Chapter 09 - Generative AI Foundations

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Introduction

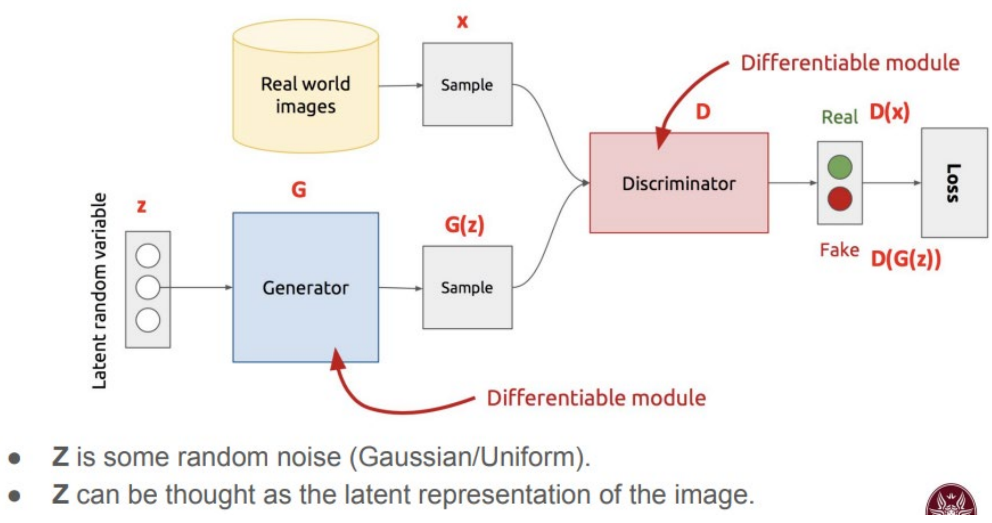
Machine learning models can be classified into:

- **Discriminative**: Make predictions based on conditional probability
- **Generative**: Model aims at returning a probability distribution
    - We model the distribution from which we able to generate samples.

2. Generative Adversarial Networks

GANs are called like this because they are:

- Generative = They generate data
- Adversarial = Because we use an adversarial training model
- Networks = Since we use Deep Learning



- Z is some random noise (Gaussian/Uniform).
- Z can be thought as the latent representation of the image.

z is a parameter controlling the generation process. Actually, it's a point in the latent space of images we want to generate. If we want to generate a person face, z is a latent representation of a person.

3. Training phase

In order to train the discriminator, we freeze the generator. Namely, during backpropagation, we do not backpropagate the gradient through the generator parameters. Then we start feeding discriminator with images that sometimes come from the generator and sometimes come from the real world, effectively computing the loss on this set of samples.

At the beginning the discriminator will have an easy job since the generator it's pure noise. We actually start generating images using the generator and we use the discriminator to classify those images (which are clearly fake). The backpropagation coming from this loss updates just the generator parameters and not the discriminator.

Whenever the discriminator will correctly guess the images, we give the generator the full gradient in order to try and produce images which look more like real images.

3.1 GAN Formulation

$$\min_G \max_D V(D,G)$$

- It is formulated as a minimax game, where:
- The Discriminator is trying to maximize its reward *V(D,G)*
- The Generator is trying to minimize Discriminator reward (or maximize its loss)

$$V(D,G) = \boxed{\mathbb{E}_{x\sim p(x)}[\log D(x)]} + \boxed{\mathbb{E}_{z\sim q(z)}\left[\log(1 - D(G(z)))\right]}$$

- The Nash equilibrium of this particular game is achieved at:

$$\boxed{\begin{array}{l} P_{data}(x) = P_{gen}(x) \;\; \forall x \\ D(x) = \frac{1}{2} \;\; \forall x \end{array}}$$

Nash theorem says that there is always a point of equilibrium between discriminator and generator such that we reach a state where both will work at their best.

This happens when the distribution that is learned by the generator is the same as the distribution of the data. The output of the discriminator is 0.5, meaning that it is completely undecided.

For convergence, we require that the discriminator D recognizes samples coming from the real distribution (we call them *positives*). $D(x) = 1$ when data comes from the real distribution.

The generator aims at tricking the discriminator (we call these *negatives*). In this case we want that the value $D(G(z))$ would be as close as possible to 1 for the generated images, such that we fool the discriminator.

$$\mathbb{E}_{z\sim P_z(z)} \log\left(1 - D\big(G(z)\big)\right)$$

Let's remember that the discriminator should return 0 when something is generated by the Generator network, but in this case, we have successfully made the discriminator return almost one.

Finally, we can conclude saying that the discriminator goal is to maximize the value function $V(D,G)$, while the generator tries to do the opposite, minimizing $V(D,G)$. Namely, we want to show that there exists a point of convergence of this value function.

We can actually simplify the loss of the GAN by moving the generation portion out of the expected value, through LOTUS theorem, as follows:

Applying LOTUS, we have that

$$E_{z\sim p_z(z)} \log(1 - D(G(z))) = E_{x\sim p_G(x)} \log(1 - D(x))$$

therefore

$$\int_x p_{data}(x) \log D(x)\, \mathrm{d}x + \int_z p(z) \log(1 - D(G(z)))\, \mathrm{d}z$$

$$= \int_x p_{data}(x) \log D(x) + p_G(x) \log(1 - D(x))\, \mathrm{d}x$$

Starting from this equation, we want to find the optimal value D given G though the derivative and through the use of the Leibniz rule which corresponds to a point where this the formula above is minimized, meaning that the derivative will be 0. We use the Leibniz rule since it helps us in bringing the derivative inside the integral.

Let's consider $f(y) = a \log y + b \log(1 - y)$

Then $f'(y) = 0 \Rightarrow \frac{a}{y} - \frac{b}{1-y} = 0 \Rightarrow y = \frac{a}{a+b}$

Therefore, $y = \frac{a}{a+b}$ corresponds to taking $D(x) = \frac{p_{data}}{p_{data} + p_G}$

What happens if $p_{data} = p_G$?

- $D_G^* = \frac{p_{data}}{p_{data} + p_G} = \frac{1}{2}$

The problem is that we do not know $P_{data}(x)$ or P_G a priori, but just knowing that they exist enables us to prove that an optimal G exists. If I set $P_{data}(x) = P_G$, the discriminator will be equal to 0.5 (which is what we wanted).

Another theorem says that the optimum for the generator G is achieved if and only if is equal to P_{data} and that this value is no less than -$log4$.

3.2 Modified Loss & Issues

The loss we have defined for the generator as:

$$\mathbb{E}_{z \sim P_z(z)} \log\left(1 - D\left(G(z)\right)\right)$$

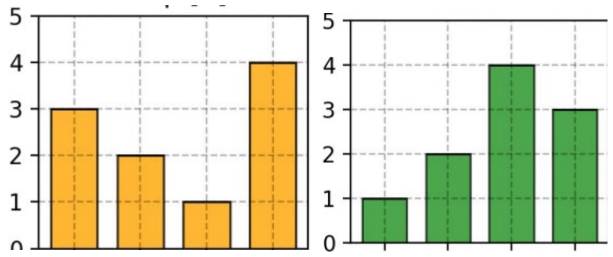
Might not be suitable enough for G to learn well (thus not providing enough gradient). Rather than training G to minimize that log function, we can train G to maximize directly $D(G(z))$, achieving much stronger gradients in early steps of the training process.

Many problems arise from the process of training a GAN:

- Collapsing of the network
- Large losses at the beginning
- Generator producing the same images
- High sensitivity to hyperparameters

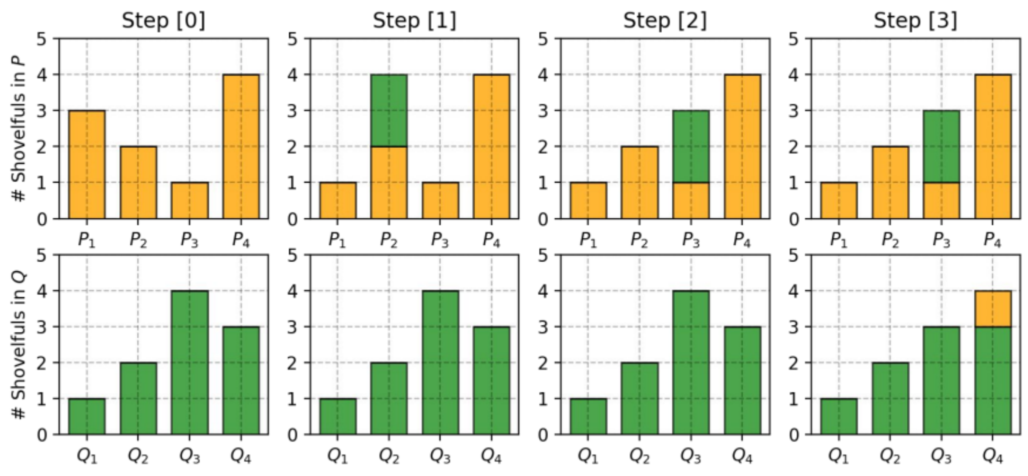
This somehow happens when the generator starts producing the same output over and over again and the discriminator's best strategy is to learn to always reject that output. But if the next generation of discriminator gets stuck in a local minimum and doesn't find the best strategy, then it's too easy for the next generator iteration to find the most plausible output for the current discriminator. Each iteration of generator over-optimizes for a particular discriminator and the discriminator never manages to learn its way out of the trap. As a result, the generators rotate through a small set of output types. This form of GAN failure is called mode collapse.

4. Wasserstein GANs



Wasserstein GANs are based on the Wasserstein Distance function. To introduce this let's imagine we have 2 different sets of piles of dirt. One (***P*** – yellow one) is composed by the first pile of 3 dirts, 2 dirts, 1 dirt and 4 dirts. Then we have a second set of piles of dirt (***Q*** – green one).

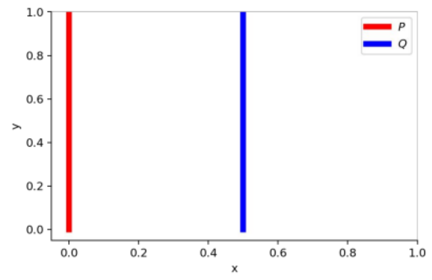
The Wasserstein distance is defined as the minimum number of movements of dirt that one need to make in order for the two distributions to match. Actually, we want to make the yellow piles of dirt equal to the green ones.



Step-by-step plan of moving dirt between piles in P and Q to make them match.

Movements in the first row (yellow) are positive. Movements on the second row (green) are negative.

We now know that JSD saturates as we can see in this plot:



- Two distributions, P and Q

$$\forall (x, y) \in P, x = 0 \text{ and } y \sim U(0, 1)$$
$$\forall (x, y) \in Q, x = \theta, 0 \leq \theta \leq 1 \text{ and } y \sim U(0, 1)$$

If we have a distance $\theta \neq 0$ between P and Q, when we compute the KL divergence, it diverges to $+\infty$.

If we compute the Johnson Shannon (JS), it's $log2$. The Wasserstein distance instead is exactly $|\theta|$.

But when $\theta = 0$, then even the Wasserstein distance is 0 because it's the absolute value of θ, which is actually 0. This distance provides then a smooth measure, which is super helpful for a stable learning process using gradient descents.

In fact, we can now rewrite the Wasserstein distance function as a loss:

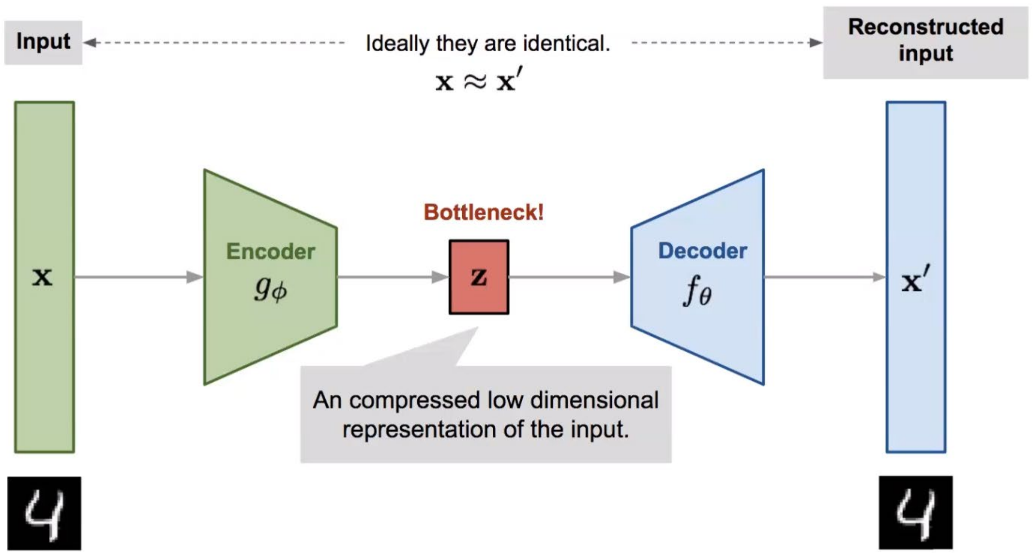
$$L(p_r, p_g) = W(p_r, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{z \sim p_r(z)}[f_w(g_\theta(z))]$$

Before explicating this loss though, one might consider that the distance between all the possible joints distributions (as per Wasserstein distance function definition) is not computable, so we take the sup (thanks also to the Kantorovich duality theorem).

In the modified Wasserstein-GAN, we replace the bad cost function depending on the discriminator and the generator etc. by a much nicer function that directly maximizes the similarity between p_r (real distribution) and p_g (distribution of the generated samples).

In this way we are just forcing the generator to have a distribution of data that is as close as possible (according to the Wasserstein distance) to the original distribution.

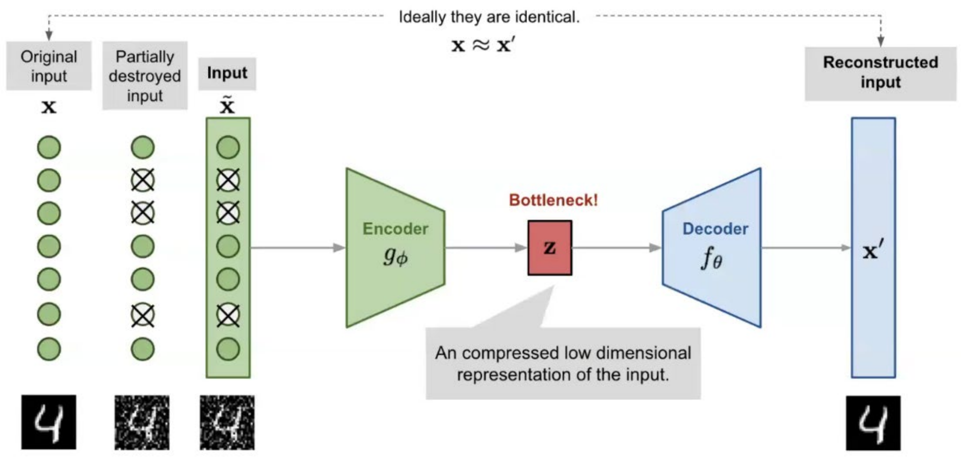
5. Autoencoders



Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

It takes an input X that gets encoded by the Encoder Network (using parameters ϕ) and generate a latent representation Z which has a much smaller dimension of X. Then, this gets decoded back to X' using a Decoder Network (using parameters θ). That's why it's called Autoencoders, since we want to decode exactly the inputs.

There is a small issue though, Z has not many information, it's not meaningful.

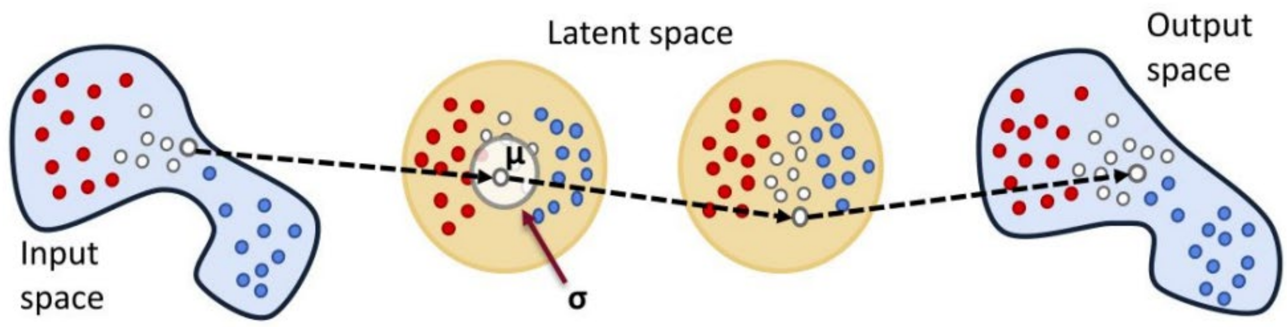


Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Original input
$\mathbf{x}$
Partially destroyed input
Input
$\tilde{\mathbf{x}}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
Reconstructed input
$\mathbf{x}'$
An compressed low dimensional representation of the input.

<u>DENOISING AUTOENCODERS</u>

In this case though, we inject a noisy input to the encoder in order to have a better representation of x' which is more robust to noise. We still compute the loss based on the original input, though.

6. Variational Autoencoders



In VAE, the idea is following: We have an input space that follows a distribution, but now I don't want to learn a representation for each point, but a representation for the whole input space.

This is better because if we can represent the probability distributions, we are actually able to generate, starting from that distribution, new samples. Moreover, in case of VAE, we take as input an input space and we try to identify a latent representation for that input space and we try to generate an output space that resembles the input one.

The autoencoder, then, is actually encoding the whole input and now just a point as in traditional encoders. The advantage is that we can actually move around the latent space, always generating valid points.

Let's label this distribution as p_θ , parameterized by θ. The relationship between the data input x and the latent encoding vector z can be fully defined by:

1. Prior $p_\theta(z)$ = Actually the shape of the latent space (usually a normal distribution)
2. Likelihood $p_\theta(x|z)$ = Used to move back from the latent space into the input space
3. Posterior $p_\theta(z|x)$ = Used to represent the input space in the latent space

Let's assume that we know the real parameters θ for this distribution. To generate a new sample, we sample the latent variable from the prior and with the parameters that we have learnt until that moment, we are ready to generate a new sample by conditioning on the generated latent vector z.

$$\theta^* = \arg\max_\theta \prod_{i=1}^{n} p_\theta(\mathbf{x}^{(i)})$$

Differently from GANs, where we use an adversarial algorithm, here we are using MLE (Maximum Likelihood Estimation) which is the product of all the probability of each sample in our dataset. That $x^{(i)}$ are the samples in our dataset.

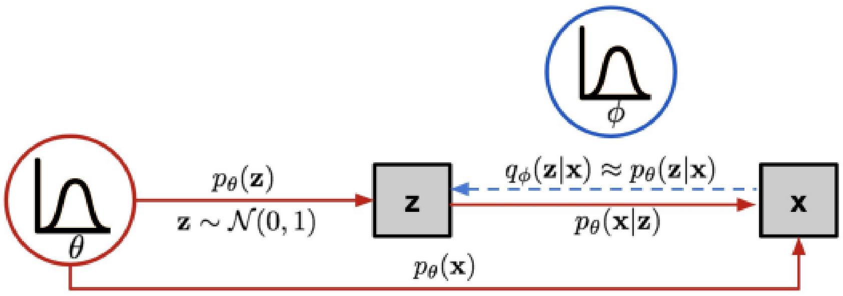
- The conditional distribution $p_{\theta^*}(x|z=z(i))$ used to generate data can be expanded as:

$$p_\theta(\mathbf{x}^{(i)}) = \int p_\theta(\mathbf{x}^{(i)}|\mathbf{z})p_\theta(\mathbf{z})d\mathbf{z}$$

- This integral is intractable.
  - It is very expensive to check all the possible values of **z** and sum them up.

This is exactly the whole VAE architecture, because it actually generates a point in the latent space and we materialize x from this point (Decoding part of the VAE).

Since there's no closed form to actually represent this integral, we can't compute MLE.



ϕ
$p_\theta(\mathbf{z})$
$\mathbf{z} \sim \mathcal{N}(0,1)$
$q_\phi(\mathbf{z}|\mathbf{x}) \approx p_\theta(\mathbf{z}|\mathbf{x})$
z
x
$p_\theta(\mathbf{x}|\mathbf{z})$
θ
$p_\theta(\mathbf{x})$

It means that taking as examples cats, representing the distribution of all possible cat images is extremely difficult. With VAEs, we want to be able to make new, realistic samples (like new cat images that haven't been seen before), so we introduce the latent space z, which has a simple, well-known distribution (like a Normal distribution). It's much easier to sample from and generate things from this simple distribution because in that way the integral is easy to compute and I can actually compute MLE.

In fact, I map the input in the latent space through $p_\theta(z)$ which is a Normal distribution.

Now, we take a real-world example (say, a cat image x) and try to find its simplified representation in the latent space (z): This is actually doing $p_\theta(z|x)$.

Going forward, we sample a point z from our simplified latent space and try to recreate a realistic-looking cat image: This is $p_\theta(x|z)$.

The problem now is that in order to pass from x to z, I don't want to use $p_\theta(z|x)$, because we said that it is intractable, but I will use another function $q_\phi(z|x)$ which depends on totally different parameters and it's, in fact, an approximation of $p_\theta(z|x)$.

- The estimated posterior $q_\phi(\mathbf{z}|\mathbf{x})$ should be very close to the real one $p_\theta(\mathbf{z}|\mathbf{x})$.
  - We can use Kullback-Leibler divergence to quantify the distance between these two distributions.
- KL divergence $D_{KL}(X\|Y)$ measures how much information is lost if the distribution Y is used to represent X.

In our case we want to minimize $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x}))$ with respect to ϕ.

6.1 Loss for VAE

Continuing, we want to find $q_\theta(z|x)$ that is as close as possible to $p_\theta(z|x)$ under the assumption that the function q is a Gaussian.

$$KL(Q_\phi(Z|X)||P(Z|X)) = \sum_{z \in Z} q_\phi(z|x) \log \frac{q_\phi(z|x)}{p(z|x)}$$

If we expand the KL divergence, it would be the sum over all the possible Z of q times q divided by p.

Through some mathematical and logarithm manipulations, we are able to rewrite the KL divergence as:

$$\log p_\theta(\mathbf{x}) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}))$$

- Negating:

$$\begin{aligned} L_{\text{VAE}}(\theta, \phi) &= -\log p_\theta(\mathbf{x}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) \\ &= -\mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z})) \\ \theta^*, \phi^* &= \arg\min_{\theta,\phi} L_{\text{VAE}} \end{aligned}$$

We rearrange these things because now we know how to compute that last expectation and the KL divergence between q and probability of z, arriving to the definition of our loss that we want to minimize, which we call **ELBO** (Evidence Lower Bound). We have two parameters θ and ϕ because one it's for the expectation and the other one is for the KL.

If we minimize this loss, we are actually maximizing $-L_{VAE}$ and given that the KL divergence is always greater than 0, then the $-L_{VAE}$ is always $\leq \log p_\theta(x)$:

$$-L_{\text{VAE}} = \log p_\theta(\mathbf{x}) - D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) \leq \log p_\theta(\mathbf{x})$$

The result of optimizing $-L_{VAE}$ is that we could get as close as possible to the logarithm of the original distribution which was our goal.

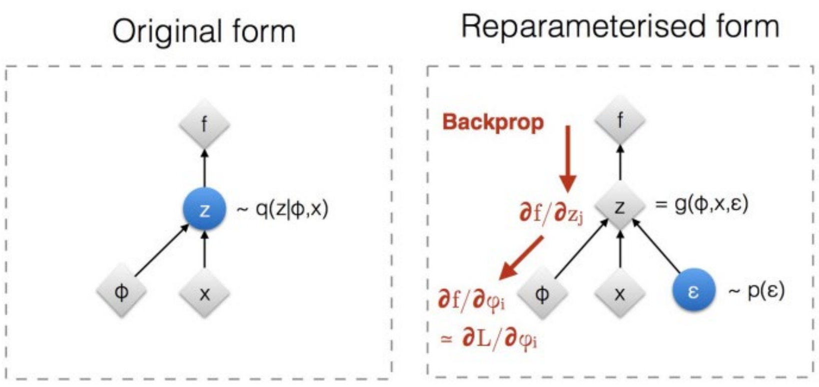
The problem now, though, is that we need to sample at each step from my own distribution (q_ϕ).

This hiddenly states that we have a problem because we want to use this as a loss, hence we need the loss to be differentiable, otherwise we can't compute the gradient.

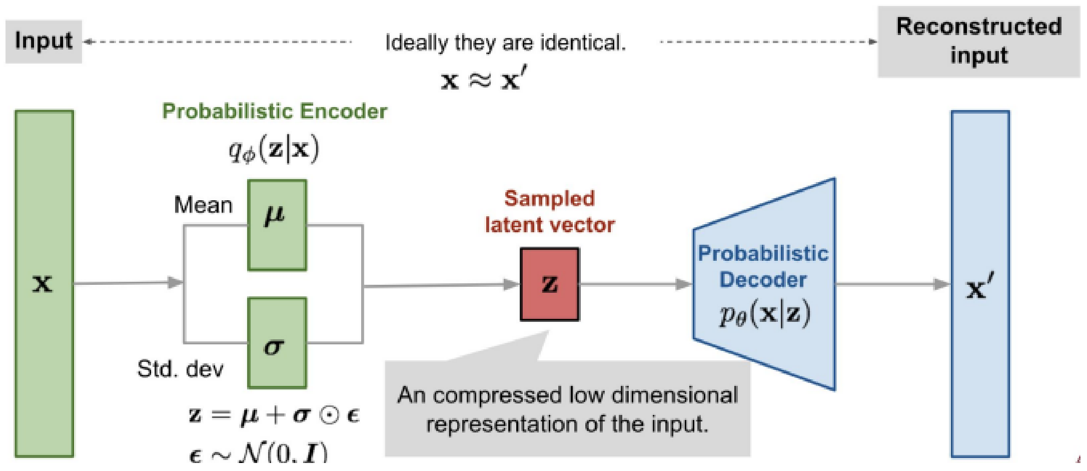
If we want to overcome this, we can use the reparameterization trick in order to make it differentiable.

6.2 Reparameterization Trick

Let's recall that we want to sample $z|x$, but backpropagation can't handle directly sampling from a distribution (like our new q_ϕ function). The reparameterization trick rewrites the sampling process in a way that it combines a deterministic function (in this case our q_ϕ function) with a random noise term (in this case ϵ) which is differentiable. This allows the error signal to flow smoothly back through the network during training.



This leads us to the true architecture of the VAE, which is this one:


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
$\mathbf{x}$
Mean
$\boldsymbol{\mu}$
Std. dev
$\boldsymbol{\sigma}$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

We take X, we project it into two vectors μ, σ (coming from the reparameterization trick) and then we have ϵ which is sampled for each minibatch. This will give us z. From this we'll learn a probabilistic decoder (just a Neural Network).

We could apply this also to Recommender System through the matrix-factorization algorithm, which uses a VAE that takes as input U (users) and V (items) and tries to learn the distribution of all the possible U and V in order to better account for mistakes or errors (for example when a user doesn't like a movie, that cell of user-movie will be filled with an empty cell. We want to learn, through VAE, a distribution which smooths out these errors).