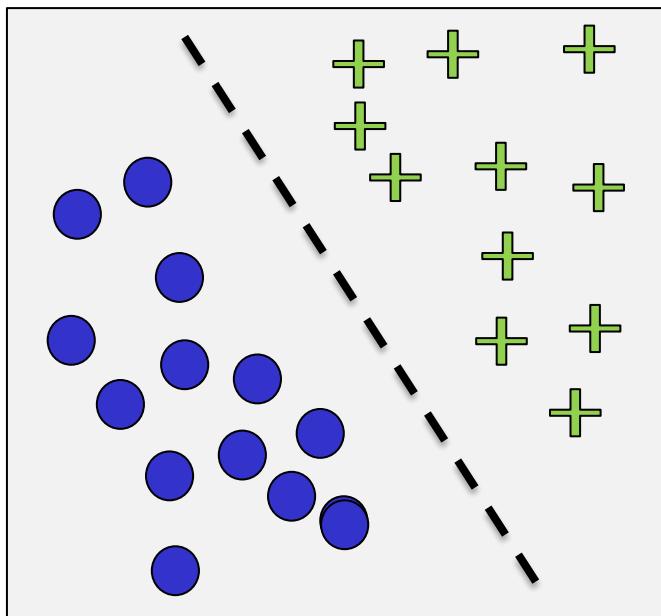


Réseaux de neurones
IFT 780

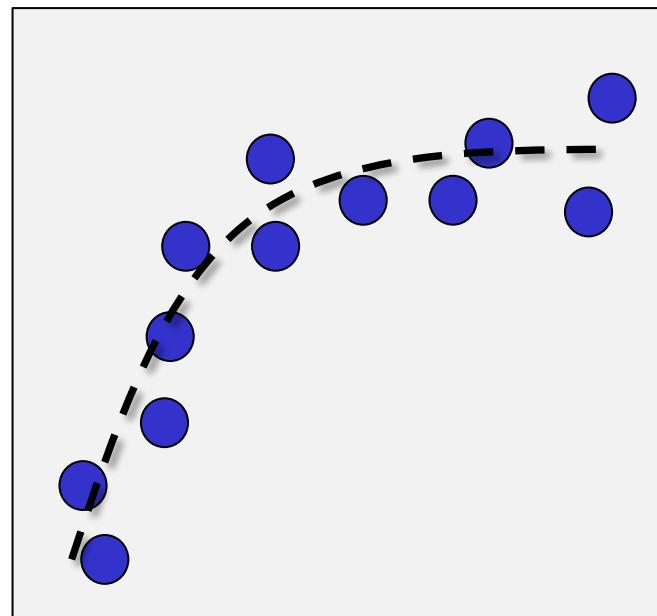
Modèles génératifs
Par
Pierre-Marc Jodoin

Jusqu'à présent : apprentissage supervisé

Classification



Régression



Jusqu'à présent : apprentissage supervisé

Segmentation



Description



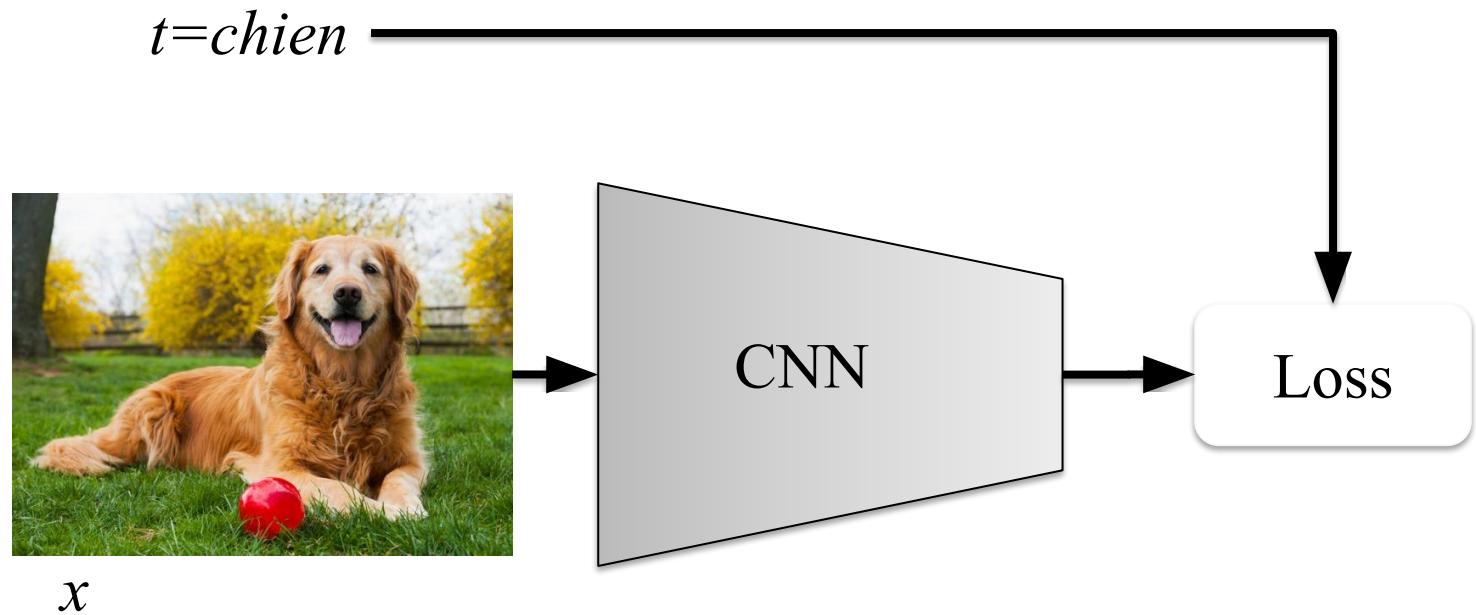
an elephant standing in a grassy field with
trees in the background

Apprentissage supervisé

Deux grandes familles d'applications

- **Classification** : la cible est un indice de classe $t \in \{1, \dots, K\}$
 - Exemple : reconnaissance de caractères
 - ✓ 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
 - ✓ 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

$$D = \{(x_1, t_1), (x_2, t_2), \dots, (x_N, t_N)\}$$

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

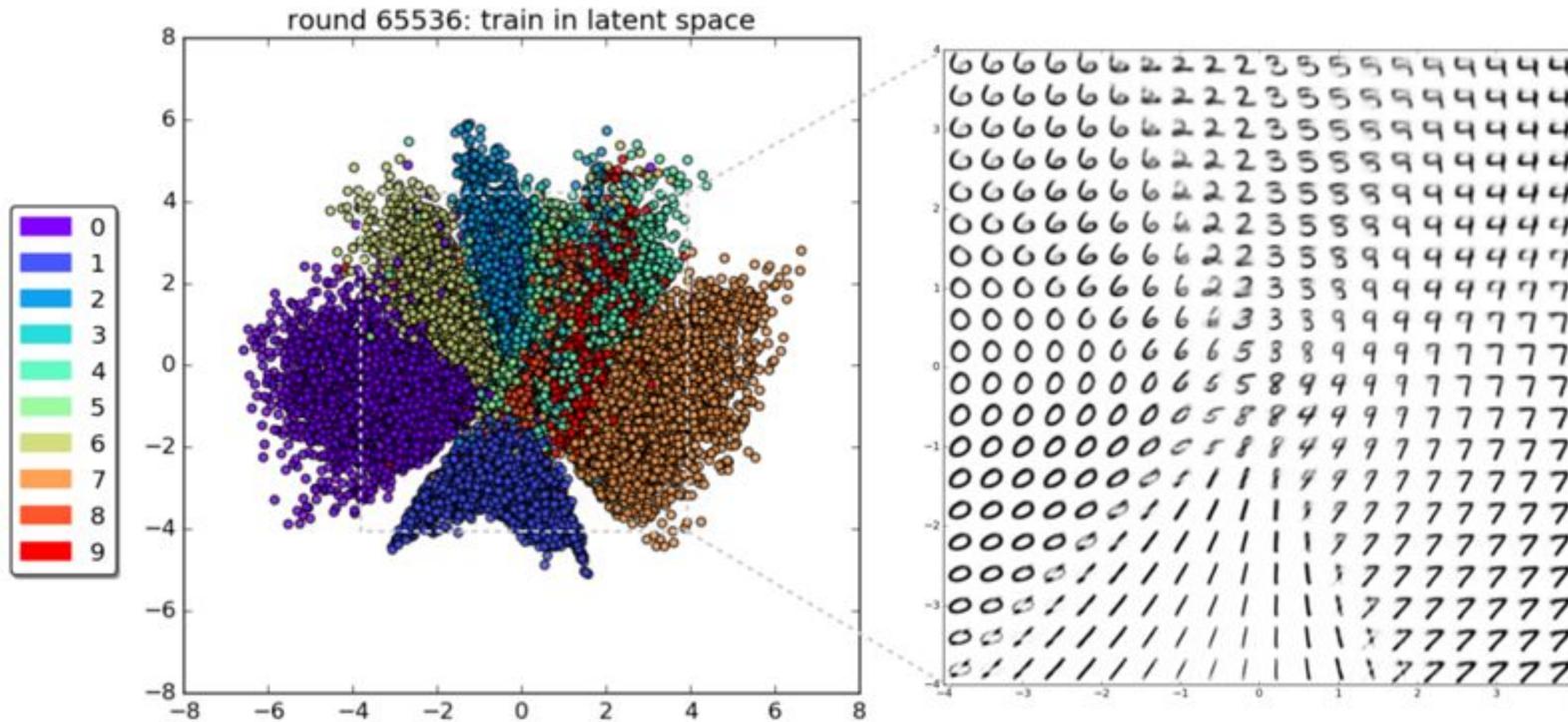
$$D = \{x_1, x_2, \dots, x_N\}$$

Apprentissage non supervisé

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

Applications : clustering (p.e. k-means), visualization, comprehension, etc.

Exemple : visualization de la distribution des images MNIST



Modèles discriminatifs vs. génératifs

$p(t|x)$

probabilité de la classe t
en sachant x

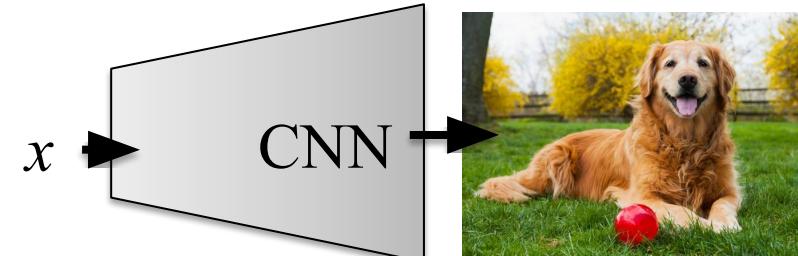
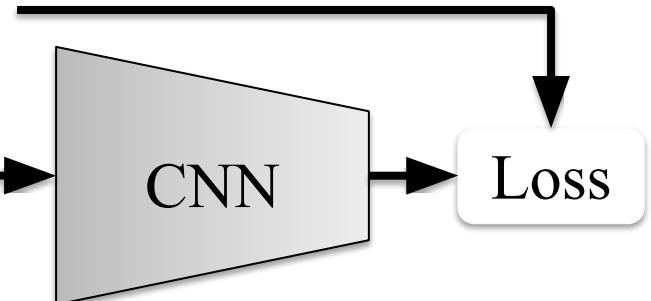
$p(x)$

probabilité de x

$t=chien$



x



Modèles discriminatifs vs. génératifs

$p(t|x) \rightarrow p(\text{chien} | \text{dog})$

probabilité de la classe t
en sachant x

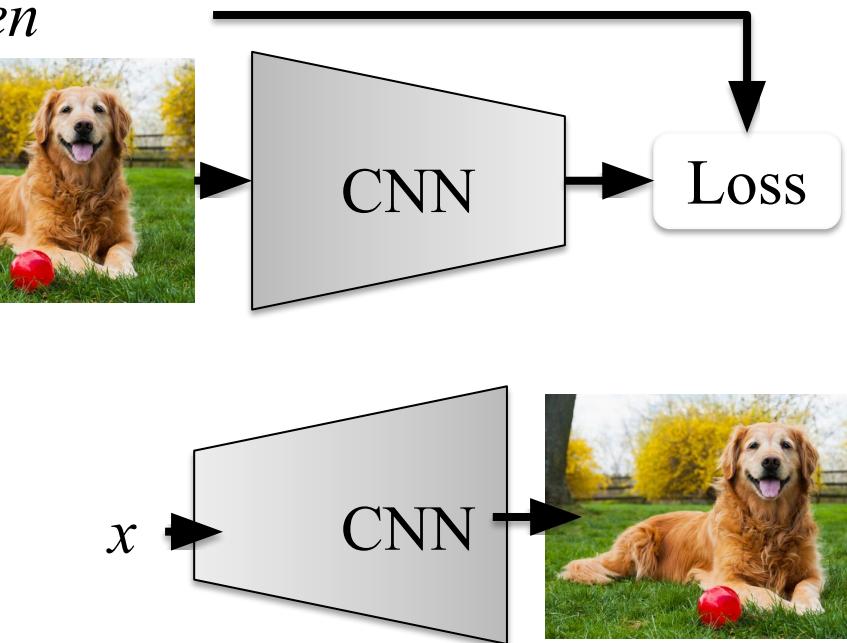
$p(x) \rightarrow p(\text{dog})$

probabilité de x

$t = \text{chien}$



x

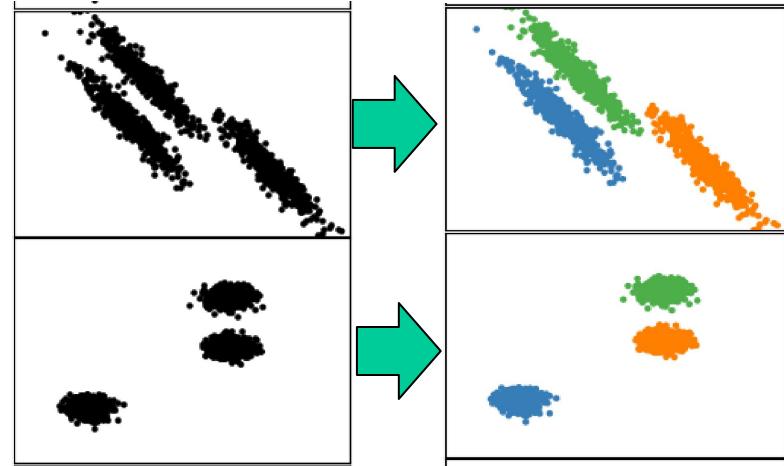
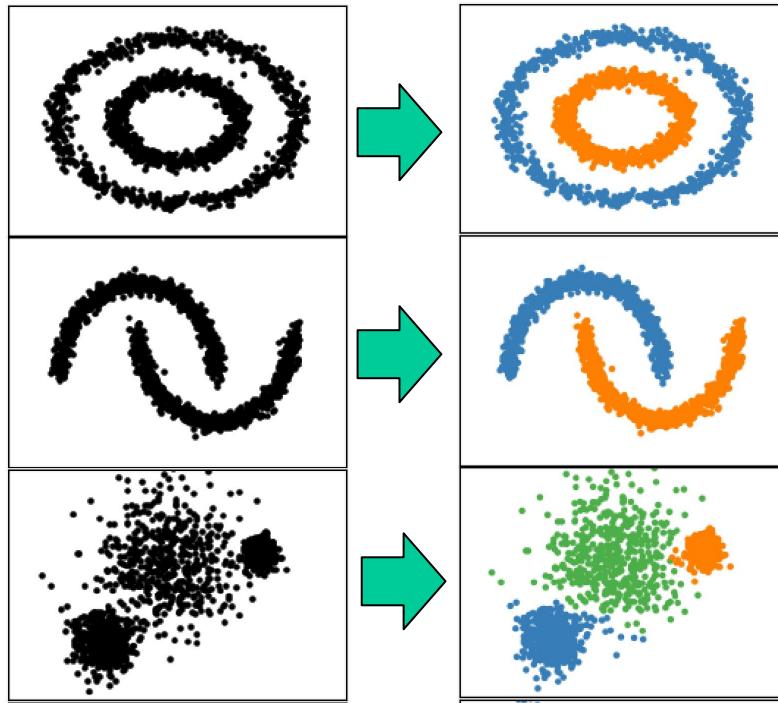


Apprentissage non supervisé

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 $p(x, y)$, $p(x | y)$, $p(y)$
que $p(x)$

Apprentissage non supervisé

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

Rappel:

- si x_1 et x_2 sont des variables aléatoires indépendantes:
 $p(x_1, x_2) = p(x_1)(x_2)$ et $p(x_1|x_2) = p(x_1)$
- si x_1 est dépendant de x_2 , mais pas l'inverse:
 $p(x_1|x_2) \neq p(x_1)$ MAIS $p(x_2|x_1) = p(x_2)$
 $p(x_1, x_2) = p(x_1|x_2)p(x_2)$

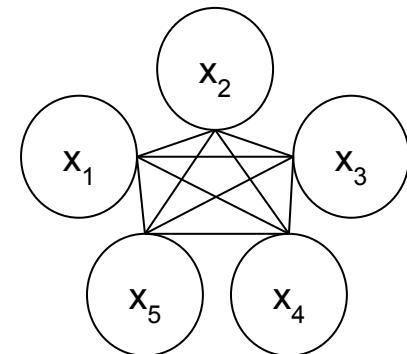
Exemple: x_1 = lire dans le jardin
 x_2 = il pleut dehors

Apprentissage non supervisé

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

Rappel:

si $\vec{x} = (x_1, x_2, x_3, x_4, x_5)$ sont des variables aléatoires dépendantes



$p(\vec{x}) = p(x_1, x_2, x_3, x_4, x_5)$, compliqué à calculer car il faut considérer chaque combinaison

Règle en chaîne des probabilités: $p(x_1, x_2, \dots, x_n) = \prod_{i=1}^n p(x_i | x_1, \dots, x_{i-1})$

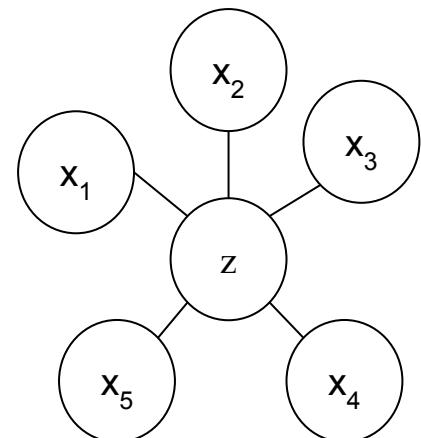
Pas si mal pour 5 variables, mais dans une image, chaque pixel est une variable !

Apprentissage non supervisé

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

Pour simplifier le problème on présuppose que les variables x_i dépendent d'une variable latente z et sont indépendantes entre elles

$$\begin{aligned} p(\vec{x}, z) &= p(x_1, x_2, x_3, x_4, x_5, z) \\ &= p(z)p(x_1|z)\dots p(x_5|z) \\ &= p(z) \prod_{i=1}^5 p(x_i|z) \\ &= p(z)p(\vec{x}|z) \end{aligned}$$



On peut toujours présumer que z existe !

$$\begin{aligned} p(\vec{x}) &= \int p(\vec{x}, z) dz \quad \text{<= marginalisation} \\ &= \int p(\vec{x}|z)p(z)dz \end{aligned}$$

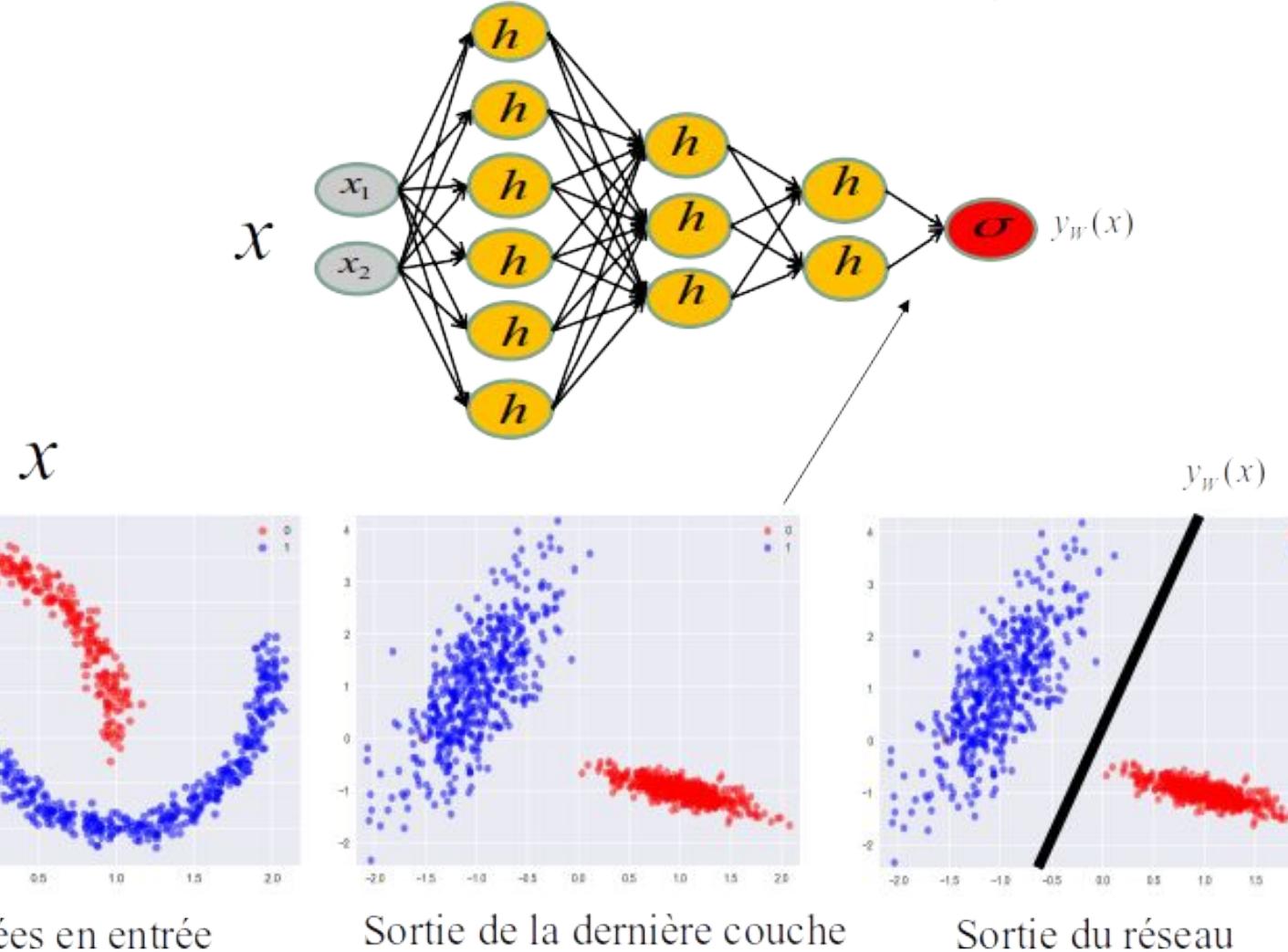
En apprentissage non-supervisé

nous nous appuierons sur

2 propriétés des réseaux de neurones

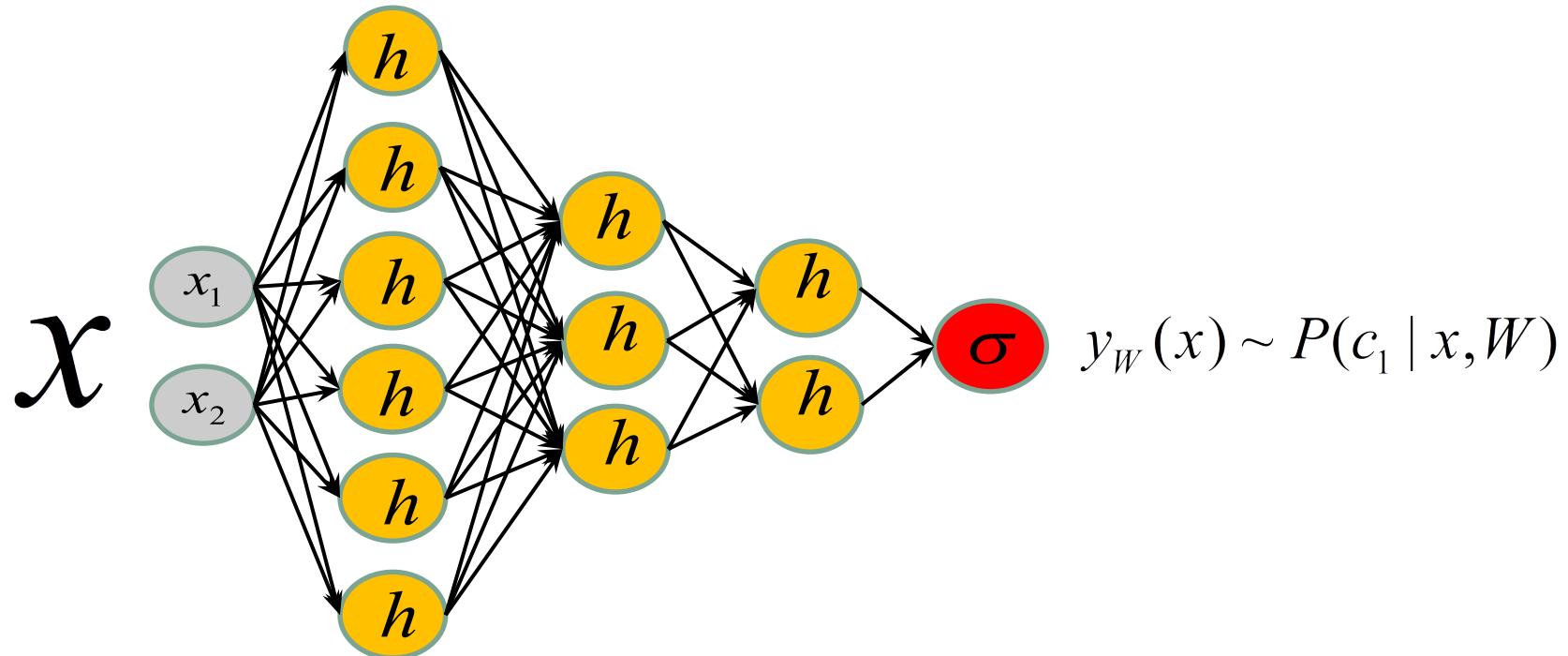
Propriété 1

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**).



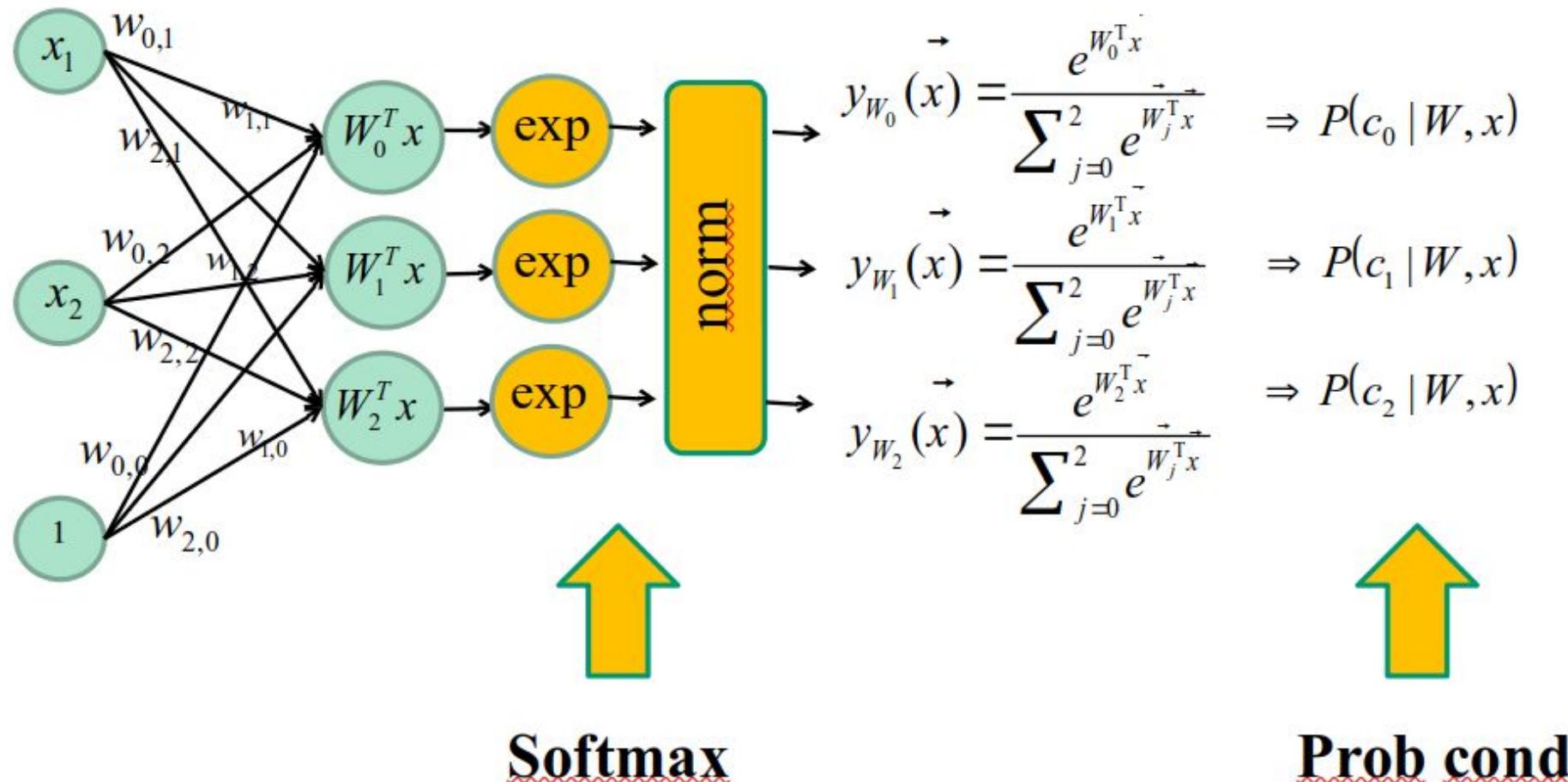
Propriété 2

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



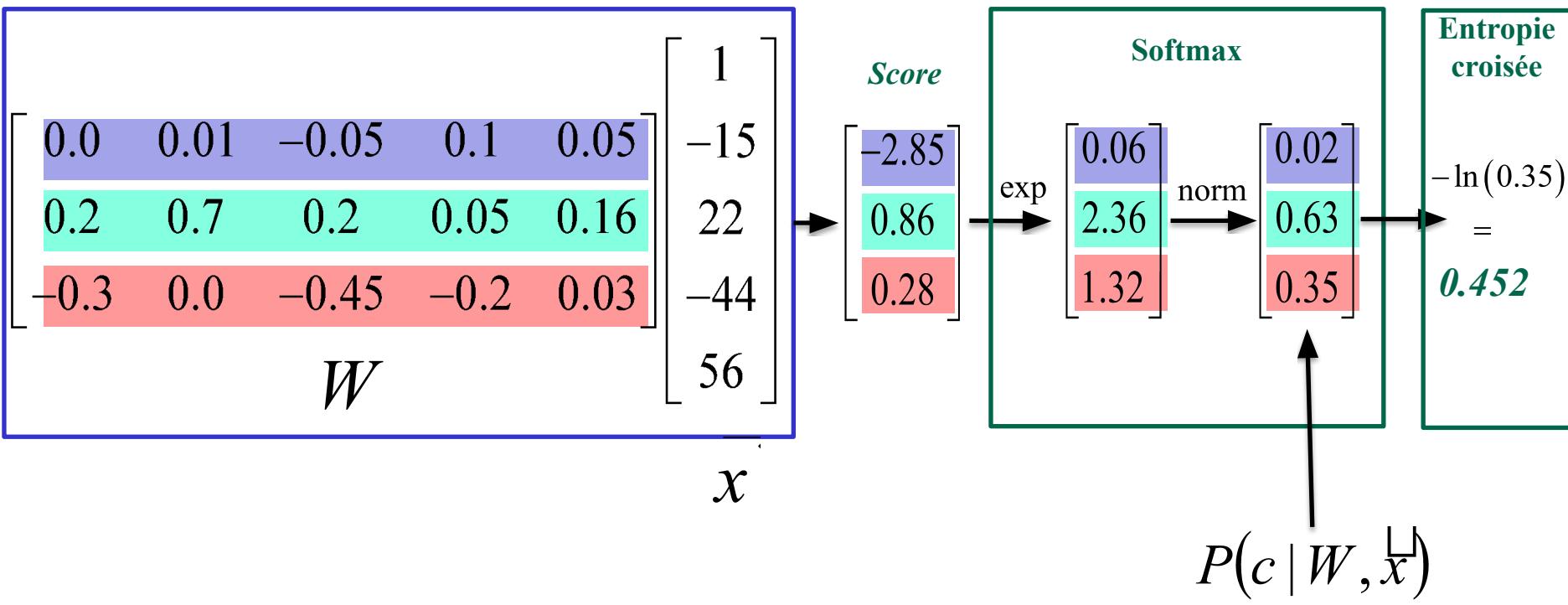
Propriété 2

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



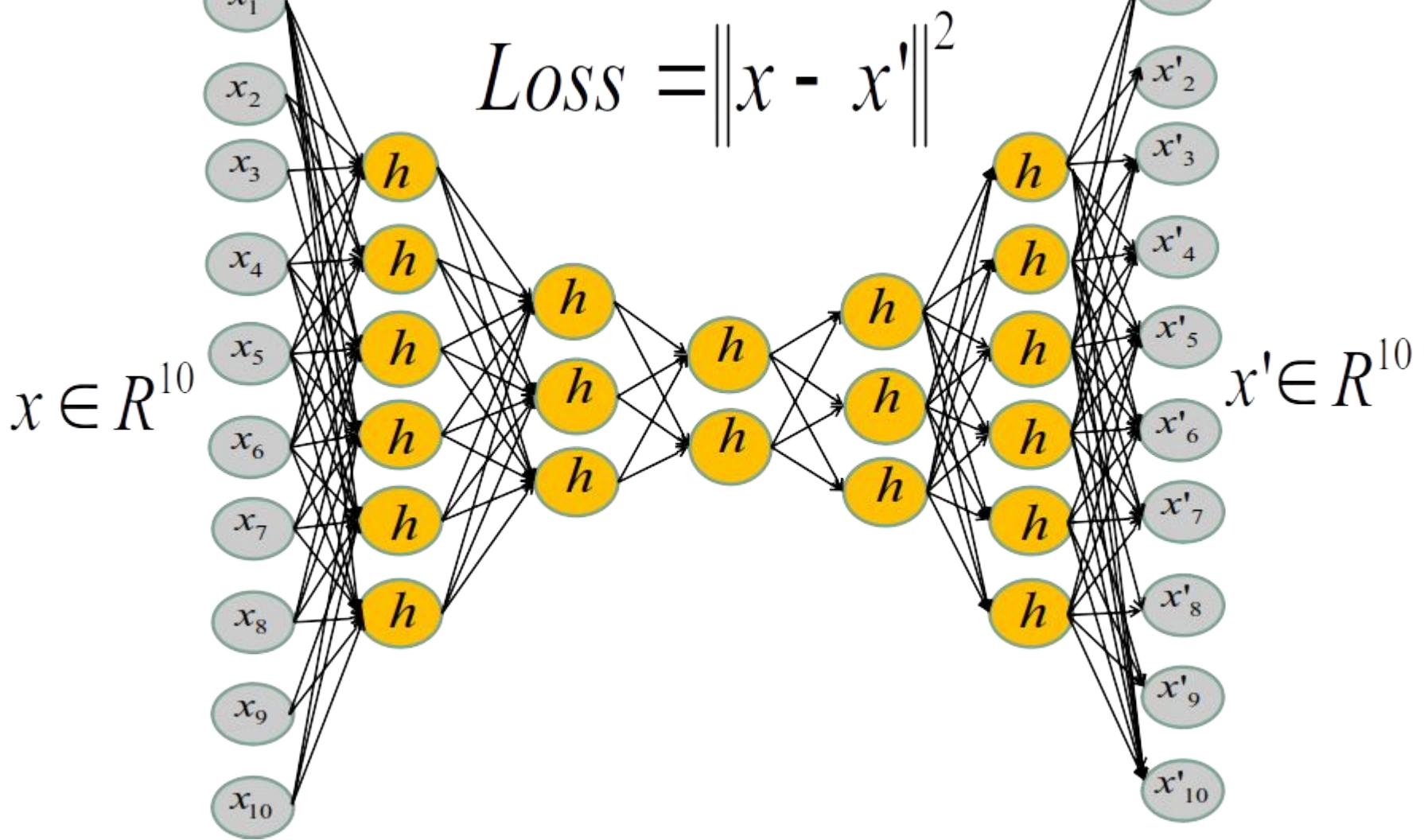
$$\vec{x} = \begin{bmatrix} -15 \\ 22 \\ -44 \\ 56 \end{bmatrix}, t = 2$$

Rappel

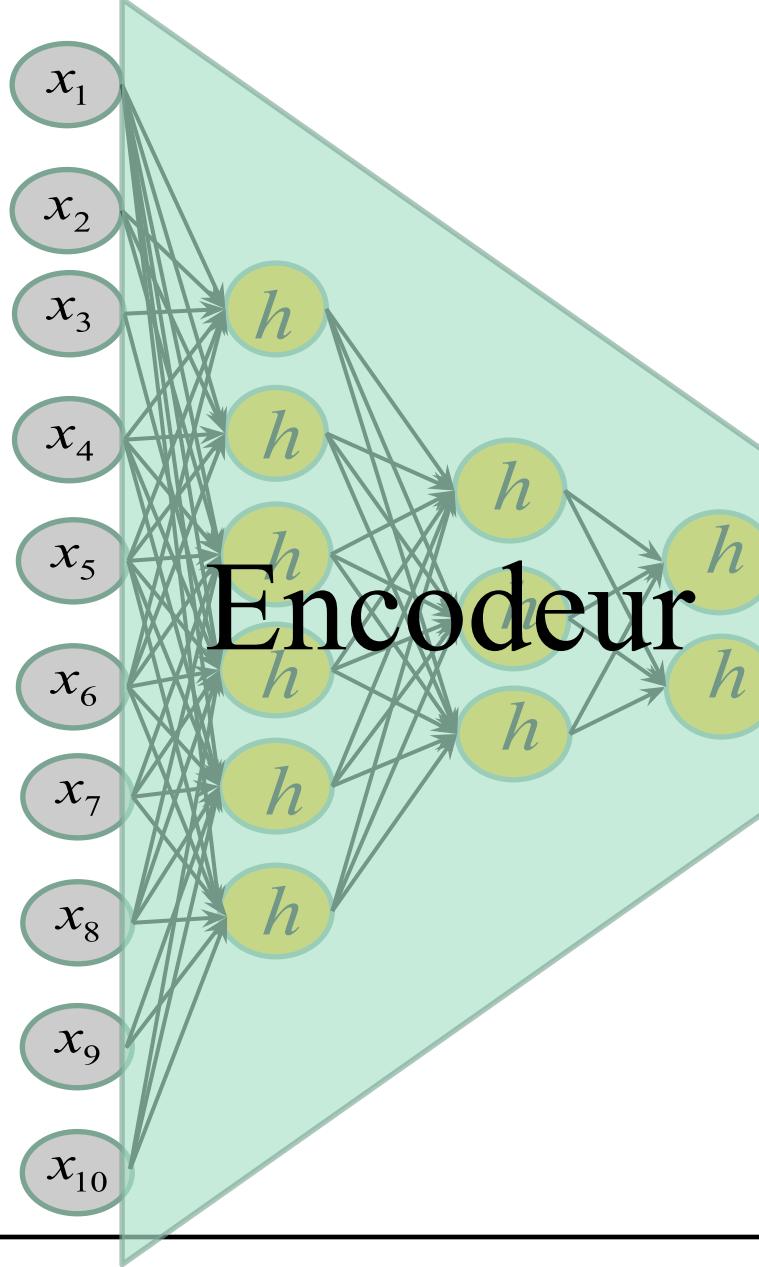


Comment utiliser un réseau de neurones pour apprendre
la **configuration sous-jacente** de données non étiquetés?





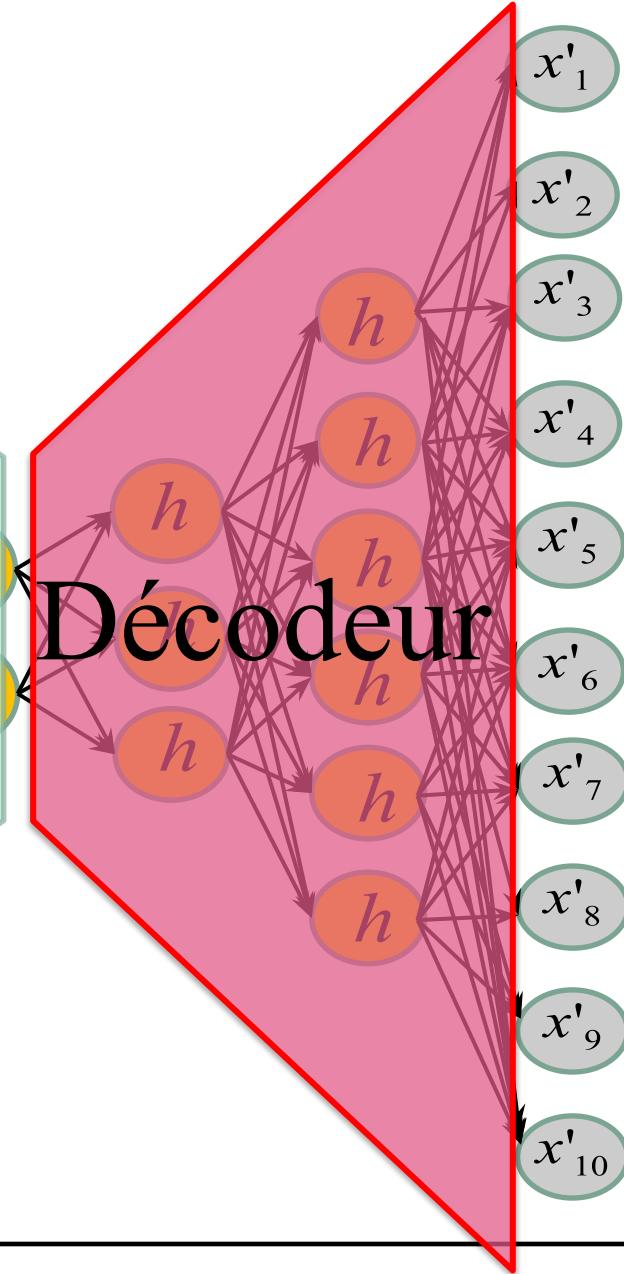
x

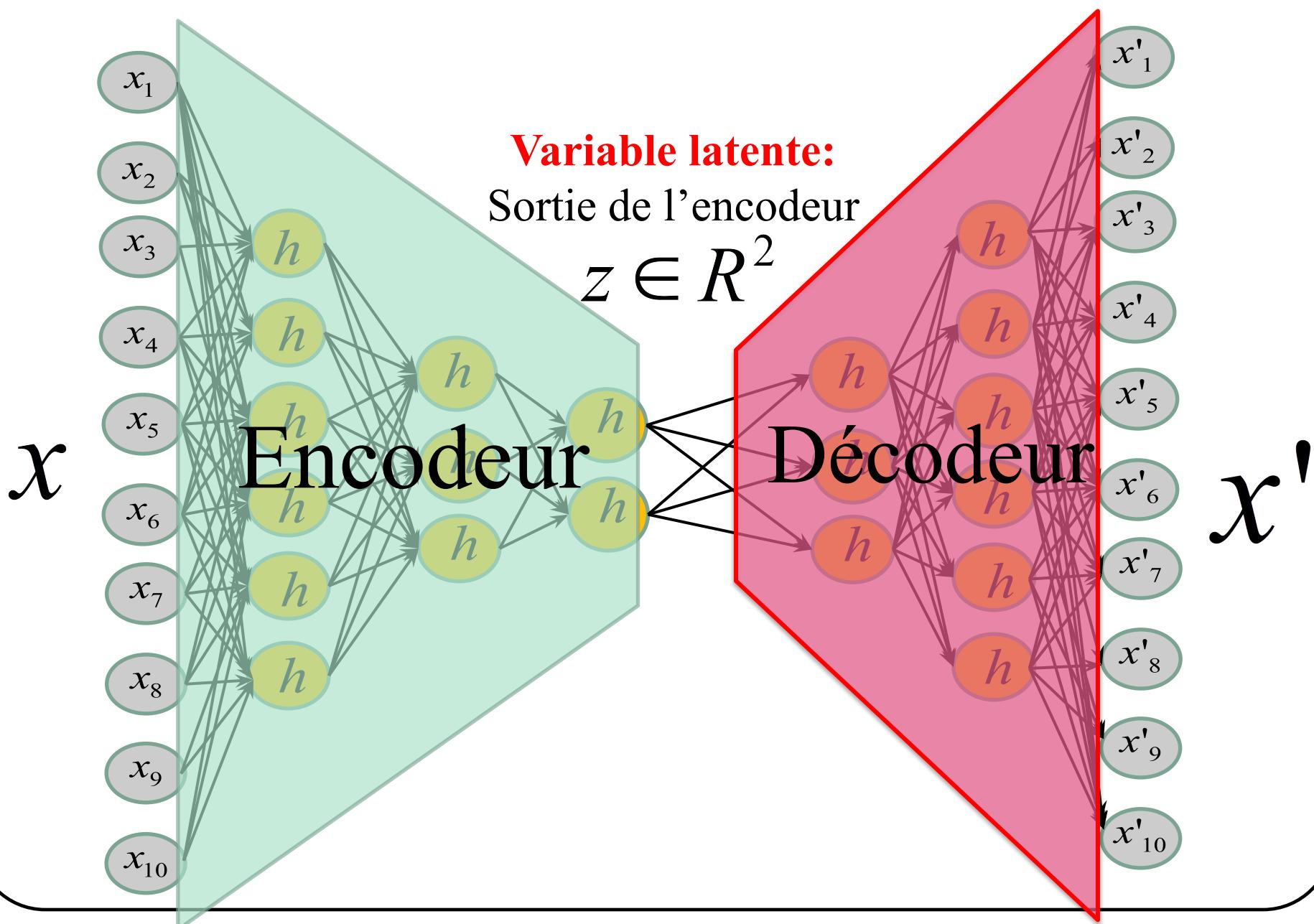


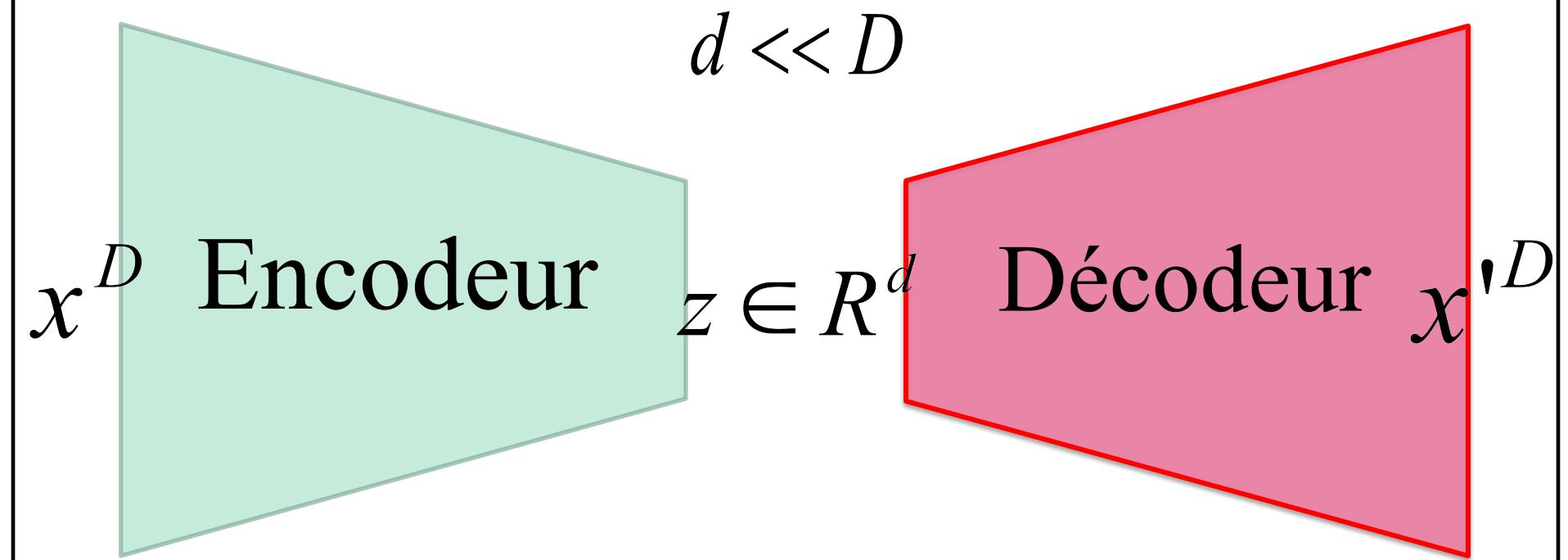
Encodeur

Décodeur

x'

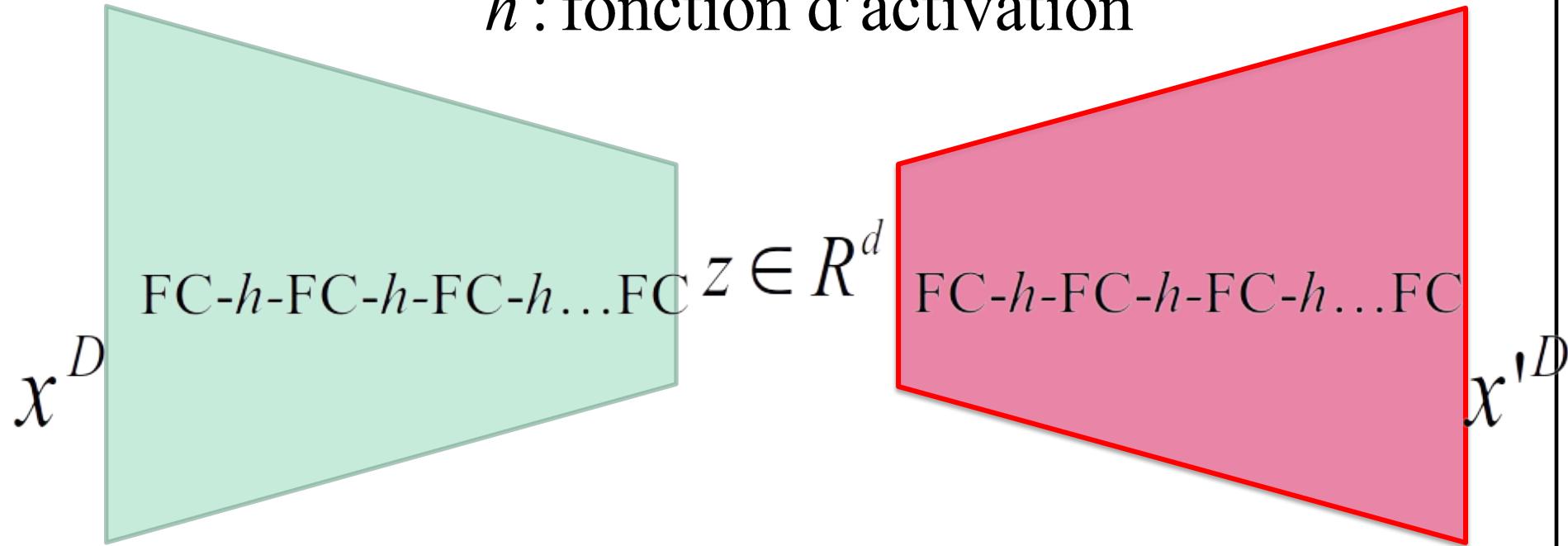




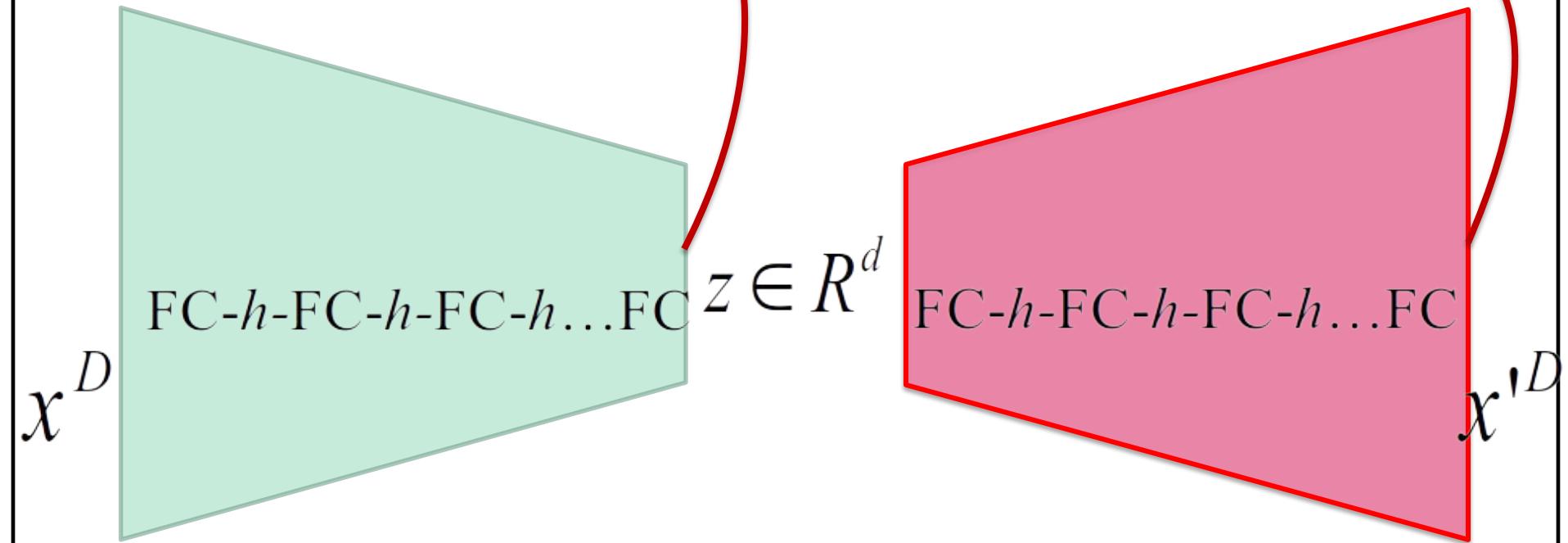


Couches pleinement connectées

h : fonction d'activation

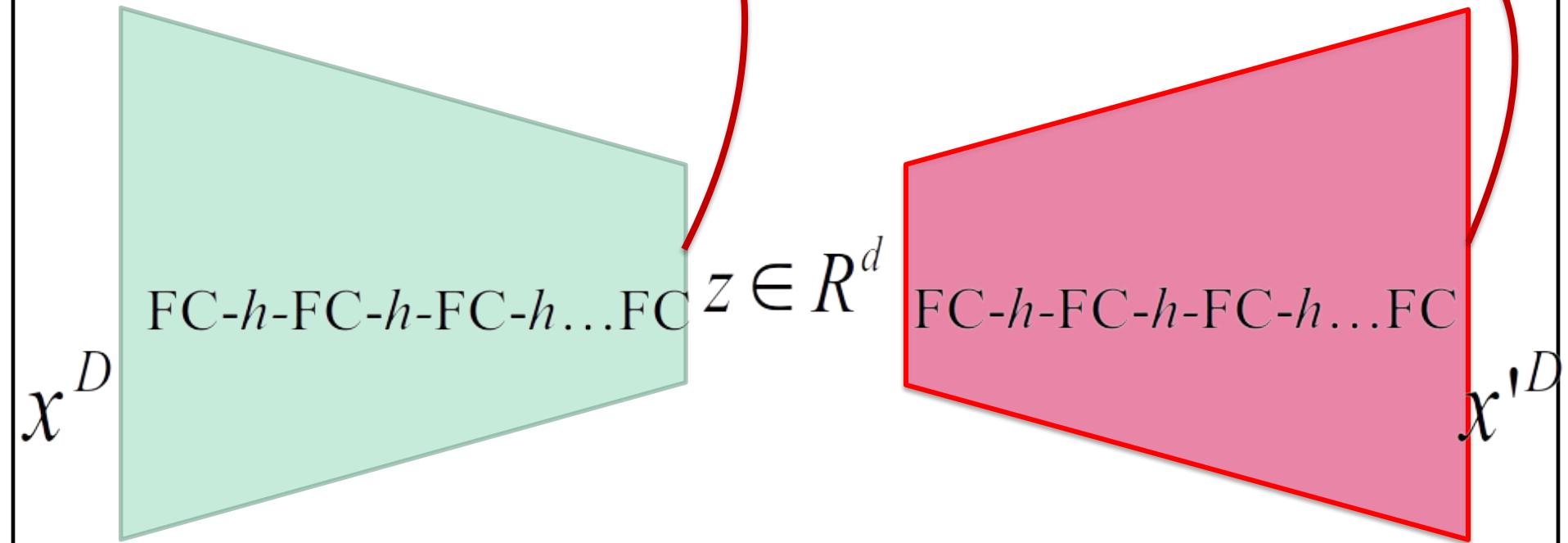


En général...
pas de fonction d'activation à la sortie
de l'encodeur et du décodeur



Selon vous, **pourquoi?**

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

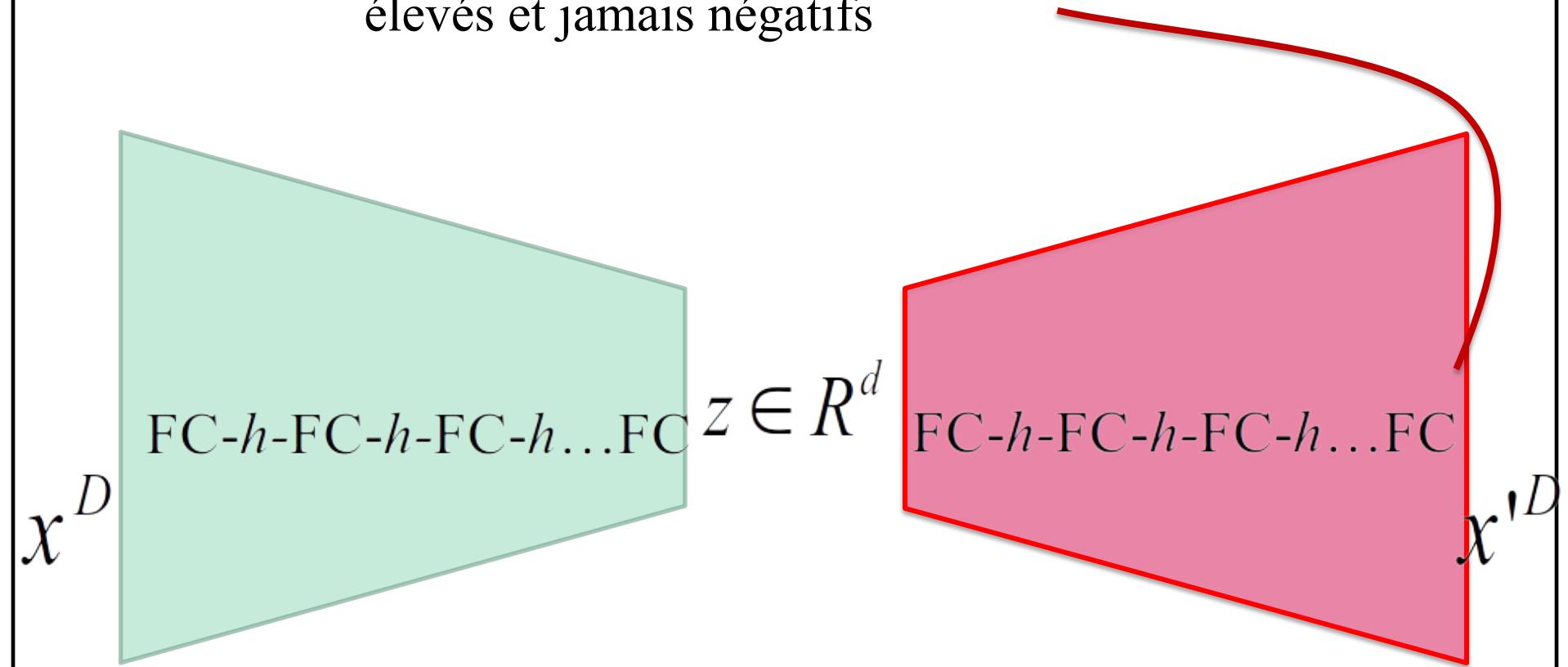


Selon vous, **pourquoi?**

Parce qu'on veut que le x encodé soit réellement un point
dans l'espace latent

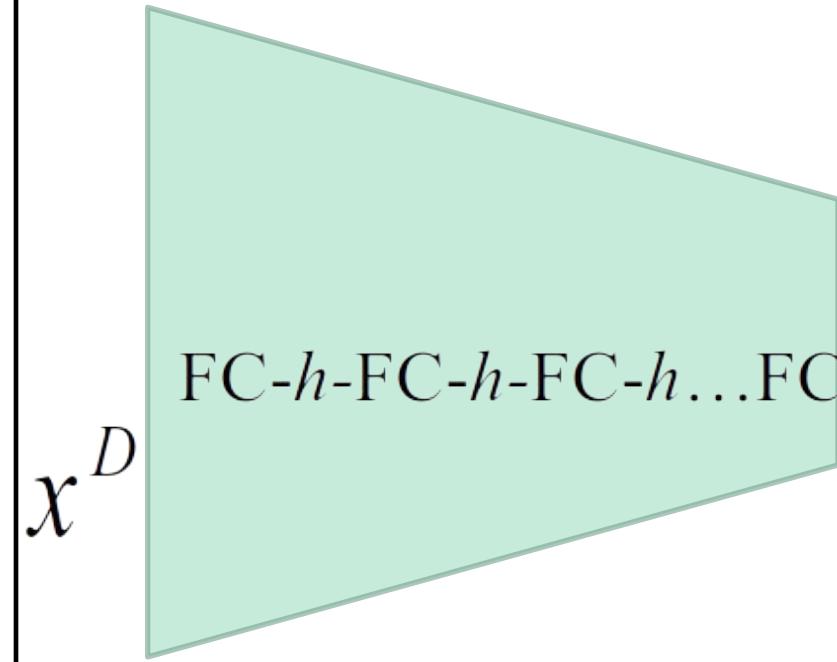
Parfois sigmoïde en sortie lorsque les pixels ont des niveaux de gris entre 0 et 1.

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



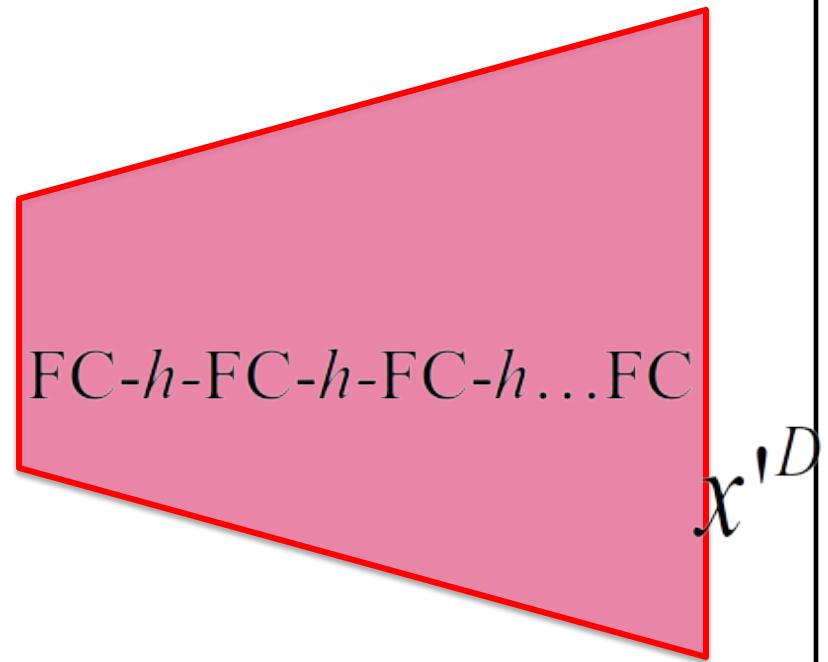
Le nombre de neurones

Décroît ou se maintient
d'une couche à l'autre



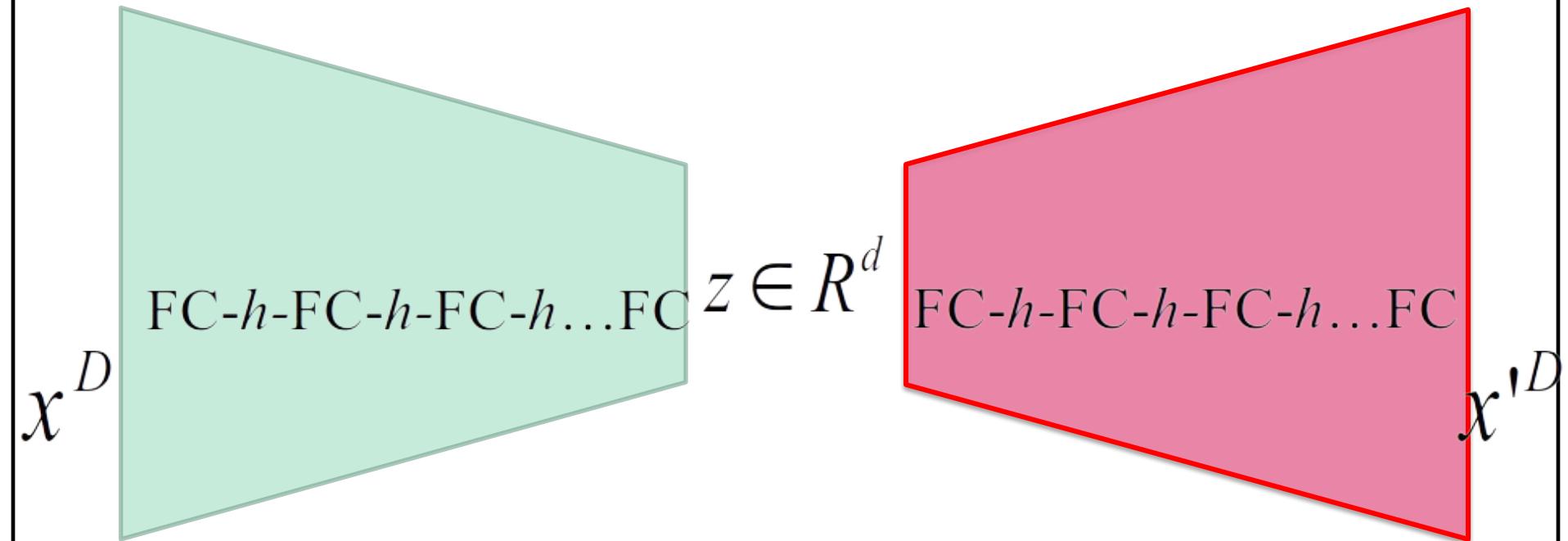
Le nombre de neurones

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

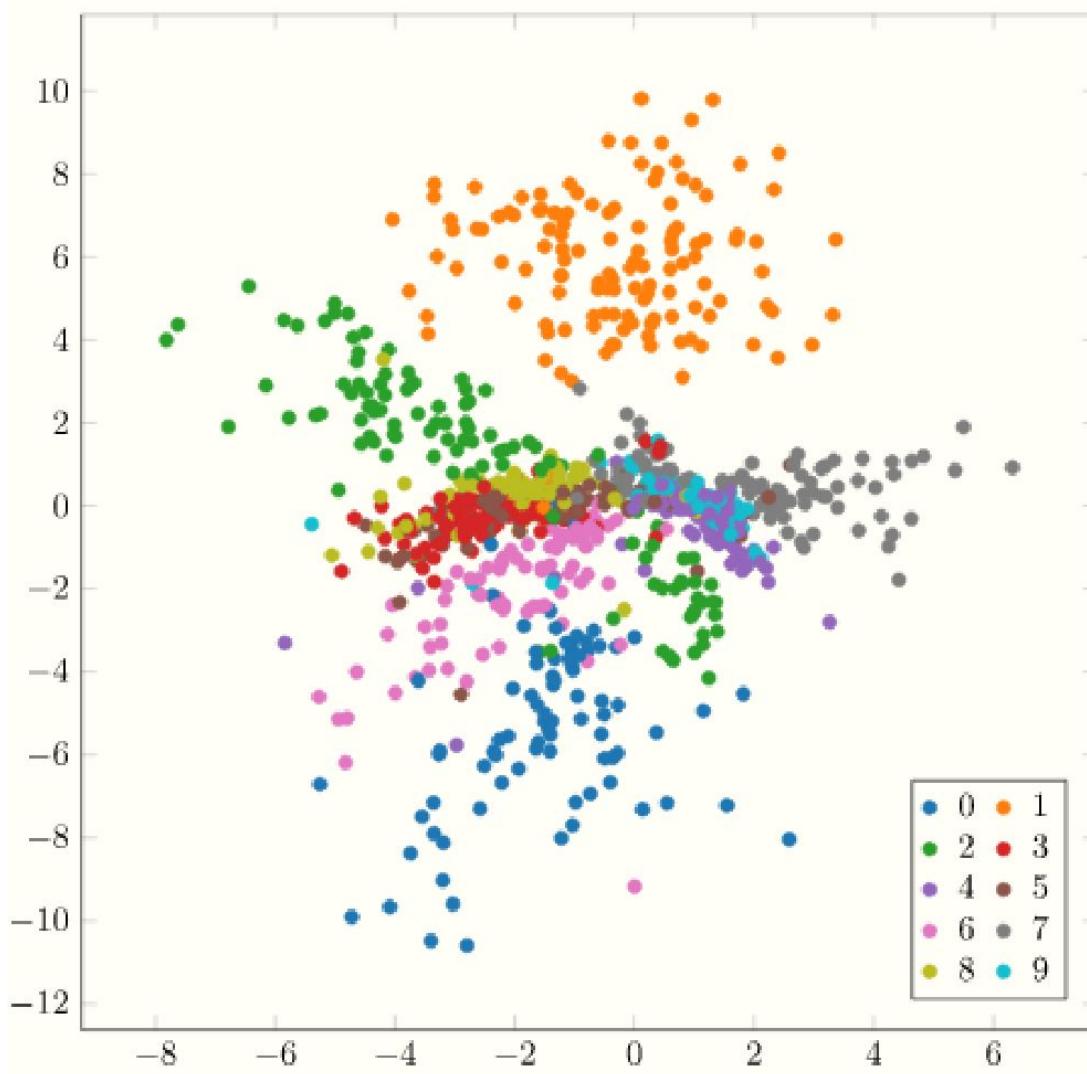
Autoencodeur jouet de MNIST

```
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
```

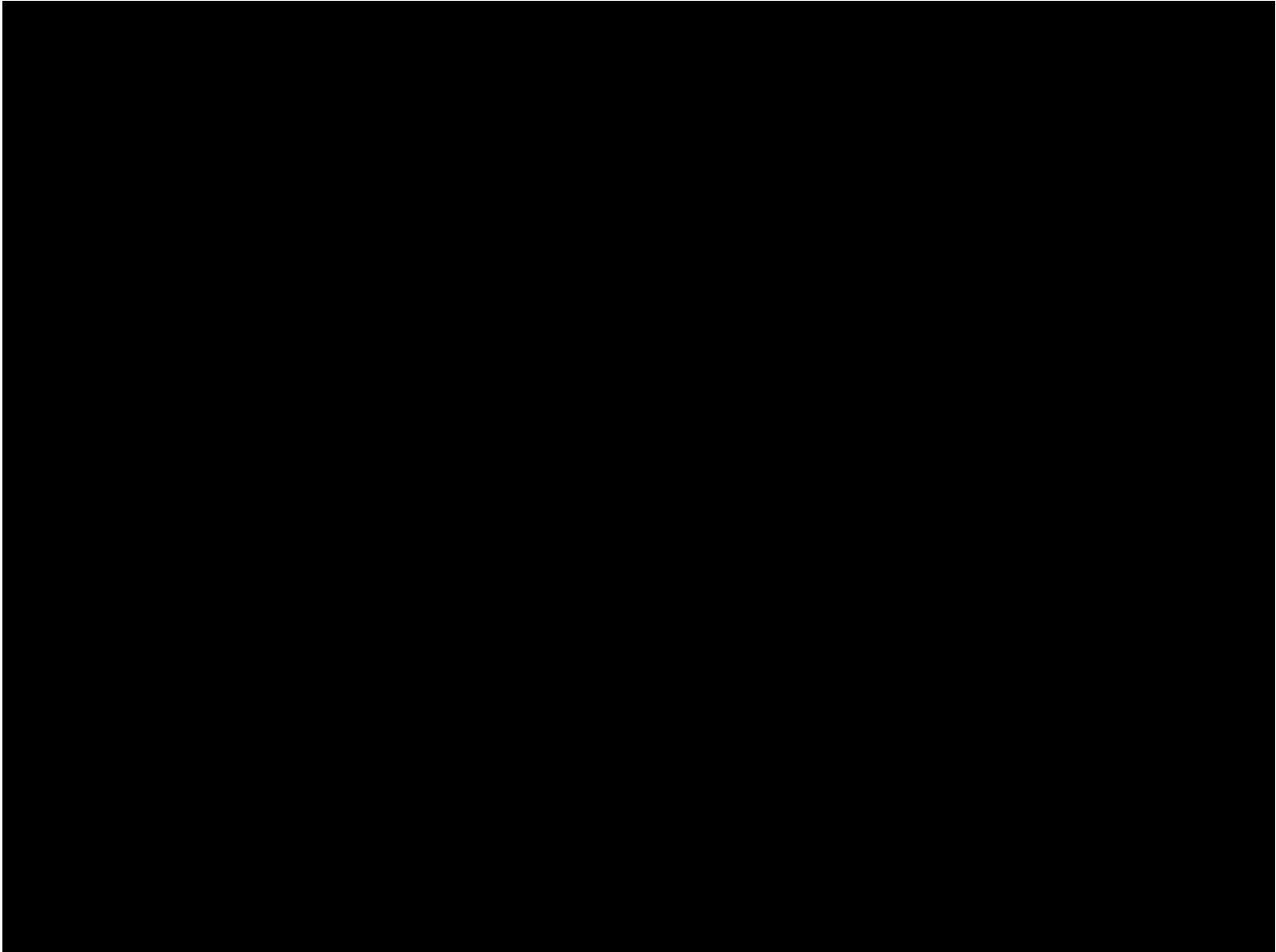
symétrie

Visualisation de l'espace latent pour 1000 images MNIST

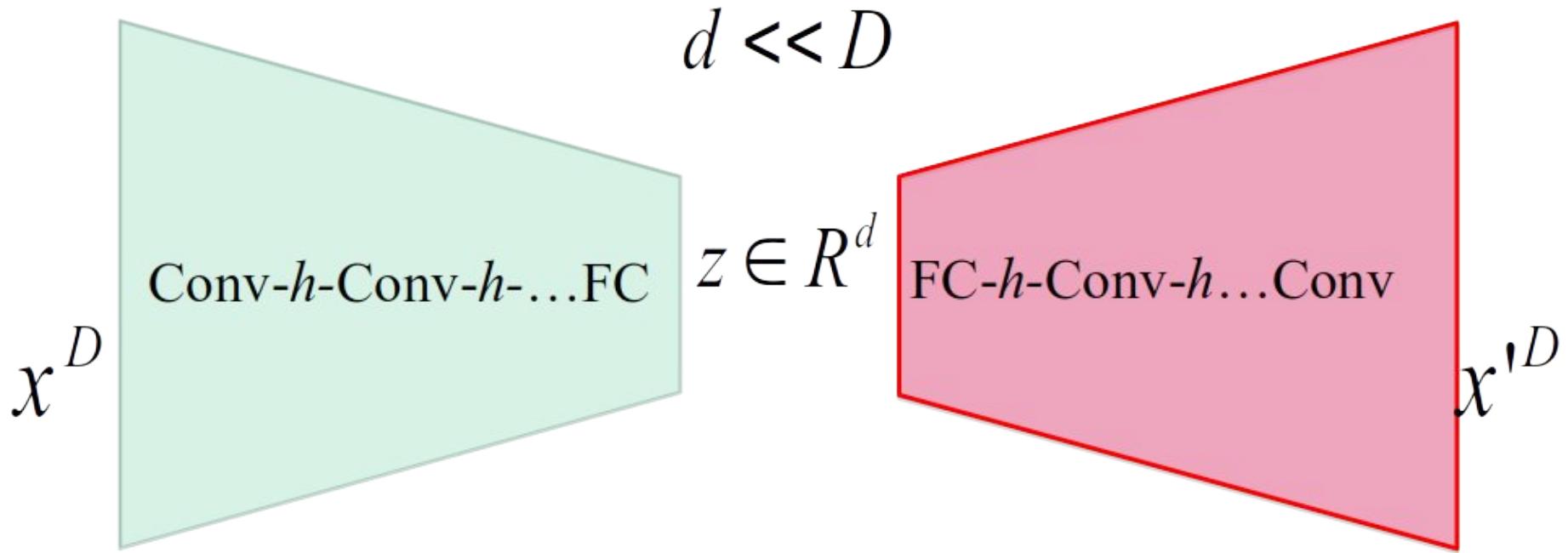
chaque image correspond à 1 point 2D



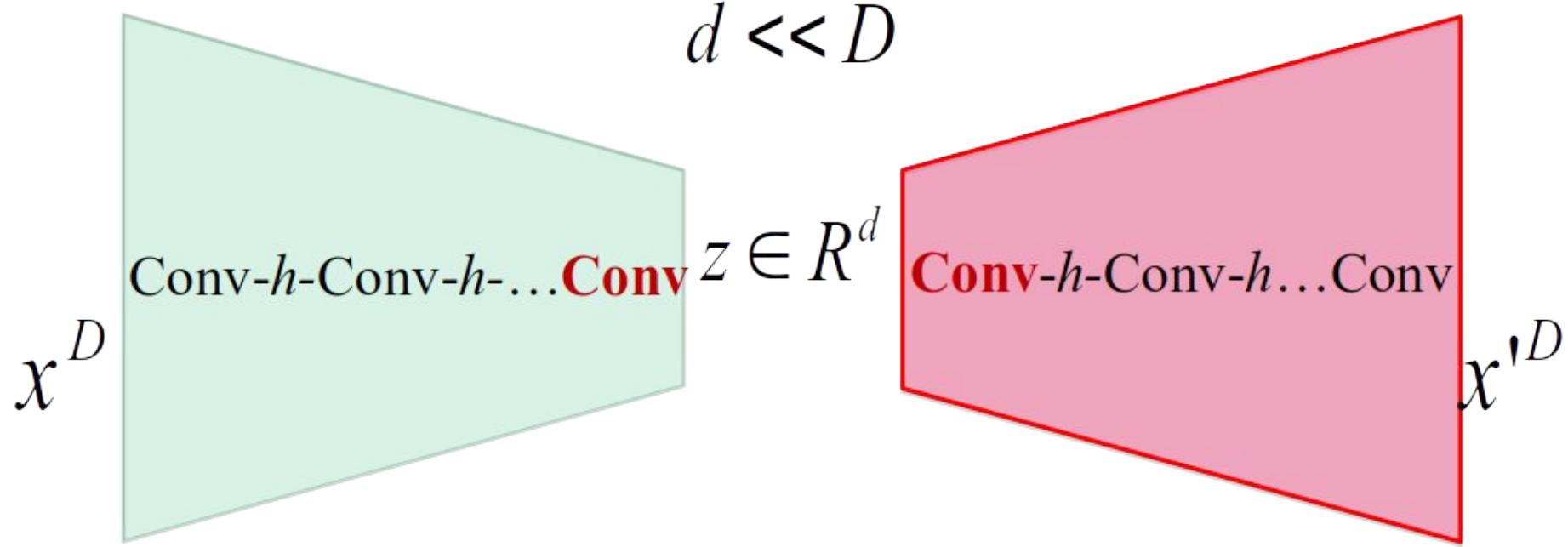
Générer des images avec le décodeur.



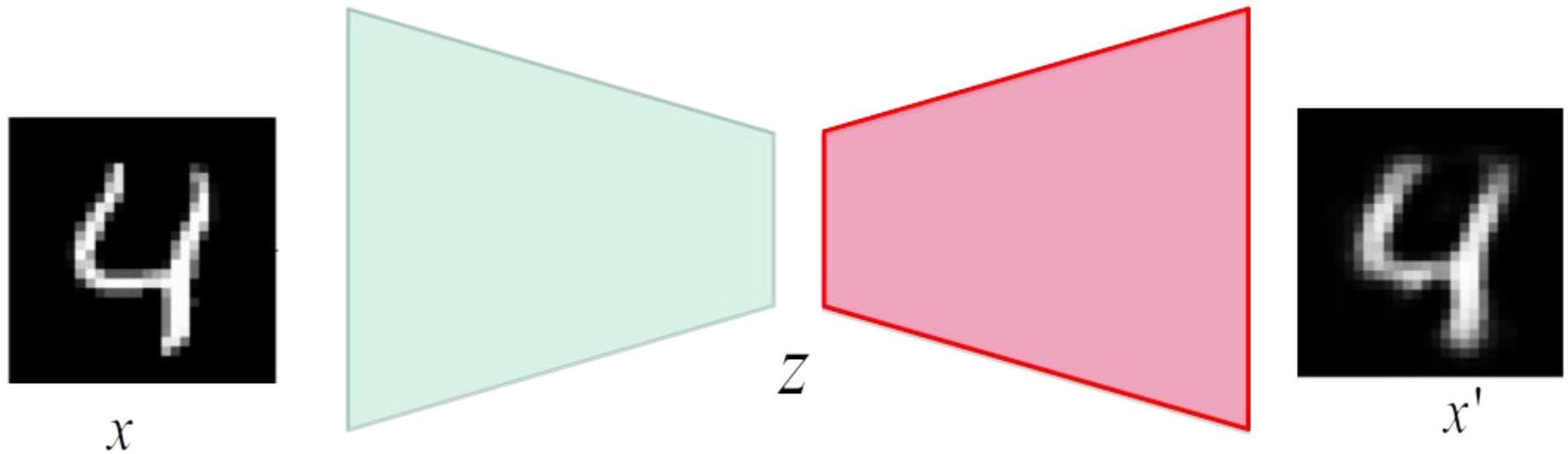
Couches convolutives



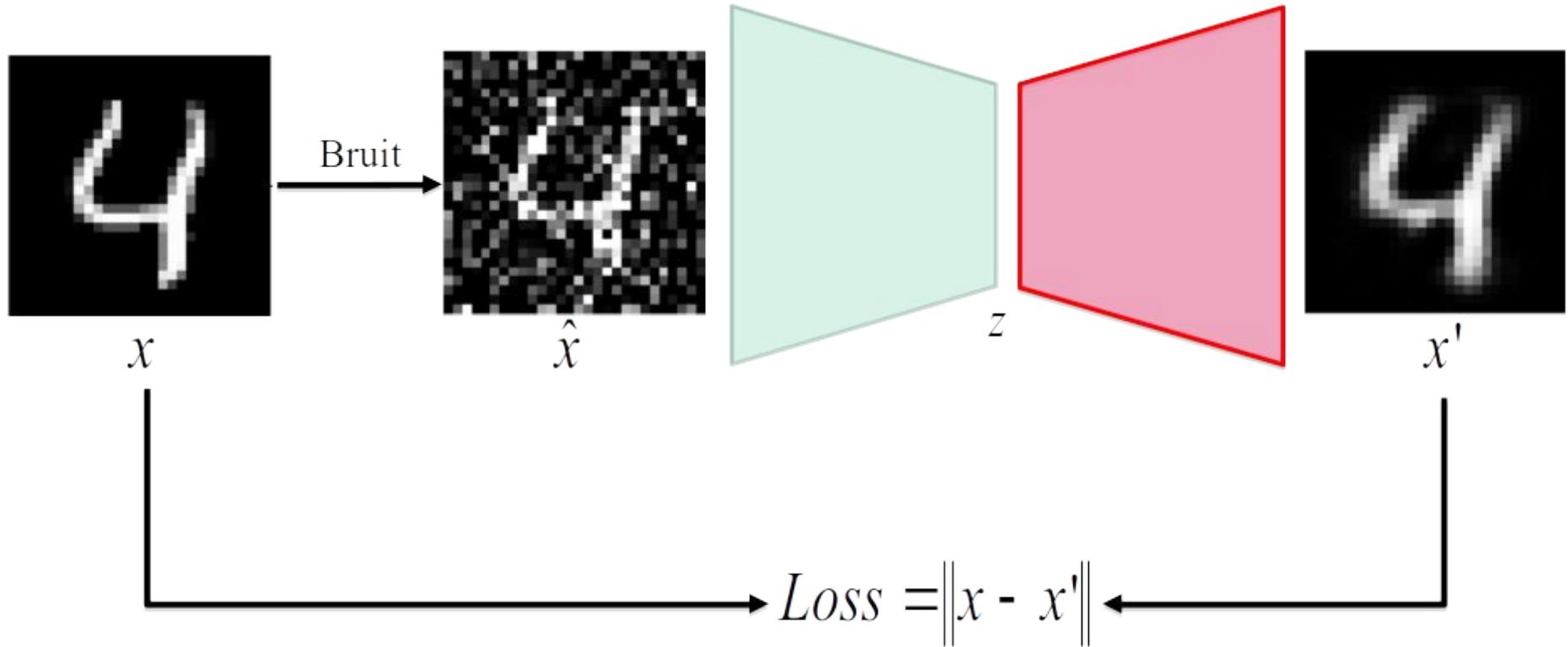
Autoencodeur pleinement convolutif



Autoencodeur de base

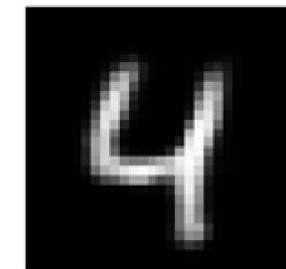
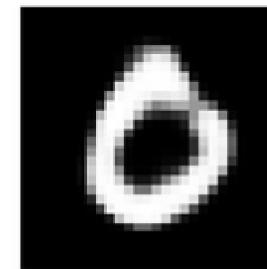
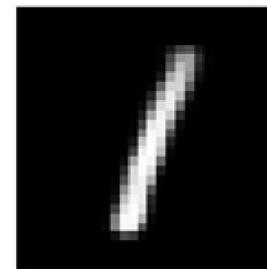
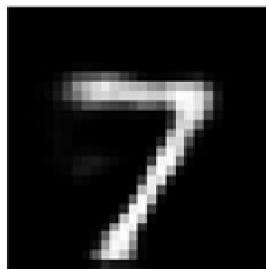
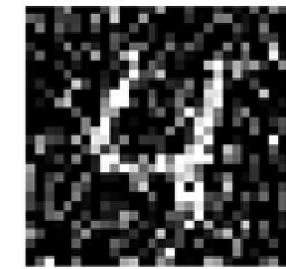
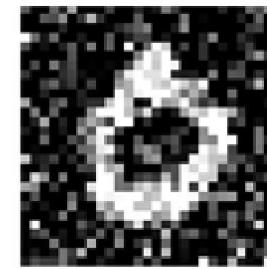
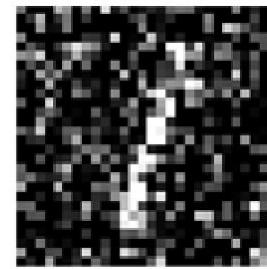
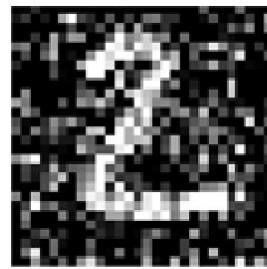
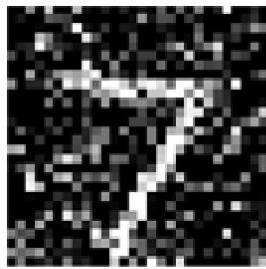


Autoencodeur pour débruitage



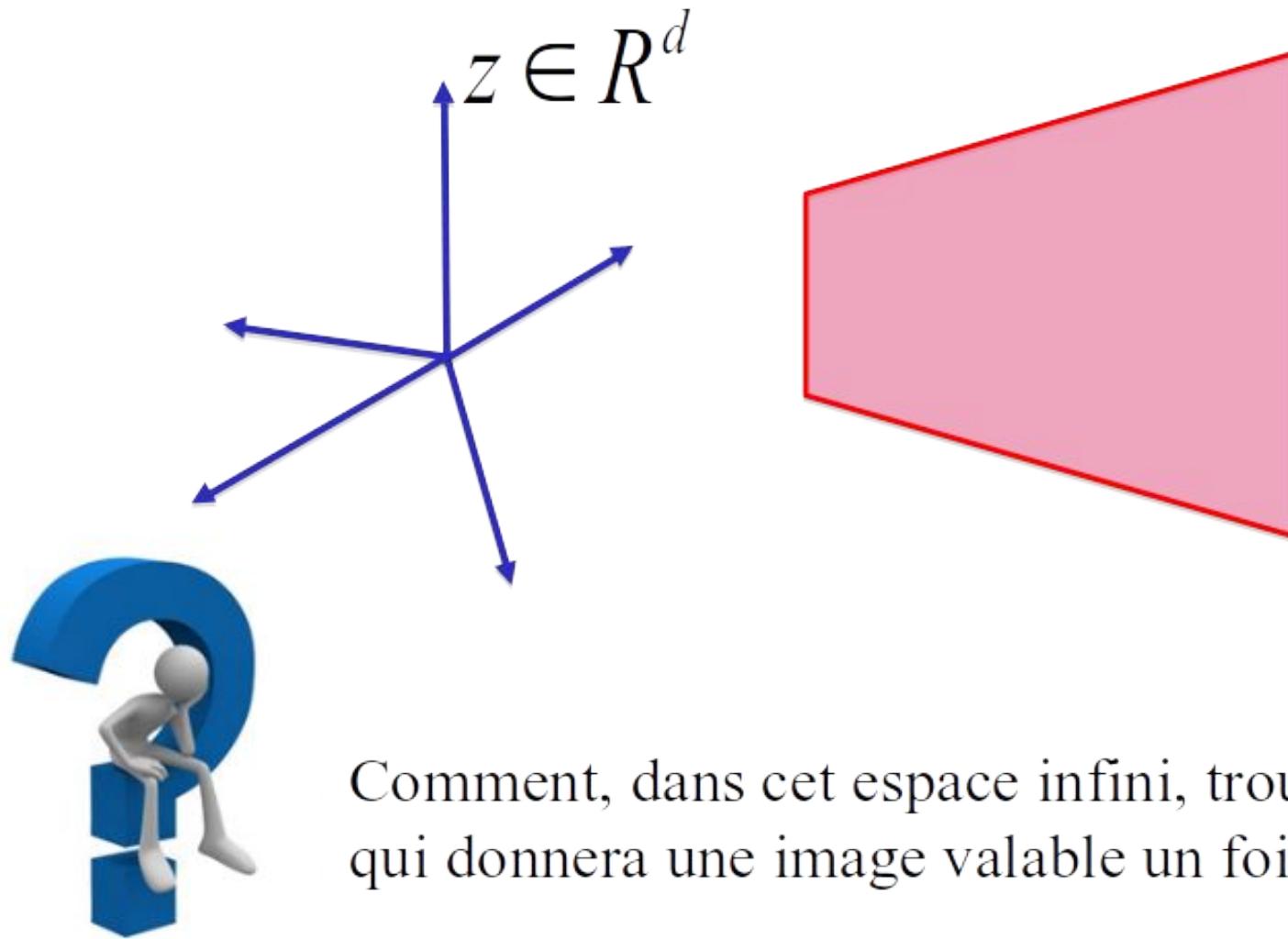
Une fois entraîné...

Signal bruité en entrée



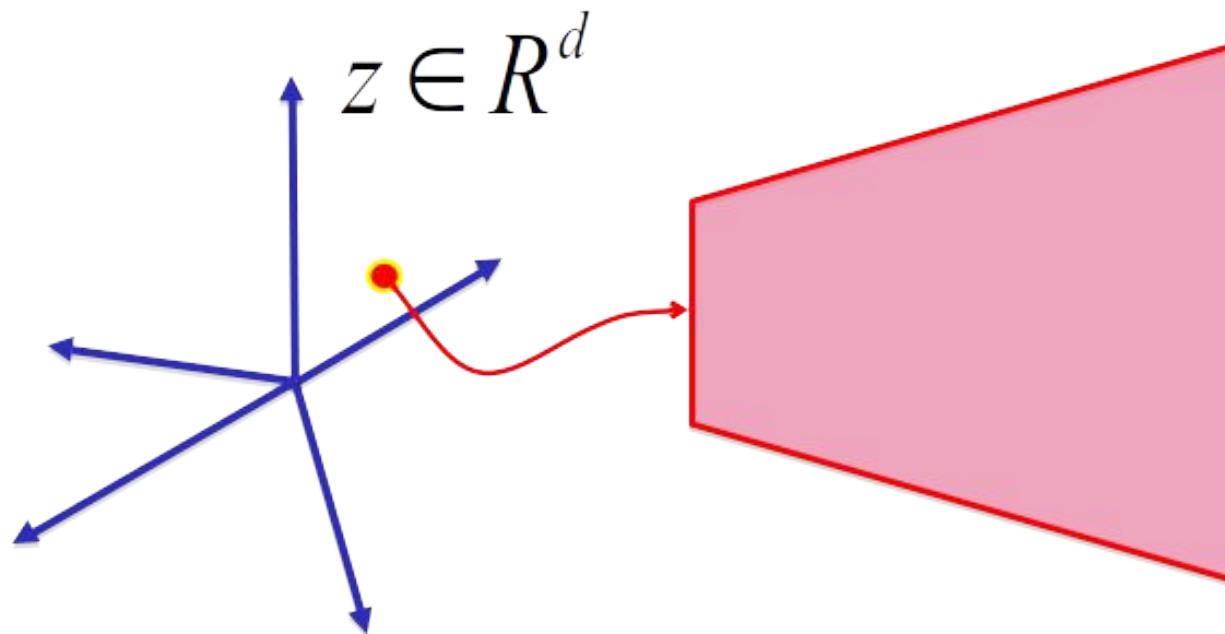
Signal reconstruit et débruité

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

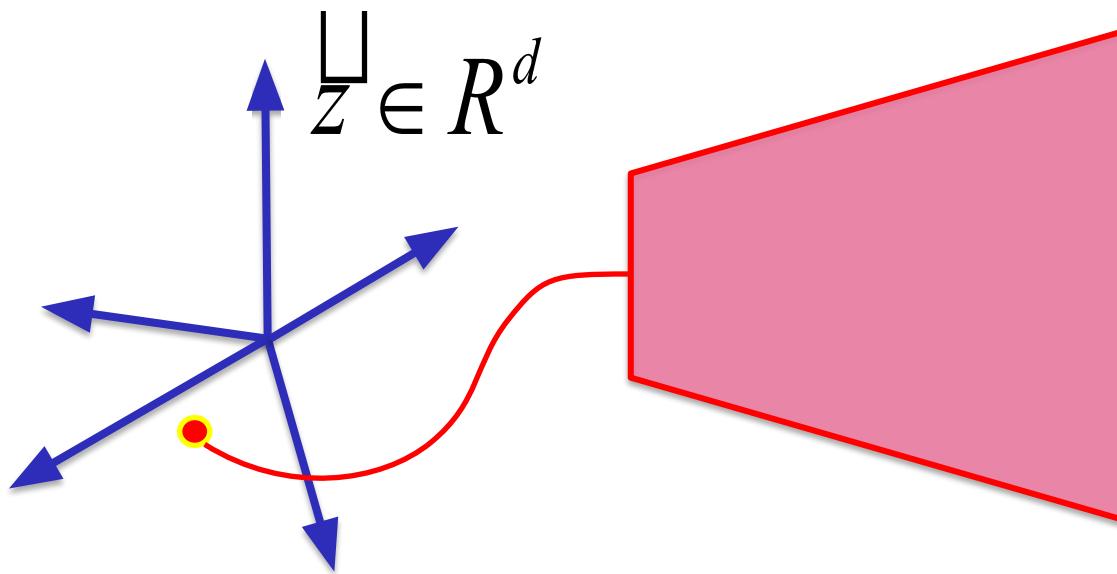


Comment, dans cet espace infini, trouver un point z qui donnera une image valable un fois décodée?

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...

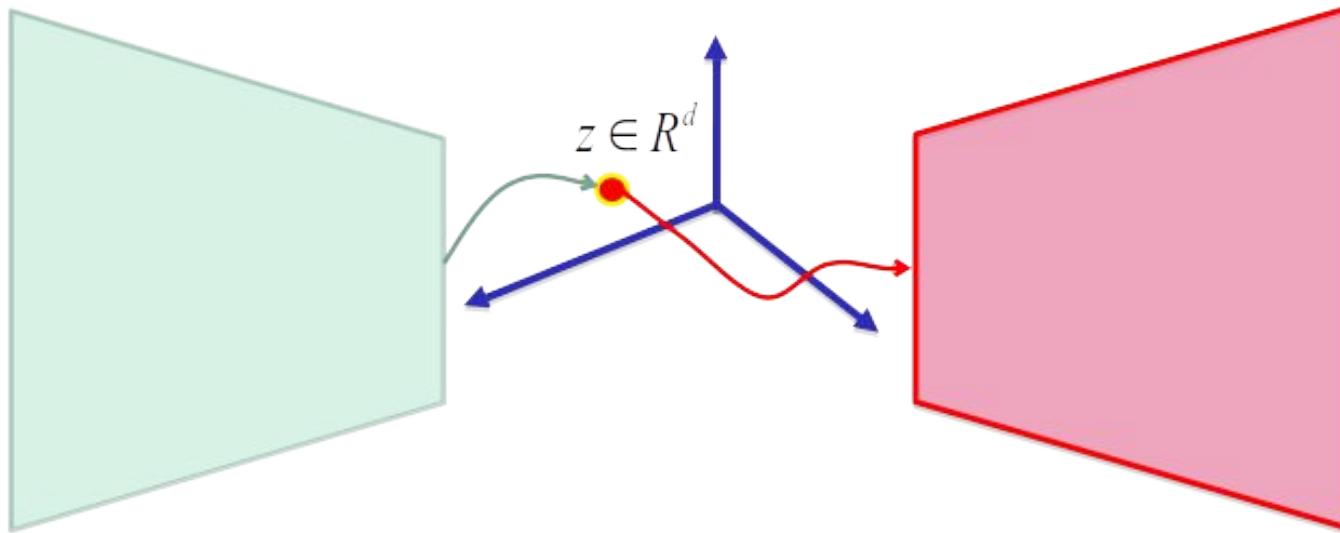


Apprendre à reproduire une **distribution** $p(\mathbb{z})$
connue de sorte qu'un **point échantillonné**
et décodé de cette distribution correspond à
un signal reconstruit valable

Autoencodeur de base



x

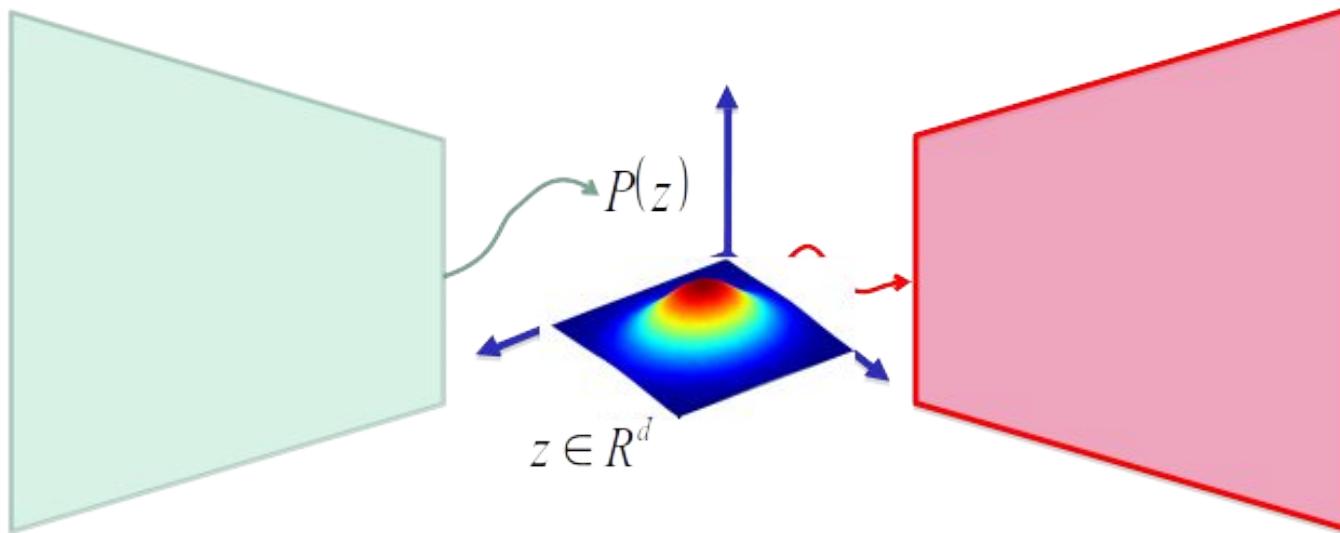


x'

Autoencodeur variationnel



x

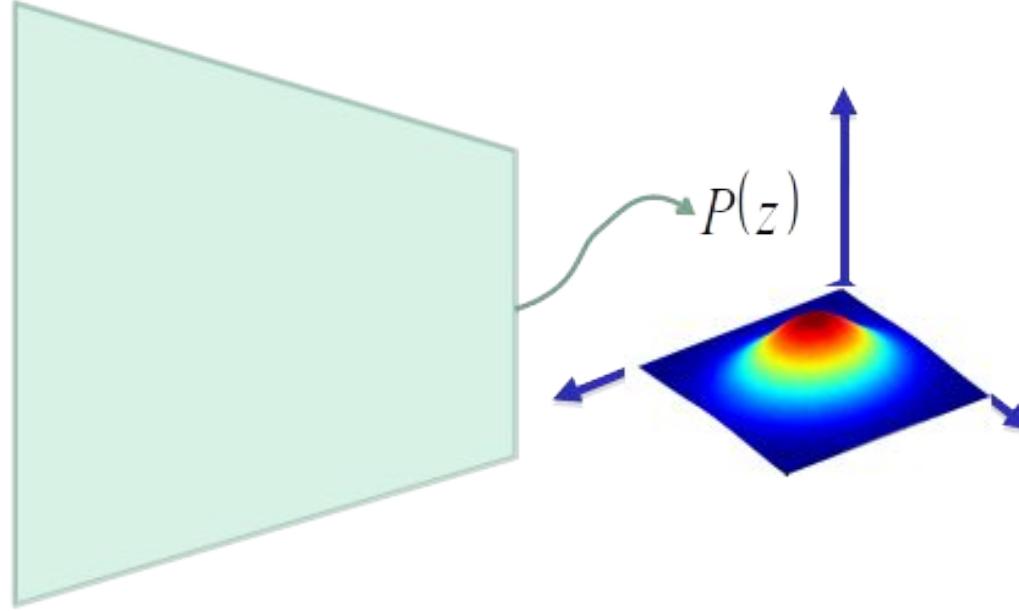


x'

Autoencodeur variationnel



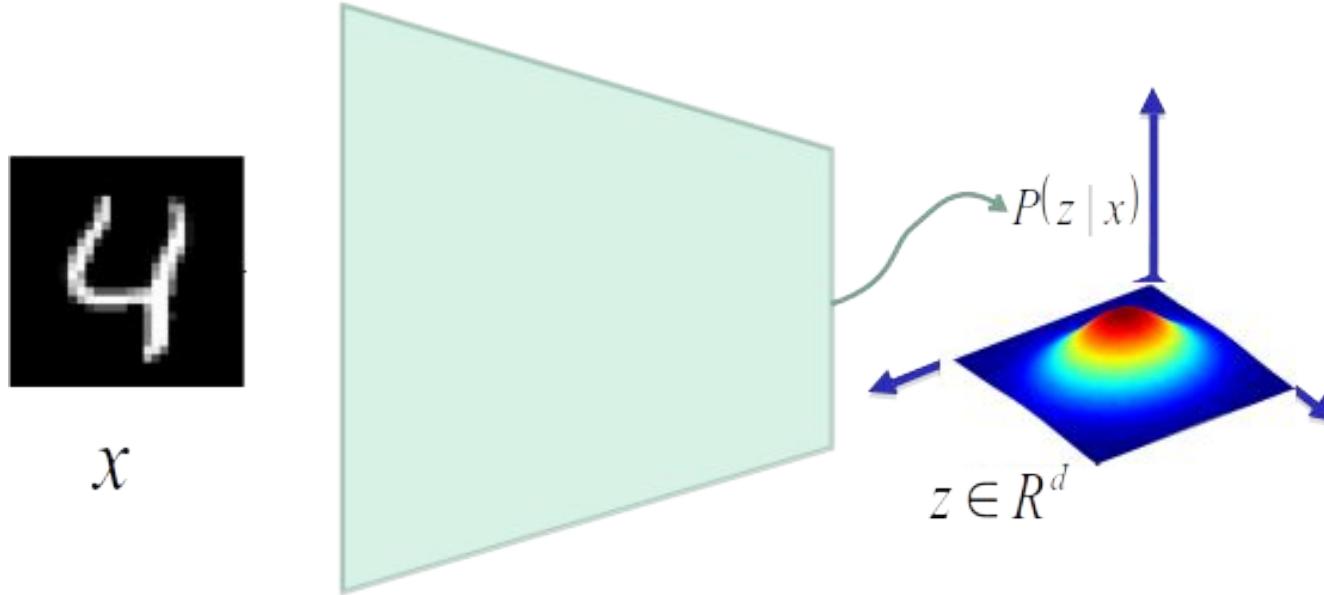
x



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

Autoencodeur variationnel

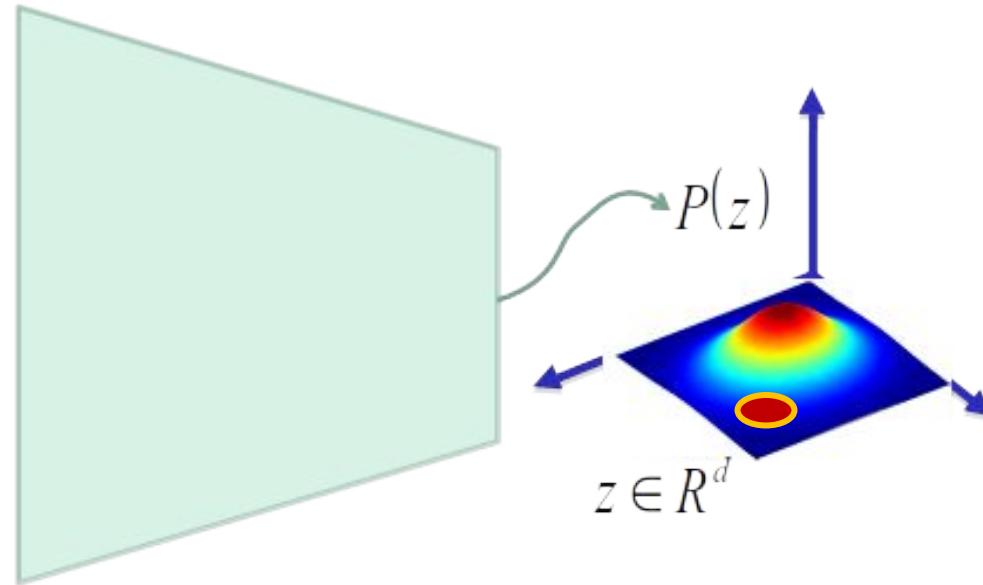
Remarque 1



Puisque la distribution de z dépend de x
on dira que la distribution apprise est

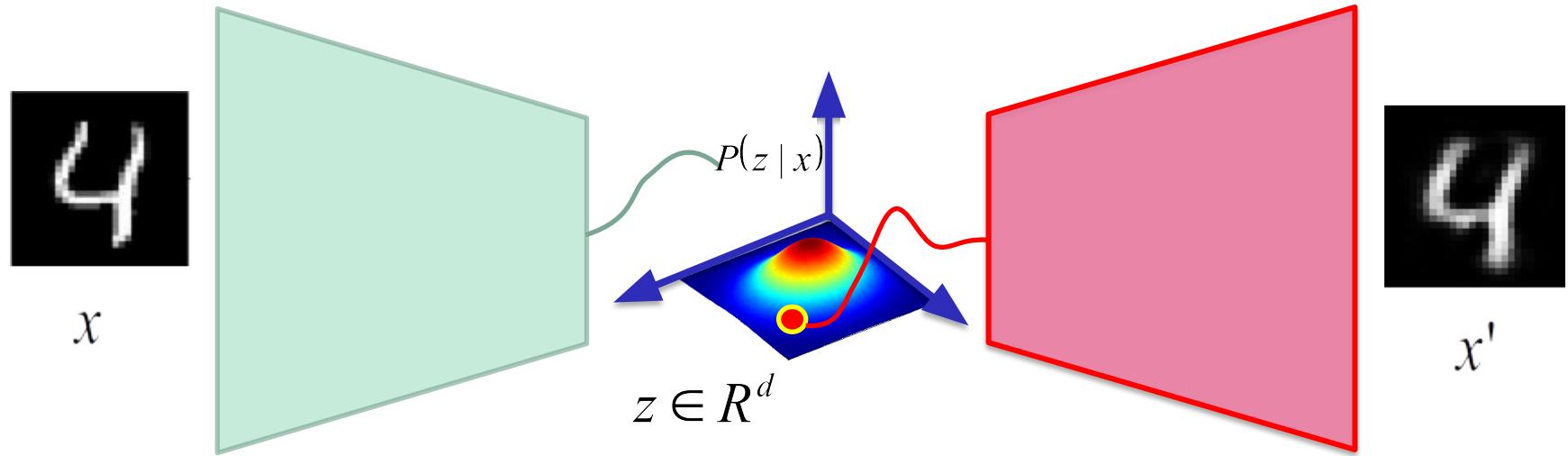
$$P(z | x)$$

Autoencodeur variationnel



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

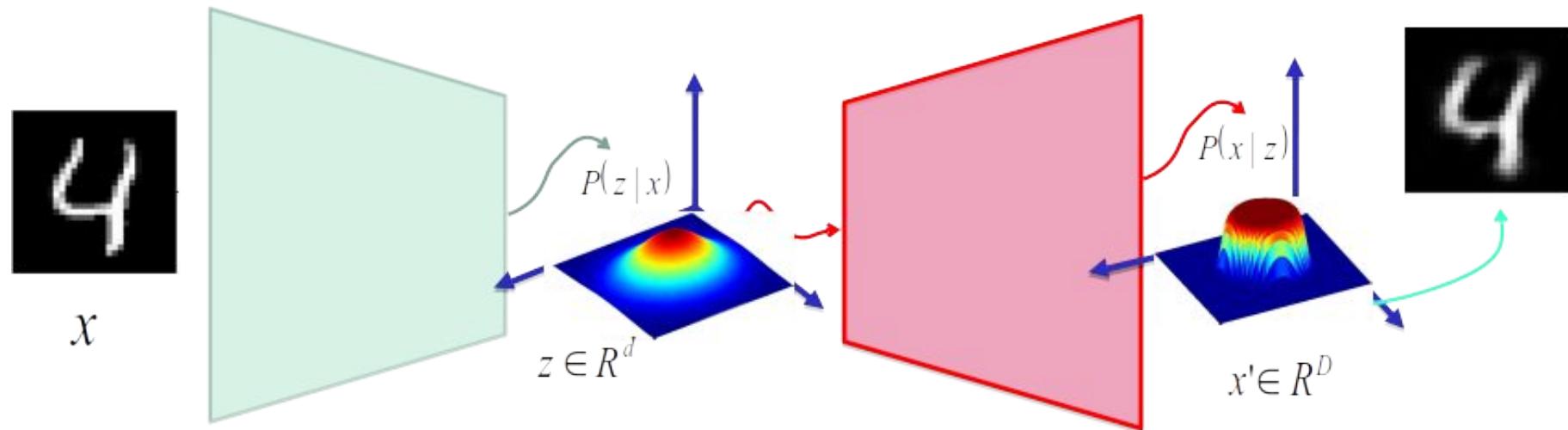
Autoencodeur variationnel



On reconstruit x'

Autoencodeur variationnel

Remarque 2



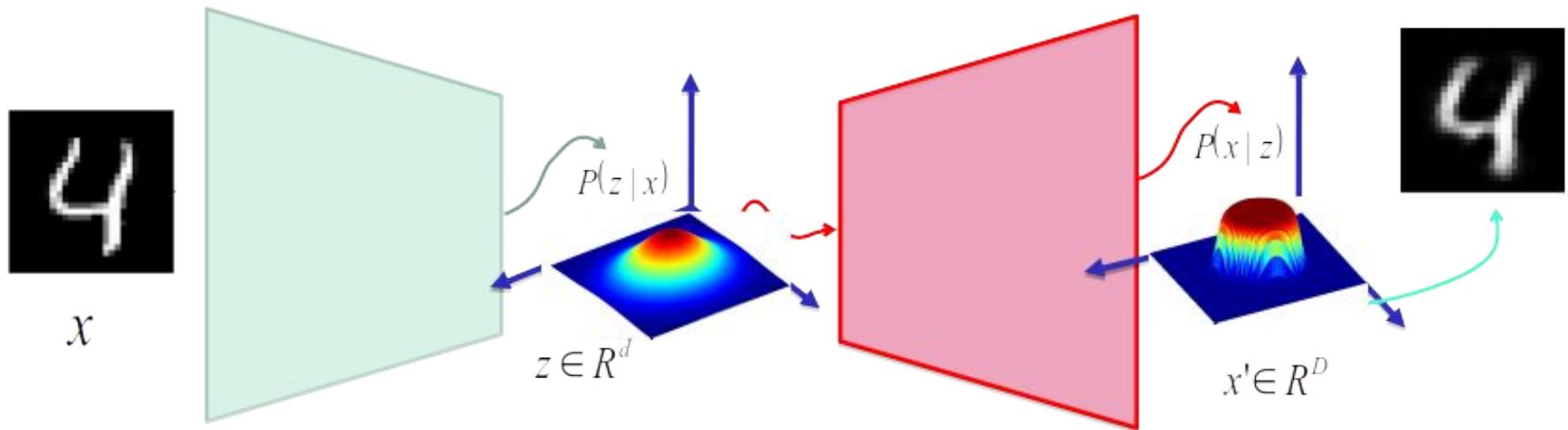
Le **décodeur** peut également produire une distribution de probabilités

$$P(x|z)$$

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

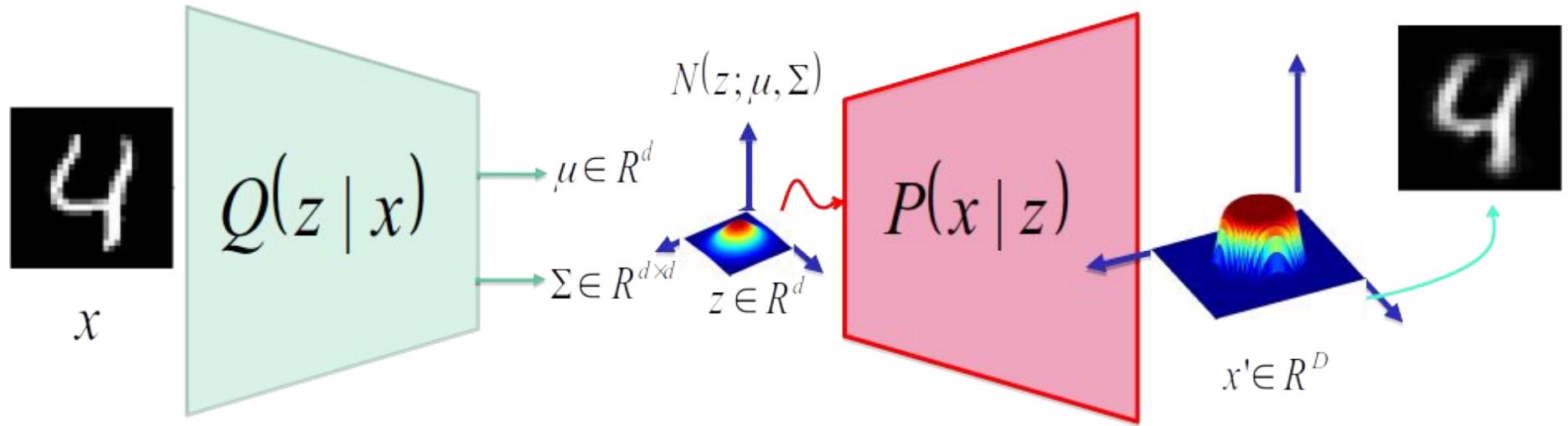
Autoencodeur variationnel

Remarque 3



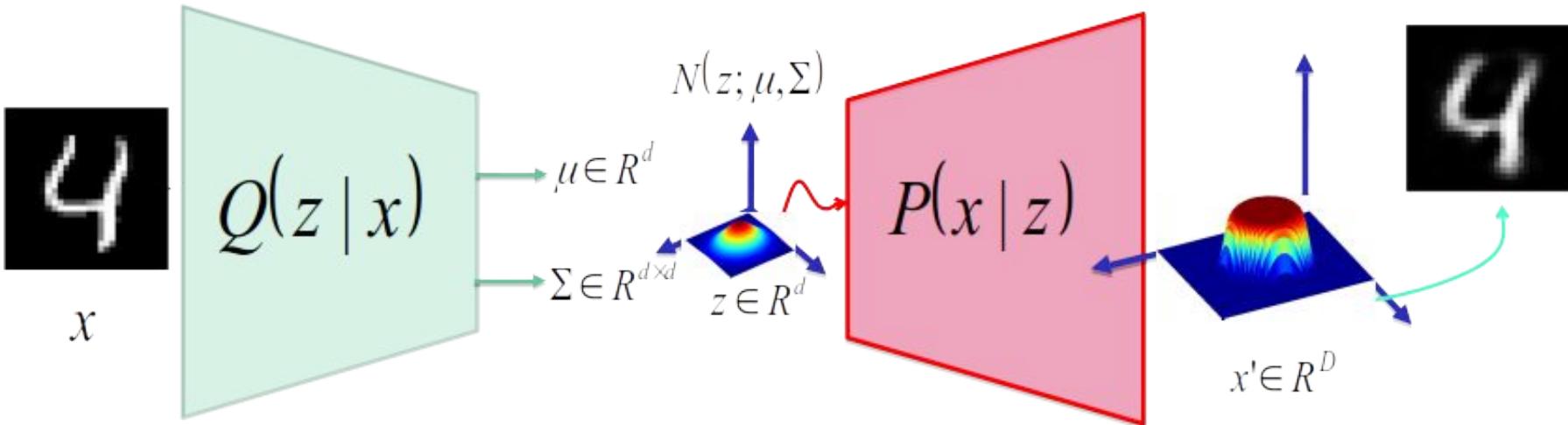
La distribution $P(z|x)$ peut être très complexe et difficile à échantillonner, on va donc l'approximer par une distribution plus simple... une gaussienne

$$Q(z|x) \approx P(z|x)$$



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

Autoencodeur variationnel Remarque 4

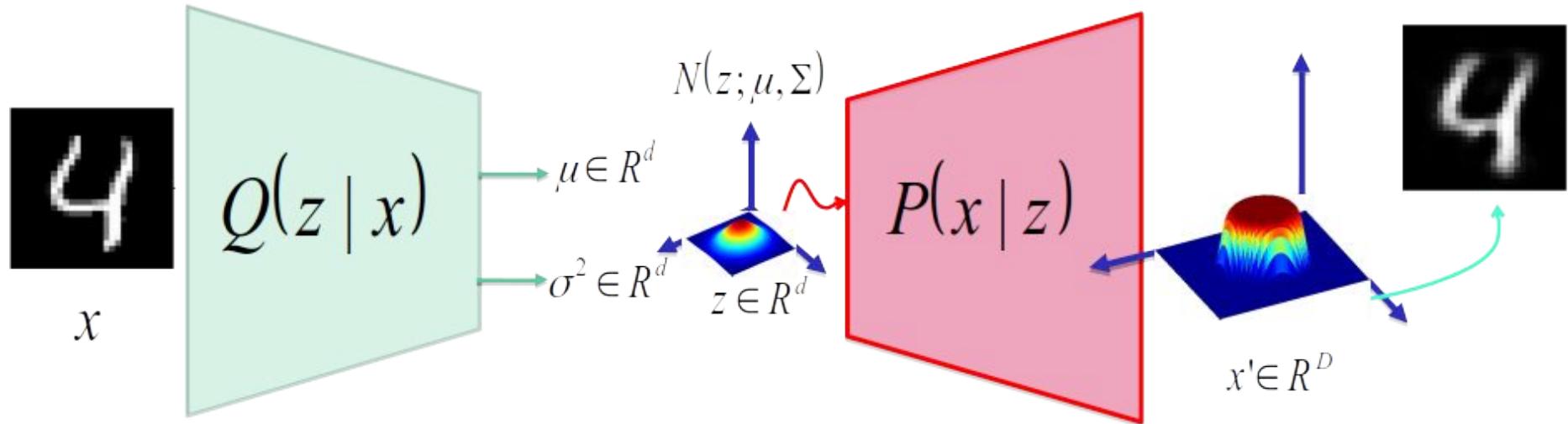


Pour simplifier les calculs, on va supposer que Σ est diagonale

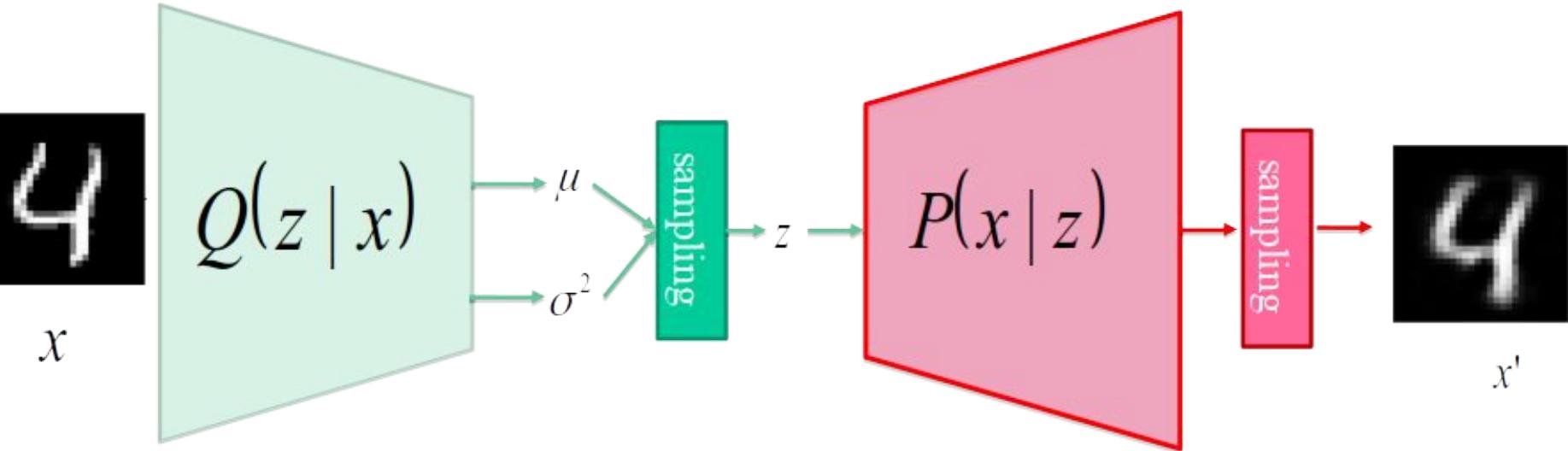
$$\Sigma = \begin{pmatrix} \sigma_1^2 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3^2 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & \sigma_d^2 \end{pmatrix}$$

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

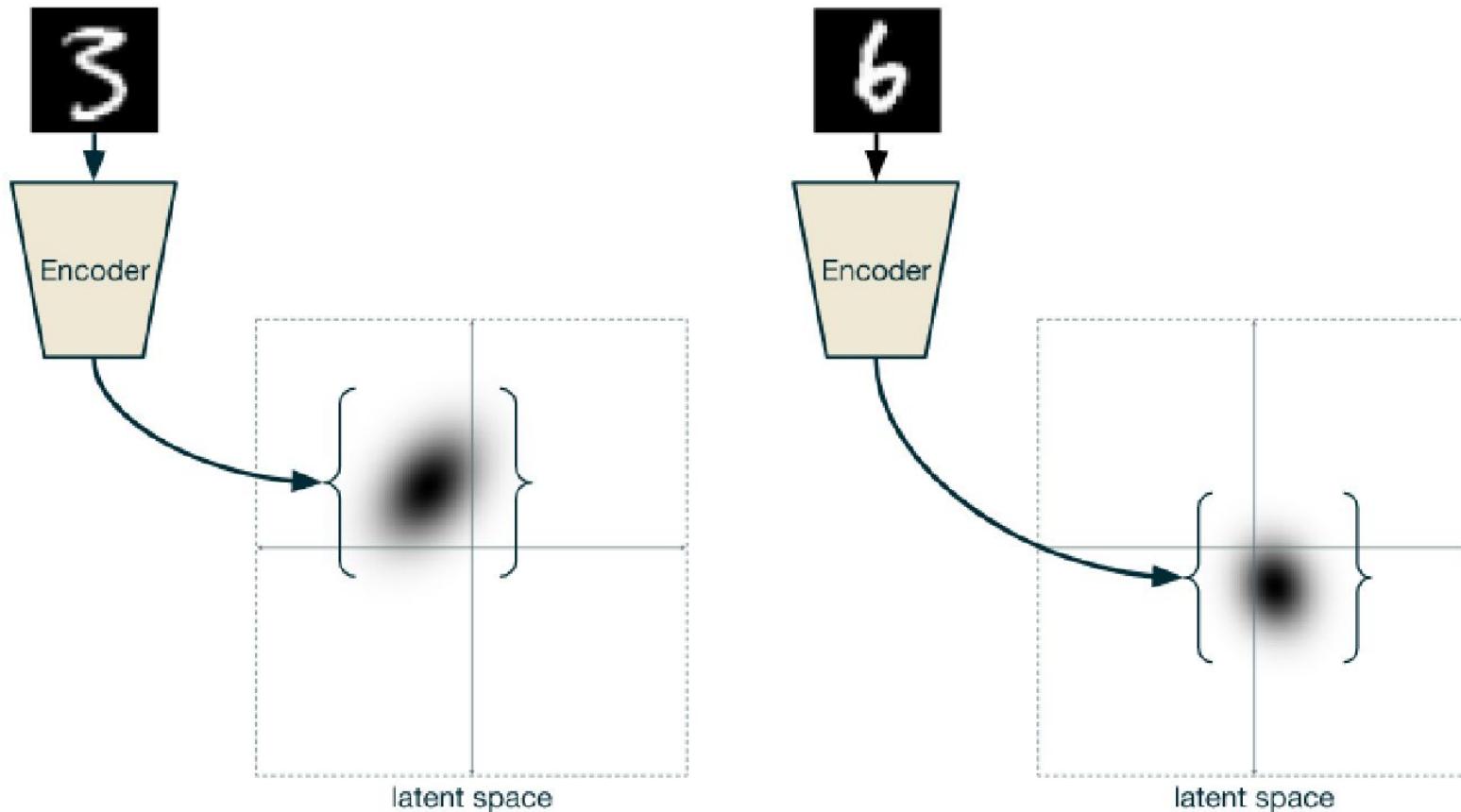
Autoencodeur variationnel



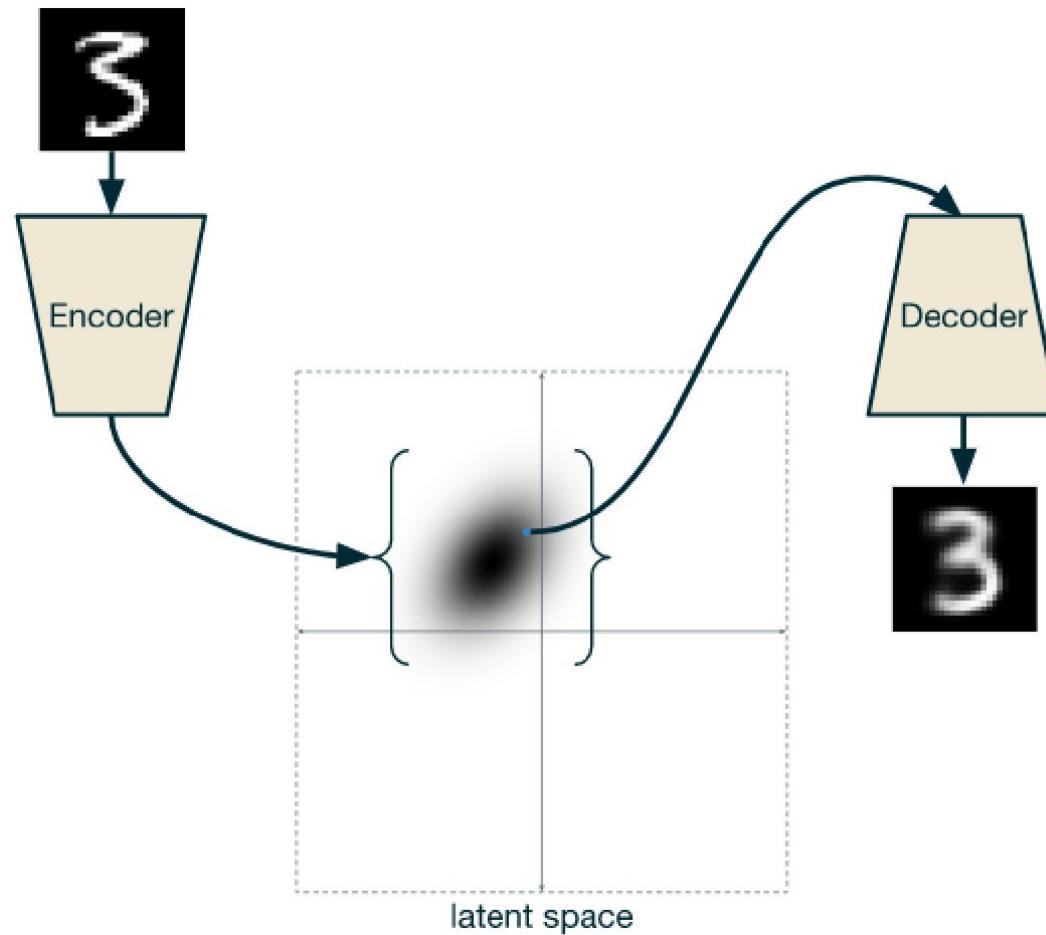
Autoencodeur variationnel



Autre façon de voir les choses...

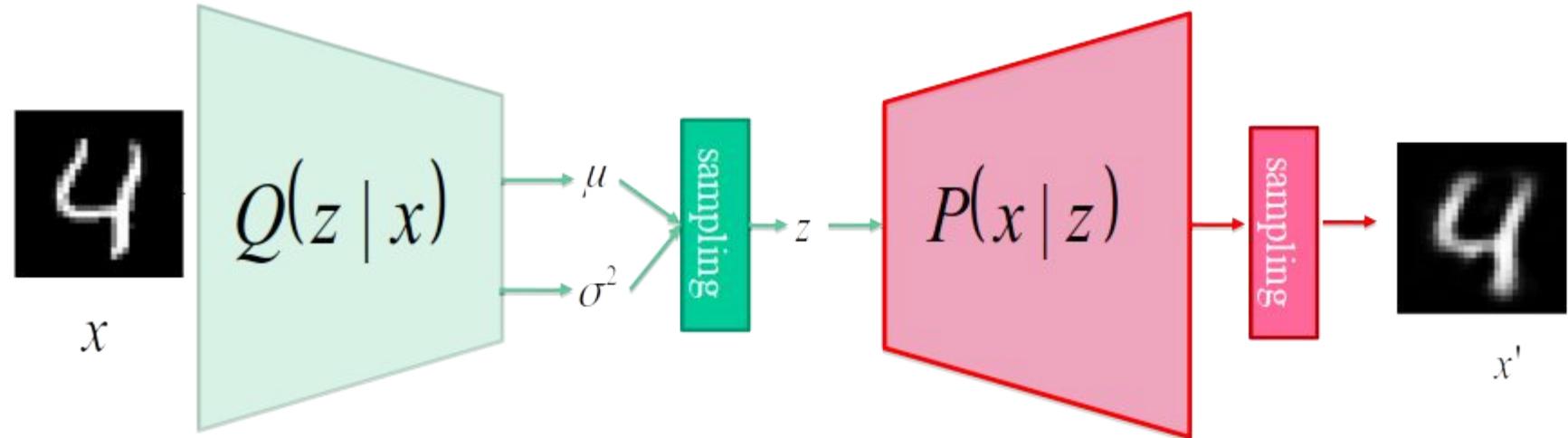


Autre façon de voir les choses...



Autoencodeur variationnel

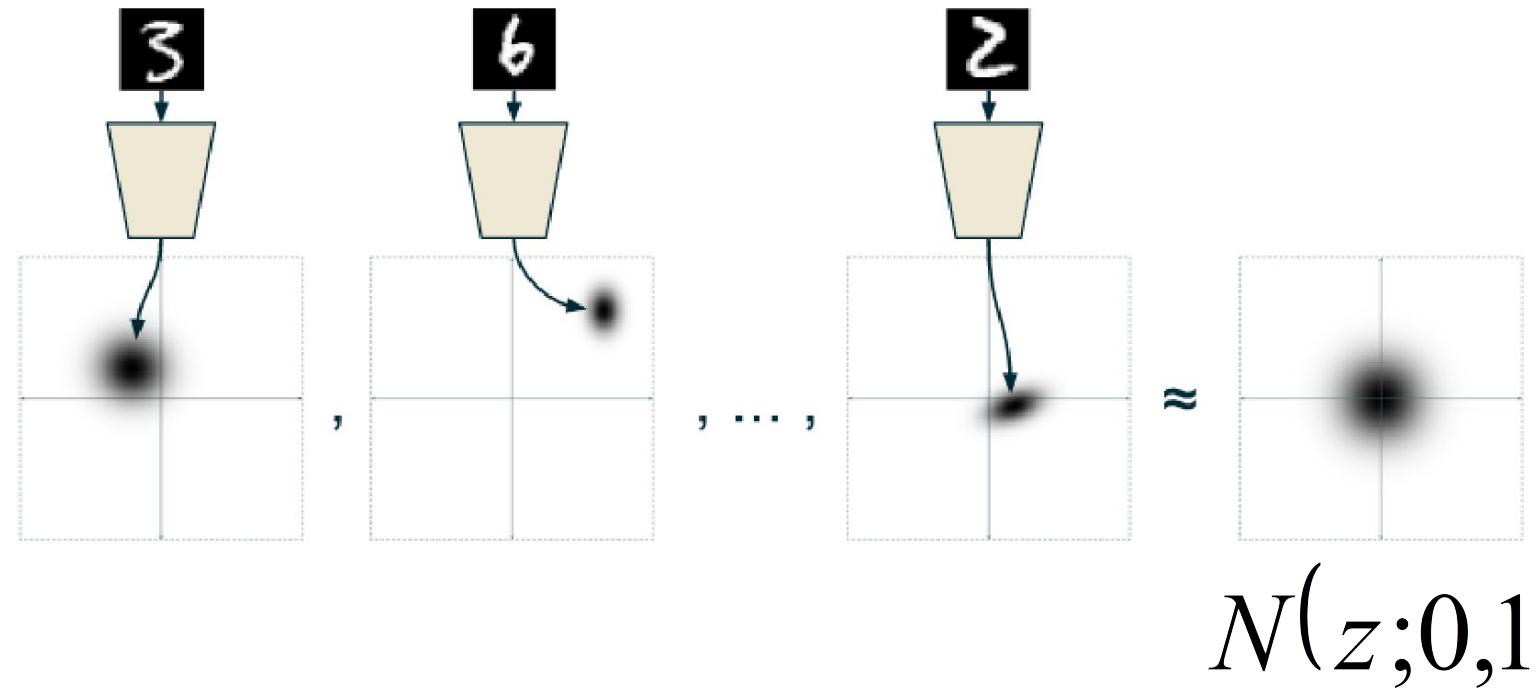
Remarque 5



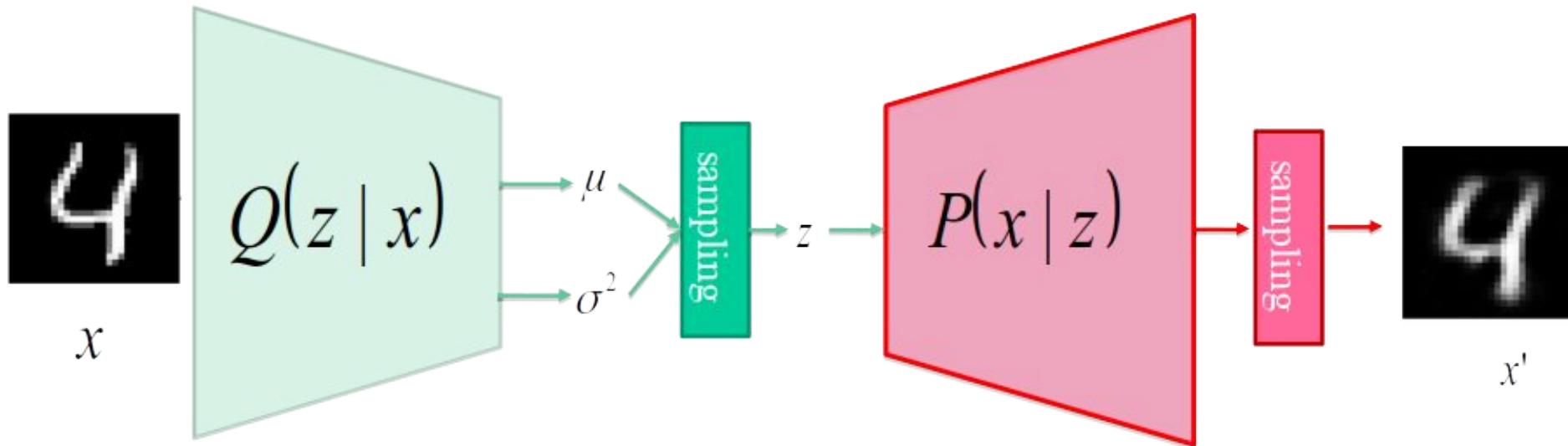
Distribution *a priori* de z : gaussienne centrée à 0 et de variance 1

$$P(z) = N(z; 0, 1)$$

Autre façon de voir les choses...



Autoencodeur variationnel



ELBO loss : Evidence Lower Bound

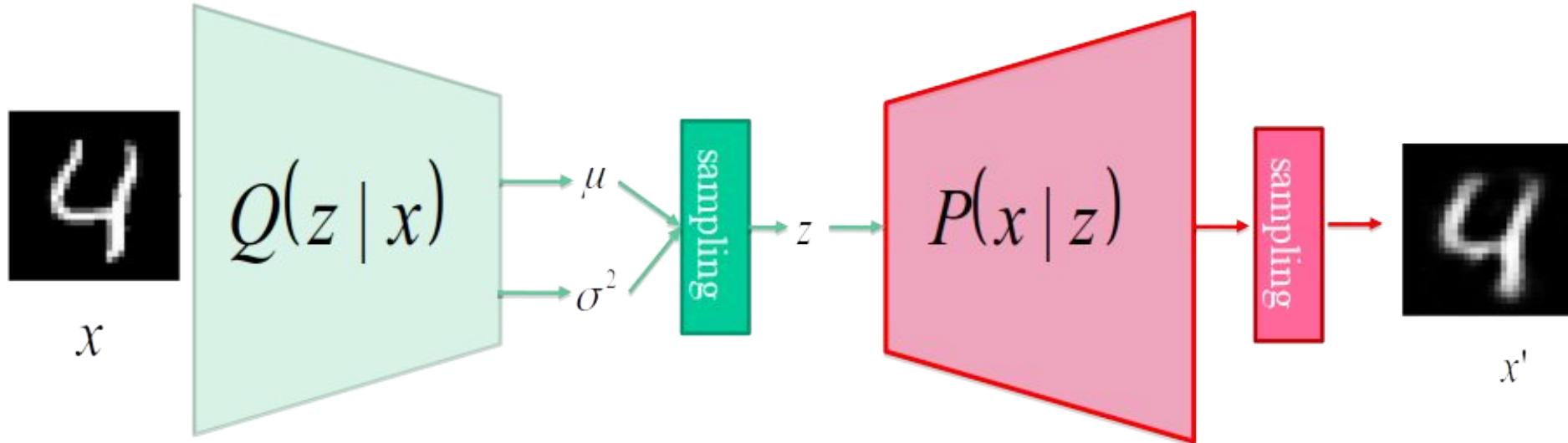
$$Loss = KL(N(z; 0, 1), N(z; \mu, \Sigma)) - \log(P(x | z))$$

Perte encodeur

Perte décodeur

Autoencodeur variationnel

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



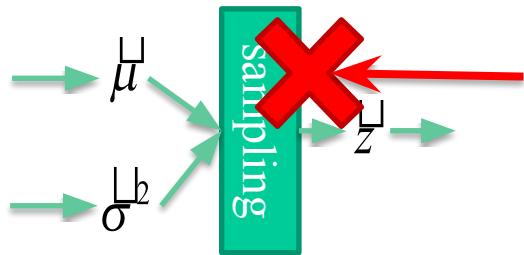
ELBO loss : Evidence Lower Bound

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

Perte encodeur

Autoencodeur variationnel

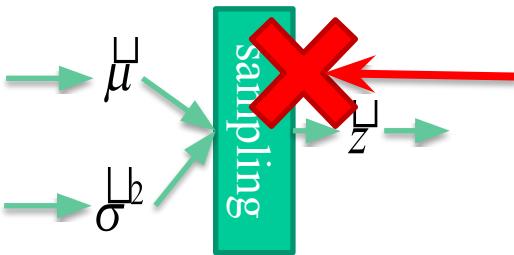
Remarque 6



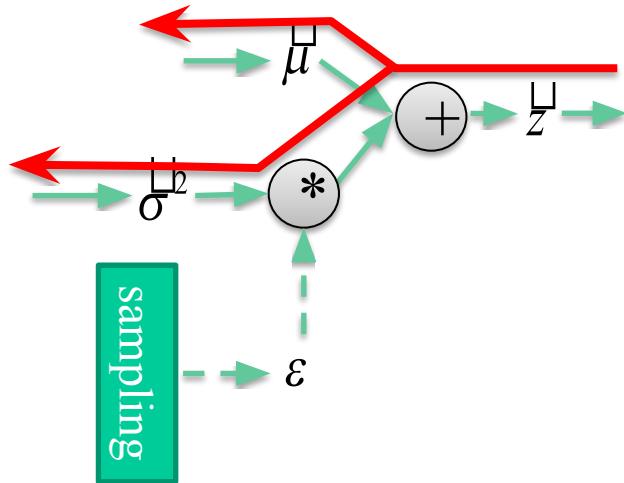
Pas de rétro-propagation à travers
un processus d'échantillonage
 $z \sim N(\mu, \Sigma)$

Autoencodeur variationnel

Remarque 6



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



Reparameterization trick

$$\underline{z} = \underline{\mu} + \varepsilon \underline{\sigma}$$

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

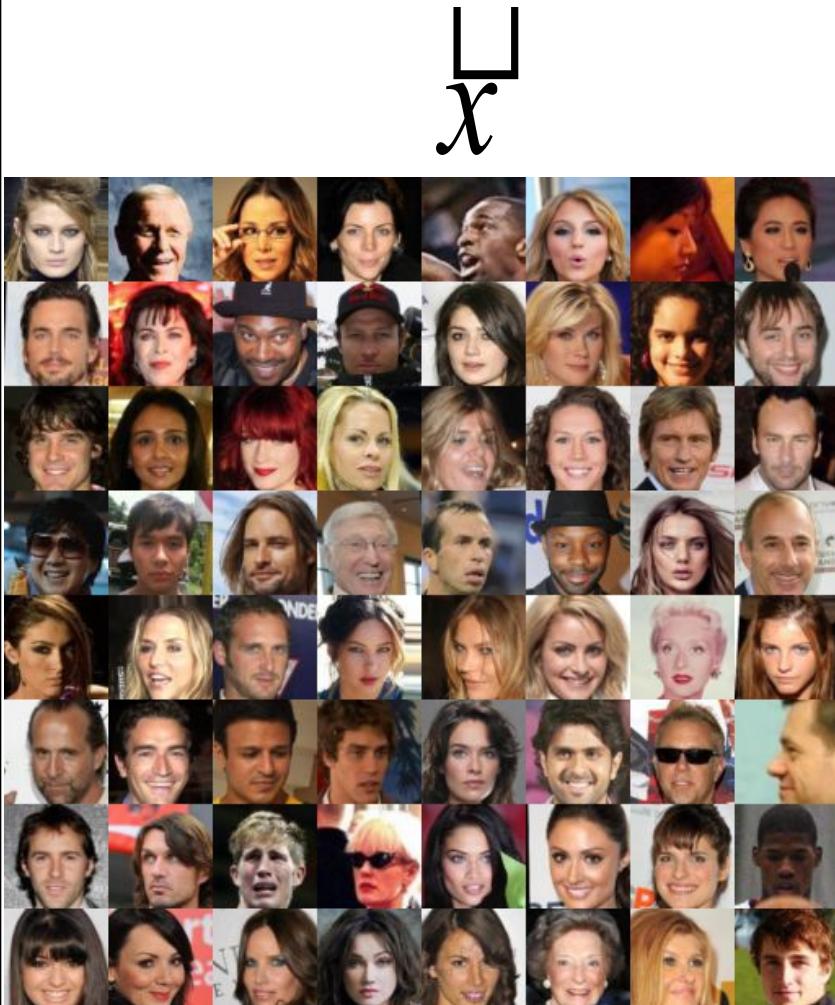
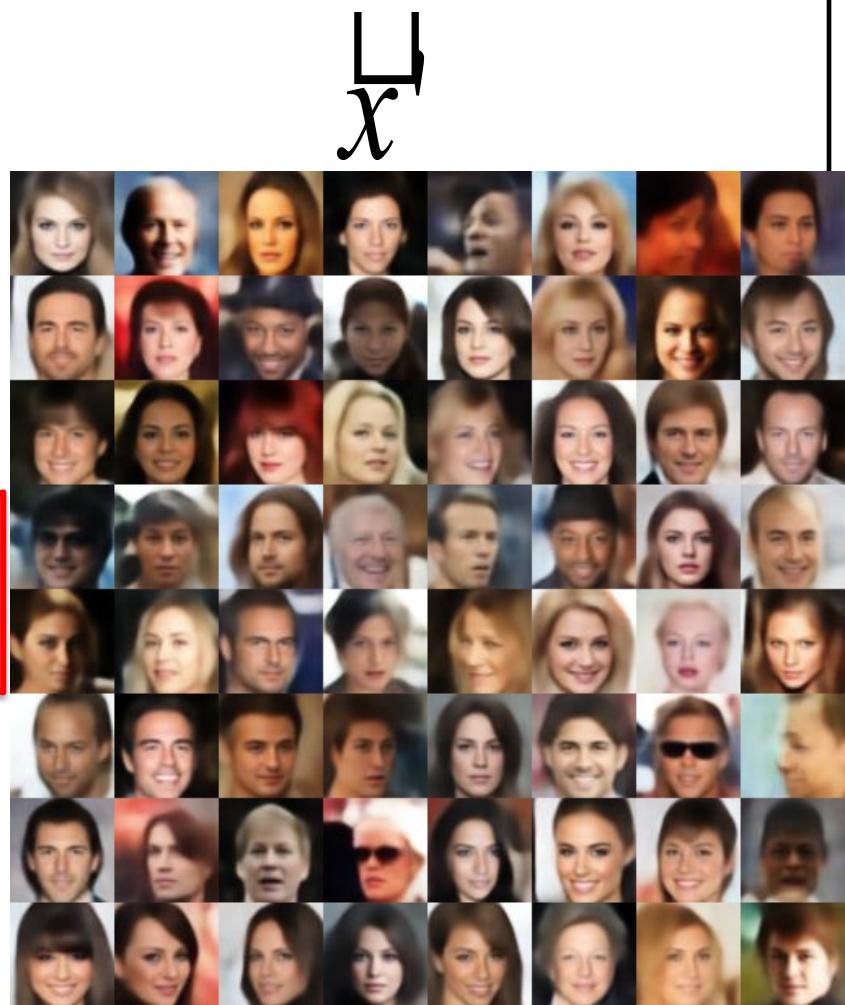
Autoencodeur variationnel jouet MNIST : d=32 dim

```
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())

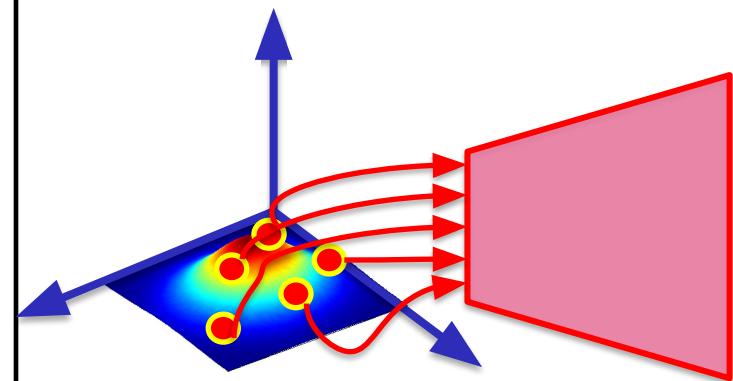
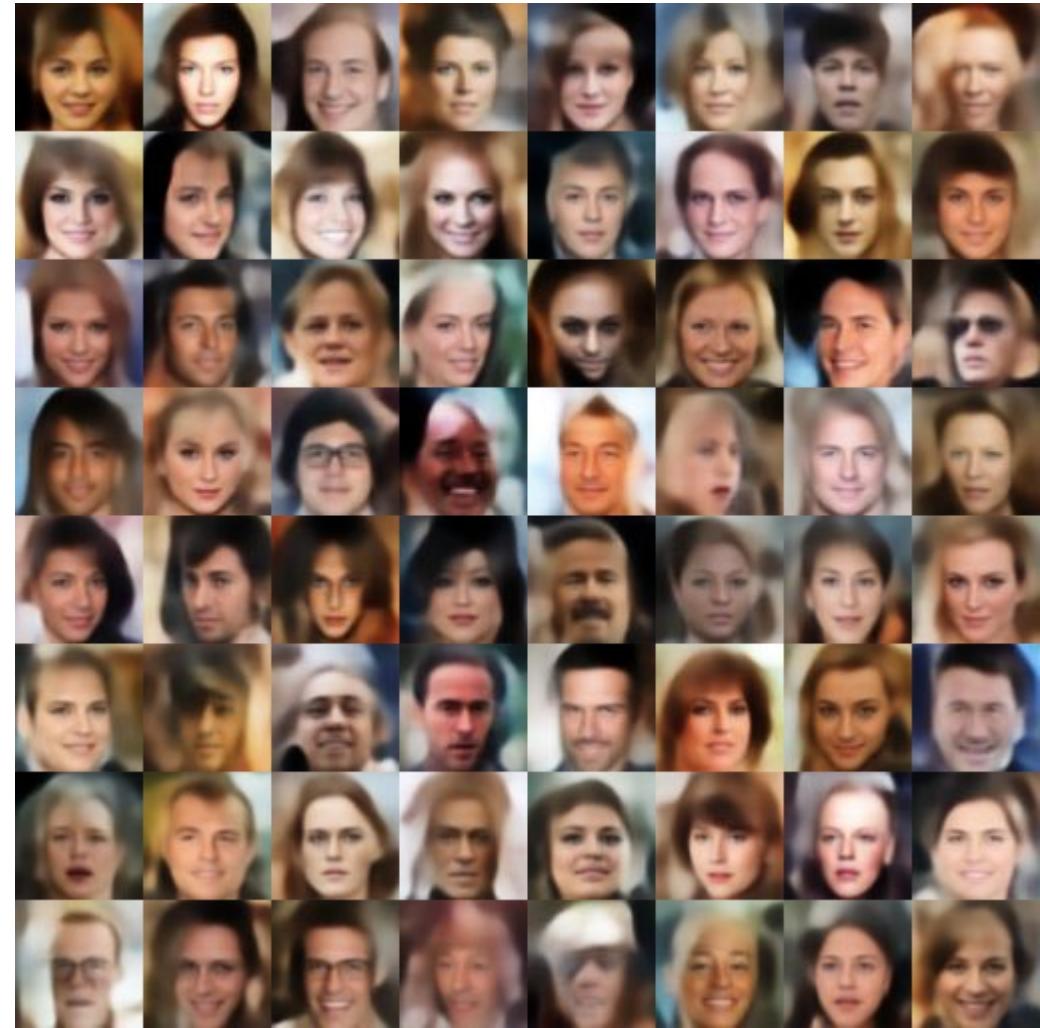
    return BCE + KLD
```

Ex.: base de données *CelebA*

 \bigsqcup_x  \bigsqcup_x

Ex.: base de données *CelebA*

Décodage d'échantillons aléatoires \mathbb{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 \hat{x} en échantillonnant $P(\hat{x})$

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



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

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

Modèle génératif

Distribution *a priori*



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

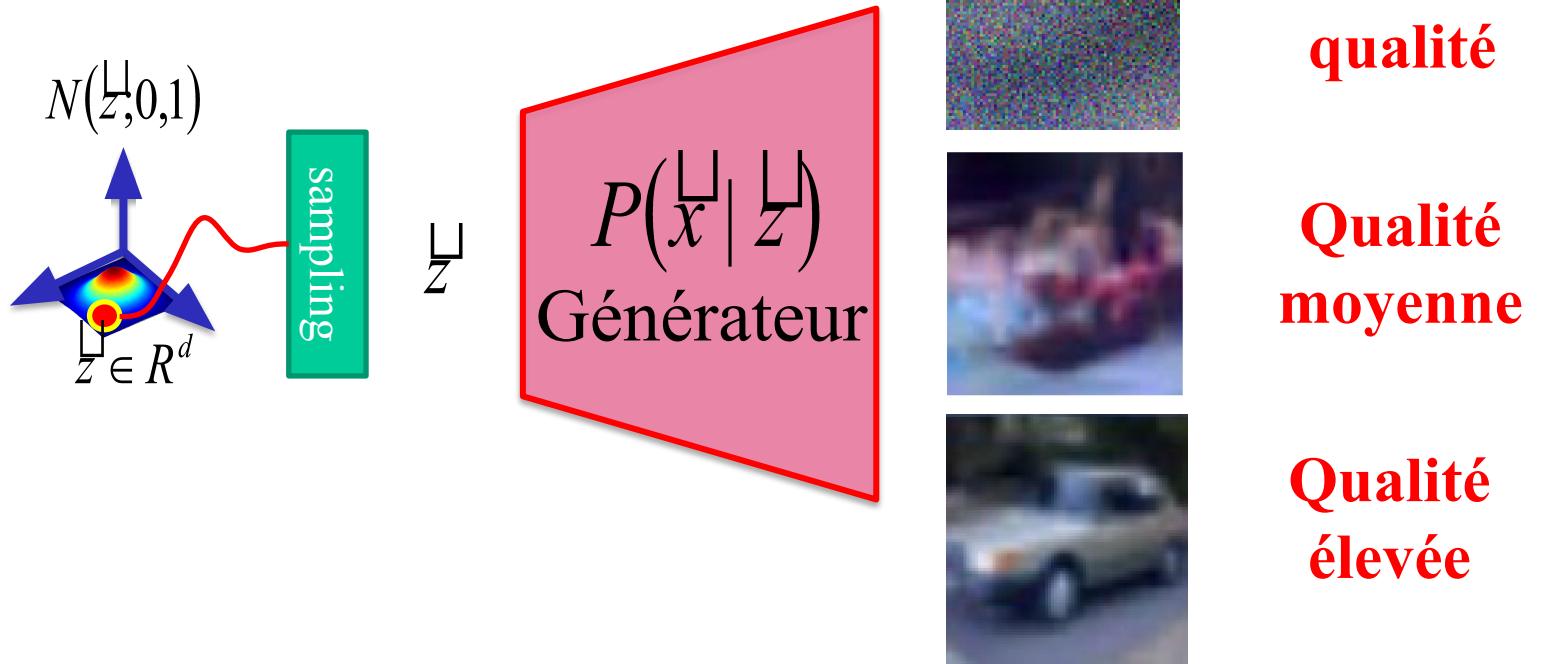
$$P(z) = N(z; 0, 1)$$

Comment estimer $P(x|z)$?

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



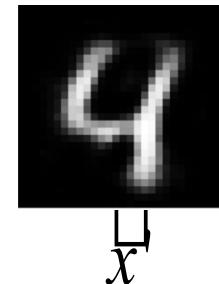
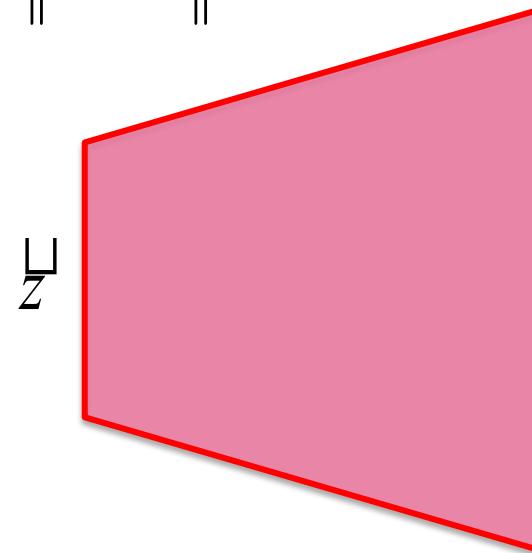
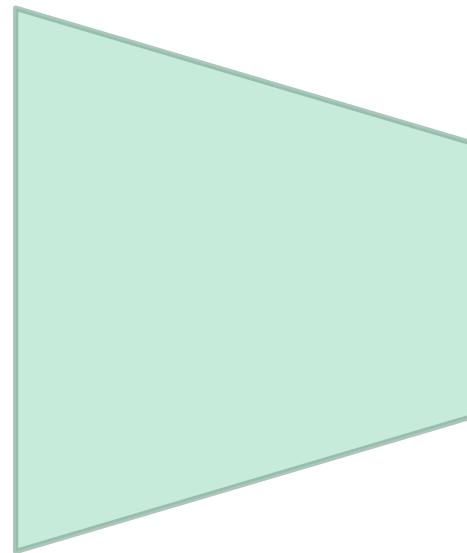
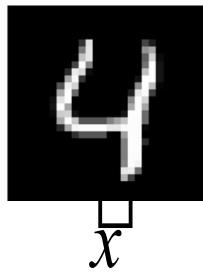
Dépendamment des performances du décodeur, les images générées
seront de qualité très variable.



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

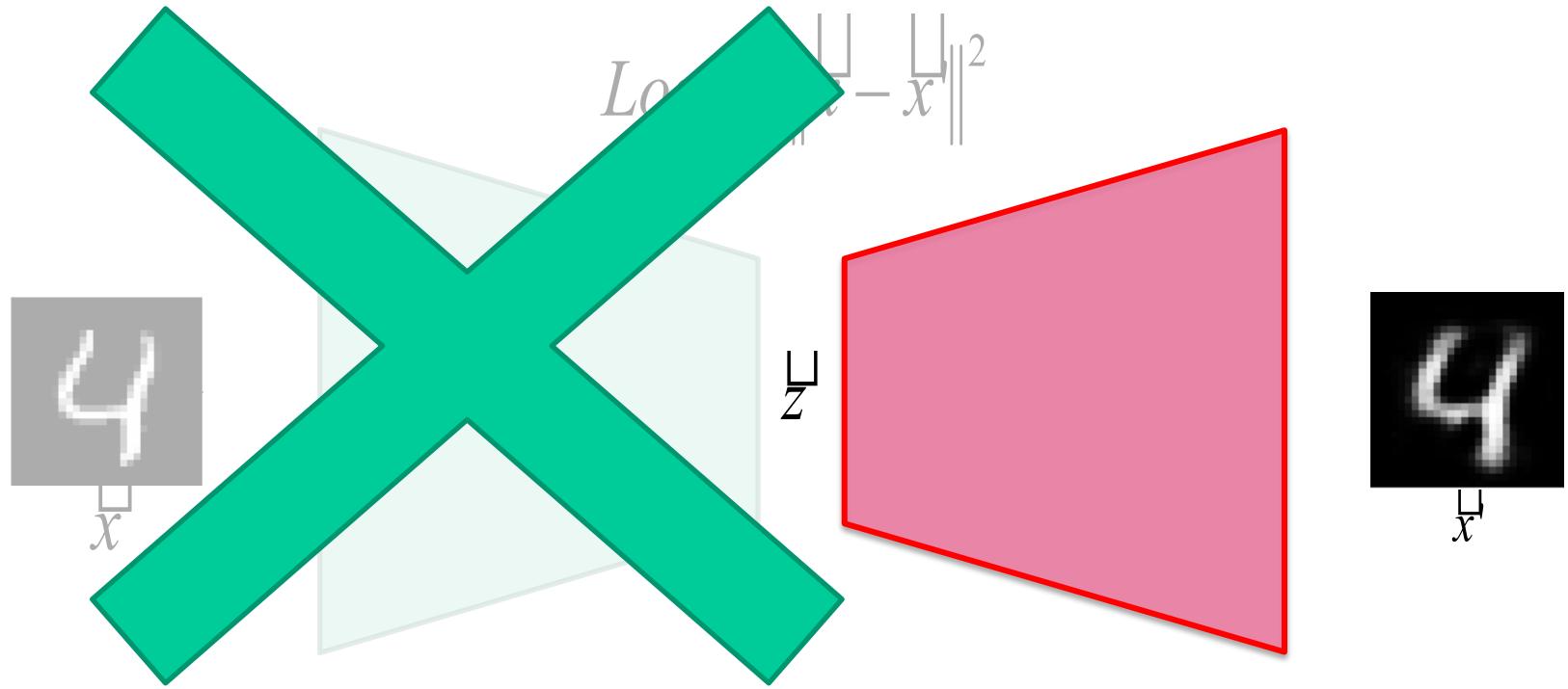
Pour un autoencodeur (variationnel ou non) c'est facile!
car on a un encodeur et une image de référence

$$Loss = \|\underline{x} - \underline{\hat{x}}\|^2$$

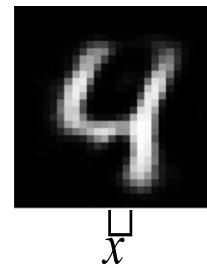


Pour entraîner un décodeur, il faut une **fonction de perte** (*loss*) qui mesure la qualité (degré de réalisme) des images produites

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



$$\mathbb{Z} \rightarrow G(\mathbb{L})$$



Loss sans cible de référence?



Approximer la perte à l'aide d'un

2^e réseau de neurones

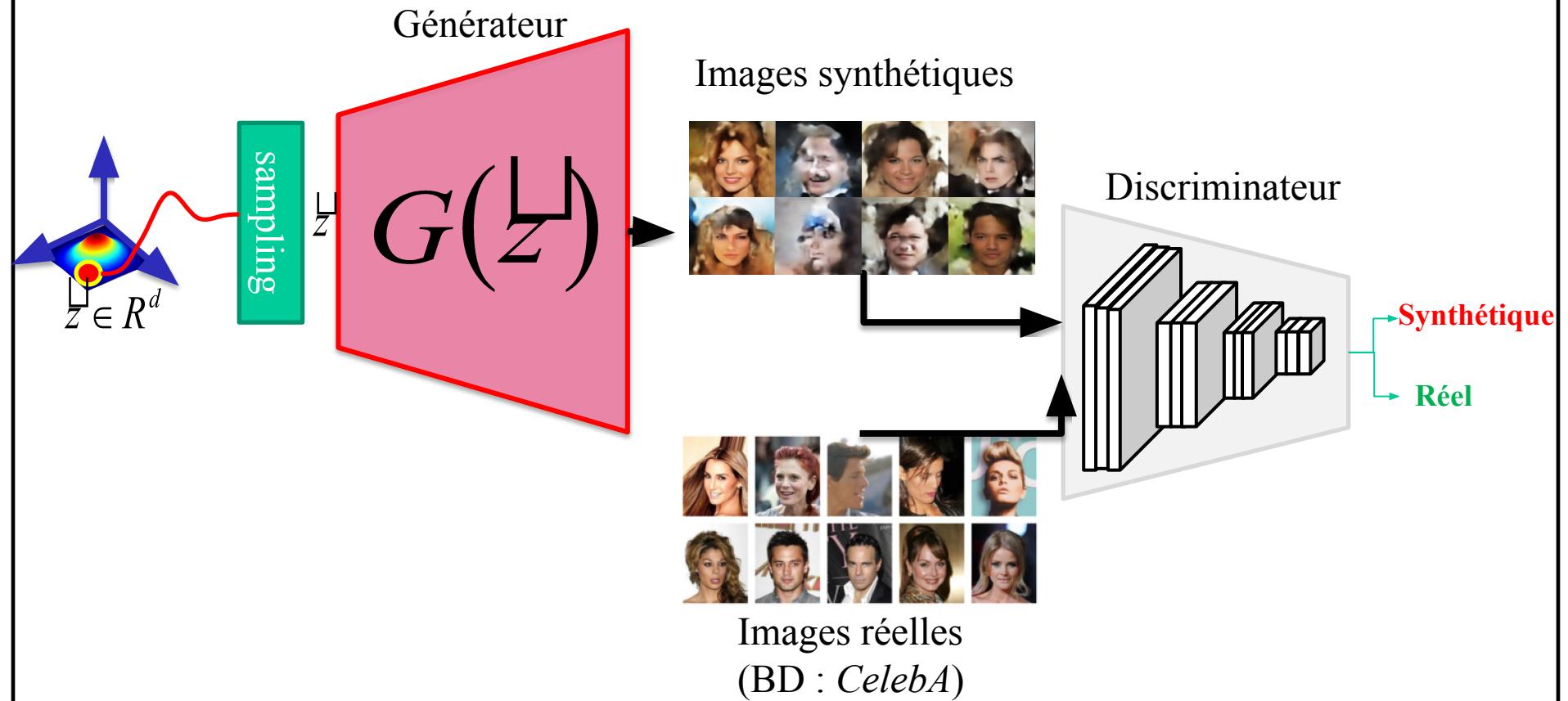
et d'une

Base de données de référence

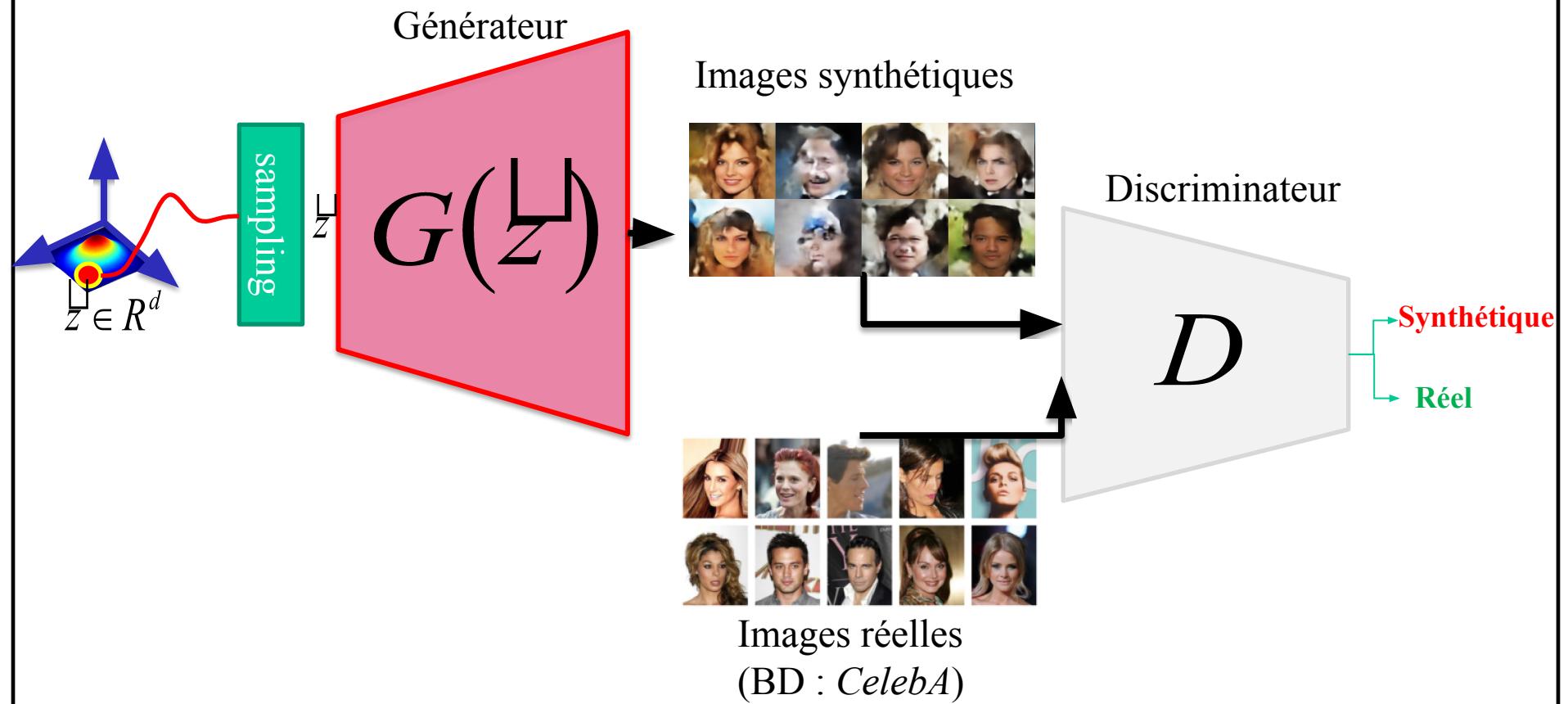
$\frac{1}{Z}$

référence?

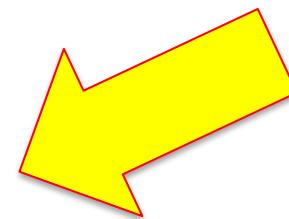
Pour entraîner un décodeur, il faut une **fonction de perte** (loss) qui mesure la qualité des images produites



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



Données étiquetées

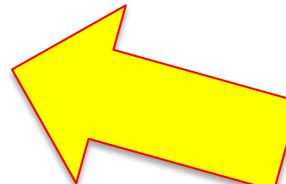


Produites par le
générateur

$t = 0$ ('synthétique')



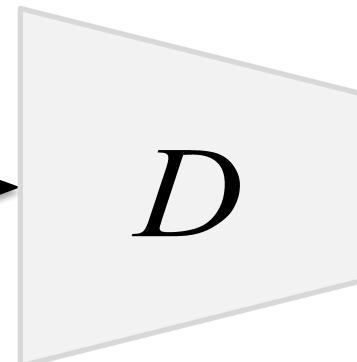
$t = 1$ ('réel')



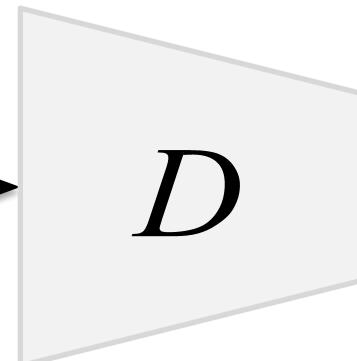
Issues d'une vraie BD

Deux réseaux aux **objectifs différents** :

Discriminateur : différentie les images synthétiques des images réelles



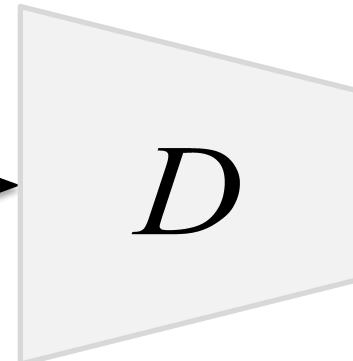
Synthétique



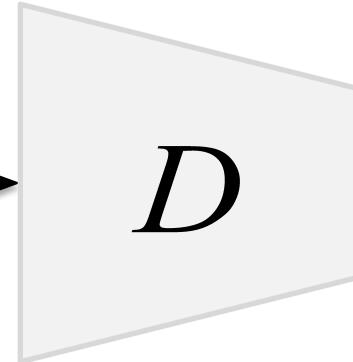
Réel

Discriminateur : classifieur binaire (régression logistique)

=> Perte l'entropie croisée



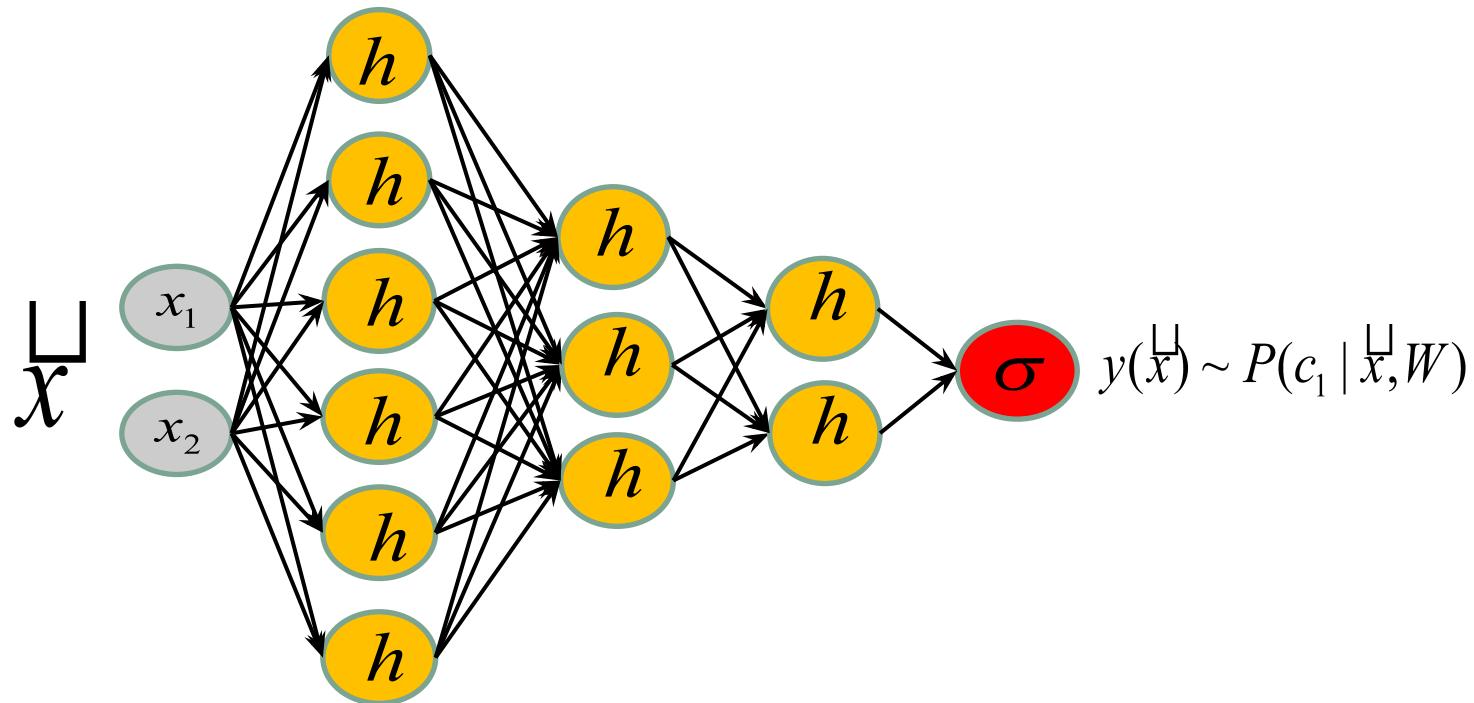
Synthétique



Réel

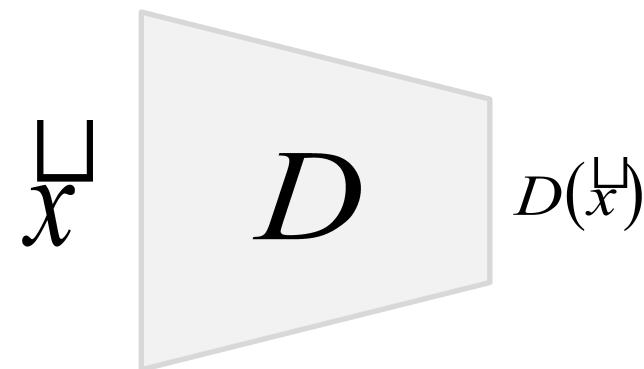
Rappel, entropie croisée pour une régression logistique binaire:

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



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

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



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

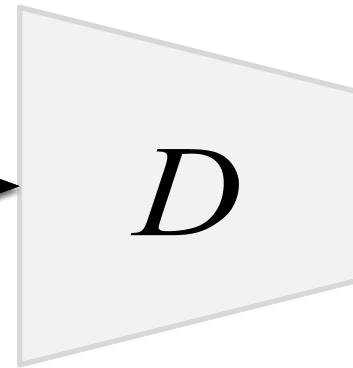
$$L_D = \frac{1}{N} \sum_i -t_i \ln(D(\underline{x}_i)) - (1 - t_i) \ln(1 - D(G(\underline{z}_i)))$$



$$G(\underline{z})$$



$$\underline{x}$$

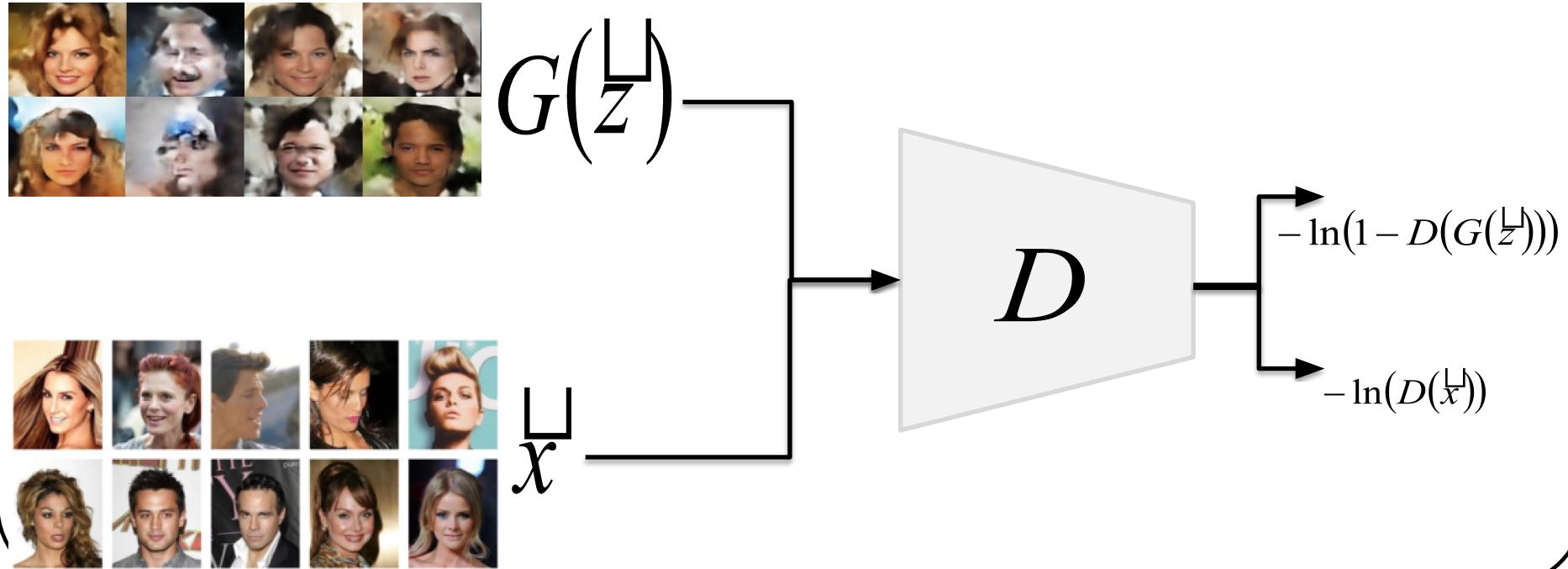


$$\begin{aligned} &D(G(\underline{z})) \\ &D(\underline{x}) \end{aligned}$$

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

$$L_D = -\frac{1}{N_{reel}} \sum_i \ln(D(\boxed{x}_i)) - \frac{1}{N_{syn}} \sum_j \ln(1 - D(G(\boxed{z}_j)))$$

Perte images réelles
Perte images 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

$$L_D = -\frac{1}{N_{reel}} \sum_i \ln(D(\boxed{x}_i)) - \frac{1}{N_{syn}} \sum_j \ln(1 - D(G(\boxed{z}_j)))$$

Perte images réelles
Perte images synthétiques

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

Objectif du discriminateur

Paramètres du discriminateur

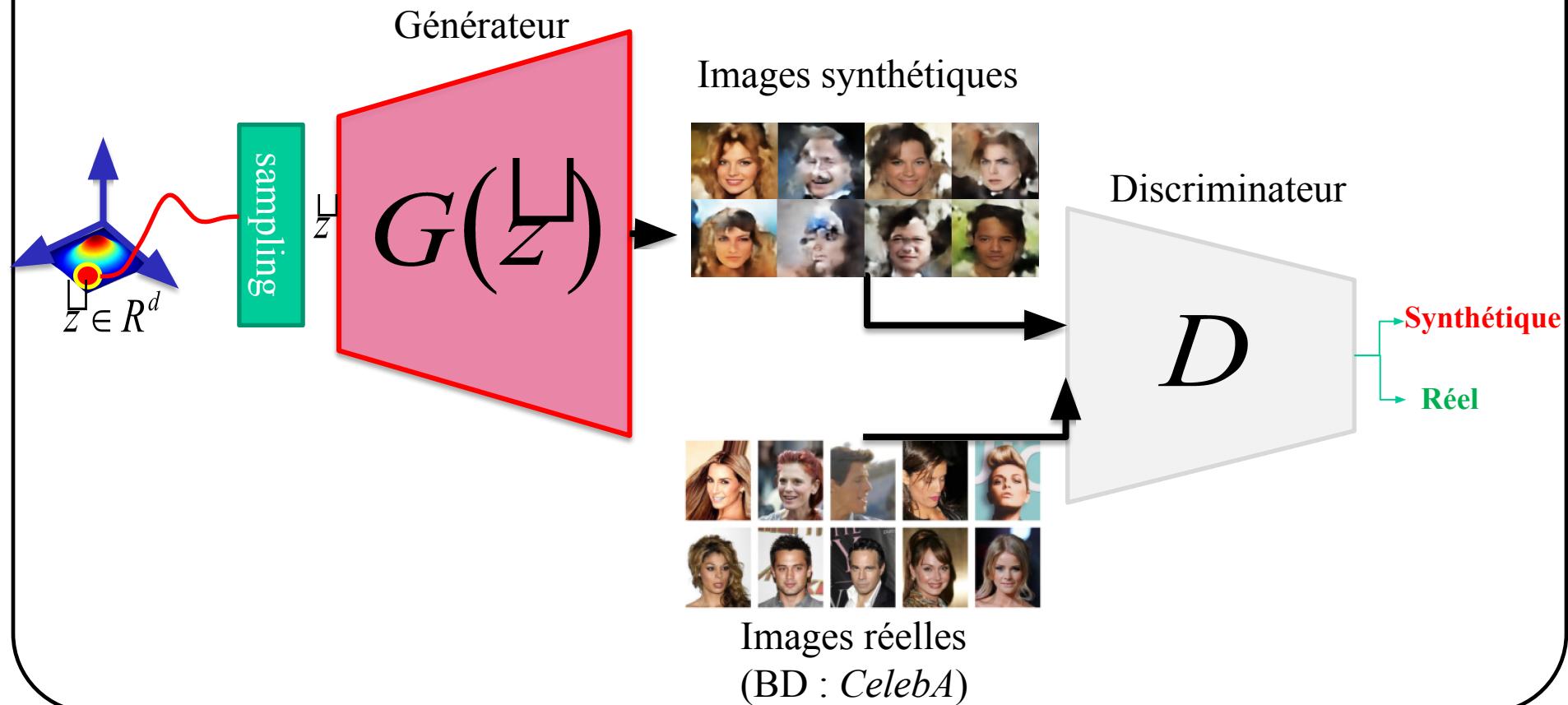
$$W_D = \arg \min_{W_D} -IE_{\bar{x} \sim P_{reel}} [\ln(D(\bar{x}))] - IE_{\bar{z} \sim P_z} [\ln(1 - D(G(\bar{z})))]$$

Ou encore, de façon équivalente

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

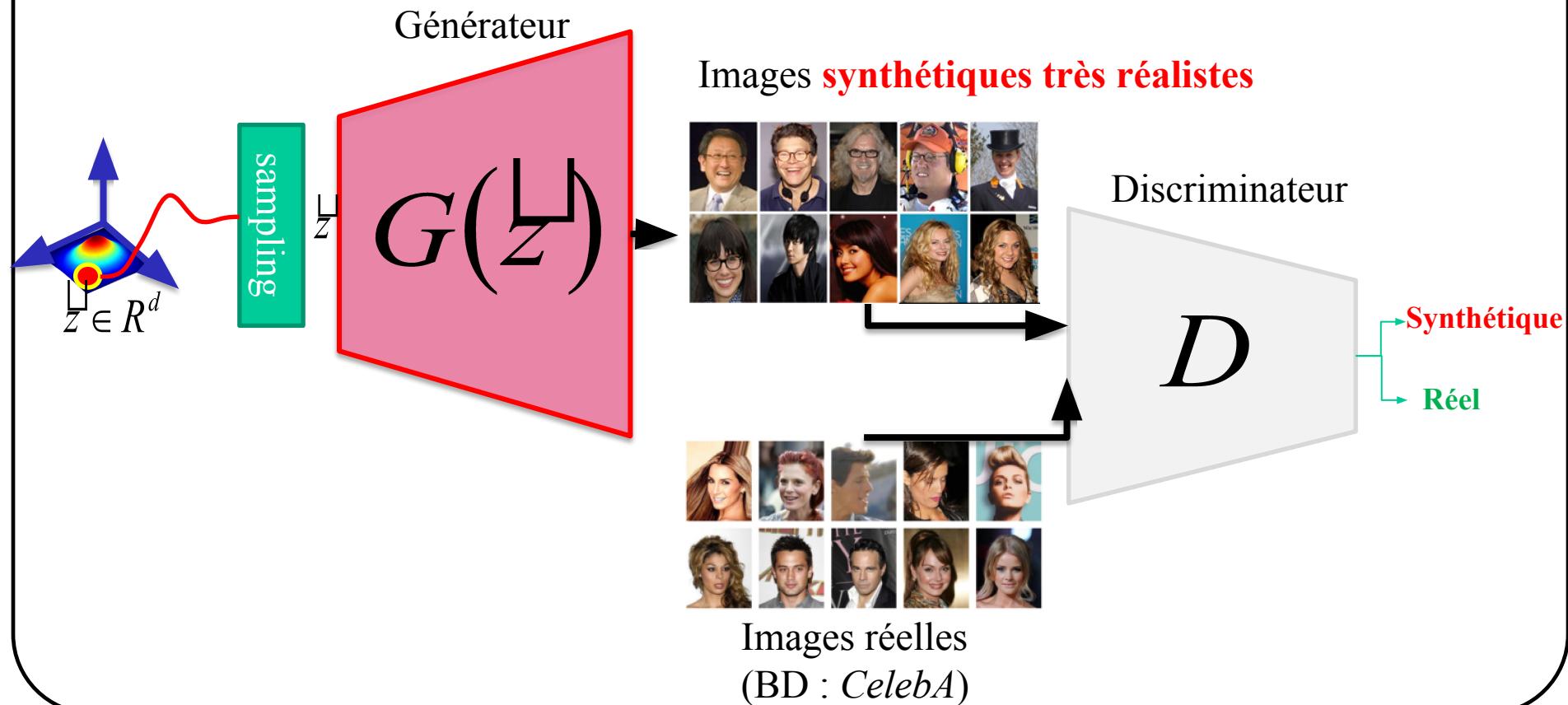
Objectif du générateur

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

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

Objectif du générateur :

produire des images synthétiques indistinguables des images réelles

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

« 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

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

on maximise plutôt celle-ci

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

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:

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

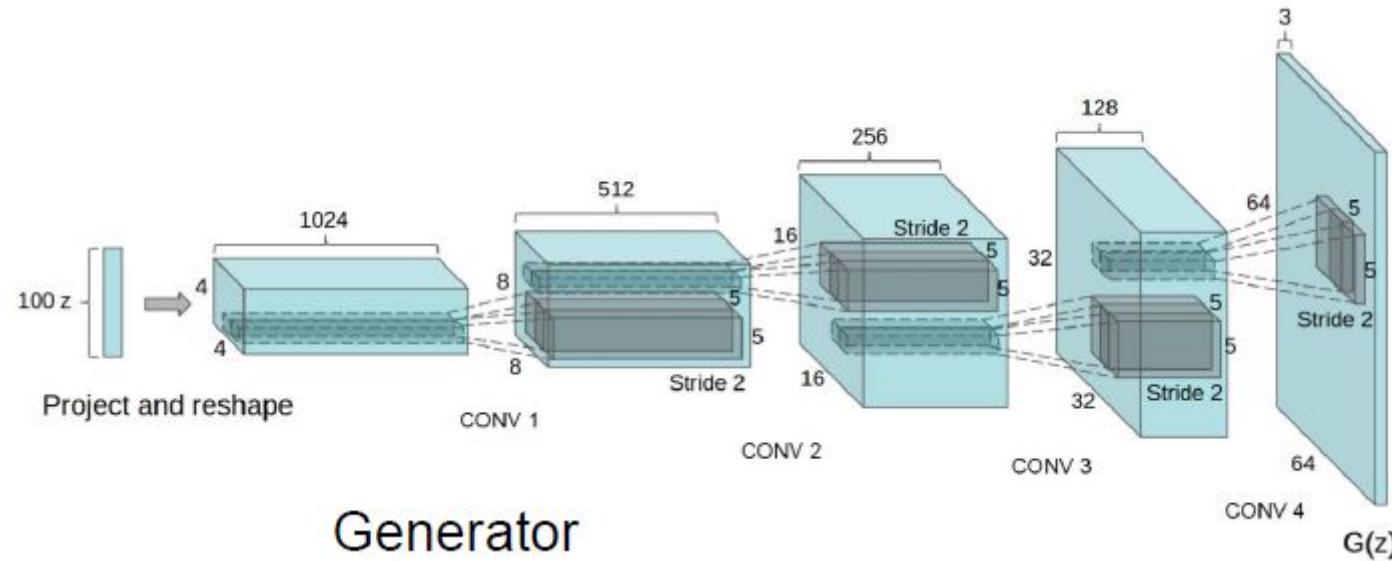
end for

- 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):

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

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

- Conv
- Pool

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

Recom

- Conv
- Leaky ReLU partout

Autre recommandations

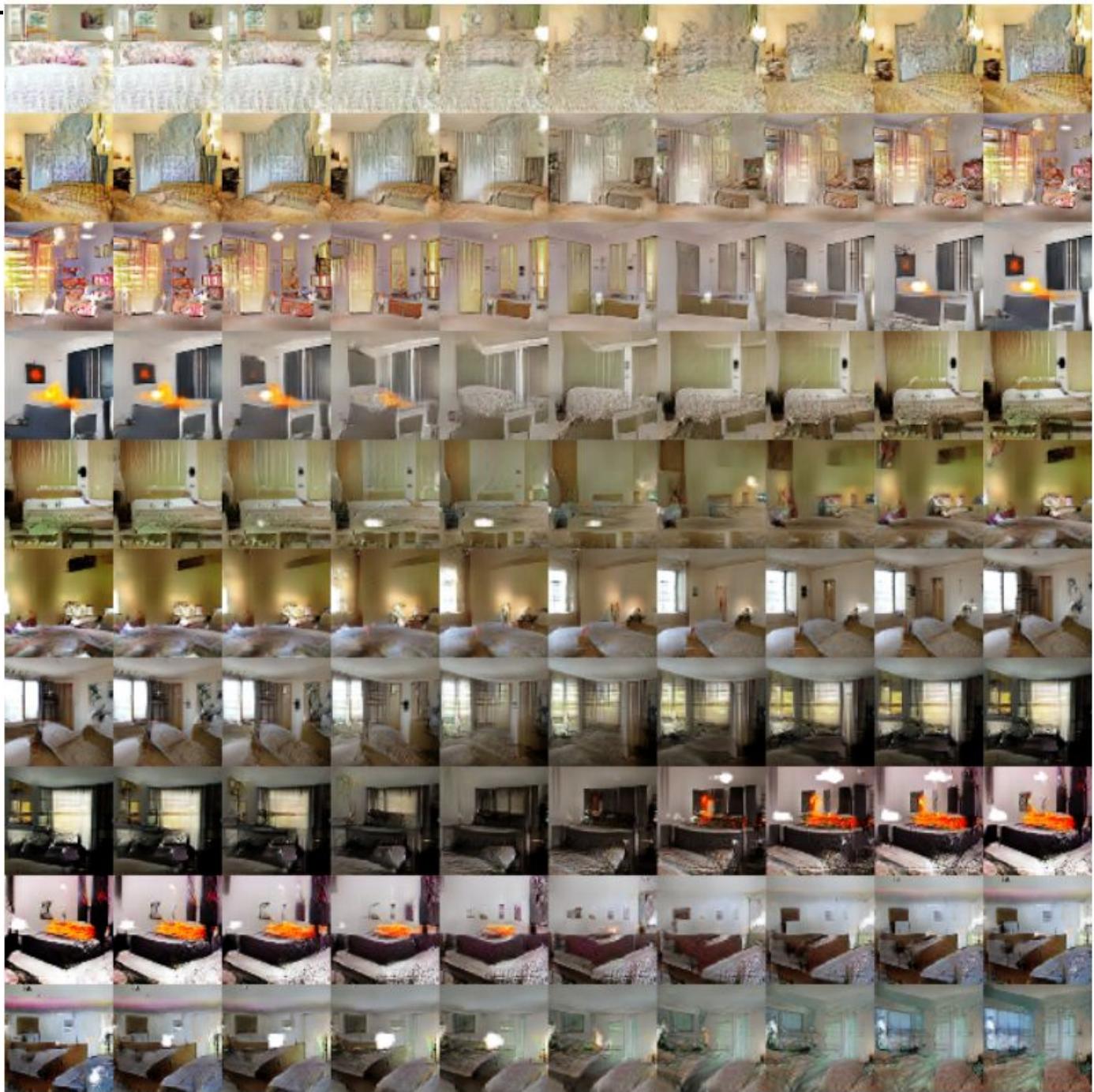
- BatchNorm partout
- Pas de FC, juste des conv

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

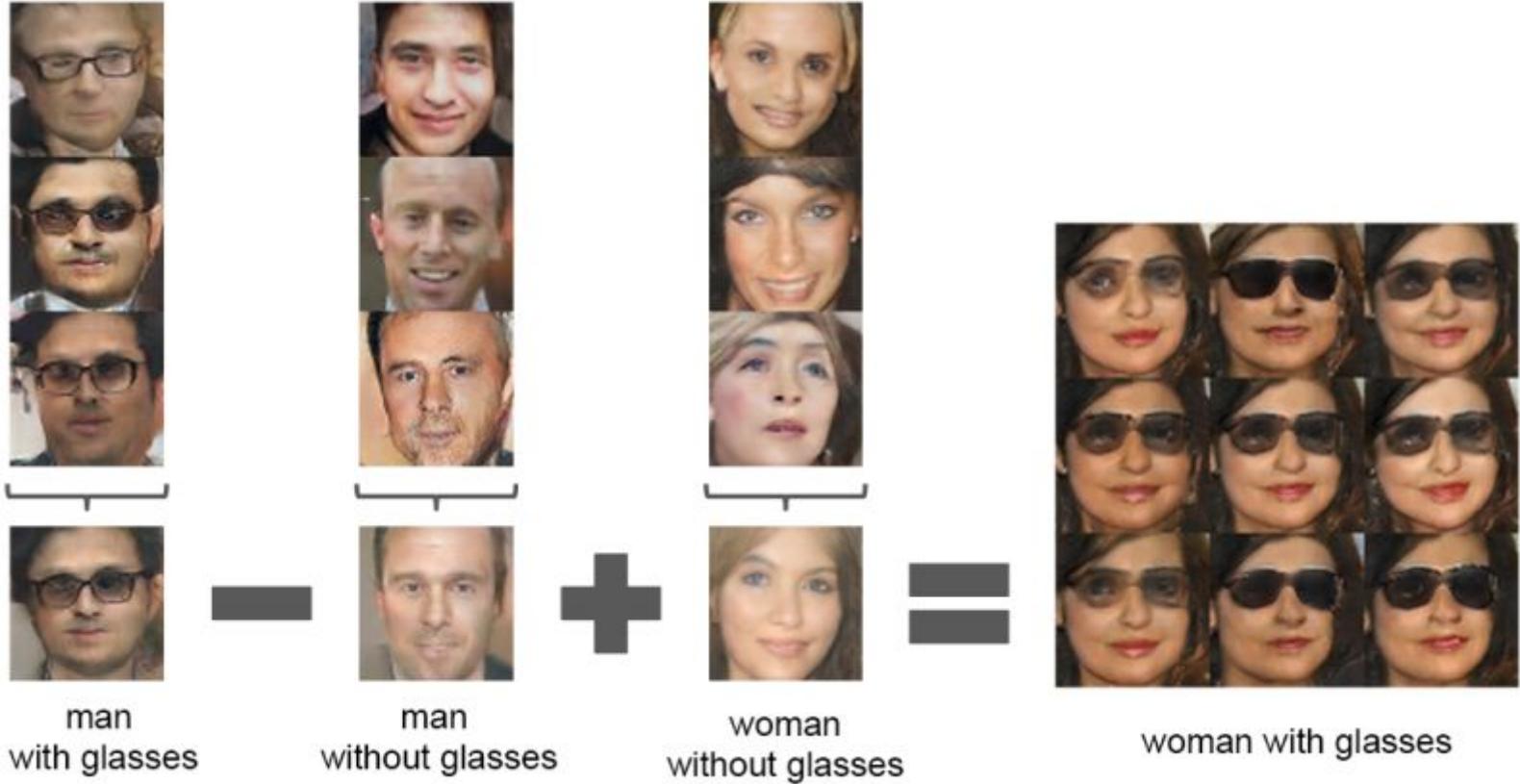
Deep Convolution Generative Adversarial Net (DCGAN)



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

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

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

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

LS GAN



(a) Generated by LSGANs.

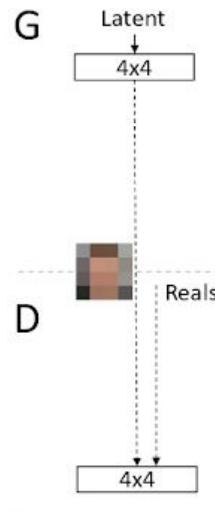


(b) Generated by DCGANs (Reported in [11]).

progressive GAN

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

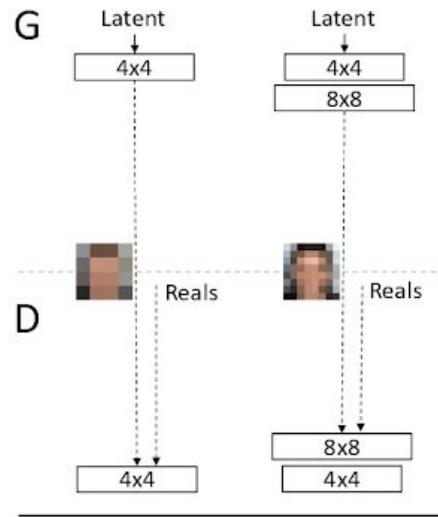
On commence avec des images de **faible résolution : 4x4 pixels**



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

progressive GAN

Et progressivement, on augmente la résolution de l'image

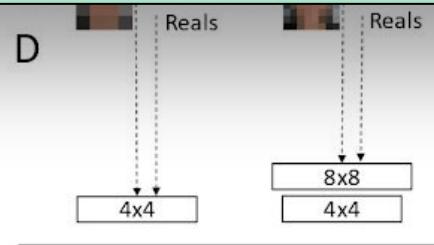


“Progressive GAN” Karras et al. ICLR’18

progressive GAN

Et progressivement on augmente la résolution de l'image

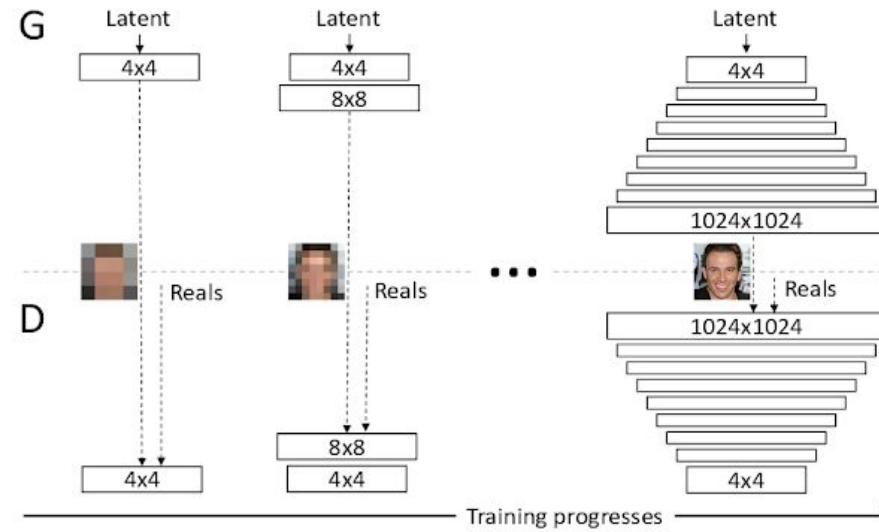
“Progressive Growing GAN requires that the capacity of both the generator and discriminator model be expanded by adding layers during the training process”



“Proggessive GAN” Karras et al. ICLR’18

progressive GAN

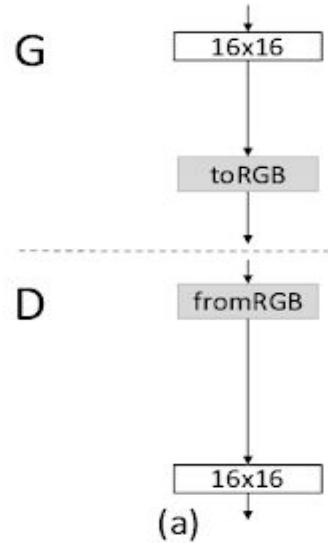
Et progressivement, chaque couche qu'on ajoute vient bonifier la couche précédente : cela se fait à l'aide d'une **opération « résiduelle »**.



“Progressive GAN” Karras et al. ICLR’18

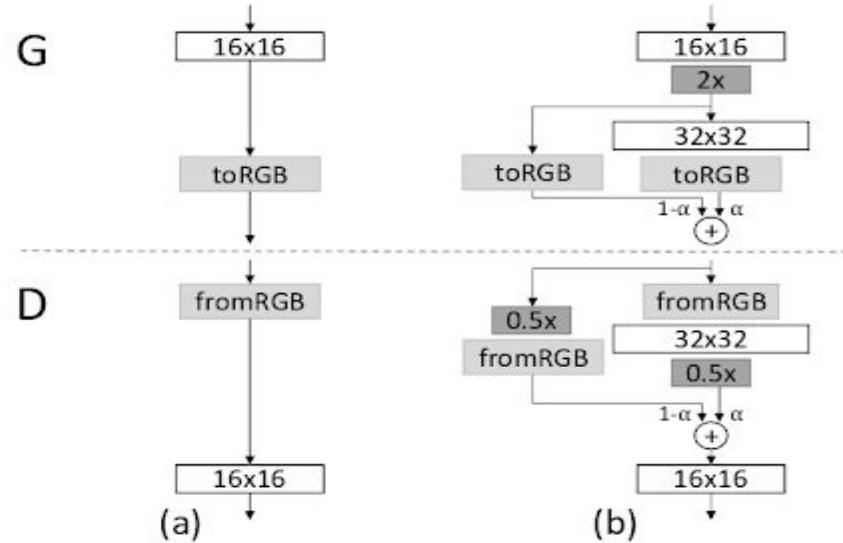
Ajout de couches

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



Ajout de couches

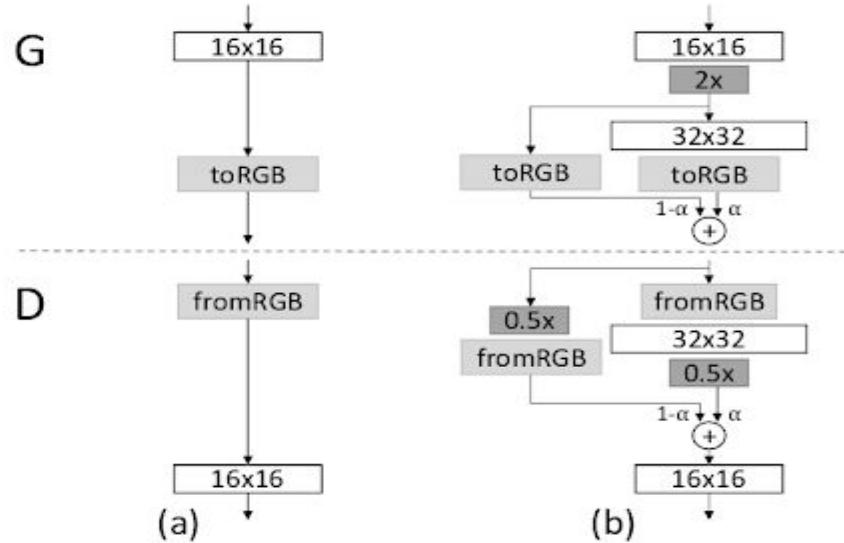
... On **ajoute une nouvelle couche** de résolution 2Rx2D (ici 32x32) au générateur ET au discriminateur.



Ajout de couches

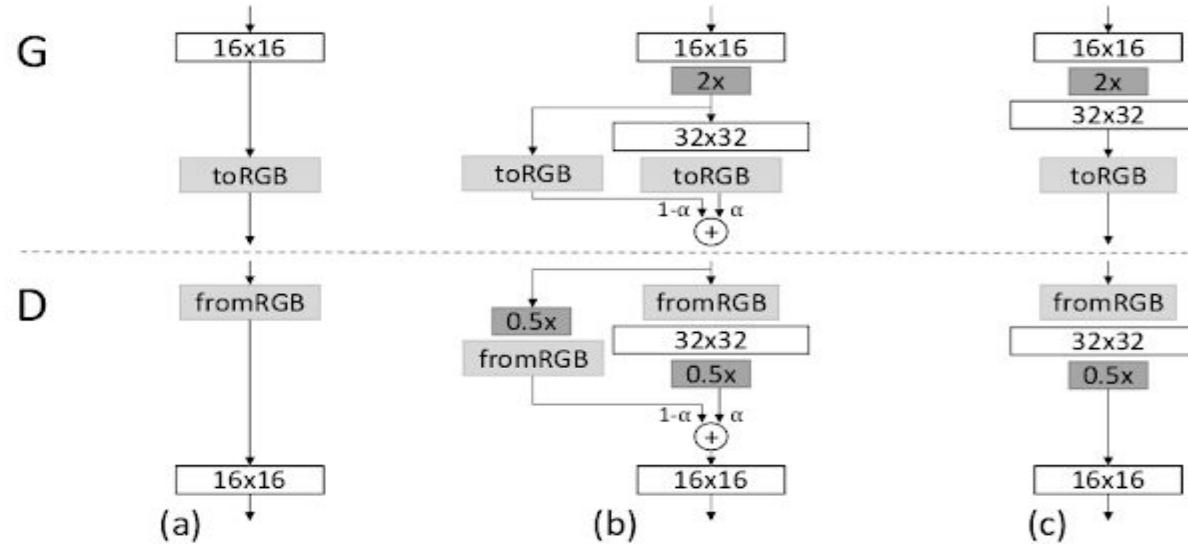
... mais pour éviter un choc, on ajoute une **composante résiduelle** comprenant un facteur α

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



Ajout de couches

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



“Progressive GAN” Karras et al. ICLR’18

Generator	Act.	Output shape	Params		Discriminator	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			23.1M		Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
					Fully-connected	linear	1 × 1 × 1	513
					Total trainable parameters			23.1M

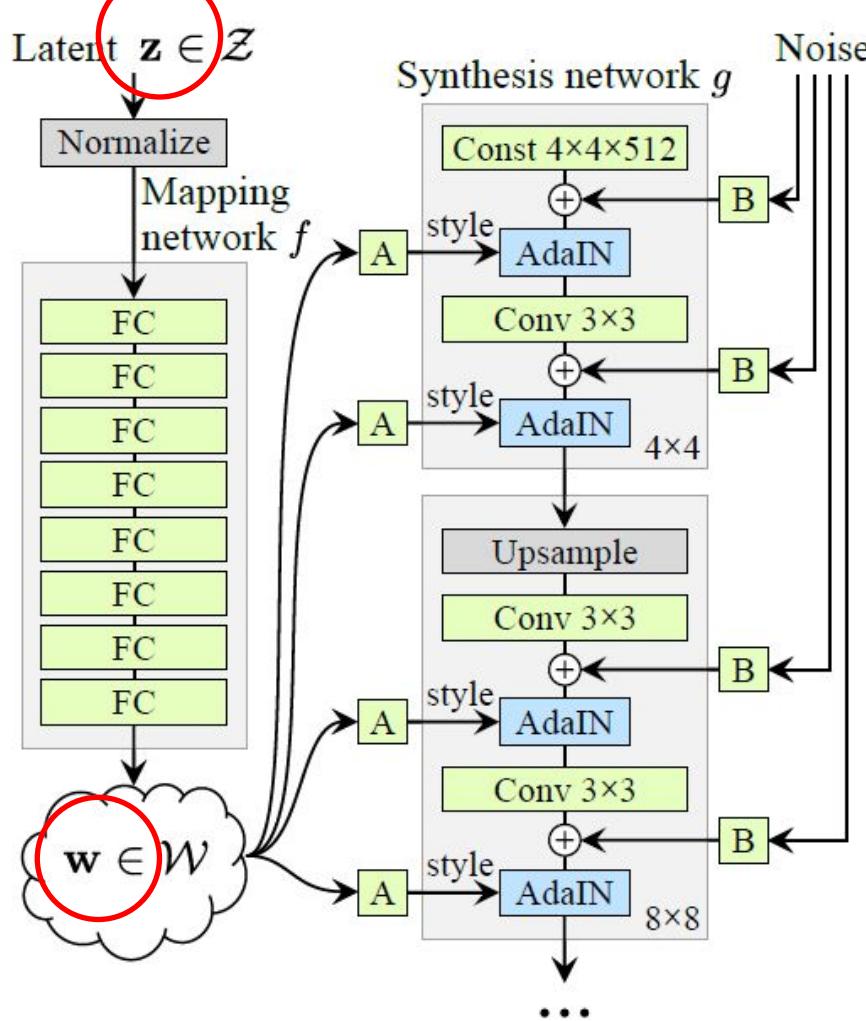
Table 2: Generator and discriminator that we use with CELEBA-HQ to generate 1024×1024 images.

“Progressive GAN” Karras et al. ICLR’18



<https://youtu.be/XOxxPcy5Gr4>

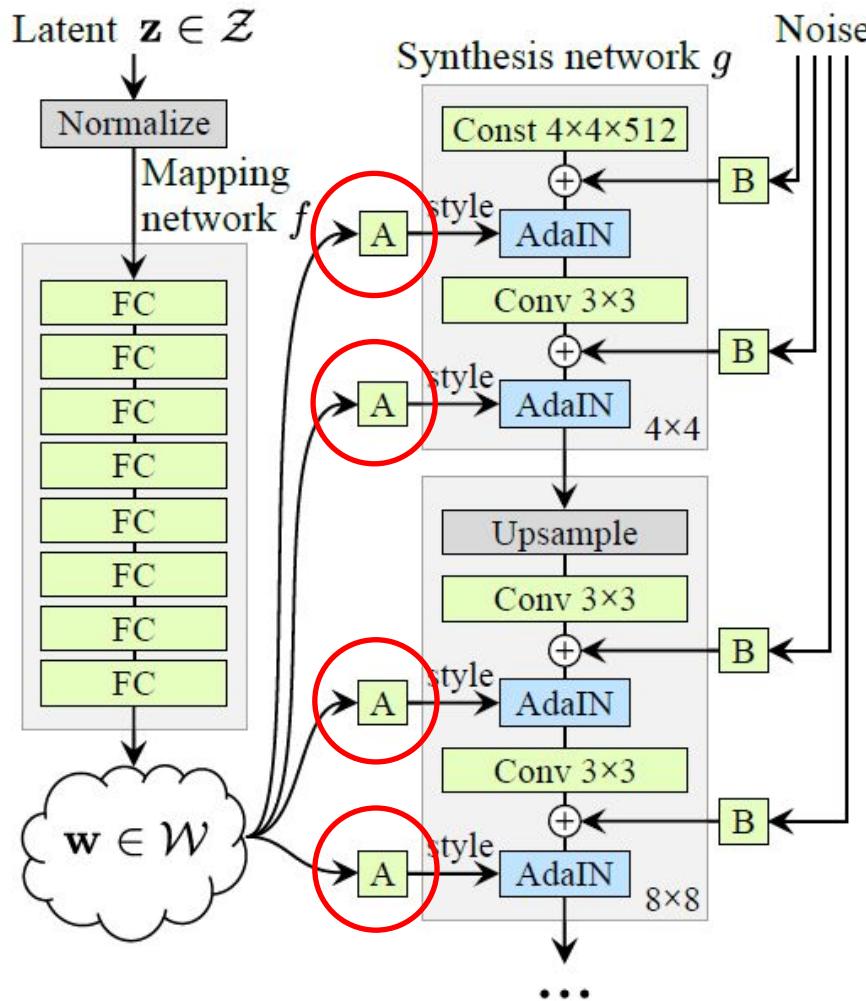
Style GAN



(b) Style-based generator

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

Style GAN

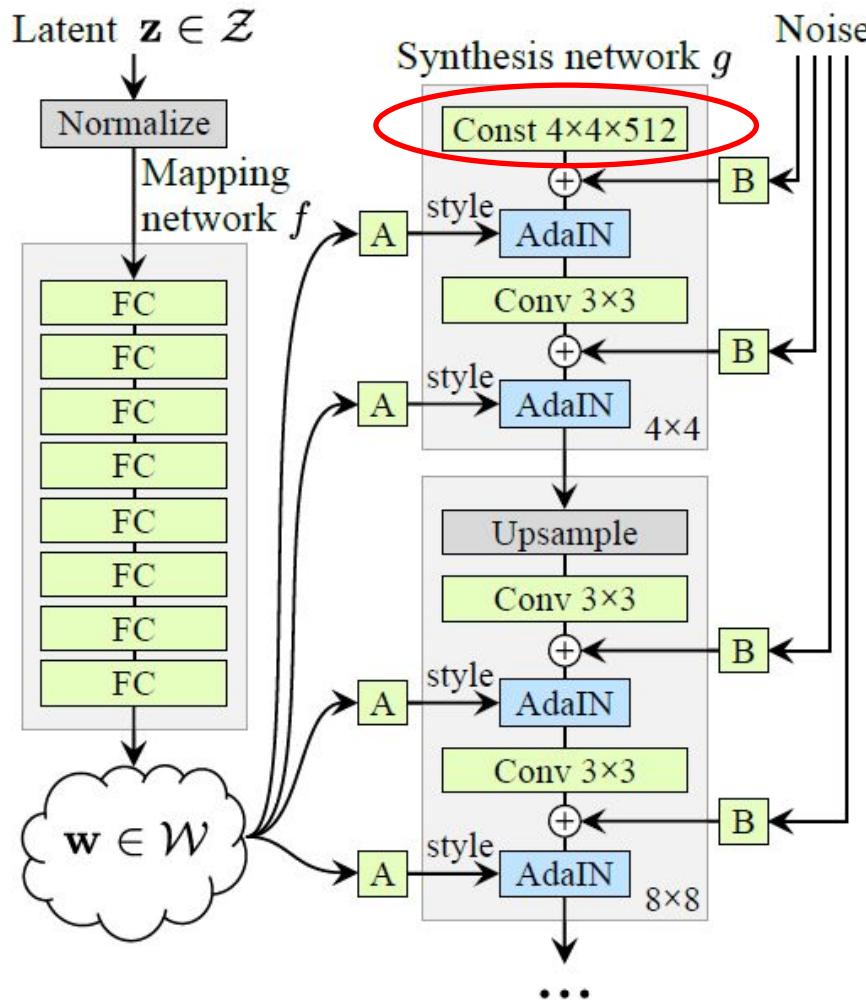


(b) Style-based generator

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

Et ce, à **chaque couche** du réseau

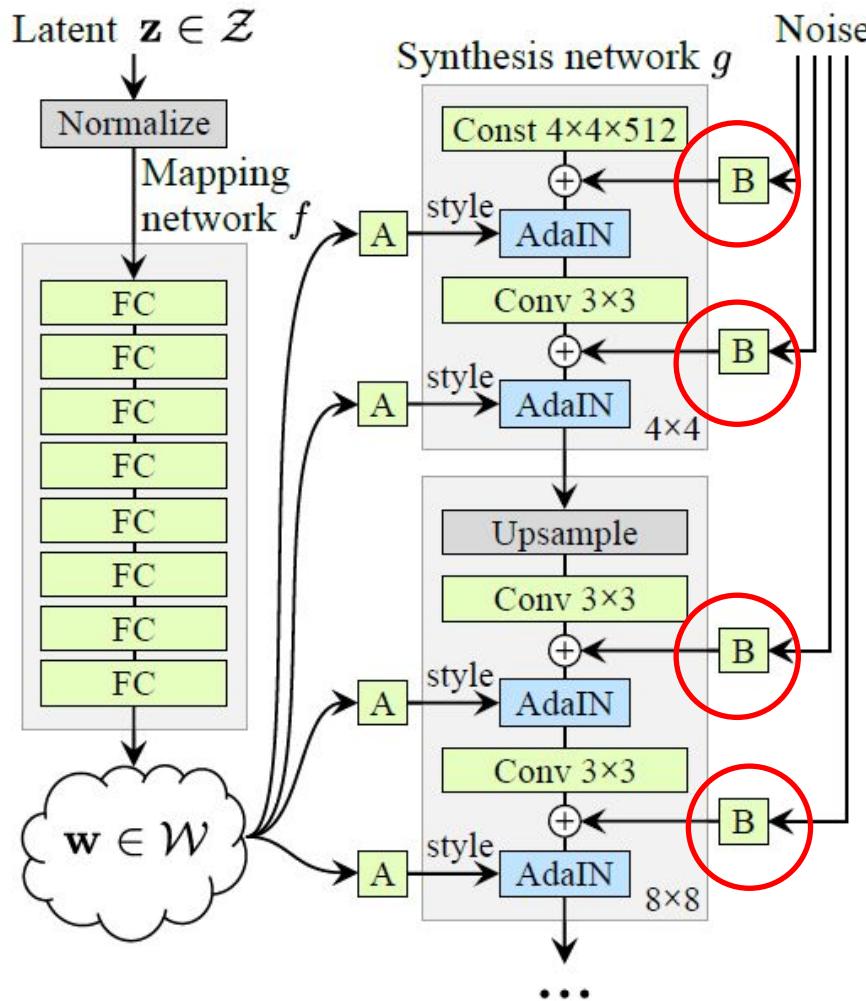
Style GAN



(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

(b) Style-based generator

Effet du bruit



(a) Generated image

(b) Stochastic variation

(c) Standard deviation

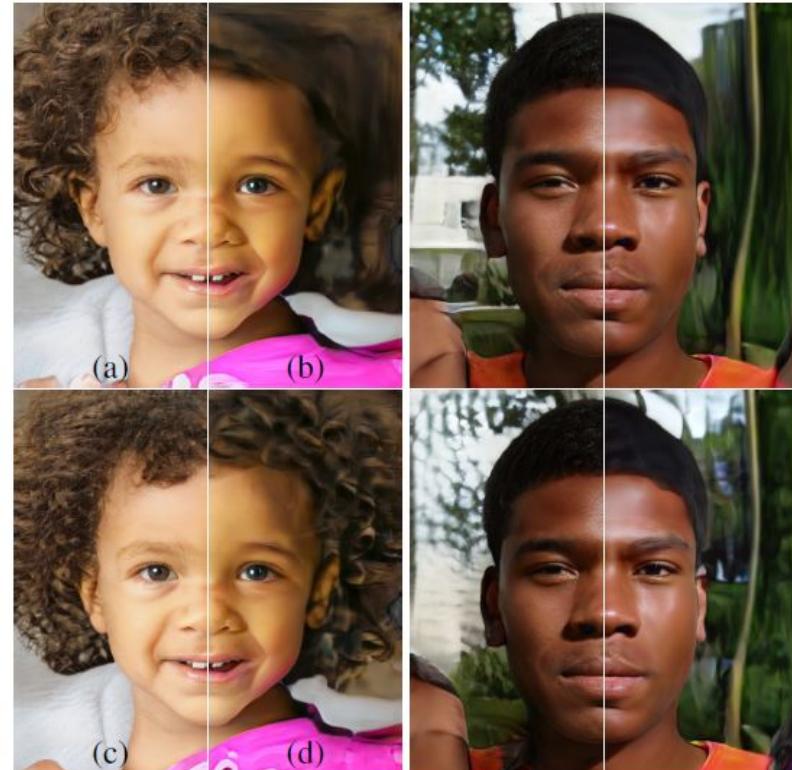
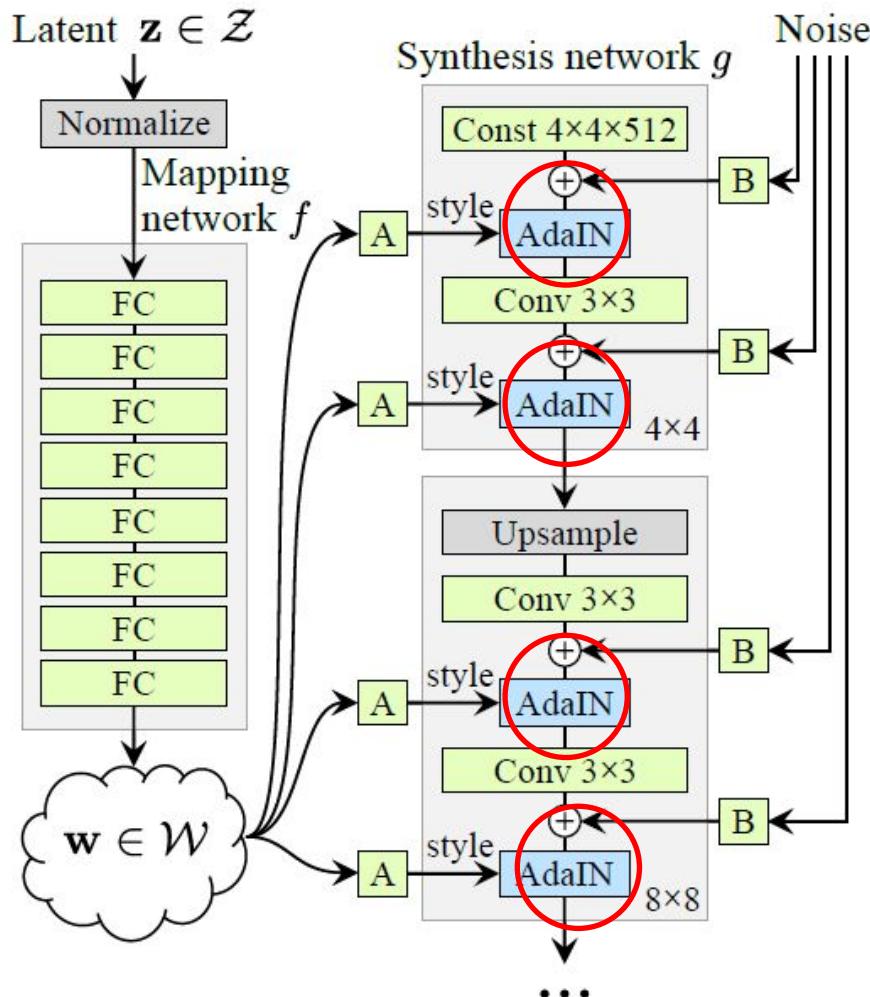


Figure 5. Effect of noise inputs at different layers of our generator. (a) Noise is applied to all layers. (b) No noise. (c) Noise in fine layers only ($64^2 - 1024^2$). (d) Noise in coarse layers only ($4^2 - 32^2$). We can see that the artificial omission of noise leads to featureless “painterly” look. 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



(b) Style-based generator

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

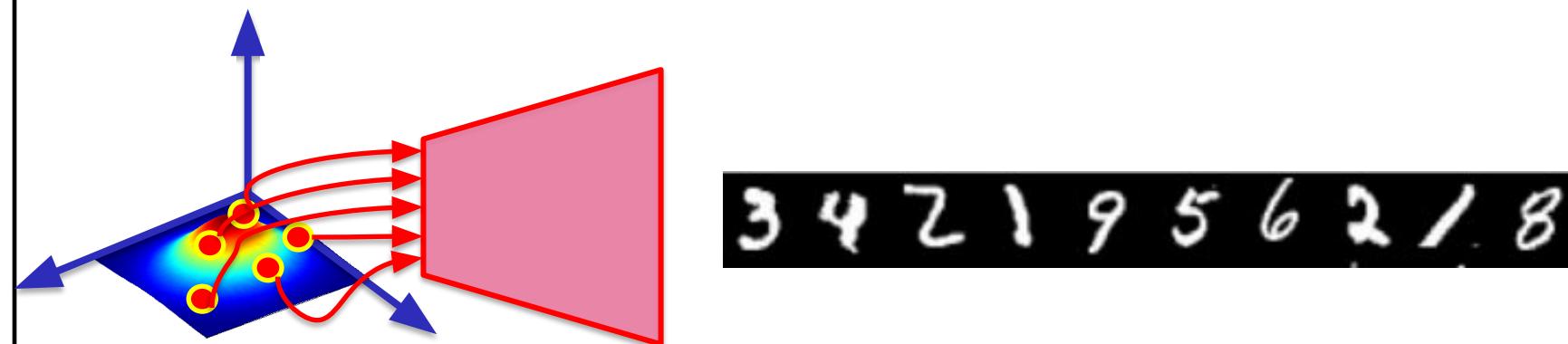
Entraînement progressif comme pour **progressive GAN**

Style GAN



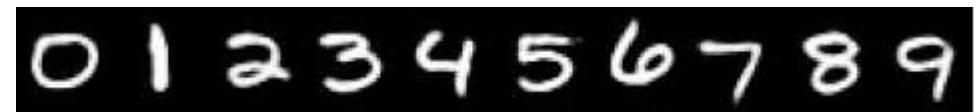
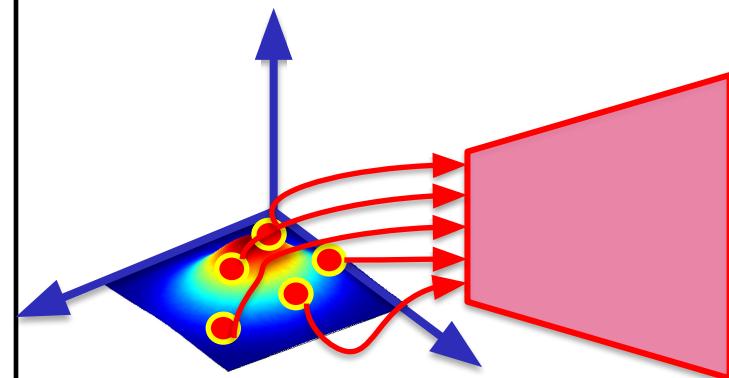
Défi avec les GAN

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



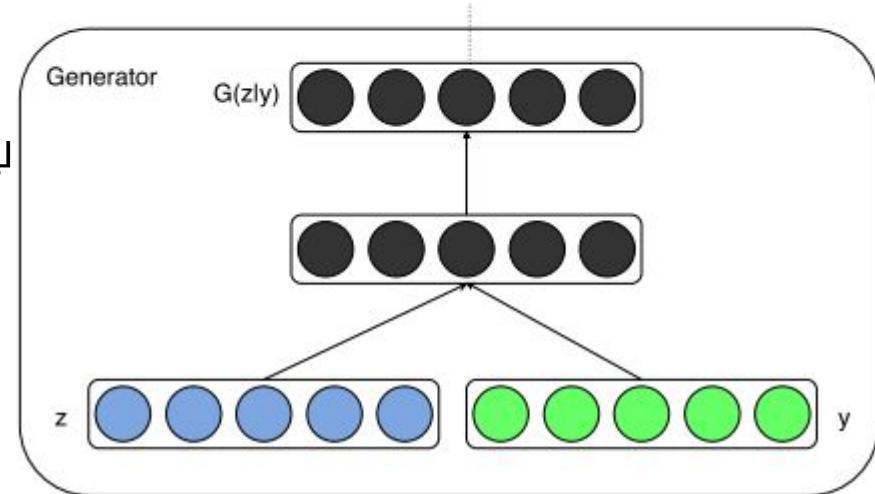
Défi avec les GAN

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?

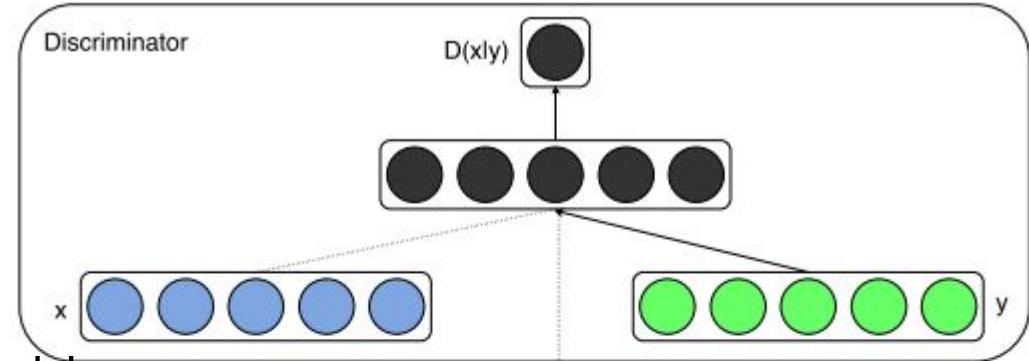


Gan conditionnel

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

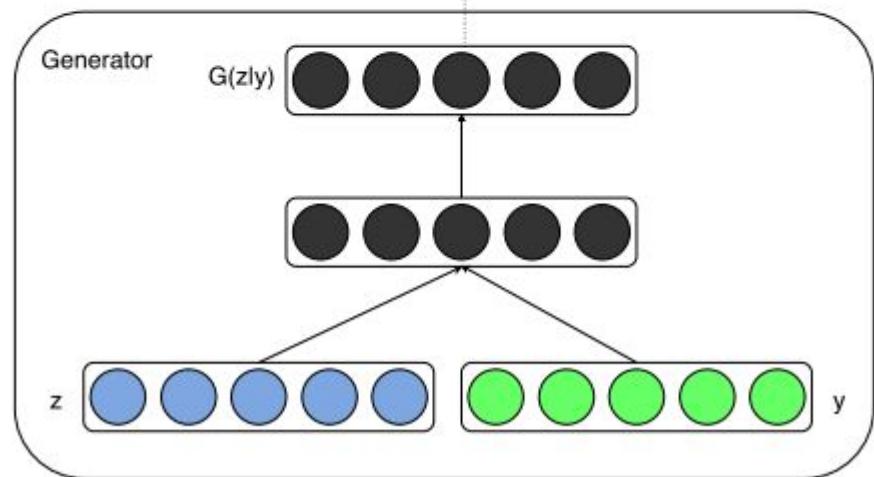


Gan conditionnel

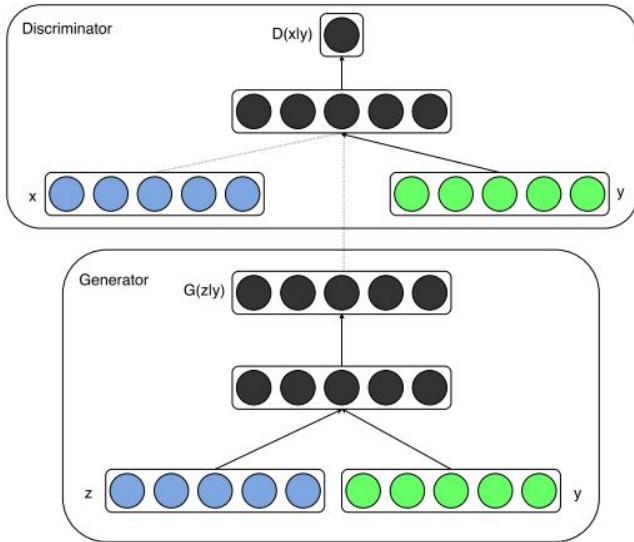


Et de discriminer une image \hat{x}

avec **le même « one-hot »** \hat{y}



Gan conditionnel



GAN de base

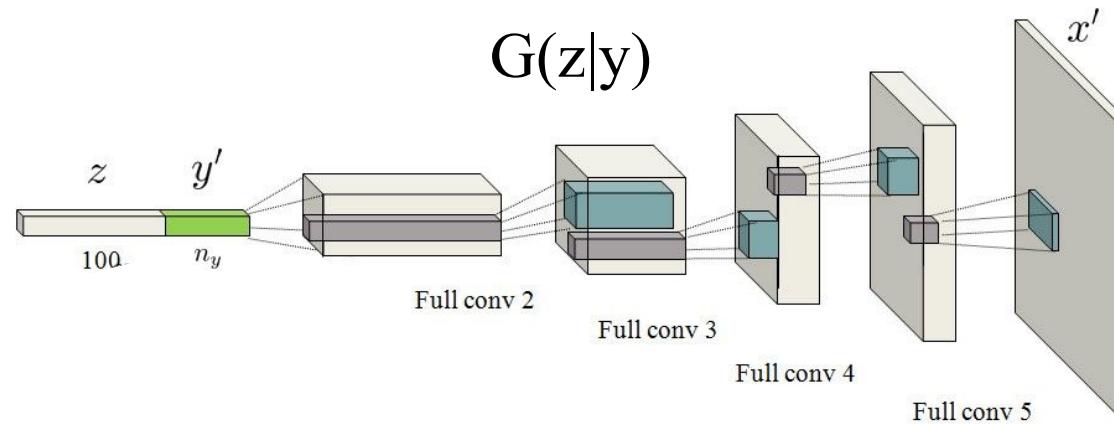
$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

GAN conditionnel

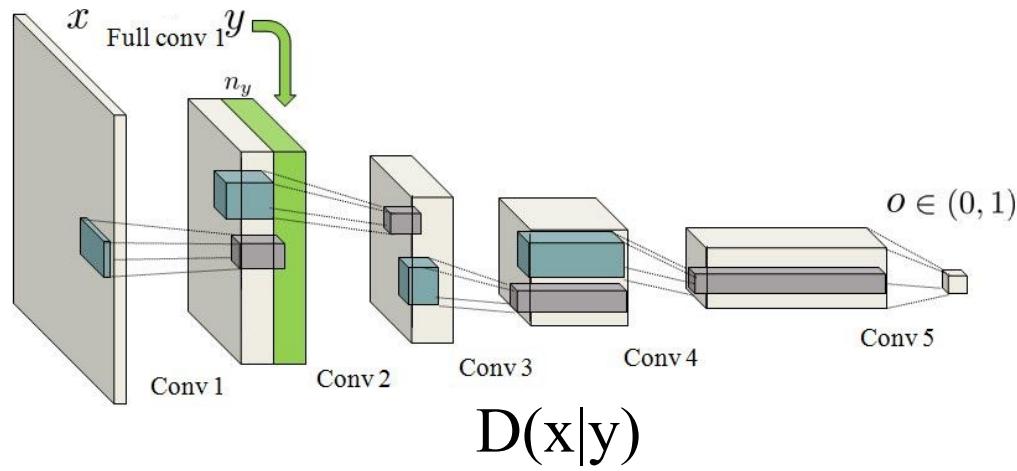
$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x}|\mathbf{y})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z}|\mathbf{y})))].$$

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

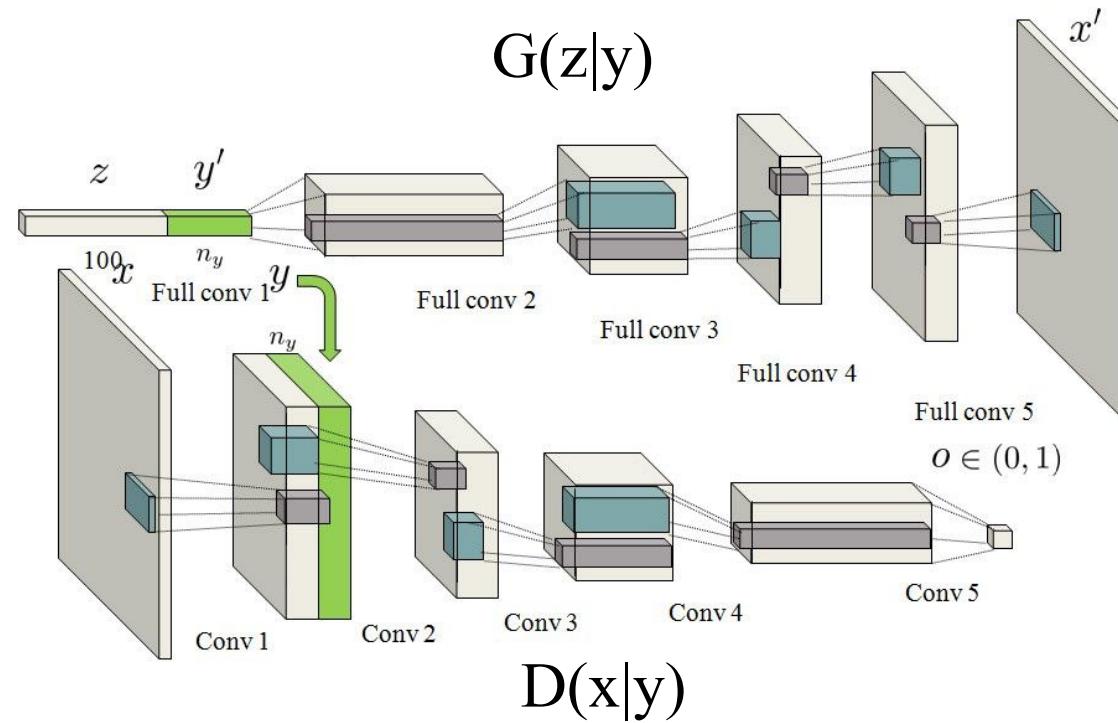
Version convolutionnelle



Version convolutionnelle



Version convolutionnelle





“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>