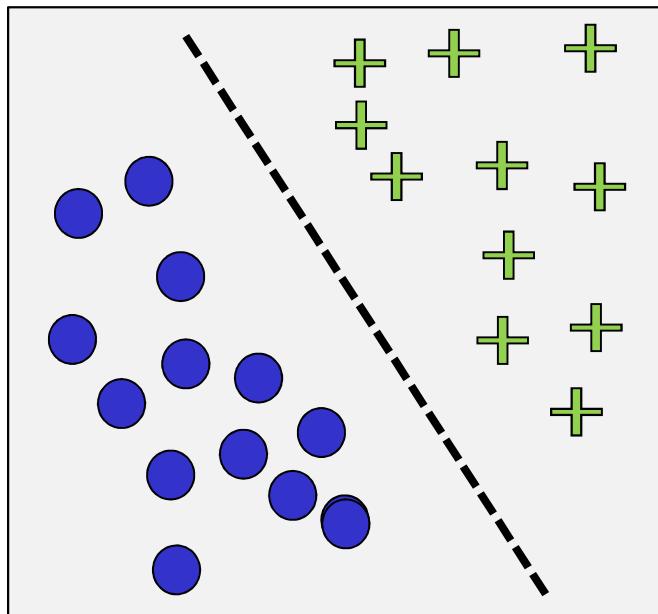


Réseaux de neurones  
IFT 780

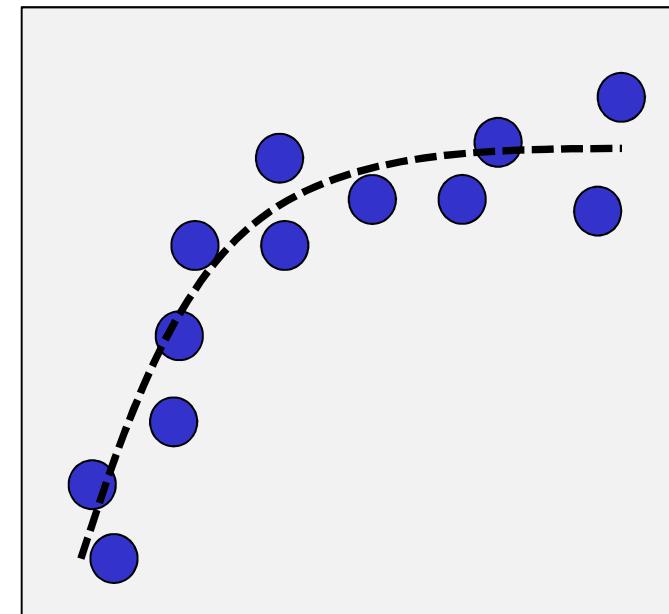
Modèles génératifs  
Par  
Pierre-Marc Jodoin

# Jusqu'à présent : apprentissage supervisé

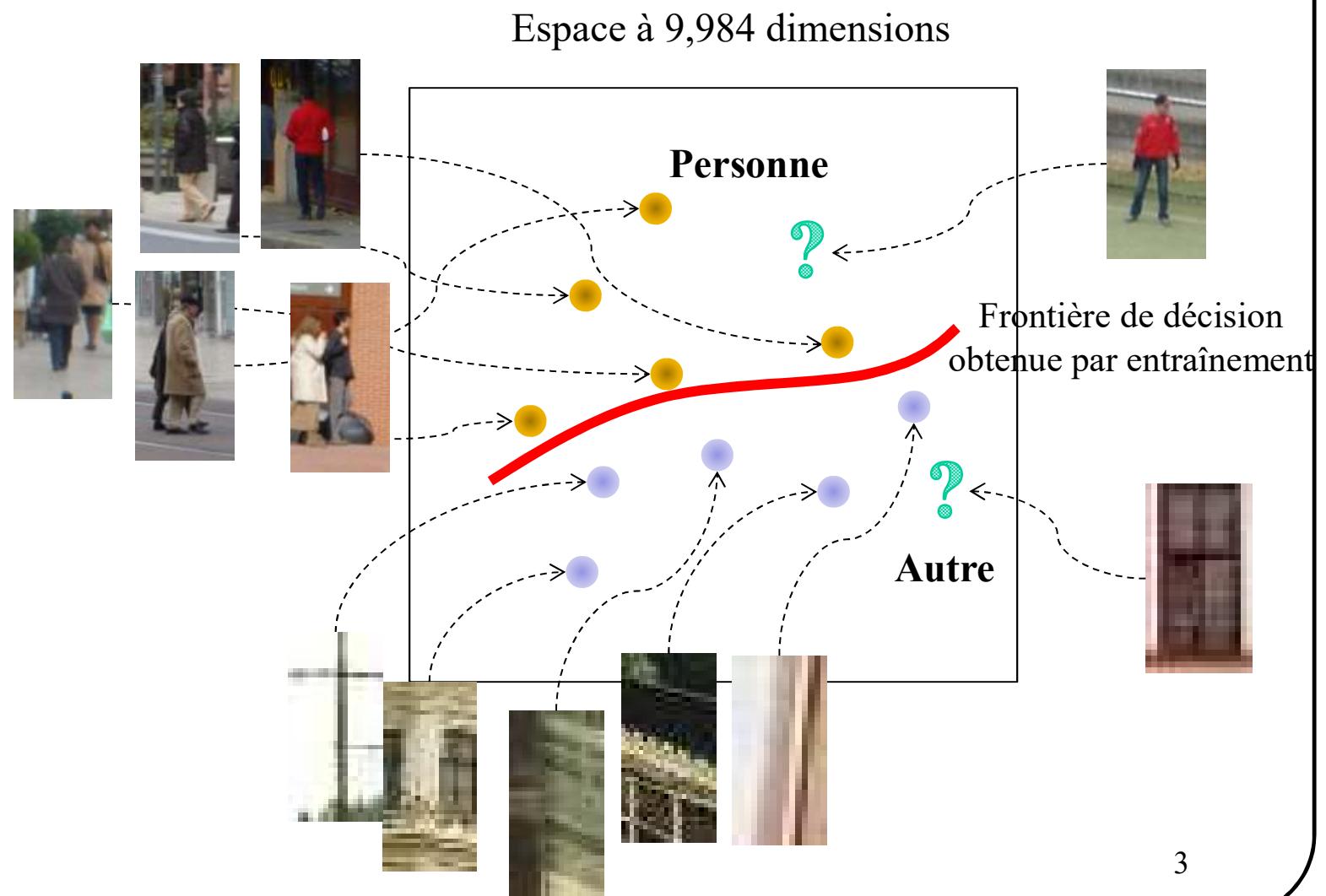
Classification



Régression



## *Inria person dataset*



# Apprentissage supervisé

Deux grandes familles d'applications

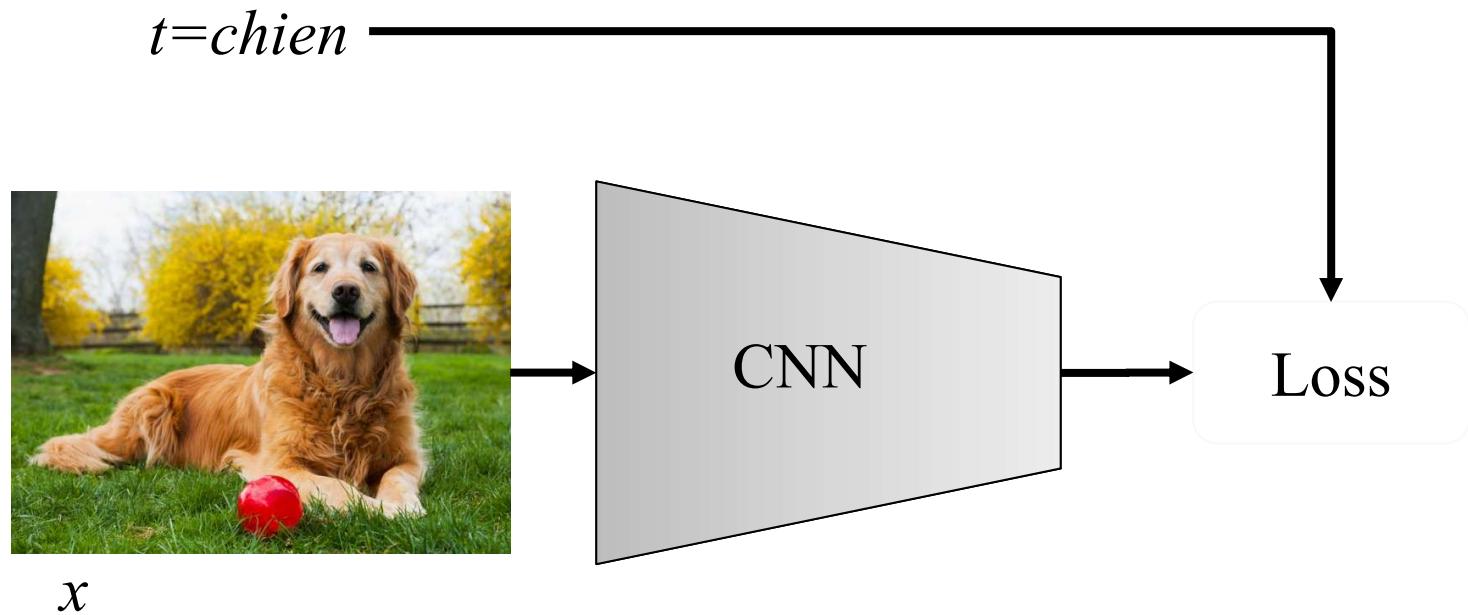
➤ **Classification** : la cible est un indice de classe  $t \in \{1, \dots, K\}$

- Exemple : reconnaissance de caractères
  - ✓  $\vec{x}$  : vecteur des intensités de tous les pixels de l'image
  - ✓  $t$  : identité du caractère

➤ **Régression** : la cible est un nombre réel  $t \in \mathbb{R}$

- Exemple : prédiction de la valeur d'une action à la bourse
  - ✓  $\vec{x}$  : vecteur contenant l'information sur l'activité économique de la journée
  - ✓  $t$  : valeur d'une action à la bourse le lendemain

# Apprentissage supervisé avec CNN



# Supervisé vs non supervisé

**Apprentissage supervisé** : il y a une cible

$$D = \{(\vec{x}_1, t_1), (\vec{x}_2, t_2), \dots, (\vec{x}_N, t_N)\}$$

**Apprentissage non-supervisé** : la cible n'est pas fournie

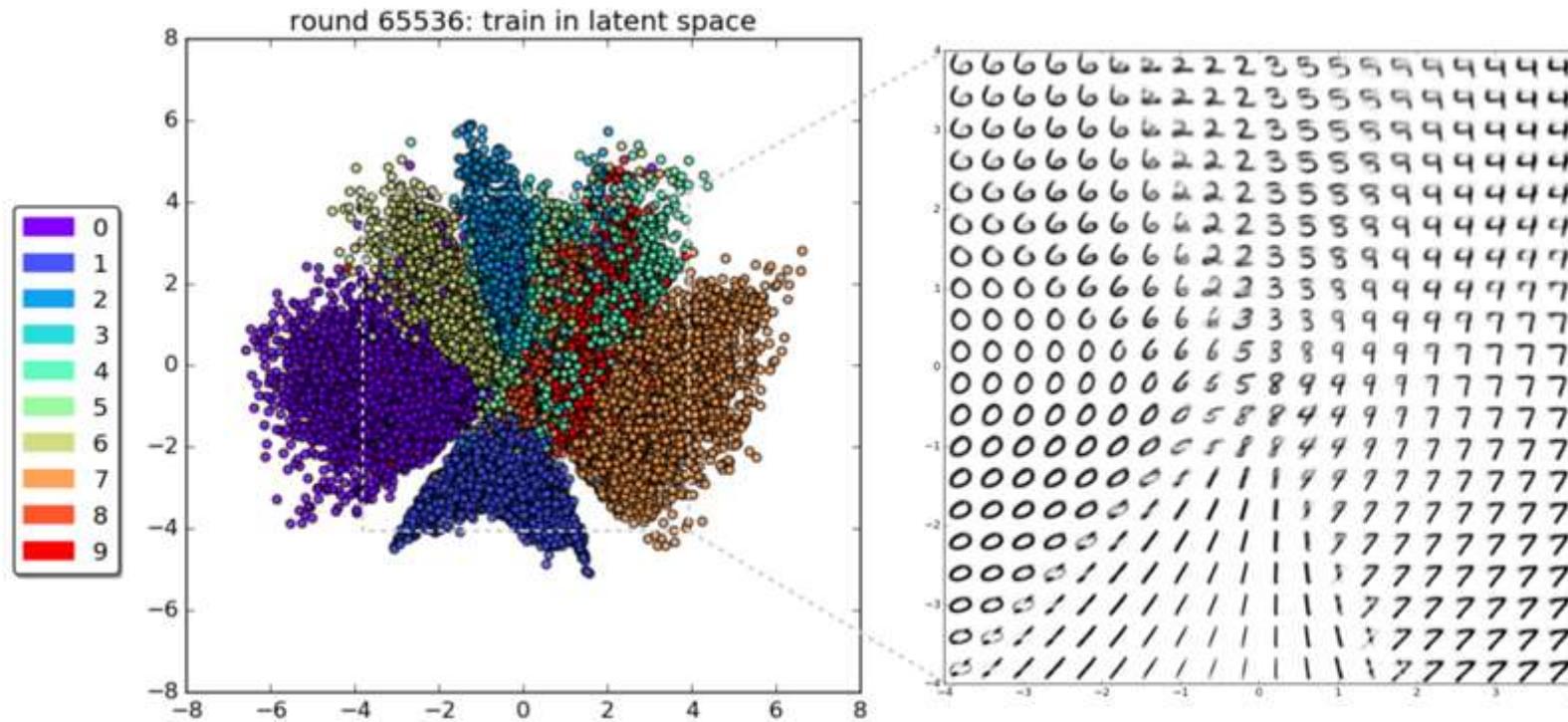
$$D = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$$

# Apprentissage non supervisé

Comprendre la distribution sous-jacente de données **non-étiquetées**

**Applications** : clustering, visualization, comprehension, etc.

**Exemple** : visualization de la distribution des images MNIST

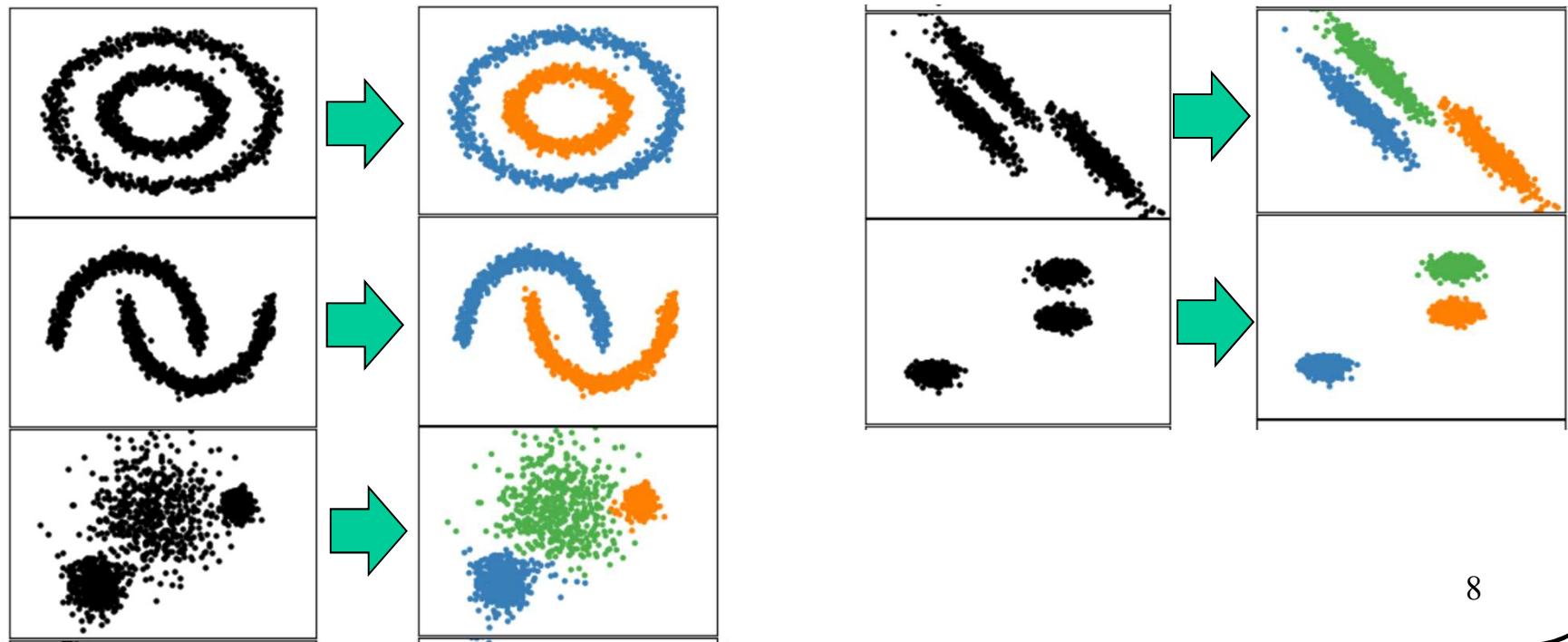


# Apprentissage non supervisé

Souvent, l'apprentissage non-supervisé inclut un (ou des) **variables latentes**.

**Variable latente:** variable aléatoire non observée mais sous-jacente à la distribution des données

**Ex:** clustering = retrouver la variable latente “cluster”



# Pourquoi une variable latente?

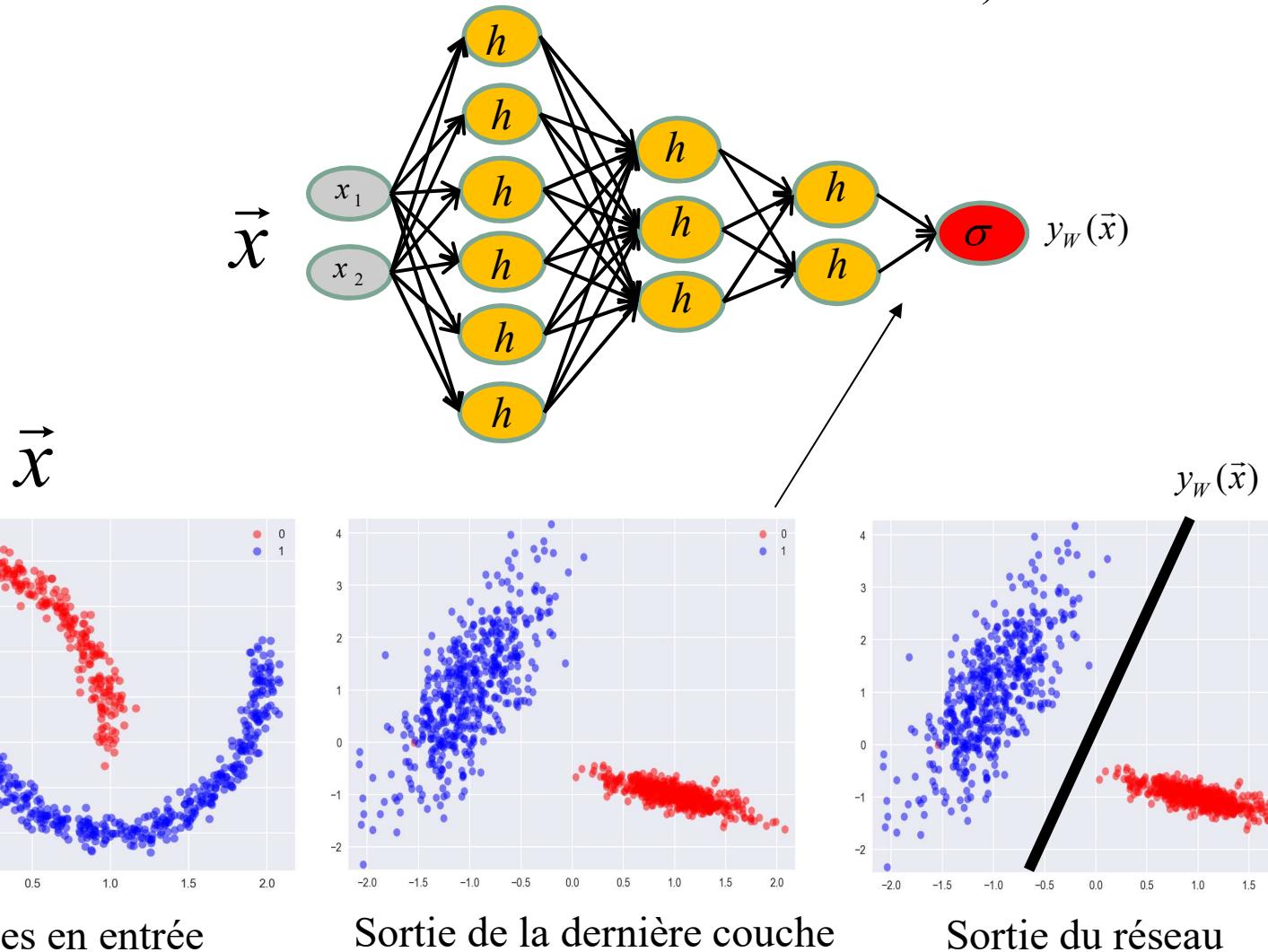
Plus facile de représenter  $p(\vec{x}, y)$ ,  $p(\vec{x} | y)$ ,  $p(y)$   
que  $p(\vec{x})$

**Plus d'info au tableau.**

En apprentissage non-supervisé  
nous nous appuierons sur  
**2 propriétés** des réseaux de neurones

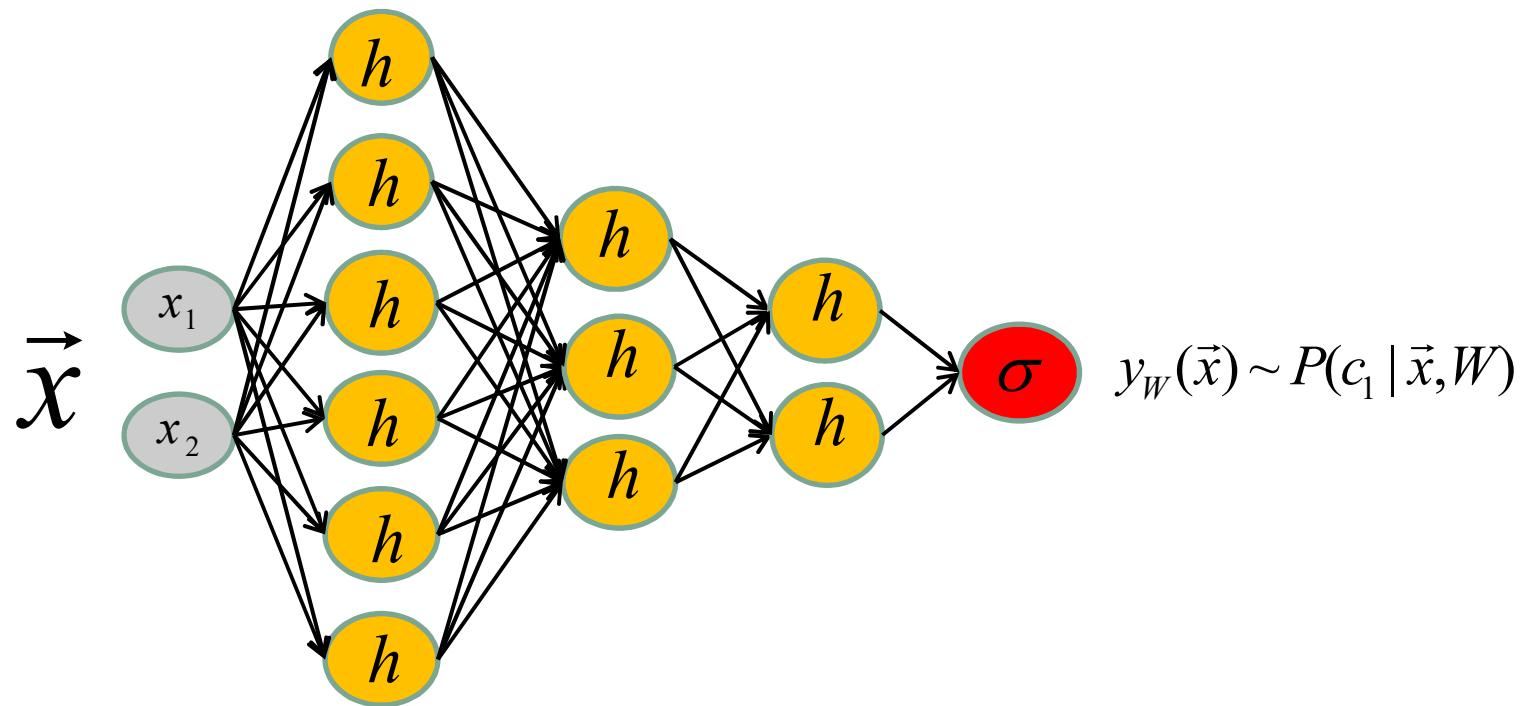
# Propriété 1

Les réseaux de neurones sont d'excellentes machines pour projeter des données brutes vers un **espace dimensionnel plus faible** dont les propriétés dépendent de la *loss* (ici **espace linéaire car la sortie du réseau est un classifieur linéaire**).



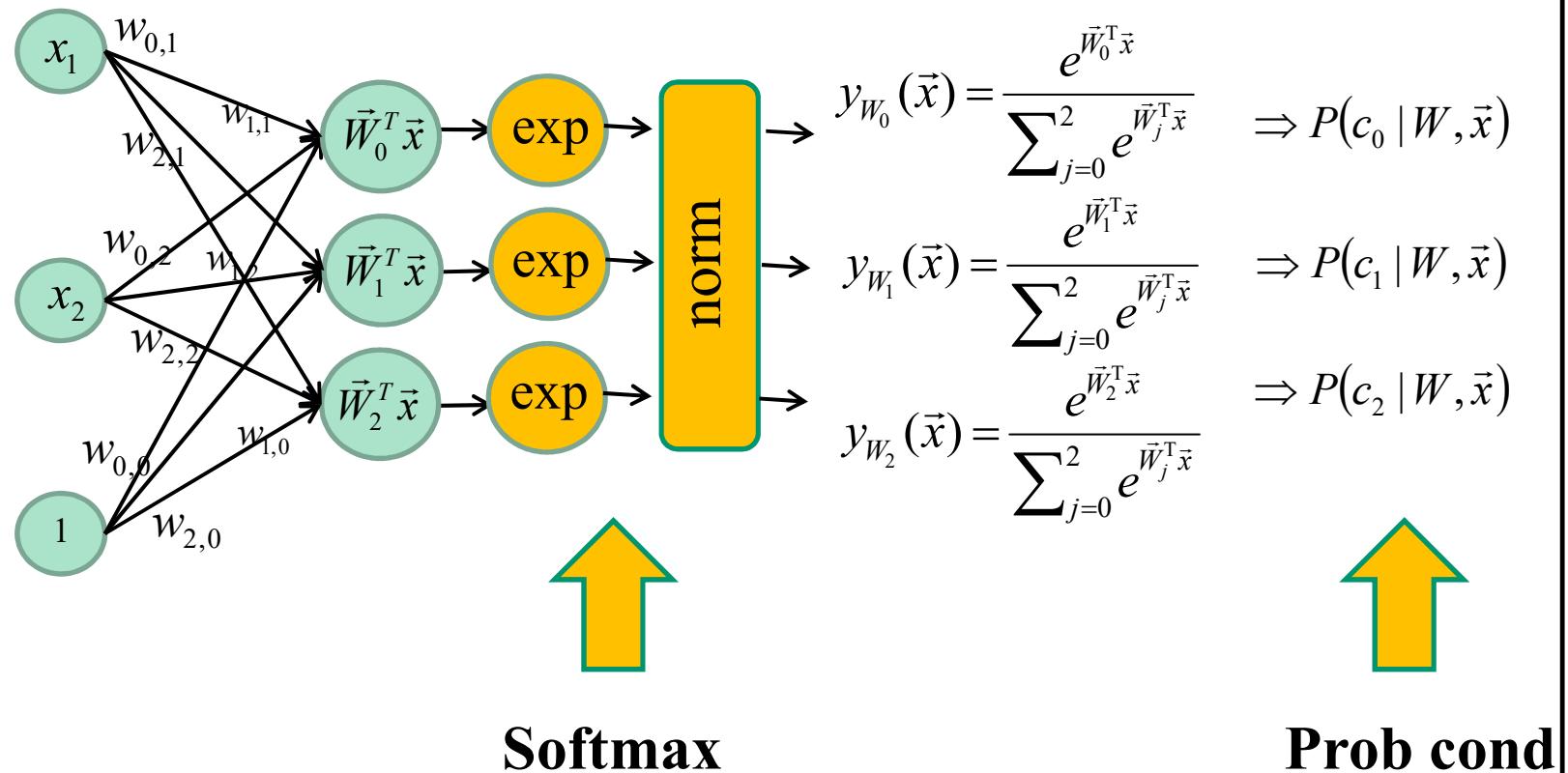
# Propriété 2

Les réseaux de neurones sont d'excellentes machines pour estimer des **probabilités conditionnelles**.



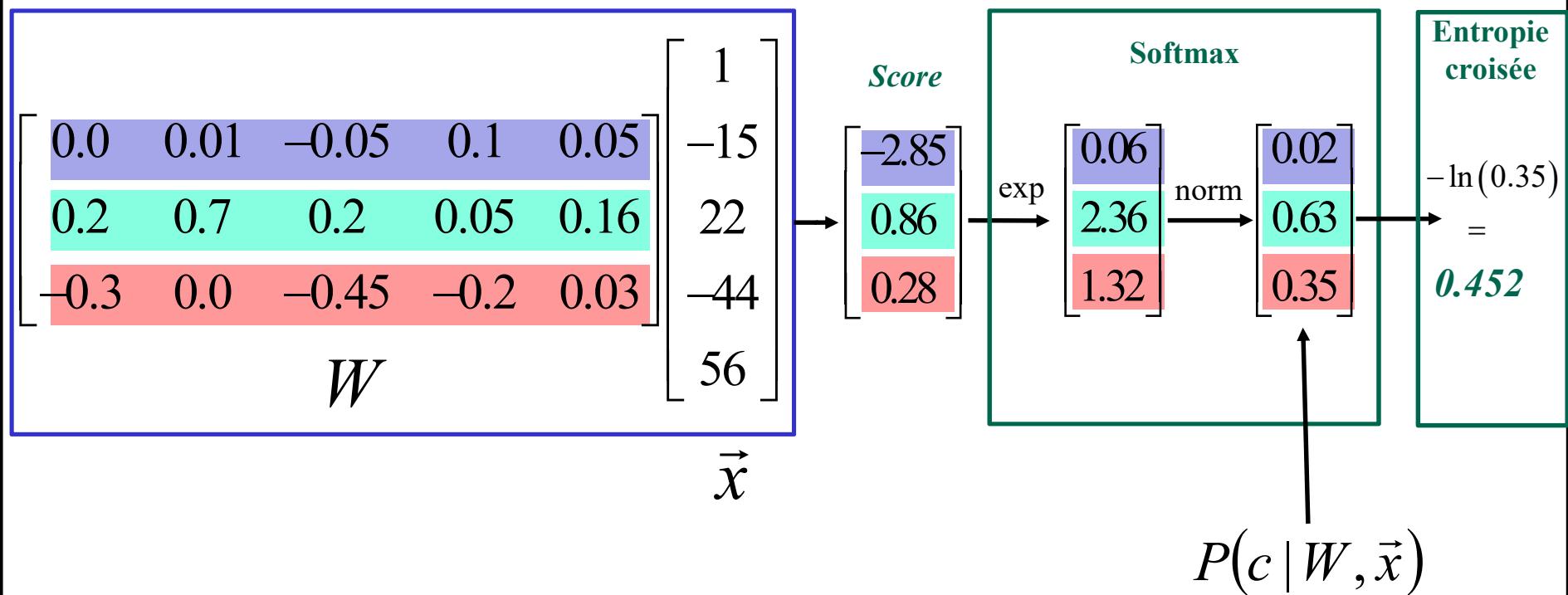
# Propriété 2

Les réseaux de neurones sont d'excellentes machines pour estimer des **probabilités conditionnelles**.



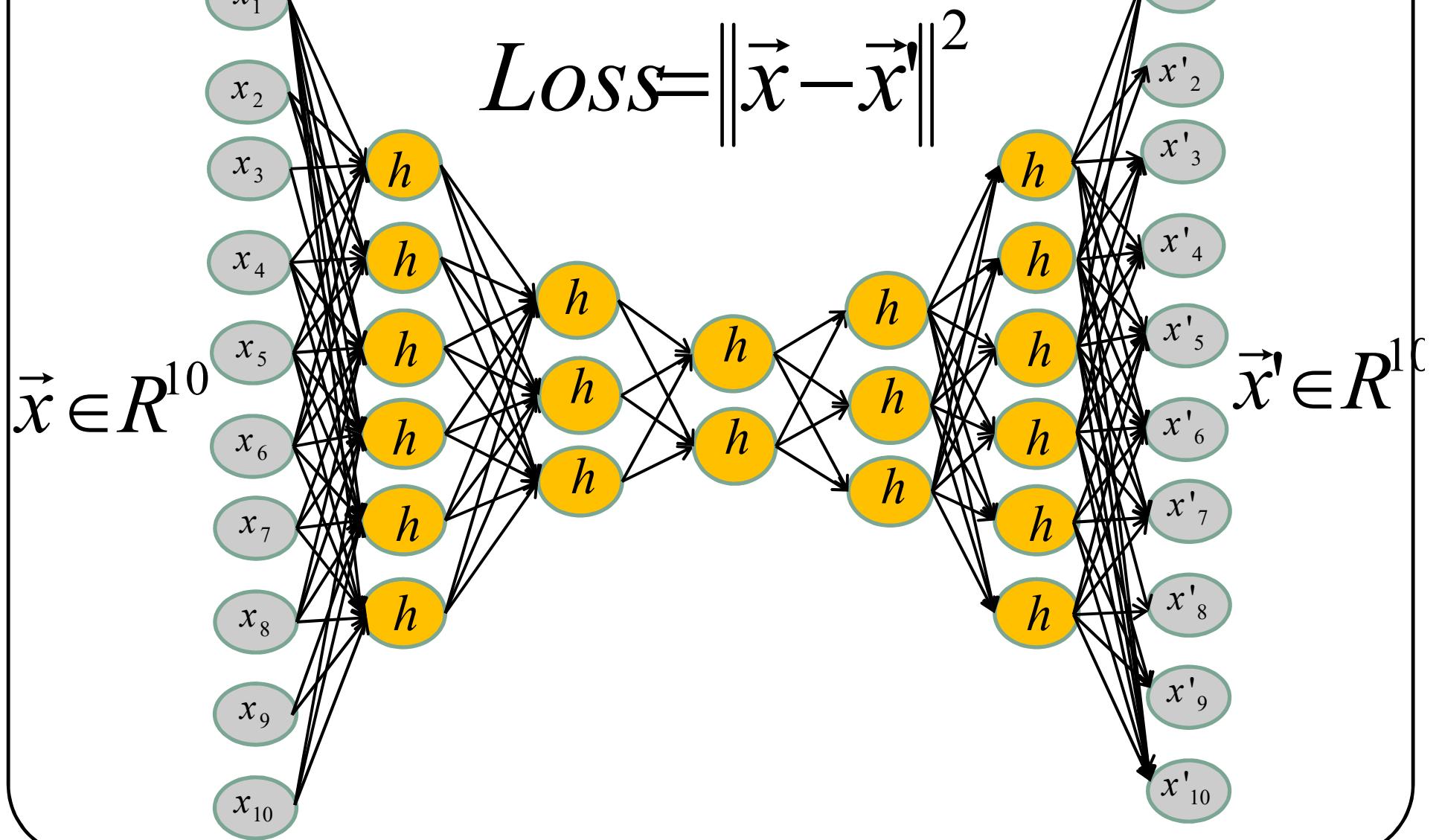
## Rappel

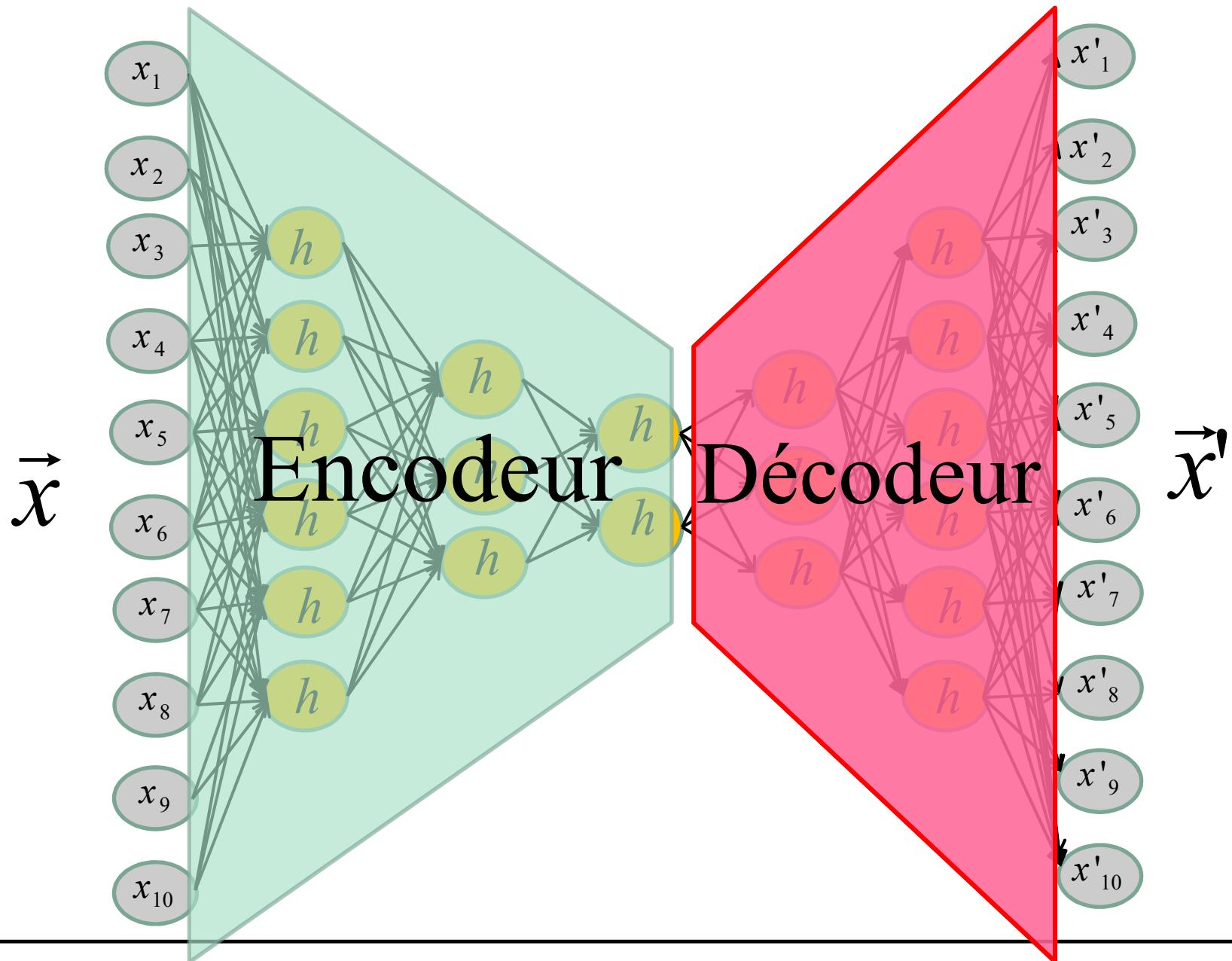
$$\vec{x} = \begin{bmatrix} -15 \\ 22 \\ -44 \\ 56 \end{bmatrix}, t = 2$$

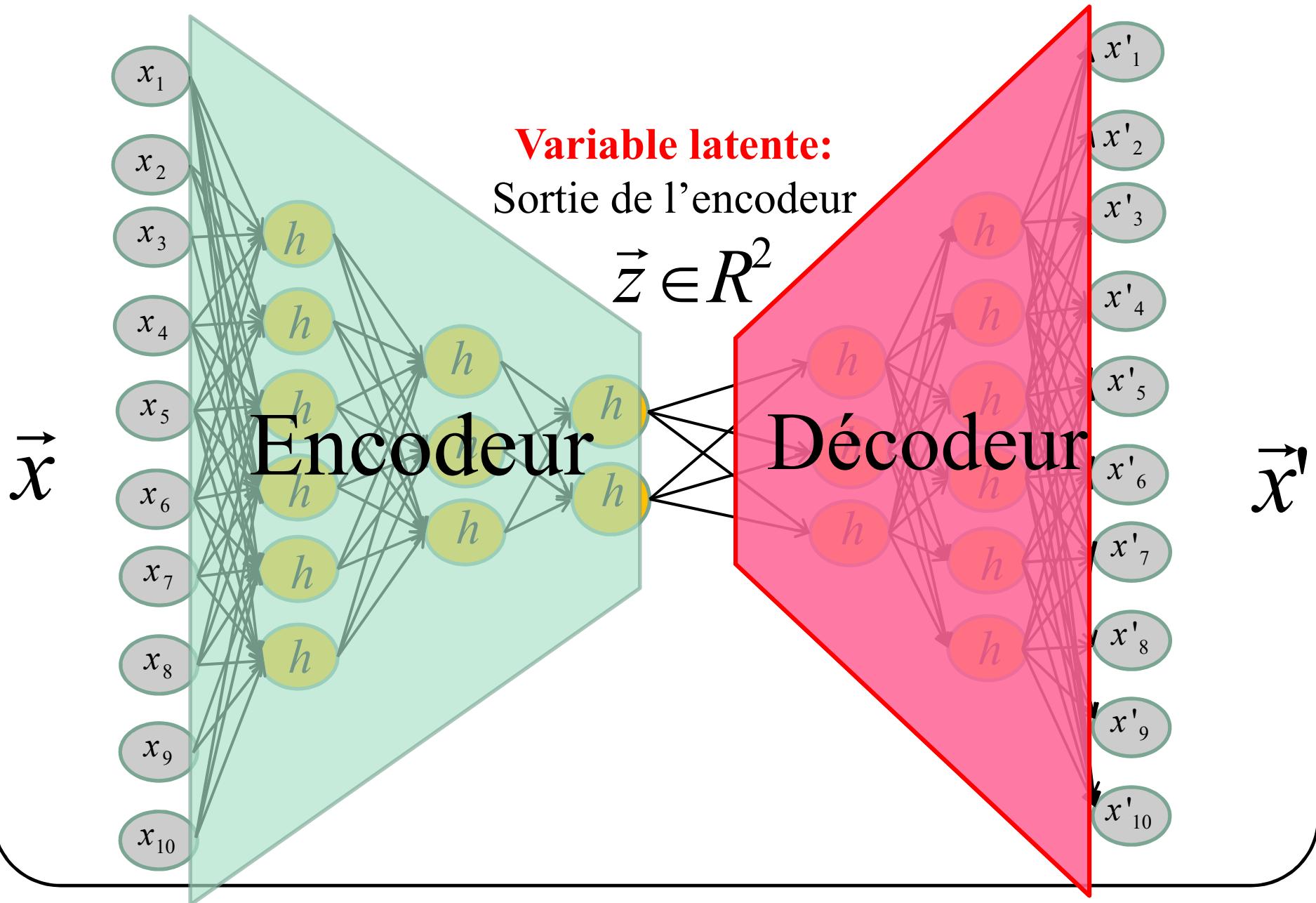


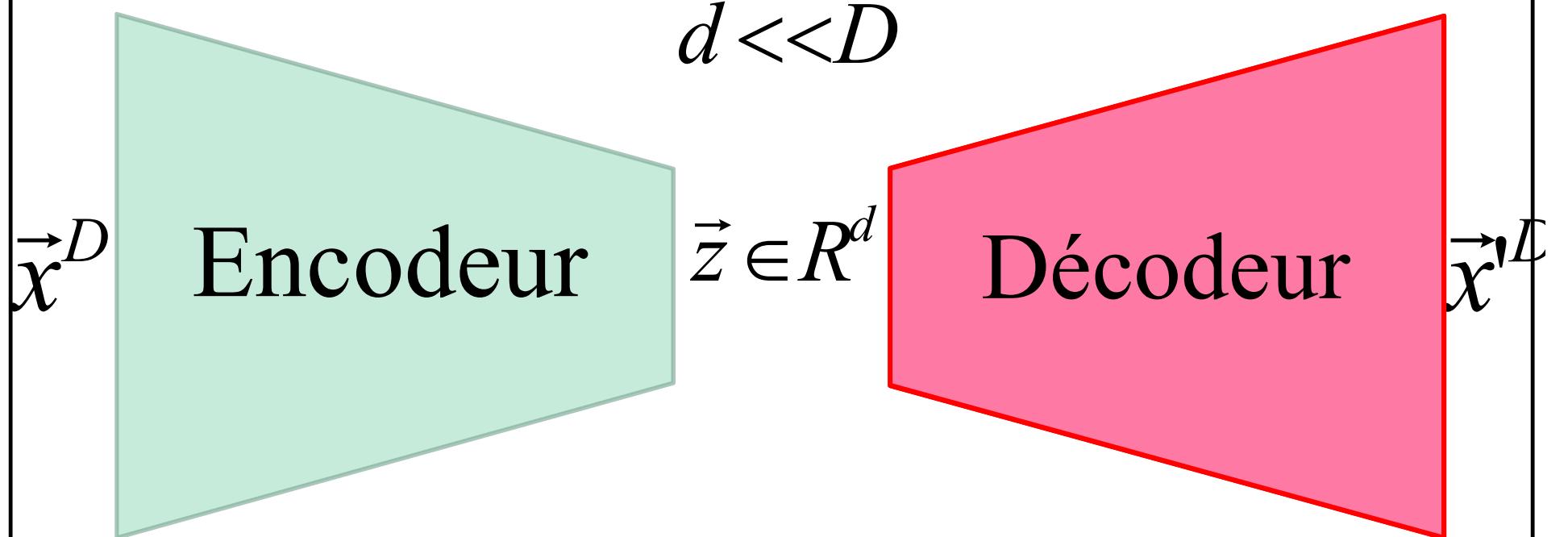
Comment utiliser un réseau de neurones pour apprendre  
la **configuration sous-jacente** de données non étiquettés?





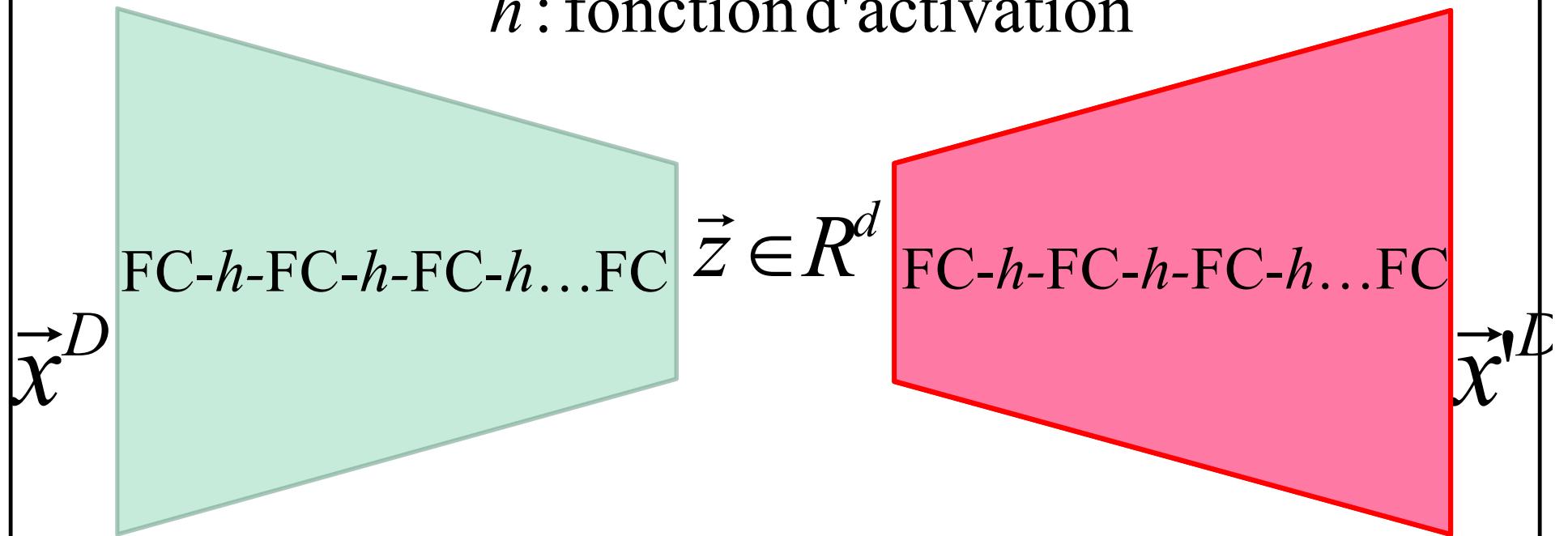




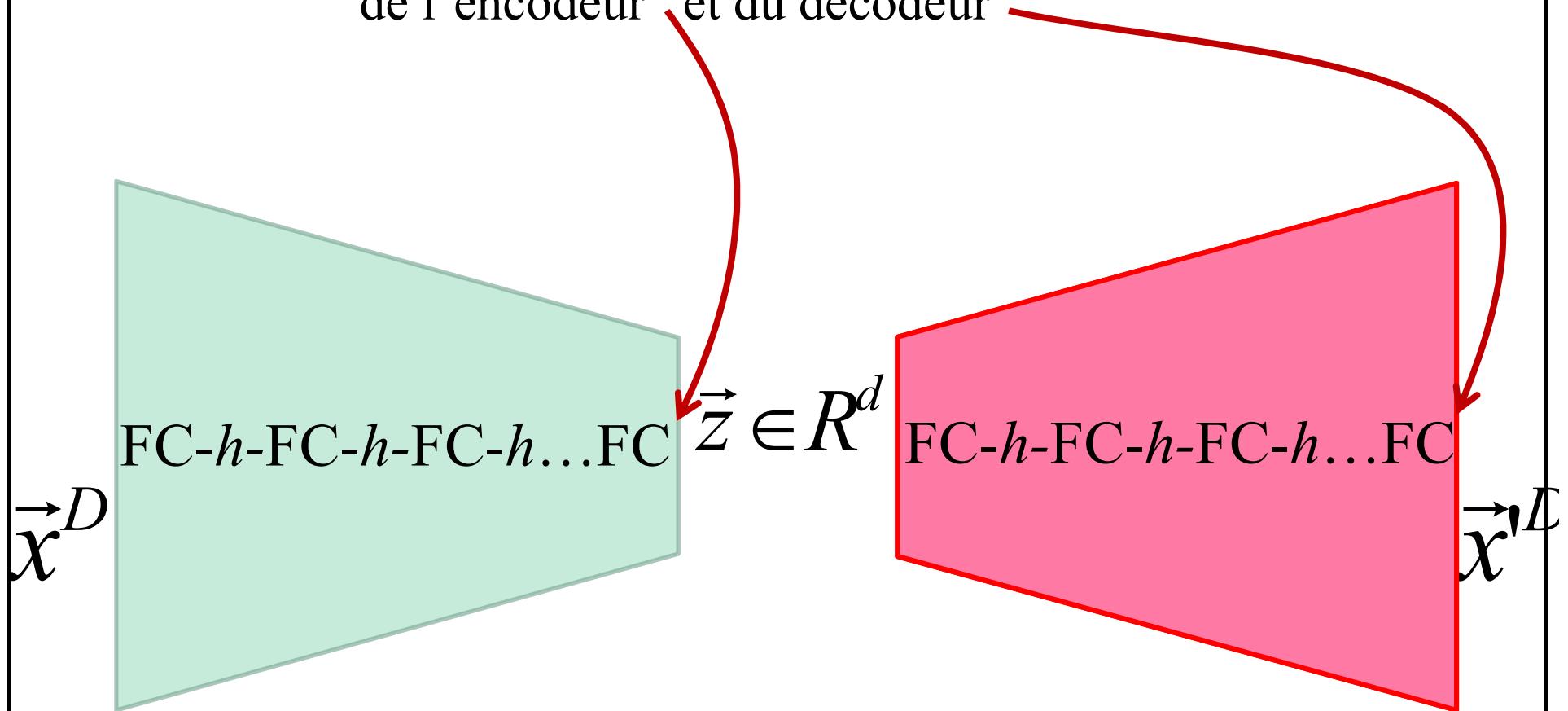


# Couches pleinement connectées

$h$ : fonction d'activation



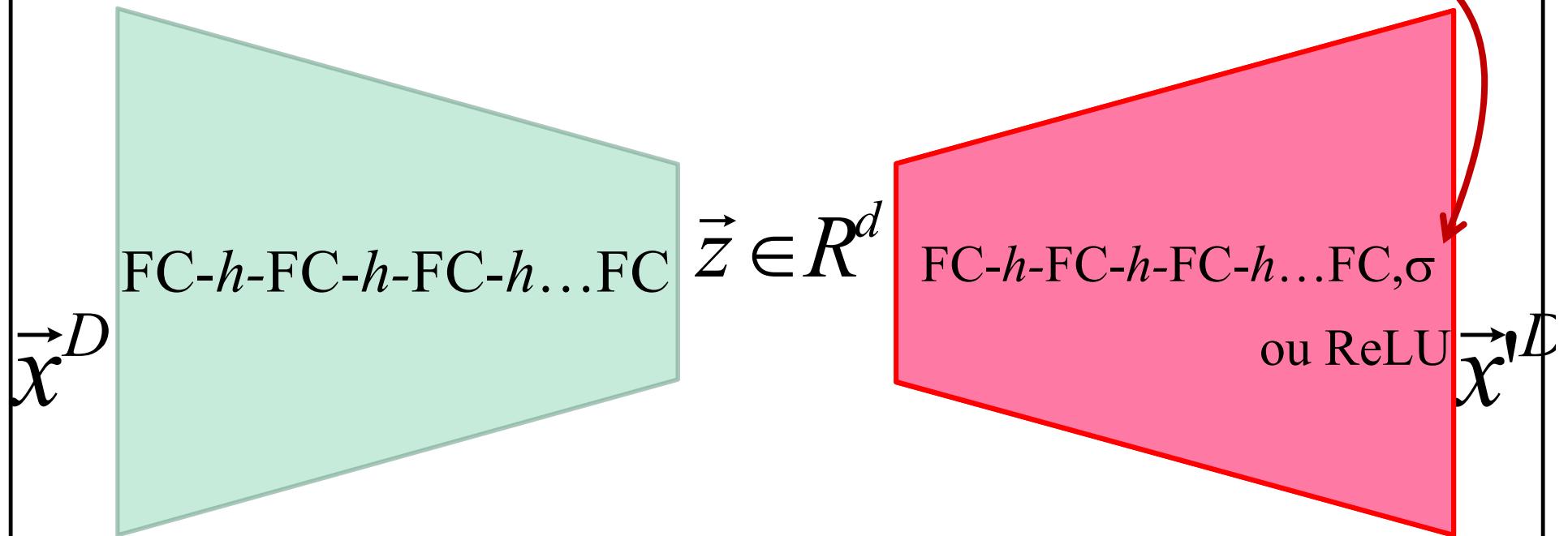
En général...  
pas de fonction d'activation à la sortie  
de l'encodeur et du décodeur



Selon vous, **pourquoi?**

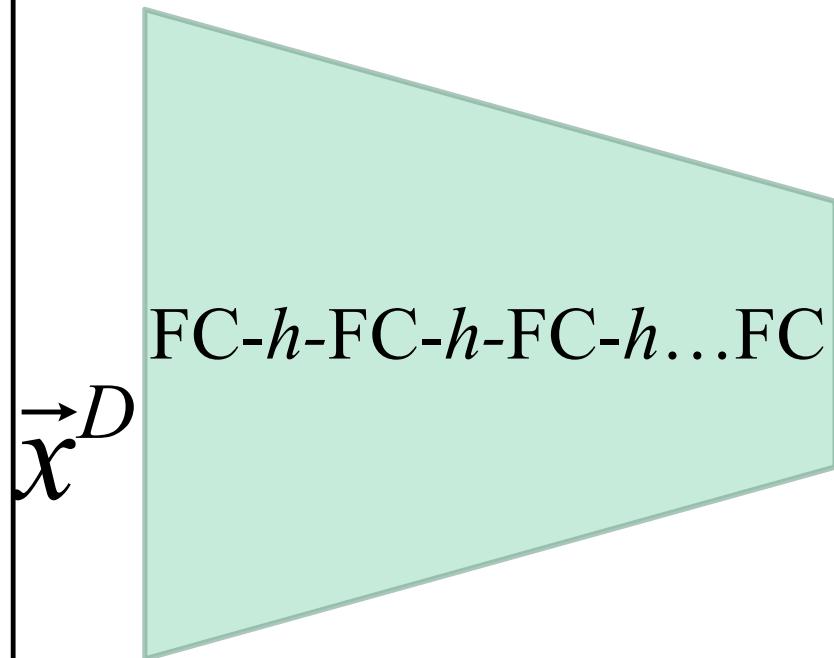
Parfois sigmoïde en sortie lorsque les pixels ont des niveaux de gris entre 0 et 1.

ou ReLU lorsque les niveaux de gris peuvent être élevés et jamais négatifs



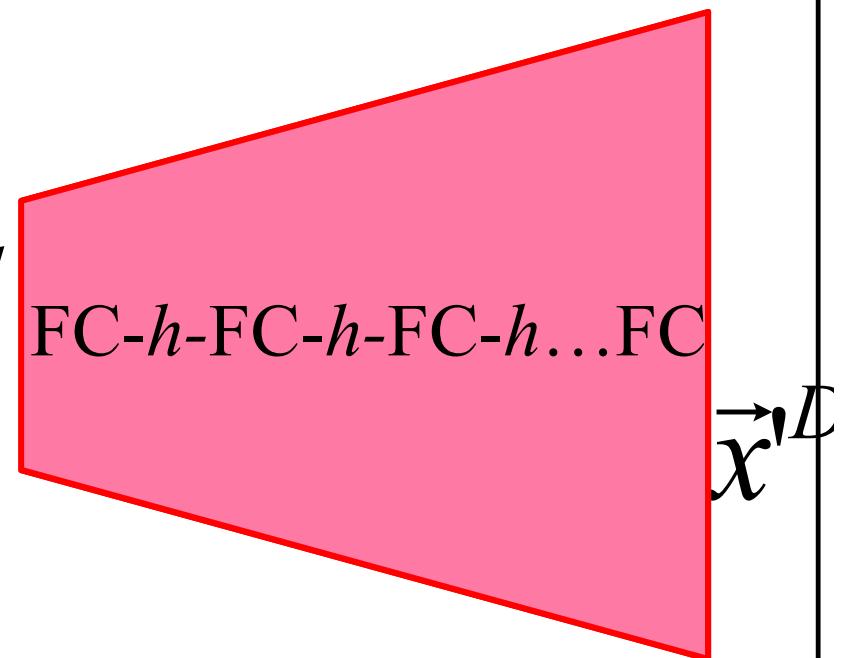
Le nombre de neurones

**Décroit ou se maintient**  
d'une couche à l'autre



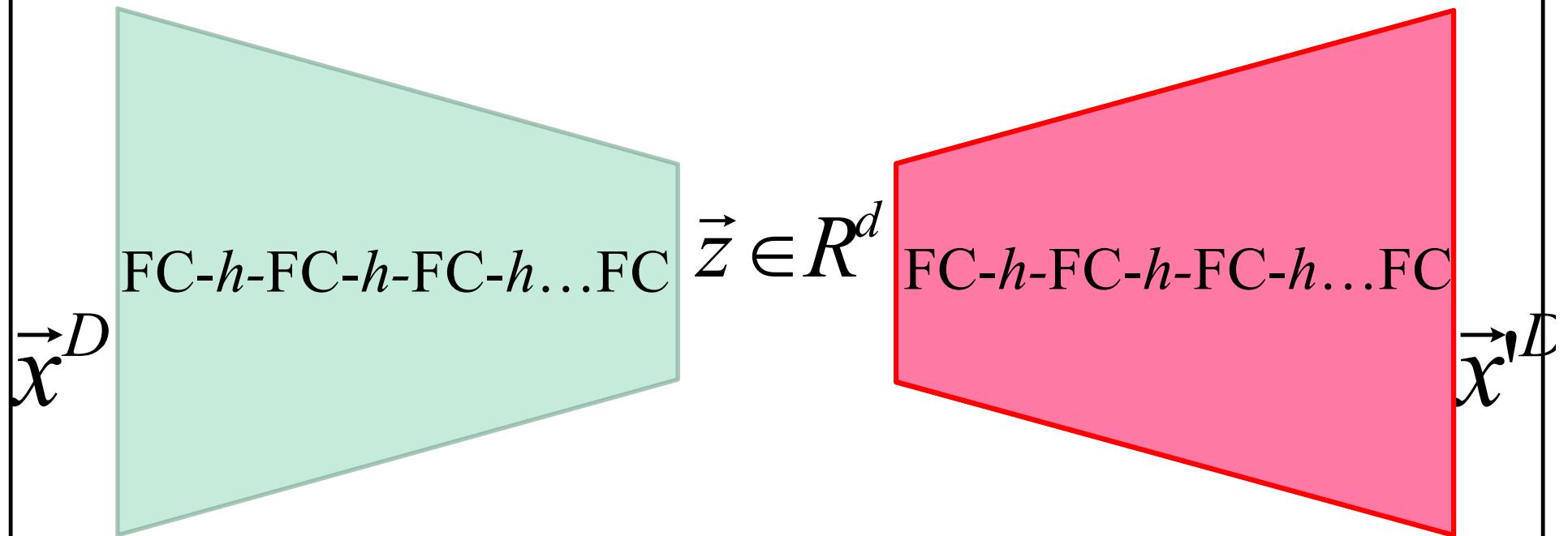
Le nombre de neurones

**Augmente ou se maintient**  
d'une couche à l'autre



Très souvent...

La structure de l'encodeur est le dual de celle du décodeur



# Autoencodeur jouet de MNIST

```
class autoencoder(nn.Module):

    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128), nn.ReLU(True),
            nn.Linear(128, 64), nn.ReLU(True),
            nn.Linear(64, 12), nn.ReLU(True),
            nn.Linear(12, 2))

        self.decoder = nn.Sequential(
            nn.Linear(2, 12), nn.ReLU(True),
            nn.Linear(12, 64), nn.ReLU(True),
            nn.Linear(64, 128), nn.ReLU(True),
            nn.Linear(128, 28 * 28))

    def forward(self, x):
        z = self.encoder(x)
        x_prime = self.decoder(z)
        return x_prime
```

Espace latent 2D



# Autoencodeur jouet de MNIST

symétrie

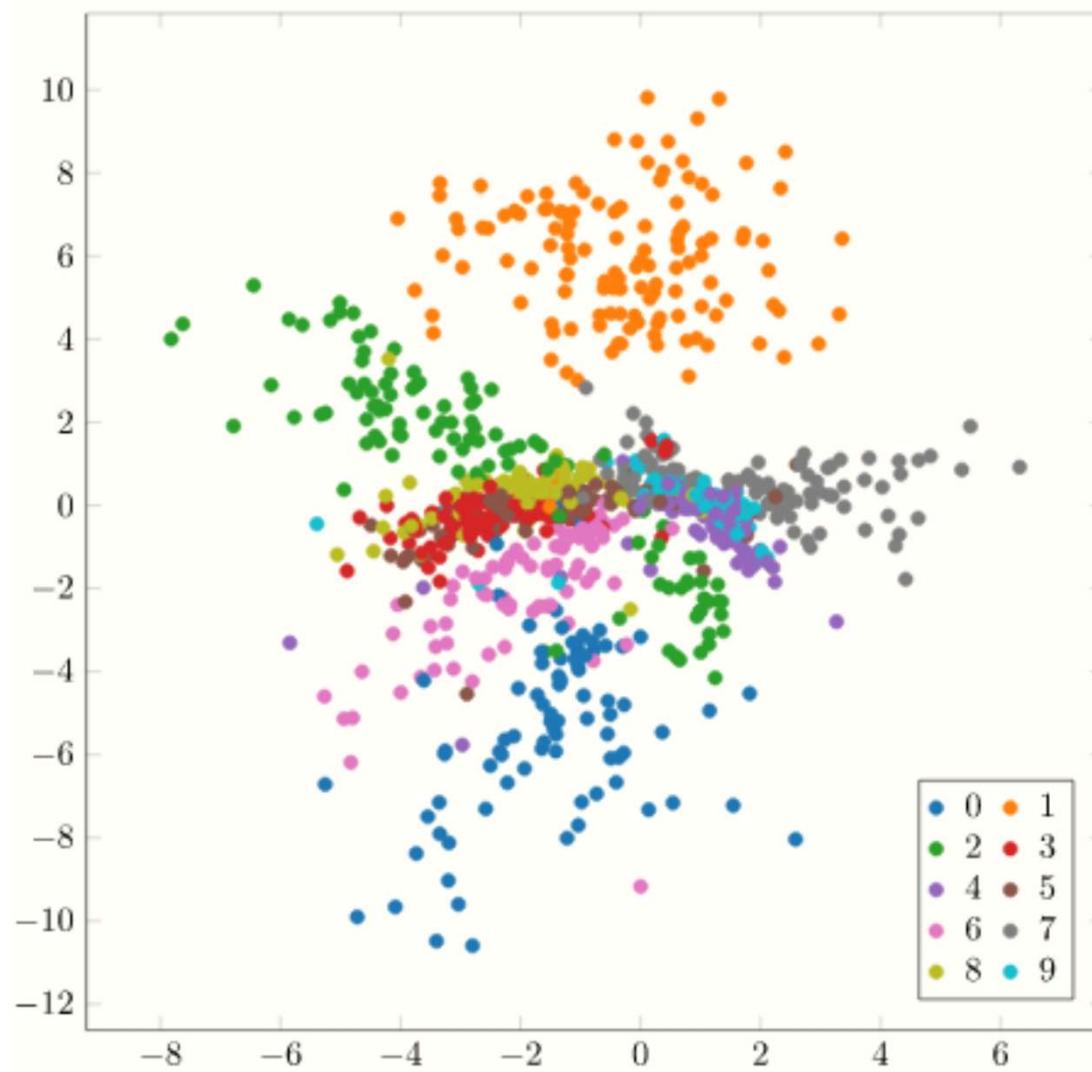
```
class autoencoder(nn.Module):

    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128), nn.ReLU(True),
            nn.Linear(128, 64), nn.ReLU(True),
            nn.Linear(64, 12), nn.ReLU(True),
            nn.Linear(12, 2))

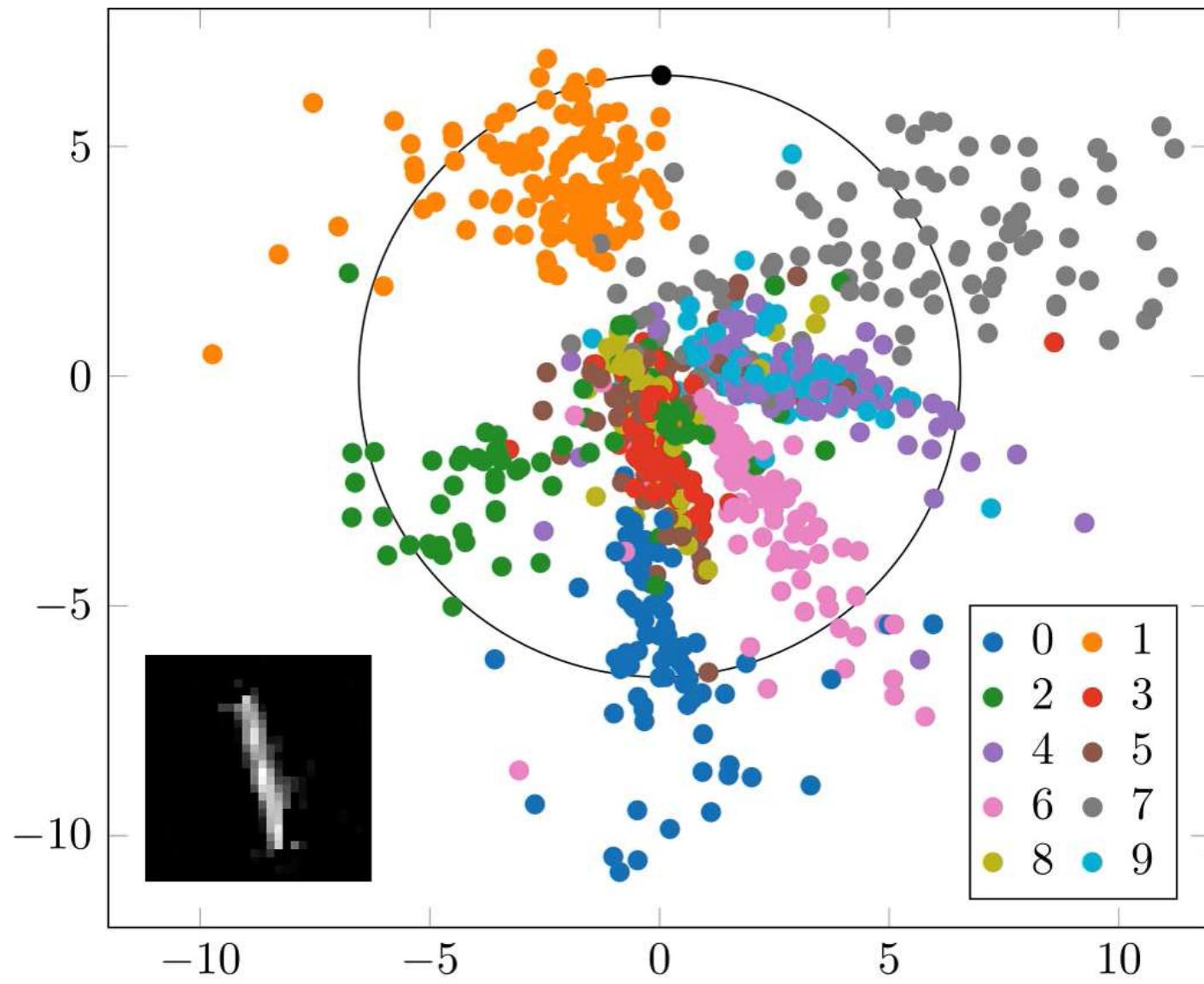
        self.decoder = nn.Sequential(
            nn.Linear(2, 12), nn.ReLU(True),
            nn.Linear(12, 64), nn.ReLU(True),
            nn.Linear(64, 128), nn.ReLU(True),
            nn.Linear(128, 28 * 28))

    def forward(self, x):
        z = self.encoder(x)
        x_prime = self.decoder(z)
        return x_prime
```

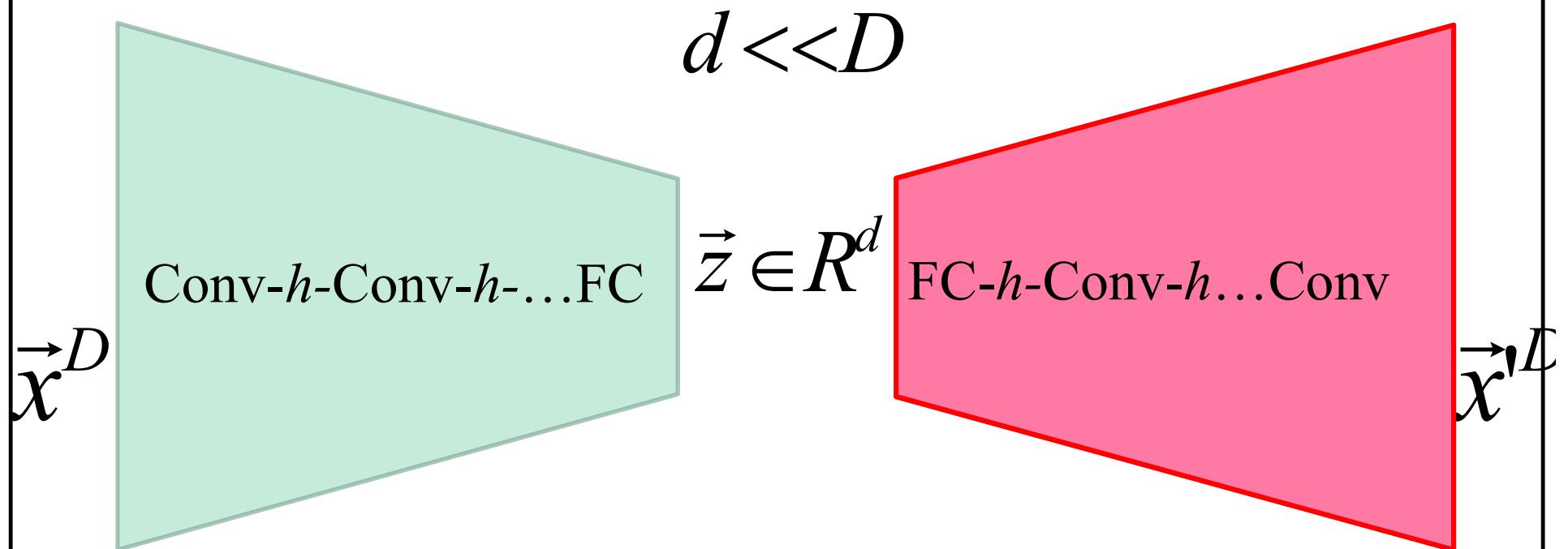
Visualisation de l'espace latent pour 1000 images MNIST  
chaque image correspond à 1 point 2D



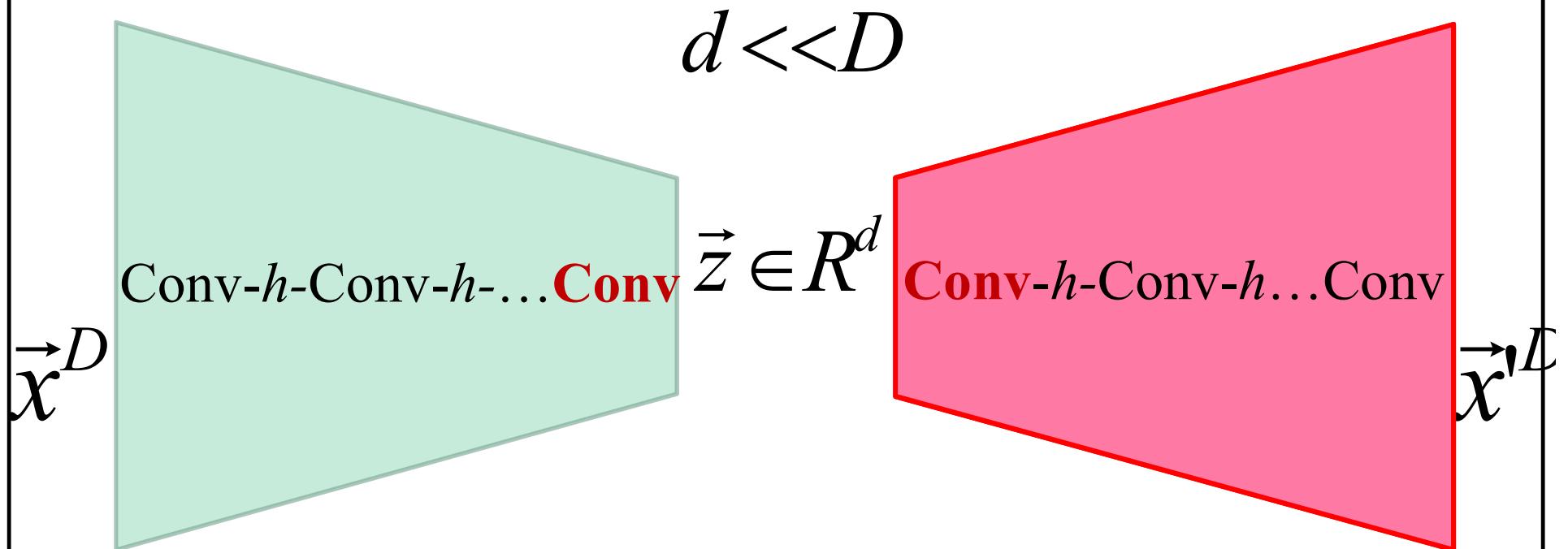
## Générer des images avec le décodeur.



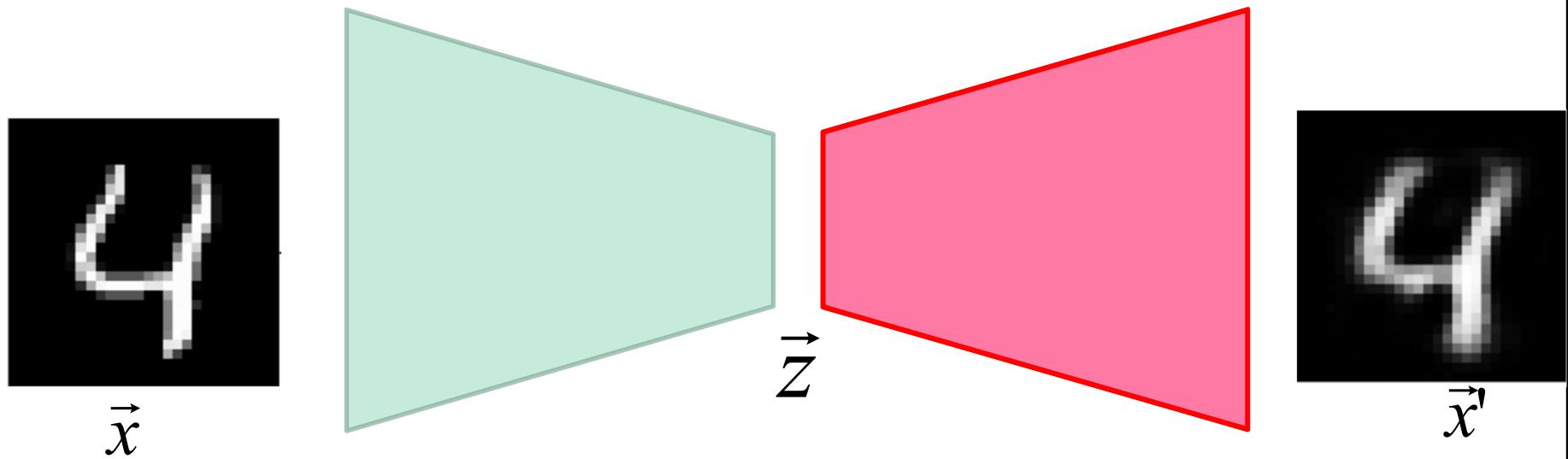
# Couches convolutives



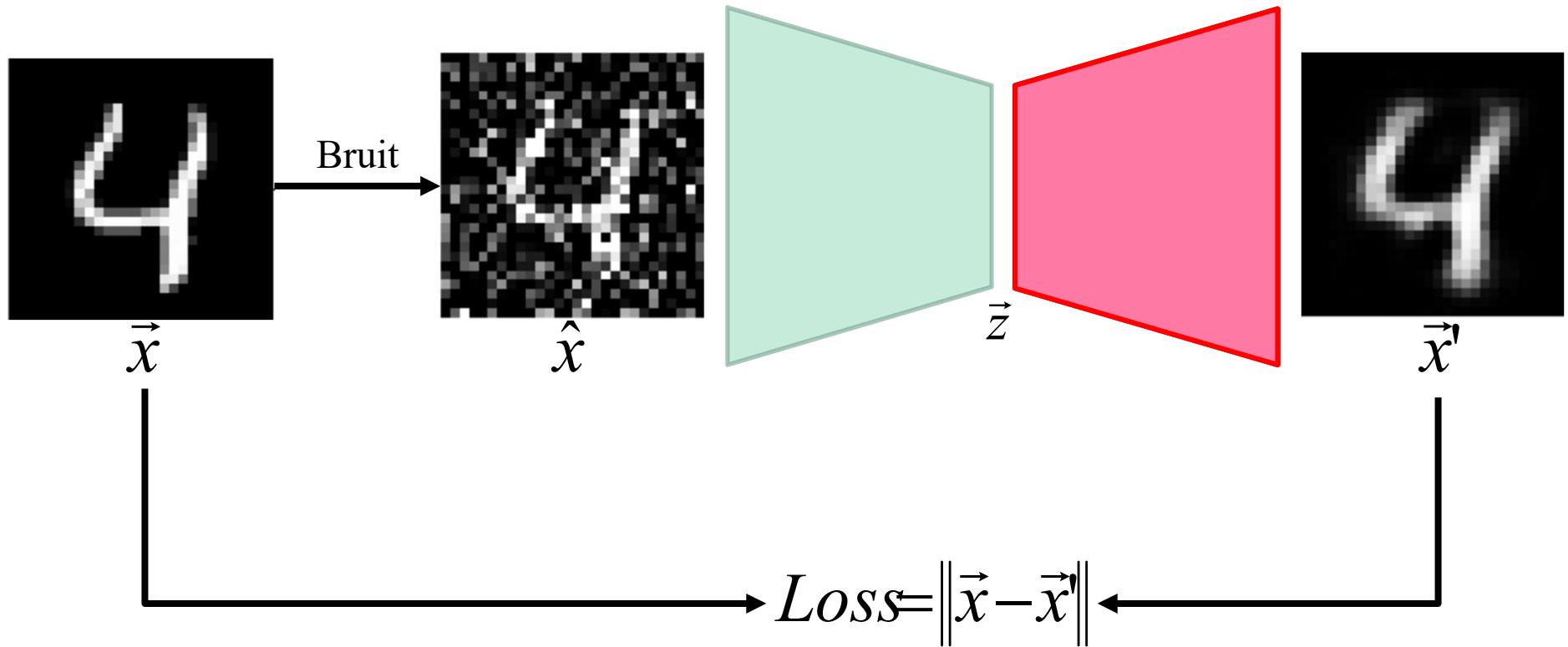
# Autoencodeur pleinement convolutif



## Autoencodeur de base

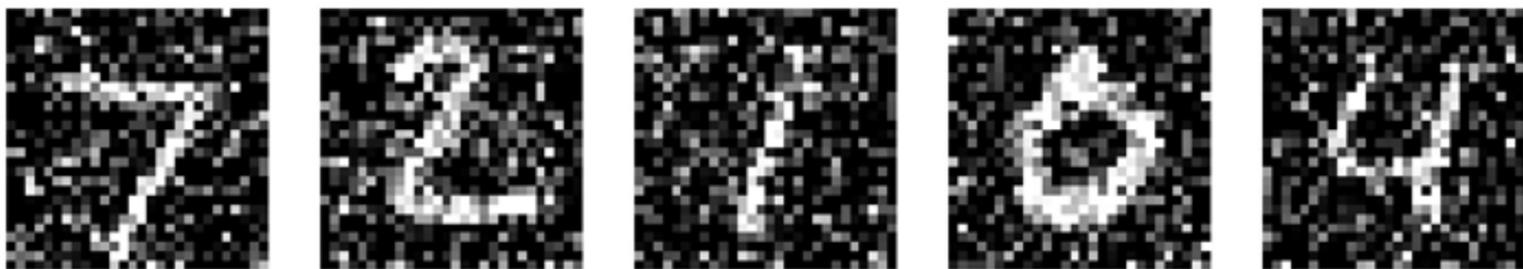


## Autoencodeur pour débuitage



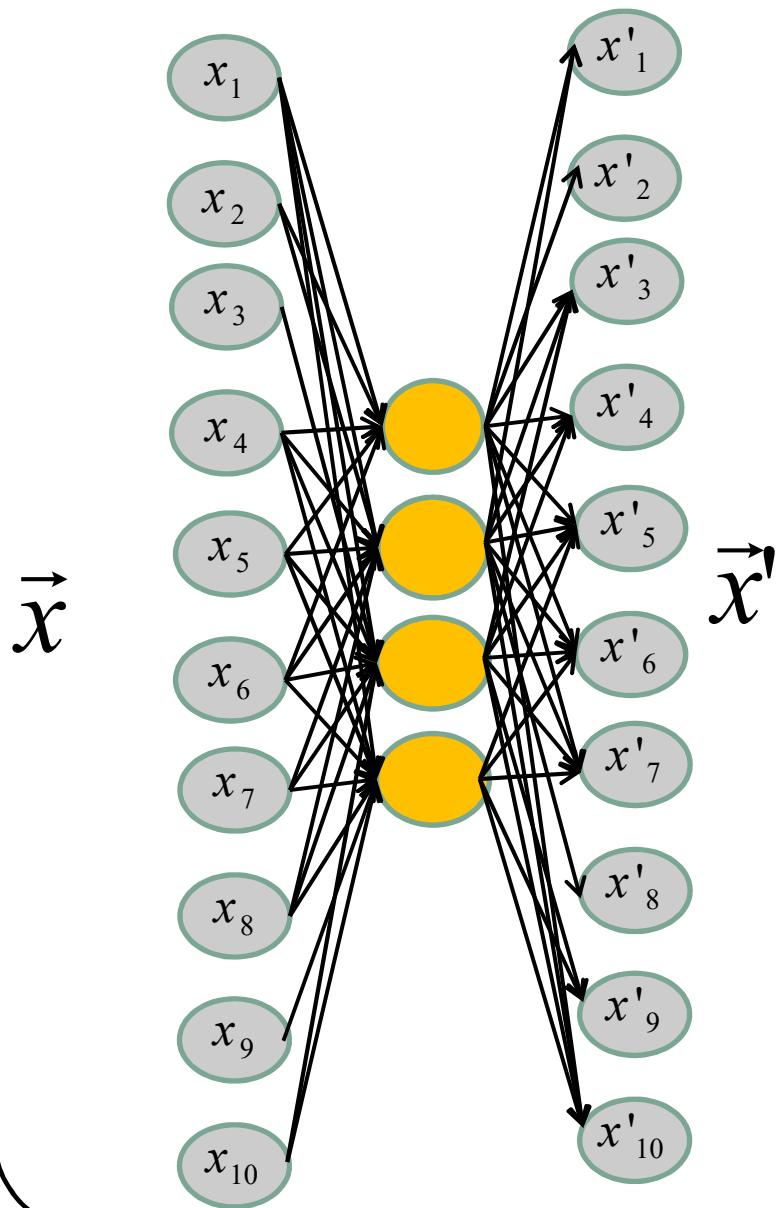
# Une fois entraîné...

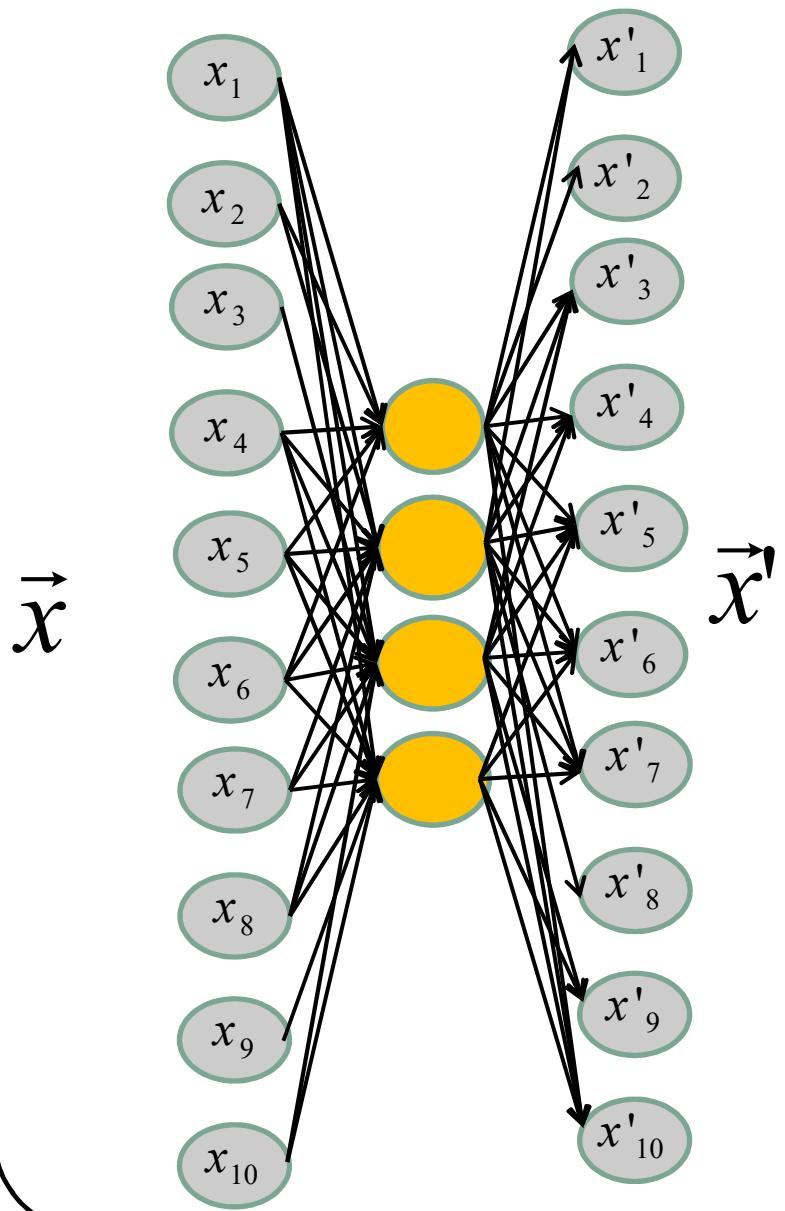
Signal bruité en entrée



Signal reconstruit et débruité

Autoencodeur à **une couche**  
et **sans fonction d'activation**



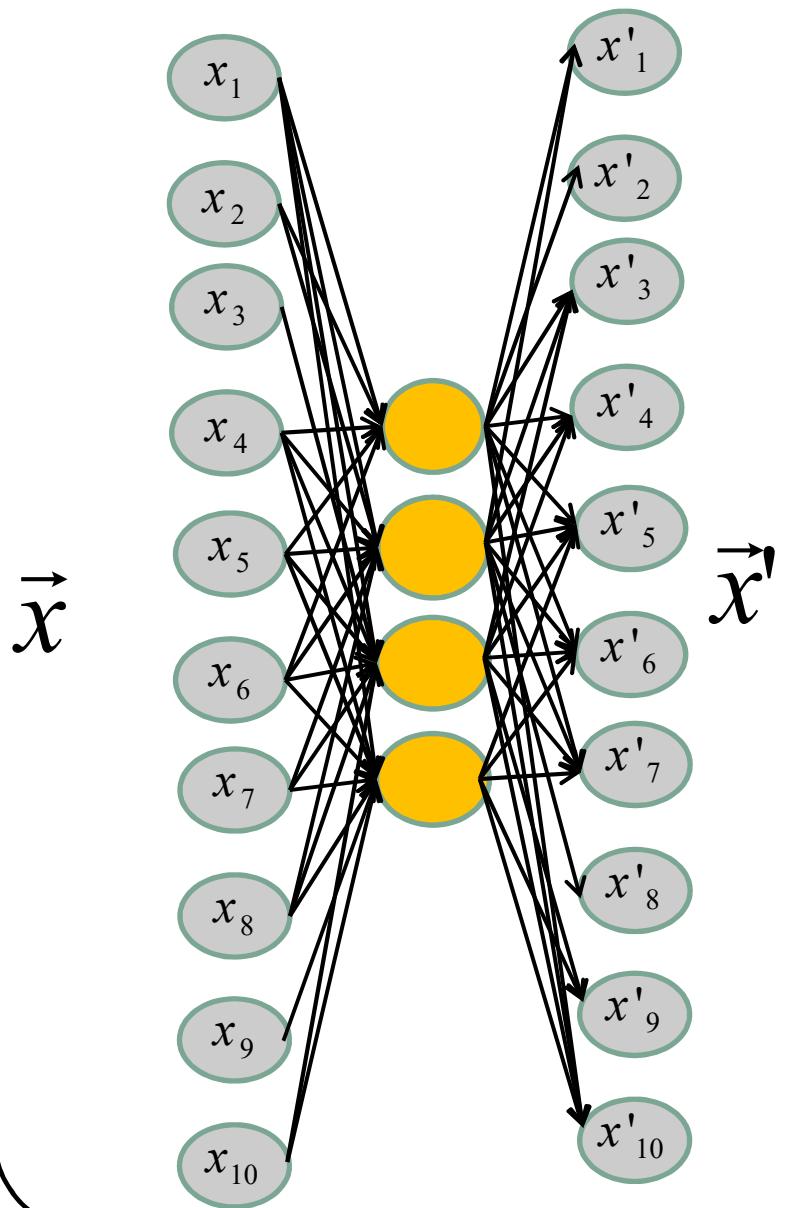


$$W_1 \in R^{4 \times 10}$$

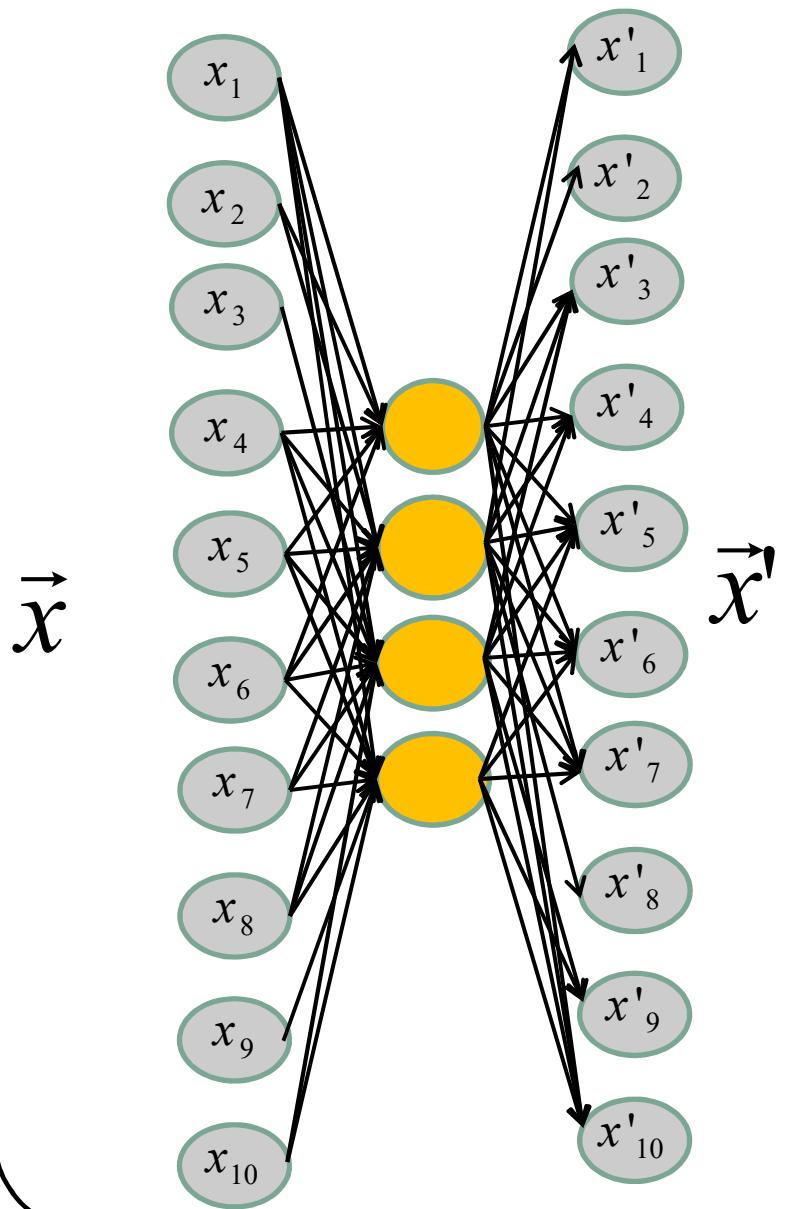
$$W_2 \in R^{10 \times 4}$$

$$\vec{z} = W_1 \vec{x}$$

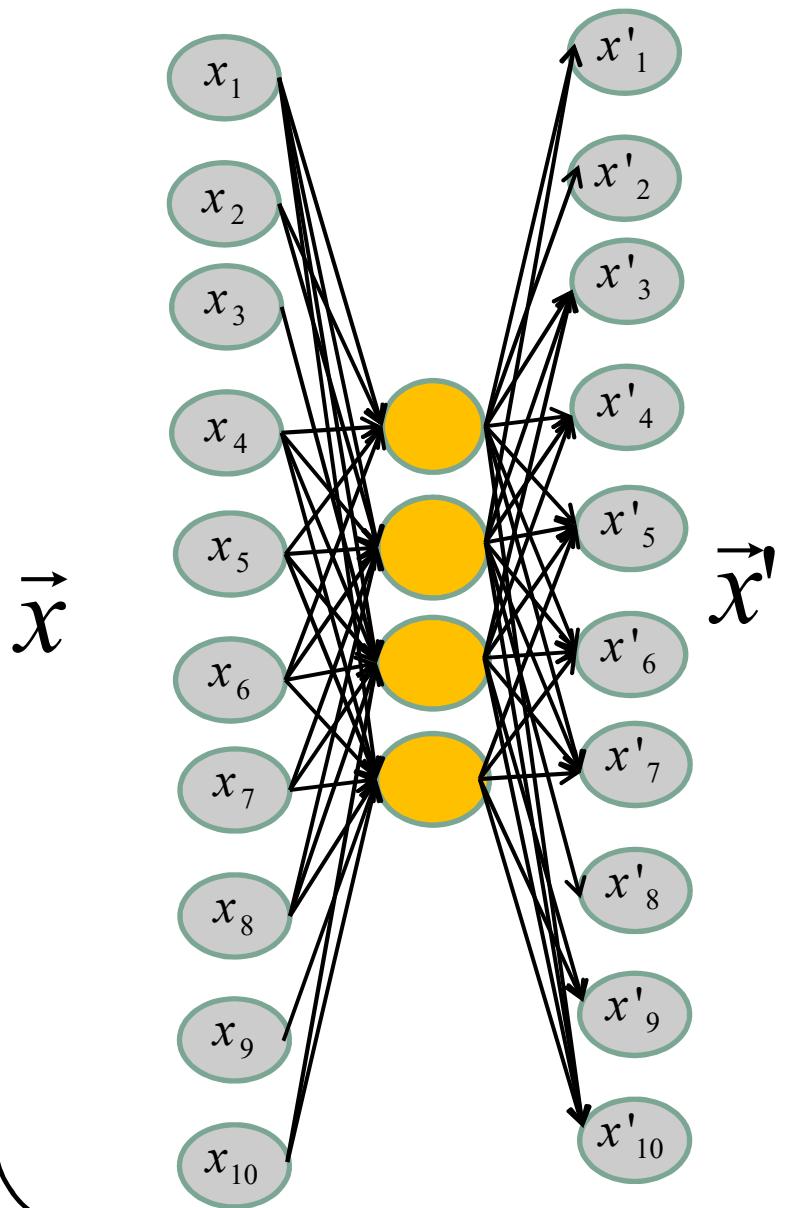
$$\vec{x}' = W_2 \vec{z}$$



$$\left. \begin{array}{l} \vec{z} = W_1 \vec{x} \\ \vec{x}' = W_2 \vec{z} \end{array} \right\} \vec{x}' = W_2 W_1 \vec{x}$$

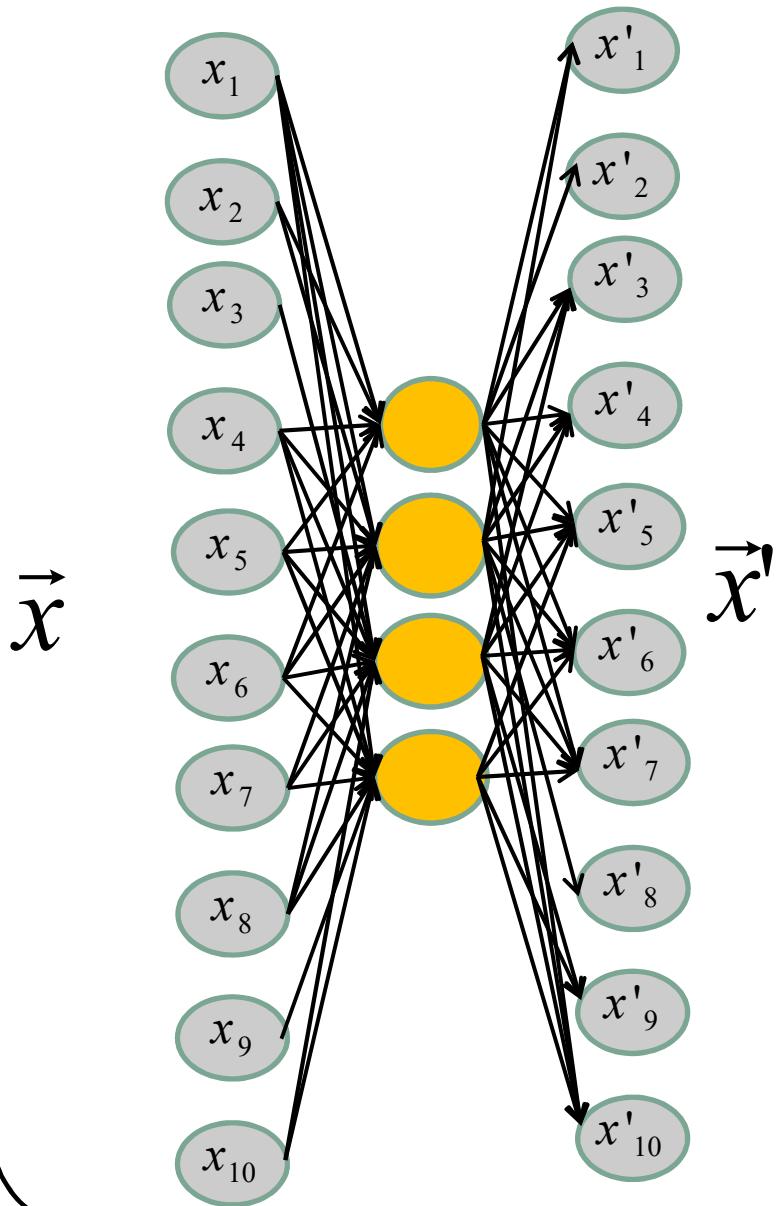


$$Loss = \|\vec{x} - W_2 W_1 \vec{x}\|^2$$



On peut démontrer que la solution est

$$W_1 = (W_2^T W_2)^{-1} W_2^T$$

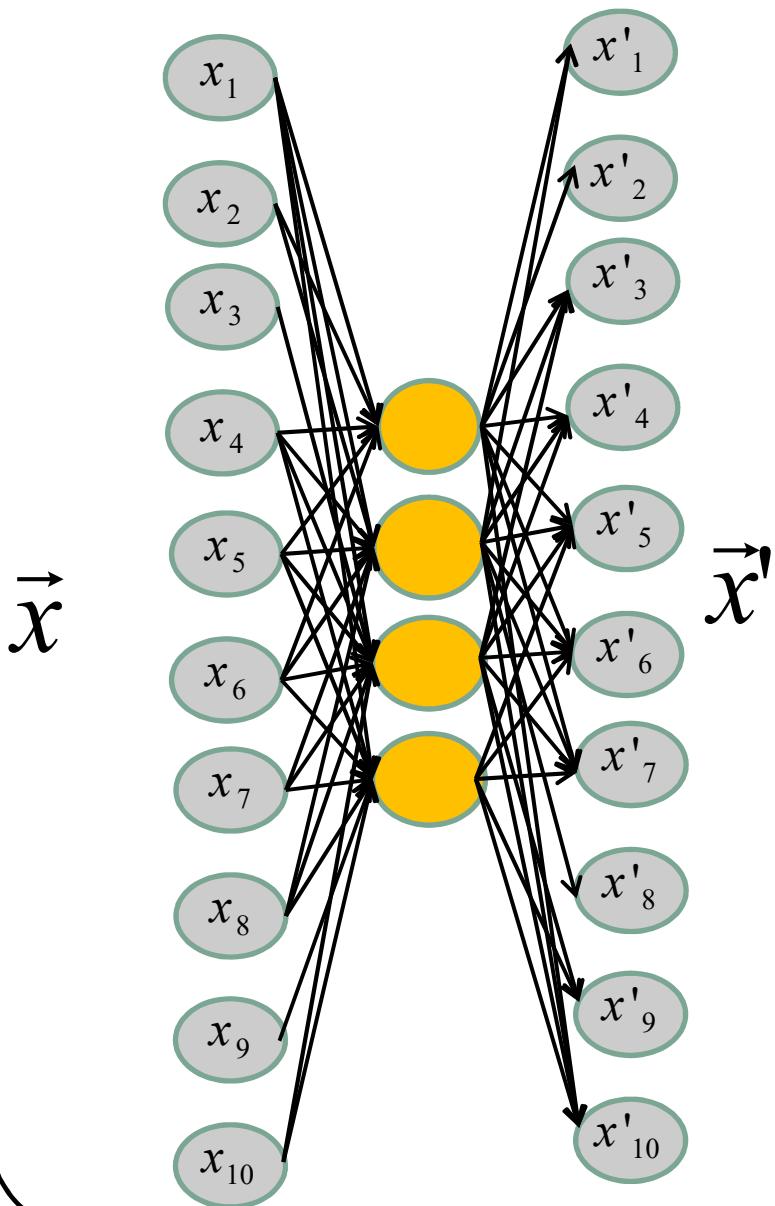


On peut démontrer que la solution est

$$W_1$$

Matrice constituée des principaux  
**vecteurs propres**  
 de la matrice de  
**variance-covariance.**  
 des données

**DONC**



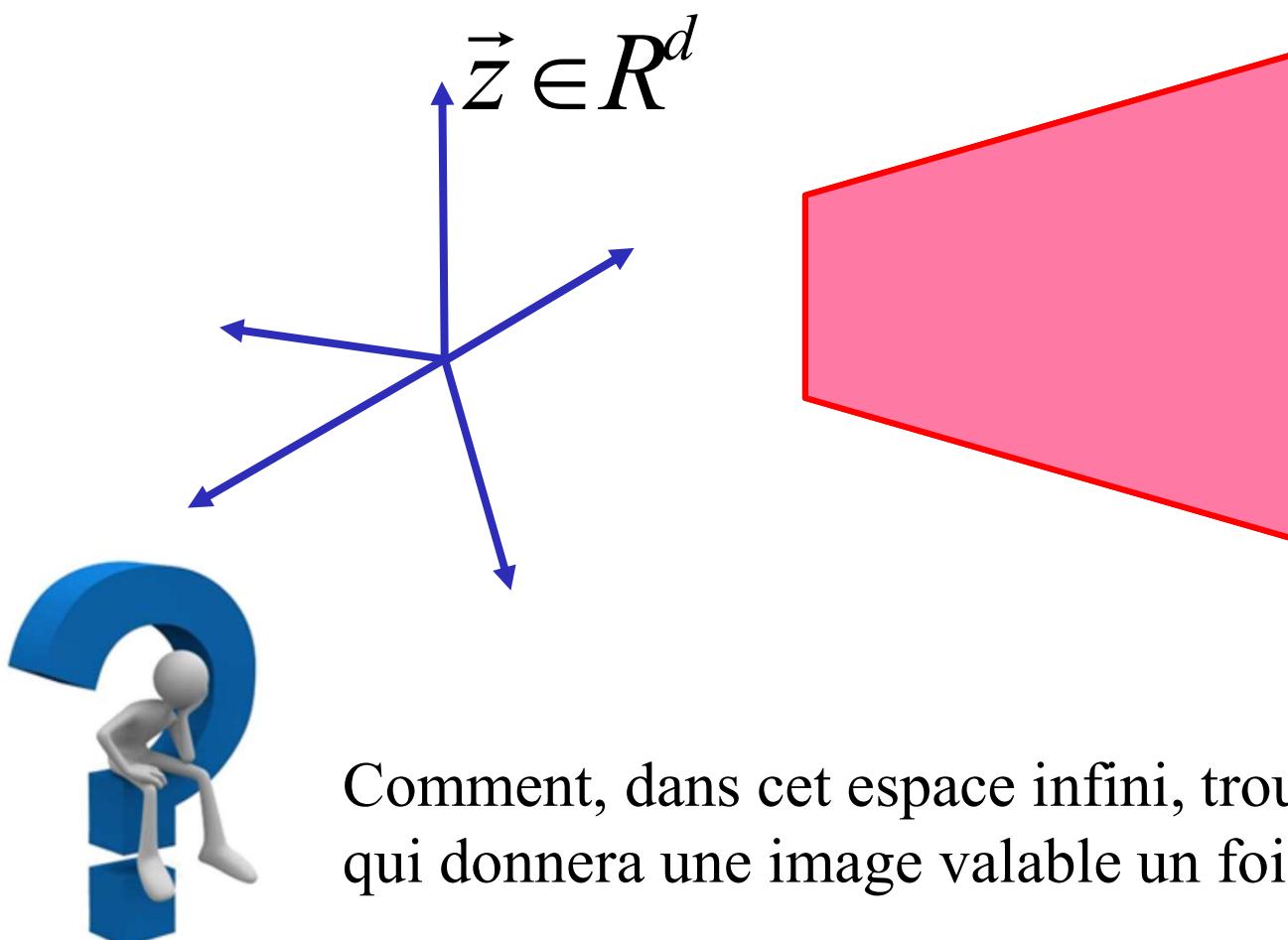
Autoencodeur linéaire

=

Analyse en composante principale  
(ACP)

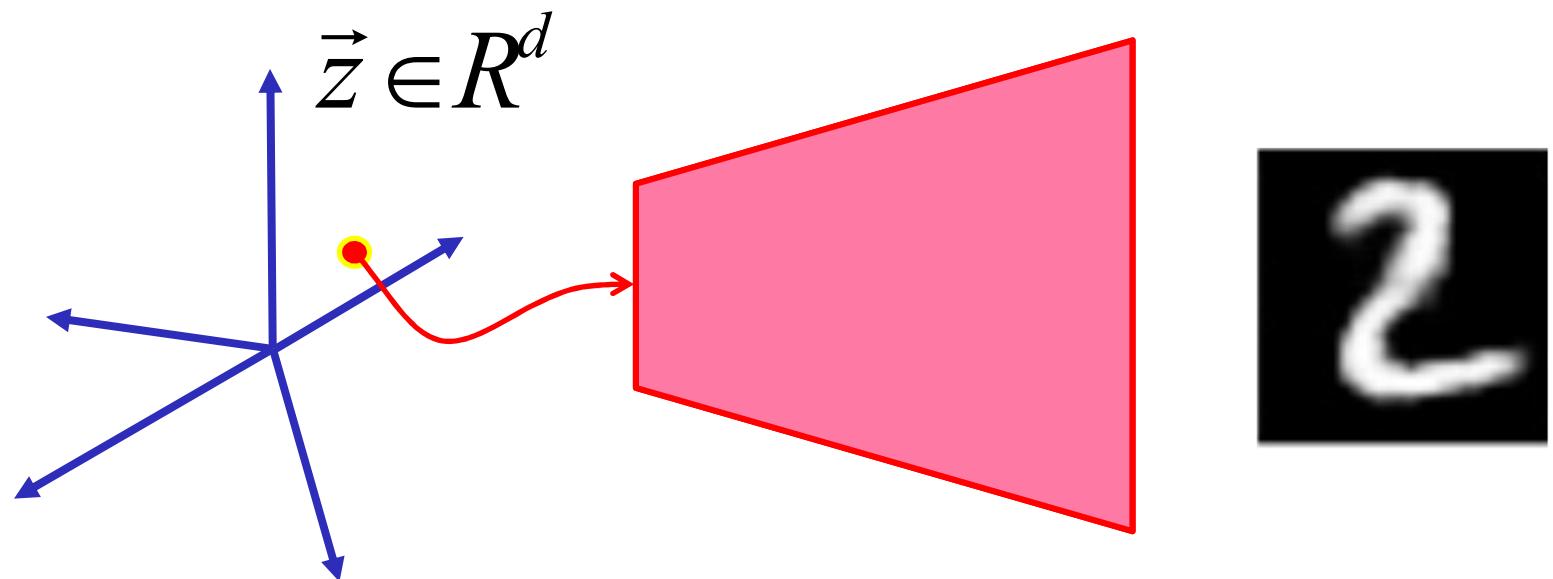
E. Plaut, **From Principal Subspaces to Principal Components with Linear Autoencoders**, arXiv:1804.10253v3, 2018

En général, l'espace latent possède entre 16 et 128 dimensions.  
Ça peut être parfois plus, et parfois moins.

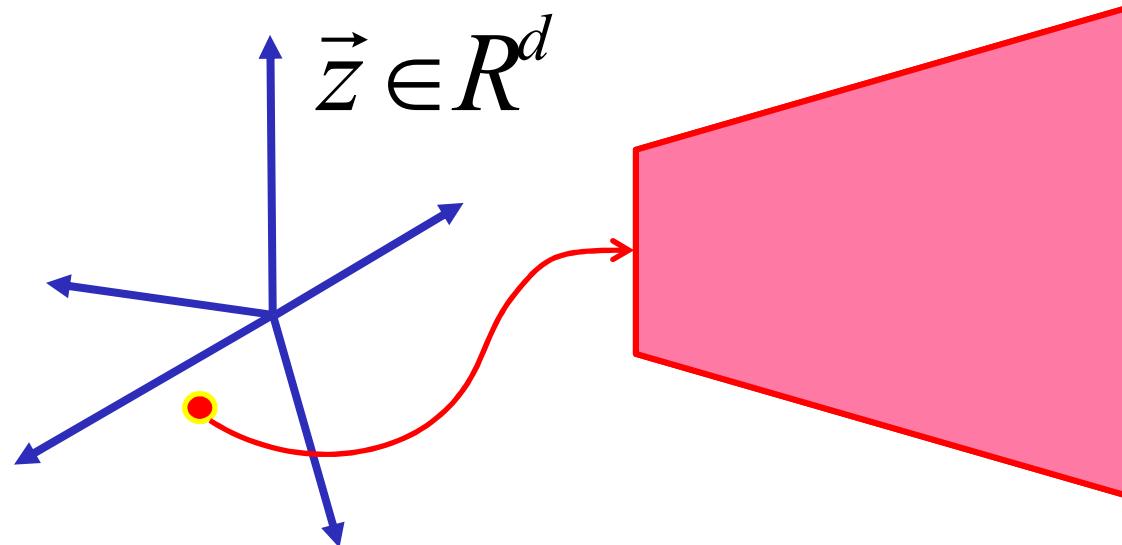


Comment, dans cet espace infini, trouver un point  $\vec{z}$  qui donnera une image valable un fois décodée?

Avec de la chance, on peut sélectionner un point au hasard et reproduire une « bonne » image (ici une image « MNIST »)



Malheureusement, la vaste majorité du temps, on reproduira du bruit

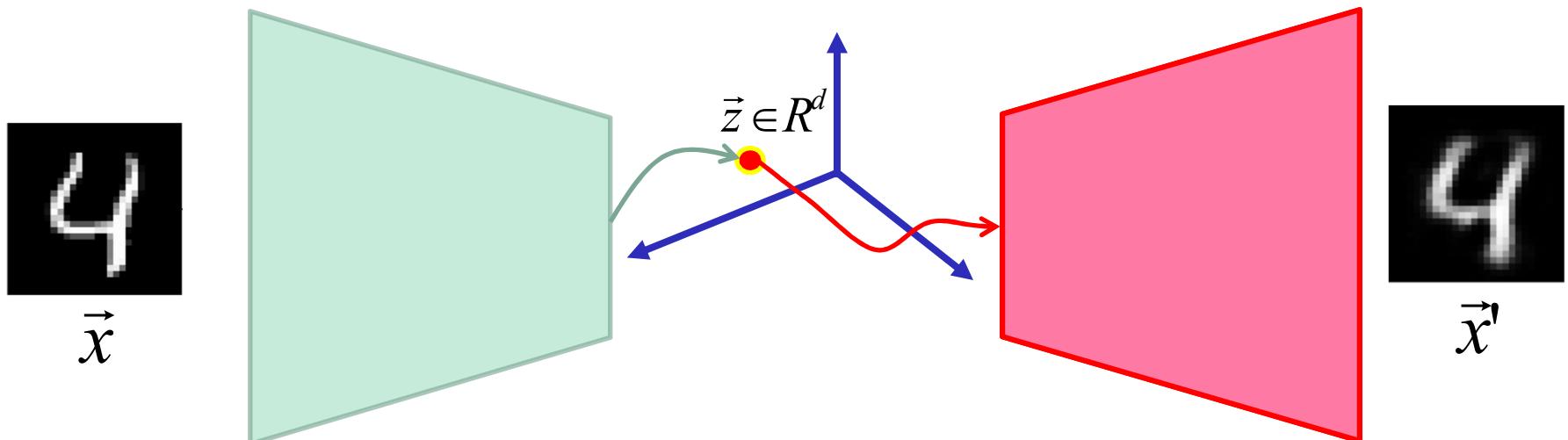


Au lieu d'apprendre à **reproduire**  
**un signal d'entrée...**

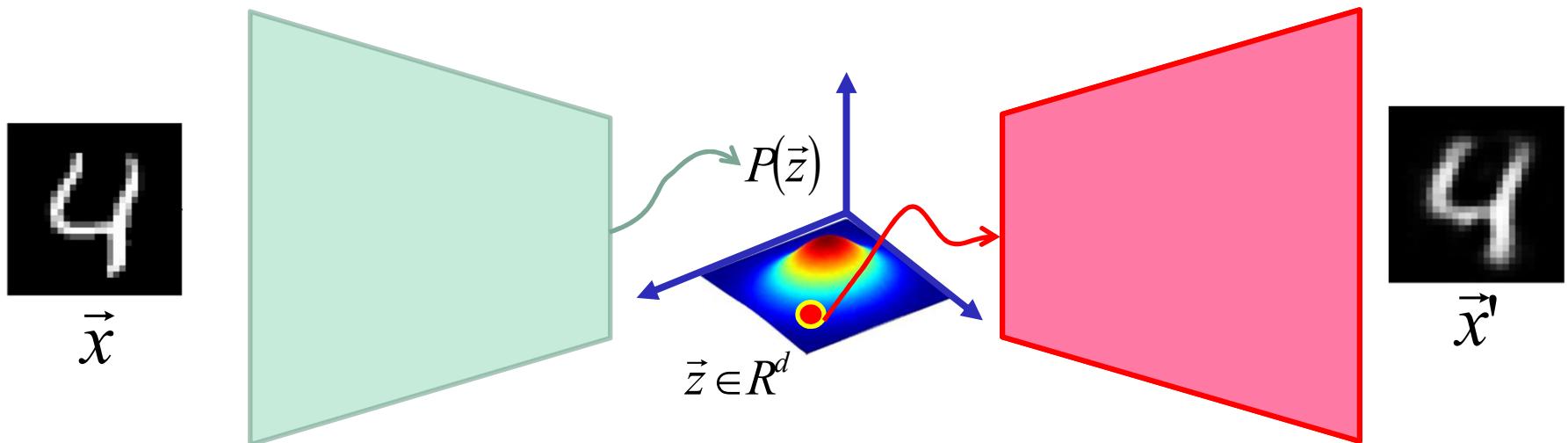


Apprendre à reproduire une **distribution**  $p(\vec{z})$   
**connue** de sorte qu'un **point échantillonné**  
**et décodé** de cette distribution correspond à  
un signal reconstruit valable

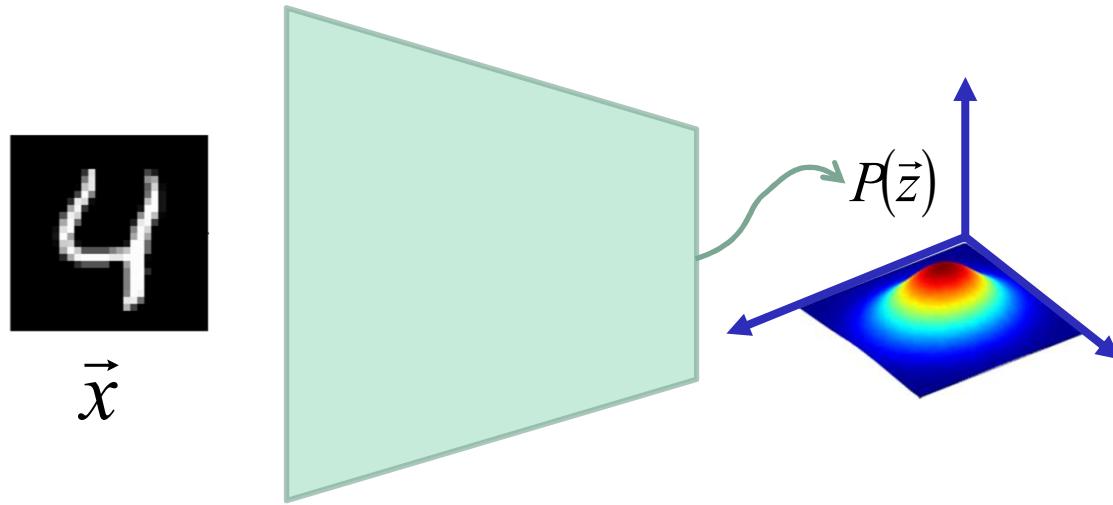
## Autoencodeur de base



## Autoencodeur variationnel

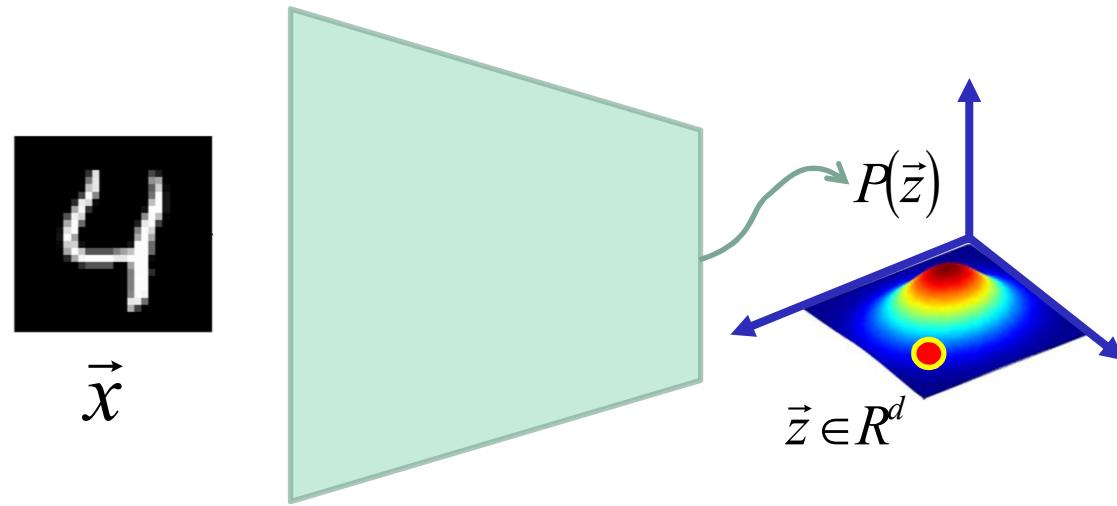


## Autoencodeur variationnel



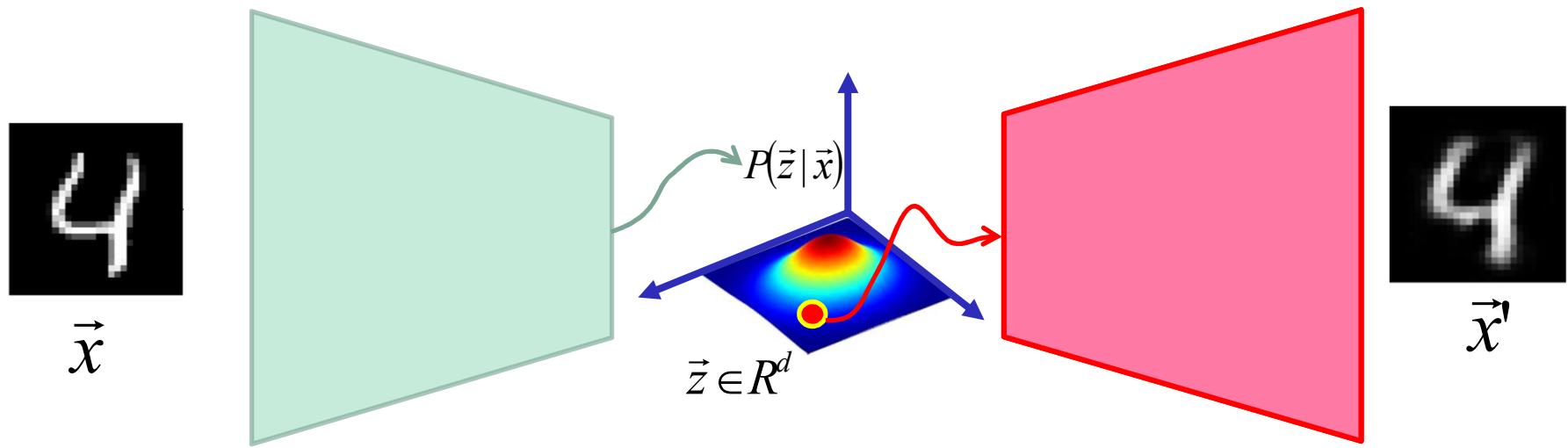
L'encodeur produit une distribution  $P(\vec{z})$   
et non juste un point  $\vec{z}$

## Autoencodeur variationnel



On échantillonne un  $\vec{z} \sim p(\vec{z})$  au hasard

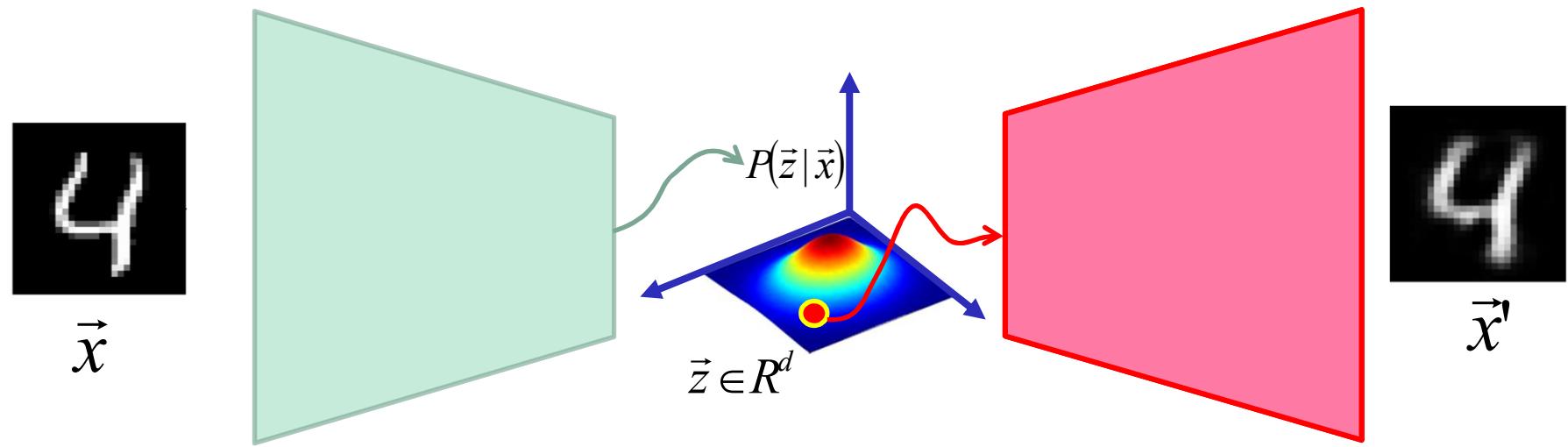
## Autoencodeur variationnel



On reconstruit  $\vec{x}'$

## Autoencodeur variationnel

### Remarque 1

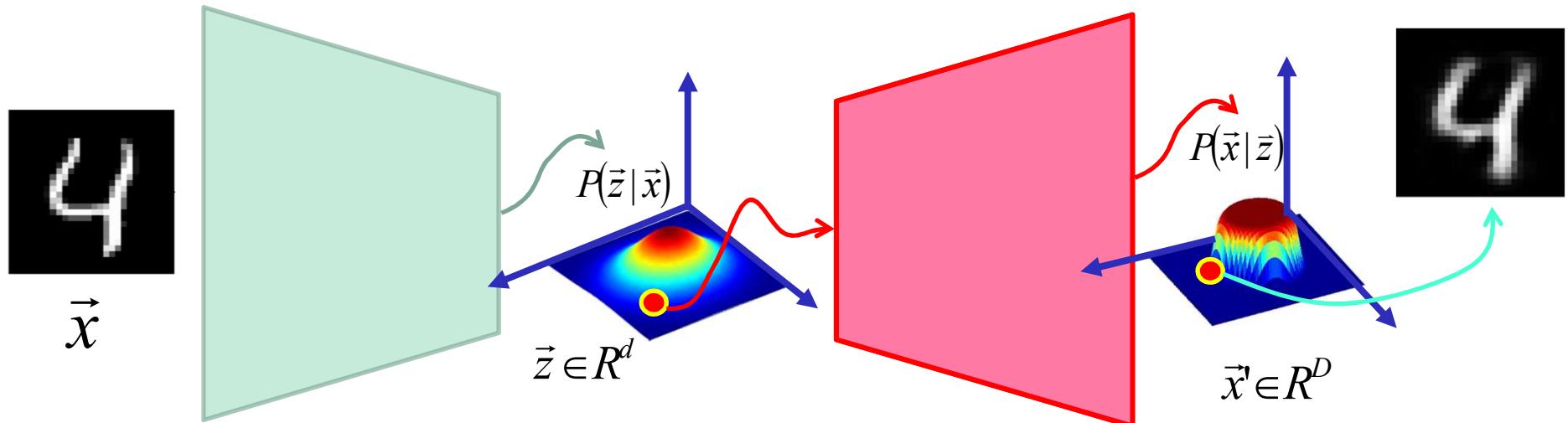


Puisque la distribution de  $\vec{z}$  dépend de  $\vec{x}$   
on dira que la distribution apprise est

$$P(\vec{z} | \vec{x})$$

## Autoencodeur variationnel

### Remarque 2



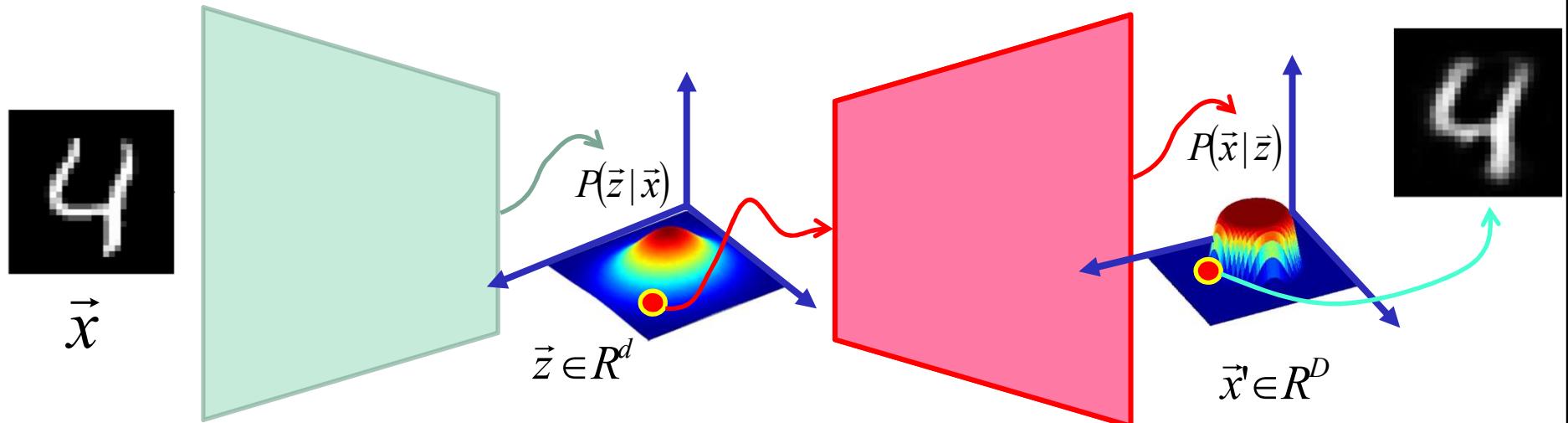
Le **décodeur** peut également produire une distribution de probabilités

$$P(\vec{x}' | \vec{z})$$

et  $\vec{x}'$  est un point échantillonné au hasard de  $P(\vec{x}' | \vec{z})$

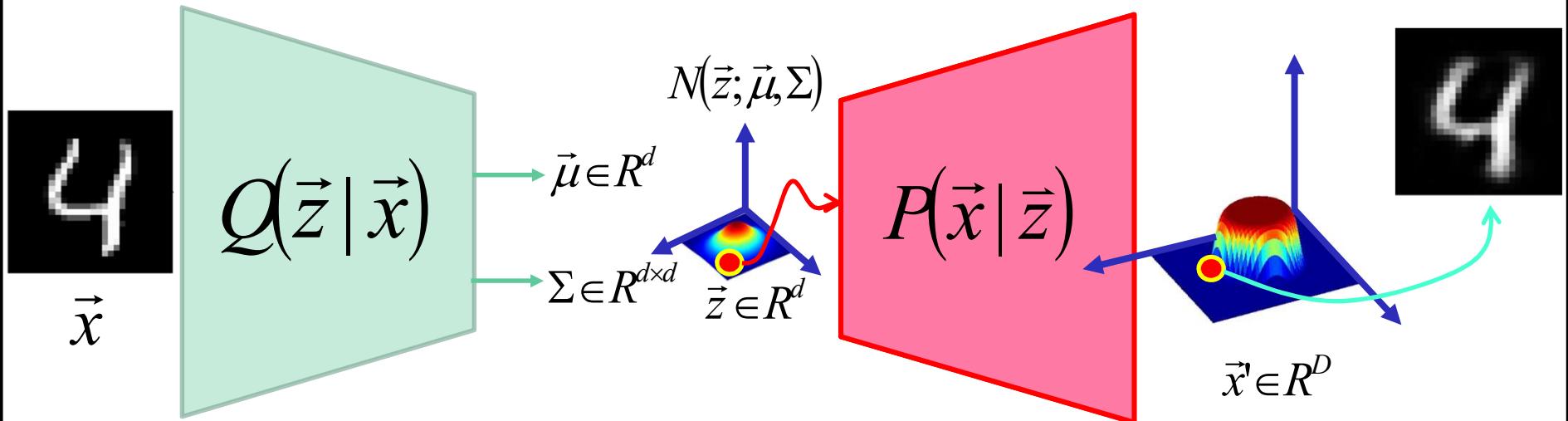
## Autoencodeur variationnel

### Remarque 3



La distribution  $P(\vec{z} | \vec{x})$  peut être très complexe et difficile à échantillonner, on va donc l'approximer par une distribution plus simple... une gaussienne

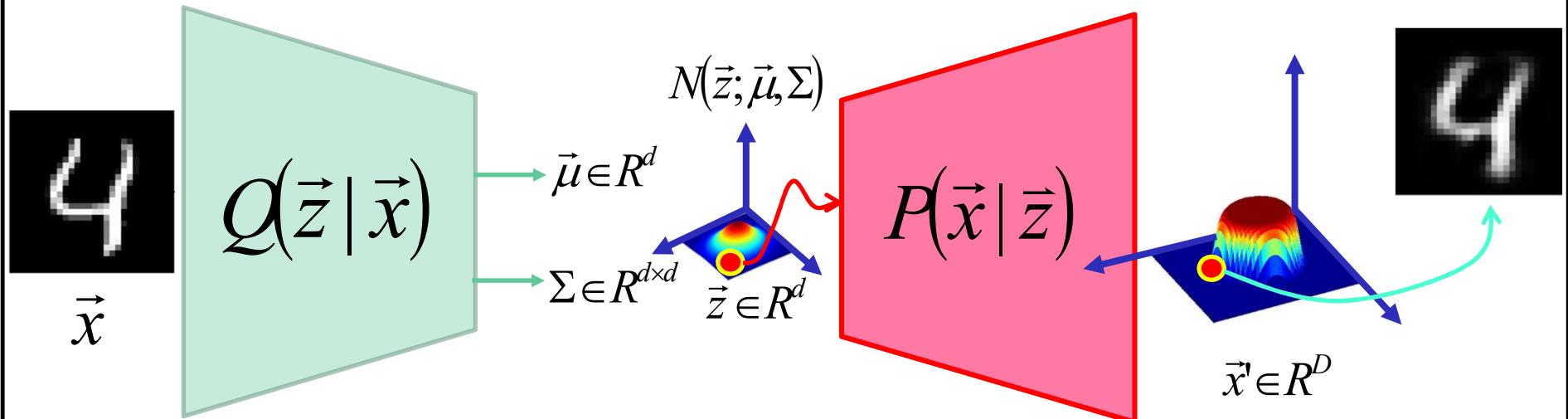
$$Q(\vec{z} | \vec{x}) \approx P(\vec{z} | \vec{x})$$



$$Q(\vec{z} | \vec{x}) \sim \text{Gaussienne}$$

## Autoencodeur variationnel

### Remarque 4

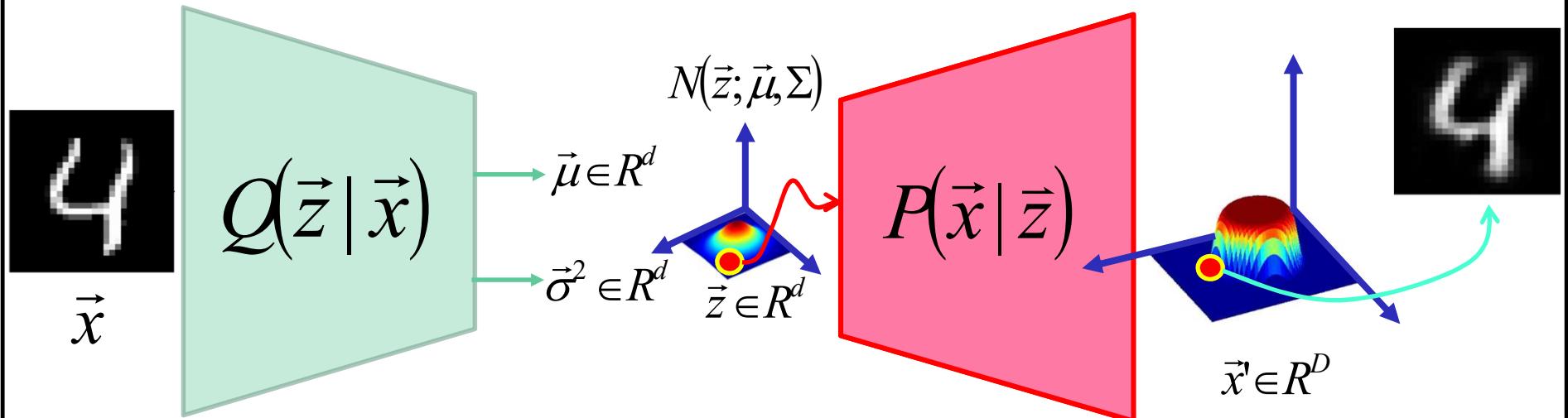


Pour simplifier les calculs, on va supposer que  $\Sigma$  est diagonale

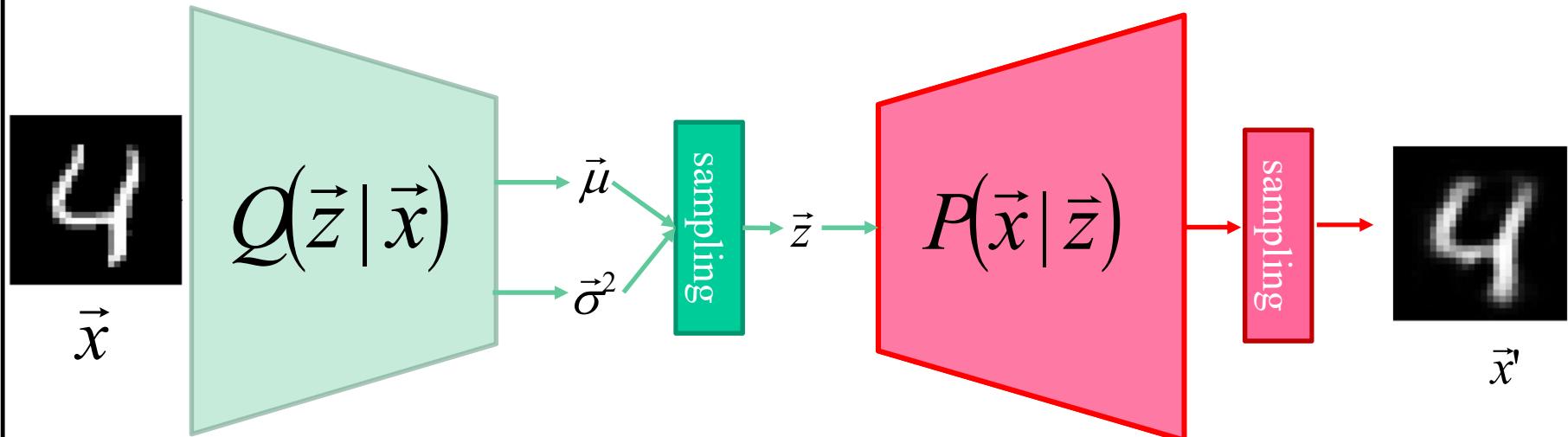
$$\Sigma = \begin{pmatrix} \sigma_1^2 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3^2 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & \sigma_d^2 \end{pmatrix}$$

On va donc **prédirer un vecteur de variances et non une matrice**

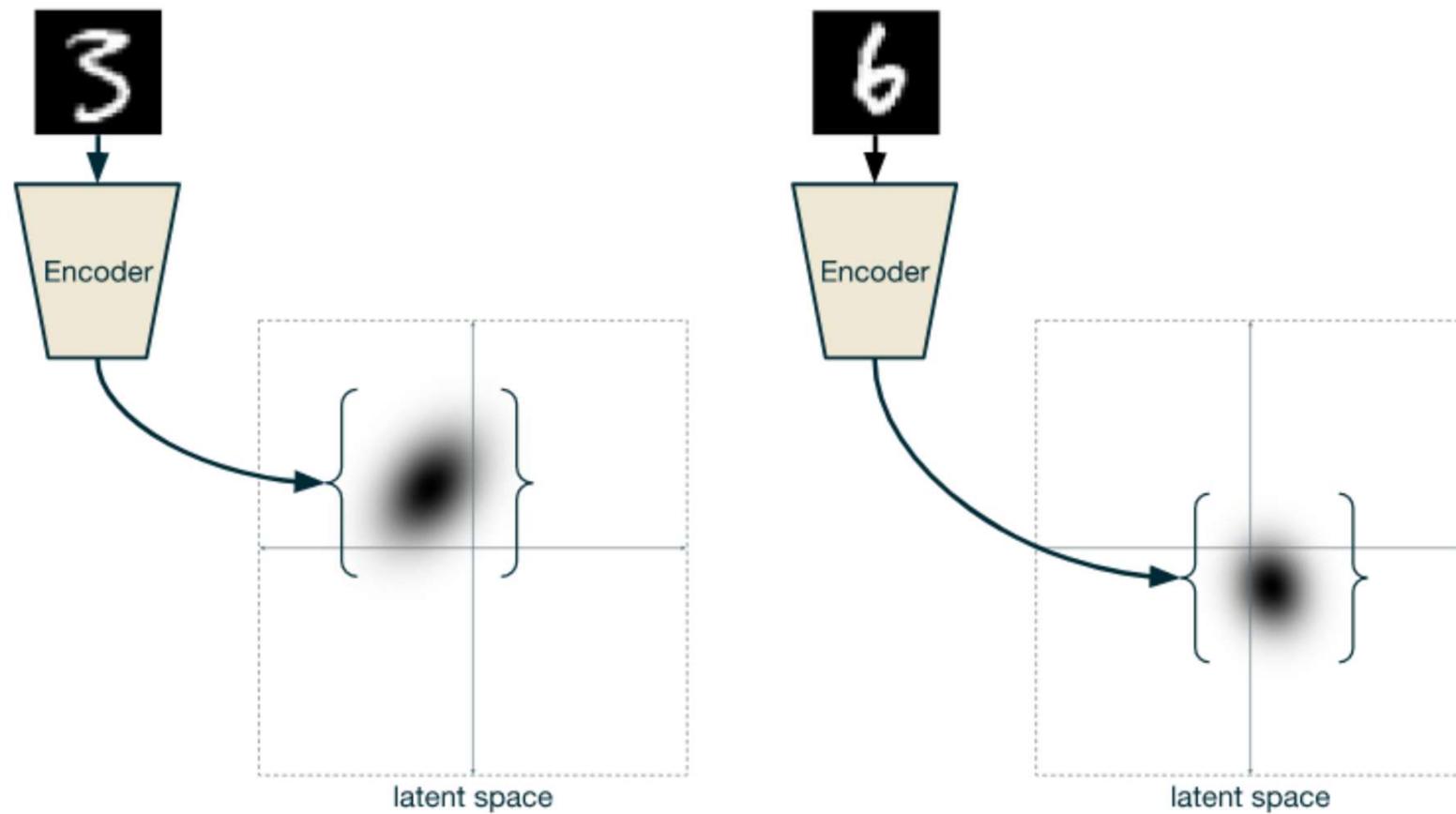
## Autoencodeur variationnel



## Autoencodeur variationnel

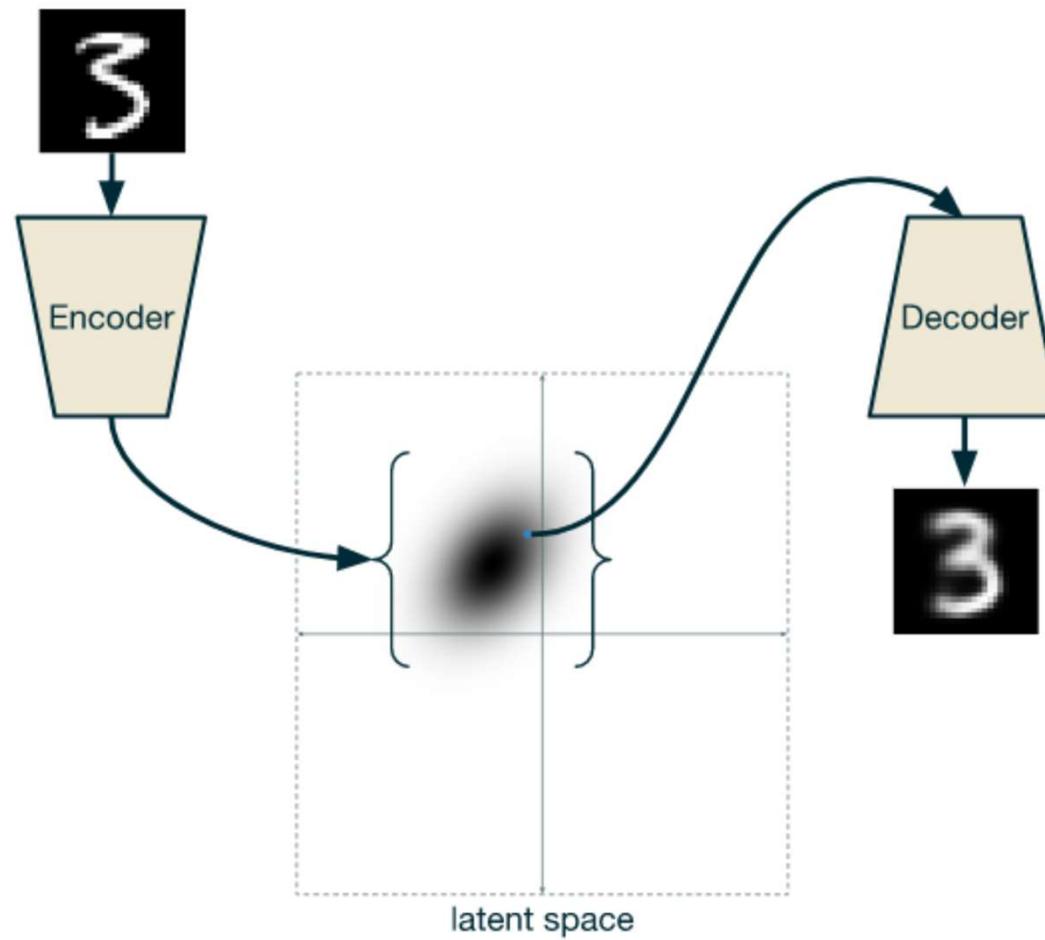


# Autre façon de voir les choses...



<https://ijdykeman.github.io/ml/2016/12/21/cvae.html>

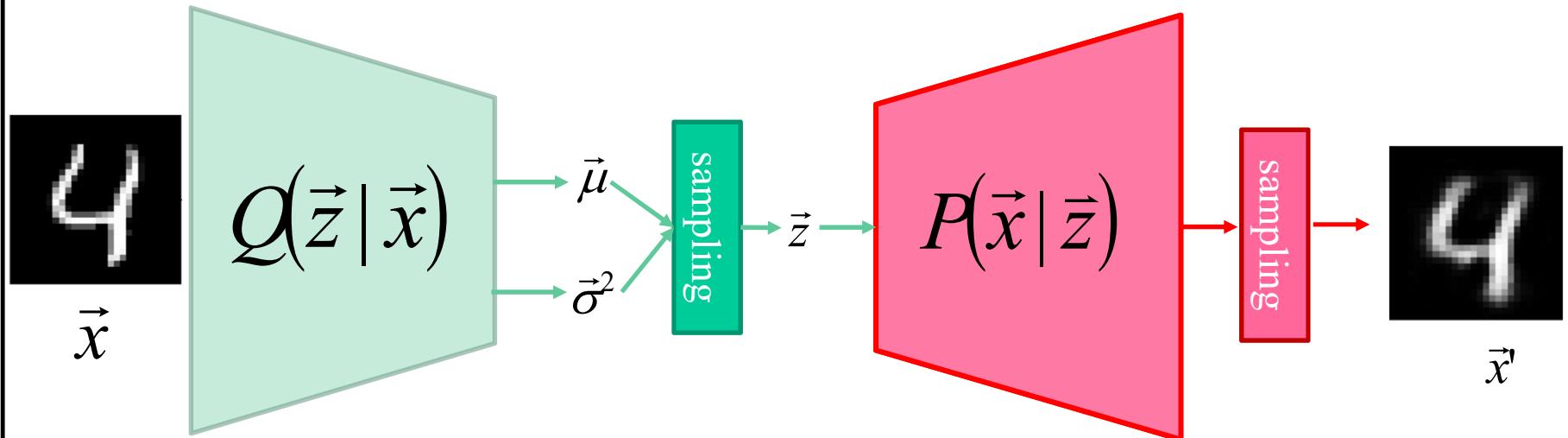
# Autre façon de voir les choses...



<https://ijdykeman.github.io/ml/2016/12/21/cvae.html>

## Autoencodeur variationnel

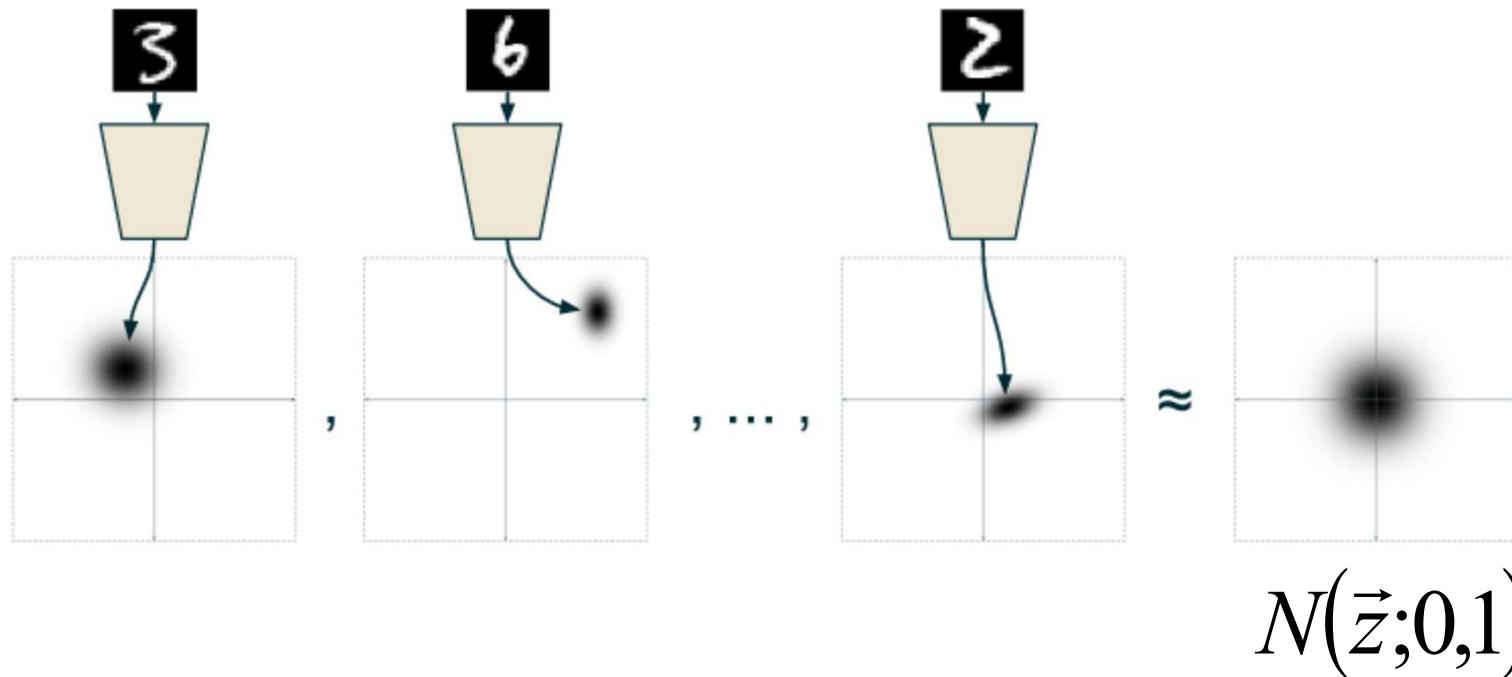
### Remarque 5



Distribution *a priori* de  $\vec{z}$  : gaussienne centrée à 0 et de variance 1

$$P(\vec{z}) = N(\vec{z}; 0, 1)$$

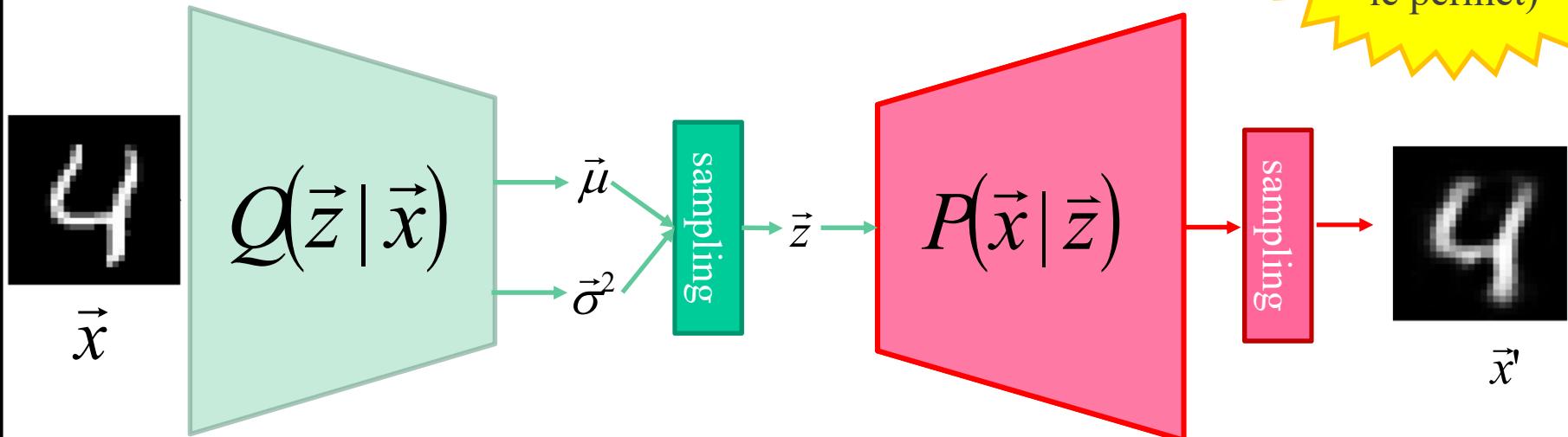
# Autre façon de voir les choses...



<https://ijdykeman.github.io/ml/2016/12/21/cvae.html>

## Autoencodeur variationnel

Preuve au tableau  
 (si le temps le permet)

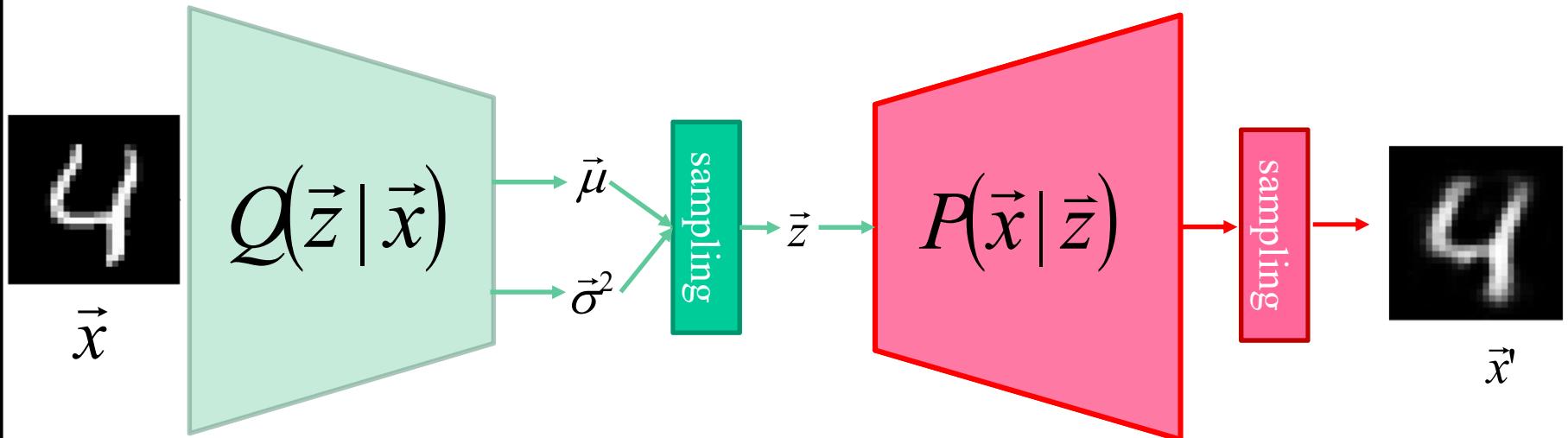


*ELBO loss : Evidence Lower Bound*

$$Loss = \underbrace{KL\left(N(\vec{z}; 0, 1), N(\vec{z}; \vec{\mu}, \Sigma)\right)}_{\text{Perte encodeur}} - \underbrace{\log(P(\vec{x} | \vec{z}))}_{\text{Perte décodeur}}$$

# Autoencodeur variationnel

D.Kingma, M.Welling, **Auto-Encoding Variational Bayes**, arXiv:1312.6114v10 ([Annexe B](#))

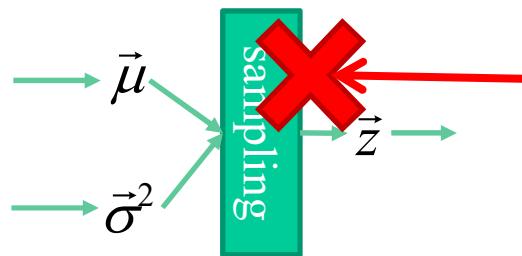


*ELBO loss : Evidence Lower Bound*

$$Loss = \frac{1}{2} \sum_{i=1}^d \underbrace{\left( 1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2 \right)}_{\text{Perte encodeur}} - \underbrace{\log(P(\vec{x} | \vec{z}))}_{\text{Perte décodeur}}$$

## Autoencodeur variationnel

### Remarque 6

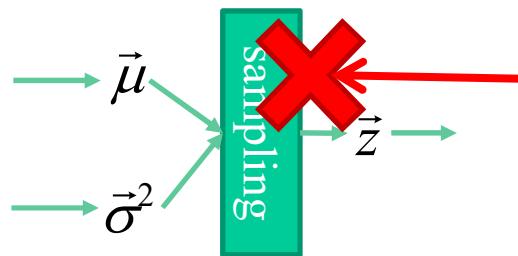


Pas de rétro-propagation à travers  
un processus d'échantillonage

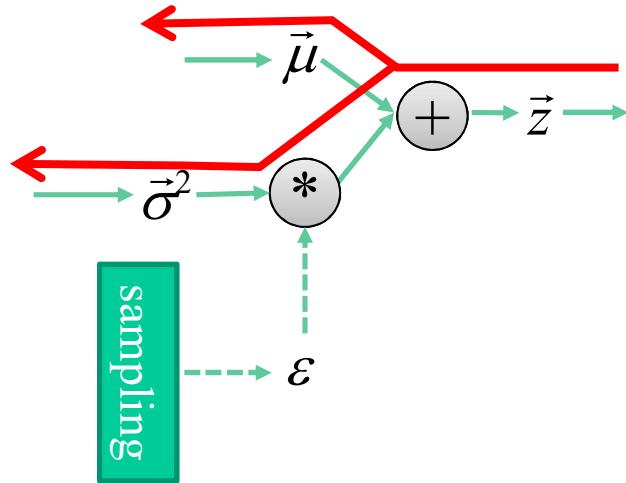
$$\vec{z} \sim N(\vec{z}; \vec{\mu}, \Sigma)$$

# Autoencodeur variationnel

## Remarque 6



Pas de rétro-propagation à travers  
un processus d'échantillonage  
 $\vec{z} \sim N(\vec{z}; \vec{\mu}, \Sigma)$



*Reparameterization trick*

$$\vec{z} = \vec{\mu} + \varepsilon \vec{\sigma}^2$$

# Autoencodeur variationnel jouet MNIST : d=32 dim

```
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128), nn.ReLU(True),
            nn.Linear(128, 64), nn.ReLU(True),
            nn.Linear(64, 32*2)
        self.decoder = nn.Sequential(
            nn.Linear(32, 64), nn.ReLU(True),
            nn.Linear(64, 128), nn.ReLU(True),
            nn.Linear(128, 28 * 28))

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    def forward(self, x):
        enc_x = self.encoder(x)
        mu = enc_x[:, :32]
        logvar = stats[:, 32:]
        z = self.reparameterize(mu, logvar)
        return self.decoder(z), mu, logvar
```

} Reparameterization  
trick

}

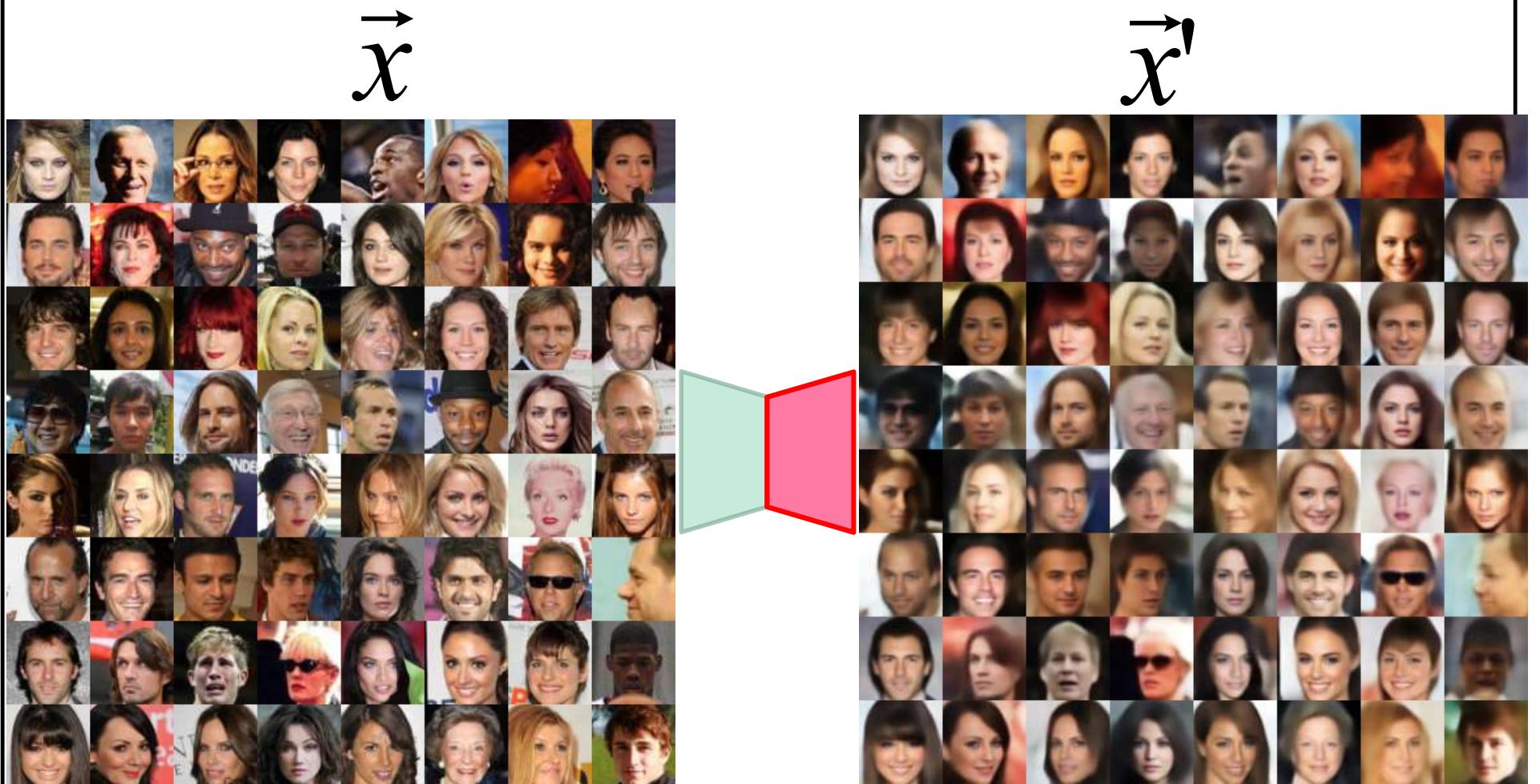
## Autoencodeur variationnel jouet MNIST : d=32 dim

```
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')

    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

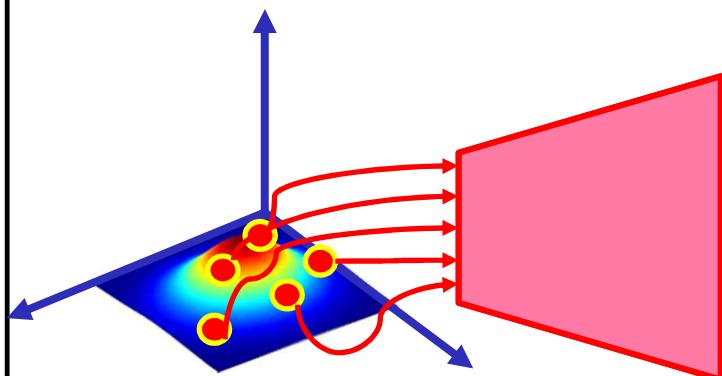
    return BCE + KLD
```

# Ex.: base de données *CelebA*



# Ex.: base de données *CelebA*

Décodage d'échantillons aléatoires  $\vec{z}$



# Plusieurs tutoriels, VAE

- <https://ijdykeman.github.io/ml/2016/12/21/cvae.html>
- <https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/>
- <https://towardsdatascience.com/deep-latent-variable-models-unravel-hidden-structures-a5df0fd32ae2>
- C. Doersch, **Tutorial on Variational Autoencoders**, arXiv:1606.05908

# GAN

Generative Adversarial Nets

On voudrait générer des images  $\vec{x}$  en échantillonnant  $P(\vec{x})$

=> **TROP DIFFICILE** car  $P(\vec{x})$  trop complexe



Comme précédemment, pour simplifier le problème, on pourrait introduire une variable latente  $\vec{z}$  et ainsi modéliser

$$P(\vec{x}, \vec{z}) = P(\vec{x} | \vec{z})P(\vec{z})$$

Modèle génératif

Distribution *a priori*

Comme pour les VAE, on utilisera une **distribution *a priori*** facile à échantillonner : une **gaussienne!**

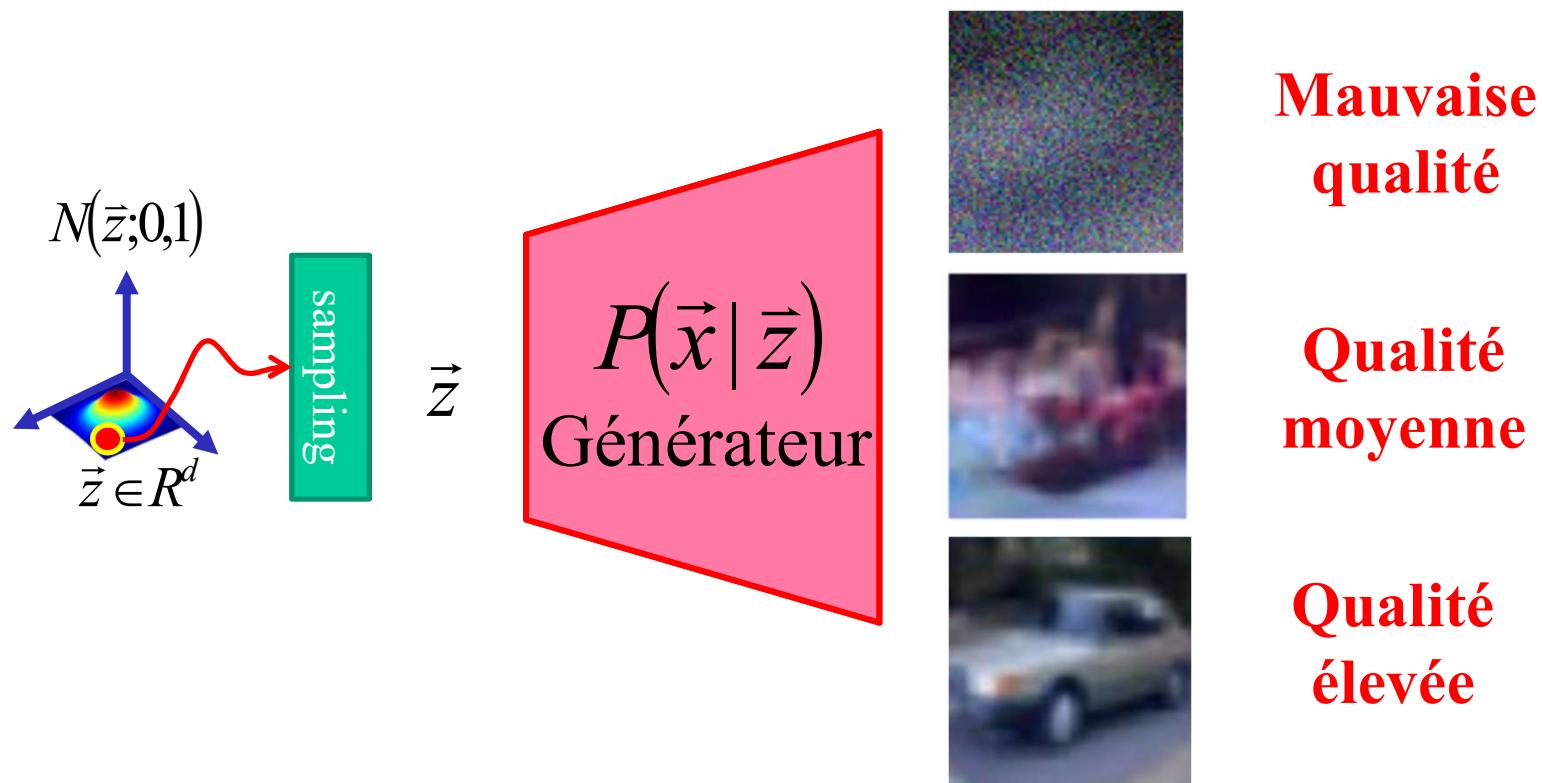
$$P(\vec{z}) = N(\vec{z}; 0, 1)$$

Comment estimer  $P(\vec{x} | \vec{z})$  ?

À l'aide d'un réseau de neurones car ce sont **d'excellentes machines pour estimer des probabilités conditionnelles**



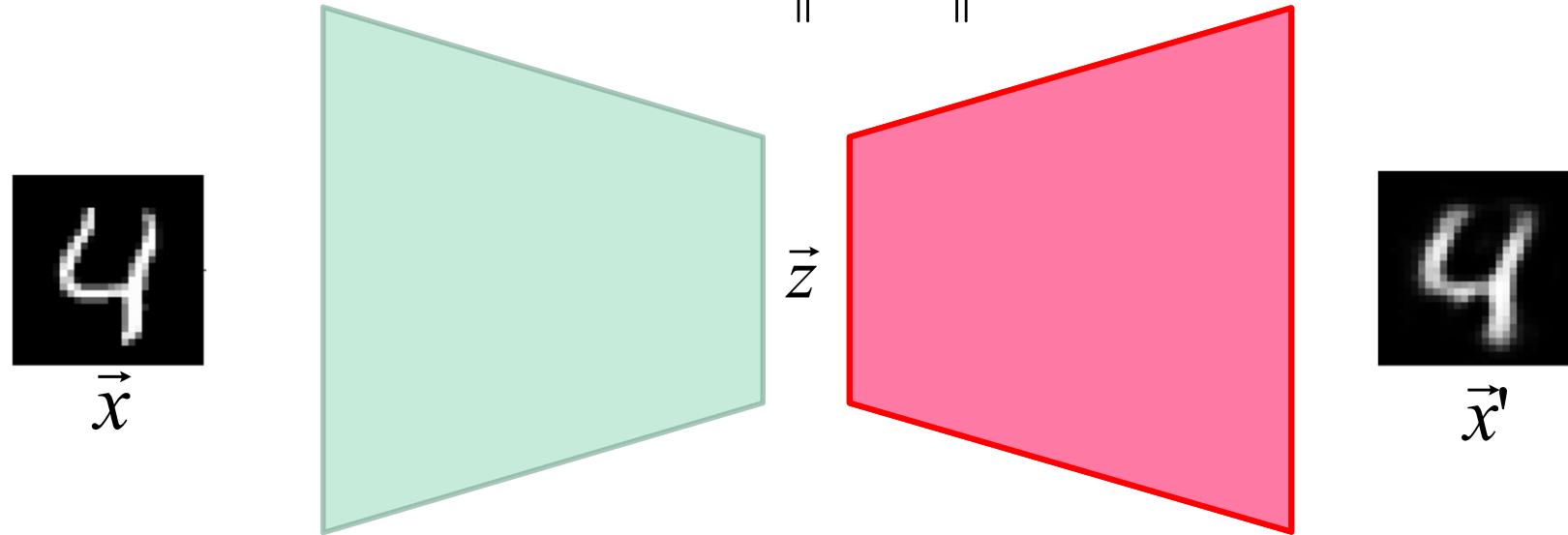
Dépendamment des performances du décodeur, les images générées  
**seront de qualité très variable.**



Pour entraîner un décodeur, il faut une **fonction de perte** (*loss*) qui **mesure la qualité (degré de réalisme) des images produites**

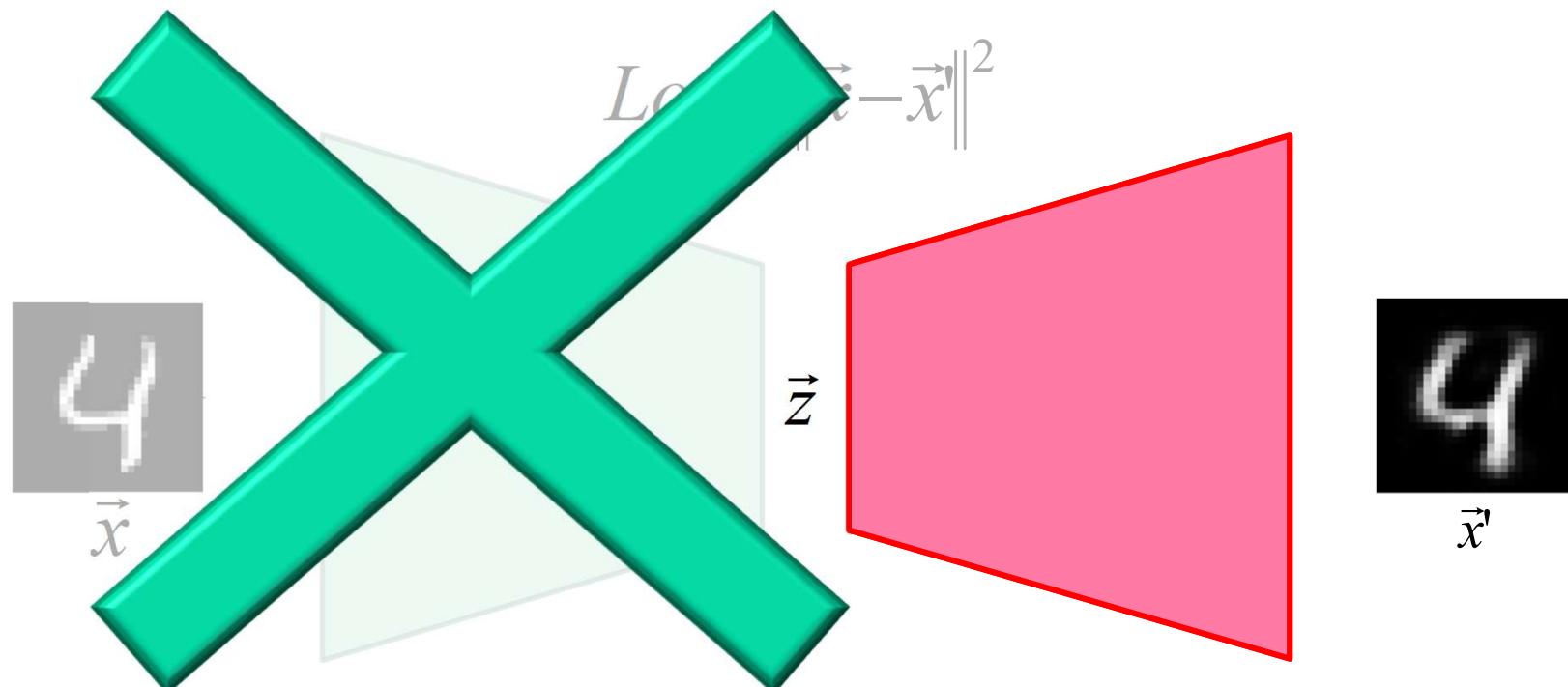
Pour un autoencodeur (variationnel ou non) c'est facile!  
car on a un encodeur et une image de référence

$$Loss = \|\vec{x} - \vec{x}'\|^2$$

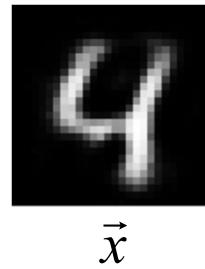


Pour entraîner un décodeur, il faut une **fonction de perte** (*loss*) qui **mesure la qualité (degré de réalisme) des images produites**

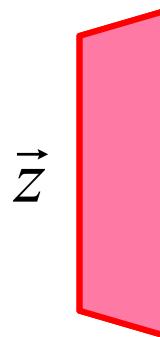
Comment faire pour un réseau **sans encodeur**?



$$\vec{z} \quad G(\vec{z})$$



Loss sans cible de référence?



Approximer la perte à l'aide d'un

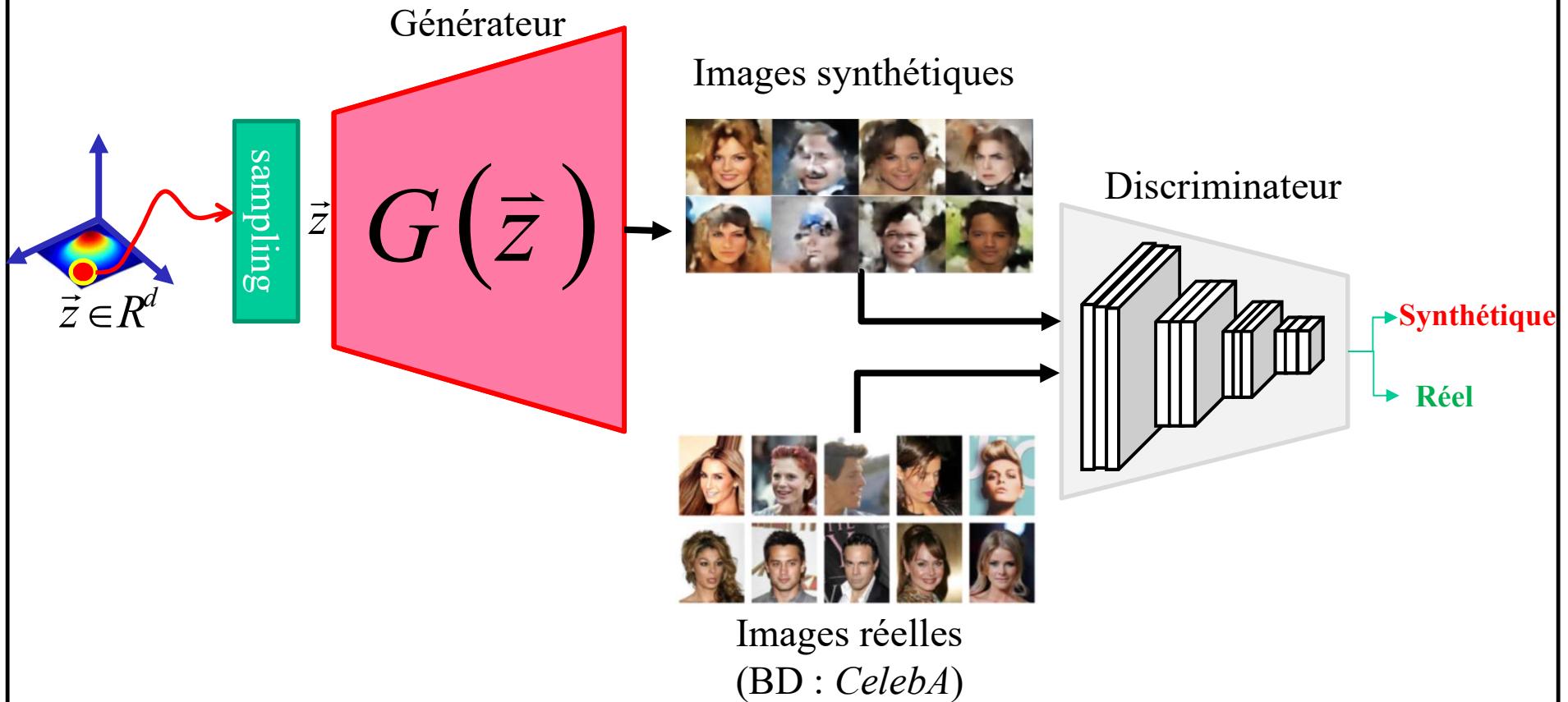
**2<sup>e</sup> réseau de neurones**

et d'une

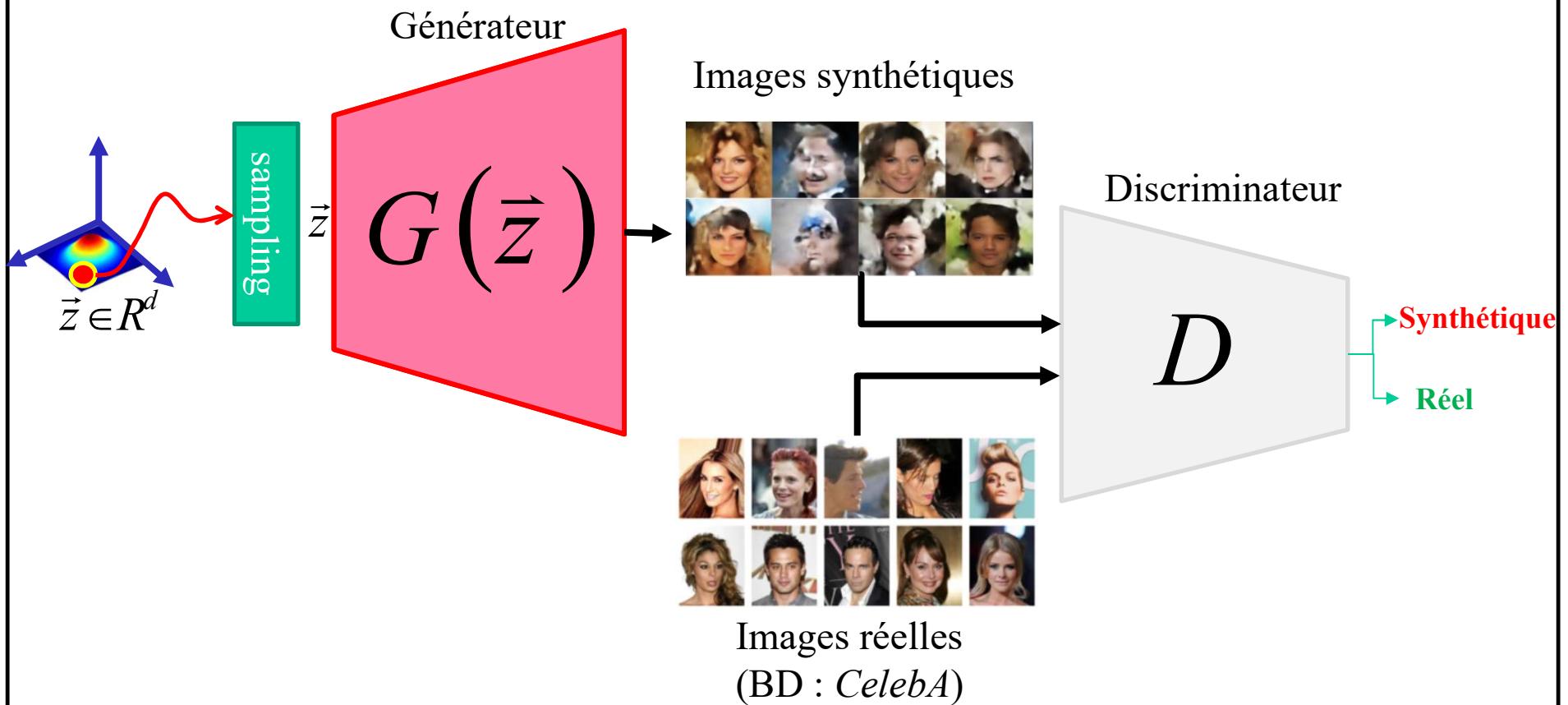
**Base de données de référence**

référence?

Pour entraîner un décodeur, il faut une **fond de perte** (loss) qui mesure la qualité des images produites



Pour entraîner un décodeur, il faut une **fonction de perte** (loss) qui mesure la qualité des images produites



# Données étiquetées



Produites par le générateur

$t = 0$  ('synthétique')

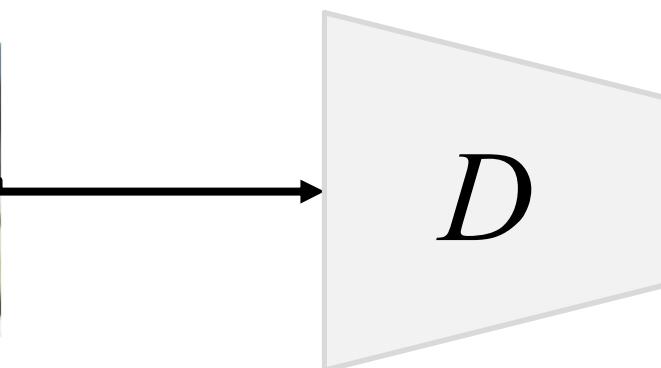


$t = 1$  ('réel')

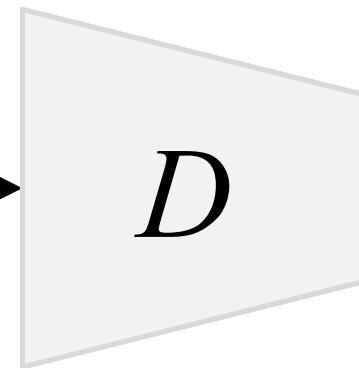
Issues d'une vraie BD

Deux réseaux aux **objectifs différents** :

**Discriminateur** : différentie les images synthétiques des images réelles



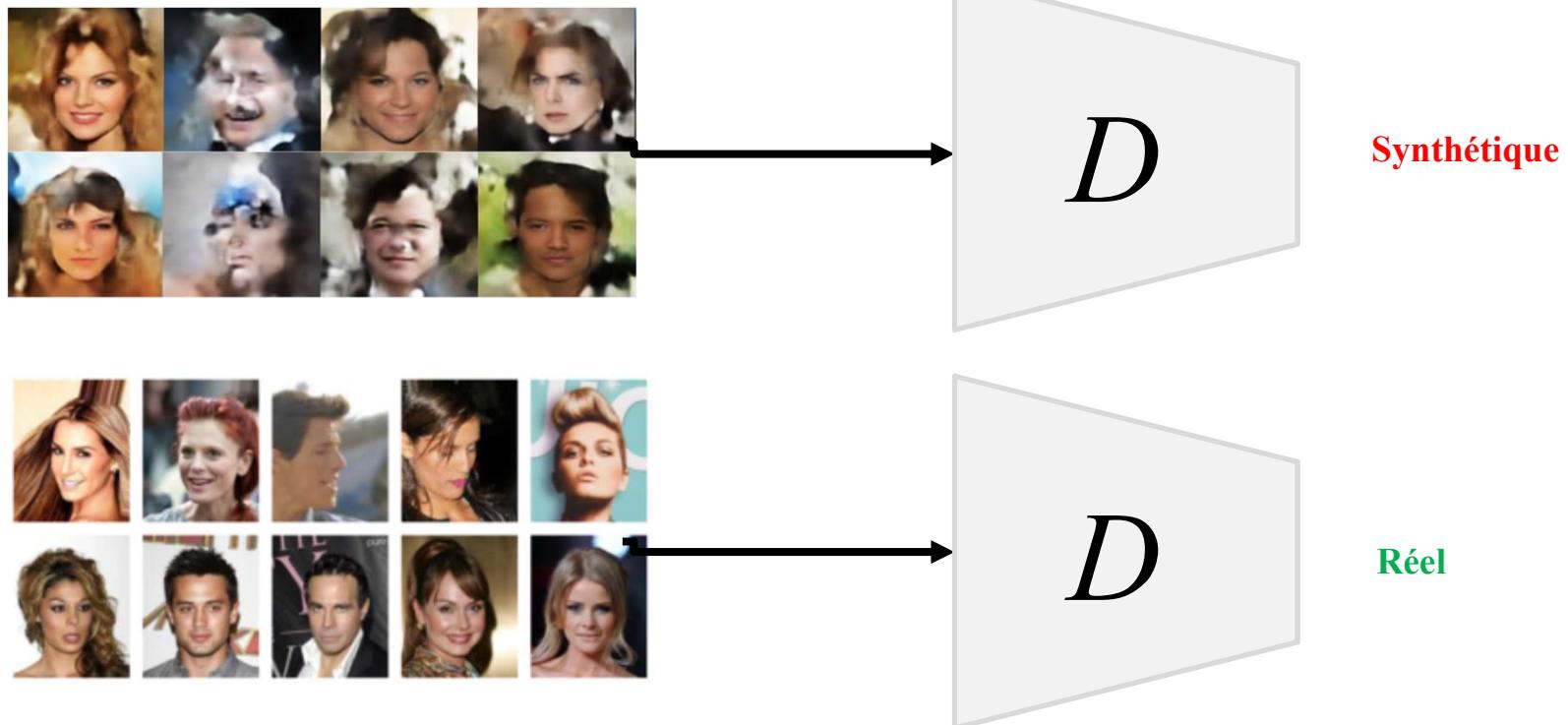
Synthétique



Réel

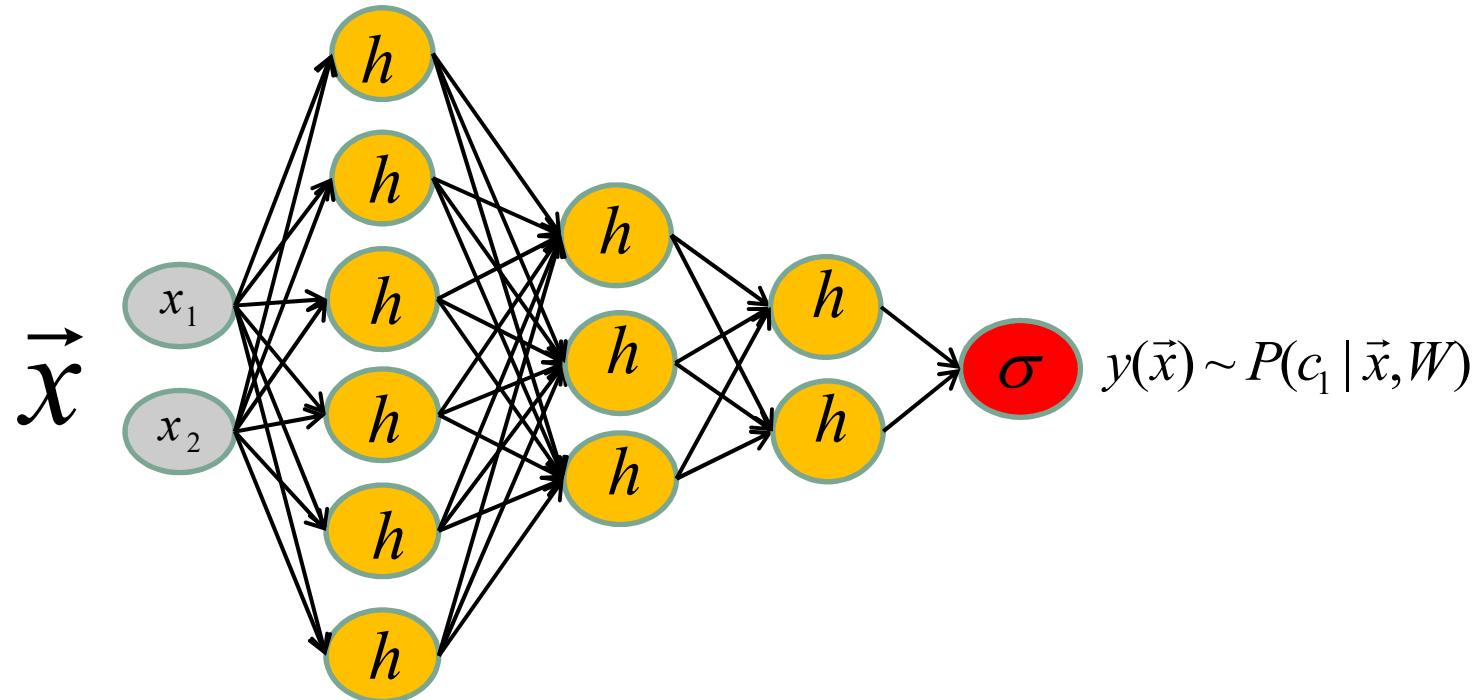
## Discriminateur : classifieur binaire (régression logistique)

=> Perte l'entropie croisée



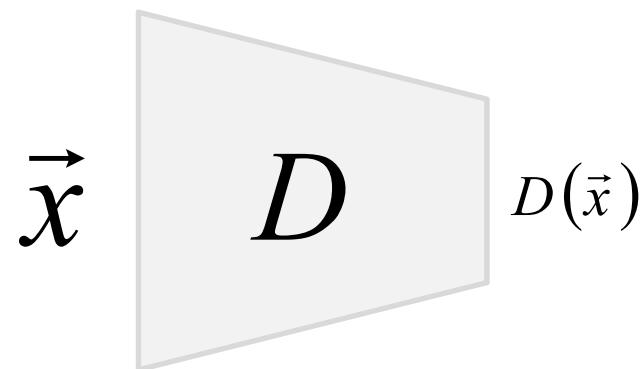
**Rappel, entropie croisée** pour une régression logistique binaire:

$$L_D = \frac{1}{N} \sum_i -t_i \ln(y(\vec{x}_i)) - (1 - t_i) \ln(1 - y(\vec{x}_i))$$



Le réseau discriminateur est représenté par la **lettre D**

$$L_D = \frac{1}{N} \sum_i -t_i \ln(D(\vec{x}_i)) - (1 - t_i) \ln(1 - D(\vec{x}_i))$$

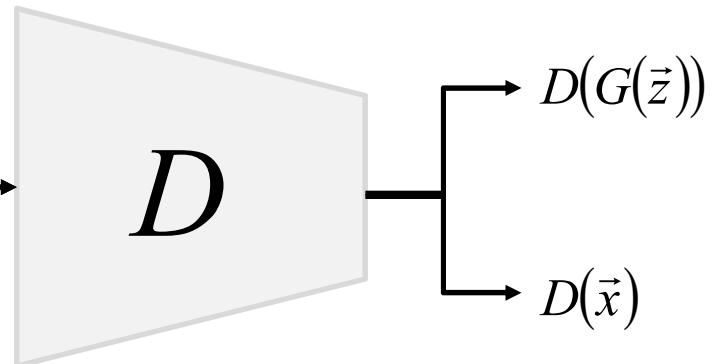


Puisque les images **synthétiques** ont été générées par le **générateur**

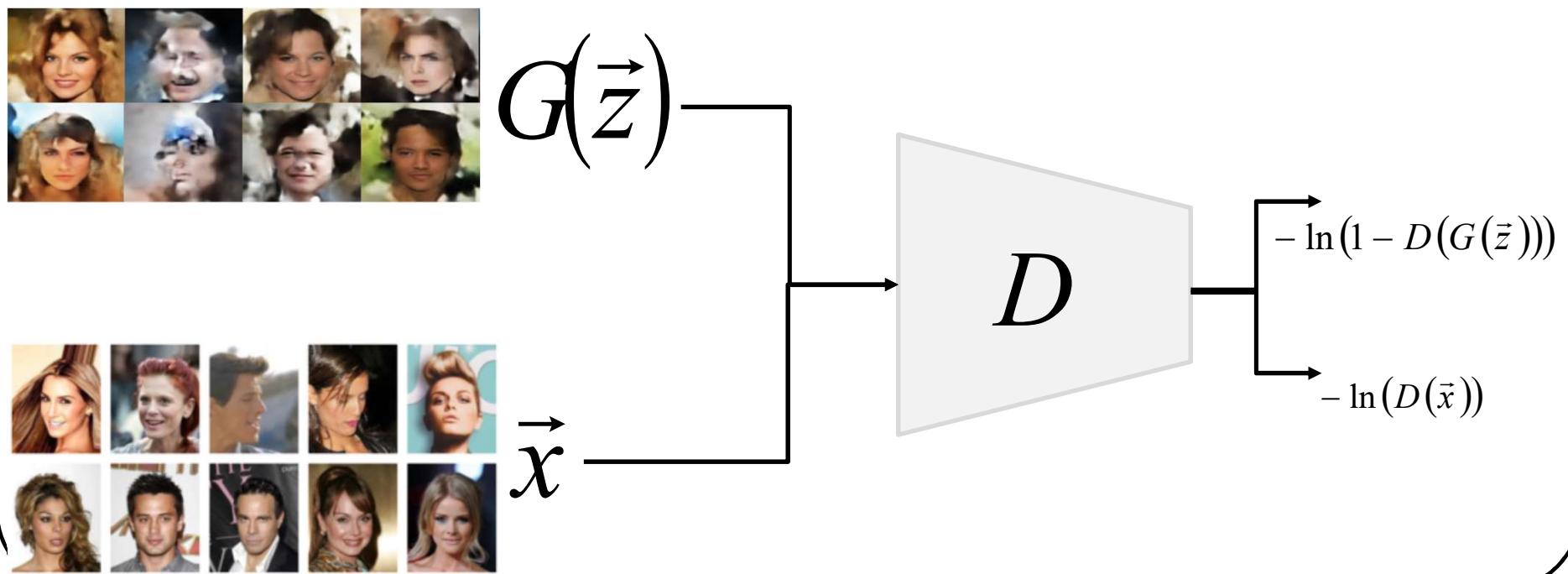
$$L_D = \frac{1}{N} \sum_i -t_i \ln(D(\vec{x}_i)) - (1 - t_i) \ln(1 - D(G(\vec{z}_i)))$$



$$G(\vec{z}) \longrightarrow \vec{x}$$



Sans perte de généralité, séparer la loss des images réelles et synthétiques



## Rappel: Espérance mathématique et approximation Monte Carlo

$$IE[x] = \int xp(x)dx$$

$$IE[f(x)] = \int f(x)p(x)dx$$

## Rappel: Espérance mathématique et approximation Monte Carlo

$$IE[x] = \int xp(x)dx$$

$$\approx \frac{1}{N} \sum_{i=1}^N x_i \quad \text{où } x_i \sim p(x)$$

approximation  
Monte Carlo

$$IE[f(x)] = \int f(x)p(x)dx$$

$$\approx \frac{1}{N} \sum_{i=1}^N f(x_i) \quad \text{où } x_i \sim p(x)$$

## Rappel: Espérance mathématique et estimateur Monte Carlo

$$L_D = - \underbrace{\frac{1}{N_{reel}} \sum_i \ln(D(\vec{x}_i))}_{\text{Perte images réelles}} - \underbrace{\frac{1}{N_{syn}} \sum_j \ln(1 - D(G(\vec{z}_j)))}_{\text{Perte images synthétiques}}$$

$$L_D = -IE_{\vec{x} \sim P_{reel}} [\ln(D(\vec{x}))] - IE_{\vec{z} \sim P_z} [\ln(1 - D(G(\vec{z})))]$$

## Objectif du discriminateur

### Paramètres du discriminateur

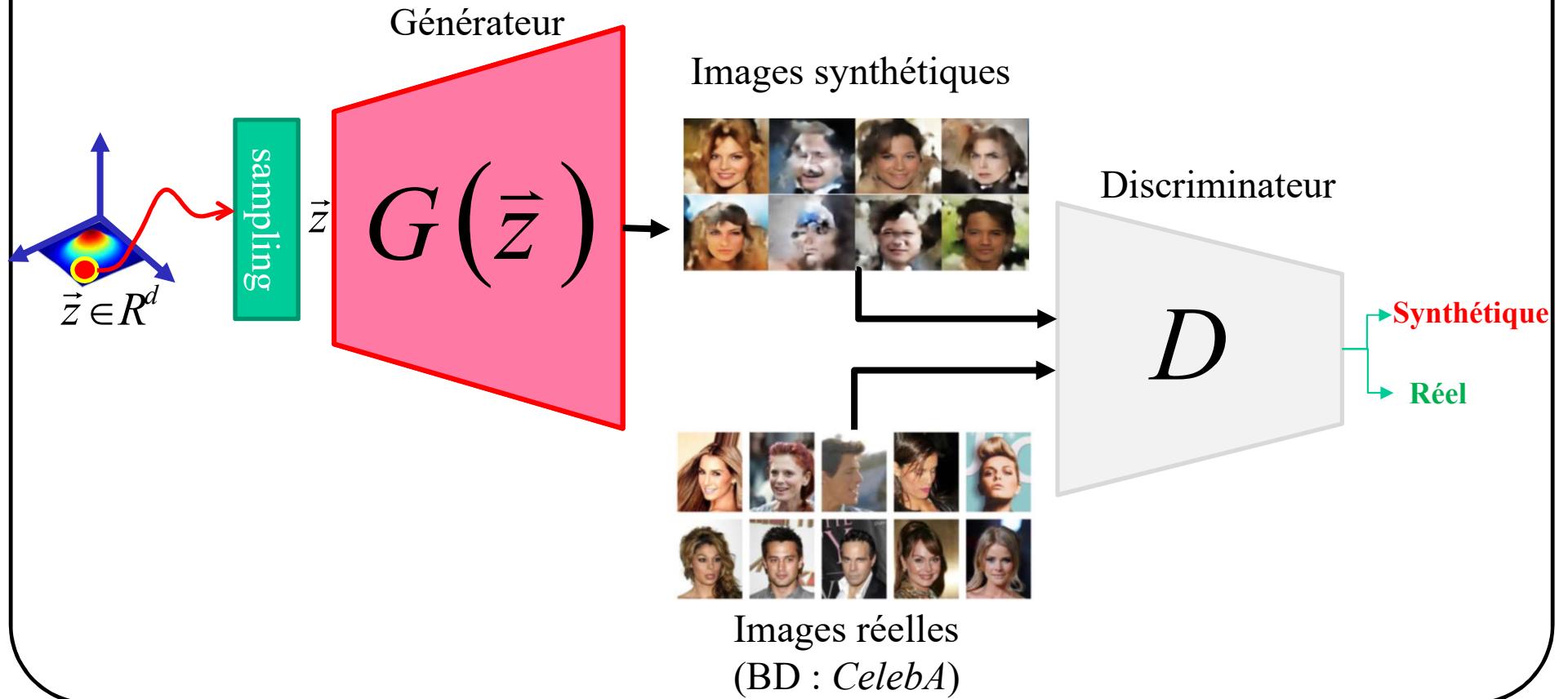
$$W_D = \arg \min_{W_D} -IE_{\vec{x} \sim P_{reel}} [\ln(D(\vec{x}))] - IE_{\vec{z} \sim P_z} [\ln(1 - D(G(\vec{z})))]$$

Ou encore, de façon équivalente

$$W_D = \arg \max_{W_D} IE_{\vec{x} \sim P_{reel}} [\ln(D(\vec{x}))] + IE_{\vec{z} \sim P_z} [\ln(1 - D(G(\vec{z})))]$$

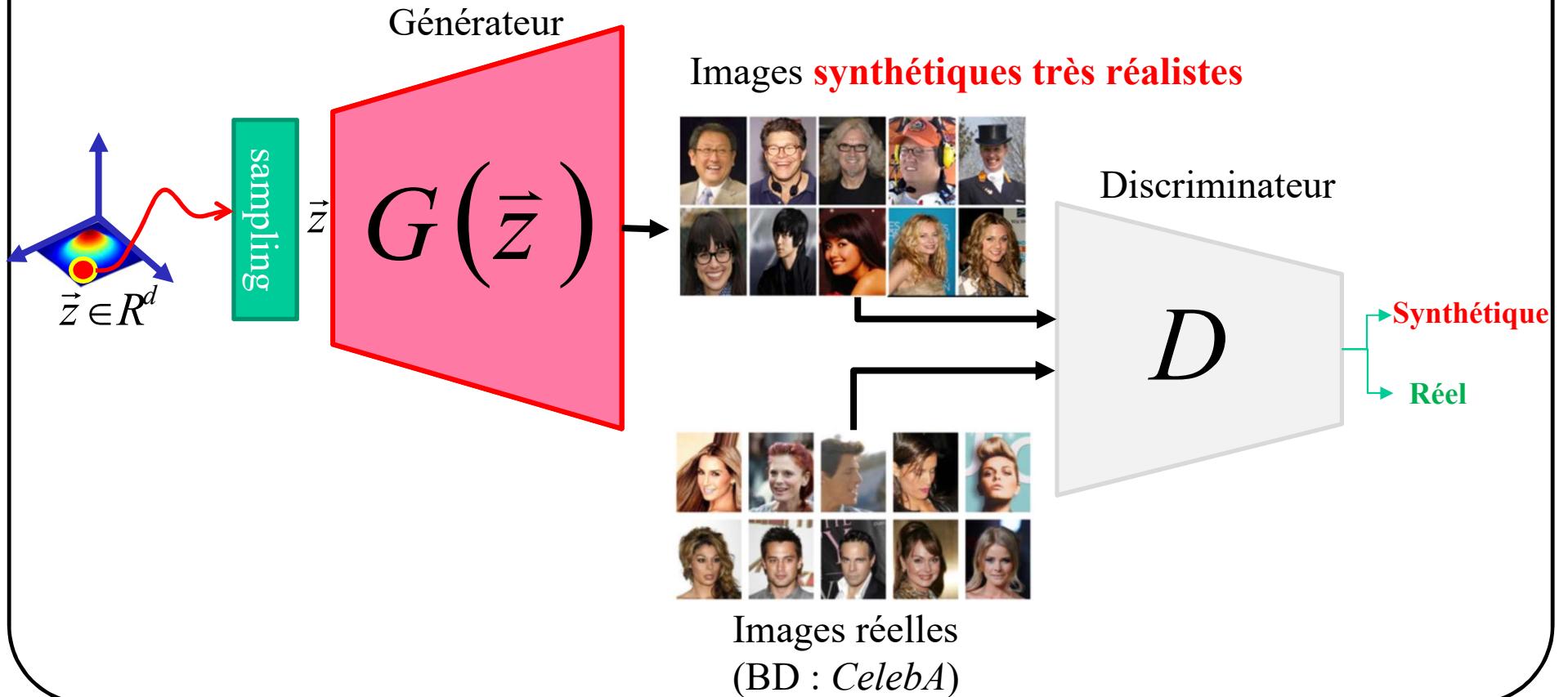
## Objectif du générateur

Produire des images aussi réalistes que celle de la BD de référence



## Objectif du générateur

S'il y parvient, le discriminateur ne pourra plus les distinguer des images réelles. La loss du discriminateur sera alors élevée.



**Objectif du discriminateur :**

bien distinguer les images réelles des images synthétiques

$$W_D = \arg \max_{W_D} IE_{\vec{x} \sim P_{real}} [\ln(D(\vec{x}))] + IE_{\vec{z} \sim P_z} [\ln(1 - D(G(\vec{z})))]$$

**Objectif du générateur :**

produire des images synthétiques indistinguables des images réelles

$$W_G = \arg \min_{W_G} IE_{\vec{z} \sim P_z} [\ln(1 - D(G(\vec{z})))]$$

# « Two player » mini-max game

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

Ian Goodfellow et al., “Generative Adversarial Nets”, NIPS 2014

## NOTE

dans les faits, on ne minimise pas cette loss

$$W_G = \arg \min_{W_G} \cancel{IE}_{\vec{z} \sim P_z} [\ln(1 - D(G(\vec{z})))]$$

on maximise plutôt celle-ci

$$W_G = \arg \max_{W_G} IE_{\vec{z} \sim P_z} [\ln(D(G(\vec{z})))]$$

**for** number of training iterations **do**

**for**  $k$  steps **do**

- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[ \log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

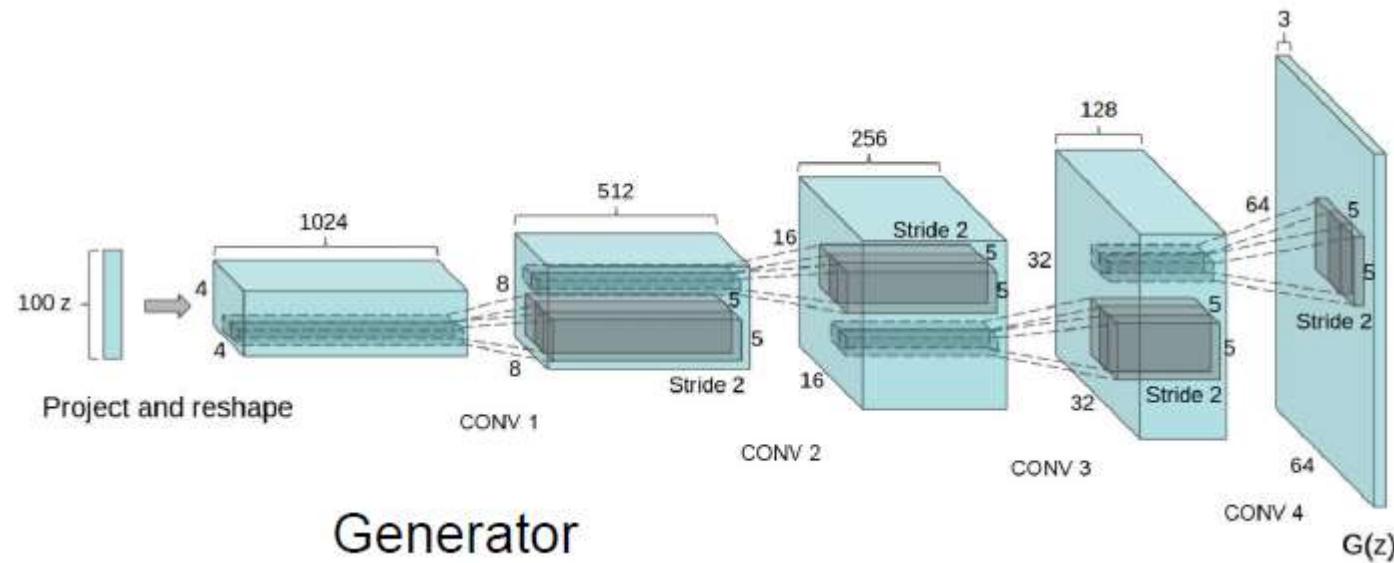
**end for**

- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

**end for**

# Deep Convolution Generative Adversarial Net (DCGAN)



Radford et al, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, ICLR 2016

# Deep Convolution Generative Adversarial Net (DCGAN)

## Recommandations discriminateur

- Conv stride>1 au lieu des couches de pooling
- ReLU partout sauf en sortie : tanh

## Recommandations générateur

- Conv transpose au lieu de upsampling
- LeakyReLU partout

## Autre recommandations

- BatchNorm partout
- Pas de FC, juste des conv

Radford et al, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, ICLR 2016

# Deep Convolution Generative Adversarial Net (DCGAN)

## Recommandations discriminateur

- Conv2d
- LeakyReLU partout

<https://github.com/soumith/ganhacks>

## Recom

- Conv2d
- LeakyReLU partout

## Autre recommandations

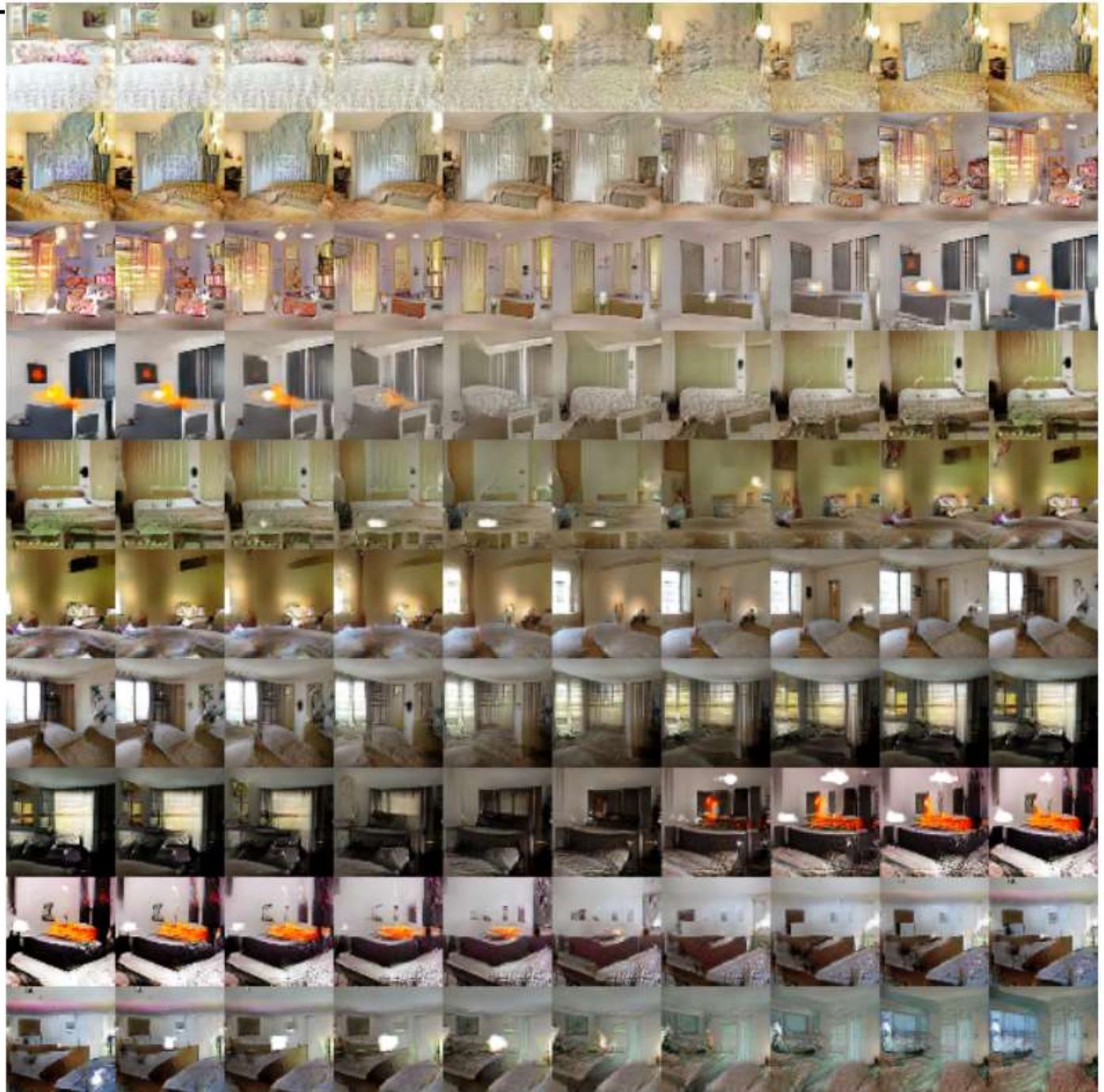
- BatchNorm partout
- Pas de FC, juste des conv

Radford et al, “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, ICLR 2016

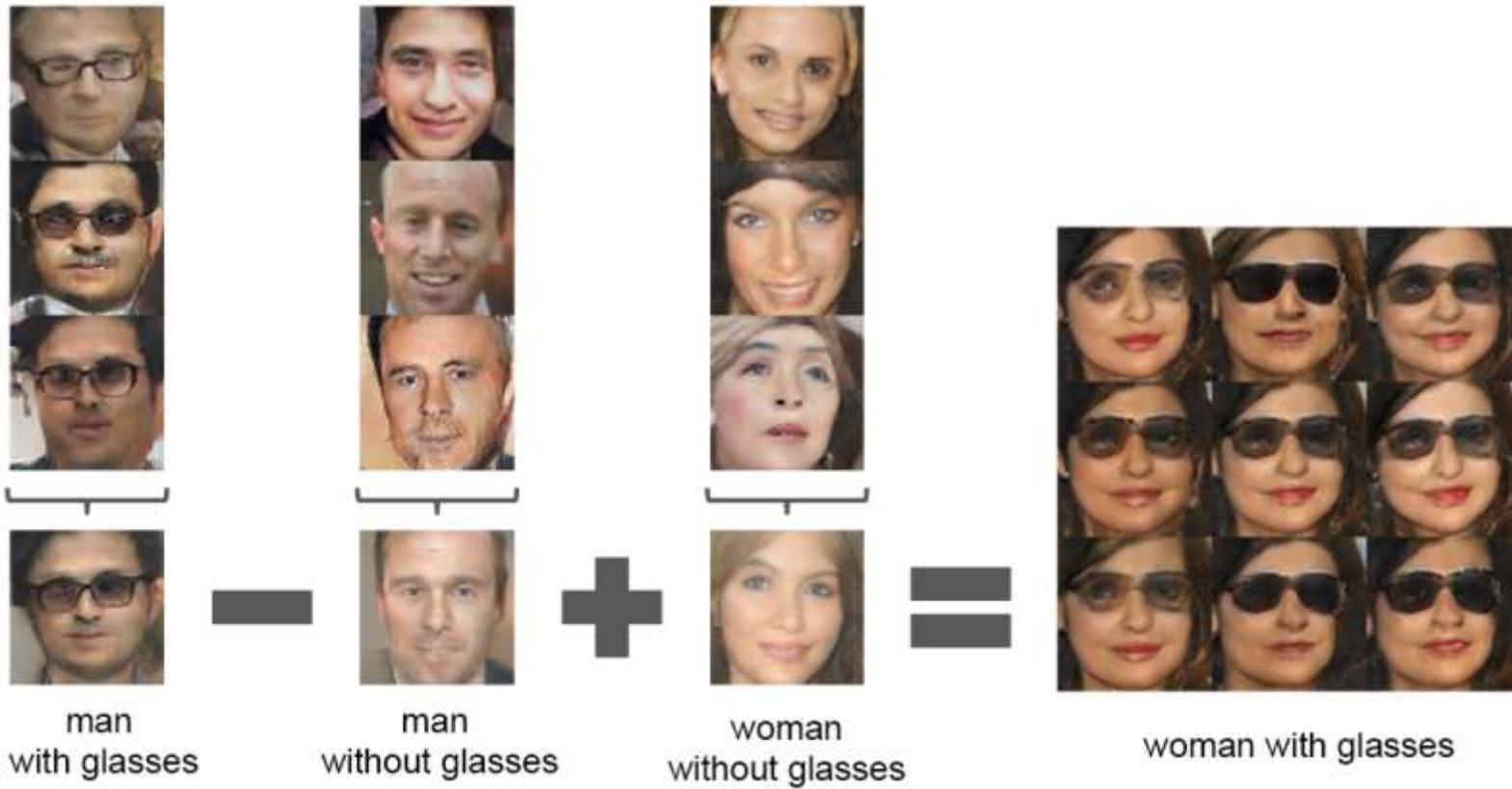
# Deep Convolution Generative Adversarial Net (DCGAN)



Interpolation  
entre 9 vecteurs  
latents aléatoires



# *“Vector arithmetic for visual concepts”*



# Problèmes d'instabilité

- Si discriminateur et générateur et n'apprennent pas ensemble:
  - disparition des gradients
  - effondrement des modes
  - on ne peut générer d'images à haute résolution
- Plusieurs solutions proposées:
  - *Wasserstein GAN* (*utilise “earth mover distance”*)
  - *Least Squares GAN* (*utilise distance d'erreur quadratique*)
  - *Progressive GAN*
  - ....

# LS GAN

## Problème des GANs de base

“**sigmoïde** de sortie” **oublie** les exemples correctement classifiés et loin du plan de séparation

$$\max_D V(D) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

$$\min_G V(G) = \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \text{ or } \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} - [\log D(G(\mathbf{z}))]$$

# LS GAN

## Problème des GANs de base

“sigmoid”

séparation

Le discriminateur ne s'entraîne plus lorsque les images synthétiques sont très différentes des images réelles

$$\min_G V(G) = \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \text{ or } \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} - [\log D(G(\mathbf{z}))]$$

# LS GAN

Quand on utilise **l'erreur quadratique**, même les exemples « trop bien classifiés » contribuent aux gradients du générateur. Le but est de rapprocher les images synthétiques des images réelles

**Pour LS GAN, la sortie du réseau n'est plus une sigmoïde**

$$\begin{aligned}\min_D V_{\text{LSGAN}}(D) &= \frac{1}{2} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [(D(\mathbf{x}) - 1)^2] + \frac{1}{2} \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [(D(G(\mathbf{z})))^2] \\ \min_G V_{\text{LSGAN}}(G) &= \frac{1}{2} \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [(D(G(\mathbf{z}))) - 1]^2,\end{aligned}$$

“Least Squares GAN” Mao et al. ICCV’17

# LS GAN



(a) Generated by LSGANs.



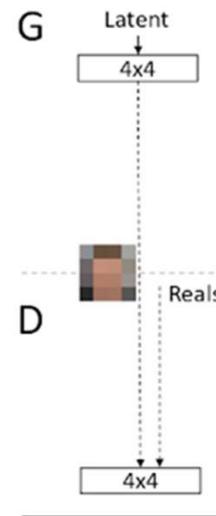
(b) Generated by DCGANs (Reported in [11]).

“Least Squares GAN” Mao et al. ICCV’17

# progressive GAN

On veut générer des images à **haute résolution**

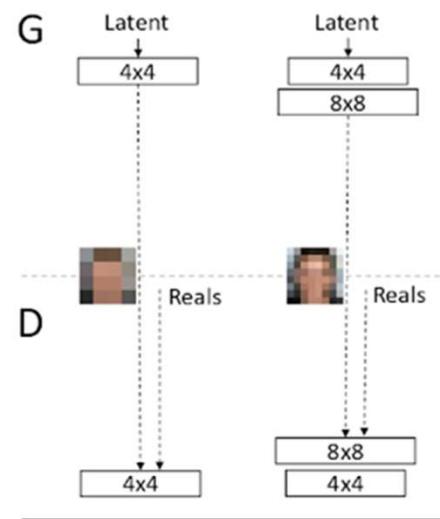
On commence avec des images de **faible résolution : 4x4 pixels**



*“Progressive GAN” Karras et al. ICLR’18*

# progressive GAN

Et progressivement, on augmente la résolution de l'image

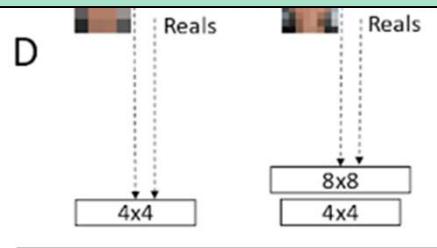


*“Progressive GAN” Karras et al. ICLR’18*

# progressive GAN

Et progressivement on augmente la résolution de l'image

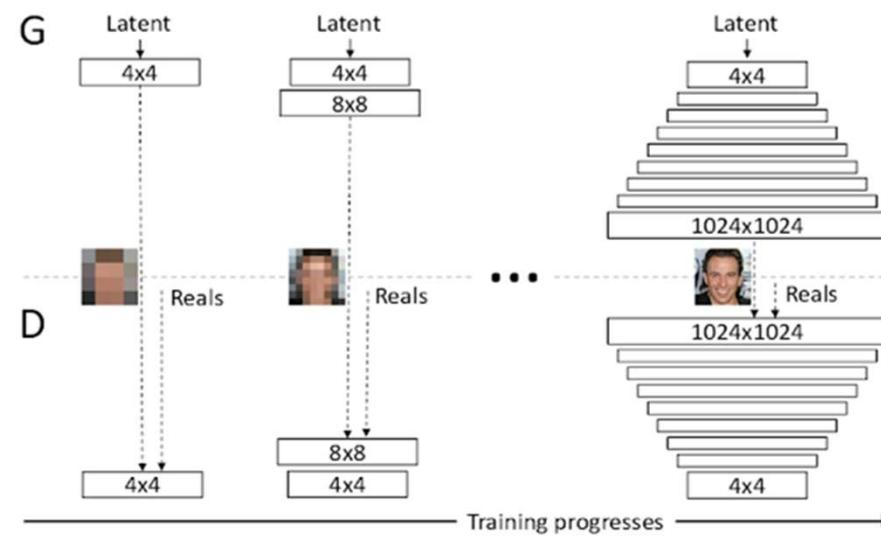
*“Progressive Growing GAN requires that the capacity of both the generator and discriminator model be expanded by adding layers during the training process”*



*“Progressive GAN” Karras et al. ICLR’18*

# progressive GAN

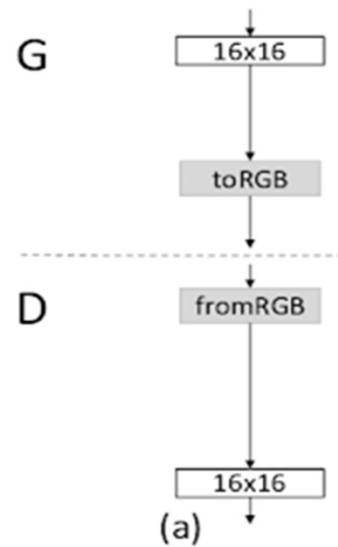
Et progressivement, chaque couche qu'on ajoute vient bonifier la couche précédente : cela se fait à l'aide d'une **opération « résiduelle »**.



*“Progressive GAN” Karras et al. ICLR’18*

# Ajout de couches

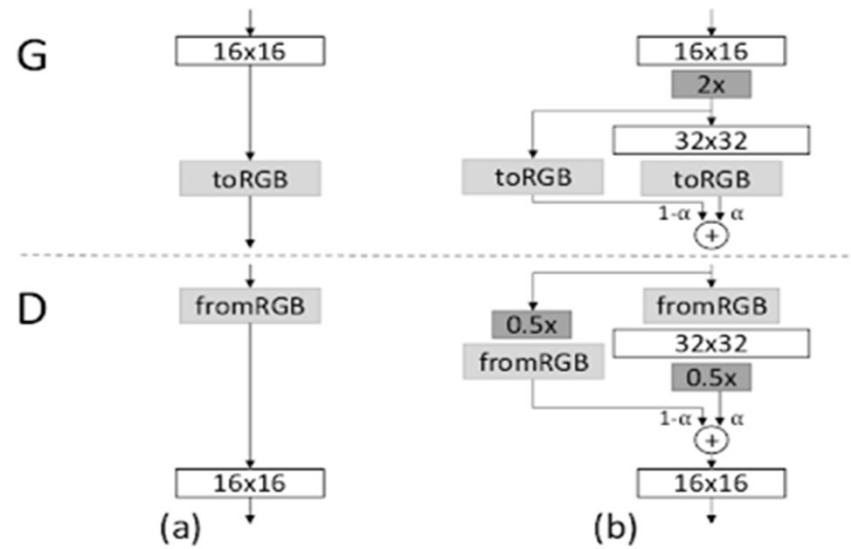
Lorsque **l'entraînement** d'une couche de résolution RxR (ici 16x16) est **terminé**...



“*Progressive GAN*” Karras et al. ICLR’18

# Ajout de couches

... On **ajoute une nouvelle couche** de résolution 2Rx2D (ici 32x32) au générateur ET au discriminateur.

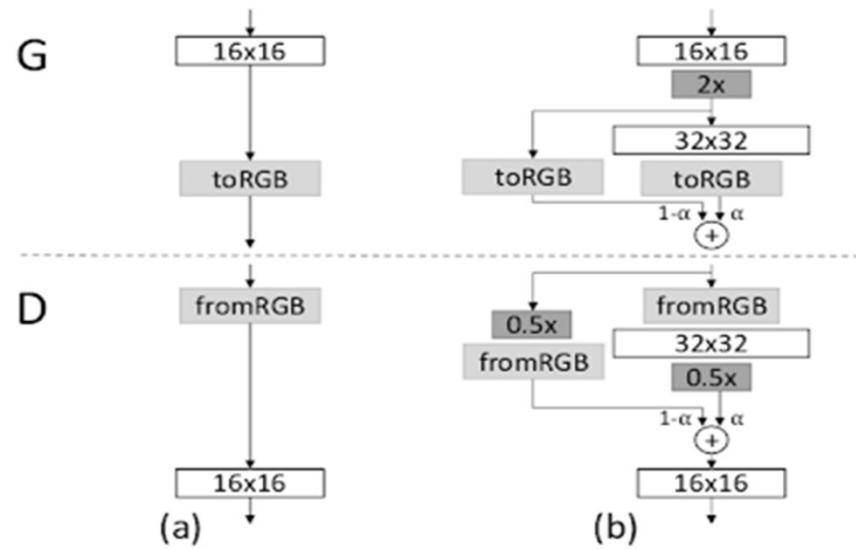


“Progressive GAN” Karras et al. ICLR’18

# Ajout de couches

... mais pour éviter un choc, on ajoute une **composante résiduelle** comprenant un facteur  $\alpha$

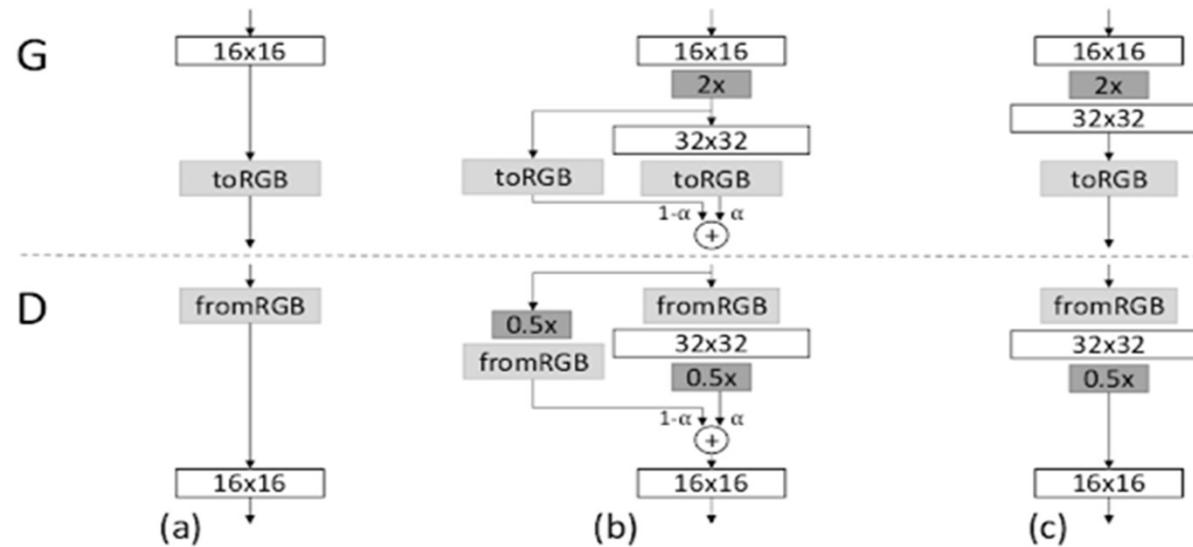
**Au début de l'entraînement,  $\alpha=0$  et progressivement  $\alpha$  augmente pour atteindre  $\alpha=1$  à la fin**



“Progressive GAN” Karras et al. ICLR’18

# Ajout de couches

Lorsque l'entraînement est terminé, on enlève la composante résiduelle.



“Progressive GAN” Karras et al. ICLR’18

# “Progressive GAN” Karras et al. ICLR’18

<b>Generator</b>	Act.	Output shape	Params		<b>Discriminator</b>	Act.	Output shape	Params
Latent vector	–	512 × 1 × 1	–		Input image	–	3 × 1024 × 1024	–
Conv 4 × 4	LReLU	512 × 4 × 4	4.2M		Conv 1 × 1	LReLU	16 × 1024 × 1024	64
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M		Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Upsample	–	512 × 8 × 8	–		Conv 3 × 3	LReLU	32 × 1024 × 1024	4.6k
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M		Downsample	–	32 × 512 × 512	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M		Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Upsample	–	512 × 16 × 16	–		Conv 3 × 3	LReLU	64 × 512 × 512	18k
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M		Downsample	–	64 × 256 × 256	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M		Conv 3 × 3	LReLU	64 × 256 × 256	37k
Upsample	–	512 × 32 × 32	–		Conv 3 × 3	LReLU	128 × 256 × 256	74k
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M		Downsample	–	128 × 128 × 128	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M		Conv 3 × 3	LReLU	128 × 128 × 128	148k
Upsample	–	512 × 64 × 64	–		Conv 3 × 3	LReLU	256 × 128 × 128	295k
Conv 3 × 3	LReLU	256 × 64 × 64	1.2M		Downsample	–	256 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	590k		Conv 3 × 3	LReLU	256 × 64 × 64	590k
Upsample	–	256 × 128 × 128	–		Conv 3 × 3	LReLU	512 × 64 × 64	1.2M
Conv 3 × 3	LReLU	128 × 128 × 128	295k		Downsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	128 × 128 × 128	148k		Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Upsample	–	128 × 256 × 256	–		Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	64 × 256 × 256	74k		Downsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	64 × 256 × 256	37k		Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Upsample	–	64 × 512 × 512	–		Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	32 × 512 × 512	18k		Downsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k		Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Upsample	–	32 × 1024 × 1024	–		Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	16 × 1024 × 1024	4.6k		Downsample	–	512 × 4 × 4	–
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k		Minibatch stddev	–	513 × 4 × 4	–
Conv 1 × 1	linear	3 × 1024 × 1024	51		Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Total trainable parameters			<b>23.1M</b>		Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
					Fully-connected	linear	1 × 1 × 1	513
					Total trainable parameters			<b>23.1M</b>

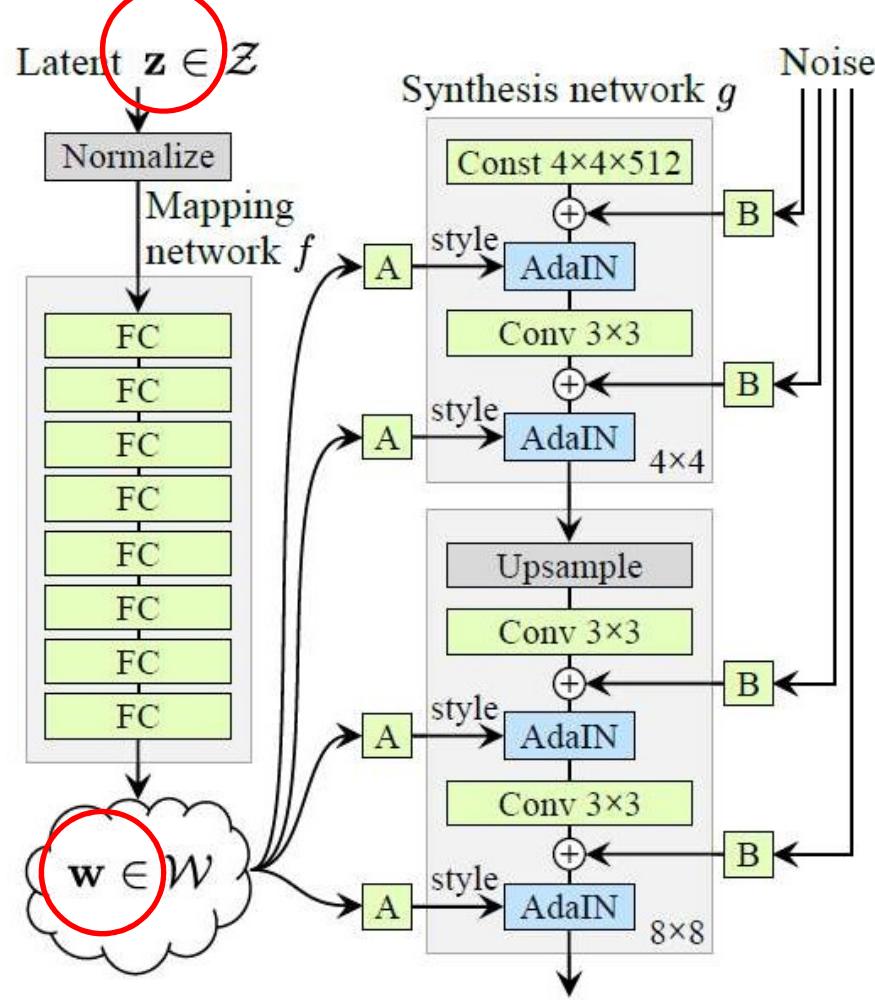
Table 2: Generator and discriminator that we use with CELEBA-HQ to generate  $1024 \times 1024$  images.

*“Progressive GAN” Karras et al. ICLR’18*



<https://youtu.be/XOxxPcy5Gr4>

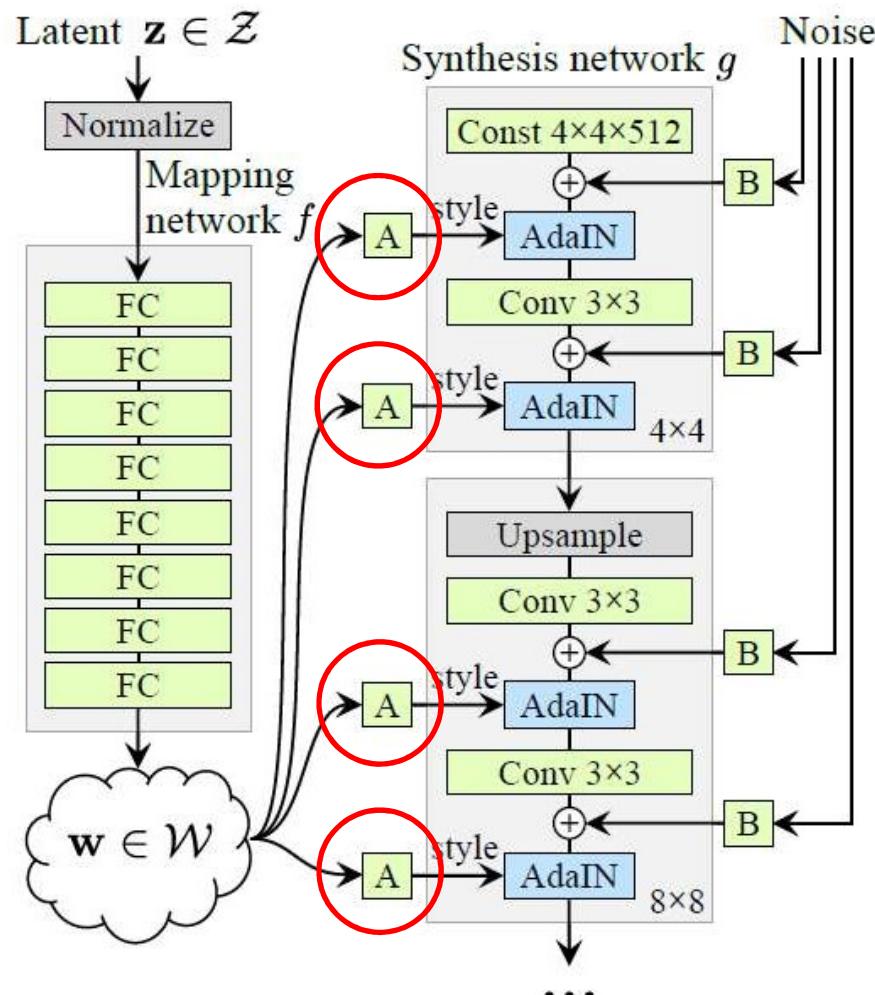
# Style GAN



(b) Style-based generator

Le Générateur reçoit une version modifiée “ $w$ ” par 8 couches FC du vecteur latent “ $z$ ”

# Style GAN

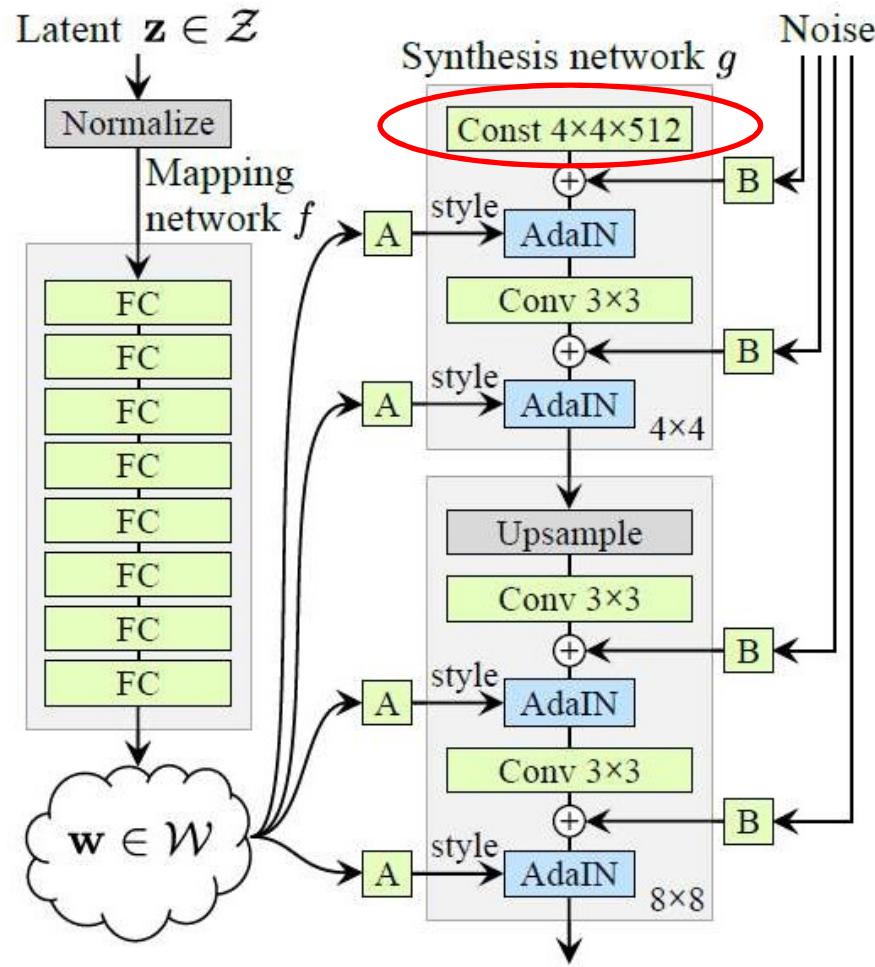


### (b) Style-based generator

Le Générateur reçoit une version modifiée “w” par 8 couches FC du vecteur latent “z”

Et ce, à **chaque couche** du réseau

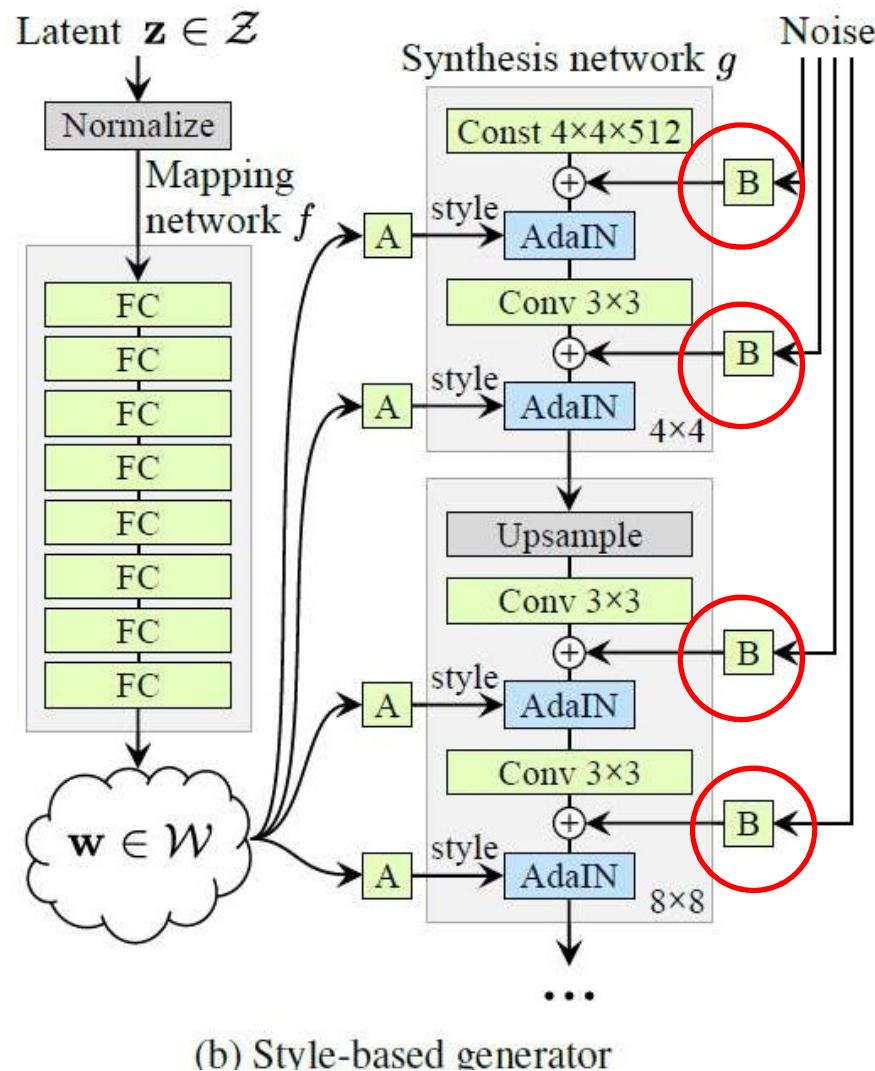
# Style GAN



(b) Style-based generator

L'entrée du réseau est un **tenseur constant** appris par entraînement

# Style GAN



Du **bruit** est multiplié à chaque carte d'activation de **chaque couche** du réseau

# Effet du bruit

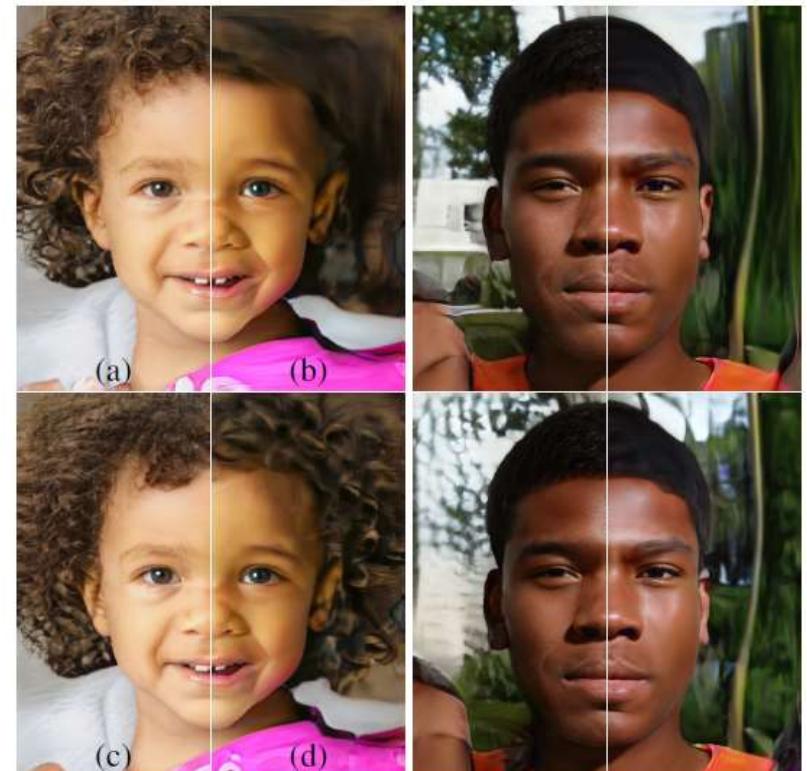
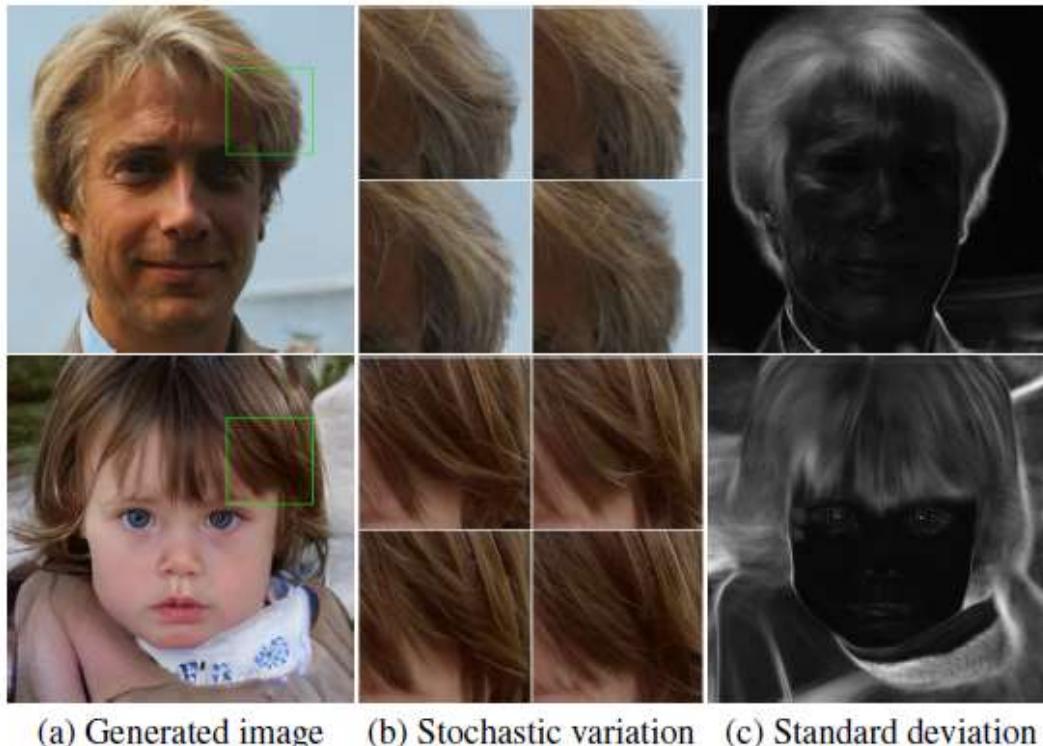
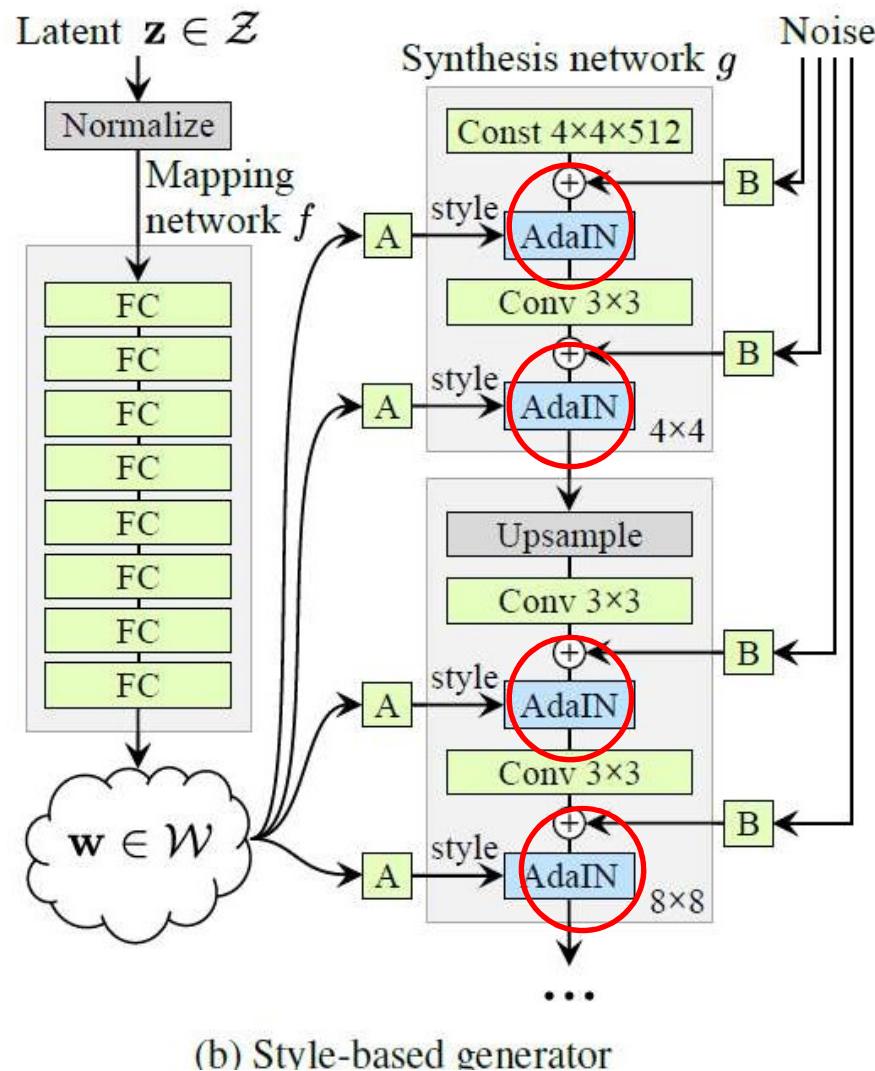


Figure 5. Effect of noise inputs at different layers of our generator. (a) Noise is applied to all layers. (b) No noise. (c) Noise in fine layers only ( $64^2 - 1024^2$ ). (d) Noise in coarse layers only ( $4^2 - 32^2$ ). We can see that the artificial omission of noise leads to featureless “painterly” look. Coarse noise causes large-scale curling of hair and appearance of larger background features, while the fine noise brings out the finer curls of hair, finer background detail, and skin pores.

# Style GAN



AdaIN: adaptive instance normalization

$$AdaIN(x, y) = \sigma(y) \frac{x - \mu(x)}{\sigma(x)} + \mu(y)$$

Comme du batchNorm, mais dont les 2 opérateurs affines sont fournis par  $\mathbf{w}$

# Style GAN

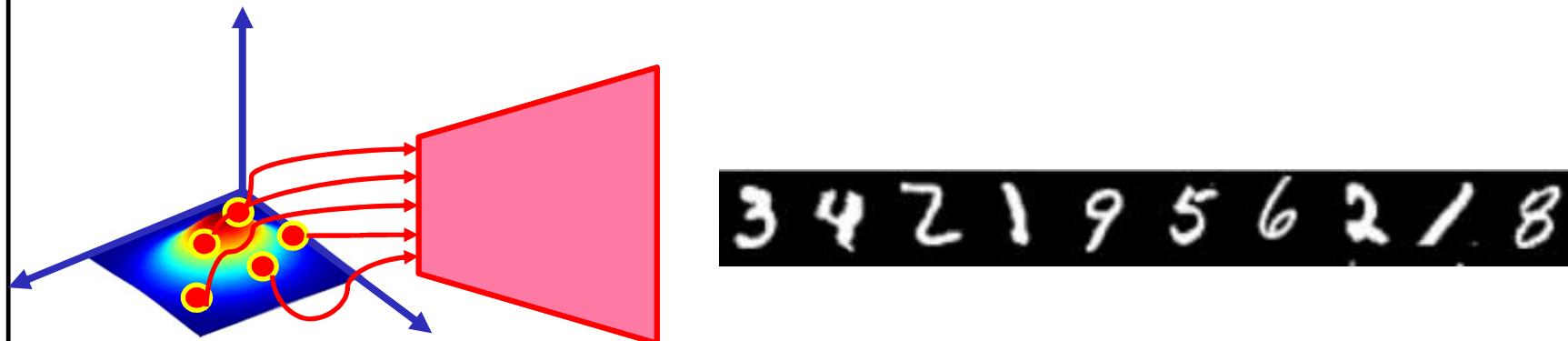
Entraînement progressif comme pour **progressive GAN**

# Style GAN



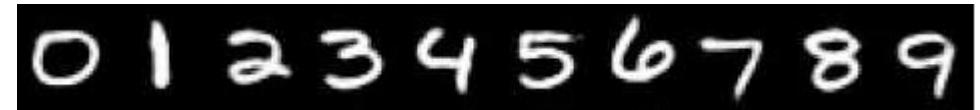
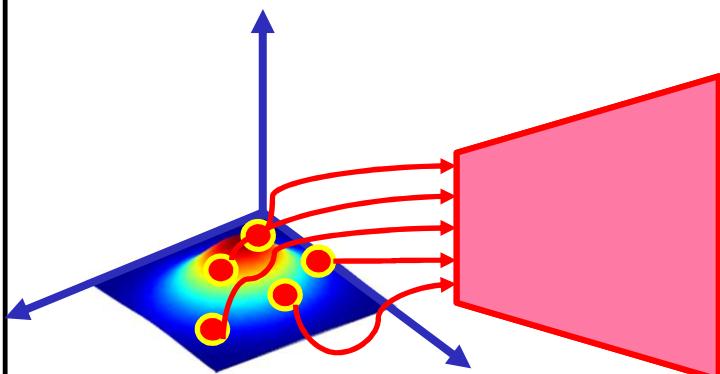
# Défi avec les GAN

Soit un GAN entraîné sur MNIST, si je décode **10 vecteurs latents pris au hasard**, j'aurai les images de **10 caractères aléatoires**.



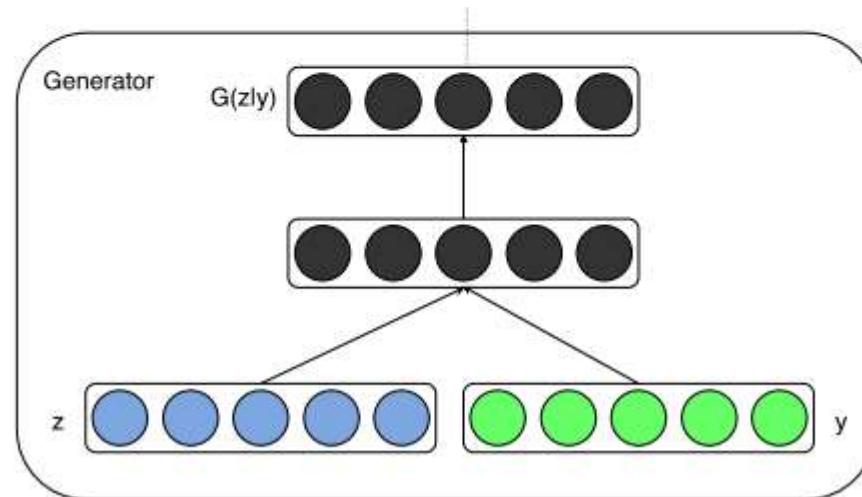
# Défi avec les GAN

**Question:** comment générer des images de catégories prédéterminées? Ex. comment sélectionner 10 vecteurs latent afin de produire la séquence de caractères : 0,1,2,3,4,5,6,7,8,9?



# Gan conditionnel

L'idée est d'encoder un vecteur latent  $\vec{z}$  ainsi qu'un **vecteur de classe** « one-hot »  $\vec{y}$

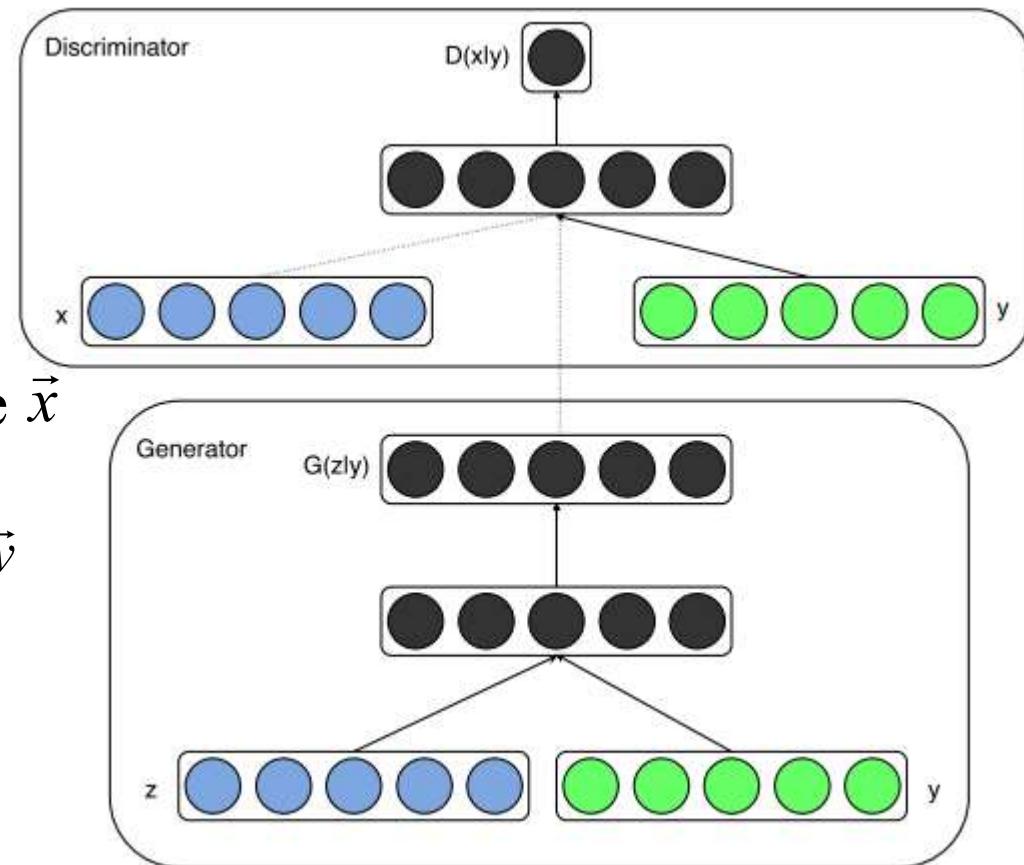


Mirza, Mehdi & Osindero, Simon. (2014). Conditional Generative Adversarial Nets. arXiv:1411.1784v1

# Gan conditionnel

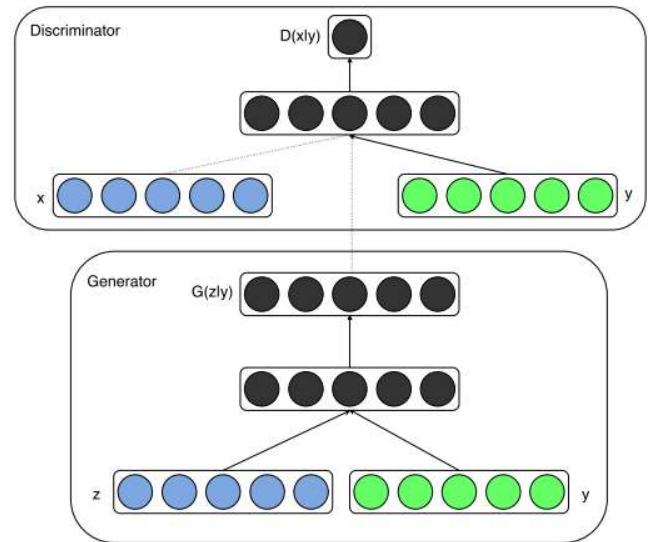
Et de discriminer une image  $\vec{x}$

avec **le même « one-hot »**  $\vec{y}$



Mirza, Mehdi & Osindero, Simon. (2014). Conditional Generative Adversarial Nets. arXiv:1411.1784v1

# Gan conditionnel



## GAN de base

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

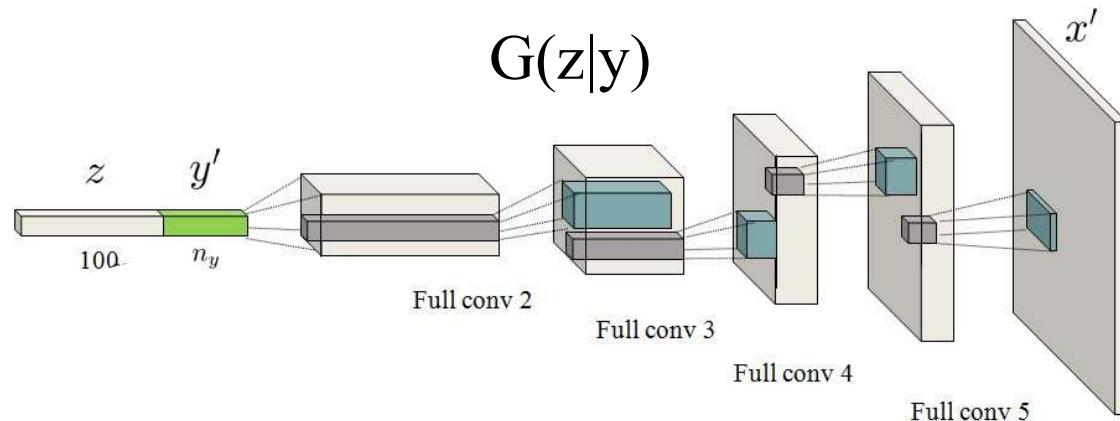
## GAN conditionnel

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x}|\mathbf{y})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z}|\mathbf{y})))].$$



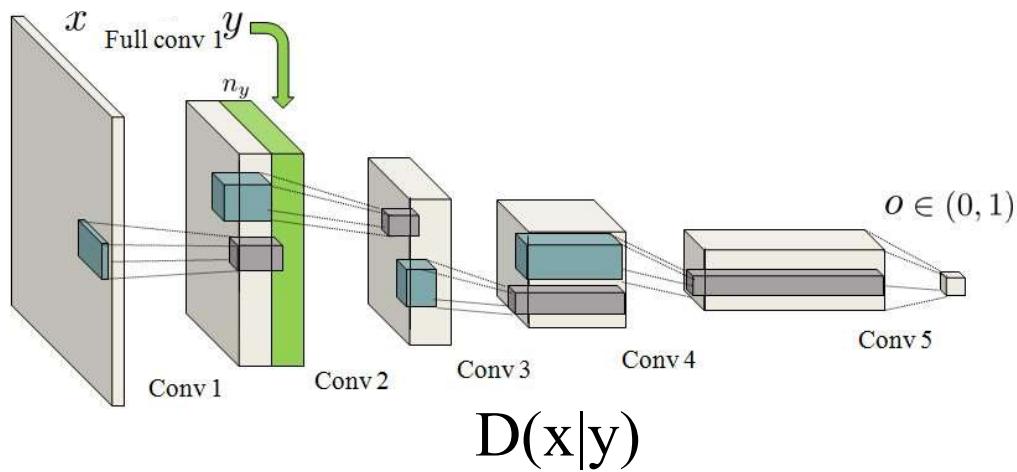
Mirza, Mehdi & Osindero, Simon. (2014). Conditional Generative Adversarial Nets. arXiv:1411.1784v1

# Version convolutionnelle



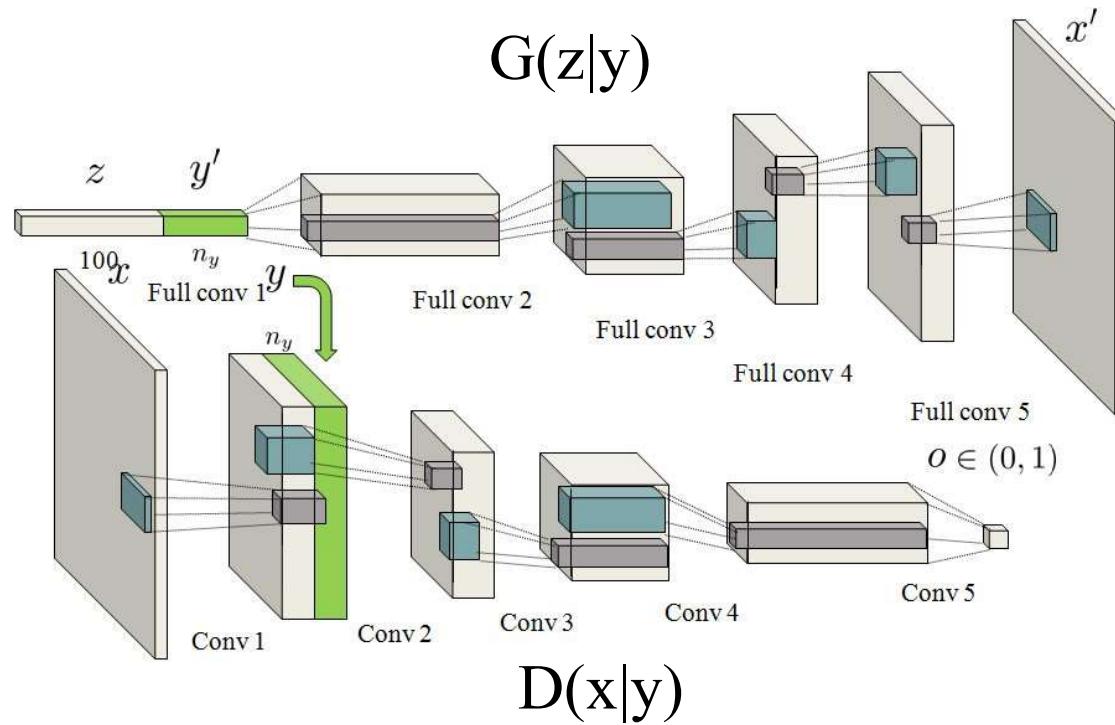
<https://medium.com/@sam.maddrellmander/conditional-dcgan-in-tensorflow-336f8b03b7b6>

# Version convolutionnelle



<https://medium.com/@sam.maddrellmander/conditional-dcgan-in-tensorflow-336f8b03b7b6>

# Version convolutionnelle



<https://medium.com/@sam.maddrellmander/conditional-dcgan-in-tensorflow-336f8b03b7b6>



“*t-shirt*”, “*pants*”, “*pullover*”, “*dress*”, “*coat*”, “*sandal*”, “*shirt*”, “*sneaker*”, “*bag*”, “*ankle boot*”.

Code pytorch pour plus de 30 modèles  
de GANs

<https://github.com/eriklindernoren/PyTorch-GAN>

Belle vidéo sur les GANs montrant comment on peut manipuler l'espace latent et comment certains les utilisent pour produire des « *deep fake* »

<https://www.youtube.com/watch?v=dCKbRCUyop8>