

GENERATIVE ADVERSARIAL NETWORKS

Par Kevin Bouchard Ph.D.

Professeur titulaire en intelligence artificielle et apprentissage automatique
Laboratoire d'Intelligence Ambiante pour la reconnaissance d'activités (LIARA)
Directeur de l'Espace innovation en technologies numériques Hydro-Québec
Président du Regroupement québécois des maladies orphelines (RQMO)
Université du Québec à Chicoutimi

www.Kevin-Bouchard.ca

Kevin_Bouchard@uqac.ca

1

TABLE DES MATIÈRES

1. Introduction au GAN
2. Principes de base
3. DCGAN
4. CGAN
5. Autres GAN en bref
6. Exemple complet

INTRODUCTION

- Dans ce module, nous introduisons un type particulier de réseaux de neurones
- Inventés en 2014 par Goodfellow, les Generative Adversarial Networks (GAN) sont devenus très populaires dans les dernières années
- Ce sont des réseaux génératifs
 - Contrairement aux auto-encodeurs, ils peuvent créer de nouvelles données avec un encodage arbitraire
- Dans ce module, nous:
 1. Introduisons les concepts de base des GAN
 2. Comment les implémenter sous Keras
 3. Nous discuterons des principaux types de GAN

SURVOL DES GAN



- Commençons par une petite vidéo assez impressionnante démontrant l'étendue de leurs pouvoirs



*Progressive GAN - 2017

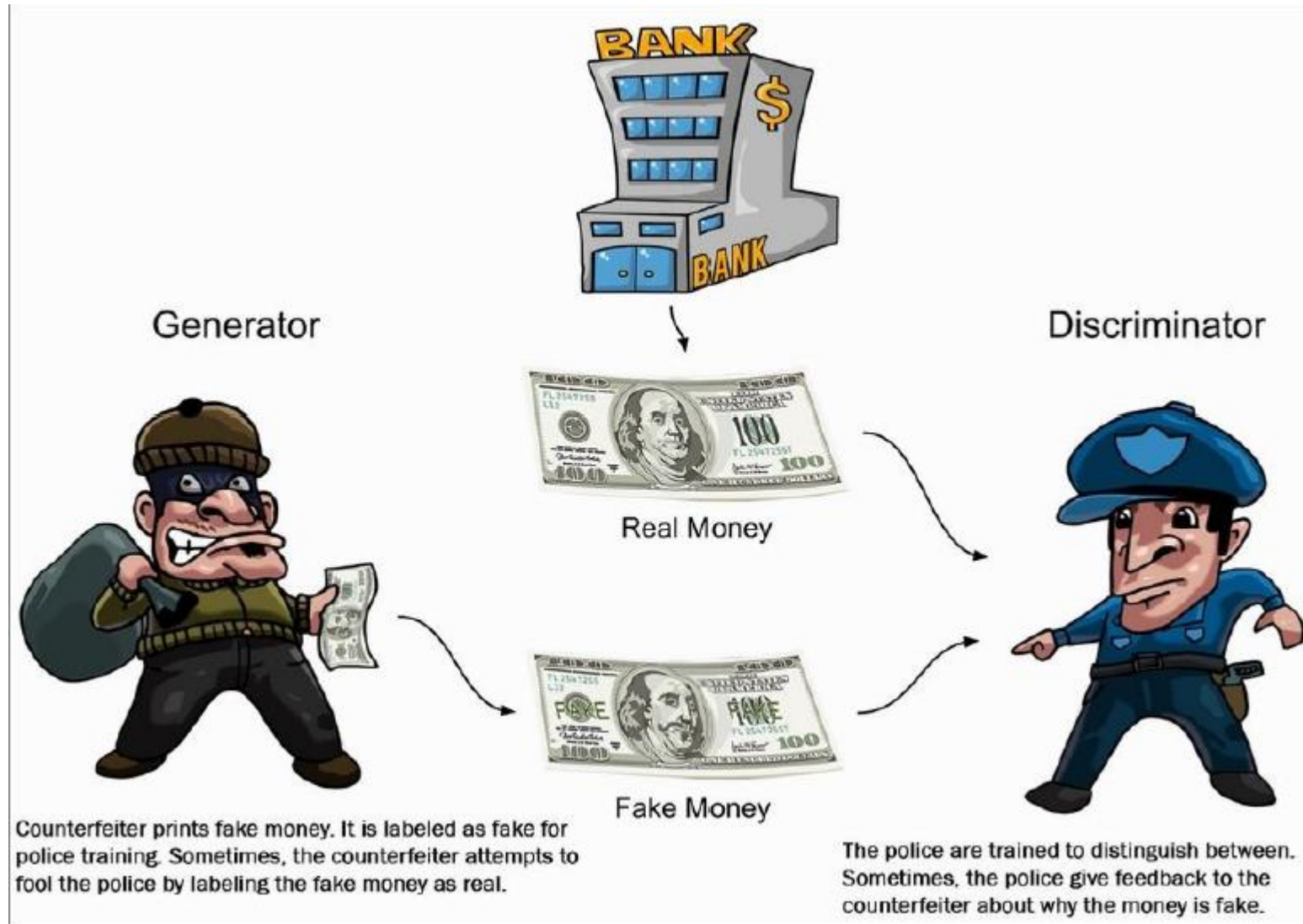
SURVOL DES GAN

- Cette qualité de génération de visages n'est pas possible avec les VAE simple que nous avons vu précédemment
- Les GAN réussissent ces exploits en entraînant deux réseaux en compétition coopérative
 - **Générateur**: il vise à apprendre à générer de fausses données (images, mais aussi textes, sons, etc.) pour tromper le discriminateur
 - **Discriminateur**: il va tenter d'apprendre à détecter les fausses données (problème de classification binaire +/-)
- Au fil de l'entraînement, le discriminateur n'arrivera plus à détecter les fausses
- Il pourra donc être éliminer et nous utiliserons le générateur pour créer de nouvelles données

DÉFIS DES GAN

- L'intuition derrière les GAN est simple
- Comment trouver un bon compromis dans l'entraînement entre le générateur et le discriminateur?
 - Nous avons besoin d'une compétition saine entre les deux
 - Sinon, ils ne parviendront pas à apprendre tous les deux!!!
- Les GAN sont en fait très difficiles à entraîner!
 - Si le discriminateur converge plus vite, le générateur ne reçoit plus suffisamment de gradients pour converger!!!
- Les GAN souffrent aussi du problème d'effondrement
 - Le générateur produit des sorties presque similaires pour différents encodages latents!

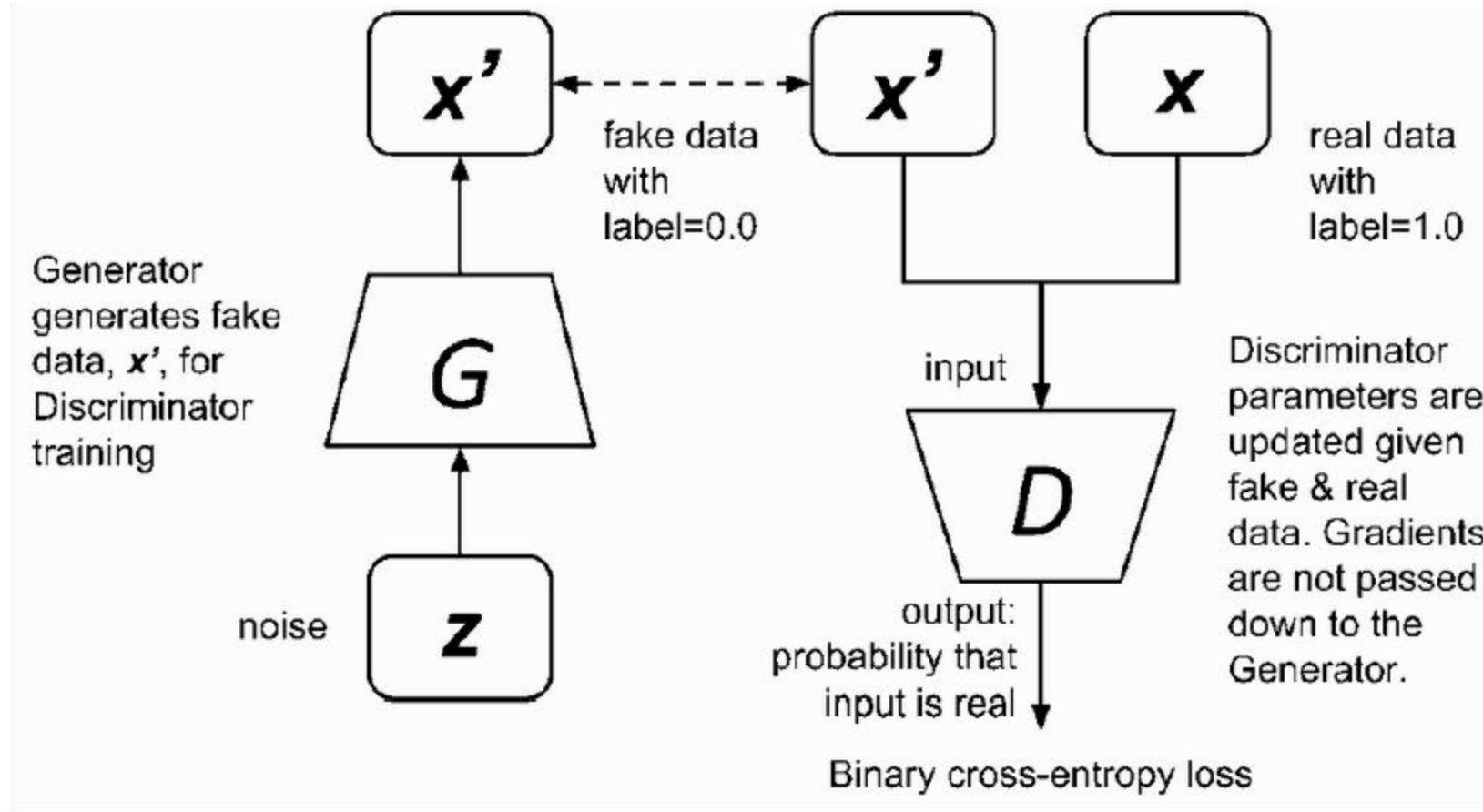
PRINCIPES DE BASE



PRINCIPES DE BASE

- L'entrée du générateur est du bruit (vecteur aléatoire)
- Sa sortie est la donnée synthétisée
- L'entrée du discriminateur est une vraie ou une fausse donnée
 - Les vraies données viennent d'un vrai ensemble d'entraînement
 - Les fausses du générateur
- Sa sortie est 1 ou 0 (100% vraie ou 0% vraie)
- Notez que nous n'avons pas besoin d'étiquettes!!!

PRINCIPES DE BASE



LE DISCRIMINATEUR

- Le discriminateur peut être entraîné avec la fonction de coût d'entropie croisée binaire (vue précédemment)

$$\mathcal{L}^{(D)}(\theta^{(G)}, \theta^{(D)}) = -\mathbb{E}_{x \sim p_{data}} \log \mathcal{D}(x) - \mathbb{E}_z \log (1 - \mathcal{D}(\mathcal{G}(z)))$$

- La perte est la somme négative de l'attente d'une identification correcte des données réelles
- et l'attente de 1.0 moins identifier correctement les données synthétiques
- Le log ne change pas l'emplacement de minimum locaux/globaux
 - Il évite les extrêmes!

LE DISCRIMINATEUR

- Deux minibatch de données sont fournies au discriminateur durant l'entraînement
 - x les données réelles avec étiquette 1
 - $x' = G(z)$ les fausses données du générateur avec étiquette 0
- Pour minimiser la fonction de perte, les paramètres du discriminateur $\theta^{(D)}$ seront mis à jour
 - avec backpropagation
 - en identifiant les vraies données $D(x) \rightarrow 1.0$
 - et les données synthétiques $1.0 - D(G(z)) \rightarrow 1.0$
- z est le vecteur de bruit arbitraire utilisé par le générateur pour synthétiser nouveaux signaux

LE GÉNÉRATEUR

- Pour entraîner le générateur, le GAN considère le total des pertes du discriminateur et du générateur comme un jeu à somme nulle
- La fonction de perte est donc le négatif de celle du discriminateur
- Du point de vue du généré $\mathcal{V}^{(G)}(\theta^{(G)}, \theta^{(D)}) = -\mathcal{L}^{(D)}(\theta^{(G)}, \theta^{(D)})$ e minimisée
- Du discriminateur, elle doit être maximisée
- C'est donc un problème minimax!!!

$$\min_G \max_D V^{(G)}(\theta^{(G)}, \theta^{(D)})$$

RELATION PAR RAPPORT AUX VAE

- Les VAE minimisent indirectement une fonction objective
- Les GAN tentent de minimiser directement la fonction objective en formant les discriminateurs pour apprendre la fonction objective d'un générateur fixe
 - Combien pouvons-nous changer le Générateur?
 - Tout en ayant le Discriminateur comme une bonne approximation ?
- Le cadre des GAN peut inclure des mesures alternatives de divergences pour objectif

FORCES DES GAN

- On peut utiliser la backpropagation
- Les données générées par les GAN ressemblent aux données d'origine
 - Donc, d'un point de vue humain, elles semblent plus réalistes
- Pas besoin d'étiquettes pour les entraînés
 - C'est de l'auto-apprentissage!!!
 - On peut aussi dire que c'est non-supervisé, mais je préfère le premier terme qui est plus spécifique

FAIBLESSES DES GAN

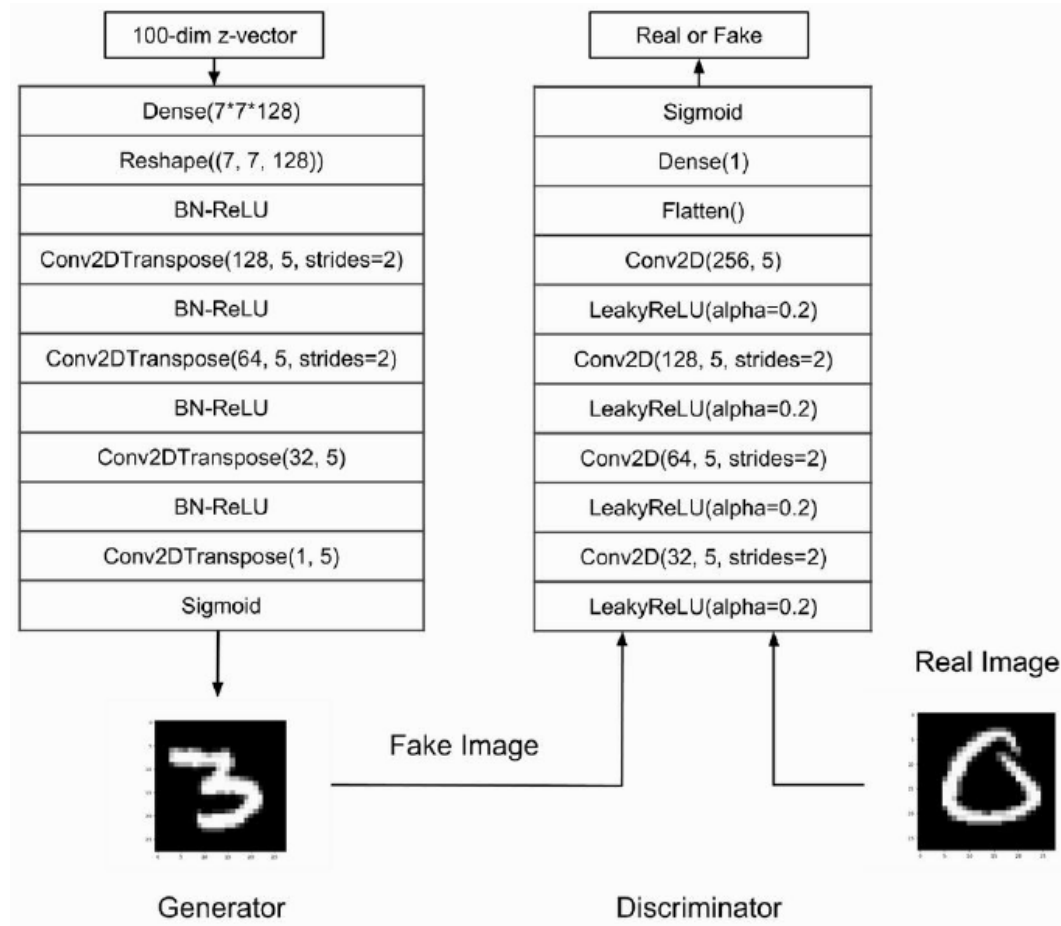
- Critères d'arrêt peu clairs
- Facilement emprisonnés dans des optima locaux qui mémorisent les données d'entraînement
- Modèle génératif difficile à inverser pour récupérer le z latent du x généré
 - Ce qui est parfois un avantage!
- Aucune mesure d'évaluation donc difficile à comparer avec d'autres modèles
- Difficile à entraîner (outils immatures pour l'optimisation)
 - De moins en moins!



DCGAN



DEEP CONVOLUTIONAL GAN (DCGAN)



- Strides > 1 plutôt que pooling
- Juste des couches de convolution, pas de denses!
 - Sauf au début pour la source
- Batch normalization: stabiliser l'apprentissage en normalisant l'entrée à chaque couche pour avoir une moyenne nulle et une variance unitaire
 - Pas de BN dans la couche de sortie du générateur et couche d'entrée du discriminateur.
- Leaky dans le discriminateur pour éviter d'avoir des activations à 0 (ReLU)

Model: "generator"

Layer (type)	Output Shape	Param #
z_input (InputLayer)	[(None, 100)]	0
dense_1 (Dense)	(None, 6272)	633472
reshape (Reshape)	(None, 7, 7, 128)	0
batch_normalization (Batch Normalization)	(None, 7, 7, 128)	512
activation_1 (Activation)	(None, 7, 7, 128)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 128)	409728
batch_normalization_1 (Batch Normalization)	(None, 14, 14, 128)	512
activation_2 (Activation)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 64)	204864
batch_normalization_2 (Batch Normalization)	(None, 28, 28, 64)	256
activation_3 (Activation)	(None, 28, 28, 64)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 28, 28, 32)	51232
batch_normalization_3 (Batch Normalization)	(None, 28, 28, 32)	128
activation_4 (Activation)	(None, 28, 28, 32)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 28, 28, 1)	801
activation_5 (Activation)	(None, 28, 28, 1)	0

Total params: 1,301,505

Trainable params: 1,300,801

Non-trainable params: 704

Layer (type)	Output Shape	Param #
discriminator_input (InputLayer)	[(None, 28, 28, 1)]	0
leaky_re_lu (LeakyReLU)	(None, 28, 28, 1)	0
conv2d (Conv2D)	(None, 14, 14, 32)	832
leaky_re_lu_1 (LeakyReLU)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 7, 7, 64)	51264
leaky_re_lu_2 (LeakyReLU)	(None, 7, 7, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	204928
leaky_re_lu_3 (LeakyReLU)	(None, 4, 4, 128)	0
conv2d_3 (Conv2D)	(None, 4, 4, 256)	819456
flatten (Flatten)	(None, 4096)	0
dense (Dense)	(None, 1)	4097
activation (Activation)	(None, 1)	0

Total params: 1,080,577

Trainable params: 1,080,577

Non-trainable params: 0

GÉNÉRATEUR

- On génère les fausses images à partir de vecteurs de taille 100
 - Aléatoire entre [-1 et 1]
- Les noyaux sont de taille 5x5
- Après la couche sigmoid, nous avons 28 x 28 x 1 avec chaque pixel normalisé entre [0.0,1.0]
 - correspondant à nos pixels en niveaux de gris [0, 255]
- VOIR CODE
 - `build_generator`

DISCRIMINATEUR

- Très similaire à nos CNN standard
- L'entrée est 28 x 28 x 1 (nos images MNIST)
- 4 couches de convolution avec stride=2 sauf le dernier
 - On downsample les filtres
 - Encore de taille 5
- Un termine par une couche dense (1 unité)
 - Sigmoid, prédiction entre 1.0 et 0.0 (probabilité d'être vraie ou fausse)
- VOIR CODE
 - `build_discriminator`
 - `build_and_train_models()`

DCGAN

- Attention! Si la batch normalization est utilisé dans le discriminateur notre GAN ne converge pas
- Même chose si le stride=2 est changé dans les deux dernières couches plutôt que les premières



500

5000

10000



15000

20000

25000



30000

35000

40000

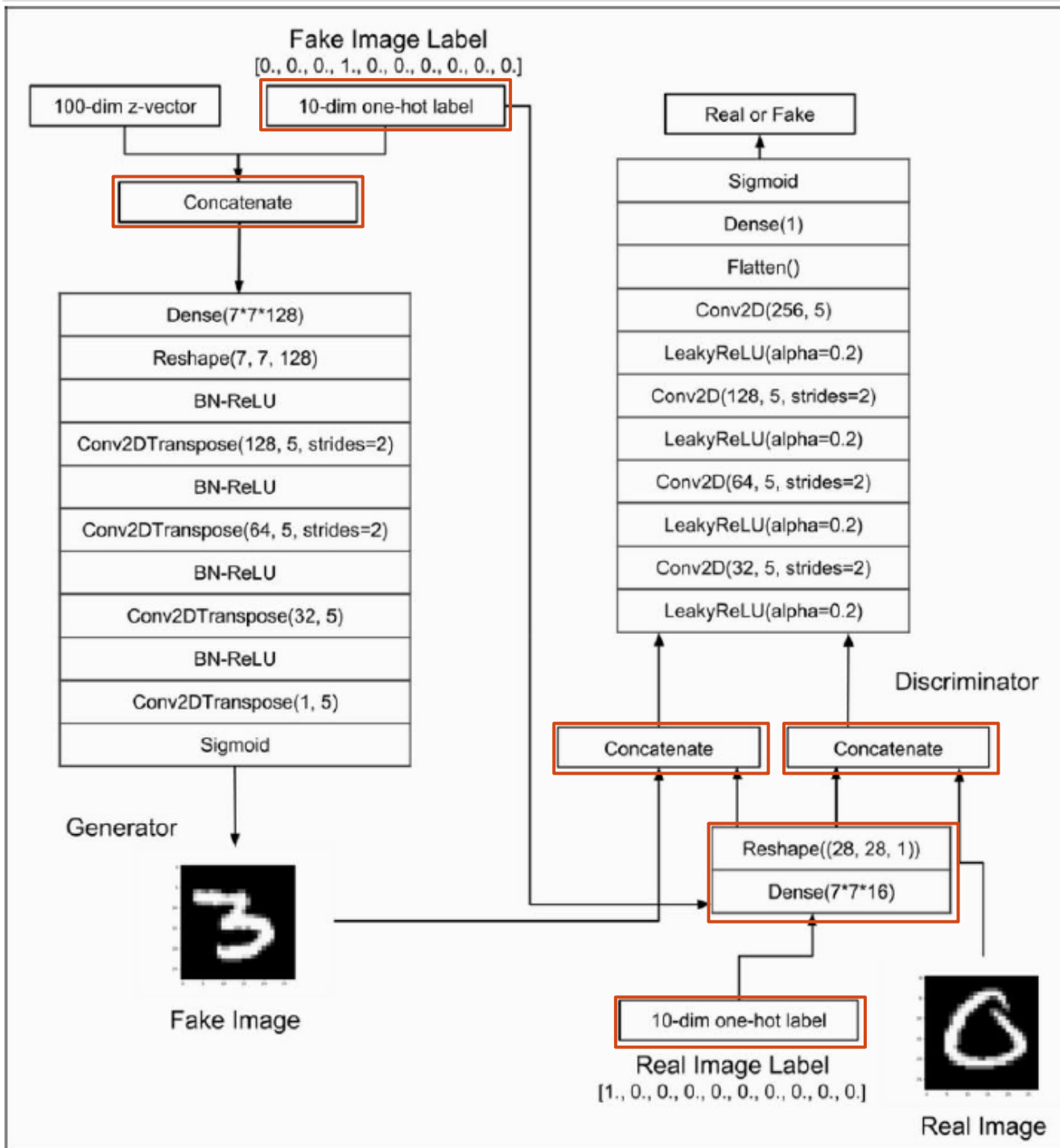


CONDITIONAL GAN



CONDITIONAL GAN (CGAN)

- Nos fausses images avec DCGAN étaient générées aléatoirement
 - Aucun contrôle sur les chiffres spécifiques produits
- Ce problème peut être résolu par les Conditional GAN (CGAN)
 - On impose une condition aux entrées du générateur et du discriminateur
 - La condition se présente sous la forme d'une version vectorielle unique du chiffre



CONDITIONAL GAN (CGAN)

CGAN est similaire à DCGAN, à l'exception du one-hot supplémentaire à l'entrée

Pour le générateur, l'étiquette one-hot est concaténée avec le vecteur latent avant la couche dense

Pour le discriminateur, une nouvelle couche dense est ajoutée

EN GROS...

- Le générateur apprend à générer de fausses images d'un vecteur d'entrées et du chiffre spécifique
- Le discriminateur est toujours un classeur binaire, mais se base sur l'image concaténée à l'étiquette du vrai chiffre
- Les équations de perte sont maintenant:

$$\begin{aligned}\mathcal{L}^{(D)}(\theta^{(G)}, \theta^{(D)}) &= -\mathbb{E}_{x \sim p_{data}} \log \mathcal{D}(x | y) - \mathbb{E}_z \log (1 - \mathcal{D}(\mathcal{G}(z | y') | y')) \\ \mathcal{L}^{(G)}(\theta^{(G)}, \theta^{(D)}) &= -\mathbb{E}_z \log D(\mathcal{G}(z | y'))\end{aligned}$$

- Voir code brièvement:
 - `build_generator`
 - `build_discriminator`



500



5000



10000



15000



20000



25000



30000



35000



40000

CGAN

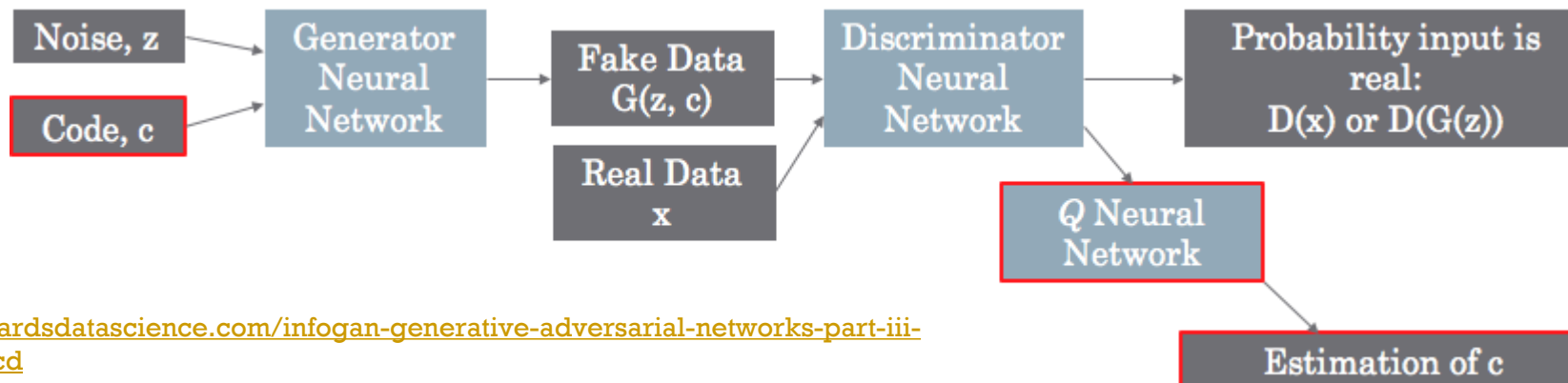


AUTRES MODÈLES DE GAN



INFORMATION MAXIMIZING GENERATIVE ADVERSARIAL NETWORK (INFOGAN)

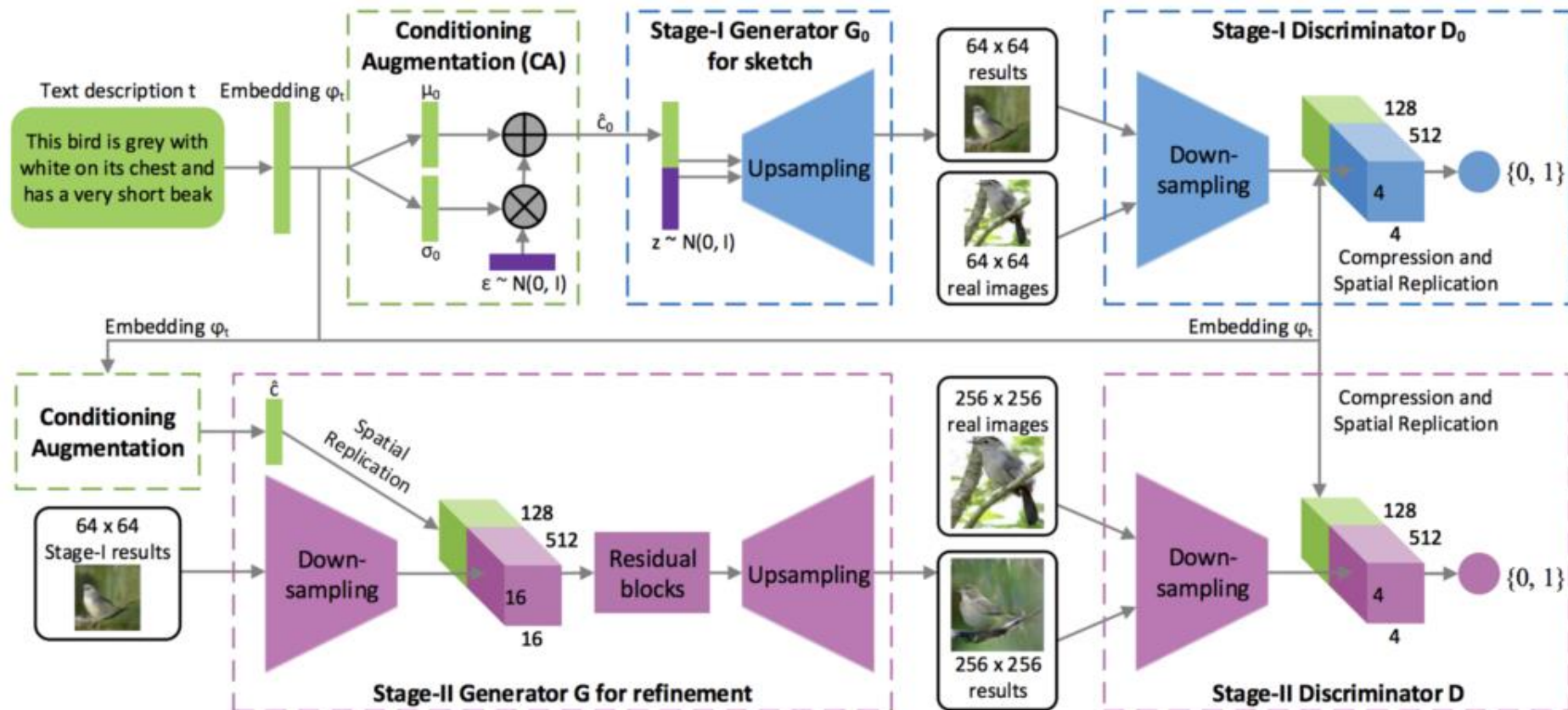
- Un InfoGAN est une extension qui tente de structurer l'entrée ou l'espace latent pour le générateur
- Plus précisément, l'objectif est d'ajouter une signification sémantique spécifique aux variables dans l'espace latent
- Plutôt que d'utiliser un seul vecteur de bruit non-structuré:
 - (1) z traité comme source de bruit incompressible
 - (2) c , code latent qui cible les caractéristiques sémantiques structurées de la distribution






*<https://towardsdatascience.com/infogan-generative-adversarial-networks-part-iii-380c0c6712cd>

STACKGAN - 2017

- StackGAN est une extension du GAN pour générer des images à partir de textes à l'aide d'une pile hiérarchique de modèles CGAN

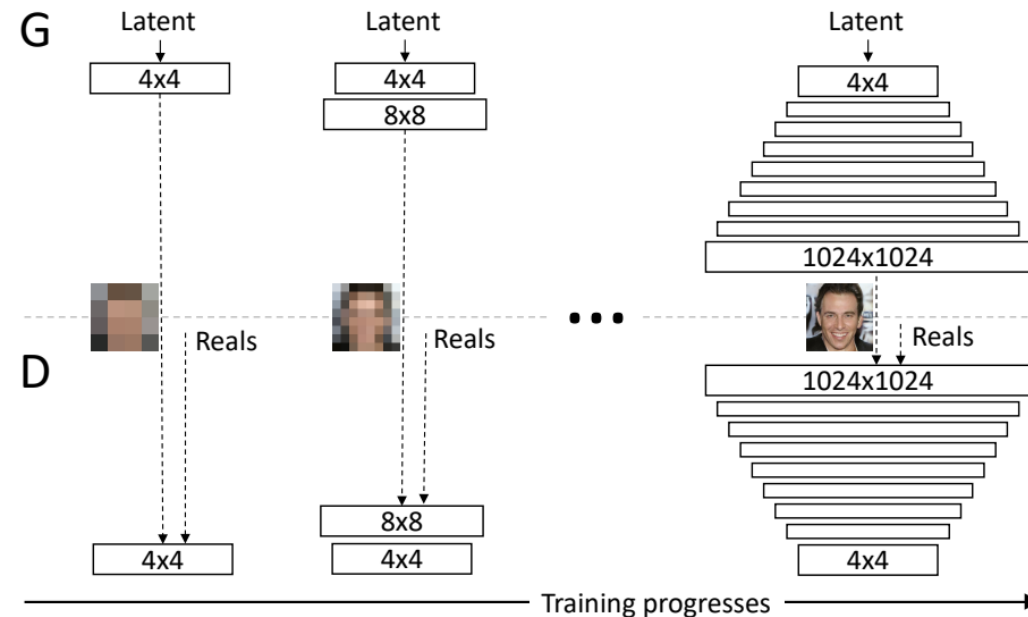


STACKGAN - 2017

Text description	This flower has a lot of small purple petals in a dome-like configuration	This flower is pink, white, and yellow in color, and has petals that are striped	This flower has petals that are dark pink with white edges and pink stamen	This flower is white and yellow in color, with petals that are wavy and smooth	A picture of a very clean living room	A group of people on skis stand in the snow	Eggs fruit candy nuts and meat served on white dish	A street sign on a stoplight pole in the middle of a day
64x64 iAN-INT-CLS								
256x256 StackGAN								

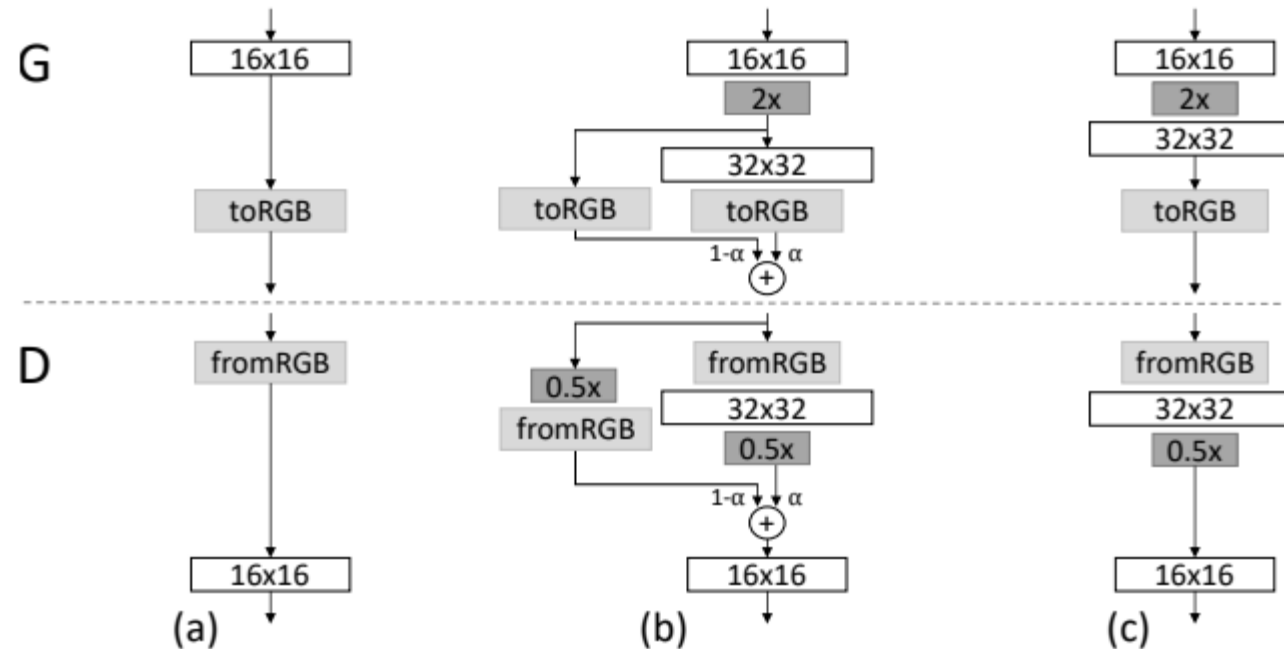
PROGRESSIVE GROWING GAN - 2017

- On augmente progressivement la profondeur du modèle durant l'entraînement
- On garde le générateur et le discriminateur progressif!!!
- Toutes les couches sont entraînables durant tout le processus



PROGRESSIVE GROWING GAN

- La transition d'une résolution à une autre est traitée comme un bloc résiduel



STYLEGAN - 2018



STYLEGAN - 2018

- Une des clés par rapport au GAN est l'utilisation de AdaIN:

$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

- Chaque feature map est normalisée séparément, mise à l'échelle et biaisée en fonction du style
- Ensuite, on fait progresser la taille de l'image successivement!

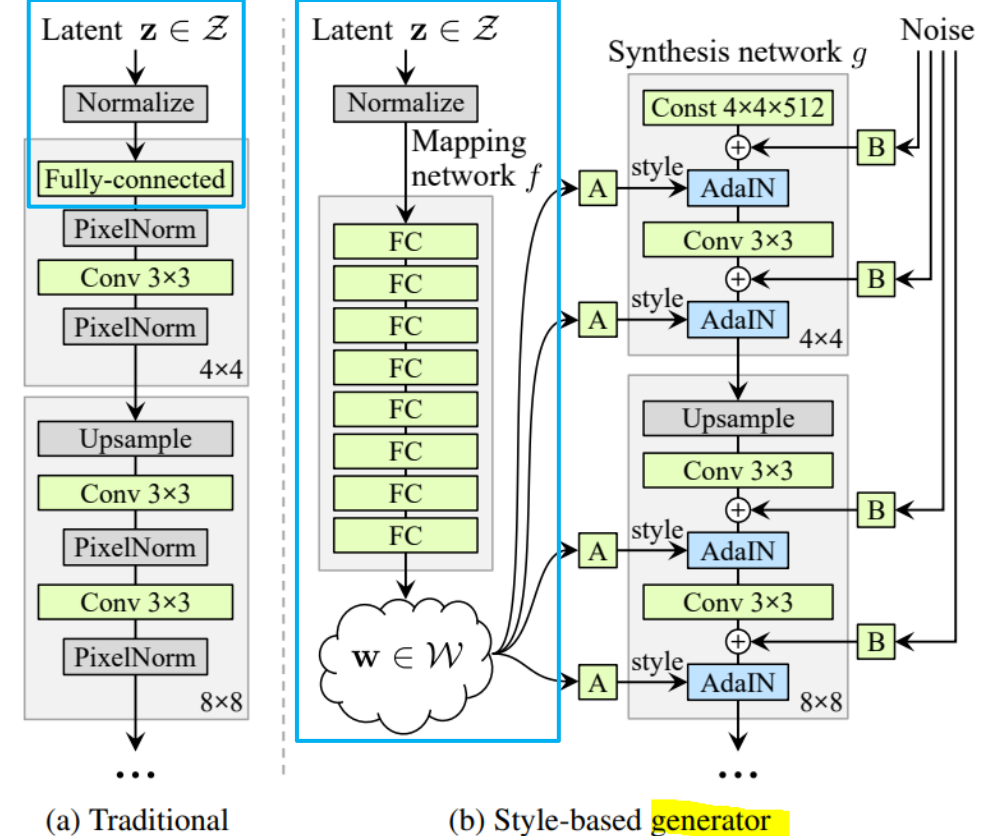
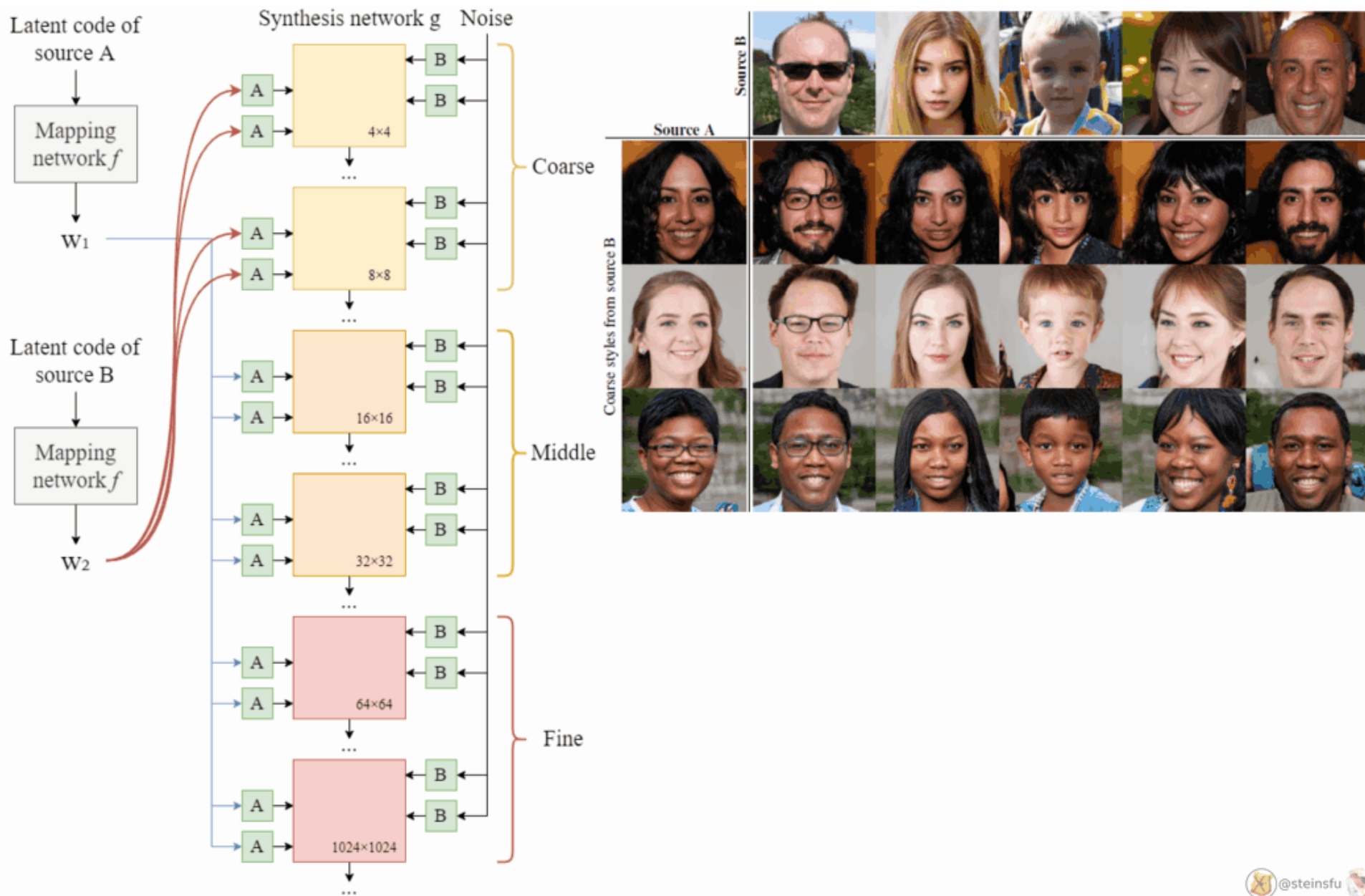


Figure 1. While a traditional generator [30] feeds the latent code through the input layer only, we first map the input to an intermediate latent space \mathcal{W} , which then controls the generator through **adaptive instance normalization** (AdaIN) at each convolution layer. Gaussian noise is added after each convolution, before evaluating the nonlinearity. Here “A” stands for a learned affine transform, and “B” applies learned per-channel scaling factors to the noise input. The mapping network f consists of 8 layers and the synthesis network g consists of 18 layers — two for each resolution ($4^2 - 1024^2$). **The output of the last layer is converted to RGB using a separate 1×1 convolution, similar to Karras et al. [30].** Our generator has a total of 26.2M trainable parameters, compared to 23.1M in the traditional generator.

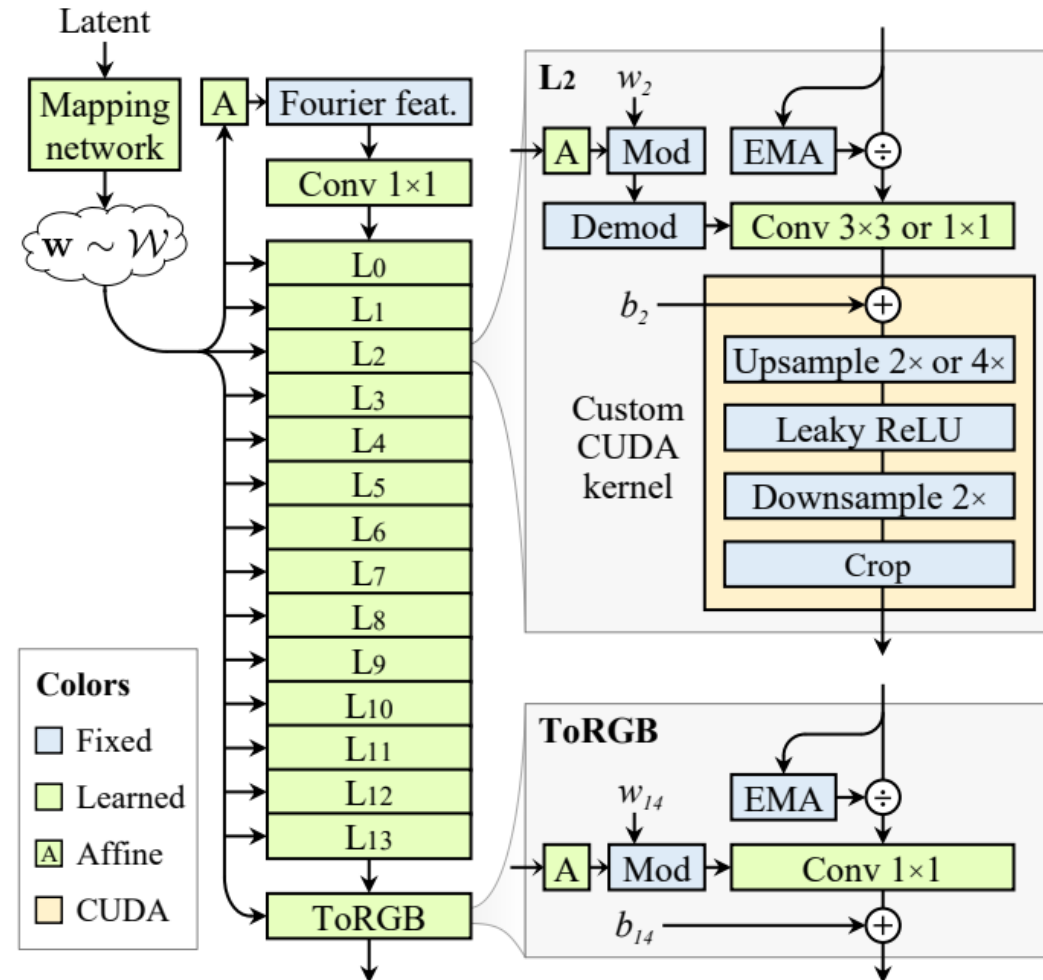
STYLEGAN - 2018

- L'entraînement d'un StyleGAN se fait taille par taille
 - D'abord, on entraîne le générateur et le discriminateur avec des tailles d'images 4x4 jusqu'à stabilité
 - Ensuite on ajoute un bloc au deux pour doubler la taille et on entraîne de nouveau
 - Etc.
- On utilise aussi le *bilinear upsampling* plutôt que le *nearest neighbor* que nous avons vu précédemment
- <https://github.com/NVlabs/stylegan>
- <https://arxiv.org/pdf/1812.04948.pdf>



STYLEGAN 3 -2021

- Les versions subséquentes de StyleGAN visaient chacune à corriger des problèmes spécifiques
- La documentation est assez complexe, mais exhaustive (avec vidéos):
 - <https://nvlabs.github.io/stylegan2/versions.html>
- Dans la V3, on vise à retirer le crénelage



b) Our alias-free StyleGAN3 generator architecture

D'AUTRES MODÈLES

- Least Squares GAN (LSGAN)
 - Cycle-Consistent GAN (CycleGAN)
 - Big GAN
 - Self-Attention GAN (SAGAN)
 - Siamese GAN (SiGAN)
 - Etc.
-
- <https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/>

POUR ALLER PLUS LOIN...

- Bon article sur les GAN: <https://arxiv.org/pdf/1801.09195.pdf>
- ThisPersonDoesNotExist: <https://this-person-does-not-exist.com/en>
- <https://nvlabs.github.io/stylegan2/versions.html>
- Image Inpainting: <https://news.developer.nvidia.com/new-ai-imaging-technique-reconstructs-photos-with-realistic-results/?ncid=nv-twi-37107>
- AI artists: <https://aiartists.org/ai-generated-art-tools>
- Applications des GAN: <https://github.com/nashory/gans-awesome-applications>



EXEMPLE COMPLET



GÉNÉRATION DE TEXTURES

- Avec un GAN, il est possible de générer des textures pour les jeux vidéo
- Encore une fois, le défi ici est d'avoir un bon ensemble d'entraînement
- Regardons comment nous pourrions y parvenir en utilisant l'ensemble de données Cifar100:
- Nous utiliserons un WGAN `from keras.datasets import`
`cifar100`
 - Un WGAN utilise la distance de Wasserstein plutôt que la divergence de Kullback–Leibler ou celle de Jensen–Shannon
 - Sans entrer dans les détails, cette distance est considérée comme meilleure pour l'entraînement des GAN

WASSERSTEIN GAN (WGAN)

- Basé sur les f-GAN

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size. n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator's parameters.

```
1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w \left[ \frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)})) \right]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSPProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSPProp}(\theta, g_\theta)$ 
12: end while
```

```

class WGAN():
    def __init__(self):
        self.img_rows = 32
        self.img_cols = 32
        self.channels = 3
        self.img_shape = (self.img_rows, self.img_cols, self.channels)
        self.latent_dim = 100

        # Following parameter and optimizer set as recommended in paper
        self.n_critic = 5
        self.clip_value = 0.01
        optimizer = RMSprop(lr=0.00005)

        # Build and compile the critic
        self.critic = self.build_critic()
        self.critic.compile(loss=self.wasserstein_loss,
                           optimizer=optimizer,
                           metrics=['accuracy'])

        # Build the generator
        self.generator = self.build_generator()

        # The generator takes noise as input and generated imgs
        z = Input(shape=(self.latent_dim,))
        img = self.generator(z)

        # For the combined model we will only train the generator
        self.critic.trainable = False

        # The critic takes generated images as input and determines
        validity
        valid = self.critic(img)

        # The combined model (stacked generator and critic)
        self.combined = Model(z, valid)
        self.combined.compile(loss=self.wasserstein_loss,
                              optimizer=optimizer,
                              metrics=['accuracy'])

```

*code: Lanham M., *Hands-On Deep Learning for Games*, Packt Publishing, 2019.

ÉTAPE 1

Code d'initialisation et de création

-Forme pour l'image

-paramétrage du WGAN

-construction du discriminateur

-construction du générateur

-compilation du WGAN


```
def build_critic(self):
    model = Sequential()
    model.add(Conv2D(16, kernel_size=3,
strides=2, input_shape=self.img_shape,
padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Conv2D(32, kernel_size=3,
strides=2, padding="same"))
    model.add(ZeroPadding2D(padding=((0,
1), (0, 1))))

    model.add(BatchNormalization(momentum=0.8
))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Conv2D(64, kernel_size=3,
strides=2, padding="same"))

    model.add(BatchNormalization(momentum=0.8
))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Conv2D(128, kernel_size=3,
strides=1, padding="same"))

    model.add(BatchNormalization(momentum=0.8
))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(1))
    model.summary()

    img = Input(shape=self.img_shape)
    validity = model(img)
    return Model(img, validity)
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 16, 16, 16)	448
leaky_re_lu (LeakyReLU)	(None, 16, 16, 16)	0
dropout (Dropout)	(None, 16, 16, 16)	0
conv2d_1 (Conv2D)	(None, 8, 8, 32)	4640
zero_padding2d (ZeroPadding2D)	(None, 9, 9, 32)	0
batch_normalization (Batch Normalization)	(None, 9, 9, 32)	128
leaky_re_lu_1 (LeakyReLU)	(None, 9, 9, 32)	0
dropout_1 (Dropout)	(None, 9, 9, 32)	0
conv2d_2 (Conv2D)	(None, 5, 5, 64)	18496
batch_normalization_1 (Batch Normalization)	(None, 5, 5, 64)	256
leaky_re_lu_2 (LeakyReLU)	(None, 5, 5, 64)	0
dropout_2 (Dropout)	(None, 5, 5, 64)	0
conv2d_3 (Conv2D)	(None, 5, 5, 128)	73856
batch_normalization_2 (Batch Normalization)	(None, 5, 5, 128)	512
leaky_re_lu_3 (LeakyReLU)	(None, 5, 5, 128)	0
dropout_3 (Dropout)	(None, 5, 5, 128)	0
flatten (Flatten)	(None, 3200)	0
dense (Dense)	(None, 1)	3201
Total params: 101,537		
Trainable params: 101,089		
Non-trainable params: 448		

ÉTAPE 2

Création du discriminateur

*code: Lanham M., *Hands-On Deep Learning for Games*, Packt Publishing, 2019.

```
def build_generator(self):
    model = Sequential()
    model.add(Dense(128 * 8 * 8,
activation="relu",
input_dim=self.latent_dim))
    model.add(Reshape((8, 8, 128)))
    model.add(UpSampling2D())
    model.add(Conv2D(128, kernel_size=4,
padding="same"))

    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))
    model.add(UpSampling2D())
    model.add(Conv2D(64, kernel_size=4,
padding="same"))

    model.add(BatchNormalization(momentum=0.8))
    model.add(Activation("relu"))
    model.add(Conv2D(self.channels,
kernel_size=4, padding="same"))
    model.add(Activation("tanh"))
    model.summary()

    noise = Input(shape=(self.latent_dim,))
    img = model(noise)
    return Model(noise, img)
```

```
Model: "sequential_1"

Layer (type)                 Output Shape              Param #
=====
dense_1 (Dense)              (None, 8192)              827392
reshape (Reshape)            (None, 8, 8, 128)         0
up_sampling2d (UpSampling2D) (None, 16, 16, 128)       0
conv2d_4 (Conv2D)            (None, 16, 16, 128)       262272
batch_normalization_3 (Batch (None, 16, 16, 128)       512
activation (Activation)      (None, 16, 16, 128)       0
up_sampling2d_1 (UpSampling2 (None, 32, 32, 128)       0
conv2d_5 (Conv2D)            (None, 32, 32, 64)        131136
batch_normalization_4 (Batch (None, 32, 32, 64)        256
activation_1 (Activation)    (None, 32, 32, 64)        0
conv2d_6 (Conv2D)            (None, 32, 32, 3)         3075
activation_2 (Activation)    (None, 32, 32, 3)         0
=====
Total params: 1,224,643
Trainable params: 1,224,259
Non-trainable params: 384
```

ÉTAPE 3

Création du générateur

*code: Lanham M., *Hands-On Deep Learning for Games*, Packt Publishing, 2019.

```

def train(self, epochs, batch_size=128, sample_interval=50):
    (X_train, y), (_, _) = cifar100.load_data(label_mode='fine')
    Z_train = []
    cnt = 0
    for i in range(0, len(y)):
        if y[i] == 33:
            cnt = cnt + 1
            z = X_train[i]
            Z_train.append(z)

    Z_train = np.reshape(Z_train, [500, 32, 32, 3])
    Z_train = (Z_train.astype(np.float32) - 127.5) / 127.5

    valid = -np.ones((batch_size, 1))
    fake = np.ones((batch_size, 1))

    for epoch in range(epochs):
        for _ in range(self.n_critic):
            idx = np.random.randint(0, Z_train.shape[0], batch_size)
            imgs = Z_train[idx]
            if epoch % sample_interval == 0:
                self.save_images(imgs, epoch)

            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            gen_imgs = self.generator.predict(noise)

            d_loss_real = self.critic.train_on_batch(imgs, valid)
            d_loss_fake = self.critic.train_on_batch(gen_imgs, fake)
            d_loss = 0.5 * np.add(d_loss_fake, d_loss_real)

            for l in self.critic.layers:
                weights = l.get_weights()
                weights = [np.clip(w, -self.clip_value, self.clip_value) for w
in weights]
                l.set_weights(weights)

            g_loss = self.combined.train_on_batch(noise, valid)

```

ÉTAPE 4

Entraînement du réseau

*code: Lanham M., *Hands-On Deep Learning for Games*, Packt Publishing, 2019.

RÉSULTATS

- Exemples de textures de type forêts
- Vous pouvez facilement générer différents types de textures à l'aide de divers paramètres
- Vous pouvez les utiliser comme textures ou champs de hauteur dans Unity ou un autre moteur de jeu

