LEMANS SCHOOL OF AI | SESSION 4.04

SESSION ANIMEE PAR
JEAN-BAPTISTE & PAUL

LEMANS
SCHOOLOFAI
ARTIFICIAL INTELLIGENCE STUDYGROUP MADE BY DOERS FOR DOERS IN LE MANS

SESSION D'APPROFONDISSEMENT

GENERATIVE ADVERSARIAL NETWORKS

JEUDI 05 NOVEMBRE 2020 | 18H30 | LE MANS INNOVATION

Sommaire

- Rappel principes des GAN
- Entrainement d’un GAN
- Evaluation d’un GAN
- StyleGAN

rappel

de quoi s’agit-il ?

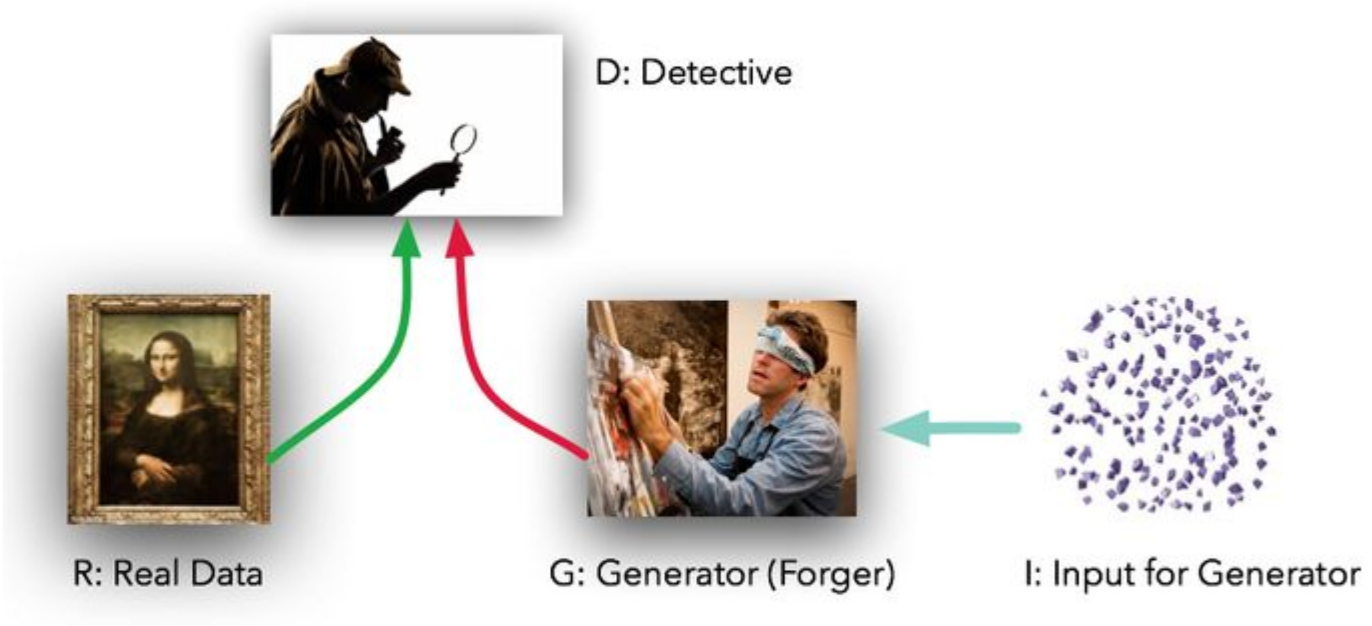
- **GAN** = type de modèle de DL

    🎯 la production de données fausses qui semblent réelles

- la particularité : entraînement non supervisé entre deux réseaux de neurones :

    un **générateur** contre un **discriminateur**.

architecture d'un GAN

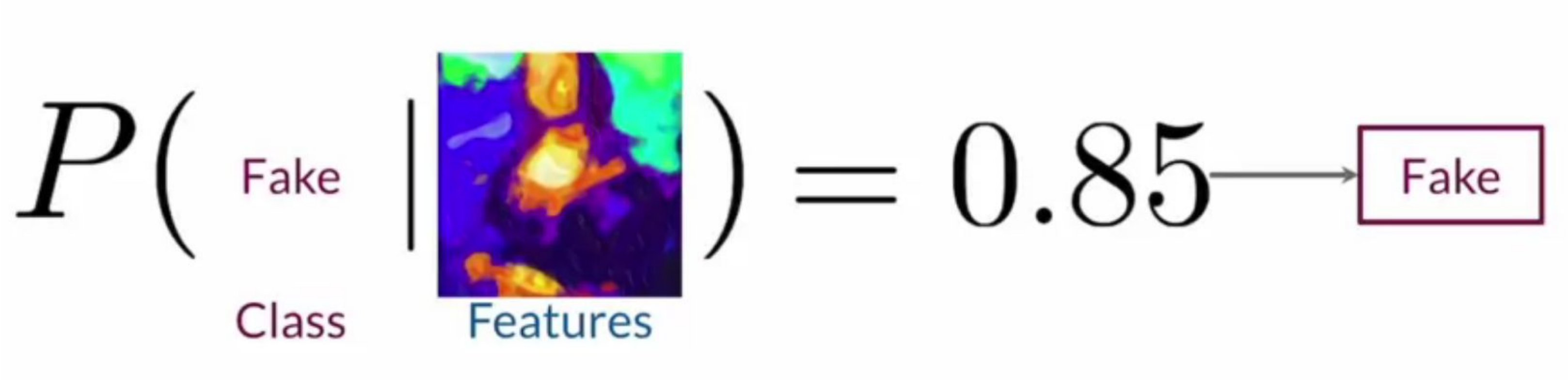


Discriminateur

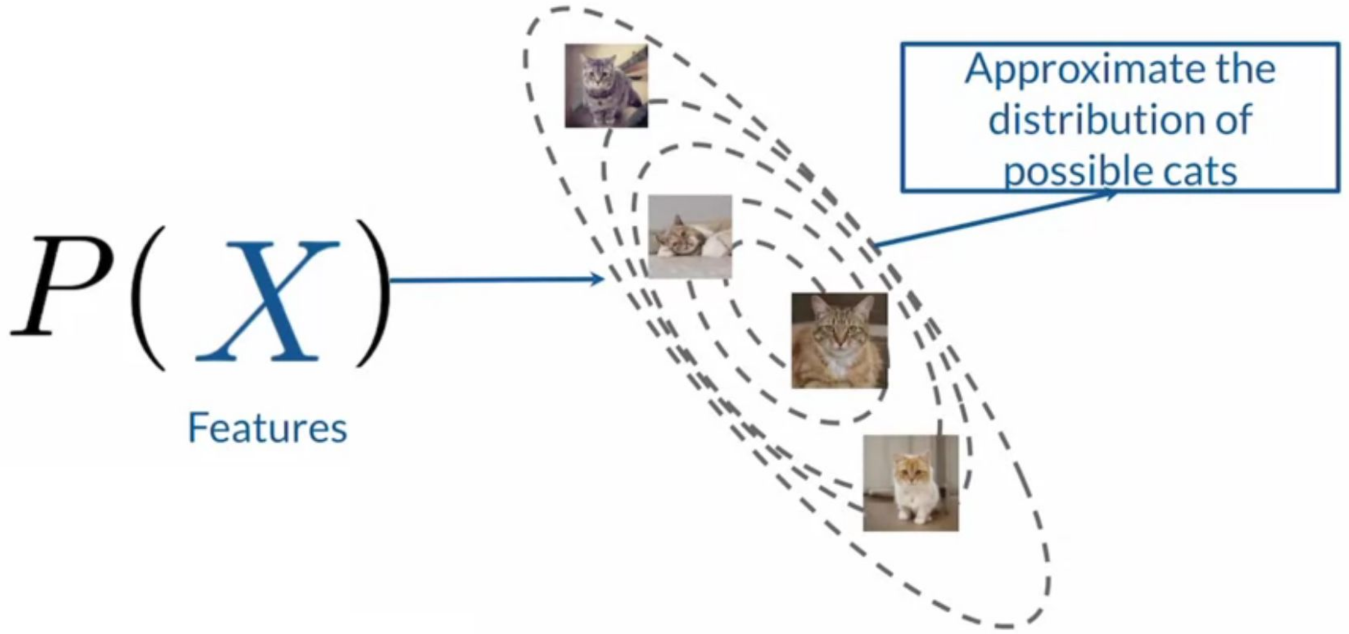
un réseau de neurones de type classifieur

$$P(Y \mid X)$$

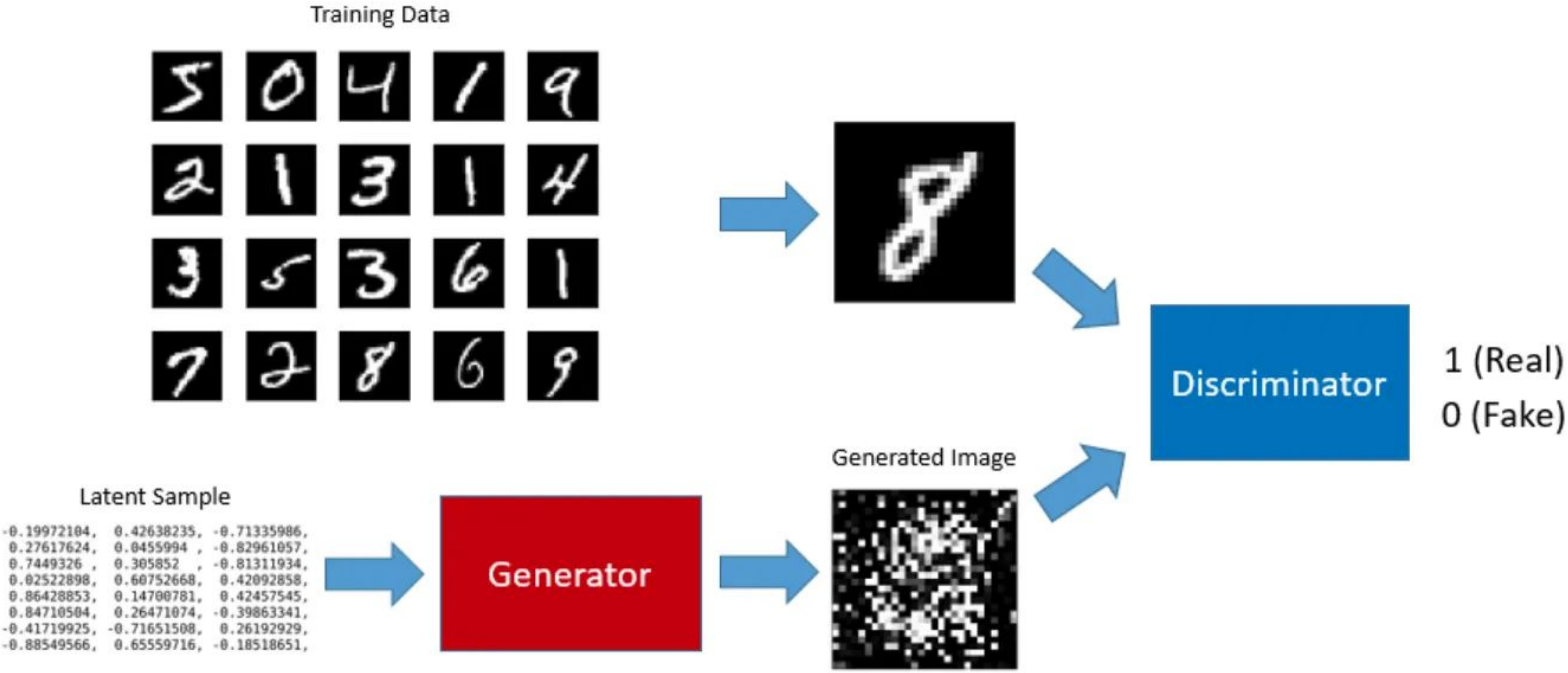
Class Features



Générateur



$P(X)$
Features
Approximate the distribution of possible cats

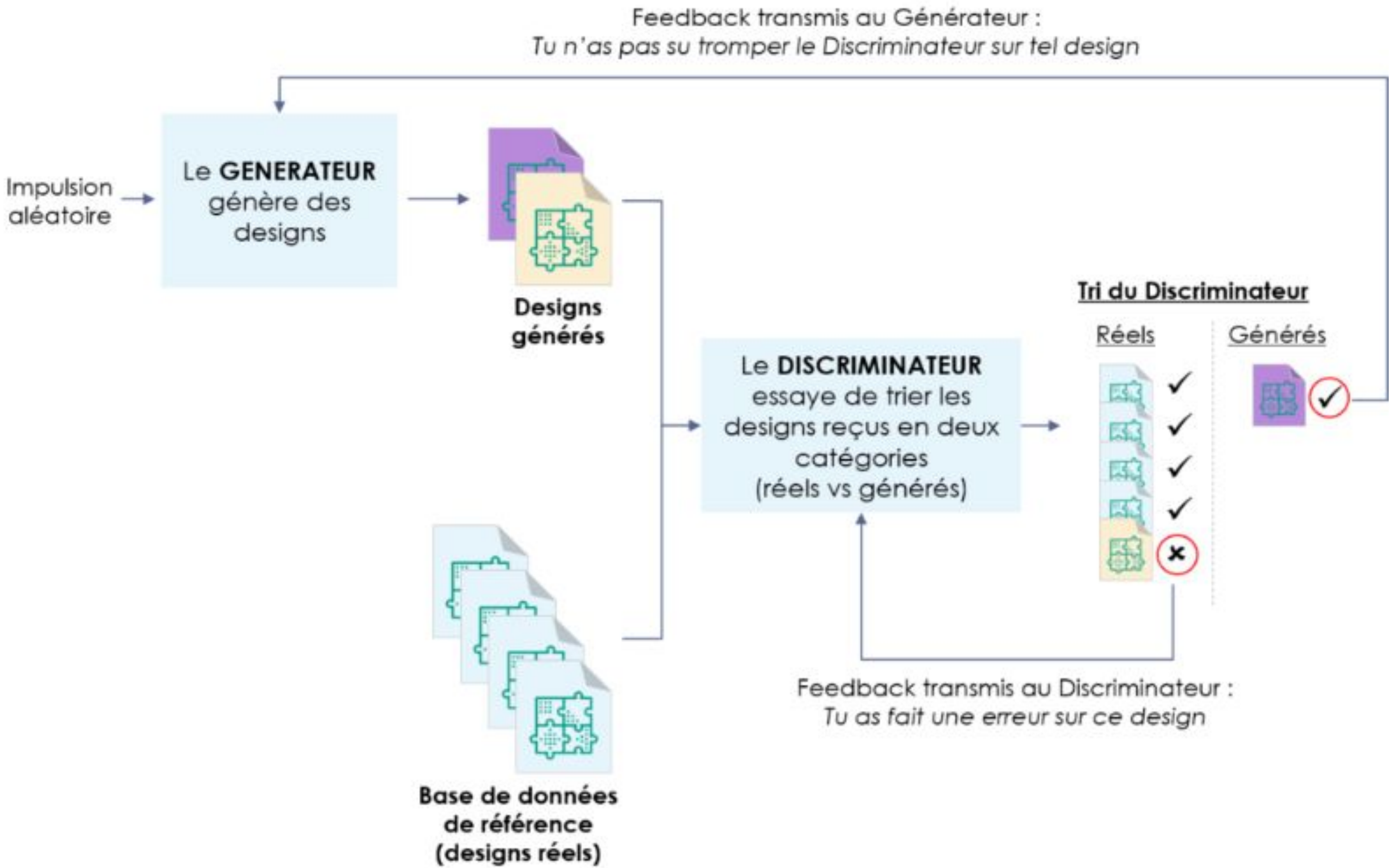


Training Data
Discriminator
1 (Real)
0 (Fake)
Generated Image
Latent Sample
-0.19972104, 0.42638235, -0.71335986,
0.27617624, 0.0455994 , -0.82961057,
0.7449326 , 0.305852 , -0.81311934,
0.02522898, 0.60752668, 0.42092858,
0.86428853, 0.14700781, 0.42457545,
0.84710504, 0.26471074, -0.39863341,
-0.41719925, -0.71651508, 0.26192929,
-0.88549566, 0.65559716, -0.18518651,
Generator

entrainement

Étapes d'un entraînement d'un GAN

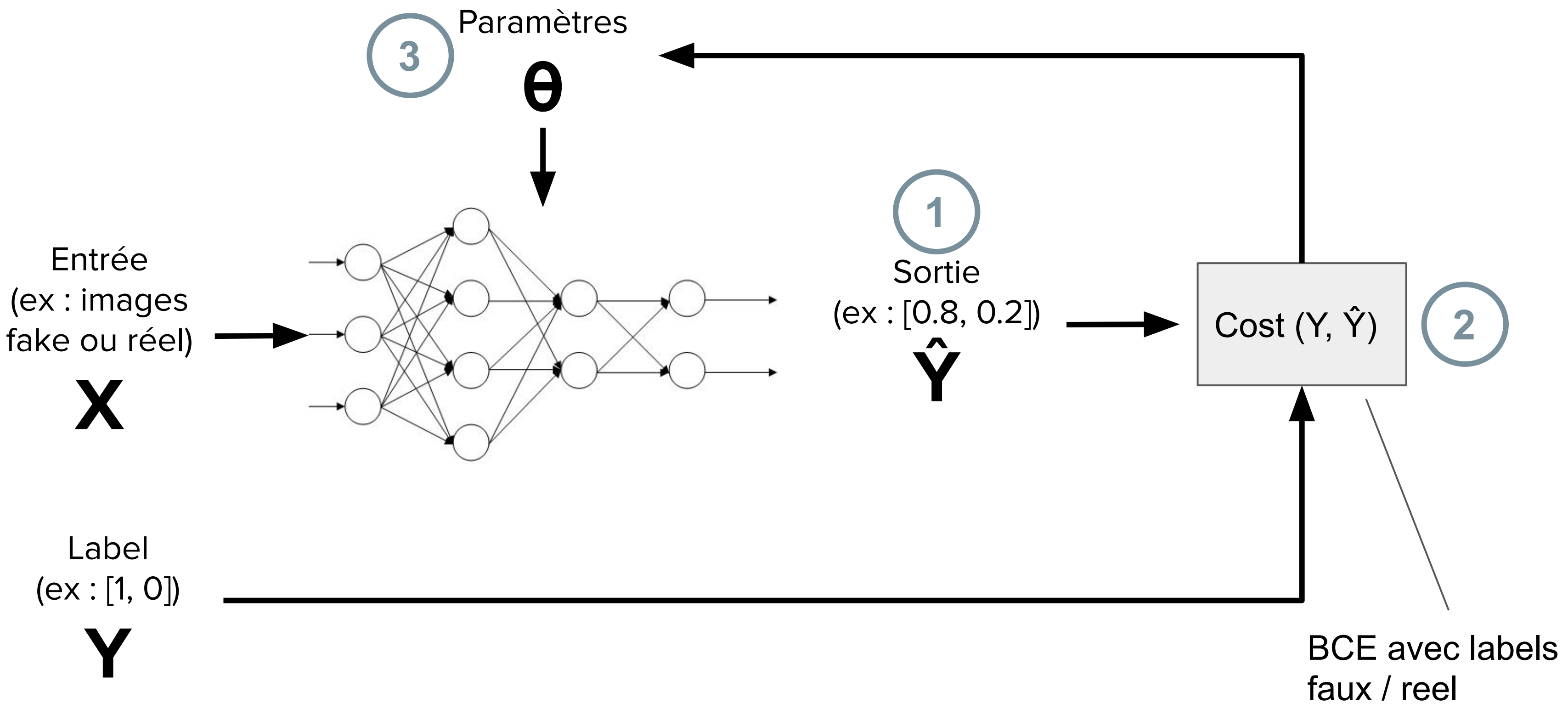
- 1/ pré-entrainement du discriminateur
- 2/ entraînement du discriminateur
- 3/ entraînement du générateur


Feedback transmis au Générateur :
Tu n'as pas su tromper le Discriminateur sur tel design
Impulsion aléatoire
Le **GENERATEUR** génère des designs
Designs générés
Base de données de référence (designs réels)
Le **DISCRIMINATEUR** essaye de trier les designs reçus en deux catégories (réels vs générés)
Tri du Discriminateur
Réels
Générés
Feedback transmis au Discriminateur :
Tu as fait une erreur sur ce design

2/ entraînement du discriminateur

- on fournit au réseau **discriminateur** les données contrefaites générées par le réseau générateur et il doit déterminer lesquelles sont vraies et lesquelles ne le sont pas.

Entrainement du discriminateur

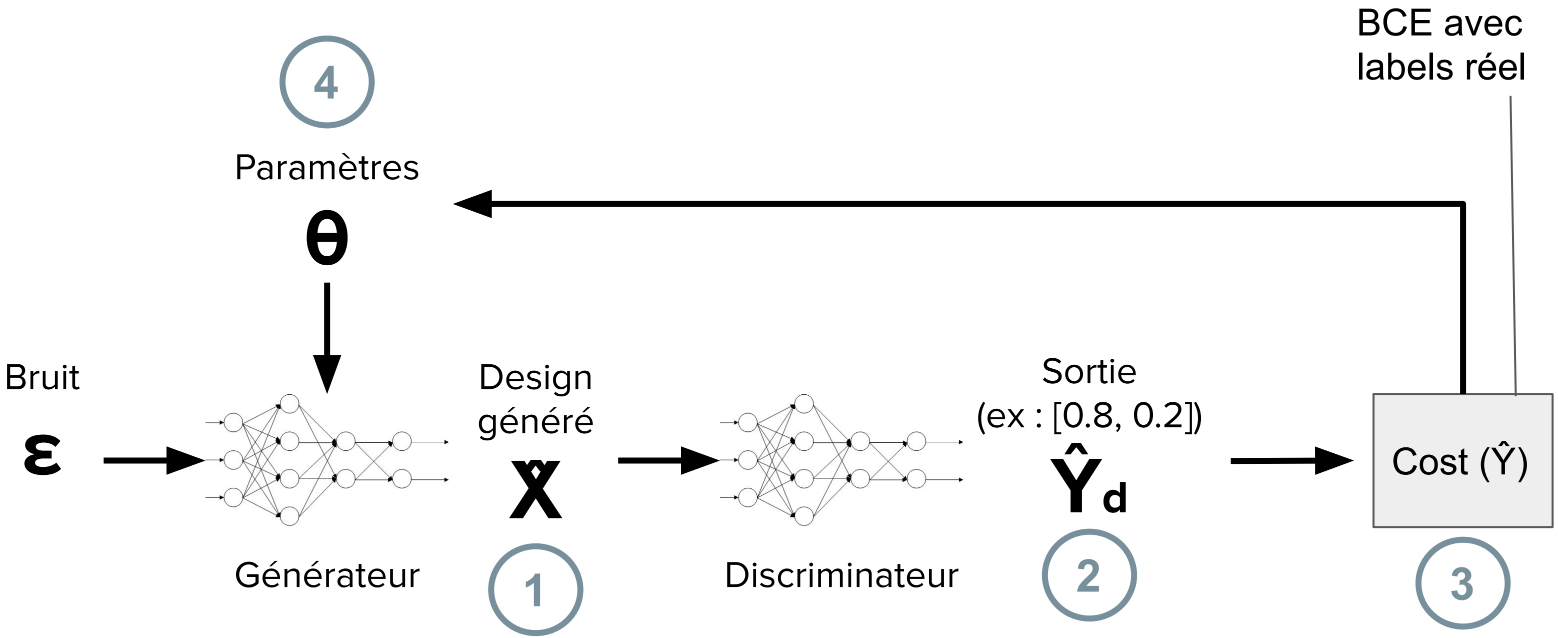


3/ entraînement du générateur

le réseau **générateur** s'améliore avec les résultats issus du deuxième entraînement du réseau discriminateur.

Le réseau **générateur** apprend à reconnaitre les failles du discriminateur et cherche à les utiliser pour générer des ensembles de données contrefaites encore plus réalistes.

Entrainement du générateur



Objectif pour :

- le Générateur : Ŷd = [1, 0]
- le Discriminateur : Ŷd =[0, 1]

challenge

- le challenge :
  - les 2 réseaux doivent apprendre progressivement ensemble
  - trouver un équilibre

- si Discriminateur trop fort (prédiction 100% faux)

  le Générateur n’apprend pas (pas de mise à jour des poids)

- si Générateur trop fort (prédiction 100% réel)

  L’apprentissage est terminé

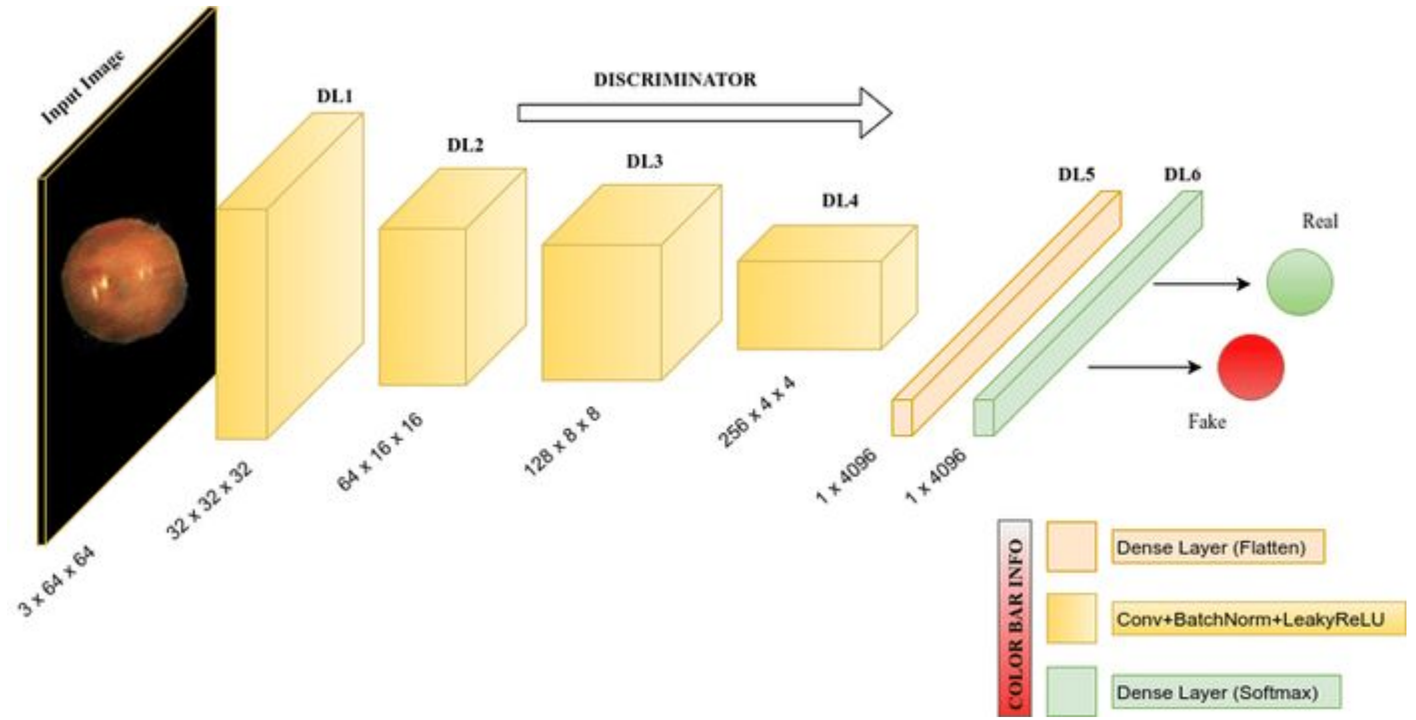
DCGAN

- pour améliorer les performances d’un modèle de génération d’image, le **Deep Convolutional GAN** est souvent utilisé

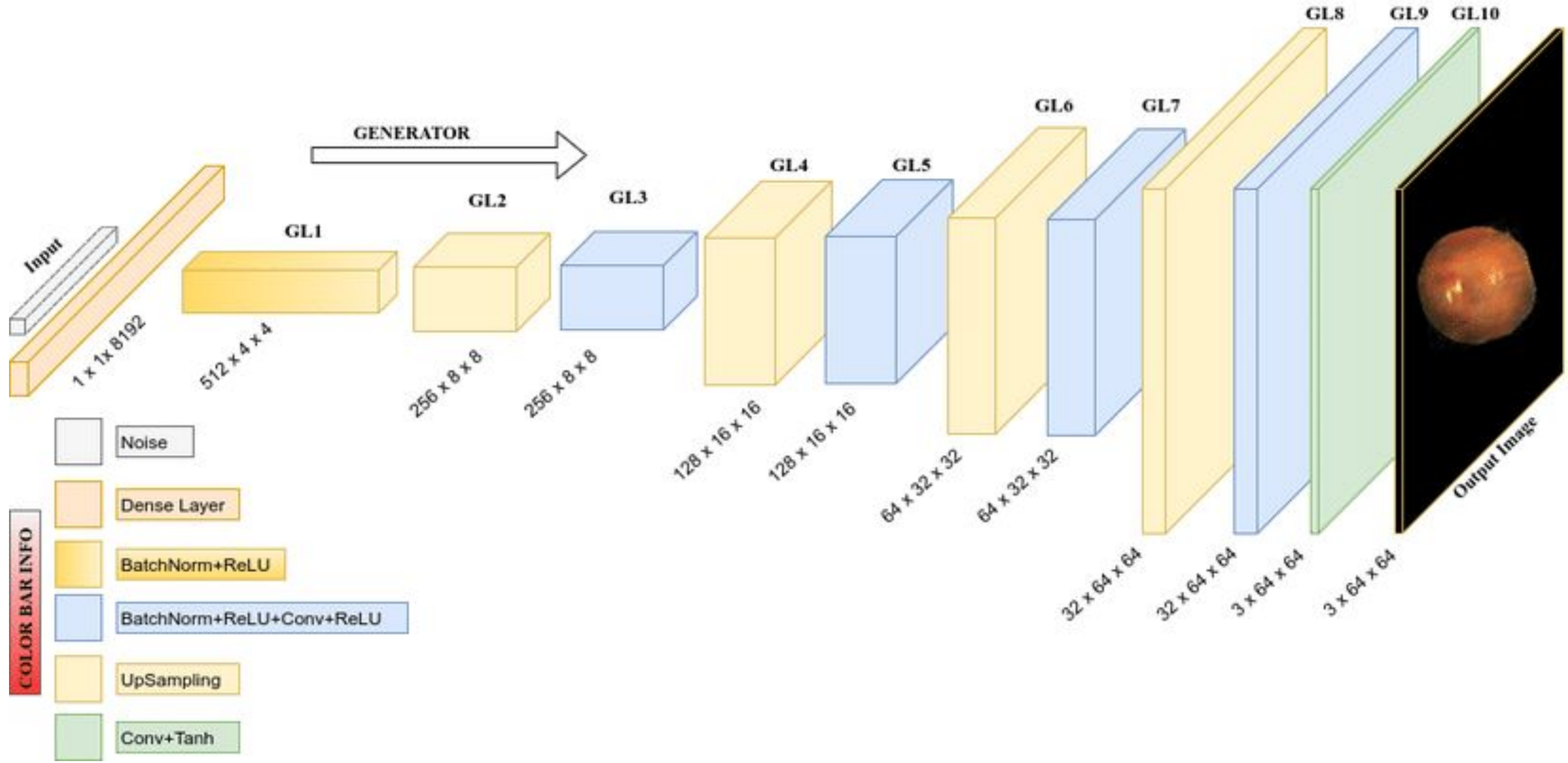
- Le **discriminateur** : un réseau à convolution

- le **générateur** : un réseau à convolution transposée ou “déconvolution”

Réseaux à convolution (discriminateur)

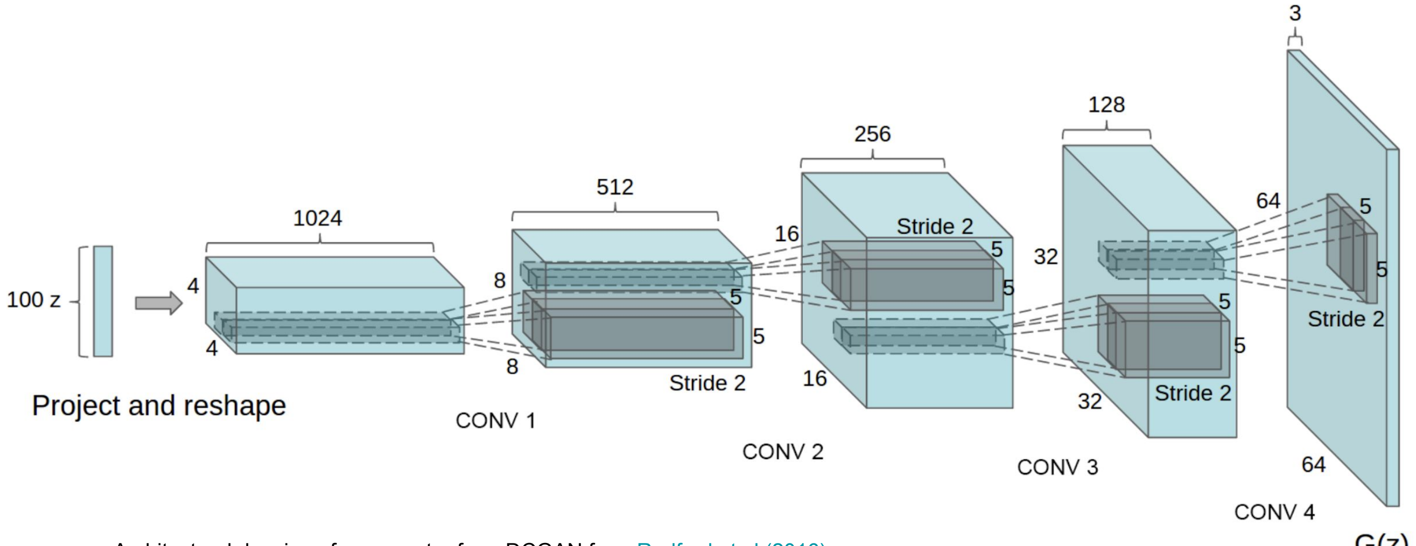


Réseaux à déconvolution (générateur)



DCGAN - exemple d'implémentation

- avec Pytorch
- dataset de chiffres manuscrits de MNIST
- objectif : générer une image synthétique d'un chiffre



Architectural drawing of a generator from DCGAN from [Radford et al (2016)](#).

DCGAN - le générateur

```python
class Generator(nn.Module):
    '''
    Generator Class
    Values:
        z_dim: the dimension of the noise vector, a scalar
        im_chan: the number of channels in the images, fitted for the dataset used, a scalar
              (MNIST is black-and-white, so 1 channel is your default)
        hidden_dim: the inner dimension, a scalar
    '''
    def __init__(self, z_dim=10, im_chan=1, hidden_dim=64):
        super(Generator, self).__init__()
        self.z_dim = z_dim
        # Build the neural network
        self.gen = nn.Sequential(
            self.make_gen_block(z_dim, hidden_dim * 4),
            self.make_gen_block(hidden_dim * 4, hidden_dim * 2, kernel_size=4, stride=1),
            self.make_gen_block(hidden_dim * 2, hidden_dim),
            self.make_gen_block(hidden_dim, im_chan, kernel_size=4, final_layer=True),
        )

    def make_gen_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
        '''
        Function to return a sequence of operations corresponding to a generator block of DCGAN,
        corresponding to a transposed convolution, a batchnorm (except for in the last layer), and an activation.
        Parameters:
            input_channels: how many channels the input feature representation has
            output_channels: how many channels the output feature representation should have
            kernel_size: the size of each convolutional filter, equivalent to (kernel_size, kernel_size)
            stride: the stride of the convolution
            final_layer: a boolean, true if it is the final layer and false otherwise
                      (affects activation and batchnorm)
        '''
        # Build the neural block
        if not final_layer:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride=stride),
                nn.BatchNorm2d(output_channels),
                nn.ReLU(inplace=True)
            )
```

DCGAN - le discriminateur

```python
class Discriminator(nn.Module):
    '''
    Discriminator Class
    Values:
        im_chan: the number of channels in the images, fitted for the dataset used, a scalar
              (MNIST is black-and-white, so 1 channel is your default)
    hidden_dim: the inner dimension, a scalar
    '''
    def __init__(self, im_chan=1, hidden_dim=16):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            self.make_disc_block(im_chan, hidden_dim),
            self.make_disc_block(hidden_dim, hidden_dim * 2),
            self.make_disc_block(hidden_dim * 2, 1, final_layer=True),
        )

    def make_disc_block(self, input_channels, output_channels, kernel_size=4, stride=2, final_layer=False):
        '''
        Function to return a sequence of operations corresponding to a discriminator block of DCGAN,
        corresponding to a convolution, a batchnorm (except for in the last layer), and an activation.
        Parameters:
            input_channels: how many channels the input feature representation has
            output_channels: how many channels the output feature representation should have
            kernel_size: the size of each convolutional filter, equivalent to (kernel_size, kernel_size)
            stride: the stride of the convolution
            final_layer: a boolean, true if it is the final layer and false otherwise
                      (affects activation and batchnorm)
        '''

        # Build the neural block
        if not final_layer:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size, stride=stride),
                nn.BatchNorm2d(output_channels),
                nn.LeakyReLU(0.2)
            )
        else: # Final Layer
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size, stride=stride)
```

DCGAN - parametre entrainement

```
criterion = nn.BCEWithLogitsLoss()
z_dim = 64
display_step = 500
batch_size = 128
# A learning rate of 0.0002 works well on DCGAN
lr = 0.0002

# These parameters control the optimizer's momentum, which you can read more about here:
# https://distill.pub/2017/momentum/
beta_1 = 0.5
beta_2 = 0.999
device = 'cuda'

# You can tranform the image values to be between -1 and 1 (the range of the tanh activation)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
])

dataloader = DataLoader(
    MNIST('.', download=False, transform=transform),
    batch_size=batch_size,
    shuffle=True)
```

DCGAN - initialisation

```
gen = Generator(z_dim).to(device)
gen_opt = torch.optim.Adam(gen.parameters(), lr=lr, betas=(beta_1, beta_2))
disc = Discriminator().to(device)
disc_opt = torch.optim.Adam(disc.parameters(), lr=lr, betas=(beta_1, beta_2))

# You initialize the weights to the normal distribution
# with mean 0 and standard deviation 0.02
def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
    if isinstance(m, nn.BatchNorm2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
        torch.nn.init.constant_(m.bias, 0)
gen = gen.apply(weights_init)
disc = disc.apply(weights_init)
```

DCGAN - entrainement

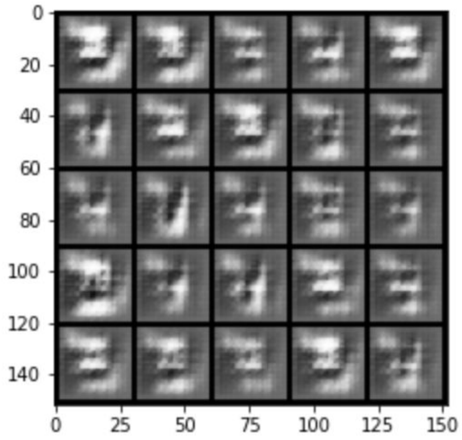
```python
n_epochs = 50
cur_step = 0
mean_generator_loss = 0
mean_discriminator_loss = 0
for epoch in range(n_epochs):
    # Dataloader returns the batches
    for real, _ in tqdm(dataloader):
        cur_batch_size = len(real)
        real = real.to(device)

        ## Update discriminator ##
        disc_opt.zero_grad()
        fake_noise = get_noise(cur_batch_size, z_dim, device=device)
        fake = gen(fake_noise)
        disc_fake_pred = disc(fake.detach())
        disc_fake_loss = criterion(disc_fake_pred, torch.zeros_like(disc_fake_pred))
        disc_real_pred = disc(real)
        disc_real_loss = criterion(disc_real_pred, torch.ones_like(disc_real_pred))
        disc_loss = (disc_fake_loss + disc_real_loss) / 2

        # Keep track of the average discriminator loss
        mean_discriminator_loss += disc_loss.item() / display_step
        # Update gradients
        disc_loss.backward(retain_graph=True)
        # Update optimizer
        disc_opt.step()
```

DCGAN - entrainement (suite)

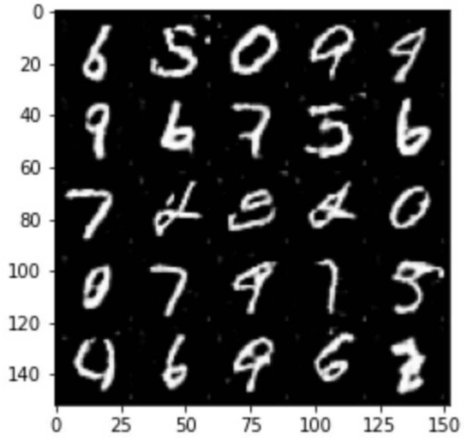
```
# Keep track of the average discriminator loss
mean_discriminator_loss += disc_loss.item() / display_step
# Update gradients
disc_loss.backward(retain_graph=True)
# Update optimizer
disc_opt.step()

## Update generator ##
gen_opt.zero_grad()
fake_noise_2 = get_noise(cur_batch_size, z_dim, device=device)
fake_2 = gen(fake_noise_2)
disc_fake_pred = disc(fake_2)
gen_loss = criterion(disc_fake_pred, torch.ones_like(disc_fake_pred))
gen_loss.backward()
gen_opt.step()

# Keep track of the average generator loss
mean_generator_loss += gen_loss.item() / display_step

## Visualization code ##
if cur_step % display_step == 0 and cur_step > 0:
    print(f"Step {cur_step}: Generator loss: {mean_generator_loss}, discriminator loss: {mean_discriminator_loss}"
    show_tensor_images(fake)
    show_tensor_images(real)
    mean_generator_loss = 0
    mean_discriminator_loss = 0
cur_step += 1
```

DCGAN - résultat (suite)

Step 1000: Generator loss: 2.0383475271463407, discriminator loss: 0.24150314955413335



Step 23000: Generator loss: 0.6993783376812931, discriminator loss: 0.6958930463790896



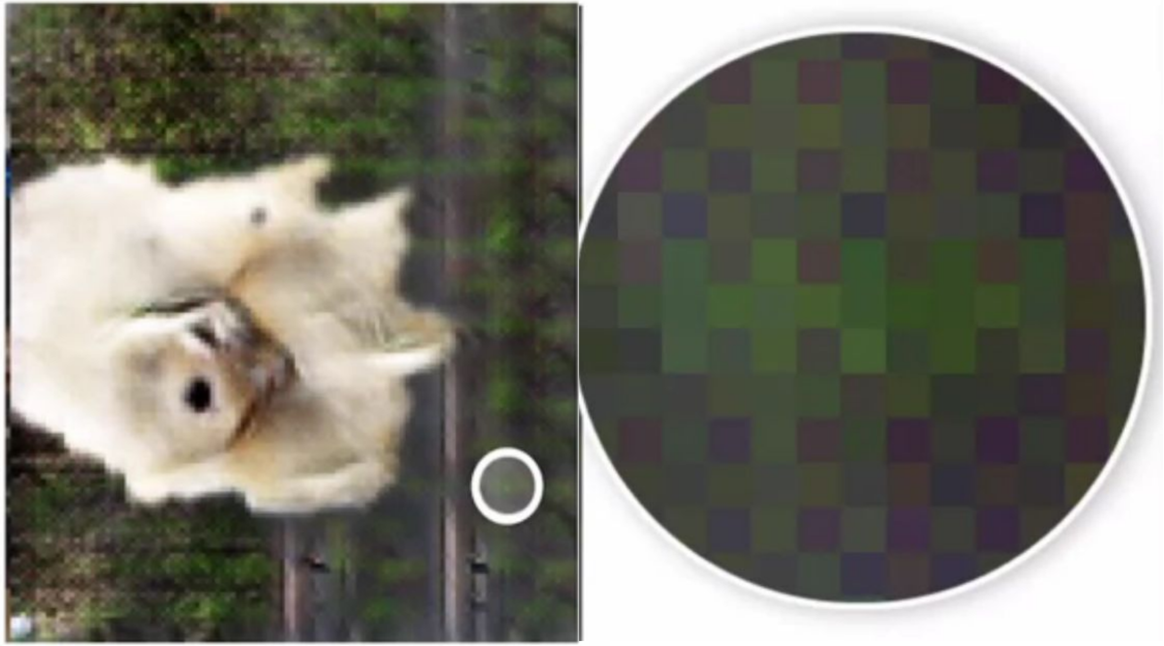
Remarques pour l’amélioration du modèle

- Utilisation de tanh pour la couche output du générateur (Scale the image pixel value between -1 and 1.)
- Batch Normalization aide à la stabilisation de l'entraînement
- Eviter le max pooling
- utiliser la fonction d’optimisation Adam

Problème de l’effet damier

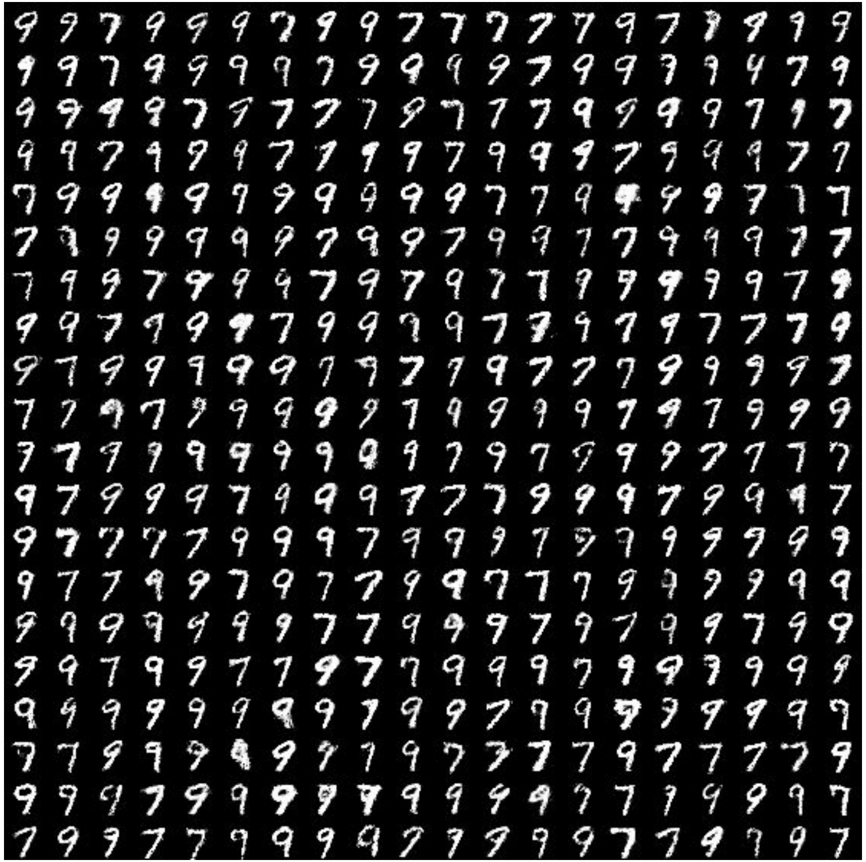
- Problème courant pour les **déconvolutions**

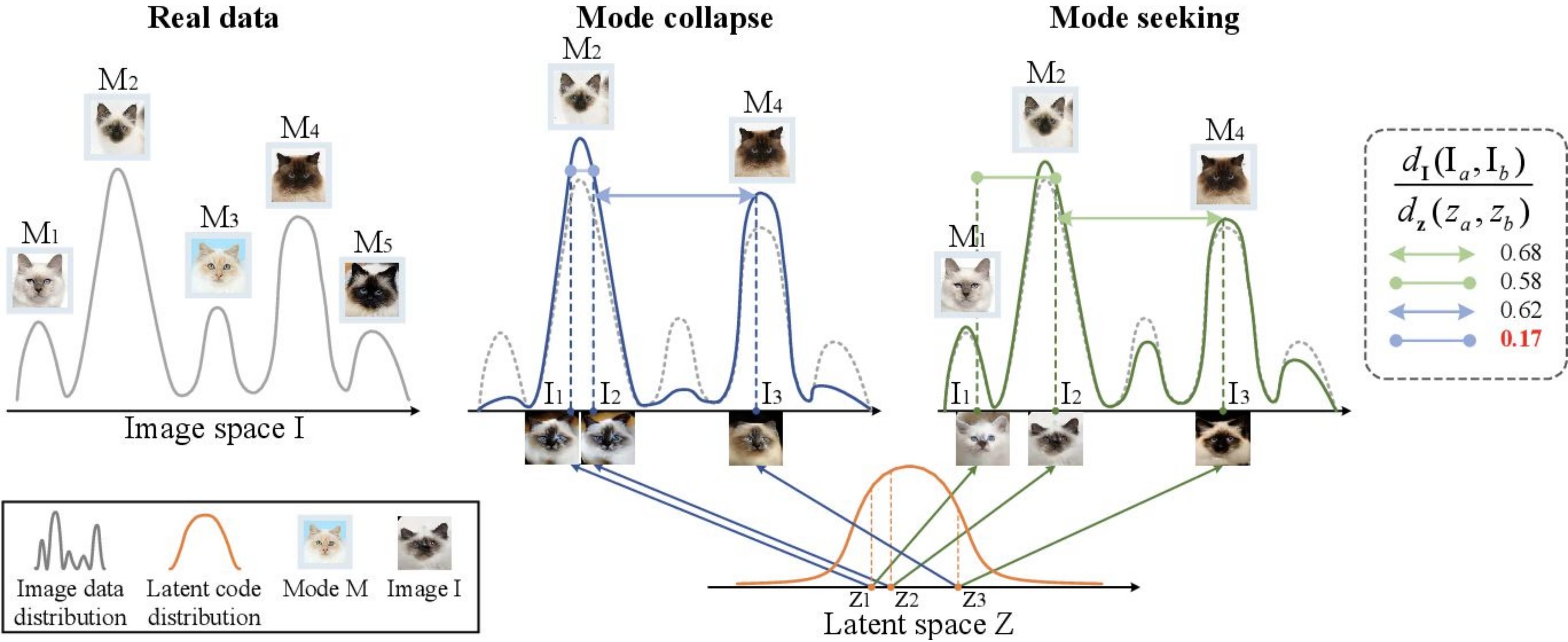
- l'utilisation d'un upsampling suivi d'une convolution est une technique de plus en plus populaire aujourd'hui pour éviter ce problème de damier.



Mode collapse

- problème fréquemment rencontré
- observable ici par une représentation d'un sous-ensemble des chiffres (ici 9, 7, 1)





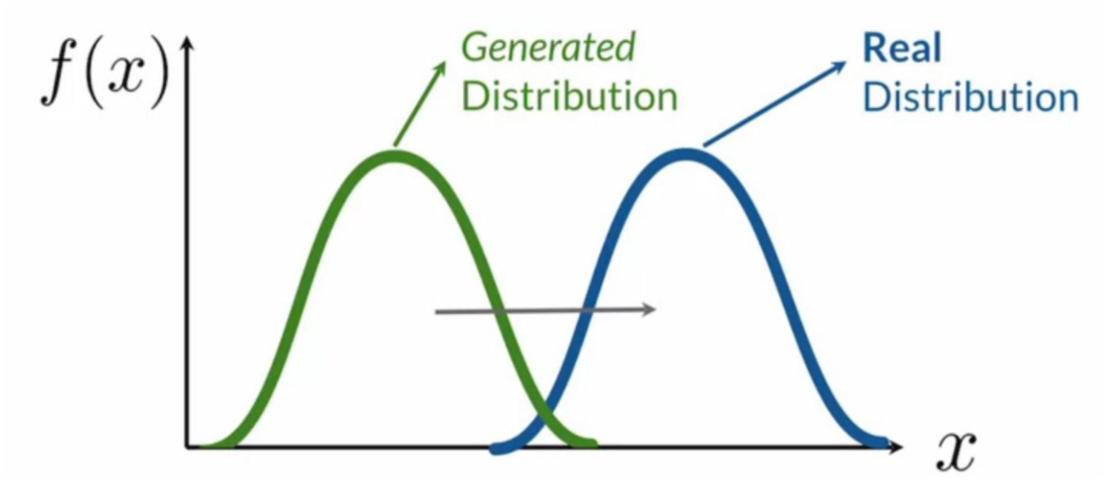
Real data
Mode collapse
Mode seeking
M1
M2
M3
M4
M5
I1
I2
I3
Image space I
$\frac{d_{\mathbf{I}}(\mathrm{I}_a, \mathrm{I}_b)}{d_{\mathbf{z}}(z_a, z_b)}$
0.68
0.58
0.62
0.17
Z1
Z2
Z3
Latent space Z
Image data distribution
Latent code distribution
Mode M
Image I

Problème Vanishing gradient

- c’est un problème lié à l’utilisation la fonction de coût BCE (Binary Cross Entropy)
- c’est un problème qui survient quand le **Discriminateur** s’améliore trop vite

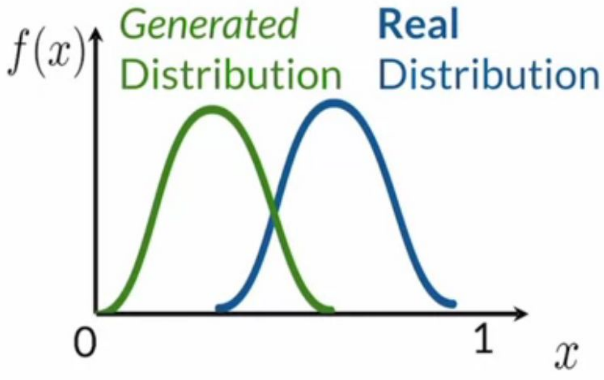
Problème Vanishing gradient (suite)

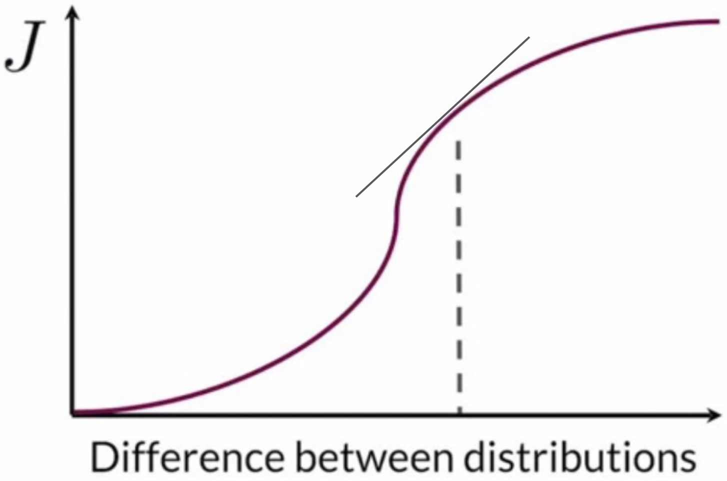
objectif du générateur : tendre vers la distribution réelle



Problème Vanishing gradient (suite)

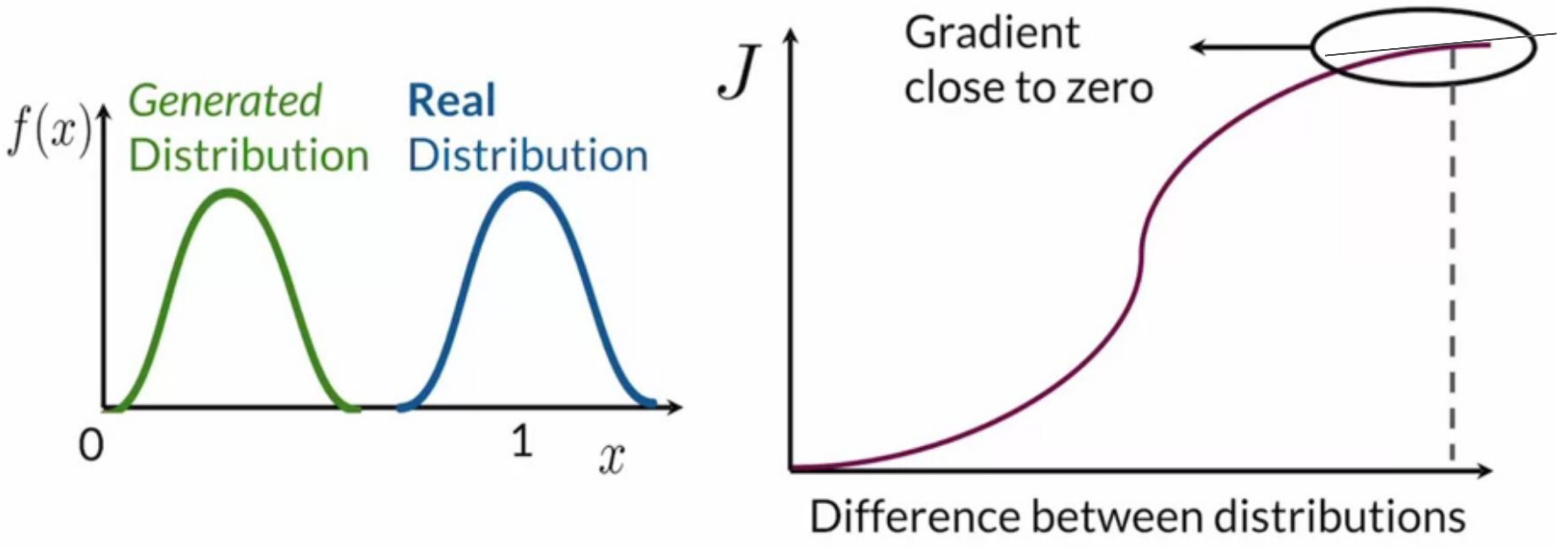
gradient non-zero = le générateur s’améliore





Problème Vanishing gradient (suite)

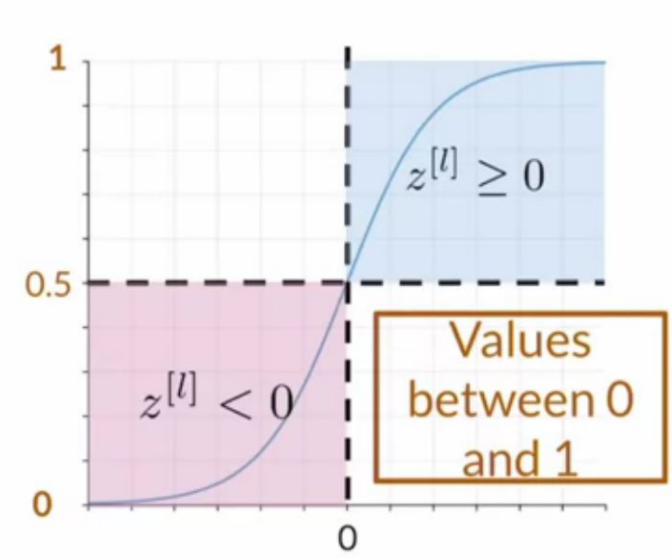
gradient proche de zéro = le générateur ne peut plus s'améliorer



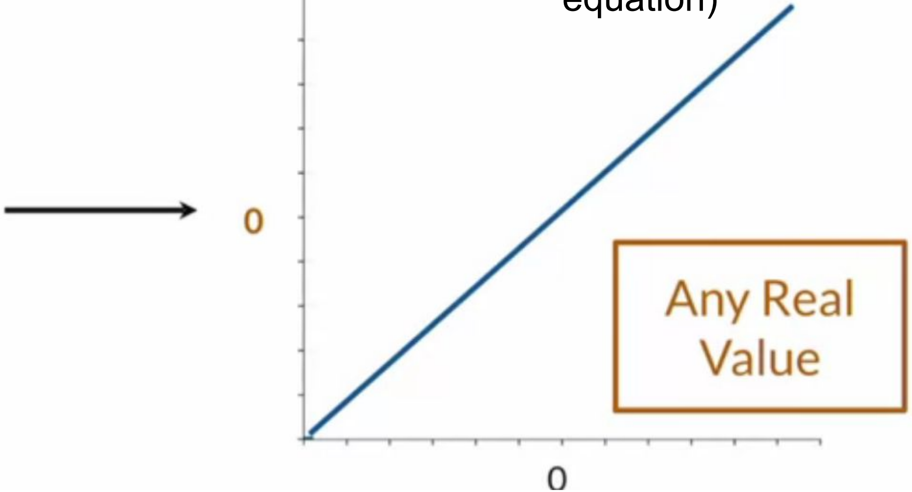
W-loss

- fonction de coût alternative : Wasserstein loss
- approximatisation de la Earth mover’s distance

BCE



W-loss (pas de log dans l’ équation)



W-loss

- résout les problèmes :
  - mode collapse
  - vanished gradient

- implique souvent la mise en place de pénalités sur les poids lors de la rétropropagation

Exemples de programme illustrant des modèles instables :

https://www.aiproblog.com/index.php/2019/07/07/how-to-identify-and-diagnose-gan-failure-modes/

Évaluation

Évaluation

- Inception-v3 and Embeddings
- Fréchet Inception Distance (FID)

Inception-v3 architecture

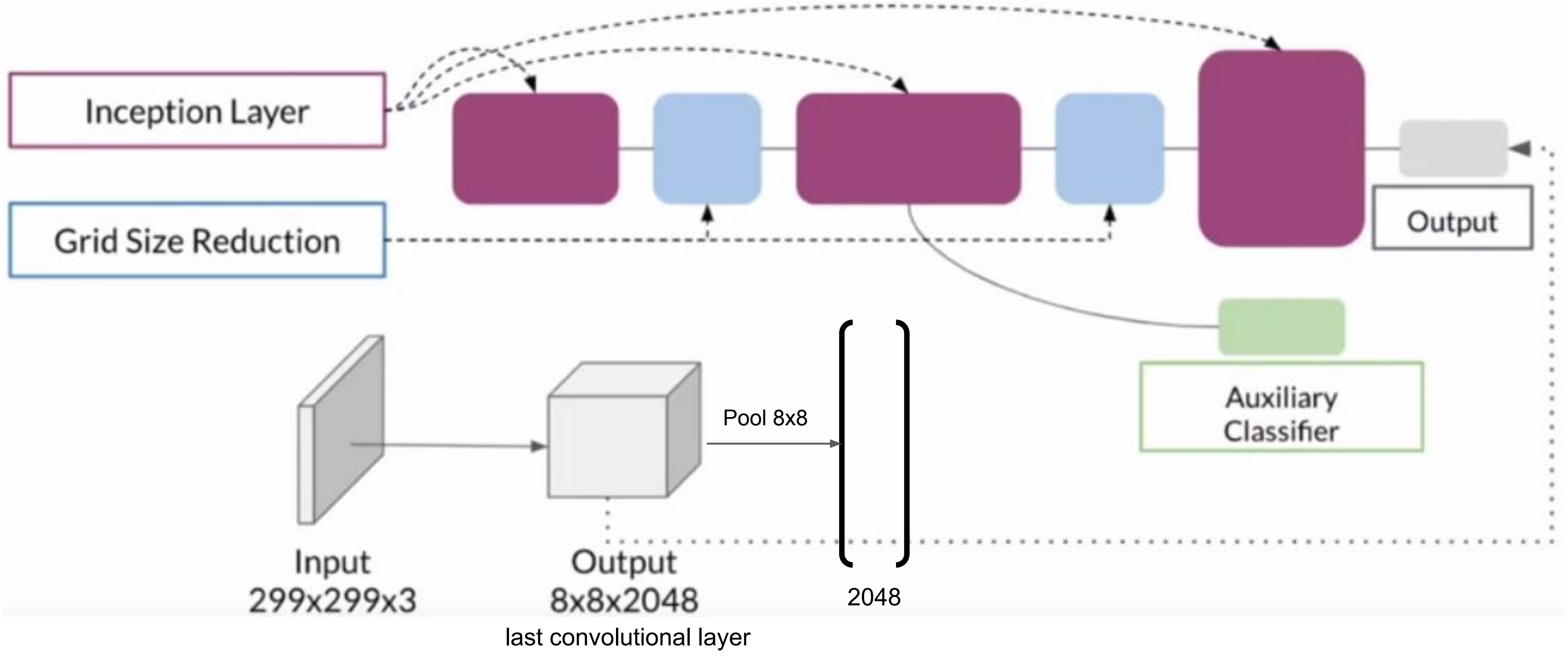
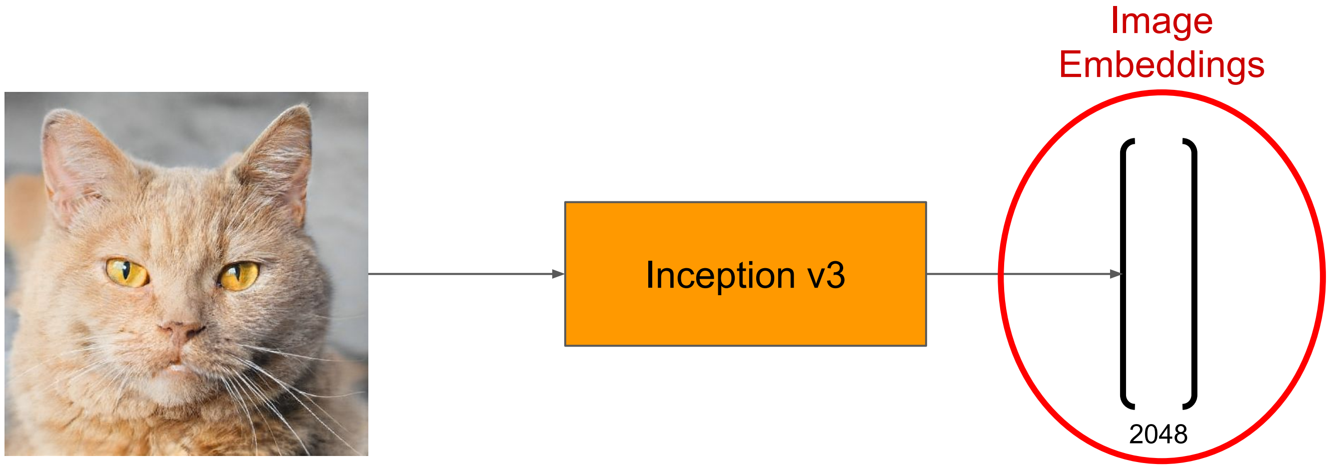
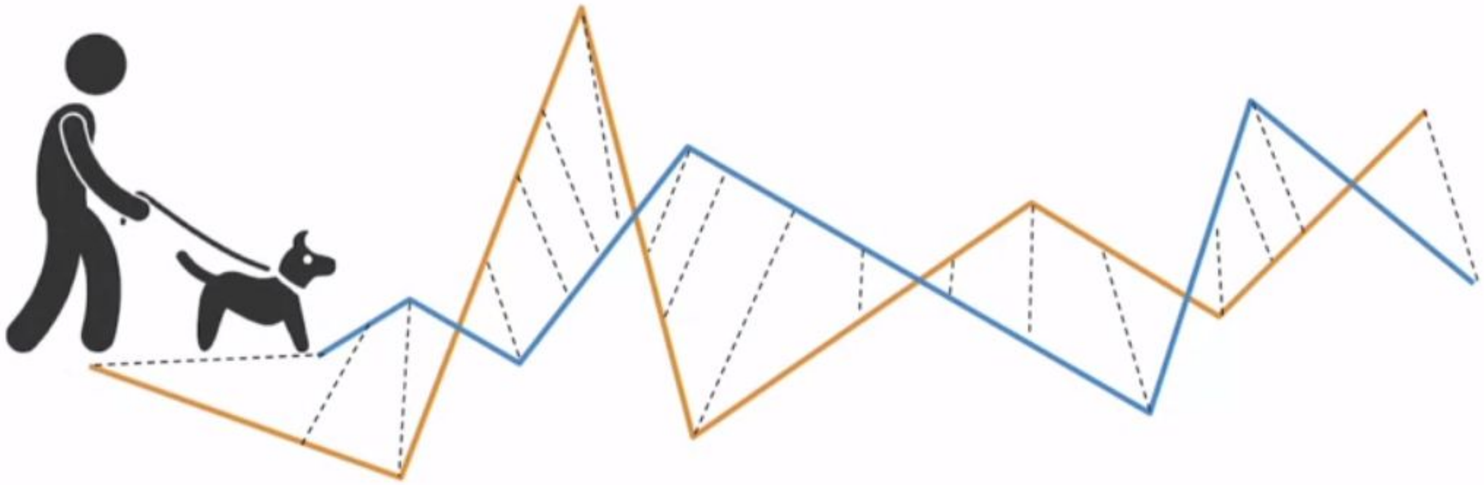


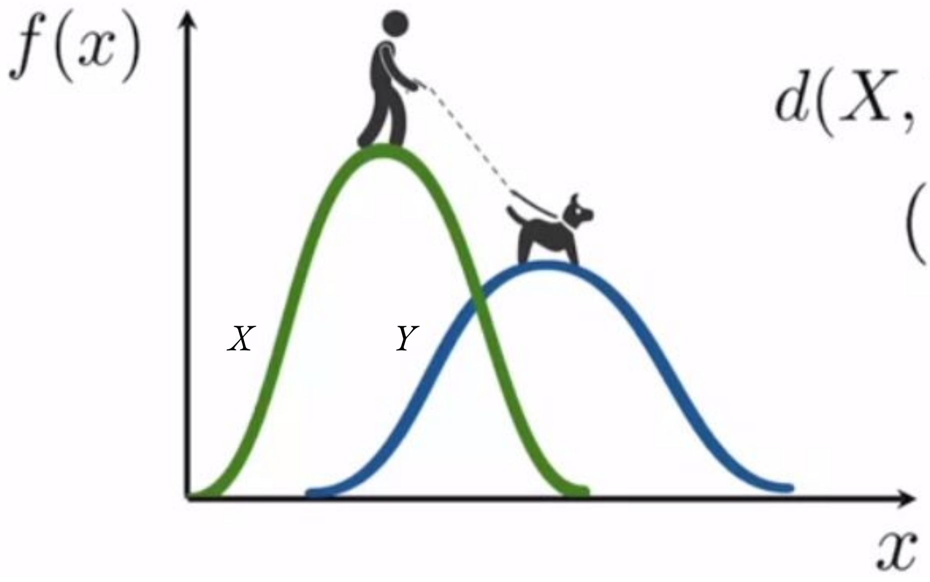
Image Embeddings



Fréchet distance



Fréchet distance 2 dimensions



$$d(X,Y) = (\mu_X - \mu_Y)^2 + (\sigma_X - \sigma_Y)^2$$

Moyenne

Écart-type

Fréchet distance 2 dimension to multidimension

$$(\mu_X - \mu_Y)^2 + (\sigma_X - \sigma_Y)^2$$

$$\downarrow$$

$$||\mu_X - \mu_Y||^2 + \mathrm{Tr}\left(\Sigma_X + \Sigma_Y - 2\sqrt{\Sigma_X \Sigma_Y}\right)$$

Fréchet inception distance (FID)

$$\text{FID} = \left\|\mu_X - \mu_Y\right\|^2 + \operatorname{Tr}\left(\Sigma_X + \Sigma_Y - 2\sqrt{\Sigma_X \Sigma_Y}\right)$$

X : Reel image Embeddings
(ex : 50 000 vecteurs de dimension 2048)
Y : Fake image Embeddings
(ex : 50 000 vecteurs de dimension 2048)

Conclusion :
Plus FID est petit plus les images sont proches donc plus FID est petit meilleur est notre générateur.

FID limits

- Utilisation d’un modèle inception pré-entraîné qui ne va pas forcément récupérer toutes les features des images. (dépend de la proximité de son dataset d’entraînement avec les images à évaluer)
- A besoin d’un large échantillon d’image pour éviter un maximum le bruit. (Au moins 50 000 images)
- Lent à exécuter

Pour résumer

- FID calcule la distance entre les image réelles et fakes
- FID utilise inception et la distance de Fréchet
- FID a besoin d’un large échantillon pour bien fonctionner

StyleGAN

Et Aujourd'hui, quel est l'état de l'art?

2014

2015

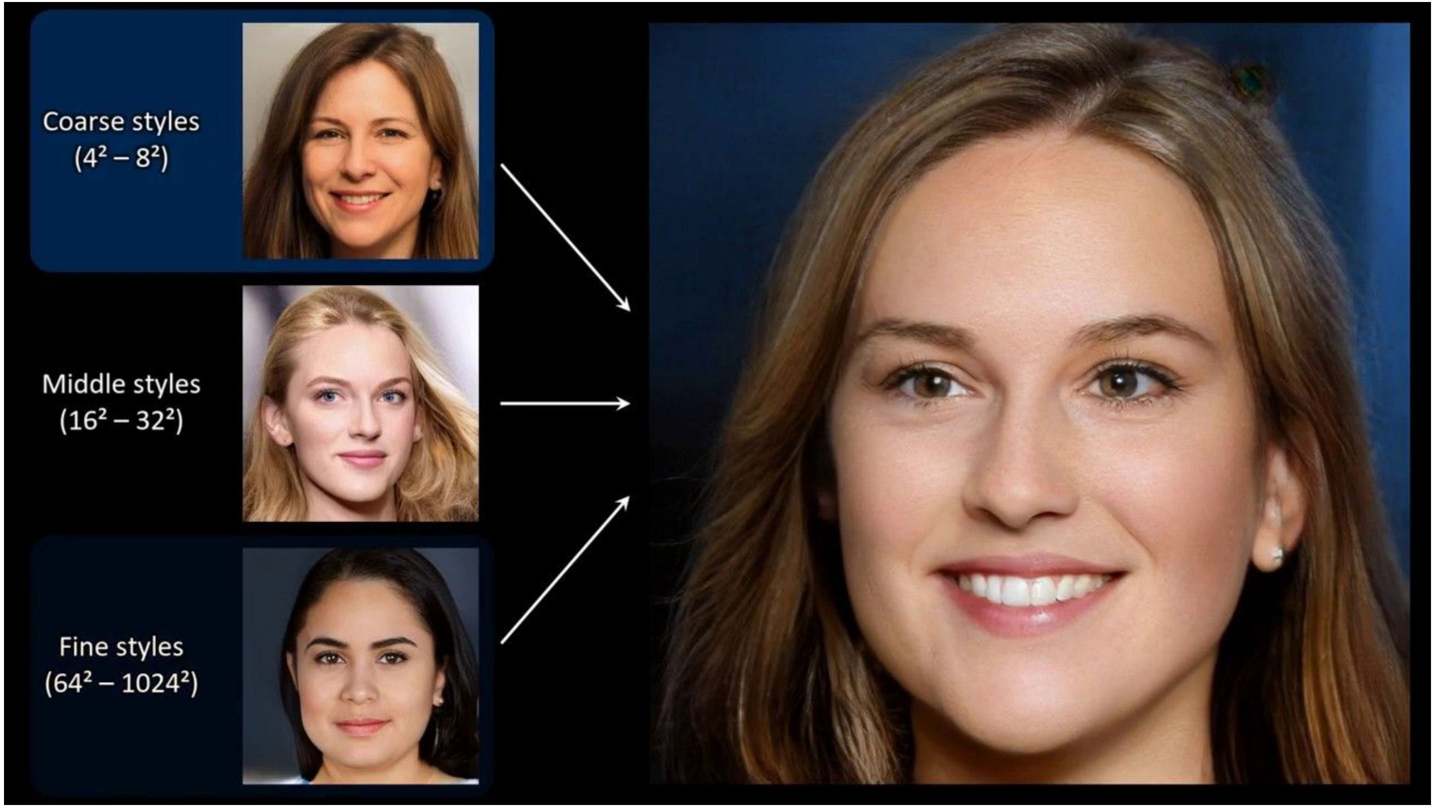
2016

2017

2018

https://twitter.com/goodfellow_ian/status/1084973596236144640/photo/1

StyleGAN

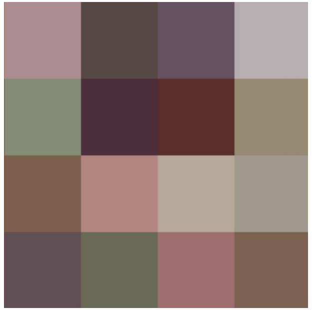


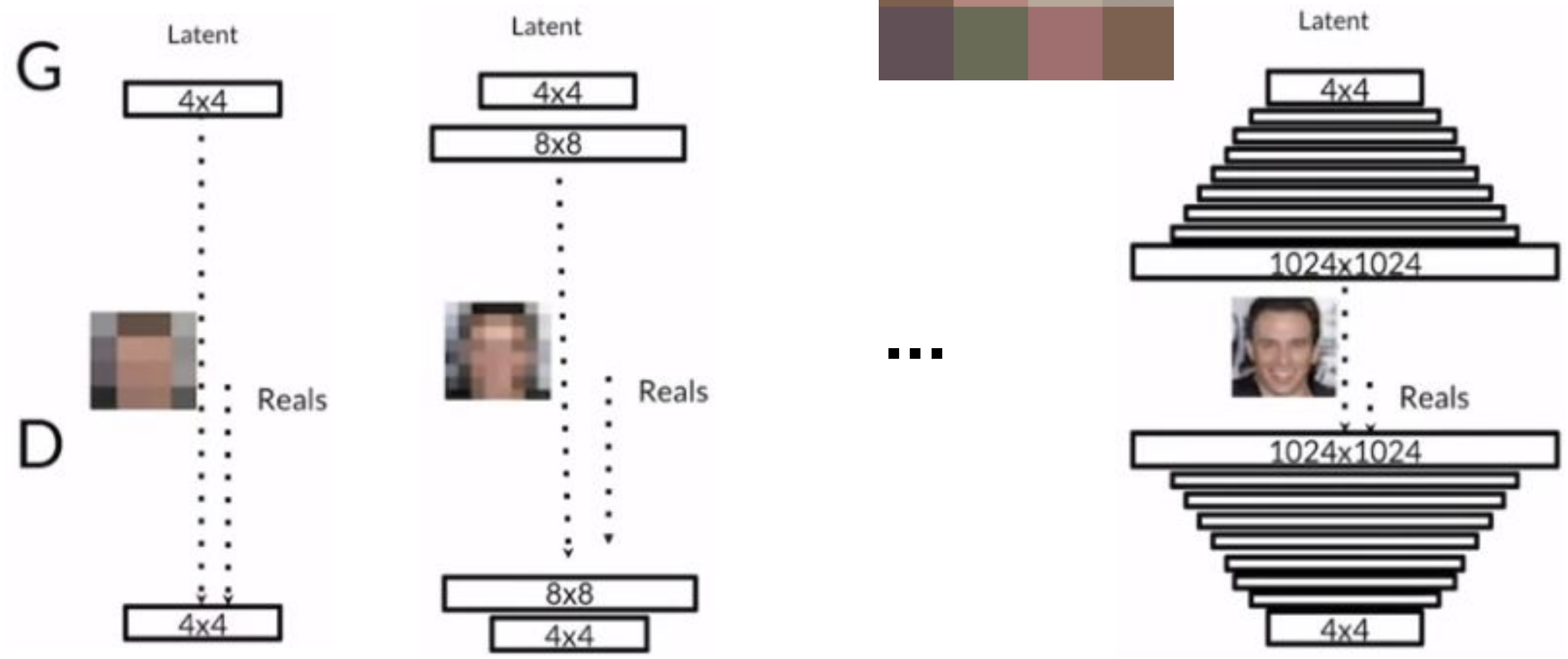
Coarse styles
(4² – 8²)
Middle styles
(16² – 32²)
Fine styles
(64² – 1024²)

StyleGAN

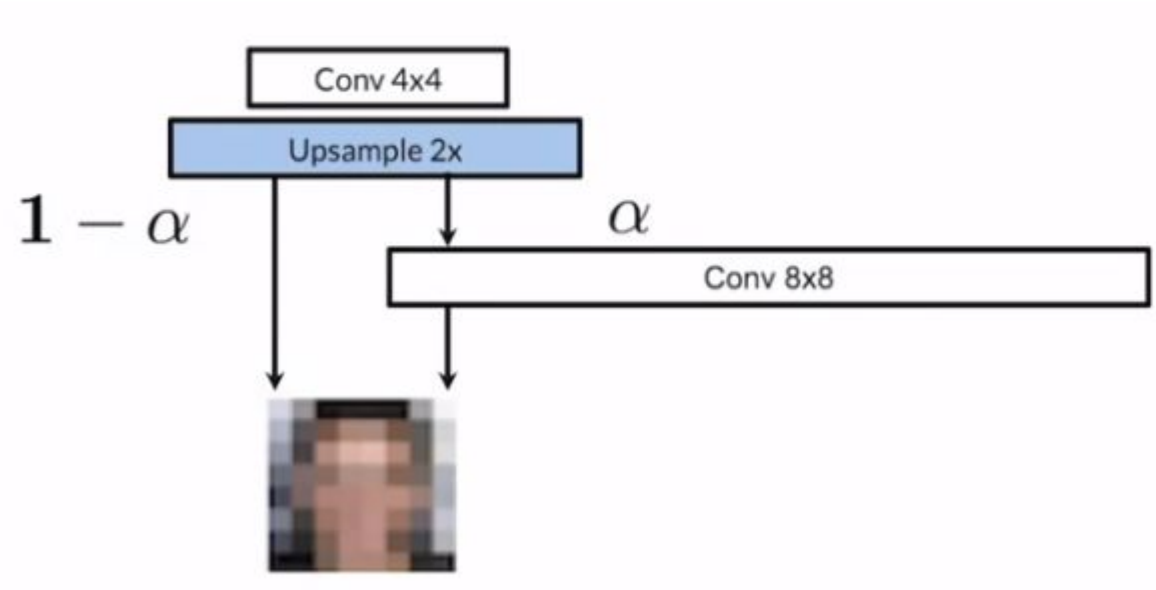
- Objectifs de StyleGAN
  - Plus grande fidélité
  - Plus grande diversité
  - Plus de contrôle sur les features
- Composants de StyleGAN :
  - Progressive Growing
  - Adaptive Instance Normalization (AdaIN)
  - Noise mapping network
  - Style mixing

Progressive growing

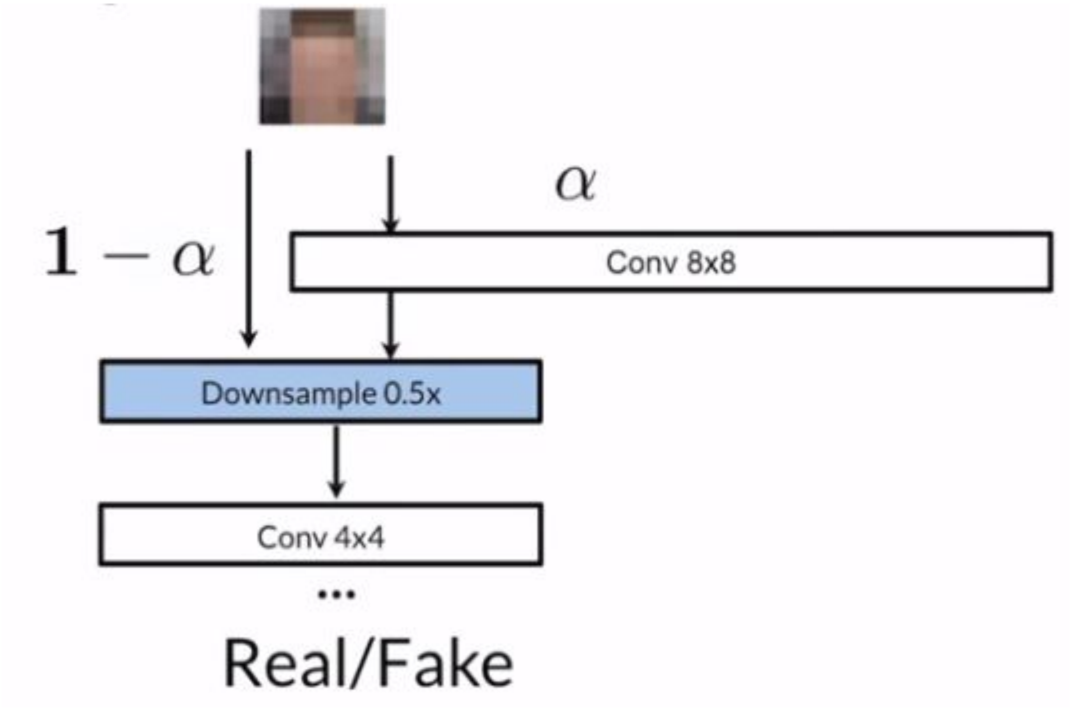




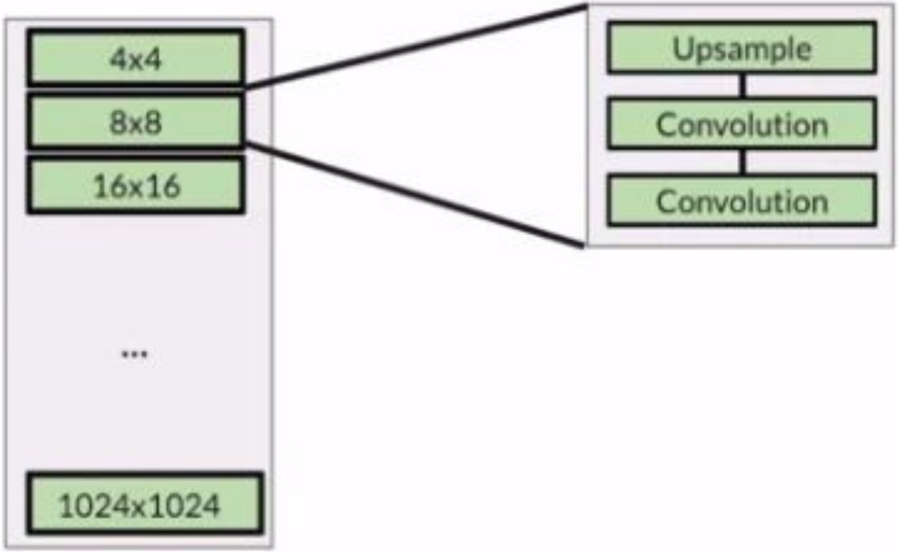
Progressive growing : Generator



Progressive growing : Discriminator



Progressive growing in context



Progressive growing Implementation

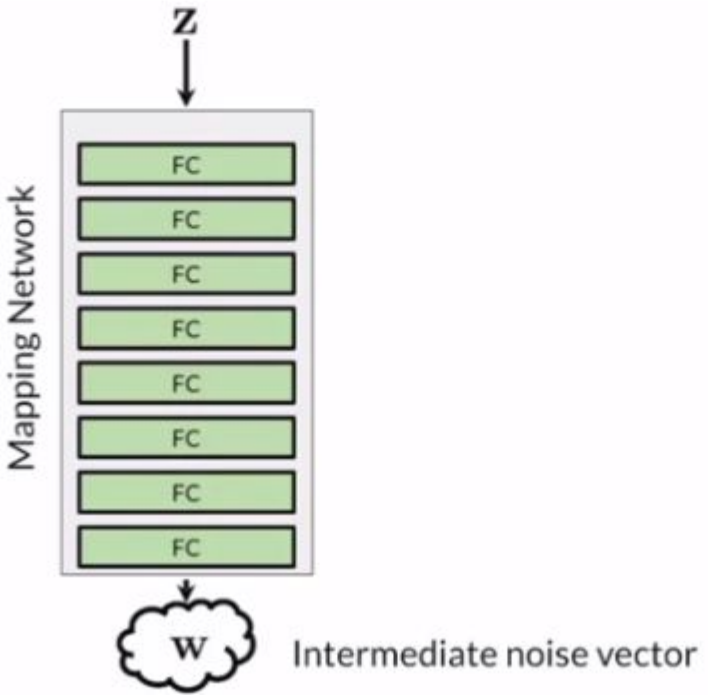
```python
class MicroStyleGANGeneratorBlock(nn.Module):
    '''
    Micro StyleGAN Generator Block Class
    Values:
        in_chan: the number of channels in the input, a scalar
        out_chan: the number of channels wanted in the output, a scalar
        w_dim: the dimension of the intermediate noise vector, a scalar
        kernel_size: the size of the convolving kernel
        starting_size: the size of the starting image
    '''

    def __init__(self, in_chan, out_chan, w_dim, kernel_size, starting_size, use_upsample=True):
        super().__init__()
        self.use_upsample = use_upsample
        # 1. Upsample to the starting_size, bilinearly (https://pytorch.org/docs/master/generated/torch.nn.Upsample.html)
        # 2. Create a kernel_size convolution which takes in
        #    an image with in_chan and outputs one with out_chan (https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
        # 3. Create an object to inject noise
        # 4. Create an AdaIN object
        # 5. Create a LeakyReLU activation with slope 0.2

        if self.use_upsample:
            self.upsample = nn.Upsample((starting_size, starting_size), mode='bilinear')
        self.conv = nn.Conv2d(in_chan, out_chan, kernel_size, padding=1) # Padding is used to maintain the image size
        self.inject_noise = InjectNoise(out_chan)
        self.adain = AdaIN(out_chan, w_dim)
        self.activation = nn.LeakyReLU(0.2)

    def forward(self, x, w):
        '''
        Function for completing a forward pass of MicroStyleGANGeneratorBlock: Given an x and w,
        computes a StyleGAN generator block.
        Parameters:
            x: the input into the generator, feature map of shape (n_samples, channels, width, height)
            w: the intermediate noise vector
        '''
        if self.use_upsample:
            x = self.upsample(x)
        x = self.conv(x)
        x = self.inject_noise(x)
        x = self.activation(x)
        x = self.adain(x, w)
        return x
```

```python
class MicroStyleGANGenerator(nn.Module):
    '''
    Micro StyleGAN Generator Class
    Values:
        z_dim: the dimension of the noise vector, a scalar
        map_hidden_dim: the mapping inner dimension, a scalar
        w_dim: the dimension of the intermediate noise vector, a scalar
        in_chan: the dimension of the constant input, usually w_dim, a scalar
        out_chan: the number of channels wanted in the output, a scalar
        kernel_size: the size of the convolving kernel
        hidden_chan: the inner dimension, a scalar
    '''
    def __init__(self, z_dim, map_hidden_dim, w_dim, in_chan, out_chan, kernel_size, hidden_chan):
        super().__init__()
        self.map = MappingLayers(z_dim, map_hidden_dim, w_dim)
        # Typically this constant is initiated to all ones, but you will initiate to a
        # Gaussian to better visualize the network's effect
        self.starting_constant = nn.Parameter(torch.randn(1, in_chan, 4, 4))
        self.block0 = MicroStyleGANGeneratorBlock(in_chan, hidden_chan, w_dim, kernel_size, 4, use_upsample=False)
        self.block1 = MicroStyleGANGeneratorBlock(hidden_chan, hidden_chan, w_dim, kernel_size, 8)
        self.block2 = MicroStyleGANGeneratorBlock(hidden_chan, hidden_chan, w_dim, kernel_size, 16)
        # You need to have a way of mapping from the output noise to an image,
        # so you learn a 1x1 convolution to transform the e.g. 512 channels into 3 channels
        # (Note that this is simplified, with clipping used in the real StyleGAN)
        self.block1_to_image = nn.Conv2d(hidden_chan, out_chan, kernel_size=1)
        self.block2_to_image = nn.Conv2d(hidden_chan, out_chan, kernel_size=1)
        self.alpha = 0.2

    def upsample_to_match_size(self, smaller_image, bigger_image):
        '''
        Function for upsampling an image to the size of another: Given a two images (smaller and bigger),
        upsamples the first to have the same dimensions as the second.
        Parameters:
            smaller_image: the smaller image to upsample
            bigger_image: the bigger image whose dimensions will be upsampled to
        '''
        return F.interpolate(smaller_image, size=bigger_image.shape[-2:], mode='bilinear')

    def forward(self, noise, return_intermediate=False):
        '''
        Function for completing a forward pass of MicroStyleGANGenerator: Given noise,
        computes a StyleGAN iteration.
        Parameters:
            noise: a noise tensor with dimensions (n_samples, z_dim)
            return_intermediate: a boolean, true to return the images as well (for testing) and false otherwise
        '''
        x = self.starting_constant
        w = self.map(noise)
        x = self.block0(x, w)
        x_small = self.block1(x, w) # First generator run output
        x_small_image = self.block1_to_image(x_small)
        x_big = self.block2(x_small, w) # Second generator run output
        x_big_image = self.block2_to_image(x_big)
        # Upsample first generator run output to be same size as second generator run output
        x_small_upsample = self.upsample_to_match_size(x_small_image, x_big_image)
        # Interpolate between the upsampled image and the image from the generator using alpha
        interpolation = torch.lerp(x_small_upsample, x_big_image, self.alpha)
        if return_intermediate:
            return interpolation, x_small_upsample, x_big_image
        return interpolation
```

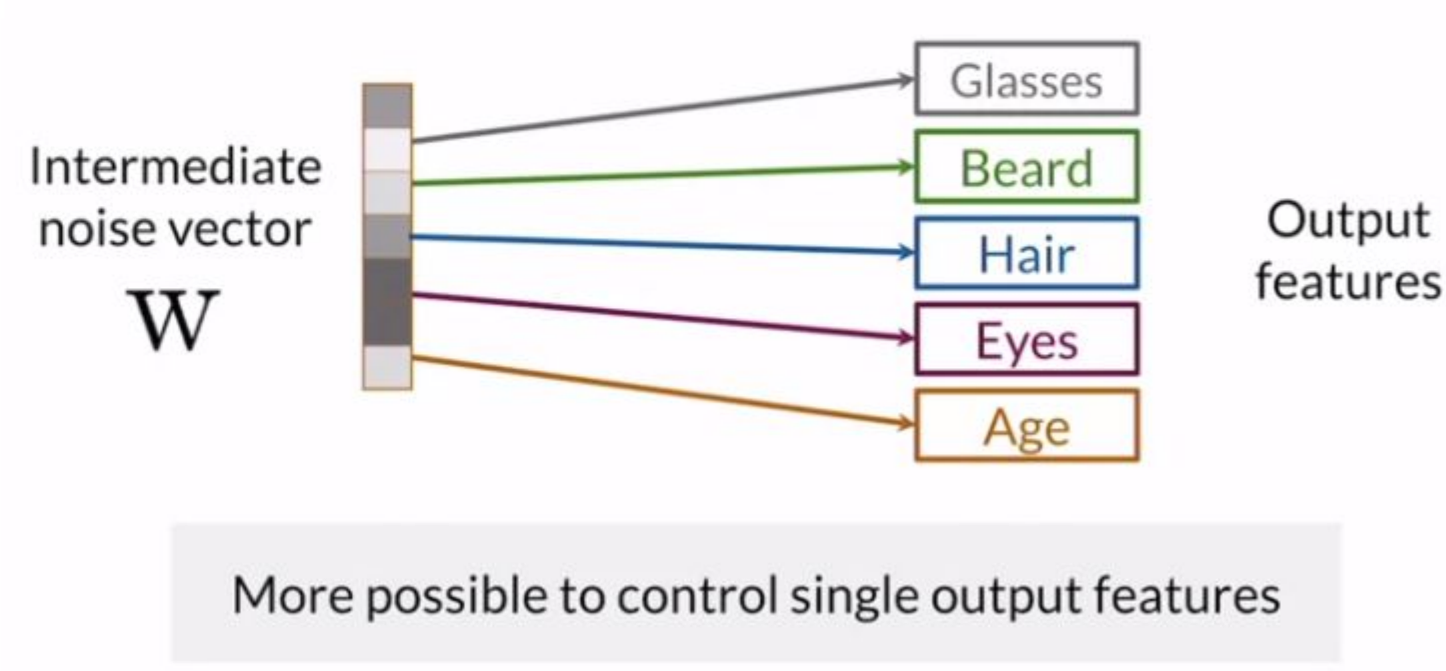
En résumé

- Progressive growing double progressivement la taille des images générées
- Progressive growing entraîne un entraînement plus rapide, plus stable et une meilleur résolution d’image

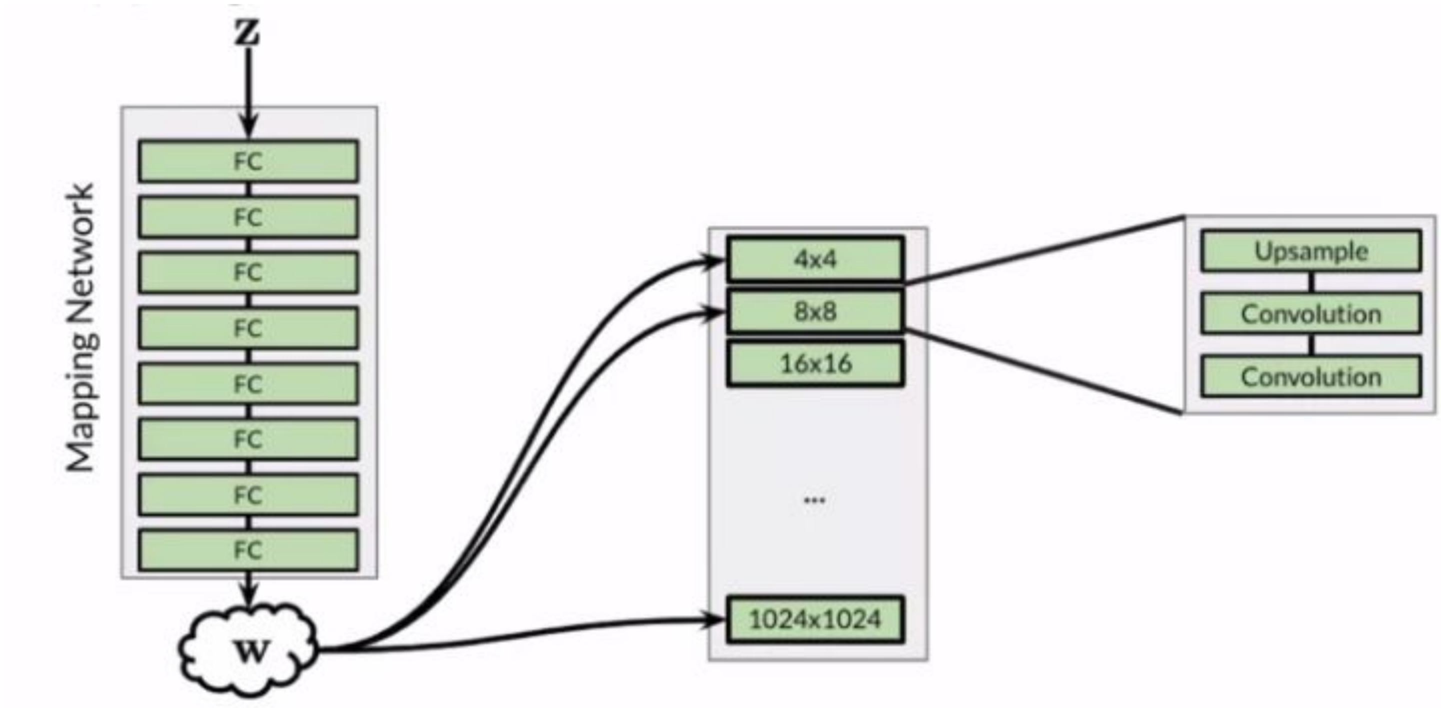
Noise Mapping Network



Noise Mapping Network



Noise Mapping Network + Progressive Growing



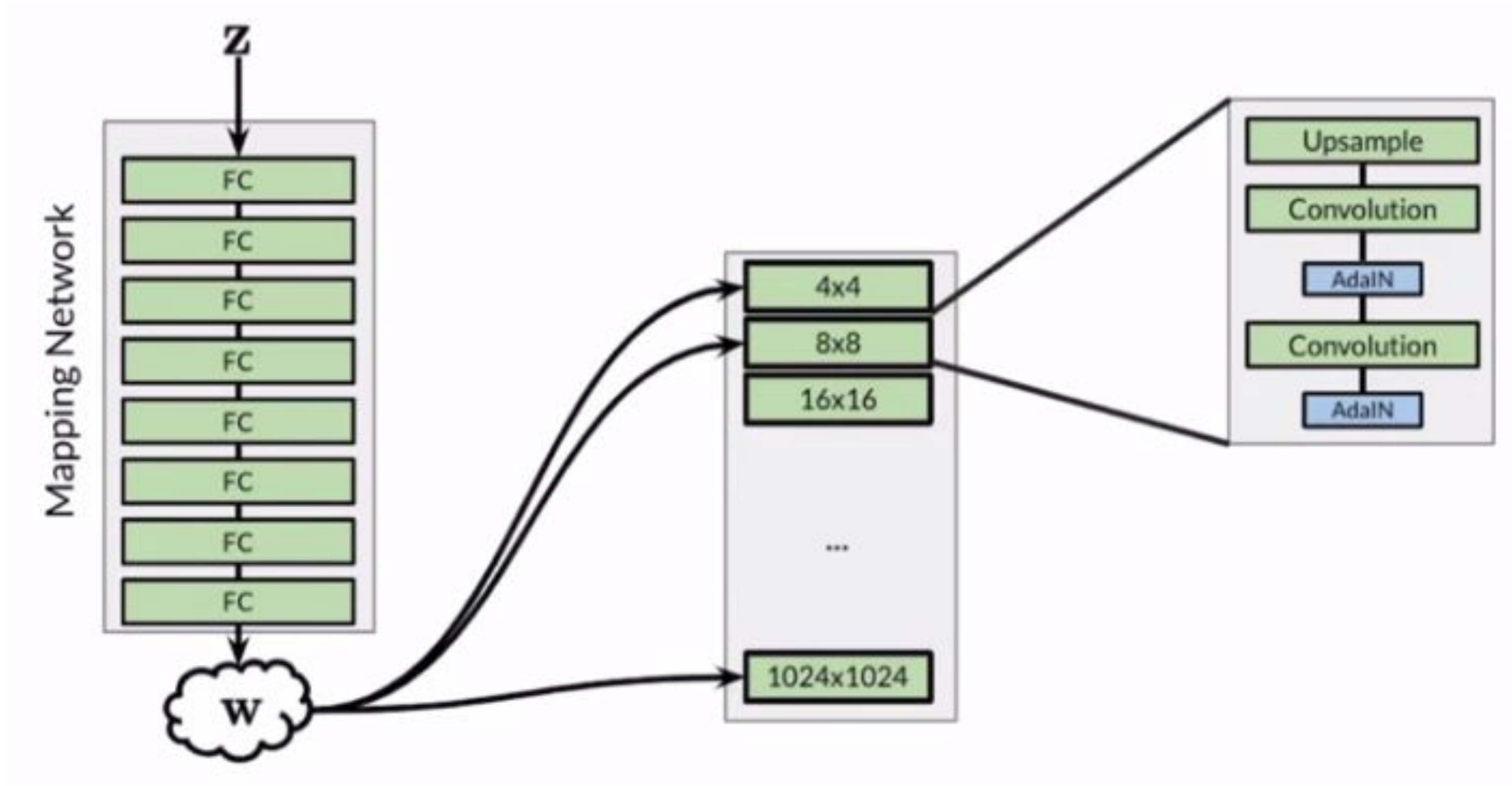
Noise Mapping Network implementation

```
class MappingLayers(nn.Module):
    '''
    Mapping Layers Class
    Values:
        z_dim: the dimension of the noise vector, a scalar
        hidden_dim: the inner dimension, a scalar
        w_dim: the dimension of the intermediate noise vector, a scalar
    '''

    def __init__(self, z_dim, hidden_dim, w_dim):
        super().__init__()
        self.mapping = nn.Sequential(
            nn.Linear(z_dim,hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim,hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim,w_dim)
        )

    def forward(self, noise):
        '''
        Function for completing a forward pass of MappingLayers:
        Given an initial noise tensor, returns the intermediate noise tensor.
        Parameters:
            noise: a noise tensor with dimensions (n_samples, z_dim)
        '''
        return self.mapping(noise)
```

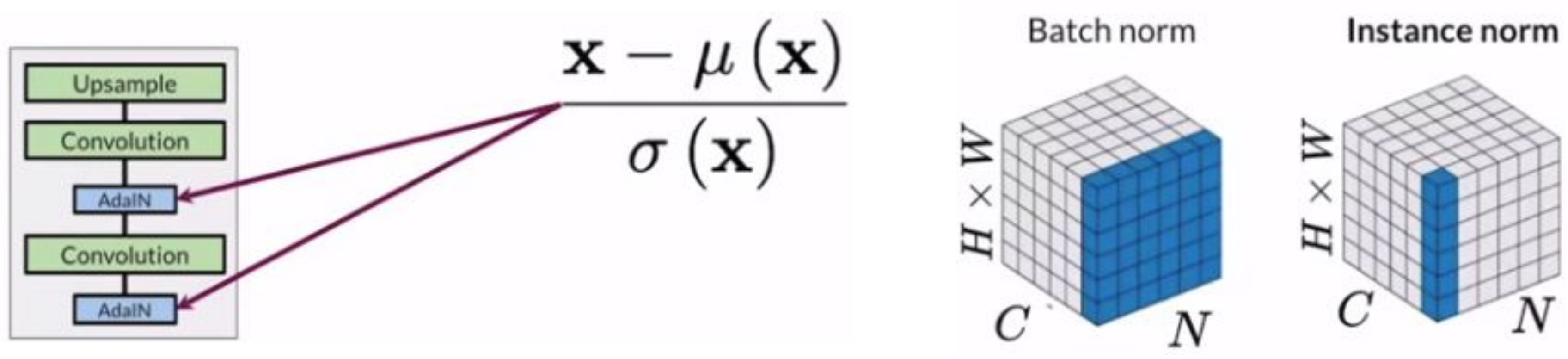
En résumé

- Noise Mapping permet de mieux contrôler les features
- W le vecteur de bruit intermédiaire est injecté à différents niveau dans le générateur

Adaptive Instance Normalization (AdaIN)

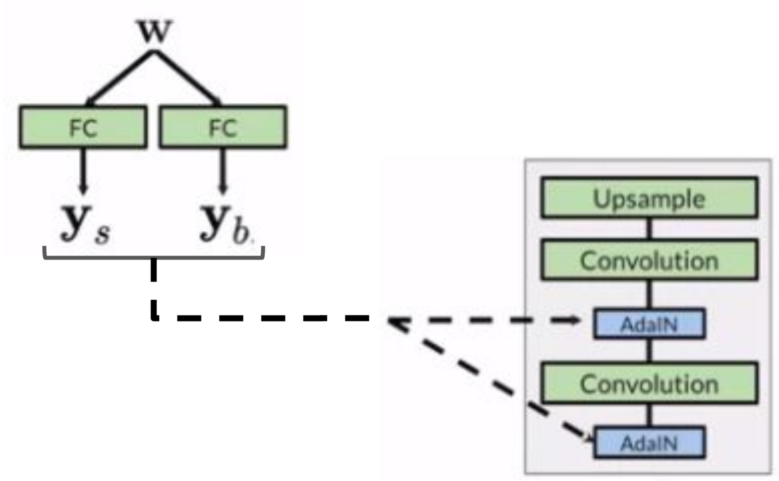


Adaptive Instance Normalization (AdaIN)



Etape 1 : Normaliser la sortie des couches de convolution en utilisant l'**Instance Normalisation**

Adaptive Instance Normalization (AdaIN)



$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \underbrace{\frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)}}_{\text{Étape 1 : Instance normalization}} + \mathbf{y}_{b,i}$$

Etape 2 : Appliquer l’**Adaptive Style** en utilisant W le vecteur de bruit intermédiaire

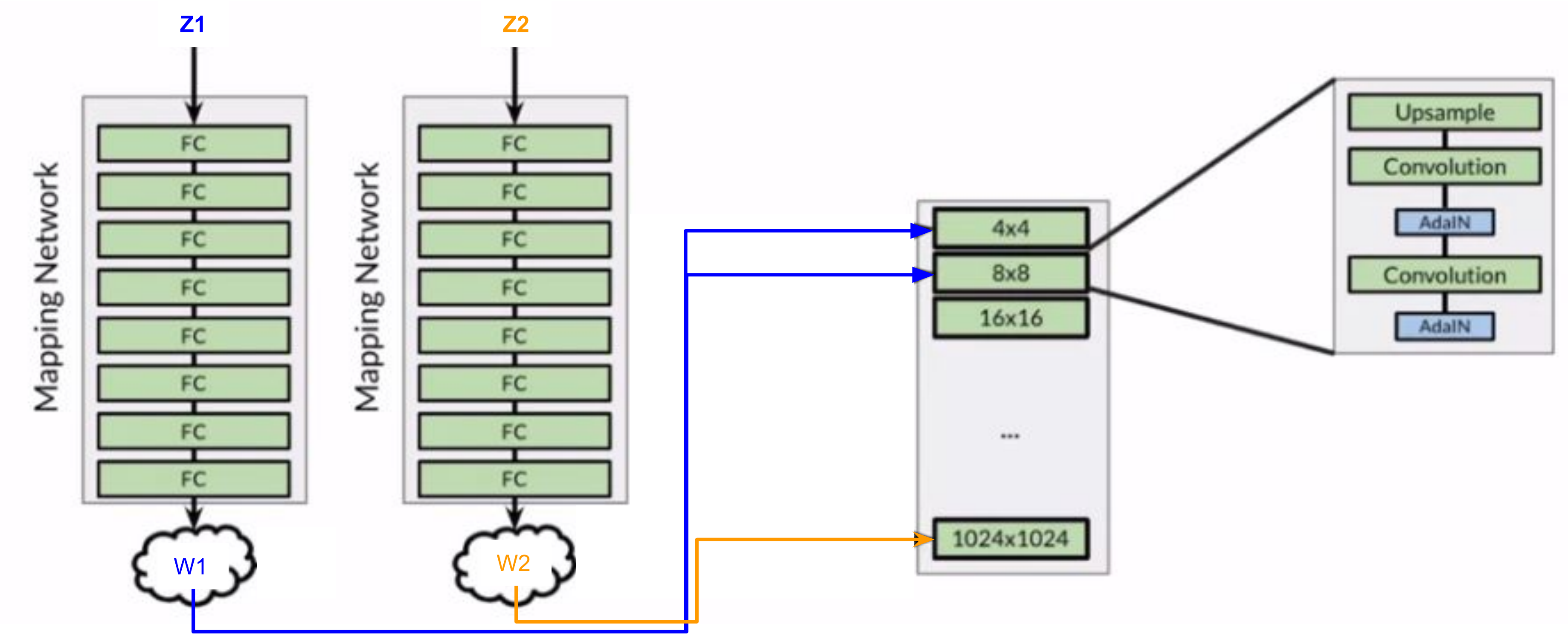
AdaIN implementation

```python
class AdaIN(nn.Module):
    '''
    AdaIN Class
    Values:
        channels: the number of channels the image has, a scalar
        w_dim: the dimension of the intermediate noise vector, a scalar
    '''

    def __init__(self, channels, w_dim):
        super().__init__()

        # Normalize the input per-dimension
        self.instance_norm = nn.InstanceNorm2d(channels)

        # You want to map w to a set of style weights per channel.
        # Replace the Nones with the correct dimensions - keep in mind that
        # both linear maps transform a w vector into style weights
        # corresponding to the number of image channels.
        self.style_scale_transform = nn.Linear(w_dim, channels)
        self.style_shift_transform = nn.Linear(w_dim, channels)

    def forward(self, image, w):
        '''
        Function for completing a forward pass of AdaIN: Given an image and intermediate noise vector w,
        returns the normalized image that has been scaled and shifted by the style.
        Parameters:
            image: the feature map of shape (n_samples, channels, width, height)
            w: the intermediate noise vector
        '''
        normalized_image = self.instance_norm(image)
        style_scale = self.style_scale_transform(w)[:, :, None, None]
        style_shift = self.style_shift_transform(w)[:, :, None, None]

        # Calculate the transformed image
        transformed_image = style_scale * normalized_image + style_shift
        return transformed_image
```

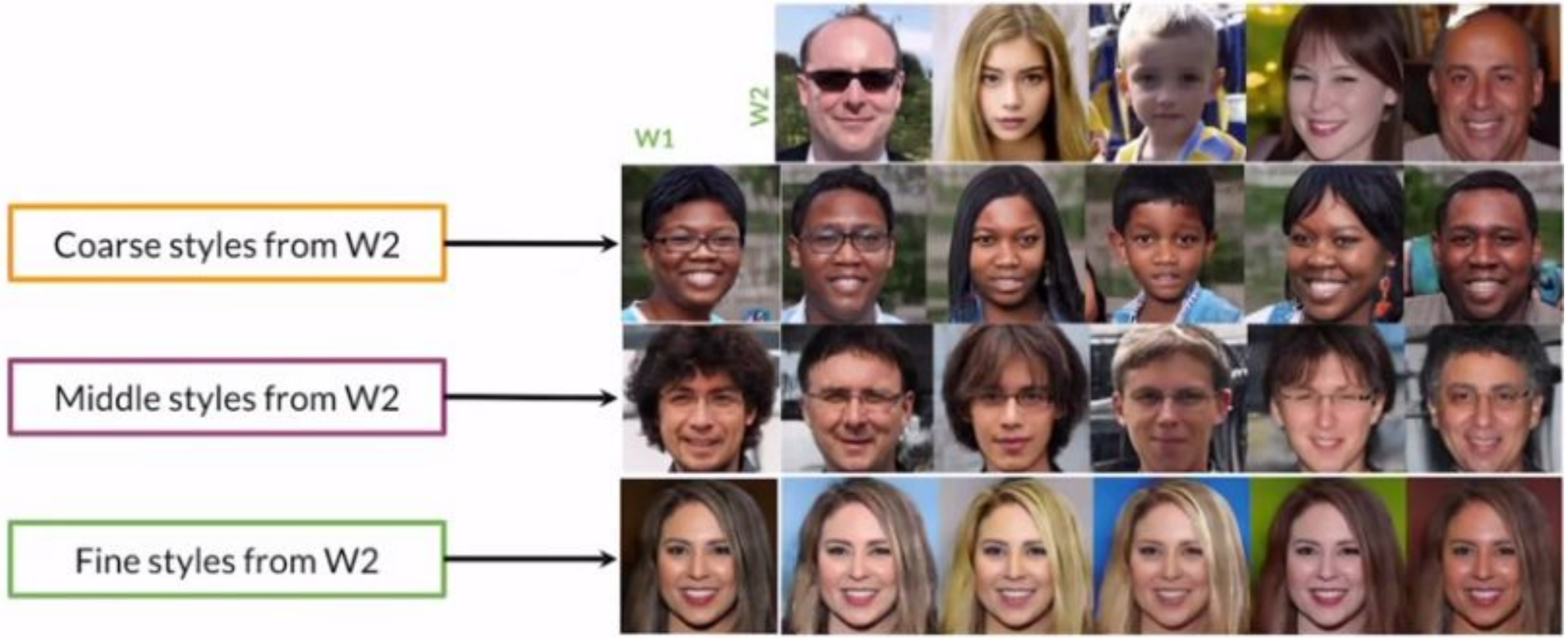
En résumé

- Instance Normalization est utilisé pour normaliser individuellement les exemples avant de leur appliquer un style
- AdaIN transfère les informations de style du vecteur de bruit intermédiaire w vers l'image générée.

Style Mixing



Style Mixing



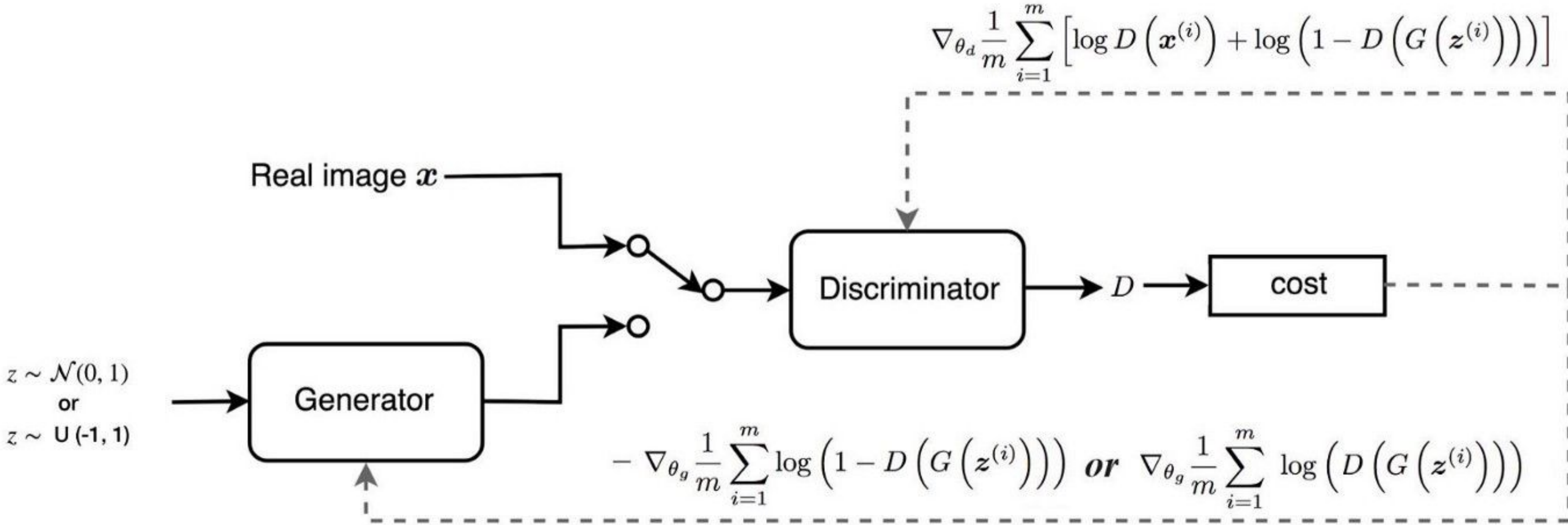
Available from: https://arxiv.org/abs/1812.04948

Merci !

Annexe

collection of implementation gan

https://github.com/eriklindernoren/Keras-GAN



$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$
Real image $\boldsymbol{x}$
Discriminator
D
cost
$z \sim \mathcal{N}(0, 1)$
or
$z \sim$ U (-1, 1)
Generator
$-\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$ ***or*** $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$

Binary cross-entropy cost function

real $y^{(i)}=1$ **fake $y^{(i)}=0$**

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\boxed{[y^{(i)}\log h(x^{(i)},\theta)} + \boxed{(1-y^{(i)})\log(1-h(x^{(i)},\theta))]}$$

Prediction Label Features Parameters

$h(x^{(i)},\theta)$	$\log h(x^{(i)},\theta)$	$\log(1-h(x^{(i)},\theta))]$
~0	- ∞	~0
~1	~0	-∞

Objectif pour :

- le Générateur : J(θ) plus grand possible
- le Discriminateur : J(θ) plus proche de 0 possible