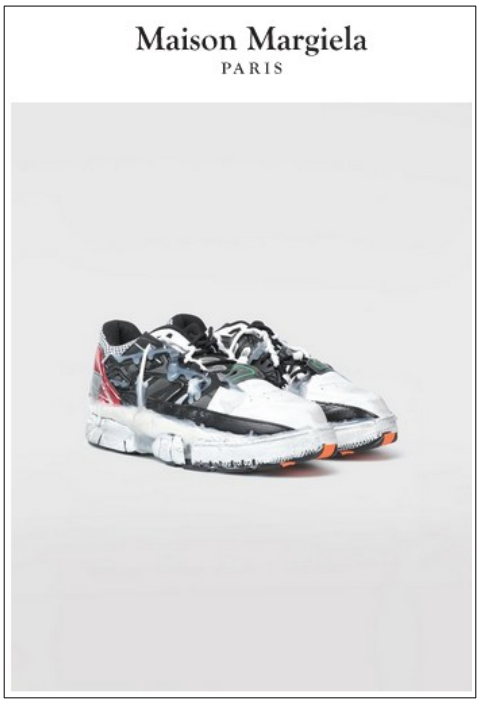
Génération d’images

Deep Learning

StyleGAN2 ADA en PyTorch



Source

20211115 Catherine LE

Sommaire

Rapport est riche en illustrations pour la moitié de son contenu.

Thématique

Notre client, célèbre enseigne de prêt à porter haut de gamme, cherche à renouveler sa collection de baskets tous les six mois. Il utilise pour cela un algorithme de génération d'images appelé Deep Convolutional Generative Adversarial Network dit DCGAN.

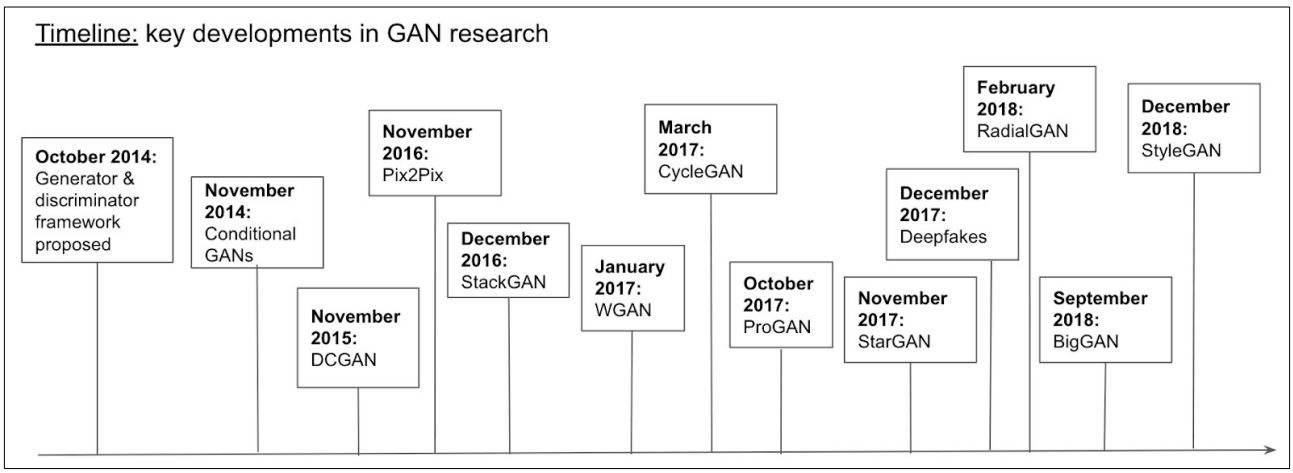
Il souhaite améliorer la méthode utilisée en production et adopter une méthode plus récente générant des images de meilleure qualité.

Dans ce projet, la thématique choisie sera la Génération d'Images par des algorithmes de Deep Learning. Les algorithmes génératives d'images font appel à des « réseaux de neurones antagonistes génératifs » appelés Generative Adversarial Network ou GAN [Goodfellow et al. 2014].

La baseline s'appuiera sur un des modèles GAN, le DCGAN qui se compose principalement de couches de convolution. Suite aux premiers tests sur des images de baskets, les résultats du DCGAN sont satisfaisants d'autant plus qu'il est possible de personnaliser ce modèle en fonction de la taille de l'image souhaitée. Cependant, les détails de l'image ne sont pas nets sur les contours de l'objet et flous dans son ensemble.

Depuis 2014, de nombreux modèles sont apparus, cf. figure ci-dessous. Le dernier modèle, le StyleGAN s'est développé avec l'arrivé du StyleGAN2 , du StyleGAN2 - ADA [7 octobre 2020] et du tout dernier StyleGAN3 [18 octobre 2021].

Dans cette étude, la baseline DCGAN sera comparée avec le StyleGAN2 – ADA pour Adapative Discriminator Augmentation, offrant une qualité comparable au styleGAN3 mais sans l'interface de manipulation et autres options développées récemment.



Source Image

État de l’art

1. DCGAN
==========
Shoe Design using Generative Advesarial Networks
CODE: https://github.com/hminle/shoe-design-using-generative-adversarial-networks
PAPER: https://github.com/hminle/shoe-design-using-generative-adversarial-networks/blob/master/report.pdf

2. StyleGAN
==========
A Gentle Introduction to StyleGAN the Style Generative Adversarial Network
BLOG: https://machinelearningmastery.com/introduction-to-style-generative-adversarial-network-stylegan/

3. StyleGAN2 - ADA
==========
Training Generative Adversarial Networks with Limited Data
PAPER: https://arxiv.org/pdf/2006.06676.pdf
CODE: https://github.com/NVlabs/stylegan2-ada-pytorch.git
CODE: https://github.com/jeffheaton/present/blob/master/youtube/gan/colab_gan_train.ipynb
https://github.com/jeffheaton/t81_558_deep_learning/blob/master/t81_558_class_07_3_style_gan.ipynb
CODE: https://github.com/jeffheaton/present/blob/master/youtube/gan/colab_gan_train.ipynb

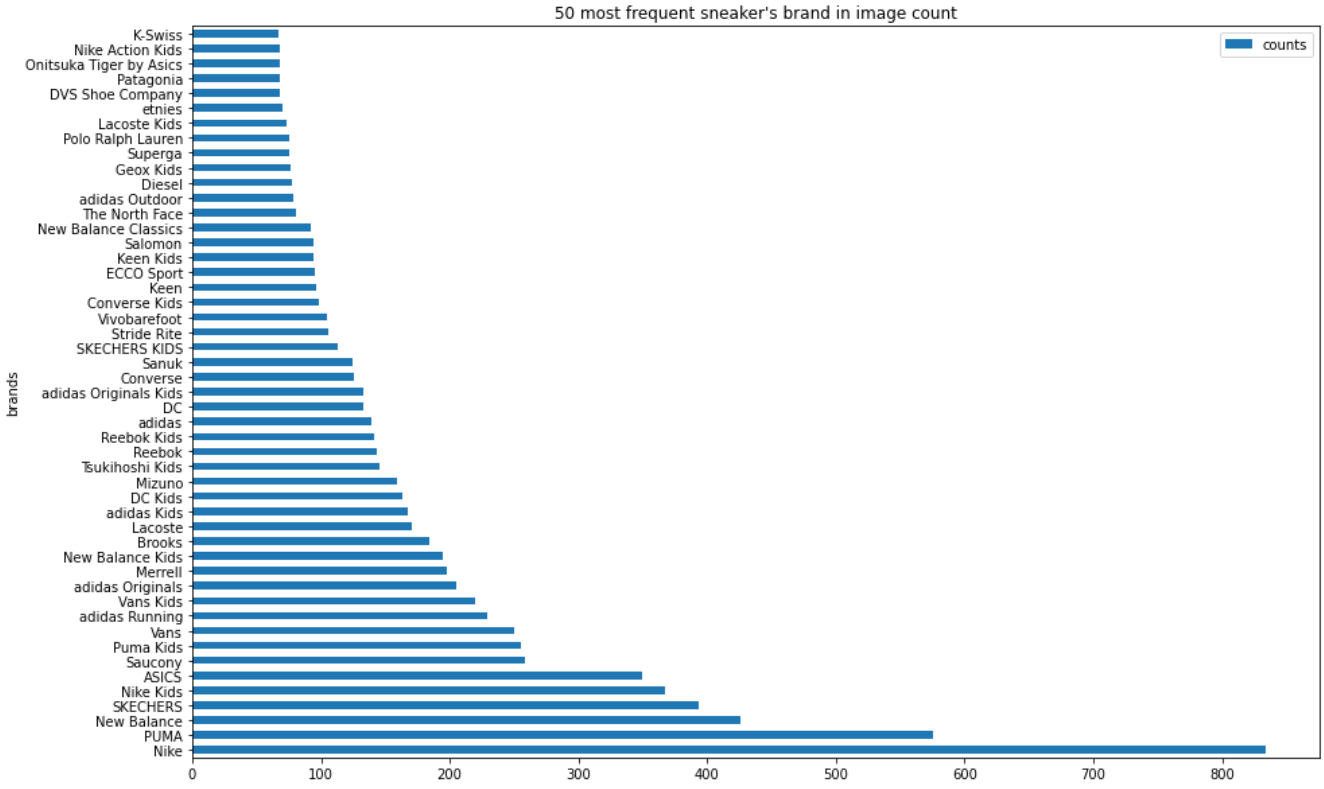
Jeu de données

UT Zappos50k est un ensemble de données d'images de chaussures composé de 50 025 images du catalogue recueillies sur Zappos.com. Les images sont divisées en 4 catégories principales - chaussures, sandales, pantoufles et bottes - suivies par leurs types fonctionnels puis par leurs marques individuelles. Les chaussures sont centrées sur un fond blanc et photographiées dans la même orientation pour faciliter l'analyse.

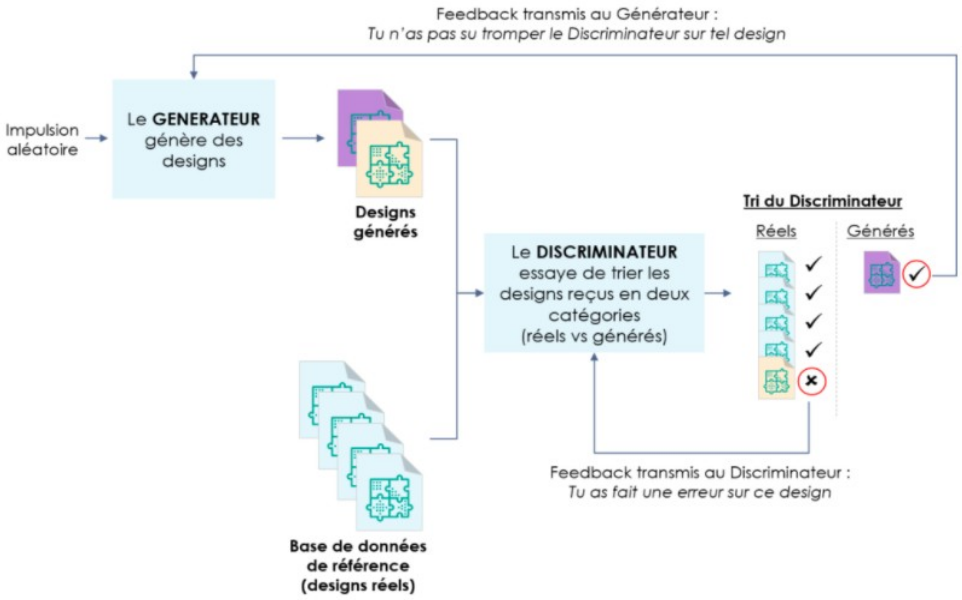
ut-zap50k-images-square
- ├Boots
- ├Sandals
- ├Shoes
  - └Boat Shoes
  - └ ...
  - └Sneakers and Athletic Shoes
- ├Slippers

- Dimensions 136 * 136 pixels
- Poids 3.8 kB
- Format JPEG

La sous-catégorie Sneakers_and_Athletic_Shoes comprend 336 sous-répertoires contenant au total 12 859 images de baskets de format .jpg. Ci-dessous, les 50 sous-répertoires des marques individuelles contenant le plus d'images.



Résumé théorique du GAN



Les algorithmes de Deep Learning générative d'images font appel à des réseaux de neurones antagonistes génératifs appelés Generative Adversarial Network ou GAN [Goodfellow et al. 2014]. Antagonistes car il s'agit d'interaction de deux réseaux neuronaux - le réseau génératif travaille à la création d'une nouvelle image, et le réseau discriminatif travaille à l'évaluation des résultats pour fournit un retour d'information au premier réseau.

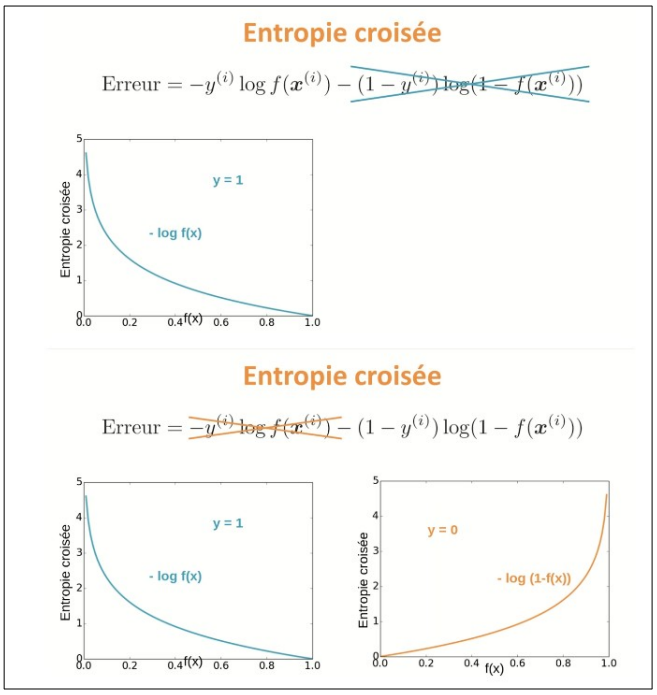
Le Générateur reçoit l'identité des designs sur lesquels il a été démasqué par le Discriminateur ainsi qu'une impulsion aléatoire. Le Discriminateur reçoit l'identité des designs sur lesquels il a été trompé par le Générateur en plus des designs générés et des designs réels.

Lors de l'apprentissage, les deux réseaux sont en positions adverses :

- le discriminateur essaie de déterminer au mieux la provenance des exemples (e.g., réel ou artificiel ?).
- le générateur essaye de tromper le discriminateur.

Source

Les modèles génératifs reproduisent eux-mêmes des données, ils peuvent donc fonctionner avec des pools de données minimaux. L'ensemble de données réelles auquel le Discriminateur a accès peut rester relativement petit, par rapport à d'autres méthodes.



Rappel: Erreur ou l'entropie croisée

Source

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

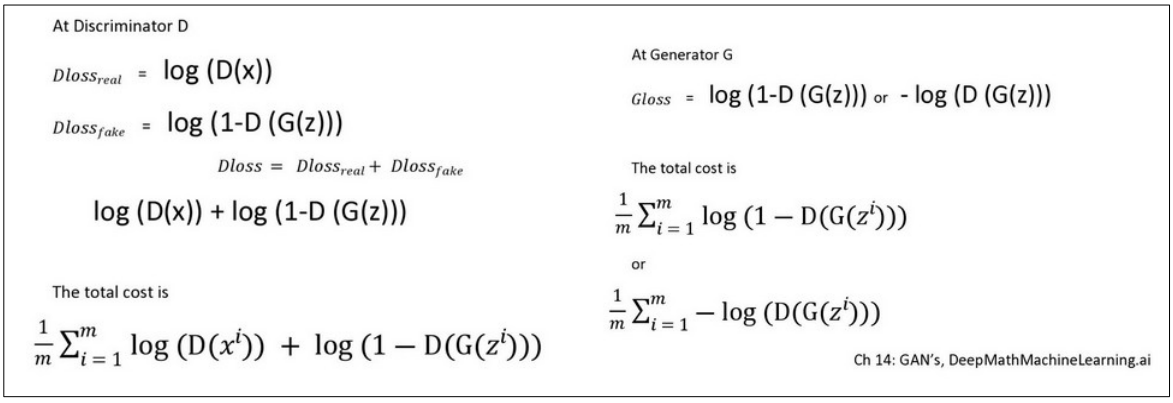
$$\max_D V(D) = \underbrace{\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})]}_{\text{recognize real images better}} + \underbrace{\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]}_{\text{recognize generated images better}}$$

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

Optimize G that can fool the discriminator the most.

- Le Discriminateur D (sortie 0 à 1) change ses poids afin de maximiser V.
- Le Générateur G (sortie image) cherche à confondre le discriminateur D, afin de minimiser V.

C'est donc deux réseaux qui s'optimisent mutuellement rendant les images générées de plus en plus réelles. Cet apprentissage se fait avec alternance entre la montée du gradient pour le discriminateur et la descente du gradient pour le générateur.



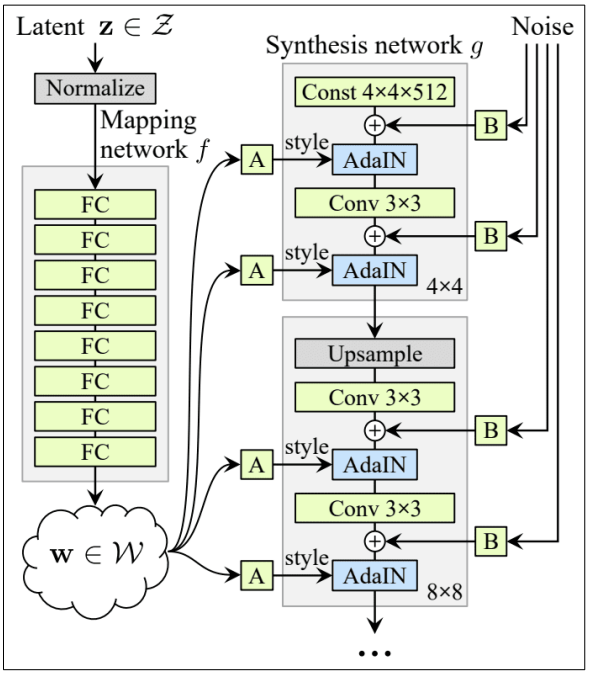
Source

Résumé théorique du StyleGAN

Le générateur StyleGAN ne prend plus un seul point aléatoire en entrée, à la place, deux nouvelles sources d'aléa sont utilisées pour générer une image synthétique :

- un réseau cartographique autonome
- des couches de bruit

Le changement suivant consiste à modifier le modèle du générateur afin qu'il ne prenne plus un point de l'espace latent en entrée mais une valeur constante 4x4x512 en entrée afin de démarrer le processus de synthèse d'image.



Architecture du StyleGAN Source

L'ajout du nouveau réseau de cartographie à l'architecture entraîne également la re-nomination du modèle de générateur sous l'appellation "réseau de synthèse".

On utilise un réseau de cartographie autonome qui prend en entrée un point échantillonné de manière aléatoire puis génère un vecteur de style. Le réseau de mappage qui est composé de huit couches entièrement connectées. Le vecteur de style est ensuite transformé par l'opération A et incorporé dans chaque bloc du modèle générateur puis passe par les couches convolutives via une opération appelée normalisation adaptative d'instance ou AdaIN. Les couches AdaIN consistent d'abord à normaliser la sortie de chaque carte de caractéristiques dites « features map » par une gaussienne standard, puis à ajouter le vecteur de style comme biais.

La sortie de chaque couche de convolution est un bloc de cartes d'activation auquel un bruit gaussien est ajouté à chacune de ces cartes d'activation juste avant les opérations AdaIN. Un échantillon différent de bruit est généré pour chaque bloc.

Utilisation du vecteur de style

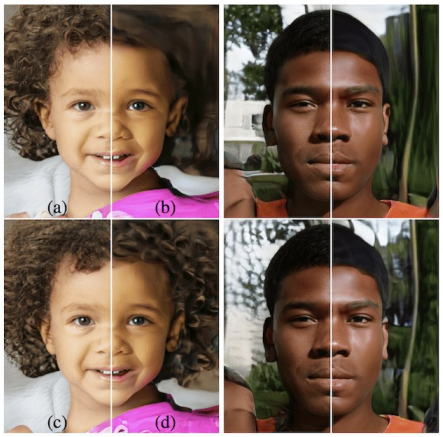
L'utilisation de différents vecteurs de style en différents points du réseau de synthèse permet de contrôler les styles de l'image résultante à différents niveaux de détail. Par exemple, les blocs des couches du réseau de synthèse de résolutions inférieures (par exemple 4×4 et 8×8) contrôlent les styles de bas niveau tels que la pose et la coiffure. Les blocs des couches du réseau de synthèse de résolutions moyennes (par exemple en 16×16 et 32×32) contrôlent les style de niveaux moyen comme la coiffure et les expressions faciales. Enfin, des blocs des couches u réseau de synthèse plus proches de l'extrémité de la sortie du réseau (par exemple 64×64 à 1024×1024) contrôlent les styles de niveaux plus fins comme les schémas de couleurs et les détails très fins.

Ensemble de visages générés en adoptant le style d'un autre ensemble de visages générés Source

L'image générée de la source B aura un style de bas niveau tel que la pose, la coiffure générale, la forme du visage et les lunettes, tandis que toutes les couleurs (yeux, cheveux, éclairage) et les caractéristiques plus fines du visage correspondent à la source

Utilisation du vecteur du bruit

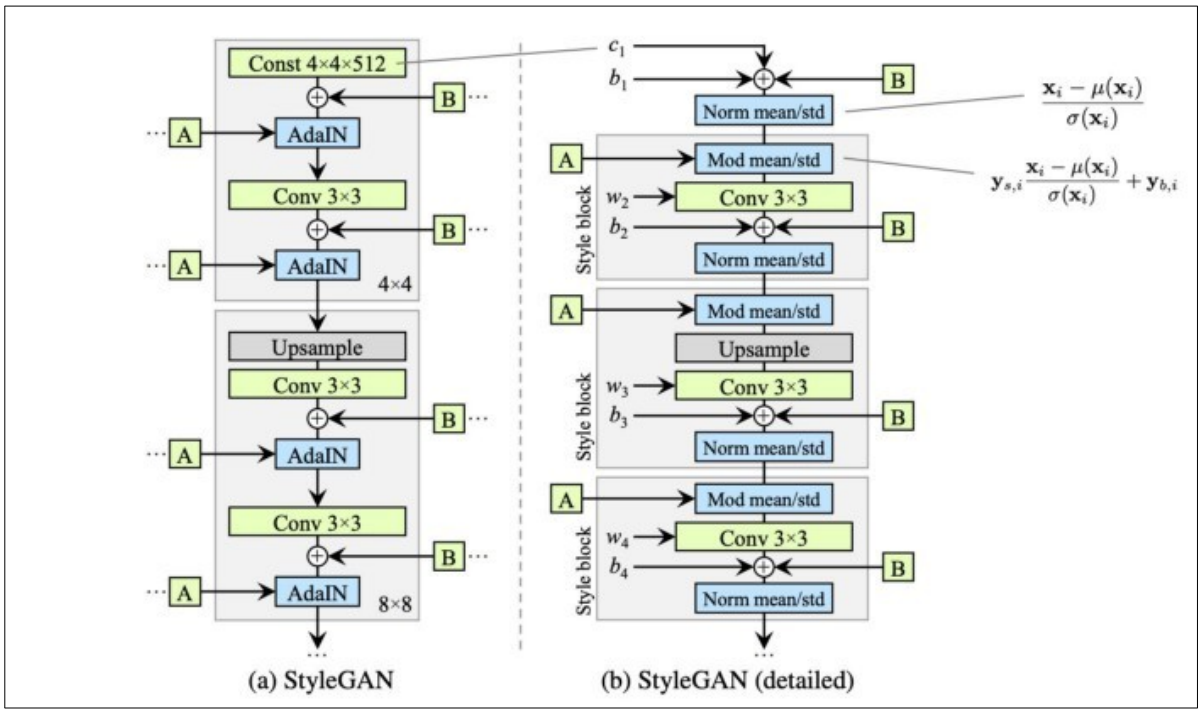
Le bruit permet de contrôler la génération de détails d'une structure plus large lorsque le bruit est utilisé sur les blocs de couches basses, à la génération de détails plus fins lorsque le bruit est ajouté aux couches plus proches de la sortie du réseau.



Variation du bruit à différents niveaux du générateur Source

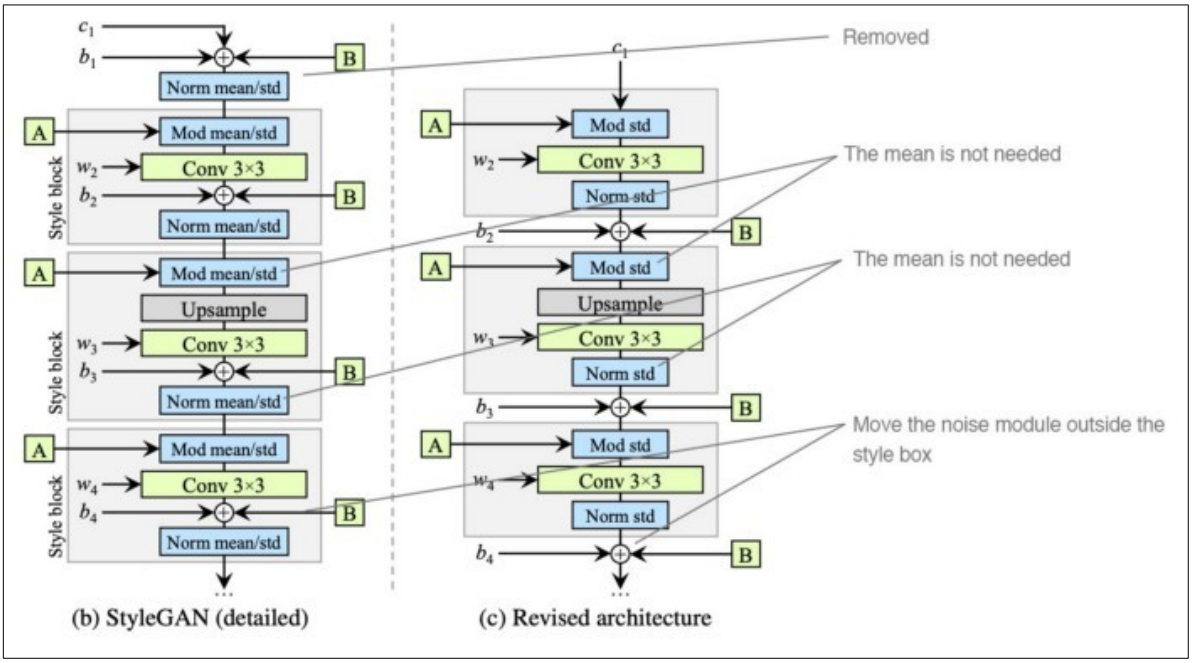
Ainsi, le StyleGAN est efficace à la fois pour contrôler le style des images générées et créer de grandes images uniques de haute qualité.

Résumé théorique du StyleGAN2



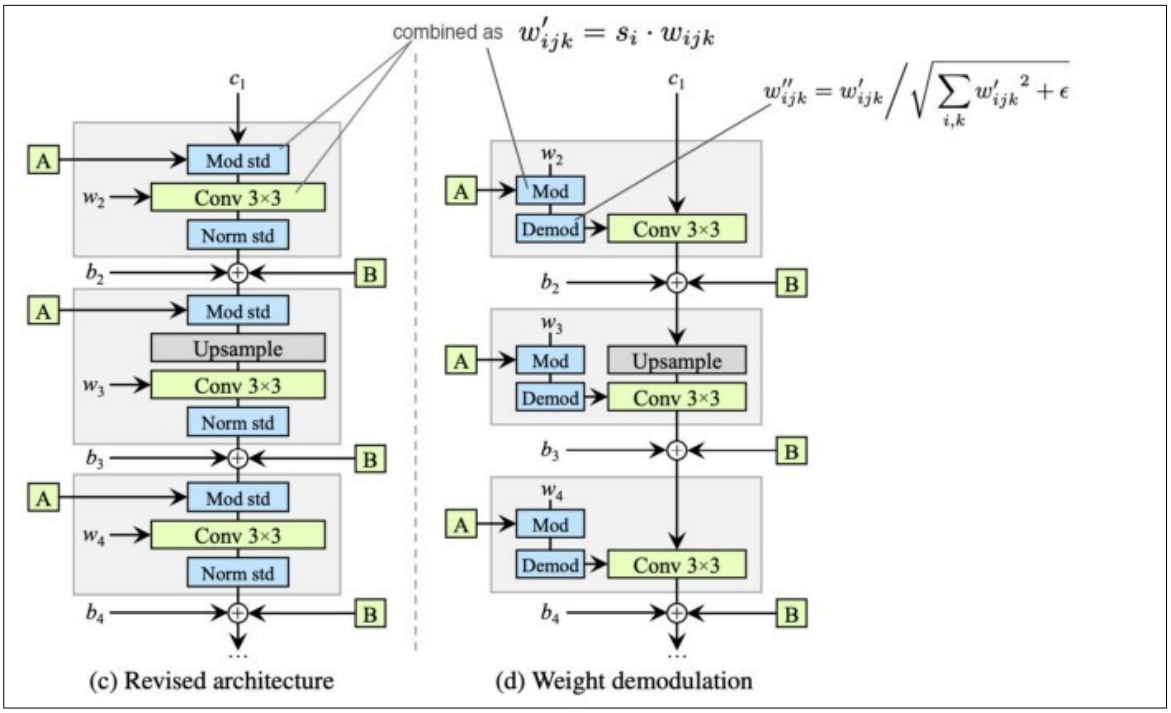
Rappel: Architecture du StyleGAN détaillé

avec la couche AdaIN splité en deux couches Source



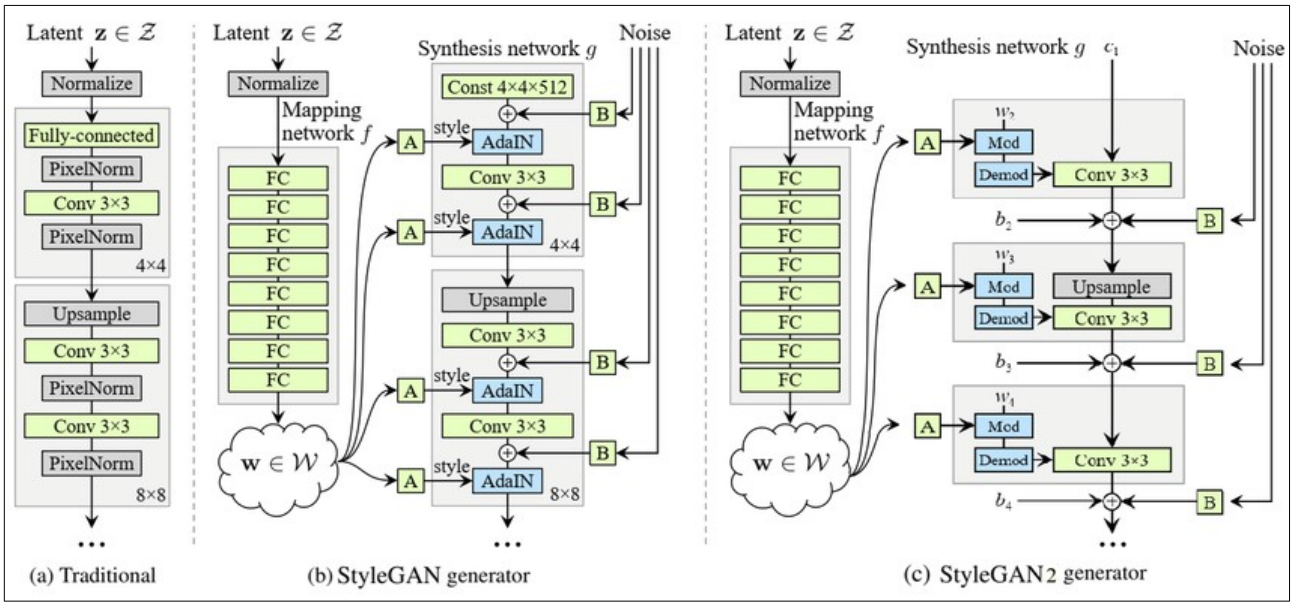
De styleGAN à StyleGAN2 Source

- Supprimez (simplifiez) la façon dont la constante 4x4x512 est traitée au début.
- La moyenne n'est pas nécessaire pour normaliser les features.
- Déplacer le bruit en dehors du module de style.



Source

Le diagramme de droite ci-dessus est la nouvelle conception avec la démodulation des poids, avec *w_ijk* les poids de la convolution. C'est semblable à la normalisation d'écart-type 1 des features map. $Mod = w'_{ijk} = s_i \times w_{ijf}$



Synthèse des architectures
GAN, StyleGAN et StyleGAN2 Source

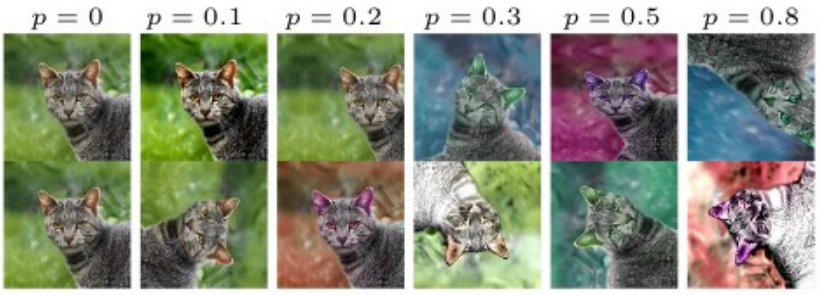
Résumé théorique du StyleGAN2 – ADA

StyleGAN2 – ADA [NVIDIA Octobre 2020]

Pour éviter l'overfitting du discriminateur, la transformation aléatoire de l'image durant les phases d'entraînement consiste à multiplier la quantité de données d'entraînement par la rotation. l'ajout de bruit, le changement de couleurs etc. pour modifier l'image et créer une version unique de celle-ci. Ils appliquent cette transformation à toutes les images présentées au Discriminateur.

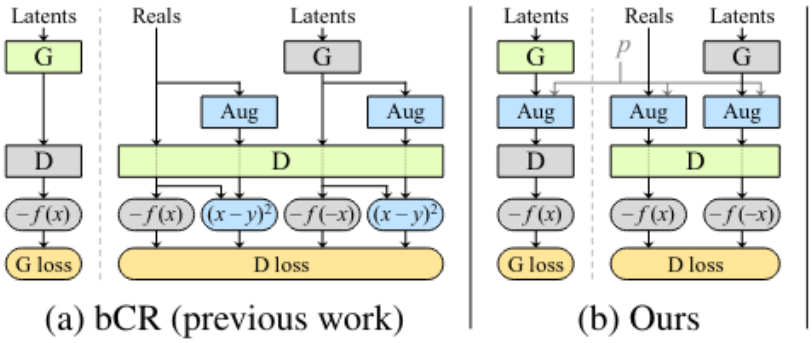
Malheureusement, cela ne peut pas être appliqué facilement à une architecture GAN puisque le Générateur apprendra à générer des images suivant ces mêmes transformations. Ils ont trouvé un moyen à ce qu'aucune de ces images transformées ne se retrouve dans les images générées. Ce phénomène est appelé « Augmentation Leaking ».

Pour se faire, le modèle fixe le paramètre *p,* la probabilité d'augmentation compris entre [0,1] mais restant inférieur à 0,8. Si l'augmentation des données est trop forte, le Générateur ne sera plus capable de lire la distribution des images réelles conduisant au phénomène du «Leak Augmentation» d'où l'image de couleur néon au sein des images générées comme le montre l'image ci-dessous.



(c) Effect of augmentation probability p

Architecture du StyleGAN2 – ADA



bCR - Balanced Consistency Regularizationet versus

DA (DCGAN2 ADA) Source

- Le Discriminateur reçoit les images augmentées provenant du Discriminateur lui-même et du Générateur.

- Le Discriminateur ne connaît pas la distribution des images réelles, n'étant pas directement lié aux images réelles. Donc il doit «découvrir» leurs distributions en passant par la phase d'augmentation des données.

- Précédent modèle: le Discriminateur connaît la distribution des images réelles.

- L'augmentation des données est paramétré par la probabilité d'augmentation *p*.

- Le second terme du D Loss disparaît.

Grâce à la phase DA, il est possible d'obtenir de bonnes performances FID - Fréchet inception distance - dès 1000 images sans être confronté à l'overfitting.

Dataset		Baseline	ADA	+ bCR
FFHQ	1k	100.16	**21.29**	22.61
	5k	49.68	10.96	**10.58**
	10k	30.74	8.13	**7.53**
	30k	12.31	5.46	**4.57**
	70k	5.28	4.30	**3.91**
	140k	3.71	3.81	**3.62**
LSUN CAT	1k	186.91	43.25	**38.82**
	5k	96.44	16.95	**16.80**
	10k	50.66	13.13	**12.90**
	30k	15.90	10.50	**9.68**
	100k	**8.56**	9.26	8.73
	200k	**7.98**	9.22	9.03

Impact du nombre d'images

sur le score FID du modèle ADA Source

Implémentation et performances du StyleGAN2 ADA

Environnement

- Linux ou Windows
- 1 à 8 GPU NVIDIA avec au moins 12 Go de mémoire.
- Python 3.7
- PyTorch 1.7.1 soit torch==1.7.1
- CUDA toolkit 11.0 ou plus récent. I> version 11.5.
- pip install : click, requests, tqdm, pyspng, ninja, imageio-ffmpeg==0.4.3

Data processing

- Redimensionnement : de 136x136 à 256x256
- Traitement et reclassement des images à l'aide du fichier dataset_tool.py
- Data augmentation ADA

```
Number of GPUs:     1
Number of images:   12856
Image resolution:   256

Image shape: [3, 256, 256]
```

Hyper paramètres pour la Data Augmentation

```
 "augment_kwargs": {
    "class_name": "training.augment.AugmentPipe",
    "xflip": 1,
    "rotate90": 1,
    "xint": 1,
    "scale": 1,
    "rotate": 1,
    "aniso": 1,
    "xfrac": 1,
    "brightness": 1,
    "contrast": 1,
    "lumaflip": 1,
    "hue": 1,
    "saturation": 1
  }
```

Hyper paramètres pour l'Optimizer

```
"G_opt_kwargs": {
    "class_name": "torch.optim.Adam",
    "lr": 0.0025,
    "betas": [0,0.99],
    "eps": 1e-08
  }
```

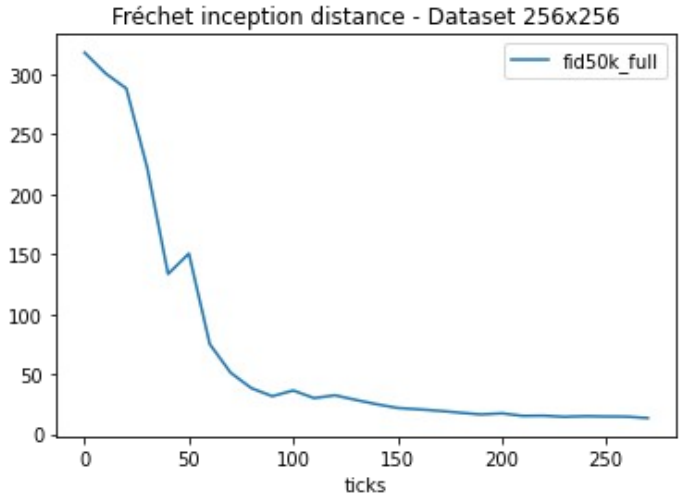
Performance: Recall et Precision


```
{"results": {
"pr50k3_full_precision": 0.46748000383377075,
"pr50k3_full_recall": 0.14141257107257843},
}
```


Performance: Fréchet inception distance

The FID or Fréchet inception distance `fid50k_full` mesure la distance entre les distributions des images générées et réelles. Plus cette cette mesure est faible, plus les images générées ressemblent aux images réelles.


```
{"results": {"fid50k_full": 11.77387331176866}, "metric": "fid50k_full",
```

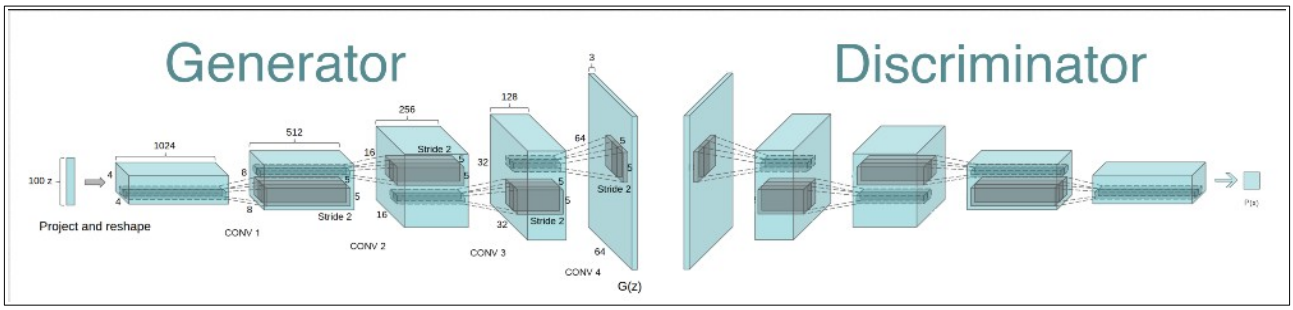
Résultats après 270 epochs

pour une durée de 23h 31m 50s

Image réelle

Image générée

Implémentation et performances du DCGAN



Source

```
_netG(
  (main): Sequential(
    (0): ConvTranspose2d(100, 1024, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU(inplace=True)
    (15): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (16): Tanh()
  )
)
```

```
_netD(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1024, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (12): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): LeakyReLU(negative_slope=0.2, inplace=True)
    (14): Conv2d(1024, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (15): Sigmoid()
  )
)
```

Caractéristiques d’un DCGAN

- Les couches Fully Connected et de Max Pooling sont supprimées.
- Le réseau G contient des couches de convolution transposée.
- Le DCGAN contient une couche de Batch Normalization sauf pour la couche de sortie du Générateur et la couche d’entrée du Discriminateur.
- Le DCGAN contient la fonction d’activation ReLU pour toutes les couches du Générateur hormis sa couche de sortie utilisant la fonction tanh.
- Le DCGAN contient la fonction d’activation LeakyReLU sur toutes les couches du Discriminateur.

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

Data processing

- Re-dimensionnement: 128x128 versus 64x64 initial
- Input normalization: cf. figure à droite
- CenterCrop: coupe l'image au centre

Choix du meilleur modèle pour la baseline

	DCGAN1 Pure DCGAN with Spherical noise	DCGAN1_ba	DCGAN1_lr	DCGAN2 Normalize Input	DCGAN3 Normalize input and One-sided label smoothing	DCGAN4 normalize input and One-sided label smoothing	DCGAN4_mod normalize input and One-sided label smoothing
epochs	50	50	50	**100**	**100**	**100**	**100**
batch_size	64	**94**	64	64	64	64	64
lr	0.0002	0.0002	**0,002**	0.0002	0.0002	0.0002	0.0002
image_size	136	136	136	136	136	136	136
scale_size	64	64	64	64	64	64	64
z_dim	100	100	100	100	100	100	100
G_features	64	64	64	64	64	64	64
D_features	64	64	64	64	64	64	64
image_channels	3	3	3	3	3	3	3
beta1	0.5	0.5	0.5	0.5	0.5	0.5	0.5
cuda	True	True	True	True	True	True	True
seed	7	7	7	7	7	7	7
workers	2	2	2	2	2	2	2
weight_decay	1.00E-04	1.00E-04	1.00E-04	1.00E-04	1.00E-04	1.00E-04	1.00E-04
Normalized input				**N(0.5, 0.5, 0.5)**	**N(0.5, 0.5, 0.5)**	**N(0.5, 0.5, 0.5)**	**N(0.5, 0.5, 0.5)**
sided_fake_label							**0.3**
sided_real_label					**0.9**	**0.9**	**0.9**
Freezing						**loss D < 0.7 loss G**	**loss D < 0.7 loss G**

`epochs`: nombre d'apprentissages
`batch_size`: taille de l'échantillon de l'apprentissage
`lr`: taux d'apprentissage pour l'optimisation de la fonction de perte
`image_size`: taille initiale de l'image
`scale_size`: taille de l'image en sortie
`z_dim`: taille du vecteur en entrée du Générateur
`G_features`: facteur du nombre de features en sortie de la première couche de convolution du Générateur
`D_features`: nombre de features en sortie de la première couche de convolution du Discriminateur
`image_channels`: nombre de features égal à 3 pour RGB
`cuda`: utilisation de la plateforme de calculs parallèles simplifiant l'utilisation du GPU
`seed`: graine définie pour des résultats reproductibles lorsque la génération aléatoire est utilisée.
`Workers`: nombre de sous-processus à utiliser pour le chargement des données. Par défaut, la valeur num_workers est fixée à zéro indiquant au chargeur de charger les données dans le processus principal.
`Normalized input`: normalisation des données en entrée
`Freezing`: stopper la remise à 0 du gradient lorsque D_losses < 0.7 * G_losses
`weight_decay`: hyper-paramètre de l'optimizer, `weight_decay` est le terme λ avec G le Gradient, W les poids, L la fonction de perte.

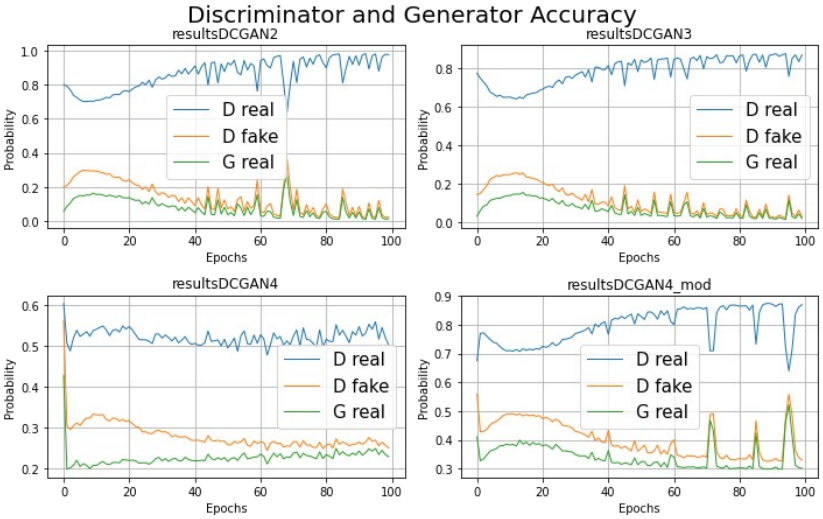
$$G^{(i)} = \frac{dL}{dW^{(i)}} + \lambda W^{(i)}$$

`Beta1`: hyper-paramètre de l'optimizer, il s'agit de l'exponentiel de `decay_rate` où `decay_rate = λ / sqrt( t)`, avec *t* nombre d'epochs

`sided_fake_label`: lissage des étiquettes où 0,3 (faux) au lieu de 0
`sided_real_label`: lissage des étiquettes où 0,9 (vrai) au lieu de 1

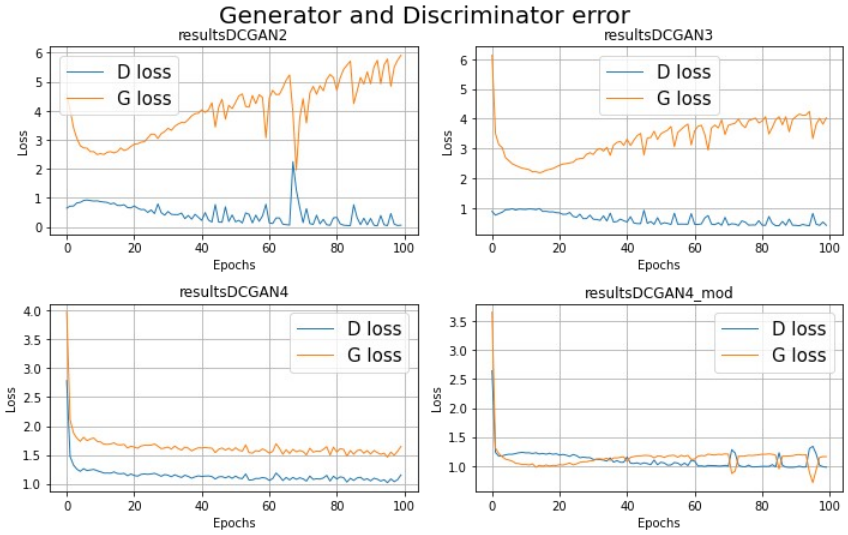
Performances

Parmi les modèles DCGAN testés, seuls les modèles avec epochs = 100 seront pris en compte afin d'obtenir des images moins opaques en sortie. Comparons les modèles restant:



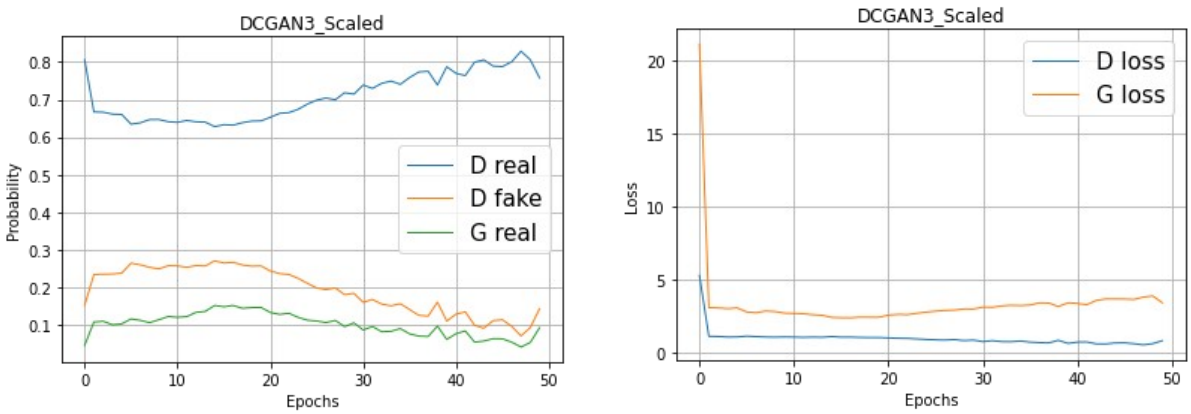
- Si l'`accuracy D_real` est élevé, le Discriminateur prédit correctement la classe des images réelles.
- Si l'`accuracy D_fake` est élevé, le Discriminateur prédit correctement la classe des images générées.
- Si l'`accuracy G_real` est élevé: `G_real = netD(fake).data.means()` - le Discriminateur prédit correctement la classe des images générées. Ces images sont retransmises au Générateur. Le Générateur va alors générer des images en les faisant passer pour des vraies.

Les modèles DCGAN4 et DCGAN4_mod obtiennent une `accuracy` trop faible ou trop instable pour le Discriminateur. Reste donc DCGAN2 et DCGAN3:



Après analyse de la fonction de perte, le DCGAN3 est plus favorable ayant une erreur de prédiction proche de 4 pour le Générateur.

Le modèle final pour la baseline sera donc le DCGAN3. Ce modèle sera réadapté pour générer des images plus grandes passant de 64x64 à 128x128.



Après redimensionnement de l’image

Panel d’images réelles (gauche) et générées (droite)
Temps d’entraînement 63,17 minutes

Analyse des résultats DCGAN versus StyleGAN2-ADA

DCGAN 128x128 **StyleGAN2 ADA 256x256**

Avec le DCGAN, quelques zones de l'image sont bruitées et floutées, les détails sont peu visibles alors que l'image n'est passée que de 64x64 à 128x128. Le DCGAN génère des images plus nettes uniquement sur des images de petites dimensions.

Le StyleGAN2 ne prend que des images de dimensions 256x256 ou 1024x1024. On arrive à mieux percevoir les détails et les contours. Ce modèle génère des images de qualité identique au input.

En conclusion, par le StyleGAN2 ADA on peut percevoir:

- les contours des lacets
- les reliefs de la semelle
- les ombres ou la profondeur
- les différences de matières

Éventuels cas d'échec

1) Choix du modèle à implémenter.

Les modèles récents donnent de très bons résultats mais en contre partie ils sont complexes et difficiles à implémenter sur un autre dataset. Par exemple, le modèle DALL-E d'OpenAI n'a pas de modèles pré entraînés sur des images baskets ni de documentation sur le transfert learning.

2) Bugs système

Processus d'installation de CUDA compatibles avec les versions de python et de ses modules. Nécessité de remettre à zéro tout le système et de réinstaller à partir de zéro les librairies et CUDA sur des sources fiables comme NVIDIA.