Neural Networks

Introduction

NN are everywhere in real life and most applications use them (speech recognition, traffic analysis, object detection, energy forecasting, music generation, autonomous vehicles, smart manufacturing, IoT applications, AI accelerators, intelligent surveillance, healthcare, scientific discovery…).

Neural networks are:

- **scalable**: from very few number of parameters to very high number of parameters (some of them are learnable, some others not)
- **composable**: inside the “black box” we have different modules
- **differentiable**: the gradient operator is used to learn and update parameters
- **optimizable end to end**

Inputs can be of any type, as well as outputs. Depending on the input type and goal to be achieved, a model is chosen.

Learning approaches:

- **supervised learning**:
  - based on data learning with which we infer knowledge in the model
  - task: regression/classification
  - the supervisor compares the exact output (labels) and the predicted output in order to adjust the parameters and make that distance close to 0 and reach the optimum
- **unsupervised learning**:
  - data contains no labels (very common situation in real world)
  - task: group/cluster of data, dimensionality reduction, self-supervised regression, self-supervised classification
  - the supervisor doesn’t exist
- **semi-supervised learning:**
  - dataset with some labels (not all labels)
- **reinforcement learning:**
  - unknown process which produces a state and a reward and based on these two components the rl algorithm decides an action
- **self supervised learning:**
  - doesn’t require labels, but we want to provide them
  - recent research

Elements of linear algebra

Linear algebra recap (useful to understand how data is represented in NN)
Linear algebra provides rules for manipulating vectors and matrices.

- scalars: physical quantities representing a number
- vectors: sequence of n scalars representable in n-dimensional space
- matrix: sequence of vectors with a number of rows and columns
- tensor: arrangement of a set of matrices

Important operations with matrices used in NN are multiplication and sum.
The sum is defined if the matrices have the same shape.
The multiplication is defined if the number of columns of the first matrix is equal to the number of rows of the second matrix.

Important operations: kronecker product, transpose, hermitian, inverse, pseudo-inverse, inner and outer product…
Important types of matrices: diagonal, semi-positive, symmetric, toeplitz matrix, projection matrix, ...
Properties: rank, determinant, trace, symmetry, …

In a simple NN the inputs are neurons associated with a scalar (weights).
Inputs and weights are represented in vector forms **x** and **w** and an inner product is applied to **x** and **w**. This process is called forward pass and returns a scalar.
In order to compare the output to some value (if the label is available) then a measurement is needed. So also the geometry distances and norms are really useful in NN theory.

Optimization Methods for Neural Networks

Intro of the lesson:
To optimize the learner algorithm we want to automatically tune the parameters in order to achieve the results we want.
In the training we start with 0 knowledge and we learn a parameterized function from available data. In this phase we have to update parameters by using an operator, the gradient. Then we stop the training when some kind of convergence is reached. So we need to define a goal.
After the training we use the information we got to test them on new unseen data.

An optimization problem is a minimisation or maximization problem. (A minimisation can be seen as a maximization). What we minimize/maximize is a cost function defined to be chosen according to data and task.
The optimization problem can be both unconstrained and constrained.
The distinction between cost function and loss is that the loss quantifies the distance between a prediction and real label of a particular sample, while the cost function is the sum over all samples of that loss.
The loss is a non negative function.
Our objective is to approximate as best as possible the function the learner algorithm has output. Also the loss function (and so the cost function) is parameterized by the parameters we have to update in order to reach the goal. Ideally we want the loss going to zero. In general, we want to arrive at the minimum of the cost function.

It is for instance useful to use convex or concave functions, since their minimum or maximum can be easily obtained by computing their gradients and put them equal to zero.
Sometimes we reach the global minimum, other times we reach the local minimum.

Some notations:

Convex function:
f is convex if

$$f(\lambda x1)+(1-\lambda)\, x2) \leq \lambda f(x1) + (1-\lambda)\, f(x2),\ \forall \lambda \in [0, 1]$$

Strictly convex function:
f is strictly convex if $\forall x1 != x2 \in R^n,\ \forall \lambda \in [0, 1]$:

$$f(\lambda x1 + (1-\lambda)\, x2) < \lambda f(x1) + (1-\lambda)\, f(x2)$$

Strongly convex function:
f is strongly convex if if $\forall x1, x2 \in R^n,\ \forall m > 0, \lambda \in [0, 1]$:

$$f(\lambda x1 + (1-\lambda)\, x2) \leq \lambda f(x1) + (1-\lambda)\, f(x2) - 1/2m\lambda\, (1-\lambda)\, ||x1 - x2||^2$$

Concave function:
f is concave if

$$f(\lambda x1 + (1-\lambda)\, x2) \geq \lambda f(x1) + (1-\lambda)\, f(x2),\ \forall \lambda \in [0, 1]$$

Strictly Concave function:
f is strictly concave if $\forall x1 != x2 \in R^n,\ \forall \lambda \in [0, 1]$:

$$f(\lambda x1 + (1-\lambda)\, x2) > \lambda f(x1) + (1-\lambda)\, f(x2)$$

Global minimum:
A point θ* is a global minimum for function J (θ) if:
$J(\theta^*) \leq J(\theta),\ \forall \theta \in R^M$
It is a local minimum if this formula holds only for an ε-radious ball centered in θ∗. It is a strict minimizer if this formula holds without equality.

Gradient: it's a vector containing the partial derivatives of the function on which it is defined.

Hessian Matrix: it contains the partial derivatives of gradient

Stationary point:
A point θ* is a stationary point of J (θ) if:
$\nabla J(\theta^*) = 0$

A matrix is Positive semidefinite matrix if: $a^T S a >= 0,\ \forall a \in R^M$

Theorem 1: necessary optimality conditions
If a point θ∗ is a local minimum then it is a stationary point and the Hessian matrix evaluated at θ∗ is positive semidefinite.

Theorem 2 (Sufficient optimality conditions):
If a point θ∗ ∈ R^M is a stationary point and the Hessian matrix evaluated at θ∗ is positive definite, then θ∗ is a strict local minimum.

Least square cost function:
It is a very common cost function since it has very important properties for this kind of application. In fact:

- It has a unique solution
- Its optimal parameters are obtained by solving a linear system of equations

The loss function is:

$$(y - f_\theta(x))^2$$

So the cost function formula is:

$$J(\theta) = \Sigma_{x_k} (y_k - f_\theta(x_k))^2$$

Examples of optimization algorithms are: gradient descent, stochastic gradient descent, mini-batch gradient descent, Adam, RMSprop, …

Linear models for regression and classification

Linear models are based on linear neural networks.

Definition of supervised dataset: set of n known examples with all the information we need, each sample is a pair, where the first element is the input to the NN x and the second is the output that the NN should predict y.
We want our prediction f to correctly predict the output given a new unseen example (x, y), so f(x) - y -> 0.

Constraints on the dataset
The dataset on which we train is the training dataset and the dataset on which we test is the test dataset. Their intersection is the empty set.
The test set is used to evaluate the model, so as to understand how good it is.
All the examples of both training and test sets are assumed to be i.i.d. so they follow the same stable distribution (identical distribution) and they are independent (no bias in the distribution).

Definition of loss function: We need to define a function (loss function) evaluating the prediction of f, so it takes 2 arguments, the estimation and the real output. Low values mean good performance, high values mean bad performance.
We use it to optimize the performances through a minimum (loss) or maximum (-loss) optimization problem.
The kind of loss is chosen according to the task.

The **expected loss** (risk) of a function f is:

$$f^*(x) = E_{p(x,y)}[l(y, f(x))]$$

This risk is uncomputable, so we proceed with the **empirical risk minimization**:

$$f^*(x) = 1/n \, \Sigma_{i=1,\dots,n} l(y_i, f(x_i))$$

It is called empirical since it uses data, risk is a synonym of loss, minimization because we want to minimize a loss.

Overfitting comes when the training error is likely to be 0 (works well with known data), but a lot of errors occur in the test set.
So we'd like to have a model learning not directly data points, but a more general approach.
When training deep neural networks, then it is more likely that overfitting happens.

To train a neural network, we need:
- input x: data
- loss function depending on the task
- f which depends on x (input) -> designing f is the most difficult part

Let's consider:
- the input as a vector of shape d containing some characteristics of the problem described
- the output is a real number

Depending on the output we distinguish between regression if y can take any values and classification if y can take only c (number of classes) possible numbers.

How to build a loss function?
Two main approaches:
- empirical: what could be a valid loss function based on data and design it by hand; reasonable but not perfect (e.g. square loss, absolute loss, squared log loss)
- probabilistic: given that our data is not perfect, the learner can make mistakes, so it is better to deal with probability (so some variance is introduced for uncertainty)

Sometimes it is easier the first, some others the second approach.
Probabilities have some good properties to be exploited.

Maximum likelihood probability (probability assigned by the model to the data):

$$f^*(x) = \text{argmax } \Pi_{(x_i, y_i)} p'(y_i|f(x_i)) \;=\; argmin\, \Sigma_{(x_i,y_i)} - \log p'(y_i|f(x_i))$$

Each sample (xi, yi) has a probability and since they're iid we can multiply them to get the global probability.
Since multiplication of small numbers is not good, then we use the sum on the logarithm of the probability.
A good f maximizes this likelihood.

Linear model
The easiest form of NN (which will be the unit of more difficult NN) is linear.
An input x has shape d and to each feature xi is given a parameter wi. This weight w_i part of a vector of the same shape as x.
f(x) is defined as the dot product between x and w plus a term called bias.

$$f(x) \;=\; w^T x \;+\; b$$

These weight and bias parameters will be updated during the optimization problem.

Least-squares optimization problem:

$$LS(w) = 1/n \; ||y - Xw||^2$$

linear model with a squared loss. least because we are looking for w having the lowest amount of squared error.
Easy to use and with good properties, but limited wrt the *linear* result it produces.

- Solving the LS problem:
  Least Square is a convex problem, so it can be solved in a closed form. We can also compute the gradient:
  ∇LS(w) = 2/n $(X^T$ (Xw − y)).
  We have to solve:
  ∇LS(w) = 0 which solution is w* = $X^T(XX^T)^{-1}y$
- Regularizing the LS problem:
  We might need to add a small amount of L2 regularization since they may cause problems in the inverse of XX^T.
  LS-REG(w) = LS(w) + λ/2 // w||^2 for some λ > 0. This makes the problem strictly convex and forces the solution to be contained in a ball of given radius, modifying the gradient and the explicit solution as:
  ∇LS-REG(w) = ∇LS(w)+λw ,
  w* = $(X^TX + \lambda I)^{-1}X^Ty$

Matrix multiplications have less complexity wrt inversion of matrices.

Classification tasks
In classification y takes a fixed set of values called classes. Biggest problems in NN were about classification. These classes are independent, the order doesn't matter.
We also need some thresholding in order to get integers, but we need to guarantee differentiability. We can also think of it as a prediction of probability distributions over the classes and we need to ensure that the output of a NN is always in the probability simplex.

Softmax probability: it can be seen as an approximation of the argmax, in fact its more accurate name is soft argmax. It is a transformation in which the input is a vector and the output is a vector with components between 0 and 1.

$$[softmax(a_i)]_i = exp(a_i) / \Sigma_j exp(a_j)$$

It can also be used as a hyperparameter, the temperature, used as the denominator of ai and aj.
Our linear model for classification becomes:
f(x) = softmax(Wx + b)

One-hot encoding
In order to compare the prediction with the ground-truth, we encode our target with the one-hot encoding. E.G. if we have 3 classes, cat, dog, bird, we have [1, 0, 0] for cat class, [0, 1, 0] for dog class and [0,0, 1] for bird class.
Formally, for any given pair (x,y), yi = 1 if x is of class i, else 0.

Cross-Entropy Loss:

$$CE(y, y') = - \Sigma i (yi \log (y'i))$$

A loss function used for classification tasks is the cross entropy loss, which provides a measure of distance between two probability distributions (the real one and the one predicted).

Only one term in the sum is not zero. We want to minimize this sum, so we are maximizing this probability: the network should take the probability corresponding to the true class and maximize it (push it as possible to 1), at the expense of all other outputs.

Log-likelihood loss

$$CE(y, \hat{y}) = -\log(\hat{y}t).$$

Logistic regression

It is a linear model for classification (not regression as the name says) using the softmax trained by optimizing the cross entropy loss. It is not possible to solve the logistic regression problem explicitly.

$$LR(W) = 1/n \, \Sigma_{i=1,..,n} CE(y_{i,}, f(x_i))$$

Binary Classification

It is a kind of classification, where the number of classes is 2. In this case we can predict a single value from f(x), 0 or 1.

f(x) is probability of class 1, 1-f(x) is probability of class 2.

To transform any real value to another one between 0 and 1 we typically use the sigmoid function, defined as σ(s) = 1/(1+exp(-s)).

Binary logistic regression model

$$\text{BIN-LR(w)} = 1/n \, \Sigma_i - y_i log\sigma(w^T x) - (1 - y_i) log(1 - \sigma(w^T x))$$

In this case, we can obtain the most probable class from the model as:

class = 1 if $\sigma(w^\top x) > 0.5$, 0 otherwise.

The differentiation of σ(s) is σ′(s) = σ(s)(1 − σ(s)).

So:

$$\nabla BIN \; LR(w) = 1/n \, \Sigma_{i=1,..,n} (\sigma(w^T x_i) - y_i) x_i$$

Logsumexp function

It is the logarithm of the sum of the exponentials: $-pi + \log \Sigma_i exp \, p_i$,

Calibration

When the model satisfies: $P(y = i|x) = f(x)_i$, then we call it calibrated.

In the condition,

- $f(x)_i$ is the result of the neural network,
- $P(y = i|x)$ is the probability of the pattern x to be classified as class i

In general, it is not true that $P(y = i|x) = f(x)_i$. So we have to check for that. To measure the calibration of a model we use the expected calibration error which formula is:

$$ECE = \Sigma_m B_m/n \; |a_m - p_m|, \quad \text{where:}$$

- m is the number of bins in an interval [0,1]
- Bm is the number of samples from the validation set, whose predicted confidence falls in bin m.
- pm is the average confidence of the model for that bin

- am is the average accuracy of the model for that bin

Typically, deeper neural networks are more confident and the value of confidence is far from that of accuracy. Instead, small models tend to be calibrated.

If the model is calibrated, we can exploit this fact according to the task and application we're designing a solution for. (ex. medical diagnosis)

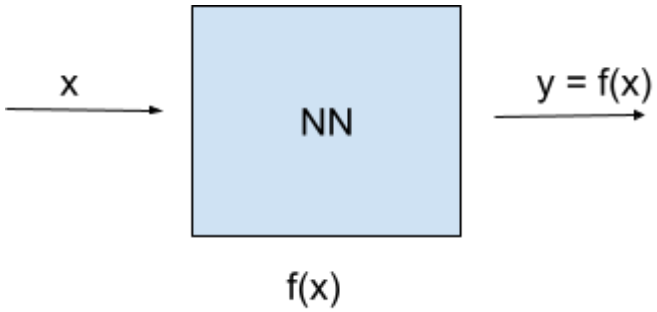
We can adjust the calibration of a model with some techniques, one of them is the focal loss.
−> So, we train a model, we test if the calibration error is not what we expect, and then we change the loss.
The focal loss puts more focus on hard misclassified examples.
Its formula is:

$$FL_{\alpha}(y, y') = -(1 - y_c')^{\alpha} log y_c'$$

where c = arg max y.

Shallow Neural Network

Intro:



Building a nn model requires some steps:

1) define the family of function to use (e.g. linear / non linear / ML)
2) define the loss function L in order to compare x and f(x)
3) Compute the gradient of the loss function in order to update the parameters

How to choose the best model? We have to know what the "box" NN does.

The "problem" of shallow neural networks comes since most applications require high dimensional space.
Linear models (linear space) are not sufficient to model data and lead to poor performance.
Functions highly nonlinear can approximate really complex data distribution. Also neural networks are nonlinear.

Generalized linear models:

Given (y, x) ∈ RxR^l , a generalized linear estimator y' of y has the form:

$$y' = f(x) = \Sigma_{k=1,..,K} w_k \Phi_k(x) + \theta_0$$

where Φ_k are preselected nonlinear functions

Cover's theorem

Let us consider N points, x1, x2, . . . , xN ∈ R^l, in general position, i.e., there is no subset of l + 1 of them lying on a (l − 1)-dimensional hyperplane.
The number of groupings, denoted as O (N, l), that can be formed by (l − 1)-dimensional hyperplanes to separate the N points in two classes, exploiting all possible combinations, is given by:

$$O(N, l) = 2 \Sigma_{i=0,\ldots,l} (N-1)! / (N-1-i)!\, i!$$

Each one of these groupings in two classes is also known as a linear dichotomy.
If N ≤ l + 1, then $O(N, l) = 2^N$.

Based on the previous theorem, given N points in R^l, the probability of grouping these points in two linearly separable classes is:

$$P = O(N, l)/2^N$$

Mapping in a higher-dimensional space

In order to exploit the previous theorem we consider N feature vectors xn ∈ R^l, n = 1, 2, ..., N, so that the following mapping is performed:

$$\phi : x_n \in R^l \rightarrow \phi(x_n) \in R^K,\ K >> l$$

The higher the value of K is the higher the probability becomes for the images of the mapping, φ (xn) ∈ R^K, with n = 1, 2, . . . , N, to be linearly separable in R^K.

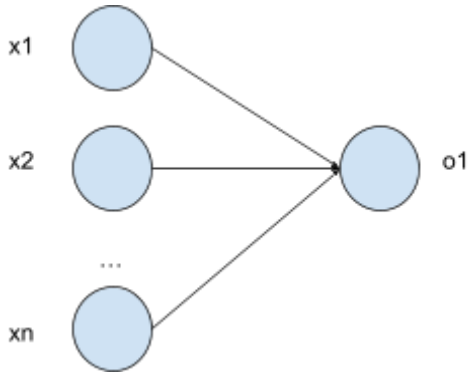
Modeling complex tasks with deep learning

Deep learning can be understood as a subfield of machine learning that is concerned with training models based on artificial neural networks (NNs) with many layers efficiently.
The basic concept behind artificial NNs was built upon hypotheses and models of how the human brain works to solve complex problems.

Shallow learning

Shallow learning is characterized by a single level of processing, which often yields an output derived from an inner product between the weights and the input, e.g.: y=Wx
Shallow models are linear classifier, regression filters, PCA and basic neural networks,...

Single-layer neural networks (fully connected layer ->each neuron is connected with the following in the next layer)



Multilayer perceptron
We can extend this structure to multiple outputs, but also to multiple intermediate layers: the hidden layers.

The hidden layers are obtained by stacking many fully connected layers on top of each other.

Each hidden layer has output h = W1x + b1 -> linear operation
Each output layer has output o = W2h + b2 -> linear operation
The final classifier is given by y = softmax(o)

Activation functions
They are differentiable operators used to introduce non-linearity in NN and to activate some neurons.

The most common activation functions are:

1) ReLU:
   - relu(x) = max(x,0)
   - relu'(x) = 1 if x>0, 0 if x<0
2) Sigmoid:
   - $sigmoid(x) = 1/1 + exp(-x)$
   - $sigmoid'(x) = sigmoid(x)\,(1 - sigmoid(x))$
3) Hyperbolic tangent:
   - tanh(x) = $1 - exp(-2x) / 1 + exp(-2x)$
   - tanh'(x) = 1 - $tanh^2(x)$

Automatic Differentiation

> backpropagation

We represent inputs X, weights W and outputs H as vectors for simplicity.
In a sequence of n layers:
$h1 = f(xw1),\ h2 = f(h1w2),\ \ldots,\ h_n = f(h_{n-1}w_n),\ y = < h_n, 1 >$

We have to compute gradients, so Jacobian matrices, in an efficient way.

Chain rule of Jacobian
$\partial[f\ o\ g] = \partial f\ o\ \partial g$

Forward mode differentiation (1964)

More formally, for each layer in the network, forward-mode autodiff proceeds as follows:
1) Compute the new output $h_i = f_i(h_{i-1}, w_i)$.
2) For wi, initialize the so-called tangent matrix:

$W_i' = \partial_{w_i} h_i$

Some layers might not have parameters, in which case skip this step.
3) For previous parameters, update their gradient using the chain rule:

$W_j' = [\partial_{h_{i-1}} h_i] W'_j$, j<i

This technique is cheap in terms of memory (it doesn't require memorization), but high-consuming in terms of time.

Reverse mode differentiation (1976)

1) Compute the output of all layers, storing each intermediate value. Set h' = 1.
2) Going in reverse, i = l, l − 1, · · · , 1, compute the gradient of the parameters of the current layer:

$\nabla_{w_i} y = h[\partial_{w_i}{}^T h_i] h'$

3) Update the gradient of y with respect to h_{i-1} exploiting again the chain rule:

$h' = [\partial_{h_{i-1}}{}^T h_i] h'$

It is cheap in terms of time, but expensive in terms of memory.
This is explainable since we typically have more memory resources instead of time.
Reverse mode in neural networks (1982).

Typically, instead of using Jacobian matrices, we use vector-Jacobian products:

$$vjp_x(f, v) = \partial^T{}_x f(x, w) \cdot v$$
$$vjp_x(f, v) = \partial^T{}_w f(x, w) \cdot v$$

How do we choose non-linearity? (Activation-functions)
Whenever a value goes through an activation function, during backpropagation we multiply by the derivative of the function. If we have many layers, we make a lot of these multiplications.
The result of this chain of multiplications can lead to 2 different but related problems: vanishing gradient and exploding gradient.

A very common choice for deep networks is the rectified linear unit (ReLU), defined as: φ(s) = max (0, s) .
Its derivative is either 1 (whenever s > 0), or 0 otherwise.
(The ReLU is not differentiable for s = 0, but this can be easily taken care of with the notion of subgradients).

Subderivatives:
The subderivative of a convex function $f : R \rightarrow R$ at a point x is a point g such that: $f(z) - f(x) \geq g(z - x) \ \forall z \in R$.

Variants of relu:

1) **Leaky-ReLU:** replaces the negative quadrant of ReLU with a small slope α which is a trainable parameter
   LR(s) = s if s>0, LR(s) = αs if s<=0
2) **exponential linear unit:** a smoother version of the ReLU
   ELU(s) = s if s>0, ELU(s) = α(exp s - 1) if s<=0
3) **Gaussian Exponential Linear Unit (GELU)**
4) **Sigmoid linear unit (SiLU) or Swish**

Handling large datasets: Stochastic Gradient Descent

When the dataset is too large, computing each output and each gradient on all examples can become infeasible. So we use a stochastic version of GD, the stochastic gradient descent algorithm, where we use only a mini-batch B of M examples from the full dataset to compute the output (and consequently update weights). The computational complexity of an iteration of SGD is fixed with respect to M (the batch size) and does not depend on the size of the dataset.
Because we assumed that samples are i.i.d., we can prove SGD also converges to a stationary point in average, albeit with noisy steps.

Instead of randomly sampling batches from the training dataset at each iteration, we generally apply the following procedure:

1. Shuffle the full dataset;
2. Split the dataset into blocks of M elements and process them sequentially, batch-by-batch;
3. After the last block, return to point (1) and iterate.

A full pass over the dataset is called an epoch.

Running SGD requires selecting the size of the mini-batch, which is an additional hyperparameter:

- **Smaller mini-batches** are faster, but the network might require more iterations to converge because the gradient is noisier.
- **Larger mini-batches** provide a more reliable estimation of the gradient, but are slower.

In general, it is typical to choose power-of-two sizes (32, 64, 128, ...) depending on the hardware configuration and the total memory available.

Neural Networks for Images and Spatial Information

Intro:
NNs are widely used for their favorable capabilities and their ability to approximate a natural behavior with an artificial model. This is complex because lots of uncontrollable factors can affect this behavior.
What we know about NN is the initial state and the output state, that's why we define it black box. But, in reality, given a task and a set of data we also decide which model / function to use, the number of layers,...

Going beyond multilayer perceptron
When the problem to solve is complex we should address deeper architectures. The complexity of a model is proportional to its deepness (and also to data). Complex systems require efficient training on a huge amount of data.
So we have to pass from a multilayer perceptron model to a neural network specialized in dealing with such problems. In this chapter, we will focus on images, tensors of matrices, and we want to exploit their spatial information.
In general, an image can contain objects in any of its parts, so we can focus only on the interested region where the object is, this is why we talk about "spatial information": nearby pixels are strongly correlated.
Few layers -> low level features
Lots of layers -> high level features

From dense layers to convolution
Dense layers (FCL) can lead to billions of parameters and this is actually a problem when the input size is huge, as in images. If we have a 28x28 image, then we need 784 neurons input to pass to hidden layers. 784 is the size of the vector obtained by taking each pixel of the matrix row by row or column by column (the order in NN doesn't matter). The training in such networks becomes hard for 2 main reasons:

- the computation of inverse matrices has highly cost
- backpropagation-related algorithm may often stuck in local minima
- vanishing/exploding gradient problem
- less generalization: adding neurons and layers makes the model too complex, so worse in generalization

A thing we can do is object detection:

1) just consider relevant pixels (e.g. all with 0 values don't give any information: discard them)
2) look at the object for which we have designed a object detection model

So we should overcome these problems and use something able to exploit properties of images. For instance, we introduce an "inductive bias" consisting of a form of prior knowledge of data.
From Image theory processing, we know that images have 2 important properties:

- translation invariance: an object in an image can appear in any part, but its information remain the same
- locality: an object in an image should depend on near pixels (belonging to the small region where it is located)

Review of the convolution operation

The convolution can be defined both for continuous and discrete functions.
Continuous case:
Given two functions $f: R^d \rightarrow R$ and $g: R^d \rightarrow R$, the convolution $f \star g$ is defined as:

$$[f \star g](t) = \int_{R^d} f(\tau) g(t - \tau)\, d\tau$$

It measures the overlap between f and g with both f and g shifted by τ and flipped.
Discrete case:

$$< f[n - k] g[n] > = \sum_{R^d} f[n - k] g[n]$$

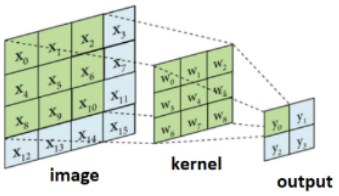
$$< f[n] g[n - k] > = \sum_{R^d} f[k] g[n - k]$$

The convolution can be easily seen as an inner product operator for each translation index.

For 2D arrays (e.g., gray-scale images), we have a corresponding sum with indices (i, j) for f and (i − τ1, j − τ2) for g respectively.

Specifically, in case of a discrete-time system, the 2D convolution can be defined, in terms of discrete translation indices k1, k2, as:

$$< f[i,j], g[i - k_1, j - k_2] >= \sum_{k_1 \in \mathbb{R}^d} \sum_{k_2 \in \mathbb{R}^d} f[k_1, k_2] g[i - k_1, j - k_2]$$



A kernel is a filter applied all over the image in order to produce some output typically reduced in dimension wrt the input image. According to its values, it captures some characteristics of the input (e.g. horizontal edges, vertical edges, lines, blurring, …).

Convolution in NN
Since we want to exploit spatial properties of images, we won't vectorize the matrices (or tensors) but we'll use layers in NN keeping the shape of the input.
The size of the image is HXW.
Let x[i, j] and h[i, j] denote an image and its hidden representation, respectively, in the pixel location (i, j).
To allow each of the HW hidden nodes receiving input from each of the HW inputs, we would switch from using weight matrices (as in MLPs) to representing our parameters as four-dimensional weight tensors:

$$h[i,j] = u[i,j] + \sum_{l_1, l_2} W[i,j,l_1,l_2] \cdot x[l_1,l_2] = u[i,j] + \sum_{k_1, k_2} V[i,j,k_1,k_2] \cdot x[i + k_1, j + k_2].$$

The one-to-one correspondence between coefficients in both tensors is obtained just by setting

V [i, j, k1, k2] = W[i, j, i + k1, j + k2].

Now let's invoke the **translation invariance principle**: a shift in the inputs x should simply
lead to a shift in the activations h. This is only possible if V and u do not actually depend on (i, j), thus:

$$h[i,j] = u + \sum_{k_1,k_2} V[k_1,k_2] \cdot x[i+k_1, j+k_2].$$

This is a convolution! We are effectively weighting pixels (i + k1, j + k2) in the vicinity of (i, j) with coefficients V [k1, k2] to obtain the value h[i, j].

Now let's invoke the **locality principle**: it is not necessary to look very far away from (i, j) in order to glean relevant information to assess what is going on at h[i, j].
This means that outside some range |k1|, |k2| > Δ, we set V [k1, k2] = 0, thus:

$$h[i,j] = u + \sum_{k_1=-\Delta}^{\Delta} \sum_{k_2=-\Delta}^{\Delta} V[k_1,k_2] \cdot x[i+k_1, j+k_2].$$

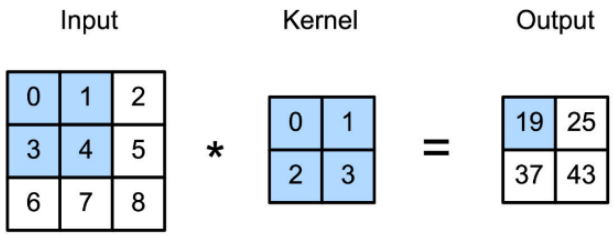
In this way, we are reducing filter size and also the cost computation.

In image processing we have fixed kernels and so we can extract some fixed and predefined effects like blur, vertical/horizontal lines,...
In neural networks we want kernels which are adaptive, so step by step we want to determine the values of those kernels.
We may have tasks like detection, segmentation, … We don't know for instance which values are good with respect to the task.
The convolutional filter is a small matrix depending on 2 indexes to slide over a tensor (the image). We can save a lot of parameters in this way wrt the FCL. The updating process of values of kernels is done according to the results they give with the input.

Convolutional Neural Networks

Representation of convolutional layer
image (input array) -> filter -> output

Cross correlation operator
This operation between input and kernel (matrix operation) is done through a cross-correlation:



The (learnable) parameters of a convolutional layer are the values of the kernel and the scalar bias.
We typically start training with random values for kernels, or following some other strategies.

Usually the size of the kernel is odd (1,3,5 or 7 are the most used). A common choice is 3x3.
Odd size is used because it preserves spatial dimensionality and padding can be performed with the same number of rows on top and bottom, and the same number of columns on left and right.

The output shape is based on the size of the input and kernel, but often we want to control it.
After lots of convolution operation the output size is much smaller than the original input, so we need something to handle this issue: the **padding**. It consists in putting extra pixels around the boundary of the input image, in order to have a greater size, so that even the output size is bigger (since it depends on the input one). Typically the extra-pixels are zeros, so they don't add information. The semantic information is still the same.

In some cases, we also want to reduce resolution: for this we use the **stride.** (stride is like the opposite of padding). By default we slide the kernel from a pixel at a time. But with stride we can control the number of steps so the pixels for which we slide the kernel (so it's like skipping some pixels). This helps in having a downsampling process. With a stride bigger than 1, we lose some information.

Cross-correlation with multiple input channels
In this case, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data.

Cross-correlation with multiple output channels
Often, we may need to have multiple output channels since representations are not learned independent but are rather optimized to be jointly useful.

Aggregating information
Often, we want to reduce the spatial resolution of the hidden representations to increase depth.
Often our ultimate task asks some global question about the image, so typically the nodes of the final layer should be sensitive to the entire input.
By gradually aggregating information, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.

Pooling operators
Pooling is an operator which downsamples the input but without learnable parameters, so they are used because they are slight.
Pooling operators are deterministic, typically calculating either the maximum (max-pooling) or the average value (average-pooling) of the elements in the pooling window.
Pooling layers can also change the output shape by padding the input and adjusting the stride.

Dilated convolution
A simple modification of the convolution operator is the dilated convolution which expands the receptive field without loss of resolution. The expansion of the filter allows it to increase its dimensionality by filling the empty positions with zeros. (similar to padding, but we “padd” over each receptive field). So we are enlarging the receptive field.
Considering the 2D case, let d be a dilation factor, given a filter kernel h [i, j], the receptive field can be widen by rewriting the convolution expression as:

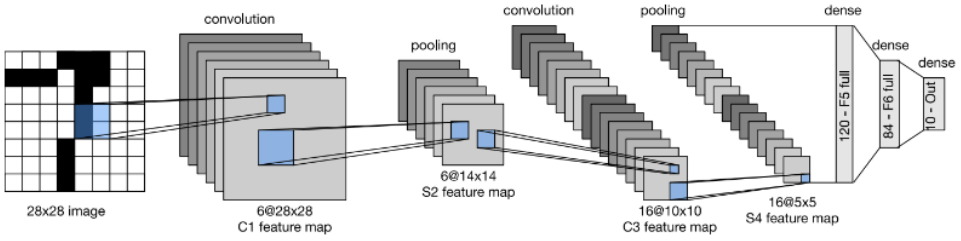
$$y[i,j] = \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} h[k_1,k_2]\, x[i - d \cdot k_1, j - d \cdot k_2]$$

Practically, the number of parameters associated with each layer is identical, while the receptive field grows exponentially as a power of two.

CNN architecture

We are now ready to put all of the tools together to deploy a fully-functional convolutional neural network, according to the following procedure:
• First, interleave multiple convolutional / max-pooling layers;
• Vectorize (flatten) the output of the last layer;
• Process the resulting vector with one or more fully-connected layers to obtain the final classification vector.
More recent CNNs add many variations on this basic architecture.

LeNet5:
LeNet5 is one of the first published CNNs, by Yann Lecun in 1998, for the purpose of recognizing handwritten digits in images.



Classic CNN pipeline

Although LeNet achieved good results on early small datasets, the performance and feasibility of training convolutional networks on larger, more realistic datasets had yet to be established.
Rather than training end-to-end systems (e.g., pixel to classification), classical pipelines looked more like this:
1 Obtain an interesting dataset.
2 Preprocess the dataset with hand-crafted features.
3 Feed the data through a standard set of feature extractors.
4 Dump the resulting representations into your favorite classifier, likely a linear model or kernel method.

Missing ingredients for deep learning

The ultimate breakthrough for deep learning in 2012 can be attributed to two key factors:
• Availability of data.
− Deep models with many layers require large amounts of data in order to enter the regime where they significantly outperform traditional methods.
− In 2009, ImageNet Challenge dataset pushed machine learning research forward providing 1 million of examples to train models.
• Availability of hardware.
− Deep learning models are voracious consumers of compute cycles.
− The computational bottlenecks in CNNs were solved by parallelizing convolutions and matrix multiplications on multiple GPUs.

Challenges in training neural networks

Training deep neural nets and getting them to converge in a reasonable amount of time can be tricky.
In particular, some practical challenges arise when training machine learning models and neural networks:
1 Choices regarding data preprocessing often make an enormous difference in the final results.
2 Activations in intermediate layers may take values with widely varying magnitudes. This alert drift in the distribution of activations could hamper the convergence of the network.
3 Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical.

Before 2012, many attempts were made to improve performance results of convolutional neural networks, by adopting different approaches.

On one hand, machine learning researchers believed the progress would have been driven by thriving and rigorous learning theories and algorithms.
On the other hand, computer vision researchers believed the progress would have been driven by bigger or cleaner datasets or by a slightly improved feature-extraction pipeline.
The truth was somewhere in between. Indeed, the progress began when it became clear that features themselves (i.e., representations) ought to be learned.

Learning representations

The process of applying a set of feature extractors to input data to obtain a representation was the most important part of a classical pipeline. Until 2012, this process was calculated mechanically.

Then, it was deemed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters.

The idea was to build models capable of capturing different levels of representations.

To this end, Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton proposed a new variant of a CNN, AlexNet, that achieved excellent performance in the 2012 ImageNet challenge.

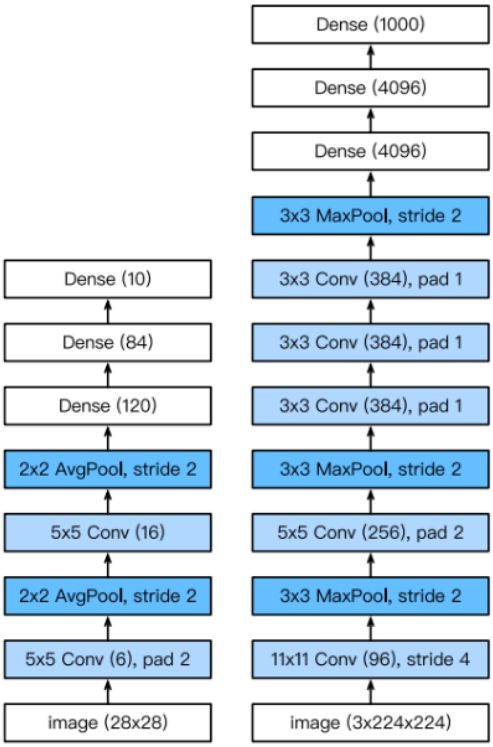
In the case of an image, the lowest layers might come to detect edges, colors, and textures.

Interestingly, in the lowest layers of the AlexNet [3], the model learned feature extractors that resembled some traditional filters.

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, etc.

Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents to separate different data easily.

AlexNET:



Reasons for AlexNet success: hardware.

Building deeper convolutional neural networks

From LeNet onwards

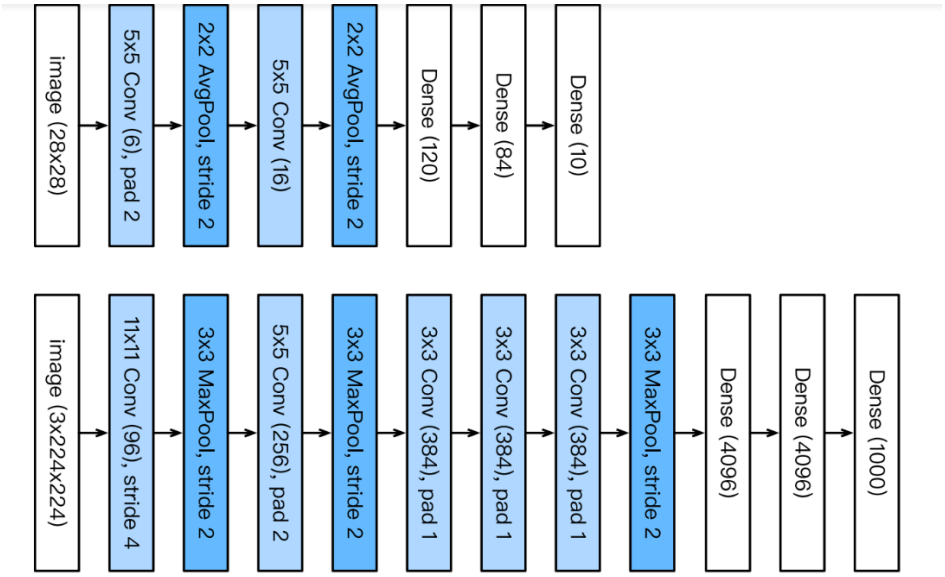
The basic building blocks came out in 1998, LeCun and his team built **LeNet 5**, a working CNN (5-7 layers) for handwritten digits recognition. (simple problems, quite simple datasets…). So these kind of "old" cnn worked but also other techniques could be used. So there weren't that kind of benefit by using the CNNs wrt preprocessing techniques.

ImageNet Challenge

However, from 2012 onwards, the combination of more computational power (especially GPUs), data, and a few algorithmic improvements quickly made deep CNNs the absolute state-of-the-art across several domains. So from 2012 we notice these improvements.
A challenge was made each 5 years by measuring the accuracy of image classification on imageNet dataset.

Overview of AlexNet

AlexNet was the first CNN to win the ILSVRC image classification competition by a large margin.



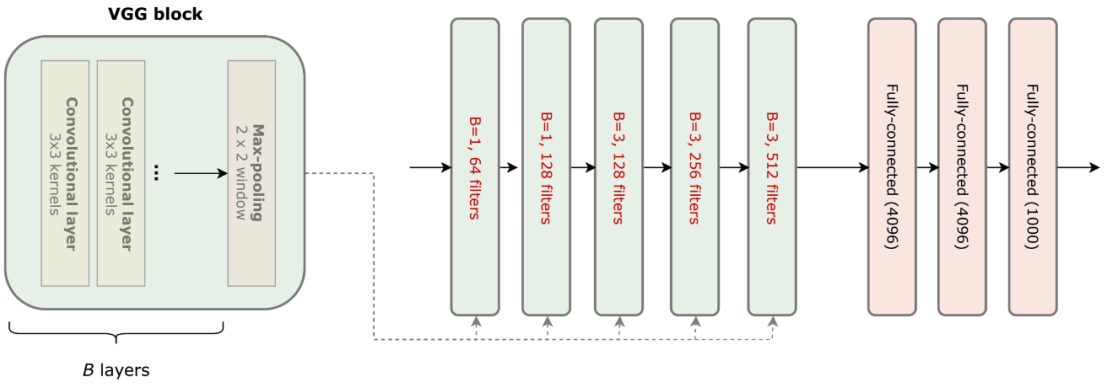
It had 8 adaptable layers (5 convolutional, 3 fully-connected). For training, it exploited several ideas, some of which relatively novel at the time:

- **ReLU** activation instead of sigmoid-like functions;
- **Data augmentation** and **dropout** to handle overfitting.

VGG

The **Oxford's Visual Geometry Group** (VGG) in 2014 popularized the idea of defining blocks composed of several layers, from which variants of a given architecture can be made according to a predefined scaling recipe. Their proposed block was very simple:

1 Several convolutional layers with size 3 × 3 and the same number of filters;
2 A single max-pooling block with 2 × 2 windows at the end of the block.



So, instead of defying an entire nn, design a block which will be repeated.

Experimenting on CIFAR-10
Consider the CIFAR-102 dataset: 60000 32x32 colour images in 10 classes, with 6000 images per class.
To see what happens when varying the depth, we experiment with a VGG-like architecture, varying the number of blocks, and the number of convolutional layers inside each block. We use a global average pooling at the end followed by a linear layer, with cross-entropy loss. We run 3 epochs, 3 repetitions each, with the Adam optimization algorithm.

By varying the depth we get doubled accuracy.

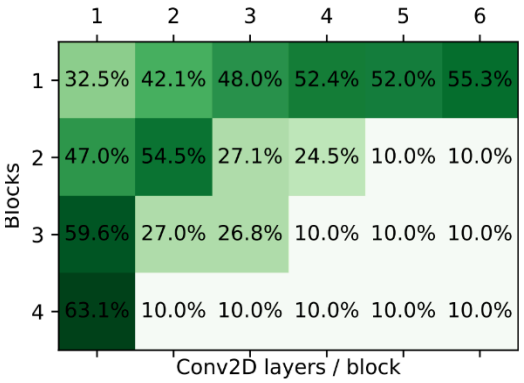


Figure 4: Test accuracy

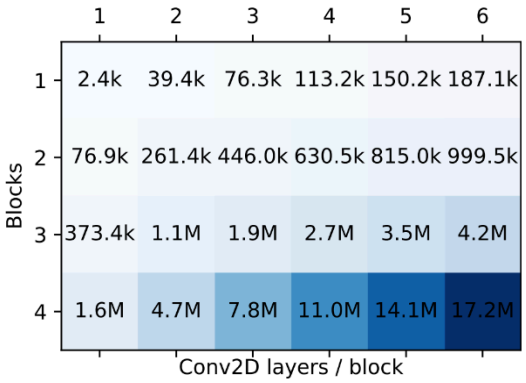


Figure 5: Parameters

• Some combinations do not even train (they get stuck at initialization). Other combinations appear much slower.
• Increasing the number of blocks looks good, but the original image is destroyed after a few max pooling operations.
• Increasing the number of layers in a block is also good, but the gains are more marginal.
• It is very easy to make the number of parameters go up.

Families of deep networks
Subsequent advances in the period 2012-2017 came from other teams competing in the ILSVRC (ImageNet) challenge and led to an explosion of new families and methods, including:
• Parallel layers and later Batch Normalization in the GoogLeNet (Inception) family.
• Residual connections in the ResNet models.
• Depthwise convolutions for efficiency in the MobileNet family.
• Neural architecture search combined with simple scaling strategies, e.g., EfficientNets and NASNets.

When moving towards a very deep regime, many strange phenomena appear, including (but not limited to):
• Scaling laws: performance scales linearly in a power law of data and compute, in a Moore-like fashion.
• Multiple descents: periods of overfitting may lead (after a while) to periods of better generalization.
• Emergent properties: scaling sufficiently may lead to a phase transition quickly moving performance on certain tasks from 0 to state-of-the-art (depending on the metric).

What about overfitting?

Overfitting happens when the performance of a model on the training set is improving, while the performance on a separate validation set is worsening.

Deep learning models are strangely resilient to classical overfitting, but they have shown some peculiar characteristics, e.g.:
1 Models trained for long enough on random data can still memorize the entire dataset and achieve perfect accuracy.
2 Models can start improving after a period of apparent overfitting (double descent).

So:

• The performance of a neural network depends heavily on the amount/quality of data,its architecture, hyper-parameters, initialization, optimization, etc. This requires a mix of good recipes, manual/automatic search, rules-of-thumbs, and experience.
• The tools we have available up to now are not enough for truly deep networks. In this lecture, we cover a number of additional tricks and layers designed especially to simplify and improve the training of the models.

Data Augmentation

Data augmentation is a technique to virtually increase the size of the dataset at training time:
1 Sample of mini-batch of examples;
2 For each example, apply one or more transformations randomly sampled (e.g., flipping, cropping, ...).
3 Train on the transformed mini-batch.
Data augmentation can be extremely helpful for overfitting, making the network more ro-
bust to small changes in the input data. In code, data augmentation can be included as part of the data processing, or as part of the model itself.

Other types of data augmentation:

- **MixUp:** combines two examples (x1, y1) and (x2, y2) by taking convex combinations with a random λ:
$$x = \lambda x1 + (1 - \lambda)x2$$
$$y = \lambda y1 + (1 - \lambda)y2$$
- **CutMix:** Define a mask M where Mij = 1 if the pixel belongs to the patch or not, we overimpose the random patch on the first image:
$$x = (1 - M) \odot x1 + M \odot x2$$
$$y = \lambda y1 + (1 - \lambda)y2$$

  where λ is again sampled from the uniform distribution in [0, 1].

Optimization Strategies

Early stopping is a procedure to find the optimal number of iterations (supposing a single descent curve):
1 Keep a portion of the dataset as the validation set.
2 For each epoch, check the validation loss (or accuracy).
3 Whenever validation loss is not improving for a while (a certain number of epochs called patience), stop the optimization process.
Early stopping is extremely common in neural networks; it highlights the difference between pure optimization and learning.

Regularization
A warning sign of overfitting can be large weights: these networks tend to be less smooth and make sharper changes in their outputs.
Regularization forces the optimization to select a network with smaller weights by penalizing large norms.

$$\mathbf{w}^* = \arg\min \left\{ \sum_i l(f(x_i), y_i) + \lambda \cdot \|\mathbf{w}\|^2 \right\}$$

λ is a hyper-parameter: with λ = 0 we have no regularization; with a λ too large, all weights would go to 0.
Regularization allows us to steer the optimization problem towards favourable solutions.
Many other types of regularization exists! For example, replacing the Euclidean norm of the weights with the sum of absolute values can lead to sparser solutions.

Weight decay
Consider the gradient update of a regularized loss:

$$-\text{Gradient of loss} = -\nabla \left[\sum_i l(f(x_i), y_i) \right] - 2C\mathbf{w} \,.$$

In the absence of the first term, the weights would decay exponentially to zero. In pure SGD, this form of regularization is also called weight decay. In other optimization algorithms, weight decay and regularization are different strategies and must be implemented differently.

On weight reparametrization

The ℓ_1 norm $\|\mathbf{w}\|_1 = \sum_i |w_i|$ is difficult to optimize with SGD, because it is not differentiable at 0. Interestingly, we can *reparameterize* $\mathbf{w}$ with two new vectors $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{w} = \mathbf{a} \odot \mathbf{b}$, and rewrite:

$$f(\mathbf{w}) + C\|\mathbf{w}\|_1 \rightarrow f(\mathbf{a} \odot \mathbf{b}) + C \left[\|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 \right] . \tag{8}$$

This has a similar loss landscape but it is much easier to optimize. In general, how we parameterize the same vector can have a significant impact on training and optimization.

Dropout

Why is data augmentation helpful?

The core idea is that we can make the network more robust by adding slight perturbations to the input. We can prove this to be a form of regularization. Dropout extends this idea to the network itself: instead of perturbing the images, we perturb the hidden layers by randomly dropping (removing) some of the connections.

Define $\mathbf{H}^{(b,f)}$ as the output of a generic fully-connected layer having f units, being fed with b inputs (mini-batch size).

With dropout, during training we replace it with:

$$\widetilde{\mathbf{H}} = \mathbf{H} \odot \mathbf{M}\,, \tag{9}$$

where $\mathbf{M}$ is a binary matrix with entries drawn from a Bernoulli distribution with probability p (i.e., $M_{i,j}$ is 0 with probability p, 1 with probability $(1-p)$).

If $M_{i,j} = 0$, the value $H_{i,j}$ is replaced with 0.

MonteCarlo Dropout

Consider a NN $f(\mathbf{x}; \mathbf{M})$ with a single layer of dropout. The output is a random variable w.r.t. the distribution of the mask $\mathbf{M}$. At inference time, a sensible approach would be to take the expected value:

$$\hat{y} = \mathbb{E}_{p(\mathbf{M})}\left[f(\mathbf{x}; \mathbf{M})\right] \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}; \mathbf{M}_i)\,, \tag{10}$$

where $\mathbf{M}_i \sim p(\mathbf{M})$ are draws of the mask, and the second step is a Montecarlo approximation. We call this Montecarlo dropout, and we can use the different samples to gather an uncertainty estimate on the output.

Dropout at inference time

A simpler and more common choice is to take the expected value of the dropout *layer* (as opposed to the entire network), since we can compute it in closed form:

$$\mathbb{E}[\widetilde{\mathbf{H}}] = p\mathbf{0} + (1-p)\mathbf{H} = (1-p)\mathbf{H}\,. \tag{11}$$

To simplify the inference, a common variant is the so-called **inverted dropout**:

$$\widetilde{\mathbf{H}} = (\mathbf{H} \odot \mathbf{M})/(1-p)\,.$$

This is useful because $\mathbb{E}[\widetilde{\mathbf{H}}] = p\mathbf{0} + (1-p)\frac{\mathbf{H}}{1-p} = \mathbf{H}$, i.e., we can simply remove the layer. Some books consider this *the* dropout implementation.

Early and late dropout

Dropout can also be switched on or off during training. Early dropout can be useful to counteract gradient variance in SGD during the initial stages of training; late dropout, instead, can be useful against overfitting.

Data normalization
Given a dataset X(n,d), it is common in machine learning to preprocess it so that each column has mean zero and unitary variance (z-scaling or standard scaling)

$$\mathbf{X}' = \frac{\mathbf{X} - \mu}{\sqrt{\sigma^2}}, \tag{12}$$

where $\mu, \sigma^2 \in \mathbb{R}^d$, $\mu_j = \frac{1}{n}\sum_i \mathbf{X}_i$ and $\sigma_j^2 = \frac{1}{n}\sum_i (\mathbf{X}_{ij} - \mu_j)^2$ are the empirical mean and variance. For example, we can do this in scikit-learn with `StandardScaler`. For many optimization algorithms, this can accelerate training significantly (preconditioning), because it makes the Hessian of $\mathbf{X}$ closer to the identity.

Batch normalization (BN), introduced in 2015, extends this idea by normalizing the outputs of each layer/block in a network,7 and then learning an optimal mean and variance for each unit.
This is not trivial, because the mean and variance of the layer's output will vary during the optimization, and recomputing them on the entire dataset can be computationally expensive. BN works by approximating the estimates using the data in the mini-batch.

Consider now a generic output $\mathbf{H}^{(b,f)}$ of a fully-connected layer (with batching). First, compute the empirical mean and variance column-wise:

$$\tilde{\mu}_j = \frac{1}{b}\sum_i [\mathbf{H}]_{i,j}, \quad \tilde{\sigma}_j^2 = \frac{1}{b}\sum_i ([\mathbf{H}]_{i,j} - \tilde{\mu}_j)^2 .$$

Second, we standardize the output so that each column has mean 0 and standard deviation 1:

$$\mathbf{H}' = \frac{\mathbf{H} - \tilde{\mu}}{\sqrt{\tilde{\sigma}^2 + \varepsilon}},$$

where $\varepsilon > 0$ is a small coefficient to avoid division by 0.

The final output of the batch normalization layer $\mathrm{BN}(\mathbf{H})$ sets a new mean and variance for each column:

$$[\mathrm{BN}(\mathbf{H})]_{i,j} = \alpha_j H'_{i,j} + \beta_j . \tag{13}$$

The $2f$ values α_j, β_j are trained via gradient descent.
Commonly, BN is inserted between a fully-connected layer and the activation function:

$$\mathbf{Z} = \phi\left(\mathrm{BN}(\mathbf{X}\mathbf{W} + \mathbf{b})\right) . \tag{14}$$

Similarly to dropout, BN requires a different behaviour outside of training, since it is undesirable that the output for an input depends on the mini-batch it is associated to.
Two common solutions are:
• After training, compute a mean and variance by running the trained model on the entire dataset, fixing the values μj , σj to that value.
• Keep a moving average of all the estimated means and variances when training, using the final value during inference.

BN is very common in convolutive layers. Consider the output H(b,h,w,c) of a generic 2D convolutive layer (b, as before, is the size of the mini-batch).
BN works exactly as before, but the mean and the variance are computed for each channel:

Why does BN work?
Despite its simplicity, batch normalization is extremely effective when training deep NNs.
Originally, its efficiency was believed to be consequence of a so-called internal covariate shift (i.e., distributions of activations changing layer-by-layer).
Nowadays, it is believed that BN works by making the optimization landscape smoother and, consequently, the gradients more predictive.

Problems of batch normalization
Batch normalization has multiple issues in practice:
• It introduces dependencies across the elements of the mini-batch, making it less suitable in, e.g, distributed optimization or contrastive self-supervised learning.
• The variance of the estimate can be very high when the batch size is small, which is a problem with very large models.
• There is a mismatch between its training and inference behaviours.

Layer normalization
It is very easy to define multiple variants of BN by varying the axes along which statistics are computed and controlled. For example, a popular variant is layer normalization, where we mean and variances are computed for each row (each input) independently. This works also with small batch sizes (even 1) and it does not add any inter-batch dependency. Importantly: in LN, α and β have the same shape as axis along which we normalize.

Residual Connections
In a residual network, we modify each block f(x) (e.g., a VGG block) by adding a skip connection:

$$r(x) = f(x)+x$$

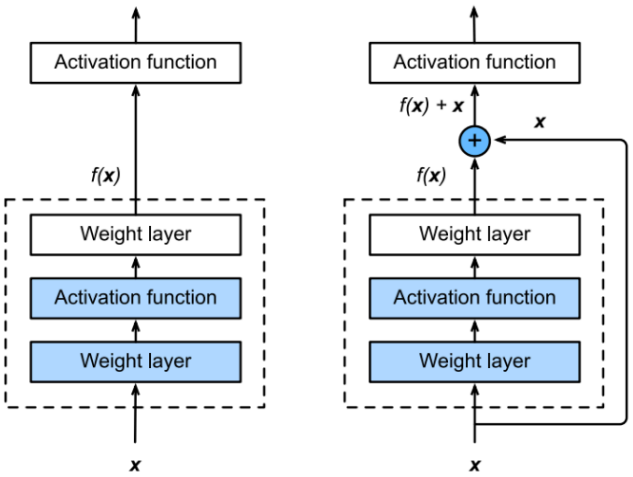
If x and f(x) have different dimensionality, we can rescale x with a matrix multiplication or a 1 × 1 convolutive block. r(x) is called a residual block.
Residual blocks work very well with batch normalization on f(x)(the residual path), because it tends to bias the network towards the skip path at initialization.

Note that:

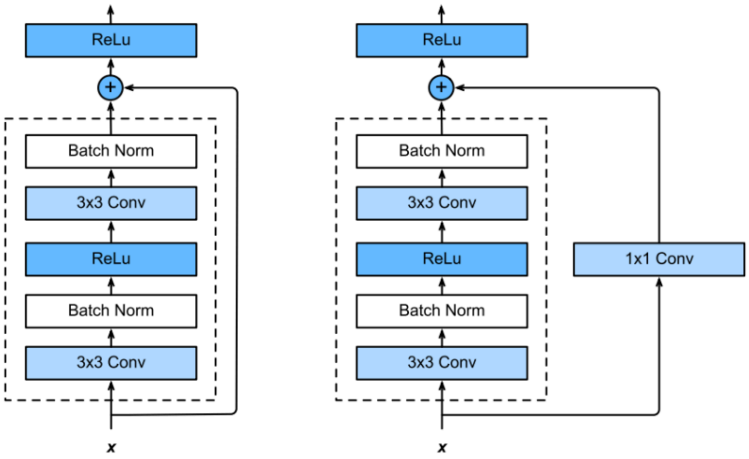
$$\partial r(x) = \partial f(x) + \mathbf{I}\,.$$

When using a residual connection, during backpropagation the gradient always flows unhindered trough the residual path, reducing any vanishing or exploding effects:

$$v^\top [\partial r(x)] = \underbrace{v^\top [\partial f(x)]}_{\text{Original VJP}} + v\,.$$

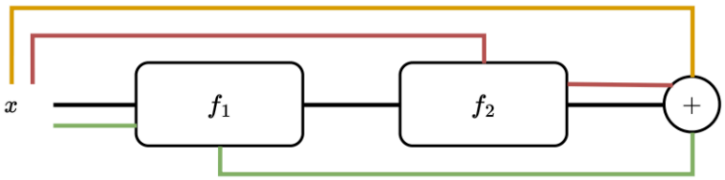


With rescaling:



ResNet as Ensambles

Two stacked residual blocks can be interpreted as creating four *paths* through the network, shown here with different colors:



The number of paths grow exponentially in the number of residual blocks, which can increase the robustness of the network and make it behave like an ensemble of shallow models (this interpretation will be very important in transformers).

Neural ODEs

Consider a network with T residual blocks, the output can be written as:

$$h_t = f(h_{t-1}) + h_{t-1}\,, t = 1, \dots, T\,, \tag{17}$$

with $h_0 = x$. In the limit $T \to \infty$ (infinite layers), each layer will only define an infinitesimal displacement. Although we cannot have infinite layers, we can handle this by conditioning a single residual block on t:

$$\frac{\partial h_t}{\partial t} = f(x, t)\,. \tag{18}$$

The previous equation can be solved by an ordinary differential equation (ODE) solver, and this class of models are called neural ODEs.

Convolutions beyond images

Many other types of data have a sequential / grid-like structure, albeit with different dimensionality: time-series, audio, videos, text, …

Temporal sequences

CNNs can be extended easily to other domains having grid-like structure of various dimensions.

For example, consider n steps of a time-series $\mathbf{x}_0, \dots, \mathbf{x}_{n-1} \in \mathbb{R}^c$, each step having c features (e.g., c different readings from different sensors).

We represent it using a matrix $\underset{(n,c)}{\mathbf{X}}$, and define a 1D convolutional layer of receptive field $2k$ as:

$$\underset{(n,c')}{[\mathbf{H}]_{i,d}} = \phi\left(\sum_{i'=-k}^{+k}\sum_{z=1}^{c} \underset{(s,c,c')}{[W]_{i'+k+1,z,d}}[\mathbf{X}]_{i+i',z}\right)\,.$$

Masked convolutions

For temporal domains, it is useful to define *causal* (masked) versions of the convolution operation.

> For a temporal sequence $\underset{(n,c)}{\mathbf{X}}$, a causal model $\underset{(n,c')}{\mathbf{H}} = f(\mathbf{X})$ is such that $[\mathbf{H}]_i$ depends only on $[\mathbf{X}]_j$ for $j \leq i$.

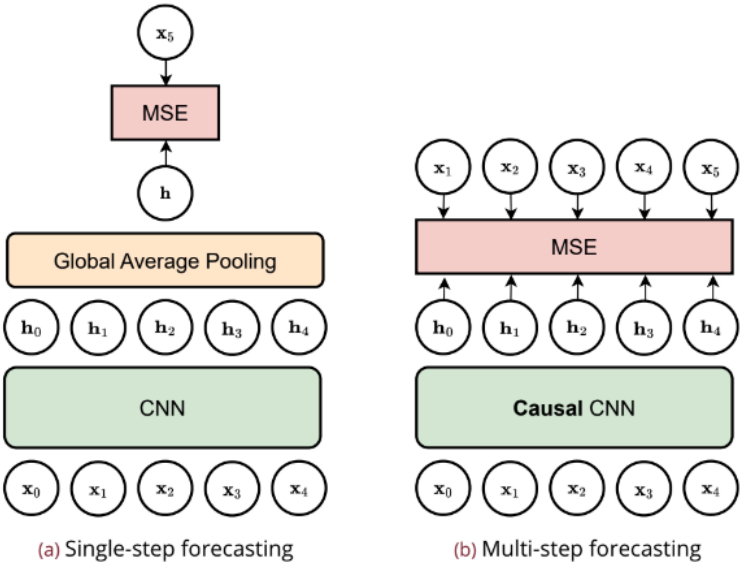
A causal convolution is defined by removing right neighbours:

$$\underset{(n,c')}{[\mathbf{H}]_{i,d}} = \phi\left(\sum_{i'=0}^{k}\sum_{z=1}^{c} \underset{(k+1,c,c')}{[W]_{i'+1,z,d}}[\mathbf{X}]_{i-i',z}\right)\,.$$

Forecasting

For time-series, a common task is forecasting, i.e., predicting the next step in the time-series.
With a causal model, we have two options:
• Pool the output representation H over n, and apply a regressor head to predict xn (also valid for non-causal models).
• Define a shifted target Y = [x1, . . . , xn], and train the model such that H ≈ Y, i.e., at each time step the network predicts the next one. This is only possible with a causal model, otherwise information would 'leak' from the input.

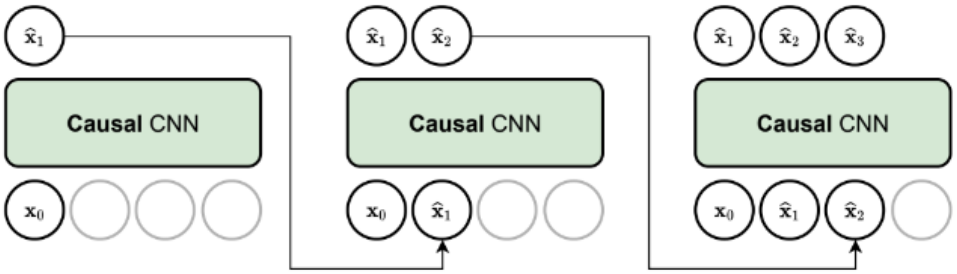
Forecasting with non-causal vs. causal models



(a) Single-step forecasting (b) Multi-step forecasting

Autoregressive models
Models of the second form are useful because they can be used to generate data in an autoregressive way. We prompt the model with the beginning of the sequence, and let it forecast the next step. Then, we concatenate it to the input and prompt again the model to generate another step, and so on.
During training, we can feed the network only with true samples (teacher forcing), or mix some of the network's own predictions. Extrapolating outside the training data windows may still be difficult.

Autoregressive generation



Video sequences

Now consider a sequence of n video frames, each of shape (h, w, c). The result is a tensor $X_{(n,h,w,c)}$.
A 3D version of the convolutional layer is straightforward:

$$[H]_{(n,h,w,c')_{t,i,j,d}} = \phi\left(\sum_{t'=-k}^{+k}\sum_{i'=-k}^{+k}\sum_{j'=-k}^{+k}\sum_{z=1}^{c}[W]_{t'+k+1,i'+k+1,j'+k+1,z,d}[X]_{t+t',i+i',j+j',z}\right)$$

Examples of audio classification

Many real-world problems require the classification of audio samples, e.g.:
1 Speech / non-speech identification (is he/she speaking now?);
2 Language identification (is it Italian?);
3 Genre / mood classification (is it rock?);
4 Determining the leading instrument;
5 Event recognition (is someone shooting?);
6 Scene recognition (are they in a bus? at a restaurant?).

Channels and sampling rate

An audio is a 1D sequence of samples, obtained with a certain sampling rate, typically in one or two channels.
For a given audio, the number of samples can be very high: at a resolution of 44kHz, we have almost half a million samples for each 10 seconds.

Can we handle the full waveform?

By having sufficient computational power, one can also work on the raw audio waveform.

When using convolutions, a key idea is the use of dilated convolutions (a.k.a. atrous convolutions, from French à trous), where neighbors are selected with exponentially increasing steps.
In this way, the receptive field of an item increases exponentially with the number of layers.

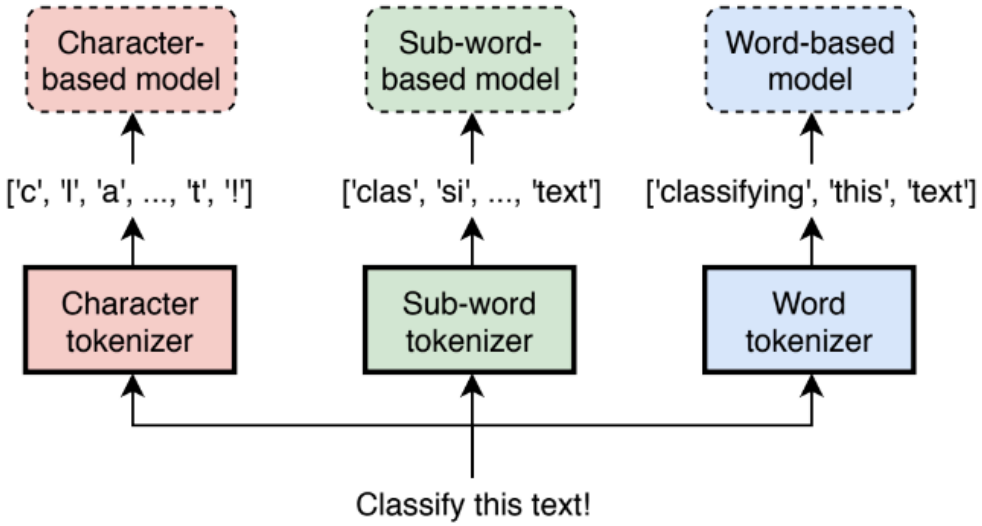
WaveNet
WaveNet (2016) was one of the first models to propose a 1D convolutional network dilated and causal convolutions for generating speech. In their design, dilation factors increased from 1 to an upper bound (e.g., 512), before restarting from 1.
Note that autoregressive generation of speech can be very slow, because the model must be called hundreds of thousands of times: although convolutive models have given way to transformers today, this problem is still true (e.g., the recent research on speculative decoding in LLMs).

Applying convolutive networks to text processing
Text data is another field with a vast range of possible applications, e.g.:
1 Recognition of a topic (is it talking about soccer?);
2 Hate speech recognition (is it respecting our code of conduit?);
3 Sentiment analysis (is it a positive review?);
4 Web page classification (is it an e-commerce website?).
Causal models applied to text are the backbone of LLM models such as ChatGPT.

Handling text data

Text must be preprocessed properly to be handled by a neural network. At the very least
we need to perform two basic operations:
1 Tokenize the text to split it into basic units (tokens) that will form a sequence.
2 Embed each token (one or more characters), i.e., convert it into numerical features usable by a neural network.
By the end of this procedure we should have a matrix X(n,d) of n tokens, each of size d.

Text tokenization



The importance of tokenization
Tokenization is crucial because it determines what the network will ‘see’. Consider that v15 of Unicode has almost 150k possible symbols, which can be combined in many forms (numbers, dates, function names, ...). Subword tokenization is very common, especially when learning the subwords from large corpora of texts (e.g., byte pair encoding, BPE).

One-hot encoding
The simplest vectorial embedding for text is a one-hot encoding according to a predefined dictionary:
• Character-level: each character is represented by a 1-of-C binary vector, where C is the number of allowable characters.
• Sub-word/word-level: similar, but each word/sub-word is represented with respect to a fixed vocabulary of sub-words / words.

• Sentence-level: each sentence can be represented by summing the one-hot encodings for the single tokens (bag-of-words).

Dense embeddings

One-hot vectors are very simplistic representations of the information contained in text. A more general solution is to learn a set of embeddings:
1 For every possible token c, initialize randomly a fixed-size vector vc.
2 During training, substitute each token in the sequence with the corresponding vector (look-up).
3 The set of vectors vc can be optimized together with the parameters of the neural network by doing gradient descent.

Dense vs. sparse embeddings

One-hot encodings are sparse, in the sense that most values are 0. In addition, they are not very informative: for example, the distance between any two encodings e0 and e1 is either 0 or $\sqrt{2}$.
Instead, trained embeddings can capture a rich semantic underlying the data, which is reflected in the possibility of doing algebraic manipulations on the embeddings themselves.
However, they can also capture biases of the data.

Pre-trained text embeddings

Text embeddings can also be pre-trained using a variety of algorithms:
1 Word2Vec (Mikolov et al., 2013);
2 Global Vectors for Word Representation (GloVe) (Pennington et al., 2014);
3 Embeddings from Language Models (ELMo) (Peters et al., 2018);
4 Generative Pre-Training (GPT) (Radford et al., 2018);
5 Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2018).

Neural Network for sequential information processing

In this case, we want to look at samples that appear in a sequential order: we want to exploit the information of data appearing in sequential order.

Processing sequential information

Many kinds of signals appear as sequences of data.
This data depends on the time information. (for spatial information we have a matrix and so on), now we have time involved. So we need to understand the information instant by instant.

Recurrent neural networks are based on this state variable, which are able to store some past information useful for understanding input data.

Dealing with sequence data

We have to introduce some statistical tools and new neural networks.

RNNs can be seen as non linear generalization of statistical standard linear model for time-series analysis which applications are in statistics, finance, meteorology, geophysics,... But also signal transmission, communication, signal transmission, control and communication engineering.
The goal is making some forecasting, so we want to predict some kind of metrics.

In the context of data mining, pattern recognition and machine learning the time series analysis can be used for clustering, classification, query by content, anomaly detection.
Also in natural language processing, raw text can be converted into sequences of the appropriate form and then analyzed for language modeling tasks.

Sequence data: quantitative attributes

They are used to define some kind of measure of data.

Time series is a sequence of random variables x0, x1, …, sampled from some kind of stochastic process (from a distribution).
Usually the observations are acquired at equally spaced time intervals, denoted as sampling rate.
Thus, time series can be described by quantitative statistic attributes:

$$\mu = \mathrm{E}\{\mathbf{x}\} \qquad \textit{first order statistics}$$

$$\boldsymbol{\Sigma} = \mathrm{E}\left\{(\mathbf{x}-\mu)(\mathbf{x}-\mu)^{\mathsf{T}}\right\} \qquad \textit{second order statistics}$$

$$r^{(m)}[n_1,\ldots,n_m] = \mathrm{E}\{(x[n_1])(x[n_2])\cdots(x[n_m])\} \qquad \textit{high order statistics}$$

$$c^{(m)}[n_1,\ldots,n_m] = \mathrm{E}\{(x[n_1]-\mu_1)(x[n_2]-\mu_2)\cdots(x[n_m]-\mu_m)\} \qquad \textit{high order statistics}$$

Sequence data: qualitative attributes

We want to classify this sequence of data based on the association among the samples.

• **Time series with no memory, or memoryless**, (e.g., white noise): each sample is independent of each other. (interested in very short window of samples, we have only the past information)

• **Time series with short memory** (ergodic processed, i.e., not white noise): each sequence is an example of all the other sequences of the same kind that could have been observed. (little area where we can find quantitative measures)

• **Time series with medium memory** (ergodic processes with presence of trend, seasonality, cycle, not stationary): usually we transform it into a short memory series.

• **Time series with long memory** (non-ergodic processes): long memory time series imply a form of dependence between observations of different epochs that persists even when the epochs are very distant from each other on the time axis. Short-memory series transformation requires a more sophisticated modeling (heavy tail phenomena). (dependence of samples which are very far in the input)

Statistics in sequence data

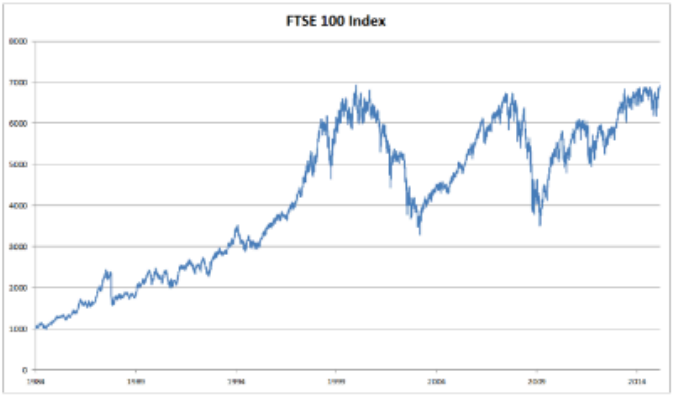


Figure 1: Example of sequential signal: stock prices index over 30 years [2].

Let's denote the prices by $x_t \geq 0$, i.e., at time $t \in \mathbb{N}$ we observe some price x_t. To do well in the stock market on day t, a trader should want to predict x_t via:

$$x_t \sim p(x_t \mid x_{t-1}, \ldots, x_1).$$

We want to compute the conditional probability.

Statistical tools for sequence data: autoregressive models I

Autoregressive models are used when I don't need all past samples, I just consider a window of them.

The trader could use a regressor, but there is just one major problem: the number of inputs, xt−1, . . . , x1 varies, depending on t.

The number of inputs increases with the amount of data, thus potentially being computationally intractable.

The neural networks that we are going to introduce now will mainly revolve around how to estimate p(xt | xt−1, . . . , x1) efficiently.
We can consider two main strategies requiring specialized statistical tools for estimation:

• Assume that a potentially rather long sequence is not really necessary and adopt an autoregressive (AR) model.

• Keep some summary ht of the past observations, at the same time update ht in addition to the prediction xˆt and adopt a latent autoregressive models.

For each instant we consider one latent variable storing the main information.

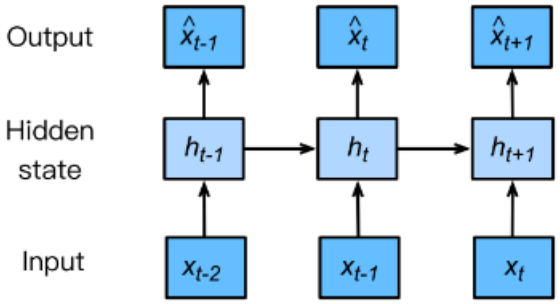


Figure 2: A latent autoregressive model [2].

Autoregressive models estimate the training time series as:

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \ldots, x_1).$$

Markov models

Whenever the AR approximation of using only xt−1, . . . , xt−τ instead of xt−1, . . . , x1 holds to estimate xt, we say that the sequence satisfies a Markov condition.
If τ = 1 (window = one sample), we have a first-order Markov model and P(x) is given by:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1).$$

Such models are particularly nice whenever xt assumes only a discrete value, since in this case dynamic programming can be used to compute values along the chain exactly.

Order of decoding

In principle, there is nothing wrong with unfolding P(x1, . . . , xT) in reverse order:

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T).$$

In fact, if we have a Markov model, we can obtain a reverse conditional probability distribution, too.
In many cases, however, there exists a natural direction for the signals, namely going forward in time.
It is clear that **future events** cannot influence the past. Hence, if we change xt, we may be able to influence what happens for xt+1 going forward but not the converse.

Time series prediction: dataset preparation
Let us consider a generated time series (e.g., a sine function with additive noise).

- Take time series and extract features and labels in order to train NN

We map the data into pairs yt = xt and xt = (xt−1, . . . , xt−τ).

Time series prediction: training

To process the time series, we consider a simple architecture consisting of:

- two fully connected layers,
- ReLU activation,
- l2 loss.

The model shows a small training loss, thus we would expect the model to work well.
We now evaluate how well the model is able to predict future samples.

Time series prediction: one-step-ahead prediction

Let us consider the one-step-ahead prediction, the prediction results look nice as expected,
but we need always to move only 1 step forward at each iteration.

Time series prediction: multi-step-ahead prediction

We may move forward with a larger step k > 1 and implement a multi-step-ahead prediction.
However, we need to consider that, as k increases, the error can diverge rather rapidly from the true observations.
For instance, weather forecasts for the next 24 hours tend to be pretty accurate but beyond that the accuracy declines rapidly.
There is quite a difference in difficulty between interpolation and extrapolation: the temporal order of the data must be always respected when training on a time series, i.e., never train on future data.
For causal models (e.g., time going forward), estimating the forward direction is typically a lot easier than the reverse direction.

Recurrent Neural Network

Latent variable models

In sequence data, the conditional probability of a sample xt at position t only depends on the past n-1 previous samples.
Since we should need a lot of memory to store them (intractable: the number of parameters would increase exponentially), we may need a latent variable model, so a model in which instead of memorizing the past samples we use just one latent variable (the actual sample plus an information of this latent variable: a summary of all past samples).

Hidden state variables

This latent variable has to be translated in NN form: a layer.
A latent variable ht is also called as hidden variable or hidden state. The hidden state at time t could be computed based on both xt and ht−1:

$$h_t = f(x_t, h_{t-1}).$$

After all, ht could simply store all the data it observed so far. But it could potentially makes both computation and storage expensive. Note that we also use h to denote by the number of hidden units of a hidden layer, but they are not to be confused! Recurrent neural networks are neural networks with hidden states.

Neural network without hidden states

Consider a multilayer perceptron with a single hidden layer with a minibatch X ∈ R (n×d) with sample size n and d inputs, similarly to the network previously used for time series prediction.
Let the hidden layer's activation function be φ. The hidden layer's output H ∈ R (n×h) is:

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h)$$

where Wxh ∈ R (d×h) is weight parameter, bh ∈ R (1×h) the bias parameter, and h is the number of hidden units for each hidden layer.
The hidden variable H is used as the input of the output layer.

The output variable O ∈ R (n×q) is given by:

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q.$$

where Whq ∈ R (h×q) is the weight parameter, and bq ∈ R (1×q) is the bias parameter of the output layer.
For a classification problem, we can use softmax(O) to compute the probability distribution of the output category.
We can pick (xt, xt−1) pairs at random and estimate the parameters W and b of our network via stochastic gradient descent.

Neural network with hidden states: the recurrent neural network

Now consider a neural network with hidden states with a minibatch $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ and a hidden variable $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ at time step t.

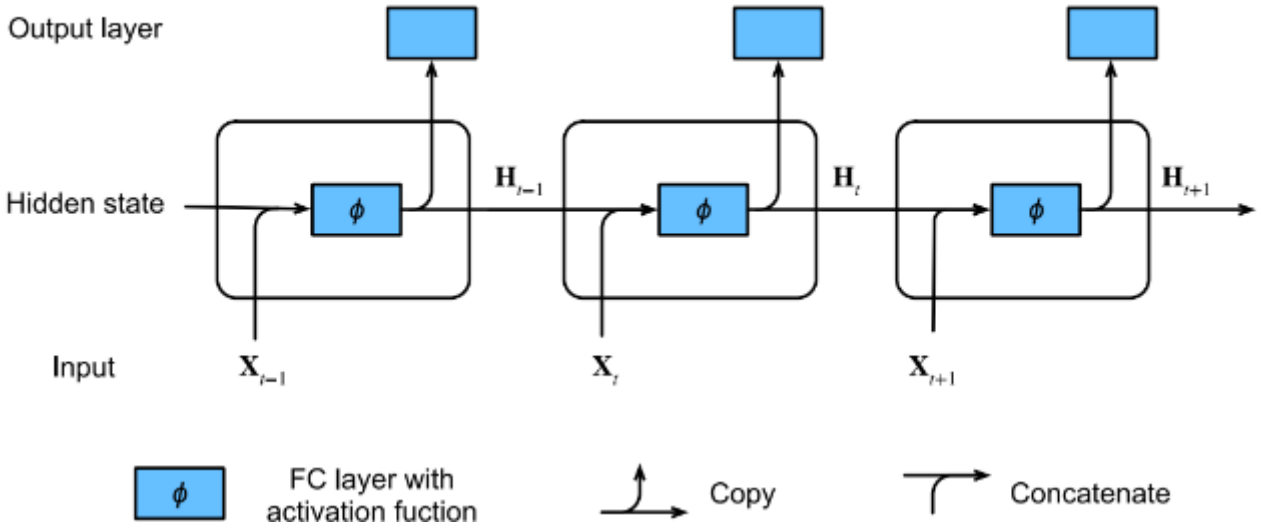
Unlike the MLP, here we save $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ to describe how to use $\mathbf{H}_{t-1}$ in the current time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h). \tag{3}$$

The neural network described by (3) and containing the additional term $\mathbf{H}_{t-1}\mathbf{W}_{hh}$ denoting the hidden state, is called **recurrent neural network** (RNN).

The output of the RNN is defined like an MLP without hidden states, i.e., by (2).

It is worth noting that the number of RNN model parameters does not grow as the number of timesteps increases.



Numerical instability of the gradient computation

For a sequence of length T, we compute the gradients T times for iteration, which might result in numerical instability, e.g., the gradients may either explode or vanish for large T. Recall that when solving an optimization problem, we take update steps for the weights wt in the general direction of the negative gradient gt on a minibatch, say wt+1 = wt − η · gt. Let's further assume that the objective is well behaved, i.e., it is Lipschitz continuous with constant L, i.e.:

$$|l(\mathbf{w}_t) - l(\mathbf{w}_{t+1})| \leq L \|\mathbf{w}_t - \mathbf{w}_{t+1}\|.$$

Thus, if we update wt by ηgt, we will not observe a change by more than Lη ||g_t||. This may slow down the convergence speed but it may also limit the gradient extent in wrong directions.

Gradient clipping

If the gradient is too large we may reduce the learning rate η.
If we only rarely get large gradients, we may clip the gradients by projecting them back to a ball of a given radius, θ:

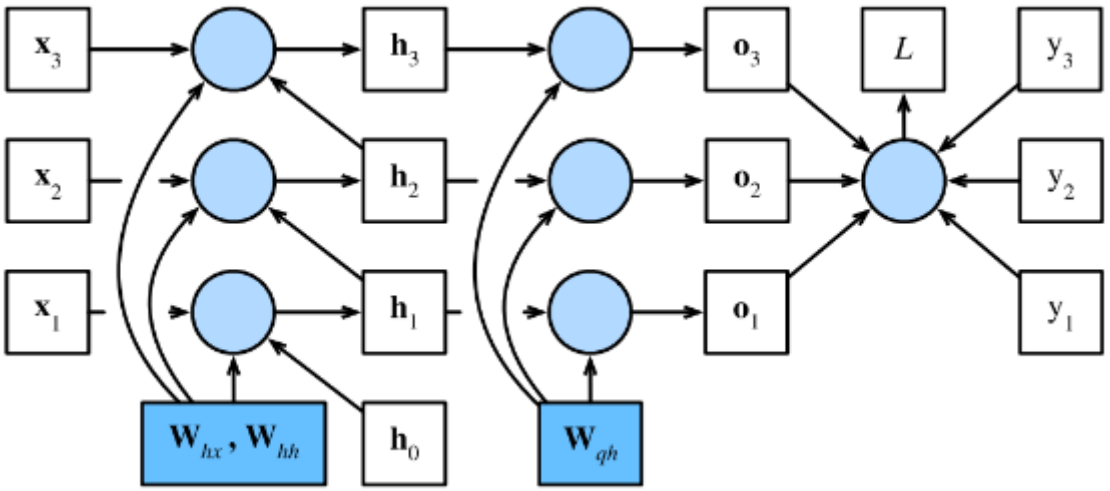
$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}.$$

Due to the gradient clipping, the gradient norm never exceeds θ and the updated gradient is entirely aligned with the original direction g.
It also has the desirable side-effect of limiting the influence any given minibatch can exert on the weight vectors. Gradient clipping provides a quick fix to the exploding gradient.

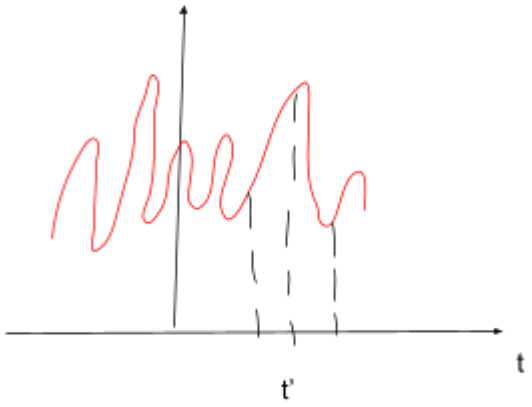
Backpropagation through time

Forward propagation in an RNN is relatively straightforward.
On the other hand, the backpropagation in an RNN is more specific and it is known as backpropagation through time.
It requires us to expand the RNN one timestep at a time to obtain the dependencies between model variables and parameters.
Then, based on the chain rule, we apply backpropagation to compute and store gradients.
Since sequences can be rather long, the dependency can be rather lengthy.

Computational graph of the backpropagation through time

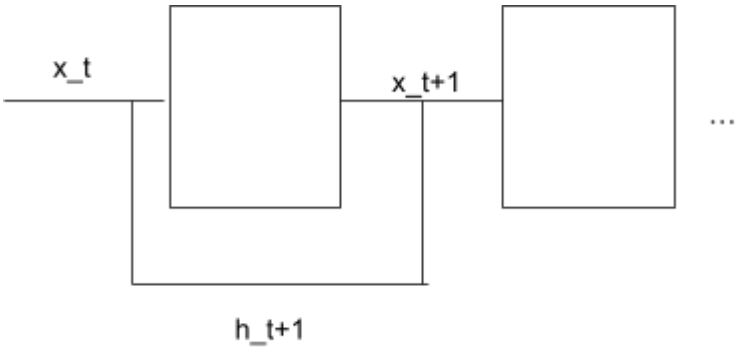


Neural Networks for Long Short-Term Memory Sequences



From the point of observation t' we can derive some information from a very short past wrt t'. In this case we talk about short time correlation. But I can also consider longer windows where we may find some kind of information (long term correlation) which might not be uniformly distributed over the window.
There is not a stochastic process describing this correlation. So we need more advanced architectures.
With larger window we may have some problems with gradients and memory.

RNNs (classical)



How to consider larger windows with this structure?
Traditional recurrent neural networks are not sufficient to solve many of today's sequence learning problems, such as:
• numerical instability during gradient calculation;
• need for deeper representation;
• need for a bidirectional design with both forward and backward recursion;
• solving sequence-to-sequence problems.

Another problem of traditional RNNs is to connect previous information to the present task.
RNNs are absolutely capable of handling short-term dependencies, but they may incur some problems as long as the gap between the previous relevant information and the actual point becomes very large.

Gated Recurrent Units

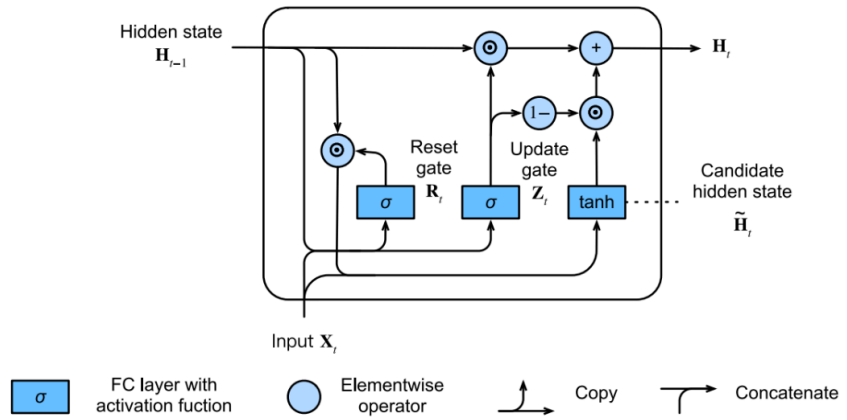
More easy and recent than LSTM.
Based on the gating mechanism.They are able to avoid problems of gradient computation:

• An early observation might be highly significant for predicting all future observations.
We would like to have some mechanisms for storing vital early information in a memory cell.
• Some symbols might carry no pertinent observation. We would like to have some mechanism for skipping such symbols in the latent state representation.
• There might be a logical break between parts of a sequence. Thus, it would be nice to have a means of resetting our internal state representation.

So, We need a mechanism able to:
- store some information from the past
- skip irrelevant (unuseful) sequences from the past
- reset latent representation

all this in an autonomous way.

The gated recurrent unit (GRU) is one of the modern RNN models, whose peculiarity lies in the presence of a hidden state gating, including dedicated update and reset gates.



Graphically:

We use for the input x_t the hidden state of the previous cell.
Then we pass x_t and h_t-1 to both a reset and update gates (2 matrices) after having applied some activation function.
Typically, we use a sigmoid function to transform input values to the interval (0, 1).

Assuming X_t in R (nxd) and H_t-1 in R (nxh):

Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z).$$

with $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$.

Next, we integrate the reset gate Rt with a regular latent state updating mechanism, leading to a candidate (possible) hidden state:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h) \quad (1)$$

where $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ and $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, and the symbol $\odot$ is the Hadamard (elementwise) product operator.

This matrix (with the same number of rows and columns) selects the useful information by multiplying element-wise the matrix H_t-1 and R_t.
Whenever the entries in the reset gate Rt are close to 1, we recover a conventional RNN.

Finally, we incorporate the effect of the update gate Zt, which determines how much of the candidate state H̃_t is used.
We simply take elementwise convex combinations between both Ht−1 and H̃_t.
This leads to the final update equation for the GRU.

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

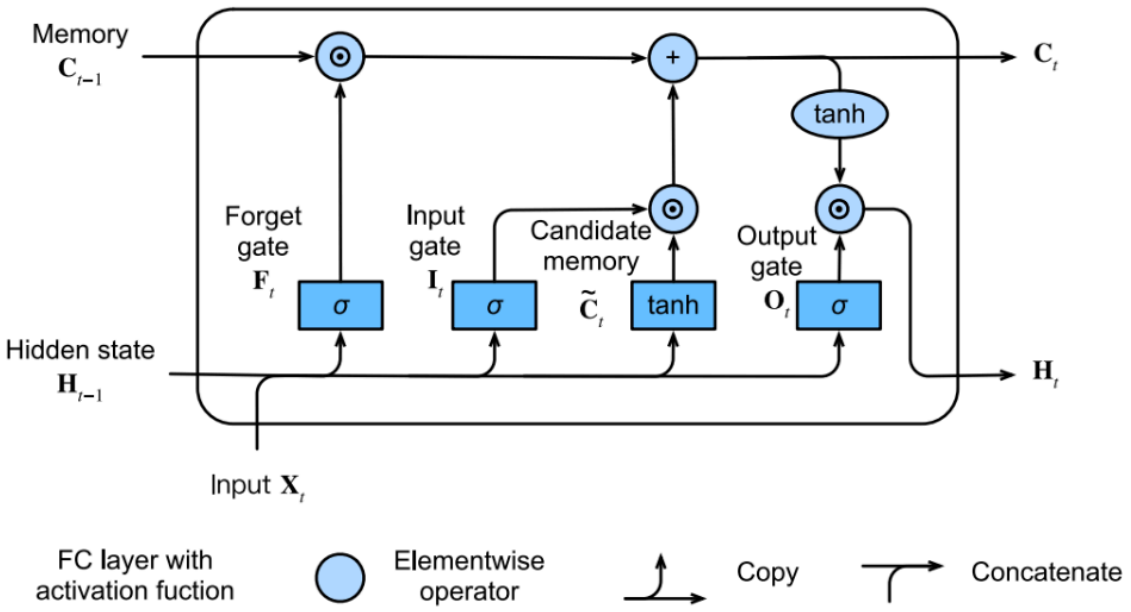
Whenever the update gate Zt is close to 1, we simply retain the old state.
In contrast, whenever Zt is close to 0, the new latent state Ht approaches the candidate latent state H̃_t.

In summary, GRUs have the following two distinguishing features:
• Reset gates help capture short-term dependencies in time series.
• Update gates help capture long-term dependencies in time series.
These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for time series with large timestep distances.

Long Short-Term Memory Networks

Long Short Term Memory (LSTM) networks are a special kind of RNN, capable of learning long-term dependencies.
The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time.
The LSTM shares many of the properties of the GRU. Its design is slightly more complex than GRU but predates GRU by almost two decades [2].
LSTM's design is inspired by logic gates of a computer. It introduces a memory cell (or cell) with the same shape as the hidden state, engineered to record additional information.
To control the memory cell we need a number of gates to decide when to remember and when to ignore inputs in the hidden state via a dedicated mechanism.

Graphically:



Just like with GRUs, the data feeding into the LSTM gates is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ and $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$.

These inputs are processed by a fully connected layer and a sigmoid activation function to compute the values of input, forget and output gates, which are all in the range of $[0, 1]$.

The input gate $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, the forget gate $\mathbf{F}_t \in \mathbb{R}^{n \times h}$, and the output gate $\mathbf{O}_t \in \mathbb{R}^{n \times h}$ are calculated as follows:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned}$$

with $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$, and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$.

Next, we design the memory cell.

We first introduce the *candidate* memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$.

It uses a $\tanh$ function with a value range for $[-1, 1]$ as the activation function:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c).$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$, and $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$.

In GRUs, we had a single mechanism to govern input and forgetting.

Similarly, in LSTMs we have: the input gate $\mathbf{I}_t$ that governs how much we take new data into account via $\tilde{\mathbf{C}}_t$, and the forget gate $\mathbf{F}_t$ that addresses how much of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells $\mathbf{C}_{t-1}$ will be saved over time and passed to the current timestep.

This design was introduced to alleviate the vanishing gradient problem and to better capture dependencies for time series with long range dependencies.

Last, we need to define how to compute the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$.

In LSTM, the output gate is simply a gated version of the $\tanh$ of the memory cell.

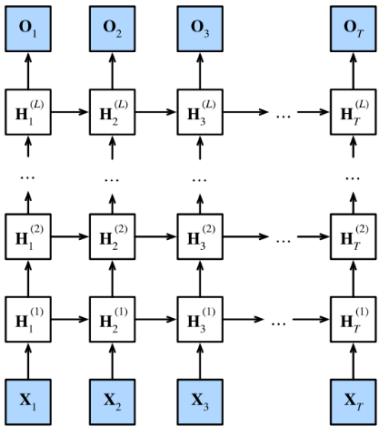
This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

Whenever the output gate is 1 we effectively pass all memory information through to the predictor, whereas for output 0 we retain all the information only within the memory cell and perform no further processing.

Deep Recurrent Neural Networks

Stacking multiple recurrent layers



Up to now, we only discussed RNNs with a single unidirectional hidden layer.
The specific functional form of how latent variables and observations interact was rather arbitrary. This is not a big problem as long as we have enough flexibility to model different types of interactions.
In the case of the perceptron, we fixed this problem by adding more layers.

Within RNNs this is a bit more tricky, since we first need to decide how and where to add extra nonlinearity.
We could stack multiple layers of RNNs on top of each other.
This results in a mechanism that is more flexible, due to the combination of several simple layers.
In particular, data might be relevant at different levels of the stack.

Functional dependencies

Let us consider a minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$.

The hidden state of hidden layer l ($l = 1, \ldots, L$) is $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$, the output layer variable is $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ and a hidden layer activation function ϕ_l for layer l.

The hidden state of layer l is expressed as:

$$\mathbf{H}_t^{(l)} = \phi_l \left(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{xh}^{(l)} + \mathbf{b}_h^{(l)} \right).$$

with $\mathbf{W}_{xh}^{(l)}, \mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, and $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$.

The output layer is only based on the hidden state of hidden layer L:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q$$

with $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$, and $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$.

Bidirectional Recurrent Neural Networks

The dynamic programming problem

In sequence learning, the goal of modeling the next output, given any past information, is not the only scenario we might encounter.
In order to illustrate the motivation why one might use deep learning and why one might pick specific architectures, we describe the dynamic programming (DP) problem.
DP is a method for solving problems by breaking them down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.
The next time the same subproblem occurs, instead of recomputing its solution, it is possible to simply look up the previously computed solution.

Previous information in hidden Markov models

Let us recall a hidden Markov model designed as a probabilistic graphical model as follows.

At any time step t, we assume that there exists some latent variable h_t that governs our observed emission x_t via $P(x_t|h_t)$.

Moreover, any transition $h_t \leftarrow h_{t+1}$ is given by some state transition probability $P(h_{t+1}|h_t)$.

Thus, for a sequence of observations we have the following joint probability distribution over the observed and hidden states:

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \text{ where } P(h_1 \mid h_0) = P(h_1). \quad (2)$$

This probabilistic graphical model is then a hidden Markov model as depicted in Fig. 11.

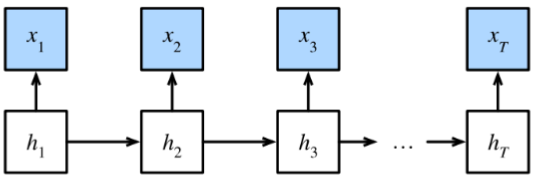


Figure 11: A hidden Markov model [1].

If we need to compute the conditioned distribution in (2) but we have no latent variable, we should consider summing over all the possible combinations of choices for $h_1, \ldots, h_T$, which is computationally demanding.

Dynamic programming in hidden Markov model

An efficient and elegant solution for that computation is given by the *dynamic programming*.

It consists in summing over latent variables $h_1, \ldots, h_T$ in turn with a *forward recursion*:

$$P(x_1, \ldots, x_T) = \sum_{h_1,\ldots,h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \sum_{h_1,\ldots,h_T} \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t)$$
$$= \ldots = \sum_{h_T} \pi_T(h_T) P(x_t \mid h_t).$$

However, the same result can be also obtained by using a *backward recursion* (see [1]).

Both the recursions allow us to sum over T latent variables in $\mathcal{O}(kT)$ (*linear*) time over all values of $(h_1, \ldots, h_T)$ rather than in exponential time.

This is one of the great benefits of the probabilistic inference with graphical models.

Bidirectional model

If we want to have a mechanism in RNNs that offers comparable look-ahead ability as in hidden Markov models, we need to modify the RNN design.
Instead of running an RNN only in the forward mode starting from the first token, we start another one from the last token running from back to front.
Bidirectional RNNs add a hidden layer that passes information in a backward direction to more flexibly process such information.
This is not too dissimilar to the forward and backward recursions in the DP of hidden Markov models.
The main distinction is that in DP these equations had a specific statistical meaning.
Now they are devoid of such easily accessible interpretations and we can just treat them as generic and learnable functions.

Definition of a bidirectional neural network

For a given timestep t, the minibatch input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d) and the hidden layer activation function is ϕ.

In the bidirectional architecture, we assume that the forward and backward hidden states for this timestep are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$. h indicates the number of hidden units.

We compute the forward and backward hidden state updates as follows:

$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),$$
$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}).$$

where the weights $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, and bias parameters $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ are all model parameters.

Next, we concatenate the forward and backward hidden states $\overrightarrow{\mathbf{H}}_t$ and $\overleftarrow{\mathbf{H}}_t$ to obtain the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ and feed it to the output layer.

In deep bidirectional RNNs, the information is passed on as *input* to the next bidirectional layer.

Last, the output layer computes the output $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q):

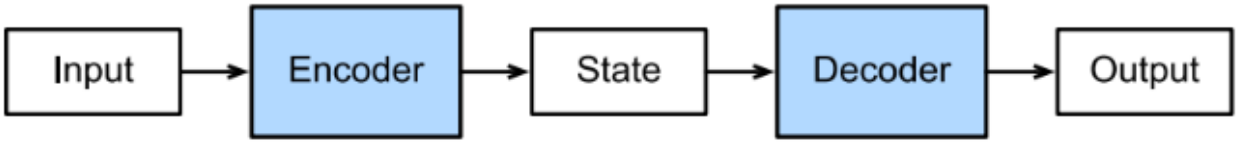
$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

Here, the weight parameter $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ and the bias parameter $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer. The two directions can have different numbers of hidden units.

Encoder-Decoder Architecture

Processing variable-length sequences

In many application domains, input and output sequences can show both a variable length.
To handle this type of sequence, we can design an architecture with 2 main components.
• An encoder, which takes a variable-length sequence as the input and transforms it into a state with a fixed shape.
• A decoder, which maps the encoded state of a fixed shape to a variable-length sequence.



Transformer

Intro

Historically:
FCL (limit in terms of kind of inputs: bunch of vectors) -> CNN (state of the art for images in cv tasks, but limited i.e. text inputs with different lengths, dependencies are not local, graphs, audio…) -> RNN (not able to handle very long sequences, too slow, problems with backpropagation…)
So we've moved to transformers with one core layer (the attention layer) from which the neural network is built. It is easily generalizable (Really widely used in several tasks) and computed (multiplications of matrices).

The problem of convolutions
Good when input data has local properties (for example in images). But the assumption of locality is not always good for example when dealing with texts:

- a subject which is far from the verb (sparse and long range dependencies)
- the important words might not be the one near to the current word
- they have variable lengths
- dynamic input

Recurrent neural networks
The main idea here is to exploit the state containing all useful information, but this is not always true.
Another issue is that RNNs are slow for backpropagation issues so they are heavy in time, but also in space.

Transformer
The alternative came out in 2017 with transformers, based on the attention layer (or multi head attention layer).
The original Transformer (Vaswani et al., 2017), was an encoder-decoder model for NLP tasks.
Today, similar models are gaining interest in audio, computer vision, biology, etc.

How can we go from convolution to attention?

Moving beyond locality

Assume we are in NLP, so our input is a sequence of vectors where each of these vectors are tokens (that can be pieces of text embedded in a vector space). We can transform lots of input types in this way (like images, audio,...).
The size of these vectors is a hyperparameter and is called embedding size.

We can write a 1D convolutional layer (ignoring padding) of kernel size k as:

$$\mathbf{h}_i = \sum_{j=-k}^{k} \mathcal{W}_j \mathbf{x}_{i+j} \,,$$

where W (2k,d,d) is the kernel tensor. What we want to do is remove the locality. We want this summation to run all the sequence. We don't want one set of weights for every position.
I also want to have a low number of parameters, but the only way in convolution to handle long-term dependencies is to augment the dimension of the filter, but this leads to a larger sensor. Adding new positions basically means adding matrices and this at a certain point can be equivalent to having a FCL.

Continuous convolution

Key idea: if i don't want one matrix per position, what I can do is instead parametrize this filter. So I add a small neural network, so basically a function g going from N to R (dxd), which takes as input the shift and learns the parameters of the kernel for each possible position i:

$$\mathbf{h}_i = \sum_{j=-i+1}^{n-i} g(i+j)\mathbf{x}_j \ .$$

These are called continuous convolutions, and they work well with image-like data with, e.g., variable sizes and resolutions. So, g is a fixed (no increasing number of parameters) neural network (like 2 dense layers that takes a scalar input and returns a dxd outputs, in matrix form).

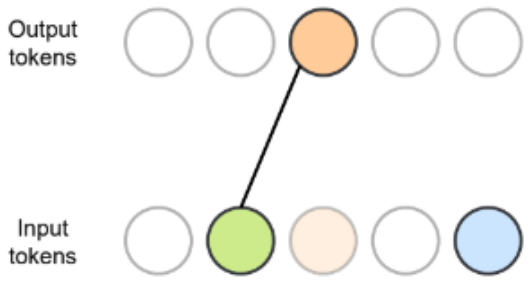
Non-local neural networks

The previous model works well at handling non-locality, but it still assumes that dependencies are regular, i.e., they only depend on j (so, on the position of the sequence). This would be ok for images, but not for texts. We can make it more general (non-local nn) by letting them depend on the values of tokens instead:

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha(\mathbf{x}_i, \mathbf{x}_j)\mathbf{x}_j \ .$$

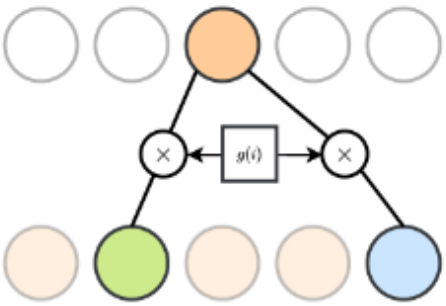
It is a function of the two tokens that we're considering i and j.
This is an example of a non-local neural network model. By a proper choice of the weighting function α(·, ·) we can obtain the MHA layer.

Differences between Convolution, Continuous Convolution and Non-Local NN:
In the pictures we are considering the orange neuron.



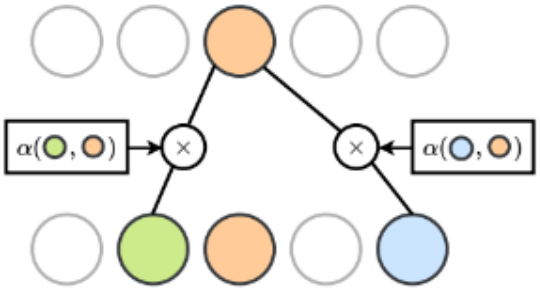
(a) Conv1D, kernel size 3

In convolution (kernel 3 and 1D), we consider the green (very close) and blue (bit farther) neurons we have fixed connection. The blue one has no connection with the output because it is outside the receptive field.



(b) Continuous convolution

In this case, both green and blue are connected and their weight will only depend on the distance. The connections here are variable but depend on the position.



(c) Non-local model

In non local models, we have again a function of vectors themselves (so green, orange and blue, orange). Here, the connections are variable but depending on the value of vectors (tokens).

Choosing the weighting function

It's called "attention function" because it represents the importance that a token should give to another one.
We make a few assumptions to simplify the layer:

• The output of α is a scalar (not a matrix). We call α the **attention scoring function** (or attention function), and its outputs the attention scores for token i.

• For each token, its attention scores are normalized in the simplex (they are positive and they sum to one).

With this formulation, each token will have a 'budget' of attention to allocate, i.e., increasing an attention score necessarily decreases the attention over the remaining tokens.

The higher the score is, the higher is the importance to give to a certain token (similarity measure).
The lower the score is, the more the 2 tokens are dissimilar.
That's what we expect the function to do.

Attention layer

There are many choices for the attention function; commonly, we use the *normalized* dot product (large vectors would produce large results), because it is fast and efficient to parallelize:

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{\sqrt{d}} \mathbf{x}_i^\top \mathbf{x}_j$$

Putting everything together we obtain (always for a single token): we have the dot product between the token i and the token j which is the attention score. we normalize this score with the softmax and we use them to average the token xj.

$$\mathbf{h}_i = \sum_{j=1}^{n} \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{x}_i^\top \mathbf{x}_j \right) \mathbf{x}_j$$

X is appearing in 3 shapes: xi which is the one that will be updated, xj which is the one we are comparing and xj which is the one that we are summing .
So, it's like we are seeing if two vectors points the same direction

Adding some parameters

Following the previous concept, we want to project xi in its 3 components:

$$\mathbf{q}_i = \mathbf{W}_q^\top \mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}_k^\top \mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}_v^\top \mathbf{x}_i$$

That we call respectively query, key and value.
So for every token we get 3 components. Let's substitute these 3 components in the previous equation:

$$\mathbf{h}_i = \sum_j \text{softmax} \left(\frac{1}{\sqrt{d}} \mathbf{q}_i^\top \mathbf{k}_j \right) \mathbf{v}_j$$

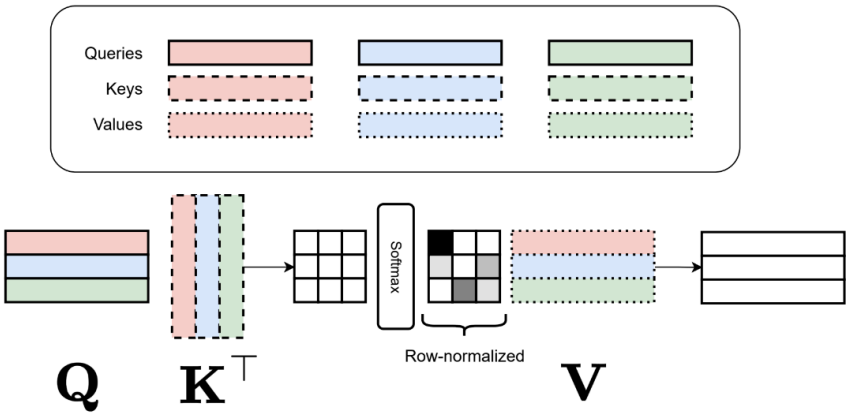
Vectorized version
Here we stack the tokens across the rows,

$$\underset{(n,q)}{\mathbf{Q}} = \mathbf{X}\mathbf{W}_q, \quad \underset{(n,q)}{\mathbf{K}} = \mathbf{X}\mathbf{W}_k, \quad \underset{(n,v)}{\mathbf{V}} = \mathbf{X}\mathbf{W}_v$$

$$\underset{(n,v)}{\mathbf{H}} = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{q}} \right) \mathbf{V}.$$

where the hyper-parameters are q and v. When the layer is applied to a batch of elements (e.g., sentences), it computes the attention function independently for every element of the batch (i.e., each token can attend only to tokens in the same sentence).

Visualizing the attention operation



The dictionary analogy

To understand the terminology, consider a Python dictionary d = dict(...). It is a collection of key/values (k,v) pairs, such that for a given query d[q] = v if k is stored inside. If the key does not exists, an error or default value is returned.

We can consider instead a 'soft' variant that always returns a value, by considering the value associated to the most similar key, even if a perfect match does not occur. If the keys, queries, and values are vectors and the distance is the dot product, this is equivalent to SA when replacing the softmax with an argmax over rows!

Multi-head attention

In order to accelerate the process we use N heads working in parallel.

heads = hyperparameters = h

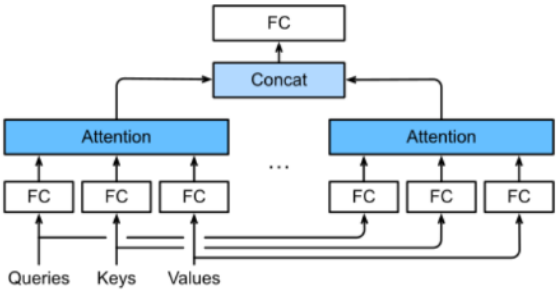
For each head is done the attention operation on 3 different matrices.

So, we now have 3h trainable matrices, or a 3 × h × q tensor assuming q = v.

The previous operation has h separate outputs; we combine them by concatenation over the embedding dimension, and a final reprojection with a trainable output matrix Wo:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_1 & \cdots & \mathbf{H}_h \end{bmatrix} \mathbf{W}_o \,.$$

As hyperparameters, we typically choose an embedding dimension m, an output size o, and a number of heads h, and we set q = v = m//h for all heads.

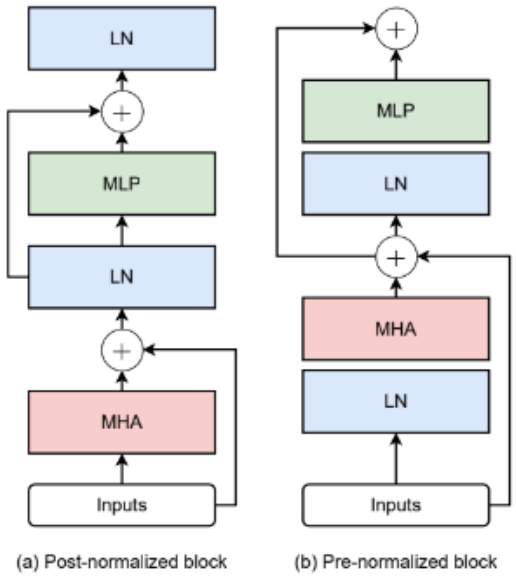


The transformer block

The transformer is an architecture using a concatenation of self attention layers.
"Attention is all you need" means that we can do everything (or almost everything) using only attention layers.
In transformers, the MHA layer is always used inside a more complex block, called the transformer block. Originally, this was composed of a MHA layer, two layer normalization operations, two residual connections, and a so-called position-wise network as follows:

1. Start with a MHA layer: $\mathbf{H} = \text{MHA}(\mathbf{X})$.
2. Add a residual connection and a layer normalization operation: $\mathbf{H} = \text{LayerNorm}(\mathbf{H} + \mathbf{X})$.
3. Apply a fully-connected model $g(\cdot)$ on each row: $\mathbf{F} = g(\mathbf{H})$.
4. Do again step 2: $\mathbf{H} = \text{LayerNorm}(\mathbf{F} + \mathbf{H})$.

(3) is a small (typically 2 layers) neural network.



Many other variants are now common, depending on the application and computational considerations, e.g.:

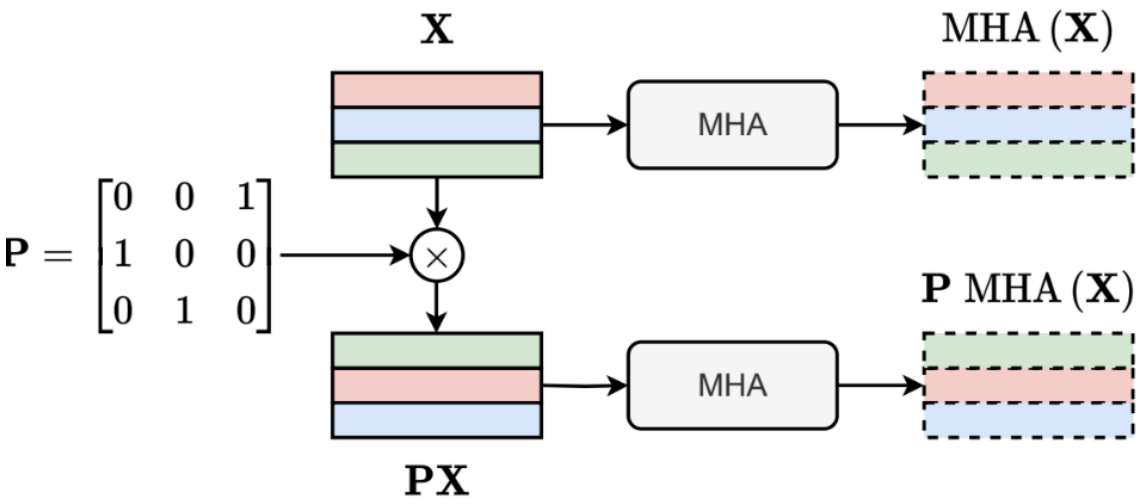
• Parallel variants perform the MHA and MLP operations in parallel, i.e.,
H = H + MLP(H) + MHA(H). In this way, the initial and final projections of the MLP and MHA layers can be fused
• Q/K normalized variants add additional LN operations over the keys and queries (ibidem).
• Multi-query variants share the same keys and values over different heads to save computations.

Designing the transformer

Until this point we have ignored the positions (so it's like we're dealing with sets, not sequences)

Positional embedding

Permutations



Transformers are permutation equivalent so: MHA(PX) = P · MHA(X).
So, In a MHA layer, their ordering is lost, because the layer is equivariant to the ordering (similar to the GAT layer for graphs).
This is not a good property to have for sequences.

Positional embeddings

Before the first MHA layer, we concatenate or sum to the input $\mathbf{X}$ a matrix of positional embeddings $\underset{(n,e)}{\mathbf{E}}$:

$$\mathbf{X}' = [\mathbf{X} \,\|\, \mathbf{E}] \quad \textit{or} \quad \mathbf{X}' = \mathbf{X} + \mathbf{E}\,,$$

where each row $[\mathbf{E}]_i$ should *uniquely* encode the position of every element of the sequence.
Using this strategy, we 'break' the equivariance:

$$\mathrm{MHA}(\mathbf{PX} \,\|\, \mathbf{E}) \neq \mathbf{P} \cdot \mathrm{MHA}(\mathbf{X} \,\|\, \mathbf{E})\,.$$

Simple positional embedding

We can encode the position for a sequence of maximum length p with a one-hot vector of dimension p, e.g.:

$$\mathbf{E}_0 = [1, 0, 0, \ldots]\,, \quad \mathbf{E}_1 = [0, 1, 0, \ldots]\,, \quad \mathbf{E}_2 = [0, 0, 1, \ldots]\,, \quad \cdots\,.$$

Or with a single increasing scalar:

$$\mathbf{E}_0 = [0/p]\,, \quad \mathbf{E}_1 = [1/p]\,, \quad \mathbf{E}_2 = [2/p]\,, \quad \cdots\,.$$

Both strategies are not particularly good empirically.

Trainable positional embeddings

We can *learn* the positional embeddings using the `tf.keras.layers.Embedding` layer:

- To each position i we associate an embedding vector of fixed dimension.
- The embeddings are trained with the rest of the network.

Note that we need to fix the maximum length of the sentence. For longer sentences, we need to linearly interpolate the set of vectors up to a larger dimension (this is the strategy used in BERT and the Vision Transformer described below).

Sinusoidal embeddings (in the original transformer paper)

Consider a single sinusoidal function of frequency ω:

$$\mathbf{E}_i = [\sin(i\omega)] \ .$$

We can interpret this as a clock with frequency ω: for two points inside a single rotation, it will give us their relative distance. For other points, the distance will be precise modulo the frequency.

Multiple sinusoidal embeddings

To uniquely identify any possible position, we can consider multiple sinusoids, each with a frequency $\omega_j, j = 1, \ldots, e$:

$$\mathbf{E}_i = [\sin(i\omega_0), \sin(i\omega_1), \ldots, \sin(i\omega_e)] \ .$$

You can think of this as a clock with e different hands, each rotating at its own frequency. This is a nice representation because it can possibly generalize to any length, without the need to impose a maximum length *a priori*.

Choosing the frequency

An empirically good choice for the frequencies (popularized by (Vaswani et al., 2017)) is:

$$\omega_j = \frac{1}{10000^{j/e}} \ .$$

For $j = 0$, this has frequency 2π. For $j = e$, this has frequency $10000 \cdot 2\pi$. In the middle, the frequency are increasing at a geometric progression.
To reduce the number of parameters, it is also common to *sum* the positional encodings instead of concatenating (in which case the dimension e is equal to d):

$$\mathbf{X}' = \mathbf{X} + \mathbf{E} \ .$$

Final version

A popular extension is to alternate sines and cosines of the same frequency:

$$[\mathbf{E}]_{i,2j} = \sin\left(\frac{i}{10000^{2j/e}}\right), \tag{5}$$

$$[\mathbf{E}]_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/e}}\right). \tag{6}$$

One important property of this encoding is that it is possible to translate an encoding via matrix multiplication:

$$[\mathbf{E}]_{i+p} = [\mathbf{E}]_i \mathbf{T}(p) \quad \text{for some } \mathbf{T}(p).$$

See https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ and references therein.

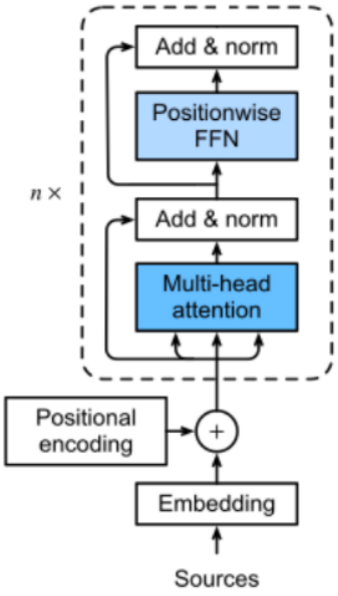
Relative positional embeddings

Another possibility is using relative positional embeddings. In this case, we modify the attention function to make it depend on the relative distance $i - j$ between tokens.
For example, attention with linear biases[4] (ALiBi) adds trainable biases b_{ij}:

$$\alpha(\mathbf{x}_i, \mathbf{x}_j, i - j) = \mathbf{x}_i^\top \mathbf{x}_j + b_{ij}. \tag{7}$$

Another common option are rotary position embeddings[5] (RoPE).

The complete transformer model



Classification token

To perform classification or regression, we can apply a final global pooling on the n tokens and one or more fully-connected layers.
An alternative that is empirically found to work well is the class token, which is an additional *trainable* token $\mathbf{c}$ added to the input matrix:

$$\underset{(n+1,d)}{\mathbf{X}'} = \begin{bmatrix} \mathbf{X} \\ \mathbf{c}^\top \end{bmatrix}.$$

The transformer model is applied to the matrix $\mathbf{X}'$ as input ($\mathbf{H} = \text{Transformer}(\mathbf{X}')$), and classification is performed on its last row:

$$\mathbf{y} = \text{softmax}(\mathbf{W}^\top [\mathbf{H}]_{n+1}).$$

Encoder-decoder models

The original transformer model was a more general model defined for sequence to sequence (seq2seq) tasks, such as machine translation (variable number of tokens in inputs and in output).
It performed an encoding of the input sequence, which was then decoded by a second, masked transformer to generate the output sequence. In order to understand it, we need to introduce two further mechanisms: masked attention and cross-attention.
For this reason, the model described before is sometimes called an encoder-only Transformer.

Building a causal transformer

In order to build a causal transformer variant, we can replace the SA layer with a masked variant:

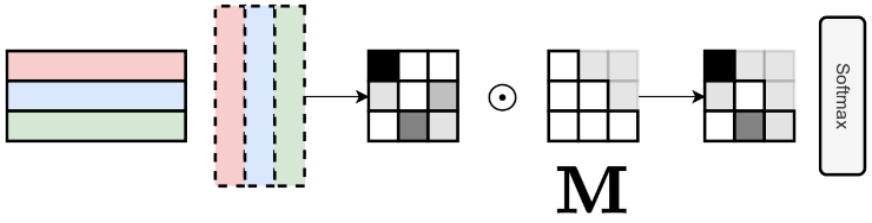
$$\mathbf{H} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M}}{\sqrt{q}}\right)\mathbf{V}.$$

where $\mathbf{M}$ has a triangular structure:

$$M_{ij} = \begin{cases} 1 & \text{if } j \leq i \\ -\infty & \text{otherwise} \end{cases}. \tag{8}$$

In practice we can use very small numbers, e.g., 10^{-10}.

Visualizing the masking operation



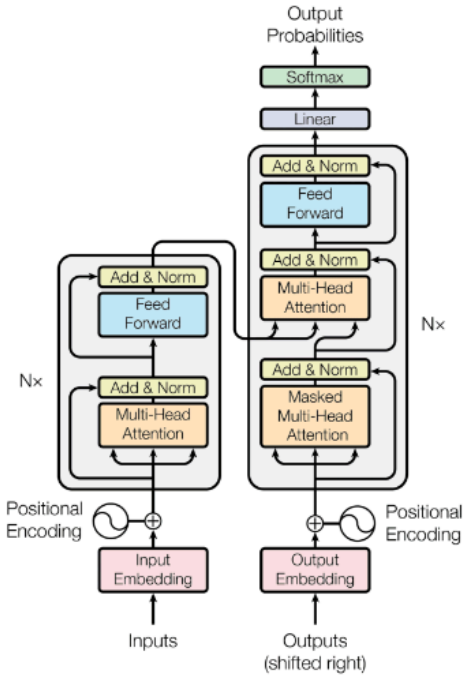
Cross-attention

Given *two* sets $\mathbf{X}$ and $\mathbf{Z}$, cross attention is defined as:

$$\mathrm{CA}(\mathbf{X}, \mathbf{Z}) = \mathrm{softmax}\left(\left(\mathbf{Z}\mathbf{W}_q\right)\left(\mathbf{X}\mathbf{W}_k\right)^\top\right)\mathbf{X}\mathbf{W}_v\,. \tag{9}$$

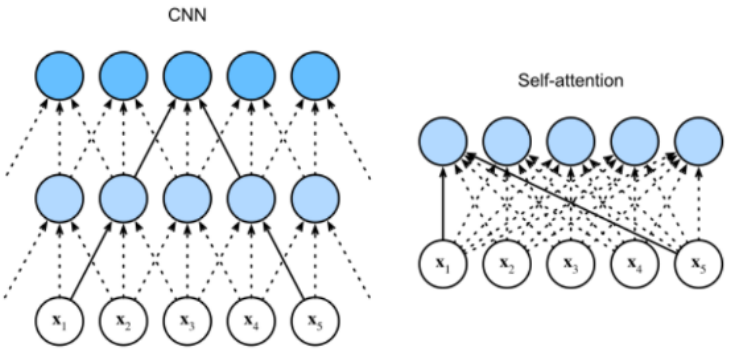
This is a useful operation that can combine information coming from different streams of information (e.g., audio-visual datasets).

Full encoder-decoder architecture



Computational considerations

Comparing convolutive layers and MHA layers



Computational cost comparison

Consider a 1D convolutional operation $\underset{(n,d)}{\mathbf{H}} = \text{Conv1D}(\underset{(n,d)}{\mathbf{X}})$ with a filter size of k. Computing the output requires $\mathcal{O}(nkd^2)$ operations.

Self-attention (with one head) requires $\mathcal{O}(nd^2)$ for computing keys, queries, and values, and $\mathcal{O}(n^2d)$ for computing the output. The n^2 term limits the applicability to long sequences, unless more advanced models are used (e.g., $n < 512$ in many text models).

However, a single layer of MHA has a receptive field of n, while the convolutive layer has a receptive field of k.

Designing linear transformers

There is a large interest in designing variants (or approximations) of SA that scale linearly in n.

A popular class of methods relies on kernel functions. We first rewrite SA for a single token as:

$$\mathbf{h}_i = \frac{\sum_j \alpha(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j}{\sum_j \alpha(\mathbf{q}_i, \mathbf{k}_j)} .$$

where in our final model we had (ignoring scalar factors) $\alpha(\mathbf{x}, \mathbf{y}) = \exp(\mathbf{x}^\top \mathbf{y})$. Note that any positive-definite $\alpha(\cdot, \cdot)$ is a valid kernel function (as in, e.g., support vector machines).

A refresher on kernel functions

For some specific kernel functions, it is possible to rewrite them as a dot product in a different *finite-dimensional*, parameter-free feature space $\phi(\mathbf{x})$:

$$\alpha(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^\top \phi(\mathbf{y}) . \tag{10}$$

(This is always possible if allowing for infinite-dimensional spaces, like in the exponential case.)

For example, $\alpha(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^\top \mathbf{y} + c)^p$ can be rewritten as (10) with $\phi(\mathbf{x})$ containing all terms $x_1^{i_1} x_2^{i_2} \dots x_d^{i_d} y_1^{j_1} y_2^{j_2} \dots y_d^{j_d}$ with $\sum_k i_k + j_k = p$.

Linearized transformers

In this case, we can rewrite the attention operation as:

$$\mathbf{h}_i = \frac{\sum_j \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j}{\sum_j \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)} = \frac{\phi(\mathbf{q}_i)^\top \overbrace{\sum_j \phi(\mathbf{k}_j)\mathbf{v}_j^\top}^{\mathbf{S}}}{\phi(\mathbf{q}_i)^\top \underbrace{\sum_j \phi(\mathbf{k}_j)}_{\mathbf{Z}}}. \tag{11}$$

where $\mathbf{S}$ and $\mathbf{Z}$ are now shared across all tokens and can be computed only once, making the operation linear in n.

Recurrent formulation

In the autoregressive case, we replace $\sum_j$ with the masked version $\sum_{j=1}^{i}$. Define the partial sums $\mathbf{S}_i = \sum_{j=1}^{i} \phi(\mathbf{k}_j)\mathbf{v}_j^\top$ and $\mathbf{Z}_i = \sum_{j=1}^{i} \phi(\mathbf{k}_j)$. We can rewrite the previous equation as:

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \mathbf{S}_i}{\phi(\mathbf{q}_i)^\top \mathbf{Z}_i}. \tag{12}$$

Since $\mathbf{S}_i = \mathbf{S}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i^\top$ and $\mathbf{Z}_i = \mathbf{Z}_{i-1} + \phi(\mathbf{k}_i)$, each new generated token requires a *constant* amount of time, making this recurrent formulation attractive for the generation of long sequences.

Memory considerations

Materializing the $\mathbf{QK}^\top$ matrix also requires $\mathcal{O}(n^2)$ memory, which is the main bottleneck in existing GPUs and similar hardware.
Modern implementations (e.g., FlashAttention[6]) can avoid this by processing tokens in multiple chunks. This can be done by storing intermediate values on the denominator of the softmax, and only applying the normalization at the end (lazy softmax).

Practical transformer models

1) **text transformer:** BERT-like, GPT-like. -> These models are called contextual embeddings because the same word in different sentences can be encoded to different vectorial representations.
Because these models are trained from the raw text alone (no specific targets) they are called self-supervised models. Their strengths is that scaling laws for transformers are empirically better than for other models (i.e., they benefit more from increasing the dataset by order of magnitude).

In natural language processing, this is also shown by the emergence of paradigms like text-prompting and zero-shot learning.

2) **Vision, Audio and Graph Transformers**

Deep Generative Models

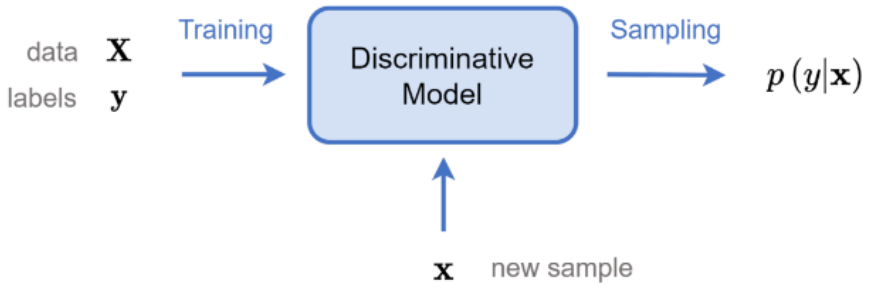
Generative model is an important branch of machine learning that complements the more widely studied discriminative modeling.

Classic neural networks may not be able to produce a probability distribution that represents inherent structure in the data and to generate examples outside of the training set.

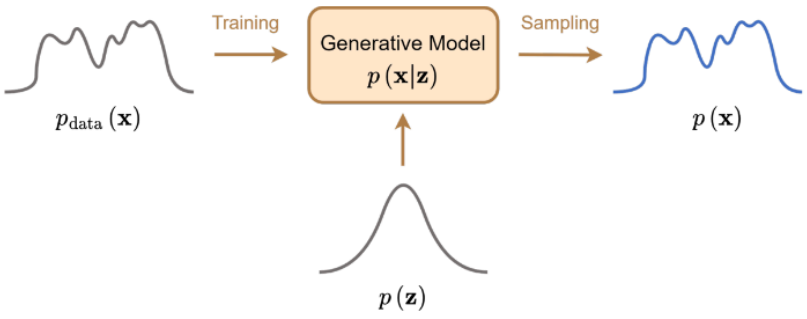
Merging generative modeling and neural networks leads to deep generative models, which are built by stacking layers as well.

Where generative deep learning comes from
It comes from:

- generative modeling: estimates the probability of observing observation x (p(x)), differently from discriminative which estimates the probability of a label y given observation x (p(y|x))
- deep learning



So the discriminative modeling process is formed by training data as input and the prediction as output.



While the generative modeling process is formed by training data as input and a new sample as output. The generative model must be probabilistic rather than deterministic and must include a stochastic element affecting the individual samples. It is performed with an unlabeled dataset. In labeled dataset we can use y to estimate p(x|y).

Generative modeling framework

- Complex and large-scale dataset X of observation x
- x drawn from some unknown distribution p_data

- a generative model estimates the distribution of p(x), thus generating observations that appear to have been drawn from p_data

The fundamental laws of a generative model
We can say that a generative model is impressive if it satisfies the following fundamental rules:
1 It must generate new samples that appear to have been drawn from p_data;
2 It must generate new samples that did not exist before in X.

Why generative modeling is attractive
• Generative processes are able to express physical laws, while considering meaningless details as noise.
• Generative models are usually highly intuitive and interpretable.
• Generative processes express causal relations, thus being able to generalize much better to new situations than mere correlations.

Some applications tasks: super-resolution, inpainting, compressed sensing, denoising

Probabilistic generative models

Reasoning about certainty

ML is all about making predictions. We might use probability in many ML applications, as well in several real-life situations. Probability gives us a formal way of reasoning about our level of certainty.
As an example, consider distinguishing cats and dogs based on photographs. Although it might sound simple, the difficulty of the problem may depend on the resolution of the image.

• If we are completely sure that the image depicts a cat, we say that the probability that the corresponding label y is “cat”, denoted p(y = “cat”) equals 1.
• If we had no evidence to suggest that y = “cat” or that y = “dog”, then we might say that the two possibilities were equally likely expressing this as p(y = “cat”) = p(y =“dog”) = 0.5.
• If we were reasonably confident, but not sure that the image depicted a cat, we might assign a probability 0.5 < p(y = "cat") < 1.

Probability is a flexible language for reasoning about our level of certainty, and it can be applied effectively in several contexts in ML.

Stochastic events and sample space
In probability theory, an observable phenomenon, or occurrence, or stochastic event x is considered based on the possibility that it may occur or not.
An event can be seen as the outcome of an experiment, also said a random variable (RV).
The set of all the possible experiments is the abstract probability space, or sample space, Ω.
Thus, the sample space contains all the values an observation x can take.

Probability and its interpretations

The probability density function, p (x), is a function that maps a point x in the sample space into a number between 0 and 1.
A parametric model, pθ (x), is a family of density functions that can be described using a finite number of parameters, θ.
There are actually at least two different interpretations of probability.
• Frequentist interpretation: to represent long run frequencies of events.
• Bayesian interpretation: to quantify our uncertainty about something.

Set of probabilities

Random variables can be described in terms of a set of probabilities.

- The **joint probability** of $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ represents the probability that both the events occur simultaneously:

$$p(x, y) = p(x \cap y) = p(x|y)\, p(y). \tag{1}$$

- Given $p(x, y)$, we can define the **marginal probability** of x as:

$$p(x) = \sum_{x \in \mathcal{Y}} p(x, y) = \sum_{x \in \mathcal{Y}} p(x|y)\, p(y) \tag{2}$$

  also known as *sum rule*. We can define $p(y)$ similarly.

- The **conditional probability** of x, given y is true, is defined as:

$$p(x|y) = \frac{p(x, y)}{p(y)} \qquad \text{if } p(y) > 0. \tag{3}$$

Likelihood

The **likelihood** $\mathcal{L}(\theta|\mathbf{x})$ of a parameter set θ is a function that measures the plausibility of θ, given some observed point $\mathbf{x}$.

It is defined as the value of the density function parameterized by θ, i.e.: $\mathcal{L}(\theta|\mathbf{x}) = p_\theta(\mathbf{x})$.

For a whole dataset $\mathbf{X}$ of independent observation we have:

$$\mathcal{L}(\theta|\mathbf{X}) = \prod_{\mathbf{x} \in \mathbf{X}} p_\theta(\mathbf{x}).$$

Since this product can be quite computationally difficult to work with, the **log-likelihood** $\ell(\theta|\mathbf{X})$ is often used instead:

$$\ell(\theta|\mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} \log p_\theta(\mathbf{x}).$$

Maximum likelihood estimation

The **maximum likelihood estimation** allows to estimate the set of parameters θ of a density function, $p_\theta(x)$, that are most likely to explain some observed data $\mathbf{X}$:

$$\hat{\theta} = \arg\max_{\theta} \mathcal{L}(\theta|\mathbf{X})$$

where $\hat{\theta}$ is the maximum likelihood estimate (MLE).

Bayesian generative modeling

One of the most popular examples of generative model in machine learning is represented by the Naive Bayes algorithm.

The basic assumption is that the features (i.e., RVs), x_k, for $k = 1, \ldots, l$, in the feature vector $\mathbf{x}$ are statistically independent, i.e.:

$$p(x_i, x_j) = p(x_i)\, p(x_j), \qquad \forall i, j = 1, \ldots, l, \quad i \neq j \tag{4}$$

Bayes Rule

The uncertainty of an output variable y_i, $i = 1, \ldots, M$, given the value of a feature vector $\mathbf{x}$, as expressed by the conditional probability $p(y_i|\mathbf{x})$, can be equivalently expressed as:

$$p(y_i|\mathbf{x}) = \frac{p(\mathbf{x}|y_i)\, p(y_i)}{p(\mathbf{x})}. \tag{5}$$

In (5), $p(\mathbf{x})$ and $p(y_i)$ are also denoted as prior probabilities, $p(y_i|\mathbf{x})$ as posterior probability and $p(\mathbf{x}|y_i)$ as likelihood.

The joint pdf can be written as a product of l marginals, i.e.:

$$p(\mathbf{x}|y_i) = \prod_{k=1}^{l} p_\theta(x_k|y_i), \quad i = 1, 2, \ldots, M.$$

This leads to a total of $2l$ unknown parameters to be estimated per class.

The role of Neural Networks in Deep Generative Models

Challenges of generative modeling

Generative modeling theory has been widely applied to traditional machine learning algorithms (e.g., naïve Bayes classifiers).

However, 1) in presence of a big amount of data, it is not easy for a model cope with the high degree of conditional dependence between features.

Moreover, 2) the larger the input data space, the more difficult it is to produce an output that satisfies the generation constraints.

Neural networks are the key to solving both of these challenges due to its ability to form its own features in a lower-dimensional space.

Dealing with complex unstructured data

Modern applications cannot ignore the complexity of real data, which may exhibit:
• High-dimensional, complicated probability distributions;
• Damaged or missing information;
• Partially labeled or completely unlabeled data;
• High resolutions and QoS constraints;
• Multimodality.

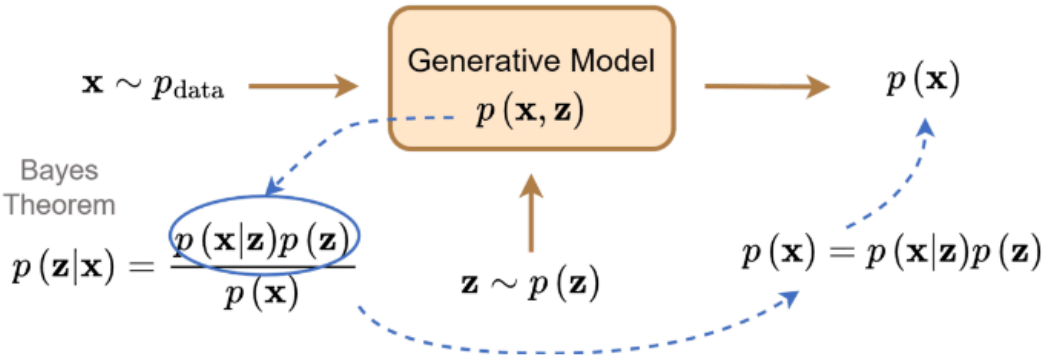
Neural networks for unstructured data

Neural networks rely on multiple stacked layers of processing units to learn high-level representations from unstructured data.
The real power of neural networks, especially with regard to generative modeling, comes from its ability to work with unstructured data.
Most often, we want to generate unstructured data such as new images or original strings of text.
This is the reason why deep learning has had such a profound impact on the field of generative modeling.

Dealing with large input data space

As said, the larger the input data space, the more difficult it is to produce an output that satisfies the generation constraints.
To reduce the input space and improve the generation perfomance, it is possible to leverage the capability of neural networks known as Representation learning.
Representation learning refers to the ability to describe each observation in the training set using some low-dimensional latent space.
Then, a function is learned to map a point in the latent space into the original domain.
Neural networks provides the ability to learn the often highly complex mapping function in a variety of ways.
So each point in the latent space is the representation of some high-dimensional image.
One of the main advantages is that we can perform operations within the more manageable latent space, while affecting the high-level properties of the image.

Neural Network parametrization



Steps for generating new data

The process to generate new data by using neural networks usually follows the next steps:

1 Sample from a known distribution

2 Optimize a likelihood-based loss (or any implicit variant).

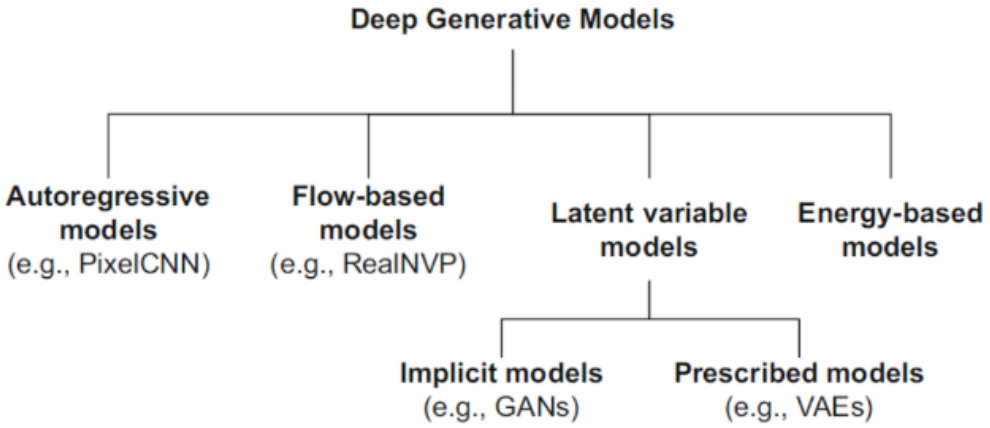
3 Train the model.

4 Decode a representation to draw a new sample.

The new sample must belong to the underlying distribution of the data points and it must never have been seen before.

Taxonomy of deep generative models

How to formulate deep generative models



Autoregressive models

The first group of deep generative models utilizes the idea of autoregressive (AR) modeling, i.e., the distribution over $\mathbf{x}$ is represented in an autoregressive manner:

$$p(\mathbf{x}) = p(x_0) \prod_{i=0}^{D} p(x_i|\mathbf{x}_{<i})$$

where $\mathbf{x}_{<i}$ denotes all $\mathbf{x}$'s up to the index i.

Causal convolutions will solve the computationally inefficiency of conditional distributions $p(x_i|\mathbf{x}_{<i})$.

Flow-based models

These models rely on a change of variable to express a density $p(\mathbf{x})$ with an invertible transformation:

$$p(\mathbf{x}) = p(\mathbf{z} = f(\mathbf{x})) \left|\mathbf{J}_{f(\mathbf{x})}\right|,$$

where $\mathbf{J}_{f(\mathbf{x})}$ denotes the Jacobian matrix, and the function $f(\cdot)$ can be *parameterized* by a deep neural network.

Several attempts have been made in the literature to perform the change of variables.

All generative models that take advantage of the change of variables formula are referred to as flow-based models.

Latent variable models

The idea behind latent variable models is to assume a lower-dimensional latent space and the following generative process:
z ~ p (z)
x ~ p (x|z)
meaning that the latent variables correspond to hidden factors in data, and the conditional distribution p (x|z) could be treated as a generator.
The most widely known latent variable model is the probabilistic Principal Component Analysis (pPCA) where p (z) and p (x|z) are Gaussian distributions, and the dependency between z and x is linear.
A nonlinear extension of the pPCA with arbitrary distributions is the Variational AutoEncoder (VAE) framework

Implicit latent variable models

So far, AR models, flows, pPCA-based models and VAEs are probabilistic models with the objective function being the log-likelihood function that is closely related to using the Kullback-Leibler divergence between the data distribution and the model distribution.
A different approach utilizes an adversarial loss in which a discriminator D (·) determines a difference between real data and synthetic data provided by a generator in the implicit form, namely:
p (x|z) = δ (x − G (z)), where δ (·) is the Dirac function.

This group of models is called implicit models, and Generative Adversarial Networks (GANs) become one of the first successful deep generative models for synthesizing realistic looking objects (e.g., images).

Energy Based Models

Physics provide an interesting perspective on defining a group of generative models through defining an energy function, $E(\mathbf{x})$, and, eventually, the Boltzmann distribution:

$$p(\mathbf{x}) = \frac{\exp\{-E(\mathbf{x})\}}{Z},$$

where $Z = \sum_{\mathbf{x}} \exp\{-E(\mathbf{x})\}$ is the partition function.

In other words, the distribution is defined by the exponentiated energy function that is further normalized to obtain values between 0 and 1 (i.e., probabilities) [12].

Applications of Deep generative models

Data Generation, data augmentation, denoising, inpainting, super-resolution, editing, image modality translation, generative semantic communications, generative biological twin…

GAN: Generative Adversarial Network

Steps for generating new data

So far, we have seen that the process to generate new data includes the following steps:
1 Sample from a known distribution.
2 Optimize a likelihood-based loss.
3 Train the model.
4 Decode a representation to draw a new sample.
The new sample must belong to the underlying distribution of the data points and it must never have been seen before.

Learning the density estimation
Consider an input x from a data distribution p_data.
A deep neural network model can be used as a sampler qφ (z) = DNN (z; φ), with z ~ p (z).
The inference model aims at inducing a density function p_model.
Both p_data and p_model are unknown, so we can only draw samples.
We need to learn the inference parameters φ to make p_model as close as possible to p_data.

Distance measures for implicit models

To measure the distance between distributions we usually adopt $\mathcal{D}_{\text{KL}}\left(p_{\text{data}}||p_{\text{model}}\right)$, which returns the objective $\mathrm{E}_{\mathbf{x}\sim p_{\text{data}}}\left\{\log p_{\boldsymbol{\phi}}\left(\mathbf{x}\right)\right\}$.

If $p_{\boldsymbol{\phi}}\left(\mathbf{x}\right)$ is not *explicit*, a distance measure different from the maximum likelihood must be adopted, e.g.:

- Jensen Shannon divergenge (JSD), maximum mean discrepancy (MMD), etc.

Comparison of divergence measures

When there is a mismatch between the data distribution and the model, different objective functions can lead to very different results

Direction of the KL divergence

Figure 5: The KL divergence is asymmetric. Suppose we have a distribution $p\left(x\right)$ and wish to approximate it with another distribution $q\left(x\right)$. We have the choice of minimizing either $\mathcal{D}_{\text{KL}}\left(p||q\right)$ or $\mathcal{D}_{\text{KL}}\left(q||p\right)$. The choice of which direction of the KL divergence to use is problem-dependent. On the left, for $\mathcal{D}_{\text{KL}}\left(p||q\right)$, we select a q that has high probability where p has high probability. When p has multiple modes, q chooses to blur the modes together, in order to put high probability mass on all of them. On the center, for $\mathcal{D}_{\text{KL}}\left(q||p\right)$, we select a q that has low probability where p has low probability. When p has multiple modes that are sufficiently widely separated, as in this figure, the KL divergence is minimized by choosing a single mode, in order to avoid putting probability mass in the low-probability areas between modes of p. Here, we illustrate the outcome when q is chosen to emphasize the left mode. If the modes are not separated by a sufficiently strong low probability region, then this direction of the KL divergence can still choose to blur the modes [8]. Using the JSD is like minimizing the reverse KL $\mathcal{D}_{\text{KL}}\left(q||p\right)$ [9].

Generator and discriminator models

Similarly to the VAE, the generative adversarial network is composed of two models that in this case are known as generator $G\left(\cdot\right)$ and discriminator (or *critic*) $D\left(\cdot\right)$.

$D\left(\cdot\right)$ aims at maximizing the log-likelihood for the binary classification problem:

- original data: real (1)
- generated data: fake (0)

$G\left(\cdot\right)$ aims at minimizing the log-probability of its samples being classified as "fake" by the discriminator $D\left(\cdot\right)$.

Generator and discriminator can be seen as the players of a minmax game:

$$\min_{G}\max_{D}\mathrm{E}_{\mathbf{x}\sim p_{\text{data}}}\left\{\log D\left(\mathbf{x}\right)\right\}+\mathrm{E}_{\mathbf{z}\sim p(\mathbf{z})}\left\{\log\left(1-D\left(G\left(\mathbf{z}\right)\right)\right)\right\}$$

Generator network

The generator receives a noise signal z and yields the generated data x:

$$x = G(z; \theta G)$$

A generator has the following properties:
• must be differentiable;
• does not require to be invertible;
• must be trainable for any size of z;
• requires z to have a higher dimension than x;
• can make x conditionally Gaussian given z.

Discriminator network

An optimal $D(\mathbf{x})$ for any $p_{\text{data}}(\mathbf{x})$ and $p_{\text{model}}(\mathbf{x})$ is always described by:

$$D(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_{\text{model}}(\mathbf{x})}$$

Estimating this ratio using supervised learning is the key approximation mechanism used by GANs.

Training GANs

We choose a minibatch stochastic gradient descent (SGD) algorithm, like Adam [1, 10].

We sample simultaneously:

- a minibatch of m training examples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_G(\mathbf{z})$;
- a minibatch of m generated examples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from noise prior $p_{\text{data}}(\mathbf{x})$.

Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_D} \frac{1}{m} \sum_{i=1}^{m} \left\{ \log D\left(\mathbf{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\mathbf{z}^{(i)}\right)\right)\right) \right\}$$

Update the generator by ascending its stochastic gradient:

$$\nabla_{\boldsymbol{\theta}_G} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D\left(G\left(\mathbf{z}^{(i)}\right)\right)\right)$$

Optional: run k steps of the discriminator for every step of the generator.

As a minmax game, equilibrium is a saddle point of the discriminator loss:

$$J_D = -\frac{1}{2} \mathrm{E}_{\mathbf{x} \sim p_{\text{data}}} \log D(\mathbf{x}) - \frac{1}{2} \mathrm{E}_{\mathbf{z} \sim p_G(\mathbf{z})} \log (1 - D(G(\mathbf{z})))$$
$$J_G = -J_D$$

The generator *minimizes* the log-probability of the discriminator being *correct*.

Non-saturating game

As an alternative to the minmax game of (1), we can define the non-saturating game strategy as [10]:

$$J_D = -\frac{1}{2} \mathrm{E}_{\mathbf{x} \sim p_{\text{data}}} \log D(\mathbf{x}) - \frac{1}{2} \mathrm{E}_{\mathbf{z} \sim p_G(\mathbf{z})} \log (1 - D(G(\mathbf{z})))$$
$$J_G = -\frac{1}{2} \mathrm{E}_{\mathbf{z} \sim p_G(\mathbf{z})} \log D(G(\mathbf{z})) \tag{2}$$

where the equilibrium is no longer describable with a single loss.

In this case, the generator *maximizes* the log-probability of the discriminator being *mistaken*.

The non-saturating game is heuristically motivated. Generator can still learn even when discriminator successfully rejects all generator samples.

Key points of GANs

- Fast sampling.
- No inference.
- Notion of optimizing directly for what you care about by using perceptual samples.

How to evaluate generation in GANs

One of the most challenging issues in GANs is related to their evaluation.
Unlike density models, you cannot report explicit likelihood estimates on test sets.
Nonparametric approaches make few assumptions about the overall shape of the estimate being modeled

Kernel density estimators

Assume observations drawn from a density $p(\mathbf{x})$ and consider a small region $\mathcal{R}$ containing $\mathbf{x}$ such that $P = \int_{\mathcal{R}} p(\mathbf{x})\, d\mathbf{x}$ [11].

The probability that K out of N observations lie inside $\mathcal{R}$, if N is large, is $K \simeq NP$.

If the volume of $\mathcal{R}$, V, is sufficiently small, $p(\mathbf{x})$ is approximately constant over $\mathcal{R}$ and $P \simeq p(\mathbf{x})\, V$, thus:

$$p(\mathbf{x}) = \frac{K}{NV}$$

Fixing V and choosing a Parzen window for K, it follows that:

$$p(\mathbf{x}) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{h^D} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right)$$

where k is a kernel that is usually chosen as Gaussian in generative model evaluation.

Inception score

Can we side-step high-dimensional density estimation?
One idea: good generators generate samples that are semantically diverse.
Let us consider a semantics predictor: trained Inception Network v3 p (y|x) and being y one of the 1000 ImageNet classes.
• Each image x should have distinctly recognizable object → p (y|x) should have low entropy.
• There should be as many classes generated as possible → p (y) should have high entropy.

On this basis, it is possible to define the inception score (IS) as

$$\begin{aligned} \mathrm{IS}(\mathbf{x}) &= \exp\left(\mathrm{E}_{\mathbf{x}\sim p_g}\left\{\mathcal{D}_{\mathrm{KL}}\left(p(\mathbf{y}|\mathbf{x})\,||p(\mathbf{y})\right)\right\}\right) \\ &= \exp\left(\mathrm{E}_{\mathbf{x}\sim p_g, \mathbf{y}\sim p(\mathbf{y}|\mathbf{x})}\left\{\log p(\mathbf{y}|\mathbf{x}) - \log p(\mathbf{y})\right\}\right) \\ &= \exp\left(H(\mathbf{y}) - H(\mathbf{y}|\mathbf{x})\right) \end{aligned}$$

Fréchet inception distance

The IS does not sufficiently measure diversity: a list of 1000 images (one of each class) can obtain perfect score.
Fréchet Inception Distance (FID) was proposed to capture more nuances.
FID is computed by embedding an image x into some feature space (2048-dimensional activations of the Inception-v3 pool3 layer), then comparing mean μ and covariance C of those random features.

$$d^2\left((\boldsymbol{\mu}, \mathbf{C}), (\boldsymbol{\mu}_w, \mathbf{C}_w)\right) = \|\boldsymbol{\mu} - \boldsymbol{\mu}_w\|_2^2 + \mathrm{tr}\left(\mathbf{C} + \mathbf{C}_w - 2\left(\mathbf{C}\mathbf{C}_w\right)^{1/2}\right)$$

Deep convolutional GAN

Most GANs today are at least loosely based on the deep convolutional GAN (DCGAN) architecture, whose key insights are:

• It uses batch normalization in most layers of both the discriminator and the generator, with the two minibatches for the discriminator normalized separately. It prevents mode collapse.
• The last layer of the generator and first layer of the discriminator are not batch normalized, so that the model can learn the correct mean and scale of the data distribution.
• The overall network structure is mostly borrowed from the all-convolutional net.
• It contains neither pooling nor “unpooling” layers. When the generator needs to increase the spatial dimension of the representation it uses transposed convolution with a stride greater than 1.
• It uses ReLU activation functions for generator and Leaky-ReLU for discriminator.
• It uses tanh output nonlinearity for generator and sigmoid for discriminator.
• The use of the Adam optimizer rather than SGD with momentum.

Summary of the main points of the DCGAN

• Incredible samples for any generative model
• GANs could be made to work well with architecture details
• Perceptually good samples and interpolations
• Representation learning
• Problems to address:
− Unstable training
− Brittle architecture / hyperparameters

Summary of classic tricks

1 Normalize the inputs between −1 and 1 and use tanh as the last layer of the generator output
2 Use a modified loss function: max log D would be preferred to min (log 1 − D) as it avoids vanishing gradient.
3 Use a spherical distribution for z: interpolations along circles are better than those along a straight line.
4 For the batch normalization, each mini-batch needs to contain only all real images or all generated images.
5 Avoid sparse gradients (ReLU, MaxPooling), which may affect the stability of the GANs.
6 Use DCGAN when yo can, otherwise a hybrid model, e.g., KL+GAN or VAE+GAN.
7 Use stability tricks from reinforcement learning.
8 Use ADAM optimizer.
9 Track failure early: check norm of the gradients.
10 Add noise to inputs to improve performance.
11 Use dropout in G in both train and test phases.

Train with labels

Labels improve subjective sample quality.
Learning a conditional model p (y|x) often gives much better samples from all classes than learning p (x) does.
Even learning p (x, y) makes samples from p (x) look much better to a human observer.
This defines three categories of models (no labels, trained with labels, generating condition on labels) that should not be compared directly to each other.

One-side label smoothing

To encourage the discriminator to estimate soft probabilities rather than to extrapolate to extremely confident classification, we can use the one-sided label smoothing.
The idea is to replace the target for the real examples, e.g., 1, with a value slightly less than 1, such as .9.
This prevents extreme extrapolation behavior in the discriminator.

It is important to not smooth the labels for the fake samples, e.g., 0.

The optimal discriminator function is:

$$D(\mathbf{x}) = \frac{(1-\alpha)\, p_{\text{data}}(\mathbf{x}) + \beta p_{\text{model}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_{\text{model}}(\mathbf{x})}$$

where β reinforces current generator behavior.

When $\beta \to 0$, then smoothing by α does nothing but scale down the optimal value of the discriminator. When $\beta \neq 0$, the shape of the optimal discriminator function changes.

Label smoothing is an excellent regularizer in the context of convolutional networks for object recognition [17, 10].

Label smoothing also prevents discriminator from giving very large gradient signal to generator.

Virtual batch normalization
The main purpose of batch normalization is to improve the optimization of the model, by reparameterization so that the mean and variance of each feature are determined by a single mean and a single variance parameter.

The normalization operation is part of the model, so that backpropgation computes the gradient of features that are defined to always be normalized.

Batch normalization is very helpful, but for GANs has a few unfortunate side effects.

The use of a different minibatch of data to compute the normalization statistics on each step of training results in fluctuation of these normalizing constants.

When minibatch sizes are small these fluctuations can become large enough that they have more effect on the image generated by the GAN than the input z has.

To mitigate this problem, reference batch normalization runs the network once on a mini-batch of inizial reference examples and once on the current minibatch of examples.

To avoid that the model overfits to the reference batch, the virtual batch normalization proposes to compute statistics for each example using the union of that example and the reference batch.

Balancing generator and discriminator
Usually the discriminator “wins”.
This is a good thing. The theoretical justifications are based on assuming D (·) is perfect.
Usually D (·) is bigger and deeper than G (·).
Sometimes run D (·) more often than G (·). Mixed results.
Do not try to limit D (·) to avoid making it “too smart”.
Use non-saturating cost and label smoothing.

Mode covering vs mode seeking: tradeoffs
For compression, one would prefer to ensure all points in the data distribution are assigned probability mass.
For generating good samples, blurring across modes spoils perceptual quality because regions outside the data manifold are assigned non-zero probability mass.
Picking one mode without assigning probability mass on points outside can produce “better-looking” samples.
Caveat: More expressive density models can place probability mass more accurately.
Example: Using mixture of Gaussians as opposed to a single isotropic Gaussian.

Mode collapse
Mode collapse may arise because the maximin solution to the GAN game is different from the minimax solution:

$$\min_G \max DV(G, D) \neq \max_D \min_G V(G, D)$$

Addressing mode collapse with minibatch features
One attempt to address mode collapse is using minibatch features.
The basic idea is to allow the discriminator to compare an example to a minibatch of generated samples and a minibatch of real samples.

By measuring distances to these other samples in latent spaces, the discriminator can detect if a sample is unusually similar to other generated samples.
Minibatch features work well!

Addressing mode collapse with unrolled GANs

The idea of unrolled GANs is to build a computational graph describing k steps of learning in the discriminator, then backpropagate through all k of these steps of learning when computing the gradient on the generator.

Non-convergence
Optimization algorithms often approach a saddle point or local minimum rather than a global minimum
Game-solving algorithms may not approach an equilibrium at all.
Exploiting convexity in function space, GAN training converges if we can modify G (·) and D (·) directly, which can be represented as highly nonconvex parametric functions.
Oscillations can be produced in long training, generating very many different categories of samples, without clearly generating better samples.
Most severe mode collapse can be caused by non-convergence.

Wasserstein GAN

Main properties
With a few changes, we want to train GANs that have the following two properties:
• A meaningful loss metric that correlates with the generator’s convergence and sample quality.
• Improved stability of the optimization process.
Specifically, a new loss function is introduced instead of the binary cross-entropy for both the discriminator and the generator, resulting in a more stable convergence of the GAN.

Earth Mover distance

We consider another distance measure besides KL and JSD, known as Earth Mover (EM) distance or Wasserstein-1:

$$\mathrm{W}\left(p\left(x\right),p\left(\tilde{x}\right)\right)=\inf_{\gamma\in\prod\left(p(x),p(\tilde{x})\right)}\mathrm{E}_{(x,\tilde{x})\sim\gamma}\left\{\left\|x-\tilde{x}\right\|\right\} \tag{3}$$

where $\prod\left(p\left(x\right),p\left(\tilde{x}\right)\right)$ denotes the set of all joint distributions $\gamma\left(x,\tilde{x}\right)$ whose marginals are respectively $p\left(x\right)$ and $p\left(\tilde{x}\right)$.

Intuitively, $\gamma\left(x,\tilde{x}\right)$ indicates how much "mass" must be *transported* from x to $\tilde{x}$ in order to transform the distributions $p\left(x\right)$ into the distribution $p\left(\tilde{x}\right)$.

Kantorovich-Rubinstein duality

$\mathrm{W}\left(p\left(x\right),p\left(\tilde{x}\right)\right)$ in (3) might have nicer properties when optimized than $\mathrm{JS}\left(p\left(x\right),p\left(\tilde{x}\right)\right)$ [20].

However, the infimum in (3) is highly intractable.

Using the Kantorovich-Rubinstein duality [22, 20], we have that:

$$\mathrm{W}\left(p\left(x\right),p\left(\tilde{x}\right)\right)=\sup_{\left\|f\right\|_{L}\leq1}\mathrm{E}_{x\sim p(x)}\left\{f\left(x\right)\right\}-\mathrm{E}_{\tilde{x}\sim p(\tilde{x})}\left\{f\left(\tilde{x}\right)\right\}$$

where the supremum is over all the 1-Lipschitz functions $f:\mathcal{X}\rightarrow\mathbb{R}$.

Search over joint distributions is now a search over 1-Lipschitz functions.

Gradient of the Wasserstein loss

If we replace $\left\|f\right\|_{L}\leq1$ for $\left\|f\right\|_{L}\leq K$ (consider K-Lipschitz for some constant K), then we end up with $K\cdot\mathrm{W}\left(p\left(x\right),p\left(\tilde{x}\right)\right)$.

Therefore, if we have a parameterized family of functions $\left\{f_{w}\right\}_{w\in\mathcal{W}}=D_{w}\left(\cdot\right)$ that are all K-Lipschitz for some K, we could consider solving the problem [20]:

$$\max_{w\in\mathcal{W}}\mathrm{E}_{x\sim p(x)}\left\{D_{w}\left(x\right)\right\}-\mathrm{E}_{\tilde{x}\sim p(\tilde{x})}\left\{D_{w}\left(\tilde{x}\right)\right\}\leq\sup_{\left\|f\right\|_{L}\leq1}\mathrm{E}_{x\sim p(x)}\left\{D\left(x\right)\right\}-\mathrm{E}_{\tilde{x}\sim p(\tilde{x})}\left\{D\left(\tilde{x}\right)\right\}=K\cdot\mathrm{W}\left(\cdot\right) \tag{4}$$

For $p\left(\tilde{x}\right)$ induced by $G_{\theta}\left(z\right)$, we can backprop through (4), thus:

$$\nabla_{\theta}\mathrm{W}\left(p\left(x\right),p\left(\tilde{x}\right)\right)=-\mathrm{E}_{z\sim p(z)}\left\{\nabla_{\theta}D\left(G_{\theta}\left(z\right)\right)\right\}.$$

In order to have parameters w lie in a compact space, something simple we can do is clamp the weights to a fixed box (e.g., $\mathcal{W}=\left[-0.01,0.01\right]^{l}$) after each gradient update.

Comparison between GAN and WGAN

- Standard GAN

$$\min_G \max_D \mathrm{E}_{x\sim p(x)} \{\log D(x)\} + \mathrm{E}_{\tilde{x}\sim p(\tilde{x})} \{\log (1 - D(\tilde{x}))\}$$

- Wasserstein GAN

$$\max_{D\in\mathcal{D}} \mathrm{E}_{\mathbf{x}\sim p(x)} \{D(x)\} + \mathrm{E}_{\tilde{x}\sim p(x)} \{D(\tilde{x})\}$$

WGAN summary

• New divergence measure for optimizing the generator
• Addresses instabilities with JSD version (sigmoid cross-entropy)
• Robust to architectural choices
• Progress on mode collapse and stability of derivative wrt input
• Introduces the idea of using Lipschitzness to stabilize GAN training
• Negative point: weight clipping requires attention

Generative Latent Variable Models: Variational Inference And Diffusion Models

Generating by latent variables

We present generative models with a different approach based on latent variables.
Let us suppose that we have a collection of images with horses and we want to learn p (x) to generate new images.
As a first step, we can ask ourselves how we should generate a horse.
There are some factors in data (e.g., a silhouette, a color, a background) that are crucial for generating an object (here, a horse).
Once we decide about these factors, we can generate them by adding details. Like painting.

Generation process from a low-dimensional latent space

We can denote the collection of images as $\mathbf{x} \in \mathcal{X}^D$ and the low-dimensional latent variables as $\mathbf{z} \in \mathcal{Z}^M$, where $\mathcal{Z}^M$ represents the *manifold*.

Factorization of latent variable models

Given the latent variables $\mathbf{z}$, the joint distribution is factorized as:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})$$

For training we have access only to $\mathbf{x}$, thus we should *sum out* the unknown, i.e., $\mathbf{z}$, by considering the marginal likelihood function:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, d\mathbf{z}. \tag{1}$$

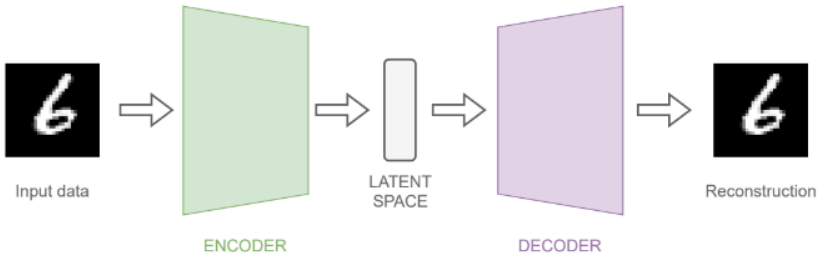
A natural question now is how to calculate this integral.

- The easiest case is that the integral is tractable.
- If not tractable, we need to compute it by utilizing a specific *approximate inference*, known as variational inference.

Autoencoder

In the presence of a larger amount of data or of a more complex problem, involving also a *nonlinear input-output relation*, we need deep learning methods to reduce the dimensionality, with the goal to enhance the representation power:

$$\min_{\mathbf{w}_1, \mathbf{w}_2} \|\mathbf{x}_i - g\left(f\left(\mathbf{x}_i; \mathbf{w}_1\right); \mathbf{w}_2\right)\|_2^2$$



Unsupervised representation learning

Unsupervised representation learning refers to a quest for disentangled, semantically meaningful, statistically independent and causal factors of variation in data.

Representation learning as implicit regularization

Alternatively, one may view this as an implicit form of regularization.

The VAE framework

The framework of variational autoencoders (VAEs) provides a principled method for jointly learning deep latent-variable models and corresponding inference models using
stochastic gradient descent.
The framework has a wide array of applications, from generative modeling to semi-supervised learning and representation learning.

The VAE can be viewed as two coupled, but independently parameterized models: the encoder, or recognition model or inference model, and the decoder, or generative model.

The variational autoencoder approach

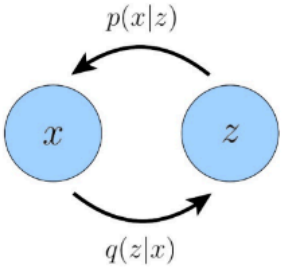


Figure 7: In the Variational Autoencoder, the encoder $q(\mathbf{z}|\mathbf{x})$, or *recognition model*, defines a distribution over latent variables $\mathbf{z}$ for a set of observations $\mathbf{x}$, while the *generation model* $p(\mathbf{x}|\mathbf{z})$ decodes latent variables into observations [1].

VAEs are particularly interesting when the dimensionality of $\mathbf{z}$ is less than that of input $\mathbf{x}$, as we might then be learning compact, useful representations.

When a semantically meaningful latent space is learned, latent vectors can be edited before being passed to the decoder to more precisely control the data generated.

Advantages of the VAE:

1) **amortized inference**

In the VAE framework, the recognition model is a stochastic function of the input variables.

This is in contrast to the case in which each data observations has a separate variational distribution, which is inefficient for large data-set.

The recognition model uses one set of parameters to model the relation between input and latent variables and it is referred to as amortized inference.

The recognition model can be arbitrary complex but is still reasonably fast because it can be done using a single feedforward pass from input to latent variables.

2) **reparameterization trick**

The parameter learning process of the recognition model introduces gradient noise.
The VAE tackes this problem by the reparameterization trick, a simple procedure to reorganize the gradient computation, thus reducing variance in the gradients.

With respect to other generative models, and like other likelihood-based models, VAEs generate more dispersed samples, but are better density models in terms of the likelihood criterion.

Marrying graphical models and deep learning

The generative model is a Bayesian network of the form p (x|z) p (z), where x represents the input and z the latent vectors.
Similarly, the recognition model is also a conditional Bayesian network of the form q (z|x).
Each conditional may hide a complex (deep) neural network, as z|x ~ f (x,), with f a neural network mapping and a noise random variable.
Its learning algorithm is a mix of classical (amortized, variational) expectation maximization, which ends up backpropagating through the many layers of the deep neural networks through the reparameterization trick.

Probabilistic models

In machine learning, we are often interested in learning probabilistic models of various natural and artificial phenomena from data.
We typically need to assume some level of uncertainty over aspects of the model.
The degree and nature of this uncertainty is specified in terms of (conditional) probability distributions.

Conditional models

Often, we are not interested in learning an unconditional model $p_{\boldsymbol{\theta}}(\mathbf{x})$, but a conditional model $p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x})$ such that:

$$p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}) \approx p_{\text{data}}(\mathbf{y}|\mathbf{x}).$$

Conditional models become more difficult to learn when the predicted variables are very high-dimensional, such as images, video or sound.

To avoid notational clutter, unconditional modeling is often assumed but we can generalize those models to conditional ones.

Parameterizing conditional distributions with neural networks

Neural networks are a particularly flexible and computationally scalable type of function approximator, denoted as $f(\cdot)$.

In case of neural network based image classification [6, 7], neural networks parameterize a categorical distribution $p_{\boldsymbol{\theta}}(y|\mathbf{x})$ over a class label l, conditioned on an image $\mathbf{x}$:

$$\mathbf{y} = f(\mathbf{x})$$
$$p_{\boldsymbol{\theta}}(l|\mathbf{x}) = \text{Categorical}(l; \mathbf{y})$$

where the last operation of $f(\cdot)$ is typically a softmax function such that $\sum_i y_i = 1$.

Structured probabilistic models

Probability distributions over a very large number of random variables involve direct interactions between relatively few variables.

Using a single function to describe the entire joint probability distribution can be very inefficient (both computationally and statistically).

Instead of using a single function to represent a probability distribution, we can split a probability distribution into many factors that we multiply together.

These factorizations can greatly reduce the number of parameters needed to describe the distribution.

When we represent the factorization of a probability distribution with a graph, we call it a structured probabilistic model or graphical model.

Probabilistic graphical models and neural networks

Directed graphical models, or *Bayesian networks*, are a type of probabilistic models where all the variables are topologically organized into a *directed acyclic graph*.

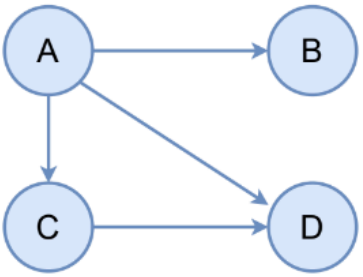


Figure 8: Example of a directed acyclic graph on four vertices.

The joint distribution over the variables of such models factorizes as a product of prior and conditional distributions:

$$p_{\boldsymbol{\theta}}\left(\mathbf{x}_1, \ldots, \mathbf{x}_M\right) = \prod_{j=1}^{M} p_{\boldsymbol{\theta}}\left(\mathbf{x}_j | A\left(\mathbf{x}_j\right)\right)$$

where $A\left(\mathbf{x}_j\right)$ is the set of parent variables of node j in the directed graph.

Traditionally, $p_{\boldsymbol{\theta}}\left(\mathbf{x}_j | A\left(\mathbf{x}_j\right)\right)$ is parameterized as a lookup table or a linear model [9].

A more flexible parameterization involves neural networks:

$$\begin{aligned} \boldsymbol{\eta} &= f\left(A\left(\mathbf{x}\right)\right) \\ p_{\boldsymbol{\theta}}\left(\mathbf{x} | A\left(\mathbf{x}\right)\right) &= p_{\boldsymbol{\theta}}\left(\mathbf{x} | \boldsymbol{\eta}\right) \end{aligned}$$

Learning in fully observed models with neural networks

If all variables in the directed graphical model are observed in the data $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^N$, under the *i.i.d. assumption*, then we can compute and differentiate the log-probability of the data under the model:

$$\log p_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{\mathbf{x} \in \mathcal{D}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$$

Under the ML criterion, we attempt to find the parameters $\boldsymbol{\theta}$ that maximize the sum, or equivalently the average, of the log-probabilities assigned to the data by the model.

ML maximization is important due to its equivalence to Kullback Leibler (KL) divergence minimization.

Directed models with latent variables

Latent variables $\mathbf{z}$ are variables that are part of the model, but which we *don't observe*, and are therefore not part of the dataset.

The marginal distribution over the observed variables $p_{\boldsymbol{\theta}}(\mathbf{x})$ is given by:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})\, d\mathbf{z} \tag{2}$$

where the joint probability $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$ is represented by a directed graphical model.

$p_{\boldsymbol{\theta}}(\mathbf{x})$ is also known as marginal likelihood and it is a *flexible* implicit distribution, since it can be represented by mixtures of distributions.

We refer as **deep latent variable model** (DLVM) to a latent variable model $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$ whose distributions are parameterized by neural networks.

Even when each factor in the directed model is relatively simple, the marginal distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ can be very complex.

This expressivity makes DLVMs attractive for approximating $p_{\text{data}}(\mathbf{x})$.

The simplest, and most common, DLVM is defied by the following factorization:

$$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$$

where the prior distribution $p_{\boldsymbol{\theta}}(\mathbf{z})$ and/or $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ are specified.

Intractabilities

The main difficulty of ML learning in DLVMs is that the marginal probability $p_{\boldsymbol{\theta}}(\mathbf{x})$ of data under the model is typically intractable.

This is due to absence of any analytical solution of the integral in (2).

Thus, differentiation is not possible, contrary to what is possible with fully observable data.

The intractability of $p_{\boldsymbol{\theta}}(\mathbf{x})$ is related to the intractability of the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$, as:

$$p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})}.$$

Approximate inference techniques allow us to approximate $p_{\boldsymbol{\theta}}(\mathbf{x})$ and $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ in DLVMs.

Encoder or approximate posterior

The VAE framework provides a computationally efficient way for optimizing DLVMs jointly with a corresponding inference model using SGD.
To turn the DLVM's intractable posterior into tractable problems, we introduce a parametric inference model qφ (z|x).
This model is also called an encoder or recognition model.
The model parameters φ are known as variational parameters, which should be optimized such that qφ (z|x) ≈ pθ (z|x).

Parameterizing the inference model

Like a DLVM, the inference model can be any directed graphical model:

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = q_{\boldsymbol{\phi}}(\mathbf{z}_1, \ldots, \mathbf{z}_M|\mathbf{x}) = \prod_{j=1}^{M} q_{\boldsymbol{\phi}}(\mathbf{z}_j|A(\mathbf{z}_j), \mathbf{x})$$

where $A(\mathbf{z}_j)$ is the set of parent variables of $\mathbf{z}_j$ in the directed graph.

And also similar to a DLVM, the distribution $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ can be parameterized using deep neural networks, e.g.:

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = f_{\boldsymbol{\phi}}(\mathbf{x})$$
$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}))$$

Tipically, a single encoder neural network $f_{\boldsymbol{\phi}}(\cdot)$ is used to perform posterior inference over all of the datapoints (amortized variational inference strategy [5]).

Maximization of the marginal log-likelihood

The optimization objective of the VAE is the **evidence lower bound** (ELBO).

For any choice of the inference model $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, the maximization of the log-likelihood can be expressed as:

$$\begin{aligned}\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x})\right\} \\ &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right\} \\ &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\frac{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right\} \\ &= \underbrace{\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\right)\right\}}_{\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})} + \underbrace{\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right\}}_{\mathcal{D}_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))}\end{aligned} \tag{3}$$

where $\mathcal{D}_{\mathrm{KL}}(\cdot)$ the KL divergence and $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\cdot)$ is the ELBO.

KL divergence and ELBO

The KL divergence between $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ and $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ in (3) is nonnegative:

$$\mathcal{D}_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right) \geq 0.$$

The ELBO in (3) can be expressed as:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right\}$$

Due to (4), the ELBO is a lower bound on the log-likelihood of the data:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{D}_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right) \leq \log p_{\boldsymbol{\theta}}(\mathbf{x})$$

Role of KL divergence and ELBO

The KL divergence $\mathcal{D}_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)||p_{\boldsymbol{\theta}}\left(\mathbf{z}|\mathbf{x}\right)\right)$ determines two distances:

1. the KL divergence of the approximate posterior from the true posterior, by definition;
2. the tightness of the bound, i.e., the gap between the ELBO $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbf{x}\right)$ and the marginal likelihood $\log p_{\boldsymbol{\theta}}\left(\mathbf{x}\right)$.

Maximizing the ELBO concurrently leads to:

1. maximizing the marginal likelihood $p_{\boldsymbol{\theta}}\left(\mathbf{z}|\mathbf{x}\right)$, which improves the generation;
2. minimizing the KL divergence, which improves the approximation of $q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)$ to $p_{\boldsymbol{\theta}}\left(\mathbf{z}|\mathbf{x}\right)$.

Stochastic gradient-based optimization of the ELBO

The ELBO allows joint optimization w.r.t. all parameters ($\boldsymbol{\theta}$ and $\boldsymbol{\phi}$) using SGD.

The gradient of individual-datapoint ELBO $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbf{x}\right)$ is in general intractable.

Unbiased gradients of the ELBO w.r.t. parameters $\boldsymbol{\theta}$ are simple to obtain:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbf{x}\right) &= \nabla_{\boldsymbol{\theta}}\,\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)\right\} \\ &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)\right)\right\} \\ &\simeq \nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)\right) \\ &= \nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right)\right)\end{aligned}$$

Unbiased gradients of the ELBO w.r.t. *variational* parameters $\boldsymbol{\phi}$ are more difficult to obtain, since the expectation depends on $\boldsymbol{\phi}$ and cannot be approximated:

$$\begin{aligned}\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}\left(\mathbf{x}\right) &= \nabla_{\boldsymbol{\phi}}\,\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)\right\} \\ &\neq \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}\left(\mathbf{x},\mathbf{z}\right) - \log q_{\boldsymbol{\phi}}\left(\mathbf{z}|\mathbf{x}\right)\right)\right\}\end{aligned} \tag{5}$$

To solve this problem, we can use a change of variables, also called the reparameterization trick [3, 4].

Change of variables

First, we express the random variable $\mathbf{z} \sim q_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ as some differentiable (and invertible) transformation of another random variable $\boldsymbol{\epsilon}$, given $\mathbf{z}$ and $\boldsymbol{\phi}$:

$$\mathbf{z} = g(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x}) \tag{6}$$

where the distribution of the random variable $\boldsymbol{\epsilon}$ is independent of $\mathbf{x}$ or $\boldsymbol{\phi}$.

Thus, the expectation $\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{\cdot\}$ can be rewritten in terms of $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$:

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}} \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{f(\mathbf{z})\} &= \nabla_{\boldsymbol{\phi}} \mathrm{E}_{p(\boldsymbol{\epsilon})}\{f(\mathbf{z})\} \\
&= \mathrm{E}_{p(\boldsymbol{\epsilon})}\{\nabla_{\boldsymbol{\phi}} f(\mathbf{z})\} \\
&\simeq \nabla_{\boldsymbol{\phi}} f(\mathbf{z}) .
\end{aligned}$$

Stochastic gradient of the ELBO under reparameterization

We can now replace an expectation w.r.t. $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, where $\mathbf{z} = g(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x})$, with one w.r.t. $p(\boldsymbol{\epsilon})$, thus the ELBO becomes:

$$\begin{aligned}
\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\} \\
&= \mathrm{E}_{p(\boldsymbol{\epsilon})}\{\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\}
\end{aligned}$$

Thus, we can form a simple Monte Carlo estimator $\widetilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ where we use a single noise sample $\boldsymbol{\epsilon}$ from $p(\boldsymbol{\epsilon})$:

$$\begin{aligned}
\boldsymbol{\epsilon} &\sim p(\boldsymbol{\epsilon}) \\
\mathbf{z} &= g(\boldsymbol{\phi}, \mathbf{x}, \boldsymbol{\epsilon}) \\
\widetilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})
\end{aligned} \tag{7}$$

The reparameterized ELBO estimator is referred to as the stochastic gradient variational Bayes (SGVB) estimator [3].

Computation of log qφ (z|x)

The computation of the ELBO estimator in (7) requires the computation of $\log q_{\phi}(\mathbf{z}|\mathbf{x})$, which is very simple if we use (6).

As long as $g(\cdot)$ in (6) is an invertible function, $\boldsymbol{\epsilon}$ and $\mathbf{z}$ are related by:

$$\log q_{\phi}(\mathbf{z}|\mathbf{x}) = \log p(\boldsymbol{\epsilon}) - \log d_{\phi}(\mathbf{x}, \boldsymbol{\epsilon})$$

where:

$$\log d_{\phi}(\mathbf{x}, \boldsymbol{\epsilon}) = \log \left| \det \left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right) \right|$$

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \frac{\partial (z_1, \ldots, z_k)}{\partial (\epsilon_1, \ldots, \epsilon_k)} = \begin{pmatrix} \frac{\partial z_1}{\partial \epsilon_1} & \cdots & \frac{\partial z_1}{\partial \epsilon_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial \epsilon_1} & \cdots & \frac{\partial z_k}{\partial \epsilon_k} \end{pmatrix}$$

It is possible to build very flexible transformations $g(\cdot)$ for which $\log d_{\phi}(\mathbf{x}, \boldsymbol{\epsilon})$ is simple to compute, resulting in highly flexible inference models $q_{\phi}(\mathbf{z}|\mathbf{x})$.

Factorized Gaussian posteriors

A common choice is a simple factorized Gaussian encoder $q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\mathbf{z}; \boldsymbol{\mu}, \operatorname{diag}\left(\boldsymbol{\sigma}^2\right)\right)$

$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = f_{\phi}(\mathbf{x})$$
$$q_{\phi}(\mathbf{z}|\mathbf{x}) = \prod_i q_{\phi}(z_i|\mathbf{x}) = \prod_i \mathcal{N}\left(z_i; \mu_i, \sigma_i^2\right)$$

After reparameterization, we can write:

$$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$$
$$(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) = f_{\phi}(\mathbf{x})$$
$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$$

The log determinant of the Jacobian is:

$$\log d_{\phi}(\mathbf{x}, \boldsymbol{\epsilon}) = \log \left| \det \left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right) \right| = \sum_i \log \sigma_i$$

where $\partial \mathbf{z} / \partial \boldsymbol{\epsilon} = \operatorname{diag}(\boldsymbol{\sigma})$.

Thus, the posterior density is:

$$\log q_{\phi}(\mathbf{z}|\mathbf{x}) = \log p(\boldsymbol{\epsilon}) - \log d_{\phi}(\mathbf{x}, \boldsymbol{\epsilon}) = \sum_i \log \mathcal{N}(\epsilon_i; 0, 1) - \log \sigma_i.$$

VAE pros and cons
+ The latent space can be very small therefore useful for many applications.
+ Fast sampling, even for the generation of multiple samples at the same time.
- VAEs approximate the distribution of the data, so the samples can be imprecise.
- VAEs are not easy to train due to the two terms in the function to be optimized.

Hierarchical variational autoencoder
A Hierarchical Variational Autoencoder (HVAE) is a generalization of a VAE that extends to multiple hierarchies over latent variables.
In HVAE, latent variables themselves are interpreted as generated from other higher-level, more abstract latents.

In particular, the HVAE involves a number T of hierarchical levels, where each latent is allowed to condition on all previous latents.

A special case of the HVAE is the Markovian HVAE (MHVAE), or sometimes known as Recursive VAE, in which the generative process is a Markov chain.

Markovian hierarchical variational autoencoder
In the MHVAE, each transition down the hierarchy is Markovian, where decoding each latent zt only conditions on previous latent zt+1.

The joint distribution and the posterior of an MHVAE can be written as

$$p\left(\mathbf{x}, \mathbf{z}_{1:T}\right)=p\left(\mathbf{z}_T\right) p_{\boldsymbol{\theta}}\left(\mathbf{x} | \mathbf{z}_1\right) \prod_{t=2}^T p_{\boldsymbol{\theta}}\left(\mathbf{z}_{t-1} | \mathbf{z}_t\right)$$

$$q_{\boldsymbol{\phi}}\left(\mathbf{z}_{1:T} | \mathbf{x}\right)=q_{\boldsymbol{\phi}}\left(\mathbf{z}_1 | \mathbf{x}\right) \prod_{t=2}^T q_{\boldsymbol{\phi}}\left(\mathbf{z}_t | \mathbf{z}_{t-1}\right)$$

The ELBO for MHVAE can be also easily derived:

$$\log p\left(\mathbf{x}\right) \geq \mathrm{E}_{q_{\boldsymbol{\phi}}\left(\mathbf{z}_{1:T} | \mathbf{x}\right)}\left[\log \frac{p\left(\mathbf{x}, \mathbf{z}_{1:T}\right)}{q_{\boldsymbol{\phi}}\left(\mathbf{z}_{1:T} | \mathbf{x}\right)}\right]$$

Variational diffusion models
A Variational Diffusion Model (VDM) can be seen as an MHVAE with three key assumptions:
1 The latent dimension is exactly equal to the data dimension.
2 The structure of the latent encoder at each timestep is not learned but pre-defined as a linear Gaussian model (i.e., a Gaussian distribution centered around the output of the previous timestep).
3 The Gaussian parameters of the latent encoders vary over time in such a way that the distribution of the latent at final timestep T is a standard Gaussian.
Furthermore, the Markov property between hierarchical transitions is explicitly maintained from a standard MHVAE.

VDM assumptions: encoder posterior

Let us denote both true data samples and latents with the same variable $\mathbf{x}_t$, where $t = 0$ represents true data samples and $t \in [1; T]$ represents a corresponding latent with hierarchy indexed by t.

Considering the first assumption, The VDM posterior is the same as the MHVAE posterior (9), but it can now be rewritten as:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{11}$$

From the second assumption we know that, unlike an MHVAE, the structure of the encoder at each timestep t is fixed as a linear Gaussian model, where the mean and standard deviation can be set beforehand or learned as parameters [12, 13].

VDM assumptions: encoder transitions

So, we parameterize the Gaussian encoder with

$$\boldsymbol{\mu}_t(\mathbf{x}_t) = \sqrt{\alpha_t}\mathbf{x}_{t-1} \quad \text{and} \quad \boldsymbol{\Sigma}_t(\mathbf{x}_t) = (1 - \alpha_t)\mathbf{I},$$

where the coefficients are chosen such that the variance of the latent variables stay at a similar scale (*variance preserving*).

α_t is a *potentially learnable* coefficient that can vary with the hierarchical depth t, for flexibility.

Encoder transitions can be denoted as:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{\alpha_t}\mathbf{x}_{t-1}, (1 - \alpha_t)\mathbf{I}) \tag{12}$$

VDM assumptions: joint distribution

From the third assumption, we know that the distribution of the final latent $p(\mathbf{x}_T)$ is a standard Gaussian.

Thus, we can then update the joint distribution of a Markovian HVAE (8) to write the joint distribution for a VDM as:

$$p(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{13}$$

where $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$.

Collectively, this set of assumptions describes a steady noisification of an image input over time; we progressively corrupt an image by adding Gaussian noise until eventually it becomes completely identical to pure Gaussian noise.

Remarks on the variational diffusion process

Note that encoder distributions q (xt|xt−1) are no longer parameterized by φ, as they are completely modeled as Gaussians with defined mean and variance parameters at each timestep.

Therefore, in a VDM, we are only interested in learning conditionals pθ (xt−1|xt), so that we can simulate new data.

After optimizing the VDM, the sampling procedure is as simple as sampling Gaussian noise from p (xT) and iteratively running the denoising transitions pθ (xt−1|xt) for T steps to generate a novel x0.

ELBO in variational diffusion models

Like any HVAE, the VDM can be optimized by maximizing the ELBO [1]:

$$
\begin{aligned}
\log p(\mathbf{x}) &= \log \int p(\mathbf{x}_{0:T})\, d\mathbf{x}_{1:T} \\
&\geq \mathrm{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \right] \\
&= \underbrace{\mathrm{E}_{q(\mathbf{x}_1|\mathbf{x}_0)} \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}_0|\mathbf{x}_1) \right]}_{\text{reconstruction term}} - \underbrace{\mathrm{E}_{q(\mathbf{x}_{T-1}|\mathbf{x}_0)} \left[\mathcal{D}_{\mathrm{KL}} \left(q(\mathbf{x}_T|\mathbf{x}_{T-1}) \,||\, p(\mathbf{x}_T) \right) \right]}_{\text{prior matching term}} \\
&\quad - \sum_{t=1}^{T-1} \underbrace{\mathrm{E}_{q(\mathbf{x}_{t-1},\mathbf{x}_{t+1}|\mathbf{x}_0)} \left[\mathcal{D}_{\mathrm{KL}} \left(q(\mathbf{x}_t|\mathbf{x}_{t-1}) \,||\, p_{\boldsymbol{\theta}}(\mathbf{x}_t|\mathbf{x}_{t+1}) \right) \right]}_{\text{consistency term}}
\end{aligned}
\tag{14}
$$

The derived form of the ELBO can be interpreted in terms of its individual components.

Individual components of the VDM ELBO

1 Reconstruction term: aims at predicting the log probability of the original data sample given the first-step latent. This term also appears in a vanilla VAE, and can be trained similarly.

2 Prior matching term: is minimized when the final latent distribution matches the Gaussian prior. This term requires no optimization, as it has no trainable parameters; furthermore, as we have assumed a large enough T such that the final distribution is Gaussian, this term effectively becomes zero.

3 Consistency term: endeavors to make the distribution at xt consistent, from both forward and backward processes. This is reflected mathematically by the KL Divergence.

This term is minimized when we train pθ (xt|xt+1) to match the Gaussian distribution q (xt|xt−1)

Drawback in the VDM ELBO formulation

In the ELBO in (14), all terms are computed as expectations and therefore can be approximated using Monte Carlo estimates.

However, the ELBO optimization of (14) might be suboptimal. Indeed, since the consistency term is computed as an expectation over two random variables {xt−1; xt+1} for every timestep, the variance of its Monte Carlo estimate could potentially be higher than a term that is estimated using only one random variable per timestep. As it is computed by summing up T − 1 consistency terms, the final estimated value of the ELBO may have high variance for large T values.

Rethinking the ELBO

To address this issue, we can derive the ELBO considering each term computed as an expectation over only one random variable at a time.

We can rewrite encoder transitions as $q(\mathbf{x}_t|\mathbf{x}_{t-1}) = q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)$, where the extra conditioning term is superfluous due to the Markov property.

According to the Bayes rule, we can rewrite each transition in (12) as:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) = \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)\, q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} \tag{15}$$

on the basis of which we can redefine the ELBO in (14) [1].

Reformulation of the lower-variance ELBO

Replacing (15) in (14) yields the ELBO of denoising diffusion probabilistic models (DDPMs):

$$\begin{aligned}
\log p(\mathbf{x}) &= \log \int p(\mathbf{x}_{0:T})\, d\mathbf{x}_{1:T} \\
&\geq \mathrm{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \right] \\
&= \underbrace{\mathrm{E}_{q(\mathbf{x}_1|\mathbf{x}_0)} \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}_0|\mathbf{x}_1) \right]}_{\text{reconstruction term}} - \underbrace{\mathcal{D}_{\mathrm{KL}}\left(q(\mathbf{x}_T|\mathbf{x}_0) \,||\, p(\mathbf{x}_T) \right)}_{\text{prior matching term}} \\
&- \sum_{t=2}^{T} \underbrace{\mathrm{E}_{q(\mathbf{x}_t|\mathbf{x}_0)} \left[\mathcal{D}_{\mathrm{KL}}\left(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \,||\, p_{\boldsymbol{\theta}}(\mathbf{x}_{t-1}|\mathbf{x}_t) \right) \right]}_{\text{denoising matching term}}
\end{aligned} \tag{16}$$

This interpretation for the ELBO can be estimated with lower variance, as each term is computed as an expectation of *at most one random variable* at a time.

Individual components of the DDPM ELBO
1 Reconstruction term: aims at predicting the log probability of the original data sample and can be trained similarly to the vanilla VAE.

2 Prior matching term: represents how close the distribution of the final noisified input is to the standard Gaussian prior. It has no trainable parameters, and is also equal to zero under our assumptions.

3 Denoising matching term: learns a desired denoising transition step pθ (xt−1|xt) as an approximation to tractable, ground-truth denoising transition step q (xt−1|xt; x0). This term is therefore minimized when the two denoising steps match as closely as possible, as measured by their KL Divergence.

Further considerations on the ELBO
Both the ELBO derivations ((14) and (16)) rely on the Markov assumption so they hold for any arbitrary MHVAE.
If we set T = 1, both the ELBO interpretations collapse into the ELBO of a vanilla VAE.

Each KL Divergence term DKL (q (xt−1|xt, x0)||pθ (xt−1|xt)) is difficult to minimize for arbitrary posteriors in arbitrarily complex MHVAEs due to the added complexity of simultaneously learning the encoder

However, in a VDM we can leverage the Gaussian transition assumption to make optimization tractable.

Sampling process

Under the reparameterization trick of the VDM encoder transitions (49), samples $\mathbf{x}_t \sim q(\mathbf{x}_t|\mathbf{x}_{t-1})$ can be rewritten as:

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon} \qquad \text{with} \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \mathbf{I})$$

and, similarly, samples $\mathbf{x}_{t-1} \sim q(\mathbf{x}_{t-1}|\mathbf{x}_{t-2})$ can be written as:

$$\mathbf{x}_{t-1} = \sqrt{\alpha_{t-1}}\mathbf{x}_{t-1} + \sqrt{1-\alpha_{t-1}}\boldsymbol{\epsilon} \qquad \text{with} \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \mathbf{I})$$

Thus, generalizing for an arbitrary sample $\mathbf{x}_t \sim q(\mathbf{x}_t|\mathbf{x}_0)$, we can write:

$$\begin{aligned} \mathbf{x}_t &= \sqrt{\prod_{i=1}^{t} \alpha_i}\mathbf{x}_0 + \sqrt{1-\prod_{i=1}^{t} \alpha_i}\boldsymbol{\epsilon}_0 \\ &= \sqrt{\overline{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\overline{\alpha}_t}\boldsymbol{\epsilon}_0 \qquad \sim \mathcal{N}\left(\mathbf{x}_t; \sqrt{\overline{\alpha}_t}\mathbf{x}_0, (1-\overline{\alpha}_t)\mathbf{I}\right) \end{aligned} \tag{17}$$

Predict the original image via deep learning

In the classical formulation, a VDM can be trained by simply learning a neural network to predict the original natural image $\mathbf{x}_0$ from an arbitrary noised version $\mathbf{x}_t$ at time index t.

If we consider the reparameterization trick and rearrange (17), we obtain:

$$\mathbf{x}_0 = \frac{\mathbf{x}_t - \sqrt{1-\overline{\alpha}_t}\boldsymbol{\epsilon}_0}{\sqrt{\overline{\alpha}_t}}$$

where a neural network can be used to learn predicting the source noise $\boldsymbol{\epsilon}_0 \sim \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \mathbf{I})$ that determines $\mathbf{x}_t$ from $\mathbf{x}_0$.

This means that learning a VDM by predicting the original image $\mathbf{x}_0$ is equivalent to learning a VDM by predicting the noise [13, 15, 1].

Generate through conditioning information

So far, we have focused on modeling just the data distribution $p(\mathbf{x})$.

However, we are often also interested in learning conditional distribution $p(\mathbf{x}|\mathbf{y})$, which would enable us to explicitly control the data we generate through conditioning information $\mathbf{y}$.

A natural way to add conditioning information is simply *alongside the timestep information*, at each iteration.

Recall the VDM joint distribution from (13):

$$p(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

Conditional diffusion model

Then, to turn this into a conditional diffusion model, we can simply add arbitrary conditioning information $\mathbf{y}$ at each transition step as:

$$p(\mathbf{x}_{0:T}|\mathbf{y}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{y})$$

For example, $\mathbf{y}$ could be a text encoding in *image-text generation*, or a low-resolution image to perform *super-resolution*.

We are thus able to learn the core neural networks of a VDM as before, by predicting the source noise or the score function according to the desired diffusion model.

Guidance

A caveat of the previous vanilla formulation is that a conditional diffusion model trained in this way may potentially learn to ignore or downplay any given conditioning information. Guidance is therefore proposed as a way to more explicitly control the amount of weight the model gives to the conditioning information, at the cost of sample diversity.
The two most popular forms of guidance are known as:

- classifier guidance
- classifier-free guidance

Classifier guidance

Let us consider the score-based formulation of a conditional diffusion model $\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t|\mathbf{y})$ at arbitrary noise levels t.

By applying the Bayes rule, we can derive the following equivalent form:

$$\begin{aligned}\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t|\mathbf{y}) &= \nabla_{\mathbf{x}_t} \log \left(\frac{p(\mathbf{x}_t)\, p(\mathbf{y}|\mathbf{x}_t)}{p(\mathbf{y})} \right) \\ &= \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) + \nabla_{\mathbf{x}_t} \log p(\mathbf{y}|\mathbf{x}_t) - \nabla_{\mathbf{x}_t} \log p(\mathbf{y}) \\ &= \underbrace{\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t)}_{\text{unconditional score}} + \gamma \underbrace{\nabla_{\mathbf{x}_t} \log p(\mathbf{y}|\mathbf{x}_t)}_{\text{adversarial gradient}}\end{aligned} \tag{18}$$

where the hyperparameter γ is introduced as fine-grained control to either encourage or discourage the model to consider the conditioning information.

Classifier-free guidance

In classifier-free guidance, the training of a separate classifier model is avoided considering that:

$$\nabla_{\mathbf{x}_t} \log p(\mathbf{y}|\mathbf{x}_t) = \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t|\mathbf{y}) - \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t)$$

which is substituted in (18) to get:

$$\begin{aligned}\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t|\mathbf{y}) &= \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) + \gamma \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t|\mathbf{y}) - \gamma \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) \\ &= \underbrace{\gamma \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t|\mathbf{y})}_{\text{conditional score}} + \underbrace{(1-\gamma)\, \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t)}_{\text{unconditional score}}\end{aligned} \tag{19}$$

Because learning two separate diffusion models is expensive, we can learn both the conditional and unconditional diffusion models together as a singular conditional model.

Classifier-free guidance is elegant because it enables us greater control over the conditional generation procedure while only requiring the training of a singular diffusion model.

Image super-resolution models: cascaded diffusion models
This forms the backbone of image super-resolution models such as Cascaded Diffusion Models, as well as state-of-the-art image-text models such as DALL-E 2 and Imagen.
Cascaded diffusion models (CDM) are pipelines of diffusion models that generate images of increasing resolution, without any classifier [21]. Conditioning augmentation was found critical towards achieving high sample fidelity.

Image super-resolution models: DALL-E 2

DALL·E 2 is a system for text-to-image generation developed by OpenAI.
When prompted with a caption, the system will attempt to generate a novel image from scratch that matches it.

DALL·E 2 involves the use of CLIP, a model that “efficiently learns visual concepts from natural language supervision”.

In particular, DALL·E 2 uses a diffusion model conditioned on CLIP image embeddings, which, during inference, are generated from CLIP text embeddings by a prior model.

Image super-resolution models: Imagen
Imagen is a text-to-image diffusion model developed by Google with an unprecedented degree of photorealism and a deep level of language understanding..
Imagen builds on the power of large transformer language models in understanding text and hinges on the strength of diffusion models in high-fidelity image generation.

Increasing the size of the language model in Imagen boosts both sample fidelity and image- text alignment much more than increasing the size of the image diffusion model.