Generative Adversarial Networks pour la génération de Pokemons

Quentin Churet
CentraleSupelec
3A-OSY-E3-DL
quentin.churet@gmail.com

Donatien Criaud
CentraleSupelec
3A-OSY-E3-DL
donatien.criaud@gmail.com

Abstract

Dans ce document nous présentons le résultats de nos travaux sur la génération d'images de Pokemons en utilisant des Generative Adversarial Networks. Nous avons choisi d'expérimenter trois méthodes différentes évoquées dans le cadre de notre cours de deep learning. Nous avons commencé par entraîner des modèles de Deep Convolutional GAN afin de générer des premiers résultats. Cependant cette méthode est très instable ce qui nous a poussé à implémenter un GAN dit de Wasserstein minimisant la distance entre la distribution des images réelles et des images générées. Bien que les résultats de ces entraînements aient été bons, le manque de données nous a posé problème. Nous avons donc expérimenté les techniques mises en oeuvre dans les Progressive Growing GANs afin d'avoir un contrôle relativement précis des différentes couches de notre réseau générateur.

NB: Les notebooks contenant nos travaux sont tous disponibles sur le repository GitHub: https://github.com/DonaCrio/gotta-gen-em-all.

1. Introduction

Les *Generative Adversarial Networks (GAN)* sont des modèles de génération qui ont fait leurs preuves dans un champ étendu de domaines. Ils ont un très grand nombre d'applications pour le traitement de l'image : Coloration d'images, génération de textures, augmentation de résolution etc... Leur utilisation a même atteint le domaine de l'art et certains réseaux sont aujourd'hui entraînés à générer de la musique classique à la manière de Bach ou à peindre *"The next Rembrandt"* [5].

Les Pokemons sont des créatures issues de l'imagination de Satoshi Tajiri leur créateur. La franchise s'est déclinée en plus de 7 générations sur une vingtaine d'années. Nous voulons donc rendre hommage à cet univers en utilisant nos connaissances acquises lors d'un cours de deep learning pour générer une toute nouvelle génération de Pokemons.

Notre démarche a été d'utiliser l'état de l'art afin d'adapter des modèles et paradigmes existants à notre objectif. L'intérêt est double : Partir des modèles les plus "simples" afin d'appliquer les acquis de notre cours ; puis concevoir des solutions plus complexes et plus adaptés à notre problème. Ainsi nous avons commencé par utiliser un *Deep Convolutional GAN* [8] afin de crée un réseau "de référence" et de lancer de premiers entraînement. Ensuite nous avons exploré des techniques de régulation permettant de conserver un équilibre entre le réseau *Generator* et le réseau *Discriminator* avec notamment l'utilisation de la *Wasserstein Loss* [1] et *Gradient penalty* [4]. Enfin, nous avons entraîné un modèle en utilisant la technique des *Progressive Growing of GANs* [7] évoqué en cours.

2. Dataset

Les données utilisées pour entraîner les modèles sont issues du dataset *Kaggle Pokemon images* [6]. Ce dataset contient une image de chaque Pokemon jusqu'à la 7ième génération. Ces images sont au format *.png* et contiennent des canaux transparents. Afin de pouvoir les utiliser avec Pytorch, il est nécessaire de les convertir en niveau des couleur RGB et de les enregistrer au format *.jpg*. Nous avons choisis de remplacer les pixels transparents par des pixels blancs afin de garder une délimitation claire entre chaque Pokemon et le fond de l'image. Une fois les données retravaillées et converties au bon format, elles ont été *pushed* sur un repository Github [2] afin de les garder accessibles à tout moment.

Le dataset ne contient pas assez d'images pour entraîner un mode adversarial donc nous avons effectué de la *data augmentation* sur chaque batch à chaque epoch. Les transformations effectuées sont présentées sur la table 1.

Transformation	Paramètres	Probabilité
Resize	size: 150px	0.5
Crop	size: 128px	0.5
Horizontal flip	/	0.5
Rotation	angle: $\pm$ 10 degres	1.0

Table 1: Transformations effectuées sur le dataset

De cette manière nous obtenons des images de Pokemon recentrées, pouvant être inversées horizontalement et ayant subit une rotation. Nous évitons de trop modifier l'orientation de ces images afin de conserver les notions de "haut" et de "bas" dans le design de chaque Pokemon.

Figure 1: Échantillon des images utilisées

Nous avons choisi de ne pas mixer plusieurs sets de données car la différence de graphismes entre ceux-ci étaient trop importantes.

3. Deep Convolutional GAN

Dans nos premiers travaux nous avons tenté de mettre en oeuvre les concepts de base des GANs. Pour cela nous avons employé une technique d'entraînement *adversarial* classique entre des réseaux à convolutions.

3.1. Principe

Le principe des *Deep Convolutional GAN* est d'apprendre à un modèle la distribution d'un set d'images afin de pouvoir en générer de nouvelles en utilisant cette distribution. Pour cela on utilise deux modèles appelés *générateur* et *discriminateur*. Le rôle du générateur est de produire de "fausses" images qui ressemble de très près aux "vraies" images de l'échantillon d'entraînement. Le discriminateur doit lui pouvoir décider de manière précise si un image qui lui est présentée est "vraie" ou "fausse". Durant l'entraînement, le générateur va donc essayer de "ruser" et faire passer des images générées comme étant de vraies images. L'équilibre est considéré comme atteint lorsque le discriminateur classifie 50% des images générées comme étant vraies (ce qui correspond à un choix au hasard).

Notons x l'image traitée et z le vecteur latent utilisé par le générateur pour créer une image. La prédiction du discriminateur sur l'image est notée $D(x)$ et l'image générée $G(z)$. Le but du générateur est donc de minimiser la probabilité que le discriminateur prédise un image générée comme étant fausse $log(1 - D(G(z)))$, tandis que le générateur doit maximiser la probabilité de classifier correctement les vraies images $log(D(x))$.

L'article [3] qui décrit l'entraînement adversarial de deux réseaux, résume celui-ci à un problème mini-max entre les deux réseaux, dont la fonction d'optimisation est la suivante:

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

où p_z est la distribution estimée par le générateur et p_x est la distribution des données réelles.

En théorie, la solution à ce problème est $p_z = p_x$, lorsque le discriminateur prédit aléatoirement si les images qui lui sont présentées sont réelles ou non. Cependant, atteindre un tel point d'équilibre est compliqué et sera discuté plus loin.

3.2. Architecture

Les *Deep Convolutional GAN* (DCGAN) sont donc une application directe des GANs décrits plus hauts et utilisent des couches de convolution et de convolution transposé dans l'architecture du générateur et du discriminateur. Nous utilisons donc naturellement l'architecture classique des GANs représentée figure 2.

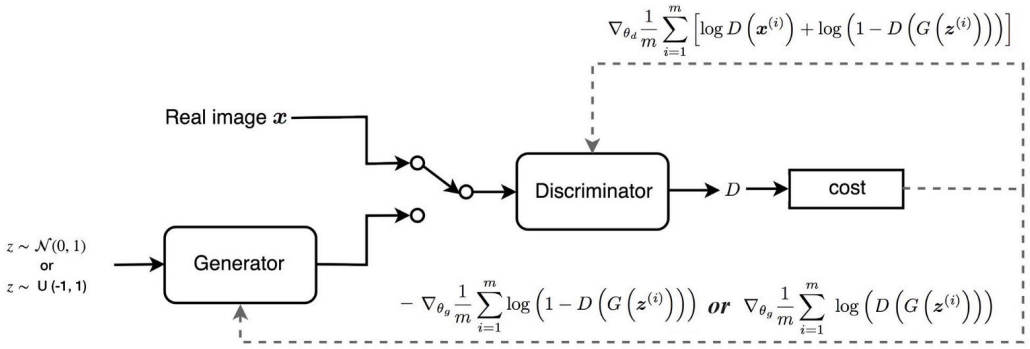


Figure 2: Architecture d'entraînement du DCGAN

Dans le cas de la génération d'images de Pokemons nous utilisons les architectures présentées figure 3.

Layer (type)	Output Shape	Param #
ConvTranspose2d-1	[-1, 2048, 4, 4]	3,276,800
BatchNorm2d-2	[-1, 2048, 4, 4]	4,096
ReLU-3	[-1, 2048, 4, 4]	0
ConvTranspose2d-4	[-1, 1024, 8, 8]	33,554,432
BatchNorm2d-5	[-1, 1024, 8, 8]	2,048
ReLU-6	[-1, 1024, 8, 8]	0
ConvTranspose2d-7	[-1, 512, 16, 16]	8,388,608
BatchNorm2d-8	[-1, 512, 16, 16]	1,024
ReLU-9	[-1, 512, 16, 16]	0
ConvTranspose2d-10	[-1, 256, 32, 32]	2,097,152
BatchNorm2d-11	[-1, 256, 32, 32]	512
ReLU-12	[-1, 256, 32, 32]	0
ConvTranspose2d-13	[-1, 128, 64, 64]	524,288
BatchNorm2d-14	[-1, 128, 64, 64]	256
ReLU-15	[-1, 128, 64, 64]	0
ConvTranspose2d-16	[-1, 64, 128, 128]	131,072
BatchNorm2d-17	[-1, 64, 128, 128]	128
ReLU-18	[-1, 64, 128, 128]	0
ConvTranspose2d-19	[-1, 3, 256, 256]	3,072
Tanh-20	[-1, 3, 256, 256]	0

(a) Architecture du générateur

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 64, 64]	6,912
LeakyReLU-2	[-1, 64, 64, 64]	0
Conv2d-3	[-1, 128, 32, 32]	131,072
BatchNorm2d-4	[-1, 128, 32, 32]	256
LeakyReLU-5	[-1, 128, 32, 32]	0
Conv2d-6	[-1, 128, 16, 16]	262,144
BatchNorm2d-7	[-1, 128, 16, 16]	256
LeakyReLU-8	[-1, 128, 16, 16]	0
Conv2d-9	[-1, 128, 8, 8]	262,144
BatchNorm2d-10	[-1, 128, 8, 8]	256
LeakyReLU-11	[-1, 128, 8, 8]	0
Conv2d-12	[-1, 128, 4, 4]	262,144
BatchNorm2d-13	[-1, 128, 4, 4]	256
LeakyReLU-14	[-1, 128, 4, 4]	0
Conv2d-15	[-1, 1, 1, 1]	2,048
Sigmoid-16	[-1, 1, 1, 1]	0

(b) Architecture du discriminateur

Figure 3: Architecture des réseaux DCGAN

Les réseaux ne possèdent pas de couches *fully connected* car plusieurs entraînements nous on montré que ces couches provoquaient une grande instabilité et une divergence des réseaux : Le discriminateur est très performant et empêche le générateur de s'améliorer. Nous explorons des méthodes pour éviter ce phénomène dans la section suivante.

3.3. Entraînement

La table 2 présente les paramètres d'entraînement utilisés pour générer des Pokemons. Nous utilisons une batch size de 16 images afin d'éviter des problèmes de mémoire en entraînant le modèle sur des échantillons de 256x256 pixels. L'état de l'art utilise des vecteurs latents de 100 ou 128 valeurs et un learning rate de 2e-4.

Paramètre	Valeur
taille vecteur latent	100
batch size	16
learning rate	2e-4
taille image	256x256

Table 2: Paramètres d'entraînement du DCGAN

3.4. Résultats

Les différents entraînements que nous avons effectué montrent que les DCGAN sont très sensibles au variations des hyperparamètre. De plus la courbe de loss obtenue est très erratique (figure 4), cependant ceci est principalement du au fait que nous entraînons chaque réseau à chaque epoch. Nous verrons plus loin que nous pouvons entraîner le discriminateur plusieurs fois avant de mettre à jour le générateur.

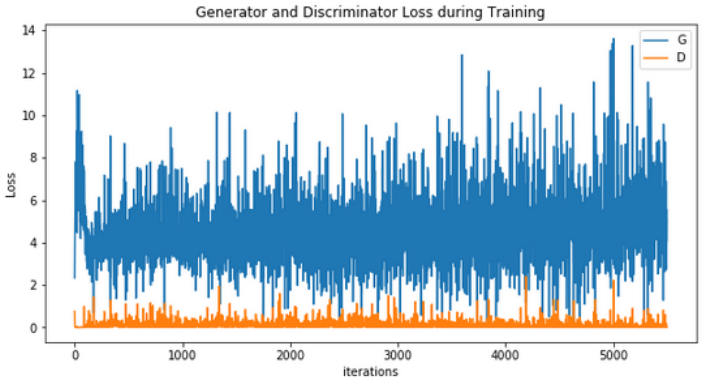


Figure 4: Fonction loss pendant l'entraînement du DCGAN

La figure 5 présente un échantillon de Pokemon générés par notre réseau. Bien que ceux-ci semblent de bonne qualité, nous pouvons remarque que beaucoup peuvent faire penser à des Pokemons déjà existant. Ce phénomène est probablement dû au fait que nous utilisons uniquement des convolutions dans nos architectures et donc nous avons tendance à reproduire et non à générer. Nous avons tenté d'ajouter des couches *fully connected* mais le DCGAN étant très sensible aux changements d'architecture, la loss du générateur explosait.

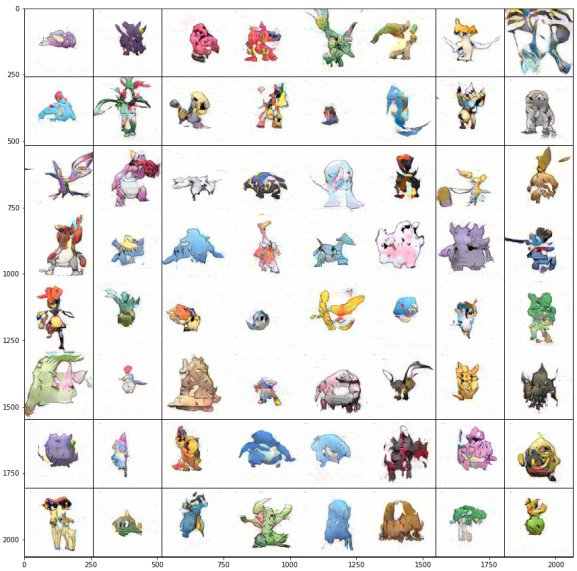


Figure 5: Échantillon de Pokemons générés

4. Équilibrage des réseaux

Comme évoqué précédemment, nos expérimentations nous ont permis de générer des Pokemons mais ceux-ci présentaient des features ressemblant fortement aux originaux. Afin d'éviter cela, nous avons ajouté des couches *fully connected* à nos réseaux. Cependant, cela a pour effet de déséquilibrer l'entraînement, aussi nous explorons dans cette section des techniques pour contrebalancer ces effets.

4.1. Wasserstein GAN et Gradient penalty

Le GAN de Wasserstein (WGAN) est introduit pour la première fois par Martin Arjovsky en 2017 dans son article "Wasserstein GAN" [4]. Il propose une méthode d'entraînement alternative permettant de mieux approximer la distribution des données dans un dataset.

Au lieu d'entraîner le générateur à "tromper" le discriminateur, le WGAN va s'attacher à donner un score de vraisemblance à chaque image. Le discriminateur est donc ici employé pour attribuer ce score à une image et non plus pour essayer de prédire la provenance de chacune d'elles. Ces changements de paradigme s'expliquent par le fait que le générateur a plutôt intérêt à minimiser la distance entre la distribution observées des images réelles et celle des exemples générés. Ceci a pour effet de stabiliser l'entraînement et de le rendre plus robuste aux changements d'architecture et d'hyperparamètres.

Contrairement au DCGAN qui utilise une classification binaire pour prédire la probabilité qu'une image soit vraie ou fausse, le WGAN utilise la distance suivante pour estimer l'écart entre deux distributions :
$W(P_1, P_2) = inf_{\gamma \in \Pi(P_1,P_2)} E_{(x,y)\ \gamma}[||x - y||]$
La figure 6 résume les différences entre les loss des deux techniques.

	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f(x^{(i)}) - f(G(z^{(i)}))\right]$	$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f(G(z^{(i)}))$

Figure 6: Différence entre les loss

4.2. Architecture

Nous changeons l'architecture de l'entraînement pour une architecture décrite par la figure 7.

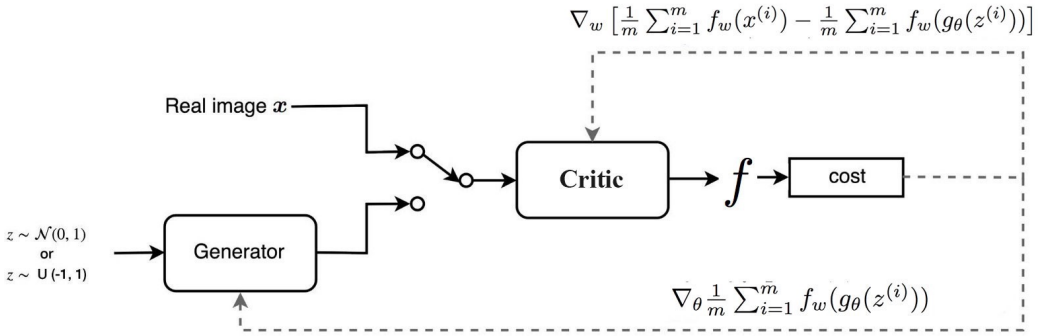


Figure 7: Architecture d'entraînement du WGAN

Les réseaux utilisés sont sensiblement les mêmes que précédemment à la différence près que l'on introduit une couche *fully connected* entre l'entrée du générateur et la première couche de convolution transposée (figure 8).

4.3. Entraînement

La table 3 présente les paramètres d'entraînement utilisés pour entraîner le WGAN. Nous utilisons une batch size de 128 images car la taille des images est de 128x128 pixels. Cette dernière n'est plus de 256x256 car cela a peu d'impact sur la qualité des images générées.

Layer (type)	Output Shape	Param #
Linear-1	[-1, 4096]	266,240
BatchNorm1d-2	[-1, 4096]	8,192
ConvTranspose2d-3	[-1, 512, 4, 4]	3,276,800
BatchNorm2d-4	[-1, 512, 4, 4]	1,024
ReLU-5	[-1, 512, 4, 4]	0
Dropout-6	[-1, 512, 4, 4]	0
ConvTranspose2d-7	[-1, 256, 8, 8]	3,276,800
BatchNorm2d-8	[-1, 256, 8, 8]	512
ReLU-9	[-1, 256, 8, 8]	0
Dropout-10	[-1, 256, 8, 8]	0
ConvTranspose2d-11	[-1, 128, 16, 16]	819,200
BatchNorm2d-12	[-1, 128, 16, 16]	256
ReLU-13	[-1, 128, 16, 16]	0
ConvTranspose2d-14	[-1, 64, 32, 32]	204,800
BatchNorm2d-15	[-1, 64, 32, 32]	128
ReLU-16	[-1, 64, 32, 32]	0
ConvTranspose2d-17	[-1, 32, 64, 64]	51,200
BatchNorm2d-18	[-1, 32, 64, 64]	64
ReLU-19	[-1, 32, 64, 64]	0
ConvTranspose2d-20	[-1, 3, 128, 128]	1,536
Tanh-21	[-1, 3, 128, 128]	0

(a) Architecture du générateur

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 32, 32]	3,456
LeakyReLU-2	[-1, 32, 32, 32]	0
Conv2d-3	[-1, 64, 16, 16]	32,768
BatchNorm2d-4	[-1, 64, 16, 16]	128
LeakyReLU-5	[-1, 64, 16, 16]	0
Conv2d-6	[-1, 64, 8, 8]	65,536
BatchNorm2d-7	[-1, 64, 8, 8]	128
LeakyReLU-8	[-1, 64, 8, 8]	0
Conv2d-9	[-1, 64, 4, 4]	65,536
BatchNorm2d-10	[-1, 64, 4, 4]	128
LeakyReLU-11	[-1, 64, 4, 4]	0
Conv2d-12	[-1, 1, 1, 1]	1,024

(b) Architecture du discriminateur

Figure 8: Architecture des réseaux WGAN

Paramètre	Valeur
taille vecteur latent	64
batch size	128
learning rate	1e-4
taille image	128x128

Table 3: Paramètres d'entraînement du WGAN

4.4. Résultats

La figure 9 présente les résultats de l'entraînement. On observe que malgré une forte amélioration des performances du discriminateur, les deux distributions restent proches, et ce même avec la présence de fluctuations importantes. cela montre bien que la distance de Wasserstein permet d'équilibrer nos deux réseaux.

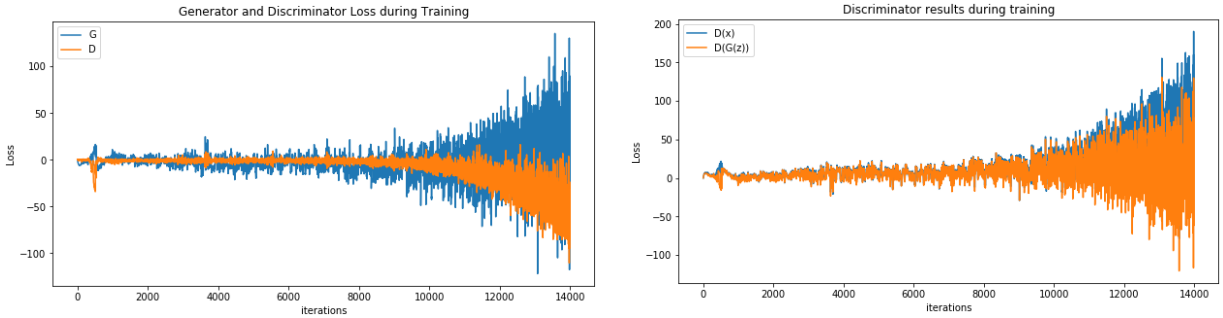


Figure 9: Résultats de l'entraînement du WGAN

On remarque aussi une amélioration des Pokemons générés (échantillon figure 10), non pas dans la résolution des images mais dans le fait qu'il n'y a maintenant plus de features reconnaissable de Pokemons existants.

La faible qualité de la génération peut s'expliquer notamment par le fait que l'on ne dispose que de 802 images de Pokemons. Nous allons essayer d'améliorer les résultats obtenus en utilisant une dernière méthode qui nous permettra de mieux contrôler les couches de convolution intermédiaires.

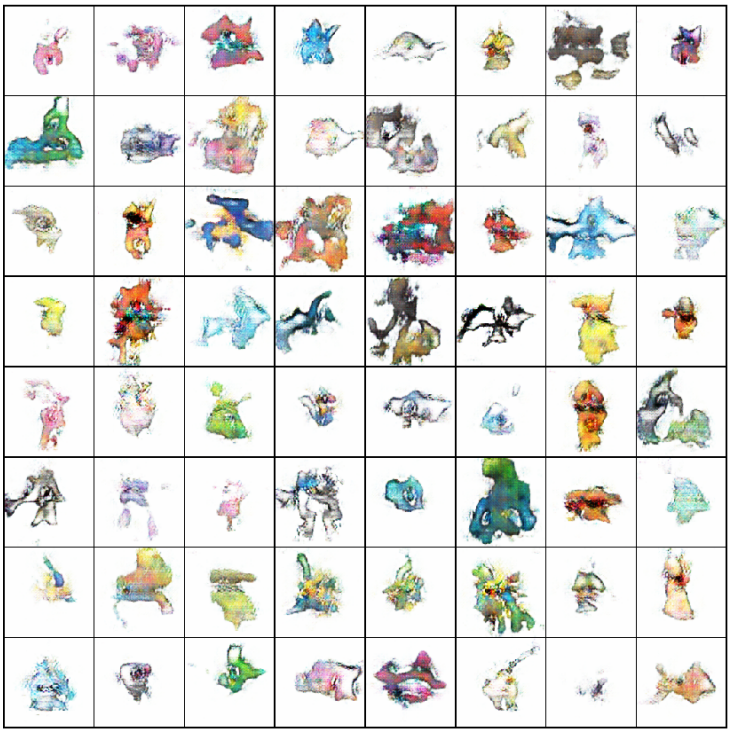


Figure 10: Résultats de l'entraînement du WGAN

5. Progressive Growing of GANs

Nous avons dans cette section tenté d'appliquer la technique de *Progressive Growing of GANs* [7] (PGAN) introduite en cours.

5.1. Principe

Le principe des Progressive GAN est de générer des fausses images de taille progressivement supérieures, en entraînant le générateur à crée des images de 4x4 pixels jusqu'à la taille souhaitée, en passant par des multiples de 2 (16x16, 32x32...). Dès lors que l'entraînement est terminé pour une taille donné, on rajoute une couche de convolution de taille supérieure, permettant d'agrandir la taille de l'image générée en sortie.

Toutefois, afin de garder la stabilité de l'image générée à une taille inférieure (le contraste étant moins important), l'input en sortie du générateur est une somme pondérée de l'output de la couche de taille inférieure, extrapolée (on multiplie les dimensions de son image par 2), et de l'output de sortie de la taille voulue. Ceci est présenté sur la figure 11 .

Au fur et à mesure de l'apprentissage, On diminue progressivement la contribution des couches précédentes à la sortie, et ce jusqu'à ce que l'image produite le soit entièrement par la dernière couche.

L'article met en avant le fait qu'il est important de normaliser le *learning rate* sur chaque couche : Étant donné que chacune génère des échantillons de taille différente, leur poids est d'importance différente sur la génération des images. Ainsi, il est nécessaire d'utiliser un multiplicateur pour adapter la back-propagation sur chaque couche. Enfin, la loss employée est la distance de Wasserstein comme décrit précédemment.

5.2. Entraînement

La table 4 présente les paramètres d'entraînement utilisés pour entraîner le PGAN. Nous utilisons une batch size de 4 images afin d'éviter des problèmes de mémoire lorsqu'on augment la taille des images générées. On utilise une batch size de 2 pour les images de 128x128 pixels pour les mêmes raisons.

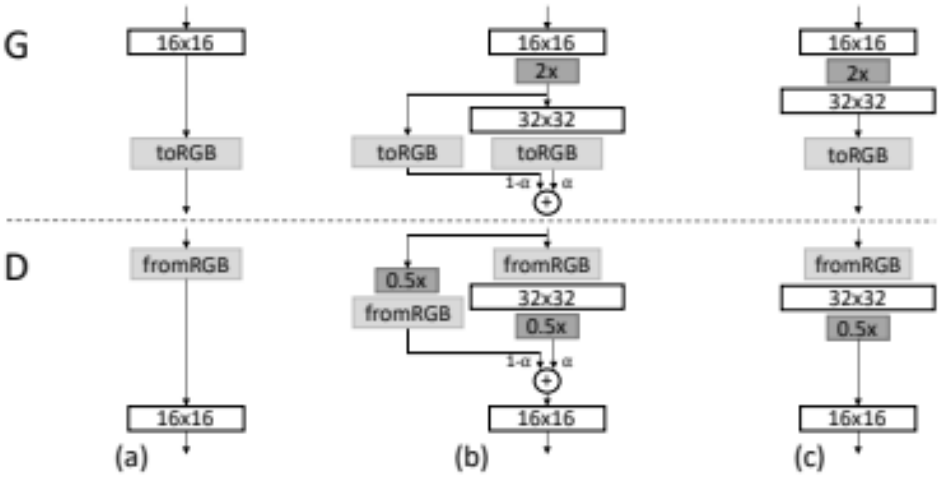


Figure 11: Architecture et évolution des réseaux à chaque itération

Paramètre	Valeur
batch size	4
batch size (128x128)	2
learning rate	1e-3
taille image	128x128

Table 4: Paramètres d'entraînement du PGAN

5.3. Résultats

On remarque immédiatement que ce genre d'entraînement par PGAN est beaucoup plus stable que précédemment. En effet, les loss présentées en figure 12 se stabilisent avec des oscillations relativement faibles.

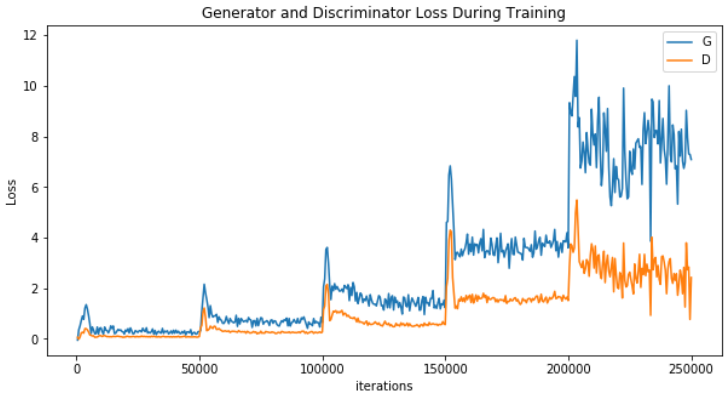
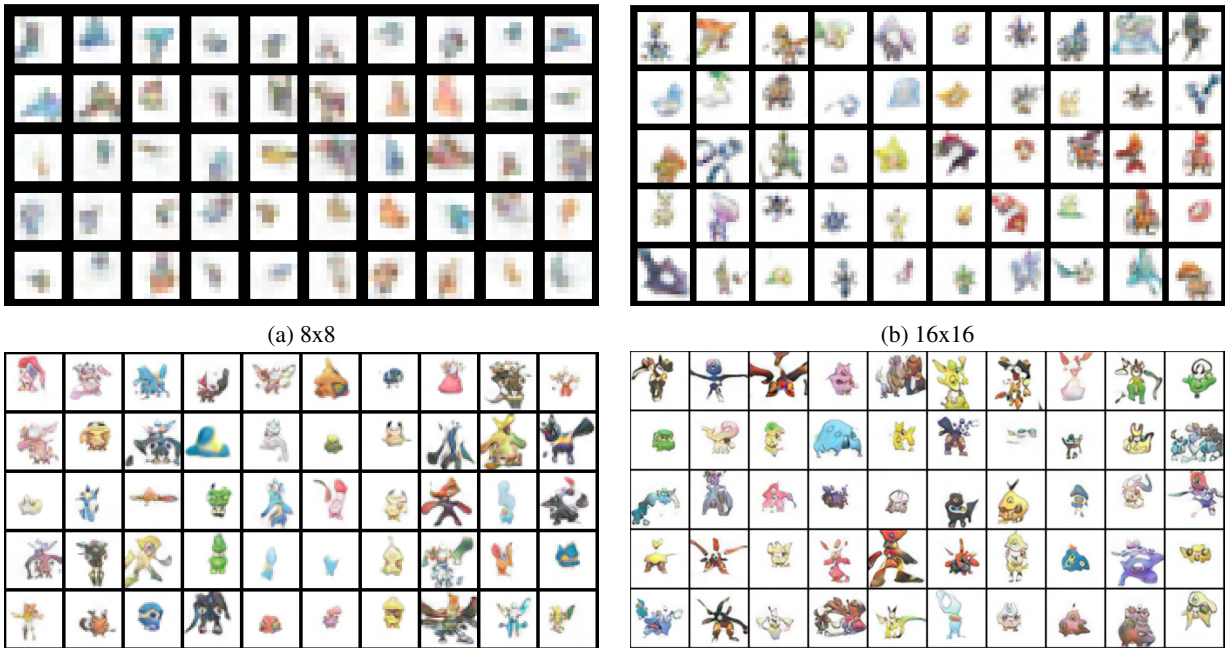


Figure 12: Fonctions loss du PGAN pendant l'entraînement

L'allure générale de ces loss montre des pics qui correspondent en fait au changement de résolution des images générées. On observe ensuite une stabilisation. De plus, les performances du discriminateur doublent lorsque l'on double la taille des images (et donc leur résolution).

On voit tout de même que ce type de réseau est très robuste à des changements d'architecture, d'hyperparamètres et de taille d'images. De plus, cela nous a permis de générer des Pokemons vraisemblables avec un dataset contenant peu d'images initialement.

Un exemple de Pokemons générés pour chaque taille d'image est présenté figure 13.



(a) 8x8 (b) 16x16

(c) 64x64 (d) 128x128

Figure 13: Évolution de la génération de Pokemons par PGAN

Un set d'images montrant l'évolution complète de la génération est disponible sur le repository GitHub [2].

6. Conclusion

La génération de Pokemons est donc bien possible en utilisant des techniques de *Generative Adversarial Networks*. Au cours de nos différents travaux nous avons pu résoudre plusieurs problèmes en améliorant nos techniques d'entraînement. Ainsi, nous avons pu éviter la reproduction de features réelles en modifiant l'architecture de notre GAN et en utilisant les Wasserstein GAN pour la stabilisation. Nous avons aussi réussi à nous affranchir de la petite quantité de données en utilisant des *Progressive Growing GAN* utilisant la distance de Wasserstein.

Nous pensons que nos travaux pourraient être grandement améliorés en concevant des PGAN plus adaptés et précis, ce que le manque d'expérience et de moyen nous a empêché de faire. De plus, il serait intéressant d'entraîner les GANs sur un type unique de Pokemon. En effet, les Pokemons de type eau on plus tendance à être bleus et les Pokemons de type air ont souvent des ailes. Ainsi les réseaux seraient plus à même de capter les distributions des données.

Voici sur la figure 14 les Pokemons qui feront peut-être un jour partie des starters de la nouvelle génération...

Figure 14: Un nouveau départ ?

References

[1] Martin Arjovsky, Soumith Chintala, , and Leon Bottou. Wasserstein gan. 2017.

[2] Donatien Criaud. Processed pokemon dataset, 2020. Pokemon sprites in RGB format and sorted by type retrieved from https://github.com/DonaCrio/Pokemon-dataset.

[3] Ian J. Goodfellow. Generative adversarial nets. 2014.

[4] Ishaan Gulrajani, Faruk Ahmed, Vincent Dumoulin Martin Arjovsky and, and Aaron Courville. Improved training of wasserstein gans. 2017.

[5] ING. The next rembrandt, 2017. https://www.nextrembrandt.com/.

[6] Kaggle. Pokemon image dataset, 2018. Original Pokemon sprites retrieved from: https://www.kaggle.com/vishalsubbiah/pokemon-images-and-types.

[7] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability and variation. 2018.

[8] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 2015.