters/hidden neurons before you see some data. h is actually h(x), is the hidden activation in response to a specific input, is not really a prior, to be an actual prior, you would need a marginalization overall the hs, or, if you like, the activation of the neurons in response to any possible input. Not really a prior, more like a point estimate of a prior, instead of using a marginalization over all the possible activations of your hidden neuron, you're selecting one single activation, which strangely enough is the activation corresponding to a specific input that you're receiving, know x and use it to penalize.
We are training our autoencoder by ML with a penalization induced by a prior that we place on the hidden layer activation. You want an over-complete autoencoder and constrain it.
Denoising autoencoder: we do not change the cost function used to train our model, but we alter the input that gets into the model. The network is encoding an x_{hat} and then decoding it from the hidden layer and then comparing the reconstructed x_{tilde} with the original input x that the network never saw an input, since he saw only $x_{hat} = C(x_{hat}|x) = x + \epsilon$ where epsilon is drawn from $N(0, \sigma^2)$, where the variance is small because you do not want something too different, which is just the result of a noise process, something that receives in input the original x and spit in output x_{hat}, a noise version of the input. I pull out an epsilon vector of noise from a normal distribution with small variance, and I add it to x, classical additive gaussian noise. Noise is a good regularizer, often used in many processes in the NN, help it to focus on the signal, and to learn that I cannot trust blindly the info that pass, because it contains noise. The autoencoders receive in input the noisy version of the input and reconstruct the clean version of the input.
Other interpretation: it is actually learning an approximation of a generative

model, is learning a denoising process from x_{hat} to x. Not seeing x but a primitive form of a distribution around x (all the noises version of x that you create and pass to the model), you are not gonna see x, but you are gonna place an area of uncertainty around each of the training samples, with radius σ^2, and then I'm not gonna see them explicitly, but instances of those gaussian distributions centered around the data point, which are the realizations of the noise process. Primitive form of distribution fitting, trying to learn an approximation of the data generating distribution, which is approximating the points in my data through tiny little gaussian placed around those points.
Learns the denoising distribution $P(X|X_{hat})$ by minimizing minus of the log of the denoising distribution, equivalent to maximize the likelihood.
Not really doing distribution fitting, because this model is too much informed by the data, x^{hat} tells you a lot of stuff about x, still a conditional distribution on which on the conditioning side there is a lot of information about the samples that you would like to place the distribution on, so it is not really very much near to P(x)
But interesting because if I'm learning that denoising distribution, if I am smart enough I can use this model to sample new data.
This dumb model, essentially I'm taking a NN, I'm confusing a little bit the inputs and asking to reconstruct the non-confused input has some theoretical connection with generative models. That started all the wave that has brought to introduce **variational autoencoders**, which are the parents of diffusion models.

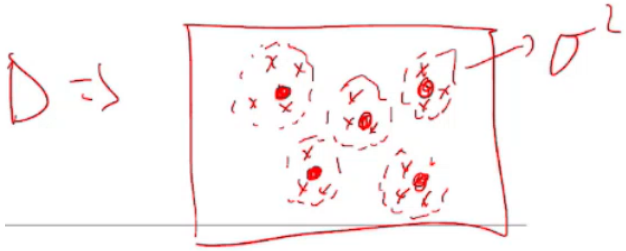


Figure 56:

Manifold assumption: your data, the data that the NN receives, may be high-dim in the original input space, think about images (N X M): better, the way we represent it is high dimensional but *the informative content inside has a much lower dimensionality*, because you can't get semantically meaningful images if you pick at random the values of the pixels.
An image is a set of pixel, NXM pixels, grey-level, the way in which I can generate such an image is in principle, I generate at random the values of the single pixel by a distribution to 0 to 155. Does generate semantically meaningful images? Well, it can, but most most of the time you will be generating salt and pepper noise. Even if my samples live in a very high dim space size N x M, the configuration of the information that is written in the pixels is not free, it can-

not take that independently, *they need to co-change*, in this very high-dim NxM space there must be a lower-dim structure, that tells me that if I move among these structures I change the pixels in my image in a consistent way (the image change meaning but there is still a meaning), if I move away from this lower dim structure I start generating disturbances in the semantic interpretation of the image
This mathematical structure I assume that exists is a manifold, a lower-dimension topological structure embedded in a much higher dim space, that represents what is the area of the space where data make sense, where data results.
My data can be as complex as living in a very articulated domain that is not even euclidean but the manifold where data live is at least locally euclidean, on a reasonably small scale I can use euclidean geometry, going over there no longer euclidean. data come to us represented in an high dim space but actually live in a lower dim space, as complex as you wish in term of twisting and turning, but if you manage to straighten it, at least locally looks like euclidean. *This low dim space will define the only relevant factor of variations*, all those directions that keep you tangential/attached to the manifold. Directions of variations are meaningful because I still remains on the manifold, if I go far from the manifold i lose semantic. With the neural autoencoder i attempted to find the manifold that contains the data, identifying only the relevant factor of variation, this is what representation learning does in general in DL: you try to find neural representation and coding of data X in the activation of hidden neurons, where activation of hidden neurons represent only the relevant factors of variation that makes sense of the data.
You can prove that what your denoising autoencoder is learning it's a vector field that kicks your noisy data backward to the manifold. Geometrically it is learning the reversed vector, that takes the perturbed data point x_{hat} and brings it back to the manifold, with the noise I'm pushing the data point out of the manifold, and then with the encoding-decoding process I am really inverting that pushing, but I do not do it once, but several times, for all the data with multiple disturbances of the manifolds, lot of re-pushing vectors for all of your data, a field of vector, a gradient/vectorial field that gets only in one direction, where from anywhere in the space: towards where the manifold is, the denoising autoencoder is actually learning the vector field that is an approximation/ a vector/ a gradient, that is the derivative of the data, *you're learning a vector field that from anywhere in space tells which is the direction in which you increase your likelihood, the direction that increases your likelihood to be a point of the data distribution.* So take a noisy version, the denoising autoencoder will give a reconstructed version that has the property to be closer to the manifold, so take one step and gets nearer to the manifold, I'm still far, throw the reconstructed inside the autoencoder and the next reconstructed version will be one step closer to the data generating manifold, repeat again, again and I will see a proper image and this is how you generate data with this model. I draw a random salt and pepper noise image and then I start sending it in input to the denoising autoencoder and I iterate the reconstruction cycle multiple times until this image makes sense, because you have learned this vector field that

from wherever you are brings toward where data exists and lives.
Imagine dealing with image like Mnist digits, 16 X 16 images, it doesn't mean that every way in which you choose these 16 by 16 pixels, will make it an image of a number, there is only a subset of the ways in which you can take your pixels that will make an image of a number. All your images of numbers, exist on a manifold, on a reduced dimensional representation of your pixels space and this manifold will define the only relevant factors of variation for your data, imagine you are only concerned about number 4, the data manifold for a world made only of images of number 4, the relevant variations is any variation that changes the way in which you write the 4, but does not change the semantics of a 4. Still remains a 4, changes in style, changes in appearance, with your manifold hypothesis you are saying that e.g. this style variation is captured by one dimension of the manifold, the changes in rotation is captured by another dim of the manifold, moving along a direction of the manifold will rotate the number, moving along another direction will change the style or the size of the numbers. Moving along those directions will keep the 4 a 4, moving outside of the manifold then it becomes a 5 or salt and pepper noise.
Contractive autoencoder: tries to do this thing in a slightly more mathematical way, if the encoder takes different versions of the input and projects them into the same encoding, then the decoder just decodes the right hidden representation.
The contractive autoencoder forces the NN, the autoencoder, to make the encoder (f function) to be robust to small variations of the input. Mathematically: you take a nice derivative of your encoder over x, the infinitesimal variation, my autoencoder as usual will be trained by a part of the loss that minimize the reconstruction plus a *penalization that penalizes the Frobenius norm of the Jacobian*, the Jacobian holds all the partial derivatives, tells you how much f, the encoding, varies if I change x infinitesimally. Change slightly x, check how much f(x) changes, it will tell the model make sure that your f changes little if you change x little, otherwise you get a strong penalization, the Frobenius norm is the trace of the multiplication of the matrix of the Jacobian, it penalizes if you really have picked responses in any of the directions is a linear operator, you have all the possible reactions of your function when you perturb your input x in all the possible directions and you want all these reactions to be small. Same effect of the denoising autoencoder, being robust to small variations of the input, but these variations are really small, infinitesimal, and I'm not telling that all the pipeline has to be robust, only the encoder, because the decoder can do is job if the encoder is robust. You can penalize also an higher order derivative. $J_{\text{CAE}}(\theta) = \sum_{\mathbf{x}\in S} \left(L(\mathbf{x}, \tilde{\mathbf{x}}) + \lambda\, \Omega(\mathbf{h}) \right)$ where $\mathbf{h} = f(\mathbf{x})$, $\Omega(\mathbf{h}) = \left\| \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right\|_F^2$ and $\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$
This is an Autoencoder with a single hidden layer, where is DL here? If you can solve the problem for a single layer, you can apply to multiple layers. Major problem: if you use lot of layers, nasty numerical problems of the gradient descent doesn't allow you to train the model, Backpropagation fails miserably due to numerical errors if you are not careful. Researchers came up with recipes

and workarounds that makes the deep learning world start.
A deep autoencoder is a NN that receives in input an x and projects into increasingly smaller representations incrementally and has mirror in one thing that decodes back, because need to be an autoencoder, if you remove the decoding part after training, only using the encoder, there is a lot of things you can do, because if you have worked properly you have discovered more or less the manifold of your data, so if x is high dimensional, noisy and confuse, $h_4$is possibly very well robust, generalizable, because it lies on the manifold and only captures the relevant factors of variation. If I have n vectors that in X-space they look different, but they're actually perturbation of the same thing, in h_4 they'll look the same thing. If you put a classifier on the top h_4 it will work better than putting it directly on the input, because all this chain of representations has done the work of getting rid of all the factors of variations that are irrelevant and the representation that is in h_4 only encodes things that survives all this cleaning of the irrelevant factors.
Autoencoders can create this nice representation on the top of which you can mount your favorite linear regressor, SVM whatever...Can be exploited also to reduction of dim to visualize the data, or even if it is high dim I can put a PCA on the top of it I would be doing a PCA on something that is small and robust in term of representation than doing the PCA on x.
How do I resolve the problem of backprop not working? I know how to train an autoencoder with a single hidden layer, can I use to solve a problem with multiple autoencoder, yes! I start with x, and I build an autoencoder for x, single hidden layer, done : i have learned the params of my autoencoder, done but only has one layer: no problem, detach the autoencoder, save what you need to save the params, and then train a second autoencoder on the h1 corresponding to the x of the training data, train it, you'll get a second layer, that you can attach to the previous, then train another autoencoder, take you training data, pass through h1, then from h1 pass to h2, and you obtain an h2 representation of your dataset that you can use to train another autoencoder to get the third layer and so on: you get a deep autoencoder. **Greedy layer-wise pretraining**, as soon as h1 is trained you never revise it, when you train h2, h2 uses representations from a frozen h1, but does not have backprop problem cause you always are propagating through 1 layer only.
Also used in cooperation with a fine-tuning phase, after you've trained your basic autoencoder you can reconstruct the whole autoencoder, try do reconstruct a sample generate an error of reconstruction and backpropagating backwards, will the gradient vanish? yes, most likely, but not entirely, whatever gradient gets to the first layer is a bit of fine tuning, this is a smart way of initialize your network to be in a reasonable area where the solution resides, instead of starting completely at random.
This was the intuition that generated the idea of DL, creating deep NN by going around the problem of vanishing gradient, by this pretraining.
Instead of a neural autoencoder, the first approach used a RBM to learn each of this individual autoencoder, a Boltzmann machine is an autoencoder, because you can plug the x, in the Gibbs sampling and you generate the h, and you

decode the h back into the x, it is the definition of an autoencoding task. **Deep Belief Network** is a *stack of pairwise RBM*, this is a deep autoencoder but not a deep Boltzmann machine, a Boltzmann machine is MRF, connectivity is undirected, here the connectivity from layer h1 upwards it is direct connectivity, because a proper Boltzmann machine $P(x|h1)$ is a standard Boltzmann machine and have undirected connectivity, as soon as I get to h1, the proper way of having h1 is that h1 is influenced by x bidirectionally, but h1 is influenced bidirectionally also by h2, you need both direction, the activation function of the Boltzmann machine neurons this time will need to be conditioned both on x and h2, if I want a purely RBM, I need to change that and training like that, fine, I can do it, but I need to be a little careful, because if I am training the first RBM between x and h1, I will learn this distribution $P(x|\theta)$ and P$(h1|W1)$ and $P(x|h1, W1)$ the ones that I sample when i go up and down with the Gibbs sampling, once that I've done that I move on, I can train these by contrastive learning and i get those distribution P(h) and P(x) and they share the same set of params w, fine, when i move forward to train the next RBM what i train is $P(h1|W2)$, you can imagine that this thing will lead to 2 different distribution over h1, the distribution over h1 that i get from below and a distribution over h1 that i get from this one, with a different set of params, W1 and W2 but this are 2 distributions for h1, so h1 should be distributed accordingly to both, I need to put the 2 in agreement, I need to average them, the first one is trained independently without knowing about h2, the second one is trained indip. and I get another distribution over h1, i need a distribution over h1 that comes from the yellow and the distribution of h1 that comes from the blue to agree, to be consistent one another, to get a single distribution for $p(h1|W1, W2)$ that armonizes both, I need to average between the two but I need to be careful because of double counting, because x influences h1, but x also influences h2, because h2 uses h1 which contanins x, if I am not careful I'll double count the effect of x when averaging the 2 distributions. Big trick: elegant dressing for something that is very ugly, how do I actually train this model: when i train the lower RBM i double the input, literally multiplying by two the contribution from x, and I train the first, when I am training the second I am multiplying by two the contributions from h2, this way the weights of w2 and w1 will be already doubled, then i will use them without without the times 2, so they will be automatically averaged when i combine them, is a huge trick to obtain averaging in a smart way without double counting for the inputs, I use them twice when I train them but when I use them in the deep version W1 and W2 as you can see they occur only once not twice like in the other mode. This is the case for only two hidden layers, when you train for a deeper network, for the layer at the bottom you double, for the layer at the top you double, this one in the middle when you train this one in the middle for instance you just train it normally and divide by two the weights.

A Restricted Boltzmann Machine (RBM) is an energy-based model consisting of a visible layer $\boldsymbol{x}$ and a hidden layer $\boldsymbol{h}$, connected by symmetric and undirected

weights $\boldsymbol{W}$. It defines a joint distribution:

$$P(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z} \exp\left(\boldsymbol{x}^\top \boldsymbol{W} \boldsymbol{h} + \boldsymbol{b}^\top \boldsymbol{x} + \boldsymbol{c}^\top \boldsymbol{h}\right)$$

An RBM can be interpreted as a shallow autoencoder, where input $\boldsymbol{x}$ is encoded to $\boldsymbol{h}$ via Gibbs sampling, and decoded back to $\boldsymbol{x}$.

A Deep Belief Network (DBN) is formed by stacking RBMs such that the first layer ($\boldsymbol{x} \leftrightarrow \boldsymbol{h}_1$) is an RBM, while upper layers form a directed belief network ($\boldsymbol{h}_1 \to \boldsymbol{h}_2 \to \cdots$). Despite their structural depth, DBNs are not Deep Boltzmann Machines (DBMs), because their upper layers are directed.

A Deep Boltzmann Machine (DBM) is a fully undirected probabilistic model where each layer communicates bidirectionally with its neighbors. For example, in a two-hidden-layer DBM: $\boldsymbol{x} \leftrightarrow \boldsymbol{h}_1 \leftrightarrow \boldsymbol{h}_2$, the hidden layer $\boldsymbol{h}_1$ receives input both from the visible layer $\boldsymbol{x}$ and the deeper hidden layer $\boldsymbol{h}_2$. This creates a problem during pretraining with RBMs: $\boldsymbol{h}_1$ appears in both RBMs $\boldsymbol{x} \leftrightarrow \boldsymbol{h}_1$ (with weights $\boldsymbol{W}_1$) and $\boldsymbol{h}_1 \leftrightarrow \boldsymbol{h}_2$ (with weights $\boldsymbol{W}_2$), leading to two independently learned distributions over $\boldsymbol{h}_1$. To correctly model $\boldsymbol{h}_1$ in the joint DBM, one needs to average these contributions. However, care must be taken to avoid double-counting the influence of $\boldsymbol{x}$, which also affects $\boldsymbol{h}_2$ indirectly.

A clever solution is to double the bottom-up and top-down contributions during pretraining, then use the weights unmodified at test time. Specifically, during training:

- When training the lower RBM ($\boldsymbol{x} \leftrightarrow \boldsymbol{h}_1$), double the influence of $\boldsymbol{x}$.
- When training the upper RBM ($\boldsymbol{h}_1 \leftrightarrow \boldsymbol{h}_2$), double the influence of $\boldsymbol{h}_2$.

In the full DBM, the activation of $\boldsymbol{h}_1$ is computed using:

$$\frac{1}{2}\boldsymbol{W}_1^\top \boldsymbol{x} + \frac{1}{2}\boldsymbol{W}_2 \boldsymbol{h}_2$$

This implicitly averages the two influences while avoiding double-counting. For networks with more than two hidden layers, the trick generalizes: the first and last RBMs are trained with doubled inputs, while intermediate RBMs are trained normally and their weights halved when composing the full DBM. This ensures a consistent marginal over shared layers and allows for effective layer-wise unsupervised pretraining of DBMs. Think of it like budgeting attention:

Imagine h_1 can listen to 2 people (input x and upper layer h_2), but the first time you train, only x is talking.

To prepare h_1 for hearing both voices later, you make x talk twice as loudly during practice. Then in the real model, you let x and h_2 talk normally, and h_1 just averages what both say.

You can use them for visualization, you can train a deep autoencoder on MNIST data and then plot the hidden representation at the final layer of your autoencoder, PCA or t-SNE it, and see that you can easily obtain different clusters of data, each color a different digit. Why? The representation that you obtain in the deeper layer of your autoencoder is consistent with keeping the

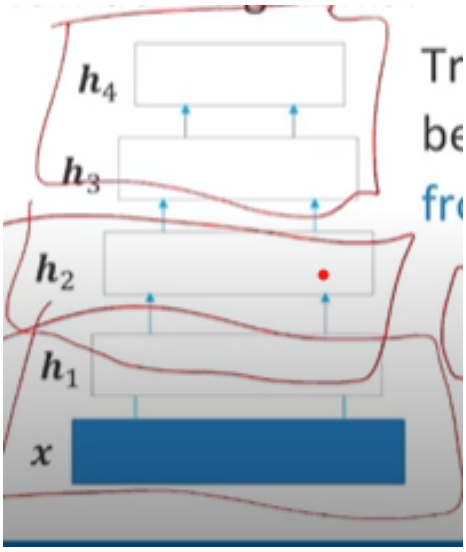


Figure 57:

semantics of the data and capturing only the relevant factor of variations, which will most likely make images of the same number represented by nearby points in the latent space.
You get some sort of regularities, gain insights on your data that would be difficult if you try to do a PCA on the original data, you have manipulated your original input data that lies on a complex original input space and projected into a smaller dim space, smoother, consistent and capturing only the relevant factors of variation.
You can use the denoising trick to obtain something even nicer: Black and white can be considered noise, the noise process of a colored image, train your autoencoder with versions of an image that has been make black and white, feed the black and white input to the autoencoder, and ask the autoencoder to reconstruct the colored image in output.
Nobody tells you that the input needs to be from the same modality, infact you can leverage this idea that you can build autoencoder easily, and build an autoencoder for a certain modality, e.g. image, and another for another modality text, independently, then get rid of the decoding part, make them work together on a fusion layer which can be obtained using a Boltzmann Machine for instance, then you gain automatically a tool to indexing images using text or viceversa, if you have images, you feed your image in the image modality, you encode it, you encode it , encode it, and then starts your Gibbs sampler that decodes back the part relative to the other modality and then from this hidden representation you use the decoder to go back into the original modality one, so from an image you get a text that represent it, or viceversa, you plug text in the respective modality, you enocode it, enocode it , encode it, get an

high level activation on the fusion layers, that generates an encoding for the image, then you look into your collection of images, all encoded at the last layer of the encoder of the image, you use Nearest Neighbor matching you find the nearest image to the encoding that you have, and you retrieve that one, that would be the image retrieved in response to a word, if you supply an image the model generated tag such beach, sea, surf, strand,...whatever. If you supply a certain number of input tags, then you can retrieve in your data collection some images you have indexed. So, you can use autoencoders to generate compressed, highly-informative representation of you text/images to for instance search to it, classical thing we do in information retrival, you generate informative representation of your information and you search it instead of the original piece of information.
You can build multimodal autoencoders by training independent autoencoders for each modality (e.g., images and text), and then connecting their latent representations via a shared layer — often modeled by a Boltzmann Machine.

Here's how it works:

Step 1: Train separate autoencoders:

One for images: learns how to compress and reconstruct images.

One for text: learns how to compress and reconstruct textual descriptions.

Step 2: Discard the decoders and connect the encoders via a shared latent "fusion" layer (e.g., a Boltzmann Machine or another neural layer). This layer learns a joint representation of both modalities.

Step 3: You can now perform cross-modal retrieval:

Given an image, encode it → sample from the joint layer → decode to generate a textual description.

Given a text query, encode it → sample from the joint layer → find the nearest image encoding in a dataset and retrieve the matching image.

This enables:

Image-to-text generation (e.g., tagging or captioning images).

Text-to-image search (e.g., finding relevant images from tags or captions).

The key idea is to align latent spaces of different modalities in a shared representation, allowing information to flow between them. Autoencoders used also in anomaly detection, they are as good as you like in whatever info you have, e.g. autoencoder over images you can make more good at reconstructing images just need to provide more hidden neurons, more params, more samples, ... but you can do it. Imagine that you can train your autoencoder on data that comes from normal behavior, normal data, you train your model, you compute the average error that your model does on that reconstruction of normal data, let's call it tau. You supply your model with new data, and check if it reconstructs correctly, within the tau margin of error the given data, whenever your model receives an anomalous piece of data won't be able to reconstruct it, cause it has only seen and has only learnt to reconstruct data there is consistent with the relevant factor of variation of normality. I will reconstruct it, but with high error, higher than the margin of error that you measure on your training data, so you can identify outliers/anomalous data. Totally unsupervised, becuase one of the problem in anomaly detection is that you can define a model of normality,

but defining what is non normal is difficult, cause you have few examples of non-normal, and most of the time you have examples of existent abnormality, but don't know about future abnormality. To make the things more robust from a statistical perspective rather than having a threshold you can have a statistical test of falling within the confidence interval etc etc but still the point is that you use the reconstruction error of the autoencoder as a sort of proxy to determine if you have abnormality, works for sequential data deciding if a transaction is anomalous for that credit card, it works also with images, you can use these models on CT scans (TAC) to identify which pixels in the image contains a tumor, because if you supply only picture of CT scan of normal brain, the autoencoder will have hard time not to reconstruct not the overall image but specif area of the image, by computing the localized reconstruction error you can get an idea of where is localized the anomaly in which part of the image.

16 Convolutional Neural Network - CNN

Well known model in image processing but can work with other type of data including sequences. An excuse to introduce *weight sharing*, and how you can reuse some symmetries in the data to design NN that are compliant with those symmetries, that have an inductive bias that is well targeting some characteristics of the data. Very important when the nature of the problem meets this inductive bias.
CNNs are behind several machine vision task, vision transformer are inspired by them. Object **semantic segmentation**, *you identify the relevant object in an image*, you don't just put a bounding box around them, but we color them, segmenting them in specific pixels according to their meaning. Very specialized task, not only recognize the visual content of an image, we want to recognize all the objects in it and not only where they're roughly located, but segmenting them out in the specific pixels that make it. CNNs make life of machine vision miserable after 2020, before people do machine vision with different things, they have to throw it all away. CNNs 2020 breakthrough in computers vision, Alex Net solution submitted to a Machine Vision competition in 2020, wins with a large margin 12% points over the runner up, supremacy of the NN in machine vision.
The intuition behind CNN are rooted in much much earlier research, Neocognitron, 80' Fukushima a Japanese researcher, proposed NN inspired to how the visual cortex is organized, at least in the early part, the one closer to the retina. Interleaving of layers, simpler cells whose role is recognizing features, and complex cells that pool together/aggregate those features, *interleaving between feature detectors and pooling* like occurs in CNNs, *trained in a fully unsupervised way* (essentially because there were so many layers for backpropagation in a vanilla and naive way to work), so *detectors were trained unsupervisedly* and a classifier on the top.
At the origins, the first CNN to be proposed, was introduced for sequences, initial layer of convolution that looked at subsequences of length 15, so you center

yourself in a certain position in the seq, you look 7 items before you, 7 after you. There are 16 convolutional filters with params. that you learn. Then another bank of filter, than there is the output production, used for seq. classification, trained with backpropr, introduce the concept of *params sharing*, 16 filter of the filter banks, you center your filter at a specific time t, then you will shift it and center at a specific time t+1, but gonna be using still the same param, so you are sharing param between time steps, you share, then t+2, shift but same filter. That is the main intuition behind CNN the fact that you share params across elems, elems of a time series, pixels of your images, nodes of a graph etc ....

Most famous one is CNN for images, LeNet from Yann LeCun 1999, reasonable number of layers, follow the Fukushima structure of the neocognitron, **interleaving** of **feature maps/filter** and **aggregator/pooling** and **final fully connected layer**.

Sharp reduction in the size of your image that means that some pooling has happened, some aggregation of the responses of different pixels, have been aggregated in a single measure, than a final layer of fully connected layers, dense layers, your classical MLP.

Why CNN instead of multilayer perceptron, we can feed an image to a MLP, you reshape an image (32x32x3) into a vector (3072), and feed to a MLP, but problems in term of the scalability in term of number of param, image are high dim. you fed your input to a MLP with 100 hidden neurons the size of the params from the input to the first layer will be 100 x 3072. The amount of elems in the input vector will easily grow very quickly with the dim/resolution of the image. With 100 or 1000 you easily get to bilions of params just with one layer. No scalability in terms of params efficiency.

Invariances: with images typically you want to recognize cats whatever position they are in an image, cats have this ability to move in every position of your images. You want to recognize them irrespectively of the context. And a cat remains a cat whether it's on the top left or the bottom right of an image. No sense in having specialized neurons to recognize cats in different positions of an image, what you have if you use a MLP. The picture of your cat, after you linearize the image into a vector, would probably be broken down into certain areas, the pixel corresponding to the cat will be in a certain area of the input vector which depends on the position. Whenever I'm gonna be inputting this to a NN, this vector will be multiply by the 307200 weights and some of those weights will specialize to recognize the cat in a specific position, if I give you a cat in a slightly different position, and this will map to another area of the vector, that won't match what are the params. that recognize the cat, because the params that recognize the cat are in a different position. You are not effectively reusing the info you have extracted from the picture in a generalizable way, you have to be **translational invariant**, then you need other, with visual info we need a lot of invariances **translational**, **photoluminance**, **rotational** and other forms including **affine**. Need to capture these features that do recognize visual content but that are not linked to the position in the image, to get rid of the positional rep you have with MLP. There are pictures where position is

relevant where do you don't want translational invariance, but in general you want it.
To keep in mind when design a proper NN for vision tasks: the other inductive bias is that similar/nearby pixels tends to have similar properties, they need to be processed together, you should be able to package them together, pixels relations are important, I have to consider the context where a pixel occurs, this become a modeling choice in my NN: **spatial neighbor must not be destroyed**, but MLP or in general when I vectorize the image I destroy the structural relationships, because I put far away things that are originally near, I destroy the structural relationships. Learning problem is much more difficult! We need a different type of NN to work with images.
Solution: *a neuron that can operate on fixed dim. spatial neighborhood of a pixel and allow to reuse info for multiple pixels.* Exact what convolution is doing, operator that slides through your data. A convolution operator between an image and a **filter**, takes a filter and applies it iteratively to every pixels in the image, slide across all the pixels of the image. The number inside of the filter were computed with a Gaussian, Laplacian of a gaussian, Sobel filter, something that filled the number in this filter with nothing more than just a matrix, but precomputed numbers from a function, now those number will be learned with our learning algo, this is the key thing, the filter will be filled with params rather than precomputed numbers.
5 x 5 filter, centered in a pixel, in a certain position of an image, so will be projected on the 5 by 5 neighborhood of that pixel than a point-wise multi. between the pixels and the correspondent elem of the filter, followed by a sum, the application of a filter to a single pixel of the image will give you one number, then you will obtain one number for each filter apposition.
Grey level image is a matrix (N by N) of pixels' intensity, a conv. filter in this context is a matrix filled with param, as many as the size of my matrix, 3x3 filter a 9-dim param vector, called also kernel, **adaptive weights** we learn it. We center our filter (kernel) on the pixel starting from the top left, center our kernel center on the first available pixel, for the moment let's stay inside the image, convo. operation element by element multiplication, sum all together, this will give you a single value the result of the convolution of your filter with the first available pixel of your image, then slide, move on pixel on the right.
Weight sharing: the same weight is multiply by different inputs. Go on until the end, final pixel you can center. The application of your filter across the image will result in another image, a filtered image, called the **feature map**, an image in which things are gonna be enhanced or suppressed based on the kind of info this filter is seeking for. A convolutional filter will learn to fire high on those pixels of the image where there is some relevant information for the task we are solving, cause the params are learned by backpropr on the task, based on the error that you commit when you predict with your network, like the filter we have seen at the beginning, but not predefined, the type of info which they are sensitive will emerge from the data.
If I apply a 5x5 conv. filter to a 32 x 32 image, you get a 28 x 28 feature map. The size of the feature map is reduced, because you don't use padding. **Mul-**

tichannel convolution, image is RGB, not grey-level, 32 x 32 x 3, you define a 5 X 5 X 3 filter that follows the depth of your image, one slice of your filter will convolve with the first slice of your image, the second slice of your image with the second slice of your filter and so on, you obtain in response a 28x28 image as result again, because this is a single filter 5x5x3, there is gonna be an element wise multiplication by the elem of your filter followed by a summation, so you sum across multiple channel.
This is the standard for general purpose image, you can decide to have separate filter, to keep channel separated. But with general purpose image you are trying to make sense of the content of the image, the fact that there is a cat in the image will depend on the mixing of the colors, if you treat the channel independently you never get the color, you need to sum, to mix it. Typically if you define a N by N by X convolution the application of that convolutional filter to your image return ONE bidimensional image, it flatten the channels.
The basic conv. filter shift by one at the time, both horizontal and vertical (**stride** of the filter) but it is a param, does not have to be 1, it is an hyper param, stride 2 jump of two pixels in both direction, or you can set different stride for different directions.
Stride reduces substantially the dim of the output feature map, you do less multiplication, multiplication costs cause image are big, especially with high resolution images (if you haven't subsampled before), then the other important thing in the feature map: things that previously are far away then become near, if the stride is 2, pixels separated by two elems in the original image with stride two will be mapped nearby in the output, if I process the obtained image with another convolutional layer they will be ending up in the receptive field of a single filter, they will be mixed together, the fact of sub-sampling, making the image coarser, allow successive layers to reason on more high level, more abstract, increasingly more abstract/general feature, to reason on pixels that are far away. This is way we introduce pooling, we need ways of making sure that through the processing of my images, in different layers I extract increasingly more abstract feature detectors, to do that i make sure that the receptive field of a filter, the pixel that are span, must grow in time.
If my image is originally 7x7 I obtain a reduced version of my image 5x5, with stride 1 and kernel 3x3 e.g. easily derivable general rule to obtain the dimension (height and width) of the feature map, depending on the stride, size of the kernel , size of the original image, important to size correctly the layer.
To preserve the size of the image, use 0 padding, start my 3x3 pixel exactly on the first pixel of my original image, padding depend on the size of the filter.
$W' = \frac{W-K}{S} + 1, \quad H' = \frac{H-K}{S} + 1$
A convolutional neuron operates accordingly to the convolution operator rather than receiving in input all the pixels of the image, and having a different weight for each pixel of the image, slides through the image and applies the same weights with weight sharing, performs multiplication between the values of the input pixels and weights encapsulated in my convolutional filter, but convolution is a fully linear operator, we need another step of non-linearity. The convolution operator generates the feature map which is a linear transformation that

needs to be subjected to a non linearity. **Element-wise non linearity**, taking each pixels in the feature map and transform through a non-linear function. Transform all the pixels independently, typically we use RELU, saturating non linearity. That's the definition of a convolutional neuron.
CNNs are not homogeneous, you find computationally units with different nature: convolutional neurons plays the role of simple cells. They learn to detect whenever there is a specific feature in my image, They respond highly where is a certain feature in my image. Pooling neurons are the complex cells, aggregate, pool all together the output of the feature neurons, will be placed after convolutional layer (not after every but after), takes in input the feature map resulting from the convolutional layers and aggregate them spatially, take input from subset of pixels in a spatial neighborhood in the feature map and aggregate them in a single value. Subsample to make the image coarser, next representation extracted with the future convolution will be more abstract. Early filters detect for instance edges, later conv. filters detect corners, later detect geometrical forms. Another use is implement another bit of invariance, invariance for instance w.r.t. to rotation.
Pooling operates on the feature map resulting after convolution layer, this gets into a pooling neuron, the pooling neuron defines a neighborhood over a grid, e.g. 2 x 2 neuron, pooling filter is typically applied with striding, e.g. stride 2, to *pool on non overlapping pixels*, whatever falls inside of the receptive field of the pooling filter will be pooled in a single number. The output image will be reduced another time. Because we take a single value for a group of pixels. The single value depends on how define pooling, typically very common choice are max pooling you preserve only the maximum value. The pooling map equivalent of the feature map for the pooling neuron is a substantially reduced representation of the input image where nearby pixels collapse in a single values, max pooling, average pooling, L2 norm pooling, random pooling etc ... Uncommon to use 0 padding in pooling since the principle of pooling is to reduce the size. Typically stride is chosen to obtain non overlapping windows.
Putting all togheter the full pipeline of a CNN: input $\rightarrow$ convolutional filter $\rightarrow$ pointwise non linearity $\rightarrow$ pooling layer, that shrinks the dim of the image and sends to the next layer. Non always you have a pooling after every convolution, often you have a certain number of convolutions then pooling, then again a certain number of convolutions followed by a pooling etc VCG architecture generalizes this concept. When we build a CNN model selection decision about this interleaving and other hyperparams. to choose, non-linear function for the convolution, size and stride of the kernel etc. Pooling is not adaptive typically the param are only in the convolutional filter.
A CNN is an architecture that receives in input an image grey-level or RGB starts transforming it through level of convolution and pooling, convolution and pooling, pooling is whenever things shrink substantially.
From A NN perspective the connectivity/density of convolutional layer is sparse **(sparse connectivity)**, because you have weight sharing across the convolutional neuron, the same set of parameters are used across pixels, so weight connectivity is said to be sparse.

The size of the original image gets reduced along the level, the thickness of those blocks/ the number of slice grows instead, few slices at the beginning, more slice at the end, each slice is a feature map, it is the response to one specific filter of an input image, and one filter is specialized to detect one type feature, at the beginning I have very basic/low level feature (an edge oriented to 45°, or 90°, some blobs of a certain size but very basic and with very few variation when we are very abstract we have increasing amount of complex feature, the semantic become increasingly complex, the number of feature to be detected grows with the depth (the ears of the dog/car, the mouth, the tail etc...). At the end a slice recognizes the ears of the cat, another slice the mouth of the cat etc this feature are fed in input to the final layers, that put all together and tries to figure out if there is a cat in the image, the job of the final fully connected layer is fusing the features extracted by this sparse convolutional layers made of convolutional neurons/feature detectors.

Sparse connectivity VS dense connectivity, dense is a layers of MLP, from the perspective of connectivity: in a dense layer each neuron receives an input from every point in the volume outputted by the convolutional, this for every neuron, high number of params **(Number of points in the volume in layer CL4 X number of neurons in FCL1)**, you pay an high price depending on how big is the volume outputted by last Convolutional layer because there is not sparse connectivity there at the final layer. Things explode. If you don't pool enough that volume becomes big. But you still need many filters to differentiate objects. More slices in each layer because we have not one but many convolutional filter, each with its own size and stride. If we have a single slide it's like having a NN with a single neuron, not very expressive, usually we have a block of filters. (filter banks)

Typically apply convolution between an input image and a block of filters. The number of param in a convolutional layer is $KxKxD1XDk$. Remeber that the application of convolution sums also over the input channels. One slice in the feature map for each of the d_k original filters. After pooling typically I reduce the H and W but D_k stays because typically common use is not to do cross channel pooling, so you preserve the number of features map after pooling.

$(I * K)(i,j) = \sum_m \sum_n I_{i-m,j-n} \cdot K(m,n)$

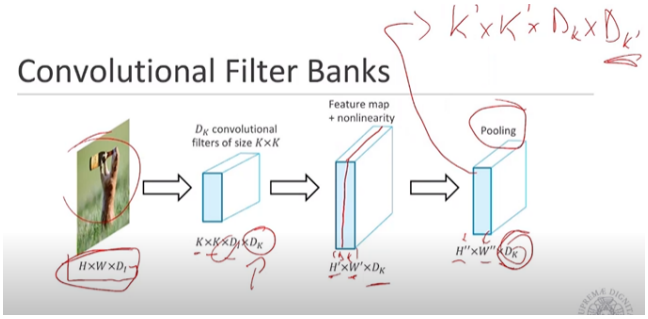


Figure 58:

$(I * K)(i, j) = \sum_m \sum_n I_{i+m,j+n} \cdot K(m, n)$
The convolution of a filter K, with an image I, centered on pixel in position i,j, indexes reasonable for my 3by3 filter e.g., we know the formula with how the the convolutional operator is implemented, it's commutative, creates a smoothing effect etc etc. If you get into libraries, you'll see that convolution is implemented in another slightly different formula, the minus disappears, which is not the convolutional operator as we know, but it doesn't really matter, this is not a predefined filter, the values in the filter are learned, the two formulas are identical if you change the value of the filter, but the kernel self-flip itself, but after all is the correlation formula not the actual convolution.
When I apply convolution there is an input, let's apply it to the seq. ad there is a convolutional layer, one neuron receives in input a limited number of inputs, a neuron receives in input the 3by3 neighborhood of a pixel, not the whole picture, one convolutional neuron for each pixel, imagine we're using 0 padding, one convolutional neuron copy repeated for each input/pixels, each convolutional neuron will not receive input from all the input but rather receive input from a subset of the input, defined by the number of pixels span by the receptive field of the filter,
Different w.r.t. multilayer perceptron connectivity where everything receives input from everything. I do not apply cross channel pooling.
Convolution over sequence: there is an input and a convolutional layer, a neuron receive an input a limited number of neighbors, e.g. a 3 x 3 neighborhood, a convolutional neuron copied for each pixel, that receives input from a limited number of pixels.
Different w.r.t. MLP connectivity where everything is connected with everything. Weight sharing: reusing the same weight for each elem of the input, the same filter for different pixels, reuse the same weight, it's a copy, exactly the same neuron copied multiple time to span across all the input, each of those neurons will have weight sharing and sparse connectivity (not take input from everything but only from a subset). In the dense counterpart full connectivity and all weights are different no weight sharing.
Strided convolution sparsify the convolutional layer, rather than having one convolutional filter for each elem of the input I have one every s, where s striding factor
Pooling receives input from a fixed number of neurons (sparse connectivity) and perform a pooling average or maximization operation. The advantage of a pooling operator is that if I have a feature map sensitive to the presence of a specific feature in a specific position of the image, same feature but present in different position of the image, role of the pooling layer is signaling when notice the presence of the feature we want to detect irrespectively of the occurrence of the feature in a specific position. It will give a bit of invariance with respect to occurrence of your feature in a specific position of your input sequence. We can define pooling over the channel, instead of pooling between pixels, I can pool between channels. Imagine to have 3 features maps, 3 different convolutional neurons, 3 different not copies, and each of them will respond to input oriented in different ways. You can cross pool between those one. The effect of pooling

here is firing a lot whenever there is a fire in the input image, irrespectively of the orientation. Cross channel pooling give invariance w.r.t. to features that are captured in different ways by different convolutional neurons e.g. rotation invariance a filter detect an image rotated by 90°, another 180° etc ...
Pooling has also effect of additional robustness w.r.t. to transformation. **The effect of stacking convolutional layers: naturally increase the receptive field of a neuron**, the amount of pixels (input info) a neuron is able to see. The effect of layering is to give to high level neurons a broader/wider view over larger pieces of input info but in a scalable and effective way, using a hierarchical organization, trees are nice because scale logarithmically, this without pooling, pooling accentuate this fact.
Hierarchical information: if you go on putting one after the other convolutional filters without 0 padding, you will end up with a single neuron that will integrate info from all input image. Compress into fewer neurons a larger areas of the input. At the cost of an ashaming number of layers, pool, use stride ect
CNN from the perspective of backprop is like just any other NN. You generate your error at the output layer, that generates the error, and then the gradient that is sent back to the convolutional neuron, when you do backprop you compute the gradient of the loss w.r.t. weight of the network. A weight is repeated/copied several times, so w.r.t. whom I'm computing the gradient? All of them, each of them contributes to the error, and **in order to get the update equation for a given weight, I'll sum the gradient for the first copy of the weight, the gradient for the second copy of the weight,...** *E.g. The gradient used to update your param w1 is the sum of the gradients over the copy of w1.*
Backpropagating your gradient from layer n to n-1 typically entails multiplying your gradient by the transpose of the weight matrix, you can write this weight matrix, to backpropagate from n to n-1, it is a weight matrix full of repetitions because you have multiple times the same weight and a lot of 0 due to sparse connectivity.
For a 3x3 filters scanning a 4x4 image with stride 1, the feature map is a 2x2, 4 output neurons and the number of input neurons is 16. You can write this matrix with a lot of 0 and repetitions.
From the perspective of training working with CNN is not incredibly different than working with standard/dense NN, we gonna compute our gradient w.r.t. multiple copies of the same weight, because the weights gonna be replicated for all the pixel, so the contribution to the gradient of that weight need to be summed up, across all the pixels that reused that weight. From the perspective of backpropagating the gradient, that amounts to multiply your gradient by the transpose of the weight matrix, but what actually is the weight matrix is a very sparse matrix with a lot of repeated stuff in it.
That convolution up there is essentially equivalent to multiply the vectorized version of the image for the transpose of this matrix, this will return the vectorized version of the feature map (forward pass), the backward pass can be obtained by multiplying the gradient coming from the feature map by the transpose of the weight matrix.

Then is all a matter of making this gradient to matrix multiplication more efficient in consideration of the fact that there is a lot of replication and that there are a lot of zero. The only key thing is when you have weight sharing you have to accumulate the contribution of the weights for all the copies of that weight.
When you are backpropagating the gradient we are somehow inverting the convolution, the usual convolution: 3x3 filter that slides over a simple 4x4 image, since there is no padding you get a 2x2 image in return. If we go back from the feature map to the original image, which is the journey that the gradient does, it turns out that is just another convolution, there is a 3x3 filter that is scanning through an image, at the center I have the 2by2 elems of the feature map obtained before, padded enough to match the size of the original image, then I run a filter over this 0 padded image and this will give me back my 4x4 image. (**inverted/transposed convolution or deconvolution**). Since I'm posing no contraint on the filter I'm using for the deconvolution, the params of this filter can be learned as well, you can device a network that in one direction convolves, and in the other direction de-convolves, gets to a point in which I convolve, i convolve an image till I get to a representation and then I deconvolve, deconvolve until I get back to the original image space. Architectures like this that tries to implement a convolutional autoencoder, in which you start from an image, compress to get your info bottleneck, once you have it decompress again to go back to the original image space, to compress use convolution and pooling, to decompress use deconvolution and unpooling. This thing works also with stride. You "zero pad" the missing elems derived from the bigger jump caused by the stride. Idea under this: **Convolution can as well go from a smaller dim image to an higher dim image**.
LeNet: trained on MNIST digits, inputs are 32by32 images, the output are number between 0 and 9, classification problem, output layer has 10 classes, in the middle: *first a convolutional layer with 6 conv. filter of size 5x5*, because you start from a 32x32 and you get to a 28x28, without padding. Infact you get 6 slides/channels after the application of the 6 filters to the original image. Then when you see sub-sampling (the features map reduce in dim but the number of slides stay as it is) you have *pooling*, 2x2 with stride 2 and infact from 28x28 you arrive 14x14. Number of channels still 6 because pooling operates channel wise. Then an *additional convolution, 16 filters 5 by 5*, then again subsampling you get down to a 5x5 image. Final effect the size of your image from 32x32 to a very tiny 5x5 but the number of filters that you have grows from 6 to 16.
Then you got your fully connected/dense layer of 120 neurons, where is the bottleneck in terms of param: between S4 and C5: dense connectivity each of the 120 neurons receives input from each point in a volume of 16 x 5 x 5 points. If you are not careful you pay an heavy price when you move from the sparse to the dense connectivity.
Before it was difficult to train because they use sigmoid activation function, very bad choice, it doesn't really help the gradient.
AlexNet: game changer in 2012 beats all the other in the ImageNet competition, a sort of convolutional NN on steroid, *we work with larger images, 227x227* (this is the right dim), size of the *kernel 11by11*, quite big, *48 11by11by3(RGB)*

kernels, that produced 48 features map, striding of 4, to 227 to 55, because too many multiplication, *this thing wouldn't fit in a GPU as well, they split images in parts and having different parts of the image in different GPUs*, otherwise the GPUs at the time couldn't fit the all dataset, the all batches and model. *Then 5x5 convolution in large numbers 128 (filters)* then 3x3 convolution in large numbers 192, followed by additional 3by3 convolution (192), quite articulated, different kernels of different size, lot of layers, still maxpooling, dense neurons, a nightmare from the perspective of the number of params. *5 conv. layers*, *3 fully connected layers*, *split in 2 parts (top/bottom)* to make it affordable to the GPU at the time.
Certain number of trick to make it affordable/trainable: key to the success → *heavy use of data augmentation*, images are high dim data, tough from a computational perspective but a dog remains a dog if you rotate by 90°, or translate. If you have a labeled image, if you artificially create another additional labeled image by rotating the image, cropping a part of the image, zoom in a part of the image. 1 milion of image in Imagenet, with 7,8 augmentation you multiply by this factor. Data augmentation is a must in Machine vision now, it is integrated as a function in the Deep learning libraries. For general purpose image, data augmentation is quite straightforward, but if you have specialized image like biomedical image, cropping part of the image might be risky, you could crop the tumor. Use Sobel filter to augment dataset, because it is creating perturbation in the image but it is not changing the shapes, it is highlighting aspect that are relevant in your shape such as edge, or additional noise, colorization etc...AlexNet used it heavily, and to train the monster is **introduced Relu**, saturating non linearity. LeNet used sigmoid, was difficult to train, sigmoid is in a complicated relationship with the gradient, and **AlexNet used heavily dropout to regularize those dense layer**, full of params, I'm gonna be overfitting for certain unless I regularize.
Sigmoid is nice because it is also a probability distribution, but it flattens at +inf and -inf and when a function flattens its gradient becomes 0, even in the rest of function the gradient is really little, the first order derivative has a peak in 0 but it is 0.2, and you multiply multiple times if all of your layers have a sigmoidal activation, the speed in which you reach 0 with your gradient is quite high.
Ideally I want an activation function whose gradient if I multiply multiples times doesn't dissipate nor explode. Magic number? 1, the identity function has 1 as a gradient, but the identity is linear, to create non linearity in the positive part the function behaves like the identity function, if you take the derivative is 1, in the negative part is a flat zero, both the function and the derivative, and in 0 happens the non linearity, non linear function with a nice gradient but it is gonna be having a lot of 0, many of the neurons gonna be 0, can be supported even at hardware with a thresholding function, less then 0 return 0 otherwise it's yourself, easy to implement in chip from a gpu perspective.
The sigmoid function is also non-zero centered, poses issues in term of optimization, gradient that jumps from positive to negative every time, create issues to the gradient descent. Sigmoid can make it zero centered with tanh, slightly

better, it has a nicer derivative, still flattens, because it is a saturating non-linearity, but sometimes you need saturation, like in RNN, otherwise you risk the open loop in which your state it's building, building and building because it's an identity function.
Relu or variation Adaptive Relu, soft exponential Relu, all introduced to mitigate the effect of the negative part, if you have a particularly unfortunate selection of configuration of your hyperparam, your neurons mights end up in the negative area, where they fire 0, if you are particularly unlucky and you fire 0 for a while you have no gradient, you don't change weights, dead neuron, to countereffect this thing you can add a little bit of slope in the negative side, and you can hyperparameter select the hyper param to regulate how much slope do you have that allows to get out from the dead zone.
You can figure out, in your training, if your network is experiencing death neurons, just count the amount of 0 in your activations, having some 0 is ok, it is a form of natural regularization, problem if your neurons never change value that's the problem.
AlexNet has 62.3 Milions of param (only 6% in convolution), at that time was many, nowdays we reason in term of Bilions of param, at that time was kind too much because at the end you hadn't enough data for that number of param. Not superoptimal design due to the bottleneck between the sparse convolutional layer and the dense final layer, each of the 2048 neurons receive input from all the volume of the last convolutional layer which is quite big. 5-6 days to train on GTX GPU. The interesting thing in CNN if you have 62.3 Milions of param only 5/6% of those params are in convolution, but you spent 95% of the time doing convolution, convolution needs to be efficient, you do a lot of multiplication
A couple of years after → VGGNet, not an incredible novelty, just using convolution and pooling nothing new. The key contribution of this model is instead of having this confusion on the number of the size of the convolutional kernel I'm gonna be using, when to put the max pooling operator or the average pooling, I don't know which one to use, let's standardize this: we do only 3by3 convolution, only 2by2 max pooling and we pool after two or three convolution, every time, standardized, we start from an image, we get down with our convolution, then I have the fully connected layer and the prediction, decent in term of performance, much much better than AlexNet, didn't win ImageNet but very influential, but you can still find several pretrain VGG nets around, VGG12, VGG16, VGG18, it changes with the number of layers that you have.
Problem: 140 milions of param, 85% of them in fully connected layers, standardization is good but this was the moment when people realized that we cannot assume indefinite growth in term of memory and GPU availability.
So to resume VGGNet, e.g. VGG16 for the number of layers **standardize convolutional layer**: 3x3 convolutions with stride 1, 2x2 max pooling with stride 2 and NOT after every convolution. 16 convolutional + 3 fully connected layers. 140 Milions of params (85% in FC).
GoogLeNet or InceptionNet: inception module → nice name for essentially convolutions, every design choice in this model is oriented towards make things

scalable, get rid of those hundred of milions of params of the previous design: first bottleneck of the previous architecture, when you move from the sparse to the dense layer, best way to reduce the number of param, get rid of the dense layer, there is no dense layer, just fusing, summing up the convolutional info from the final conv. layer before it is provided to an output layer, you take directly convolutional layer, you aggregate them with your favorite aggregator, you have pooling for instance for that, you can do cross-channel pooling, get a single image, vectorize it, feed to a predictor layer, no dense connectivity, no MLPs, and output the prediction based on that.
There isn't just a single output layer; why possibly *injecting/placing other outputs here and there, to inject gradients*, the gradient generated at the very output layer, propagates for a few layers then it dies, here the second output head injects its gradient that continue/substain the propagation more in depth...and so on. You *generate intermediate gradient*, you inject it in the network by having intermediate predictions, the moment in which you get rid of the dense layer before outputting the prediction you can have prediction everywhere, you need to exercise a little bit of care, because of course your output in the early layers are gonna be wrong with extreme likelihood w.r.t. to the very final output, because output at the end use more refined information to generate the prediction, so they are likely to be less wrong, so the intermediate gradient, are gonna be stronger w.r.t the one that you obtain from the all chain of backprop. but they are gonna likely to be noise, control the noise that you get from this gradient, cut the gradient/crop it/scale it and make it sure that it does not cancel may be the little bit of good gradient that comes from the end, the magnitude of the gradient matters and you don't want to overwrite with noise gradient.
So two interesting design choices: elimination of the dense layers, which in turn allows to have predictions plugged in different position of the network that allows you to have good gradient even if your network is deep. But then there is another thing that attacks the problem of excessive complexity in terms of param: you want more reduction of params without having to rely only the 3 by 3 convolution from the VCG, because in principle having convolution at different size is a richness, you are extracting info from the image at different degrees of resolution, so in principle you want that, but when you have 7by7, 11by11 convolution you pay this with a lot of multiplications and with a lot of params, because it's not only 7by7, it's 7by7 times the number of channel of the input image, that's where the inception model comes into play, the key thing of the inception module is when you focus on the previous layer, with its big bunch of image, with all its channels coming from the previous layer, 190 channels cause I have 190 convolutions in the previous layer, if I feed this directly to a 5by5 convolution, I will have 5by5by190 by the number of filters that I have, too many params, how do I reduce? Before sending the info from the previous layer to the 3by3 and 5by5 convolutions which are the larger one, I do a 1by1 convolution, then I have my previous layer that gets a 1by1 filtering, then a 1by1 filtering followed by a 3by3 filtering, then a 1by1 filtering followed by a 5by5 filtering, a 1by1 filtering but after a 3by3 max pooling, so you get these four different branches that create a multi-resolution view of your image, and

then you fuse them, you concatenate them into a single piece that you then pass to the next layer, and the leap of faith is that the 1by1 one convolution before the 3by3 convolution and the 5by5 convolution somehow will help containing the number of params. Why is that? The image that comes from the previous layers is e.g. 56 x 56 x 64, 64 channels if I want to convolve with a 1 by 1 I have to shape my filter: 1 x 1 x 64. Take 5 of them, if I convolve this 5 kernels with my image the output size is: 56x56x5. Because convolution sum over the channels, the effect of running 1 by 1 convolution is to mix the channels, shrink the images before you pass them to the other convolutional layer that otherwise would have too many multiplications too many param. Instead of sending a 56x56x64 image to a 3by3 filter, I feed a 56x56x5 if you want to use 5 1by1 filter, 1by1 convolution are useful because they don't combine anything on the spatial level, a 1by1 filter looks at one pixel at a time, but looks at the pixel in all of the 64 different interpretations that you have in the different channels, and sums them/combines them with parameters that are learned, in practice *learn how to mix the channels*. Using all this tricks/workarounds give you a network with *5 milions of params, beats VGG 1 over 20 params*, if you are smart you can make CNN work on a budget. 12X params than AlexNet.
GoogLeNet rebranded as InceptionNet, you can find several version v2,v3,v4 change only the amount of filters that you have, you have more filter, bigger filter, including the use of **Batch Normalization**.
Batch normalization is a layer, something that you can throw in the middle of a NN, you have two layers, and you can decide that between this two layers you want to put a *batch normalization layer*. When you are dealing with very deep NN and you are training with minibatching, *you changes the params between minibatches* and *in minibatches you've different samples*, this 2 things all together might lead to the fact that the distribution of activation of layer L, there is a peak in the distribution of activation (in the initial part consider the activations as a vector) and then gets to 0, some neurons in the initial part are more active (there is a peak of activation) and the ones in the final part of the layers are less active, this for minibatch 1, for minibatch 2 two it is the opposite, flat at the beginning, and high at the final part, this per-se is not an incredible problem, unless you realize that layer L then gets in input to layer L+1, layer L+1 expects that L in batch 2 behaves as in batch 1, but it doesn't, it has a different distribution of activations, this thing confuse layer L+1 when updating its param on batch 2, *internal covariance shift*, this change in distribution of the activation, which might make learning not exactly smooth. When we're training the NN we are normalizing the input, the input to the first hidden layer, why shouldn't we be normalizing the input to the L+1 layer, use the same principle to say let's apply normalization not only to the input we fed to the NN but also to the input that feed into the L+1 layer.
How does it work? B is a generic minibatch, we compute the activation of layer L, consider the activation of one neuron in layer L for minibatch sample i, i take the average for the activation of a neuron over a specific layer L, the summation runs over the samples of the minibatch. I take a sample, from the minibatch, plug into the NN, propagated the forward prop. until I compute the output

of the layer L and save it, do the same for the second sample of the minbatch, the third, the fourth, the fith and so on, then compute the average activation of that neuron for the minibatch and the standard deviation of that neuron for that minibatch, then I standardize the output of the l-th neuron by substracting to its actual output the mean and dividing by the variance/standard deviation plus epsilon to avoid division by 0. Now I get a zero mean, unitary variance, standardized output from that neuron. Rethinks this done for all the neurons in a layer L, in vectorial form, you get that their neuron are standardized, their activation gets zero centered with unitary variance, if we think about our neurons activation distribution as a vector we obtain a normal/gaussian.
This is the standardized output of layer L, I send it as an input to layer L+1, layer L+1 when moving between batches will always see distribution that are 0 centered and with unitary variance.
Modification: i take my rescaled output and before sending it to the next layer as an input i multiply by γ and I add β, gamma and beta are params learned by backprop, the effects of this param. is exactly that of canceling if needed the normalization, gamma can cancel the effect of the standardization, beta can cancel the effect of the subtraction of the mean, we are giving the next layer potentially a standardized input but the backprop can decide to de-standardized if needed, you don't have to keep it 0 centered.
Batch normalization is a scaling/normalization layer, but it has parameters trained by backprop, the effect of batch normalization is on one hand is a sort of smoothing of the optimization problem but it is also used for regularization.
$\mu_b = \frac{1}{N_b}\sum_{i=1}^{N_b} x_i, \quad \sigma_b^2 = \frac{1}{N_b}\sum_{i=1}^{N_b}(x_i - \mu_b)^2, \quad \hat{x}_i = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \epsilon}}, \quad y_i = \gamma\hat{x}_i + \beta$
ResNet: introduced the era of the Ultra-Deep network, ResNet152 has 152 convolutional layers, why they manage to train such a deep monster? They start to think why the training of these networks does not work, is it because the task is too complex? Let's use a dumb task, let's train a model that receives in input an image and generates in output an image, even without going through an information bottleneck, come on, at least at training time it must be able to memorize the image and reproduce it in output, if you do it with 20 layers fine, with 56 layer not fine, even in training, it's not a generalization problem! Not a problem of complexity, it's purely a problem of backpropr, the network cannot transform through the layers decently enough, cannot learn a stupid transformation such as do nothing to the original input.
If I want to have a NN that is naturally capable of solving this problem, even without learning, what are the solution, to make floating the input to the output with some jumps between layers, **residual connections/skip connections**: the idea is what if I create a NN where everything that enters a certain pack of convolution, is called x, an image, x goes through the usual convolution and gets transformed in f(x), the result of applying some filtering to my image, the result of the application of the convolution to image, if I want to implement the identity mapping, the solution adopted so far is let's hope that f(x) learns to be the identity function, but wasn't working, to make it more keen by design to learn the id function, I use a **bypass connection** and I say that in output

at the residual block I get x + f(x), this is naturally identity function because x enters and it is available at the end, f(x) has to focus on doing little, because x is already been copied.
Think about a lesser stupid task, in general when working with images, through the transformation that are operating by the convolution, what you are actually doing is taking an input and modify a little bit every time you go through a layer, most of the time you want to preserve within the image and add additional elaboration extracted from the filters, so the residual connectivity will allow to do it, *preserve the structure of the image with the skip/bypass connection x*, while allowing *f(x) to focus only on understanding the incremental operation that needs to be learned and added to the general structure of x*. It is a much simpler problem that needs to be learned by the transformation f. This is the effect on the forward pass, alleviate the burden of the function f, the convolution does not have to waste param to store info about the input, *when backpropagating the gradient flows backward*, there are two paths, one that goes through the convolution and one direction that flows directly, the activation function here is the identity with derivative 1. The gradients there flows freely and rejoin with the gradient that goes through the convolution. Skip connection allows the gradient generated at the output to jump and flow freely backward without being hampered and integrated between time with the little gradient that you have in the between blocks. This thing if you push to the limit has a continuous interpretation, it is a differential equation, x' = x + f(x) where f has an interpretation of dx so x', this is actually pushed to the limit, any transformation done by this residual block has been an incremental infinitesimal modification to x, x is the neural interpretation of the neural network, in principle imagine that this block shared the weights w, you can replicate this process, copy as many time as you like, till infinite, you get a continuous system of ODE that describes how an input plugged at the beginning of the network, an image, gets transformed through neural representation in time, where time is essentially every layer that has a skip connection, if you push through the limit, time is no longer t+1, t+2, ... but it is a continuous time and you get a fully continuous interpretation of how h, a neural embedding of an input, evolves continuously through time where time is actual layers.
You can actually interpret layers as a discretization of a system of ODE, in which you decide to have a layer each time you discretize your system of ODE, using this intuition you can study the property of a NN as you study the property of a dynamical system including the fact that you can keep it stable, non dissipative, or chaotic → Neural ODE.
Residual connection are useful to smooth the forward process and help non dissipating the gradient, very useful with very deep network. *Residual connectivity can lead to instability due to the variance between the layers that builds up, after every skip connection. the variance resulting at the end of the skip connection will increase depending on the depth of the skip connection*, it has been showed that this continues copy of the input increases the variance, of course large variance means difficulty in optimization.
To control the variance standardize, use batch normalization, residual connec-

tions and batch normalization are usually used together.
At some point you might want your NN to run on very power/computationally constraint devices, having 5 bilion params and a lot of multiplication might not be ideal, so you want to further push on the reduction of the multiplication and params, that leads to a class of models called **MobileNets**: a class of models that takes your usual standard convolutional filters, I have N dK*dK filters, that operates on M input channels, a lot of params and a lot of multiplication, because whenever you convolve with an image that has size D1xD2 the complexity of that operation is D1xD2xDkxDkxMxN, that's the number of multiplication that you have, you wanna reduce it, using smartly 1by1 convolution, you decouple the M factor with the N factor, you run the DkbyDkby1 convolution repeated M times, and the result of this is fed to 1by1 convolutions depth M repeated N times, this decouples the previous long multiplications in two terms that are summed, reduces the number of param and the number of multiplication, typically used in Mobile device.
MobileNets introduce a computationally efficient alternative to standard convolutions by using **depthwise separable convolutions**. In a regular convolutional layer, the cost of applying N filters of size $d_k \times d_k$ over an input with M channels and spatial dimensions $D_1 \times D_2$ is $D_1 \times D_2 \times d_k \times d_k \times M \times N$, which can be expensive on resource-constrained devices. Depthwise separable convolution splits this operation into two parts: a *depthwise convolution*, where a single $d_k \times d_k$ filter is applied to each of the M input channels separately (cost: $D_1 \times D_2 \times d_k \times d_k \times M$), followed by a *pointwise convolution*, where N filters of size 1×1 are used to mix the M output channels (cost: $D_1 \times D_2 \times M \times N$). This reduces the total computational cost to $D_1 \times D_2 \times (d_k^2 \times M + M \times N)$, which is significantly lower than the cost of standard convolution. This trick allows MobileNets to run efficiently on mobile and embedded devices with limited compute and power.
VGG huge, slow and not incredibly effective, the ones that work better are Inception and ResNet, best trade off between number of params/accuracy, lot of inductive bias more than a VGG but inductive bias is good if it is the right one. MobileNet are incredibly efficient, you might be willing to sacrifice some performances if they do not fit in the hardware that you have.
DenseNet is similar to ResNet but rather than summing they're copying. DenseNet retains and forwards all previous outputs by concatenating them — instead of combining them destructively (like summing in ResNet).
What thing you're ending up doing with CNN or with any NN you're gonna be using from machine vision, you re never gonna be training from scratch anything serious, **reusing pre-trained model or foundation models**. You use **transfer learning**, I take a NN that somebody else has trained on a huge amount of data, possibly on a completely uncorrelated type of data, e.g. general image, then I take all the convolutional layers, get rid of the dense ones, replace them with new dense one and retrain the new dense layers with a specialized small dataset e.g. biomedical images.
In the end conv. filters are filters, especially those closer to the input, they learnt to specialize on reusable knowledge like presence of edges, blobs things

that make sense for general images and specialized image, the things that are more specialized are the deeper layers, the one closer to the task, to the output, those you typically want to get rid of them, overwrite them or fine-tune them, typically, but depends on the size of your dataset, you can freeze the convolutional layers, every framework for DL allow you to decide which layers are adaptive and which one are kept frozen during back-prop, you can stop the gradient through them, and only train the one that are closer, or maybe the later convolution, it depends on the data you have, you can also model select different configuration of the freezing.
How is data represented in your Neural representation of the CNN, given my input data, my images, I encode them through my CNN, the last layers before the prediction layer encodes the summarized info extracted by the CNN about the input image, I can collect for all the images of my training set, 1024 dim vector, one for each image, PCA or t-SNE and project this into a 2dim space, then I plot nearby images that in 2-dim projection of the embedded neural representations are near, what I can discover is that the network has learned to put together things that are visually very similar (similar color), or that contains similar animals. But what about the intermediate layers? Visualize intermediate layers is not that easy, the dumb version is visualize the kernel weights, the kernel weights are 3by3, 5by5 things, I can look at them and see if they make any sense? they make reasonable sense for early convolutional layers, in which you visualize filter that detects if you have an edge oriented in a certain way but not informative.
I need something smarter, we human interpret things decently if they are represented in our perceptual space, the space where the image is taken, *the feature map is a good representation for the neural network, not for the human, a good rep. for the human will take that rep and map it back in the space of the image, to understand how that rep. maps to certain feature of the image.* Can we do that? We need an operator that from the conv. layer brings us back to the image space $\rightarrow$ transpose convolution /deconvolution. To understand which features are discriminated/detected by a certain feature map i want to map the activation of the convolutional kernel back in pixel space, this requires to reverse convolution, to do that simply use Deconvolution.
DeConvNet: you have your nice little CNN, trained to do all your classification stuff, in order to interpret it you attach the equivalent deconvolutional NN where in place of the pooling you have the unpooling, maxpooling selects the value corresponding to the maximum value in a neighborhood of five, unpooling copies that single value in all the neighborhood, you need a transfer of info between maxpooling and unpooling but it is only the index of the winning elem, and to invert the convolution I have the deconvolution, my CNN through all the conv. layers gets me to the dense representation and from that dense representation i can predict the task, Deconvolutional Neural Network starts from that and from the dense rep. start rebuilding a picture in output, it is clearly an encoder/decoder architecture, I can use this to inspect the layers, I plug an image I get to a specific layer in which I'm interest into, to a specific kernel i'm interest into, e.g. i'm interested in kernel number 42 because it has

all the answers, so to get an interpretation of kernel number 42, I zero the slices of all the feature maps except 42 and then i use the DeConvNet to reconstruct what has happened or viceversa, i keep only the others and zero only the 42, and see what happen in the image space, doing so you start understanding what are the areas of an original image to which specific filters focus on.
Even use to understand what are the image patches that respond more to a specific filter, the image patches of the original image space that respond more, e.g. filter responds to a triangular pattern or dominance of a certain colour, this for layer one, low level, this are still interpretable, if you go to layer 2, start becoming psychedelics, difficult to interpret, if you go on the image areas you start understanding what are the features the filter is actually looking into. You can realize that there is a filter whose job is to recognize wheels of a car,..
Another thing you can do to understand, is *introducing alteration and measuring effect of that alterations* $\rightarrow$ occlusions, classical in interpretability, you have an intervention on your network in causal terms, and you see how things change, based on that intervention, for instance you can analyze the heatmap that occur when you slide a gray patch on the image, and you check how much the position of the grey image change the likelihood of predicting that particular class, if you position the grey patch in an area of the image rather than in another one, the accuracy of predicting the right class label (pomerarian) drops because it's a blue area, if you position in the red area the network will keep saying oh it is a pomererian, it means that the network is focusing when predicting on the central part, the face of the dog.
Dense CNN, they have skip connection, but in the ResNet whenever skip connection whose coming I know that I was summing the input of the skip connection with the f(x) from the convolution, in this case instead I'm adding a channel. The input to each layer is copied as an additional channel, one to everybody, *the specific layer gets copied in all the future layers*, it's a way to fast forward information, *to integrate low level info available at the early stage of the network with high level abstract info available at the end of the network.* Again you have the nice effect of having the gradient flows.
Causal Convolutions: whenever you want to predict something in the future, imagine that you have convolutions in time, no longer dealing with images, but dealing with sequences, we can have CNN working on seq, my filters instead of being 3x3 are 3x1 e.g., or 2by1 convolution, encode the current input with a convolution that also looks to the previous element, it's a causal convolution because the filter instead of centering on one time and having observation on the right and on the left, I only have observation on the left, because I can only observe the past, not the future when predicting, a sort of asymmetric convolution.
Here we are trying to generate an output prediction that can be for instance the next elem in seq. and we use 2by1 conv. at each layer. Many layers are stacked and the field of view in order to output a prediction include more than two elem, in order to make 1 pred. I am capable of looking 3/4/5 time instance in the past. Ok, but very short past, to look more far away in the past I can have 5by1 conv, but then I have too many params. Is there any smarter ways

of doing it? Yes, I delete my convolution, so I will keep having only two convolutions, but when running these convolution instead of taking the current and the past input I take the current and I skip the immediate past and I take the thing immediately before, like striding, you jump one thing, but with striding you don't predict for a specific pixel or elem of the time series, here you convolve every point of the time series what you dilate is the kernel, you don't apply the kernel continuously but with jumps. There is a dilation factor: 1,2,3,4, but they are all 2by1 convolution, and if you do like that, whatever it is outputted I'm able to see the whole history in input. Can I obtain the same effect otherwise? Yes, you keep having only 2by1 convolutions without dilution, but then you gonna be needing 20 layers, this is a way to keep the size of your filter small, the number of layers contained, while giving a lot of context in order to make the prediction.
To recap, causal convolution to prevent a convolution from allowing to see in the future, the problem is the context size that grows slow with depth. To tackle this problem Dilated Convolution:

$$(I * K)(i, j) = \sum_m \sum_n I(i - lm, j - ln)\, K(m, n)$$

where l is the dilation factor, at the first hidden layer dilation factor of 1, second hidden layer dilation factor of 2, 3° hidden layer l = 4, final hidden layer l = 8. Similar to striding, but size is preserved.
Semantic segmentation is the problem of getting an image in input and rather than predicting one lable for the whole image, I *predict one label for each pixel in the image*. Not longer looking for an embedding of an image into a dense vector out of which I can generate a prediction for the full image. I want to keep an embedding for each pixel and predicting a class for each pixel.
How can I obtain this? Dumb naif thing, keep using my classical CNN, and when i get to my dense layer i make sure to have in the dense vector the same number of pixels as the output image and then I reshape as an output image, but pixels are a lot, too many params. Other option **use deconvolution, encoder/decoder architecture**, I have a CNN up to the point in which I reduce the size and then apply learned deconvolution to generate in output a segmentation, the loss to train this thing is your classical classification loss the sum of the classification error in each of the pixels trained by backprop. In Deconv. maxpooling indices transferred to decoder to improve the segmentation resolution. During encoding, max-pooling is used to reduce the spatial dimensions (downsampling) while keeping the most salient features. But this operation loses spatial information—it only keeps the maximum value, not where that value came from. In the decoder (upsampling) stage, we try to recover the spatial resolution. If we don't know where the maxima were in the original input, any upsampling (e.g., using transposed convolution or interpolation) becomes a best guess, which leads to blurry or misaligned segmentations.

However, if we store the indices of the max values from the encoder's pooling layer, we can use those during unpooling to put the activations back in their original positions. This technique is often used in models like SegNet.

U-nets because of the shape of the model, has an encoder on the left side and a decoder on the right side, you are predicting a pixel mask in output, you have classical convolutional layer, every time you have red arrow the image reduces its size, so those are pooling layer, then you get to a point in which you have the equivalent of the dense representation, a vector which represent the all image content obtained by iterative application of conv. layer and pooling, conv. layer and pooling until you get to a single vector, to which I start applying up-convolution and inversion, I recreate the original image, I get back to the image space, very similar to the DeConvNet, there is also a link that transfer info from the convolution at the left side to the convolution on the right side which is comparable, it is a residual connection, you want to recreate the shape of the image at the final layer of the upconvolution, do you want to waste param to recreate the shape of the dog? NO!, Let's use a residual connectivity to do it, and the same you do at every level, you put in communication levels in such a way that the deconvolution is informed by equally adequate info coming from the convolution, again this has also the nice effect that when you generate the gradient and you backpropagate through the skip connection, you get a straighthrough passage for the gradient that doesn't have to go through the whole network. And also dilated convolution works nicely here. Use Dilated Convolutions: always perform 3x3 convolutions with no pooling at each level. Context increase without pooling (changes map size) and increasing computational complexity.
From 1 Prediction for the whole image to 1 Prediction for every pixel in the image, reshaping doesn't work very well, too big dense layer waste a lot of param, best way look to skipping things always as much as possible down to the level of convolutions, that may not be multiplication efficient but it is param efficient cause it use param sharing, $\rightarrow$ use Deconvolution architecture that copies in output exactly what they have in input but the pixels from the original RGB are labelled in output with their meaning. Recalls an encoder/decoder architecture but it is not autoencoder cause it is decoding something slightly different, convolution compress the original image and up-convolution that leverages this compressed and refined and extracted info, bringing it back to the original size of the image and in the end the reconstructed original image with the labels in place of the pixels, there is a softmax in each pixel, you want to predict a class for each pixel, both convolution and up-convolution are learned. If you have pooling you have to transfer the pooling indexes when you unpool/unsample becuase you need to give responsibility to the winning pixel in the max competition when you reconstruct output.
U-net just an encoder/decoder arch. at the bottom there is the information bottleneck, all conv. layers, red line a pooling layer, shrink the image received in input to a very compact representation, a single vector and then the right hand side again plays with up convolution, back to the original size where we can reconstruct the single pixel and assign the single pixel a class. Characterizing aspect of this model: the *residual connection between the encoder and the decoder*, provides finer grain low level info, little abstract, you transfer this info to the neural layers that have the very abstract info because it has gone all the

way through the information bottleneck so they have only the abstract info, you integrate the high level abstract info in the right hand side with the low level info coming from the decoder, fuse them and obtain the semantic segmentation, in the backward path when backprop the residual connection helps, because gradient flows freely in that direction cause their activation is the identity function.
Additional approach Dilated convolution, introduced for seq. case in which we have a seq. in input, and since we wanted to have my prediction in output to have a big view over the original input seq. while keeping really small convolution rather than convolving the current point with the immediate predecessor, convolve the current point with two predecessor before, or four before $\rightarrow$ dilution factor. The same thing can be projected into a bidimensional problem on our images, 3by3 filter centered on my central pixel of my image, level 2 is still a 3by3 filter that rather than be compact is dilated, still centering on the same center pixels, but rather than apply my convolution to my direct neighbor, I jump of 1 neighbor, the effect is that you enlarge your ability to view over more pixels.
Layering/Stacking dilated convolutions make sure that when I go up one level my convolution, I am enlarging a lot the spatial receptive field.
Rephrased in the problem of semantic segmentation, I'm going to be predicting the class label for that pixel considering info of all that area, and this apply to all the pixel. Different from strided convolution, which is unuseful for the purpose of semantic segmentation, because we want to convolve with every single pixel, here we are convolving fo every single pixel but when convolving we are skipping pixels. You get a broader field of view but doing lesser multiplication because you can keep your filter smaller.
Semantic segmentation is about segmenting out the different semantically consistent pixel into an image. Counting of object is not included, I'm just trying to figure out if a pixel is from a certain type of object but I don't know if there are 2 of them, 3, or what is the respective geometric relationship it's purely a segmentation with a semantic in it. **Object detection**: to count how may object given an image, for only the objects of interest, meaning I'm not interested in identifying the ground or other confounders that might be in the image, only focus on object of interest and put a bounding box (regular rectangle), sorrounding it, has closely as possible to the object, a label on the top of it and possibly a confidence level.
Reference architecture: **Faster R-CNN** a family of models, the characteristic is *reasonable trade off between speed and accuracy*, implements *a two level process*, **the network has a first step in which identifies where there should be bounding box**, and **a second stage in which giving the identified and cleaned up bounding box, try to identify what's inside of it**. There are *single stage detector*, the most famous is **YOLO-You Only Look Once**, fuse in a single stage the identification phase with the classification stage, you gain a lot of time, but lose accuracy. Faster but influenced by unbalancing in the object visual class, so it has to predict majority classes over minority classes, you need to be careful on how your dataset is structured.

Faster R-CNN is a two-stage object detection framework that improves both accuracy and efficiency compared to earlier models like R-CNN and Fast R-CNN. It begins with a convolutional neural network (CNN), which extracts a deep feature map from the input image. On top of this feature map, a *Region Proposal Network (RPN)* is applied to generate candidate object proposals. The RPN slides a small window over the feature map and, for each location, outputs k anchor boxes with different scales and aspect ratios. Anchor boxes are pre-defined bounding boxes of various sizes and aspect ratios used as reference templates to detect objects at different scales and shapes. For each position on the feature map, the model places k anchor boxes (e.g., tall, wide, square). The network learns to adjust these anchor boxes using bounding box regression so they tightly fit actual objects in the image. They act like starting guesses for object locations. Each anchor is evaluated for objectness (binary classification, evaluate the probability that a proposed region (e.g., an anchor box) actually contains an object (as opposed to background)) and refined with bounding box regression (coordinates and size). The proposals are then passed through a *RoI (Region of Interest) pooling* layer, which crops and resizes features corresponding to each proposal to a fixed size. These features are then fed into a fully connected network to perform final object classification and further bounding box refinement. Faster R-CNN significantly reduces computation by sharing convolutional features between the RPN and the detection head. However, as a two-stage detector, it remains slower than single-stage methods like YOLO, but generally offers higher accuracy, especially on small or densely packed objects.
How does it work: put a CNN on the top of your image, you need a feature map, at some point the outcome of a pipeline of convolutions is a feature map, that is then fed in input to a piece of the network that is called the region proposal network that generates a fixed number of proposal or regions: the number of bounding boxes and their position: 2 driving factor: 1- gives you an idea of in which pixels there should be a bounding box, which it's driven by a classification loss, then a regression loss that measure how well you are doing in actually positioning the bounding box, it means that you need to generate an x and y position of the top left corner, the size and the confidence on the presence of the bounding box. You are gonna be having multiple BB overlapping and at some point to decide to if to keep them all, whether to fuse them, whether to throw some away, after the initial region proposal phase there is ROI pooling, region of interest, a phase in which you have 2K candidates, you start deciding which to keep, and which to thrown away.
Then to the surviving bounding box you apply classification, will give you in the end your classification loss about the detected object and the bounding box regression loss about the exact position. Phase 1) Trying to position a fixed number of BB in the most likely position in the image, where the most likely is induced by the feature map, the fact that the feature map at some point will have area of higher intensity intepreted as area of more likely to have an object there, and region proposal network will learn how to overlay those anchor boxes where they believe they should be more.
This can be extended rather than just having the BB what if I get the BB

together with the semantic segmentation, the BB tells me that exists a certain number of countable objects, the semantic segmentation tells me I don't want just the rough position of the object but exactly the outlining of the object at a pixel level, on the top of the level on which you have already the classification loss that classifies your object and position the bounding box where the object should be you add an additional model that does the job of the dilated convolution, try to segment. From the feature map tries to label each pixels through U-net, dilated convolution, you put a softmax in the end and you try to label at that level the pixel also knowing the classification, this extenstion of faster R-CNN that does also the semantic segmentation is called the masked R-CNN Caffe has a structured JSon-like way to represent the architecture of the NN (protobuffer) together with a standardized way of pickling/save the weights. They started the ModelZoo, a repo where pushing all the trained model, started the epochs of downloading a model, reusing the model, fine tuning etc...
Batch normalization works nicely with images but not as nicely with seq. data. Why? I work on minibatch and computing avarages over the samples in a minibatch, assuming the things you are averaging are i.i.d., images inside of mb are i.i.d. measurement over a sequence inside of a mb are not i.i.d by def they are dependent, when you use batch normalization with recurrent stuff you need to be a little careful, you're essentially averaging over stuff that is dependent, in that case you typically you use **activation normalization**, *rather than averaging over the samples in a mb you average over the neuron in a certain layer.*
Bypass/skip/residual connection also known as jumping knowledge. Convolution works nicely with images, with sequences and also with seq. of images (videos), CNN that applies convolution both on a spatial level of the image and at the level of time, the question is you first convolve in time and then in space or you first convolve in space and then in time, bunch of model that alternates the two. CNN works nicely especially when you deal with short video, action recognition. Video is very high dim, convolution is cheap in terms of params you pay the price of having a limited memory, ability to consider time info, you typically apply to restricted problem like action recognition, classification of biomedical videos, things that last short time.

17 Gated Recurrent Networks

Why we feel the *need of having gates in RNN?* Entails reasoning on the **gradient vanish/exploding problem**, specifically for the RNN, **in RNN gradient issues are particularly evident because of weight sharing**.
Seq. are compound info, made of a certain number of non independent information, the observations has a dependency in time, in the order in which I collect the observations. I want to capture this dependency structure, in the inductive bias of the model that is gonna be processing it. In probabilistic model this is done by taking the first order Markov assumption, the observation at the current time depends on the observation at the previous time and that's

it, provided that I know the early past I can predict the present. With RNN rather than focusing on compressing the history of my past observation, into a single hidden state which is a discrete object like in the HMM, I compress all this history into a vector of neuron activations.
This concept of compressing history into a single value is more general than that, it's not history that I'm interested into it's context I'm interested into so, whenever we are working with compound data, data with dependencies, irrespectively of the nature/form of the data, whenver I'm taking decision about piece of the data I need to understand the context in which that particular observation needs to be interpreted, In seq, the context is the subsequence preceding the current observation, in bidi seq, I will be taking decision about an elem in the middle using the context of what precedes it and what's after it, it's not any longer about history, it is about taking a piece of info and put in into the right context before taking a decision, making a prediction about that specif elem, from the perspective of representation learning this means that I take the central item and representing in a neural space and the neural representation of this item will not be depending only on the specific input but will depend on the input and on the context, neural representation of input conditioned on a specific context, in seq, it is a simple context, cause it's linear, it's either the past, or the future or the past and the future. Further level of generalization, seq. are just an example of compound information, there is plenty of compound info which have a different form, tree: express different form of dependencies, observations are in each node of the tree, i want to be able to learn a representation for e.g. one observation in a node that takes into consideration the context in which that observation occurs. What is the context of a node? Parents, children. Seeking a representation of this piece of non independent information, that depends on this local piece of information and the context (the full descendant for instance or the ancestors or both of them).
Other class of data aside from vectors that is structured data, compound data, data that is made of observation and relationship between them, that require specific modeling, sequence are the entry point, then we will see tree, and then you can push it up to graph, hypergraph/multigraph... the richer the structural dependencies, the more difficult is to come up with a model that is able to capture it and to do it in a reasonable time.
Efficient ways to compute good compressed representation of the context, that I'm gonna be needing in order to make predictions about the specific item, in order to predict about time step t=3, I need a good representation of the context until time 3, good, compact, informative, that allows me to interpret whatever observation will come at time 3 and place in the context of whatever came before, 2 things that play a role in sequential data processing current observation, and the compressed rep. of the context i.e. the state of the RNN, **the hidden neurons of the RNN are the compressor of the context, essentially a vector of recurrent neuron activation** and this thing generalizes to tree easily, **Recursive Neural Network** generalizes the concept of recurrency to multiple preceding context.
RNN paradigm is all about learning how to push this context into a vector,

has different ways of doing it, fully adaptive approach in which the NN has the ability to adapt all the params, randomized approach like Reservoir Computing in which the compressor itself is non-learning, it is just nicely initialized so there is a function that can generate compressed representation of the history and then you learn only the predictor, gated recurrent network is our focus. The compressor function needs to be able to compress into the same size, into a vector of fixed size, context of variable size.
It's all about crafting, again, the proper inductive bias, what kind of design decision we need to take, in order to obtain a NN that is compliant with the requirement we have expressed before: **the ability to compress variable size context into single size representation**, the first thing that we need is **weight sharing across time**, stationarity, cause if I want the ability to take a 10 step seq. being mapped into 100 neurons, but at the sime time can take a 15 step seq. or 100 step seq. and to map in the same 100 neurons→ copy exactly the same set of weight for the amount of time that you need, number of step you have in your seq. **weight stationarity or weight sharing is the leading inductive bias in RNN**. *A lot of copies of weights which is also the cause of the gradient vanish, the reason way it explodes so easily with RNN.* You want the RNN to be easily trainable, we'll see that this won't be exactly the case, infact that's why we use transformer instead of RNN nowdays, although there has been a rediscovery of recurrent approaches also in transformer by Mamba and state space models, because they are easy to train, RNN are nice for this inductive bias of being able to copy its param for an indefinite number of time, **they have the potential ability of remembering everything from the past**, you don't have to set a maximum amount of allowed context, the limit is the number of neurons that you have and the precision of your arithmetic in the calculator, these are the limits in principle. This is due to the fact that we use recurrency, the same thing you do not have with transformer because you need to define the context window with transformer, you need to set the maximum amount of token you can look at, and that is fixed, and you can't say "Oh I want more,", sorry you have to retrain it, but **what you pay with the recurrent approach is that everything needs to be synchronized**, in order to compute the next time step I need to compute the previous time step in the seq, the same does not apply to attention based transformers, they can be computed in parallel for all the elems of the sequence, so this is also a limit when you train, you are less parallel, approaches such state space model, Mamba and alike kind of traded between the two, because they give you the possibility to unfolding indefinitely in time, if you use the recurrent formulation but at the same time they allow you to use a vectorized version that you can use in training with a kernel that allows you to compute everything in parallel.
What are the task that you can define on the top of these sequences: this generalize to structured data of any form: element to element task, you have a seq in input and for each of the elem in input you want to predict something in output (PoS tagging or activity recognition, given the current measurement of the accelerometer predict for every instant if I am sleeping, I am running ... but it's not an instantaneous prediction, you cannot take the decision only

looking at the current reading of the accelerometer otherwise you won't be able to differentiate sleeping from running. Seq-to-item, you look to the full seq and you are interested in a prediction at the end, classical emotion recognition, I give you a sentence you tell me if this sentence is expressing a positive or a negative feeling, angry or happy, single prediction at the end of a full seq. of input. Item to seq. an item can be an image and the seq. in output the caption, the string describing it. Seq-to-seq, different from elem-to-elem I have two sequences, one in input and one in output, but they are desynchronized, there is no syncronicity, classical example is machine translation, seq. in input is a sentence in english, the seq. in output is the exact sentence in german, the input seq could have a different length from the output seq. and a different grammatical structure, what happens at the beginning of a sentence in english might end up in the end of a sentence in another language, the two seq. cannot be synchronized, the only way to deal with them is first parse the input seq. and then generate, and this gives us a first int on what is generation, the second seq.

De-facto starting point for recurrent NN is the vanilla RNN, it receives an input at the time t, it integrates through this function A the input at time t, with some feedback loop in order to generate h_t that is the neural representation of this isolated point x_t put into the context of what came before given by the feedback loop, so what is flowing in the feedback loop is the context which I use to interpret x_t and embed everything in h_t which is the revised version of x_t placed into the context. This context in the simplest case is exactly h_{t-1}, I interpret the current input at time t in the context of the previous hidden state h_{t-1}.

This thing should resemble a NN, there is an input, there is an activation function that generates an output, there should be synaptic weights, a weight that weights the input that get into the hidden neuron W_{in}, and then there is a weight in the other connection that we have W_h, the connection that weights the contribution from the previous time.

These weights are copied multiple time, this is a compacted version of RNN. The exploded version that looks more like a NN: a RNN is made of a certain number of neurons, each of them receive in input the current input x_t , weighted by a parameter for the specific neuron, and there is another input that comes from the previous instantiation, weighted according to W_h, these are summed up, passed through a non linearity, generate in output a version h_t that compounds all the hidden states of all the recurrent neurons and goes to the next time step.

The activation potential in the weighted summation from the contribution for the input and the past hidden state, and the current hidden state is resulting from a tanh of this local potential. This the classical Elman RNN, and then you can use these h_t to generate an output for the whole network, y_t is the prediction generated at time t, in case of elem-to-elem case, in case of seq-to-elem case I just produce this output at time T, when the seq is finished, well, actually you know that the network will compute an output every time, is just that you don't inject the error.

$$\mathbf{y}_t = f(\mathbf{W}_{\text{out}}\,\mathbf{h}_t + \mathbf{b}_{\text{out}})$$
$$\mathbf{h}_t = \tanh(\mathbf{g}_t)$$
$$\mathbf{g}_t = \mathbf{W}_h\,\mathbf{h}_{t-1} + \mathbf{W}_{\text{in}}\,\mathbf{x}_t + \mathbf{b}_h$$

Inductive bias of copying the weights as necessary to adapt to the length or topology of the seq, the application of the model to a seq produce the **unfolding or unrolling**. You take your RNN, that is a sort of compacted version of the NN, that says ok I expect to have an input and a feedback, these are the params associated with the input and these are the param associated with the feedback, then you get an actual seq, you need to unroll that recursive model, you have to copy those params, and instantiate them for every elem of your structured data, seq in this case, this will bring you to the actual instantion of the RNN for that specific seq.
The same set of input and recurrent weights copied as much as necessary, to unfold on all the sequence.
This will map a seq. of arbitrary length to a fixed length encoding, which is h_t a vector in R^k where k is the number of neurons in the hidden layer and irrespectively of how long the seq. is if I read the final h, the h of big T, h_T, it will be a fixed length vector that encode the full seq. indep. of how long it is. This vector h needs to encode, all the info that I need to use in order to make a prediction, for instance if at time 3 I need to infer something about x3 that requires info from the past, I need to be able to make the link between h_3 and x_1, I need that a network learns that it needs to store x1 info into the hidden state and that info needs to last until to h_3. Classical example: if you need to predict the role of the word in position x3, in order to be able to predict that role in the sentence you need to look into the surrounding.
The problem: learning long term/range dependencies, learning to propagate, to leave a trace of x1 in a hidden state, in far way hidden state is difficult, not only a learning problem, it's a problem of how we craft these networks. Making sure that there is a trace of x1 in h_t or h_{t+1} is yes a problem of setting the params in such a way that x1 gets into the state h at time t+1, params should be set in such a way that extract from x1 the relevant information and put into the hidden state and that is a learning problem but there is another problem, provided that this params W_{in} have learned how to extract info from X1 and place it into H, then there is the problem of how this things get to very far and distant H_{t+1}, every time h is updated, the h generated at h1 that contains possibly a trace of X, gets updated in the next step, get rewritten in the successive steps, ... this is forward phase, it's not backward, it's not a problem of the gradient, having multiple rewriting of my hidden states and if I'm not doing something specific, those rewriting can overwrite relevant info that is inside h1. Forward propagation problem: the process that reads the info in input and propagates to the future is not efficient enough, this thing is not gonna work anyway, even if I do not have gradient vanishing, it is a leaky process, every time that I'm updating h, $h' = h + c(x_i)$, it is this process here that is leaky, the

repeated application of this will cancel the information from the previous step. Very similar as having a loss in physical system, if you have a physical system that cannot preserve energy, if you let it run at some point it will run out of energy, if you have a lossy information system if you let it run at some point it will loose information, else said the derivative ($\frac{dh_t}{dx_k}$) will be 0, not the gradient thanks which you learn, it's a forward gradient, you need your system to be sensitive to the inputs, you need your h to be able to keep track/to keep a trace of the input that is introduced as some time for as much as possible if you wanna remember about things.
Exploding and vanishing of the gradient: **gradient vanish** my gradient gets to 0 $\rightarrow$ weights do not change, no learning, **gradient explosion** $\rightarrow$ I am gonna be oscillating a lot, I can miss the minima, I can oscillate around it $\rightarrow$ it's an optimization problem. So the exploding of the gradient is treated as an optimization problem, you smoothen your gradient, you cut it, the vanishing is nasty problem no way of creating info out of nothing.
Gradient vanishing originates where the gradient originates, when you make a prediction at time t, at training time, you output something, on the base of that output you generate an error, and that is your loss at time t, so I need to update my weights, I take the derivative of the loss w.r.t to my weights, the weights (copied) are the last layer, the previous one, the previous again, ... when we have weights sharing we compute the gradient w.r.t. to all the instantiations of all the copy of the weight, the derivative of L_t w.r.t. to a specific weight will be a certain number of derivatives, how many? T the length of the seq. I sum them to get a single gradient, $\frac{dL_t}{dW_{in_t}}$ gives you the contribution of this copy at time t of the weight to the loss, the derivative $\frac{dL_t}{dW_{in_1}}$ gives you the contribution of the copy of the weight at time 1 to the error that you make and you need to sum them all, otherwise you lose info, and *it is exactly this derivative that creates the link between the time step 1 at the time step T, the long range dependency.*
In order to compute the gradient between L_t that is generated at time T, and the gradient of the copy of the weight that is in position 1, backpropagation or better chain rule of calculus, requires that I expand my derivative with a lot of multiplication. I generate the error at the end, I cannot take directly the derivative w.r.t. to the weights, let me take the derivative with respect to the hidden layer for instance that is directly attached to them, fine, then I take the derivative of that hidden state w.r.t. to the previous hidden state, w.r.t. to the previous hidden state, w.r.t. to the previous hidden state, by application of the chain rule you get all those intermediate derivatives $\frac{dh_t}{dh_{t-1}}$ until you get to the point where you need to be and there you can take the derivative of h1 w.r.t. W_{in_1} $\frac{dh_1}{dW_{in_1}}$, these are directly attached and you know how to compute this. A lot of multiplication and this is typically a numerical problem. Consider the derivative of the loss computed w.r.t. to the weights, generic weight, could be the recurrent or the input weight. This entails summing over all the copies of that weight, and all the copies are at different time step so $\frac{dL_t}{dh_k} \cdot \frac{dh_k}{dW(k)}$, first application of the chain rule will tell me that if I need to compute the contribution to the loss of W for a specific W(k), the overall gradient for a weight W is

the sum of the gradient for all the time steps, of the derivative of Lt w.r.t. to k-th copy. $\frac{dh_k}{dW(k)} \rightarrow$ this I know how to compute because it is h_k differentiated w.r.t to the k-th copy of w. By application of the chain rule you introduce h_t

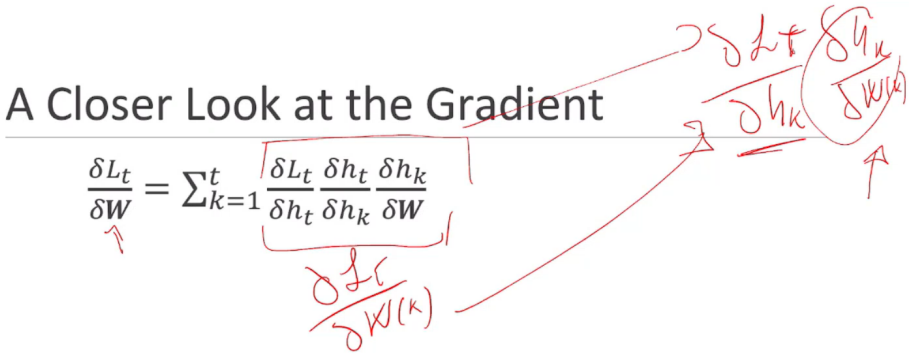


Figure 59:

because you can directly differentiate L_t w.r.t. to h_t. The nasty point is that term number 1 easy peasy, the derivative of the loss w.r.t. to the hidden state, directly attached to it, I know how to compute it, third term derivative of the h_k w.r.t. to the k-th copy of my weight, h_k = sigma(W(k) h_{k-1} + something for the input) where sigma is activation function, I know how to compute it, they are directly dependent one another, ok, fine, so the problem must be in two, a single suspect left, and it's number 2. Number 2 hides a lot of multiplications, because in order to be able to compute the derivative of the state at time T w.r.t. to a state at time k, I will have to unfold it as the multiplication of the derivatives from time T until time k, I can only compute derivatives between adjacents states because h_t = sigma(W h_{t-1} + something for the input), I know how to compute. so the application of the chain rule i.e. the multiplication will be as long as the distance between T an k, so the more distant the k the longer that multiplication. Just to mention: in general we are gonna be taking that derivative with respect to all the weights rather than having a single value, we have the Jacobian, the matrix with all the partial derivatives. The final form for our gradient, the sum overall the copies for all the times os some local derivatives where the local derivatives in the middle are just long product. Now we can focus on that part, and try to understand where could be the problem: we have vanishing gradient, the gradient it's small, it's norm gets to 0, why happens? look ok the norm of the highlighted part and try to understand what are the factors that influence it, and it will be the cause of the fact that this norm is going to 0, by inspecting why that happens we understand what we should be doing in order to avoid that the norm goes to 0, since we're interested in vanishing gradient we will derive an upper bound of this norm of the gradient and we'll see that this upper bound is near 0 if we take some decision, so we gonna take decision to make sure that this upper bound is larger.

This thing here creates issue because I'm concerned about the magnitude of the gradient, that gradient is going to be more articulated than a single value, so what I'm gonna be looking at is the norm of that gradient, to understand why the norm of that gradient goes to 0, so in order to do that I'll focus only on the part highlighted in cyan.

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

Figure 60:

Seeking a bound for the gradient that allows to understand why it gets squashed to 0, h_l is the activation of the l-th layer of a RNN, the activation function of a RNN is typically tanh, the input to my recurrent layer are two, the actual input to the NN at time l weighted by W_{in} and the input that comes from the previous hidden state h_{l-1} weighted by a different set of weights, as part of this long list of multiplication I'll be computing the derivative of h_l w.r.t. to h_{l-1} which can be easily computed because it directly depends from h_{l-1}. That will be something that looks like, the multiplication between the weight vector of the hidden neurons because that would be the derivative of the argument of the tanh times the derivative of the tanh, the first term D is what accounts for the derivative of the tanh, it is just a vectorized version, in matrix form, because that thing it is not a single scalar, but a derivative, w.r.t. to all the params, so it's a matrix, a Jacobian. The Jacobian of the activations is a diagonal matrix, with a structure that comes from the definition of the derivative of the tanh. So that derivatives account for a matrix multiplication the diagonal matrix with the Jacobian of the activation function and the weight matrix, if I plug it in the expression i have before, that will lead me to an expression for which I will look at the magnitude.

$\mathbf{h}_l = \tanh(\mathbf{W}_{hl}\mathbf{h}_{l-1} + \mathbf{W}_{in}\mathbf{x}_l)$

$\mathbf{D}_{l+1} = \text{diag}\left(1 - \tanh^2(\mathbf{W}_{hl}\mathbf{h}_l + \mathbf{W}_{in}\mathbf{x}_{l+1})\right)$

$\frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l} = \mathbf{D}_{l+1}\mathbf{W}_{hl}^{\top}$

$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \prod_{l=k}^{t-1} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l} = \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \prod_{l=k}^{t-1} \mathbf{D}_{l+1}\mathbf{W}_{hl}^{\top}$

The norm of the product can be upper bounded by the product of the norm, and upper bound is what we are interested into, so becomes the products of the norm of the first gradient, but this is a single a value, it is not really harmy, unless it is 0 does not really create issue, the things we should be concerned about is the long product that goes from k to t-1, this is the product of two norms, which is roughly equivalent to take the power of the norms, which is roughly equivalent to look at the spectral properties of the two matrices. **Under reasonable conditions we can approximate the two norms of those two matrices D and W_h^T with their spectral radius, the largest eigenvalue in absolute term**, this is the final upper bound to the nasty part of the

gradient.
When are things going wrong? the derivative of L_t w.r.t. h_t is not a big issue, the problem stands from those powers, $\sigma(D)^{k-1}$ and $\sigma(W_h^T)^{k-1}$, which are the largest eigenvalues, the spectral properties of those two matrices are crucial, will make the difference.
If the spectral radius of those two matrices is 1, we are happy, when we have for instance matrices with spectral radius equal to 1? Unitary matrices for instance, if you have unitary matrices we have exactly what we need so for instance **if my weight matrix is unitary or orthogonal matrix then I'm particularly happy**. The structure of the weights, intended in a spectral term matters because it gives you more or less ability to propagate the gradient, from what does it depends the spectral structure of the matrix? From the values inside of my weights, certain types of matrices that I like and others that I like less, same reasoning applies to D, D is a diagonal matrix filled with 1 on the diagonal and that would make us very happy, because would have the right spectral radius, minus unfortunately, tanh squared whatever...so not exactly what we are looking for, **we are gonna be happy to have activation functions whose Jacobian leads to unitary again orthonormal matrices**, is this the case for tanh? no, because the term on the diagonal that subtract to 1 is non zero. That is the source of the problem: If those spectral radius are smaller than 1 $\rightarrow$ shrinking effect, otherwise the bound grows, if larger than 1, the bound grows to infinity but not really an incredible concern, if the gradient magnitude is large it becomes a matter of proper optimization, when it goes to 0 there is a problem of total dissolution of info.
If we want to have non dissipating gradient: we need to have those sigma as close as possible to one. My weights associated with recurrent connection have a spectral radius near to one, possibly one, and the activation function that we choose has a first derivative that gives rise to the identity matrix. (Hint on why RELU is the solution.)
Not for tanh, not for generic recurrent weight matrix, no way which by pure chance the model learning will give you a recurrent matrix with the properties that you like, you have to enforce them. You cannot enforce them just once, that's a weight matrix, every time you finish going through a mini-batch you update it so you change the values, you can always make a matrix orthogonal it is the price that you pay to make it orthogonal that is a problem, you cannot afford do to it very minibatch.
Instead of having strictly orthogonal matrix I can impose some training loss penalty that make sure that asymptotically that matrix will tend to be with spectral radius 1, distort the gradient with a direction that makes sure that the weights are chosen in such a way they converge towards unitary spectral radius matrix but you do not have any insurance that instantaneously, at a specific epoch, at a specific mini-batch that matrix is indeed with spectral radius equal to 1. Solution but we loose pieces of the gradient anyways.
Exploding gradient $\rightarrow$ gradient clipping, check the norm of your gradient, if it is smaller than the threshold ok, if it isn't you clip it, this does not alter the direction in which you move only the length of the step.

So for the gradient vanishing rescaling has no sense, we need a solution that gives us a spectral radius equal to 1, start at looking at those two term D and W and understand what we can do with that, this will require **choosing the activation function in an appropriate way and constraining the recurrent weight matrix**. Tanh not good → linear activation function has the property that we like, identity matrix when we compute the Jacobian, ideally choosing an activation function that is linear, the first term of my product is the identity matrix and then everything amounts to choosing the appropriately recurrent weights matrix → orthogonal. *Problems: it is an accumulator of input, takes an input, throw it inside of the state, takes another input throw it inside of the state, ... since we are in a feedback loop and you're only accumulating stuff that hidden state will tend to explode*, because will get the direction of steepest change, the Principal component of your space and will follow it, and we will grow until explode. Unbounded, uncontrolled, information just taken from the input and throw inside of the state, no control and in a feedback loop system this typically leads to explosion of the state.
Check that those weights respect the spectral property, the solution is to have for instance orthonormal matrices, unitary matrices, if I apply this idea combined with linear activation that in principle gives arise to our perfectly unchangeable gradient, the gradient always flows freely, if you think about it, what skip connections are doing, connections that have no weight, or better they have 1 as a weight (the identity matrix), the activation function is the identity, this is the perfect solution for the gradient perspective: caveat: unless you go for the id matrix *maintaining orthogonality or unitarity of the weight matrix will cost you a lot*, slightly better for unitary matrix because if you are smart enough you can use the complex domain to spend a little bit less in term of multiplication, smart factorization that helps you to reduce number of multiplication you have, papers that propose *unitary RNN*.
Using identity function and using identity recurrency will lead to this: $\mathbf{h}_t = \mathbf{h}_{t-1} + c(\mathbf{x}_t)$ my current state is my past state + the new contribution from the input, c(x) doing something to the input before feeding, maybe non-linearly transforming but doesn't really matter, because we know from the bound we derive, whatever choice we take w.r.t. to the input and its weights doesn't really matter w.r.t. the explosion/vanishing of the gradient, only the recurrent part matters. In principle what we need to use, but leads to saturation. Desired spectral properties but does not work in practice as it quickly saturates memory (e.g. with replicated/non-useful inputs and states). We want to be able to "control the forgetting".
Let's work around this one to make it feasible, let's build this linear recurrence but before we need an additional mechanism. *The problem with that formulation is the fact that we allowed uncontrolled writing on my memory that leads to explosion of the memory*, can I come up with a *mechanism that controls how much I write into a vector and this mechanism is a neural mechanism* (part of the NN). A mechanism that tells me how much of an input information possibly transformed gets into my state, that mechanism is called **gate or gating unit**. Used to implement mixture of experts, a modular neural network/model,

instead of being a monolitic NN which outputs a single prediction, is modularized in *k independent experts*, they all receive the same input, and each of them independently takes a decision and votes, classification or regression, *each of them is outputting a prediction*, how to generate the prediction of the all net? *Weighted voting*, rather than taking the majority voting, *we multiply each of the vote by a weight between 0 and 1 and make sure they sum to 1, exactly what gates do.* k local experts, NN of some sorts, MLP, all of them generate an output that it is aggregated, there is a *third party thing, called gating network*, in order to combine appropriately the votes we need weights between 0 and 1 that some to 1, do we know any kind of Neural Layer that outputs k alpha values between 0 and 1 that sum to 1 $\rightarrow$ *softmax layer* which is an additional module of your network that receives in input x, because you want to select the most relevant expert based on the specific input that you are receiving, imagine that you're an expert,you are the better one w.r.t. another depending the geography, a local expert of a certain region, another expert of another one, through this mechanism, this soft gating you'll learn by backprop that if the current input x belongs in a certain area you will have an higher alpha weight e.g. for the expert number 1 and a smaller for another expert, and the best expert of that region will have increasingly more weight depending on how much is closer to the decision boundary of the geographical region. So we have k alpha values, the local expert rather than a prediction might be outputting their hidden states, so the hidden state h1 from the expert 1 is multiplied for α_1, ... they are all sum up together and then you pass this info to an output prediction layer, you can combine directly the prediction or the hidden state, interesting thing: this is a different kind of connection, we are used to connection that goes from a neuron to another neuron weighted by a synaptic weight, we have just introduced a *different form of connection this is a connection that goes from one neuron to an edge*, it has a different nature, *it is multiplicative*, if you look this from the perspective of Statistical learning theory this has a larger VC-dim, it is a more expressive/complex model, because of the multiplicative nature of the interaction between the neuron.

Now, we can use this mechanism for our problem, we have a multiplicative neuron that we can use to dampen or take all of the info we would like to write in the hidden state, by making sure that the output of the gate modulates how much of the input gets into the hidden state. **Use this mechanism to solve the problem of uncontrolled writing in our linear memory**.

Our linear memory is also called **Constant Error Carousel (CEC)**, other way of saying if I have that configuration of the hidden state updates the gradient will flow forever, fine, we know because it has perfect spectral properties, flow forever is also a mess, so we start controlling it. Making sure that my hidden state doesn't explode, *not all of the hidden state at the previous time gets into the new hidden state*, that is the job of the **forget gate**, a gating mechanism that computes at time t one value (multiple values actually) and what it does it tells you I want to forget things so f_t will be near 0 or I want to retain a lot of what you know from the past $\rightarrow$ f_t will go to 1, and everything that is in between 0 and 1, sigmoid $\rightarrow$ very bad for backprop, for poor spectral

properties of the Jacobian, but handy to dampen stuff, so my *forget gate will be a sigmoidal neuron*, whose output is multiplied by h_{t-1}, no way h_t will explode, because if it keeps building up it will be shut down by a forget gate.
$f_t = \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f)$, $h_t = f_t \odot h_{t-1} + c(x_t)$
This thing *avoid the explosion*, am I retaining the nice properties in term of the gradient never dampening? Of course no, I'm multiplying it by something that we know having very bad spectral properties, the sigmoid. No free lunch.
LSTM - Long Short Term Memory: births from the ideas of the Constant Error Carousel and on the controlling of the writing, start from vanilla RNN and start plugging into it, first we plug the CEC, the linear recurrent state, it will have a different name than h_t, h_t will become the state that is available to everybody, whereas the linear state I'm gonna keeping for myself as a neuron, what will happen it's this *connection between c_t, the CEC at time t, and the c_{t-1}, this implements the linear recurrency*, becuase I'm reserving h_t for other stuff that will come. If it is implemented like that it arises issue $\rightarrow$ add a **forget gate**. The forget gate makes sure that the info from the past, that comes from the CEC, doesn't necessarily gets all copied into the current state, but there is another thing that gets written in the state, which is the input, I may want to control how much of the input is written in the current state, in the poor CEC, so I'll be creating the **input gating**, it is a sigmoid whose job is to dampen the input, my recurrent will receive my current input x_t, the past public hidden state h_{t-1}, they will be transformed through a non-linear parametrized transformation (tanh), this gets you a g_t, the result of this combination/transformation, that thing is then multiplied by the output of the input gate, not all g_t is written but only the part that is allowed by the input gate, which is still a sigmoid, the thing that survives to this $g_t \odot i_t$ multiplication is allowed to enter the CEC together with the CEC from the previous time step, now we add also the **forget gate**, a different gate, so the input gate has params, so does the forget gate, they are neurons anyways, they have their own separate set of params, the job of the forget gate is to dampen the information that comes from c_{t-1} and whatever survives gets into the CEC. So, the CEC gets updated with dampened version of cc_{t-1} and dampened version of the transformation of the input, dampened by the gating units. Now I have the new state c_t that in principle I can give to the next unit, before doing so I have a non-linear transformation. I'm not yet satisfied, because I don't want everybody to read my internal state, I don't want people outside of my neuron, to be able to read all info, I add another gate, output gate, whose job is to decide how much of the internal state, transformed through a non linearity, of my neuron is delivered in output and this can be reused in the next state, passed over to an output layer, whatever, the only public thing is h_t and you decide how much it to show. Note the standard weight sharing of the RNN applies to the input, output, forget gate, the params of the non linear transformation of the neuron itself are all copied multiple times, for all the time step, but they are different one another the input gate has different param w.r.t. to the forget gate etc....Imagine that my job is to capture if in a seq. of zero there are 2 ones that are far away. This task does really require that you remember all the zeros, just need to be able to

count two 1, your hidden state must be able to count up to two, I implement it with gates: whenever I get a 0, the network will learn to keep the input gates closed, cause I don't need to count the fact that there is a 0, never gets into the memory, as soon as I get a 1 I write it in the memory, do I output through my output gate the fact that I have a one in memory? no, it is not helpful, knowing there is 1 one, it's not helpful for the prediction layer, I'll keep it inside, the output gate is shut, if at some point another 1 arrive, it is allowed inside of the memory by the input gate, it is written into the memory, then the output gate will know there are two ones, ok, go on, fire and open the output gate and keep it open until the end of the seq. when you can make a prediction.
It's not really that you are using the power of propagation of the gradient through many hidden layers, it's a cheat, have a very long sequence but my RNN reduces this very long seq to a smaller seq, cause the input gets cancel out the useless, not needed to memorize parts. 000000100001000→11. Artificially cutting pieces of the input seq. and nearing far away dependencies, dependencies are no longer far away because you decide what to consider and what to ignore.
This is a cinematic exaggeration of the thing but it is the intuition behind why it works, because *this thing is far from the ideal numerical conditions, the fact that it works depends on the dynamics of the reading and writing, in fact when you train the LSTM or any other Gated Recurrent unit, you stop your gradient, you truncate your gradient, when you do backpropagation through time BPTT you don't backprop through all seq. you backpropgate for 30/40/50 instances, as much as you can get, then the gradient fades anyway, it's not that you can propagate forever.*
Element-wise multiplication (Hadamard product) because you want to zero not all the component of the vector but selectively the components that you don't want to write.
You package *multiple LSTM cells in a single a layer and then you have multiple layers of that*, what is the effect of that? **Layering is as usual a way to extract increasingly more abstract features from your sequence**, character → syllabus → word→sentence. Hierarchical hierarchy you have in Deep Learning. If you wanna do sentiment recognition doing it on the basis on the basis of how characters put together in a syllable is difficult, if you do it on the basis of how words put together in a sentence it's easier, more generalizable.
How do you train LSTM? Now with **BPTT**, in the past there was a weird mix between BPTT and Real Time Recurrent Learning and it wasn't working, between 2005 some researchers come up with a fully correct derivation, there wasn't any automatic differentiation libraries, actual learning equations had to be derived by hands.

1) Input and Forget Gates:

$$I_t = \sigma(W_{Ih}h_{t-1} + W_{Ix}x_t + b_I)$$
$$F_t = \sigma(W_{Fh}h_{t-1} + W_{Fx}x_t + b_F)$$

2) Input Potential and Cell State:

$$g_t = \tanh(W_{gh}h_{t-1} + W_{gx}x_t + b_g)$$
$$c_t = F_t \odot c_{t-1} + I_t \odot g_t$$

3) Output Gate and Hidden State:

$$O_t = \sigma(W_{Oh}h_{t-1} + W_{Ox}x_t + b_O)$$
$$h_t = O_t \odot \tanh(c_t)$$

LSTM are plenty of params, you want to regularize $\rightarrow$ norm 2, norm 1 regularization or dropout that applies only to NN, because it disconnects neurons, the intuition behind **dropout** is that in order to avoid that the NN uses too many of its param *you force the NN to use less of the neuron that it has.* If you want to use all of the neurons, but not at the same time, you *randomly disconnect some of them each time, you pull a random matrix (mask) of 0s and 1s and you apply as a mask to the neurons. Literally you are zeroing the output of some neurons*, always pulling up a different mask each of the time, when you do seq. classification you typically fix the set of neurons at the beginning of the seq. and you use the same mask until the end. **Probability of shutting down a neuron**, or the probability of retention of a neuron, $0.8 -- > 80\%$ of the neurons retained on average, 20% will be on average shut off, control the amount of regularization. p selected with model selection. It works because rather than training a single network you are training a potentially infinite number of network, with different topologies, aside from working as a regularizer, this actually creates a committee effect, committee machine of weak learner are more powerful/resilient/robust than a single weak learner. Also a trick to estimate confidence, dropout is typically applied at training time, *at test time you just multiply the activations by the params of retention*, you cannot use the full activation because during training the network was used to have the 80% of the activation alive so you need to scale your activation. *You can use the dropout at test time, you don't predict with the full net you apply dropout, you disconnect some neurons and predict, then take same input disconnect neurons predict and so on, a population of prediction, mean and average prediction, build a confidence interval for the prediction of the NN, very empirical give an estimate of how much the NN is convinced about the prediction, if the prediction is brittle disconnecting some neurons will easily flip it one way or another, otherwise a tighter variance.*
You can also apply dropout on the single connections, rather than disconnecting neurons you disconnect single connection (**dropConnect**), higher cost in term of the mask (for all the weights not all the neurons).

Activity regularization, a couple of activity regularizer: you can use your friend L2 activity regularization, seen for instance in autoencoder, where we was regularizing the loss, adding a penalty term based on the norm, one in that case, of the hidden state activation, we do something similar here, we add a term to the cost function which regularizes, using norm 2 this time, the hidden layer activation (recurrent), if applied together with dropout just make sure that this regularization it is only applying to stuff where you are not dropping out, and then there is a weight trading the cost of this regularization, *the effect of this regularization is to make sure that the activations of the recurrent neuron are kept close to 0, that don't grow excessively. Since you have tanh you keep the model working in the very dynamic area of the tanh.*
You can impose some form of activity regularization in time, between two consecutive time step, you don't want that states jump too much, some sort of smoothness constraint.
L2 activity regularization:

$$\alpha \left\| \mathbf{m} \odot h^t \right\|_2^2$$

Temporal activity regularization:

$$\beta \left\| h^t - h^{t+1} \right\|_2^2$$

Truncated gradient propagation: when you are propagating your grad from time to the beginning of time, the numerical properties of LSTM do not really allow for backprop through many time instances, not worthy to go through all the seq. if there are 100 backprop stages, you cut it, you stop after e.g. 50 propagation, stability and efficiency, you save a lot of multiplication. Truncation factor you can define it when you set up the optimizer.
GRU - Gated Recurrent Unit: rather than using 3 gates you use only 2. **Reset gate** and an **update gate**, the combined work of the two gave you a similar behavior that you have with 3 gates, not exactly the same, because the 2 gates need to be able to do the job of 3 at the same time, intuition: reset gate controls how much of the input gets into the hidden state h_t, no CEC here, I directly operate on h_t, I control how much of the past h_t and the current x_t gets into the new h_t proposal, how much of this proposal enters the actual h_t that is outputted after is controlled by z_t, z_t trades between the proposal and the existing information. The reset gate r is the one controlling how much you're feeding in input and the z, the update gate tells you how much you're integrating the proposal, before you put it out, you linearly combine it with the past state. So whatever you are getting as an actual state outputted to the external world, is a weighted combination of exactly the past state h_{t-1} and this proposal that has been computed internally by deciding how much of the current input and past h_t gets into the new proposal and there is a merging of the two.
z_t is a sigmoid, if z_t is 0.5 you take 50% of the old state and you mix with the 50% of the new proposal.

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

$$\tilde{h}_t = \tanh(W_{hh}(r_t \odot h_{t-1}) + W_{hx}x_t + b_h)$$
$$z_t = \sigma(W_{zh}h_{t-1} + W_{zx}x_t + b_z)$$

$$r_t = \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r)$$

These are neural models for sequential data, all we say for HMM applies also here, including the fact that I can have an HMM where one chain reads stuff left to right and another chain that reads stuff from right to left, then you get to a point in which you fuse the two information in order to predict something. You can do exactly the same here, you have a 2 layers/modules architectures, in which one LSTM layer sees time that flows from 1 to T and another LSTM module that see time that flows from T to 1, and you fuse the 2 hidden state in order to predict something. Application: character recognition task from handwritten digit, it works better if looked to the whole thing before generating the actual prediction, it is easier to understand if there is a p after an m, same thing if you are in genomic data, with bioinformatic data, you typically have a seq and you want a prediction on a portion of the seq to be influenced by what comes before and what comes next.
The first neural generative model of language wasn't a transformer, was a LSTM, comes natural, because it is a purely autoregressive model: a model in which given the input until now, predicts the next input, classical autoregression, takes the input that is outputted and feeds it back in input and predicts the next input token, this was applied to input that were characters, letters so you have like 3 different layers of LSTM, you were seeing in input letter h and you predict e, then you see e and you predict l, l and you predict the next l, .. you are constructing the word hello, training this thing is simple, fully self-supervised, cause you are predicting the next character when you are training, and when you are testing it works automatically: you just write the initial letter, then allow the model to predict the next character, feed it in input, predict the next character, feed it input, and you go on and you generate everything...The intermediate layer of this model are LSTM, the output layer are a softmax over the letters, so what you are doing is predicting the distribution of the most likely letter, you take the most likely letter or tossing a coin and take the letter, and fed in input. When you do training you do **teacher forcing**, when you train the model the next input to the NN is not the prediction, is the true following letter, at the beginning the output of the NN are random you can't use them. Trick: if you train the model only on correct stuff, the model when trained will produce hella instead of hello, but not a problem cause it will be corrected by teacher forcing, it keeps saying hello I'm a model, at test time if it produces Hella it doesn't produce I, because it never sees something wrong like hella, so the state is wrong, I would be highly likely that the model will mistake the next

letter, the workaround to this is allow the model to see mistake during training, put a *simulated annealing scheduling to the teacher forcing*, at the beginning your teacher forcing is fully active, so the model will train on perfect data, at some point you start reducing the probability of using teacher forcing, the model will start every now and then through its epochs seeing something wrong and weird and it'll learn that it has to correct, until you get to convergence time where the model will only predict based on a fully autoregressive mode with no teaching forcing.
Every time you train anything in an autoregressive way, robustly, you will have to use this kind of scheduling of the teacher forcing.
Neural memory: what happens to the representation of words in a NN if you train it on a certain type of language, you get all the best and all the worst of humans, including becoming extremely sexiest, the same for etnicity, there are words like blonde, brunette, goose that are projected in the same area (t-SNE projection of the embedding of the word), the NN has only the bias that gets from the reading, reading shapes your mind so read good stuff.
LSTM works with seq. but seq. are only the beginning of structured data, I can have trees, in a tree I wanna predict about a node taking into consideration the context of what is below, the two children up-trees, i can extend my LSTM to work on tree, **tree-based LSTM**: it finds a CEC, an hidden state for the current node, by integrating info from these two cl and cr which are the hidden states of the 2 nodes below, if I want to train a classifier that classifies trees is to train a LSTM of this form, that starts visiting the tree from the bottom, from the leaves, encodes first the leaves, basing on the fact they have null children, so basing only on the info attached to them, generate a c vector, an encoding for all of them, and when you have generated an encoding from these one, I combine using gates the knowledge from the two children, recursively, until to the top of the root when I get to a unic c, and use this one to predict on the full tree.
Sentiment analysis using parse tree instead of seq. of words: why they can be helpful, words highly correlated for predicting the sentiment may be very far away in the sequence, but when you integrate them in a tree they become very close, because a tree works on the logarithm of the n distance between the words, so relating 2 words and understanding that those 2 words are related in a specific way for the task of predicting the sentiment, it's much easier from the perspective of capturing that connection if you are using a tree rather than a seq. for the pure fact that the length of the jumps that the gradient needs to take is smaller is logarithmic rather than linear. Provide that you can generate reasonable structures replicable, and with parse tree is ok cause they follow the structure of the language that shortens the gradient flow.
Images: father of the Vision language model, can I *use this stuff into a system in which I have an image and that images gets automatically labeled with a caption: item-to-sequence task*, my image gets into my NN, I need to find the proper encoding of an Image: CNN can process my image, gets to the point in which it obtains the dense vectorial encoding, this become my x in input to my LSTM, the LSTM of previous that was trained character by character for

instance, or word by word, so my encoding gets in input to my NN, that is then generating in an autoregressive way the caption of the image, *I need to train just images and associated captions.*
Don't look at RNN as a model that can process sequential/structured data, *the key point of RNN is having NN with a state than can remember thing, that can compress info into an internal vector.* You can apply *RNN to generate images, pixel by pixel in a fully autoregressive way, the NN are generating the color of the first pixel on the top left, then moves on and generate the color of the second pixel, than the third, ...* How does it know to generate consistent pixels? Because it remembers what pixel has generated before in the hidden state, so the key thing is that they are a NN with a state, a memory.

18 Attention-based architectures

How **neural attention** is implemented and why it is needed? *seq-to-seq paradigm*, we will see different forms of attention, **cross-attention**, **self-attention**, leading to transformers and the use of attention in vision tasks. Consider LSTM or GRU as black-box. Consider layers of these cells. A RNN made of multiple cells, multiple layers. GRU, and in general RNN, are naturally designed to handle time varying, length/size varying input seq. $\rightarrow$ weight sharing mechanism, we copy the weights as needed for the length of the seq. The problem arises when is varying the seq. in output, paradigm in which there is a seq. in input of length n, and output seq. length m, asynchronous w.r.t. to the input seq. seq-to-seq setting: carefully rethink the way in which we process a seq. **Setting in which the input seq. and the output seq. they are desynchronized, possibly of different length. Using RNN in this setting is not a good idea.** Sequential data is compound data, is made of piece of info linked by relationships, always a good idea, to interpret a piece of info in the right context, predefine a context of interest, everything that is interesting to understand what's going on in a particular time of the sequence, whatever was before that time, and that is undesigned context or an **adaptive mechanism that tells you which part of the rest of information is relevant to interpret a piece of information**. Fixed or adaptive context $\rightarrow$ **attention** *gives adaptive focus on different pieces of information when interpreting other pieces*, among all the predecessor of the current observation should I be considering all of them, or select only the relevant one, what attention is doing is selecting only the relevant pieces.
Sequence to sequence: sequence transduction, in formal language: automata recognize input sequences, transducer are state machine that transforms an input in an output seq. the same concept applies to ML, task receive in input a seq and return another one, no synchronization between input and output seq and seq. of different length, machine translation is the main example. *I first need to encode all the input and then after that start outputting the output.* All a matter of how we encode the context. Naive approach: a set of Recurrent layer, the RNN in the first part receive in input the input seq. x1 the first word, x2 the second word, in english language, then I output nothing while I have

an input token, or else said I output a specific output which is "no output", at some point I will receive in input a specific token that signals that the input seq is finished. I will train the model whenever you see the seq. finish token start returning/producing the output, the words of the output sequence/the translation. Problem: first of all I have only a set of param for 2 languages, a single RNN which is handling both the english and italian version in the same set of params, mixed 2 languages with the same set of param, it is the same RNN that first processes the input and then the output, also if I have to relate "the" to "il" that is a very long way for the gradient to work, for the information about "the" to propagate in the forward phase, a trace of the word "the" needs to stay in the hidden state of the RNN for enough time until it reaches the point in time in which it will have to be translated into the word "il", both a problem in gradient backpropagation, both in the forward path, in keeping actually trace of the word which is semantically in the hidden state of the RNN for a long time, for all the all input seq. and part of the output seq., while keep adding info to that because you keep receiving other words in input and they all will be squeezed into the only context vector which is the hidden state of your RNN. This idea of leveraging the unfolding of a single RNN even if it has multiple layer does not really work.
Something different that uses a modularization of the network: **two phases**: **encoder** (when I receive in input the info) and **decoder** (in which I decode in output info) phase, that is not made explicit in the structure of the previous NN, because there was actually a single NN, instead of having an atomic thing use *two modules to isolate the different language in different module each with the corresponding params.* The job of the encoder is to take in input a seq. and compress into a fixed size repre., indip of the length of the input sentences, seq. of different length to be projected and encoded into the same dim embedding, a vector with an equal dim. The encoder will read x1, x2, ...xn, e.g. with a RNN, h1 will be the embedding/activation of the recurrent layer corresponding to (encoding) x1, h2 embedding corresponding to having seen the input x2 after x1, so conditioned also on the info in h1, a classical RNN. *At the end of the input seq. i will have n hidden vector, encoding for each elem of the input seq.*, I need a single vector that compress all the info of the input seq: good idea take h_n: the bias of the RNN tells me that h_n in principle has info about the all seq. or I can sum all of them but in the end, I need one single vector, a **context vector**, summarizing info about the full input seq in a single vector. In the original the choice was to choose the context vector to be the embedding of the last token of my sequence
The context vector is used to inform the decoder, responsible to output the output sequence, *the first module was a RNN, also the second module is a RNN, because it has to deal with outputting the seq.,* how can I decode in output a seq, first of all I need to inform the decoder about the content of the input seq. by passing the context vector to the decoder, multiple way to do that: dumbest way set the initial state of the RNN, it assumes that the hidden size of the encoder is identical to the hidden size of the decoder, not super general, also when you are decoding s_n the memory of the initial content from the input

into s_n is lost or is gonna be very little, because you have inputted the context only at time 1, and then through time you have added info and you might have lost the original meaning of the sentence that you want to translate, must *better approach c gets into all the decoding stages, all the steps of decoding.*
Fully autoregressive way: RNN at time 1, will receive input from the context vector and it's own personal input at time 1, x'_1 is the token that says start translating, ¡start¿ and encode the info that the net should start output, I will output the first word W1 by generating through a softmax defined over the vocabulary, over the token whatever, it will be the first word of the translated seq.: then autoregressive modeling: w1 becomes the input at time 2, together with the context and so on, the model knows that now it has to output a second word which is then fed in input and so on, use teaching forcing to train, interleaving with some error to teach to the net how to correct mistakes.
First architecture: modularized: encoder separated from the decoder (resembles the encoder/decoder transformer architecture, the design comes from this): separate parametrization, autoregressive modeling of the language or more in general output generation, info about the input seq. compressed in a vectorial representation that should carry all the semantic of the input info
seq-to-seq architecture: RNN both as the encoder and the decoder, a context vector that compress, acts as info bottleneck between the encoder and the decoder that is the last hidden state corresponding to the final elem of the input seq, first neural machine translation NN that achieves decent translation accuracy in NLP. They can share params but typically uncommon. *They can also train them independently, you can train the first model to become a good embedder of english language and the second a decoder of the italian language: you do masked language modeling the first and autoregressive language modeling in the second case and then put them all together and refine them.* This model works well if I revers the order of generation of the output sequence. **Reversing the input seq. in encoding typically resulted in increased performance**, doesn't apply to all the languages but typically in western languages, the last word in the input language tends to occur near the end of the output language, by reversing you make more easier to get that dependencies, because **there is less length of backprop, less length of forward propagation, you have a lot of info in the hidden state about the last word of the input seq.**
You need to reason on the fact that *the way you are composing your context info is not adequate*: if you keeps thing ordered in the language way when you train to decode tavolo, in your context you have a lot of stuff/info and a compressed version of table, there is going to be that you are trying to decode 'tavolo' having in your context info a lot of stuff that comes from the things that I have already decoded and a vague history of what was the real semantic of all the words in the original input seq. In reality to decode 'tavolo' the only thing you need to know is the word 'table', possibly the article, the encoding these information to support in predicting table, the context that I need for "tavolo", are those 2 words of the input sentence plus the predecessor in the output sentence, to predict "il" in output I don't really care in knowing the encoding of table, I need only the encoding from "The" and "cat" as context, *different words will need*

different context, **my context aggregation strategy needs to be selective**, need to be able to pick up the pieces of info that I need to put together in a specialized context for each of the element of the output seq that I need to generate, that is way **we need attention that gives selective, personalized context**. Getting rid of the mechanism that creates a single context c from the full input seq., **use an attention module**: *the attention module needs to receive all info from all the embedding encoding of the input sequence/tokens* and need to generate a context c, cannot be just the selection of the last elem or the summation of all the input elems, but *needs to be a function of the word that I'm trying to decode now*, at some point there will be another input that comes from the decoder that tells which word I'm trying to decode, which is the word I need to translate: *the attention mechanism will have the job of learning to understand out of all the embedding that I receives which are the ones to keep*
Attention module: black-box view, a module that receives in input the set of encodings, h_1, ...,h_n of the same size, needs to be comparable, vectors of the same size, at some point you will aggregate them, then you need the additional information, the context that tells you what are the things are relevant for my context s, a sort of switcher, in the seq-to-seq architecture of before, s is going to be some sort of info about the word I'm actually decoding, in output I get an aggregated c context vector, typically same size input h_i, tailored to s, function of s.
Inside: gates, we need a mechanism to select what pass and what don't pass, we need gates multiplicative connections that decides how much of the info flows over a synaptic connections gets to the target neuron .
At the beginning: the first part of the attentional mechanism (cross-attention) **computation of relevance**: it means a match between the context information and each of the input, a number that expresses how much s and h_i likes each other, e_1, e_2,... is relevance measure, how much input h_i is relevant for context s, one values for each of the input vector: s and h_1 are the same size: vector operator similarity measures: **dot product**, in the general case s and h_1 come from different spaces, so the operation that returns a coefficient of similarity is a Neural layer whatever you like, Neural layer that does an embedding of h_i into the space of S through a linear projection (S in $R^d \leftarrow W_1 h_1$) that embeds h_1 in the space of S than you do a dot product or you can have a MLP that receives an input s and h_1 and return in output e_1, the constant is that you have parameter, you'll learn how to assign the right relevance to the input w.r.t. to the context with backpropagation through all the modules. Advisable Properties of this thing: let's call the relevance modules MLP1, MLP2,... *if I use different MLPs will have different paramaters, relevance checking that is position dependent*, consider h_i binary, 1 or 0, it is positional, the relevance of s with 1 will depends where the 1 occurs, *the parametrization will be position dependent*, input 1000 will be different than having 0100, same number of 1 (a single 1) and all the ones set to zero in input to my attention layer, in the first input pattern the first elem is set to 1, in the second case is the second elem set to 1 and the other rest 0, but under this setting e_1 on the first set of input will be different from e_2 for the second set of input, cause the matching/the rele-

vance is computed using a different set of param, the MLP that computes them are different, positional dependent attention, **if I want to obtain something that is positionally independent, weight sharing, all the MLP shares the same number of params**, attention module is independent of length, **weight sharing gives you independence w.r.t. to the positioning and independent of length**, so gives the ability to have an attention module that accepts a variable number of inputs, if I have 5 input, I copy 5 times my relevance computation network, if 6 inputs I copy 6 time and so on. You cannot have both, position dependence and variable number of input
The first layer computes the relevance, you get a certain number of scalars, generic numbers, I don't like these numbers unconstrained, I like them between 0 and 1 that sums to 1, **gets all the e_i through a softmax**, so that after this normalization they can be considered params of a convex combination. Take my α params, for each of the input, multiplies by the corresponding input and gets summed together, α_1 x h_1 + α_2 x h_2 x...α_nxh_n, weighted sum, the output of my attention module is the weighted summation of the input of my attention module, where the weights α_i will give much weight or less weight to the input that are consistent or inconsistent with the context. Just a way to smartly sum the input. Relevance contribution mechanism that tells me how much take from each of the input.
The attention module receives input from all the encodings of my input seq, and if I want to be able to encode the input seq. without being position dependent I know that I need to make a weight sharing choice inside the attention module, this will also allow me to take into consideration a number of input to the attention modules that varies and that varies with the length of the input sentence, then my attention module will get these contextual information, additional input, and will output the compressed seed or context for the decoding.
How the attention module plugs into the seq-to-seq architecture, h_i become the input to my attention module, how I get the context? s2, what I'm trying to do now is generate through the attention module the vector c that embeds all the contextual information that I need to have in order to decode in output y3, the previous word decoded, plus the context that tells me the right information from the input seq. How can I obtain the right information from the input seq: through an attention mechanism contextualized to what? s2, because I'm trying to compute s3 here, so the last available context computed is s2, s2 summarizes all the info of the words generated so far, s2 becomes the s-th input to the attention module, **all the hidden state of the input sequence are contextualized in the context of s2 and the final c that I obtain is the c put into the context of s2**, that is used to generate s3 which is used to output the word y3. This the final seq. to seq. model, with the attention module, that was for some time the state-of-the-art in Machine translation. Important because taught to the community two things: if you work with a certain type of data that can be decomposed in pieces you need an encoder-decoder architecture, and an attention mechanism to be very efficient in select which part of the data to attend when you are processing data, which is what influenced the design of Transformer. **Transformer** has an **encoder-decoder** architecture with *heavy*

use of attention, a different type of attention.
When you are translating with attention, you get a matrix that tells you how to put in relation single words in the input seq. with single words in the output seq. When you are translating "signed" the network is focusing/making a lot of attention to 'a été signè', attention is essentially a multinomial vector, a column vector that sums to 1, αs. Sort of an interpretability gate to the network.
All of these modules are plugged into a neural architecture, in the forward phase you'll produce the output, in the backward phase the gradient should be propagated into each module, to adjust the param, if the network share the weights you still have to backpropagate through each of them and then sum the gradient in order to update the single parameter, if you have different set of params you adjust each different set of param indip. But you need to be able to backpropagate to it, you can because there is a softmax and MLP, instead of using sum aggregation, you can use some other creative forms of aggregation, including random sampling, rather than outputting a single context that is the sum of the input, I can sample the most probable input, most probable according to α, the real problem comes when you backprop., cause there is a **random sampling operation that is not differentiable**, but that is only an option (soft attention $\rightarrow$ you aggregate all of the input and you can back-propagate while hard attention rather than aggregation, you are selecting, but when you sample you lose differentiability). **Transformers** took inspiration from the encoder and decoder, are not only RNN, but transformer like layer, so **feedforward NN**, get rid of recurrence, **a lot of attention, MLP transformation, batch normalization, residual connectivity**.
The key thing of transformer is the use of self-attention **a specific form of attention**, *different from the additive cross-attention*, **additive or multiplicative attention**: a *MLP computes an additive attention*, the s seed and the input h, they both get in input to the MLP, so they are combined by summation, **with self-attention things are combined with dot product so is multiplicative**, you can also have multiplicative cross attention. **Self-attention:** instead of having my s info to be something different, that comes from a different module, s comes from the input h, you get in input an h_1,...,h_n, **s does not come from the external world but it is one of the input h**, you compute the attention of the inputs with themselves, in round, first time all the input against h_1, then all input against h_2, and so on, everything against everything, you compute the attention of the input against each other, in round, I need a parametrized transformation in order to compute the relevance, I need to transform the inputs through some parametric transformation, a linear transformation simplest form, W x h_1, W weight matrix, instead of projecting the input only on a single vector through a linear embedding, project x_1 into 3 different vectors: **query, key and value**, *each of them generated by a different set of params, matrices*, the first step, starting from each of the input, some magic going on that transform the input info into 3 vectors, key value and query, obtained with 3 different matrices.
After having generated my 3 interpretation of the input, I need to compute the relevance, my equivalent of my e_i value in the cross attention, let's assume that

now I use h_1 as context, so the first elem is the one against which I'm comparing all the other elems of the sequence including myself, I will take input 1, I will take one of its views, which is the query vector generated by one of the transformation, and compute the **matching between the query vector and all the keys of all the other input**, this will give me my relevance scores/e_i values. Computed by matching the query with each key. One value that is equivalent to e_1, e_2, e_3, is a score still un-normalized, pass everything through a softmax, to become αs params of my convex approximation, number between 0 and 1, the sum of all these α will be 1, you use the αs to generate the aggregated values by multiplying each α for the input, not the original input but the third transformation the **value**, and I sum them, the final output is the summation of the three generated vector, the effect of the score α_1 that is zero is zeroing the contribution of input vector number 1, $\alpha_2 = 0.5$ score and $\alpha_3 = 0.5$.
Same step as attention: computation of the unnormalized relevance, computation of the softmax values, use of those softmax values to obtain by a weighted summation a single output for the whole attention module: the only thing changing rather than using directly the inputs I use 3 different transformation of the input, to compute all of the 3 different aspects: **relevance computed as a query-key matching mechanism**, dot product between the query and the key, the score is then normalized and **the output is then computed by multiplying the normalized score with the value, not the input** this gives you more representational power to the attention mechanism. You can use the different set of params, one to learn how to represent stuff in an adequate way in order to be a query for the other points, one transformation allows to obtain proper key for each of the input and one that manipulates input in order to have properly shaped values for the output prediction. **3 set of weights for this neural model, a set of weights for query, one for keys and one for values**. This module is non-linear, if I only use linear projecting, the output is a linear transformation of the input, the slight non-linearity is in how you compute α, because there is a softmax in the middle, but the actual transformation from input and output is a linear transformation, through α you can compose non linearly a linear transformation, this is a way you put MLP between attention modules, to push non linearity then up, otherwise if you combine only attention mechanism you get bunch of almost linear transformation, one after the other and does not get very representative . So to recap: transformation of an input into 3 different things: the query, the key and the value, obtained by a linear transformation, multiplication of the input by a matrix sigma q of parameters plus a bias parameter. I have transformation of an input into 3 different things obtained by a linear transformation, multiply the input by a weight matrix plus a bias. Same for the keys, same for the values, different set of weights for each of them. My input is a sequence of length n, where each elem is a d dimensional vector, I compute the query, the key and values as matrices, it means I can compute each of the elem for each of the input in **parallel**, can't do the same with RNN, in order to compute the embedding for an elem I need to have computed the embedding for all the preceding elems in the sequence, in the attention mechanism you don't have to do that, you can compute in

one-shot all together in parallel, all the cross attention between all your elem of your input, in parallel if you have a GPU. There is a lot of desynchronized computation useful to be parallelized.
$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
Scaled multiplicative self attention: multiplicative cause I'm multiplying the key to the query, this is how I compute the relevance matching, scaled because i have a sqrt of the dimension, the scaling of the self attention is to make the computation through a softmax smoother, if you don't scale the softmax by the size the largest of the relevance vector e_i will get of the alpha and the rest of the alpha will be set to 0, you'll tend to have very peak multinomial distribution, by rescale you redistribute a bit of the probability mass to the other alphas, otherwise winner take all behavior in the softmax, the result of softmax is a vector of alpha multiplied elem by elem and summed by the value vector, all this stuff can be run in parallel. Attention in practice is seldom used with a single set of weights rather I compute multiple attention for each layer, **multihead attention**: instead of having a single set of query, key and value I will have a query, key and values set of weight for each different channel, capture the fact that words can have multiple meaning, capture this multiple meaning in a single embedding vector is difficult, multiple embedding vector one for each different channel and the different channel can capture different meaning of your words, replicating the concept of attention. If you apply this multi channel attention, each input elem is embedded by the attention vector into a different set of output embedding that are then concatenated or aggregated by summation e.g. arrive to be a single input that gets to a mlp (in general a non linear transformation), you get the final non linear embedding of the whole self-attention transformer-like layer and then it gets into another layer. This become an atomic module that you can plug in different architectures. **Encoder-Decoder architecture of a simple transformer**: encoder, the thing that in the previous architecture was encoding with a RNN the input seq., the decoder what in the previous schema was the RNN for the output later, between them the context information that flows from one to the other, at the top where do you predict the next word, when you are doing machine translation or generative language model, multi head attention is used in many places, **you use a lot of layer-wise normalization not batch normalization cause it assume independence of elems in the minibatch but you are working with sentences NLP so layer normalization**
Is self attention a good mechanism to model temporal dependencies?? People use transformer with natural language, seq. data, time series, but are transformer a natural model for that? **If I permute the order of words in a sentence, does the embedding of self-attention changes? If I use the same set of weight/weight sharing for all the input there is no positional embedding**, "I love cats" equals to "Cats love I", same embedding, **transformer is not a model for sequential data is a model for multi set**, the inductive bias of transformer has anything to do with modeling sequential dependencies, only modeling how things co-occur in an unordered way in an input bunch, an

input bunch that is unordered and can vary in size is a set not a seq. the fact that an item of a set can occur multiple time, make it a multi set not a sequence If you don't to anything seriously to handle this thing this is a model that has not the proper inductive bias to deal with sequence, sequentially order info like language. You need **positional encoding**, not positional weights, those are a nightmare, you still want weight sharing between position, but you need to tell the transformer look there is an order between the items you are seeing, How do you do that? **Taking the embedding generated by the self attention (which are embedding that does not take the order into consideration) and summing together those embedding and embedding that takes into consideration the position of a word/observation in a sentence or in a seq.**, you need an additional component (positional/encoding) absolute or relative, you take the embedding generated from the self attention I need an additional component position encoding : e.g fixed absolute positional encoding you want a vector, to sum to embedding generated by the self attention module, same size as the embedding generated by the attention module, let's focus on y axis, I want to embed the position over a seq. that has a maximum length of 50, each row of that matrix is the vector encoding the position of the i-th elem of the seq. will have the same size as my embedding from the self attention, I'm gonna be summing them. How are them computed? How do I compute/represent a location in memory: bits 000000, 100000 0100000 110000... embedding of location using bits, can I use these vector and sum to the input embedding? NO, you obtain nicely refined representation of the semantic info of words represented by this vector in R^d were the fifty digit after the 0 has semantic relevance and the you sum one thing that has a one in a specific position of the vector, completely destroy the semantic info, in general metric model such NN **you don't never wanna sum a discrete information to a very dense piece of info, the discrete kills the continuous**. In general you don't want to sum a discrete info to a dense vector, need a way to embed the same info but in a **smoother real based embedding, use sin and cos (sinusoidal)**, those embedding are generated by creating an embedding of a position, then apply sin and cos transformation, my sin and cos will have a frequency, the deeper you go into the depth of your representation the less thing change there is only very low frequency change in stuff, at the beginning you are encoding near zero, very high freq. representation, what you are representing through sin and cos is the bit flipping mechanism, when you move between 0 and 1 you are flipping 1 bit, when moving from 1 to 2 you flip 2 bit the less signicant bit is flipped back to 0 and the second less significant bit to 1, when you are moving to the next representation you are flipping 1 bit, when moving to the next one you are flipping 3 bits, this matrix represent the bit flipping mechanism, the amounts of bits that you flip depending on the number you are trying to represent using couple of sin and cos, theorem that tells you if you wanna learn the dependencies between time t and time t+delta t, that there is a sort of temporal dependencies if you use sin and cos encoding of the delta of the time than this things is learnable by a linear transformation. It is a way to embed position into a dense space in a way that easily learnable by a linear

model, anyway there are other encoding. And gives the model the ability to know if a specific word is in a specific position, as you are hammering inside of the transformer the position, it by itself has no inductive bias of the ordering of stuff.
To provide the Transformer with information about the position of tokens in a sequence, sinusoidal positional encodings are added to the input embeddings. These encodings are defined as $\mathrm{PE}_{(p,2i)} = \sin\left(\frac{p}{10000^{2i/d}}\right)$ and $\mathrm{PE}_{(p,2i+1)} = \cos\left(\frac{p}{10000^{2i/d}}\right)$, where p is the token position, i is the dimension index, and d is the total model dimension. This approach allows each position to be uniquely represented across dimensions using sine and cosine functions of varying frequencies. Lower dimensions capture short-range dependencies with high-frequency oscillations, while higher dimensions encode longer-range dependencies with slower oscillations. These functions are fixed and not learned, enabling the model to generalize to sequences longer than those seen during training and to infer relative positions from absolute encodings.
This attention based mechanism can be ported to vision: I can **compute attention over pixels**, *automated captioning model that receives in input a figure and then decodes in output a sentence that describes the figure*, this wan is using attention, when it is decoding birds is looking in the central part of the image, where are the pixels that actually contains the bird, *attention costs to compute it, rather than computing attention at the level of single pixel, you divides into patches, and compute the attention on the patches not on the pixel.* My image caption mechanism receive in input an image and outputs the caption, we already know that in order to output a sentence in an output, I need a RNN or a transformer, while in order to encode an image in input a need a CNN this time I use a CNN together with attentional modules, *the CNN will encode rather than the full image will be encoding the single patches of the image, independently by the CNN, then single sub-images are put together by an attention mechanism into a single embedding for the full image, then then this is fed in input to the LSTM that learn how to relate each of the patches of the input image to a specif word that it needs to generate*, like in a classical translation schema, before the input was a sequence of words, now a seq. of image patches each of them encoded via a CNN and then transformer for images.
Vision Transformer: straightforward application of the transformer to vision task, nothing really different from the standard transformer in the learning machinery. *Direct application of the transformer layer.* With the transformer we expects an input that is multi set of things, or a seq. if you use positional embedding, with images we have a seq. of pixels but that would be too much: think about it, a key thing of the transformer is that you compute self-attention, it means attention between an element and all other elem of your piece of info, in image will mean attention between each pixel vs every other pixel, also if you do not add positional embedding it means that an image with the pixels in a certain ordering or another image withe same pixels but a scrambled ordering have the same semantic, that is not true, so the model by itself does not have a real inductive bias for images, it's order independent, *you totally lose the structure*

of the pixels which instead you have in convolutions, you impose a structuring of the pixels cause you consider pixels organized in sub-windows, **how do you recover this inductive bias with transformers?, rather than working at a pixel level you work at image patch level**, *you take your image and you split into sub patches*, an image get split for instance typically in 16by16 sub-patches, which are just tiles(piastrelle) extracted from the original image by cutting into pieces, **then you give an ordering over that**, from the top left corner to the bottom right corner, the original order of the convolution, your image becomes a seq. of patches and as that is given in input to your transformer, and then **the computed self attention is between these 16by16 elems and the rest of the machinery remains identical, and since you don't want to lose the global ordering of these patches, what you add to them is positional embedding exactly the same concept applied to seq. data**, the observation at time t becomes the image patch at that specific position, the time information t, embedded through a positional embedding become a positional embedding of my patch in my 16 by 16 puzzels, **encoder only approach no decoder you train only the encoder**, you encode the input info and with the embedding obtain over the whole image you for instance classify the presence of a visual task: like in a CNN where you set up an image recognition/classification task, **what you gain w.r.t to a CNN? you lose inductive bias, you lose in computational complexity because of the attention but we gain in masked language modeling applied to images**, I can train my model giving in input a certain number of patches, all of them but one, and the one being masked with grey pixel and train the model to reconstruct the missing pixels, like the masked language modeling when you mask one word and you use the info of the surrounding word to reconstruct it, **self-supervised learning** no need of labeled data in order to do that, you pretrain your model, classical pre-training - fine-tuning schema that you have in NLP, ported to images: in which you pretrain your model using a language modeling task, in this case visual language modeling task, that do not require, no supervised info, so in a fully self-supervised way, once you have completed this thing, you take the few thousands labeled images that you have and you fine-tune the encoder part of the model to solve your supervised task with much fewer labeled elems. When to choose one (CNN) over the other (Vision transformer)? It depends on the nature of your task, the latter has less inductive bias in order to recover the loss of inductive bias you need a lot of data, and this is typical for transformer, low inductive bias model so lot of data in order to learn it from the data, **you gain being able to recover that inductive bias from unlabeled data**, you lose in computational performance/cost but you may gain in term of accuracy, depending on how much data you have available.

19 Neural Memories

How to deal with very long term dependencies in RNN, RNN suffers from this problem of not being able to fairly connect the dots between things that are far

away into the sequences, how to attack this? And why attention-based networks work, despite not having an adequate inductive bias for seq. data.?
Two perspective: one is understanding **how we can make the memory update itself more effective**, the other one is about **how we can represent richer form of memory**. Second topic: memory, but not as we have seen so far, not store just the state of the past input, but **modular NN** in which **part of the NN has one job that is storing information**, *long term storage information rather than dynamic memories like usual state of a RNN* → **differentiable neural memories**. Differentiable memory read, write, indexing. **Hierarchical and multi-scale RNN**: gated RNNs do not solve the problem of long term dep., *only in principle we are able to completely cancel the gradient vanish*, but in practice in order to do that we obtain a memory that can be saturated by all the input info that flows into the network, so we reintroduce gates and gates reintroduce gradient vanish, we *solve the problem of medium-term dep*, gates allow to shorten the distance between elem in the seq. by deciding not to update the memory/what to store in the memory, by closing the input gate and making sure that info does not get into the memory or by opening or closing the forget gate, sure that the memory is not tampered.
In making the memory more effective there is something to do with not updating the state, this is a good idea if you want to preserve your memory. With LSTM or GRU they can tackle medium term memory task, but *if you go truly on long term memory tasks, which you require a NN to store long range data they don't work as effective*. Consider MNIST digit as seq. of pixels, (linearize/vectorize images), train a RNN to classify the long seq. of pixels into numbers, need to put in relationship pixels very far away in the seq. representation → LSTM and GRU fails. We'll see architectures to address this problem: **key thing optimize how you use the recurrent state**. **Skipping update it's a good idea if you want to preserve your memory → vanishing effect proportional to the number of layers/unfolding you have in your network**, in a RNN you have a num. of layers equal to the number of unfolding that you do in the net. 100 elems seq. → 100 layers in the unfolded, from the gradient perspective you have to backpropagate through 100 layers. Make some of those layer disappear, you'll reduce the effect of your gradient vanish, the intuition is that avoiding to update the memory, making disappear some layers have a positive effect. *Mechanism to introduce this skipping/no update?* One we already know → CNN → shorten the path of the gradient, adequate to treat time series, or seq. info, **use monodim filter** e.g. [w1, w2, w3], weight sharing assumption, in a CNN the same set of param. will be reused for each elem of the seq., this will allow me to deal with length varying seq. Advantage in terms of gradient vanishing: consider your time series x1, x2, ...,xn and a 1 x 3 filters, in time series causal convolution, center the filter on the last elem, at the beginning centered in x3 and apply consistently, do the convolution w3*x3 + w2*x2+ w1*x1, only a single filter and a single layer, you generate your output and you generate your error/loss, make a prediction at x3 and generate an error for that prediction, need to back-propagate that error to understand the dependencies between the elem in my seq, what could have influenced the error that I make at

x3? x3, x2 and x1, the sub seq. seen so far. For how long do I have to backpropagate the gradient to create the link between x3 and x1? How many layers to backprogate through ? Only this one, so zero essentially, else said the distance between x3 and x1 from the gradient perspective is 0, direct feed from where the gradient is generated and x1, they are treated all together in the same layers, within the size of the convolution window, **all the elem that fall within the size of the convolutional filter will be in direct connection from the gradient perspective**. No gradient vanish. Gonna be having multiple layers, stacking convolutional layers of 1 x 3, two layers of convolution: x1,x2,x3...,x6 and so on processed at the first level by a first convolutional layer, then another convolutional layer that operates on 3 elem, e.g. h3,h4,h5 the results of the convolution run before, focus on h5, which means x5, it is the response of the convolutional filter centered in x5, h5 has access to what kind of info? h5 has access to all info about h5,h4,h3 directly, cause they fall into the same window, but through h3 will also know things about x1 and x2. *Increase of the receptive field.* From the gradient perspective the distance between x5 to x1, is 1, the only single layer between them that is 1, only a single layer between them, but they are distant 5 in the seq, if a treat a seq with a RNN the distance in terms of layer unfolding that will have from x1 to x5 is 4, 4 unfolding of the NN to get info from x1 to x5. With a convolutional approach I'm gonna be having a distance between them of 1, only a layer between them, the convolution layer, if seq is length N, filters size k, you need $log_k(N)$ layers. With $log_k(N)$ layers you'll span the whole seq with k-dim filters, with a RNN you need n-1 unfolding to go from x1 to xN, the gradient propagation in CNN is O($log_k(N)$), in RNN scales as O(N), and here *we speak in terms of multiplication, and multiplications matters here cause you multiple small quantities, the CNN from the perspective of gradient propagation is less affected from gradient issues*, no free lunch we lose unbounded memories, RNN does not define the size of the memory, here you are implicitly defining, convolutional filters of size 3 and a specific numbers of conv. layers fix the size of your memories, with RNN does not happen, the layers of RNN can be copied as much as you like, unfolding as much as you like $\rightarrow$ infinite memories at the cost of gradient vanishing problem.

There are models that does seq-to-seq learning using CNN instead of RNN with, in the middle, the usual attention mechanism. You can interchange CNN with RNN knowing you are gaining something and at the meanwhile you're losing something.

Why attention helps when you are dealing with long range dep, if you are recurrent, your gradient need to go through to n layers, so it dampen to connect elem 1 and elem number n in you seq., if you are convolutional is $log_k(N)$ better, **if you are attention based length is 1, always, you are independent from the size of your seq**, why? What does it do the self attention mechanism? *It takes every elem of the seq. and cross-ref with every other elem in the sequence in the same layers, so the first elem of the seq. is running self attention with the last elem of the seq, with the second to the last, with the third to the last and so on, in the same layer you are putting in direct correlation all the elems of the seq, there is no layer distance between them*, then we use more layer in

a transformer because of extracting more abstract features, but not to put into connection far away things like in a RNN. *That is what you actually gave, you are cheating on the structure of the seq. because you are really putting in direct connection elems that are far away, you don't consider that elem could be far away, at least in the inductive bias, then you reintroduce that with the positional encoding but you don't have to backpropagate through many layers just to have the connection between far away items in the seq.*
No-free lunch even **in self attention you pay in complexity per layer: given a sequence of length n, you are comparing every elem of the sequence with every elem of the seq.** $O(n^2 * d)$ **where d is the dimension of the embedding**, huge lot of operations but all parallelizable, if you don't mind about the planet and have enough GPU no problem.
RNN has a nice linear computational complexity per layer, you just integrate the past observation with the current one for length n, but it is enterely seq. completely synchronized operation in order to process now I have to already process yesterday otherwise I cannot do it, so the complexity there is different and of course *you pay path length for the gradient.*
Convolutional NN seats in the middle, quadratic complexity in the embedding size but sort of linear complexity in n, depending on how big you choose your convolutional filter. And like self attention you can run things in parallel, you are not expecting to have completed previous convolution, on the previous element, in order to compute the convolution on the current elem.
The complexity per layer you pay also at prediction time, not only at training time, depending on your application you need to be careful on the model you choose.
The self-attention restricted I'm not computing the self attention **all elems vs all elems** but **all vs a subset of all elems**. Complexity r*n, where r depends on how much you depart from n.
In RNN we have these issues to have N layers, we **need to skip some of these layers to reduce the number of multiplication and so the gradient vanish**, 2 families of approaches: one is **static, predetermining when I'm designing the network when to skip the updates of the state, adaptive, I'll learn when to skip the updates**. When implementing you can decide to *skip the update at the unit level: some units skip some not or at the level of block of unit, your net is modularized some units all together skip the update other are not skipping and so on or you skip the updates for entire levels.*
Unilevel static skipping: Zoneout, sort of regularization mechanism, works like dropout in a sense, Zoneout is implemented positively rather than negatively at the forward level, for every unit *you draw a mask to decide if the state of that unit is updated or not*, if you pull up a specific unit and you decide to zone it out, rather than set it to 0, you *keeping the previous activation.* What I obtain? There is the usual unfolding in time of your network, based on the input you are receiving x1,x2,..,xt and in the recurrent layer there is gonna be multiple neurons that has been unfolded, imagine that a neuron is selected for Zone-out, another it isn't, *the unit that is not selected will update its state from*

time t-1 to t as usual with a recurrent update, the one that is selected for zone out will update the state by copying h_{t-1} in h_t, you skip the actual update, you're actually copying, sort of identity, *through identity gradient flow nicely, no dampening, sort of stochastic shortening of your unfolding, reducing in a randomized way the amount of dampening you have in the gradient*, **static because even if it dynamically determined which units are being update and which not, by random sampling, no learning in which unit to update and which not to update**, it's random sampling→ static method, **effect of regularization but helps also with gradient vanishing, but you lose differentiability, your gradient needs to be approximated**. So to recap Zoneout, it is a form of regularization, at each time step, force some units to keep the same value. Random sample which unit to zone out. It is an hard gate, lose differentiability, needs to approximate the gradient. The idea is that avoiding (some) update improves the gradient propagation. Zoneout and Dropout are regularization techniques used in neural networks to prevent overfitting by introducing stochastic behavior during training. **Zoneout:**

$$\mathcal{T} = d_t \odot \tilde{\mathcal{T}} + (1 - d_t) \odot 1$$

Dropout:

$$\mathcal{T} = d_t \odot \tilde{\mathcal{T}} + (1 - d_t) \odot 0$$

Here, d_t is a binary mask sampled from a Bernoulli distribution, $\tilde{\mathcal{T}}$ is the transformed value (e.g., updated hidden state), and $\odot$ denotes element-wise multiplication. In Dropout, masked units are zeroed out, whereas in Zoneout, masked units retain their previous value (represented as 1 here, implying identity or retention). This subtle difference allows Zoneout to maintain temporal consistency in recurrent models like RNNs.

Rather than looking at unit level, let's look how to modularize the network to make *block of unit operates in a synchronized way and at a different update freq. than other*, **clockwork RNN**: *modules in my RNN that operates at different clock*, like a processor, my input is sequential data, so it is input in time, we have a Recurrent layers, is modularized, the thing within the dashed rectangles are not copied, but different neurons, *module M1 contains a certain number of neurons, M1, M2 ,...,Mg each one has a t on the top, clock, operational frequency at which they operate*, strange connectivity, **every recurrent module receives input from the input and every rec module receive input only from the modules on the right hand side**, Module 1 receives input from all other modules, **Module G receives input from no one**, **Module 1 receives from all the other**, **Mg operates with a very very slow clock**, say your data comes at a certain sampling freq. say 100 Hz, Mg does not work at 100 Hz, it works at 1 Hz, it is slower, compute the updates $h_t = f(x_t, \text{and the previous state})$ not for all the t, but once every ten, once every hundred/thousands, the update that is computed by this module here is passed to the module on the left side, that operates at a frequency still slow but faster than G, when you move from right to left there are module that operates at faster operational time ($T1 < T2 < T3 < ... < Tg$, clock time of the mod-

ules), Mg will feed to Mg-1 that will feed to Mg-2 and so on, modules that are slower feeds to module that are faster, you get *modules hidden state that captures very slow changing features if they operate at low clock but they manage to capture very long range dep.*, cause their state is update once every e.g. hundred times, for them one update covers a length of hundred in the original time series, shorten the distance between elems, but you don't lose info, you gave modules at different freq. slow, medium and fast. you get hidden state that represents info changing slowly, info that changes at a medium pass, and info that changes very quickly. Equivalent to a single recurrent layer, in which the structure of the W_r matrix, the recurrent matrix, is defined in blocks, blockwise RNN, say we have k neurons, the recurrent matrix for your standard fully RNN, its k by k, every recurrent neuron in connection with everything recurrent neuron not, all neurons instead here speak with other neurons, neurons at the bottom right will speak only with themselves, block in the bottom right of this matrix with a certain amount of numbers different from 0, and the all zero in the same columns cause not connected with previous modules, also block in the top left of your matrix with the internal connectivity and below in those columns all the params of connectivity with the other modules. Block structured matrix with a lot of zeros, whenever you don't have connectivity between neuron in different modules. *From the parametrization perspective is a block structured RNN* but not only that because together with the parametrization *you have also the fact you're not updating all the hidden states at the same time*, this is a single layer, you can use whatever you want of the recurrent neuron, simple RNN, lSTM, GRU, convolutional filters, echo state ... **you just need to be able to define a block structured weight matrix and that some neurons are update at different speed than others**. This was a static method again. You define a priori which neurons will be updated and which will skip the update.
Define a skip freq. that varies depending on the module.
Adaptive method: **Skip RNN**, cause they skip update, similar to the LSTM, rather than having a single neuron you have a neuron with some gates, below s_t is the recurrent state part of the neuron, s_{t-1} is the previous state enters my network and x_t current input enters my neurons, my usual update is $s_t = f(s_{t-1}, x_t)$, f non linear, but there is a (logical) gate that whenever is one s_t gets updated as usual, when the gate is equal to 0, different (non)update rule $s_t = s_{t-1}$, skipping the update, copy the old hidden state s_{t-1} in s_t. How to decide when to switch on the 0 or the 1, u_t is that decision, or 0 or 1, other neuron $f_{binarize}$ that outputs a binarized output (0 or 1) used to make that decision of copy or update, and that is a neuron, that has an activation function, that takes as input u_{t-1} the previous decision, what I did in the past in terms of decision to be active or not, the activation function is to binary values, a thresholded gaussian or a threshold/step function. But a neuron with its own params, that decide when it is 0 or 1, when update or not the memory. At every time I check what binarize decision update/not update I take in the past, I take the decision for the current time step, I use to update the state and to the next time step I 'll send my recurrent state update or not updated and the u_t that I used in this layer. This binarize decision is not differentiable again but it is a step

function. Visualize this model on linearized MNIST digit, you feed into your NN the pixels in order from the top left to the bottom right, in red overlay the pixels for which u_t is set to 1,and in blue when set to 0, the model has learned to skip to update the hidden state when there is no pen stroke in the picture, all the areas of the image when the model learns that does not matter they're not used to update the state, no info to put in the hidden state, and you are also shortening artificially the length of your propagation cause you are applying the id function when not update the state and gradient flows nicely, the gradient is generated at the pixel on the bottom right, when you create the classification, no dampening because the gate is closed in the blue area, **really compressing the unfolding of your network, lesser and lesser multiplication of your gradient**, your memory gets cleaner, no noisy measurement in input when not relevant (forward stage) AND IN BACKWARD YOU DO NOT MULTIPLY SMALL NUMBERS whenever is not necessary, your gradient is cleaner and nicer. So shortening the seq. helps the gradient to flow. no dampening in all the blue areas, because the gates are closed, a lot of compression of the relevant elems of the seq, you do not multiply small numbers when not necessary.
Hierarchical Sequential Structure, look at the structure of your problem to reduce the number of updates, **in many sequential problem there is a multiscale hierarchical organization** e.g. in natural language, when written, you can interpret character by character high freq, word by word medium freq. or sentence by sentence low freq., use this structure in the problem to inform the structure of you RNN→ Model in which the hidden states operates consistently with this structure of your data. **Hierarchical Multiscale RNN**, Multiple recurrent layers to extract features at different levels of abstraction, how many of them? NLP problem, input info characters that forms my natural language, first recurrent hidden layers, will learn to put together characters in to word, the second to put word into sentence, this will happen naturally even in a Vanilla RNN, **can I modify the way in which the update works in the different layers to respect the structure of the problem**, instead of a single operation I have 3 different operation on my hidden state: UPDATE standard LSTM/GRU hidden state update, h_t = whatever magic thing the gate in a LSTM do of h_{t-1} the usual update, COPY is what the skip RNN is doing, $h_t = h_{t-1}$, FLUSH is the operation that says ok, I'm layer 1, done my job, I send my hidden state h_t at layer 1 to the layer 2, and reset my state to 0 (flushing). Imagine you are layer 1, the first layer after the input, you see a t in input and you decide to update your state, you say an h in input and you decide to update your state, same for an 'e'... then you see a '_' space in input and you say stop updating and flush my state, send to the next layer, an hidden state computed in correspondence of the 'e', which gets all info that you need about 'the', to the next layer and you reset cause now you are ready to read a new word, the second layer am i receiving something useful at time 1 from below? no skip, at time 2? no skip, at time 3? no skip, at time 4?yes I receive the flush from below, ok! So update and you'll keep updating until below you see a point, sends an update at the second layer, the model sees it has an encoding of a full stop so it flushes to the level above. Connection intra layer can only

be copy or update, connection that go from hidden layers l to hidden layer l+1 are flush or copy connection and the model decide when activate them. **Your model really learns when to use update, copy and flush this thing work when your problem has a hierarchical structure.**

20 Neural Reasoning

Try to create a module that we can use to store long term info. The difference that is between Cache and RAM, rather than RAM and the disk. The cache is the dynamic memory, the RAM is a memory that as soon as you don't switch of the RNN the information stays. Do in the context of **Neural reasoning: attempt to make sure that my NN is capable of running algorithm**, computer science alg., reasoning alg. (logic), the algorithm only works if all the preconditions are met, *I need to store the input in order to run the alg., but NN works also with approximation*, I have to navigate a Robot from point A to point B, I can always use a path planning alg. if I have the cost of the paths, if I'haven't I need to interpolate, to use proxy info (sensor information) doesn't tell me the cost of the path but some info, measured from the camera of the robot e.g. the NN can learn to put info captured from the camera of the robot in relationship with the cost of the path and run the path planning alg. internally. *Need the ability to store somewhere the info*, in order to run an alg. I need a place where to store the data or intermediate computation of the alg. question answering, simple problem that nowadays people solve with NN, requires storage of information, I give you 4 facts and a question, you will know the answer by your logical reasoning, you have applied in your mental model of the word some reasoning, but for a NN to implement it either you are transformer and artificially put everything in the same seq., just a matter of computational complexity or **you need to store the info in a neural memory and query it to solve a question answering problem or a logical entailment**. This requires memorizing fact, question and answers. A bit too much for the dynamic RNN memories you need an external memory. Recurrent (sequential) models as general programs. Reasoning abilities, typical of classic algorithms or classic AI. We don't use a classical algorithm cause we may not have a proper input e.g. a pathfinding graph but only sensor info. The model needs to learn to encode the structure from the raw data and then solve the problem

Memory networks, *there is a NN that does the job, make the predictions, gets the input, whatever*, **connected with an external mechanism that is itself a NN** (I need to be able to differentiate everything) but is modularized and they behave differently, **one mechanism to write into the Neural memory and one to read from a memory**. **Read on pre-written memory**: **input feature map** $\rightarrow$ *encodes the input in a space (feature vector) to confront this input with whatever is stored in the memory*,

Generalization part: decides what input (or function of it) to write to the

memory, **output feature map** that reads the relevant memory slots and a **response** that integrates the info read from the memory with the info received in input and generates a response.
Fist approach can only **read pre-written memory already populated by somebody**, question answering problem: receive a question: "Where is the apple?", The NN needs to reason on the facts it knows, the different position in ordering where John went, **the question is in input in the NN**, **the facts are stored/precompiled in my memory**. **Put in relation the question with the facts stored in my memory**: question is a seq. of words, embed the question into a **vectorial representation q** that *represents my question compressed in a vector, of the size of the vectors I have in my memory*, **my memory is a matrix of vectors, where each vectors is the embedding of a fact/sentence**, M_i is a vector in memory that embeds some info, I have a certain number of facts, and I have my *query vector u* of same size of M_i, we can make the inner product between the query and all the elems stored M_i, for all the different choice of i, goes to a softmax to **compute attention coefficients**, in order to understand how to use a memory is taking my input u and compute the cross attention between u and all the vectors embedding the fact M_i in my matrix/memory, α_i or p_i attention weights, the large p_i memory M_i is highly relevant for u, very long vector of attentional weights same size as my memory, **the weights are creating an indexing vector for the memory**, tells me how much to read from each cell/fact stored in memory. $p_i = 0.5$ read 0.5 from M_i if $p_j = 0$ read 0 from M_j. Read: going through the embedding of my facts into vector, my facts can be embedded into a set of weight to compute the reading vector/the memory addressing vector, the alpha weights, and into a different embedding space same facts for the purpose of reading. I'm gonna be doing a memory read selecting from each fact embedded as a vector c_i, in proportion to the corresponding attention vector p_i, summing them up with a weighted summation, and getting **a single read O which is the convex combination of all the memories weighted by their attention weight**, **an attention vector is an addressing vector for a neural differentiable memory** that tells me how much I'm reading from every elem stored in memory, and every element stored in memory is just a vector of information, aggregating them all in a single read O and O is my final read from the memory, integrated with u, multiplied by some params and used to predict the answer, **at this point I generate the error I can backpropagate through C, A ... and update all weights**. **Everything here is entirely differentiable**, when reading the memory I'm not reading a single memory but all cells with probability 0, no hard choice, due only to the fact that the softmax assign 0 to that particular memory, all differentiable, gradients flows nicely, I can learn all the embedding matrix and all the params of this model... This only implements a reading from a neural memory, the facts are predefined, I'm learning how to embed them into m_i, c_i but no control on the fact that somebody has introduced for me. This can be iterated, here we have a single read from my memory, the model I see before implement a single access to the memory in order to output a response, but *sometimes I have to go through multiple time through my facts*, *I just replicate the models*

as many times, you can compose, memory network with 3 step of reasoning on the facts in order to get to a prediction. The number of unfolding are fixed, you define them architecturally but in principle **you would like the Neural Net to do multiple access to the memory for the number of steps that it needs to unfold, mechanism that can unfold over time implementing this? RNN**, *the part that decides to generate reading vectors to read from the memories come from a RNN, that unfolds itself for as many layers as necessary, so this reading from the Neural Memories comes from a RNN that unfolds as necessary and this constitutes the base for the Neural Turing Machine*, **use a RNN to do multiple steps of reading from the memory and introduce a step of writing into it**. To recap, memory network extension, you can predefine a number of read in the neural memories, to simulate multiple step or reasoning, just replicate/stack the architecture, often with tied weights, but this forces a fixed number of steps, if you want to dynamically query the memories as many time as you want, combine with a RNN.
Neural Turning Machine has a **controller** which decides which parts of the memory to read, which part of memories to write, has **reading Heads** and **writing heads** and has **a neural memory** that can be read and ca be written. Memory networks that can read and write memories at both training and test, end-to-end differentiable.
You can easily create a NN that can answer to fact where the facts are about an image, how do you tranform an image into facts, you divide it into patches, you encode each of the image patches through a CNN into an encoding, the facts in your memory are just a vectorial encoding obtained by a CNN of the single patches that you have in your image, and whenever you receive a question "What are sitting in the basket of a bicycle?" This is a question you need a way to encode it into a single vector, use a LSTM encoding into a vector same size of the representation of the input info, use my memory mechanism to match the query that comes from the Natural Language question, with facts contained into the part of the images, and identify in which part of the image there is a match between the query and the facts, through the soft attention mechanism, combining everything together, output an answer, because I can reason on the single pieces of the image an put in relationship with the question. And if you look to the attentional weights of the different facts you can see which part of the memory you have read to generate the answer for that question. Cause the fact are the parts of the image.
Neural Turing machine allows to unfold this reasoning for possibly an infinite number of times and to read and write from the memory and to do it in a fully end-to-end differentiable way because we want the gradient to flow everywhere
Neural controller acts as an interface between the neural memory and the external word, is the one that **receives the query from the external word, the controller receives a question, the input received at time 0**, and **the picture is another input**, *the neural controller decide to takes the input, question and image, write them into the memory and start reasoning on them*, **by unfolding in time**, **start seq. of read and write from the memory** to try to understand if there is an animal in the picture e.g that has

been positioned in memory by the controller itself at the first time step and **the seq. of reading and writing are self decided by the controller that can do that because it is a RNN**, *at every time step of unfolding of the NN the controller decides whether to read or write from the memory or where to stop and output a prediction*, this is job RNN can do, not more difficult of generating a seq. of words and decides when to stop the generation. **The controller is a RNN**, has an input possibly at each time step, but sometimes you can receive a dumb input, no input, question, an image inputed at the first time step and then I'm all giving only dumb input cause it is time for the Recurrent controller to ponder, to reason, about the input it has received. *At each time step the RNN output something*, at the beginning the output is not relevant so it outputs a blank until it arrives to the final time step in which output a true target/a true prediction. **The controller internally interacts with the memory**, at each time step the controller generates read and write to the memory, so for instance at the first time step will generate a seq. of writing to write all the input info it has received, at the second time step probably it won't write anything on the memory it will read something that uses at the third time step to write something, reread next to make a prediction, but all determined in a complete adaptive way from the task. To recap Neural Controller, an RNN emitting vectors to control read and write from the memory. The **key to differentiability is to always read and write the whole memory**, use soft-attention to determine how much to read/write from each point.
Memory read: Neural controller generates an addressing vector to read from the neural memory, as we read the memory in a computer by generating a seq. o addressing in the RAM to be read. Only to preserve differentiation I need to read always the whole memory if I don't want to read something the weight will be set to 0. A memory in a Neural Turing Machine is just a matrix of vectors, I read all vector proportionally to some coefficients αs, that sum to 1 and each elem between 0 and 1, the higher the coefficient the more I read from the cell, the attention vector modulates how much I read. So whenever I read from the memory I'm gonna be summing all these vectors modulated by the weight in the attention vector. So my memory is made of a matrix of vectors, each vector is a memory, each vector is associated with an attention weight, and the addressing vector to read the memory is a vector of attentional weights, and a memory read is just a summing of all the vector that I have in memory weighted by the corresponding attention weight, fully differentiable, then it is just a matter of being able to generate appropriately the address in memory. The memories r is the sum overall the memories of α_i M_i the weight of the read and the vector corresponding to the read. **Also memory write on all the cells**, a specific location entails selections that is not differentiable, write the value in all the cell/slot of the memories, but I can do it smartly by using an addressing vector,the value w needs to be written at this address a lot and in the others I don't write much of the content. **Very little alpha values, it means that the new value w, will be combined very little with the past value of the memory**, so the new memory will be very similar to the original one, instead the cell the corresponding to much bigger alpha values will

be more influenced by the new value that I'm writing down. Writing something on the memory is writing something in all the cells of the memory but proportionally to what the addressing mechanism tell me to do. This will keep the thing totally differentiable, writing into the memory is just a summation of two vectors w and M_i. $M_i = alpha_i$ w + (1-$alpha_i$) M_i.
So to recap memory write, we have a value to write w, there is attention distribution vector α describing how much we change each memory. Write operation is actually performed by composing erasing of the previous cell value and adding of the new value.
Neural Turing machine: Neural network that can read and write a memory, where you can store info on a more voluntary basis and longe term basis than having it captured in a dynamical state like a RNN, a Neural memories is a matrix of vectors, each fact/memory in the memories is a vector in this matrix, the position of the vector is the location of the memory, I address the memory with attention vector, how much to read from every location in the memory, or how much to write, memory reads and writes in order to be differentiable need to read the full memory, the addressing vector is a vector of same size as number of location I have in memory and how much I' m gonna be reading from the corresponding cell in memory is told by the attention coefficient, the thing I am reading is just a convex combination of those vectors/locations weighted by the attentional vector, reading a memory amounts to generate a properly addressing vector, writing a memory is writing a same vector in all the existing location in memory, write them all to be differentiable, write as much as alpha values from attentional vector tell me to do.
How to generate this attentional vector, reading and writing is the job of reading and writing head, **generating attentional vector is the role of the controller**, **attention focusing by the NTM is managed by this controller and it is multi step process**, not an unique phase/step, but rather generate the final read by combining content-based addressing with other steps like scanning e.g. The first step in the generation of the attentional vector for the reading and writing, is creating as usual an attentional vector by comparing a query vector, generated for instance by the input I've received or the internal state of the controller, it is a query vector that tells me I need to do something on the memory that is related to this query vector, that get confronted with all the vector in memory through a dot product, and this will, by appropriate application of a softmax, become the usual candidate attentional vector, after this first step I obtain an addressing vector which represents how much the single memories match the query, it's pure content based addressing in terms of memory. **Once you have this content based initial address called A for simplicity we do something to it, we interpolate it, we mix A with the previous attention vector**, I keep the previous attention vector used for the previous access into the memory either for reading and for writing, **I mix A with the previous attention vector, with an interpolation amount that is regulated by a value generated by the controller**, the controllers learn how much to trade between the new candidate vector for the content based addressing and the previous attention vector and then I get B in response, to have

smoothness in the access to the memory, avoid inconsistent jump between two different access in memory, the interpolator could decide to switch off completely the contribution from the previous attention vector, **once you have B is not finished because now you have a content based addressing vector to access the memory that has been mitigated by the previous access** and that is now **integrated with an information that comes from a shift distribution filter that is convolved with the attentional vector**, we are literally applying the convolution, that shift distribution vector is generated by the controller and **moves the probability mass of the softmax either to the left or to the right or focuses to the center, the goal is to shift my access to the memory in certain directions**, the NTM stores in the memories various numbers M1,M2, M3 .... in the Neural Memories and you ask the NTM for the larger number, you usually would do a for loop on the vector as a good computer scientist, that is what does the shift distribution filter, first access to the memory with a very high coefficient/distribution to the first elem and zero or little things on the rest, this will give an accessing vector at the first stage that will allow you to read from number one, then the shift filters apply this thing here a shift that takes the probability mass you have here and move it to there, by having my distribution filter like -1 0 1, when you convolve with the vector above you shift little the probability mass of the above vector on the right hand side, through repeated application of the shift distribution filter you are allowed to scan linearly your memory, with just content based addressing you can't do that, **shift distribution filter allow you to do scanning of the memory, provided that you can generate adequately the filter values** A revised attentional vector, **sharpening to access as few locations as possible** rather then little a little bit of all of them, to obtain a more focused access, here all is differentiable cause you read everything and you write everything, **all these vectors are just predictions outputted by a RNN**, which is the thing that is implementing the recurrent neural controller, everything is end-to-end differentiable, an error is generated whenever you output your predictions, so to me the larger number in the vector is 42, you generate the error, and you backpropagate through all the decision that the NTM has taken and you adapt the params.

The real value of these model is not solving actually practical problem is just to show what in theory can be done with such a model and enhance a Neural Network with data structures, the neural memory is the simplest case, you can implement queues, stacks ect... How do you embed in a NN a different form of inductive bias, the one of the structure that we create as computer scientist, this model implements data structures that you can't manage to do it with RNN, LLM solve this task now but in a different way, they do not have an actual area where they store info in about the input and that this area is structured according to a data structure, they solve it by essentially having seen program to do that so they can generate a program to solve your task.

21 Generative Deep Learning

Generative learning is much more than generation. Generative/Probabilistic Deep Learning model is about focusing on distribution rather than learning how to solve a task, model more general and robust, when you focus too much on discrimination, on separating things rather than understanding things, lot of problems in which you can incur, you might be easily fooled. Deep learning models focuses a lot on build walls between distinct classes as a discriminative model would do, build a separating curve to reach a decent classification accuracy, if I am a bastard attacker, I create some input that is exactly like one class and I classify like another class, looks exactly like an elem of a class but the label is of the other class, the model has only learned how to distinguish them, *adversarial attack keep the appearance of a class and label of another, adversarial attack reverse the direction of the gradient to decide how to position examples*, to generate a sample that positions exactly on the side of the wall that I like, while keeping the appearance to fool the model, take a picture and inject some invisible to the naked eyes to this picture so that the stop sign is interpreted as a sign that tells you have precedence, go ahead with your car, attack that flip/change just one pixel (one pixel attack) that change the prediction of the model, sticker attached on the stop sign and that will alter the perception of the model in real life. Focusing instead of **learning the characteristic/the data generating distribution** is a good idea because it becomes more difficult to fool you, not only learning how to generate new data, but learning things about your problem/data distribution.
Understanding data you understand the world better, if you want to put AI running around in the world, it's better that you make sure the model understand the world rather than showing it understand.
Understand the data variance, you understand that some changes to your data make sense and others do they make sense? Distinguish between different styles of an image w.r.t. to non-sensical images.
How do we learn distribution from a DL perspective, powerful function approximator that are NN, you can take very little assumption of the generating data distribution, from the probabilistic model we take the full theoretical framework that tells us how to approximate nicely the distribution. *Take potentially unlabeled data and learn p(x) the data distribution, fit our model, NN theta to learn the distribution $P_\theta(x)$, which closely approximates the data generating distribution P(x).*
Why Generative? Focusing too much on discrimination rather than on characterizing data can cause issues: reduced interpretability, adversarial examples. Generative models (try to) characterize data distribution: understand the data $\rightarrow$ understand the world, understand data variances $\rightarrow$ learn to steer them, understand normality $\rightarrow$ detect anomalies. Approaching the problem from a DL perspective: **Given training data, learn a (deep) neural network that can generate new samples from (an approximation of) the data distribution**. We have our TR data, distributed $\sim P(x)$, we have our NN θ and we want to generate data, according to $\sim P_\theta(x)$, an approximation of the data

distribution. Two approaches: **explicit** that learn a model density $\sim P_\theta(x)$, **implicit** that learn a process that samples data from $\sim P_\theta(x) \approx P(x)$. Rather than learning an approximation of the data generating distribution, we gonna be learning an approximation of the data generating process. **2 approaches: Explicit vs Implicit**. The first really try to learn the density P_θ and make sure that this matches the data generating distribution P(x) as much as possible, the second instead we'll learn a process not a distribution, a process that samples new data and from these data that we sample, the likelihood is gonna be consistent/comparable with the data generating distribution, but we never have an approximation of the distribution, only a sampling process.
The all field of Generative DL can be partitioned in explicit and implicit model: **the first train a model to approximate directly this distribution**, there are multiple ways in which we can train a model to approximate a distribution, *the key difference is whether we try to approximate a distribution using only visible variables*, so the kind of distribution we operate on is a **likelihood**, the likelihood is defined over only visible data, if we directly try to model the likelihood we are in a word where we learn based only from visible data, not incredibly different from what transformer does, transformer is an auto-regressive model, **the decoder part of a GPT is an explicit, purely visible, generative learning model, you generate the next token based on the previous token, all visible information, the modeling of the distribution is purely based on visible information**. **Latent based model** instead, because the distribution based on only visible things is too complex to be modeled, I cannot factorize into simpler distribution, I invent some unexisting, not visible, random variable than things factorize. The first one leads to **tractable densities**, provided that we know how to compute the visible variables, *as soon as we introduce latent variable a nasty integral shows up, an integral over the non-observable latent variable, because they are not observable we need to marginalize them out*, only our latent variables, since we are in a NN, are going to be neurons, so it's going to be an integral over the activation of neurons, so kind of a nasty one to deal with, something that cannot be touch in closed form, needs to use approximation, which kind of approximation variational approximation based on **ELBO** or **sampling based**, approximating expectation with sampling. Explicit model, fully visible variable models $\rightarrow$ **Sampling RNN** about images, or **Flow-based models**, *generalization of sampling RNN, powerful explicit models that can really learn the data likelihood.* These are the only class of full explicit models that can actually learn the data likelihood, with the data likelihood you can do for instance anomaly detection. **Variational based latent variables models$\rightarrow$ Variational autoenconders** and **Diffusion-based model** explicit, latent representation of my information. In the stochastic approach, we have already seen an example, a model that is a NN with latent space repre. trained with Gibbs sampling, is Boltzmann machine. **Implicit hand we are learning sampling process**, we'll see 2 methods: **direct based** and **stochastic based**: *where in your model you position the sampling, if you position it smartly outside the chain of differentiation it means that you keep everything differentiable and you get a direct model, if*

you instead place the sampling within the core of the model, also that works, but your gradient at some point won't be differentiable and you'll need approximation in order to be able to compute the gradient. For the **direct method** we'll see **Generative Adversarial Network**, for stochastic method a relevant model is Generative Stochastic Networks but we won't see .
Density learning with full observability, **models that actually learn the data likelihood**, learning with fully visible information is essentially application of the chain rule, we have some data X can be images, we want to learn a distribution over images, we factorize over the elems that make up an image, the pixels, product over all the pixels of an image, I just need to decide which ordering to scan my image to generate the image, pixel by pixel, one after the other, left to right, top to the bottom, *the conditioning side tends to grows quite very much as the image grows*, when you generate the guy on the top left depends only on itself, when you generate the last bottom right depends on the rest of the image, *when the conditioning side is too big, use the independence assumption*, instead of conditioning on the value of all the preceding pixels, condition only on the nearby/neighboring pixels. You know what instead of conditioning on the value of all the preceding pixels, I'll be conditioning on the value of the neighboring pixels, plus some additional info that sort of summarizes all the pixels that I've seen so far, a model that is able to remember the info I have seen previously? **A RNN to parametrize this distribution**, because it can learn what pixels it has been seen before
I will generate the probability of a pixel having a certain intensity value given the known intensity of its predecessor.
To recap: if we do learning with fully visible information, the joint distribution can be computed from the chain rule, it's a Bayesian Network: $P(\mathbf{x}) = \prod_i P(x_i \mid x_1, \ldots, x_{i-1})$. So we are modeling the probability e.g. of an image, by using the chain rule, learning the joint probability of all pixels, i.e. of an image, it amounts to learning the probability of a pixel having a certain intensity value, given the known intensity of its predecessor, then just apply the chain rule $\rightarrow$ job for a RNN. To do that we need to define for sure a sensible ordering for the application of the chain rule, and the conditional distribution can be difficult to compute in case of long conditioning sets. All of these probabilities, in the simplest case simplify to $P(x_i|x_{i-1})$, this makes an image exactly a seq of pixels and I am modeling the generative behavior by saying that the current pixel is generated knowing the previous pixels and possibly some info about the predecessor, compacted into a state vector, recall very much the problem of generate the next character of a sentence, something already seen and modeled with LSTM and any other language-based model, where we generate in an autoregressive mode the next character, here we generate in an autoregressive mode the next pixel of an image, use a RNN to do that. Generate our image of a cat producing the RGB value for each pixel in output for this LSTM, conditioned on the values of the previous pixels, when we use a RNN we generate in output the RGB value for a pixel, receiving in input the RGB value from the predecessor, and using the hidden state info given by the recurrent layer that informs about the all history of all the preceding pixels. *Whatever pixel I*

am predicting at this point I'll be informed explicitly by the previous pixel RGB values and implicitly through the state info of the recurrent net about all the history of the pixels that I have seen before. **Pixel Recurrent NN**. So we are approximating the conditional probability, scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN. PixelRNN models the joint distribution of image pixels as a product of conditional distributions using different generation orders. PixelCNN uses masked convolutions to condition each pixel on a local context of previously generated pixels. Row LSTM processes the image row by row, where each pixel depends on prior pixels in the same row and the row above. Diagonal BiLSTM extends this by sweeping across diagonals in both directions, allowing richer contextual modeling from multiple directions. These represent different scheduling policies for autoregressive image generation. You can decide you don't want to generate things by scanning linearly, but rather you want to generate things to go in both direction at the same time, two indep. mechanism, different policies to generate the seq. that generates the pixels. I can generate a pixel value by looking at the pixel values of the neighboring pixel in the previous layer of a CNN or you can do it with a row base LSTM ... **many schedule of generation of pixel**. Images that from very remote can look realistic, but they are not so good if you look it closer, *generating image pixels by pixels is a nightmare*, it takes forever to generate a 32x32 seq. one pixel at a time, when you generate the final pixel, conditioned on the hidden state that should remember things on the first pixel that was in the image, **gradient vanish and explosion**, fine but it does not work, you actually learning the distribution over your pixels, data likelihood model of the original data, pixel by pixels is simply to much. Model that does the same thing, operates and put a distribution on real chunk of the data, but the chunk cannot do the single pixel→ flow based model operate on a much bigger scale, model more scalable and more capable of capture more complex interaction between faraway things.

Problem is too complex to be addressed with only visible info → distribution don't factorize → only one thing to do, introduce simplifying assumption → **introduce latent variable**, $p_\theta(x)$ does not factorize nicely, we introduce z, a latent factor, that will allow us to obtain a distribution $p_\theta(x|z)$ which in principle should be much much easier to manipulate than $p_\theta(x)$, modeling only the data generating distribution/data likelihood p(x) is too difficult, even if I try to factorize I cannot really simplify the distribution, there is no other way that *introducing an additional set of variables, that allow me to reason in terms of two easier distribution and make sure that the two obtained distribution are simpler*, $P(x|Z)$ and $P(Z)$, and my hope is that those distribution are gonna be easier to treat than $p_\theta(x)$. Latent space z and what this factorization is saying, from a generative perspective, is that if I want to compute the probability of x what I need to be doing is pull up a Z from a prior distribution P(z) and using this distribution P_θ decode x from the z I've drawn, since there is an integral don't do it once, do it multiple time, on average, in expectation I would have the P(x) that you like, underlying this thing is potentially a sampling process and p(z) is a very easy distribution you can easily sample from it.

From visible to latent variables: with only visible information, we try to learn the θ parametrized model distribution $P_\theta(\mathbf{x}) = \prod_{i=1}^{n} P_\theta(x_i \mid x_1, \ldots, x_{i-1})$. Now we introduce a latent process, regulated by unobservable variables z: $P_\theta(\mathbf{x}) = \int P_\theta(\mathbf{x} \mid \mathbf{z})\, P_\theta(\mathbf{z})\, d\mathbf{z}$. This is typically intractable for nontrivial models (cannot be computed for all z assignments). **Z is the distribution over neuron activation**, cause it's a NN, my random variables here are neuron activation, Z is actually a vector of neural activation, **simplest distribution to sample P(z) is a gaussian**, and now that I have Z, I know a simply enough way to transform a z into an x? yes **a NN, that takes in input a z and spit out an x**. **Problem? Integral and the fact that my z are not informed by x, when training the model z needs to be informed by x to be decode back**. Intractable in close form we need to work on that, a neural network with latent variables? **Autoencoder**, P(x—Z) the autoencoder takes x, put into a z, and decode it back, the z generated is informed about x, works like that needs to make it more probabilistic, instead of being a deterministic transformation between the original x and the reconstructed one, we need to insert some stochasticity, denoising autoencoder have a probabilistic interpretation, noisy version of x and return x, **here x is not mapped into a single z but on a distribution of multiple x**, here I'm trying to learn an autoencoder that autoencodes in probability an x into z and decodes in probability z into x, in probability means that **x is not mapped into a single z but to a distribution over z, multiple z with a certain probability**, same for the decoding distribution, you gave me a z and with a certain probability I will give you a set of x, the decoder is the part involved in my factorization $P(x|z)$, if I want to map this into my model I need to focus on the decoder part and make sure that learn $P(x|z)$ and that somehow here I have a data generating distribution that gave me P(z), the encoding should model P(z). **We are gonna be sampling the latent variables from a true prior P(z) and then reconvert into x**. Of course I do not have access to the true once so I'm gonna be making assumption on what it is form. **Z will come from a normal**, standard gaussian, zero means and unitary variance, the decoder $P(x|z)$ will be approximated by a NN, very deterministic, if I choose my z like a simple standard gaussian distribution, 0 mean and unitary variance the z has no info about the original x, **no information in Z by the original x how do we expect to reconstruct/decode back x?? the gaussian we are sampling needs to be informed from x**, the params of a gaussian are the mean and the variance, the only way that I have to control and make sure that the gaussian from which I am sampling has info about x and sigma is that **the variance and mean of that gaussian needs to be function of x**, instead of sampling from a standard/normal distribution **I will sample from a gaussian with mean μ_x and variance σ_x** how do I get these two values???? This makes sense, if I have this Gaussian, sampling a Z from this gaussian where the mean and std dev are informed by x will give me a z that is informed by x, where do I get to μ and my σ?? by a NN, the encoder, the job of the encoder is to take x and generate z, but z gets sampled so the job of the encoder is to generate μ_x and σ_x. Passing over to the gaussian and the gaussian will generate the z and that will be used by

the decoder to generate x. **Problem??? Z is generated by sampling and sampling is not differentiable!!!!** We need to do something more. *I cannot send the gradient to the encoder in such a way it generates the right mean and variance*. But this is the overall framework: the encoder becomes the part of the net that generates the mean and the variance that are informed by x, from which (with a gaussian with those params) I sample a Z, that is informed by x, which I pass to the decoder, which learns in a deterministic how to transform that z back into x. No differentiability because of the sampling operation.
So to recap: we were looking for a NN with latent variables, because we need them in order to simplify the problem, but we have already seen an example AE neural net that takes a x encodes in a z and decodes back x'. It is not difficult to cast a probabilistic twist on AE (by making encoder-decoder maps probabilistic). x in this case is encode in a $P_e(z|x)$, a distribution and decoded in a $P_d(x'|z)$ another distribution. A Deeper Probabilistic Push: As an additional push in the probabilistic interpretation, we assume to be able to generate the reconstruction from a sampled latent representation. z $\rightarrow$ Sample latent variables from the true prior P(z), x' $\rightarrow$ samples from the true conditional $P(x'|z)$. Of course we don't have access to the true distributions, so we have to approximate them.
Variational autoencoder (VAE) - The catch: Sample z, latent variable, from a simple distribution such as gaussian $\mathbf{z} \sim \mathcal{N}(\mu(\mathbf{x}), \sigma(\mathbf{x}))$, represent the $P(x'|z)$ distribution through a NN g (the decoder). At training time sample z conditioned on data x, and train the decoder g to reconstruct x itself from z.
VAE Training: ideally one would like to train maximizing the likelihood. $\mathcal{L}_D = \prod_{i=1}^{N} \int P(\mathbf{x}_i \mid \mathbf{z})\, P(\mathbf{z})\, d\mathbf{z}$, but it is not as easy as it seam, intractable $\rightarrow$ **variational approximation**. *Training by maximizing the maximum likelihood problem with the introduction of the latent variable using back prop ascending gradient*, that integral, what I am interest into is the complex distribution of the data, u-shape, complex distribution which I do not know how to model, this latent variable model is telling me, rather than model the thing in the x space directly, I introduce a volume full of slices, each slice an instantiation of z and infinite once because continuous z, once I concentrate on a slide, on a specific choice of z, I can model x with a gaussian centered on x, very simple distribution, how can I obtain the original complex distribution? By integrating out over all possible choice of z, and all x is a gaussian, so I am obtaining this complex distribution by summing all the little gaussian that I have in the distribution of x and obtain this complex distribution as infinitesimal sum of this gaussian. Transformed a very complex u-shape distribution into a sum of very simple gaussian distribution, in principle it works in practice no cause of the damn integral that does not allow me to write things in a close form. Rather than maximizing this thing which I don't know how to maximize I will maximize the expectation lower bound $\rightarrow$ ELBO $\rightarrow$ rather than maximize this thing that is complex I 'll maximize the ELBO $--$ variational approximation, my ELBO: $\log P(x \mid \theta) \geq \mathbb{E}_Q[\log P(x, z)] - \mathbb{E}_Q[\log Q(z)] = \mathcal{L}(x, \theta, \phi)$
Maximizing the ELBO allows approximating from below the intractable log-likelihood log P(x)

$\mathcal{L}(x, \theta, \phi) = \mathbb{E}_Q[\log P(x \mid z)] + \mathbb{E}_Q[\log P(z)] - \mathbb{E}_Q[\log Q(z)] = \mathbb{E}_Q[\log P(x \mid z)] - \mathrm{KL}(Q(z \mid \phi) \,\|\, P(z \mid \theta))$
Where $P(x|z)$ is the decoder estimate of the reconstruction based on a sampled z, from the second term we see from the KL that we need a Q(z) function to approximate P(z). Remember that the sampling introduces non differentiability.
On the left hand side log likelihood problem you don't know how to maximize, on the right hand side of the inequality there is what you can optimize. You can slightly manipulate $P(x, z)$ is not a distribution we know, we now $P(x|z)$, substitute with the chain rule and thanks to the logarithm it separates nicely, then the functional of Q that I am gonna be approximating, **Q is the magical approximation function that will use to optimize my model even if I cannot compute the exact posterior P**. The obtained equation is very interesting, **the first term is a reconstruction term/error**, maximizing this thing I will be maximizing the fact that I generate sample x which looks realistic, this math term in english tells that this thing is maximized when the x looks exactly like something that I have in real data, said in other ways this thing is maximized when the reconstruction error is low (minimized), the second term is difference between two logarithm in expectation, should immediately recall Kullback-Liebler, it's a KL-divergence between the Q and the P, the variational distribution over Z, and the exact uninformative prior over Z, the gaussian. This start to become something we can optimize, when you plug all the necessary info in the first term, it becomes a MSE, you are gonna be assuming that your data is gaussianly distributed and the first term will become MSE, the second term is KL-div, untractable in general but special case, **KL-div between two gaussians**, the P is a gaussian, so we make Q a gaussian and that thing writes in closed form, the integral will close and will be a difference of means and a difference of variances, something very very simple, once you assumed that your 2 distribution Q and P are gaussian, the KL can be written in closed form. P is already gaussian, just need to make Q a gaussian. But we need Q to be a function that can approximate P, **which thing I m gonna be using to implement Q? a NN, the encoder**, the encoder is actually a gaussian, the work of the encoder is to take in input x and return μ_x and σ_x which is exactly the mean and the variance of a gaussian, my Q is actually a gaussian, so this thing saves the day except the non differentiability thing, but whenever you are in this case, in which you have my z that is sampled from a normal, where the mean and the variance comes from a NN, the sampling is a non differentiable operation, the gradient cannot get in two mu and sigma, cause there is a non differentiability barrier, what I need is Z to be gaussian distributed and located to have mean mu and variance sigma, is there alternative ways to obtain this?? one way is to pull z direct from a gaussian with those parameters, **another way is to pull z from a gaussian with 0 mean and unitary variance and then relocated by translation to have mean mu and rescaled to have variance sigma, re-parametrization**, we sample some noise, a vector ϵ from a gaussian with 0 mean and unitary variance, ϵ is a vector will have the same size as z, drawn from a normal but i want Z to be centered in mu and

with variance sigma, $z = \epsilon * \sigma + \mu$, now epsilon is mu-centered and with sigma variance. Now it's much much better, when I generate the gradient it's spread overall the directions the gradient needs to go during backprop, it tries to go in the mu path, can it goes there? yes, there is no non-diff. operation, it tries to go in the sigma path and it menages, can it goes to the epsilon path? NO, it is not differentiable, do I care? NO, there are no parameters to optimize there.
Now I have a full schema that I can train: Variational autoencoder: I have a decoder, a purely deterministic decoder that given a z returns an x, and will learn to generate x that are good reconstruction of good quality due to the first part of the ELBO, my z is now differentiable in the mu(x) and sigma(x) part because z is not generated by drawing it directly from a gaussian but first we draw an epsilon from a normal gaussian and the we relocate it to have mean mu and variance sigma, then I just need to attach the missing neural network, that is the encoder whose job is give me in input x, I encode your x into 2 values, mu(x) and sigma(x), then my z will have all the necessary info to reconstruct my x, x is encoded into a mu of x and sigma of x, that are vectors compressing information of x, it's a latent encoding of x, which becomes the aggregating encoding z with a little stochasticity, z has enough info on x it integrates mu(x) and sigma(x).
Autoencoder with stochasticity, make sure that it is no longer a deterministic one but a probabilistic one, it learns a distribution.
To recap: reparametrization trick, instead of sampling z from $N(\mu(x), \sigma(x))$ that introduces non differentiable opearation for your param $\mu(x)$ and $\sigma(x)$ sample ϵ from N(0,1) and reparametrize by adding $\mu(x)$ and multiplying by $\sigma(x)$, now sampling is limited to non differentiated variable and you can backpropagate through mu and sigma. So the full architecture at training time, we have the encoder Q that takes in input x and output $\mu(x)$ and $\sigma(x)$, the encoder network represents the distribution $Q(z|x, \phi)$ with ϕ params. The espilon sampled by the normal, is reparametrized by the mu and sigma informed of x, the decoder do the reconstruction, the decoder network represent the distribution $P(x'|z, \theta)$ with θ params.
VAE Training: Training is performed by backpropagation on θ and ϕ to optimize the ELBO.

$$\mathcal{L}(x, \theta, \phi) = \underbrace{\mathbb{E}_Q \left[\log P\left(x \mid z = \mu(x) + sqrt(\sigma(x)) * \epsilon, \theta\right)\right]}_{\text{Reconstruction}} - \underbrace{\mathrm{KL}\left(Q(z \mid x, \phi) \,\|\, P(z \mid \theta)\right)}_{\text{Regularization}}$$

The KL can be computed in closed form when Q(z) and P(z) are Gaussians, $\mathrm{KL}\left(\mathcal{N}(\mu(x), \sigma(x)) \,\|\, \mathcal{N}(0, 1)\right)$. Train the encoder to behave like a Gaussian prior with 0 mean and unitary variance.
From ELBO we know that we have 2 set of params, the set of model params $P_\theta(X|Z)$ and the params of the variational distribution $Q_\phi(Z)$, fine it's just 2 different neural network, 2 neural network one with param θ one with param ϕ, they are different, encoder and decoder, the function that we will be optimizing: the first term is the reconstruction error of x given a z, there is a relocation of a randomized eps, and the second term is still my KL, the second term is now making explicit one thing, I know now that Q is my encoder, my z is not $Q(z)$

is $Q(z|x)$ because z is informed by x, Q is so called informed distribution, now I can reason on what that KL-div means, the KL-div pushes the 2 distribution to be alike, which one can be pushed, only the first one, the second one P(z) is a gaussian, is defined apriori, we have decided that P(z) is the simplest case of a gaussian, 0 mean and unitary variance, the KL div there is telling us that I want Q to behave as P, so this is telling me that I want the informed gaussian, in which z is informed by x behaves as an uninformative gaussian P(z), so you Q generates Z that are as much as possible, indistinguishable from z that have no trace what so ever of x, so is putting entropy in my distribution, in Q, since the first term make sure that the backprop is pulling to have a lot of info about x in Z, otherwise I cannot reconstruct correctly otherwise I will make a lot of error in the first term, the second term instead is saying no wait you should behave like a non informative gaussian, put less information possible about x in z, so the second term is regularizing the first term, it's putting entropy in my q distr. which otherwise will tent to do if you leave it as is, Wow they are not watching let's put z equal x and we've done, NN are truly bad students, like those that try to learn things by heart, if you let them go they will really do that, KL part regularizes the reconstruction part by adding entropy to your distribution and make sure they are not too predictive. Kl between two gaussian nothing scary it is really a difference of mean and difference of variance, and it is actual KL between a normal with mean mu(x) and sigma(x) std and a normal centered in 0 and 1 std. Informed gaussian vs non-informed gaussian and this has a closed form solution.
VAE LOSS - Another view on differentiability: In principle we would like to optimize the following loss by SGD: $\mathbb{E}_{x\sim D}\left[\mathbb{E}_{z\sim Q}\left[\log P(x \mid z)\right] - \mathrm{KL}\left(Q(z \mid x, \phi) \,\|\, P(z)\right)\right]$ which can be rearranged following the reparametrization trick

$$\mathbb{E}_{x\sim D}\left[\mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\left[\log P\left(x \mid z = \mu(x) + \sigma(x)^{\frac{1}{2}} \odot \epsilon, \theta\right)\right] - \mathrm{KL}\left(Q(z \mid x, \phi) \,\|\, P(z)\right)\right]$$

No expectation is w.r.t distributions that depend on model parameters →We can move gradients into them
VAE Loss, in principle this is the ideal function we would like to approximate by ELBO, this is the function we cannot optimize by backprop because of the expectation run on Z drawn from Q, **whenever you have expectation taken w.r.t. to z that are drawn from parametrized distribution, the drawn/sampling from distribution that are parametrized break the differentiability**, infact with the parametrization trick this is what we are optimizing: the expectation from the x taken from the data (no params in the data, completely empirical distribution, differentiable) of the expectation of epsilon taken from a normal, normal no param perfect of the terms where the z is relocated as appropriate, what made the whole thing differentiable, is the fact that we traded the expectation over z from a parametrized distribution with the expectation over epsilon from a non parametrized distribution with makes everything smoothly differentiable. If you take the VAE Loss from an information theoretic point of view is actually counting the amount of information that is moved from one distribution from the other one, the **first term**

is actually counting the number of bits that you need to reconstruct x from z, the information theoretic interpretation of that first term is that, under the ideal encoding, we do not have the ideal encoding cause z is typically drawn from Q(z) not from P(Z) the original one. The first term is really telling me how much I need to have from z to recover x, the second term is telling me how many bits I need to use to convert a completely uninformed thing z into an informed thing.
So, information theoretic interpretation:
$\mathbb{E}_{x\sim D}\left[\mathbb{E}_{z\sim Q}\left[\log P(x \mid z)\right] - \mathrm{KL}\left(Q(z \mid x, \phi) \,\|\, P(z)\right)\right]$
First term: Number of bits required to reconstruct x from z under the ideal encoding (i.e. $Q(z|x)$ is generally suboptimal). Second term: Number of bits required to convert an uninformative sample from P(z) into an informative sample from $Q(z|x)$. **Information gain**: Amount of extra info that we get about X when z comes from $Q(z|x)$ instead of from P(z). This whole thing can be trained by backprop, purely like MLP or CNN, simple NN, once you trained them, and you train them by using the encoder and the decoder, cause you need the encoder to generate the mu_x and the $sigma_x$ for the KL div, once you have done training you take the encoder and you throw it away. Cause the job here is to generate new data, how do I generate new data? I don't need the encoder to do that, I just need the decoder, sampling a z from a normal, I have a z, I throw it through the decoder that will give me a new data, because decode z into a sample x that looks as much as possible alike real data. Sampling the VAE (a.k.a. testing): At test time detach the encoder, sample a random encoding (from a normal gaussian) and generate the sample as the corresponding reconstruction
Contractive autoencoder: input x got perturbed infinitesimally but the constraint on the Frobenius norm of the Jacobian of the encoder makes sure that all the infinitesimal perturbed x are gonna be ending up into the same point in the latent space, original sample $\rightarrow$ you generate a space of infinitesimal perturbed point around the orignal sample, than by contractivity, map into the same point in the latent spaze z, what a VAE DOES: the same thing but not with infinitesimal variation, around an x there is a distribution, and you sample things from that distribution, everything that is in that distr. gets project into another distribution in the z space, very simple a local gaussian, every thing that is in a locality of my x gets projected into a distribution here and you decode everything this in this distribution back into z, really nice way of getting a coverage of your space, a lot of gaussians that overlap and span your space that map to to a lot of gaussian that overlap and span your space, nicer/smoother approximation, not sample based, it is based on placing tiny little patches to create a little coverage of your space, generalize much much better than first one, you have no coverage there, if you don't have data with VAE you still have the coverage of a functional that tells you that things are gonna behave reasonably smoothly there. Latent space will encode in nearby and contiguous area of the latent space, encoding of picture of the same number in the MNIST digit, or encoding of picture of people that looks in the same direction in the face dataset. There are area of the latent space that controls the color of the

hair, others areas that control the color of the eye, areas that control the gender, the presence or absence of glasses. Take one sample that is a male person with short hair and glasses, it will be mapped into the corresponding area, then take a male with long hair and no glasses you can create a distance vector between those two points and starts interpolating, and for each point along this direction, you take and you decode, you take and you decode it, your original person when moving along that direction will start loosing glasses and growing hair, **the space is organized**.
If you control this, *here we are sampling in a complete uncontrolled way*, I might want to be sampling people with a specific gender, or people with a specific hair color or not, this can be obtained by making sure that when I'm training my encoder and decoder, aside from supplying the picture I supply a vector of binary value, that tells long/short hair, glass on/off, this way the encoding z will be informed by this specific symbolic information, when you generate you are gonna be informing your decoder with these infos, z which is sampled will account for the general identity of the person, and by changing the values of y, the person of that identity will acquire glasses, will lose glasses, will acquire long hair ... $\rightarrow$ Conditional Generation.
Conditional generation (CVAE): want to generate conditionally some info, not completely freely, uncontrolled. Prepare some y e.g. a vector, with info you want to condition on, that at training time you provide to your encoder and your decoder. This way you are learning the conditional distribution $P(x|y)$. At inference time you provide y to your decoder to generate conditionally.

22 Implicit models - Adversarial Learning

Implicit models are a different family of approaches rather the other we have seen until know, that learns explicit probabilistic distribution, here we talk about **how can we learn sampling processes**, good excuse to introduce adversarial learning approaches. So far we attacked the problem of learning a distribution in such a way we can generate new data, that's our ultimate goal for the time being, **explicit approach we learn the density with the model**, *density that can be sampled reasonably easy*, in the VAE we were doing so by splitting the model density in two component, the encoder and the decoder, having a simple way to sample some noise in latent space from a gaussian, then relocated with a certain mean and variance, for the purpose of training and then decode it back with a deterministic decoder, that was our sampling process and the decoding of this randomly sampled gaussian noise was the generation of the realistic sample.
Implicit approach rather than making sure that the model learns a density, that somehow approximate the data distribution, **we will learn directly a way to sample, no density involved here, just learning a process to sample realistic data**, we are *gonna be generating samples that have a distribution that is somewhat overlapping to the true data distribution. But no explicit modeling of the distribution.* In the taxonomy of Generative DL, it falls out the Implicit direct approaches, *that takes the stochastic part of the model, the sampling, out*

of the chain of differentiation, in such a way that the model remains fully differentiable. To summarize what we have done with VAE, we were learning an approximation of the generating distribution, following the usual factorization of the joint distribution into 2 distribution, while you introduce latent variables, the z. With generative adversarial network there won't be any of such probabilities involved, there will be sampling from a simple distribution again, but nothing else, the leading model in this family is **GAN (Generative Adversarial Network)**, Game theoretic inspired approach, no refined game theory here, but there are 2 players in this game, in this NN, there is a **generator** and a **discriminator** that plays a zero sum game essentially. The point is that if we look at the original data, sampling from the original data distribution is literally almost impossible task, because imagine that we want to obtain is potentially model that can sample images, videos, molecules,... if we sample from the original data, even if I have the original distribution, sampling from it would be a very very difficult task, finding a sampling process to that.
The catch it's always the same, reduce/manipulate our problem, in such a way that the sampling is always performed from simpler distribution from which we know how to sample from: Gaussian, Uniform or some simple Multinomial. In particular we are gonna be sample from a gaussian, because it is simple, we know how to do that, we have reliable tools to do that. And then since we exercise in simplicity in the first point, we exercise in complexity in the second point, we put a lot of the complexity of the model into how we transform random noise into an actual realistic looking sample. We do that through a NN or a seq./composition of NN whose job is literally to transform noise into samples from the training distribution. How to do this into a scalable and sort of easily learnable way? Very challenging task of transforming noise into realistic data.
General overview: 2 players game, **2 modules in the network**, no surprising, also VAE has 2 models encoder and decoder, that works in a synergic way, trying to work together to try to minimize the same thing, since GAN is a game theoretic approach **the 2 modules will be opponent**, two games or better same game but with **different function to optimize**, contrasting games, the game of one is the loss of the other. Non cooperative games. The first element of this model is a **generator network**, whatever articulated form of NN you come up with, but the job of this network is the actual job you would like to realize, a network whose job is to transform random noise, z is sampled from your easily sampleable distribution, a gaussian, in the VAE z is the stochastic encoding of the input info, the original image, that enters the encoder, gets transformed into a vector which is perturbed by noise, stochastically perturbed encoding of the original image, latent space representation of the original image, here in GAN z is a latent space representation of the image that is to be generated, actually no explicit latent space, **z is just a sample from a gaussian**, that spans in a R^d space, this space will acquire a semantic meaning just alike a latent space, but not because you shape it into being semantically organized using for instance the KL trick in the variational autoencoder, here it acquires some sort of sem. organization implicitly somehow, because **the generator learns to map certain portion of the space into a certain type of image**, it's all the job of the

generator you are not structuring the space. You do have your random noise, whatever sort of NN which can takes in input a vector and output an image, for the time being is a black box, **Generator G, G(Z) is the image/sample corresponding to decoding of the random noise**. The thing that make it work, they are not going to be only those images that play, **they are going to be images from a dataset**. *At some point I will have the availability of 2 sets of images: the set of images generated by the generator, based on the sampled z, which will generate a fake images dataset, not real set of images, than we have the real dataset, set of true images/realistic sample.* The second player is called the **discriminator network**, a network that can receive in input in this case an image, **any NN that receives in input an image and classify**, let's take a CNN, **given an image in input tells me if it is original or fake, binary classification problem**. Pipeline: random noise being generated, transformed into an image that is fed to a discriminator network that generates a prediction, generate an error there that you can back-propagate
You can backpropagate from the original or fake prediction down to the generator network, because there is no non differentiable operation evident, the only one is the sampling of z, but that operates over a non-parametrized distribution is normal gaussian, with 0 mean and unitary variance. Only there is the branch in which the input to the discriminator network sometimes comes from the fake, some times come from the real, so that's how it is going to be play the game, so 2 agents: generator whose job is to generate images that look so close to the original images that the discriminator will fail in telling you whether they are fake or real, this is the **contrasting game**, job of the discriminator is being as good as possible, at telling you which images are original and which are fake, the job of the generator is generate images that are so realistically looking that the discriminator fails whenever fake images are inputed to classify them as fake.
Convert this wordy description in a loss function that conveys exactly this game between the 2 net, to train the model. **ALTERNATE OPTIMIZATION**

$$\mathcal{C} = \min_{\theta_G} \max_{\theta_D} \mathbb{E}_{x \sim p_{\text{data}}} \left[\log D_{\theta_D}(x)\right] + \mathbb{E}_{z \sim p_z} \left[\log \left(1 - D_{\theta_D}(G_{\theta_G}(z))\right)\right]$$

Loss that at least originally was used to train the generative autoencoder, and translates to math the intuition discussed before, the generator trying to generate images to fool the discriminator, the discriminator trying to discriminate whether the images in input are fake or true. There is a tension between the generator trying to gain something and gaining something making the discriminator lose. The generator needs to play one game and the discriminator another game. Which turns out to be that min and that max that you have in that loss, differently from the loss that you are used to so far, we don't have a maximization of a likelihood, or a minimization of a loss function, **we have one elem of the net that tries to minimize my loss and the other elem that tries to maximize my loss**, it's a game and θ_g and θ_D are clearly the params of the generator and the discriminator respectively. **The discriminator is trying to maximize that loss function**, **there are 2 expectations one**

over X another over Z, *those 2 expectations rather being integral they will be average over minibatches, over your training data.* Expectation over X that is drawn from the real data, from D the dataset. In principle taken from the data generating distribution P_D, the true distribution, you have the empirical approximation given by data from your training set. For data that comes from the TR set, I have there a manipulation of the usual cross entropy, it's a binary cross-entropy. The only thing I'm predicting at the end of all this thing is fake or original. 0 or 1. D_{θ_D} is the discriminator, if you are the discriminator what you want to predict in the first term? If you are the discriminator, x is an images from the TR set, you want to predict true (1), the log(1) is 0, if you are the discriminator you want the first term to be as close as possible to 0, and *the second part is the expectation over z that comes from the gaussian N(0,1)*, z sampled, goes into the generator and returns an image, the image gets into the discriminator, if I am the discriminator I want to output 0 here, log(1-0) = 0. That's why the objective of the discriminator is to maximize this equation, **this loss is at most 0**, it goes with the log between 0 and 1, it's -inf to 0. If you maximize this part of the log the best you can obtain is 0, the job of the discriminator is to maximize this function because if you do so you will behave as a good discriminator saying fake to the fake images and good to the original images. From the perspective of the generator instead: the first term does not play any effect, *the generator does not have any control over the first term*, the only thing that matters is the second term, if you are the generator you want from the second term that the discriminator gets wrong, if you want that from a sampled z, transformed into an image by the generator, the discriminator predicts 1, true, than log (1 - 1) $\rightarrow$ -inf. Pushing this whole thing to its min. You are gonna be trying to minimize this overall equation, it's a game between two agents, each with its different set of params, working on this overall loss that can be controlled only partially by the generator. Just a matter of changing those expectations that are expectation running over actual density function with empirical expectation that you can run over minibatches.
This turn out to be 2 different functions to be optimized, so whenever you are actually training this model, when updating the params of the discriminator you are maximizing the function C_d, when you are updating the params of the generator you are gonna be minimizing the loss function in C_g. How? **Alternatively**.

1. Discriminator Gradient Ascent Optimize the discriminator parameters θ_D by maximizing the following objective:

$$C_D = \max_{\theta_D} \mathbb{E}_{x \sim p_{\text{data}}} \left[\log D_{\theta_D}(x)\right] + \mathbb{E}_{z \sim p_z} \left[\log \left(1 - D_{\theta_D}(G_{\theta_G}(z))\right)\right]$$

2. Generator Gradient Descent Optimize the generator parameters θ_G by minimizing the following objective:

$$C_G = \min_{\theta_G} \mathbb{E}_{z \sim p_z} \left[\log \left(1 - D_{\theta_D}(G_{\theta_G}(z))\right)\right]$$

Note: This generator loss leads to vanishing gradients when $D_{\theta_D}(G_{\theta_G}(z))$ is close to 0. In practice, we often use the non-saturating heuristic loss:

$$C_G = \min_{\theta_G} \mathbb{E}_{z\sim p_z} \left[-\log\left(D_{\theta_D}(G_{\theta_G}(z))\right)\right]$$

The Issue and a Solution: the cost that the Generator receives in response to generate G(z) depends only on the Discriminator response, flat gradient when sample is plainly fake. Use the non-saturating heuristic loss, that is maximize the likelihood of the discriminator being wrong.

$$C_G = \max_{\theta_G} \mathbb{E}_{z\sim p_z} \left[\log\left(D_{\theta_D}(G_{\theta_G}(z))\right)\right]$$

It's a classical example of alternate maximization, we have seen other case, **EM alg. another case in which you first update the params of the variational distribution Q and then you update params of the model**, here in step 1 you update the param of discr. while in step 2 you update param of the generator. Then how this is done, especially at the beginning when they start to introducing the GANs, and that were very tricky to train, there was a subtle balance between updating the discriminator and the generator, people do 10 rounds of training a discriminator, 1 round of updating the discriminator, related to the fact that is more difficult training a generator than a discriminator, and if the discriminator gets too good too quickly, the generator can't learn anything, everything that the generator sees in his loss function depends on how good the discriminator is at doing is job, **if the discriminator it's extremely good in doing is job, there is no way in which the generator will get any gradient out of this loss**. Because that loss will be completely flat. That is why you need to train them together.
Optimize this loss does not really work, if you use a minimax loss like this one, in the area in which the generator should operate there is a flat gradient, the generator has really hard times in learning. They first come up with a non-saturating heuristic that actually gives you the slope that you need to train the discriminator, this heuristic amount to having the generator solve a maximization problem, instead of minimizing the ability of the generator, is maximizing the confusion of the discriminator, which change slightly the optimization problem, getting to a nice little slope when you need to update your generator, but still complex problem, because when the discriminator is very good no change you can perform on the parameter of the generator that can really alter the loss, gradient descent adaptation does not really work very nice in this kind of problem.
Those expectation in the loss formulations eventually they just become sums, empirical expectation, **stability trick: you update for instance the discriminator multiple times before updating the generator because otherwise you get crazy gradients**, the update of the discriminator which is based on the expectation over data becomes a sum over M samples taken from a minibatch. Same thing from the generator, an expectation over z drawn from the gaussian is the sum over m samples of Z drawing from the Gaussian, you are

gonna be updating your discriminator with the average gradient over the TR data and you are gonna be updating your generator with the average gradient over your fake samples.
People tries all the ways, to obtain smoother gradient for the generator, you first get sort of the discriminator out of speed decently, so you don't obtain crazy gradient because of the fact the discriminator cannot do its job, but that happens that in later stages becomes a problem cause discriminator is too good and generator is too bad. **At the beginning you do a K loops on the discriminator**, after you do the same on the generator, what happens you were essentially training your model and at a certain point your loss die and gets to 0, people tries to resuscitate the loss, backward before the death of the loss and randomization to avoid to die again.
A Hard Two-Player Game: the optimal solution of the min-max problem is a saddle point. Little stability. Lot of heuristic work. One of the problem is given from the fact that we are dealing with a saddle point problem, the solution to that min-max game is a saddle point it really takes very little to fall on each side of the saddle, on the minima or on the maxima, very unstable, there is a solution that solves the problem from a radically more principled perspective. This loss doesn't work because if you analyze this eq., if you literally unfold all the details of the probability distribution involved into this theoretical eq., run all the expectations, rearrange the terms what happens is that you get two terms, the first that penalizes when the generator generates samples which have no support in the data, so from a probabilistic perspective it means that when the generator is putting probability mass on the learned probability on areas where in the original data distribution there is no support. Said in other ways the first term of the reworked loss favors the fact that your generator is generating realistically looking data, the data that is generated by the generator must lies in area of the space where real data lives. It needs to come from high probability points in the original data distribution. The data generator is generating needs to be of high quality, low reconstruction error if you look from an autoencoder perspective, the second term instead favors coverage, you're behaving very badly if the distribution that you are learning does not overlap to the original distribution, the distribution learned by the generator needs to be in a good overlap with the distribution from the data. The point is that while the first term is parametrized by the generator's parameters the second term is not parametrized by the generator params, it means that **the generator by the loss is pushed to generate realistically looking data but it is not pushed to make the learned distribution overlap entirely with the original distribution**, there is no contribution from the coverage term, **the mode collapse or mode failure** because *the generator is only pushed to generate high quality data, it isn't pushed to generate all the data*, it just need to generate one image that is realistically looking and feed it to the discriminator that would satisfy the first term high quality, but not the second term, does not matter, his parameter are not adjusted based on the second term, the coverage term, everything fails because the model is only pushed to generate some realistically looking data, not to generate samples that are consistent overall with

the all the distribution of the data, in all the point, **no coverage enforced**. The key point to find the actual solution to this problem is to **reformulate the min-max problem as a density matching problem**, making sure that the loss function that is used to train this model is a **loss function that pushes the 2 distribution (the generated and the original) to overlap each other**, at least from the perspective of the generator, then discriminator job remains that of telling you fake from falls.
Wasserstein GAN's objective Attempts to solve the hardness of training adversarial generators by optimizing the Wasserstein distance (EMD) between the generator and empirical distribution filtered trough the discriminator function D

Generator Objective

Generator Objective

$$G^* = \arg\min_G W(\mu, \mu_G)$$

Wasserstein Distance

$$W(\mu, \mu_G) = \sup_{\|D\|_L \leq 1} \mathbb{E}_{x\sim\mu}[D(x)] - \mathbb{E}_{x\sim\mu_G}[D(x)]$$

$\|D\|_L \leq 1 \rightarrow$ requires optimizing D under a constraint Lipschitz seminorm: clipping D weights (slow to converge) loss: **Wasserstein distance is defined over distribution**, also called the Earth Mover distance (EMD) amount of hearth, soil, that you need to move to fill a hole. How much work, like in physical terms, do I need to transform one distribution, the generator distribution into another distribution, the data distribution. The Wasserstein distance measures this, **the amount of work you need to be doing for taking the probability mass, if you think about the probability mass/denisty function for the generator and the other one is the actual true data distribution, the amount of work you need to do to pick pieces of soil from one and put back to another, to transform one into the other**. Formulation in the continuous domain, that is a min max problem only rather than having a maximization we have a supremum (estremo superiore), because we have a constrained optimization problem, **the generator is trying to minimize the distance between its distribution and the distribution from the data**, without ever having explicitly written the distribution because we are not learning the distribution here explicitly, so there is nowhere there the distribution written, *only expectation w.r.t. to the output of the discriminator, this loss is in practice written again only in terms of expectations over the outputs of the discriminator*, **the generator is pushing expectations of the discriminator when run on data close to expectation of the discriminator when run on generated samples**, (the second expectation is $\mathbb{E}_z\left[D(G(z))\right]$), *it pushes this 2 expectation to overlap, to be close, instead the discriminator*

wants to keep this apart. There is a constraint placed on discriminator, Lipschitz Seminorm, your discriminator cannot be an all powerful discriminator it needs to be a discriminator with limits, whose weights live on a compact, so they are bound, this comes from the formulation of the problem **constrained optimization**, that requires the gradient of the function you are being optimizing to have bounded norm, the norm needs to be bounded under 1. Lipschitz constant 1. In practice means that the weight of my discriminator needs to be kept on a compact, the simplest way to do that is to clip your weights, to make sure that the weights of the discriminator never exceeds a certain threshold in absolute value, **clip your weight (of the descriminator) to be between [-0.5,0.5]**, so you clip the weight in such a way the remain in this compact. It's a minimax loss with some constraints on the weights of the discriminator and the expectations that you get are empirical averages. The fact that *the discriminator is limited, also somehow in a sense is not allowed to exaggerate with the params, it cannot easily get more powerful than the generator, you want a powerful enough discriminator, but not an oracular discriminator.* The GAN of today all use the Wasserstein Loss. This thing here also pushes the generator to generate things consistent with the actual data distribution where data exists. Up to a fact that y You're matching in expectation, so there is an averaging effect. If you focus on a single mode/peak in your data, you're never gonna be able to approximate appropriately the expectation unless that mode precisely coincide with the expected value that is a very singular configuration. **With WGAN loss, what you obtain in space you always have a gradient mostly piecewise linear behavior in your WGAN loss, wherever you place in the space of the params of the generator you will always have a gradient**, this loss is telling you need to do work to shift ground from the distribution on the right to the distribution on the left and you have a gradient for that, the red one is what you get from the GAN discriminator, large portions where you get 0 gradient.
So effect of the Wasserstein Loss: Classical GAN loss results in saturation (discriminator with zero loss). WGAN provide gradients across all the range of training conditions.
Now we have a loss function that works for generative autoencoders, how do I model the rest of the net? need a NN for the generator and a NN for the discriminator, let's say we are talking about generating images, the discriminator needs to be a net that takes in input an image and output a binary classification, any CNN, visual transformer, what about generator, we need a NN that receives in input a vector z and outputs an image: Upconvolution layers/Deconvolution/Transposed convolution, a NN that at first step takes a vector and outputs a small image, progressing in the net an original small images with a lot of channels get transformed in a larger image with fewer channel, until you get at the end with a very large image with 3 channel RGB, exactly the definition of inverting a convolution: transposed convolution: usual convolution only with jumps, upconvolutions are just convolutions with parametrized filters trainable with backpropagation, implementing the generator is just using a deconvolutional NN, implementing the discriminator is just using a classical

CNN.
DCGAN: first reasonable quality model that popularizes the use of Generative Adversarial Network for image generation, this model has a lot of success for image generation, at the beginning images were ugly generated, but they started do things, done also with VAE, let me sample a z1 and decode into an image, a random vector z2 and decode to an image same for a vector z3, then I can do some arithmetic with this "latent vector" $z_1 - z_2 + z_3 = z_G$ and then decode zG, in principle since I am operating on a metric space in Z_G I should obtain visual feature of these 3 samples combined. You can add alpha values, interpolation values, e.g between z1 and z2 and z3 you can obtain different images when you decode. When you subtract z1 male with sunglasses - z2 male without sunglasses, you remove the shared component which is the gender. You are left with all the gender related features removed and the sunglasses stays. If you sum the result to z3 a woman without sunglasses you obtain a woman with sunglasses. $\alpha(z_1 - z_2) + (1 - \alpha)z_3$ you can get people with less or more sunglasses in the picture.
There is no explicit structuring of the z spaces, which result from a sampling of the gaussian, but the structure is implicit because of the generator, which has learned to interpret the different area of the original space giving it a semantic. Low quality images $\rightarrow$ to high resolution images. You can't do it in one shot, **to have high resolution large dim images you need to do it incrementally**. **Progressive GANs**: if you want to generate a realistically looking a 1024x1024 image you need start training a GAN to generate a 4 by 4 image, which means that your DCGAN, the deconvolutional network, will have only the first part, the 4 by 4 with a lot of channel. Fine you train it. And of course you also train the discriminator, the discriminator matches the complexity of the generator, at the beginning you have a very simple discriminator that takes in input a 4by4 and decides false or not with only 4by4 pixels. Once you have done it, you move one step ahead , and you do it with a 8by8, you increase a little bit your DeConvNN and your CNN with **the original weights inherited from the previous iteration**, and **when you train you also adjust, they are not freezed, but they start from a pretrained situation** in which they know how to do their job on a 4by4, you repeat this process until you get to a point in which you generate a 1024x1024. Actually you do it with a progressive fading of the early trained models in favors of the new models. Let's imagine we start by generating a 16 by 16, you train your model to generate a 16by16 and the same you do for the discriminator, when you move on to generate a 32by32, where you're generating the 32by32, **when you are training the model, what you're actually doing is that the image that you're sending into the discriminator, is not generated directly by 32by32 but rather is generating by interpolating this 16 by 16 image generated by the previoulsy trained model, that pass through this bypass connection here, residual connectivity with 32by32 image generated by the new block**, $(1 - \alpha)$ of the image that comes from the 16by16 plus α of the image that comes from the 32by32. You start with α being very small at the beginning of training in such a way that the image that you are sending

to the discriminator, is mostly due to the 16by16 image of course you upsample this image to a 32by32, the image that you send to the discriminator depends heavily on the previously trained 16by16 layers, as you go on the training, in epochs, when you start to rely more on the 32by32 you start pushing α to 1 in such a way that the image you send to the discriminator depends mostly to the 32by32. Slightly smoother way to move from model (a) to (b) see slide otherwise you have peak in gradient, strong discontinuity that does not help training.
Conditional Generation: learn a mapping from an observed side information x and a random noise vector z to the fooling samples y. $G : \{x, z\} \rightarrow y$ **Conditional generation**: rather than generating images that are only subject to randomization, you might want to control generation of images, for instance you want to control the age group of people you are generating, this age group becomes an **additional conditioning information that I am supplying to my generator**, x is an encoding of the age group, whenever you are training your model **you train based on z that captures the generic content of the image, the appearance, the look of the face, the color of the eye, every possible aspect but the thing modeled by x, because it is supplied in input**, x models the age, for instance you can draw a z, a single random vector, with will denote the identity of the person, whatever it is, you supply the specific x, age group, and you decode and obtain the first decode on the first column (see slide 20), then if you keep the z fixed and you change x, you get the same person, because z that captures everything but age is fixed, and by changing x you make the person grow older.
The same thing can be pushed into things where condition information x is more articulated than being gender information, input can be semantic segmentation of a scene, and you want the model to generate a realistically looking image consistent with the semantic segmentation, or the input could be a sketch of a bag and you want the model to generate the photorealistic image of the bag, the structure of the bag in input will be complemented by the vector z which is defining all the aspects that aren't captured from the original sketch, so the color, z will capture the color, the material, the lighting condition. Whenever you are clamping you are actually inserting some conditioning information that will make sure that the model when trained will interpret everything that comes from the stochastic z vector as being all the rest of info it needs to capture except the ones that you are using to condition the generation.
Conditional GAN $\rightarrow$ you supply in input to your generator your latent variable sampled from the gaussian z, the attribute vector like the gender c, and the generator learns to generate an image x_j that is then sent to the discriminator, **together (with the generated image) the discriminator receives in input the attribute vector that can comes directly from the input**, the discriminator is trained receiving in input the image (real or fake) and the attribute information. This require full labeling.
There is another option: works specially when conditioning information is class information, I want to generate a person who is blonde vs dark-hair, rather than giving in input to the discriminator the class information, **giving to the dis-**

criminator only the image and ask the discriminator to tell me if the image is real or not and predict also the class information(the prediction of which class is portrayed in the image), this will inject more info into the params. of the model and also pushes the generator to generate images that are more consistent with the class, because if they are not consistent with the class the discriminator will get the classification wrong. The discriminator, is not only predicting if the image is real or not but it is also trying to guess which class belongs to, just a trick to force the generator to be more accurate with generating sample, because if it does not generate sample that are consistent with the class the discriminator cannot do its job so the loss grows. This is called **Auxiliary classifier GAN**

The other option is the **infoGAN**: *the conditioning information at training time is not inputted, it is sampled from a different distribution*, z_j is from a gaussian and c_j comes from a multinomial with a certain number of dimension, by splitting the input information for the generator into these 2 parts you are trying to push the separation of information inside of the generator, making sure that the generator tries to capture in z_j only aspect that are consistent with a continuous formulation like color, identity, while capturing into c_j of those aspects that instead are controllable by very specific discrete aspect, like gender, age group, this is called **disentangling**, *when you push your model to capture in different dim of the latent space, very specific and isolated aspects.* I don't want gender to be a concept that is distributed in multiple dim. of a continuous vector z_j, I want gender to be very specific, discrete attribute captured by one elem of the discrete vector c_j, disentanglement means this separation of info into diff. dim of the input vector, and at generation time I can keep sampling z_j from a normal and operating on the components of c_j flipping them, changing their value to see how images are generated differently and understand for instance that dim 1 of the vector c_j controls gender, dim 2 controls age group, dim 3 presence of glasses or color of the eyes. For the rest infoGAN looks very similar to the auxiliary classifier GAN because the discriminator is trained not only to output the prob. of the image being real but also to predict c as if it was a classification, always to push the model to learn to structure the way in which they interpret the original space.

Many of this model for conditional generation require data to be aligned, *if I want to generate an image of a person belonging to a certain age group*, I need to have my original images labeled by age group. *You have seen those app that transform every picture you send them into a Monet Painting, those are style transfer, that take in input an image, you are conditioning a GAN with the picture you're sending, and you obtain in output the image decoded by the generator.* I cannot expect to have a dataset to train on painting of Monet with the corresponding original picture, I obtain it with a cycle GAN. **CycleGAN** is made of multiple models, *imagine that I want to generate a very useful GAN that transforms horses into zebras, I don't want to leverage a dataset in which i have painted all the horses like zebras or all the zebras by horses to preserve alignment of the images.*

I start with a picture of an horse, I send in input to the generator, this generator

receives an horse as conditioning info plus a z, that is randomly sampled, and outputs a generic zebra, so the job of this generator is really that of converting between the 2 domains: generate a possible realistic zebra from an horse, how to assure that? I have a discriminator that is trained on the zebras to tell you if it is a zebra or not, by this direction here I get a gradient that pushes **generatorA2B** to generate starting from the horse a realistically looking zebras otherwise I get an error when I send to the discriminator. **There is a consistency loss from the conditioning picture of the horse and the generated zebra that scores how much those 2 elems are similar, like a reconstruction error that tells you how similar they are, when I generate a zebra that fools the discriminator but it is a zebra in a completely different setting than the original picture you get an high loss, so you get also a gradient from that**. Then there is a third part in which the generated zebra gets converted back from a second generator into an horse, trained by another discriminator that tells you if this horse comes from the generator or from original set of horses. **In all these things no direct alignment between training set images**, only between images that are generated, and there are **3 losses: the usual adversarial loss, inconsistency loss between the 2 generated elems of the different domain and a cyclic loss that tells me if the decoded back horse looks alike the original horse**.
CycleGAN - Style transfer without pairing:

CycleGAN Architecture CycleGAN consists of two generators, $G_{A\to B}$ and $G_{B\to A}$, and two discriminators, D_A and D_B. The generator $G_{A\to B}$ maps images from domain A to domain B, while $G_{B\to A}$ performs the reverse. The discriminator D_B distinguishes between real images from domain B and fake images generated by $G_{A\to B}$, and similarly, D_A distinguishes between real images from A and those generated by $G_{B\to A}$. The training involves two types of losses: adversarial loss, which encourages each generator to produce outputs indistinguishable from the target domain, and cycle-consistency loss, which ensures that a sample mapped from one domain to the other and back remains close to the original, i.e., $G_{B\to A}(G_{A\to B}(x)) \approx x$ and vice versa. This allows training without paired data, making CycleGAN effective for unpaired image-to-image translation tasks.
Until now 2 family of models: VAE that auto-encodes original data x, into an explicit latent space Z, and decode it back into the original input data, in which the KL-div is used to ensure that the encoding of the variational autoencoder to be simple enough to be confused with a gaussian, and then the Adversarial approach in which z is taken from a gaussian and then by this game is transformed from an image. Can I obtain the best of the 2: *a model in which I'm auto-encoding things into a latent space, in which I'm pushing properties into the latent space rather than by KL-div by an adversarial loss* → **Adversarial Autoencoder** (AAE). Autoencoder: you start from an image, gets into an encoder, you get a z, and decode it back into your image, usually that z, there

is a component of my loss, which is the KL-div between the q distribution of the encoder, which is a gaussian, and the normal distribution, a gaussian with 0 mean and unitary variance, this is how typically works, here I still have my gaussian with 0 mean and unitary variance that now is the generator of true data, true data are vectors z that comes from the gaussian, and then there is a **discriminator, whose job is to tell me if the input info that I am receiving comes from the encoder q, which is the generator or from the actual data that comes from the gaussian**, so I can remove the KL-div and substitute with a usual adversarial loss based on the discriminator, that is the trick: the effect of the GAN used in place of the KL-div is to obtain the same goal that the encoding z generating by q are indistinguishable from the perspective of the discriminator from z that comes from an actual gaussian with 0 mean and unitary variance that was essentially what you want to obtain with the KL-div. Take the loss of the VAE autoencoder and substitute the KL with the Wasserstein loss, that's it. Training AAE:

Modified Variational Objective

$$\mathcal{L}(x) = \mathbb{E}_Q \left[\log P(x|z)\right] - \underbrace{\mathrm{KL}\left(Q(z|x) \,\|\, P(z)\right)}_{\text{substituted by adversarial loss}}$$

Reconstruction phase - Update the encoder and decoder to minimize reconstruction error. **Regularization phase** - Update discriminator to distinguish true prior samples from generated samples; update generator to fool the discriminator. Adversarial regularization allows to impose priors for which we cannot compute the KL divergence. There are multiple training phase. In the reconstruction phase you update both the encoder and the decoder to minimize the reconstruction loss, cause it is an autoencoder, in the regularization phase you update the discriminator to distinguish the true prior samples from the the generated ones and you update the generator to fool the discriminator, so in this second phase you train the discriminator to become better at telling when things come from the Gaussian and the encoder to become better in sending z to fools the discriminator, the encoder gets trained twice, by the reconstruction loss and by the adversarial loss, in the adversarial phase.
Why do I want to do that if I have the KL div that writes in close form for the Gaussian, because I may want to shape my latent space to be consistent with distribution which are not gaussian and for which I don't know how to compute the KL-div, you can make sure that the autoencoder latent space behave like any distribution you can think of, provide you can sample from it, it doesn't need to be a distribution for which you can write the KL div in closed form . You need sample from the original data, **you can shape your latent space to be consistent with a distribution which is consistent with a distribution that is fully empirical, just data**. Train my z vectors to be consistent with a latent space organized as mixture of gaussian, organized as a flower each petal is gaussian, points that belongs to a same petal may be images of the same class, same for a different petal, but images that looks very

much alike live in the central part. This organization of the latent space allow to represent images very distinctive of one class into a very far away part of the petal, and images that belongs to that class but can get very mixed with other images cause looks alike to other images they are gonna be represented by the encoder in the central part of the flower.
We can train NN also by having them compete one another - adversarial training. For instance in VAE whenever we do have the need to regularize the latent code to make it behave like a normal gaussian, one possible way of doing that is Kl-div in standard VAE, or we can use adversarial learning, sample z, an encoding/latent vector for my encoder should be indistinguishable from a random sample from a gaussian by appropriately training a discriminator in a min-max game in adversarial training.
Remove in the training of the autoencoder by removing the KL regularization and substituting with an adversarial loss. 2 phases, one phase that minimizes the reconstruction error, and a regularization phase in which the encoder and the discriminator plays a min-max game, and their params are adapted. This thing make sense when you want to impose on the latent space a prior other than the gaussian for which you can write in close form the Kl-div, so for instance we can play with mixture of gaussian or more you can make sure that your model is adversarially training by knowing exactly to which sepal it needs to belong to, we can feed this info in the adversarial part. I can have a task classical MNIST with multiple style of digit, 10 different classes, you wanna capture also that the way you write your number can change in style, model with adversarial autoencoder, I know at least at TR time what's the digit, let's make sure that the autoencoding of the model, the original image is autoencoded into a latent code as usual z, but then the decoding is based both on z learned by the autoencoder, as well as on info that I supply externally, that identifies which digit is used with a 1-hot. As a result of this thing we are saying to the model, you don't have to waste space in z to try to encode the identity of the number, no need to do that, because I'm supplying it with a specific 1-hot encoding, so z will focus specifically on capturing everything that is not the identity of the number such as the rotation or the style, used to generate conditionally stuff, set the 1-hot encoding of the digit that I want, sample a random z from my gaussian, and then feed this concatenation to the decoder and obtain a 1 with a style determined by z, if i want to generate a 2 with the same style, keep the z fixed and I flip the 1-hot encoding. AAE - Style Transfer (supervised): Incorporate label information explicitly to force z to capture class-independent information (e.g. style).
You can push the same intuition further, rather than having to rely on the fact that this thing is purely supervised info, I can say that my autoencoder is made in such a way in which I have both this dense embedding for the style and the 1-hot encoding for the digit identity, but this digit identity not available as supervised info but has to be treated as classical embedding in a VAE, it is an embedding generated by the decoder, now I have to tell to the model how this thing needs to behave/to be distributed, need to have a KL-term also for that part, but since adversarial autoencoder can be trained to behave consistently

to samples drawn from a categorical distribution of my choices, a multinomial informed by the data, discriminator to push the generator, so q, to generate y that are consistent with the distribution of the categories in my data. When I'm training the model make sure that 1s for instance are mapped to a specific gaussian of a mixture of gaussian, my latent space flower shaped, all these petals of the flowers are number, each of them is a gaussian, each of them is responsible for generating a specific type of digit and the spread of the point is the style or the rotation aspect. If I want that as a prior of my latent space shape I have to make sure that when I train the model the y needs to behave consistently with that assignment, whatever training example I'll have I'll confront zeros with 1-hot encoding of the identifier of the 0 component of my mixture of gaussian, whenever I see 1 in input I need to be sure that the encoding y needs to be consistent with sample drawn from the identifier 1 of the gaussian. Having fixed from which gaussian the things come I can make sure that the z part instead is consistent with a gaussian because having fixed one gaussian the z then are gaussian distributed. At training time I can use this supervised info, knowing the fact that the number is 0 to assign to a specific gaussian, at test time I can sample it from my distribution and decode accordingly. Semi-supervised info because it can use it a TR time, not directly as the info fed into the model, but rather pushing the encoder to guess which is the identity of the digit, the way to do it is by shaping the expected distribution of your latent space accordingly. There is a lot of tricks on GAN including the fact that your are gonna throw away a lot of samples when you generate, generating new data with a GAN you are substantially decoding a random vector of noise, that comes from a gaussian with 0 mean and unitary variance, it comes with the non negligible chance that you can sample some vector that comes from the outskirt of the gaussian, that comes from the external part and on the external part those are low probability vectors on which probably the generator is not well trained → low quality stuff. People tend to reject low probability sample. AAE – Semi-supervised learning: factorize latent code in: one hot encoding vector y, continuous code z. Distribution of y made little distinguishable from a multinomial (induced from data).

23 Diffusion Models

Back to the world of **explicit models**, models that in a way or another expect to do *Maximum likelihood learning, approximation of the likelihood.* Diffusion model is essentially **a model that uses 2 process, one that adds noise and one that removes noise**. They have an interpretation as VAE, a very specific VAE, where latent space has the same size as input space. **Forward and reverse process**. **Sequential processes** but are **bad in terms of computational complexity**. How can we obtain the effect of a sequential process using non sequential operation? How do we met this process of *incrementally adding noise* and *incrementally removing something* that we can scalably training. Get to Dall-E like stuff.

Diffusion model used also for conditional generation, conditioned e.g. based on text, *text can be easily embedded into a vector using a language model and used as a sort of info into a model whose job is to generate image.*
Diffusion Models falls into the category of **explicit** GDL models, because they **maximize the likelihood**, but doing so using **untractable densities** because we postulate that we use **latent variables**, instead of using latent variables which are vectors, **we use latent variables that are modification of the original input images**, *latent embedding is just the original input transformed, same size.* When we have intractable densities we need to approximate, the chosen way to approximate things in this model is variational approximation→ **ELBO**.
General scheme of a diffusion model: **2 processes** in a diffusion model: **forward process** and **reverse process**, the first has the role that the encoder has in a VAE, *starts with the data and ends up in an encoded version of the data, the reverse process is sort to akin to the decoder, start with the latent rep. and gets back in the original image*, the process of encoding is particularly strange because it works by adding noise, **the forward process iteratively, incrementally adds noise to my image**. Input X, Z_1 equivalent of latent encoding at layer 1, Z_2 latent encoding at layer 2, ..., Z_N, **N step of encoding of the original image**, this encoding is destroying the original image by addition of noise, incrementally, the decoder, reverse that thing, from Z_N reconstruct Z_1 and from Z_1 reconstruct x, the reverse process is a denoising process
I have a process that from data gets to full noise, **characterize noise as gaussian with 0 mean and unitary variance**, so I get withe noise that comes from a gaussian. If I have done my job well, this thing has learned to take a sample from a Gaussian, withe noise, and transform it back into data. I used a process to destroy an image, to transform into noise, noise characterized as drawn from a bidimensional gaussian, in such a way that I can train, on known data, the reverse model to invert a sample from noise into the corresponding image, and I have this ground truth cause I'm generating it with the forward process. The intuition is that given an image from my TR set I generate the corresponding target, noisy version, by incremental addition of noise, and then I use this info to revert the thing, to train a learning model that associates a specific vector of noise to a reconstructed data, if I have done this job well I can throw the encoder at inference time, just use the decoder, sample some bidi noise, feed to the decoder, and generate an image.
This is in stage/step, how to make this thing, that is potentially very sequential/synch. in the more efficient way possible??
Forward process, *we call it encoder but no NN actually involved*, and infact it does not need to be, the job of forw. process, is adding a little bit of noise at each step, can **use a standard noise model**.

$$z_1 = \sqrt{1-\beta_1}\, x + \sqrt{\beta_1}\, \epsilon_1, \quad \epsilon_1 \sim \mathcal{N}(0,1)$$
$$z_t = \sqrt{1-\beta_t}\, z_{t-1} + \sqrt{\beta_t}\, \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0,1)$$
$$\text{for } t = 2, 3, \ldots, T$$

$\beta_t \in [0,1]$ is the noise schedule. This transformation, starting from x, to z_1, to z_2, until z_T and at each step I need to add a little bit of noise according to a predefined noise model. This is hinting at the fact that there is a transformation of x into a sequence of latent variables, underlying this process here there is a very well known kind of process, in which z_t depends on z_{t-1}. It's a **Markov chain** defined by this **transition distribution** $q(z_t|z_{t-1})$, how I obtain the next image starting from the previous one, under the condition that z_0 = x, a Markov process with an initial state z_0 and then a transition distribution so forward diffusion process/noise model $q(z_t|z_{t-1})$ is telling me how I obtain z_t, the next image, starting from the current one, z_{t-1}, by adding a little bit of noise to z_{t-1}, noise model, at the beginning I want to obtain z_1 from x, how do I obtain it? with a simple-enough to characterize model, needs to be smart in the choice of the noise model,$z_1 = x + \epsilon_1$, take x and add ϵ_1, some noise, with same size as input image and needs to be easy to sample, ϵ is a sample from a normal with 0 mean (image centered in 0) and unitary variance, taking x and add to each pixel an amount of noise that is independently sample from that normal, incredibly harsh if I do that, mitigate the effect of noise addition with the term in the square root, *trade-off between how much I am preserving from the original image x and how much noise I am adding to it.* At the beginning I will have β_1 very close to 0, I can choose β close to 0, z_1 very similar to x with a little bit addition of noise, that I can replicate for all the other step, at time z_t I can have the new noise version at time t, which is mostly all the noise version at time t-1, which still preserves a vague remembrance of the input through all the unrolling of this equation, plus the newly sampled noise, just added a tiny bit cause β very close to 0. This if β is static, but β has a **noise schedule**, β not static but changing with time according to any reasonable schedule that start from 0, stays very close to 0, up to a certain time than it grows to reach 1 at some point, at the beginning of my Markov chain when t near to 0, e.g. when computing z_1, β very close to 0, perturbing very little the image, for the first stages of my diffusion process noise injection in the original image very limited, at the end of the process allowed larger injection of noise. Whatever we are doing now, we need to revert after, **if we are bad and we use the opposite schedule, first destroy a lot and then a little bit, I'm getting rid of much of the info in my image since the beginning, overwriting it completely, and then long long tail of propagation of just noise, no much sense**, whereas it makes sense start destroying the very high frequencies information in the image, the first aspect to be destroyed in a signal are the high freq. info when you add noise, you want to destroy very slowly the details at the beginning, then start hammering your image with a lot of noise, low freq. needs a lot of noise to destroy, that's way radio in low freq. works nicely.

So to resume, I get a **very simple additive noise model**, *at each step I am adding a little bit of gaussian noise and the amount of noise I am injecting depends on this time dependent schedule*, the fact that I am using additive noise model and that noise comes from a normal gaussian is the key trick to allow me to say what is the nature of this distribution q, cause if ϵ_1 is gaussian than z_1 will be gaussian, and z_2, ... my transition function is gaussian $q(z_t|z_{t-1})$, is a gaussian , not 0 mean and 1 stdv cause that is the noise, but it is the noise relocated.

$$q(\boldsymbol{z}_t \mid \boldsymbol{z}_{t-1}) = \mathcal{N}\left(\sqrt{(1-\beta_t)}\boldsymbol{z}_{t-1},\, \beta_t \mathbf{I}\right), \quad \text{where } \boldsymbol{z}_0 = \boldsymbol{x}$$

$$q(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_T \mid \boldsymbol{x}) = q(\boldsymbol{z}_1 \mid \boldsymbol{x}) \prod_{t=2}^{T} q(\boldsymbol{z}_t \mid \boldsymbol{z}_{t-1})$$

Distribution of z_t is a gaussian centered in the mean given by z_{t-1} times the annealing coefficient and with variance that is β_t that comes from the multiplication factor applied to the noise. **Every step is a gaussian that changes slightly with the scheduling**. Now I can apply my factorization according to the Markovian independency, if I want to model the joint distribution of all the latent variable, a sort of posterior, latent variable z given the observable one x, that posterior factorizes easily using the Markovian independency, product of course of the first transformation x $\rightarrow$ z1, and the iterative product of all the state transition until the end, product of all gaussian. **So far so good, but so far so sequential**, to go from x from z_t you have to obtain z_1, then z_2 ,... I want more speed, **diffusion kernel** that speeds up stuff, I can operate substitution z_{t-1}, I can express in terms of z_{t-2}, it is a recursive definition, then I rewrite z_{t-2} in terms of z_{t-3} until I arrive to x, if I unfold all of these in time I will get a very long thing that multiplies a lot of $\sqrt{(1-\beta_s)}$ from t-1 to 1 that in the end multiplies by x, which is good, **if I can pre-compute the product of those square roots, given an x, I can obtain the noise version at each step without the recursive formulation**, that's the definition of **diffusion kernel**.

$$q(\boldsymbol{z}_t \mid \boldsymbol{x}) = \mathcal{N}\left(\sqrt{\alpha_t}\boldsymbol{x},\, (1-\alpha_t)\mathbf{I}\right)$$

$$\alpha_t = \prod_{s=1}^{t}(1-\beta_s)$$

It tells you that if you want to have the distribution of z_t given x, **obtain straightaway the noise version of x at time t**, what you need to do is just sample it from a normal with mean $x * \sqrt{(\alpha_t)}$ and stdv $(1-\alpha_t) * I$ where α_t is that precomputed long product of sqrt, so this **can be precomputed one time for all**, and can be used to **obtain the noisy version of x at every possible step of my forward process without having to precompute the others**, break a fully sequential process, into a **potentially parallel process**, when you do training you want to generate noise versions of x at different

time steps, and you can do that in parallel processing, no synch, you can scale it by using a GPU, we have obtained effectively a way to generate all the noise version you need at every time step, in one-shot, that are consistent with the Markovian process defined before.

$$q(\boldsymbol{z}_t) = \int q(\boldsymbol{z}_t, \boldsymbol{x})\, d\boldsymbol{x} = \int q(\boldsymbol{x})\, q(\boldsymbol{z}_t \mid \boldsymbol{x})\, d\boldsymbol{x}$$

I can also compute the **marginal** of my distribution $q(z_t)$, can easily be computed with the diffusion kernel, I introduce x by marginalization and then apply my chain rule and rewrite the marginal of $q(z_t)$ as the **marginalization over q(x) times the diffusion kernel**. The **forward process** is implementing the **morphing of a distribution**, through the diffusion process, thanks to the diffusion kernel, you obtain a tool that *given an initial probability distribution q(x), data generating distribution, very complex, articulated, I can obtain a new distribution $q(z_t)$, for all the original data by application of this diffusion kernel, I obtain a new distribution $q(z_t)$ that is tending for t that goes to infinity to a gaussian.* Forward process: transforming q(x), in principle by iterative application of noise, you are transforming this very complex multimodal, distribution, into a very simple, very easily samplable gaussian with 0 mean and unitary variance, because through a diffusion process you are taking an original sample and throwing a lot of 0-centered unitary variance noise to it, there is a limit to how much noise you can throw to stuff, at some point it will become a sample from a normal and the diffusion kernel gives you a way to jump immediately to this simpler distribution. **Morphing distribution with no param so far.**, just morphing distribution using a predefined transformation
Markovian process with **normal transition distribution**, can be rewritten explicitly getting rid of the seq. using the diffusion kernel, **jump as we like through time**.
Learning: *inverting this process*, morphing a complex high-dim distribution in a very simple one it's easy, you are destroying info, **take an uninformative distribution and morphing it back into a informative distribution, data generating distribution**. It is not doable without marginalizing out the data. **Unless we make some simplifying assumptions you cannot generate gold from garbage**, the things work only because you are not really starting from $q(z_t)$ pure garbage, but with some spoiler about x, otherwise it will be impossible to regenerate sample from the data generating distribution.
In the denoising process I am gonna be starting with a sample in the simple to sample space of z_t and then **incrementally apply a reversed noise process** until I get back to the original data space. Previously I had $q(z_t|z_{t-1})$, this is the mixing process, in the backward direction I model $q(z_{t-1}|z_t)$ I am inverting exactly the thing over there, by **iterative application of sampling from this demixing/denoising distribution** I am gonna be hitting at some point z_0 which is gonna be x, an actual image. What is the nature of this distribution? Let me write in terms of distribution that I know, apply Bayesian theorem that allows me to flip the 2 things, rewrite by the Bayesian theorem and find a

Gaussian distribution $q(z_t|z_{t-1})$ which I know how to characterize, who are the other 2 guys? the marginal distribution over z_{t-1} or z_t mmmmmm,.... completely untractable, **marginalization overall the data that contains the data generating distribution**.... you can't do much.
The denoising distribution is intractable. $q(\mathbf{z}_{t-1} \mid \mathbf{z}_t) = \frac{q(\mathbf{z}_{t-1})\, q(\mathbf{z}_t|\mathbf{z}_{t-1})}{q(\mathbf{z}_t)}$ Because of $q(z_t)$ that marginalizes out of all q(x). It turns out that we cannot de-mix noise if we don't know the starting point x, if we do, then we can show that $q(z_{t-1}|z_t, x)$ is Normal.
If I had the data generating distribution I am done. Find an approximation to make it approachable, If I know x, if I can condition also on x, $q(z_{t-1}|z_t)$, this thing becomes tractable, because **you lose the marginalization inside of the q, because x is fixed, it is a specific x**, you start trying to reconstruct from noise and knowing where you are gonna be ending up, because you are saying that if you have $q(z_{t-1}|z_t, x)$, you know where you are z_t and where you wanna get at the end of the times x, can you give me the slightly denoised version of z_t of course it is easy in this setting, if I know z_t and I know where I want to arrive, translated in math terms: **this distribution here** $q(z_{t-1}|z_t, x)$ **is our friend gaussian**, but I don't have x when I am decoding, because I am using this distribution to generate x, **I cannot have x at inference time, but I can use x during training**, *knowing that I can use x during training, helps me to find an approximation of this thing that is reasonably gaussian, at inference time I'm gonna be denoising without considering x*, denoising starting from z_t and obtaining z_{t-1}, but knowing that at training I can have it so the distribution that I have learned is gaussian when I know x, so I will model q to be a gaussian, I will training using info from x, so that I am under the condition of the gaussian, **another fact that make it doable is that I can assume** $q(z_{t-1}|z_t)$**, even at inference time when x is not known, if the amount of noise I added between** z_{t-1} **and** z_t **in the noise process is small enough**, β should be changing slowly and I need a sufficient amount of step, if β changes slowly to get from actual data x to fully random noise z_T will require a lot of steps. If I want **to make this thing feasible, I have to leverage the info about x during training, fine, and to make sure that this thing q stays gaussian also at inference time I will be doing my noise process in such a way that there are many steps with a little noise added each time.**
If I wrap up the all of this, the denoising process under all this reasoning become gaussian, the parametrization of the q, there are two q, the $q_f(z_t|z_{t-1})$ that adds noise, and the $q_r(z_{t-1}|z_t)$, this is the demixing/denoising, the first is gaussian from definition of the noise process, the second is gaussian under all the above condition, so I can learn it, the first one is predefined, the second one is learned, how do I learn it? Q_r, reverse or demixing distribution, of the reversed process is **learned by a NN** that receives in input z_t and returns in output a mean and the variance of the gaussian. Model of that q_r distr. amounts to have a NN that receives in input z_t and returns in output the mean and the variance of the gaussian, then, How do i generate a Z_{t-1}? I sample it

from a gaussian with that mean and that variance. My model on the reverse part, P_θ is the learning model, the first one is distribution from which I sample z_T the noise, but z_T is the normal gaussian, z_T is sampled from the gaussian. Sample my z_T from the normal and whenever I have to sample a generic Z_{t-1} I sample from this normal parametrized by my NN, **my NN output is the mean of the gaussian from which I am gonna be drawing** Z_{t-1}, how do I generate? by reading in input z_t and t. Unless I want to train a different model for each of the different times, I need to tell to the NN from which time I am decoding. Decoding step is transforming a z_t from a time instant t in z_{t-1} and t can change, I am modeling all these jumps: I can have multiple NN for each of this time step, they can be separate and parametrized differently , a nightmare, number of param proportional to the number of step and we know that they must be a lot, not the best idea. If you cannot use different param for different time step use a single NN to implement all of those transformation, weight sharing, but that creates issues, my β scheduling start from 0 and grows with time, the amount of noise, that I add to the image change with time, when I subtract the amount of noise to substract to extract a z_{t-1} from z_t changes with time, need to communicate time to the NN, an encoding of time is needed in order to help the poor NN to understand how much noise to extract.

$$P_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t) = \mathcal{N}\big(\boldsymbol{\mu}_\theta(\boldsymbol{z}_t, t), \sigma_t^2 \mathbf{I}\big)$$

My backward process from a learning perspective becomes a single NN, replicated at each time step, that at every time step, receives the previously noisy image and return the mean of the slightly less noise image, then I sample from a normal, that normal, to get that image. At least in this simple model the variance is not predicted by the NN, σ_t^2 is just a schedule for the variance that can match the schedule fixed for β in the forward process.
Can I learn also sigma through a NN? yes, but for simplicity we don't do that in this first case.
So at some point I want a z_{t-1} how do I obtain it? I use my NN to generate μ using in input z_t and t and I obtain the mean of the normal $\mu(z_t, t)$, now I take exactly the normal with its variance $N(\mu(z_t, t), stdv)$ and I sample it from there, or better I sample an ϵ, from a N(0,1) and relocated to have that mean and variance.
The all job of the **denoising process**, is to **train this NN that return means of gaussian**. Training follows the classical log-likelihood maximization view:

$$\log P_\theta(\boldsymbol{x}) = \log \int P_\theta(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_T, \boldsymbol{x})\, d\boldsymbol{z}_1 \ldots d\boldsymbol{z}_T$$

$$= \log \int P_\theta(\boldsymbol{x} \mid \boldsymbol{z}_1) \left(\prod_{t=2}^{T} P_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t) \right) P_\theta(\boldsymbol{z}_T)\, d\boldsymbol{z}_1 \ldots d\boldsymbol{z}_T$$

...which of course is intractable.
I train it by writing the likelihood as it is explicit model, **the log likelihood is the log probability of my data**, but I don't know this distribution $P_\theta(x)$

cause it is a latent variable model, let me introduce all the latent variables, all the Z_t until Z_T, **beautiful marginalization over all the continuous latent variables**, I introduce my Z_t and I get my joint completed distribution, on this distribution here I apply the markovian condition and factorize according to the chain rule and the simplification induced by the markovian condition so this can be rewritten as just a multiplication of distributions, I have applied the Markov chain in the inverted direction, because I want all the distributions that are parametrized here, no the q ones that aren't parametrized, otherwise there is little to learn. $P_\theta(z_T)$ is not parametrized is a normal gaussian, but inside the other 2 terms I will have my NN whose job is to predict the mean, this are all gaussian $P(x|z_1)$ and $P(z_{t-1}|z_t)$ are all Gaussians by the defs given before, this is a **logarithm of an integral of a product of gaussian**, I can characterize the involved distributions, **problem is a log of integral of a product not tractable analytically**. How do I optimize this thing??
Introduce the encoder distribution q Let $\bar{\boldsymbol{z}} = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_T)$. Then:

$$\log P_\theta(\boldsymbol{x}) = \log \int P_\theta(\bar{\boldsymbol{z}}, \boldsymbol{x})\, d\bar{\boldsymbol{z}}$$

$$\geq \int q(\bar{\boldsymbol{z}} \mid \boldsymbol{x}) \log \frac{P_\theta(\bar{\boldsymbol{z}}, \boldsymbol{x})}{q(\bar{\boldsymbol{z}} \mid \boldsymbol{x})}\, d\bar{\boldsymbol{z}} \quad \text{(ELBO)}$$

Find an approximation of this present form, use the **variational approximation, ELBO**, instead of maximizing the log of the integral of that thing, we maximize the integral of that stuff over there that is just the ELBO, I have been smart in introducing the Q in my ELBO the q that I am using is just $q(\bar{z}|x)$ why? I exactly know how to compute that, it is the noising process or the denoising process depend on how you compute it, it is a distribution that given x, gives you all the different versions of your z_t, it is computable, that thing you can manipulate because you know all your distribution involved: $P_\theta(\bar{z}, x)$ is your NN essentially, and the $q(\bar{z}|x)$, comes from the denoising process at training time, where I know x, so I manipulate all that stuff and I can get to this little beauty here:

$$\mathbb{E}_{q(\boldsymbol{z}_1|\boldsymbol{x})} \left[\log P_\theta(\boldsymbol{x} \mid \boldsymbol{z}_1)\right] - \sum_{t=2}^{T} \mathrm{KL}\left(q(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t, \boldsymbol{x}) \,\|\, P_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)\right)$$

The first term is your usual friend **reconstruction error**, expectation run over the z_1 drawn from the distribution that returns z_1 given x, so *the first step of the noisy distribution*, gaussian, easy to sample, of the log of the prob of x given z_1, this is my NN, the final application of my NN at final stage, the one between z_1 and x.
So this thing inside here is a gaussian whose mean I can compute, because it is outputted by the NN, the log of a gaussian makes all the exponential disappear, and *left only essentially with the norm two of the difference between the mean and the x*, you can figure out that if you plug the eq. of a gaussian here,

something like the norm 2, residual error comes out, this thing is perfectly approachable, in principle I need to compute this expectation in practice I cannot, it requires integrated out z_1, whenever I cannot compute the expectation in practice, I can sample, take on elem from the dataset x, and use the q function to obtain the corresponding noise version at z_1 and then I estimate with the usual average over my dataset, literally a reconstruction error, the second term is a KL-div between the gaussian predicted by my NN $P_\theta(z_{t-1}|z_t)$, and?? It is a KL-div, it measures how much 2 distr. kind of overlap/ are different one another, one of them is parametrized the other is not, the only one that can be adjusted is P_θ the one from the NN, the only one that has parameter, the NN that emits the mean of the gaussian, P_θ is gaussian, and its the only distribution that can be adjusted, **the second term will push the gaussian from the NN to behave as the distribution** $q(z_{t-1}|z_t, x)$ that is a computable distribution we discuss before, $q(z_{t-1}|z_t)$ not computable because of the nasty Bayesian rule expansion, but that becomes a gaussian when I can also observe x, the KL-div between 2 gaussian I can compute in close form, the KL-div will have some log inside and some expectation, if I plug the eq. for the gaussian P_θ and the gaussian for q, those are 2 gaussian, this KL-div at the end will become a difference between means, the mean predicted by the NN P_θ and the actual mean, that I know from the actual denoising process, that I can characterize because I create the noise that was added there, so I now exactly at TR time how much noise was added to Z_{t-1} starting from x to create Z_t so I can reverse this process, and I can tell you exactly what is z_{t-1} obtained by starting from x substracting from z_t the noise I have created at time t, cause I am at training time, I've completely control over there, I know where I've started x,I know how much noise I added at time t, I'll push my NN to predict means for my gaussian consistent with the z_{t-1} created by adding that noise, so this is the intuition, that term will push my NN to output predictions that are consistent with the different levels of noise that I created at different time, cause this will going to be sum for all time.
At inference time I need this parametrized distribution $P_\theta(Z_{t-1}|Z_t)$ to generate data, at inference time I'll generate a z_T by drawing from a normal and then repeatedly apply P_θ until get to my x.
$q(z_{t-1}|z_t) \rightarrow$ cannot use the target q to train it, cause I don't have it, it is a difficult one, even if we cannot characterize this, there is another version better characterized $q(z_{t-1}|z_t, x)$, this is well characterized, it is gaussian with a certain mean and variance, and this make sense, if I know from where I start, x, I can invert the noise, to go back to where i come from. $q(z_{t-1}|z_t, x)$ it is gaussian and I know how to compute at TR time, because I have x and all the z_t cause I have generated through my forward pass.
At TR time I can estimate q, if I have this particular q at training time, I can use it to generate targets to learn my P, that is the principle, i want P_θ to behave like my original q, but you can't, so we approximate the original q with the TR version that is gaussian, and we make P_θ approximate the Q with the x at training time, once trained it won't need x, it doesn't have it, but at TR time we need it to creates the target, if $q(z_{t-1}|z_t, x)$ is gaussian also P_θ is

gaussian, so I'm gonna be having a NN that outputs the means of the gaussian P_θ and TR will make sure that the gaussian means predicted by the NN will match the actual gaussian mean of $q(z_{t-1}|z_t, x)$ and I know the mean of this one, it is a gaussian that extract from z_t in the direction of x a z_{t-1}, it can be completely determined by algebraic computation. The means for this gaussian can be computed and I can use them as target for P_θ.
Your original log-likelihood is something that you cannot manipulate, I can define the ELBO, in order to define the ELBO I need to define a variational distribution, q function that will act as a sort of approximator term, if I introduce that q function and when I do the integral I know that the integral goes out, the logarithm enters here and everything works as usual, except that q(z) is not really the parametrized distribution you used to, that is a $q(z|x)$, it is the distribution here $q(z_{t-1}|z_t, x)$ that I am using at TR, when I manipulate it will become that, this one is not parametrized, it is a gaussian with a set mean that is a combination of x and z_t there are no params there, it is a non adaptive distribution, since I cannot manipulate the original distribution, and the original distribution, the log-likelihood contains a P_θ, which is the adaptive distribution, I am introducing the $q(z|x)$ approximated distribution that I know how to compute, as if it were a variational function, but q won't move cause there are no params in there, the thing that will move is P_θ, it is a slightly different thing w.r.t. the once that we were used to. When we introduced Q with the VAE q was the encoder, was a fully variational like distribution, it had params, in this case the q I am introducing is a q that acts as a target rather than a distribution, but the ELBO does not tell anything about the fact about the q that we introduce must be a parametrized distribution, the all derivation of the ELBO works anyways. Clearly the quality of the ELBO approximation will depend on the ability of the P_θ to work appropriately with the q(z) that I have introduced, by aside from that the all derivation of the ELBO holds.
With simplification I get this nice equation in which the first term is a classical reconstruction term and the second term derives exactly from the fact I introduced this q distribution through the ELBO that now acts a target for the true parametrized distribution P_θ. So whenever I'm gonna be maximizing this thing I'm gonna be taking the derivative of this equation w.r.t. to θ. The first term pushes the param of the model to make sure that the x_s that are generated are consistent with the data, they well reconstruct the image in the final passage from z_1 to x, the second term make sure that in all the other transitions from t to t-1 the P_θ distribution behaves consistently with my target gaussian which is my approximated denoising distribution when I know x, totally legal at TR time, if I plug the definition of KL-div into that, the first term is the logarithm of the normal of the final denoising step, P_θ is gaussian I can put in place of that a Gaussian with the mean predicted by the NN that predicts the gaussian for that point, and it will be the gaussian from which I will draw x, I do i get the mean of this gaussian, from a NN that receives in input z_1, so in the formula you can read the log of the gaussian whose mean is obtained by applying the NN μ_θ to z_1 and the time t that in this case is 1. The second term is relating with the KL-div, sum overall time, that are not the first time in the chain,

essentially the difference of two term, term A and term B, term B is the simpler one, the mean of the gaussian predicted by the NN at time t, from which I'll pull out z_{t-1}. What is A? the actual mean of that gaussian that comes from q, when you do the math is the actual mean of the gaussian $q(z_{t-1}|z_t, x)$ and this is know, obtained by a few calculation, this is the target mean, *what this loss is doing is pushing the predicted mean by P to be equal to the target means for all the time step, pushing the NN at each time step of my diffusion process to predict the right means, where right is defined by the means of this gaussian q, and the first term ensure that we are generating realistically looking data when denoising at the final step.*

$$\mathcal{L} = \sum_{\boldsymbol{x}} \left(\left[-\log \mathcal{N}\left(\boldsymbol{\mu}_\theta(\boldsymbol{z}_1, t), \sigma_1^2 \mathbf{I}\right) \right] + \sum_{t=2}^{T} \frac{1}{2\sigma_t^2} \left\| \frac{1-\alpha_{t-1}}{1-\alpha_t} \sqrt{1-\beta_t} \boldsymbol{z}_t + \frac{\sqrt{\alpha_{t-1}}\beta_t}{1-\alpha_t} x - \boldsymbol{\mu}_\theta(\boldsymbol{z}_t, t) \right\|^2 \right)$$

Training practical view. Loss can be heavily simplified by reparameterizing so that the model predicts the noise $\epsilon_\theta(.)$ that was mixed with the original data, rewriting x as:

$$\boldsymbol{x} = \frac{1}{\sqrt{\alpha_t}} \boldsymbol{z}_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}} \boldsymbol{\epsilon}_t$$

Inserting x above into the ELBO yields after a while to a different loss where the network now predicts the unit noise given current noised input z_t. So that loss, seen before, won't be the one that we are gonna be using, my NN will be generating gaussian means that have to be the same size as the image and actually at the end will be the image, this NN is predicting averages, and this means are the ones from which you draw the different denoised version until you get to the fully denoised version which is x. Change a little bit how I use my NN, if I manipulate the definition of z_t in my diffusion kernel, that give me straightaway z_t starting from x, instead of having it from z_t i solve it w.r.t. to x. This equation gives me exactly the x knowing the noisy version of x at time t and *exactly how much noise has been injected in the network at time t*, this is saying that if I know a noisy version of my image and exactly how much noisy is injected I can recover x, I can plug into my equations and re-arrange stuff in my ELBO and what iget?? If I use this eq. that tells me that x can be reconstructed exactly if I know the error, rather than predicting with a NN z_{t-1} why I just don't predict ϵ_t , it is easier, previously at time t I have z_t and through my denoised version i predict z_{t-1}, by using the NN that generates the mean of z_{t-1}, now I say if I am at time t, and I know z_t why don't I just predict how much noise it is introduced in the net at time t??

If I can do that again I can use the diffusion kernel, I don't have to do all the denoising step by step, I can sample at random one time step in my time, take the corresponding z_t generate the noise with the prediction of the NN, confront it with the actual noise and train the NN, take another time in random, generate the prediction about the noise, confront with the actual noise that I have injected and so on, purely supervised learning, a NN trained to predicted noise injected into an image, the input to this NN is z_t shape of the image, the output of the NN is ϵ_t shape of the image, all the noise injected in all the pixels, not a

very complex NN, any NN that given an image can predict an image in output, U-net, Conv-DeconV, vision transformers,...

$$\text{Loss}(\theta) = \sum_{\boldsymbol{x}} \sum_{t=1}^{T} \left\| \epsilon_\theta \left(\underbrace{\sqrt{\alpha_t}\boldsymbol{x} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_t}_{\boldsymbol{z}_t}, t \right) - \boldsymbol{\epsilon}_t \right\|^2$$

Under this assumption, I obtain a nicer, simpler, loss, sum overall examples in the dataset, sum overall time in the diffusion, I am confronting the prediction of the NN that predicts the current version of the noise injected at time t, with the ground truth noise injected at time t, and this is a NN that receives in input x and the noise introduced until that time, $\epsilon_\theta(z_t, t)$.
In the end, the implementation of this thing is really simple, as soon as you get rid of all the craziness coming from the derivation, you design your model to be a NN that receives an image in input and outputs an image, choose your favorite one, take your TR data, for example in a minibatch you pick up one image, to which you wanna apply the diffusion, choose a time at random, pick up at random one elem from the dataset, generate noise at random, and then feed the NN with Z_t , generated on the fly Z_t by taking x and applying the noise you have generated using the diffusion kernel, generating on the fly Z_t, feed it to the NN, have the NN predict the injected noise, based on the information in which the noise is mixed with the image, the NN makes its prediction, you confront its prediction with classical MSE loss with the actual noise that you have injected, that will be your loss, apply the gradient done, no seq. relationships between time here, no longer concept of time as such in terms of dependency as time is sampled uniformly, using at full speed the diffusion kernel that allows me to compute whatever I need in a specific instant in time just with a single shot.
A NN that have the behavior that I need, image (noisy version of X) $\rightarrow$ NN $\rightarrow$ image (the error same size of the input), my NN will be for instance a Convolutional/Deconvolutional NN, the first part of the NN does all the convolution compression and the second part that does all the deconvolution/decompression until I generate the noise/error image at the end.
Trick: **copy connectivity**, when you deconvolve a bit you gonna be having a copy of the corresponding convolve part just to provide you the info that you need, all trick that you have in NN like U-net.
This NN needs to receive in input also the time, because at different times I expect to be injecting more or less noise, what time is it I tell it with a specific input, I need a way to embed time info, like positional embedding in transformer, we can use sinusoidal/cosinusoidal embedding. **Time is fed typically as an additional channel before convolution**, with sinusoidal positional embeddings, time features are fed to all the residual blocks.
The terms that control the variance of the gaussian are related with the noise schedule are those that trade between the image and the noise that I am adding, typical choices for the forward and reverse diffusion are that you choose β_t to be linearly increasing, start with little contribution from noise then increasing,

σ_t^2 typically match β_t so it is not typically a param, you can learn σ_t by sort of operating on the bound that we have on the ELBO, and optimize also σ_t to push the ELBO closer to the loglikelihood, β_t can also be learned, the intuition is that if you minimize the variance of the TR objective you get a better noise schedule... just refinements. But important start with a little noise at time 0 to not destroy immediately and too quickly the info and you'll never recover.
Injecting noisy slowly in the beginning and then possibly in a stronger way at the end is one aspect, the other aspect that you can't force too quickly is the speed of the diffusion process, you cannot expect to implement the data diffusion model operating on 5 step will work, for a certain number of reasons including the theoretical one, we can approximate the true denoising distribution q only if we take small steps and enough long, big enough number of steps in the diffusion. This is not a problem at TR time cause you have the diffusion kernel, when you are sampling you do not have it, when your model is gonna be trained, at inference time, you will pull some random noise from the gaussian, and then apply iteratively your NN to denoise it for the amount of step you have defined in the TR process, no way to skip that, cause you don't have the starting point x, the diffusion kernel work with x, but in generation mode you are looking x, that's way chatgpt takes so long to generate image, **it has to diffuse for the needed number of step.**
This gives us a way to generate x, by pulling it from P(x) without having P(x) explicitly in reality, we never compute P(x), explicit model because at some point we write a likelihood, but we never learn the likelihood, we learn distribution q that are all Gaussians and none of those gaussian is any close to the actual data distribution, but we get a smart way to sample from it, **different from GAN in a GAN you never have anywhere nothing near to a probability distribution, no distributions in a GAN**, here there is, our q, P_θ that we learn, but none of them is actually the data generating distribution or an approximation of it, the flow models instead learn an approximation of the p(x) distribution, but all these said, we know we have a smart way to sample from p(x), fine, but I want more, because I want to sample from $P(x|y)$ conditional distribution or conditional generation, generate image based on some conditioning information. **Classifier guidance**: on one hand you have your diffusion model, that can do his job of generating your image x, **unconditionally**, on the other hand I have **another neural model**, *that possibly can reuse part of the diffusion model, whose job is receiving in input an image and classify it into a certain number of classes*, for instance 3 classes of visual images (cats, dogs and cows), this conditional process is a process in which y or c, is c one of these 3 classes, I train the diffusion model on images of cats, dogs and cows, *in parallel I train a classifier that receives in input images and tells me if they are are cats, dog or cows,* but don't receive in input the x images, it receives in input z_t, the noisy version of the images, whenever I'm gonna be generating the z_{t-1}, the denoised version of the image, I'm gonna be taking the part that comes from the denoised version of the image, I'm gonna be taking the part that comes from the reversed diffusion process that I know, the $\hat{z}$ is just the mean predicted by my diffusion model + σ_t * ϵ which is the noise that I added

around the mean, this part here is just some that I can obtain with my diffusion model, if I use only this part here I get my denoised version but not informed about the cluster I want to obtain, make sure to shift that into the direction given by the derivative of my classifier w.r.t. to its input. I have a candidate image that comes from the diffusion model, I have another thing that comes from this derivative that has the same size cause it is a jacobian, the derivative of the classifier w.r.t. to all the points in the image z_t, and then I combine the 2 in order to get the denoised image z_{t-1}, orienting the diffusion process to consider not only the info about the visual info, but also the semantic deviation due to the fact that I don't want any image, I want those images that are in the area of the space where cats are, or dogs are or cows are, this classifier through its derivative guide your generation process, what possible go wrong? the fact that training a classifier to actually recognize a cow out of a very noisy version of a cow might not be exactly the smartest task you wanna do, lot of very bad gradient from that.
What we want to achieve is learning how to inform the generation of z_t with some external data c, and we are somehow approximating it with 2 terms, the first term comes from the diffusion model, the second term comes from the classifier, we are obtaining the direction in which we change our z_t by combining two directions, an unconditional direction and some direction that is sort of pushing us strongly where cows or cats or dogs reside, and we have some term γ that regolates the trade-off between the 2, if you look at the problem from this perspective it's interesting cause you can interpret those 2 distribution as doing different things, let's name the first B and the last A, if we don't put B in there, what could possibly A give us?? A is a very strong single direction, once you fix C, and in our case C are only 3 of them, that gradient over there will push you into 3 major directions, one for the dog, one for the cow, one for the cats, so term A, if you think for the perspective of generating images will generate very stereotypical images, the most stereotypical for a cow, for a dog and for a cat, if you only follow that, it is the gradient that goes in 3 directions essentially. The first term is instead adding the sparkle of creativity to it, rather than getting the stereotypical cow you will get the Milka cow and all the creative variations of the cow, trading between the two terms (γ) you will have more realistically looking but less creative images or more creative images at the risk of departing very much for the guidance.
Another approach is that you are tautologically obtaining $P(z_t|c)$ by mixing $P(z_t)$ with $P(z_t|c)$, what the heck?? You want exact the same effect as before, the ability to trade between being very creative or very stereotypical, by having two distributions: one is conditional and the other unconditional, but you want them to be fused into one? you get the single model that do both of them, a single model that can predict conditional and unconditional distribution, it is not incredibly difficult to set it up, because class info from 1...C the visual classes, I want a model that receives in input also the class info in order to generate x, this is my diffusion model, receive in input the class info produces a new x, if you have C classes, in order to obtain a model that generates both conditional and unconditional I append the magic zero class, that tells the model to gener-

ate unconditionally, you need an input that you can switch off at random, by setting 0, and you will have to alternate when you are training between training based on conditional info, and training based on complete unconditional decoding. Drop out the conditioning info every now and then when you are training, this way you get a model able to integrate between the 2, if you want to decode an image into a specific class, you sample from the gaussian distribution, you fed into your diffusion denoising process, first without any conditioning, then with the conditioning and you trade the two and you get exactly your mixed image. $\gamma = 1 \rightarrow$ no creativity , $\gamma = 0 \rightarrow$ an artist.
In practice, want to generate a dog, my model receive an input the model and an encoding of the class ID, that gets into the model how you input the time in a diffusion model, however you have to encode or with a 1-hot, or with a semantic informed encoding like a BERT embedding.
Guided Generation – Classifier Guidance: Guide diffusion process using auxiliary data c, using the gradient of a trained classifier as guidance:

- Train the diffusion model unconditionally
- Train a classifier $P(c|z_t)$ where c are conditioning labels
- Add an extra term when sampling the diffusion model, i.e. when reconstructing z_{t-1} from z_t, that modifies the reconstruction in the direction given by the gradient of a classifier

$$\boldsymbol{z}_{t-1} = \hat{\boldsymbol{z}}_{t-1} + \sigma_t \boldsymbol{\epsilon} + \sigma_t^2 \frac{\partial \log P(c \mid \boldsymbol{z}_t)}{\partial \boldsymbol{z}_t}$$

Classifier guidance: issues $\rightarrow$ Classifier guidance comes from mixing the predicted score function of the unconditional diffusion model with the classifier gradients:

$$\frac{\partial \log P_\gamma(\boldsymbol{z}_t \mid c)}{\partial \boldsymbol{z}_t} = \frac{\partial \log P(\boldsymbol{z}_t)}{\partial \boldsymbol{z}_t} + \gamma \cdot \frac{\partial \log P(c \mid \boldsymbol{z}_t)}{\partial \boldsymbol{z}_t}$$

Classifier receives a noisy input z_t at each step. Most of z_t is of no use for predicting c $\rightarrow$ arbitrary classifier gradients.
Classifier-free Guidance Derive guidance from Bayes rule.

$$\frac{\partial \log P_\gamma(\boldsymbol{z}_t \mid c)}{\partial \boldsymbol{z}_t} = (1-\gamma) \cdot \frac{\partial \log P(\boldsymbol{z}_t)}{\partial \boldsymbol{z}_t} + \gamma \cdot \frac{\partial \log P(\boldsymbol{z}_t \mid c)}{\partial \boldsymbol{z}_t}$$

Training conditional diffusion with dropout (randomly removing conditioning). Conditioning replaced by flag input (presence/absence of conditioning) $\rightarrow$ single model for conditional/unconditional diffusion.
How to generate an **high resolution image**? Same as progressive GAN, *you have a diffusion model that diffuses on small dim images, then another diffusion model that upsample, that upsample, ...*
You can use your **diffusion model to do semantic segmentation**, *the denoised version of your image could be a semantically segmented image*, and *part of the loss is how good you are at classifying the single pixel in the corresponding*

semantic class.
Conditional generation: it is not surprising now how I generate an image based on text, I take my text, I need to find an embedding for my text, it becomes my vector c, that my use in my conditional generation, if I don't use classifier guidance but just the version with c in input to the diffusion model this thing works immediately.
Conditional progressive diffusion model to generate images controlled by text, a certain number of **convdeconv net** or **U-net** that operates **in cascade** *you first decode the general appearance/low resolution of your dog*, so how it works: *I have an initial model that condition only on text, sample a random noise, decode into a tiny little dog that looks like the shape of a dog, this info gets in input as additional conditioning info to the second layer whose job is simple, cause it has to learn to upsample that until you get to dog of the reasonable size that you like.*
Dall-E does not work like that? if I had to generate an image every time by diffusing using a diffusion process defined over so many different images that would be a nightmare in terms of time, **Dall-E works on the concept of latent diffusion**, you want to generate an image, but **creating a diffusion process over images is too costly**, how do I do that? Let me take an image, and let me assume that I know how to generate a compressed version of the image, let's call it encoder, that generates a compressed vector z, run the diffusion rather than on the original image I diffuse on my compressed latent encode, get my Z_T by applying the noise process, than backward i denoise Z_T until I get to Z_0 which is not yet the image, is **the reconstructed latent code** of the image, then I need just a magical decoder whose job is transforming back into an image, **the diffusion operates only at the level of the vector**, with an autoencoder in the external part you solve your problem, it's **a VAE with a diffusion process in the middle on the latent space**. Of course you can add conditioning info, the text that you need to control the generation and you know that **the text has to be inputted together with time, as an additional source of info when you are running the diffusion in your U-Net**, the way you can fuse textual information with the visual generation, you're creating at the level of the U-Net is for instance as suggested by Q,K and V **attention or whatever other mechanism you want to use to fuse 2 channels of info**. This is how **DALL-E** works, it is **a diffusion model defined over latent encodings**, the latent encodings are smart, are **CLIP embedding**, embedding that by nature are obtained by a model that is trained multimodally, a large language vision model, the way in which this model is trained is to make sure that if you have **text and associated images they are gonna be encoded in vector that are very similar and text and not associated images are gonna be encoded in vectors that are different**, so it gives you these 2 encoders and 2 decoders one for text the other for images, it's pretrained, in Dall-E whenever you write your text and you request a specific type of image that will be converted by the clip encoder into an embedding than you have your diffusion, diffusion process or another one that transforms this text clip into the image clip, so it is a transformation

of the textual information into the image related embedding and then you can use diffusion decoding as seen before to obtain the image.
We rely on a noise model that is for continuous type info, does not work well on discrete data. Use a encoder/decoder that transform discrete info into dense vectors and run a diffusion on the latent space. The price that you pay is that **you never had the actual likelihood**, we never estimate, we give you a data point, and tell me how probable this sample is according to your model? You can't do it with a diffusion model.
Cascaded Conditional Generation: framework for generating images based on conditional inputs like images, text, or other forms of guidance. The model generates data in stages (or cascades), refining the output progressively (e.g., from low-res to high-res). Each stage uses conditional information to guide the generation more precisely. Conditional Input Mechanisms: **Scalar–vector embedding + spatial addition** Scalars (e.g., class labels) are embedded into vectors, then broadcast spatially and added to the feature maps. **Image-channel-wise concatenation** The conditional image (e.g., a segmentation mask or depth map) is concatenated along the channel dimension with the input image or feature maps. This helps the model directly "see" the guidance image. **Text-vector embedding + spatial addition or cross-attention**. Text inputs (e.g., prompts) are embedded via language models. These embeddings are then: added spatially like positional encodings (simpler). Or processed via cross-attention, where the generator attends directly to the text (more powerful for complex guidance).
Latent Space Diffusion: Latent Space Diffusion is a generative modeling technique that performs the diffusion process in a lower-dimensional latent space rather than in the high-dimensional pixel space, leading to significant computational savings. An encoder first maps the input image x into a latent representation z, where the diffusion process (i.e., iterative noise addition and removal) occurs. A denoising U-Net ϵ_θ refines the noisy latent vector z_T through multiple steps, leveraging cross-attention mechanisms to integrate conditioning signals such as text, semantic maps, or image features. After denoising, a decoder reconstructs the final image $\tilde{x}$ from the clean latent vector. This approach allows efficient and flexible image generation conditioned on diverse modalities. **DALL·E 2** is a text-to-image generation model that operates through a two-stage process: **representation learning** and **image synthesis**. First, it leverages **CLIP to learn a joint embedding space for text and images, aligning their representations through a contrastive loss**. Given a text prompt, the model encodes it into a CLIP text embedding, which is then passed through a learned prior (either autoregressive or diffusion-based) to generate a corresponding image embedding. In the second stage, a diffusion decoder reconstructs a high-quality image from this embedding. This separation of semantic understanding and image synthesis allows DALL·E 2 to generate diverse and semantically aligned visuals from textual descriptions.
So to recap: Generate data from noise through a learned incremental denoising with fixed steps. Diffusion process can be reversed if the variance of the gaussian noise added at each step is small enough. Training goal is to make sure that the

predicted noise map at each step is unit gaussian. During generation, subtract the predicted noise from the noisy image at time t to generate the image at time t-1. Diffusion can be computationally involved. Need to take many small steps. Vanilla diffusion on a latent space same size as the original data. Guided generation can improve sample quality (and reduce diversity). Latent space diffusion improves efficiency of generation and generalizes which data that can be used (including discrete objects) and allows introducing semantic structuring in latent space.

24 Normalizing flow

We still have a process going in one direction and another going in the other direction, in the diffusion model the process going left to right was static. *The direction that goes from the data point x to the randomized projection in Z_T is completely static in Diffusion models*, how could you possibly learn a likelihood consistent with the data generating distribution out of it?? We need to relax that somehow, Normalizing flow relax that and falls into the world of **fully visible models, no latent variables**, otherwise we fall into intractable densities so we are not able anymore to get to a good estimate of the likelihood. We need not to have z, we need to have **distributions to be defined in terms only of visible info**, with sampling RNN, the autoregressive model, this was incredibly difficult cause we were generating one pixel at time to do that. The intuition is what if we can **add piece by piece to the information**, if I want to *generate something can I generate something piece by piece*?? Can I, at every step change slightly the piece of info and obtain a slightly changed version, then change a little bit more , and a little bit more... With Normalizing Flow you have **2 processes** *the black one: you take the castle and you get a completely normalized castle, and then the red part that starts from the normalized castle and reconstruct the original castle.*
We want to transform a very complex data generating distribution into a very simple distribution, I have to do very small operation, I take my original castle, I detach a lego piece of the castle and position in a specific position of the table, then take another block and position in a specific position of the table until I have destroyed/decomposed my castle but in a very ordered way, now I have all my building blocks on a specific order on the table, I take the last piece that was destroyed and put in the right place, then take the second last etc and I step by step invert the process and regenerate the castle. Normalizing flow does exactly this, take the original complex distribution and change it slightly a tiny bit ,and then change it slightly , ...again, again **until you have morphed you original distribution into a gaussian again**, from which we can sample easily, now I take the gaussian and repeat the inverse of the operation I did the previous time step, **there is a constraint, whatever operation you run in the black step it needs to be run/inverted in the red direction**, *the function that you use to transform one thing into another must be bijective.* **They need to be invertible**. That function we are gonna be using to this

morphing are **change of variable function**. Every time we change a bit our distribution, our distribution previously was defined over $P(Z1)$ now is defined over $P(z2)$ where z2 is a change of variable w.r.t. to Z1, i.e it contains inside z1 modified accordingly, this will morph a complex distribution into a simple one, and this is a fully invertible thing and I can go back.
Learn a probabilistic model by transforming a simple distribution into the complex data generating distribution using a deep network: easy to sample and evaluate the probability, requires a specialized architecture where each layer must be invertible. **Probabilistic Change of Variable**: take a **tractable base distribution** P(z) over latent variable z and a model density P(x) over data x. Apply a change of variable function (possibly learned with params θ) $x = f(z; \theta)$. In addition, we are going to require that f is invertible $z = f^{-1}(x; \theta)$
If I want to generate data, I sample one example from the normal, which is simple to do, and decoding through my process into actual data. Why don't use diffusion model? Because if I want to **estimate p(x)**, I take x and start applying my change of variable to x, until I get to a z, which has a distribution which is a gaussian, I can **compute p(z) and multiply p(z) for all the changes of variable and get p(x)**, this will give you an **explicit likelihood model**. A model that allows both **density estimation** and **sampling**. In order to do density estimation, we plug the castle, we apply all the normalizing transformation, we multiply whatever happens in the process for the final probability of the point z in the base density and we get an accurate estimate of the original densities. The first direction is called **inverse or normalizing direction**, cause takes a density and transform into the normal, and the second is the **forward or generative** direction cause takes a sample from the base density and transform into a sample from the data density. Substantially different from a diffusion model as we are morphing distribution not noising sample as a by-product we gain the ability to take a sample drawn from a gaussian and transformed into a sample that comes from our distribution but we are doing density learning. This works well because the forward process is simple enough to implement, sampling from a gaussian is simple, and as long as all this steps implemented by the red arrow are simple enough this is a simple sampling process.
Change of variable, *you can morph a density into another one by a change of variable*, assume you have a simple base density and a model density, our data density, more complex, **I can transform from the base density to the model density by a change of variable**, I can obtain an x in the space of the model density by transforming z in the domain of the base density through a **function f which is a trainable function with param**, since we are doing ML here, f is a function that is transforming something that I pick up in base density space z, this function returns me the corresponding point in the model space x.
Whenever I want to map something from a z-space to an x-space and I have a function that do it, it is easy. Provide that f is invertible I can invert this process with f^{-1}. Through my function I can go from x domain to z domain and viceversa, because it is invertible, but there are 2 distributions there, whatever is happening needs to be consistent with the fact that the whole probability

mass needs to be conserved, the integral over the z space is 1, the integral over the x space is 1, **when morphing an infinitesimal interval dz into an infinitesimal interval dx the infinitesimal amount of mass needs to be preserved**, if I am picking up up a point in the z space that is near to the peak of the gaussian, I have a quite big probability/area, when projecting through f in my space x, if I end up in a space where the density is small this needs to stretch into a larger dx, the amount of probability mass needs to be conserved, so in other words when reasoning in probabilistic f is doing a morphing of the 2 distribution, stretching or compressing the space in order to conserve the probability mass. **f needs to be invertible**, we need want to implement both the normalizing and the generative flow but needs also to be **consistent with this preservation of the density mass**. Some property of this function f that we are gonna be using to say how much an area that comes from z is morphed by the function into an area in x.
Any operator to tell me if the space is locally contractive or expanding given a function?? **The Jacobian**, simple linear operator, that tells us how much the space is morphing around a specific point in our domain.
Linear 1D Change of Variable. Normalizing Flow define complex densities by transforming a base one by invertible mappings (bijections). Simplest case in 1D is a univariate Gaussian base density $z \sim \mathcal{N}(0, 1)$. Simplest change of variable (forward) by linear transformation $x = f(z; \mu, \sigma) = \mu + z\sigma$. Inverse then (under $\sigma! = 0$) $\rightarrow z = f^{-1}(x; \mu, sigma) = \frac{(x-\mu)}{\sigma}$. With P(z) known we want to find P(x)
1-dim change of variable, simplest base density is 1d gaussian, z comes from a normal, simplest change of variable, an affine transformation, relocating my gaussian in a different position, it is invertible. Also in higher dim I want a simple function, parametrized but easily inverted, we need to preserve our probability mass, necessary condition$P(z)dz = P(x)dx$, that is P(z) in a infinitesimal interval dz is equal to P(x) in an infinitesimal interval dx, if we want to know the form of my P(x) distribution I can solve w.r.t. to that, x is a function of z, so dx/dz is actually a derivative $x = \mu + z * \sigma = f(z)$, first derivative of my invertible mapping f, I want the absolute value, cause I want the volume of my function f, the transformed distribution P(x) can be obtained from the base distribution P(z) by multiplying it for the derivative of the change of variable function w.r.t. to z. The jacobian s going to be the multidim generalization of this first derivative w.r.t. to z, the derivative of your function f w.r.t. to each of the dim that gives you the Jacobian, for the unidim case my model distribution can be obtained from the base distribution through a change of variable by **taking the base dist. P(z) and multiply for the absolute value of the first derivative of my change of variable function**.
Linear 1D - Mass conservation: The volume may change but the density must be preserved. The necessary condition for this is $P(z)dz = P(x)dx$. The probability of data x under the transformed distribution is

$$p_X(x) = p_Z(z) \left| \frac{dx}{dz} \right|^{-1} = \frac{P(z)}{\sigma}$$

This tells me how to trnasform a base dist. into a slightly less base distr. through a change of variable f.
$z = (x - \mu)/\sigma$ and σ is the derivative of f. w.r.t z
My distribution P is gonna be a normal with mean μ and variance σ^2. $x = \mu + \sigma * x$. This is **the forward transformation** *from the base distribution to the complex distribution, it is the generative direction*, implemented with a single step, now I want to implement this transformation between a base distribution and a very complex data distribution, not in a single step, otherwise f becomes too complex, you want to have f simple, but operates repeated change of variable to transform a gaussian into a very complex distribution, **compose changes of variable**. **Generative perspective**: I can sample a z_0 from the gaussian N(0,1), now through change of variable I can obtain z_1 with f_1, for instance with an affine transformation, now let's apply a second change of variable $f_2(z_1)$ to obtain x, not directly from z_0 but trough an intermediate z_1, what is the form of this distribution ?? A chain of multiplication of Jacobian, or derivatives in this case, P(x) is gonna be the prob of the base distribution times the derivative of transforming a sample from the base distribution into a sample in the space z_1, times the derivative of transforming a sample from z_1 into a data samples. N different warping, I will have a chain of N different multiplications, one for each of the change of variables. Composing linear transformation since they are closed under composition, it is equivalent to a single linear transformation, we are not gonna be sticking on linear transformation.
Linear 1D – Iterated Forward Pass

- Sample x through 2 mappings (transformations):

$$\begin{aligned} z_0 &\sim P_z \\ z_1 &= f_1(z_0) \\ x &= f_2(z_1) \end{aligned}$$

- Density obtained by composing forward transformations:

$$P(x) = P(z_0) \left| \frac{dz_1}{dz_0} \right|^{-1} \left| \frac{dx}{dz_1} \right|^{-1}$$

Inverse flow: **Given a sample from x space projects into my base distribution**: it is a change of variable, only instead of using f i use f^{-1}, cause is the same change of variable function as before just inverted, I am operating with bijective function.
Linear 1D – Inverse Flow

- We may be interested in estimating the density of a given input sample x.
- Requires building the inverse flow ($g = f^{-1}$):

$$\begin{aligned} z_1 &= f_2^{-1}(x) = g_2(x) \\ z_0 &= f_1^{-1}(z_1) = g_1(z_1) \end{aligned}$$

- And computing the density accordingly:

$$P_x = \left|\frac{dz_1}{dx}\right| \left|\frac{dz_0}{dz_1}\right| P(z_0)$$

The generative flows it is useful cause I can sample from a very simple distribution and reconstruct an x through change of variable. In order to get a density I want to be able to project a specific sample from the data x, into my base distribution and estimate the density that comes from data. Given an input sample x, I start operating the reverse process, given x I obtain z_1 by the inverse of f_2, f_2 before was the function that given f_1 returns me x, then once I have z_1 I obtain z_0 by the inverse of f_1, z_0 is a sample from the base distribution and I know how to compute a density for a normal gaussian. **So my density for a specific point x, P(x) is gonna be given by $P(z_0)$ and all the warping necessary to transform the complex density into the base density, a product of first derivatives.**
We've seen transformation between 1-dim variables but **in general we want to transform between vectors**, x and z will gonna be vectorial random variable each with its own density P(x) and P(z) and their flow needs to be invertible and differentiable. If we want to generalize all said before in this new vectorial setting to transform a base prob P(z) into P(x) I multiply by the inverse determinant of the partial derivatives dx/dz or equiv df(z)/dz, this is a Jacobian as we're in a vectorial space, that is the matrix with all the partial derivatives, but I don't really need the matrix with all the partial derivatives, only its determinant that gives me a measure of the volume change between two spaces when you morph it thanks to f.
The changed area is the Jacobian determinant when you move from x to z.
We need the determinant of the Jacobian we don't need the Jacobian, or to invert it, computing the determinant of a general matrix is difficult cubic.
Multidimensional Flow

- Extend the approach to the multi-dimensional case.
- $\mathbf{x}, \mathbf{z}$ are vector-valued random variables with densities $P(\mathbf{z})$ and $P(\mathbf{x})$.
- Flow $f(\mathbf{z})$ is invertible and differentiable (closed under composition).
- The transformation $\mathbf{x} = f(\mathbf{z})$ leads to the probability change:

$$P(\mathbf{x}) = P(\mathbf{z}) \left|\det\left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}\right)\right|^{-1}$$

- The determinant of the Jacobian provides information on the rate of change of the volume affected by the f transformation.
- The area (or volume) change under f can be computed with vector calculus, and it turns out to be the Jacobian determinant.

In the general multi step case we are gonna be having a **forward mapping** and an **inverse mapping** just like the base case, the inverse mapping is the normalizing flow, that takes the model density and flows it into the normal base density and what we use to compute density, *a direction that let us to relate a specific point in the x space to a point in z where I can easily compute the density.* P(x) based on P(z) times a certain number of morphing, trick to have a density estimate , if I learn the params of my inverse mapping appropriately I'll get something that gives me **the data density P(x) as a function of P(z), easy, a gaussian times a certain number of multiplication of determinant of Jacobians of inverse function f_i^{-1} w.r.t. to z_i**: *need to compute this multiplication in an efficient and numerically stable way.*
The forward mapping is used to generate, draw a z from my simple base distribution and then through a series of f, direct transformations, I will get $z_1 = f_1(z_0)$, then I obtain $z_2 = f_2(z_1)$, until I get to $z_n = x$, my decoded data space sample. In this case the forward mapping flow, the morphing, is described by something similar but I am differentiating w.r.t. to different variables , P(x) is the product of the base distribution simple gaussian, because it is the only that I actually know, the other distrib. are obtained through a change of variable, and then the multiplication of the determinant of the Jacobian of f_i given z_{i-1} this time, because it is a forward I obtain $z_1 = f_1(z_0)$ so z_i is a function of z_{i-1}. And actually I take the inverse of the determinant. The requirements are the same: need to be able to compute efficiently and in numerically stable way the determinant of my Jacobian.
Some Considerations & Desiderata

- Can use log densities for numerical stability and learning:

$$\begin{aligned}\log P(\mathbf{x}) &= \log P(\mathbf{z}_0) + \sum_{i=1}^{N} \log \left|\det J_{\mathbf{z}_{i-1}}(f_i^{-1})\right| \\ &= \log P(\mathbf{z}_0) - \sum_{i=1}^{N} \log \left|\det J_{\mathbf{z}_i}(f_i)\right|\end{aligned}$$

- Can optimize the parameters θ of $f_i(\cdot\,;\theta)$ using gradient-based optimization of the log-likelihood above.
- Requirements for f_i:
  - f_i must be invertible and differentiable (and remain so throughout learning).
  - The composition of f_i functions must be expressive enough to map a normal distribution into an arbitrary target distribution.
  - The determinant of the Jacobian should be easily computable (e.g., using diagonal or triangular structure).
  - Computation of f_i must be efficient for sampling.

– Computation of f_i^{-1} must be efficient for inference and learning.
– Computation of f_i must be numerically stable.

Very many determinant to compute, very many of them to multiply one another, not that difficult to obtain. The forward mapping is used to generate, draw a z from my simple base distribution, z_0, through a series of f_i direct transformation, $z_0 \rightarrow z_1 \rightarrow z_2 ... \rightarrow z_n = x$ my decoded data space sample. It's gonna be P(x) = to the product of the base distribution, times the multiplication of the determinant of the Jacobian of f_i given z_{i-1} because it is a forward, $z_1 = f_1(z_0)$. One thing to make stuff more numerically stable, when learning, reasoning in terms of **log probabilities**, which we'll get also nicely in the world of **maximum log likelihood estimation when we train the model**, so we're gonna be using log densities for stability and learning, log P(x) and when you apply logarithm to that factorization up there that is just a product of stuff, we get the sum of logarithm. That's your training equation, you train the model to maximize this thing here, the first term is the log of a gaussian, it's a dumb thing, in the second term there is gonna be something that has the derivatives of your functional mappings f in there in a way or another, not all of them actually but only those that are relevant for the computation of the determinant, if you choose your model to be smart enough that determinant will only depends on some few partial derivatives and will be very easy to compute. You need to differentiate this loss w.r.t. your model params, the params of your f function, because we need to learn those f function but making sure they stay bijective. We will learn our model by doing a maximization w.r.t. to θ of the log-likelihood, with θ that are the params of my function f_i, different f_i will have different params, we will be differentiating w.r.t. to all of them. f_i needs to be invertible and differentiable, if we choose to implement f_i with a NN surely remains differentiable we need to be careful on how we craft our NN because they will need to remain invertible. f_i cannot be some dumb linear transformation, because it is not expressive enough, cannot map a normal into an arbitrary distribution, some non linearity is required. We need to compute the determinant easily cause we have to compute many of them all the time, through learning and trough sampling, the Jacobian needs to have some nice properties, diagonal or triangular matrices have easy determinant to compute only looking on the elem on the diagonal, we are gonna look for such matrices. How to implement this stuff with NN? **Neural flow layers**, we need to look into *flows as invertible neural layers*, again our nice affine transformation is fantastically invertible, I can use a linear layer, $f(z) = b + Wz$, that is a transformation that can be implemented with a NN, b bias term, W weight term, but it is linear. Not sufficiently expressive.
Another thing that is easy to invert is **Pointwise nonlinearities**, your function f, applied to z, is gonna be a function f applied to the single Independent element of your z, provided that f are smooth enough, like Relu (piecewise-linear) or smooth splines this works reasonably but it is **pointwise cannot capture correlation**, your param θ will never capture the correlation between z_1 and z_2 becasue never see z_1 and z_2 together only independently, between the affine

transformation and the Pointwise non linearity we have the hints of how things can be done, we might be able to do things, in such a way that **a part of my vector z is a linear transformation, a part of my vector z is a non linear transformation and those two are combined smartly → coupling flows**: image a generic transformation in the forward flow from z to z', take my z at a specific time for instance z_0, z is a vector and I split in 2 part z_1 and z_2, the first part I copy, and report as it was in z_1', then I use z_1 to feed it through network, which I call θ cause it generates a param θ, so z_1 is used to generate the params of my function f that transform z_2 in z_2'. z' has the first half identical to z, and the second part is a transformation f, of the second half of the first vector, z_2, where the params of my transf. f are given by θ which is based on the first half of z. θ is generated by a NN that generates the params from one of the simple transformation that we see before. The inversion of this thing can be obtained, provided that f is invertible, because the NN operates to generate θ so it's not a problem, to invert z_1 is easy, copy z_1' to invert z_2 I need θ to give me back my params, and we are able cause they are based on z_1 that is equal to z_1', f is invertible by structure, so I can reconstruct z_2. The key problem here is that z_1 is not transformed, but this is just one step of transformation, **after I can swap** and do the same thing with z_2 that is copied and z_1 that is changed and then swap it again. **NICE - Non-linear Independent Components Estimation**: $z_2' = z_2 + \theta(z_1)$, z_2 plus a displacement/shift, given by a NN, z_2' is a copy shifted by a quantity of z_2, the amount to shift is decided by my NN θ, linear transformation, nicely invertible, there is a source of non linearity hidden in the NN, z_1 is non linearly transformed, but there is the copy that preserves also the original info that makes fully invertible. You just need a NN to predict your θ, you can train with your log(P(x)) seen before. In order to train this model we need the determinant of the Jacobian of our transformation, the Jacobian of our transformation is triangular I only care about the diagonal things. You can stack multiple of this coupled flows, if I stack 2 layer of this coupled flow between them I introduce a **DETERMINISTIC** (to preserve invertibility) **shuffle operation**. Each of this layer is a different Neural layer plus some affine shifting, and each θ has its own param. You need some **more non-linearity**, those are just shifting, do some more articulated in f, θ, the NN, will again produce two set of param in output, one used for a scaling function, all the operation here are pointwise, hadamard product, even the exponential is pointwise, the exponential is to ensure that this thing is positive, and the others are params that operates the shift. θ_B how much you shift z_2, and θ_A how much you scale z_2. Again this is easily invertible and again a very nice diagonal Jaconia, to compute the determinant just product of elem on the diagonal. On the bottom right you have the derivative of your mapping w.r.t. to z2, that is the scaling elements, one for each dim of z2.

Jacobian Matrix of the Affine Coupling Layer

Consider a transformation that splits the input vector into two parts: $\mathbf{z}_1$ and $\mathbf{z}_2$, and defines the output as $\mathbf{z}_1' = \mathbf{z}_1$ (unchanged) and $\mathbf{z}_2' = \exp(\theta_A(\mathbf{z}_1)) \odot \mathbf{z}_2 + \theta_B(\mathbf{z}_1)$, where $\odot$ denotes element-wise multiplication.

The Jacobian of the transformation from $[\mathbf{z}_1, \mathbf{z}_2]$ to $[\mathbf{z}_1', \mathbf{z}_2']$ is a block matrix

of the form:

$$J = \begin{bmatrix} \frac{\partial \mathbf{z}_1'}{\partial \mathbf{z}_1} & \frac{\partial \mathbf{z}_1'}{\partial \mathbf{z}_2} \\ \frac{\partial \mathbf{z}_2'}{\partial \mathbf{z}_1} & \frac{\partial \mathbf{z}_2'}{\partial \mathbf{z}_2} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{z}_2'}{\partial \mathbf{z}_1} & \text{diag}\left(\exp(\theta_A(\mathbf{z}_1))\right) \end{bmatrix}$$

- The top-left block, $\frac{\partial \mathbf{z}_1'}{\partial \mathbf{z}_1} = \mathbf{I}$, is the identity matrix since $\mathbf{z}_1$ is passed unchanged.
- The top-right block, $\frac{\partial \mathbf{z}_1'}{\partial \mathbf{z}_2} = \mathbf{0}$, is a zero matrix because $\mathbf{z}_1'$ does not depend on $\mathbf{z}_2$.
- The bottom-left block, $\frac{\partial \mathbf{z}_2'}{\partial \mathbf{z}_1}$, is generally non-zero since both $\theta_A(\mathbf{z}_1)$ and $\theta_B(\mathbf{z}_1)$ depend on $\mathbf{z}_1$. However, this block is not needed when computing the Jacobian determinant.
- The bottom-right block, $\frac{\partial \mathbf{z}_2'}{\partial \mathbf{z}_2}$, is a diagonal matrix where each diagonal entry is $\exp(\theta_A(\mathbf{z}_1)_i)$, because the transformation applied to each component of $\mathbf{z}_2$ is independent and element-wise.

This triangular Jacobian structure allows for efficient computation of the determinant as the product of diagonal elements, which is essential for normalizing flow models. How this it is implemented? θ it's applied on image, θ is gonna be some sort of CNN, some sort of Rectified convolution, you use 0 padding to preserve size, here you use rectified convolution to compute convolution without losing dim without having to 0 pad. Use residual connection and batch normalization, all easily invertible stuff.

Conditional Transformation in Coupling Layers

In flow-based models, particularly in affine coupling layers, we often split the input vector $\mathbf{z}$ into two parts:

$$\mathbf{z} = [\mathbf{z}_1, \mathbf{z}_2]$$

The transformation is applied only to $\mathbf{z}_2$, conditioned on $\mathbf{z}_1$. This is represented as:

$$\mathbf{z}_2' = \mathbf{f}(\mathbf{z}_2; \theta(\mathbf{z}_1))$$

where $\theta(\mathbf{z}_1)$ parametrizes the transformation of $\mathbf{z}_2$. The final output is:

$$\mathbf{z}' = [\mathbf{z}_1, \mathbf{z}_2']$$

Components of the Transformation $\mathbf{f}(\cdot)$:

- **Rectified Convolutions:** Convolutional layers followed by non-linear activations (e.g., ReLU). The convolutions are *size-preserving* and use *no zero-padding*, keeping the spatial structure intact.
- **Residual Connections:** The output of the convolutional block is added to its input, helping the model learn identity mappings and improving gradient flow:

$$\text{output} = \text{ConvBlock}(\mathbf{x}) + \mathbf{x}$$

- **Batch Normalization:** Normalizes activations across the batch to stabilize training, ensuring mean-zero and unit variance for each feature channel.

These design choices make the function $\mathbf{f}$ both expressive and invertible — essential properties for flow-based generative models.
Multiscale Flows. To get a richer flow without loosing invertibility, rather than operating on the full stack of my data all together, at every step, I can focalize on a certain piece of info, when you implement the flow, you implement it on things that have the size of the original data, if your original data is an image a lot of multiplication /processing going on, you might want to be smarter and to this thing in an incremental way → **Multiscale flows**: you focus on a piece of your vector and you copy the rest, base density vector size of the image, at the first flow layer you only focus z_1, you transform only z_1 and you copy z_2, and you'll get a new transformation with z_1 changed and z_2 copied and you have not touch the rest of the vector, now you transform this whole part here you have just produced and you copy z_3 from before, now you transform all of these 3 blocks and you copy z_4 and now your size of the full sample is ok. **Every time you implement a coupling layer on a subset of your vector**. In order to implement this on image you need a mechanism to mask which part of the big vector z you are considering in each coupling layer and in which not. The efficient way of doing that is by having mask that tells you which part of your big sized vector you are considering and which not, **split the image in a black and white-checkerboard pattern**, copy the white and modify the black. **RealNVP** not only introduces this mask to take the spatial info into account, but also I am transforming my image every now and then by an operation called **squeezing**, completely invertible, that does not at all change the semantic meaning of an image but shuffle a little bit how you put together the pixels in order to compute your info, after you have done your flow on the checkerboard pattern you can take the matrix and transform an s x s C, where C is num of channel, matrix and squeezing into the equivalent s/2 x s/2 x 4C channels, I am rearranging a matrix into 4 different matrices in which the first and the fifth pixels becomes the first and second pixel in this first channel and the 9 and 11 pixels are down here, rearranged my initial matrix into a tensor, and now I am gonna be transforming one and copy another and then I am gonna be rearranging them again into an image , apply another mask, all to mix the the info and split the structure of the image in a crazy way.

RealNVP – Masking and Squeezing
RealNVP (Dinh et al., ICLR 2017) introduces two key techniques to build expressive yet tractable invertible models: **masking** and **squeezing**.

Affine Coupling Layer with Masking In RealNVP, the input vector $\mathbf{z}$ is split via a binary mask $\mathbf{b} \in \{0,1\}^d$:

$$\mathbf{z}' = \mathbf{b} \odot \mathbf{z} + (1 - \mathbf{b}) \odot (\exp(\theta_A(\mathbf{b} \odot \mathbf{z})) \odot \mathbf{z} + \theta_B(\mathbf{b} \odot \mathbf{z}))$$

- $\mathbf{b}$ is a binary mask selecting part of the input to remain unchanged.

- The transformation on the other part is conditioned only on the masked (unchanged) portion.
- Elementwise operations are used to preserve invertibility.

Checkerboard and Channel-Wise Masking

- RealNVP alternates **pixel-wise masking** (checkerboard pattern) and **channel-wise masking** to maximize the dependency coverage.
- Pixel-wise masking is applied before squeezing.
- Channel-wise masking is applied after squeezing.

Squeezing Operation

- Squeezing reshapes a tensor from shape (s, s, c) to $(s/2, s/2, 4c)$.
- This transformation brings spatially adjacent pixels into the channel dimension.
- It improves the coupling layer's ability to model local dependencies.

Multiscale Architecture RealNVP's full model stacks multiple masked coupling layers and interleaves them with squeezing steps. This multiscale structure increases expressiveness while maintaining tractable inverse and Jacobian computations. **GLOW** introduces a way to avoid all these mixing, permutation, squeezing, instead of having all this channel wise restructuring why don't we use good old **1-dim conv, full learnable**, **through 1 dim convolution you can implement any possible random permutation of the pixel**, you bypass the squeezing part in which you organize thing in a channel-wise matrix. And instead of using batch normalization they use actnormalization, that don't need a batch but a single sample, just a linear rescaling. Only thing you have to be smart in the way you implement your invertible one by one convolution, it will turn out you will have to compute the determinant of the Jacobian at some point, make sure that your W factorizes efficiently with an LU decomposition so the determ. can be computed efficiently. **GLOW – Multiscale Coupling Flow with Invertible 1×1 Convolutions**

GLOW (Kingma and Dhariwal, 2018) extends RealNVP by introducing **invertible 1×1 convolutions** for channel mixing. This allows for more flexible permutation of channels across coupling layers.

Affine Coupling Layer As in RealNVP, GLOW uses an affine transformation of part of the input:

$$\mathbf{z}_2' = \exp\left(\theta_A(\mathbf{z}_1)\right) \odot \mathbf{z}_2 + \theta_B(\mathbf{z}_1)$$

- The input $\mathbf{z}$ is split into $\mathbf{z}_1$ and $\mathbf{z}_2$ along the channel axis.

- $\mathbf{z}_1$ conditions the transformation of $\mathbf{z}_2$ through two neural networks θ_A, θ_B.
- This transformation is invertible and easy to compute.

Invertible 1×1 Convolutions To increase channel mixing and flexibility:

- A 1×1 convolution is applied across all channels.
- This acts as a learned permutation over channels.
- The convolution weight matrix $\mathbf{W}$ is parameterized via LU decomposition for efficient determinant computation:

$$\mathbf{W} = \mathbf{PLU}$$

- The log-determinant of $\mathbf{W}$ can be computed efficiently using:

$$\log |\det(\mathbf{W})| = \sum_i \log |u_{ii}|$$

  where u_{ii} are the diagonal elements of $\mathbf{U}$.

Multiscale Architecture

- Start with an RGB tensor of shape $(H, W, 3)$.
- Apply a **periodic squeezing operation** to reduce spatial size and increase channels.
- Alternate between coupling layers and invertible convolutions.
- At each scale, some channels are factored out (split and stored), forming a hierarchical latent structure.

GLOW's design improves expressivity, invertibility, and training efficiency, while retaining exact log-likelihood computation. In GLOW in particular in RealNVP we have seen that we have this weird thing in which we operate on splitting an image on a pixel by pixel basis, with a checkerboard pattern and then on a channel by channel base, where the way in which you arrange the channel where sort of permuting with random or predefined permutation, GLOW does not care about the checkerboard pattern, no split in the pixel spatial dimension, you split only the channels, and the channel are not permuted explicitly we are using invertible one by one convolution as a sort of generalization of permutation of channel, **1by1 convolution operates only on the channel dimension**, don't alter the dim of an image you operate on a 1 by 1 pixels you operate only on the channel and if you have the same number of filters as the original number of channel you are not changing the original image. With normalizing flows you can never alter the dim of your representation can only transform in a noisier or less noisier version but you cannot shrink or enlarge, cause they are

not invertible, 1x1 convolutions are generalization of permutation over channel, especially when you express the weight matrices as LU factorization, L is the permutation matrix. **GLOW is essential your classical friend affine coupling layer in which you are only splitting between channel, z1 and z2 are just split of the different channel of the images, 1by1 convolution play the role that before was of the shuffling of the channel**. actnorm like batch normalization but don't require a batch. Put this GLOW layer in between usual stages of a normalizing flows. You can take an original picture and find in which z_0 embedding it is mapped in the space of the base density (normal space), you do the same for another guy you obtain its z_0' and then you interpolate between this 2 vector and you start decoding using the opposite flow. Celebrity dataset with auxiliary information, smiling, hair color, pale skin,...you can take one of those auxiliary info, extract all the images that are associated with blond hair, make the average image, project this average image in the z_0 space, do the same with black hair, and then take an image and move it in the direction of more blonde or more black color ...when operate on the z_0 space nice gaussianly organized space operation between vectors there tends to be smooth

Autoregressive flow: generalization of pixel RNN/pixel CNN, one way in which you can compute your flow is by an autoregressive decomposition, z_1 is copied or manipulated slightly little through an invertible function f, z_1 in parallel is used to generate a param to alter z_2 , and z_1 together with the original z_2 is used to generate the params to alter z_3 ecc... any transformation to a partition z_n in your original vector z is conditioned on the info on all the subpartition from z_1 to z_{n-1} , classical autoregressive setting in which we model $P(z_n|z_1...z_{n-1})$, autoregressive but at the same time can be computed in parallel, because the second transformation e.g. depends on z_1 not on the transformation of z_1, all these normalizing pass can be computed in parallel. Problem: the backward direction is not parallel is sequential.

Autoregressive Flows Generalization of coupling flows that treats each input dimension as a separate block: Forward and inverse directions have different costs (parallel/sequential).

MASKED autoregressive flows: when I want to generate z_i', the i-th partition that I am generating as new value of z_i in the flow, I can do that with my usual invertible function f, which works on z_i, original piece of data that I want to transform through the flow, and some params. that comes from all the previous partition of my information from 1 to i-1, but here we have x not z, we are conditioning on the actual data, not the transformation that I have perfomed on my data. Similar to what we have in pixel RNN $P(x_i|x_1, ..., x_{i-1})$ sort of a generalization of autoregressive model, but you always have the data here, if you go in the normalizing direction no problem you have the data and you arrive till the base density z_0, if you go in the opposite way the sampling direction from z_0 to data space you don't have x, or you have x but you first need to decode x for the first partition, x for the second partition, x for the third partition, before you can decode the x for the fourth partition, one direction fully parallel the other one is sequential. **Masked autoregressive flow**

is gonna be very efficient at training time (normalize flow) but not so efficient at sampling time, probably I'll use this kind of model more for density estimation than for sampling. This function here is a probabilistic function, I am transforming z_i into z_i' through a function f that depends on the current value of z_i and some params that comes from $x_{1:i-1}$ and these param are μ_i and s_i, params of a gaussian, z_i' is generated by drawing it from a gaussian with mean μ_i and variance $exp(s_i)^2$. Reparameterization trick: z_i' = μ_i + ϵ_i $exp(s_i)$ where ϵ_i is drawn from a normal. Fully probabilistic transformation of my model and this flow it is essentially implementing a transformation from the space of random vector ϵ to data x, generalization of autoregressive modeling in pixel RNN, inversion not difficult can be easily inverted and the determinant is just the diagonal of the Jacobian wich is just the exponential of the si.
Called masked autoregressive flow cause this sort of autoregressive computation that you have here, in which not all of the input influence not all of the output, only a subset of the input are used for a subset of the output in order to mantain a causal dependency can be implemented easily, if you assume that this sort of causal computation can be implemented as a dense computation but where you are masking some of the connection. MADE masked autoencoder, if you want to implement a causal or autoregressive computation in a specific layer h this h(x) the vector that contains all the activations for all the input from 1 to n, is gonna be your usual non linearity, your bias plus your weights not directly multiplied by x as you would do in autoregressive, but are before multiplied by a mask, that tells which component contribute to which neuron, to keep causal relationships. **Masked Autoregressive Flow (MAF)**

Masked Autoregressive Flow (MAF) defines a flow model using autoregressive transformations, where each output variable depends only on previous input dimensions. It leverages autoregressive models to build invertible transformations with tractable Jacobians.

Transformation Function Given a latent variable $\mathbf{z}$, MAF defines a mapping $\mathbf{x} = f(\mathbf{z})$ using autoregressive functions:

$$x_i = \mu_i(\mathbf{x}_{1:i-1}) + z_i \cdot \exp(s_i(\mathbf{x}_{1:i-1})) \quad \text{where } z_i \sim \mathcal{N}(0, 1)$$

Here:

- μ_i and s_i are functions of the previous outputs $\mathbf{x}_{1:i-1}$.
- This makes f autoregressive in $\mathbf{x}$, but still invertible.

Density of x Since z_i is standard normal, the conditional distribution of x_i is:

$$p(x_i \mid \mathbf{x}_{1:i-1}) = \mathcal{N}\left(\mu_i, \exp(s_i)^2\right)$$

Invertibility The transformation is easily invertible (used during training), since we can compute z_i from x_i as:

$$z_i = (x_i - \mu_i) \cdot \exp(-s_i)$$

Jacobian and Determinant The Jacobian of f^{-1} is lower triangular because each z_i only depends on x_i and $\mathbf{x}_{1:i-1}$. The determinant of the Jacobian is the product of the diagonal terms:

$$\left|\det\left(\frac{\partial f^{-1}}{\partial \mathbf{x}}\right)\right| = \prod_i \exp(-s_i) = \exp\left(-\sum_i s_i\right)$$

Advantages

- Tractable and efficient log-likelihood computation.
- Invertibility enables sampling from base Gaussian by passing through inverse flow.
- Triangular Jacobian simplifies determinant calculation.