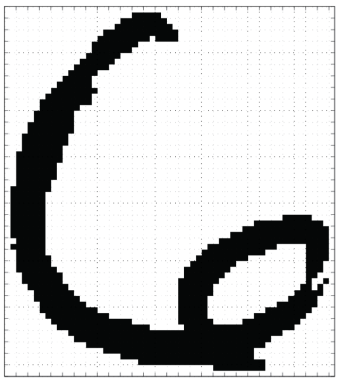
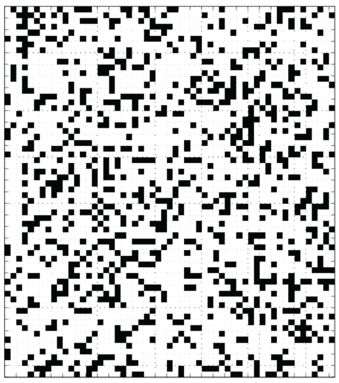
Dimensionality reduction

Introduction

In many cases (like images) we find ourselves to work with very high dimensional inputs. For example, a space $X \in \mathbb{R}^{w \times h \times c}$ has $w \times h \times c$ degrees of freedom. But often it happens that not all the combination of the input are meaningful. In fact actual data may present a much lower variability (for example if we are analyzing a dataset of numbers, only certain combination of pixels make sense in the context of the dataset):



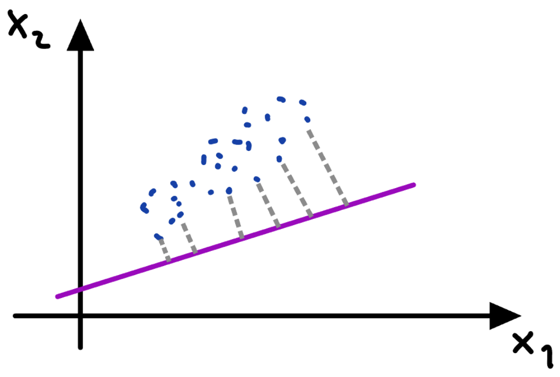


Even if this is a valid combination of pixels, it doesn't make sense in a digits datset.

The field of data reduction studies the possible transformation that can be applied to the input space in order to obtain compact (but still meaningful) representation of it in lower dimensions. When performing these transformations the risk is to lose information, or obtaining only partial representations of the problem. The following methods aim at finding the best tradeoff between dimensionality reduction and information retention.

Principal component analysis (PCA)

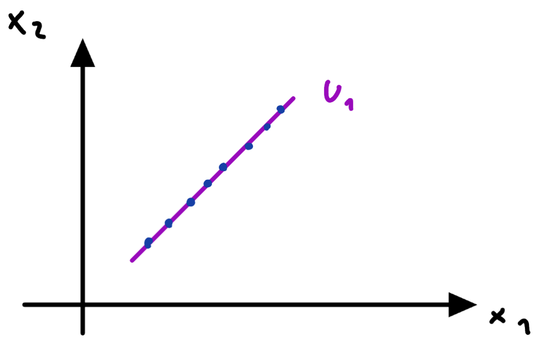
Imagine that we want to apply a transformation of a 2D dataset in $\mathbb{R}^2$ into $\mathbb{R}$. What we do is basically projecting every point in 2 dimensions onto a 1D line:



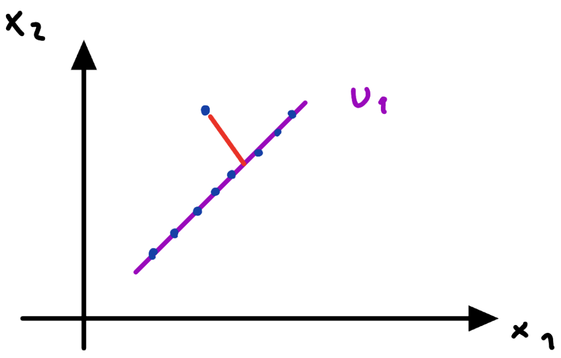
But *what is the best representation?*

What we want is the transformation that guarantees the minimum error (meaning the maximum retention of information).

The simplest case is 2D datas that already lie on a linear pattern. Here we have no error in the representation u_1:

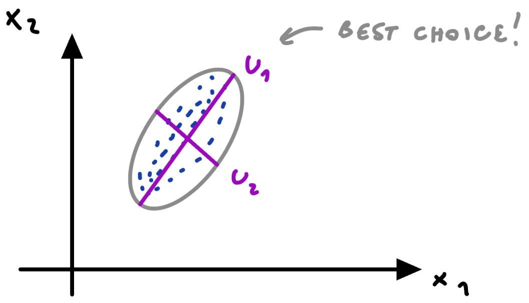


If we introduce some variability to the data, we start loosing informations in the representation:



Actually, u_1 still remains the best choice, because despite loosing the information for one point, still retains the informations about the majority of points.

In general, the best choice is the one that follows the direction of maximum variance of the data, like in the following example:



In general, what we want is to produce a transformation $\mathbb{R}^M \rightarrow \mathbb{R}^D$ where D is the number of directions with the maximum variance. We denote the directions of the input space as $x_1, ..., x_M$ and the ones of the transformes space as $u_1, ..., u_D$. The resulting output vector will be:

$$< u_1^T x_n, ..., u_i^T x_n, ..., u_D^T x_n >$$

A fundamental property of the input space is that all of its components are orthogonal, meaning that holds the property that $u_i \cdot u_j = 1$ if they are the same, 0 otherwise. These directions will follow the base vectors that maximize variance of the projected points.

Maximizing the variance

First, we find the mean of the data points $\bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n$. Then we find the data-centered matrix, which is the result of rescaling all data in order to have mean 0:

$$X = \begin{bmatrix} (x_1 - \bar{x}^T) \\ ... \\ (x_N - \bar{x})^T \end{bmatrix}$$

This procedure is also called normalization of the input data.

We can now fix a certain direction u_1 with the goal of using it to find the direction of maximum variance.

The mean for the projected points will be $u_1^T \bar{x}$. Their variance will correspond to the sum of the squared distances from the mean:

$$\frac{1}{N} \sum_{n=1}^{N} [u_1^T x_n - u_1^T \bar{x}]^2 = u_1^T S u_1$$

Where $S = \frac{1}{N} \sum_{n=1}^{N} (x_n - \bar{x})(x_n - \bar{x})^T = \frac{1}{N} X^T X$.

What we want is to solve the maximization problem by finding the direction that maximizes the variance:

$$max_{u_1} u_1^T S u_1$$

The solution is: $S u_1 = \lambda_1 u_1$. This means that u_1 is the eigenvector for S and λ_1 is its eigenvalue. Hence, the maximum direction will correspond to the maximum among the eigenvalues of S.

Here is a summary for this approach:

Given a dataset, we compute the data-centered matrix and the covariance matrix S. From S, we extract the λ_i eigenvalues and we order them in a decreasin order. If we are transforming the data into a D-dimensional space, we take the D highest eigenvalues, that correspond to the ones that will be the optimum for the variance maximizaton.

The largest higenvalue of the correlation matrix λ_i is called **principal component**.

Each transformed data point will be represented as:

$$x_n = \sum_{i=1}^{D} \alpha_{ij} u_i$$

Where $\alpha_{ij} = x_n^T u_j$. By exploiting the property of orthonormality of the transformed space, we can reduce the expression as:

$$x_n = \sum_{i=1}^{D} (x_n^T u_i) u_i$$

When reducing the representation to an M dimensional space, we are approximating the point in lower dimensions. Hence, we can express this approximation as the results of two contributions:

$$\tilde{x} = \sum_{i=1}^{M} z_{ni} u_i + \sum_{i=M+1}^{D} b_i u_i$$

Now we can define the error between the original point and the approximated one as the summ of the squared distance between them:

$$J = \frac{1}{N} \sum_{i=1}^{N} ||x_n - \tilde{x}_n||^2$$

How do we reduce this error?

Remember that the approximation is made up by two contributions. The first one is given by the coefficients $z_{ni} = x_n^T u_j \text{ for } i = 1, ..., M$ which is computed from the actual points, while the real error depends on the last $M + 1, ..., D$ contributions, that only depend on the mean of the contributions. Thus, the actual representation of the error is:

$$x_n - \tilde{x}_n = \sum_{i=M+1}^{D} [(x_n - \bar{x})^T u_i] u_i$$

If we sum the contribution of the errors for all the samples in the dataset, then we obtain:

$$J = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=M+1}^{D} (x_n^T u_i - \bar{x}^T u_i)^2 = \sum_{i=M+1}^{D} u_i^T S u_i$$

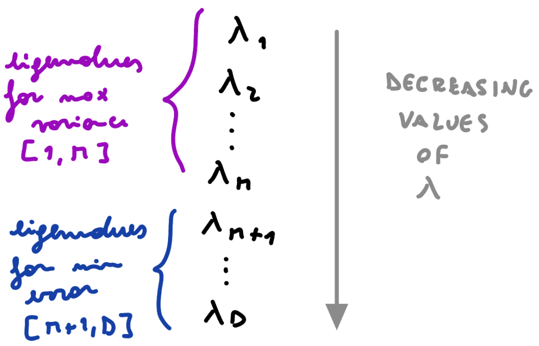
What we have obtaines is a representation of the error based on the covariance!

The formulation we see in particular is the same that we've seen when maximizing the variance, thus, the solutions of the minimization problem for the error will depend on the same relationship as before:

$$Su_i = \lambda_i u_i$$

This means that the minimization of the error will depend on the eigenvalues of S. In order to obtain the minimum error, we will simpy have to choose the $M+1, ..., D$ lowest eigenvalues of S.

This means that the whole problem of maximizing variance and minimizing the approximation error can be boiled down to a single operation. In fact, if we choose the highest M eigenvalues for the maximization of variance, we will automatically obtain the remaining $M+1, ..., D$ eigenvalues that minimize the error:



PCA for high dimensional data

In some cases, it can happen that the number of samples N is smalled than the number of dimensions D of the space they belong to. When this happens, computing the eigenvalues of the covariance matrix is inefficient, since the computation will have to deal with a total of $D \times D$ dimensions.

In cases where $N < D$ we can make a strategic choice for the computation of the eigenvalues by considering the matrix XX^T, which is an $N \times N$ dimensional matrix. Using this trick will allow us to find the same eigenvalues that we need, but searching in a smalled space.

Probabilistic PCA

With probabilistic PCA we define a linear relationship between the actual data $x \in \mathbb{R}^D$ and lower dimentional latent variable $z \in \mathbb{R}^M$. The linear realationship is modeled as:

$$x = Wz + u$$

We make two fundamental assumptions. The first one is that the latent variables follow a normal Gaussian Distribution:

$$P(z) = \mathcal{N}(z; 0, I)$$

The second one is that the conditional probability of x given z follows a linear Gaussian relationship:

$$P(x|z) = \mathcal{N}(x; Wz + \mu, \sigma^2 I)$$

An interesting way of interpreting this probabilistic relationship is that it can be used as a generative model. In fact if we know all the parameters $< W, \mu, \sigma >$, we can sample new data from the obtained distribution based on the values of z.

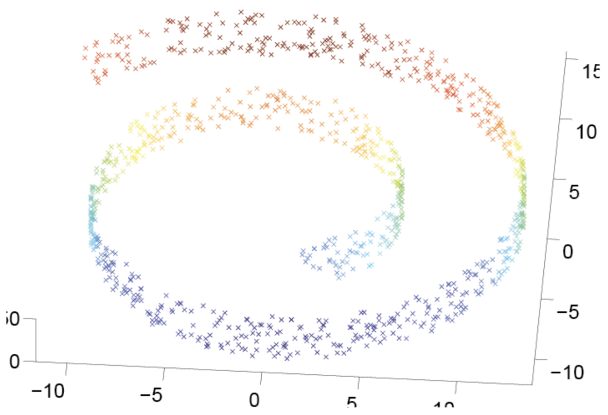
The goal with probabilistic PCA is to find the parameters $< W, \mu, \sigma >$ that maximize the likelihood of the previous relationship:

$$argmax_{W,\mu,\sigma}\ P(x|W, \mu, \sigma^2) = \sum_{n=1}^{N} ln\ P(x_n|W, \mu, \sigma^2)$$

When we set the derivatives to 0, we obtain the solution of the maximization problem in a closed form (here we omit the actual solution).

Non-linear latent variables models

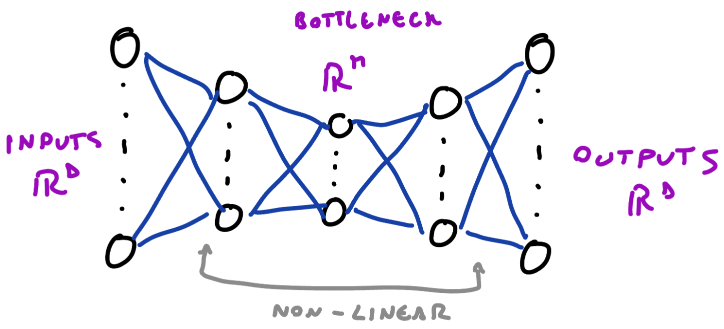
Up until now, we've seen that PCA assumes a linear relationship between the data and the latent variables, but actually this is not always the case. Sometimes there is the need for transforming data that follow a non-linear pattern, like the ones shown in the following image:



In order to deal with these problem we need to use non-linear models, and neural networks serve well this purpose. The following methods show how we can exploit them in various cases.

Autoencoders

Autoencoders are a special type of neural networks in terms of architecture. In particular the input and output layer will be of the same dimensions D, while the hidden layer will be of a lower dimension M (called bottleneck):

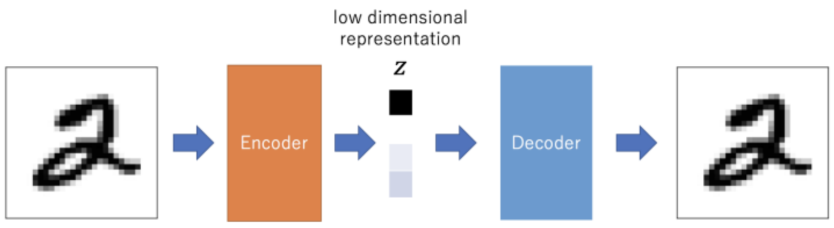


They work by transforming the input x_n into an intermediate representation $z \in \mathbb{R}^M$ and finally reconstructing it into the output x'_n.

An ideal model will perfectly reconstruct the given input such that $x_n = x'_n$.

The great intuition here is that we train the network in order to perform well this operation, the model will have learned how to represent the input space into a reduced space by loosing the minimum amount of information.

The training of an autoencoder is done by giving x_n both as input and as target value. The representation of x_n in the latent space is denoted as x_n.

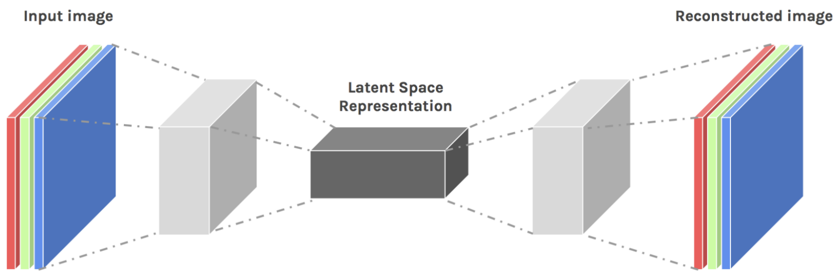


The first part of the network that transforms the input space is called encoder, while the one that reconstructs the output from the latent space is the decoder.

Notice that after training, if we remove the encoder, we can use the decoder as a generative model that outputs new samples starting from the values assigned to the latent space.

Convolutional autoencoders

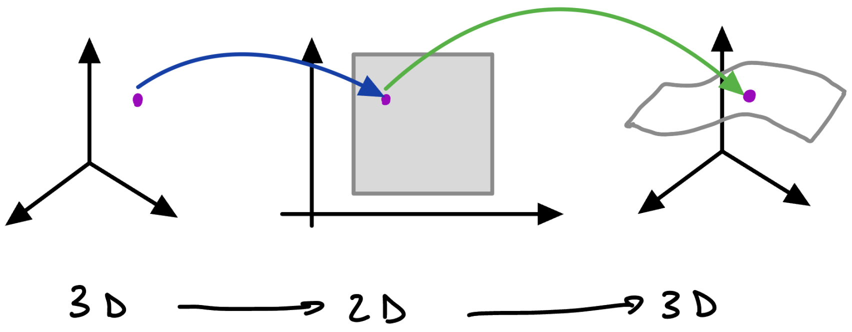
In case of high dimensional data such as images we use convolutional autoencoders, which exploit the concept of convolutional operations to encode the inputs.



The representation in the latent space will then be decoded by a convolutional transposed layer, which in brief does the reverse convolution operation. As with normal autoencoders, we use the same images as input and targets in order to train the network. By doing this, we will be able to use convolutional autoencoders to generate new images.

What is happening behind the scenes?

When using autoencoders we are using a non linear transformation to encode the data in a lower dimensional space. This data will then be re-projected onto the original higher dimensional space, and will tipically lie on a manifold. The following is a representation of what happens when transforming a 3D input in 2D:



Autoencoders for anomaly detection

There are cases when we want to understand whether a certain input is not part of the dataset. This field of study is called anomaly detection and aims at telling apart abnormal that. Since we want to distinguish normal from abnormal, the problem is also called one-class classiication.

A problem when considering what data to consider is that collecting samples of anomalies does not always make sense, since new anomalies are not predicatble (new anomalies can be very different form pas ones). Thus the approach is based on training only on a dataset that represent normal samples.

Remember that the task is basically a binay classification where we eant to find a function such that $f : X \to \{normal, abnormal\}$. The problem is that we have a dataset that contains only samples from one class $D = \{(x_n normal)_{n=1}^N\}$.

How do we deal with that?

A simple but effective solution is the following:

1. Start from a dataset containing unlabelled data $D = \{x_n\}$.
2. Train the autoencoder on the dataset and compute the final training loss.
3. Define a certain threshold based on the loss. For example $\delta = mean(loss) + std(loss)$
4. When we have a new sample x', we can try to use the autoencoder to reconstruct it. We then compute the loss of the autoencoder on this reconstruction. If this loss is higher than the threshold, we consider the new sample as an anomaly, otherwise we consider it as normal.

This method works better when used on an ensemble of autoencoder, where we use the average of the losses for each reconstruction.

Generative models

As we have seen, autoencoders can be used to generate new samples in the input space, but it's not the purpose they were born to serve. Instead, there are models that were specifically designed for generating new data that is similar to a distribution. The goal of these models is to find a dimensionality reduction that is able to reconstruct an output that is as similar as possible to the input. In particular, we will focus on Variable Autoencoders and Generative adversarial networks.

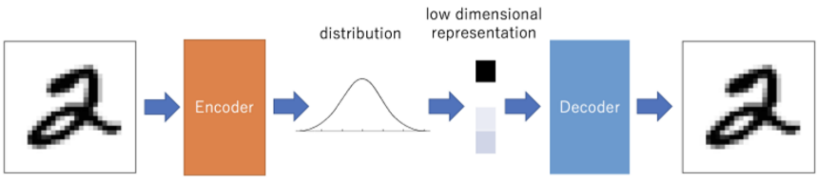
Variational Autoencoders (VAEs)

Classical autoencoders are very good at encoding informations in a compact latent space, but despite the latent space can be used to generate new sample, we don't have proper control on the generation process. In fact since there is no constraint on how the latent space is organized, two points very closed to each other can produce really different outputs.

We have to find a way to properly characterize regions of the latent space in order to map them to determined outputs in the originalspace. Variational autoencoder serve this purpose by making sure that the distributions in the latent space are well defined in terms of Gaussian distribution.

This means that VAEs will learn the distribution of certain data in terms of Gaussians, such that points that are clos in the latent space are also similar in the latent space. This will give us much more control on the process of generations, since we know that small perturbation in the latent space will produce similar results.

VAEs implement this concept by defining the latent space in terms of probability distributions that dipend on the parameters μ, σ. Therefore, the encoder will produce a distribution instead of a vector in the latent space, while the function of the decoder will be to sample from the computed distributions:



How do we produce these distributions?

The model will still be trained by giving both as taget and input the same value, but what changes is the definition of the loss function, which looks like this:

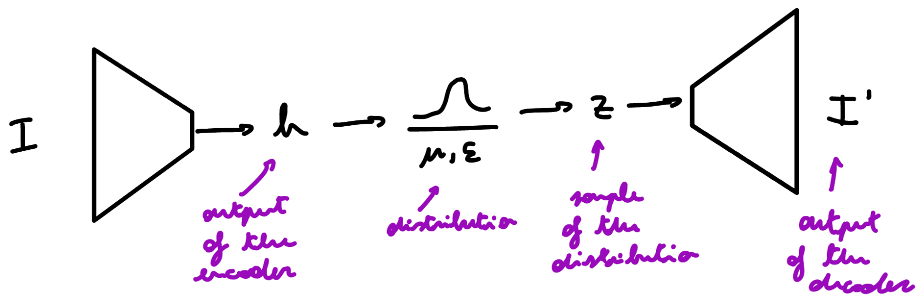
$$loss = MSQE + \lambda \cdot KL(\mathcal{N}(z; \mu, \Sigma), \mathcal{N}(z; 0, I))$$

Notice that while we kept the mean squared error term (that measures the difference from the original and the reconstructed sample), we added a new term KL. This new term is known as the Kullback-Leibler divergence, and is a method to measure the divergence between two distributions. In particular, this term will allow us to shape the latent space to be as close as possible to a normal distribution. The term λ will regulate the importance of this morphing.

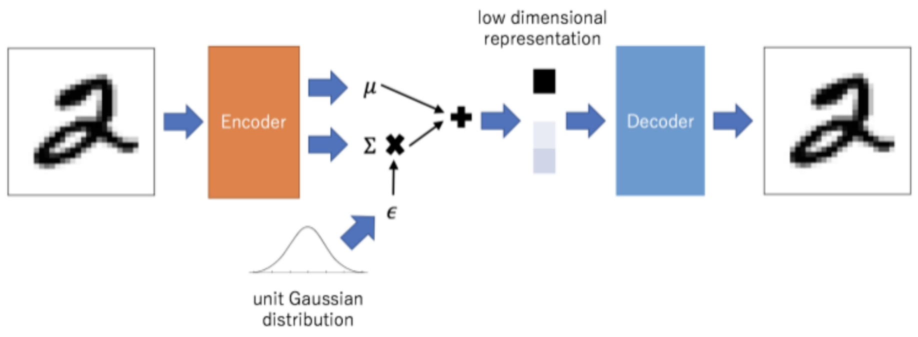
Minimizing the error will allow us to find the best tradeoff between a good reconstruction of the output and a good representation of the latent space.

And now, how we get an output from the obtained distribution?

We will need to sample from this distribution. Basically we want a method that works like this:



While this operation is easy to do in the forward step from input to output, we have a serious problem when implementing backpropagation. In fact we would have to perform some sort of reverse samplimng, but sampling is not differentiable. This problem is solved by substituting the distribution with a single value ϵ to the distribution (re-parametrization) when computing the gradient:

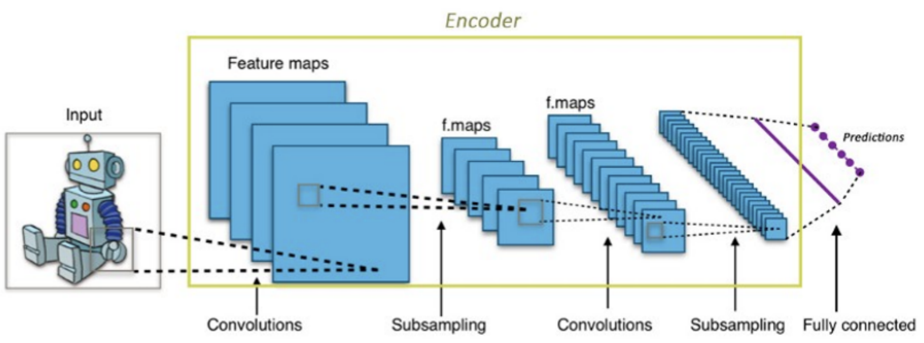


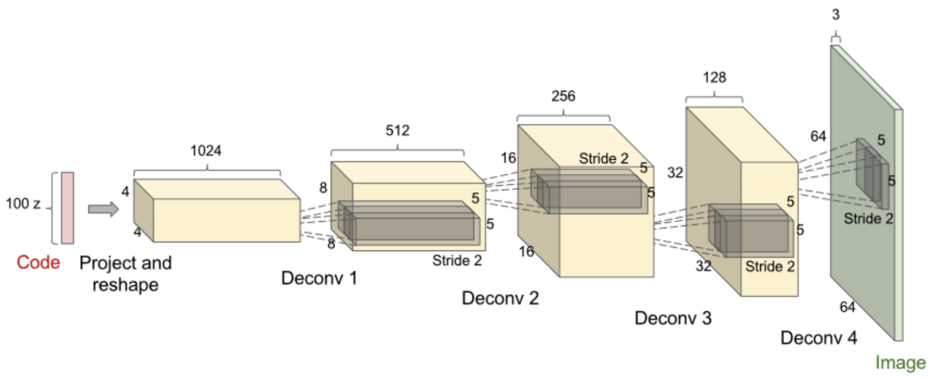
Generative Adversarial Networks (GANs)

Even if VAEs are better at the generation task, we have to remember that autoencoders were born for the task of dimensionality reduction. But when thinking about generative tasks, we don't actually need this feature. What we will study here is a family of models that were born with the specific purpose of generating data.

We will focus on a family of algorithms based on the concept of adversarial training. In particular, the core idea is to make two networks compete on the same task.

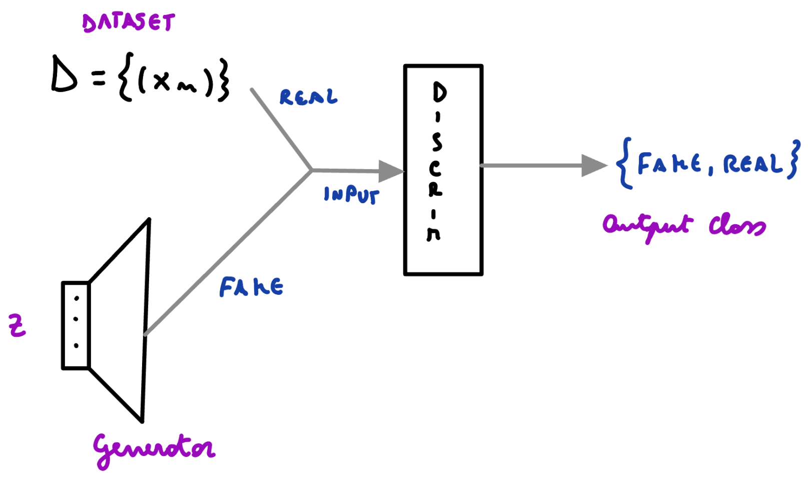
The typical archtecture of Generative Adversarial Networks is to have two components: a **discriminator** and a **generator**.





In particular, the discriminator is actually a decoder that works as usual. WIth respect to autoencoders, we won't need an encoder, but our only task will be to train the generator.

Our input will be an unlabelled dataset $D = \{(x_n)_{n=1}^N\}$. Now we introduce the discriminator, which is a binary classifier, whose task is to tell apart the images coming form the dataset (considered as real) from the ones generated by the generator (considered as fake). Basically, the discriminator has to understand whether an input data is real or fake.



Here, the two dicriminator and generator entities have different point of views. In fact the generator is good if he’s able to fool the discriminator, meaning that it can make it classify generated images as real. Instead, the discriminator works well when it’s able to properly lable generated data as fake. This is the core idea of the adversarial conept.

The training phase of a GAN has two main phases that are iterated:

1. Train the discriminator while freezing the weights of the generator.
2. Train the generator while freezing the weights of the discriminator.

During the first phase, the images produced by the generator will be labelled as fake such that the discriminator will keep getting better. Vice-versa, during the decond phae the images from the generator will be labelled as real.

The first time we train the discriminator, the generator’s latent space is initialized with random values.

When doing the training that while the loss of one entity increases, the other one decreases. We will stop training when the loss of the two components will converge to a minimum. Notice that we never want the loss of one compomenent to be zero, since this would mean that the other loss will diverge to infinity.

The GAN will work when tipically when the loss of the discirminator will reach about 50%.

Adversarial attacks to ML models

A very dangerous use of GANs is that they can be trained to fool already existing ML models. In particular this can be very impactfuls when using them against anomaly detectors.