Introduction to Variational Autoencoders (VAEs) and Implementation with Pytorch

References:

1. Beginner guide to Variational Autoencoders (VAE) with PyTorch Lightning
2. Variational Autoencoder Demystified With PyTorch Implementation
3. Understanding Variational Autoencoders (VAEs)
4. Youtube: Variational Autoencoders

The list of sections is given below:

1. Introduction

By virtue of a huge amount of data, deep learning based generative models have shown their powerful ability to produce highly-realistic pieces of content of various kinds, sunch as images, texts and sounds. In this note, we introduce one of the outstanding deep generative models called Variational Autoencoders (VAEs).

Face images generated with a Variational Autoencoder (source: Wojciech Mormul on Github).

In a nutshell, a VAE is an autoencoder whose encodings distribution is regularized during the training in order to ensure that its latent space has good properties allowing us to generate some new data. Moreover, the term “variational” comes from the close relation there is between the regularization and the variational inference method in statistics.

In Section 2, we will review some important notions about dimensionality reduction and autoencoders that will be useful for the understanding of VAEs. Then, in Section 3, we introuduce VAEs with its mathematical representation. At last, in Section 4, we give an implementation of VAEs on MNIST and cifar-10 using Pytorch.

2. Dimentionality reduction and autoencoders

In machine learning, dimensionality reduction is the process of reducing the number of features that describe some data. This reduction is done either by selection (only some existing features are conserved) or by extraction (a reduced number of new features are created based on the old features) and can be useful in many situations that require low dimensional data (data visualisation, data storage, heavy computation...). Although there exists many different methods of dimensionality reduction, we can set a global framework that is matched by most (if not any!) of these methods.

First, let’s call **encoder** the process that produce the “new features” representation from the “old features” representation (by selection or by extraction) and **decoder** the reverse process. Dimensionality reduction can then be interpreted as data compression where the encoder compress the data (from the initial space to the **encoded space**, also called **latent space**) whereas the decoder decompress them. Of course, depending on the initial data distribution, the latent space dimension and the encoder definition, *this compression can be lossy*, meaning that a part of the information is lost during the encoding process and cannot be recovered when decoding.

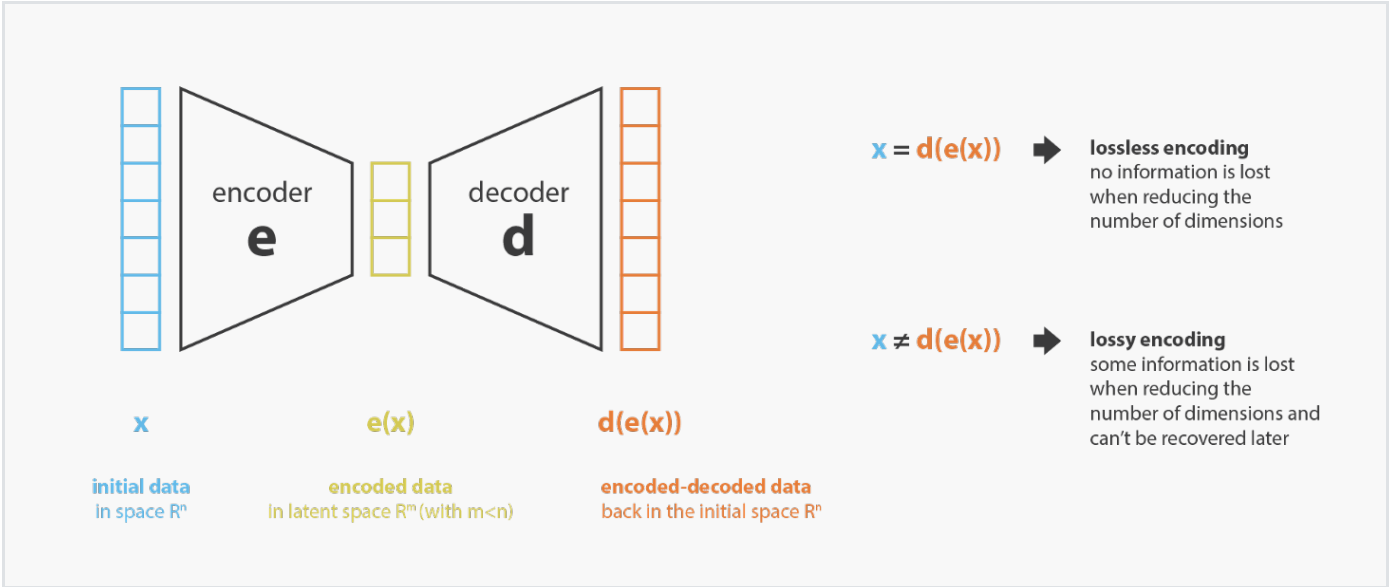


Illustration of the dimensionality reduction principle with encoder and decoder (source: [3]).

For a given set of possible encoders and decoders, we are looking for the pair that **keeps the maximum of information when encoding** and, so, **has the minimum of reconstruction error when decoding**. If we denote respectively E and D the families of encoders and decoders we are considering, then the dimensionality reduction problem can be written

$$(e^*, d^*) = \arg \min_{(e,d) \in E \times D} \epsilon(x, d(e(x)))$$

where $\epsilon(x, d(e(x)))$ is the reconstruction error. For example, we can use square loss

$$\epsilon(x, d(e(x))) = \|x - d(e(x))\|^2$$

Now, the general idea of autoencoders is pretty simple and consists in **setting an encoder and a decoder as neural networks** and to **learn the best encoding-decoding scheme using an iterative optimisation process**. So, at each iteration we feed the autoencoder architecture (the encoder followed by the decoder) with some data, we compare the encoded-decoded output with the initial data and backpropagate the error through the architecture to update the weights of the networks.

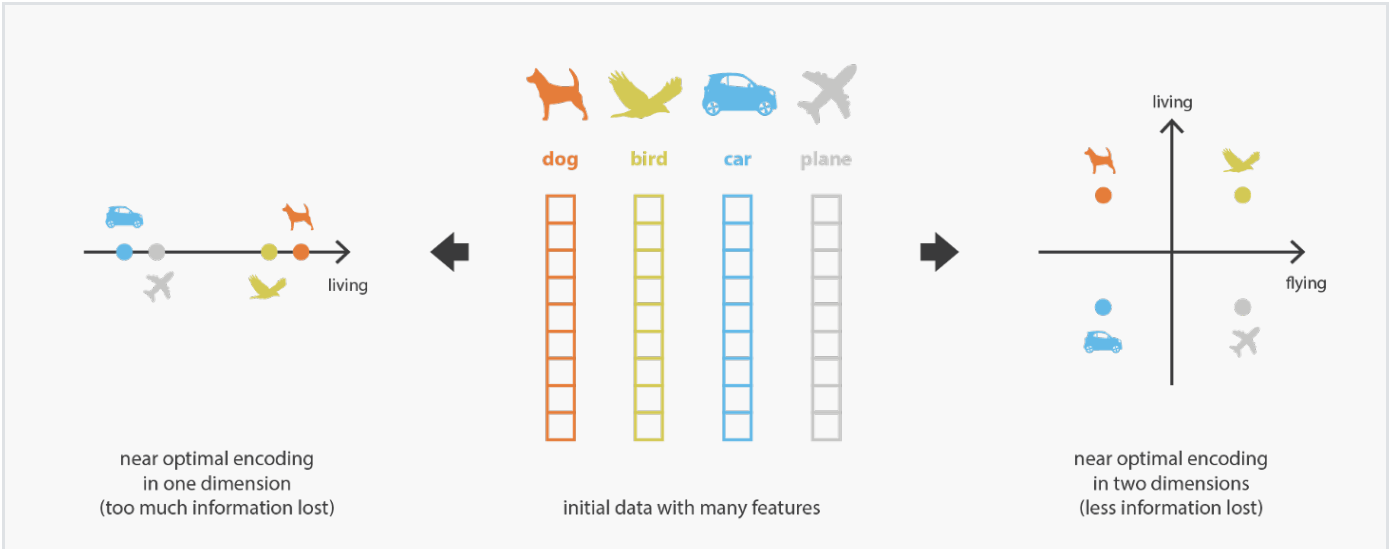
Thus, intuitively, the overall autoencoder architecture (encoder+decoder) creates a **bottleneck** for data that ensures only the main structured part of the information can go through and be reconstructed.

The more complex the architecture is, the more the autoencoder can proceed to a high dimensionality reduction while keeping reconstruction loss low. Intuitively, if our encoder and our decoder have enough degrees of freedom, we can reduce any initial dimensionality to 1. Indeed, an encoder with "infinite power" could theoretically takes our N initial data points and encodes them as 1, 2, 3, ... up to N and the associated decoder could make the reverse transformation, with no loss during the process.

Here, we should however keep two things in mind.

- First, an important dimensionality reduction with no reconstruction loss often comes with a price: the lack of interpretable and exploitable structures in the latent space (**lack of regularity**).
- Second, most of the time the final purpose of dimensionality reduction is not to only reduce the number of dimensions of the data but to reduce this number of dimensions **while keeping the major part of the data structure information in the reduced representations**.

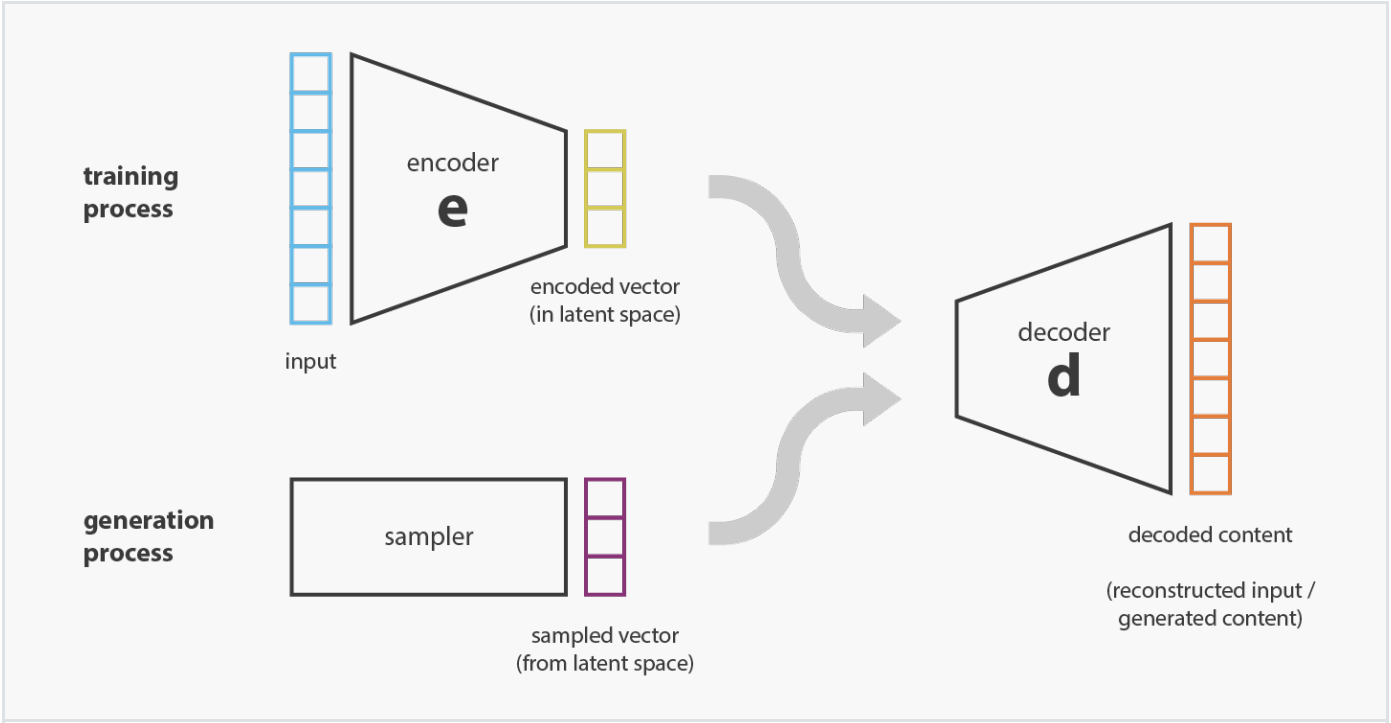
For these two reasons, the dimension of the latent space and the "depth" of autoencoders (that define degree and quality of compression) have to be carefully controlled and adjusted depending on the final purpose of the dimensionality reduction.



When reducing dimensionality, we want to keep the main structure there exists among the data (source: [3]).

Limitations of autoencoders for content generation

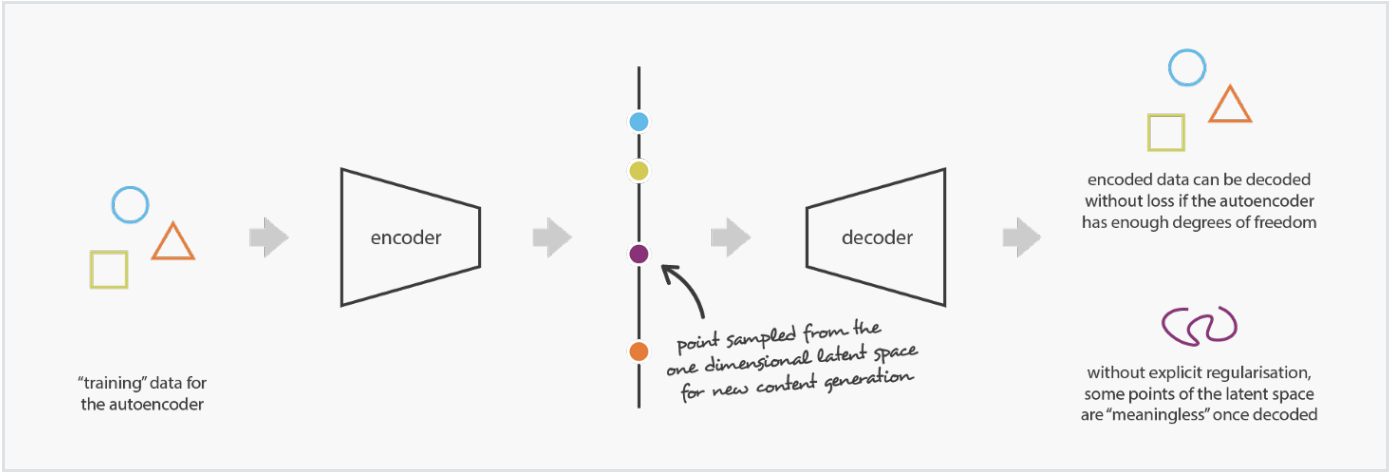
The problem is "what is the link between autoencoders and content generation?". We could be tempted to think that, if the latent space is regular enough (well "organized" by the encoder during the training process), we could take a point randomly from that latent space and decode it to get a new content.



We can generate new data by decoding points that are randomly sampled from the latent space. The quality and relevance of generated data depend on the regularity of the latent space (Source: [3]).

But it is pretty difficult (if not impossible) to ensure, a priori, that the encoder will organize the latent space in a smart way compatible with the generative process we just described.

To illustrate this point, let's consider the example we gave previously in which we described an encoder and a decoder powerful enough to put any N initial training data onto the real axis (each data point being encoded as a real value) and decode them without any reconstruction loss. In such case, the high degree of freedom of the autoencoder **leads to a severe overfitting** implying that some points of the latent space will give meaningless content once decoded.



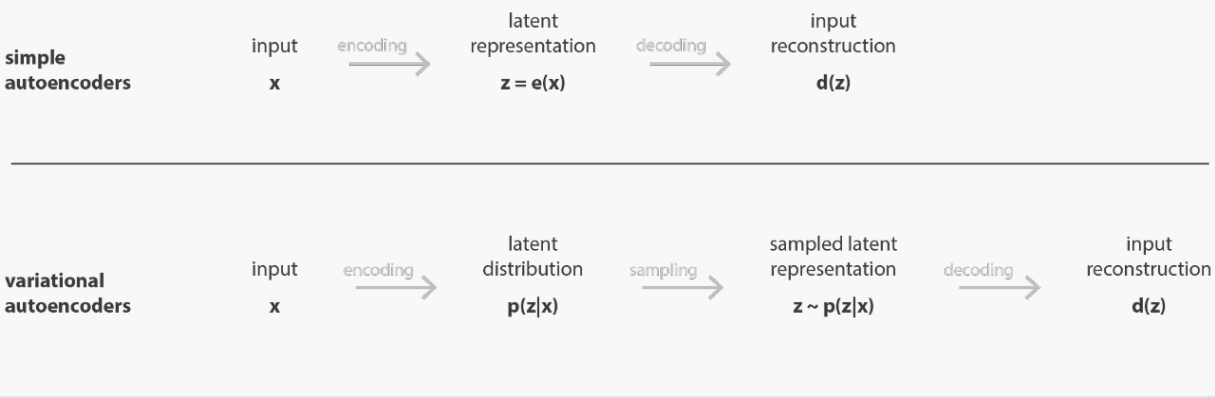
Irregular latent space prevent us from using autoencoder for new content generation (source: [3]).

When thinking about it for a minute, this lack of structure among the encoded data into the latent space is pretty normal. Indeed, nothing in the task the autoencoder is trained for enforce to get such organization: **the autoencoder is solely trained to encode and decode with as few loss as possible, no matter how the latent space is organized**. Thus, if we are not careful about the definition of the architecture, it is natural that, during the training, the network takes advantage of any overfitting possibilities to achieve its task as well as it can... unless we explicitly regularize it!

3. Variational autoencoders (VAEs)

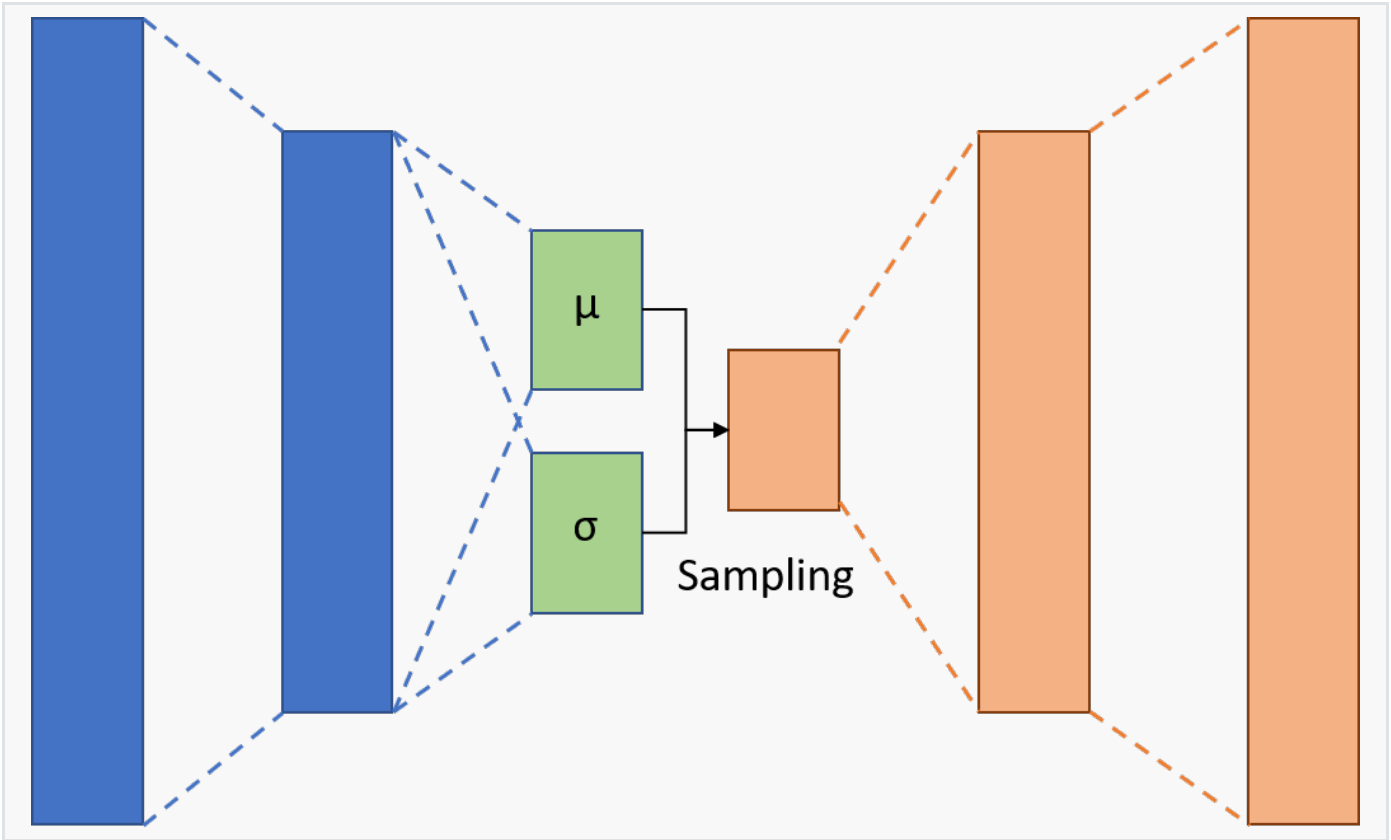
A variational autoencoder can be defined as being an autoencoder whose training is regularized to avoid overfitting and ensure that the latent space has good properties that enable generative process. Just as a standard autoencoder, a variational autoencoder is an architecture composed of both an encoder and a decoder and that is trained to minimize the reconstruction error between the encoded-decoded data and the initial data. However, in order to introduce some regularization of the latent space, we proceed to a slight modification of the encoding-decoding process: **instead of encoding an input as a single point, we encode it as a distribution over the latent space**. The model is then trained as follows:

- first, the input is encoded as distribution over the latent space
- second, a point from the latent space is sampled from that distribution
- third, the sampled point is decoded and the reconstruction error can be computed
- finally, the reconstruction error is backpropagated through the network



Difference between autoencoder (deterministic) and variational autoencoder (probabilistic) (source: [3]).

We assume the data is generated from a prior probability distribution and then try to learn how to derive the data from that probability distribution. This probability distribution will be a multivariate normal distribution $N(\mu, \Sigma)$ with no covariance. This allows the latent probability distribution to be represented by 2 n-sized vectors, one for the mean and the other for the variance.



VAE Architecture (source: [1])

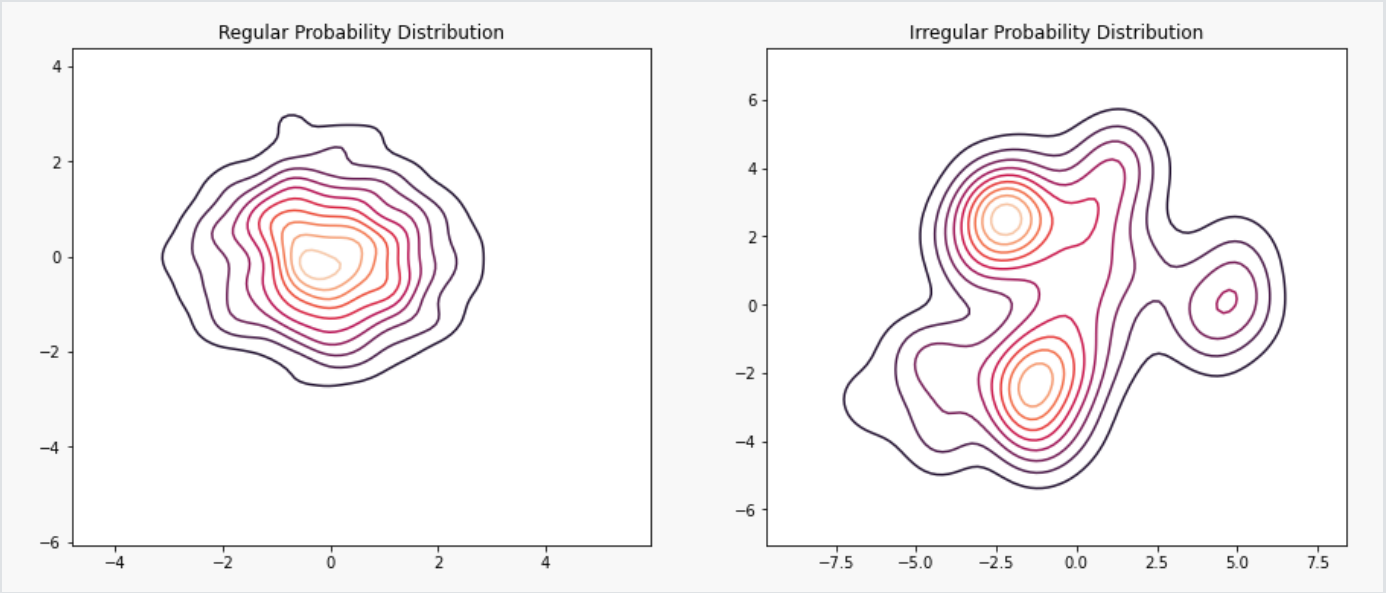
Regularization intuition

The only fact that VAEs encode inputs as distributions instead of simple points is not sufficient to ensure continuity and completeness. Without a well defined regularization term, the model can learn, in order to minimize its reconstruction error, **to "ignore" the fact that distributions are returned and behave almost like classic autoencoders** (leading to overfitting). To do so, the encoder can either return distributions with tiny variances (that would tend to be punctual distributions) or return distributions with very different means (that would then be really far apart from each other in the latent space). In both cases, distributions are used the wrong way (cancelling the expected benefit) and continuity and/or completeness are not satisfied.

The regularity that is expected from the latent space in order to make generative process possible can be expressed through two main properties:

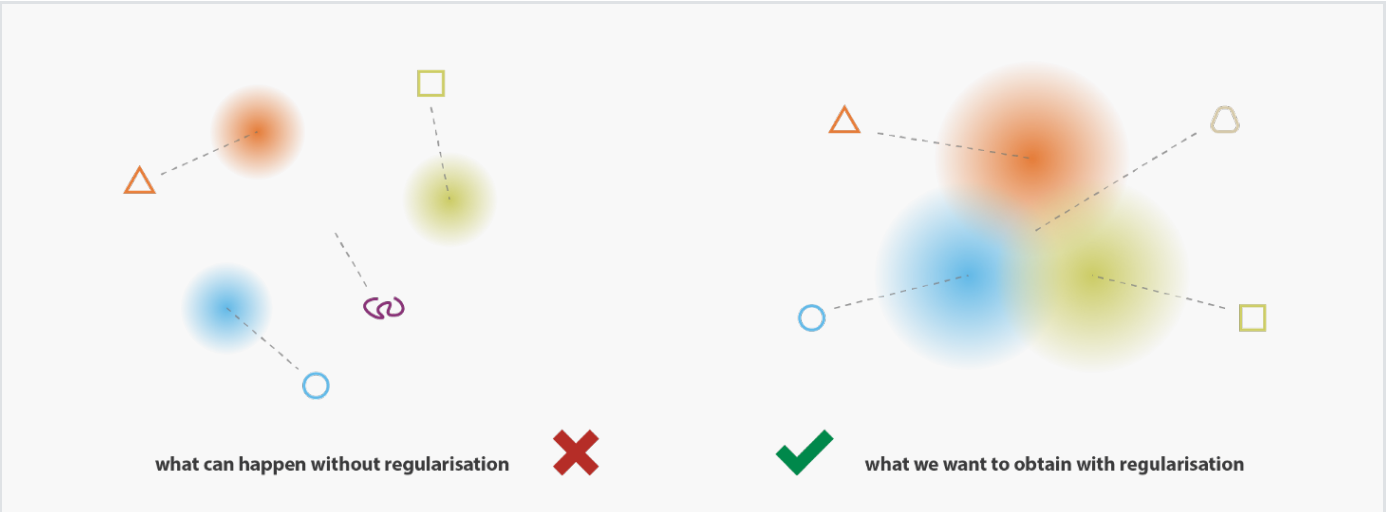
- **continuity**: two close points in the latent space should not give two completely different contents once decoded;
- **completeness**: for a chosen distribution, a point sampled from the latent space should give "meaningful" content once decoded.

Difference between a "regular" and an "irregular" latent space (source: [3]).



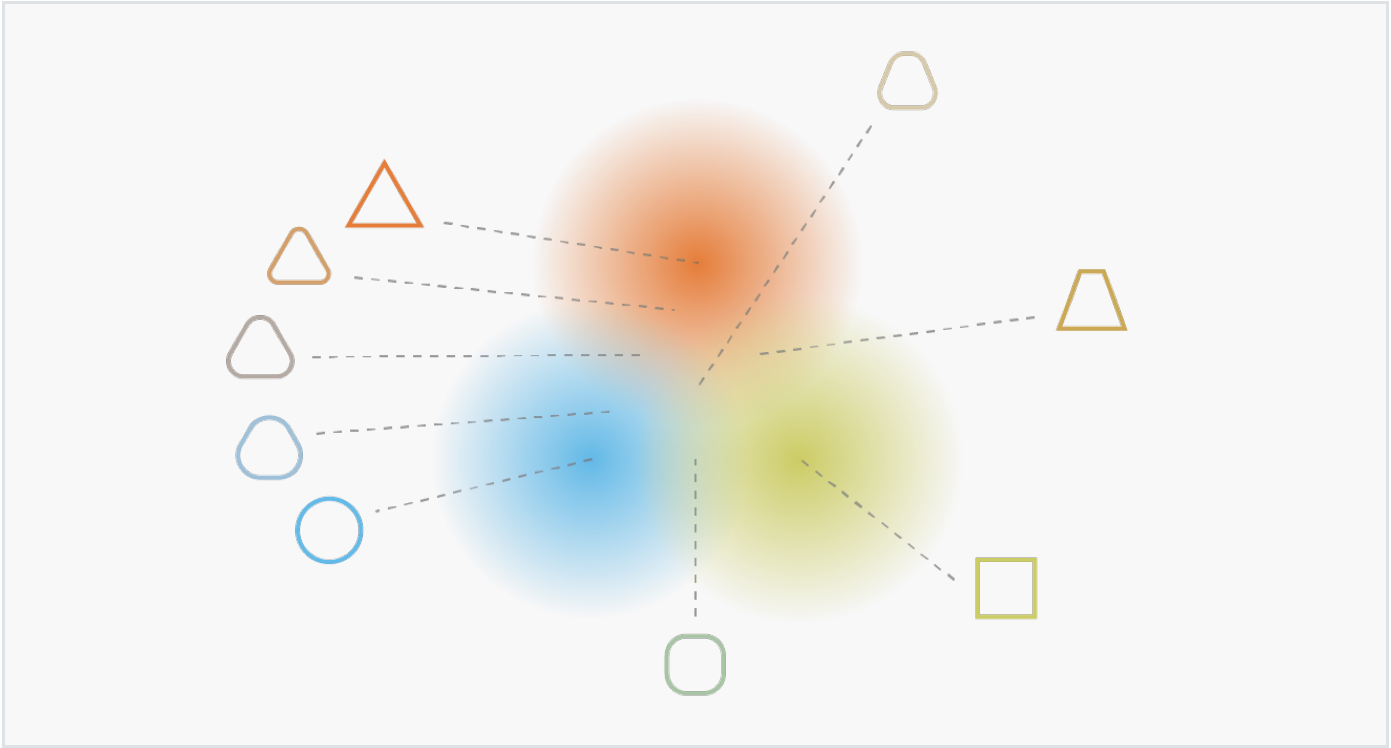
Irregular probability distributions can contain “pockets” where the decoder performs poorly because there are few examples that are mapped to that region (source: [1]).

So, in order to avoid these effects **we have to regularize both the covariance matrix and the mean of the distributions returned by the encoder**. In practice, this regularisation is done by enforcing distributions to be close to a standard normal distribution (centred and reduced). This way, we require the covariance matrices to be close to the identity, preventing punctual distributions, and the mean to be close to 0, preventing encoded distributions to be too far apart from each others.



The returned distributions of VAEs have to be regularized to obtain a latent space with good properties (source: [3]).

Now, we can observe that continuity and completeness obtained with regularization **tend to create a “gradient” over the information encoded in the latent space**. For example, a point of the latent space that would be halfway between the means of two encoded distributions coming from different training data should be decoded in something that is somewhere between the data that gave the first distribution and the data that gave the second distribution as it may be sampled by the autoencoder in both cases.

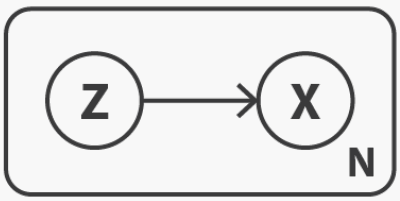


Regularization tends to create a “gradient” over the information encoded in the latent space (source: [3]).

Probabilistic framework and assumptions

Let’s begin by defining a probabilistic graphical model to describe our data. We denote by x the variable that represents our data and assume that x is generated from a latent variable z (the encoded representation) that is **not directly observed**. Thus, for each data point, the following two steps generative process is assumed:

- first, a latent representation z is sampled from the prior distribution $p(z)$
- second, the data x is sampled from the conditional likelihood distribution $p(x|z)$



Graphical model of the data generation process (source: [3]).

> Note: We only obeserve x.

With such a probabilistic model in mind, we can redefine our notions of encoder and decoder. The **“probabilistic decoder” is naturally defined by** $p(x|z)$, that describes the distribution of the decoded variable given the encoded one, whereas the **“probabilistic encoder” is defined by** $p(z|x)$, that describes the distribution of the encoded variable given the decoded one.

We make the following assumptions:

$$p(z) \equiv \mathcal{N}(0, I)$$
$$p(x|z) \equiv \mathcal{N}(f(z), cI) \qquad f \in F,\ c > 0$$

Here f is assumed to belong to a family of functions denoted F that is left unspecified for the moment and that will be chosen later. Let's consider, for now, that f is well defined and fixed. In theory, as we know $p(z)$ and $p(x|z)$, we can use the Bayes theorem to compute $p(z|x)$: this is a classical Bayesian inference problem. However, this kind of computation is often intractable (because of the integral at the denominator) and require the use of approximation techniques such as variational inference which makes the approach pretty general and more robust to some changes in the hypothesis of the model.

Variational inference formulation

In statistics, variational inference (VI) is a technique to approximate complex distributions. The idea is to set a parametrised family of distribution and to look for the best approximation of our target distribution among this family. The best element in the family is one that minimize a given approximation error measurement (most of the time the **Kullback-Leibler divergence** between approximation and target) and is found by gradient descent over the parameters that describe the family.

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

KL Divergence for Continuous Probability Distributions (source: Wikipedia)

Here we are going to approximate $p(z|x)$ by a Gaussian distribution $q_x(z)$ whose mean and covariance are defined by two functions, g and h, of the parameter x. These two functions are supposed to belong, respectively, to the families of functions G and H that will be specified later but that are supposed to be parametrised. Thus we can denote

$$q_x(z) \equiv \mathcal{N}(g(x), h(x)), \quad g \in G,\ h \in H$$

So, we have defined this way a family of candidates for variational inference and need now to find the best approximation among this family by optimizing the functions g and h (in fact, their parameters) to minimize the Kullback-Leibler divergence between the approximation and the target $p(z|x)$. In other words, we are looking for the optimal g^* and h^* such that

$$
\begin{aligned}
(g^*, h^*) &= \underset{(g,h)\in G\times H}{\arg\min}\ KL(q_x(z), p(z|x)) \\
&= \underset{(g,h)\in G\times H}{\arg\min}\ \left(\mathbb{E}_{z\sim q_x}(\log q_x(z)) - \mathbb{E}_{z\sim q_x}\left(\log \frac{p(x|z)p(z)}{p(x)}\right)\right) \\
&= \underset{(g,h)\in G\times H}{\arg\min}\ (\mathbb{E}_{z\sim q_x}(\log q_x(z)) - \mathbb{E}_{z\sim q_x}(\log p(z)) - \mathbb{E}_{z\sim q_x}(\log p(x|z)) + \mathbb{E}_{z\sim q_x}(\log p(x))) \\
&= \underset{(g,h)\in G\times H}{\arg\max}\ (\mathbb{E}_{z\sim q_x}(\log p(x|z)) - KL(q_x(z), p(z))) \\
&= \underset{(g,h)\in G\times H}{\arg\max}\ \left(\mathbb{E}_{z\sim q_x}\left(-\frac{||x - f(z)||^2}{2c}\right) - KL(q_x(z), p(z))\right)
\end{aligned}
$$

In the second last equation, we can observe the tradeoff there exists — when approximating the posterior $p(z|x)$ — between maximizing the likelihood of the "observations" (maximization of the expected log-likelihood, for the first term) and staying close to the prior distribution (minimization of the KL divergence between $q_x(z)$ and $p(z)$, for the second term). This tradeoff is natural for Bayesian inference problem and express the balance that needs to be found between the confidence we have in the data and the confidence we have in the prior.

For the second term, since $p(z)$ is standard normal, if we assume $q_x(z)$ is a multidimensional Gaussian distribution with diagonal covariance matrix (variables independence assumption), i.e., $q_x(z) \equiv \mathcal{N}\left((\mu_1, \cdots, \mu_k)^T, diag\{\sigma_1^2, \cdots, \sigma_k^2\}\right)$, then we have

$$
KL(q_x(z), p(z)) = \frac{1}{2}\sum_{i=1}^{k}(\sigma_i^2 + \mu_i^2 - 1 - \ln(\sigma_i^2))
$$

Thus, for every fixed $f \in F$, we can get the best approximation of $p(z|x)$, denoted $q_x^*(z)$. Then, we want to choose the function f that maximizes the expected log-likelihood of x given z when z is sampled from $q_x^*(z)$. **In other words, for a given input x, we want to maximize the probability to have $\hat{x} = x$ when we sample z from the distribution $q_x^*(z)$ and then sample $\hat{x}$ from the distribution $p(x|z)$.** Thus, we are looking for the optimal f^* such that

$$
f^* = \underset{f\in F}{\arg\max}\, \mathbb{E}_{z\sim q_x^*}(\log p(x|z)) = \underset{f\in F}{\arg\max}\, \mathbb{E}_{z\sim q_x^*}\left(-\frac{\|x - f(z\|^2)}{2c}\right)
$$

Gathering all the pieces together, we are looking for optimal f^*, g^* and h^* such that

$$
(f^*, g^*, h^*) = \arg\max\left(\mathbb{E}\left(-\frac{\|x - f(z)\|^2}{2c}\right) - KL(q_x(z), p(z))\right)
$$

We can identify in this objective function the elements introduced in the intuitive description of VAEs given in the previous section: the reconstruction error between x and $f(z)$ and the regularisation term given by the KL divergence between $q_x(z)$ and $p(z)$.

We can also notice the constant c that rules the balance between the two previous terms. The higher c is the more we assume a high variance around $f(z)$ for the probabilistic decoder in our model and, so, the more we favour the regularisation term over the reconstruction term (and the opposite stands if c is low).

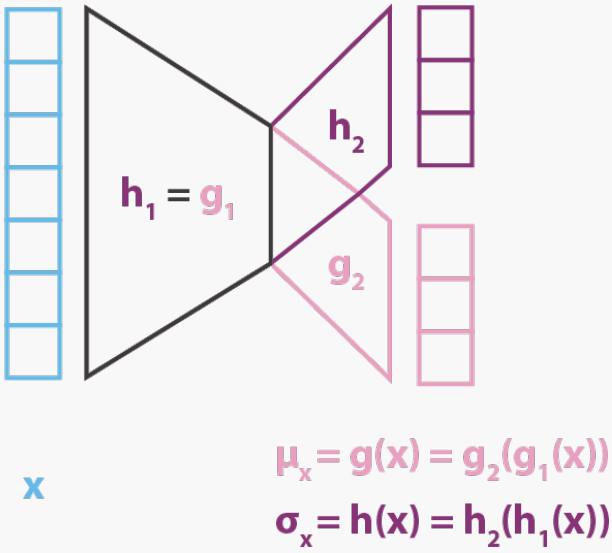
Bringing neural networks into the model

As we can't easily optimize over the entire space of functions, we constrain the optimization domain and decide to express f, g and h as neural networks. Thus, F, G and H correspond respectively to the families of functions defined by the networks architectures and the optimization is done over the parameters of these networks.

Recall that $q_x(z) \equiv \mathcal{N}(g(x), h(x))$. In practice, g and h are not defined by two completely independent networks but share a part of their architecture and their weights so that we have

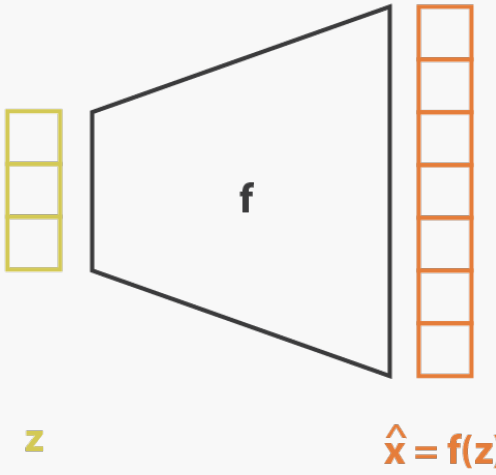
$$g(x) = g_2(g_1(x)) \qquad h(x) = h_2(h_1(x)) \qquad g_1(x) = h_1(x)$$

Here $h(x)$ is supposed to be a square matrix. However, in order to simplify the computation and reduce the number of parameters, we make the additional assumption that $q_x(z)$ is a multidimensional Gaussian distribution with diagonal covariance matrix (variables independence assumption). With this assumption, $h(x)$ is simply the vector of the diagonal elements of the covariance matrix and has then the same size as $g(x)$.



Encoder part of the VAE (source: [3]).

Contrarily to the encoder part that models $p(z|x)$ and for which we considered a Gaussian with both mean and covariance that are functions of x (g and h), our model assumes for $p(x|z)$ a Gaussian with fixed covariance. The function f of the variable z defining the mean of that Gaussian is modelled by a neural network and can be represented as follows



Decoder part of the VAE (source: [3]).

The overall architecture is then obtained by concatenating the encoder and the decoder parts.

If you look closely at the architecture, generating the latent representation from the $g(x)$ and $h(x)$ vector involves a sampling operation. The problem with the sampling operation is that it is a stochastic process and gradients cannot backpropagate back to the $g(x)$ and $h(x)$ vector.

For this, we utilize the **reparametrization trick** which allows us to separate the stochastic and deterministic parts of the operation. How this works is that we sample from a standard normal distribution $\mathcal{N}(0, I)$ and use the $g(x)$ and $h(x)$ vector to transform it. So the latent variable z is generated by

$$z = h(x)\zeta + g(x), \quad \zeta \sim \mathcal{N}(0, I)$$

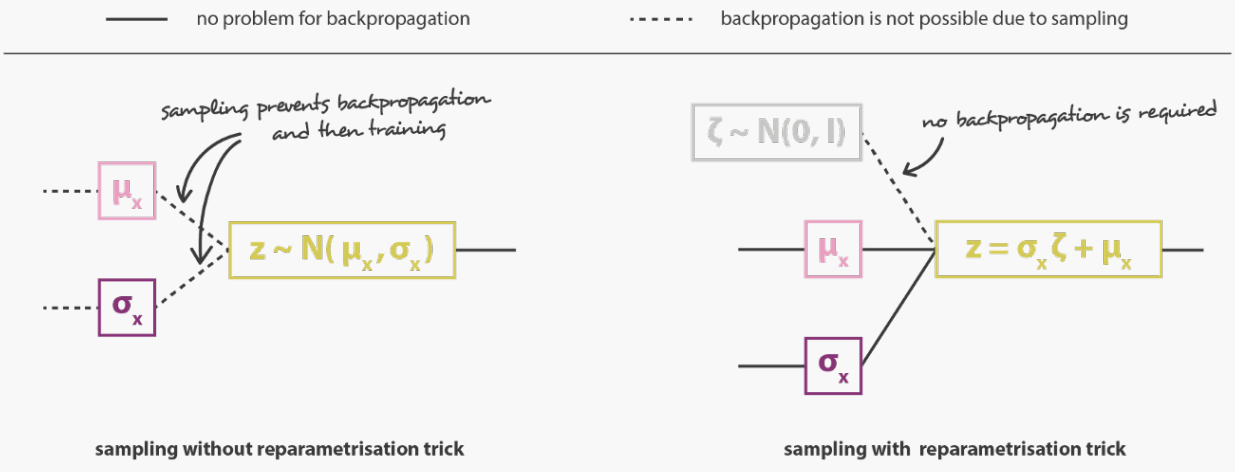
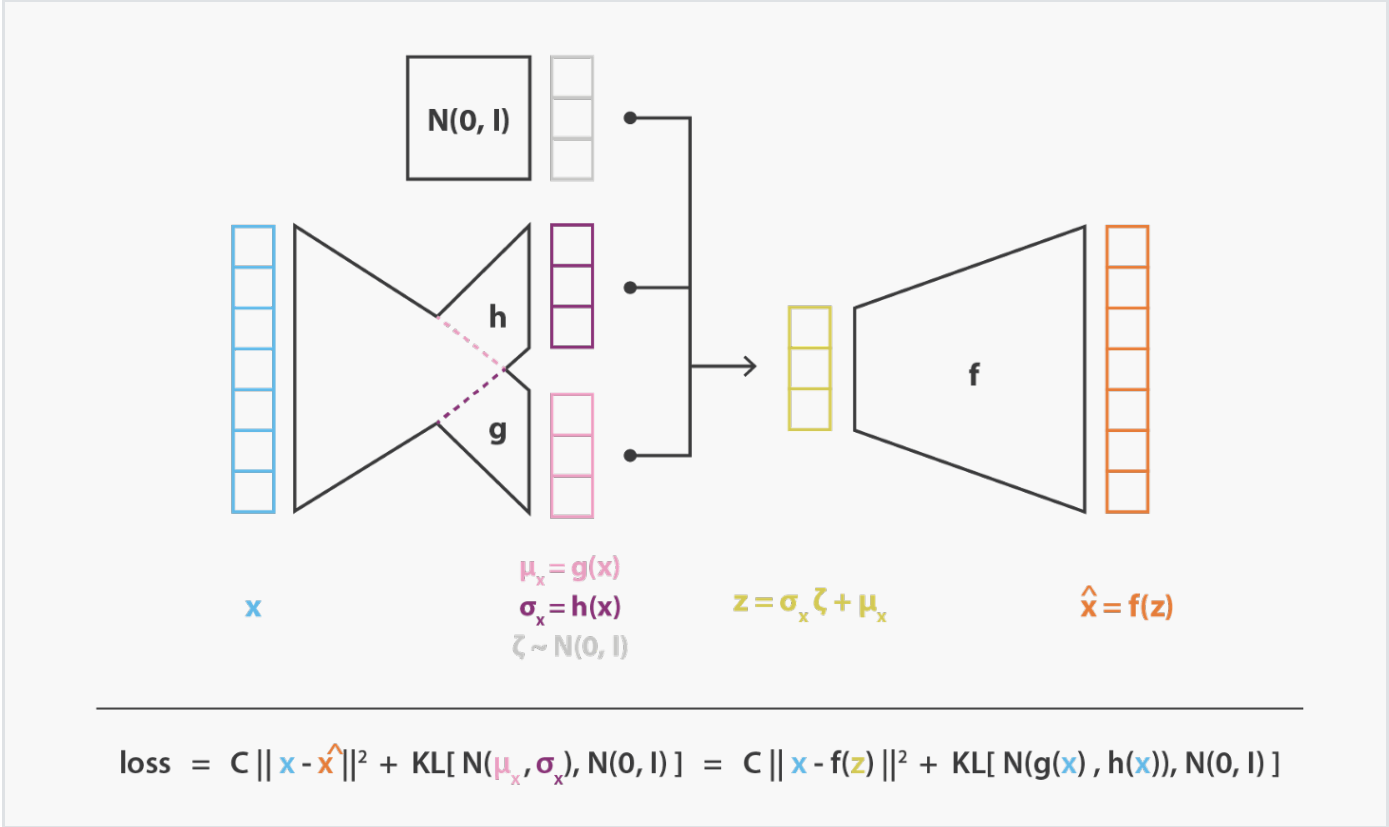


Illustration of the reparametrisation trick (source: [3]).

Finally, the objective function of the variational autoencoder architecture obtained this way is given by the last equation of the previous subsection in which the theoretical expectancy is replaced by a more or less accurate Monte-Carlo approximation that consists, most of the time, into a single draw. So, considering this approximation and denoting $C = 1/(2c)$, we recover the loss function derived intuitively in the previous section, composed of a reconstruction term, a regularization term and a constant to define the relative weights of these two terms.



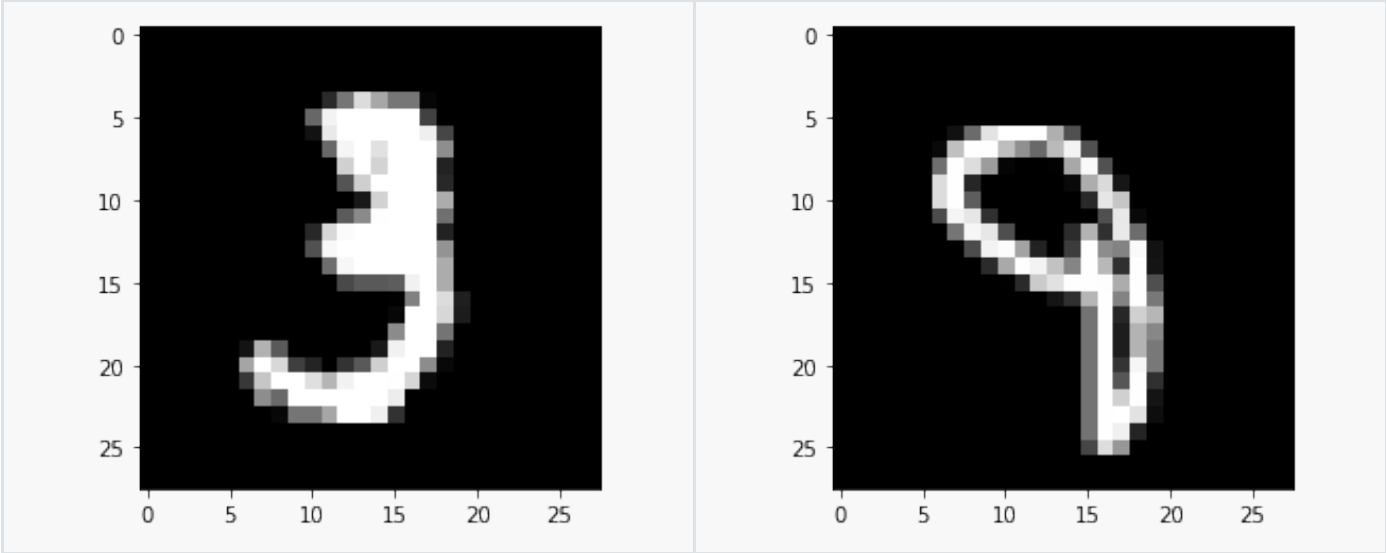
Variational Autoencoders representation (source: [3]).

4. Implementation

In this section, we give two examples to show how we use Pytorch to implement VAEs. Although we specifically use two different datasets, for simplicitgy, we will not go through the details other than the VAEs model itself (the data preprocessing, training and testing provess, etc). And note that the main difference between the two datasets we are using (MNIST and cifar-10) is the number of channels. Images in MNIST are in grey mode and thus have only one channel, while images in cifar-10 have three channels ('R', 'G', 'B').

MNIST

MNIST is a very classic dataset in deep learning. In MNIST dataset, each image is 28x28 pixels wide and has a label an integer from 0 up to 9. Here we show two of the images in the dataset:



As expected with most images, many of the pixels share the same information and are correlated with each other. This also means that the information in these pixels is largely redundant and the same amount of information can be compressed. This compressed form of the data is a representation of the same data in a smaller vector space which is also known as the latent space.

```
class VAEonMNIST(pl.LightningModule):
   def __init__(self,alpha = 1):
        #Autoencoder only requires 1 dimensional argument since input and output-
size is the same
        super().__init__()
        self.encoder =nn.Sequential(nn.Linear(784,196),
                                    nn.ReLU(),
                                    nn.BatchNorm1d(196,momentum = 0.7),
                                    nn.Linear(196,49),
                                    nn.ReLU(),
                                    nn.BatchNorm1d(49,momentum = 0.7),
                                    nn.Linear(49,28),
                                    nn.LeakyReLU())
        self.hidden2mu = nn.Linear(28,28)
        self.hidden2log_var = nn.Linear(28,28)
        self.alpha = alpha
        self.decoder = nn.Sequential(nn.Linear(28,49),nn.ReLU(),
                                     nn.Linear(49,196),nn.ReLU(),
                                     nn.Linear(196,784),nn.Tanh())

   def reparametrize(self,mu,log_var):
        #Reparametrization Trick to allow gradients to backpropagate from the
        #stochastic part of the model
        sigma = torch.exp(0.5*log_var)
        z = torch.randn(size = (mu.size(0),mu.size(1)))
        z= z.type_as(mu) # Setting z to be .cuda when using GPU training
        return mu + sigma*z

   def encode(self,x):
```

```
        hidden = self.encoder(x)
        mu = self.hidden2mu(hidden)
        log_var = self.hidden2log_var(hidden)
        return mu,log_var

    def forward(self,x):
        batch_size = x.size(0)
        x = x.view(batch_size,-1)
        mu,log_var = self.encode(x)
        hidden = self.reparametrize(mu,log_var)
        return self.decoder(hidden)
```

See the whole code here.

cifar-10

The following model is fully decoupled from the data. This means we can train on imagenet, cifar-10, or whatever you want. Hence, here we leave out discussion for the dataset and focus on the code. Again, we only present the part of the codes. The whole code can be seen here.

```
import pytorch_lightning as pl
from torch import nn
import torch
from pl_bolts.models.autoencoders.components import (
    resnet18_decoder,
    resnet18_encoder,
)


class VAE(pl.LightningModule):
    def __init__(self, enc_out_dim=512, latent_dim=256, input_height=32):
        super().__init__()

        self.save_hyperparameters()

        # encoder, decoder
        self.encoder = resnet18_encoder(False, False)
        self.decoder = resnet18_decoder(
            latent_dim=latent_dim,
            input_height=input_height,
            first_conv=False,
            maxpool1=False
        )

        # distribution parameters
        self.fc_mu = nn.Linear(enc_out_dim, latent_dim)
        self.fc_var = nn.Linear(enc_out_dim, latent_dim)

        # for the gaussian likelihood
```

```
        self.log_scale = nn.Parameter(torch.Tensor([0.0]))

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=1e-4)

    def gaussian_likelihood(self, x_hat, logscale, x):
        scale = torch.exp(logscale)
        mean = x_hat
        dist = torch.distributions.Normal(mean, scale)

        # measure prob of seeing image under p(x|z)
        log_pxz = dist.log_prob(x)
        return log_pxz.sum(dim=(1, 2, 3))

    def kl_divergence(self, z, mu, std):
        # --------------------------
        # Monte carlo KL divergence
        # --------------------------
        # 1. define the first two probabilities (in this case Normal for both)
        p = torch.distributions.Normal(torch.zeros_like(mu), torch.ones_like(std))
        q = torch.distributions.Normal(mu, std)

        # 2. get the probabilities from the equation
        log_qzx = q.log_prob(z)
        log_pz = p.log_prob(z)

        # kl
        kl = (log_qzx - log_pz)
        kl = kl.sum(-1)
        return kl

    def training_step(self, batch, batch_idx):
        x, _ = batch

        # encode x to get the mu and variance parameters
        x_encoded = self.encoder(x)
        mu, log_var = self.fc_mu(x_encoded), self.fc_var(x_encoded)

        # sample z from q
        std = torch.exp(log_var / 2)
        q = torch.distributions.Normal(mu, std)
        z = q.rsample()

        # decoded
        x_hat = self.decoder(z)

        # reconstruction loss
        recon_loss = self.gaussian_likelihood(x_hat, self.log_scale, x)

```

```
        # kl
        kl = self.kl_divergence(z, mu, std)

        # elbo
        elbo = (kl - recon_loss)
        elbo = elbo.mean()

        self.log_dict({
            'elbo': elbo,
            'kl': kl.mean(),
            'recon_loss': recon_loss.mean(),
            'reconstruction': recon_loss.mean(),
            'kl': kl.mean(),
        })

        return elbo
```