

Generative Adversarial Network

Loüet Joseph & Karmim Yannis
Spécialité **DAC**



Table des matières

1 Generative Adversarial Network	3
1.1 Principe général.	3
1.2 Architecture des réseaux et pratique.	4
2 Conditional Adversarial Network	18
2.1 Principe général.	18
2.2 Architecture cDCGAN pour MNIST.	19
2.3 Architecture cGAN pour MNIST.	21

1 Generative Adversarial Network

1.1 Principe général.

Pour l'apprentissage d'un GAN on cherche à optimiser le problème suivant :

$$\min_G \max_D \mathbb{E}_{\mathbf{x}^* \in \mathcal{D}_{\text{ata}}} [\log D(\mathbf{x}^*)] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \quad (1)$$

1.)

$$\max_G \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log D(G(\mathbf{z}))] \quad (2)$$

$$\max_D \mathbb{E}_{\mathbf{x}^* \in \mathcal{D}_{\text{ata}}} [\log D(\mathbf{x}^*)] + \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))] \quad (3)$$

L'équation 2 veut dire que pour un discriminateur D fixé, la fonction objective du générateur G va être de "tromper" le discriminateur D en lui faisant classer les exemples générés comme des données réelles, avec $D(G(\mathbf{z}))$ proche de 1.

L'équation 3 veut dire que pour un générateur G fixé, la fonction objective du discriminateur D est de classer les données réelles avec une probabilité la plus proche possible de 1, et inversement de classer les données générées avec une probabilité la plus proche possible de 0 afin que la partie $1 - D(G(\mathbf{z}))$ soit la plus haute possible.

Si on utilise seulement une des deux fonctions objective, on entraînera seulement D si on utilise l'équation (3) ou G si on utilise l'équation (2).

2.) Idéalement, le générateur G transforme la distribution $P(\mathbf{z})$ comme étant égal à la distribution $P(\mathbf{X})$ des données réelles.

3.) L'équation (2) n'est pas directement dérivée de l'équation (1) afin d'éviter la saturation du gradient. La vraie équation ici aurait été :

$$\min_G \mathbb{E}_{\mathbf{z} \sim P(\mathbf{z})} [\log 1 - D(G(\mathbf{z}))] \quad (4)$$

1.2 Architecture des réseaux et pratique.

4) Paramètres par défaut. Voici nos résultats après avoir implémenté l'architecture DCGAN 32x32 avec les hyperparamètres par défaut :

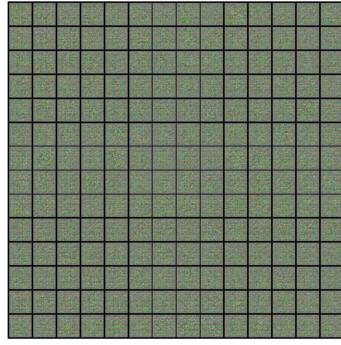


FIGURE 1 – Images générées à la première itération par le générateur avec les hyperparamètres par défaut.



FIGURE 2 – Image générées à la fin de l'apprentissage par le générateur avec les hyperparamètres par défaut.

Avec les hyperparamètres par défaut qui suivent ceux de l'article DCGAN, on obtient déjà des résultats satisfaisant. Dès la 100ème itération le générateur commence à produire des formes de visages assez symétrique. Le modèle converge rapidement puisqu'à partir de la 2000ème itération les visages sont assez ressemblant avec nos résultats finaux (8000ème itération).

Pour les **loss** on remarque que la loss du discriminateur décroît rapidement, ce

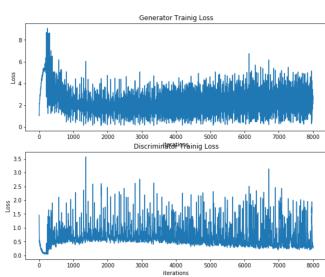


FIGURE 3 – Loss générateur et discriminateur avec les hyperparamètres par défaut.

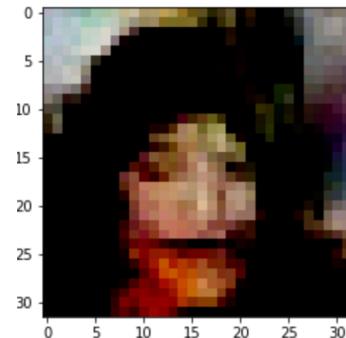


FIGURE 4 – Exemple de visage généré par le modèle.

qui est cohérent puisqu'au début les images générées sont encore très bruitées et ne ressemblent pas du tout à des visages. On remarque ensuite que lorsque

le générateur apprend et que sa loss décroît on a une légère hausse de la loss du discriminateur, ce qui nous montre que le générateur réussi à tromper le discriminateur. Malgré la grande variance de la loss du générateur elle reste relativement centré autour de 2.3 dès la 2000ème itération.

La diversité des images générées est également satisfaisante puisqu'on voit des images de femmes, d'hommes, de personnes âgées, des fonds assez variés, des visages souriants... Ce qui montre que notre générateur capte une distribution assez fidèle d'un visage humain.

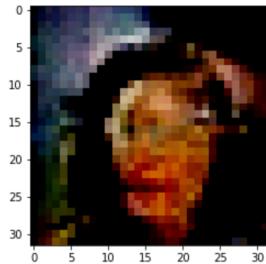


FIGURE 5 – Exemple de visage généré peu réal avec les paramètres par défaut.

Cependant le modèle reste assez instable puisqu'on peut obtenir des images relativement fidèle à un visage humain comme la **Figure 4** et des images très irréalistes comme la **Figure 5**. Cette instabilité se reflète dans nos loss puisque une image comme la **Figure 5** ne "trompe" pas le discriminateur, contrairement à la **Figure 4**.

5) Exploration des hyperparamètres.

Momentum. On modifie uniquement les Momentum des deux réseaux de l'optimiseur Adam de 0.5 pour 0.9 :

La modification de valeur du momentum a un impact important sur notre

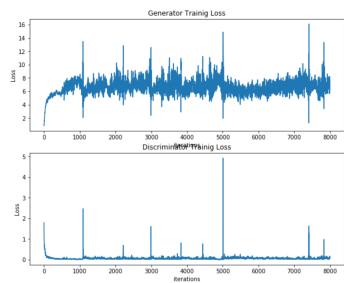


FIGURE 6 – Loss générateur et discriminateur avec un momentum de 0.9.



FIGURE 7 – Exemple de visage généré par le modèle avec un momentum de 0.9.

modèle, puisque le générateur a une loss beaucoup plus élevé et moyennant autour de 6.7 contrairement à 2.3 auparavant. Le discriminateur quant à lui n'a plus de difficulté à déterminer si l'image générée est réel ou non, et sa loss est donc beaucoup moins élevée.

Implémentation de la "vraie" loss dérivée de l'équation d'origine. Nous avons implémenté la loss de l'équation (2) qui était conseillé d'utiliser par les auteurs dans l'article originel des GAN afin d'éviter l'évaporation de gradient. Néanmoins nous allons implémenter la loss d'origine (4) dérivée de l'équation (1) qui a été présenté dans l'article GAN.

Modification de l'équilibrage des deux modèles. Par défaut on apprend autant sur notre générateur que sur notre discriminateur. On peut choisir de modifier l'équilibrage de notre apprentissage en apprenant plus l'un que l'autre.

. On apprend 5x plus le générateur :

En apprenant notre générateur 5x plus, le discriminateur a bien plus de mal

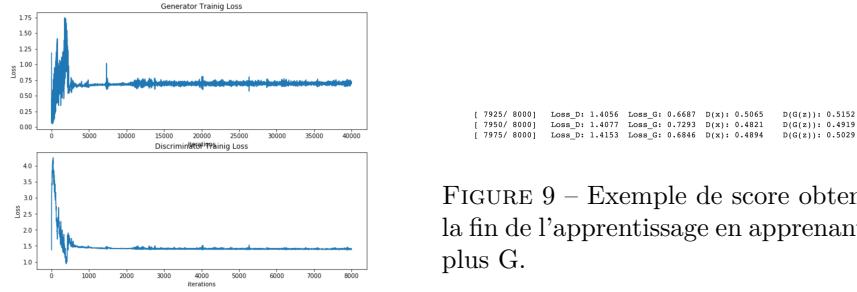


FIGURE 8 – Loss générateur et discriminateur en apprenant 5x plus G.

à déterminer si les images générées par $G(\mathbf{z})$ sont vraies ou fausses. Cependant il a également du mal à reconnaître les vrais images \mathbf{x} , et on a globalement $D(G(\mathbf{z})) \sim D(\mathbf{x}) \sim 0.5$. Comme on effectue plus d'étapes d'apprentissage sur le générateur, notre modèle est plus long à apprendre, idem pour D.

. On apprend 5x plus le discriminateur :

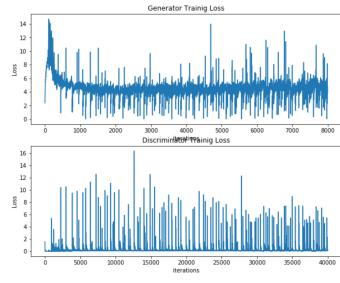


FIGURE 10 – Loss générateur et discriminateur en apprenant 5x plus D.



FIGURE 11 – Exemple de visages obtenus à la fin de l'apprentissage en apprenant 5x plus D.

La loss du discriminateur est très instable mais reste en moyenne très basse, on remarque aussi que le générateur apprend bien et converge, même si sa loss est en moyenne plus élevé qu'avec les hyper paramètres de base.

Enfin on remarque qu'on obtient des résultats plus réaliste sur la génération de visage en privilégiant plus l'apprentissage du discriminateur qu'en privilégiant l'apprentissage du générateur. Néanmoins les résultats restent meilleurs avec un équilibre de 1 pour 1 entre le discriminateur et le générateur.

Modification du learning rate On va expérimenter et modifier le learning_rate des deux réseaux, qui est par défaut 0.0002 pour G et D .

. **Learning_rate_D = 0.001 et Learning_rate_G = 0.0002** Le changement du learning_rate de D n'a pas un grand impact sur la loss de notre générateur, cependant on remarque la loss de D est en moyenne légèrement plus élevé qu'avec les hyper paramètres de base. Les visages générés sont à peu près semblable. On peut souligner également le fait que la probabilité qui est attribué par le discriminateur sur les images réelles $D(\mathbf{x})$ est beaucoup plus instable, on obtient plus souvent des probabilités inférieures à 0.5, ce qui nous pousse à croire que D a plus de mal à apprendre avec ce learning_rate.

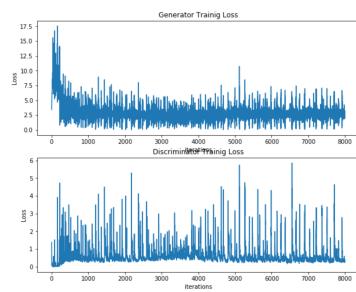


FIGURE 12 – Loss.



FIGURE 13 – Exemple de visage généré par le modèle à la fin de l'apprentissage.

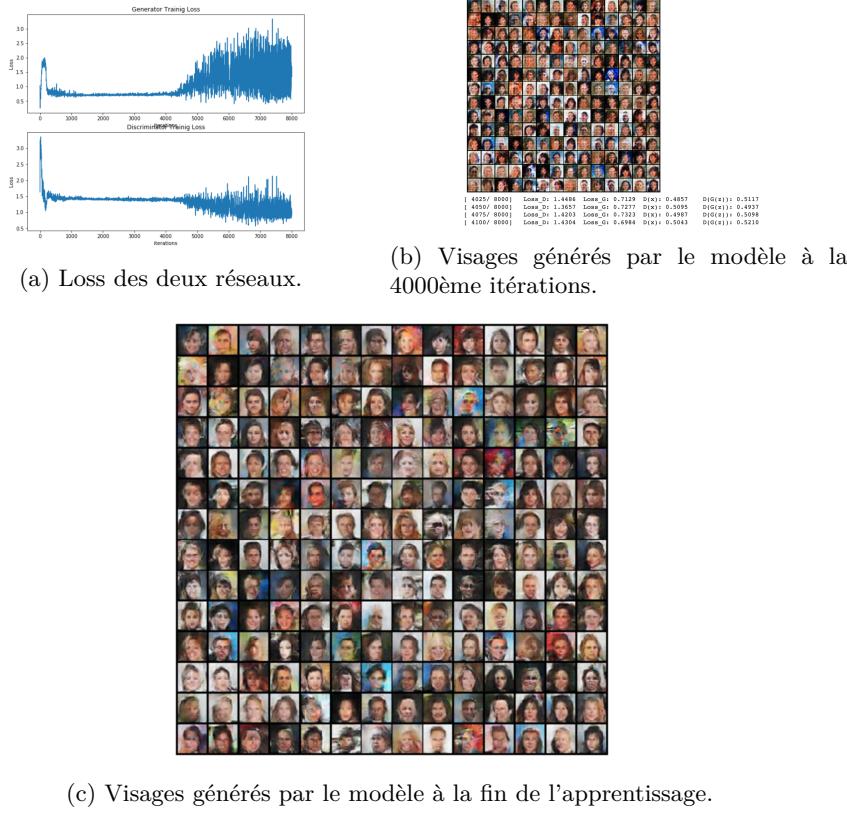


FIGURE 14 – Learning_rate G = 0.001

. **Learning_rate_D = 0.0002 et Learning_rate_G = 0.001** On modifie maintenant lr_G en l'augmentant d'un facteur 5, on accélère la descente de gradient de G , et D a donc du retard sur son apprentissage et peine à reconnaître comme faux les images générées par $D(G(\mathbf{z}))$, on a donc une loss pour G qui est très faible jusqu'à la 4000ème itérations. On pourrait croire alors que les visages générés sont très réaliste, cependant si l'on regarde de plus près on remarque que $D(\mathbf{x})$ a également une probabilité assez faible. Cela veut dire que notre générateur trompe certe, mais trompe un discriminateur D peu entraîné et les visages que l'on voit sur la figure 14(b) sont peu réaliste. Une fois que D rattrape son retard, sa loss diminue et celle de G augmente et on retrouve des courbes plus fidèle a ce qu'on avait avec les hyper paramètres de base.

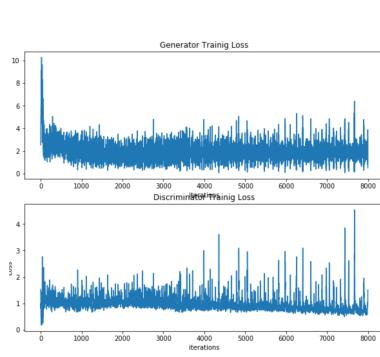


FIGURE 15 – Loss.



FIGURE 16 – Exemple de visage généré par le modèle à la fin de l'apprentissage.

. **Learning_rate_D = 0.001 et Learning_rate_G = 0.001** On a modifié les learning_rate des deux modèles, on a plus de stabilité et une loss en moyenne moins élevée pour G (environ 1.95 au lieu de 2.4 avec les paramètres de base). Cependant la loss de D est plus élevé et D a plus de mal à bien classifier les exemples réels \mathbf{x} . Il est donc plus facile pour G de tromper D et par conséquent même si la loss de G est moins élevé, il ne semble pas générer d'exemple plus réaliste qu'avec les hyper paramètres par défaut.

Modification du nombre d'epochs. On reprend maintenant les paramètres par défaut, seulement on effectue 25 epochs pour notre apprentissage.

Comme on pouvait l'attendre au vu du comportement que notre modèle avait

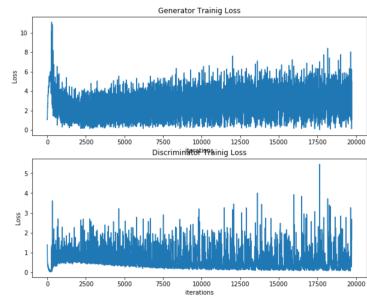


FIGURE 17 – Loss générateur et discriminateur sur 25 epochs.



FIGURE 18 – Exemple de visage généré par le modèle à la fin des 25 epochs.

avec les paramètres par défaut, les loss du générateur et du discriminateur n'évoluent pas beaucoup plus en 20 000 itérations qu'en 8 000 itérations. Les images générées sont relativement semblable. Le discriminateur continue néanmoins de décroître au fur et à mesure des epochs, ce qui montre bien que le générateur a atteint ses limites.

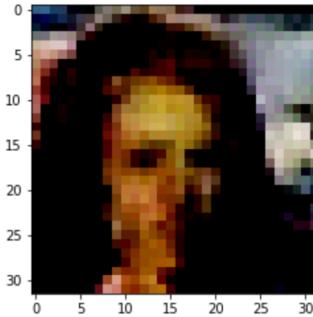


FIGURE 19 – Exemple de visage généré peu réal après 30 epochs.

Modification de la taille du vecteur latent n_z . On reprend les paramètres par défaut et on va modifier la taille du vecteur latent n_z en entrée de notre générateur qui était par défaut 100, afin de voir l'influence de ce paramètre sur notre modèle.

$n_z = 10$

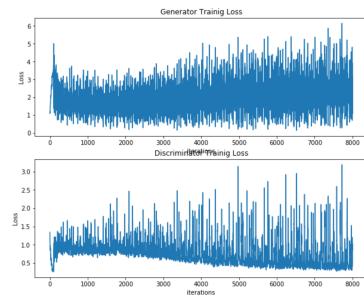


FIGURE 20 – Loss générateur et discriminateur pour une taille d'entrée $n_z = 10$.



FIGURE 21 – Exemple de visage généré par le modèle pour une taille d'entrée $n_z = 10$.

Pour une taille de vecteur latent $n_z = 10$ le modèle semble se comporter de la même façon que les paramètres par défaut, si ce n'est un peu mieux. On a une moyenne de loss pour G de 1.97, il faut cependant faire attention en raison de l'instabilité des loss et du modèle.

$n_z = 1000$

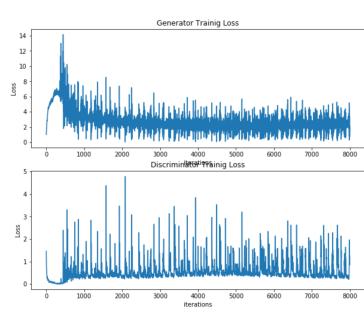


FIGURE 22 – Loss générateur et discriminateur pour une taille d’entrée $n_z = 1000$.



FIGURE 23 – Exemple de visage généré par le modèle pour une taille d’entrée $n_z = 1000$.

Pour une taille de vecteur latent $n_z = 1000$ le modèle est moins performant. On a une moyenne de loss plus élevé pour G de 2.67 mais le générateur semble mieux converger que le discriminateur. Le discriminateur a plus de mal à bien converger avec cette taille de vecteur n_z cependant sa moyenne de loss est relativement proche des configurations précédentes.

Génération d'images correspondant à des interpolations linéaires. Pour deux vecteurs latents \mathbf{z}_1 et \mathbf{z}_2 on génère des images à partir des interpolations linéaires $\alpha\mathbf{z}_1 + (1 - \alpha)\mathbf{z}_2, \alpha \in [0, 1]$.
 $\alpha = 0.3$.

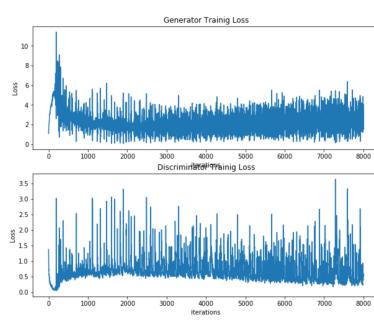


FIGURE 24 – Loss générateur et discriminateur pour un $\alpha = 0.3$.



FIGURE 25 – Exemple de visages générés par le modèle à la fin de l'apprentissage pour $\alpha = 0.3$.

$\alpha = 0.8$

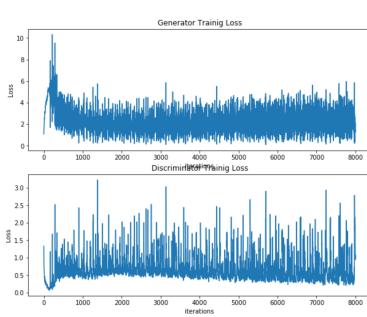


FIGURE 26 – Loss générateur et discriminateur pour un $\alpha = 0.8$.



FIGURE 27 – Exemple de visages générés par le modèle à la fin de l'apprentissage pour $\alpha = 0.8$.

Les images générées en utilisant des interpolations linéaires entre deux vecteurs latents \mathbf{z}_1 et \mathbf{z}_2 sont sensiblement pareil que nos résultats avec les para-

mètres par défaut en utilisant un vecteur \mathbf{z} en entrée, que ce soit avec $\alpha = 0.8$ ou $\alpha = 0.3$. Pareil pour les loss de nos générateurs et discriminateurs, on peut peut être voir que les loss des générateurs avec des interpolations entre deux vecteurs sont plus stable.

Dataset CIFAR-10. On essaie maintenant notre modèle 64x64 qu'on utilisera également pour celeba64 sur le dataset CIFAR10. Le jeu de données CIFAR-10 comprend 60000 images couleur 32x32 réparties en 10 classes, avec 6000 images par classe, il y a 50000 images d'entraînement qu'on utilisera. On réadapte la taille de nos images afin de pouvoir utiliser l'architecture en 64x64.



FIGURE 28 – Images générées par le modèle 64x64 sur le dataset CIFAR.

Le modèle génère des images assez satisfaisante, cependant le problème est plus difficile puisqu'on a 10 classes assez différentes (des animaux, des véhicules...) on manque donc parfois de réalisme mais on peut reconnaître certaine forme sur les images générées.

Architecture 64x64. On expérimente maintenant notre modèle sur le dataset Celeba64, qui est le même dataset que celeba cependant avec des images en plus haute résolution.

On adapte notre modèle en ajoutant au générateur et au discriminateur une couche de convolution.

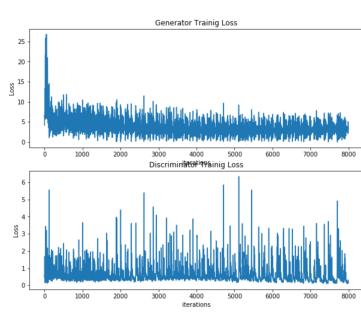


FIGURE 29 – Loss génératrice et discriminatrice pour le dataset celeba64 avec les hyperparamètres par défaut.



FIGURE 30 – Visages générés par le modèle pour le dataset celeba64.

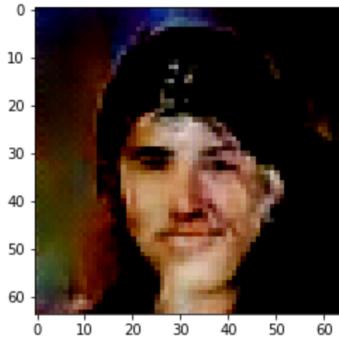


FIGURE 31 – Exemple de visage générée par le modèle en apprenant sur le data-set celeba64.

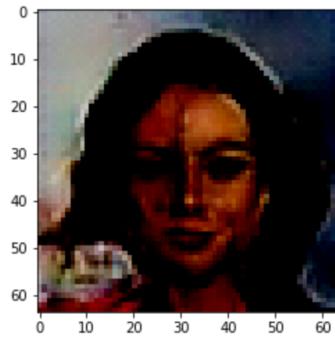


FIGURE 32 – Exemple de visage générée par le modèle en apprenant sur le data-set celeba64.

Le modèle prend plus de temps à générer les images en plus haute résolution 64x64, cependant on obtient des résultats satisfaisants avec des visages plus nets. Les visages générés conservent une bonne symétrie, on observe la même variété de visage que pour le dataset 32x32. La loss du générateur est beaucoup plus stable, avec une moyenne néanmoins un peu plus haute que pour le dataset 32x32.

2 Conditional Adversarial Network

2.1 Principe général.

6) Le nouveau problème que l'on cherche à optimiser en utilisant les équations (2) et (3) est le suivant :

$$\max_D \mathbb{E}_{x^* \in \text{Data}, y \in A} [\log(cD(x^*, y))] + \mathbb{E}_{z \sim P(z), y \in A} [\log(1 - cD(cG(z, y), y))] \quad (5)$$

$$\max_G \mathbb{E}_{z \sim P(z), y \in A} [\log(cD(cG(z, y), y))] \quad (6)$$

7) Le générateur dans l'exemple de la **Figure 6** peut être conditionné aux variables jeune, vieux, homme, femme.

8) Dans l'exemple posé on transforme une vidéo capturée par une voiture en hiver en vidéo qui serait prise en été. Les variables peuvent donc être ici les attributs météorologique : couleur du ciel, présence de nuage, de neiges, feuille sur les arbres...

9) Dans l'exemple ci on a des vidéos de voiture autonome dans un milieu urbain on peut avoir donc comme attribut de conditionnement : Les espaces verts, la présence de piétons, de camions, d'immeubles, de signalisations, des données météorologiques...

2.2 Architecture cDCGAN pour MNIST.

10.) Avec les hyper paramètres de base on obtient bien des résultats satis-

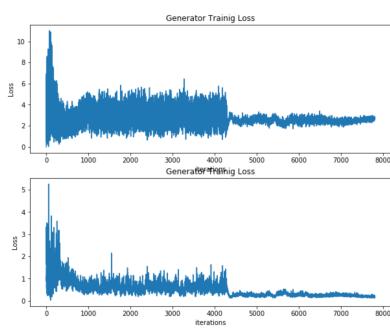


FIGURE 33 – Loss de notre générateur et discriminateur à la fin de l'apprentissage de cDCGAN.

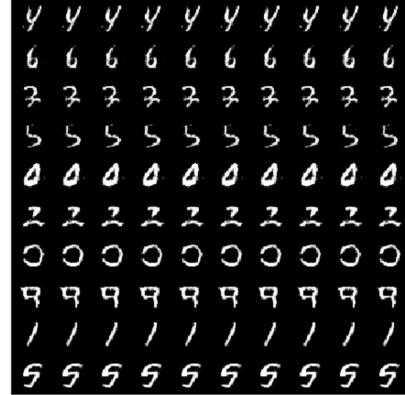


FIGURE 34 – Exemple de nombres générés par cDCGAN à la fin de l'apprentissage de MNIST.

faisant pour la génération de nombre conditionné sur le dataset MNIST. L'apprentissage est assez instable sur les 4000 premières itérations et les courbes sont assez comparable au modèle DCGAN. Après les 4000 itérations nos deux réseaux convergent et se stabilisent pour donner des résultats beaucoup plus nettes et vraisemblable. On voit sur la figure 35 que les nombres générés sont assez indistinguables dans l'ensemble.

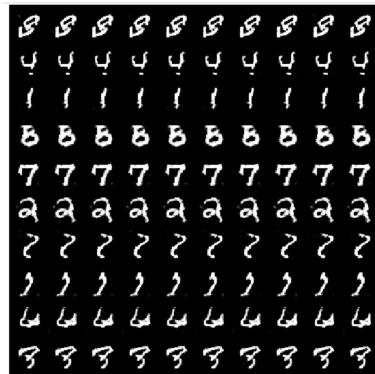


FIGURE 35 – Les nombres générés par notre modèle à la 4000ème itérations un peu avant que les loss se stabilisent.

11.) Dans notre cas le vecteur d'entrée \mathbf{y} est un vecteur one-hot de taille 10 déterminant à quel chiffre on se conditionne pour la génération, $cD(\mathbf{x}, \mathbf{y})$ détermine si la donnée conditionnée en entrée est réelle ou non.

On pourrait se passer de donner le vecteur \mathbf{y} en entrée de cD en construisant le vecteur y à l'aide d'une branche classifieur dans cD qui retourne un softmax de taille 10 et que l'on transforme en vecteur one-hot, ce qui reviendrait à avoir un vecteur \mathbf{y} . Ensuite à l'aide de ce vecteur construit dans cD on l'utilise de la même manière pour déterminer si \mathbf{x} conditionné à \mathbf{y} est une donnée réelle ou non.

2.3 Architecture cGAN pour MNIST.

12.) Avec les hyper-paramètres de base, on obtient des résultats satisfaisants

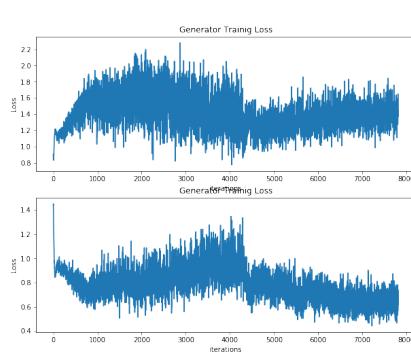


FIGURE 36 – Loss de notre générateur et discriminateur à la fin de l'apprentissage de cGAN.

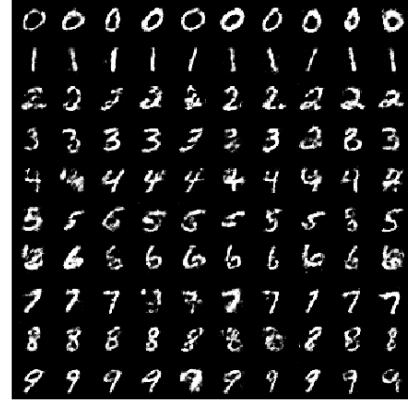


FIGURE 37 – Exemple de nombres générés par cGAN à la fin de l'apprentissage de MNIST.

pour la génération de nombre conditionné sur le dataset MNIST. L'apprentissage est assez instable. On voit sur la figure 37 que les résultats obtenus sont beaucoup moins régulier (pour un même chiffre) que les résultats du cDCGAN où tous les résultats d'un même chiffre semble égaux. Cette diversité de résultat peut expliquer l'instabilité des deux loss. En partant de ce principe, on peut se demander si l'on obtient pas une forme de stabilité à partir de 4000 itérations comme pour le cDCGAN.

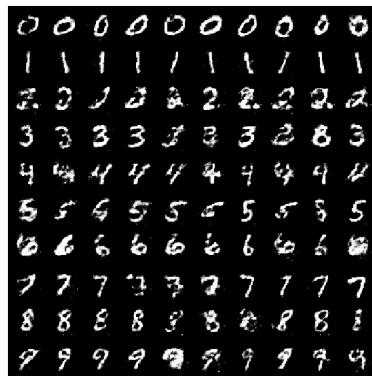


FIGURE 38 – Les nombres générés par le cGAN à la 4000ème itérations

On peut ici remarquer quelques irrégularités pour le cGAN (pixels blancs n'appartenant pas au chiffre) qui n'apparaissent pas pour le cDCGAN.

13.) Il est relativement plus difficile de générer des nombres conditionnées à partir d'un cGAN qu'avec un cDCGAN car la notion d'espace est plus forte dans cDCGAN que dans cGAN. En effet, la première convolution transposée sur le vecteur y du cDCGAN permet d'avoir 256 filtres à partir du vecteur y en créant une notion d'espace. En revanche, sur le cGAN, nous avons un *fully-connected* sur le vecteur y ne créant pas de notion d'espace. Cela peut expliquer, dans un premier temps, la non-uniformité des résultats pour le cGAN au contraire du cDCGAN et, dans un second temps, la difficulté du cGAN à créer des nombres conditionnés.