Ecole Publique d'Ingénieurs en 3 ans

Réalisation d'un Réseau Antagoniste Génératif

Tuteur: Loïck Lhote
loick.lhote@unicaen.fr

BROOD Sarah
DAVID Adrien
PINSON Kévin

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Présentation du projet

Dans le cadre de notre projet de 2ème année, nous avons décidé de découvrir les réseaux antagonistes génératifs (que nous nommerons ici GAN). Dans un premier temps il était obligatoire de faire des recherches sur le fonctionnement de ces réseaux, de leur histoire, diversité et de leurs limites, ce que nous avons fait dans la première partie de ce projet.

Ici nous nous intéressons à la partie pratique de notre projet. Notre objectif est de créer un réseau antagoniste génératif permettant de créer des images de chiffres manuscrits en se basant sur la base de données très connue MNIST.

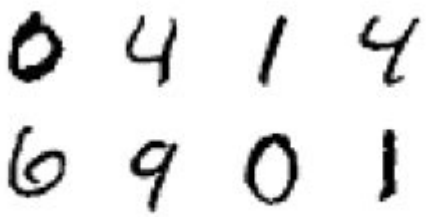

Figure 1 - Exemple de données de la base MNIST

Notre implémentation se trouve en ligne à cette adresse : gan-mnist.

Technologies utilisées

Nous avons choisi le langage Python car il est simple à utiliser et qu'il offre beaucoup d'outils et une variété de librairies conçues pour nous faciliter la vie. De plus ce langage est facile à prendre en main en particulier pour des personnes ayant déjà de l'expérience dans divers langages.

Lors de nos recherches, la grande majorité des exemples d'implémentation de GAN était faite en Python et notamment en utilisant la librairie TensorFlow et son module Keras. Cette librairie gratuite et open source est une référence pour faire de l'intelligence artificielle. Elle est utilisée par Google, Apple ou encore NVIDIA.

Nous avons pu utiliser nos ordinateurs personnels (plus particulièrement un, avec une carte graphique paramétrée pour utiliser Tensorflow) pour entraîner le réseau avec des images de taille réduite (28x28 pixels).

Pour finir nous avons décidé de ne pas expliquer les temps d'exécution du programme car ils sont trop dépendants de l'ordinateur sur lequel le programme est exécuté.

PRÉSENTATION DE NOTRE PROGRAMME

Le type de réseau antagoniste génératif que nous avons implémenté est un DCGAN : un réseau profond convolutif antagoniste génératif. Ces réseaux ont été décrits dans l'article Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks par Alec Radford et Luke Metz. C'est un GAN qui utilise des réseaux convolutifs profonds.

On rappelle qu'un réseau antagoniste génératif met en compétition deux réseaux de neurones. Le premier s'appelle discriminateur et a pour rôle de faire la différence entre les données créées et les données réelles (celles de la base de données MNIST dans notre cas). Le réseau générateur crée des images en essayant de duper le discriminateur et de les faire passer pour des données réelles. Les deux réseaux adaptent leurs pondérations internes grâce à un feedback adapté à chacune de leurs missions.

On peut décrire le fonctionnement de cet algorithme avec le schéma de la figure 2 (source).

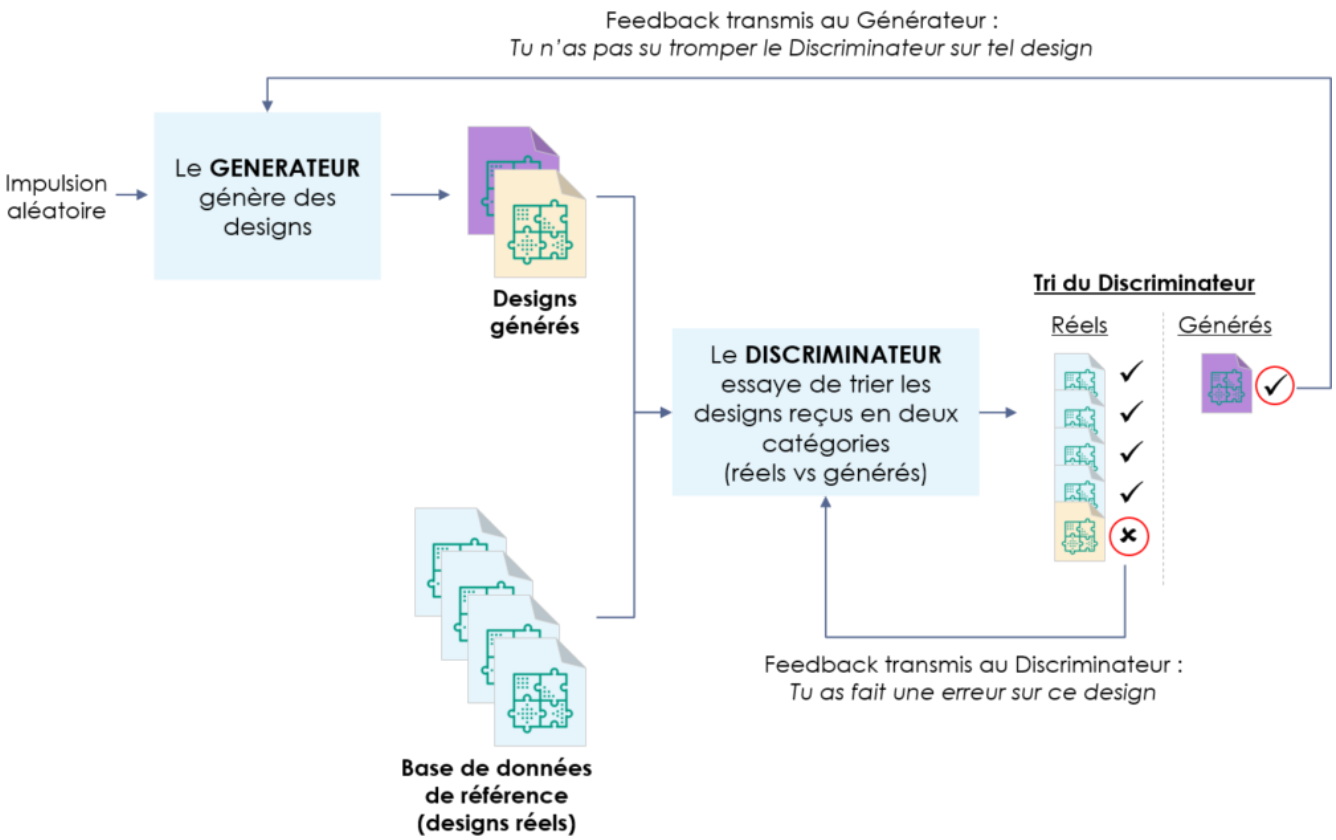

Figure 2 - Schéma du fonctionnement d'un GAN

Construction des différentes parties

Ici nous expliquerons d'abord la dernière implémentation que nous avons retenue pour dans un second temps revenir sur nos difficultés d'implémentation.

1. Discriminateur

Le discriminateur est un réseau de neurones convolutif qui doit répondre *vrai* si une entrée est un chiffre manuscrit. Son but est de distinguer les chiffres générés par le générateur (les "fausses images") des images originales (celles de la base de données MNIST).

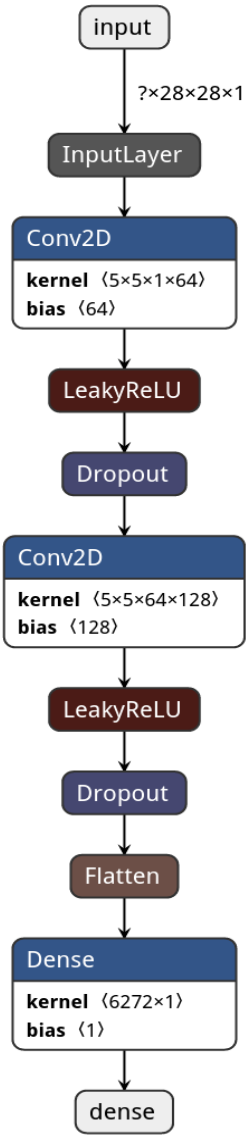

Figure 3

Le discriminateur est représenté ici à gauche dans la figure 3.

Dans la pratique, le réseau prend en paramètre une image d'une dimension 28x28.

Premièrement, une couche convolution (Conv2D) est appliquée à l'image. Cette couche effectue un filtrage par convolution. On calcule le produit de convolution entre un filtre et une portion de l'image, en balayant toute l'image. Le filtre est en fait un des "concepts" de l'image, c'est le réseau qui le définit lui-même car en réalité ce sont ses neurones (pour une image de chien un concept pourrait être le nez, les oreilles ...).

Pour schématiser nous pouvons prendre la représentation ci-dessous et dire par rapport à notre implémentation qu'ici les paramètres sont tels que l'on demande à ce qu'il y ait 64 filtres différents (les cercles sur le schéma), et qu'il faut utiliser des portions d'images de 5x5 pixels (le carré sur l'image).

L'ajout de biais permet au réseau de pouvoir décaler sa fonction d'activation indépendamment des données reçues et de ce fait d'être plus précis dans ces prédictions. L'utilisation de biais est utile voire indispensable sur les petits réseaux comme c'est le cas pour le discriminateur.

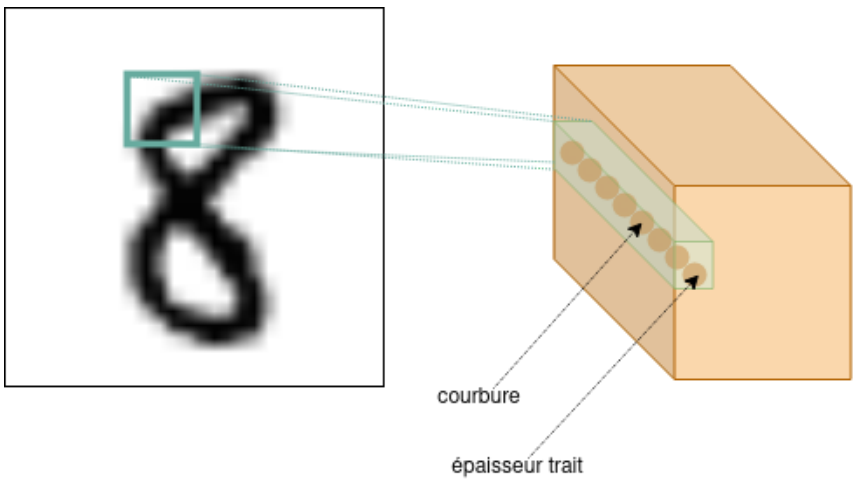

Figure 4 - Schématisation de la convolution

Nous utilisons ensuite une couche dite de correction (LeakyReLU). Cette couche exagère la pondération de chaque neurone ce qui permet d'améliorer l'efficacité du traitement. Nous pouvons la définir comme suit :

$$f(x) = \begin{Bmatrix} x & si\ x > 0 \\ 0.03x & sinon \end{Bmatrix}$$

Une couche de *dropout* est alors ajoutée. Ici *dropout* ne signifie pas décrochage mais plutôt désactivation. On joue à la roulette russe et on désactive pendant une itération d'apprentissage aléatoirement des sorties de neurones avec une probabilité (ici de 0.3). Cette action permet d'améliorer les performances de l'algorithme et sa précision. Nous l'utilisons aussi pour éviter un potentiel sur-apprentissage.

Ces 3 couches sont répétées une nouvelle fois. Seule la taille de la couche de convolution change et change ainsi le nombre de filtres de 64 à 128.

La couche pour aplatir (Flatten) permet de transformer le volume 3D disponible en un vecteur 1D. Nous pouvons observer sur la figure 7 que le cube de 7 par 7 par 128 est transformé en un vecteur colonne de 6272, ce qui facilitera le travail de la couche suivante. La figure 5 représente l'application de la couche pour un petit volume.

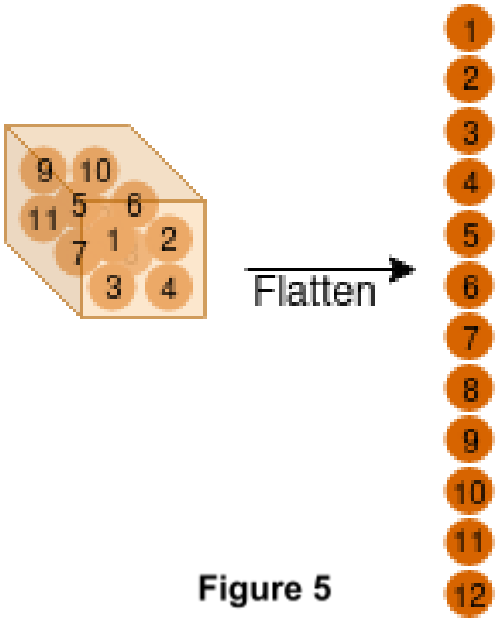

Figure 5

La couche Dense permet d'obtenir une valeur de retour de taille 1 donc un booléen à partir d'un vecteur de 6272 valeurs. Cette couche est "très fortement connectée" car chaque neurone en sortie est connecté à tous les précédents. Dans notre cas, le neurone de sortie est connecté aux 6272 précédents. La figure 6 représente l'application de la couche Dense pour un petit vecteur.

Cette couche utilise la fonction sigmoid qui retourne un booléen. La fonction est définie comme suit :

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

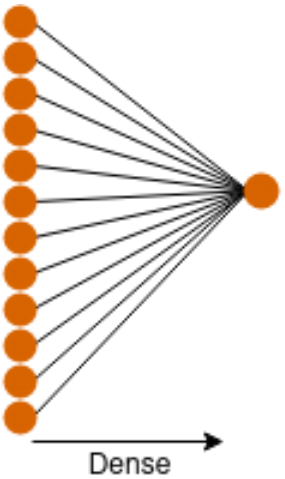

Figure 6

Il est possible de schématiser les transformations appliquées grâce à la figure 7. Il est possible alors de bien représenter le volume correspondant aux neurones présents.

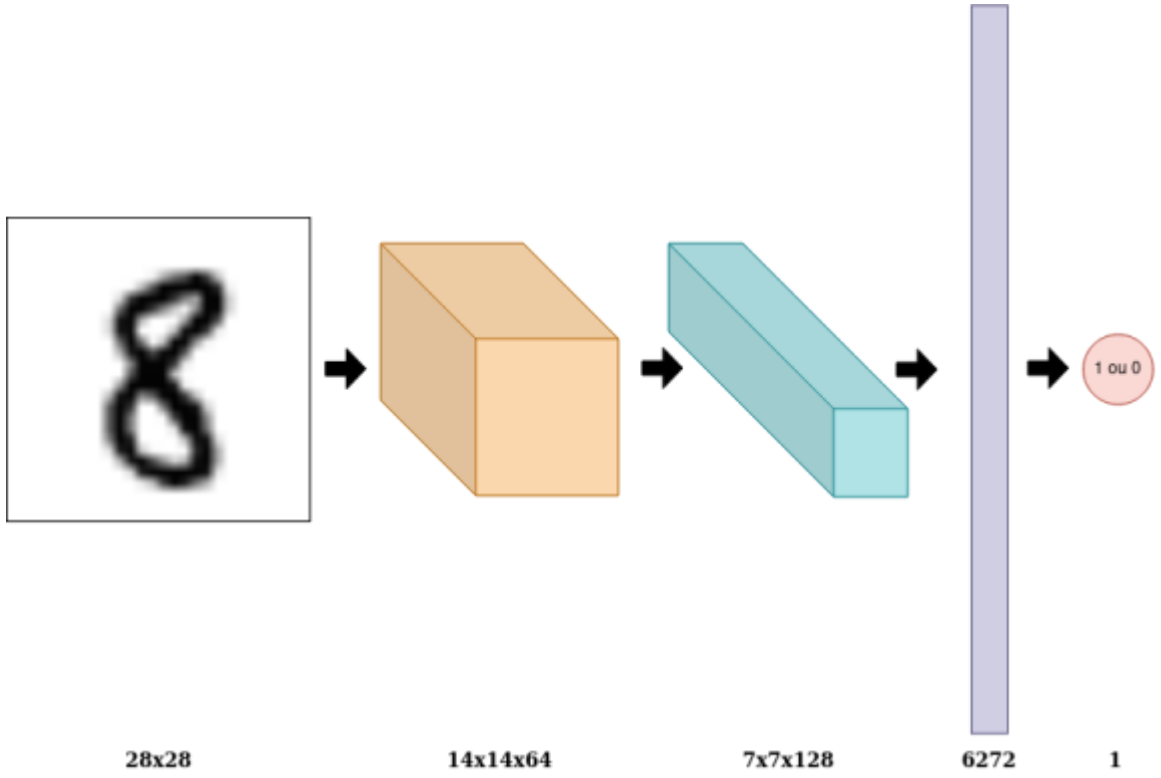

Figure 7 - Schéma du réseau discriminateur

Nous avons choisi d'utiliser ces fonctions, valeurs et enchaînements de couches grâce à nos recherches préalables. Contrairement au générateur, ce réseau a une représentation simple et bien logique que nous avons pu comprendre dès le départ, une forme d'entonnoir à modéliser. C'est en jouant avec les différents paramètres durant de nombreux essais et à des recherches plus poussées sur chaque type de couche que nous avons pu trouver (à notre sens) les meilleurs.

2. Générateur

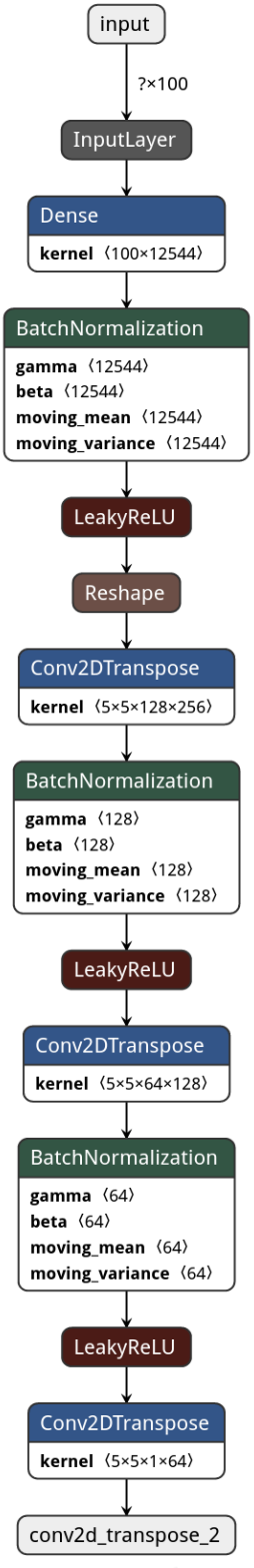

Figure 8

Le générateur a un seul objectif : duper le discriminateur et donc générer des images semblables à celles présentes dans la base de données MNIST. Pour se faire il prend en entrée du bruit pour finalement produire une image d'une dimension 28x28. Comme son adversaire, il s'agit aussi d'un réseau de neurones convolutif.

Le générateur est représenté à gauche dans la figure numéro 8.

Pour générer les données d'entrée du générateur, nous créons des vecteurs de bruit définis par une distribution normale. Nous en créons autant que la taille d'un batch (défini plus loin).

Nous allons ici faire presque le chemin inverse par rapport à ce qui a été fait avec le discriminateur. Tout d'abord nous avons une couche Dense qui permet de générer à partir du bruit, un vecteur de 12544 neurones. Cette couche permet à chaque neurone de se baser sur tous les neurones précédents. Nous obtenons alors encore du bruit mais avec la dimension que nous désirons. Ici et dans toutes les couches suivantes, il n'est pas pertinent d'utiliser un biais car il est important de garder les pixels blancs.

Une couche de normalisation est alors utilisée. En effet, avec la superposition des couches on peut observer un certain décalage. Cette couche permet d'améliorer la coordination entre les couches. Elle effectue un simple redimensionnement des données pour qu'elles aient une moyenne à 0 et un écart-type de 1. Pour les images, cela permet d'éviter un effet néfaste sur les résultats à cause du décalage. Pour finir, celle-ci permet d'améliorer considérablement les résultats de l'algorithme en stabilisant et accélérant la formation du réseau.

Une couche de correction (présentée plus haut) est ajoutée. Ces deux couches sont placées tout du long du réseau pour qu'il soit davantage rapide.

Ici nous avons encore un réseau de neurones "plat". Or pour pouvoir avoir des informations sur les concepts de l'image nous avons besoin de volume. C'est pour cela qu'une couche pour redimensionner (Reshape) est utilisée, elle permet de redimensionner à la dimension souhaitée notre réseau. Ici la taille qui est demandée n'est pas aléatoire, en effet on a 7 * 7 * 256 = 12544.

On utilise alors une couche de déconvolution (Conv2DTranspose). Cette opération est l'inverse de la convolution. Nous l'avons illustré dans le schéma de la figure 9.

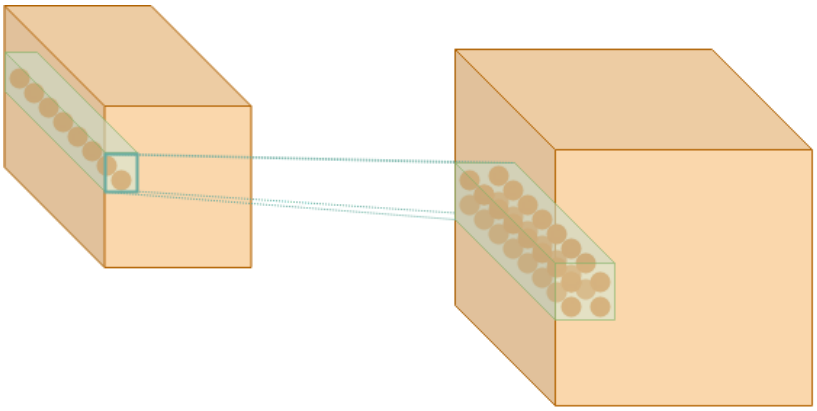

Figure 9 - Schéma de la déconvolution

Les trois couches précédentes sont répétées deux fois hormis la couche de redimensionnement car nous avons déjà un volume grâce à la première couche. Le but ici est d'obtenir la forme de l'image. On diminue alors la profondeur pour augmenter les autres dimensions.

Finalement la dernière couche de déconvolution permet d'obtenir les mêmes dimensions qu'une image MNIST. On utilise sur cette dernière couche une fonction d'activation, la tangente hyperbolique qui a la forme suivante :

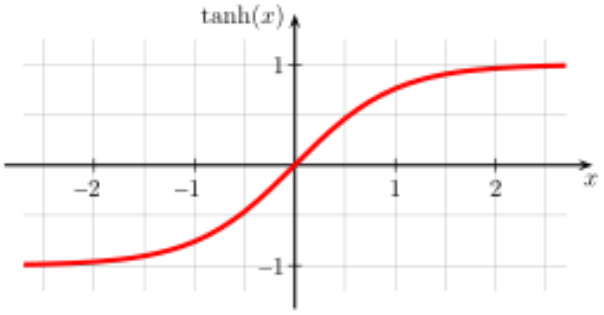

Figure 10 - Courbe de la tangente hyperbolique

La fonction *sigmoid* pourrait être aussi pertinente car elle existe entre 0 et 1. Et c'est ce type de sortie que nous cherchons car les neurones portent une probabilité mais la fonction tangente est plus efficace.

Nous avons schématisé les transformations appliquées grâce à la figure 11.

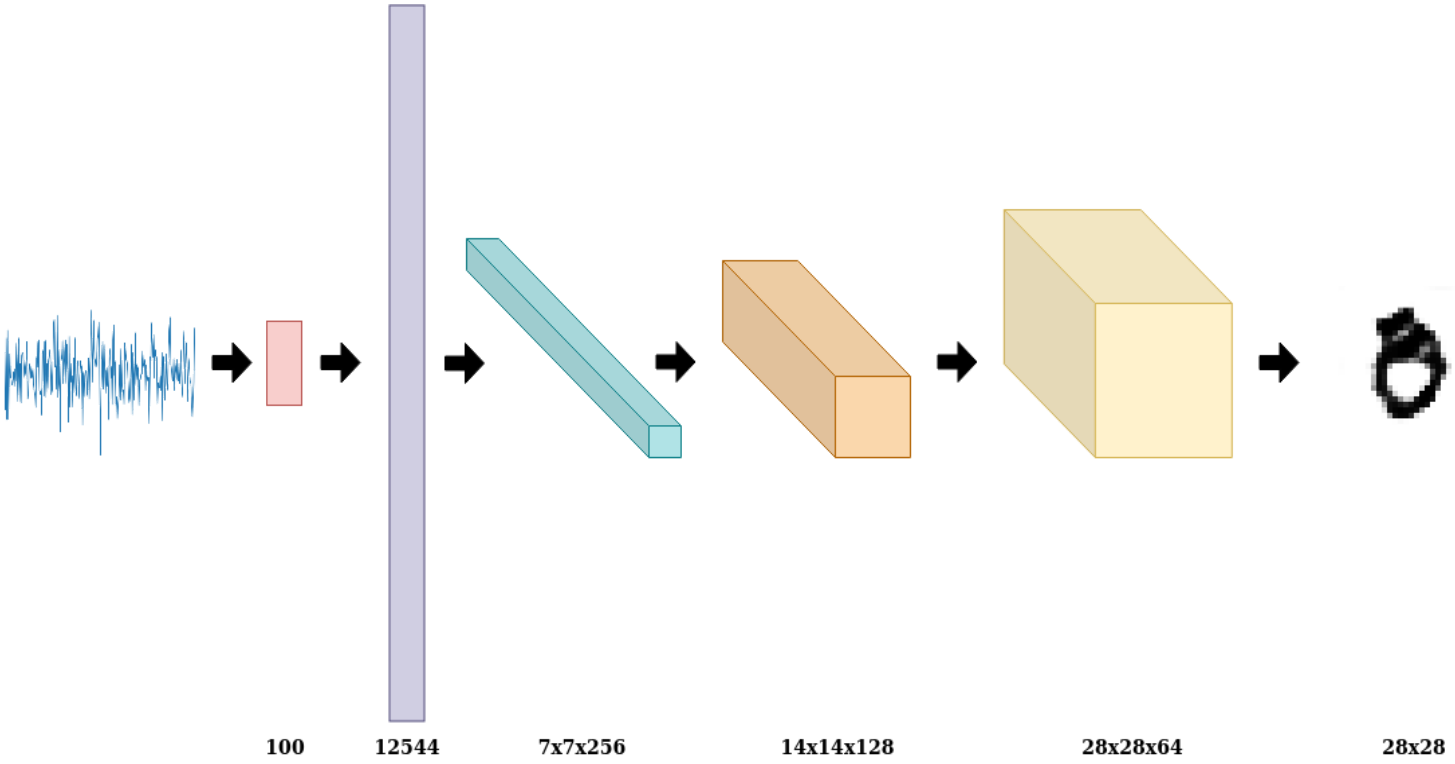

Figure 11 - Schéma du réseau générateur

La construction du générateur a été plutôt compliquée pour nous car il nous semblait dans un premier temps qu'il était strictement l'inverse du discriminateur. Or il est bien plus complexe que cela et les premières couches finalement utilisées sont loin de ce que nous imaginions. Nous avons eu des difficultés à comprendre la relation entre les tailles des volumes et des vecteurs, et le fonctionnement des convolutions. Après diverses recherches et en étudiant l'état de l'art nous avons pensé que cette forme était la meilleure.

Des couches comme celles de normalisation ne nous paraissent pas utiles dans un premier lieu alors qu'elles sont très importantes dans l'optimisation du réseau. En effet, sans les couches de normalisation, le réseau est bien plus lent à "comprendre" le type d'image que nous cherchons à créer et les formes sont bien plus floues. Elles permettent aussi de réduire considérablement le nombre d'*epochs* nécessaires pour l'entraînement du réseau. Vous pourrez comparer les résultats enregistrés d'un générateur sans normalisation et un avec dans la partie résultats de ce rapport.

3. Lien entre les différentes parties

Pour entraîner un réseau, il faut lui donner en paramètre des données. Ces données vont passer plusieurs fois dans le réseau. Quand toutes les données sont passées une fois : une *epoch* a été effectuée. Notre boucle principale se base sur le nombre *d'epochs* que l'on souhaite réaliser au total. Au sein d'une *epoch,* les données sont divisées en *batch.*

À l'intérieur d'un *batch*, le générateur va créer autant d'images qu'il y en a dans un *batch*. Puis le discriminateur va effectuer une prédiction pour chaque image du batch ainsi que pour chaque image créée par le générateur. Les décisions concernant le générateur vont passer dans une fonction qui va le récompenser en se basant sur sa capacité à duper le discriminateur. Puis, les décisions concernant le discriminateur vont elles aussi passer par une fonction qui va le récompenser pour sa capacité à identifier les vraies données, mais aussi à identifier les fausses données. À partir de ces résultats, les réseaux mettent à jour le poids de leurs neurones.

On se retrouve donc avec un code proche de celui-ci :

```
for epoch in MAX_EPOCH:
  for batch in dataset:
      noise = random([BATCH_SIZE])
      fake_images = generator(noise)
      fake_decision = discriminator(fake_images)
      real_decision = discriminator(batch)
      loss_discriminator = discriminator_loss(real_decision,fake_decision)
      loss_generator = generator_loss(fake_decision)
      discriminator_optimizer.apply_gradients(discriminator,loss_discriminator)
      generator_optimizer.apply_gradients(generator,loss_generator)
```

On répète l'opération tant qu'on ne trouve pas que le générateur est conforme à nos attentes.

4. Paramètres

Au cours du développement du GAN, nous avons dû choisir un ensemble d'éléments permettant le fonctionnement de ce dernier. Il a fallu entre autres choisir quelle fonction utiliser pour diriger le générateur et le discriminateur. Ce choix a été le même pour les deux réseaux et fut la fonction *cross_entropy* de *TensorFlow*. Cette fonction est dessinée sous le schéma suivant.

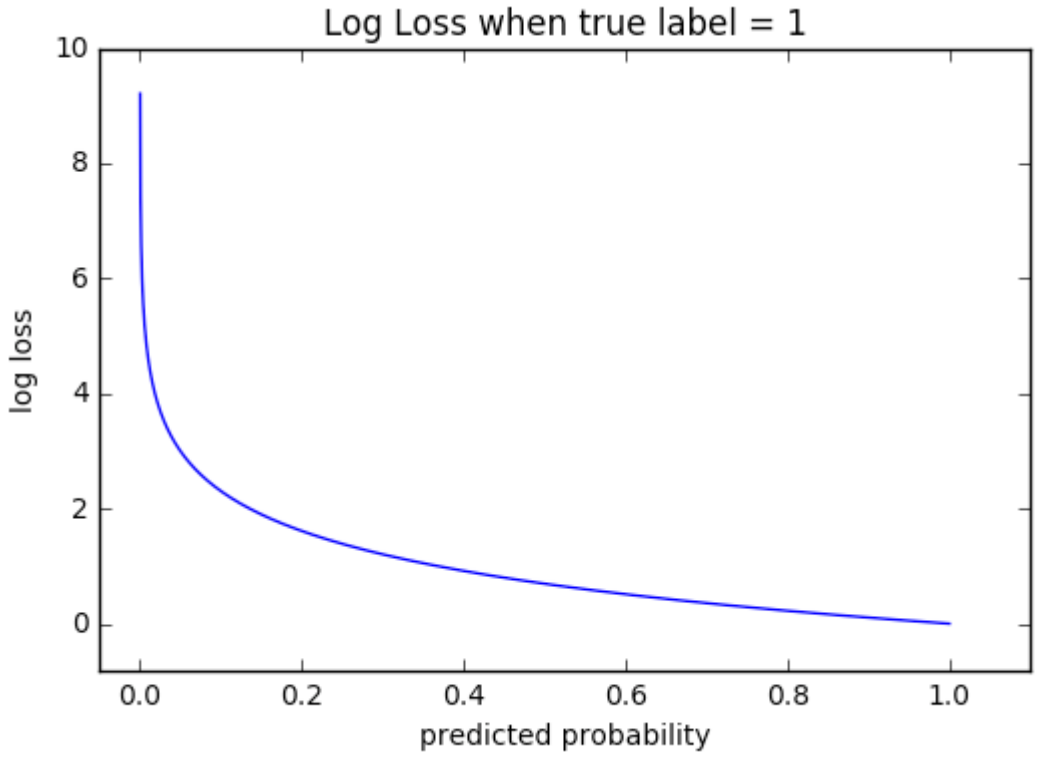

Figure 12 - Fonction cross_entropy

Prenons le cas d'une image que le générateur a produit. Le discriminateur va donc retourner une valeur entre 0 et 1, pour 1 l'image est vraie (le générateur a donc réussi) et pour 0 l'image est fausse (le générateur a échoué). Par la suite, le générateur va récupérer le résultat du discriminateur. Lorsque le résultat va passer dans la fonction, s'il est proche de 1, il aura alors une perte (*loss*) faible et va modifier par la suite que de très peu le poids de ses neurones. Dans le cas inverse, il aura une perte très élevée et va modifier de façon plus conséquente son réseau. Le même concept est utilisé pour le discriminateur. L'avantage de cette fonction est que si le réseau n'est pas du tout dans la bonne direction alors des grands changements vont être apportés. Tandis que dans le cas inverse, le réseau va aller petit à petit et pouvoir ainsi être le plus précis possible.

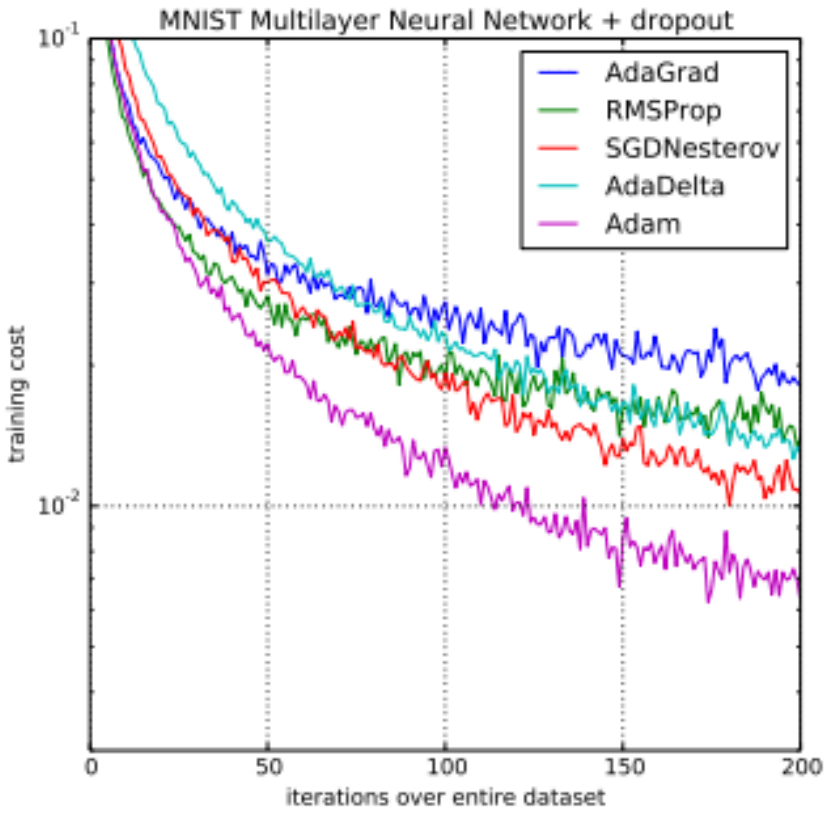

Figure 13 - Comparaison des différentes fonctions

Ce ne fut cependant pas le seul choix que nous avons dû réaliser. En effet, il faut aussi choisir quelle fonction utiliser pour la descente de gradient. Nous avons opté ici pour la même fonction dans les deux réseaux qui est la fonction Adam.

Nous l'avons choisi car il semble avoir les meilleures performances, en particulier lorsque l'on fait du training sur la base de données MNIST. Les résultats de la figure 13 proviennent de : *Adam: A Method for Stochastic Optimization, 2015.*

Entrainement du gan

1. Temps d'entrainements

Le temps d'entraînement de la dernière version de notre algorithme lancé avec Tensorflow paramétré pour utiliser une carte graphique (GeForce GTX 1050 Ti dans notre cas), est d'environ 40 minutes.

Sans les couches de normalisation nous avons besoin d'environ 1 heure, et avec les premières configurations de notre algorithme nous n'attendions pas la fin des epochs car le temps d'entraînement était doublé voire triplé, et nos résultats n'étaient pas bons.

2. Difficultés rencontrées

Une des principales difficultés a été l'utilisation de TensorFlow. En effet, l'installation et la mise en œuvre n'ont pas été très aisées. En particulier pour la descente de gradient et la sauvegarde des réseaux dans des fichiers. De plus, nous avons trouvé que c'était globalement plus dur de debugger le programme, surtout quand cela concerne les réseaux qui paraissent parfois comme des boîtes noires.

Une autre difficulté a été les temps d'entraînement. Un d'entre nous possédait une carte graphique ce qui a été d'une très grande utilité pour l'entraînement. Cependant pour ceux qui n'en avaient pas les calculs semblaient durer une éternité et il était très difficile de s'assurer que ce que nous avions fait était réellement utile.

La dernière difficulté a été le paramétrage des réseaux. Certaines fonctions de TensorFlow semblaient assez floues sur leur fonctionnement et le nombre de paramètres possibles pour chaque fonction rendait la création des réseaux très difficile.

Résultats

Nous avons pu exécuter plusieurs fois l'algorithme, pour trouver les bons paramètres mais aussi tout simplement pour se rendre compte que notre premier algorithme ne fonctionnait absolument pas, et donc avoir des résultats différents.

Les trois figures suivantes représentent un tableau des images générées pour une exécution du programme.

Ratés

Au début du développement, comme dit précédemment, nous ne laissions pas le programme se terminer naturellement car il était trop long à s'exécuter.

Nous ne reconnaissions même pas de chiffres comme on peut le constater sur la figure 14.

La forme de nos réseaux et le lien n'était pas encore très adaptés.

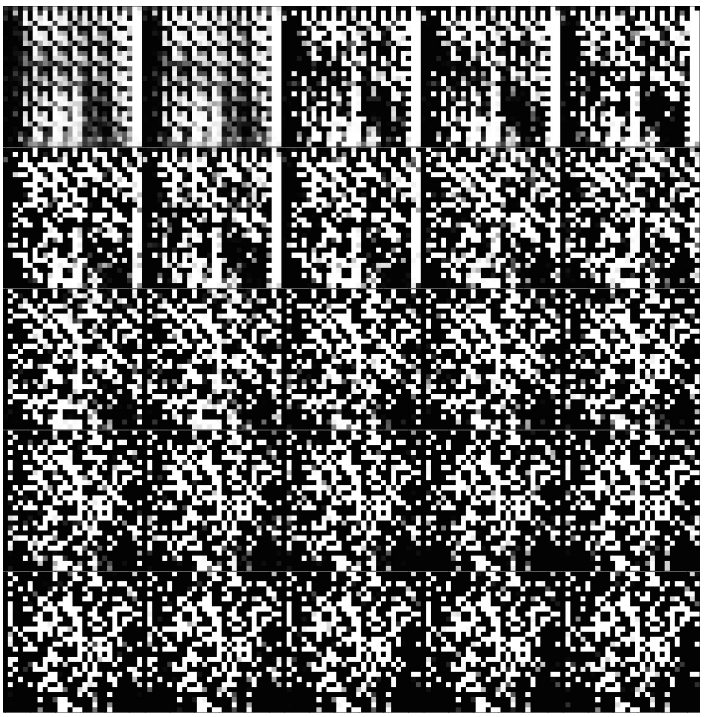

Figure 14

Sans Batch Normalization

Un de nos premiers programmes qui fonctionnait ne comportait pas de couche de normalisation dans le réseau générateur.

Comme expliqué plus haut, cette couche permet d'augmenter drastiquement les résultats de l'algorithme.

Les figures 15 et 16 se lisent de gauche à droite et de haut en bas et représentent un exemple de ce que le générateur a pu produire à une *epoch* donnée. Dans les deux cas, il y a 2 epochs de différence entre chaque image. Ainsi l'image dans la première ligne de la première colonne présente un échantillon à la fin de l'*epoch* 1, celle sur la deuxième colonne de la première ligne l'*epoch* 3, et ainsi de suite.

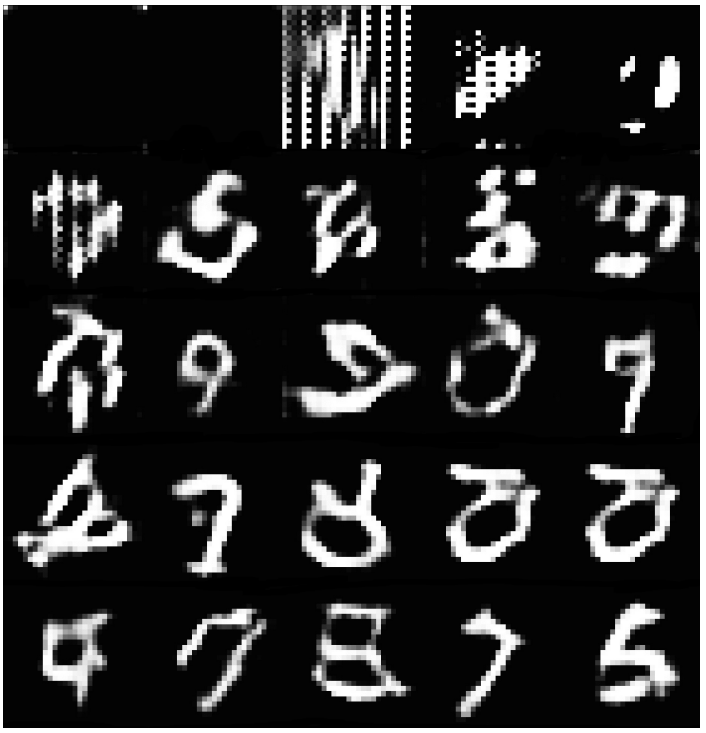

Figure 15

Ici en comparant avec les résultats définitifs on peut voir que les résultats mettent plus longtemps à devenir cohérents. De plus, les meilleurs résultats sans normalisation semblent moins bons que les meilleurs résultats qui en possèdent. Il y aussi une impression de stagnation de l'évolution.

Résultats définitifs

Nous avons décidé de garder comme résultats définitifs les résultats suivants. Nous pensons qu'ils ne sont pas parfaits mais les images MNIST étant de dimensions 28x28 pixels nous ne pouvions pas avoir des images très nettes.

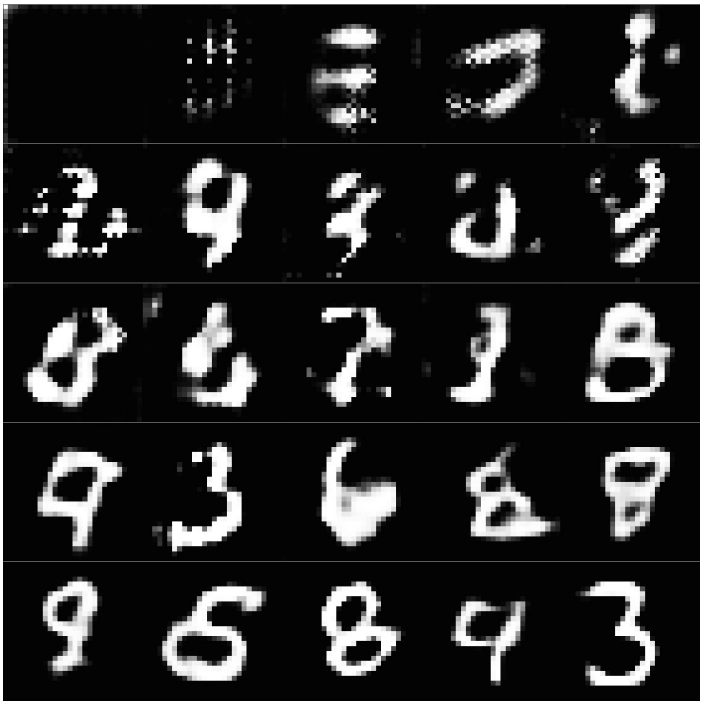

Figure 16

Conclusion

Après bon nombre de péripéties, nous avons donc un GAN MNIST fonctionnel. Ce projet nous a permis de découvrir en profondeur le fonctionnement des GAN, mais aussi de manipuler la librairie TensorFlow. Il a également permis de comprendre une multitude de choses concernant le deep learning et les réseaux de neurones à convolution. Notamment le concept d'*epoch*, de *batch* et la variété des paramètres possibles afin de créer des réseaux. Nous sommes très contents d'avoir réussi à implémenter un GAN et sommes aussi satisfaits des chiffres qu'il génère.

Bibliographie

- Notre dernier rapports et ses sources
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks – Alec Radford & Luke Metz
- Rectifier (neural networks) – Wikipédia
- Bias Vector – DeepAI
- Keras CONV2D Class – Geeks for Geeks
- Documentation Python de Tensorflow
- Produit de convolution – Wikipédia
- https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#Leaky_ReLU
- Documentation de l'API de Keras – Activations
- A Gentle Introduction to Batch Normalization for Deep Neural Networks – Machine Learning Mastery
- Gentle Introduction to the Adam Optimization Algorithm for Deep Learning – Machine Learning Mastery
- Loss Function – ML CheatSheet
- Epoch – DeepAI
- Batch Size – Radiopaedia
- Netron par Luzroeder (schéma vertical des réseaux)
- DiagramEditor (outils pour les schémas)


ENSI
CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE