h e g

Haute école de gestion
Genève

Etude et application des réseaux adverses génératifs à la conversion de voix

Travail de Bachelor réalisé en vue de l'obtention du Bachelor HES

par :

Frédéric CHARBONNIER

Conseiller au travail de Bachelor :

Jean-Philippe TRABICHET, professeur HES – chef du département

Genève, le 31 mai 2021

Haute École de Gestion de Genève (HEG-GE)

Filière Informatique de Gestion

Hes·so GENÈVE
Haute Ecole Spécialisée de Suisse occidentale

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre de Bachelor of Science en Informatique de Gestion.

L'étudiant a envoyé ce document par email à l'adresse remise par son conseiller au travail de Bachelor pour analyse par le logiciel de détection de plagiat URKUND, selon la procédure détaillée à l'URL suivante : https://www.urkund.com .

L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Genève, le 31 mai 2021

Frédéric Charbonnier

Remerciements

Je dédie ce travail de Bachelor à ma fille Clara et à ma compagne Jenny.

Un immense merci du fond du cœur à ma famille, mes proches, mes collègues et mon conseiller au travail de Bachelor pour leur soutien et leur confiance.

Un clin d'œil à mon père qui me regarde de là-haut…

Résumé

L'objectif de ce travail de Bachelor est de survoler le fonctionnement des réseaux adverses génératifs, du signal vocal et de la conversion de voix pour tenter de mettre en place une solution permettant de convertir le timbre d'une voix A en une voix B tout en gardant le contenu du message initial.

Pour comprendre intuitivement l'approche réalisée pour atteindre cet objectif, ce document débute par un survol des concepts du deep learning, des réseaux adverses génératifs, du signal vocal ainsi que de la conversion de voix.

Des cas pratiques seront ensuite réalisés dans le cadre d'une démarche évolutive, dans laquelle je commencerai par développer et entrainer un réseau adverse génératif simple permettant de générer des points dans un espace donné. Puis, je ferai évoluer ce réseau pour générer des chiffres écrits à la main et enfin je le modifierai pour réaliser des conversions de voix.

Enfin, je testerai Google Voice Match avec les voix converties pour observer le comportement d'un système spécialisé en reconnaissance vocale avec des échantillons générés à l'aide d'un réseau adverse génératif.

Le code source développé pour les cas pratiques est disponible sur mon dépôt GitHub : https://github.com/charbonnier-fred/GAN-TB-2021-HEG

Table des matières

Liste des figures

1. Introduction

Le domaine de l'intelligence artificielle rassemble toutes les techniques permettant à des ordinateurs de simuler et de reproduire l'intelligence humaine (Roder, 2019).

Les travaux sur le sujet ont débuté dans les années 1940 à 1950 où l'on parlait alors de cybernétique, science modélisant, à l'aide de flux d'informations, les systèmes biologiques, psychiques, politiques et sociaux (Vannieuwenhuyze, 2019).

En 1948, le mathématicien Allan Turing décrivit, dans son article « Intelligent Machinery », des réseaux de neurones connectés aléatoirement capables de s'auto-organiser. Puis en 1950, dans son article « Computing Machinery and Intelligence », il proposa le populaire « Jeu de l'imitation » permettant de tester les capacités d'une machine à imiter la conversation humaine. Pour que la machine réussisse ce test, il faut qu'à l'issue de quelques jeux de questions et de réponses, la personne ne sache pas si elle a engagé une conversation avec un être humain ou une machine (Vannieuwenhuyze, 2019).

Mais l'intelligence artificielle a officiellement vu le jour en tant que science en 1956 au Dartmouth College (New Hampshire, Etats-Unis) lors d'une université d'été organisée par les scientifiques John McCarthy, Marvin Minsky, Nathaniel Rochester et Claude Shannon (Vannieuwenhuyze, 2019). Cette nouvelle discipline académique suppose que toutes les fonctions cognitives humaines puissent être décrites de façon très précise et donc être reproduites sur ordinateur. Dès lors, il serait possible de créer des systèmes capables d'apprendre, de calculer, de mémoriser, et de réaliser des découvertes scientifiques ou des créations artistiques (Vannieuwenhuyze, 2019).

Depuis, l'intelligence artificielle s'est largement implantée dans notre quotidien, comme par exemple dans la reconnaissance automatique de caractères, dans les filtres anti-spam, elle sert également à protéger contre la fraude bancaire, recommander des livres, des films, identifier les visages sur les réseaux sociaux, traduire automatiquement des textes d'une langue à une autre (Azencott, 2019).

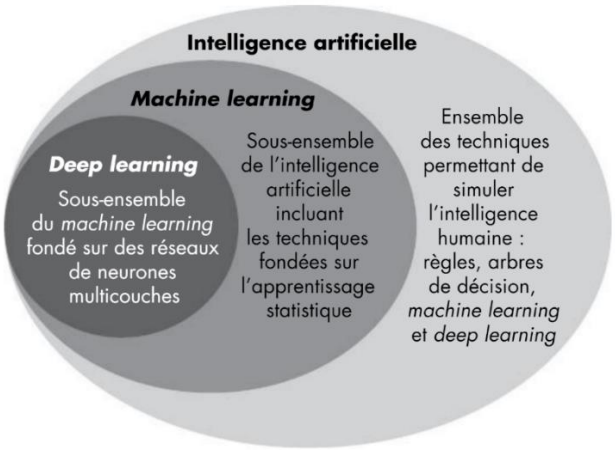
L'intelligence artificielle permet aussi de donner une voix à nos assistants personnels tels que Google Home ou Alexa. Ces mêmes assistants sont également capables d'identifier notre empreinte vocale afin nous fournir des résultats personnalisés, comme nos rendez-vous de la semaine, nos derniers e-mails ou encore de valider une transaction financière.

L'intelligence artificielle a également montré des capacités à manipuler facilement des données aussi bien visuelles qu'auditives, en produisant des deepfakes suffisamment réalistes. On peut désormais synthétiser un visage, modifier ses caractéristiques morphologiques, changer les expressions faciales ou même échanger des visages. Ces mêmes manipulations sont également possibles avec l'audio. On peut synthétiser une voix, modifier son expressivité ou encore échanger des voix. L'hyperréalisme des résultats obtenus inquiète (sécurité, manipulation d'opinion) et de nombreux acteurs tels que Google, Facebook ou Amazon travaillent sur la détection de ces deepfakes. Mais ces capacités peuvent aussi montrer des impacts positifs, comme par exemple redonner la voix à des personnes handicapées ou encore créer du contenu vidéo pédagogique original (en faisant intervenir des personnages historiques, etc…) (Rukubayihunga, 2020).

Dans les années à venir, l'intelligence artificielle nous permettra probablement d'améliorer la sécurité routière (notamment grâce aux véhicules autonomes), de donner une réponse d'urgence aux catastrophes naturelles, de développer de nouveaux médicaments ou encore d'augmenter l'efficacité énergétique de nos bâtiments, de nos industries et de nos villes (Azencott, 2019).

L'intelligence artificielle est composée de deux sous-ensembles. Le premier est appelé le machine learning. Il s'agit de l'utilisation des statistiques pour donner la faculté aux machines d'apprendre. Le second sous-ensemble est appelé deep learning et désigne les algorithmes capables de s'auto-améliorer grâce à des modélisations inspirées du fonctionnement du cerveau humain tels que les réseaux de neurones reposant sur un grand nombre de données (Vannieuwenhuyze, 2019).

Figure 1 : Sous-ensembles de l'intelligence artificielle



Source : Roder, 2019

Actuellement lorsque nous parlons d'intelligence artificielle, il est préférable de parler directement de machine learning ou de deep learning (Vannieuwenhuyze, 2019).

2. Le machine learning

En 1956, Arthur Samuel définit le machine learning comme un « domaine d'étude qui donne aux ordinateurs la capacité d'apprendre sans être explicitement programmé » (Samuel, 1959).

On peut alors différencier un programme « classique » qui reçoit des données en entrée et utilise des procédures spécifiquement programmées pour générer des résultats en sortie, d'un programme de machine learning qui utilise des données en entrées et les résultats obtenus en sortie pour définir des procédures (Azencott, 2019).

Prenons l'exemple d'une société de vente en ligne qui souhaite connaître les 3 produits les plus vendus d'une année terminée. Il suffit d'appliquer un algorithme classique, à savoir une simple addition, il n'est donc pas nécessaire d'utiliser un algorithme d'apprentissage.

Cependant, si cette même société souhaite connaître les 3 produits qui seront probablement les plus vendus de l'année en cours, il n'est alors plus possible d'appliquer une simple addition étant donné que les informations sont incomplètes. Il est alors possible d'utiliser un algorithme de machine learning sur la base de l'historique des ventes effectuées pour générer un modèle prédictif permettant de répondre à ce besoin.

Le machine learning peut servir à résoudre 3 types de problèmes (Azencott, 2019) :

- Ce que l'on ne sait pas (encore) résoudre ;
- Ce que l'on sait résoudre, mais dont on ne se sait pas formaliser en termes algorithmiques comment nous les résolvons (par exemple la reconnaissance d'images, la compréhension du langage naturel) ;
- Ce que l'on sait résoudre, mais avec des procédures beaucoup trop gourmandes en ressources informatiques.

Le machine learning est donc utilisé lorsque les données sont abondantes mais que les connaissances sont peu accessibles ou peu développées (Azencott, 2019).

Les algorithmes d'apprentissage peuvent aussi aider les humains à apprendre sur les données en révélant l'importance relative de certaines informations ou la façon dont elles interagissent entre elles pour résoudre un problème particulier (Azencott, 2019). Par

exemple, à l'aide d'une base de données contenant l'historique de transactions bancaires il serait possible d'identifier les données pertinentes pour la détection de fraudes.

Quels sont les méthodes d'apprentissage qui permettent à une machine d'apprendre à résoudre ces différents problèmes ?

2.1 Apprentissage

L'apprentissage-machine est l'ensemble des méthodes permettant d'entrainer un système, au lieu de le programmer explicitement. Il existe 3 formes d'apprentissages : l'apprentissage supervisé, l'apprentissage par renforcement et l'apprentissage non-supervisé (Azencott, 2019).

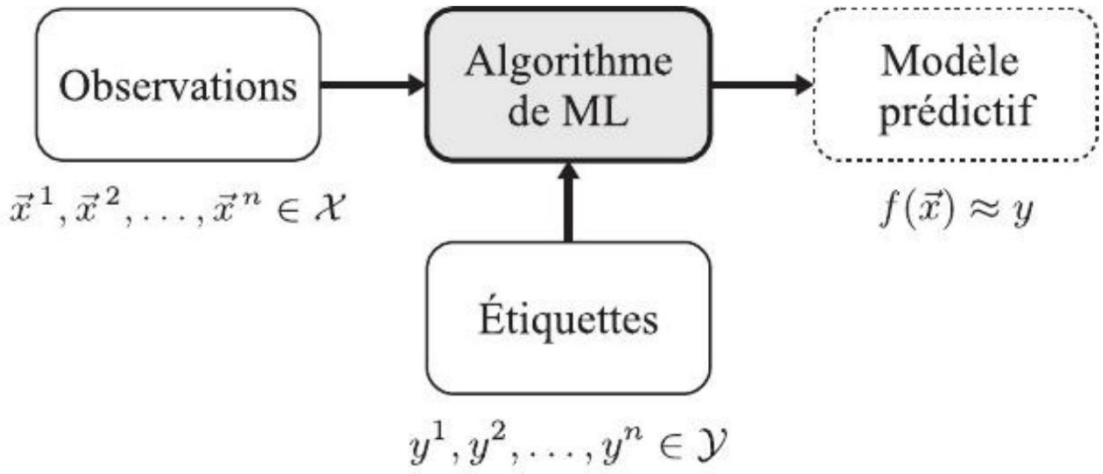
2.1.1 Apprentissage supervisé

L'apprentissage supervisé entraine le système à accomplir une tâche à partir d'exemples d'entrée et de sortie correspondante (Le Cun, 2019).

Stéphane Roder nous apprend que 90% de l'intelligence artificielle de ces dernières années est fondée sur cette technique d'apprentissage (Roder, 2019).

On utilise les exemples du passé (observations et étiquettes) pour en déduire une fonction de prédiction, le modèle prédictif, qui va nous permettre de généraliser notre connaissance pour une utilisation future (Roder, 2019).

Figure 2 : Apprentissage supervisé



Source : Azencott, 2019

Si les étiquettes sont binaires (vrai ou faux), c'est-à-dire qu'elles indiquent l'appartenance à une classe, alors on parle de classification binaire (Azencott, 2019).

Un exemple simple pour illustrer la classification binaire est le contexte d'un filtre anti-spam. Il y a une liste de mails (les observations) et pour chaque mail, l'indication précisant s'il s'agit d'un spam ou non (les étiquettes) qui prendra la valeur 0 ou 1. Le modèle prédictif est alors généré à l'aide des observations et des étiquettes existantes, son objectif sera de classifier les futurs mails comme potentiel spam ou non.

Si les étiquettes sont discrètes, et correspondent donc à plusieurs classes, on parle alors de classification multi-classe (Azencott, 2019). Par exemple, si les observations et les étiquettes correspondent à des images de plantes et le libellé à l'espèce correspondante.

Enfin, si les étiquettes sont à valeurs réelles, on parle alors de régression (Azencott, 2019). Par exemple avec les actions en bourse, la date du jour et leur valeur d'achat.

2.1.2 Apprentissage par renforcement

L'apprentissage par renforcement entraîne le système par interaction avec un environnement par essais et erreurs (Le Cun, 2019). En retour de ces actions, le système obtient une récompense, qui peut être positive si l'action était un bon choix, ou négative dans le cas contraire (Azencott, 2019).

Les applications principales de l'apprentissage par renforcement se trouvent dans les jeux, comme le go ou les échecs, et dans la robotique. L'apprentissage consiste à définir une stratégie permettant d'obtenir systématiquement la meilleure récompense possible (Azencott, 2019).

Par exemple, aux échecs, la machine va simuler elle-même de nombreuses parties pour essayer différents scénarios de jeu en progressant à chaque erreur et à chaque réussite.

2.1.3 Apprentissage non-supervisé

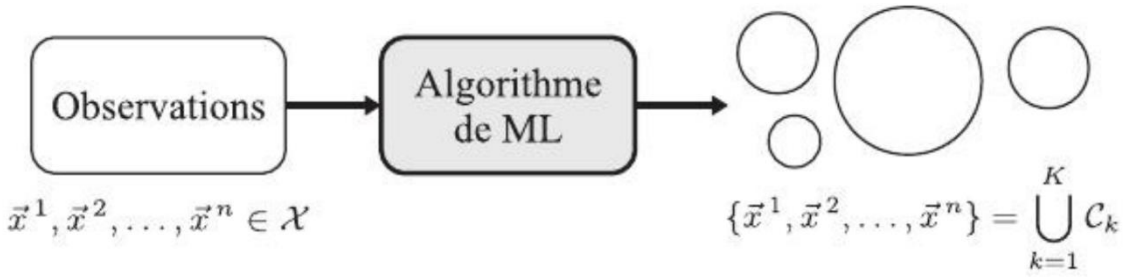
Avec l'apprentissage non supervisé ou auto-supervisé, le système découvre les interdépendances entre les variables d'entrée sans être entrainé pour une tâche particulière (Le Cun, 2019).

Figure 3 : Apprentissage non-supervisé

Source : Azencott, 2019

Le problème d'apprentissage, appelé Clustering, consiste à identifier des groupes dans les données pour comprendre leurs caractéristiques générales et déduire les propriétés d'une observation en fonction du groupe auquel elle appartient (Azencott, 2019).

Figure 4 : Clustering



Source : Azencott, 2019

Cela peut être appliqué, par exemple, dans le cadre d'une société de vente en ligne qui souhaiterait catégoriser les clients ayant des comportements similaires en plusieurs groupes, pour ainsi mieux comprendre leur profil d'acheteur et ainsi cibler plus efficacement certaines catégories de clients avec des promotions et du contenu personnalisé.

Un autre problème d'apprentissage, appelé la réduction de dimension, consiste à réduire les temps de calcul et l'espace mémoire nécessaire au stockage de données, mais aussi à améliorer les performances d'un algorithme d'apprentissage supervisé entraîné par la suite sur ces données (Azencott, 2019).

Figure 5 : Réduction de dimension

Source : Azencott, 2019

Par exemple, nous avons une base de données de plusieurs millions de photos de plantes et nous souhaiterions pouvoir extraire et stocker seulement les caractéristiques importantes de ces images afin de pouvoir les utiliser pour entrainer un algorithme de classification.

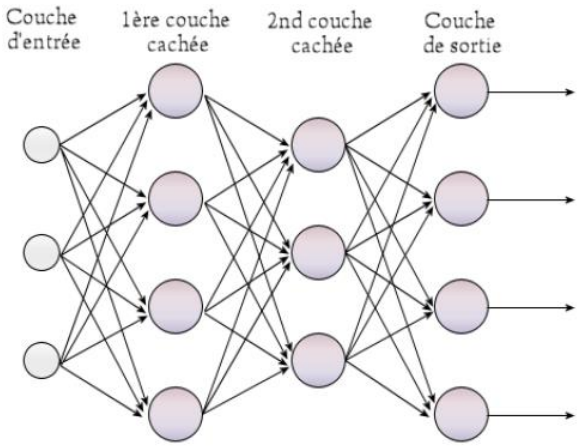
Lorsque le machine learning s'appuie sur l'usage de neurones artificiels s'inspirant du cerveau humain, nous parlons alors de deep learning (Vannieuwenhuyze, 2019).

2.2 Le deep learning

Les réseaux de neurones sont au cœur du deep learning. Ils sont polyvalents, puissants et extensibles, ce qui les rend parfaitement adaptés aux tâches d'apprentissage automatique complexes, comme la classification de milliards d'images, la reconnaissance vocale, la recommandation de vidéos auprès de centaines de millions d'utilisateurs ou l'apprentissage nécessaire pour battre le champion du monde du jeu de go (Géron, 2019).

On parle de deep learning (apprentissage profond) car les réseaux de neurones multicouches permettent l'apprentissage grâce à des représentations de données à de multiples niveaux d'abstraction (Di, Bhardwaj, Wei, 2018).

Figure 6 : Réseau de neurones multicouches



Source : Perceptron multicouche, 2020

Chaque rond représente un neurone artificiel, les flèches entre chaque neurone représentent les liens entre neurones et le sens de ces flèches représente le sens du signal. La couche d'entrée est chargée de réceptionner les données d'entrées. Les couches cachées effectuent les différents traitements et la couche de sortie livre les données de résultats.

Comment fonctionne un neurone artificiel ?

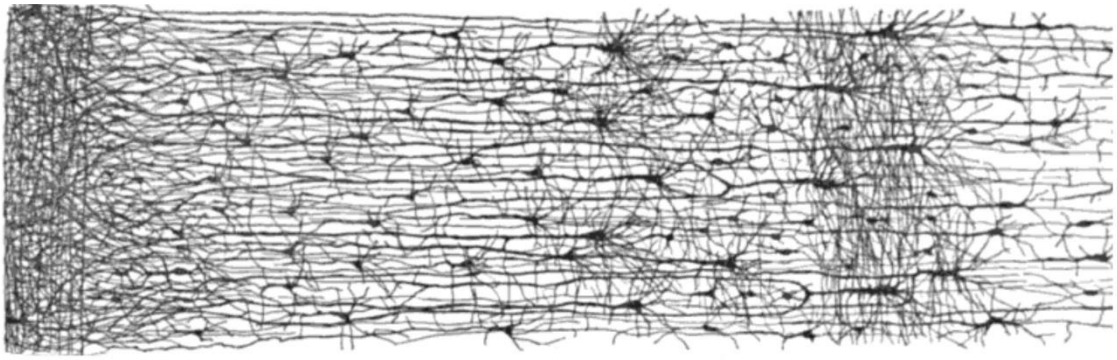
2.2.1 Les neurones

Le neurone artificiel se base sur le fonctionnement du neurone biologique qui est une cellule que l'on trouve principalement dans le cerveau des animaux (Géron, 2019).

Un neurone biologique produit de courtes impulsions électriques qui sont ensuite transmises aux neurones interconnectés. Lorsqu'un neurone reçoit en quelques millisecondes un nombre suffisant de ces impulsions électriques, il déclenche alors ses propres impulsions électriques pour les transmettre aux neurones suivants (Géron, 2019).

Des calculs complexes peuvent être réalisés par un réseau de neurones, de même manière qu'une fourmilière complexe peut être construite grâce aux efforts combinés de simples fourmis. Il semblerait que dans le cerveau, les neurones soient souvent organisés en couches successives (Géron, 2019).

Figure 7 : Réseau de neurones biologiques

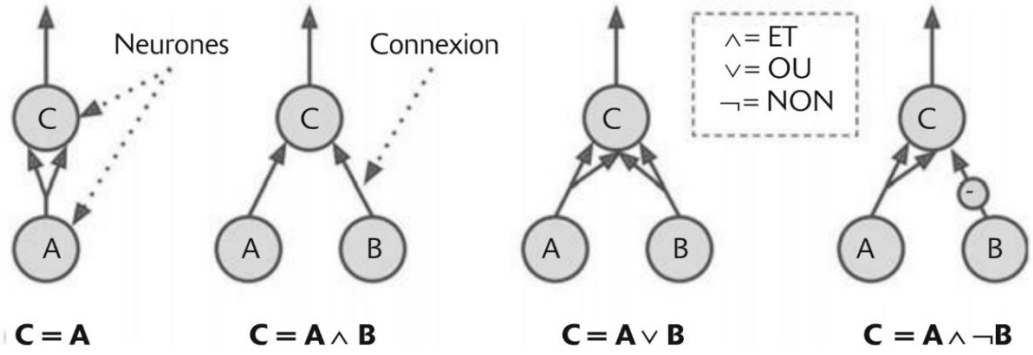


Source : Ramón y Cajal, 1899

Le neurophysiologiste Warren McCulloch et le mathématicien Walter Pitts ont présenté en 1943 un modèle informatique simplifié du fonctionnement combiné des neurones biologiques dans le but de résoudre des calculs complexes à l'aide de la logique propositionnelle (Géron, 2019).

Ce schéma montre l'implémentation de fonctions logiques (ET, OU, NON) grâce à des neurones artificiels interconnectés :

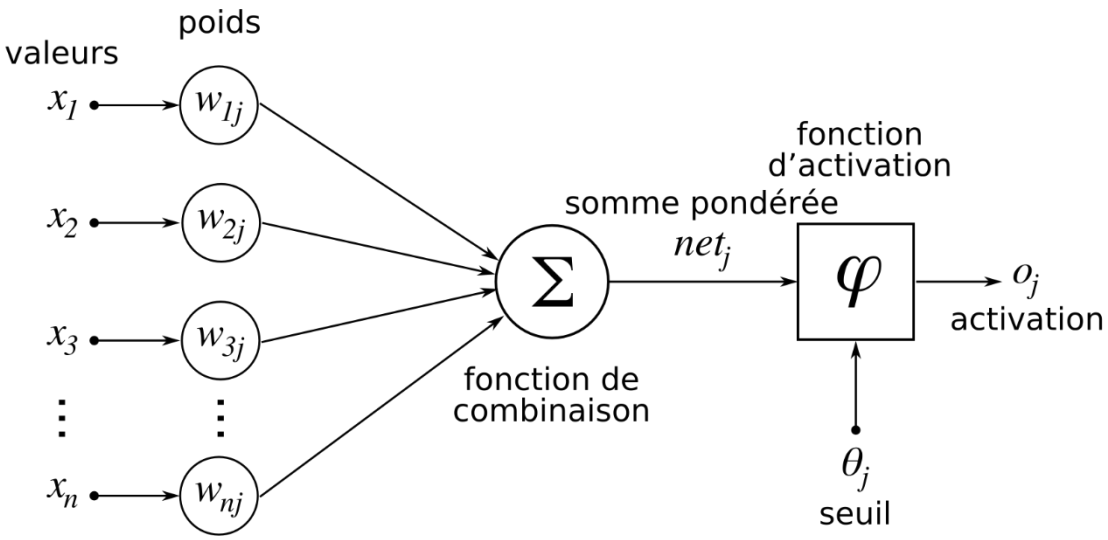
Figure 8 : Fonctions logiques et neurones



Source : Géron, 2019

Plus concrètement, un neurone artificiel est une fonction mathématique qui prend des valeurs pondérées en entrée, les additionne, transmet la valeur à une fonction de seuil, aussi appelée fonction d'activation, et s'active ou non. On peut voir un neurone artificiel comme une porte logique qui s'ouvre et se ferme en fonction des valeurs qui lui sont fournies (Israel, 2019).

Figure 9 : Neurone artificiel simple

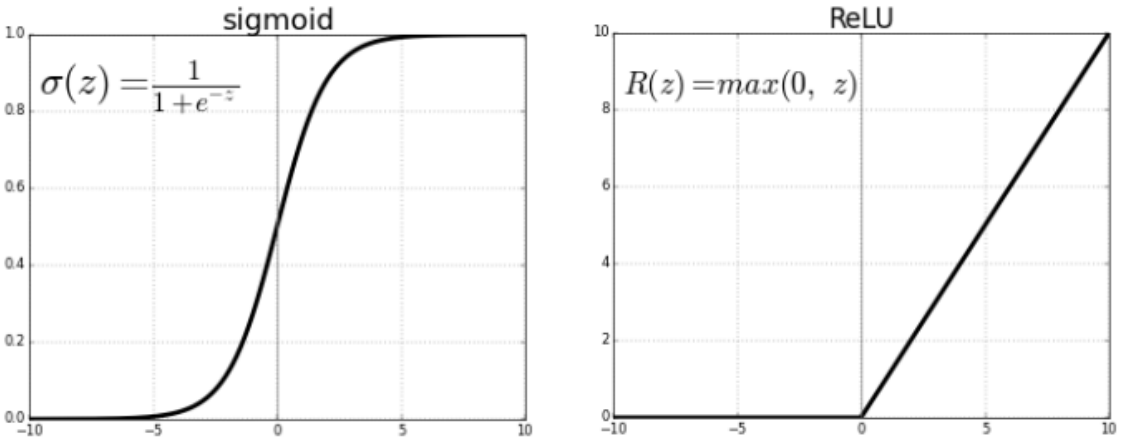


Source : Réseau de neurones artificiels, 2021

Ce neurone reçoit en entrée les valeurs x, pondérées par les poids w. Il effectue la somme pondérée de ses entrées puis passe cette valeur à travers la fonction d'activation pour produire la sortie o.

La fonction d'activation agit comme un filtre qui fournit une valeur de sortie (Israel, 2019). Il existe plusieurs fonctions utilisables avec un neurone artificiel, telles que la fonction de seuil binaire, la fonction sigmoïde, la fonction tangente hyperbolique ou encore la fonction ReLU (Vannieuwenhuyze, 2019).

Figure 10 : Sigmoïde et ReLU



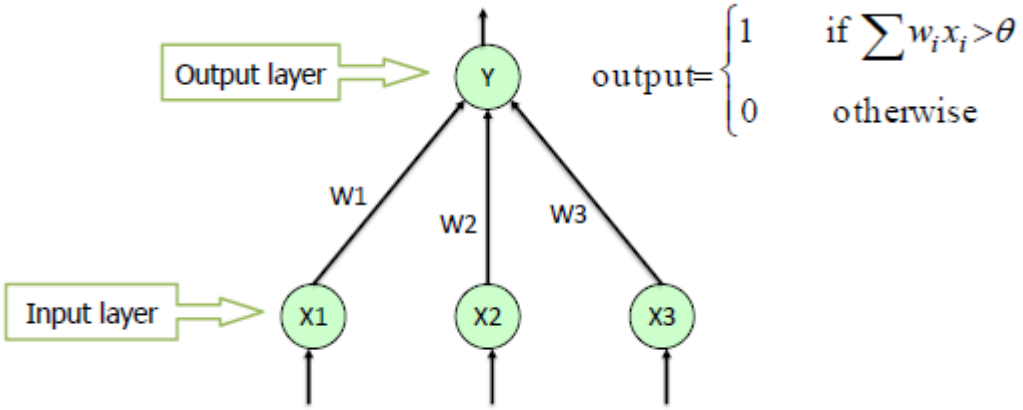
Source : Fuchs, 2018

Avec la fonction ReLU, si la valeur d'entrée est en dessous du seuil, alors la fonction retournera 0 sinon elle retournera la valeur d'entrée.

2.2.2 Les réseaux de neurones

Le perceptron monocouche est le premier réseau de neurones capable d'apprendre par expérience. Il a été créé par le psychologue américain Frank Rosenblatt en 1956. Le perceptron est un classifieur linéaire, il modélise une frontière linéaire pour classifier des données en deux ensembles (Alonzo et Audevart, 2019).

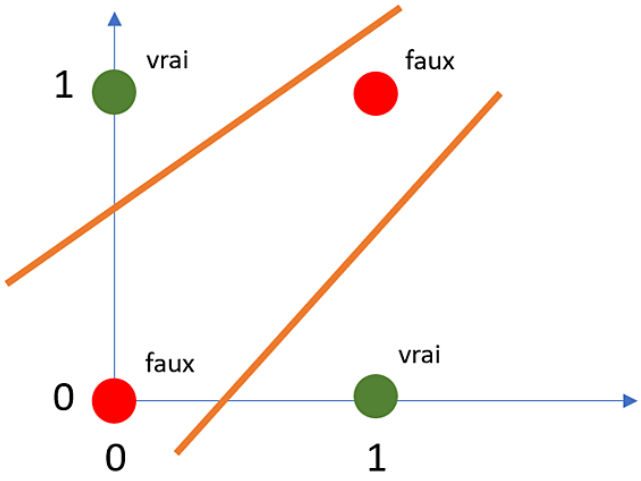
Figure 11 : Perceptron monocouche



Source : Saed, sans date

Le perceptron monocouche est relativement limité car il ne permet pas de reconnaître des classes d'objets non linéairement séparables (Alonzo et Audevart, 2019). En effet il n'est pas possible d'implémenter la fonction "Ou exclusif" appelée "XOR" (Alonzo et Audevart, 2019).

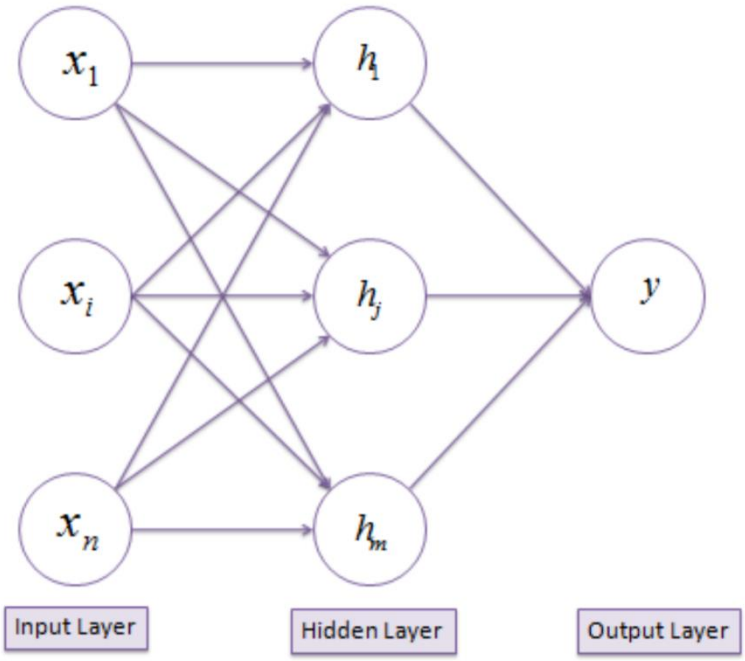
Figure 12 : Ou exclusif



Source : Vannieuwenhuyze, 2019

Pour implémenter cette fonction, il est nécessaire d'utiliser des perceptrons multicouches. Un perceptron multicouches est un ensemble de neurones organisés en couche. Le signal d'entrée se propage d'une couche à l'autre jusqu'à la sortie, en activant ou non, au fur et à mesure, des neurones, propageant ou non une information (Alonzo et Audevart, 2019).

Figure 13 : Perceptron multicouches



Source : Saed, sans date

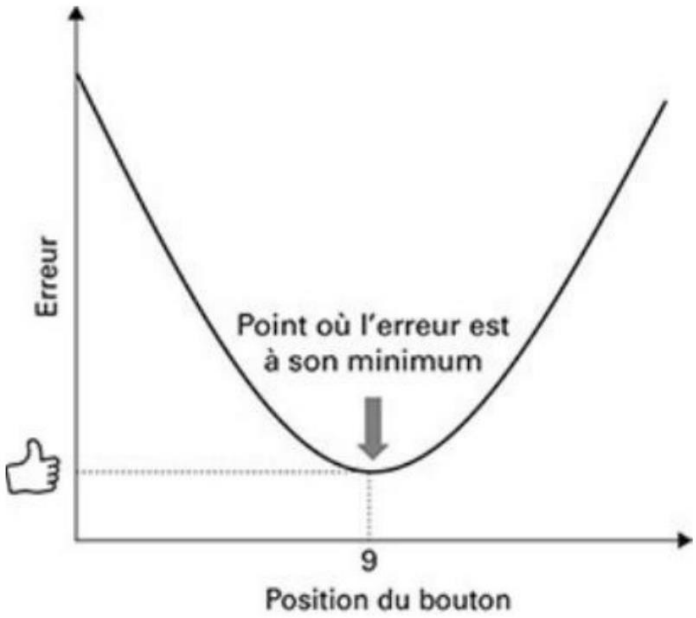
Le grand défi pour un réseau de neurones consiste à trouver la valeur optimale des poids entre chaque neurone pour adapter le modèle aux données de la base d'apprentissage (Alonzo et Audevart, 2019).

L'étape consistant à réaliser la somme pondérée des entrées et à utiliser une fonction d'activation pour obtenir une valeur de prédiction est appelée la phase de propagation. Le sens de cette propagation part des points d'entrée vers les points de sorties pour réaliser les calculs (Vannieuwenhuyze, 2019).

Une fois la prédiction réalisée, nous allons la comparer avec la prédiction attendue en calculant la différence entre la valeur attendue et la valeur prédite, cette différence est appelée l'erreur de prédiction. L'erreur de prédiction peut également être calculée à l'aide d'une fonction de perte pour quantifier la gravitée d'une erreur. Par exemple, l'erreur quadratique permet d'obtenir la même gravité pour une erreur positive ou négative (Vannieuwenhuyze, 2019).

Une fois cette erreur de prédiction obtenue, nous pouvons parcourir le réseau de neurone en sens inverse (de la sortie vers les entrées) afin d'ajuster l'ensemble des poids en tenant compte de l'erreur commise. Cette phase est appelée la rétropropagation de l'erreur. Pour parvenir à minimiser cette erreur de prédiction, nous allons effectuer une opération appelée la descente de gradient, qui consiste à ajuster, petit à petit, les différents points du réseau jusqu'à atteindre l'erreur de prédiction minimum, appelé le point de convergence (Vannieuwenhuyze, 2019).

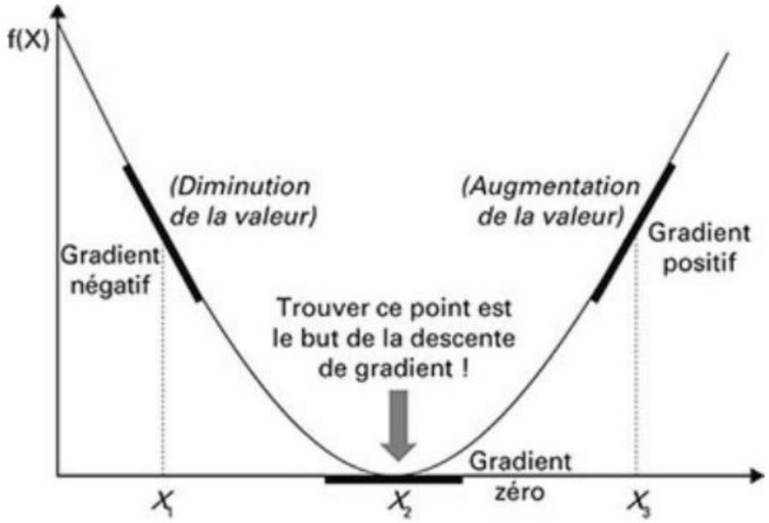
Figure 14 : Point de convergence



Source : Alonzo et Audevart, 2019

Une métaphore assez connue pour illustrer la descente de gradient est la descente de montagne dans une nuit noire. La seule indication que nous avons à disposition est le sens de la pente, mais nous ne pouvons pas du tout visualiser le bas de la montagne. Nous avançons donc, pas par pas, dans le sens de la pente jusqu'à atteindre le bas de la vallée. La taille des pas représente la taille de l'ajustement qui sera appliqué aux différents poids du réseau de neurones. Cet ajustement est effectué à l'aide d'un hyperparamètre appelé le taux d'apprentissage. Ce taux d'apprentissage ne doit pas être trop petit, au risque de mettre très longtemps avant d'arriver au point de convergence, il ne doit pas non plus être trop grand car nous pourrions dépasser le point de convergence et voir l'erreur augmenter (Vannieuwenhuyze, 2019).

Figure 15 : Descente de gradient



Source : Alonzo et Audevart, 2019

Maintenant que nous avons une idée du fonctionnement des réseaux de neurones, quel type de réseau est utilisé pour reconnaitre des images ?

2.2.3 Réseaux convolutifs

Les réseaux principalement utilisés dans le domaine de l'analyse d'images sont appelés réseaux convolutifs. Ces réseaux sont capables de prendre en compte la proximité des pixels et tirent ainsi parti de la forte corrélation que peuvent avoir les pixels voisins (Alonzo et Audevart, 2019).

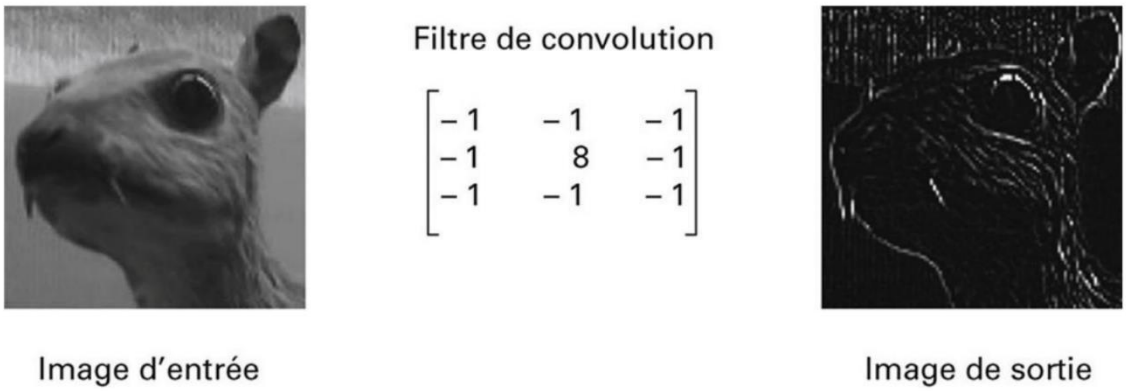
Les réseaux convolutifs sont basés sur le fonctionnement du cortex visuel, tel que présenté par Hubel et Wiesel dans leurs travaux avec des chats (Hubel, Wiesel, 1963), et la rétropropagation du gradient (Le Cun, 2019).

La convolution est une composante importante de cette architecture, qui a des similitudes avec les cellules simples du cortex visuel (Le Cun, 2019).

Une matrice de convolution, ou filtre de convolution, est une matrice de petite taille qui est utilisée pour détecter les contours, reconnaitre les formes et les textures, améliorer la netteté, etc… La matrice de convolution est appliquée à l'image zone par zone (Alonzo et Audevart, 2019).

Voici un exemple d'application d'un filtre de convolution mettant en valeur des détails ayant une variation rapide de luminosité :

Figure 16 : Filtre de convolution

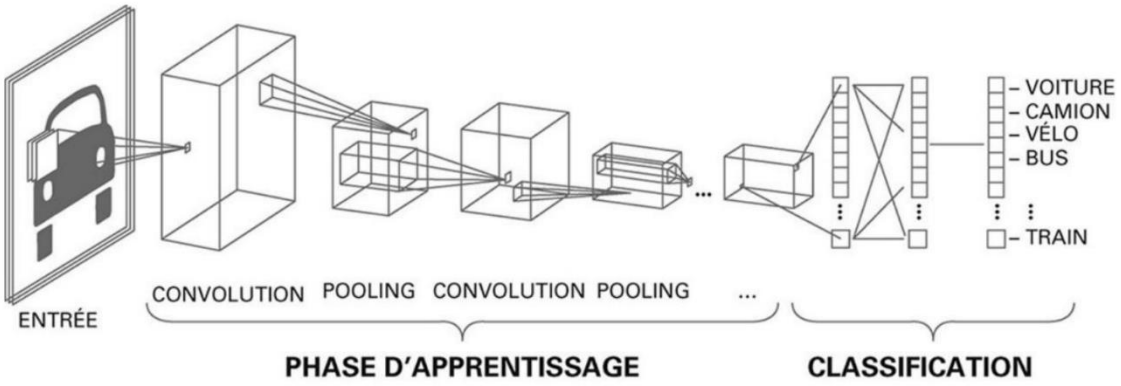


Source : Alonzo et Audevart, 2019

Une autre opération utilisée dans les réseaux convolutifs est le pooling, celle-ci permet de réduire la taille d'une image en ne conservant que les pixels les plus importants (Alonzo et Audevart, 2019).

Un réseau convolutif est constitué d'un empilement de couches de convolutions, de fonctions d'activation ReLU et de pooling (Le Cun, 2019).

Figure 17 : Réseau convolutif



Source : Alonzo et Audevart, 2019

Cette succession de couches permet d'apprendre de nouvelles caractéristiques. Le réseau illustré ci-dessus va extraire les lignes puis les formes pour mettre en évidence les caractéristiques permettant de catégoriser les images de véhicules fournies en entrée (Alonzo et Audevart, 2019).

De nos jours, un réseau convolutif peut comporter une centaine de ces couches (Le Cun, 2019).

Les réseaux adverses génératifs

Les réseaux adverses génératifs (en anglais « generative adversarial networks » ou GAN) ont été introduits en 2014 par Ian GoodFellow et son équipe dans la publication « Generative Adversarial Nets » de l'Université de Montréal (GoodFellow, 2014).

Les GAN sont une approche de la modélisation générative utilisant des méthodes de deep learning, telles que les réseaux de neurones convolutifs (Brownlee, 2019a).

La modélisation générative est une tâche d'apprentissage non-supervisée qui implique la découverte et l'apprentissage automatique de régularités et de patterns dans les données d'entrée de manière à ce que le modèle puisse générer de nouveaux échantillons ressemblant aux échantillons de l'ensemble de données d'origine (Brownlee, 2019a).

Les GAN sont une technique visant à former un modèle génératif en l'encadrant comme un problème d'apprentissage supervisé. Cette technique s'appuie sur deux sous-modèles (Brownlee, 2019a) :

- Un modèle générateur est formé pour générer de nouveaux échantillons
- Un modèle discriminateur tente de classer les échantillons comme réels (tirés de l'ensemble de données d'origine) ou faux (généré par le modèle générateur)

Ces deux sous-modèles sont entrainés ensemble dans un jeu à somme nulle, contradictoire, jusqu'à ce que, en théorie, le modèle discriminateur se trompe la moitié du temps, ce qui signifiera que le modèle générateur produit des échantillons suffisamment réalistes (Brownlee, 2019a).

2.3 Analogie des GAN

Pour comprendre le fonctionnement des GAN, faisons l'analogie grâce à trois acteurs différents : Un inspecteur, la banque nationale suisse (BNS) et un faussaire.

L’inspecteur (le modèle discriminateur) est chargé de vérifier si les billets reçus sont bien de vrais billets. Initialement, cet inspecteur n’a aucune expérience en détection de faux billets.

La BNS fournit l’inspecteur avec de vrais billets (l’ensemble de données d’origine) afin que l’inspecteur puisse améliorer son expérience.

Enfin, le faussaire (le modèle générateur) imprime des faux billets. Son objectif est que ses faux billets soient suffisamment réalistes pour tromper l’inspecteur.

L’inspecteur et le faussaire ont tous deux un objectif contradictoire, ils vont progresser ensemble tout en essayant de surpasser l’autre.

Chaque cycle d’apprentissage se déroule comme ceci :

1) L’inspecteur reçoit une liasse composée de billets (d’échantillons) provenant de la BNS (vrais billets) et du faussaire (faux billets) sans être au courant de leur provenance.

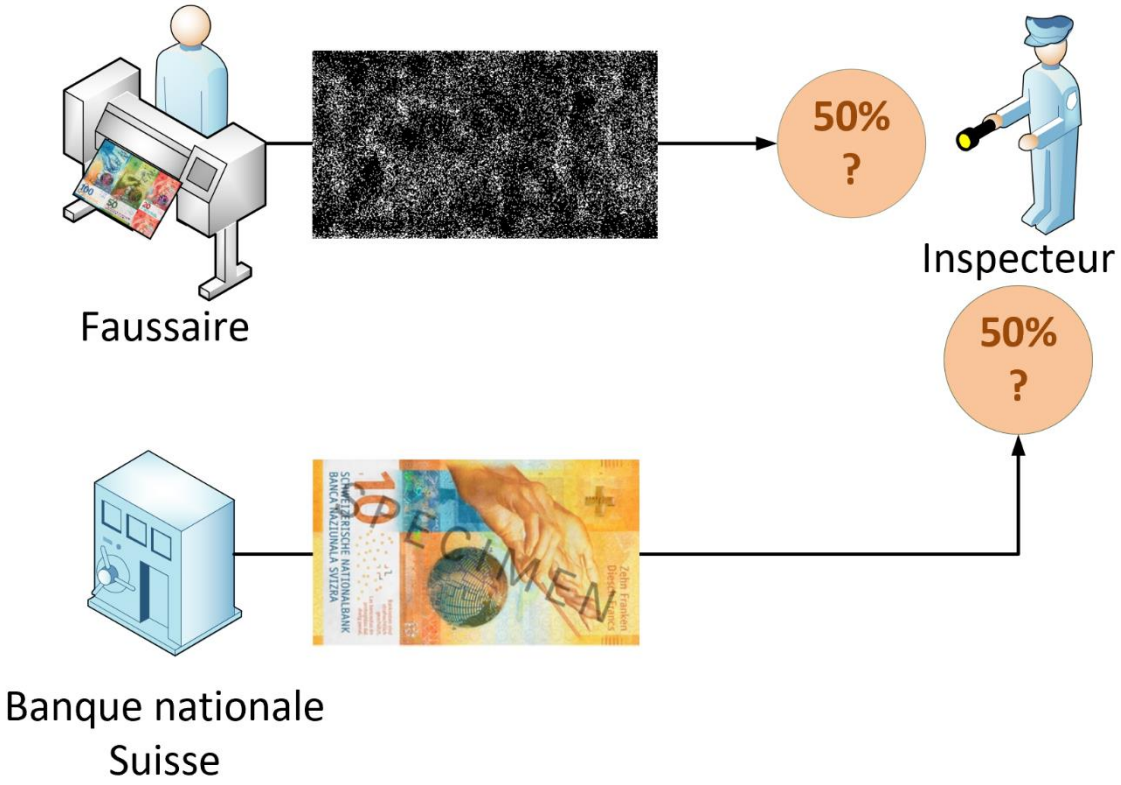
2) L’inspecteur donne, pour chaque billet, une probabilité, basée sur son expérience, qu’il s’agit d’un vrai billet (la prédiction sortie du modèle discriminateur).

3) L’inspecteur est mis au courant de la provenance de chaque billet (étiquettes « vrai », « faux »). Il calcule son erreur de prédiction sur la base de ses précédentes prédictions et de la provenance des billets, puis il fait évoluer (rétropropagation de l’erreur) ses techniques d’analyse.

4) Le faussaire reçoit les pourcentages attribués par l’inspecteur à ses faux billets. Il calcule son erreur en fonction de l’erreur obtenue par l’inspecteur avec ses faux billets (le contraire de l’erreur de prédiction de l’inspecteur) et fait évoluer (rétropropagation de l’erreur) ses techniques d’impression de faux billets.

Durant le 1er cycle d’apprentissage, l’inspecteur, qui n’a vraiment aucune expérience en la matière, attribue aléatoirement des pourcentages pour les billets de la BNS et du faussaire, et ce, malgré le niveau d’amateurisme très élevé du faussaire :

Figure 18 : Analogie apprentissage 1



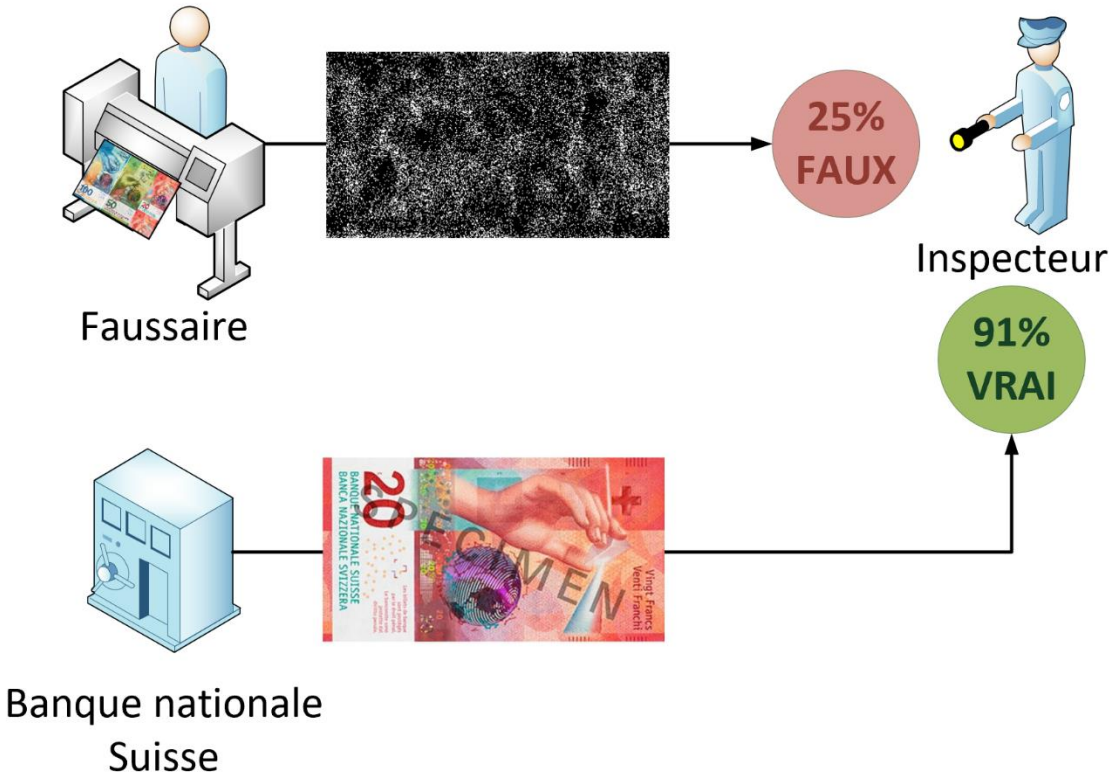
Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Source des billets : Banque nationale suisse, sans date

A la fin du 1er cycle, l'inspecteur améliore ses techniques d'analyse en fonction de ses erreurs.

Lors du 2ème cycle d'apprentissage, l'inspecteur peut alors, grâce à l'expérience acquise lors du 1er cycle, détecter plus facilement les faux billets du faussaire.

Figure 19 : Analogie apprentissage 2



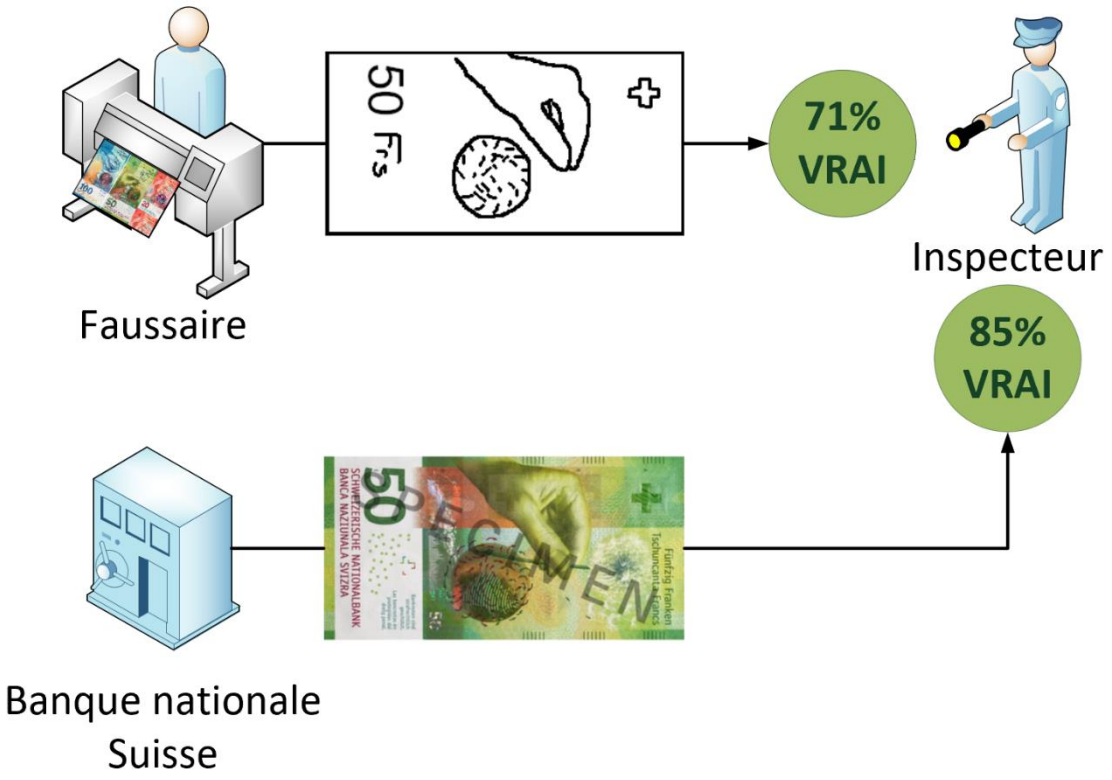
Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Source des billets : Banque nationale suisse, sans date

A la fin du 2ème cycle, constatant les mauvais pourcentages attribués à ses faux billets, le faussaire améliore ses techniques d'impressions.

Après quelques cycles supplémentaires, permettant l'évolution des techniques du faussaire, celui-ci parvient à tromper l'inspecteur :

Figure 20 : Analogie apprentissage 3



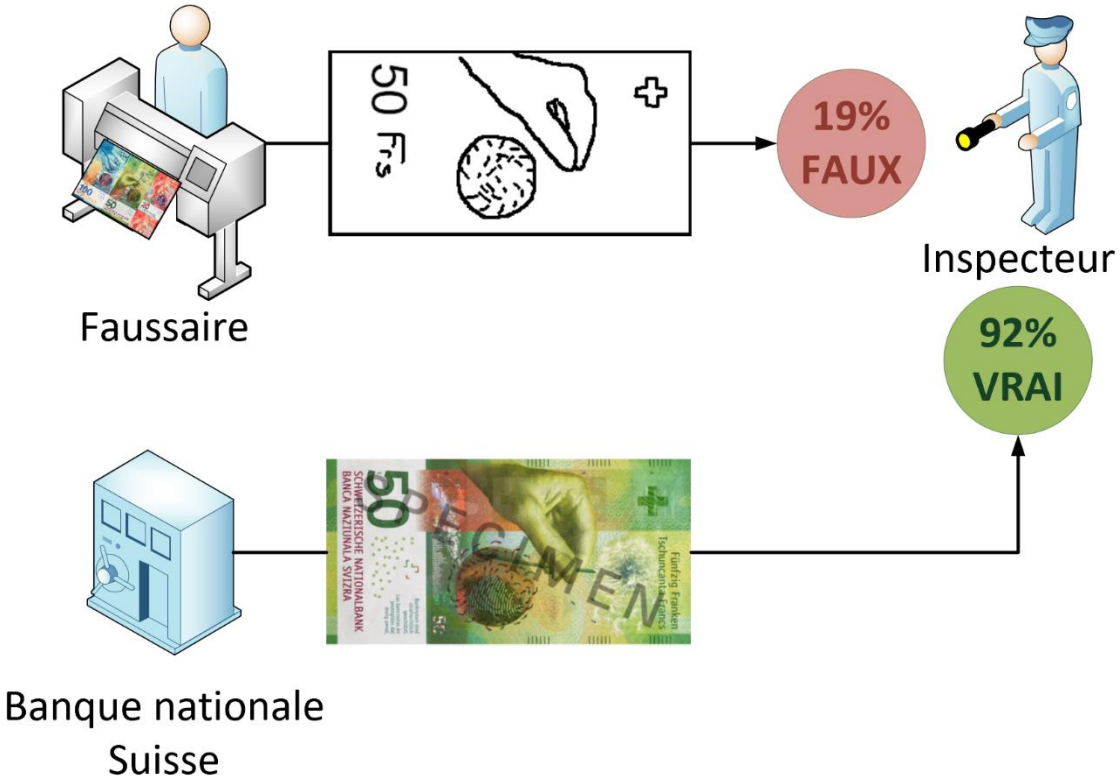
Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Source des billets : Banque nationale suisse, sans date

Constatant les pourcentages élevés attribués aux billets provenant du faussaire, l'inspecteur se voit forcer d'améliorer à nouveau ses techniques d'analyse.

Lors du cycle suivant, les billets du faussaire sont à nouveau mieux détectés par l'inspecteur :

Figure 21 : Analogie apprentissage 4

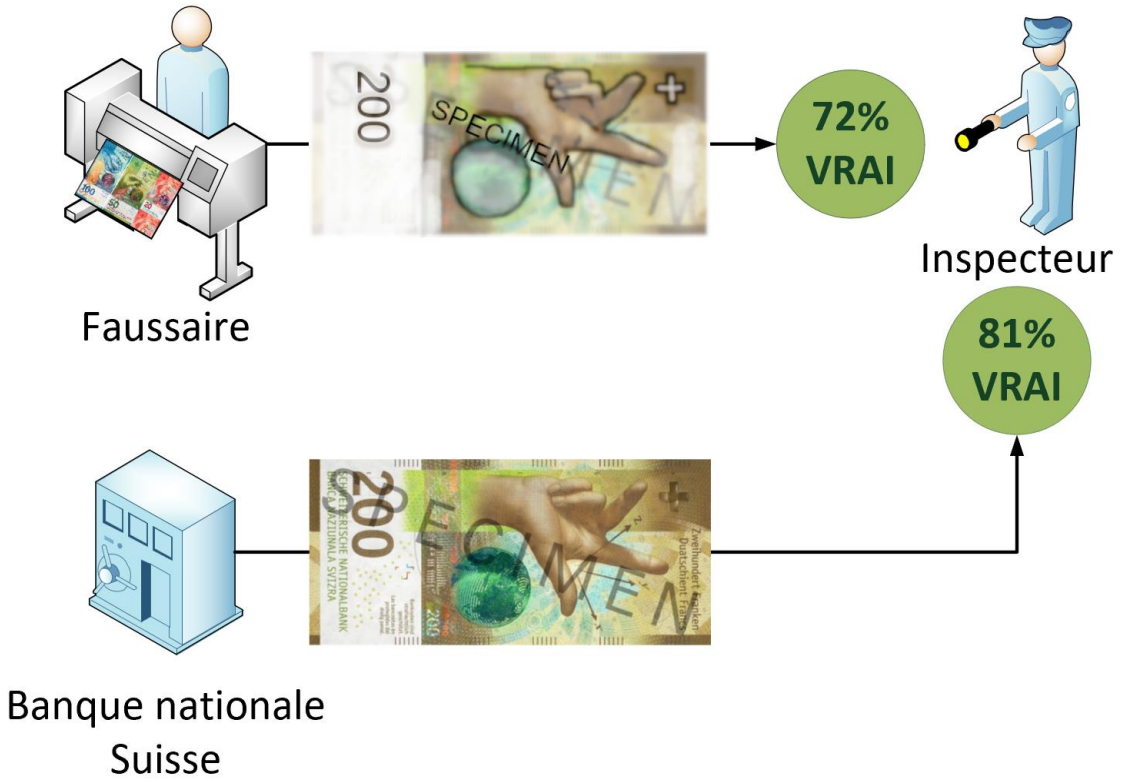


Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Source des billets : Banque nationale suisse, sans date

Ce qui pousse le faussaire à améliorer à nouveau ses techniques d'impressions.

Figure 22 : Analogie apprentissage 5

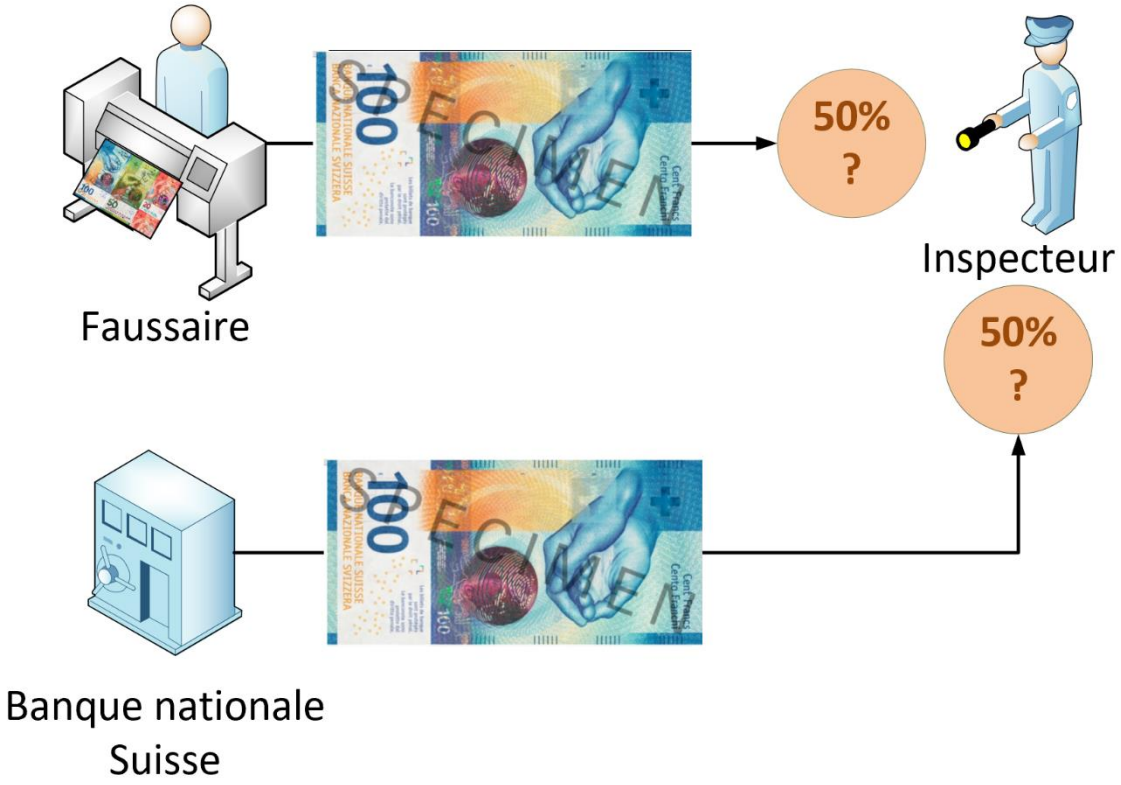


Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Source des billets : Banque nationale suisse, sans date

Ces cycles d'apprentissage continuent, jusqu'à ce que le faussaire soit capable d'imprimer des faux billets tellement réalistes que l'inspecteur n'arrive plus à faire la différence entre les vrais et les faux billets.

Figure 23 : Analogie apprentissage 6

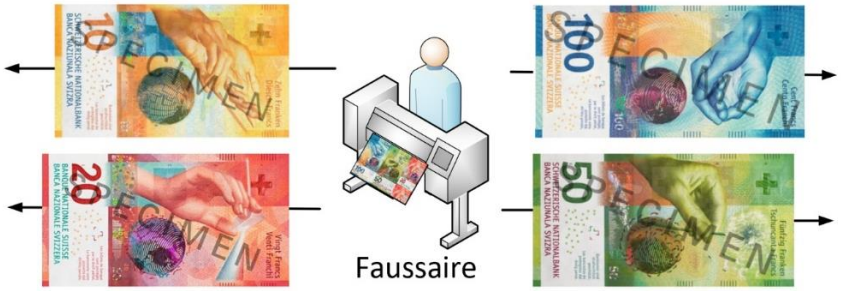


Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Source des billets : Banque nationale suisse, sans date

Dès lors, si l'on sort le faussaire de cette architecture, celui-ci peut générer, de façon totalement indépendante, des faux billets très semblables à des vrais :

Figure 24 : Analogie apprentissage 7



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

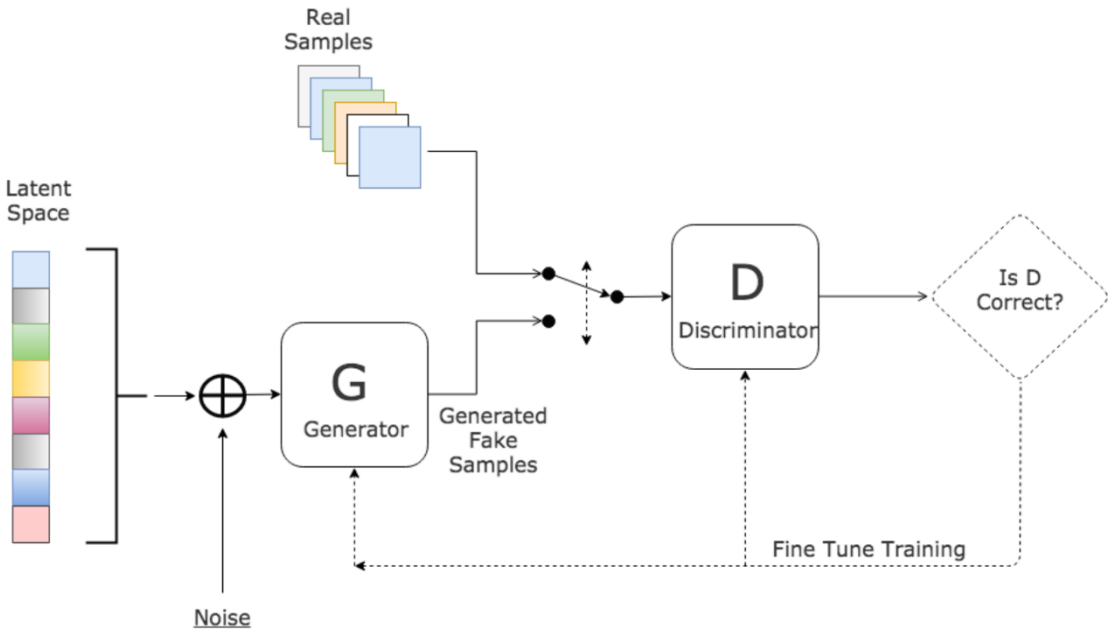
Source des billets : Banque nationale suisse, sans date

Cette démonstration permet de mieux saisir le concept de mise en concurrence entre le modèle génératif (faussaire) et le modèle discriminatif (inspecteur). Chaque modèle évolue grâce à l'autre jusqu'à ce que le modèle génératif soit capable de produire, de façon totalement indépendante, des échantillons très proches de l'ensemble de données d'origine.

2.4 Architecture

Voici les éléments qui constituent l'architecture plus formelle du GAN d'origine, proposé par Ian GoodFellow (GoodFellow, 2014) :

Figure 25 : Architecture GAN



Source : Gharakhanian, 2016

- Le discriminateur D : Chargé de détecter les vrais échantillons des faux
- Le générateur G : Chargé de générer des faux échantillons
- Les vrais échantillons x : Provenant du jeu de données d'origine. Peut être des images, de l'audio ou encore du texte
- Faux échantillons $G(z)$: Générés par le générateur G. Peut être des images, de l'audio ou encore du texte
- Bruit z : Vecteur de bruit issu d'une distribution normale (l'espace latent)
- La perte du discriminateur $V(D)$
- La perte du générateur $V(G)$

2.4.1 Discriminateur

Le réseau de neurones discriminateur essaie de différencier les vraies données de celles générées par le réseau de neurone générateur. Ce réseau classifie les données dans des catégories prédéfinies. Généralement, dans un réseau de type GAN, la classification binaire (vrai ou faux) est utilisée (Ahirwar, 2019).

Le discriminateur est une fonction $f : x \rightarrow y$, où x représente les échantillons et $y = D(x)$ représente la probabilité prédite par D que x fasse partie des vrais échantillons. Si les données utilisées sont images, x représente une image et y la probabilité prédite par D que cette image fasse partie des vraies images (Valle, 2019).

2.4.2 Générateur

Le réseau de neurone générateur utilise des données existantes pour générer de nouvelles données. Son objectif principal est de générer des données (images, vidéo, audio ou texte) à l'aide d'un vecteur contenant des nombres générés aléatoirement appelés bruit provenant d'un espace latent (Ahirwar, 2019).

Le générateur est une fonction $f : z \rightarrow \tilde{x}$, où z représente un vecteur contenant du bruit issu de l'espace latent et $\tilde{x} = G(z)$ représente les faux échantillons générés par le générateur. Si les données utilisées sont des images, alors $\tilde{x}$ représente une image générée par le générateur. A noter que le ~ sur x permet de différencier les images issues des vrais données x des images générées par le générateur $\tilde{x}$ (Valle, 2019).

2.4.3 Les vrais et faux échantillons

Vrai (ou réel) et faux sont les termes utilisés pour désigner les données utilisées dans les GAN. Vrai se réfère aux données qui proviennent du jeu de données d'origine que nous voulons apprendre, et faux se réfère aux données produites par le générateur (Valle, 2019).

2.4.4 Le bruit

Le vecteur de bruit z vient alimenter le générateur. Ce vecteur de bruit provient de l'espace latent qui contient les valeurs d'une distribution connue, telle que la distribution uniforme ou la distribution gaussienne (Valle, 2019).

2.4.5 La perte du discriminateur et du générateur

La fonction de perte utilisée pour le GAN d'origine se décrit par les règles suivantes (GoodFellow, 2014) :

- Le discriminateur doit approuver tous les échantillons provenant du jeu de données d'origine ;
- Le discriminateur doit rejeter toutes les échantillons générés par le générateur pour le tromper ;
- Le générateur doit tromper le discriminateur en l'obligeant à approuver tous ses échantillons générés.

Cette fonction de perte est illustrée par la fonction « minmax » suivante (GoodFellow, 2014) :

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

x représente un vrai échantillon.

$D(x)$ retourne la prédiction du discriminateur de la probabilité que le vrai échantillon soit vrai.

$G(z)$ retourne un faux échantillon produit par le générateur.

$D(G(z))$ retourne la prédiction du discriminateur de la probabilité que le faux échantillon soit vrai.

Le discriminateur tentera de maximiser la fonction en influant sur $D(x)$ sachant que $\log(D(x))$ va de $-\infty$ à 0.

Le générateur tentera de minimiser la fonction en influant sur $G(z)$ se trouvant dans D.

2.5 Entrainement

Comme vu avec la fonction de perte du générateur et du discriminateur, ces deux réseaux de neurones ont des objectifs opposés. Le discriminateur tente de distinguer les échantillons générés des vrais échantillons, tandis que le générateur tente de produire des échantillons suffisamment réalistes pour tromper le discriminateur. Puisque les GAN sont constitués de deux réseaux de neurones aux objectifs contraires, ils ne peuvent pas être entraînés comme un réseau de neurones traditionnel (Géron, 2019).

Chaque itération d'entraînement comprend deux phases (Géron, 2019) :

1) Entrainement du discriminateur : Un lot d'échantillons réels est récupéré à partir du jeu de données d'origine, auquel est ajouté un nombre égal de faux échantillons produits par le générateur. Les étiquettes sont fixées à 0 pour les faux échantillons et à 1 pour les vrais échantillons. Le discriminateur est entraîné sur ce lot étiqueté pendant une étape, en utilisant une perte d'entropie croisée binaire (indicateur utilisé pour évaluer la précision des prédictions). Il est important de noter qu'au cours de cette étape, la rétropropagation optimise uniquement les poids du discriminateur.

2) Entrainement du générateur : Un lot de faux échantillons est produit à partir du générateur et, une fois encore, le discriminateur doit prédire si les échantillons sont vrais ou faux. Cette fois-ci, aucun vrai échantillon n'est ajoutée au lot et

toutes les étiquettes sont fixées à 1 (comme pour les vrais échantillons). L'objectif du générateur est que ses échantillons générés soient considérés (à tort) comme vrais par le discriminateur. Il est indispensable que les poids du discriminateur soient figés au cours de cette étape, de sorte que la rétropropagation affecte uniquement les poids du générateur.

A noter que le générateur ne voit jamais directement les vrais échantillons, il reçoit uniquement les gradients qui proviennent du discriminateur. Plus le discriminateur est bon, plus les informations sur les vrais échantillons contenus dans ces gradients « de seconde main » seront pertinentes. Le générateur peut ainsi réaliser des progrès significatifs (Géron, 2019).

2.6 Evaluation

Voici deux approches permettant d'évaluer les GAN dans un contexte d'utilisation d'images.

2.6.1 Les mesures qualitatives

L'évaluation visuelle des échantillons par des humains est l'un des moyens les plus courants et les plus intuitifs pour évaluer les GAN (Borji, 2018).

Cependant, cette inspection manuelle a de nombreuses limitations (Borji, 2018). Elle est subjective, ne reflète pas pleinement la capacité des modèles, est difficile à reproduire et son coût est corrélé au nombre d'échantillons à inspecter. Les plateformes de Crowdsourcing peuvent aider à la mise en place de ce type d'évaluation.

Voici quelques approches courantes (Borji, 2018):

Plus proches voisins : Comparaison des images générées avec leurs plus proches voisins du jeu d'entrainement. Très utile pour détecter le surentrainement.

Evaluation et jugement de préférence : Evaluation des modèles par rapport à la fidélité des images générées. On pourrait par exemple générer des images à partir de 3 modèles différents et sélectionner les 2 images préférées.

Catégorisation rapide des scènes : Mélange des images d'entrainements et des images générées puis catégorisation (avec un délai de réflexion très court) entre « vraie image » ou « fausse image ». Ce principe est le fonctionnement discriminatif des GAN mais en remplaçant le discriminateur par un humain.

2.6.2 Les mesures quantitatives

Ces mesures permettent de fournir un score numérique. Il en existe de nombreuses, en voici quelques courantes (Borji, 2018) :

Average Log-likelihood : Utilisée par Ian Goodfellow dans son article original du GAN (GoodFellow, 2014). Cette méthode, également appelée Estimation par noyau ou encore méthode de Parzen-Rosenblatt, a pour but d'estimer la densité de probabilité d'une variable aléatoire.

Cependant, elle s'est avérée ne pas être une bonne mesure pour évaluer les GAN car elle favorise les modèles triviaux et ne reflète pas la fidélité visuelle des échantillons (Theis, 2016).

Inception Score : Peut-être la mesure la plus largement adoptée pour l'évaluation des GAN (Borji, 2018). Cela implique l'utilisation d'un réseau de neurone pré-entrainé existant utilisé pour donner un score reflétant le degré de réalisme de l'image générée.

2.7 Principales difficultés

L'entrainement d'un GAN peut s'avérer être une tâche compliquée. Voici deux problématiques importantes souvent rencontrées (Bernico, 2018) :

Stabilité :

L'entrainement d'un GAN doit être effectué progressivement, à la fois du côté du discriminateur et du générateur. Le discriminateur et le générateur se battent l'un contre l'autre pour progresser individuellement mais ils s'aident également à progresser mutuellement. Il ne faut donc pas que l'un prenne trop d'avance sur l'autre. Par exemple, l'apprentissage devient instable si le discriminateur prend trop de pouvoir sur le générateur et donc devient certain que les échantillons créés par le générateur sont faux. Le générateur ne pourra alors plus progresser.

Mode collapse :

Il arrive parfois que le générateur choisisse toujours là même méthode pour tromper le discriminateur. Dans ce cas il se restreint à générer les mêmes échantillons.

2.8 Evolutions

En 2016 Yann Le Cun, responsable du laboratoire d'intelligence artificielle chez Facebook et l'un des inventeurs du deep learning, écrivait « Ça a l'air d'être un sujet technique, mais je pense vraiment que ça ouvre une porte à tout un monde de possibilités » (Le Cun, 2016).

Les réseaux adverses génératifs n'ont cessé d'évoluer ces dernières années avec de nombreuses optimisations ou encore des applications diverses dans les domaines de l'art, de la mode, de la publicité, des sciences ou encore des jeux vidéo (Shatri 2016).

Voici des exemples de photos générées à l'aide des réseaux adverses génératifs. L'évolution de la résolution de ces photos permet d'illustrer la progression des GAN depuis 2014 :

Figure 26 : Progression des GAN

Source : GoodFellow, 2019

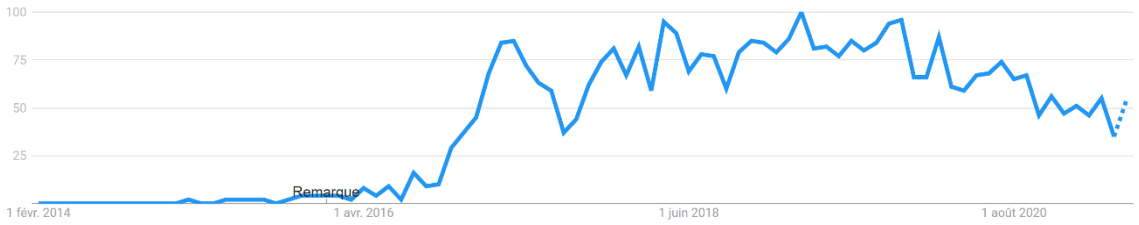
En 2019, voici une image générée depuis le site thispersondoesnotexist.com :

Figure 27 : Personne fictive

Source : Karras, 2019

L'intérêt pour les « Generative Adversarial Networks » depuis avril 2016 dans le moteur de recherche Google est significatif :

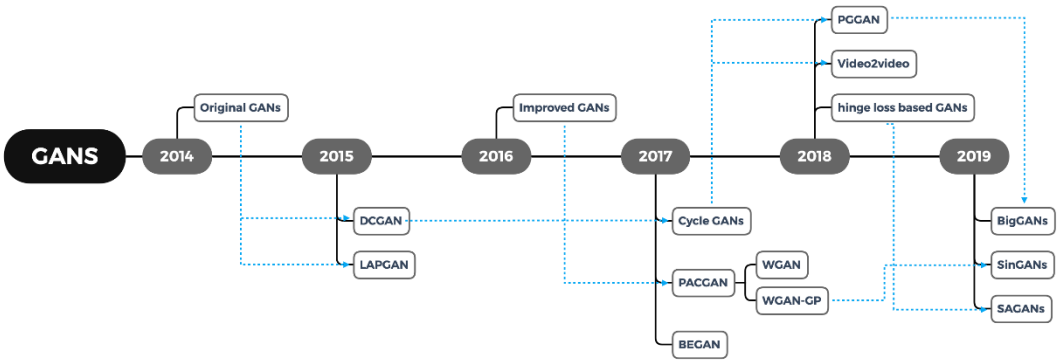
Figure 28 : Google Trends GAN



Source : Google Trends, 2021

Voici quelques variantes de GAN publiées entre 2014 et 2019 :

Figure 29 : Variantes de GAN



Source : Shatri, 2016

On peut citer, par exemple, le Deep convolutional GAN (DCGAN) qui utilise des réseaux de neurones convolutifs pour obtenir de meilleurs résultats avec la génération d'images (Radford, 2015), le Wasserstein GAN (WGAN) qui propose une alternative au GAN d'origine permettant d'améliorer la stabilité du processus d'entrainement (Arjovsky, 2017) ou encore plus récemment le BigGAN qui a été entrainé pour générer des images sur la base de différentes catégories (Brock, 2018). A noter qu'il existe encore bien d'autres variantes.

2.9 Applications

Les applications des GAN sont nombreuses, voici une liste non-exhaustive (Ahirwar, 2019) :

- Génération d'images : Les GAN peuvent générer de nouvelles images réalistes sur la base d'images réelles contenues dans un jeu d'entrainement, ces

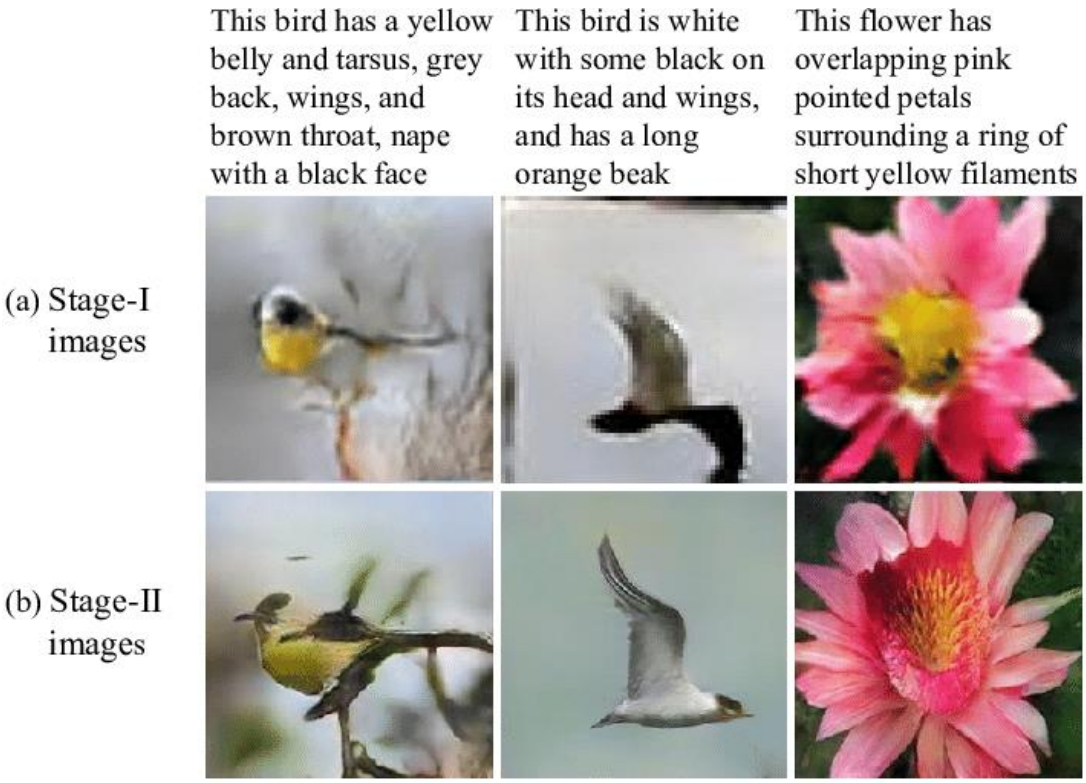
nouvelles images pourraient être utilisées pour des campagnes de marketing, la génération de logo, le divertissement, les réseaux sociaux, etc..

Figure 30 : Images générées

Source : Karras, Aila, Laine et al, 2017

- Synthèse texte-image : Il est possible de générer des images sur la base de descriptions textuelles. Cela pourrait aider l'industrie du film, à générer des images de séquences à tourner sur la base du script.

Figure 31 : Textes en images

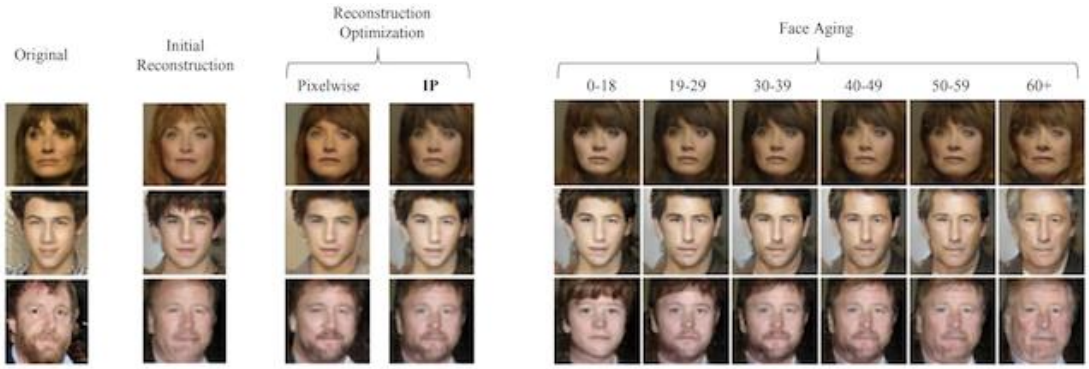


Source : Zhang, Xu, Li et al, 2017

(a) Images en basse définition générées à partir du texte. (b) Images en haute définition générées à partir des images de l'étape 1.

- Vieillissement du visage : Cela pourrait être utile dans le domaine du divertissement et dans le domaine de la sécurité. Par exemple, l'identification faciale pourrait prendre en compte le vieillissement des personnes.

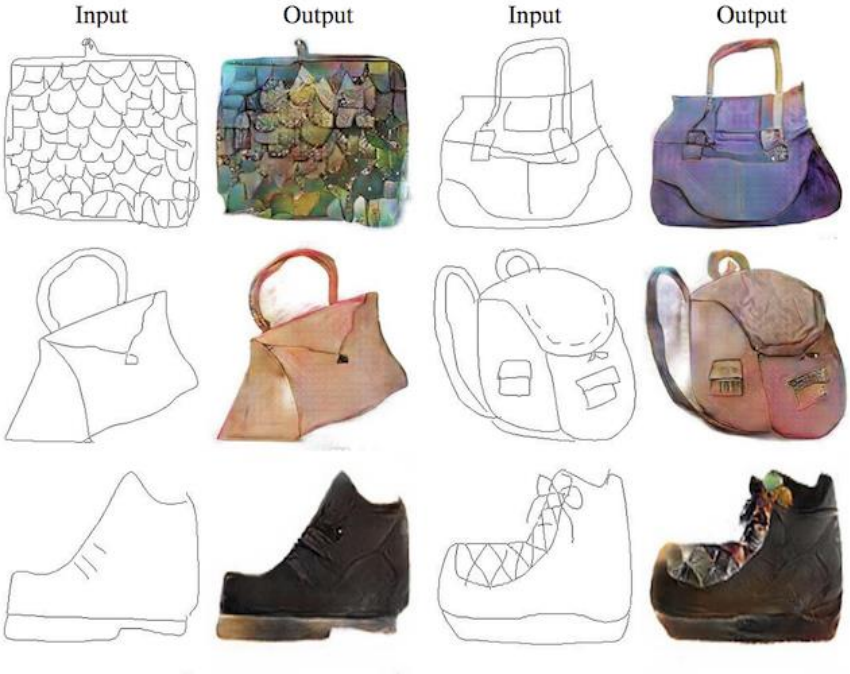
Figure 32 : Vieillissement du visage



Source : Antipov, Baccouche, Degelay, 2017

- Traduction d'image vers image : Des images prises de jour peuvent être converties en images prises de nuit, des peintures pourraient être converties avec le style de Picasso ou de Van Gogh, des photos aériennes en images satellites, des dessins faits à la main en images réalistes, etc.

Figure 33 : Traduction d'image en image



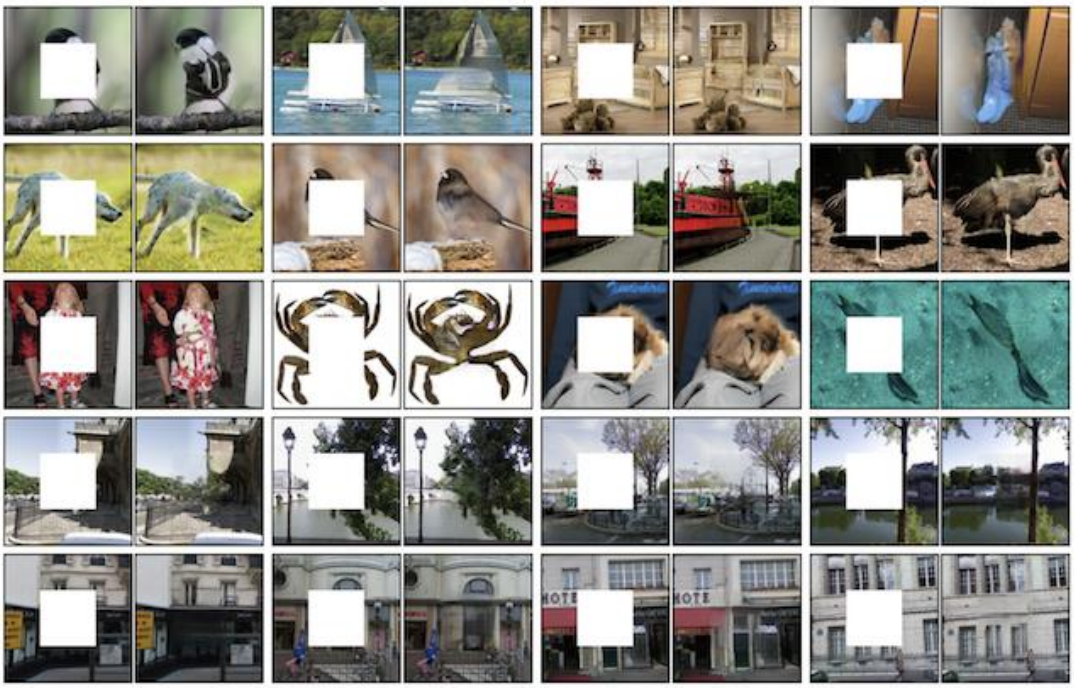
Source : Isola, Zhu, Zhou et al, 2017

- Synthèse vidéo : Les GAN peuvent être utilisés pour générer du contenu vidéo, ce qui prendrait moins de temps que si le contenu devait être créé manuellement. Cela pourrait aider à améliorer la productivité des créateurs de film, ou faciliter l'accès aux amateurs pour créer leurs propres vidéos.

- Génération d'image en haute résolution : Les photos de mauvaises qualités pourraient être améliorées sans perdre les détails essentiels.

- Compléter des parties manquantes d'images : Les GAN peuvent aider à regénérer les parties manquantes d'images.

Figure 34 : Parties manquantes complétées



Source : Pathak, Krahenbuhl, Donahue et al, 2016

3. La voix

La voix humaine est l'ensemble des sons produits par le frottement de l'air des poumons sur les replis du larynx de l'être humain (Voix humaine, 2021).

Mais qu'est-ce que le son, comment est-il représenté numériquement et comment l'utiliser avec les réseaux de neurones ?

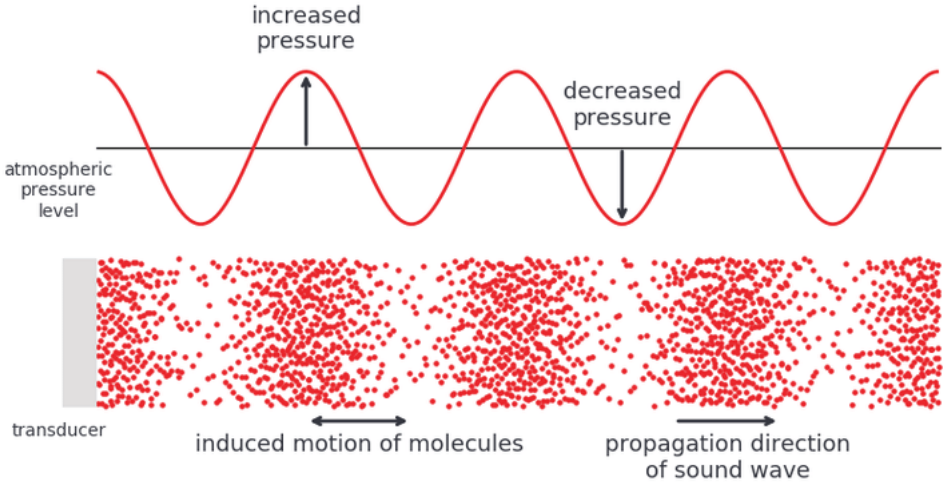
3.1 Le son

Le son est une vibration mécanique d'un fluide, qui se propage sous forme d'ondes longitudinales grâce à la déformation élastique de ce fluide. Les êtres humains, comme beaucoup d'animaux, ressentent cette vibration grâce au sens de l'ouïe (Son, 2021).

3.1.1 La forme d'onde

Dans l'image ci-dessous, le graphique des particules en bas représente les zones de basse et haute pression dans l'air causées par le son. Les zones à basse pression ont une densité de particules plus faible et les zones à pression plus élevée ont une densité de particules plus élevée.

Figure 35 : Propagation pression onde sonore



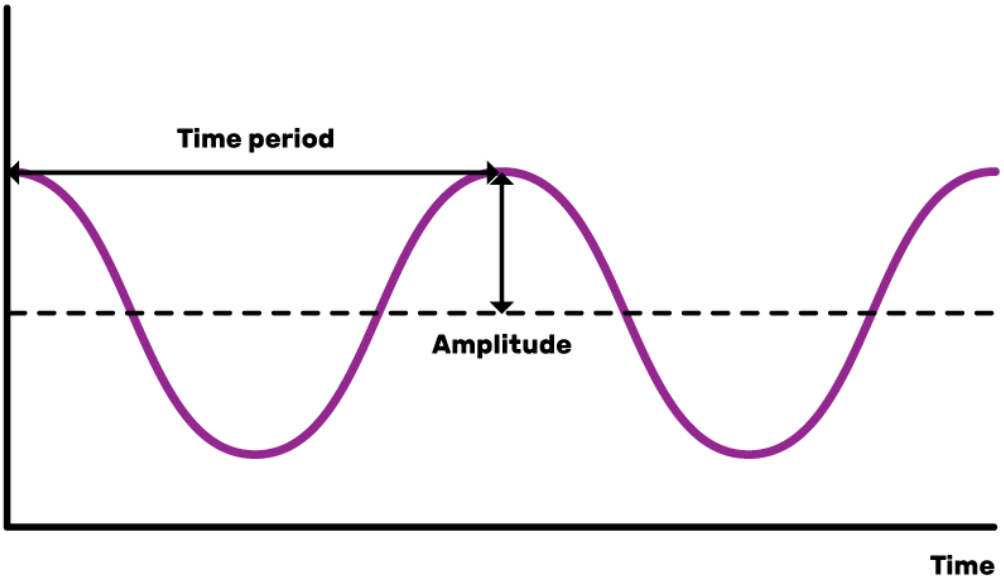
Source : Seckel, Singh, 2019

Sur la base de cette différence de pression, une courbe peut être générée, avec des pics dans les zones avec une pression d'air plus élevée et des creux dans des zones avec une pression d'air plus basse. Cette visualisation des ondes sonores est connue sous le nom de forme d'onde et fournit de nombreux détails sur le son, qui peuvent être exploités lors de l'extraction des caractéristiques de l'audio (Guduguntla, 2021).

3.1.2 L'amplitude, la période et la fréquence

Les signaux sonores se répètent souvent à intervalles réguliers. La hauteur indique l'intensité du son et est connue sous le nom d'amplitude (Doshi, 2021a).

Figure 36 : Période et amplitude



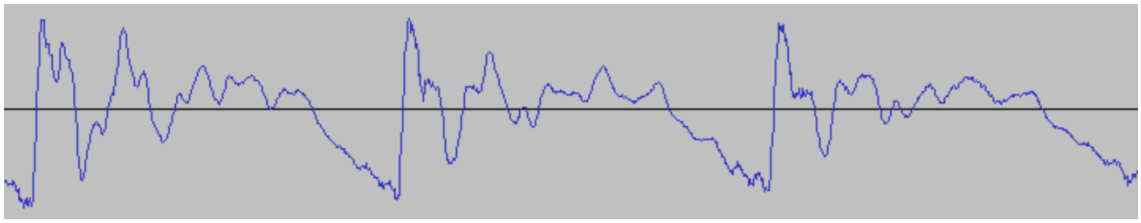
Source : The Physics of Sound, sans date

Le temps nécessaire au signal pour terminer une onde complète correspond à la période. Le nombre d'ondes produites par le signal en une seconde s'appelle la fréquence. La fréquence est l'inverse de la période. L'unité de fréquence est le Hertz (Doshi, 2021a).

L'oreille humaine est capable de distinguer des sons graves de basses fréquences allant jusqu'à 20 Hz et des sons aigus de hautes fréquences allant jusqu'à 20 000 Hz (Bayle, 2018).

La majorité des sons que nous percevons ne suivent pas des schémas périodiques aussi simples et réguliers. En effet, des signaux de fréquences différentes peuvent être additionnés pour créer des signaux composites avec des motifs répétitifs plus complexes. Tous les sons que nous entendons, y compris notre propre voix humaine, se composent de formes d'onde comme celles-ci (Doshi, 2021a).

Figure 37 : Forme d'onde voix humaine

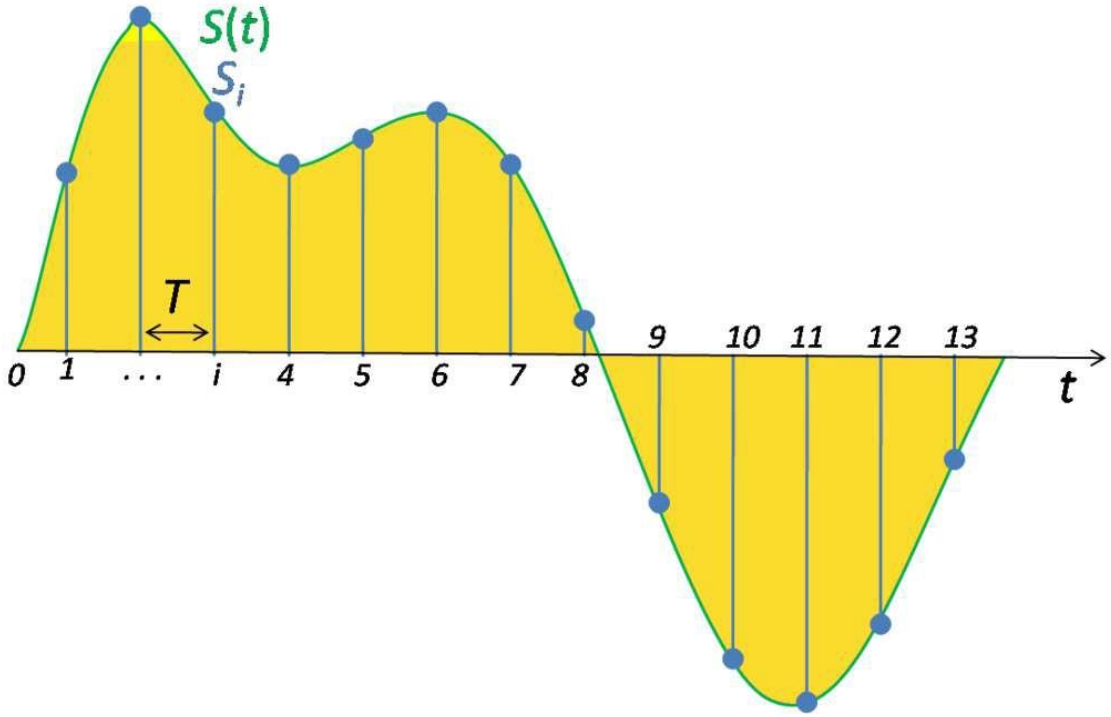


Fait à l'aide de l'outil : Audacity

3.2 Représentation numérique du son

Pour numériser une onde sonore, c'est-à-dire transformer le signal en une série de nombres, l'amplitude du son est mesurée à des intervalles de temps fixes (Doshi, 2021a).

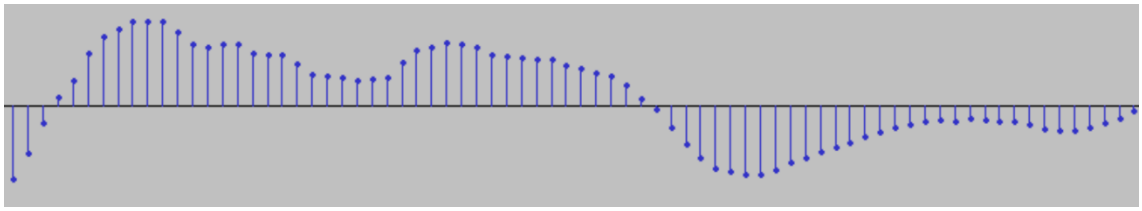
Figure 38 : Echantillonnage 1



Source : Doshi, 2021a

Chacune de ces mesures est appelée un échantillon et la fréquence d'échantillonnage correspond au nombre d'échantillons par seconde. Par exemple, une fréquence d'échantillonnage commune est d'environ 44 100 échantillons par seconde. Cela signifie qu'un clip de 10 secondes aurait 441 000 échantillons (Doshi, 2021a). Voici un exemple d'échantillonnage à 44.1 kHz d'une forme d'onde de voix humaine :

Figure 39 : Echantillonnage 2



Fait à l'aide de l'outil : Audacity

3.3 Traitement du son avec le deep learning

A l'époque précédant le deep learning, les applications d'apprentissage automatique audio dépendaient des techniques traditionnelles de traitement du signal numérique pour extraire des caractéristiques. Par exemple, pour comprendre la voix humaine, les signaux audios pouvaient être analysés à l'aide de concepts phonétiques pour extraire des éléments comme les phonèmes. Il fallait une grande expertise spécifique au domaine pour résoudre ces problèmes et régler le système pour obtenir de meilleures performances. Cependant, avec le deep learning, les techniques de traitement audio traditionnelles ne sont plus nécessaires et nous pouvons compter sur une préparation standard des données (Doshi, 2021a).

Avec le deep learning, les données audios ne sont pas traitées sous leur forme brute. L'approche courante utilisée consiste à convertir les données audios en images, puis à utiliser l’architecture des réseaux de neurones convolutifs standards pour traiter ces images. Cela se fait en générant des spectrogrammes à partir du signal audio (Doshi, 2021a).

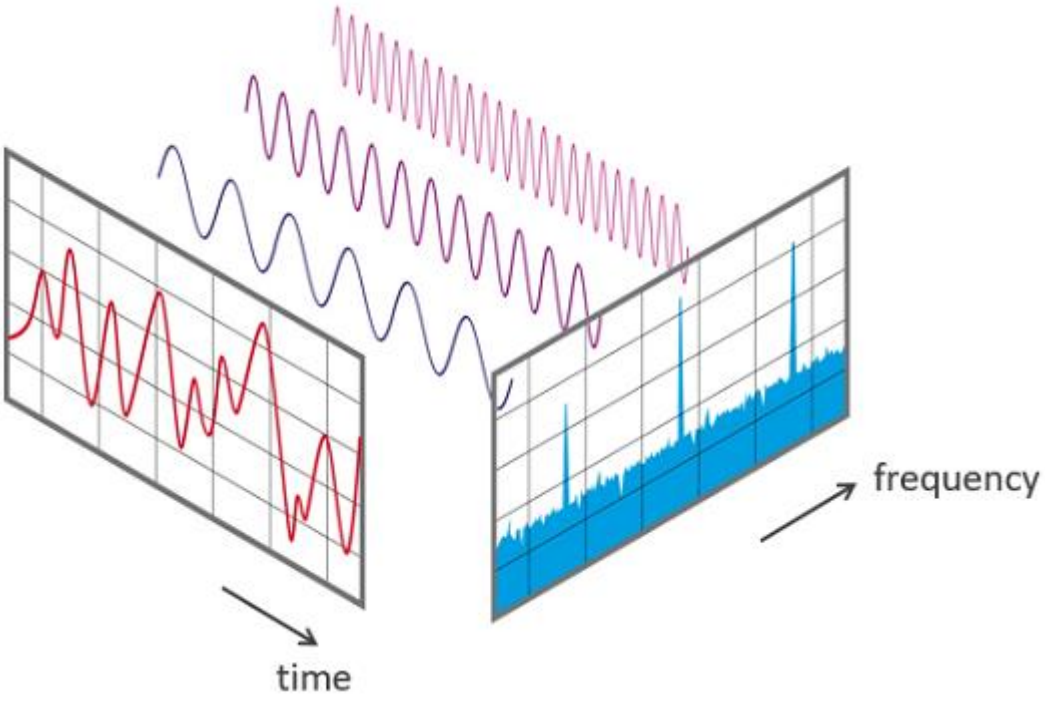
3.3.1 Spectre, spectrogramme et fréquence fondamentale

Pour rappel, les signaux de fréquences différentes peuvent être additionnés pour créer des signaux composites, représentant tout son qui se produit dans le monde réel. Cela signifie que tout signal se compose de nombreuses fréquences distinctes et peut être exprimé comme la somme de ces fréquences (Doshi, 2021a).

Le spectre est l'ensemble des fréquences qui sont combinées ensemble pour produire un signal. Il trace toutes les fréquences présentes dans le signal avec la force ou l'amplitude de chaque fréquence. Le spectre est une autre façon de représenter le même signal. Il montre l'amplitude par rapport à la fréquence. (Doshi, 2021a).

Le passage d’une forme d’ondes à son spectre revient au passage du domaine temporel au domaine fréquentiel (Doshi, 2021a) :

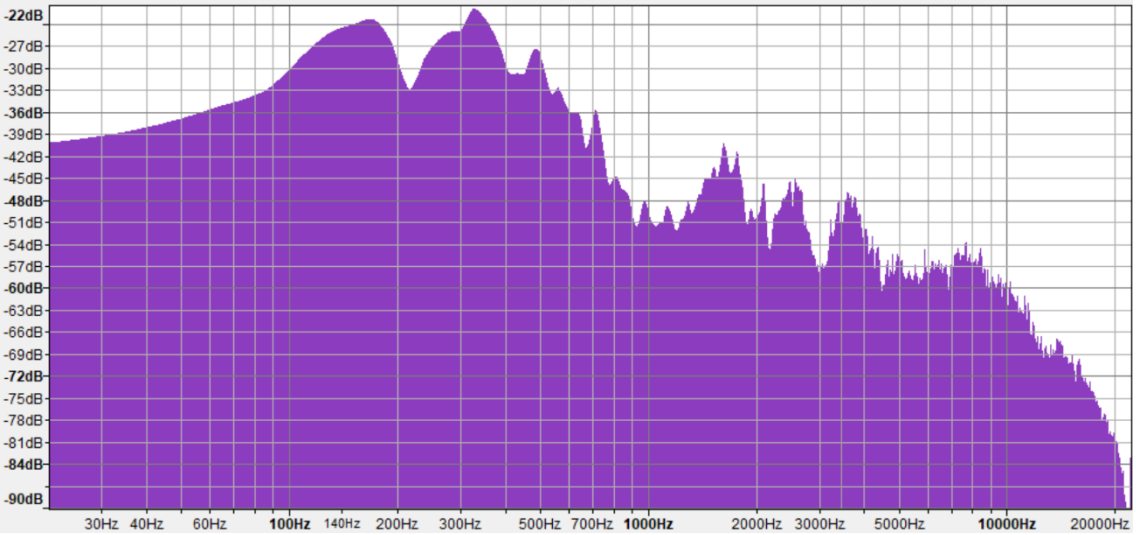
Figure 40 : Passage temporel en fréquentiel



Source : Phonical, 2017

Voici ci-dessous le spectre d'une forme d'onde de voix humaine :

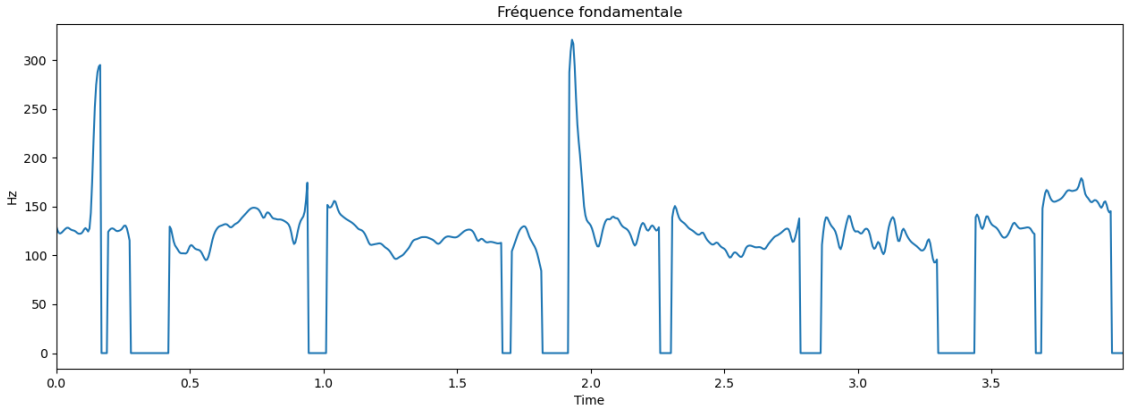
Figure 41 : Domaine fréquentiel



Fait à l'aide de l'outil : Audacity

La fréquence la plus basse d'un signal est appelée fréquence fondamentale (Doshi, 2021a).

Figure 42 : Fréquence fondamentale



Fait à l'aide de l'outil : Librosa

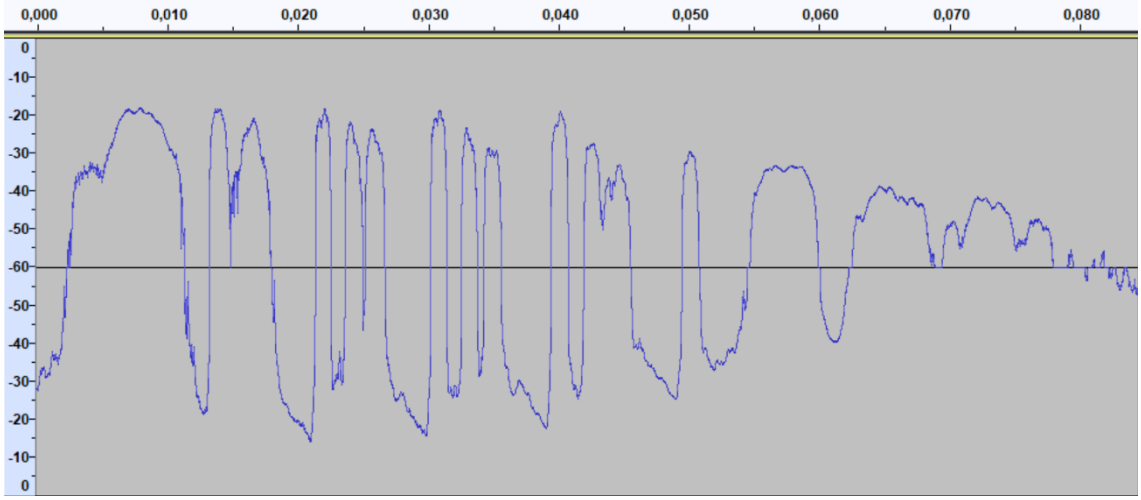
Puisqu'un signal produit des sons différents à mesure qu'il varie dans le temps, ses fréquences constitutives varient également avec le temps. En d'autres termes, son spectre varie avec le temps (Doshi, 2021a).

Le spectrogramme d'un signal trace son spectre dans le temps, comme une « photographie » du signal. Il trace le temps sur l'axe des x et la fréquence sur l'axe des y. C'est comme si nous prenions le spectre encore et encore à différentes instances dans le temps, puis les réunissions tous ensemble dans un seul graphe (Doshi, 2021a).

Le spectrogramme utilise différentes couleurs pour indiquer l'amplitude ou la force de chaque fréquence. Plus la couleur est claire, plus l'énergie du signal est élevée. Chaque «tranche» verticale du spectrogramme représente le spectre du signal à cet instant dans le temps et montre comment la force du signal est distribuée dans chaque fréquence trouvée dans le signal à cet instant (Doshi, 2021a).

Dans l'exemple ci-dessous, la première image affiche le signal dans le domaine temporel ie. Amplitude vs temps. Cela nous donne une idée du niveau sonore ou silencieux d'un signal audio à tout moment, mais cela nous donne très peu d'informations sur les fréquences présentes (Doshi, 2021a).

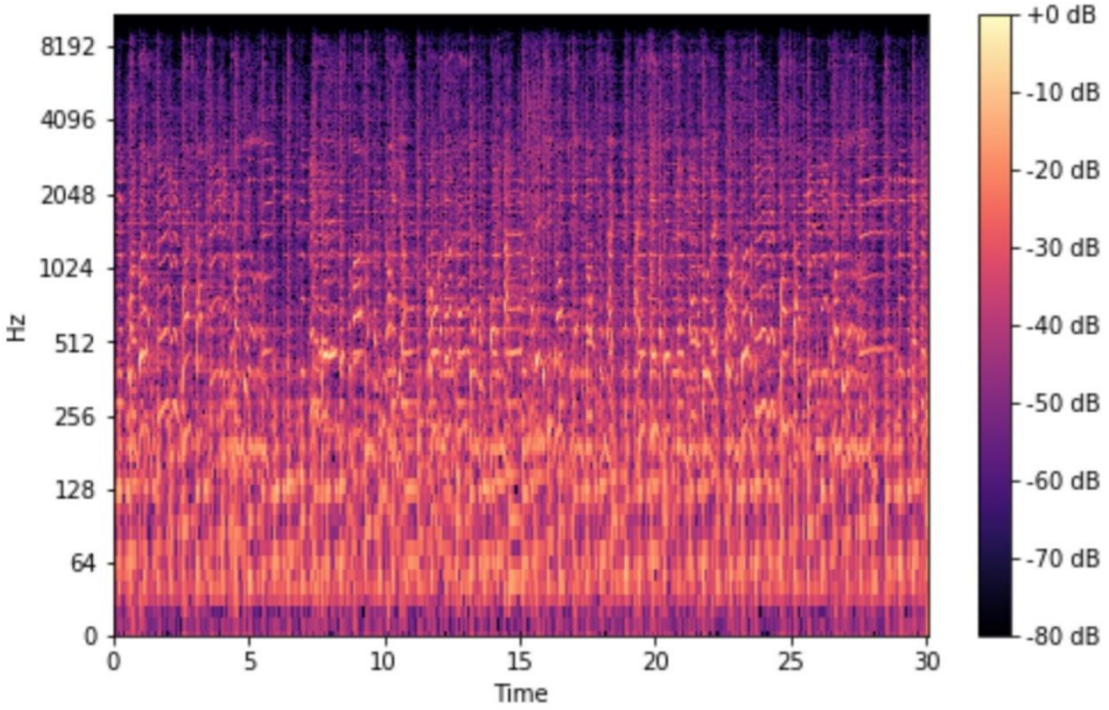
Figure 43 : Zoom forme d'onde



Fait à l'aide de l'outil : Audacity

Cette seconde image illustre le spectrogramme correspondant au signal dans le domaine de fréquence.

Figure 44 : Spectrogramme



Source : Roberts, 2020

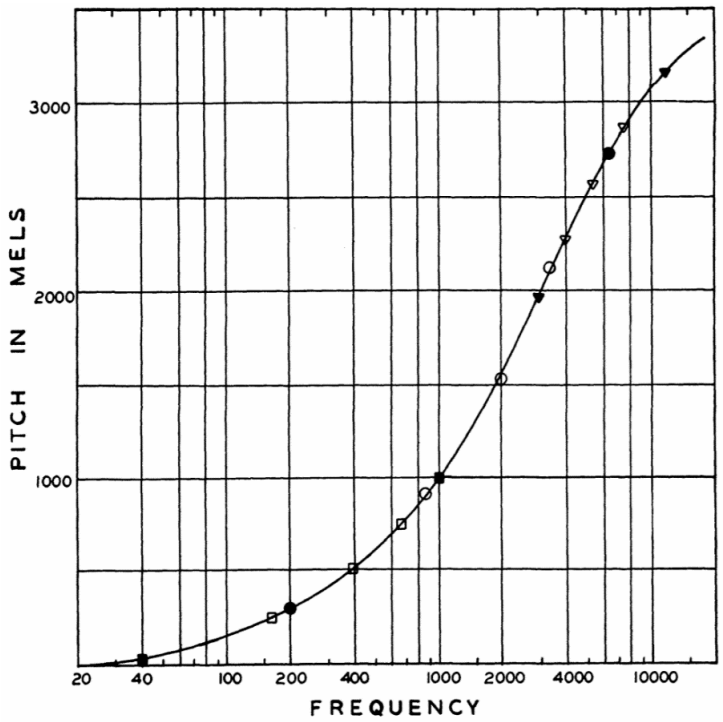
Les spectrogrammes sont produits à l'aide de transformées de Fourier pour décomposer tout signal en ses fréquences constitutives (Doshi, 2021a).

3.3.2 L'échelle de Mel et le spectrogramme Mel

Les humains détectent mieux les différences dans les basses fréquences que dans les fréquences plus élevées. Par exemple, la différence entre 500 et 1000 Hz est identifiable à l'oreille humaine mais ce n'est pas le cas de la différence entre 10000 et 10500 Hz (Roberts, 2020).

En 1937, Stevens, Volkmann et Newmann ont proposé une échelle de mesure des hauteurs du son jugées par les auditeurs comme égales en distance les unes des autres (Roberts, 2020).

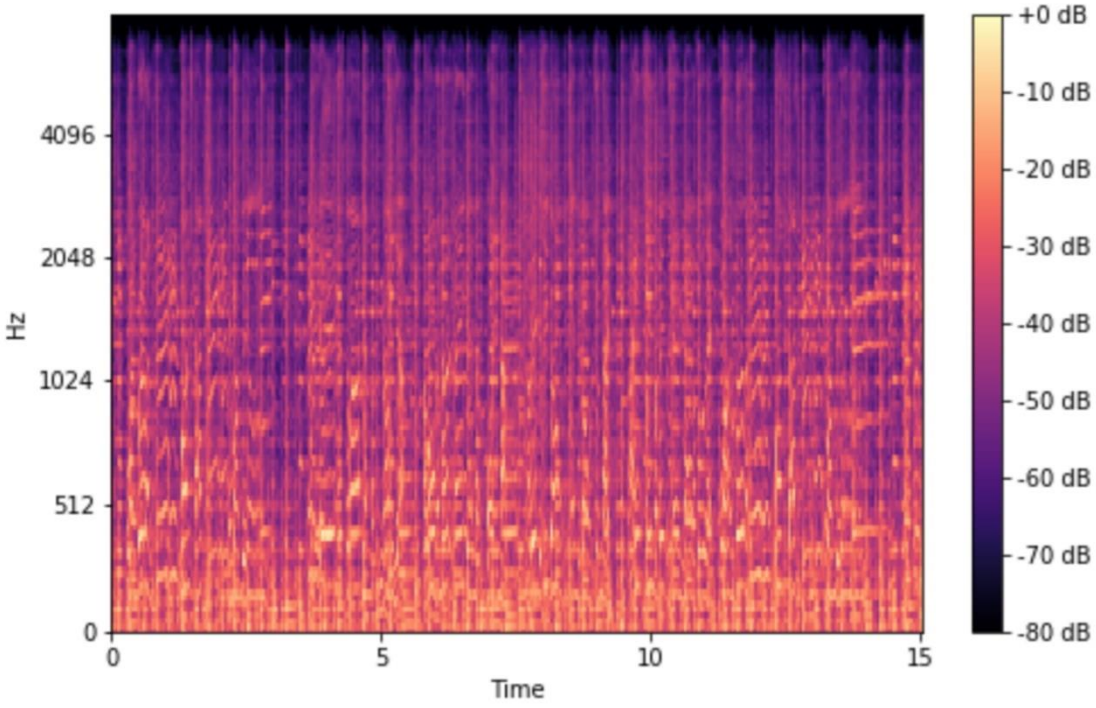
Figure 45 : Echelle des mels



Source : Stevens, Volkmann, 1940

Un spectrogramme mel est un spectrogramme dans lequel les fréquences sont converties en échelle mel (Roberts, 2020).

Figure 46 : Spectrogramme mel



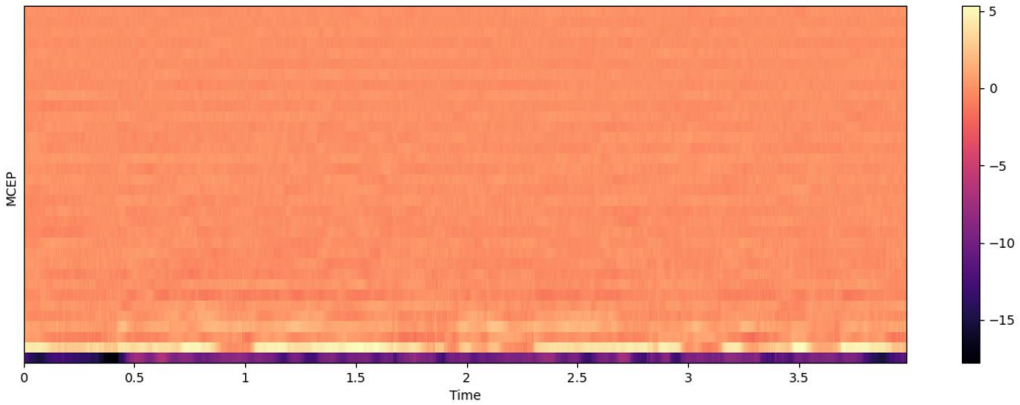
Source : Roberts, 2020

3.3.3 MCEP et MFCC

Pour les problèmes liés à la parole humaine, les Mel-cepstral coefficients (MCEP) et Mel Frequency Cepstral Coefficients (MFCC) sont couramment utilisées dans le deep learning. Il s'agit de représentations compressées des bandes de fréquences du spectrogramme Mel (Doshi, 2021b).

Les MCEP sont principalement utilisés pour effectuer de la synthèse vocale, car une fois obtenus, il est possible de reconstituer la représentation spectrale d'origine (Babu Saheer, 2013).

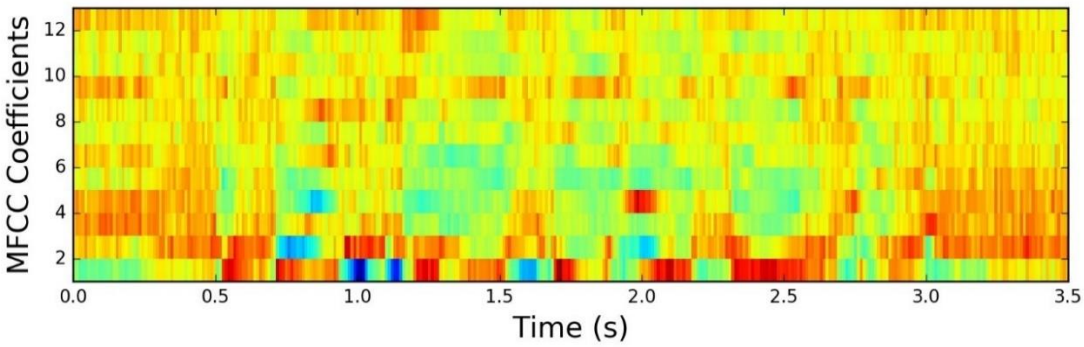
Figure 47 : MCEP



Fait à l'aide de l'outil : Librosa

Les MFCC sont plus couramment utilisés pour identifier les mots prononcés dans un extrait audio et les convertir en texte car ils ont tendance à supprimer les caractéristiques propres à l'orateur en lissant les variations spectrales pour pouvoir maximiser les informations relatives au contenu de l'extrait audio (Babu Saheer, 2013).

Figure 48 : MFCC

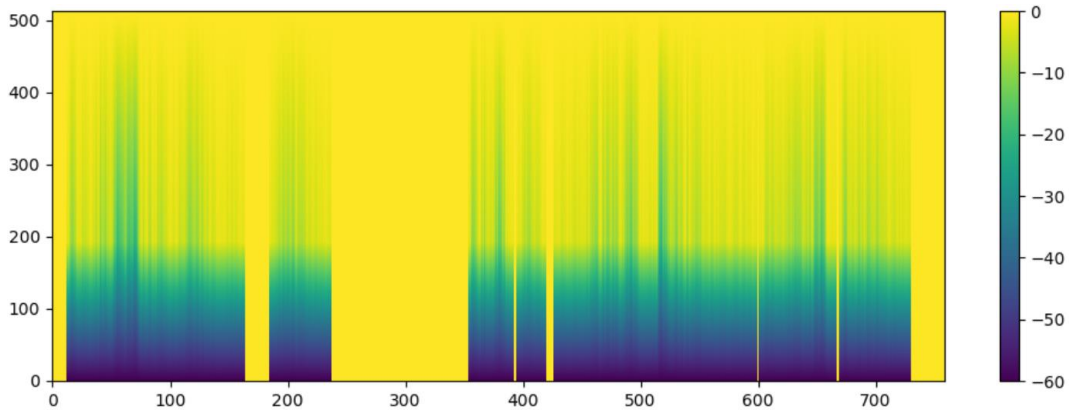


Source : Pratheeksha, 2018

3.3.4 Apériodicité

L'apériodicité est définie comme le rapport entre le signal de parole et la composante apériodique du signal (bruit).

Figure 49 : Apériodicité



Source : WORLD.jl, sans date

Elle est utilisée par les vocoders (outils pour le traitement du signal sonore) pour améliorer la qualité sonore de la parole synthétisée.

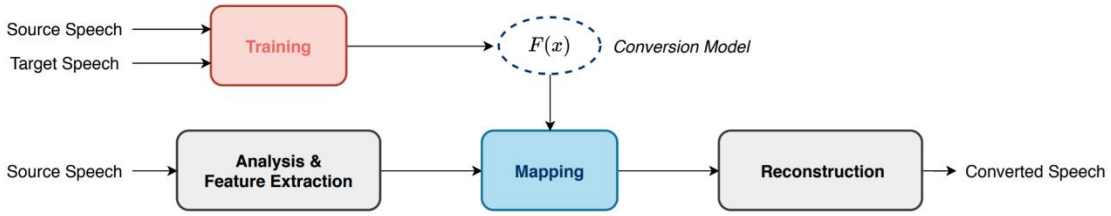
3.4 La conversion de voix

La conversion de voix consiste à modifier l'identité vocale. Plus précisément, le signal vocal prononcé par un orateur source est modifié pour être entendu comme s'il avait été prononcé par un autre orateur, l'orateur cible (Ezzine, 2017).

3.4.1 Principes d'un système de conversion de voix

Un système de conversion vocale modifie uniquement les caractéristiques de la parole dépendant du locuteur, comme la forme spectrale, les formants (zones du spectre d'un signal vocal correspondant à un maximum d'énergie), la fréquence fondamentale (F0), l'intonation, l'intensité et la durée, tout en conservant le contenu de la parole (Sisman, Yamagishi, King et al. 2020).

Figure 50 : Système de conversion de voix



Source : Sisman, Yamagishi, King et al. 2020

Si X et Y correspondent respectivement aux caractéristiques du locuteur source et cible alors la fonction de conversion peut être formulée comme ceci :

$$y = F(x)$$

Le processus standard d'une conversion de voix est composé de trois étapes (Sisman, Yamagishi, King et al. 2020) :

1) Analyse et extraction des caractéristiques de la parole : Décomposer le signal vocal sous la forme d'une représentation intermédiaire pour permettre une manipulation efficace tout en respectant les propriétés acoustiques du signal vocal d'origine. Par exemple, le vocoder Python WORLD permet d'extraire la fréquence fondamentale (F0), l'apériodicité et l'enveloppe spectrale d'un signal vocal. Il permet également d'extraire les coefficients MCEP.

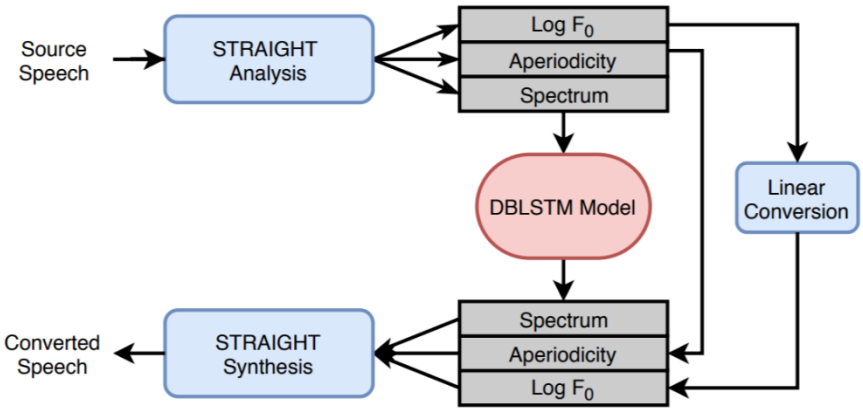
2) Mapping : modification des caractéristiques vocales à partir de la source à l'orateur cible. Par exemple, un réseau de neurones entrainé à la conversion de voix modifiera les coefficients MCEP du locuteur source pour qu'ils puissent reconstituer la représentation spectrale du locuteur cible. Parallèlement, la fréquence fondamentale (F0) du locuteur source sera convertie en une fréquence fondamentale (F0) plus représentative de la voix cible.

3) Reconstruction de la parole : Cette étape peut être vue comme une fonction inverse de la première étape, il s'agit d'utiliser les caractéristiques modifiées lors de l'étape 2 et de générer un signal vocal audible. Par exemple, le vocoder

Python WORLD permet de regénérer le signal vocal à l'aide de la fréquence fondamentale (F0), de l'apériodicité et de l'enveloppe spectrale. WORLD permet également de reconstruire l'enveloppe spectrale depuis les coefficients MCEP.

Illustration de l'extraction des caractéristiques du signal vocal du locuteur source et de sa reconstruction à l'aide du vocoder STRAIGHT (alternative de WORLD) :

Figure 51 : Extraction reconstruction signal vocal STRAIGHT



Source : Zhang, Sisman, Rallabandi et al, 2018

3.4.2 Techniques traditionnelles de conversion de voix

De nombreuses techniques de conversion de voix ont été proposées durant ces dernières années, telles que la transformation par quantification vectorielle, l'alignement fréquentiel dynamique, le modèle de Markov caché ou encore le modèle de mélange Gaussien. La transformation par quantification vectorielle, par exemple, permet de définir automatiquement un dictionnaire de « mapping » contenant la correspondance des caractéristiques acoustiques du locuteur source avec celles du locuteur cible (Ezzine, 2017).

La plupart des techniques de conversion vocale traditionnelles supposent la disponibilité de données d'entraînement parallèles. Autrement dit, la fonction de mappage est formée sur des paires d'extrait vocal de même contenu linguistique parlé par l'orateur source et l'orateur cible (Sisman, Yamagishi, King et al. 2020).

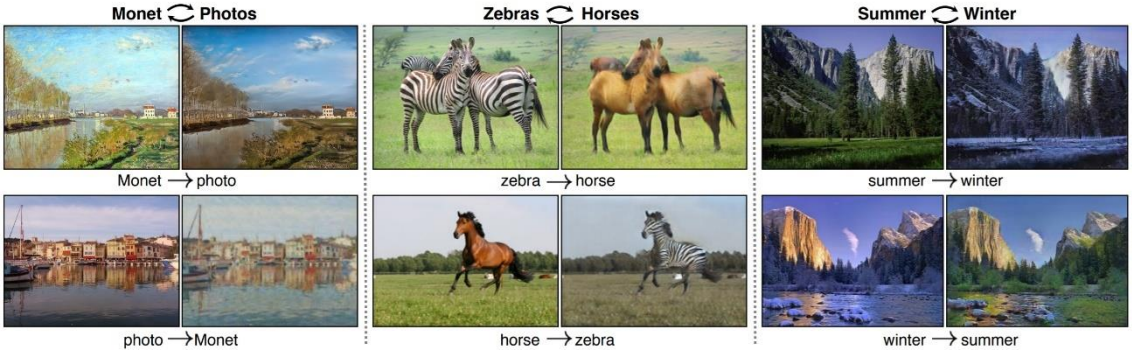
3.4.3 La conversion de voix avec le deep learning

Des études récentes montrent que le deep learning a permis de nombreux scénarios de conversions vocales sans avoir besoin de données parallèles (Sisman, Yamagishi, King et al. 2020).

L’un des concepts proposés, appelé CycleGAN-VC2 (Kaneko, Kameoka, Tanaka et al, 2019), utilise l’approche générative GAN à la conversion de voix.

CycleGAN-VC2 se base sur le modèle CycleGAN (Zhu, Park, Isola et al, 2017) initialement proposé pour convertir des images d’un domaine A vers un autre domaine B et inversement. Voici quelques exemples de conversions réalisés avec CycleGAN :

Figure 52 : Exemples conversions CycleGAN



Source : Zhu, Park, Isola et al, 2017

L’avantage de CycleGAN est qu’il peut être entrainé à l’aide de deux collections d'images totalement différentes.

L’architecture de CycleGAN se compose des éléments suivants :

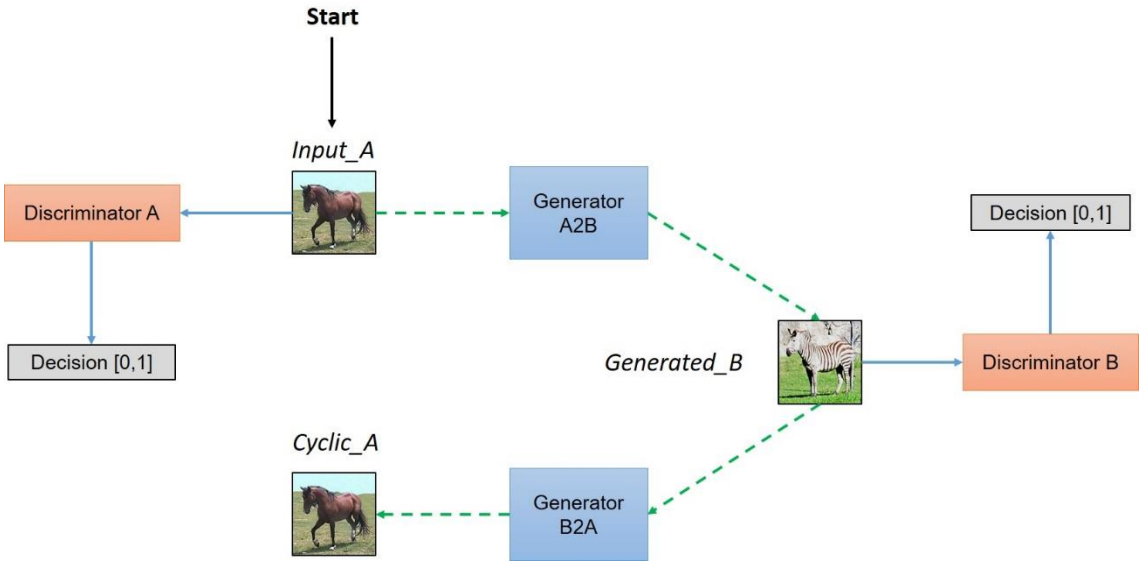
- Un **générateur A2B** chargé de convertir les images du domaine A en images du domaine B
- Un **générateur B2A** chargé de convertir les images du domaine B en images du domaine A
- Un **discriminateur A** chargé de classifier les images comme étant de vraies images du domaine A ou non
- Un **discriminateur B** chargé de classifier les images comme étant de vraies images du domaine B ou non

CycleGAN applique la logique de fonctionnement des GAN, décrite dans le chapitre « Réseaux adverses génératifs », qui consiste à mettre en concurrence un générateur avec un discriminateur pour permettre à chacun de s’améliorer (pour que le générateur améliore la qualité des échantillons générés et que le discriminateur améliore la classification des vraies et des fausses images).

Le processus d’entrainement de CycleGAN se décrit comme ceci :

Une vraie image du domaine A (Input_A) est soumise au discriminateur A et au générateur A2B. Le discriminateur A devra détecter s'il s'agit d'une vraie ou d'une fausse image du domaine A. Le générateur A2B utilisera l'image du domaine A pour générer une image du domaine B (Generated_B). L'image générée du domaine B (Generated_B) est alors utilisée pour entrainer le discriminateur B, qui devra détecter s'il s'agit d'une vraie ou d'une fausse image du domaine B. L'image générée du domaine B (Generated_B) est également envoyée au générateur B2A pour regénérer une image du domaine A (Cyclic_A). Cette image regénérée pourra être directement comparée avec l'image de base (Input_A).

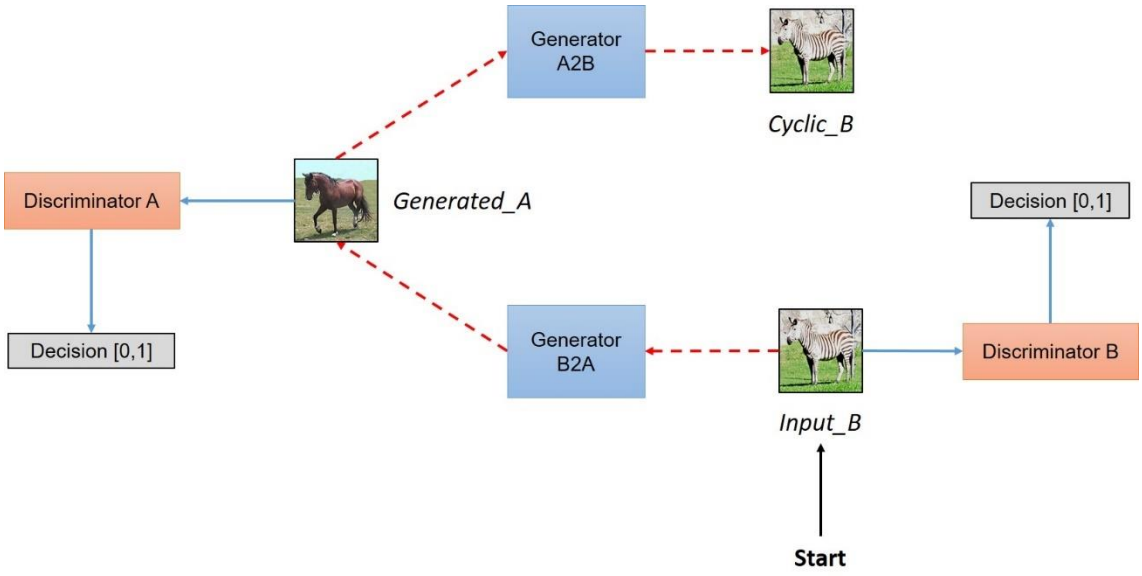
Figure 53 : Entrainement CycleGAN entrée domaine A



Source : Hardik, Archit, sans date

Le même mécanisme est effectué dans l'autre sens à l'aide d'une vraie image du domaine B (Input_B) soumise au discriminateur B et au générateur B2A. L'image générée du domaine A (Generated_A) est alors utilisée par le discriminateur A et par le générateur A2B pour regénérer l'image du domaine B (Cyclic_B) :

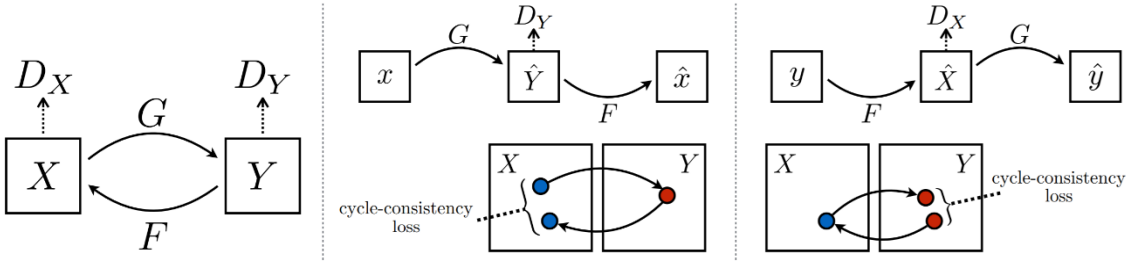
Figure 54 : Entrainement CycleGAN entrée domaine B



Source : Hardik, Archit, sans date

CycleGAN est entrainé et son équilibre cyclique est maintenu grâce aux fonctions de perte décrites ci-dessous (https://hardikbansal.github.io/CycleGANBlog/) :

Figure 55 : Fonctions de perte de CycleGAN

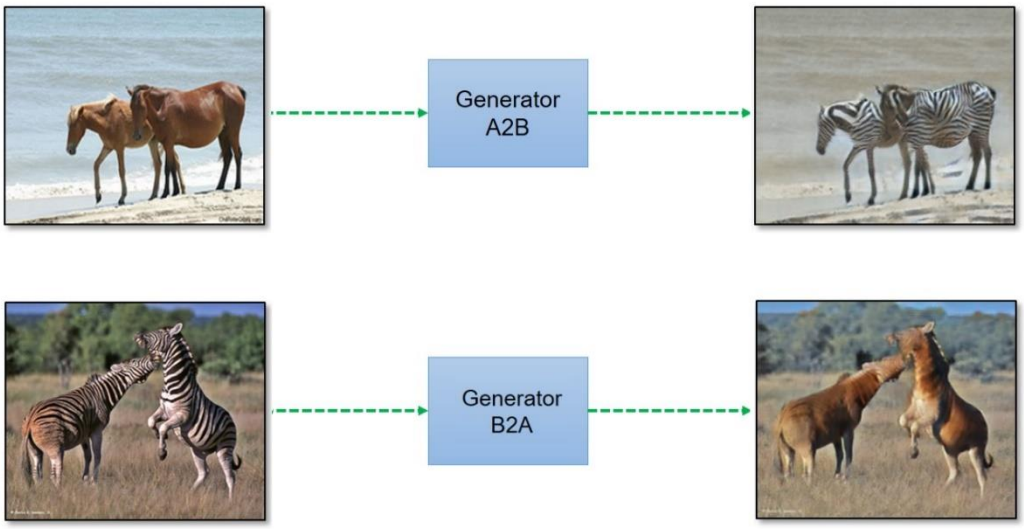


Source : Isola, Zhu, Zhou et al, 2017

- Les discriminateurs doivent approuver toutes les images provenant du jeu d'images d'origine du domaine correspondant ;
- Les discriminateurs doivent rejeter toutes les images générées par les générateurs pour les tromper ;
- Les générateurs doivent tromper les discriminateurs en les obligeant à approuver toutes les images générées ;
- L'image générée doit conserver la propriété de l'image d'origine. Donc si nous générons une fausse image à l'aide d'un générateur, par exemple le discriminateur A2B alors nous devons pouvoir revenir à l'image originale en utilisant l'autre générateur B2A pour garantir une certaine cohérence cyclique.

Une fois suffisamment entrainés, les générateurs sont capables de convertir le domaine d'une image indépendamment des discriminateurs :

Figure 56 : Résultats conversion CycleGAN



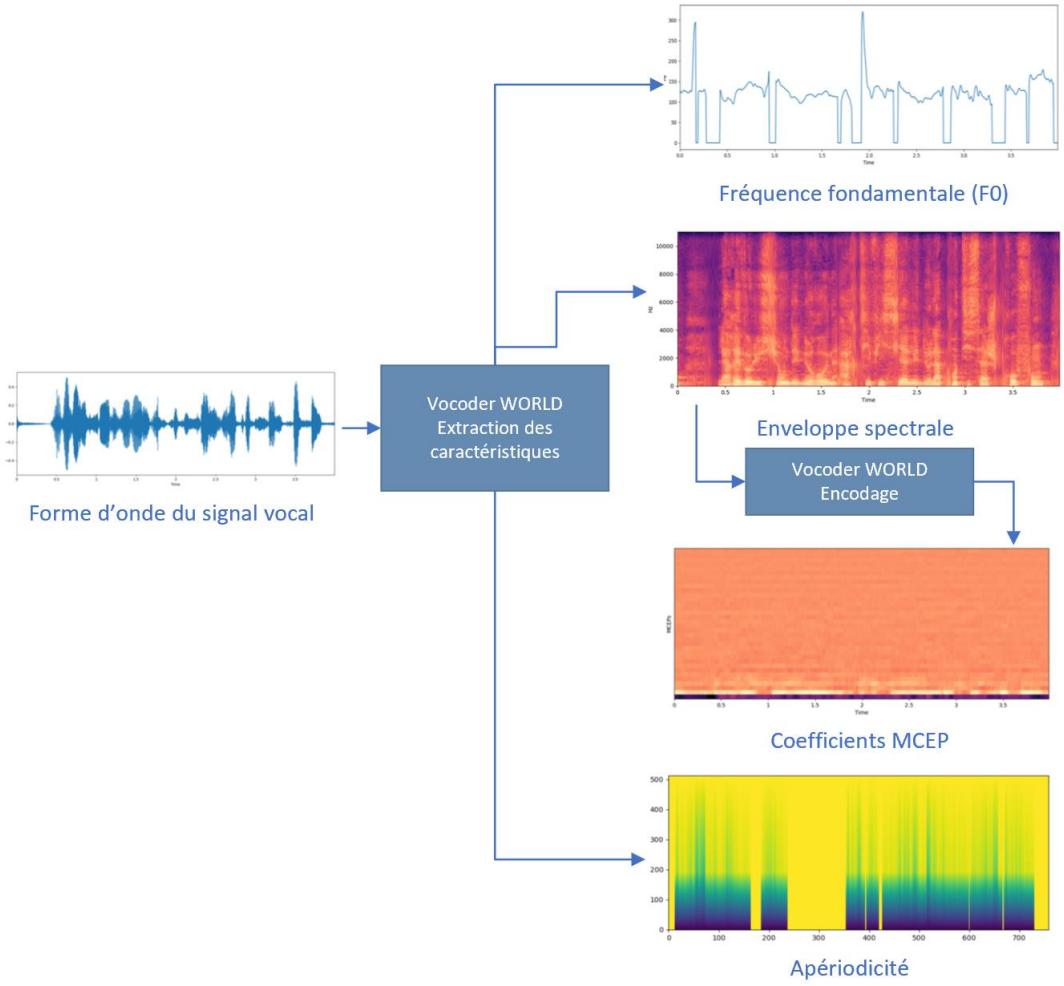
Source : Hardik, Archit, sans date

CycleGAN-VC2 fonctionne sur le même principe que CycleGAN avec quelques différences techniques. Les images utilisées représentent les coefficients MCEP extraits du signal vocal et les domaines correspondent aux différents locuteurs.

Le vocoder WORLD est utilisé pour extraire les différentes caractéristiques suivantes du signal vocal :

- La fréquence fondamentale du signal (F0) ;
- L'enveloppe spectrale, qui est ensuite convertie en coefficients MCEP ;
- L'apériodicité (AP).

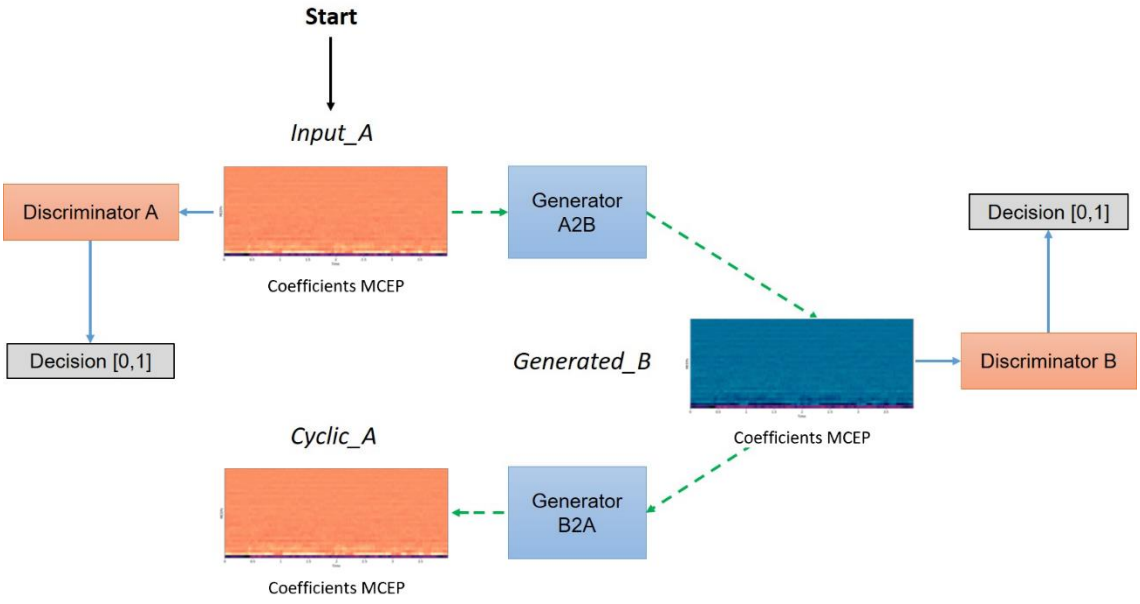
Figure 57 : Extraction caractéristiques signal vocal WORLD



Source : Charbonnier, 2021

Les coefficients MCEP sont ensuite utilisés comme image d'entrée dans l'architecture CycleGAN. Pour les MCEP du locuteur A :

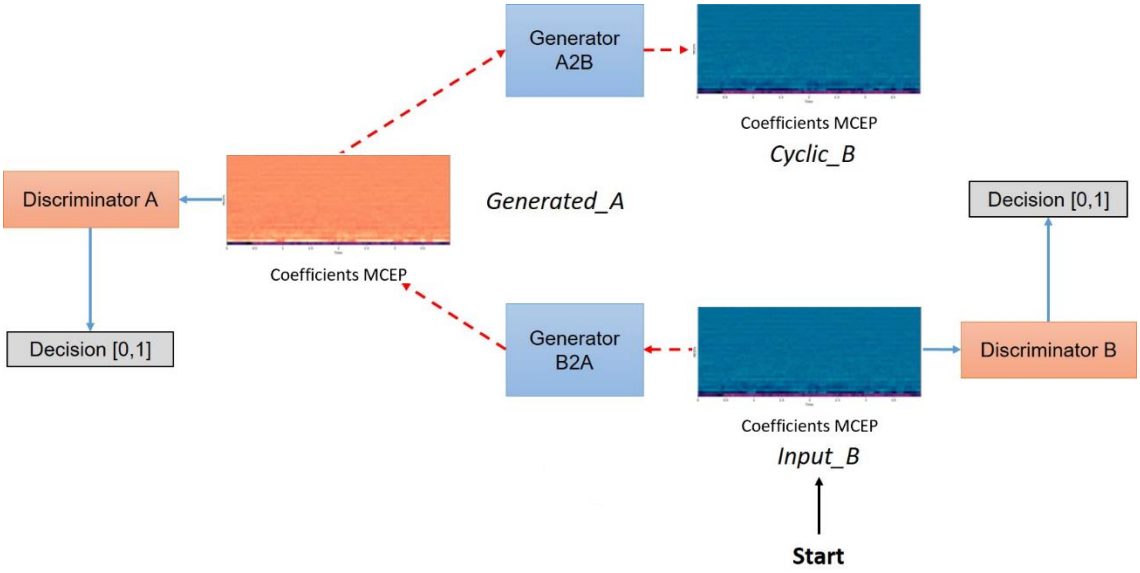
Figure 58 : Entrainement CycleGAN-VC2 entrée domaine A



Source : adapté de Hardik, Archit, sans date

Pour les MCEP du locuteur B :

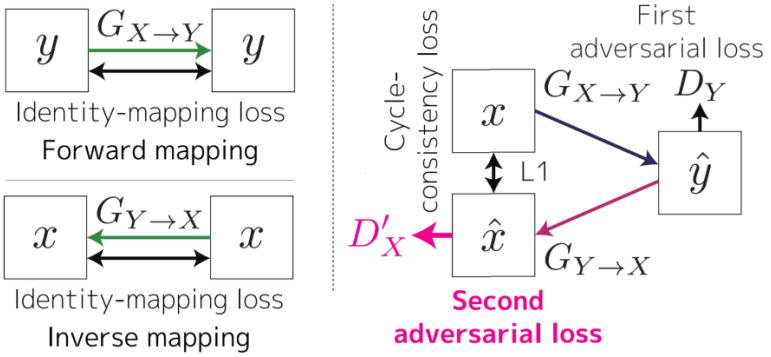
Figure 59 : Entrainement CycleGAN entrée domaine B



Source : adapté de Hardik, Archit, sans date

En plus des fonctions de perte initiales de CycleGAN, CycleGAN-VC2 implémente deux fonctions de perte supplémentaires :

Figure 60 : Fonctions de perte de CycleGAN-VC2

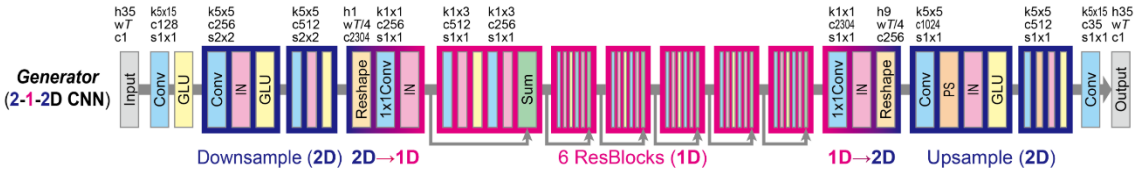


Source : Kaneko, Kameoka, Tanaka et al, 2019

- Une image générée à l'aide du générateur A2B depuis une image B doit conserver la propriété de l'image d'origine et inversement. Cela dans le but de préserver la composition de l'image entre l'entrée et la sortie ;
- Le discriminateur A doit approuver toutes les images A ayant été transformées par le générateur A2B et le générateur B2A et inversement. Cette règle atténuerait l'effet de lissage excessif d'une double conversion.

Enfin le CycleGAN-VC2 implémente un générateur constitué d'un réseau de neurones convolutif :

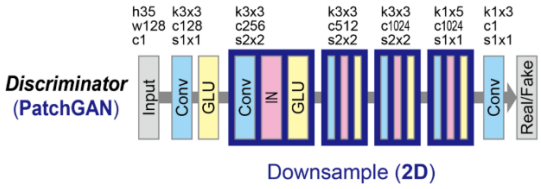
Figure 61 : Architecture générateur CycleGAN-VC2



Source : Kaneko, Kameoka, Tanaka et al, 2019

Et un discriminateur PatchGAN (qui utilise une évaluation par morceau d'image) :

Figure 62 : Architecture discriminateur CycleGAN-VC2

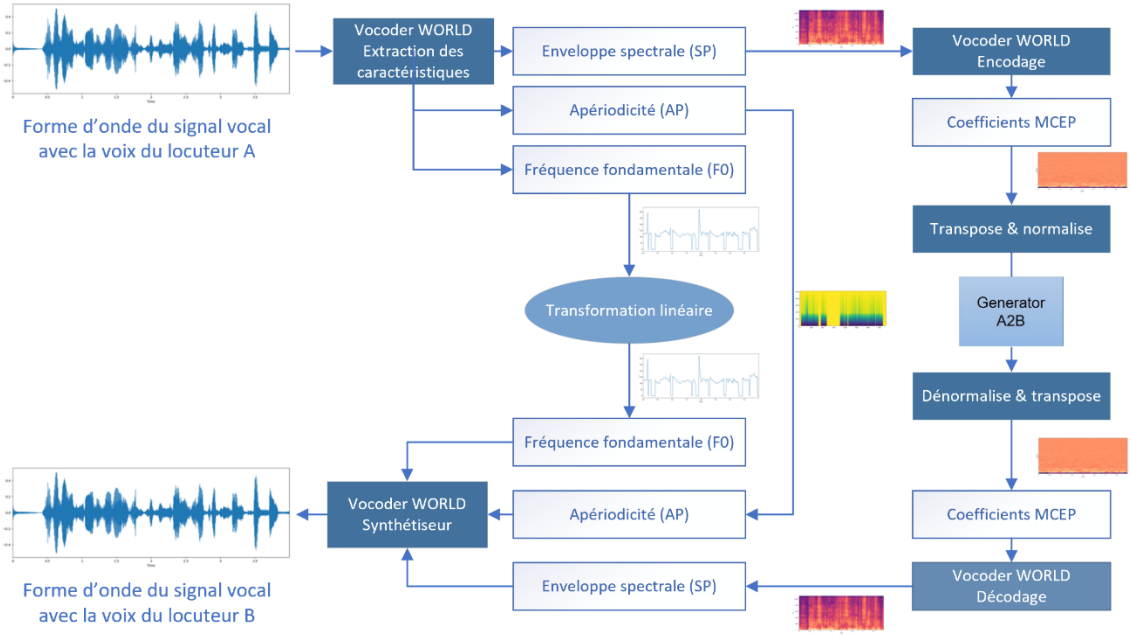


Source : Kaneko, Kameoka, Tanaka et al, 2019

Une fois CycleGAN-VC2 suffisamment entrainé, il est alors possible d'utiliser les générateurs indépendamment des discriminateurs pour convertir la voix des locuteurs en effectuant l'opération suivante :

Le vocoder WORLD est utilisé sur la forme d'onde du signal vocal du locuteur A pour extraire l'enveloppe spectrale (SP), l'apériodicité (AP) et la fréquence fondamentale (F0). L'enveloppe spectrale est encodée en coefficients MCEP. La matrice contenant ces coefficients MCEP est transposée, les valeurs sont normalisées puis fournies comme entrées au générateur A2B.

Figure 63 : Extraction conversion reconstruction signal vocal CycleGAN-VC2



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

La même opération est effectuée pour convertir la voix du locuteur B en locuteur A à l'aide du générateur B2A.

4. Cas pratiques

Je commencerai par développer un GAN « from scratch » avec une architecture simple basée sur une fonction à une dimension (1D) pour me familiariser avec les concepts fondamentaux des GAN et me confronter aux difficultés liées à l'entrainement du générateur et du discriminateur.

J'adapterai ensuite cette architecture pour obtenir un DCGAN en intégrant des réseaux de neurones convolutifs et en générant des lettres manuscrites à l'aide du jeu de données MNIST (base de données de chiffres écrits à la main).

Puis je ferai évoluer cette architecture, en me basant sur le concept CycleGAN-VC2, pour pouvoir réaliser des conversions de voix.

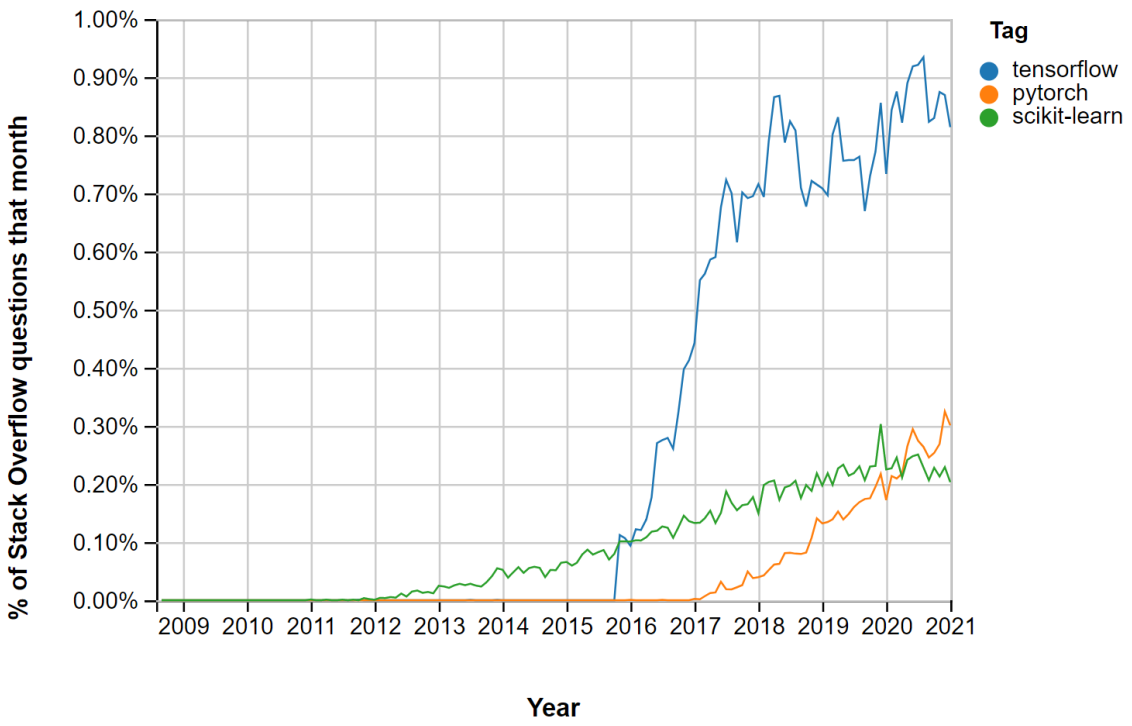
Enfin, à l'aide des conversions obtenues, je testerai le système de reconnaissance vocale de Google (Google Voice Match).

Le code source développé pour les cas pratiques est disponible sur mon dépôt GitHub : https://github.com/charbonnier-fred/GAN-TB-2021-HEG

4.1 TensorFlow

Parmi les frameworks les plus populaires en machine learning, on retrouve TensorFlow, PyTorch ou encore scikit-learn. Voici le pourcentage de questions relatives à ces trois frameworks sur Stackoverflow :

Figure 64 : Popularité framework machine learning



Source : Stack Overflow Trends, 2021

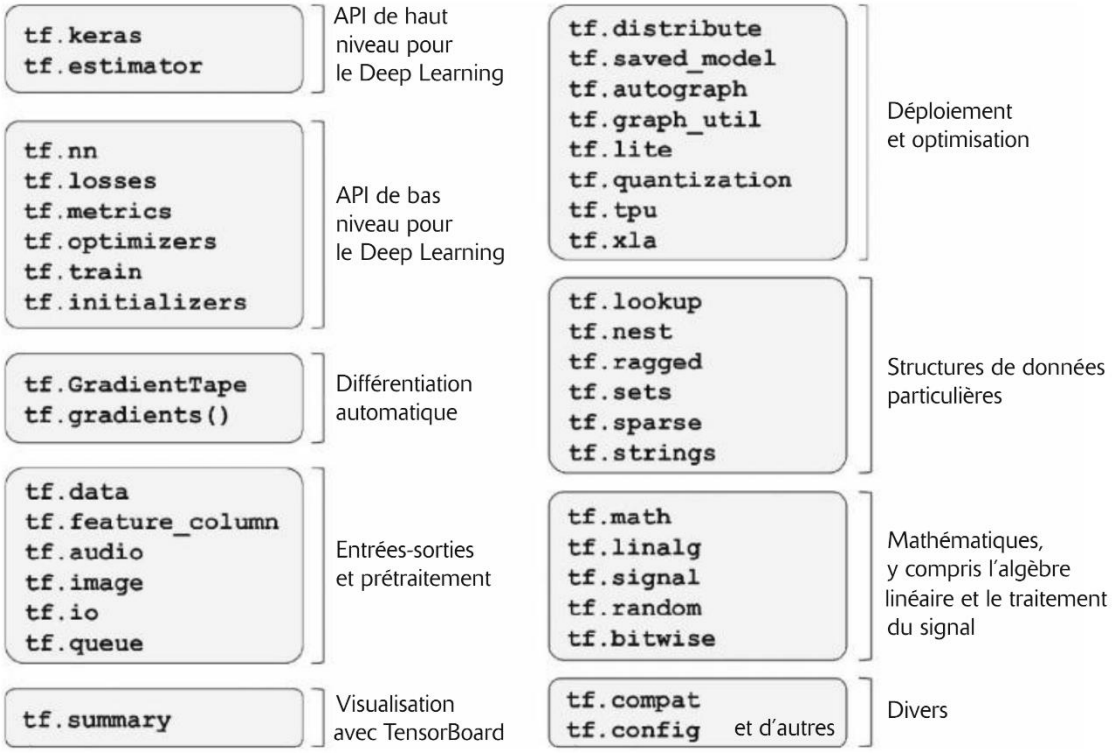
En raison de sa forte popularité, j'utiliserai le framework TensorFlow pour réaliser les cas pratiques de ce travail de Bachelor. A noter que TensorFlow fournit des API stables en Python et en C (TensorFlow, 2021), j'opterai donc pour le langage Python.

TensorFlow est un framework logiciel puissant destiné au calcul numérique. Il est particulièrement bien adapté et optimisé pour l'apprentissage automatique à grande échelle, mais il peut être employé également à toutes autres tâches qui nécessitent de lourds calculs (Geron, 2019).

Initialement développé par l'équipe Google Brain, il se trouve au cœur de nombreux services à grande échelle de Google, comme Google Cloud Speech, Google Photos et Google Search. Ce framework est passé en open source en novembre 2015 et fait à présent partie des bibliothèques de deep learning les plus populaires. De nombreux projets utilisent Tensorflow pour toutes sortes de tâches d'apprentissage automatique, comme par exemple la classification d'images, le traitement automatique du langage naturel ou encore les systèmes de recommandation (Geron, 2019).

Tensorflow offre de nombreuses fonctionnalités, telles que la construction, l'entraînement, l'évaluation et l'exécution de toutes sortes de réseaux de neurones (tf.keras), des outils de chargement et de prétraitement des données (tf.data, tf.io), des outils de traitement d'images (tf.image), des outils de traitement du signal (tf.signal), etc… Voici une vue d'ensemble de l'API Python de TensorFlow:

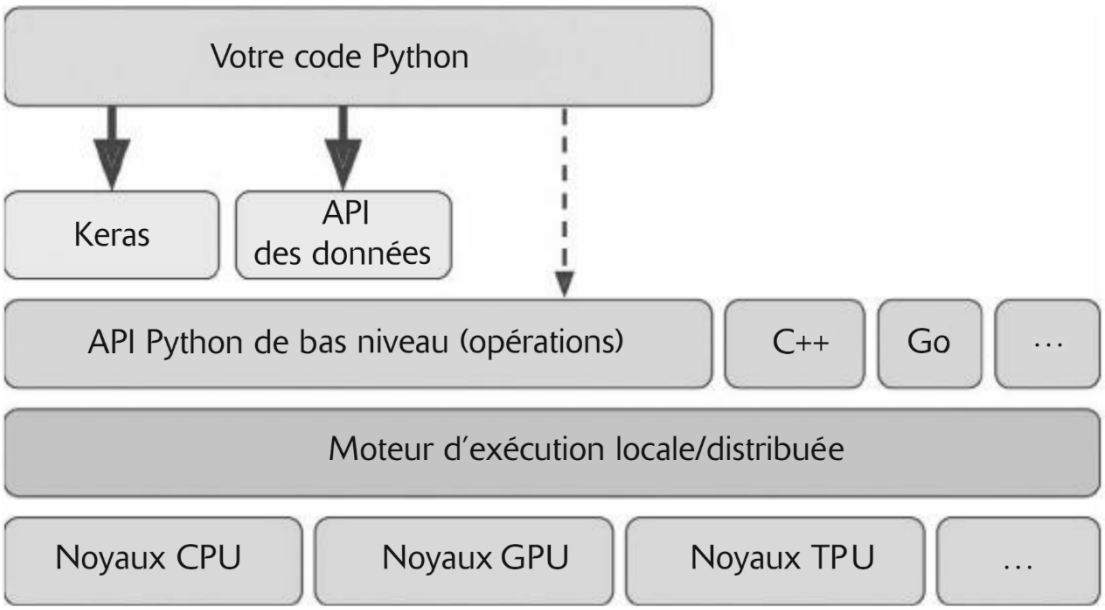
Figure 65 : API Python de TensorFlow



Source : Geron, 2019

L'architecture de TensorFlow est illustrée comme ceci :

Figure 66 : Architecture de TensorFlow



Source : Geron, 2019

Dans de nombreux cas, le code exploitera les API de haut niveau (en particulier tf.keras et tf.data). Mais s'il est nécessaire d'avoir plus de flexibilité, alors le code utilisera l'API Python de bas niveau, en gérant directement les tableaux multidimensionnels de TensorFlow (appelé « tenseurs »). A noter que le moteur d'exécution de TensorFlow se charge d'exécuter efficacement les opérations sur les processeurs et machines concernées. Chaque noyau est dédié à un type de processeur spécifique, principalement les CPU, les GPU et les TPU (Geron, 2019).

Le CPU désigne, la plupart du temps, l'unité centrale d'un ordinateur. Elle est très polyvalente et à une très faible latence pour les opérations arithmétiques, ce qui lui permet d'être utilisée avec de nombreuses applications. Cependant, elle a un faible débit lorsqu'il s'agit de réaliser de nombreuses opérations arithmétiques sur beaucoup de nombres en même temps car elle effectue les calculs séquentiellement, ce qui limite fortement le débit lors de l'entrainement de réseaux de neurones qui nécessite de nombreux calculs (Naushad, 2020).

Le GPU est une unité de traitement graphique. Elle est spécialisée pour réaliser des calculs en parallèle. Elle a une latence plus élevée pour les opérations arithmétiques que le CPU mais elle a un haut débit lors de la réalisation d'opérations arithmétiques en simultanée, ce qui est favorable pour l'entrainement de réseau de neurones (Naushad, 2020).

Le TPU a une architecture dédiée à l'entrainement des réseaux de neurones. Elle offre donc un très haut débit lors de la réalisation de calculs de manière simultanée (Naushad, 2020). Cependant, certaines techniques d'apprentissage ne sont pas compatibles avec les abstractions logicielles des TPU (Google Cloud, 2021).

Afin de bénéficier d'une vitesse favorable à l'entrainement des réseaux de neurones sans être confronté aux problématiques de compatibilité, je privilégierai l'utilisation de GPU pour la réalisation de mes cas pratiques.

Pour suivre et visualiser les différentes métriques lors de l'entrainement des réseaux de neurones, j'utiliserai le kit de visualisation de TensorFlow appelé TensorBoard (TensorBoard, sans date).

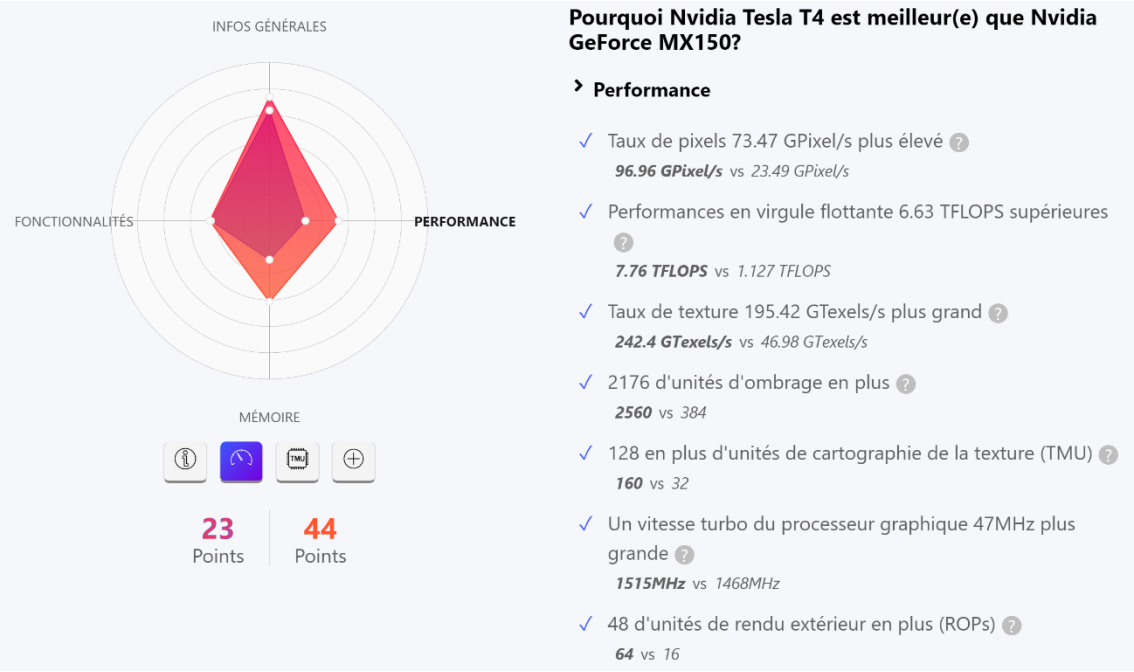
4.2 Environnement

Pour entrainer des réseaux de neurones « simples », j'utiliserai la carte graphique de mon PC personnel, une NVIDIA GeForce MX150.

Cependant, pour entrainer des réseaux de neurones plus élaborés, comme pour le traitement de données audio, j'utiliserai une carte graphique adaptée au calcul de réseaux de neurones, sur Google Cloud Platform, une NVIDIA Testa T4.

Voici le résumé d'un comparatif des performances entre une NVIDIA GeForce MX150 (rouge) et une NVIDIA Testa T4 (orange) :

Figure 67 : Comparatif performances cartes NVIDIA

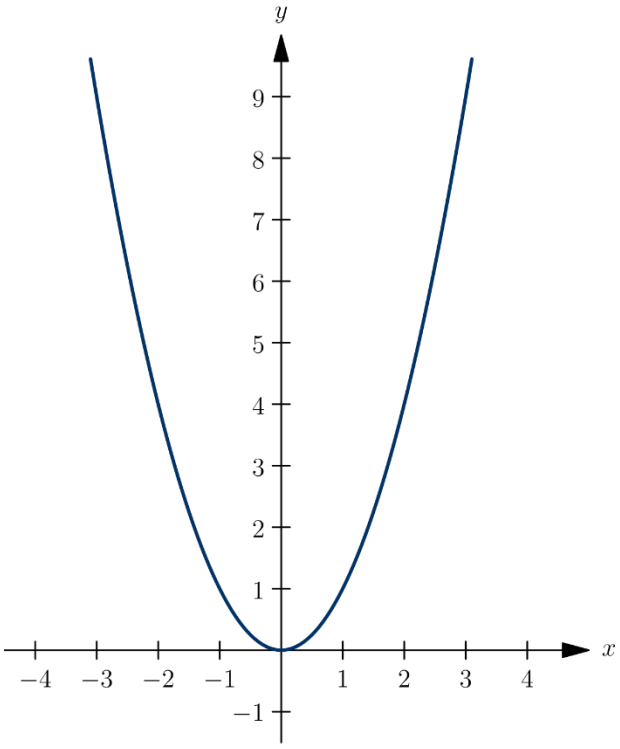


Source : Nvidia GeForce MX150 vs Nvidia Tesla T4, sans date

4.3 GAN 1D

Pour ce premier cas pratique, nous allons, à l'aide des guides et tutoriels officiels de TensorFlow (TensorFlow Core, 2021), ainsi que du livre « Generative Adversarial Networks with Python » de Jason Brownlee (Brownlee, 2019b), mettre en place un GAN simple basé sur une fonction à une dimension (1D), la fonction carré $f(x) = x^2$ illustrée ci-dessus.

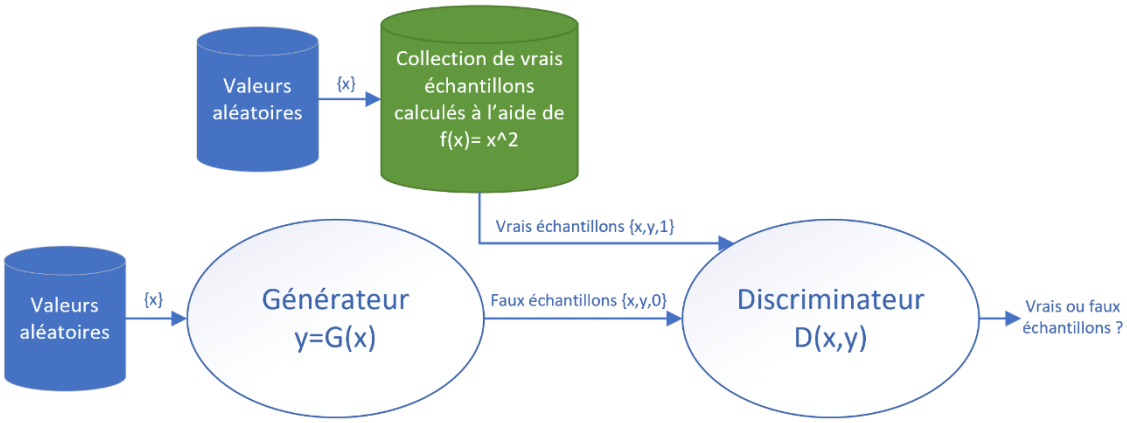
Figure 68 : Fonction carré



Source : Fonction carré, 2021

Voici l'architecture du GAN 1D à développer :

Figure 69 : Architecture GAN 1D



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Les vrais échantillons $\{x, y\}$ fournis au discriminateur seront générés à l'aide de la fonction carré. L'objectif sera d'entrainer le générateur et le discriminateur en concurrence afin que le générateur soit suffisamment entrainé pour réussir à générer des échantillons $\{x, y\}$ se situant sur la ligne illustrée par la fonction carré.

4.3.1 Développement

Je commence par implémenter la fonction carré décrite précédemment :

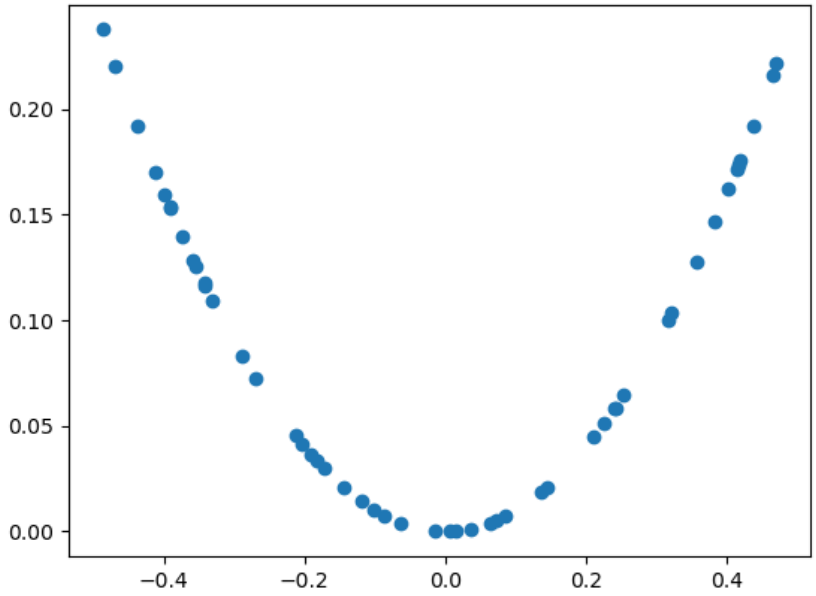
```
#Fonction carré
def fcarre(x):
    return x * x
```

Pour pouvoir produire les vrais échantillons qui seront fourni au discriminateur, je mets en place une fonction qui utilise une valeur aléatoire pour x et calcule y à l'aide de la fonction carré :

```
# Génère n vrais échantillons
def generate_real_samples(n):
    # Génère un vecteur d'entrées entre -0.5 et 0.5
    X = rand(n) - 0.5
    # Génère le vecteur de sortie de fcarre(x)
    Y = fcarre(X)
    # Transforme les vecteurs en matrice 1 dimension
    X = X.reshape(n, 1)
    Y = Y.reshape(n, 1)
    # Concatène les 2 matrice 1 dimension
    # pour obtenir une matrice à 2 dimension
    XY = hstack((X, Y))
    # Crée une matrice 1 dimension remplie de 1 pour
    # indiquer qu'il s'agit de vrais échantillons
    Z = ones((n, 1))
    return XY, Z
```

Voici une illustration de 50 vrais échantillons générés à l'aide de cette fonction :

Figure 70 : GAN 1D 50 vrais échantillons



Fait à l'aide de l'outil : TensorBoard

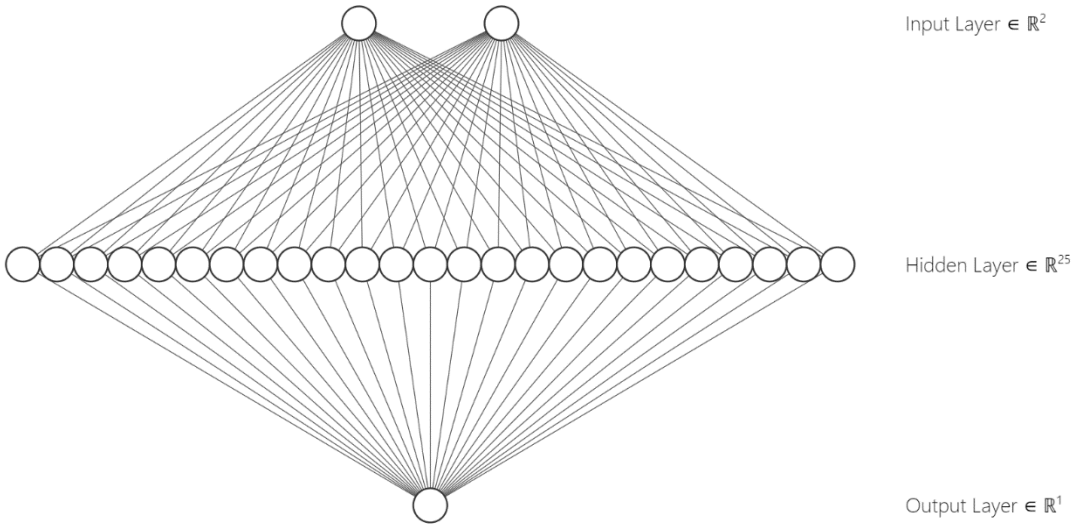
Je mets ensuite en place le discriminateur. Son objectif sera de classifier les points en deux catégories : « Vrai » pour les points qui se situent sur l'axe de la fonction carré (vrais échantillons) et « Faux » pour les autres points placés aléatoirement (faux échantillons).

Le discriminateur recevra deux valeurs d'entrées pour pouvoir traiter les coordonnées x et y de chaque point. Il retournera un pourcentage représentant l'erreur de prédiction de la classification « Vrai/Faux ».

Jason Brownlee nous indique que nous n'avons pas besoin d'un réseau de neurones complexe pour modéliser ce discriminateur car il s'agit d'un problème assez simple (Brownlee, 2019). Il conseille d'utiliser une seule couche cachée dans laquelle chaque neurone sera connecté à tous les neurones de la couche précédente (appelée couche Dense), composée de 25 nœuds avec une fonction d'activation ReLU et d'initialiser les poids à l'aide d'une distribution spécifique (appelée initialisation He). Pour la couche de sortie, nous utiliserons un seul nœud avec une fonction d'activation sigmoïde.

```python
def discriminator():
  # Entrée à 2 valeurs
  inp = Input(shape=(2,), name='input_sample')
  x = inp
  # Unique couche cachée Dense de 25 noeuds avec une fonction d'activation ReLU
  # et une méthode d'initialisation des poids He
  x = Dense(25, activation="relu", kernel_initializer="he_uniform", name="dense1")(x)
  # Couche de sortie avec une fonction d'activation sigmoïde
  last = Dense(1, activation="sigmoid", name="output")(x)
  return Model(inputs=inp, outputs=last)
```

Figure 71 : GAN 1D architecture du discriminateur



Fait à l'aide de l'outil : https://alexlenail.me

Je peux maintenant instancier le discriminateur et compiler le modèle à l'aide de trois paramètres, un algorithme d'optimisation (pour la descente du gradient), une fonction de perte et un choix de métriques (pour juger la performance du modèle). Comme proposé par Jason Brownlee, nous utiliserons l'algorithme d'optimisation appelé adam, la fonction de perte appelée cross-entropy binaire et les métriques appelées accuracy (Brownlee, 2019b).

```python
discriminateur.compile(optimizer="adam",loss="binary_crossentropy",metrics=["accuracy"])
```

Passons maintenant au développement du générateur.

Pour rappel, l'objectif du générateur est de générer un nouvel échantillon $\{y, x\}$ le plus réaliste possible sur la base d'un point x généré aléatoirement.

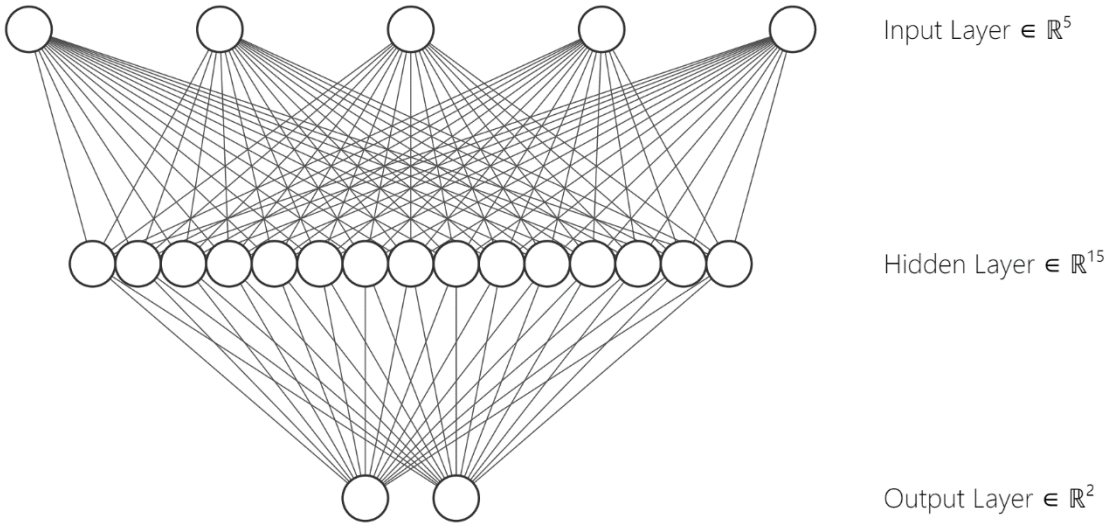
Pour créer le générateur, Jason Brownlee recommande d'utiliser une seule couche cachée Dense composée de 15 nœuds avec une fonction d'activation ReLU une initialisation des poids avec la méthode d'initialisation He (Brownlee, 2019b). La couche

de sortie sera composée de deux nœuds pour permettre d'obtenir les valeurs de x et de y qui composeront chaque échantillon, chacun de ces nœuds utilisera une fonction d'activation linéaire. Cette fonction d'activation a été choisie car la sortie souhaitée est une valeur réelle entre -0.5 et 0.5 pour le premier élément et entre 0.00 et 0.25 pour le deuxième élément. Enfin, la couche d'entrée variera en fonction du nombre de valeurs aléatoires reçues.

```
def generator(latent_dim):
  # Entrée à "latent_dim" valeurs
  inp = Input(shape=(latent_dim,), name='input_sample')
  x = inp
  # Ajout de la couche cachée de 15 noeuds avec une fonction d'activation ReLU
  # et la méthode d'initialisation des poids He
  x = Dense(15, activation="relu", kernel_initializer="he_uniform", name="dense1")(x)
  # Ajout de la sortie avec la fonction d'activation linéaire
  last = Dense(2, activation="linear", name="output")(x)
  return Model(inputs=inp, outputs=last)
```

Illustration du générateur avec un espace latent d'une dimension de 5 :

Figure 72 : GAN 1D architecture du générateur



Fait à l'aide de l'outil : https://alexlenail.me

Maintenant que le générateur est créé, il faut définir une fonction permettant de récupérer des points générés aléatoirement.

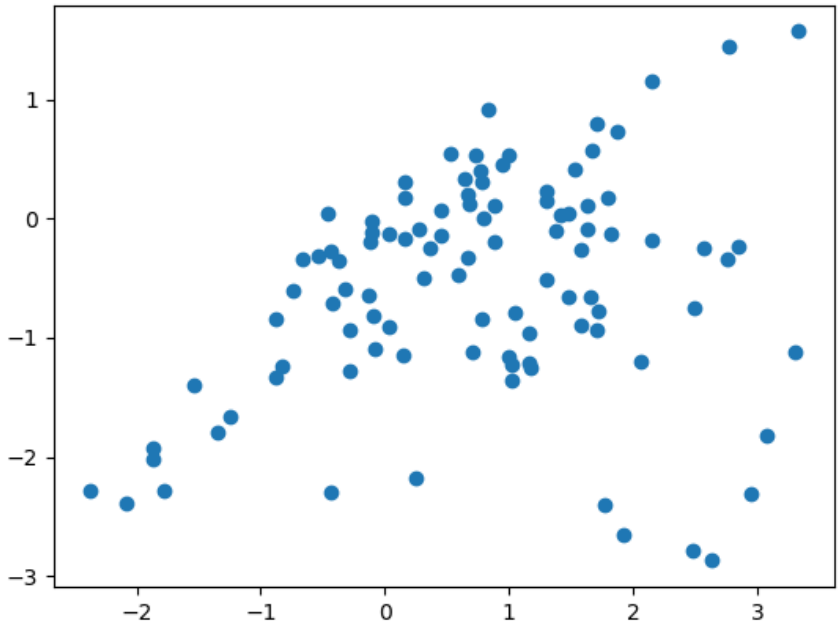
```
# Générer les points dans l'espace latent
def generate_latent_points(latent_dim, n):
    # Génère les points à l'aide de la fonction
    # randn() utilisant la distribution gaussienne
    x_input = randn(latent_dim * n)
    # Remodèle dans une matrice à plusieurs dimensions
    x_input = x_input.reshape(n, latent_dim)
    return x_input
```

Je peux maintenant utiliser les points générés aléatoirement comme entrée du générateur puis générer des faux échantillons. J'ajoute donc la nouvelle fonction suivante :

```
#Génère n faux échantillons à l'aide du générateur
def generate_fake_samples(generator, latent_dim, n):
    #Génère les points de l'espace latent
    x_input = generate_latent_points(latent_dim[0], n)
    #Génère les échantillons à l'aide du générateur
    x = generator.predict(x_input)
    #Crée une matrice 1 dimension remplie de 0 pour
    #indiquer qu'il s'agit de faux échantillons
    y = zeros((n, 1))
    return x, y
```

Voici 100 faux échantillons générés par le générateur :

Figure 73 : GAN 1D 100 faux échantillons



Fait à l'aide de l'outil : TensorBoard

Pour l'instant le générateur n'est pas encore entrainé, il s'agit du résultat obtenu à l'aide du réseau de neurones du générateur avec ses poids d'origines.

Pour former le générateur, je crée une nouvelle classe appelée GAN.

Son constructeur permettra de définir différents paramètres, tels que le nombre de tours complets sur le jeu d'entrainement (appelé époques), la taille du découpage du jeu d'entrainement (appelé lots), la dimension de l'espace latent, le taux d'apprentissage du discriminateur, le taux d'apprentissage du générateur ou encore le répertoire de log utilisé par TensorBoard.

```
class GAN(object):
  # Constructeur
  def __init__(self, epochs, batch_size, latent_dim, d_learning_rate, g_learning_rate, logdir):
    self.epochs = epochs
    self.batch_size = batch_size
    self.latent_dim = latent_dim
    # Instanciation de l'optimiseur du discriminateur
    self.d_optimizer = tf.keras.optimizers.Adam(learning_rate=d_learning_rate)
    # Instanciation de l'optimiseur du générateur
    self.g_optimizer = tf.keras.optimizers.Adam(learning_rate=g_learning_rate)
    self.logdir = logdir
    self.discriminator = discriminator()
    self.generator = generator(latent_dim)
    # Instanciation de la fonction de perte du discrinateur
    self.loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
    # Mesures de performance
    self.g_accuracy = tf.keras.metrics.BinaryAccuracy()
    self.d_accuracy = tf.keras.metrics.BinaryAccuracy()
    self.d_real_accuracy = tf.keras.metrics.BinaryAccuracy()
    self.d_generated_accuracy = tf.keras.metrics.BinaryAccuracy()
```

La perte du générateur, qui sera utilisée par l'optimiseur du générateur pour chercher à minimiser cette perte, sera calculée dans la méthode generator_loss() à l'aide des faux échantillons ayant été classifiés comme « vrai » par le discriminateur.

```
def generator_loss(self, generated_output):
  # Calcul de la perte du générateur à l'aide des faux échantillons
  # ayant été classifiés comme "vrai" par le discriminateur
  return self.loss_fn(tf.ones_like(generated_output), generated_output)
```

La perte du discriminateur sera calculée dans la méthode discriminator_loss() à l'aide des vrais échantillons ayant été classifiés comme « vrai » et des faux échantillons ayant été classifiés comme « faux ».

```python
def discriminator_loss(self, real_output, generated_output):
  # Calcul de la perte du discriminateur à l'aide des vrais échantillons
  # ayant été classifiés comme "vrai" et des faux échantillons ayant
  # été classifiés comme "faux"
  real_loss = self.loss_fn(tf.ones_like(real_output), real_output)
  generated_loss = self.loss_fn(tf.zeros_like(generated_output), generated_output)
  total_loss = real_loss + generated_loss
  return total_loss
```

Pour pouvoir visualiser la performance du générateur et du discriminateur, afin de pouvoir par la suite calibrer correctement l'évolution en concurrence des deux réseaux, je définis les méthodes ci-dessous :

```python
def generator_accuracy(self, generated_output):
  # Calcul de la performance du générateur à l'aide des faux échantillons
  # ayant été classifiés comme "vrai" par le discriminateur
  self.g_accuracy.reset_states()
  return self.g_accuracy(tf.ones_like(generated_output), generated_output)

def discriminator_accuracy(self, real_output, generated_output):
  # Calcul de la performance du discriminateur à l'aide des échantillons réels
  # ayant été classifiés comme "vrai" et des faux échantillons générés par
  # le générateur ayant été classifiés comme "faux"
  self.d_accuracy.reset_states()
  return self.d_accuracy(tf.concat([tf.ones_like(real_output), tf.zeros_like(generated_output)], 0),
    tf.concat([real_output, generated_output], 0))

def discriminator_real_accuracy(self, real_output):
  # Calcul de la performance du discriminateur à classifier
  # les vrais échantillons à "vrai"
  self.d_real_accuracy.reset_states()
  return self.d_real_accuracy(tf.ones_like(real_output), real_output)

def discriminator_generated_accuracy(self, generated_output):
  # Calcul de la performance du discriminateur à classifier
  # les faux échantillons à "faux"
  self.d_generated_accuracy.reset_states()
  return self.d_generated_accuracy(tf.zeros_like(generated_output), generated_output)
```

Pour pouvoir réaliser l'entrainement du GAN par lot, je définis une méthode train_step() qui effectuera les étapes suivantes pour chaque lot:

1) Le générateur génère des faux échantillons
2) Le discriminateur classifie les faux échantillons et les vrais échantillons
3) Calcul de la perte du générateur et du discriminateur
4) Calcul des performances du générateur et du discriminateur
5) Récupération des gradients des variables entraînables par rapport aux pertes
6) Effectue une étape de descente de gradient en mettant à jour les variables pour minimiser la perte

```python
def discriminator_generated_accuracy(self, generated_output):
  # Calcul de la performance du discriminateur à classifier
  # les faux échantillons à "faux"
  self.d_generated_accuracy.reset_states()
  return self.d_generated_accuracy(tf.zeros_like(generated_output), generated_output)

def train_step(self, real_samples):
  # Génére les points de l'espace latent
  random_latent_vectors = tf.random.normal(shape=(self.batch_size, self.latent_dim))

  with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
    # Génère les faux échantillons
    generated_samples = self.generator(random_latent_vectors)

    # Classifie les vrais échantillons
    real_output = self.discriminator(real_samples)
    # Classifie les faux échantillons
    generated_output = self.discriminator(generated_samples)

    # Calcul la perte du générateur
    gen_loss = self.generator_loss(generated_output)
    # Calcul la perte du discriminateur
    disc_loss = self.discriminator_loss(real_output, generated_output)

    # Calcul des performances du générateur
    gen_acc = self.generator_accuracy(generated_output)
    # Calcul des performances du discriminateur
    disc_acc = self.discriminator_accuracy(real_output, generated_output)
    # Calcul des performances du discriminateur avec les vrais échantillons
    disc_real_acc = self.discriminator_real_accuracy(real_output)
    # Calcul des performances du discriminateur avec les faux échantillons
    disc_gen_acc = self.discriminator_generated_accuracy(generated_output)

  # Récupère les gradients des variables entraînables par rapport à la perte
  gradients_of_generator = gen_tape.gradient(gen_loss,
    self.generator.trainable_variables)
  gradients_of_discriminator = disc_tape.gradient(disc_loss,
    self.discriminator.trainable_variables)

  # Effectue une étape de descente de gradient en mettant à jour la valeur
  # des variables pour minimiser la perte
  self.g_optimizer.apply_gradients(zip(gradients_of_generator,
    self.generator.trainable_variables))
  self.d_optimizer.apply_gradients(zip(gradients_of_discriminator,
    self.discriminator.trainable_variables))

  return gen_loss, disc_loss, gen_acc, disc_acc, disc_real_acc, disc_gen_acc
```

Enfin, je dois encore définir la méthode train() qui bouclera sur les époques pour appeler la méthode train_step() pour chaque lot.

```python
def train(self, dataset):
  file_writer = tf.summary.create_file_writer(self.logdir)
  file_writer.set_as_default()

  time_list = []

  # Boucle sur les époques
  for epoch in range(self.epochs):
    start_time = time.time()
    # Boucle sur les lots du jeu de données
    for real_samples in dataset:
      # Entraine le GAN à l'aide d'un lot de vrais échantillons
      gen_loss, disc_loss, gen_acc, disc_acc, disc_real_acc, disc_gen_acc = self.train_step(real_samples)
```

Cette méthode sera également chargée d'enregistrer les logs des différents indicateurs utiles au calibrage de l'évolution concurrentielle du générateur et du discriminateur. Elle dessinera également un diagramme de dispersion, toutes les 50 époques, afin de pouvoir visualiser les faux échantillons générés par le générateur en regard des vrais échantillons.

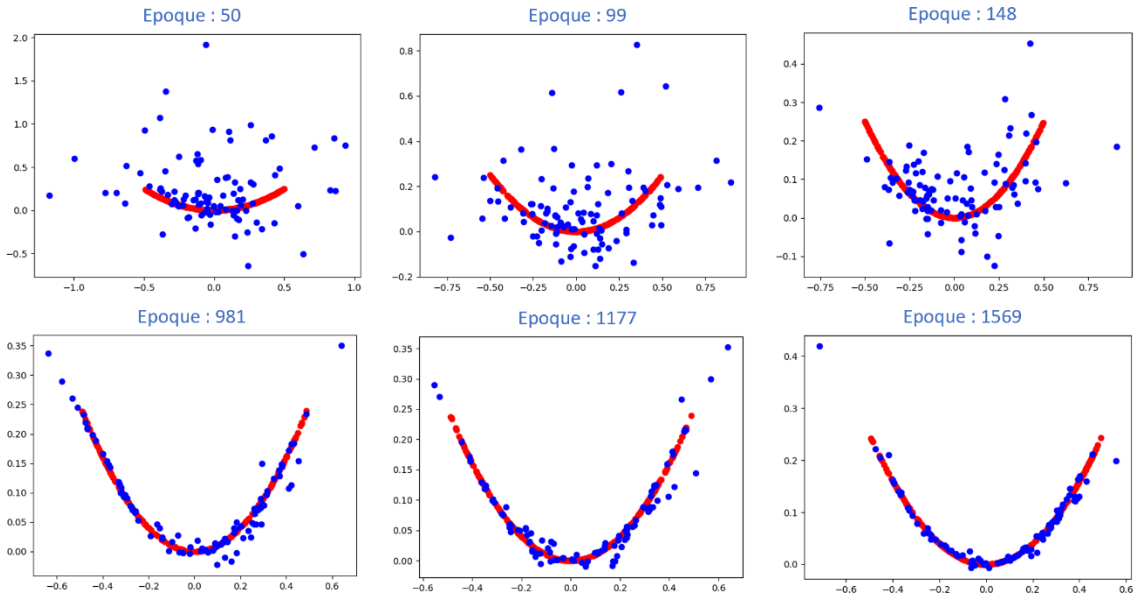
```python
      # Logue les différentes pertes
      tf.summary.scalar('1. Generator loss', gen_loss, step=epoch)
      tf.summary.scalar('2. Discriminator loss', disc_loss, step=epoch)
      # Logue les différentes performances
      tf.summary.scalar('3. Generator accuracy', gen_acc, step=epoch)
      tf.summary.scalar('4. Discriminator accuracy', disc_acc, step=epoch)
      tf.summary.scalar('5. Discriminator real accuracy', disc_real_acc, step=epoch)
      tf.summary.scalar('6. Discriminator fake accuracy', disc_gen_acc, step=epoch)
      # Toutes les 50 époques
      if (epoch) % 49 == 0:
        # Récupère 2 lots de vrais échantillons
        real_samples = np.concatenate([x for x in dataset.take(2)], axis=0)
        random_latent_vectors = tf.random.normal(shape=(100, self.latent_dim))
        # Récupère 100 faux échantillons
        generated_samples = np.array(self.generator(random_latent_vectors))
        # Dessine le diagramme de dispersion
        fig = pyplot.figure()
        ax2 = pyplot.axes()
        ax2.scatter(real_samples[:, 0], real_samples[:, 1], c="red")
        ax2.scatter(generated_samples[:, 0], generated_samples[:, 1], c="blue")
        tf.summary.image("Output generator", plot_to_image(fig), step=(epoch+1))
```

4.3.2 Entrainement et résultats

Maintenant que le GAN est développé et prêt à l'entrainement, il faut trouver le taux d'apprentissage adapté pour le discriminateur et le générateur pour pouvoir les garder en concurrence lors de l'entrainement et ainsi permettre la progression efficace des deux réseaux de neurones. La démarche de recherche du taux d'apprentissage optimal est détaillée dans l'annexe 1 « Recherche du taux d'apprentissage optimal ».

Voici les résultats obtenus au fur et à mesure des époques, les vrais échantillons sont représentés par les points rouges, les faux échantillons par les points bleus:

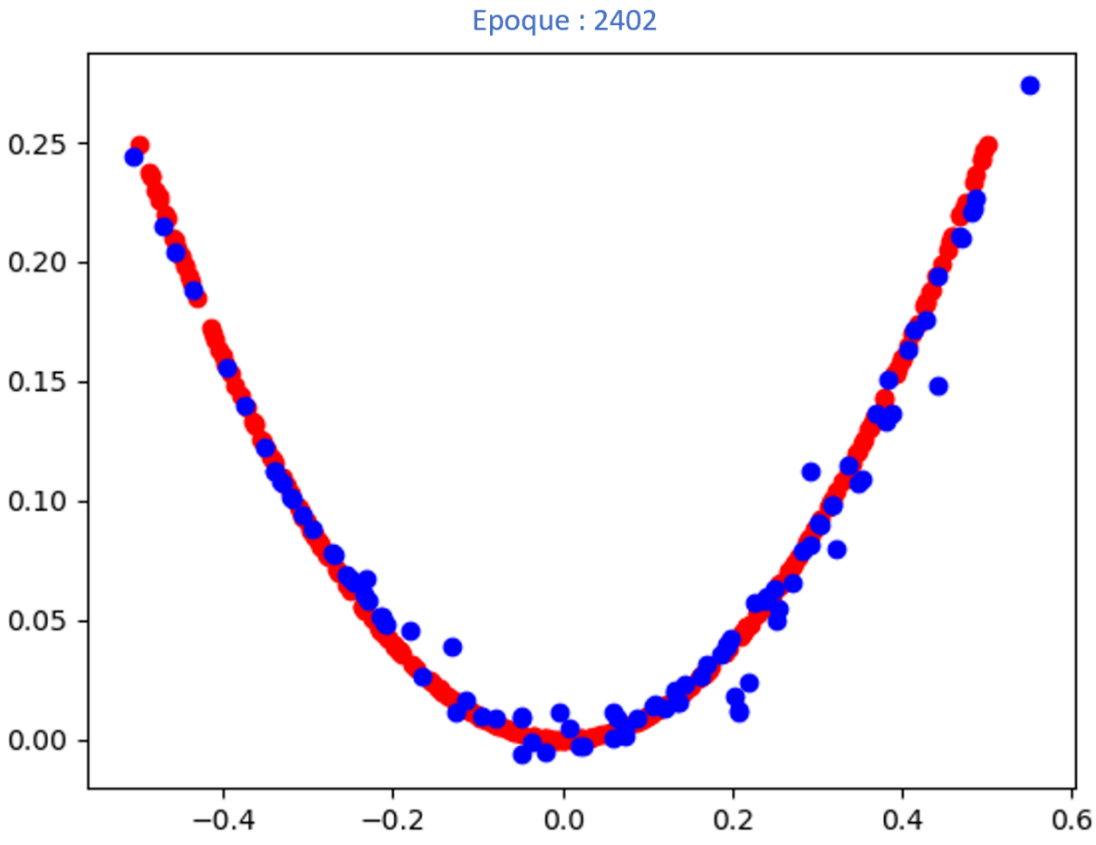
Figure 74 : GAN 1D évolution des résultats en fonction des époques



Fait à l'aide de l'outil : TensorBoard

On constate qu'après 2402 époques, les faux échantillons générés sont très proches des vrais :

Figure 75 : GAN 1D résultats après 2406 époques



Fait à l'aide de l'outil : TensorBoard

4.4 DCGAN avec MNIST

Toujours à l'aide du livre « Generative Adversarial Networks with Python » de Jason Brownlee (Brownlee, 2019b), je vais maintenant faire évoluer la structure pour mettre en place un DCGAN permettant de générer des images de nombres écrits à la main à l'aide de MNIST.

Figure 76 : MNIST

Source : Base de données MNIST, 2021

MNIST est un jeu de données contenant 70 000 images de 28 x 28 pixels en noir et blanc représentant des nombres écrits à la main entre 0 et 9.

Pour développer un GAN pour des images, Jason Brownlee préconise l'utilisation des réseaux de neurones convolutifs pour le discriminateur et le générateur (Brownlee, 2019b).

4.4.1 Développement

Pour charger le jeu de données MNIST, j'utiliserai la fonction mnist.load_data() dans Keras. Par défaut, cette fonction retourne 60 000 images d'entrainements et 10 000 images de test.

```
from keras.datasets import mnist

#Récupération des images MNIST d'entrainement et de test
(trainX, trainy), (testX, testy) = mnist.load_data()
```

Je vais commencer par faire évoluer le discriminateur.

Le discriminateur définit dans le premier cas pratique prenait en entrée deux valeurs. Il faut désormais le faire évoluer pour prendre en entrée une matrice à une dimension d'une taille de 28 x 28. Sa sortie restera identique étant donné que son objectif reste la classification binaire. Comme recommandé par Jason Brownlee (Brownlee, 2019b), j'intègre deux couches cachées convolutives.

```
def discriminator():
  # Entrée à 28 x 28 pixels et 1 canal
  inp = Input(shape=(28,28,1), name='input_sample')
  x = inp
  # Couche convolutive
  x = Conv2D(64, (3,3), strides=(2, 2), padding="same")(x)
  x = LeakyReLU(alpha=0.2)(x)
  x = Dropout(0.4)(x)
  # Couche convolutive
  x = Conv2D(64, (3,3), strides=(2, 2), padding="same")(x)
  x = LeakyReLU(alpha=0.2)(x)
  x = Dropout(0.4)(x)
  x = Flatten()(x)
  # Couche de sortie avec une fonction d'activation sigmoïde
  last = Dense(1, activation="sigmoid",name="output")(x)
  return Model(inputs=inp, outputs=last)
```

Je fais également évoluer le générateur en intégrant des couches convolutives :

```
def generator(latent_dim):
  # Entrée à "latent_dim" valeurs
  inp = Input(shape=(latent_dim,), name='input_sample')
  x = inp
  x = Dense(128 * 7 * 7)(x)
  x = LeakyReLU(alpha=0.2)(x)
  x = Reshape((7, 7, 128))(x)
  x = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(x)
  x = LeakyReLU(alpha=0.2)(x)
  x = Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(x)
  x = LeakyReLU(alpha=0.2)(x)
  # Ajout de la sortie avec la fonction d'activation sigmoid
  last = Conv2D(1, (7,7), activation='sigmoid', padding='same', name="output")(x)
  return Model(inputs=inp, outputs=last)
```

Il faut également redéfinir la fonction qui génère les vrais échantillons generate_real_samples() pour qu'elle utilise les données de MNIST :

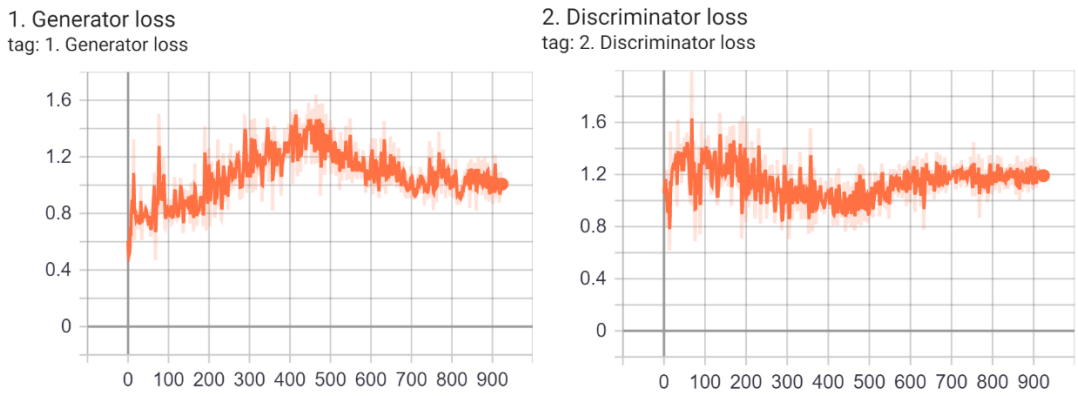
```python
# Génère n vrais échantillons
def generate_real_samples(n):
  # Charge le jeu d'entrainement depuis MNIST
  (trainX, _), (_, _) = mnist.load_data()
  # transforme les tableaux 2D en 3D en ajoutant un canal supplémentaire
  dataset = expand_dims(trainX, axis=-1)
  # Converti Int et float32
  dataset = dataset.astype("float32")
  # Change l'échelle [0,255] en [0,1]
  dataset = dataset / 255.0
  # Choisi aléatoirement des index du dataset
  ix = randint(0, dataset.shape[0], n)
  # Récupère les images sélectionnées
  x = dataset[ix]
  # Ajoute la classe y = 1 pour indiquer qu'il s'agit de vrais échantillons
  y = ones((n, 1))
  return x, y
```

Maintenant que le GAN a été adapté, nous pouvons procéder à son entrainement.

4.4.2 Entrainement et résultats

Voici l'évolution de la perte du générateur et du discriminateur après 924 époques :

Figure 77 : DCGAN évolution perte générateur et discriminateur

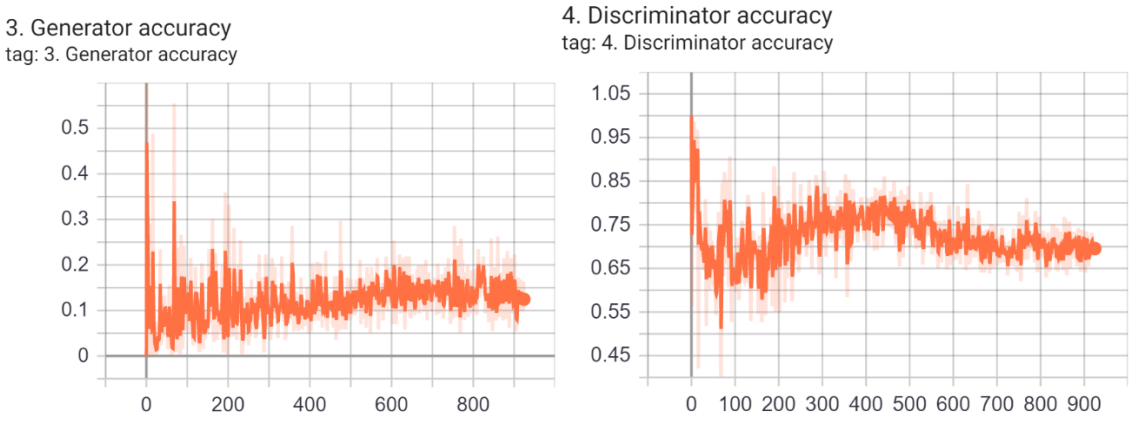


Fait à l'aide de l'outil : TensorBoard

On peut constater que la perte du discriminateur diminue entre 50 et 500 époques, ce qui reflète une progression du discriminateur. Parallèlement la perte du générateur augmente, il doit en effet augmenter son effort face à l'évolution de la performance du discriminateur pour rester dans la course.

Regardons maintenant les courbes de performances des deux réseaux :

Figure 78 : DCGAN évolution performance générateur et discriminateur



Fait à l'aide de l'outil : TensorBoard

On constate que le discriminateur garde une longueur d'avance sur le générateur (entre 65 et 75% de réussite), alors que le générateur affiche une performance relativement modérée (entre 5% et 20%). Comme relevé dans l'annexe 1 « Recherche du taux d'apprentissage optimal », il est important que le discriminateur garde une certaine avance pour pouvoir guider le générateur dans la bonne direction, sans toutefois que la performance du générateur tombe à 0%. L'équilibre illustré ci-dessus est donc favorable pour un entrainement optimal du GAN.

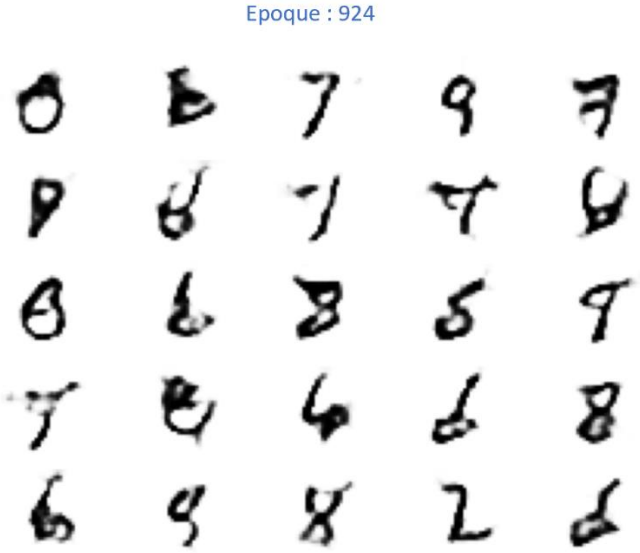
Voyons maintenant l'évolution des résultats au fur et à mesure des époques :

Figure 79 : DCGAN évolution des résultats en fonction des époques

Fait à l'aide de l'outil : TensorBoard

Après 924 époques, on peut distinguer différents nombres se dessiner parmi les échantillons générés :

Figure 80 : DCGAN résultats après 924 époques



Fait à l'aide de l'outil : TensorBoard

4.5 CycleGAN pour la conversion de voix

Après avoir réalisé ces quelques cas pratiques, je vais maintenant faire évoluer cette structure pour réaliser une conversion de voix. Pour les concepts, je m'appuierai sur la publication « CycleGAN-VC2: Improved CycleGAN-based Non-parallel Voice Conversion » (Kaneko, Kameoka, Tanaka et al, 2019) et pour le code, je m'inspirerai des dépôts GitHub « CycleGAN-VC2-PyTorch » (Kun Ma, 2020) et « Voice-Conversion-CycleGAN2 » (Aplharol, 2019).

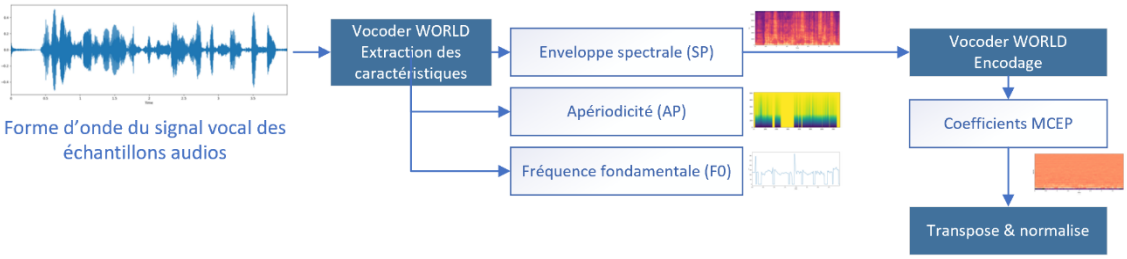
A noter que le fonctionnement de CycleGAN-VC2 (Kaneko, Kameoka, Tanaka et al, 2019) a été abordé dans le chapitre « La conversion de voix avec le deep learning ».

L'objectif sera de faire atteindre une maturité suffisante aux générateurs afin qu'ils puissent transformer des échantillons de voix A en échantillons de voix B réalistes et inversement.

4.5.1 Développement

Les réseaux de neurones seront entrainés à l'aide des coefficients MCEP transposés et normalisés extraits des échantillons audio. Pour commencer, je vais donc implémenter les fonctions nécessaires pour l'extraction des différentes caractéristiques du signal audio.

Figure 81 : CycleGAN-voix extraction caractéristiques signal vocal



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

J'implémente une fonction permettant d'ouvrir les fichiers audio format wav :

```
# Charge les fichiers wav dans une liste
# wav_dir : Répertoire
# sr : Taux d'échantillonnage
def load_wavs(wav_dir, sr):
    wavs = list()
    filenames = list()
    for file in os.listdir(wav_dir):
        file_path = os.path.join(wav_dir, file)
        wav, _ = librosa.load(file_path, sr=sr, mono=True)
        wavs.append(wav)
        filenames.append(file)
    return wavs, filenames
```

Le taux d'échantillonnage sera de 22 kHz pour se rapprocher de la valeur préconisée par Kaneko, Kameoka, Tanaka et al.

Puis la fonction permettant d'extraire la F0, l'enveloppe spectrale et l'apériodicité

```
# Extrait la f0, l'enveloppe spectrale et l'apériodicité
# d'un fichier audio à l'aide de WORLD vocodeur
# wav : Fichier audio
# fs : Taux d'échantillonnage
# frame_period : Durée d'une trame
def world_decompose(wav, fs, frame_period=5.0):
    wav = wav.astype(np.float64)
    # Extraction de la f0 et de la position temporelle de chaque trame
    f0, timeaxis = pyworld.harvest(
        wav, fs, frame_period=frame_period, f0_floor=71.0, f0_ceil=800.0)

    # Extraction de l'enveloppe spectrale
    sp = pyworld.cheaptrick(wav, f0, timeaxis, fs)

    # Extraction de l'aperiodicité
    ap = pyworld.d4c(wav, f0, timeaxis, fs)

    return f0, timeaxis, sp, ap
```

J'implémente également une fonction qui, sur la base de l'enveloppe spectrale, du taux d'échantillonnage et d'un nombre de dimensions souhaités, permet d'obtenir les coefficients MCEP :

```
# Génère la représentation Mel-cepstral coefficients (MCEP)
# sp : Enveloppe spectrale
# fs : Taux d'échantillonnage
# dim : Nombre de dimensions MCEP souhaitées
def world_encode_spectral_envelop(sp, fs, dim=34):
    # Get Mel-Cepstral coefficients (MCEP)
    coded_sp = pyworld.code_spectral_envelope(sp, fs, dim)
    return coded_sp
```

A noter que la durée de 5ms par trame ainsi que la dimension de 34 utilisée pour la représentation Mel-cepstral coefficients sont les valeurs utilisées par Kaneko, Kameoka, Tanaka et al.

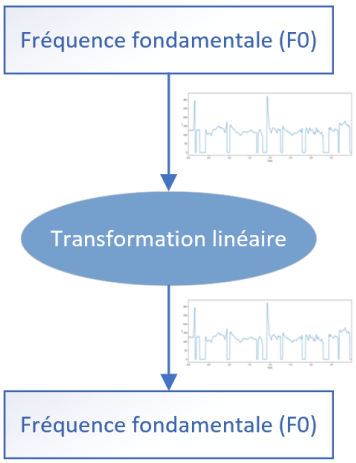
Voici les fonctions qui permettront de transposer et normaliser les coefficients MCEP :

```
# Transpose dans une liste
def transpose_in_list(lst):
    transposed_lst = list()
    for array in lst:
        transposed_lst.append(array.T)
    return transposed_lst
```

```
# Normalise la représentation Mel-cepstral coefficients (MCEP)
def coded_sps_normalization_fit_transform(coded_sps):
    coded_sps_concatenated = np.concatenate(coded_sps, axis=1)
    coded_sps_mean = np.mean(coded_sps_concatenated, axis=1, keepdims=True)
    coded_sps_std = np.std(coded_sps_concatenated, axis=1, keepdims=True)
    coded_sps_normalized = list()
    for coded_sp in coded_sps:
        coded_sps_normalized.append(coded_sp_normalization_fit_transform(coded_sp, coded_sps_mean, coded_sps_std))
    return coded_sps_normalized, coded_sps_mean, coded_sps_std
```

Nous allons devoir effectuer une transformation linéaire de la fréquence fondamentale (F0) la voix A vers la voix B :

Figure 82 : CycleGAN-voix transformation linéaire F0



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Pour réaliser cette transformation, nous avons besoin d'une fonction permettant d'obtenir la moyenne et l'écart type de la fréquence fondamentale :

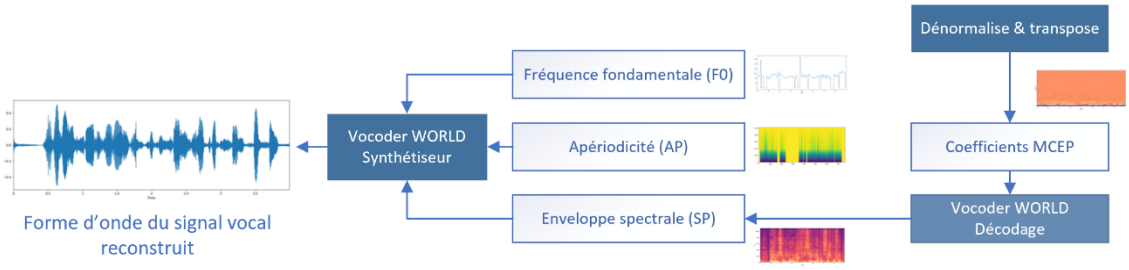
```
# Calcule la moyenne et l'écart type de la
# fréquence fondamentale
def logf0_statistics(f0s):
    log_f0s_concatenated = np.ma.log(np.concatenate(f0s))
    log_f0s_mean = log_f0s_concatenated.mean()
    log_f0s_std = log_f0s_concatenated.std()
    return log_f0s_mean, log_f0s_std
```

Et de mettre en place la fonction qui réalisera cette transformation :

```
# Normalisation gaussienne logarithmique pour la conversion du pitch
def pitch_conversion(f0, mean_log_src, std_log_src, mean_log_target, std_log_target):
    f0_converted = np.exp((np.log(f0) - mean_log_src) /
                          std_log_src * std_log_target + mean_log_target)
    return f0_converted
```

Nous devons maintenant créer les fonctions qui permettront de faire l'opération inverse afin de reconstruire les fichiers audio depuis les différentes caractéristiques.

Figure 83 : CycleGAN-voix reconstruction du signal vocal



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Pour dénormaliser les coefficients MCEP :

```
# Dénormalise les représentations Mel-cepstral coefficients (MCEP)
def coded_sps_denormalization_fit_transform(coded_sps_normalized, coded_sps_mean, coded_sps_std):
    coded_sps = list()
    for coded_sp_normalized in coded_sps_normalized:
      coded_sps.append(coded_sp_denormalization_fit_transform(coded_sp_normalized, coded_sps_mean, coded_sps_std))
    return coded_sps
```

Pour reconstruire l'enveloppe spectrale depuis les coefficients MCEP :

```
# Décode l'enveloppe spectrale
# Mel-cepstral coefficients (MCEP) -> envelope spectrale
def world_decode_spectral_envelop(coded_sp, fs):
  fftlen = pyworld.get_cheaptrick_fft_size(fs)
  decoded_sp = pyworld.decode_spectral_envelope(coded_sp, fs, fftlen)
  return decoded_sp
```

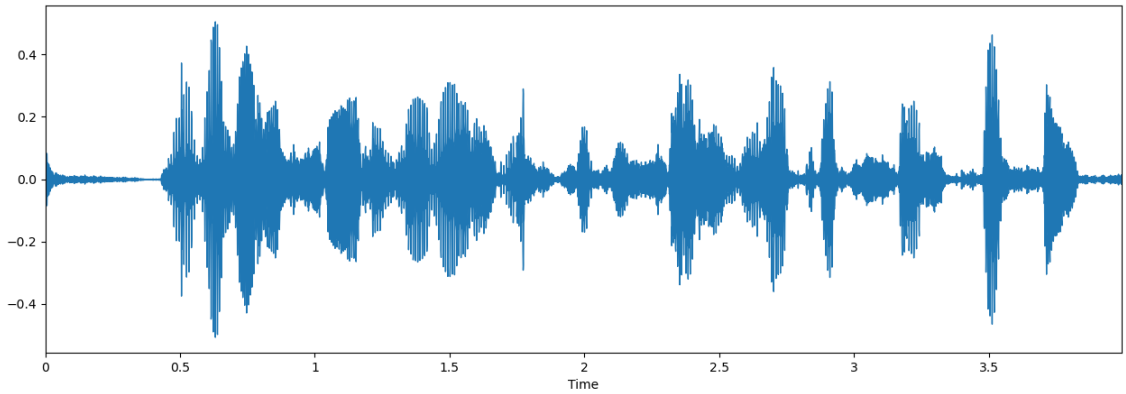
Pour regénérer les fichiers audios :

```
# Génère des fichiers audios depuis leurs
# caractéristiques
# f0 : Fréquence fondamentale
# decoded_sp : Enveloppe spectrale décodée
# ap : apériodicité
# fs : Taux d'échantillonnage
# frame_period : Durée d'une trame
def world_speechs_synthesis(f0s, decoded_sps, aps, fs, frame_period):
  wavs = list()
  for f0, decoded_sp, ap in zip(f0s, decoded_sps, aps):
    wavs.append(world_speech_synthesis(f0, decoded_sp, ap, fs, frame_period))
  return wavs
```

Pour s'assurer du bon fonctionnement de l'ensemble de ces fonctions, testons maintenant toute la chaîne de traitement (aller et retour) avec un échantillon de voix.

Voici la forme d'onde avant encodage -> transposition -> normalisation -> dénormalisation -> détransposition -> décodage :

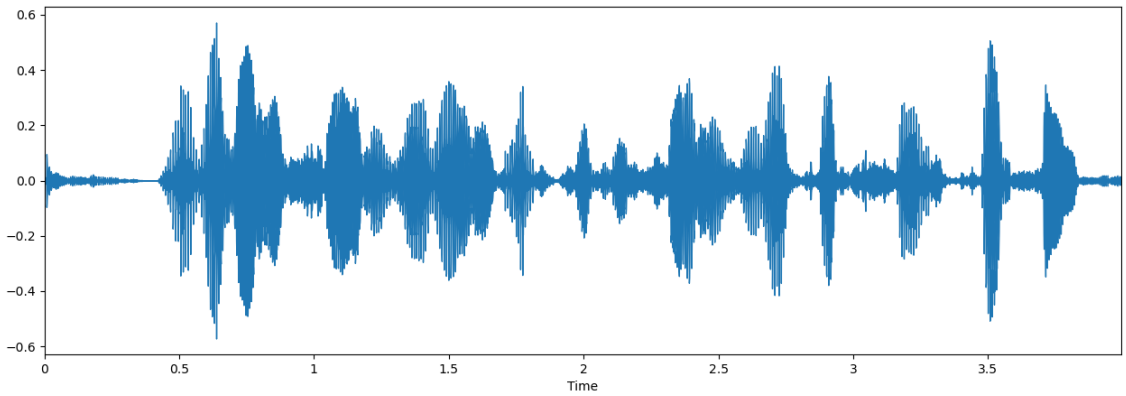
Figure 84 : CycleGAN-voix forme d'onde avant traitement



Fait à l'aide de l'outil : Librosa

Voici sa forme d'onde après encodage -> transposition -> normalisation -> dénormalisation -> détransposition -> décodage :

Figure 85 : CycleGAN-voix forme d'onde après traitement



Fait à l'aide de l'outil : Librosa

On peut observer que ses deux formes d'ondes restent relativement similaires. A l'écoute du fichier audio traité, on peut constater une légère « robotisation » de la voix, mais le locuteur reste tout à fait identifiable.

Nous implémentons maintenant une fonction de pré-traitement qui sera chargée d'effectuer l'extraction et le calcul de toutes les caractéristiques vues précédemment :

```
def preprocessing(dir_wavs_A, dir_wavs_B, dir_cache, sr, frame_period, num_mcep,
                  path_logf0s_norm, path_mcep_norm, path_coded_sps_A_norm, path_coded_sps_B_norm):
  # Vérifie que le cache soit vide
  if not os.path.exists(dir_cache):
    os.mkdir(dir_cache)
    # Chargement des fichiers wav des voix A et B
    wavs_A, _ = load_wavs(dir_wavs_A, sr)
    wavs_B, _ = load_wavs(dir_wavs_B, sr)

    # Extraction des caractéristiques des voix A et B
    f0s_A, timeaxes_A, sps_A, aps_A, coded_sps_A = world_encode_data(wavs_A, sr, frame_period, num_mcep)
    f0s_B, timeaxes_B, sps_B, aps_B, coded_sps_B = world_encode_data(wavs_B, sr, frame_period, num_mcep)

    # Calcul de la moyenne et de l'écart type du f0 des voix A et B
    log_f0s_mean_A, log_f0s_std_A = logf0_statistics(f0s_A)
    log_f0s_mean_B, log_f0s_std_B = logf0_statistics(f0s_B)

    # Transposition MCEP des voix A et B
    coded_sps_A_transposed = transpose_in_list(coded_sps_A)
    coded_sps_B_transposed = transpose_in_list(coded_sps_B)

    # Normalisation MCEP des voix A et B
    coded_sps_A_norm, coded_sps_A_mean, coded_sps_A_std = coded_sps_normalization_fit_transform(coded_sps_A_transposed)
    coded_sps_B_norm, coded_sps_B_mean, coded_sps_B_std = coded_sps_normalization_fit_transform(coded_sps_B_transposed)
```

Cette fonction sera également chargée de sauvegarder, dans un répertoire de cache, les différentes données extraites comme les coefficients MCEP ou les données relatives à la fréquence fondamentales utiles lors de la reconstruction du signal vocal :

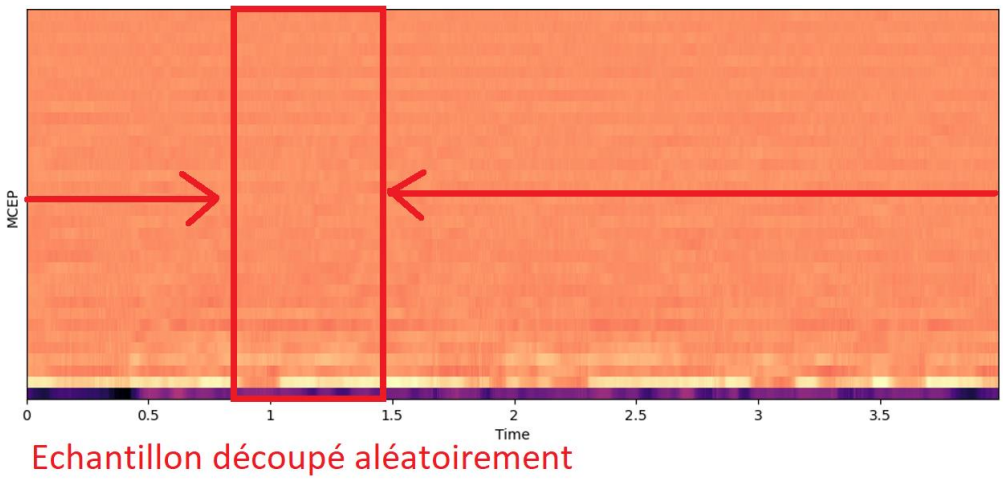
```
    np.savez(path_logf0s_norm,
             mean_A=log_f0s_mean_A,
             std_A=log_f0s_std_A,
             mean_B=log_f0s_mean_B,
             std_B=log_f0s_std_B)

    np.savez(path_mcep_norm,
             mean_A=coded_sps_A_mean,
             std_A=coded_sps_A_std,
             mean_B=coded_sps_B_mean,
             std_B=coded_sps_B_std)

    save_pickle(variable=coded_sps_A_norm,
                fileName=path_coded_sps_A_norm)
    save_pickle(variable=coded_sps_B_norm,
                fileName=path_coded_sps_B_norm)
# Si le cache n'est pas vide
else:
    coded_sps_A_norm = load_pickle_file(path_coded_sps_A_norm)
    coded_sps_B_norm = load_pickle_file(path_coded_sps_B_norm)
```

La taille des coefficients MCEP utilisée pour l'entrainement des réseaux de neurones sera fixe. Sachant que cette taille varie en fonction de la durée de l'échantillon, nous mettons en place une fonction qui permettra d'extraire aléatoirement une suite de trames pour former un échantillon découpé aléatoirement. Cette technique permettra d'enrichir le jeu de vrais échantillons audio utilisés pour l'entrainement en permettant l'utilisation d'échantillons différents découpés aléatoirement à chaque époque.

Figure 86 : CycleGAN-voix découpage aléatoire MCEP



Fait à l'aide de l'outil : Librosa

```python
# Découpe les MCEP normalisés à un nombre de frames défini
def cut_coded_sps_norm(coded_sps_A_norm, coded_sps_B_norm, n_frames=128):
    coded_sps_A_norm_split = list()
    coded_sps_B_norm_split = list()

    for coded_sp_A_norm, coded_sp_B_norm in zip(coded_sps_A_norm, coded_sps_B_norm):
        frames_A_total = coded_sp_A_norm.shape[1]
        assert frames_A_total >= n_frames
        start_A = np.random.randint(frames_A_total - n_frames + 1)
        end_A = start_A + n_frames
        coded_sps_A_norm_split.append(coded_sp_A_norm[:, start_A:end_A])

        frames_B_total = coded_sp_B_norm.shape[1]
        assert frames_B_total >= n_frames
        start_B = np.random.randint(frames_B_total - n_frames + 1)
        end_B = start_B + n_frames
        coded_sps_B_norm_split.append(coded_sp_B_norm[:, start_B:end_B])

    coded_sps_A_norm_split = np.array(coded_sps_A_norm_split)
    coded_sps_B_norm_split = np.array(coded_sps_B_norm_split)

    return coded_sps_A_norm_split, coded_sps_B_norm_split
```

Pour utiliser les générateurs avec des échantillons complets, nous devons implémenter une fonction qui permettra de formater les fichiers audio au format d'entrée des générateurs. Le formatage se résume à l'ajout de trames vides en début et fin de fichier afin de garantir que le nombre de valeurs (après traitement) soit divisible par la taille d'entrée des générateurs (afin qu'ils puissent traiter les données par morceau en divisant les échantillons complets en morceaux de la taille de leur entrée) :

```python
#Formate le fichier audio pour être compatible avec l'entrée du générateur
def wav_padding(wav, sr, frame_period, multiple = 4):
    assert wav.ndim == 1
    num_frames = len(wav)
    num_frames_padded = int((np.ceil((np.floor(num_frames / (sr * frame_period / 1000)) + 1) / multiple + 1) * multiple - 1)
                            * (sr * frame_period / 1000))
    num_frames_diff = num_frames_padded - num_frames
    num_pad_left = num_frames_diff // 2
    num_pad_right = num_frames_diff - num_pad_left
    wav_padded = np.pad(wav, (num_pad_left, num_pad_right), 'constant', constant_values = 0)

    return wav_padded
```

Maintenant que nous avons toutes les fonctions nécessaires pour alimenter les réseaux de neurones, nous allons adapter le discriminateur pour intégrer le PatchGAN utilisé dans CycleGAN-VC2 (Kaneko, Kameoka, Tanaka et al, 2019).

```python
def discriminator(num_mcep, n_frames):
  # Entrée
  inp = Input(shape=(num_mcep, n_frames, 1), name='input_sample')
  inputs = inp

  h1 = conv2d_layer(inputs = inputs, filters = 128, kernel_size = [3, 3],
                    strides = [1, 1], activation = None, name = 'h1_conv')
  h1_gates = conv2d_layer(inputs = inputs, filters = 128, kernel_size = [3, 3],
                          strides = [1, 1], activation = None, name = 'h1_conv_gates')
  h1_glu = gated_linear_layer(inputs = h1, gates = h1_gates, name = 'h1_glu')

  # Downsample
  d1 = downsample2d_block(inputs = h1_glu, filters = 256, kernel_size = [3, 3],
                          strides = [2, 2], name_prefix = 'downsample2d_block1_')
  d2 = downsample2d_block(inputs = d1, filters = 512, kernel_size = [3, 3],
                          strides = [2, 2], name_prefix = 'downsample2d_block2_')
  d3 = downsample2d_block(inputs = d2, filters = 1024, kernel_size = [3, 3],
                          strides = [2, 2], name_prefix = 'downsample2d_block3_')
  d4 = downsample2d_block(inputs=d3, filters=1024, kernel_size=[1, 5],
                          strides=[1, 1], name_prefix='downsample2d_block4_')

  # Output
  o1 = conv2d_layer(inputs=d4, filters=1, kernel_size=[1, 3],
                    strides=[1, 1], activation=tf.nn.sigmoid, name='out_1d_conv')

  return Model(inputs=inp, outputs=o1)
```

Redéfinissons maintenant le générateur sur la base de la structure de CycleGAN-VC2 (Kaneko, Kameoka, Tanaka et al, 2019) :

```python
def generator(num_mcep):
    # Entrée
    inp = Input(shape=(num_mcep, None, 1), name='input_sample')
    inputs = inp

    res_filter = 512
    batch_size = 1

    h1 = conv2d_layer(inputs = inputs, filters = 128, kernel_size = [5, 15],
                      strides = [1, 1], activation = None, name = 'h1_conv')
    h1_gates = conv2d_layer(inputs = inputs, filters = 128, kernel_size = [5, 15],
                            strides = 1, activation = None, name = 'h1_conv_gates')
    h1_glu = gated_linear_layer(inputs = h1, gates = h1_gates, name = 'h1_glu')

    # Downsample
    d1 = downsample2d_block(inputs = h1_glu, filters = 256, kernel_size = 5,
                            strides = 2, name_prefix = 'downsample1d_block1_')
    d2 = downsample2d_block(inputs = d1, filters = 256, kernel_size = 5,
                            strides = 2, name_prefix = 'downsample1d_block2_')

    # Reshape
    d3 = tf.squeeze(tf.reshape(d2, shape=(batch_size, 1, -1, 2304)), axis=1)

    # 1x1 Conv
    d3 = conv1d_layer(inputs=d3, filters=256, kernel_size = 1,
                      strides = 1, activation = None, name = '1x1_down_conv1d')
    d3 = InstanceNormalization(epsilon=1e-6, name='d3_norm')(d3)

    # Residual blocks
    r1 = residual1d_block(inputs = d3, filters = res_filter, kernel_size = 3,
                          strides = 1, name_prefix = 'residual1d_block1_')
    r2 = residual1d_block(inputs = r1, filters = res_filter, kernel_size = 3,
                          strides = 1, name_prefix = 'residual1d_block2_')
    r3 = residual1d_block(inputs = r2, filters = res_filter, kernel_size = 3,
                          strides = 1, name_prefix = 'residual1d_block3_')
    r4 = residual1d_block(inputs = r3, filters = res_filter, kernel_size = 3,
                          strides = 1, name_prefix = 'residual1d_block4_')
    r5 = residual1d_block(inputs = r4, filters = res_filter, kernel_size = 3,
                          strides = 1, name_prefix = 'residual1d_block5_')
    r6 = residual1d_block(inputs = r5, filters = res_filter, kernel_size = 3,
                          strides = 1, name_prefix = 'residual1d_block6_')

    # 1x1 Conv
    r6 = conv1d_layer(r6, filters = 2304, kernel_size = 1,
                      strides = 1, activation = None, name = '1x1_up_conv1d')
    r6 = InstanceNormalization(epsilon=1e-6, name='r6_norm')(r6)

    # Reshape
    r6 = tf.reshape(tf.expand_dims(r6, axis=1), shape=(batch_size, 9, -1, 256))

    # Upsample
    u1 = upsample2d_block(inputs = r6, filters = 1024, kernel_size = 5,
                          strides = 1, name_prefix = 'upsample1d_block1_')
    u2 = upsample2d_block(inputs = u1, filters = 512, kernel_size = 5,
                          strides = 1, name_prefix = 'upsample1d_block2_')

    # Output
    o1 = conv2d_layer(inputs = u2, filters = 1, kernel_size = [5, 15],
                      strides = [1, 1], activation = None, name = 'o1_conv')

    return Model(inputs=inp, outputs=o1)
```

Implémentons les fonctions de pertes de CycleGAN-VC2 :

```
def generator_loss(self, generated_output):
  # Calcul de la perte du générateur à l'aide des faux échantillons
  # ayant été classifiés comme "vrai" par le discriminateur
  return tf.reduce_mean(tf.square(tf.ones_like(generated_output) - generated_output))

def discriminator_loss(self, real_output, generated_output):
  # Calcul de la perte du discriminateur à l'aide des vrais échantillons
  # ayant été classifiés comme "vrai" et des faux échantillons ayant
  # été classifiés comme "faux"
  real_loss = tf.reduce_mean(tf.square(tf.ones_like(real_output) - real_output))
  generated_loss = tf.reduce_mean(tf.square(tf.zeros_like(generated_output) - generated_output))
  total_loss = (real_loss + generated_loss) / 2.0
  return total_loss

def calc_cycle_loss(self, real_image, cycled_image):
  loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))
  return loss1

def identity_loss(self, real_image, same_image):
  loss = tf.reduce_mean(tf.abs(real_image - same_image))
  return loss
```

Ainsi que d'autres indicateurs de performance qui seront utiles pour le suivi de l'entrainement :

```
def generator_accuracy(self, generated_output):
  # Calcul de la performance du générateur à l'aide des faux échantillons
  # ayant été classifiés comme "vrai" par le discriminateur
  self.g_accuracy.reset_states()
  return self.g_accuracy(tf.ones_like(generated_output), generated_output)

def discriminator_accuracy(self, real_output, generated_output):
  # Calcul de la performance du discriminateur à l'aide des échantillons réels
  # ayant été classifiés comme "vrai" et des faux échantillons générés par
  # le générateur ayant été classifiés comme "faux"
  self.d_accuracy.reset_states()
  return self.d_accuracy(tf.concat([tf.ones_like(real_output), tf.zeros_like(generated_output)], 0),
    tf.concat([real_output, generated_output], 0))

def discriminator_real_accuracy(self, real_output):
  # Calcul de la performance du discriminateur à classifier
  # les vrais échantillons à "vrai"
  self.d_real_accuracy.reset_states()
  return self.d_real_accuracy(tf.ones_like(real_output), real_output)

def discriminator_generated_accuracy(self, generated_output):
  # Calcul de la performance du discriminateur à classifier
  # les faux échantillons à "faux"
  self.d_generated_accuracy.reset_states()
  return self.d_generated_accuracy(tf.zeros_like(generated_output), generated_output)
```

Nous devons maintenant implémenter la fonction train_step() qui sera chargée d'effectuer les opérations d'entrainement et d'évolution des réseaux de neurones pour chaque lot.

Cette fonction est composée de deux parties distinctes, la première partie sera chargée de faire évoluer les générateurs, la seconde partie les discriminateurs.

Commençons par détailler la première partie. Tout d'abord nous devons générer des faux échantillons par les générateurs et effectuer des prédictions de classification à l'aide des discriminateurs :

```
def train_step(self, dataset_A_batch, dataset_B_batch):

  with tf.GradientTape() as gen_tape:
    # Génère les faux échantillons de la voix B depuis
    # les vrais échantillons de la voix A
    fake_samples_B = self.generator_AB(dataset_A_batch, training=True)
    cycled_samples_A = self.generator_BA(tf.squeeze(fake_samples_B, axis=-1), training=True)

    fake_samples_A = self.generator_BA(dataset_B_batch, training=True)
    cycled_samples_B = self.generator_AB(tf.squeeze(fake_samples_A, axis=-1), training=True)

    same_samples_A = tf.dtypes.cast(tf.squeeze(self.generator_BA(dataset_A_batch, training=True), axis=-1), tf.float64)
    same_samples_B = tf.dtypes.cast(tf.squeeze(self.generator_AB(dataset_B_batch, training=True), axis=-1), tf.float64)

    # Classifie les faux échantillons de la voix B
    d_fake_output_A = self.discriminator_A(fake_samples_A, training=True)
    # Classifie les faux échantillons de la voix B
    d_fake_output_B = self.discriminator_B(fake_samples_B, training=True)

    # for the second step adverserial loss
    d_fake_output_cycle_A = self.discriminator_A(cycled_samples_A)
    d_fake_output_cycle_B = self.discriminator_B(cycled_samples_B)
```

Puis nous calculons les résultats des fonctions de pertes des générateurs et les indicateurs de performance à l'aide des échantillons générés et des prédictions de classification :

```
    # for the second step adverserial loss
    d_fake_output_cycle_A = self.discriminator_A(cycled_samples_A)
    d_fake_output_cycle_B = self.discriminator_B(cycled_samples_B)

    cycle_A_loss = self.calc_cycle_loss(dataset_A_batch, tf.dtypes.cast(tf.squeeze(cycled_samples_A, axis=-1), tf.float64))
    cycle_B_loss = self.calc_cycle_loss(dataset_B_batch, tf.dtypes.cast(tf.squeeze(cycled_samples_B, axis=-1), tf.float64))
    total_cycle_loss = cycle_A_loss + cycle_B_loss

    identity_A_loss = self.identity_loss(dataset_A_batch, same_samples_A)
    identity_B_loss = self.identity_loss(dataset_B_batch, same_samples_B)
    total_identity_loss = identity_A_loss + identity_B_loss

    # Calcul la perte du générateur
    gen_AB_loss = self.generator_loss(d_fake_output_B)
    gen_BA_loss = self.generator_loss(d_fake_output_A)

    # Calcul perte cycle
    gen_AB_loss_2nd = self.generator_loss(d_fake_output_cycle_B)
    gen_BA_loss_2nd = self.generator_loss(d_fake_output_cycle_A)

    total_gen_loss = tf.dtypes.cast(gen_AB_loss, tf.float64) + tf.dtypes.cast(gen_BA_loss, tf.float64) + \
                     tf.dtypes.cast(gen_AB_loss_2nd, tf.float64) + tf.dtypes.cast(gen_BA_loss_2nd, tf.float64) + \
                     self.lambda_cycle * total_cycle_loss  + total_identity_loss * self.lambda_identity

    # Calcul des performances du générateur
    gen_AB_acc = self.generator_accuracy(d_fake_output_B)
    # Calcul des performances du générateur
    gen_BA_acc = self.generator_accuracy(d_fake_output_A)
```

Nous pouvons maintenant récupérer les variables entrainables des générateurs, calculer les gradients à l'aide des résultats des fonctions de pertes des générateurs et appliquer ces gradients aux variables entrainables pour faire évoluer les réseaux de neurones des générateurs :

```
# Récupère les gradients des variables entraînables par rapport à la perte
generator_vars = self.generator_AB.trainable_variables + self.generator_BA.trainable_variables
gradients_of_generator = gen_tape.gradient(total_gen_loss, generator_vars)
self.g_optimizer.apply_gradients(zip(gradients_of_generator, generator_vars))
```

Passons maintenant à la seconde partie. Nous effectuons des prédictions de classification à l'aide des discriminateurs et nous regénérons des faux échantillons par les générateurs :

```
with tf.GradientTape() as dis_tape:
  # Classifie les vrais échantillons de la voix A
  real_output_A = self.discriminator_A(dataset_A_batch, training=True)
  # Classifie les vrais échantillons de la voix B
  real_output_B = self.discriminator_B(dataset_B_batch, training=True)

  generated_samples_A = self.generator_BA(dataset_B_batch, training=True)
  d_fake_output_A = self.discriminator_A(generated_samples_A, training=True)

  cycled_samples_B = self.generator_AB(tf.squeeze(generated_samples_A, axis=-1), training=True)
  d_cycled_B = self.discriminator_B(cycled_samples_B, training=True)

  generated_samples_B = self.generator_AB(dataset_A_batch, training=True)
  d_fake_output_B = self.discriminator_B(generated_samples_B, training=True)

  cycled_samples_A = self.generator_BA(tf.squeeze(generated_samples_B, axis=-1), training=True)
  d_cycled_A = self.discriminator_A(cycled_samples_A, training=True)
```

Nous calculons ensuite les résultats des fonctions de perte des discriminateurs :

```
  # Calcul la perte du discriminateur
  disc_A_loss = self.discriminator_loss(real_output_A, d_fake_output_A)
  disc_B_loss = self.discriminator_loss(real_output_B, d_fake_output_B)

  # Calcul la perte du cycle
  disc_A_loss_2nd = self.discriminator_loss(real_output_A, d_cycled_A)
  disc_B_loss_2nd = self.discriminator_loss(real_output_B, d_cycled_B)

  total_disc_loss = (disc_A_loss + disc_B_loss) / 2.0 + (disc_A_loss_2nd + disc_B_loss_2nd) / 2.0

  # Calcul des performances du discriminateur
  disc_B_acc = self.discriminator_accuracy(real_output_B, d_fake_output_B)
  # Calcul des performances du discriminateur
  disc_A_acc = self.discriminator_accuracy(real_output_A, d_fake_output_A)
  # Calcul des performances du discriminateur avec les vrais échantillons de la voix B
  disc_B_real_acc = self.discriminator_real_accuracy(real_output_B)
  # Calcul des performances du discriminateur avec les vrais échantillons de la voix B
  disc_A_real_acc = self.discriminator_real_accuracy(real_output_A)
  # Calcul des performances du discriminateur avec les faux échantillons de la voix B
  disc_B_gen_acc = self.discriminator_generated_accuracy(d_fake_output_B)
  # Calcul des performances du discriminateur avec les faux échantillons de la voix B
  disc_A_gen_acc = self.discriminator_generated_accuracy(d_fake_output_A)
```

Et nous calculons les gradients à l'aide des résultats des fonctions de pertes des discriminateurs, que nous appliquons aux variables entrainables des discriminateurs pour les faire évoluer :

```
discriminator_vars = self.discriminator_B.trainable_variables + self.discriminator_A.trainable_variables
gradients_of_discriminator = dis_tape.gradient(total_disc_loss, discriminator_vars)
self.d_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator_vars))
```

Enfin, nous retournons les valeurs des fonctions de pertes et les indicateurs de performance pour pouvoir les utiliser pour le suivi de l'entrainement :

```
return total_gen_loss, total_disc_loss, gen_AB_loss, gen_BA_loss, cycle_A_loss, cycle_B_loss,
        identity_B_loss, identity_A_loss, disc_B_loss, disc_A_loss, gen_AB_acc, gen_BA_acc, disc_B_acc,
        disc_B_real_acc, disc_B_gen_acc, disc_A_acc, disc_A_real_acc, disc_A_gen_acc
```

Développons maintenant la fonction train() décrivant les opérations à effectuer pour chaque époque.

Dans un premier temps, pour chaque époque, les coefficient MCEP de la voix A et de la voix B sont coupés au format d'entrée des générateurs et chargés aléatoirement dans les jeux de données :

```
def train(self, coded_sps_A_norm, coded_sps_B_norm):
  file_writer = tf.summary.create_file_writer(self.logdir)
  file_writer.set_as_default()

  time_list = []

  # Boucle sur les époques
  for epoch in range(self.epochs):
    start_time = time.time()
    coded_sps_A_norm_cut, coded_sps_B_norm_cut = utils.cut_coded_sps_norm(coded_sps_A_norm, coded_sps_B_norm, self.n_frames)
    dataset_A = tf.data.Dataset.from_tensor_slices(coded_sps_A_norm_cut)
    dataset_B = tf.data.Dataset.from_tensor_slices(coded_sps_B_norm_cut)

    dataset_A = dataset_A.shuffle(buffer_size=coded_sps_A_norm_cut.shape[0]).batch(self.batch_size)
    dataset_B = dataset_B.shuffle(buffer_size=coded_sps_B_norm_cut.shape[0]).batch(self.batch_size)
```

Pour chaque lot nous appelons la fonction train_step(). Au bout de 10000 itérations, nous passerons le coefficient de la perte d'identité à 0 et nous diminuerons sensiblement le taux d'apprentissage des générateurs et des discriminateurs.

```
    # Boucle sur les lots du jeu de données
    for dataset_A_batch, dataset_B_batch in zip(dataset_A, dataset_B):
      num_iterations = coded_sps_A_norm_cut.shape[0] // self.batch_size * epoch

      if num_iterations > 10000:
        self.lambda_identity = 0
        self.g_learning_rate = max(0, self.g_learning_rate - self.g_learning_rate_decay)
        self.d_learning_rate = max(0, self.d_learning_rate - self.d_learning_rate_decay)
        self.g_optimizer.lr.assign(self.g_learning_rate)
        self.d_optimizer.lr.assign(self.d_learning_rate)
      # Entraine le GAN à l'aide d'un lot d'échantillons de la voix A et de la voix B
      total_gen_loss, total_disc_loss, gen_AB_loss, gen_BA_loss, cycle_A_loss,
      cycle_B_loss, identity_B_loss, identity_A_loss, disc_B_loss, disc_A_loss,
      gen_AB_acc, gen_BA_acc, disc_B_acc, disc_B_real_acc, disc_B_gen_acc, disc_A_acc,
      disc_A_real_acc, disc_A_gen_acc = self.train_step(dataset_A_batch, dataset_B_batch)


    wall_time_sec = time.time() - start_time
    time_list.append(wall_time_sec)
```

Entre 0 et 10000 itérations, le coefficient de la perte d'identité sera assez élevé dans le but d'orienter l'entrainement du GAN à reproduire solidement le signal d'origine et ainsi conserver le contenu du signal vocal. Après 10000 itérations, ce coefficient est passé à 0 pour laisser le GAN évoluer vers la conversion de l'identité vocale.

Nous implémentons la sauvegarde régulière des modèles des générateurs :

```
if (epoch+1) % 20 == 0:
  print("Saving Epoch {}".format(epoch+1))
  ckpt_name = "generateurAB" + "-" + str(epoch+1)
  self.generator_AB.save(self.dir_model + ckpt_name)
  ckpt_name = "generateurBA" + "-" + str(epoch+1)
  self.generator_BA.save(self.dir_model + ckpt_name)
```

Et nous loguons les différentes valeurs utiles au suivi de l'entrainement :

```
# Logue les différentes pertes
tf.summary.scalar('01. Generator loss', total_gen_loss, step=epoch)
tf.summary.scalar('02. Discriminator loss', total_disc_loss, step=epoch)
tf.summary.scalar('03. Generator AB loss', gen_AB_loss, step=epoch)
tf.summary.scalar('04. Discriminator B loss', disc_B_loss, step=epoch)
tf.summary.scalar('05. Generator BA loss', gen_BA_loss, step=epoch)
tf.summary.scalar('06. Discriminator A loss', disc_A_loss, step=epoch)
tf.summary.scalar('07. Cycle A loss', cycle_A_loss, step=epoch)
tf.summary.scalar('08. Cycle B loss', cycle_B_loss, step=epoch)
tf.summary.scalar('09. Identity B loss', identity_B_loss, step=epoch)
tf.summary.scalar('10. Identity A loss', identity_A_loss, step=epoch)
# Logue les différentes performances
tf.summary.scalar('11. Generator accuracy AB', gen_AB_acc, step=epoch)
tf.summary.scalar('12. Discriminator B accuracy', disc_B_acc, step=epoch)
tf.summary.scalar('13. Discriminator B real voice', disc_B_real_acc, step=epoch)
tf.summary.scalar('14. Discriminator B voice generated', disc_B_gen_acc, step=epoch)
tf.summary.scalar('15. Generator accuracy BA', gen_AB_acc, step=epoch)
tf.summary.scalar('16. Discriminator A accuracy', disc_A_acc, step=epoch)
tf.summary.scalar('17. Discriminator A real voice', disc_A_real_acc, step=epoch)
tf.summary.scalar('18. Discriminator A voice generated', disc_A_gen_acc, step=epoch)
```

Enfin, nous générons régulièrement des conversions de tests A->B pour écouter l'évolution de la qualité des échantillons audio générés :

```python
if (epoch) == 0 or (epoch+1) % 5 == 0:
  # Voix A->B
  filenames = list()
  filenames.append(os.listdir(self.dir_wavs_A_test)[0])
  filenames.append(random.choice(os.listdir(self.dir_wavs_A_test)))
  num = 1
  for file in filenames:
    wav, _ = librosa.load(os.path.join(self.dir_wavs_A_test, file), sr = self.sr, mono = True)
    wav = utils.wav_padding(wav, self.sr, self.frame_period, multiple = 4)
    f0, timeaxis, sp, ap = utils.world_decompose(wav, self.sr, self.frame_period)
    f0_converted = utils.pitch_conversion(f0, self.log_f0s_mean_A, self.log_f0s_std_A, self.log_f0s_mean_B, self.log_f0s_std_B)
    coded_sp = utils.world_encode_spectral_envelop(sp, self.sr, self.num_mcep)
    coded_sp_transposed = coded_sp.T
    coded_sp_norm = utils.coded_sp_normalization_fit_transform(coded_sp_transposed, self.coded_sps_A_mean, self.coded_sps_A_std)
    coded_sp_norm = np.array([coded_sp_norm])
    coded_sp_converted_norm = self.generator_AB(coded_sp_norm, training=False)
    coded_sp_converted_norm = np.squeeze(coded_sp_converted_norm)
    coded_sp_converted = utils.coded_sp_denormalization_fit_transform(coded_sp_converted_norm, self.coded_sps_B_mean, self.coded_sps_B_std)
    coded_sp_converted = coded_sp_converted.T
    coded_sp_converted = np.ascontiguousarray(coded_sp_converted)
    sp_converted = utils.world_decode_spectral_envelop(coded_sp_converted, self.sr)

    sp_original_decoded = utils.world_decode_spectral_envelop(coded_sp, self.sr)
    wav_original_decoded = utils.world_speech_synthesis(f0, sp_original_decoded, ap, self.sr, self.frame_period)

    wav_transformed = utils.world_speech_synthesis(f0_converted, sp_converted, ap, self.sr, self.frame_period)

    tf.summary.audio(f'0{num}. Original_A_{file.rsplit(".")[-2]}', tf.expand_dims(tf.expand_dims(wav_original_decoded, -1), 0),
                      sample_rate=self.sr, step=epoch)
    tf.summary.audio(f'0{num+1}. Transformed_A->B_{file.rsplit(".")[-2]}', tf.expand_dims(tf.expand_dims(wav_transformed, -1), 0),
                      sample_rate=self.sr, step=epoch)
```

```python
    fig = pyplot.figure(figsize=(16,5))
    librosa.display.specshow(np.log(sp_original_decoded).T,
                             sr=self.sr,
                             hop_length=int(0.001 * self.sr * self.frame_period),
                             x_axis="time",
                             y_axis="linear",
                             cmap="magma")
    pyplot.colorbar()
    tf.summary.image(f'0{num}. Original A', plot_to_image(fig), step=epoch)

    fig = pyplot.figure(figsize=(16,5))
    librosa.display.specshow(np.log(sp_converted).T,
                             sr=self.sr,
                             hop_length=int(0.001 * self.sr * self.frame_period),
                             x_axis="time",
                             y_axis="linear",
                             cmap="magma")
    pyplot.colorbar()
    tf.summary.image(f'0{num+1}. Transformed A-B', plot_to_image(fig), step=epoch)
    num = num + 2
```

Des échantillons de tests de conversions B->A sont également générés.

Après avoir développé les modèles et les fonctions d'entrainements, nous pouvons maintenant mettre en place le script principal qui nous permettra d'assigner les différents paramètres, d'extraire les caractéristiques des fichiers audio et de lancer l'entrainement du GAN.

```python
# Nombre d'époques
epochs = 3000
# Taille des lots
batch_size = 1
# Taux d'apprentissage des générateurs
g_learning_rate = 0.0002
# Taux d'apprentissage des discriminateurs
d_learning_rate = 0.000005
# Decay
g_learning_rate_decay = g_learning_rate / 200000
d_learning_rate_decay = d_learning_rate / 200000
```

```
# Répertoire de log
logdir="./logs/" + datetime.now().strftime("%Y%m%d-%H%M%S")
# Répertoire Voix A pour l'entrainement
dir_wavs_A_train = "wavs/voix_A_train/"
# Répertoire Voix B pour l'entrainement
dir_wavs_B_train = "wavs/voix_B_train/"
# Répertoire Voix A pour le test
dir_wavs_A_test = "wavs/voix_A_test/"
# Répertoire Voix B pour le test
dir_wavs_B_test = "wavs/voix_B_test/"
# Répertoire de cache
dir_cache = "cache/"
dir_model = "model/"
path_logf0s_norm = os.path.join(dir_cache, 'logf0s_normalization.npz')
path_mcep_norm = os.path.join(dir_cache, 'mcep_normalization.npz')
path_coded_sps_A_norm = os.path.join(dir_cache, "coded_sps_A_norm.pickle")
path_coded_sps_B_norm = os.path.join(dir_cache, "coded_sps_B_norm.pickle")
# Taux d'échantillonnage (Hz)
sr = 22000
# Durée d'une trame (ms)
frame_period = 5.0
# Nombre de dimensions MCEPs souhaitées
num_mcep = 36
# Nombre de trames utilisées pour l'entrainement
# par échantillon
n_frames = 128


print("debut preprocessing")
# Preprocessing pour l'extraction des caractéristiques audios
coded_sps_A_norm, coded_sps_B_norm = utils.preprocessing(dir_wavs_A_train, dir_wavs_B_train, dir_cache, sr, frame_period, num_mcep,
                                                          path_logf0s_norm, path_mcep_norm, path_coded_sps_A_norm, path_coded_sps_B_norm)
print("fin preprocessing")
# Instanciation du GAN
gan = GAN(epochs, batch_size, d_learning_rate, g_learning_rate, g_learning_rate_decay, d_learning_rate_decay,
          logdir, num_mcep, n_frames, sr, frame_period, dir_wavs_A_test, dir_wavs_B_test, path_logf0s_norm, path_mcep_norm, dir_model)

# Entrainement du GAN
gan.train(coded_sps_A_norm, coded_sps_B_norm)
```

Passons maintenant à l'entrainement du GAN et aux résultats obtenus.

4.5.2 Entrainement et résultats

Pour disposer d'échantillons audio, j'ai enregistré le signal vocal de deux personnes différentes à l'aide du logiciel Audacity et d'un micro USB M-Audio.

Chaque locuteur était chargé de lire, pendant environ 13 minutes, les extraits d'un livre sélectionné. A noter que les extraits choisis étaient différents entre chaque locuteur, et ce dans le but d'éviter au maximum les similarités de contenu dans les extraits audio.

L'enregistrement a été réalisé avec un taux d'échantillonnage de 44.1 Khz et a ensuite été découpé en 200 échantillons d'environ 4 secondes à l'aide du logiciel FFmpeg.

Pour chaque personne, 196 échantillons sont destinés exclusivement à l'entrainement et 4 échantillons ont été gardés pour la génération des échantillons de tests de conversions. Le nombre d'échantillons traités par époque sera donc de 392 (196 * 2).

Les échantillons de la voix A contiennent des extraits audio enregistrés avec ma propre voix.

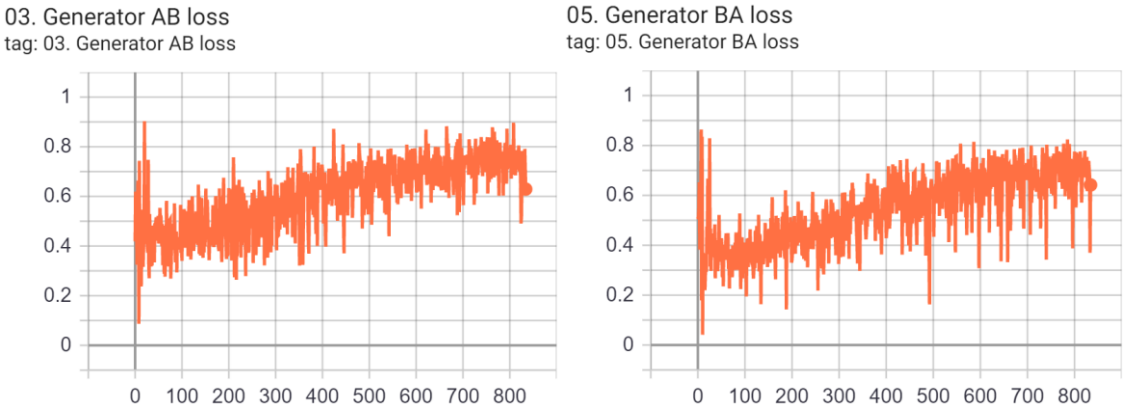
Les échantillons de la voix B contiennent des extraits audio enregistrés par mon conseiller au travail de Bachelor, M. Trabichet.

Le GAN a été entrainé sur 835 époques pendant 3 jours, 16 heures et 12 minutes à l'aide d'une carte graphique NVIDIA Testa T4 sur Google Cloud Platform.

Regardons quelques indicateurs intéressants obtenus durant cet entrainement.

Voici l'évolution de la perte totale des 2 générateurs :

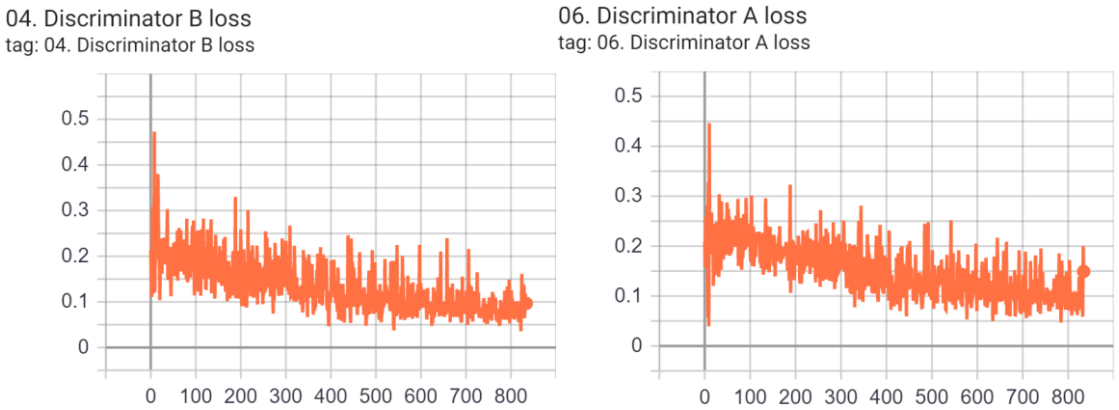
Figure 87 : CycleGAN-voix évolution perte des 2 générateurs



Fait à l'aide de l'outil : TensorBoard

Et l'évolution des pertes des 2 discriminateurs :

Figure 88 : CycleGAN-voix évolution perte des 2 discriminateurs



Fait à l'aide de l'outil : TensorBoard

Pour rappel, l'objectif de chacun de ces réseaux est de minimiser sa perte. On observe donc que les discriminateurs gardent l'ascendant sur les générateurs tout au long de l'entrainement. En effet les discriminateurs parviennent à réduire progressivement leur perte alors que les générateurs tentent de contenir son augmentation.

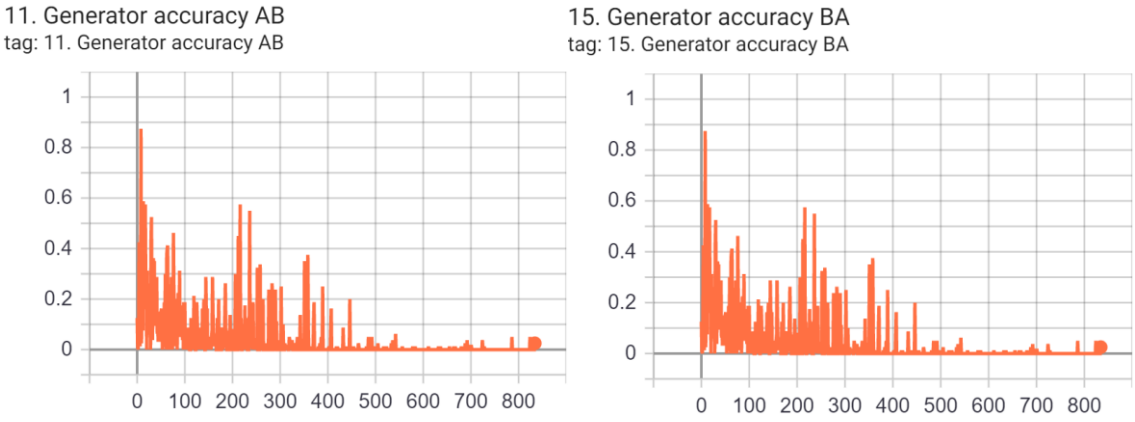
Comme constaté dans l'annexe 1 « Recherche du taux d'apprentissage optimal », l'équilibre souhaité pendant l'entrainement d'un GAN est que les discriminateurs gardent

une légère ascendance sur les générateurs afin de mieux pouvoir les guider durant l'apprentissage. Nous retrouvons donc ici un équilibre tout à fait convenable.

Ce scénario d'ascendance des discriminateurs sur les générateurs se confirme en comparant les autres indicateurs suivants :

Evolution de la performance des 2 générateurs :

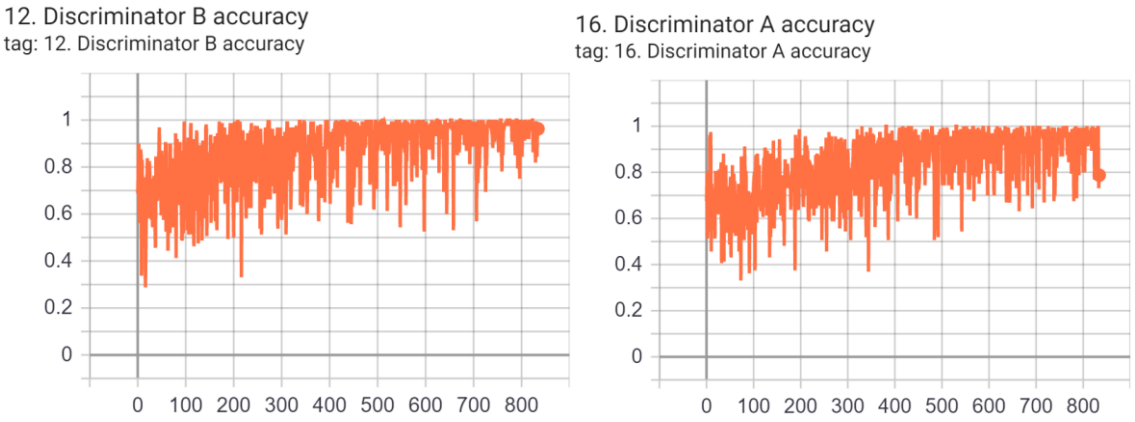
Figure 89 : CycleGAN-voix évolution performance des 2 générateurs



Fait à l'aide de l'outil : TensorBoard

Evolution de la performance des 2 discriminateurs :

Figure 90 : CycleGAN-voix évolution performance des 2 discriminateurs



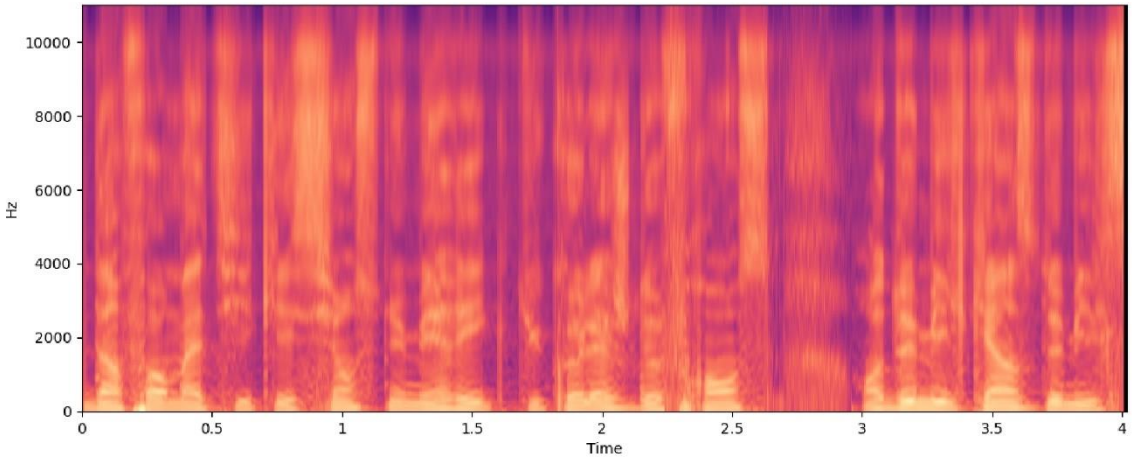
Fait à l'aide de l'outil : TensorBoard

Le suivi de ces différents indicateurs m'a permis de contrôler, tout au long du processus, le bon déroulement de l'apprentissage. En effet, j'ai dû plusieurs fois modifier les taux d'apprentissage des différents réseaux et recommencer entièrement l'apprentissage pour trouver l'équilibre souhaité.

Passons maintenant aux résultats obtenus.

Voici le spectogramme d'un échantillon audio de test de la voix A :

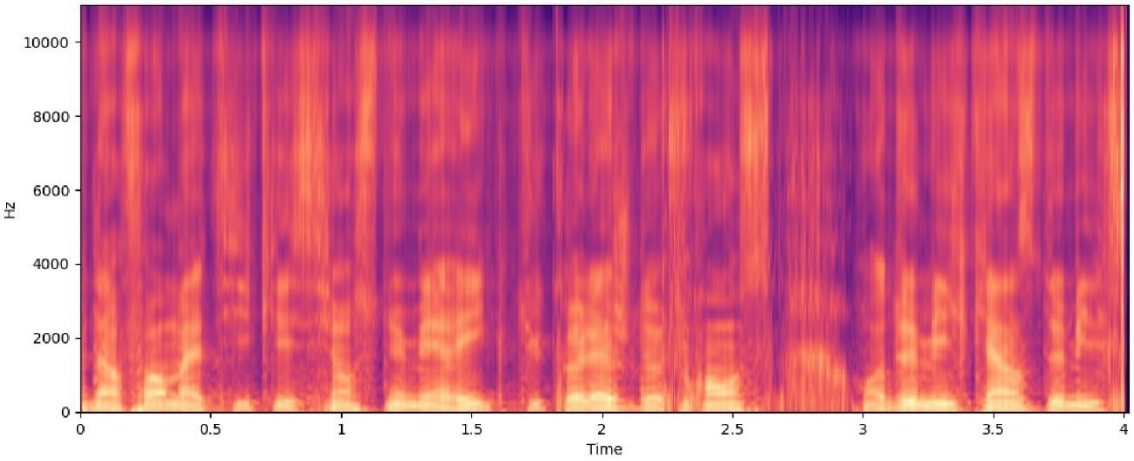
Figure 91 : CycleGAN-voix spectrogramme voix A



Fait à l'aide de l'outil : Librosa

Après avoir entrainé le GAN pendant 835 époques, voici le spectogramme du même échantillon après sa conversion en voix B :

Figure 92 : CycleGAN-voix spectrogramme voix A après conversion

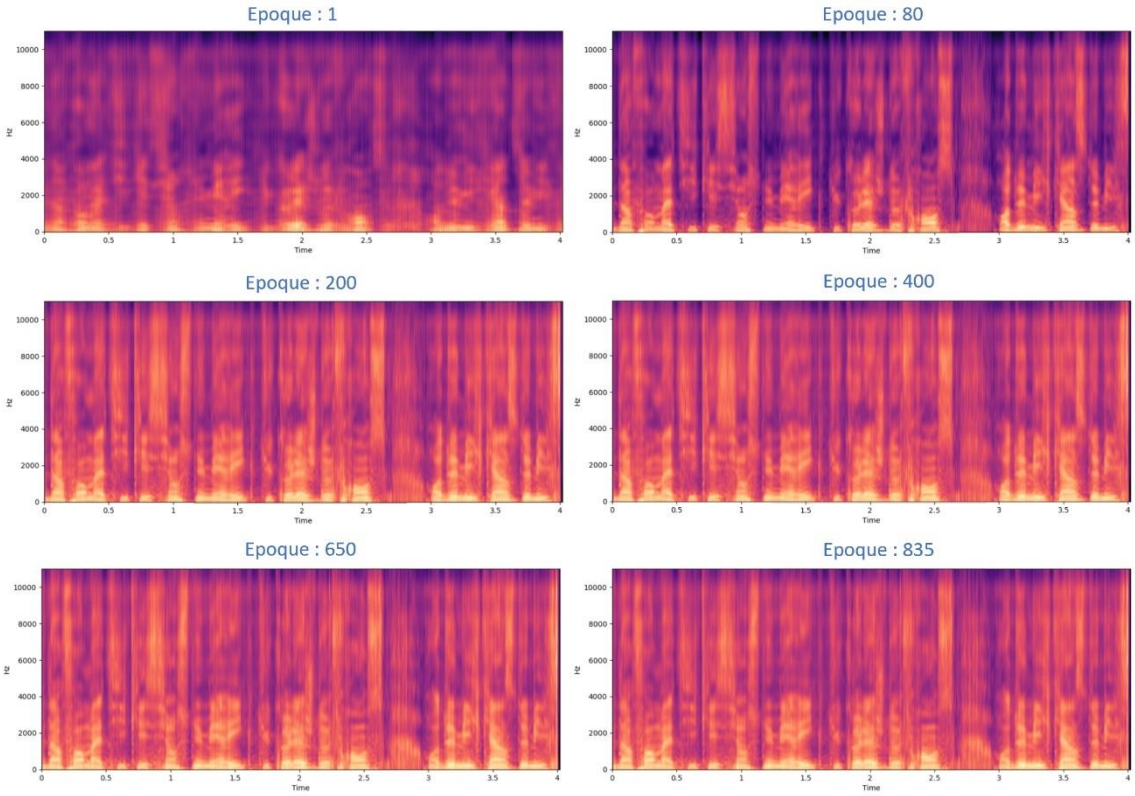


Fait à l'aide de l'outil : Librosa

En visualisant ces deux images, on peut constater de nombreuses différences entre le premier spectogramme et le deuxième, notamment au niveau de l'intensité, de la forme et de la netteté de certaines zones. Toutefois, on retrouve bien les même formes générales ce qui nous laisse supposer qu'une transformation a bien été appliquée sans pour autant altérer totalement le signal d'origine.

Pour illustrer l'impact du nombre d'époque d'entrainement du GAN sur le résultat obtenu, voici les spectogrammes du résultat de la conversion du même échantillon audio de test en fonction des époques :

Figure 93 : CycleGAN-voix évolution des résultats en fonction des époques



Fait à l'aide de l'outil : Librosa

On constate qu'entre l'époque 1 et 200 le contenu du spectogramme se densifie fortement. Puis entre l'époque 200 à 835 les différences deviennent plus difficiles à percevoir à l'œil nu.

Evaluation subjective de l'écoute des échantillons audio de test convertis :

A l'époque 1 : On perçoit le contenu du message, mais il est impossible d'identifier ni le timbre de la voix A, ni celui de la voix B. La voix est très robotisée et le message est mal articulé.

A l'époque 200 : Le contenu du message est clair, bien articulé, et on reconnait le timbre de la voix B qui est toutefois légèrement robotisé.

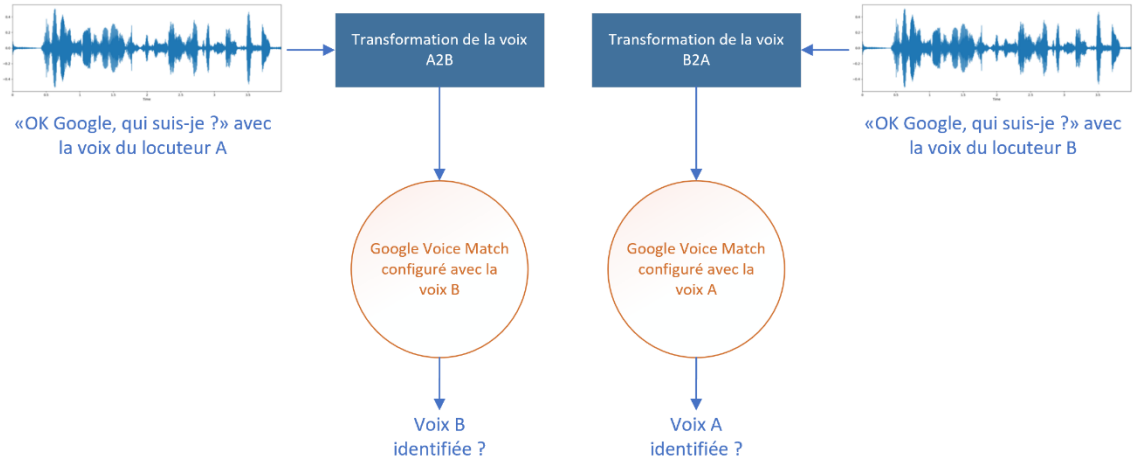
A l'époque 835 : Le contenu du message est clair, bien articulé, et on identifie clairement le timbre de la voix B qui reste toutefois très légèrement robotisé.

Le résultat obtenu après 835 époques est tout à fait acceptable pour l'objectif recherché dans ce travail de Bachelor. En effet, dans un échantillon de voix A converti en voix B, le timbre de la voix B est totalement reconnaissable et le timbre de la voix A n'est plus perceptible. Mêmes observations lors d'une conversion dans l'autre sens, B vers A.

4.5.3 Tests des résultats avec Google Voice Match

Nous allons maintenant tester le comportement du système de reconnaissance vocale Google Voice Match à l'aide de la voix convertie grâce au GAN. En utilisant la voix du locuteur A convertie en locuteur B, nous testerons si Google Voice Match nous identifie comme étant le locuteur B en inversement.

Figure 94 : Tests réalisés avec Google Voice Match



Fait à l'aide de l'outil : Microsoft Visio Professionnel 2016

Pour réaliser ces tests, j'utiliserai un GAN ayant été entrainé avec comme voix A, ma propre voix et comme voix B, la voix de ma compagne Jenny.

Les résultats sont concluants, Google Voice Match identifie bien la voix du locuteur A convertie en locuteur B comme étant le locuteur B :

Figure 95 : Résultat identification voix B générée

Fait à l'aide de l'outil : Capture d'écran Android

Et la voix du locuteur B convertie en locuteur A comme étant le locuteur A :

Figure 96 : Résultat identification voix A générée

Fait à l'aide de l'outil : Capture d'écran Android

On peut observer la fragilité de Google Voice Match face à des échantillons audio générés à l'aide de GAN. Google avait d'ailleurs empêché le déverrouillage des appareils Android avec la voix en 2019 (Pomian-Bonnemaison, 2019). Cependant, Google Voice Match n'est pas abandonné pour autant, une phase de test est d'ailleurs en cours sur Google Home pour permettre aux utilisateurs de confirmer des paiements avec leur voix (Le Goupil, 2020).

5. Conclusion

Ce travail de Bachelor m'a permis de survoler les domaines passionnants que sont les réseaux adverses génératifs et la conversion de voix.

Il est intéressant de constater à quel point les outils de deep learning disponibles aujourd'hui rendent accessible le développement d'idées et l'application de concepts dans des domaines qui étaient jusqu'à très récemment réservés aux experts.

Cette facilité nourrit également le débat en termes de sécurité dans les domaines de l'usurpation biométrique. Cette course à la sécurité n'est pas prête de s'arrêter. Finalement, elle ressemblerait presque au concept GAN dans lequel les spécialistes en sécurité, les discriminateurs, ne cesseront d'évoluer pour tenter de détecter les nouvelles techniques des hackers, les générateurs. Mais dans ce jeu de concurrence-là, j'ai bien peur que les hackers aient toujours une longueur d'avance…

Bibliographie

AHIRWAR, Kailash, 2019. *Generative Adversarial Networks Projects.* Birmingham : Packt Publishing. ISBN 978-1-78913-667-8

ALONZO, Magaly et AUDEVART, Alexia, 2019. *Apprendre demain : Quand intelligence artificielle et neurosciences révolutionnent l'apprentissage*. Malakoff : Dunod. ISBN: 978-2-10-079880-3

ALPHAROL, 2019. Voice-Conversion-CycleGAN2. GitHub [en ligne]. 4 novembre 2019. [Consulté le 26 mars 2021]. Disponible à l'adresse : https://github.com/alpharol/Voice_Conversion_CycleGAN2

ANTIPOV, Grigory, BACCOUCHE, Moez, et DUGELAY, Jean-Luc. Face aging with conditional generative adversarial networks. In : *2017 IEEE international conference on image processing (ICIP)*. IEEE, 2017. p. 2089-2093.

ARJOVSKY, Martin, CHINTALA, Soumith, et BOTTOU, Léon. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

AZENCOTT, Chloé-Agathe, 2019. *Introduction au Machine Learning*. Malakoff : Dunod. ISBN: 978-2-10-080153-4

BABU SAHEER, Lakshmi, 2013. Unified Framework of Feature Based Adaptation for Statistical Speech Synthesis and Recognition [en ligne]. Lausanne : École polytechnique fédérale de Lausanne. Thèse. [Consulté le 14 mai 2021]. Disponible à l'adresse : https://infoscience.epfl.ch/record/185083

Banque nationale suisse. Toutes les séries de billets de banque de la BNS [en ligne]. Sans date. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://www.snb.ch/fr/iabout/cash/history/id/cash_history_overview#

Base de données MNIST. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 25 avril 2021 à 16:26. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Base_de_donn%C3%A9es_MNIST&oldid=182280648

BAYLE, Yann, 2018. Apprentissage automatique de caractéristiques audio : application à la génération de listes de lecture thématiques [en ligne]. Bordeaux : Université de Bordeaux. Thèse. [Consulté le 14 mai 2021]. Disponible à l'adresse : https://tel.archives-ouvertes.fr/tel-01961637/document

BERNICO, Mike, 2018. *Deep Learning Quick Reference*. Birmingham : Packt Publishing. ISBN 978-1-78883-799-6

BORJI, Ali. Pros and Cons of GAN Evaluation Measures. *arXiv preprint arXiv:1802.03446*, 2018.

BROCK, Andrew, DONAHUE, Jeff, et SIMONYAN, Karen. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

BROWNLEE, Jason, 2019a. A Gentle Introduction to Generative Adversarial Networks (GAN) [en ligne]. 19 juillet 2019. [Consulté le 08 mai 2021]. Disponible à l'adresse : https://machinelearningmastery.com/generative_adversarial_networks

BROWNLEE, Jason, 2019b. *Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation.* Machine Learning Mastery, 2019.

DI Wei, BHARDWAJ Anurag, WEI Jianing, 2018. *Deep Learning Essentials*. Birmingham : Packt Publishing. ISBN 978-1-78588-036-0

DOSHI, Ketan, 2021a. Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques. [en ligne]. 11 février 2021. [Consulté le 09 mai 2021]. Disponible à l'adresse : https://towardsdatascience.com/audio-deep-learning-made-simple-part-1-state-of-the-art-techniques-da1d3dff2504

DOSHI, Ketan, 2021b. Audio Deep Learning Made Simple (Part 3): Data Preparation and Augmentation [en ligne]. 24 février 2021. [Consulté le 14 mai 2021]. Disponible à l'adresse : https://towardsdatascience.com/audio-deep-learning-made-simple-part-3-data-preparation-and-augmentation-24c6e1f6b52

EZZINE, Kadria, et MONDHER Frikha. A comparative study of voice conversion techniques: A review. In *2017 International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, pp. 1-6. IEEE, 2017.

Fonction carré. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 12 février 2021 à 08:59. [Consulté le 24 février 2021]. Disponible à l'adresse : http://fr.wikipedia.org/w/index.php?title=Fonction_carr%C3%A9&oldid=179832531

FUCHS, Michael, 2018. Roger Federer vs. Deep Learning: can AI predict Federer's serve ? [en ligne]. Neuchâtel : Université de Neuchâtel. [Consulté le 27 mai 2021]. Disponible à l'adresse : https://www.researchgate.net/publication/327882341_Roger_Federer_vs_Deep_Learning_can_AI_predict_Federer's_serve

GERON, Aurélien, 2019. *Deep Learning avec Keras et TensorFlow : Mise en œuvre et cas concrets.* Ed. 2. Malakoff : Dunod. ISBN: 978-2-10-079066-1

GHARAKHANIAN, Al, 2016. GANs: One of the Hottest Topics in Machine Learning [en ligne]. 19 décembre 2016. [Consulté le 27 mai 2021]. https://www.linkedin.com/pulse/gans-one-hottest-topics-machine-learning-al-gharakhanian

GOODFELLOW, Ian J., POUGET-ABADIE, Jean, MIRZA, Mehdi, et al. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

GOODFELLOW, Ian, 2019. goodfellow_ian. 4.5 years of GAN progress on face generation. Post Twitter [en ligne]. 15 janvier 2019, 01h40. [Consulté le 13 août 2020]. Disponible à l'adresse : https://twitter.com/goodfellow_ian/status/1084973596236144640

GOOGLE CLOUD, 2021. Questions fréquentes [en ligne]. 12 janvier 2021. [Consulté le 21 février 2021]. Disponible à l'adresse : https://cloud.google.com/tpu/docs/faq

GOOGLE DEVELOPERS, 2019. Background: What is a Generative Model? [en ligne]. 24 avril 2019. [Consulté le 18 octobre 2020]. Disponible à l'adresse : https://developers.google.com/machine-learning/gan/generative

GOOGLE TRENDS, 2021. Generative Adversarial Networks [en ligne]. 27 mai 2021. [Consulté le 27 mai 2021]. Disponible à l'adresse : https://trends.google.com/trends/explore?date=2014-01-07%202021-05-27&q=Generative%20Adversarial%20Networks

GUDUGUNTLA, Praneeth, 2021. Understanding Audio: What sound is and how we can leverage it. [en ligne]. 25 février 2021. [Consulté le 09 mai 2021]. Disponible à l'adresse : https://towardsdatascience.com/understanding-audio-what-sound-is-and-how-we-can-leverage-it-1e03d29cd7ce

HARDIK Bansal, ARCHIT Rathore, sans date. Understanding and Implementing CycleGAN in TensorFlow [en ligne]. Sans date. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://hardikbansal.github.io/CycleGANBlog/

HUBEL, David H. et WIESEL, T. N. Shape and arrangement of columns in cat's striate cortex. The Journal of physiology, 1963, vol. 165, no 3, p. 559-568.

ISOLA, Phillip, ZHU, Jun-Yan, ZHOU, Tinghui, et al. Image-to-image translation with conditional adversarial networks. In : *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017. p. 1125-1134.

ISRAEL, Marc, 2019. *Intelligence artificielle avec AWS : Exploitez les services cognitifs d'Amazon*. St Herblain : Editions ENI. ISBN 978-2-409-01945-6

KAMEOKA, Hirokazu, TAKUHIRO Kaneko, KOU Tanaka, et NOBUKATSU Hojo. Stargan-vc: Non-parallel many-to-many voice conversion using star generative adversarial networks. In *2018 IEEE Spoken Language Technology Workshop (SLT)*, pp. 266-273. IEEE, 2018.

KANEKO, Takuhiro, KAMEOKA, Hirokazu, TANAKA, Kou, et al. Cyclegan-vc2: Improved cyclegan-based non-parallel voice conversion. In : *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019. p. 6820-6824.

KARRAS, Tero, AILA, Timo, LAINE, Samuli, et al. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

KARRAS, Tero et al. et Nvidia, 2019. This Person Does Not Exist [en ligne]. [Consulté le 12 août 2020]. Disponible à l'adresse : https://www.thispersondoesnotexist.com

KUN Ma, 2020. CycleGAN-VC2-PyTorch. GitHub [en ligne]. 27 novembre 2020. [Consulté le 26 mars 2021]. Disponible à l'adresse : https://github.com/jackaduma/CycleGAN-VC2

LE CUN, Yann. What are some recent and potentially upcoming breakthroughs in deep learning?. *Machine learning forums* [en ligne]. 28 juillet 2016. [Consulté le 13 août 2020]. Disponible à l'adresse : https://www.quora.com/What-are-some-recent-and-potentially-upcoming-breakthroughs-in-deep-learning

LE CUN, Yann, 2019. *Quand la machine apprend. La révolution des neurones artificiels et de l'apprentissage profond*. Paris : Odile Jacob. ISBN 978-2-7381-4931-2

LE GOUPIL, Pierre, 2020. Google Assistant va permettre de payer en ligne juste avec votre voix. PhonAndroid [en ligne]. 26 mai 2020. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://www.phonandroid.com/google-assistant-va-permettre-de-payer-en-ligne-juste-avec-votre-voix.html

NAUSHAD, Raoof, 2020. CPU, GPU and TPU — Machine Learning [en ligne]. 17 juin 2020. [Consulté le 21 février 2021]. Disponible à l'adresse : https://medium.com/dataseries/cpu-gpu-and-tpu-machine-learning-5033e25b8a0f

Nvidia GeForce MX150 vs Nvidia Tesla T4. Versus [en ligne]. Sans date. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://versus.com/fr/nvidia-geforce-mx150-vs-nvidia-tesla-t4#group_performance

PATHAK, Deepak, KRAHENBUHL, Philipp, DONAHUE, Jeff, et al. Context encoders: Feature learning by inpainting. In : *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016. p. 2536-2544.

Perceptron multicouche. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 23 juin 2020 à 18:54. [Consulté le 13 août 2020]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Perceptron_multicouche&oldid=172277711

PHONICAL, 2017. View of a signal in the time and frequency domain [en ligne]. 30 novembre 2017. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://commons.wikimedia.org/wiki/File:FFT-Time-Frequency-View.png

POMIAN-BONNEMAISON, Romain, 2019. “Ok Google” et Voice Match ne peuvent plus déverrouiller votre smartphone. PhonAndroid [en ligne]. 7 mars 2019. [Consulté le 28 mai 2021]. Disponible à l’adresse : https://www.phonandroid.com/ok-google-et-voice-match-ne-peuvent-plus-deverrouiller-votre-smartphone.html

PRATHEEKSHA, Nair, 2018. The dummy’s guide to MFCC [en ligne]. 24 juillet 2018. [Consulté le 28 mai 2021]. Disponible à l’adresse : https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd

RADFORD, Alec, METZ, Luke, et CHINTALA, Soumith. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

RAMON Y CAJAL Santiago, 1899. *Comparative study of the sensory areas of the human cortex.* Worcester : Clark University. ISBN 978-1-45882-189-8

Réseau de neurones artificiels. Wikipédia : l’encyclopédie libre [en ligne]. Dernière modification de la page le 20 février 2021 à 20:28. [Consulté le 27 mai 2021]. Disponible à l’adresse : https://fr.wikipedia.org/w/index.php?title=R%C3%A9seau_de_neurones_artificiels&oldid=180115501

ROBERTS, Leland, 2020. Understanding the Mel Spectrogram. [en ligne]. 6 mars 2020. [Consulté le 14 mai 2021]. Disponible à l’adresse : https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53

RODER, Stéphane, 2019. *Guide pratique de l'intelligence artificielle dans l'entreprise : Anticiper les transformations, mettre en place des solutions.* Ed. 1. Paris : Editions Eyrolles. ISBN: 978-2-212-57122-6

RUKUBAYIHUNGA, Alia, 2020. Forbe : Deepfake : Menace imminente ou outil d’aide à la production [en ligne]. 01 mai 2020. [Consulté le 01 mai 2021]. Disponible à l’adresse : https://www.forbes.fr/technologie/deepfake-menace-imminente-ou-outil-daide-a-la-production/

SAED, Sayad, sans date. Artificial Neural Network - Perceptron [en ligne]. [Consulté le 27 mai 2021]. Disponible à l’adresse : https://www.saedsayad.com/artificial_neural_network_bkp.htm

SAMUEL, Arthur L. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 1959, vol. 3, no 3, p. 210-229.

SECKEL, Nancy et SINGH, Adi. Physics of 3d ultrasonic sensors. *Toposens GmbH, Tech. Rep.*, 2019.

SERRANO, Luis, 2020. A Friendly Introduction to Generative Adversarial Networks (GAN) [enregistrement vidéo]. YouTube [en ligne]. 5 mai 2020. [Consulté le 21 octobre 2020]. Disponible à l’adresse : https://www.youtube.com/watch?v=8L11aMN5KY8

SHATRI, Elona, 2020. A review on Generative Adversarial Networks: How did the GANs change the way machine learning works? [en ligne]. 16 juin 2020. [Consulté le 14 mai 2021]. Disponible à l’adresse : https://towardsdatascience.com/a-review-of-generative-adversarial-networks-9af21e94bda4

SHRAWANKAR, U.et THAKARE, V.M. Techniques for feature extraction in speech recognition system: A comparative study. *arXiv preprint arXiv:1305.1145*, 2013.

SISMAN, Berrak, YAMAGISHI, Junichi, KING, Simon, et al. An overview of voice conversion and its challenges: From statistical modeling to deep learning. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 2020.

Son (physique). Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 26 mai 2021 à 20:39. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Son_(physique)&oldid=183289496

STACK OVERFLOW TRENDS, 2021. Tensorflow, pytorch et scikit-learn [en ligne]. 28 mai 2021. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://insights.stackoverflow.com/trends?tags=tensorflow%2Cpytorch%2Cscikit-learn

STEVENS, Stanley S. et VOLKMANN, John. The relation of pitch to frequency: A revised scale. *The American Journal of Psychology*, 1940, vol. 53, no 3, p. 329-353.

TAKUHIRO, Kaneko et KAMEOKA Hirokazu. Cyclegan-vc: Non-parallel voice conversion using cycle-consistent adversarial networks. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pp. 2100-2104. IEEE, 2018.

TensorBoard. TensorFlow [En ligne]. Sans date. [Consulté le 26 février 2021]. Disponible à l'adresse : https://www.tensorflow.org/tensorboard?hl=fr

TensorFlow. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 19 janvier 2021 à 12:33. [Consulté le 21 février 2021]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=TensorFlow&oldid=178975575

TensorFlow Core. TensorFlow [en ligne]. Dernière modification de la page le 5 février 2021. [Consulté le 26 février 2021]. Disponible à l'adresse : https://www.tensorflow.org/overview

The Physics of Sound. Future Learn [en ligne]. Sans date. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://www.futurelearn.com/info/courses/representing-data-with-images-and-sound/0/steps/53151

THEIS, Lucas, VAN DEN OORD, Aäron et BETHGE, Matthias, A note on the evaluation of generative models. arXiv preprint arXiv:1511.01844, 2016.

VALLE, Rafael, 2019. *Hands-On Generative Adversarial Networks with Keras*. Birmingham : Packt Publishing. ISBN 978-1-78953-820-5

VANNIEUWENHUYZE, Aurelien, 2019. *Intelligence artificielle vulgarisée : Le Machine Learning et le Deep Learning par la pratique*. St Herblain : Editions ENI. ISBN 978-2-409-02073-5

Voix humaine. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 17 mai 2021 à 08:32. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://fr.wikipedia.org/w/index.php?title=Voix_humaine&oldid=182981567

WORLD.jl [en ligne]. Sans date. [Consulté le 28 mai 2021]. Disponible à l'adresse : https://r9y9.github.io/WORLD.jl/latest/index.html

ZHANG, Han, XU, Tao, LI, Hongsheng, et al. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In : *Proceedings of the IEEE international conference on computer vision.* 2017. p. 5907-5915.

ZHANG, Mingyang, SISMAN, Berrak, RALLABANDI, Sai Sirisha, et al. Error reduction network for dblstm-based voice conversion. In : *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 2018. p. 823-828.

ZHU, Jun-Yan, PARK, Taesung, ISOLA, Phillip, et al. Unpaired image-to-image translation using cycle-consistent adversarial networks. In : *Proceedings of the IEEE international conference on computer vision.* 2017. p. 2223-2232.

Annexe 1 : Recherche du taux d'apprentissage optimal

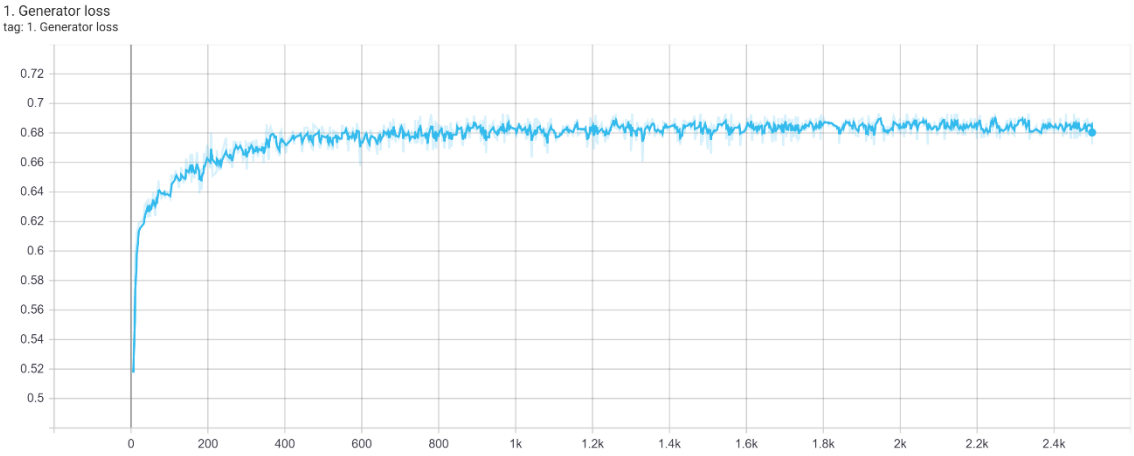
Les tests sont réalisés à l'aide d'un GAN 1D sur un jeu de données de 1280 vrais échantillons, pendant 2500 époques en utilisant une taille de lot de 128 échantillons.

Entrainement avec la mise à jour des poids du générateur désactivée :

Pour commencer, je vais tout d'abord désactiver totalement la mise à jour des poids du générateur, afin de pouvoir analyser l'évolution de la perte et les performances des deux réseaux lorsque le discriminateur prend totalement l'avantage sur le générateur.

Le taux d'apprentissage du discriminateur est fixé à 0.001, qui est la valeur par défaut proposée par Tensorflow.

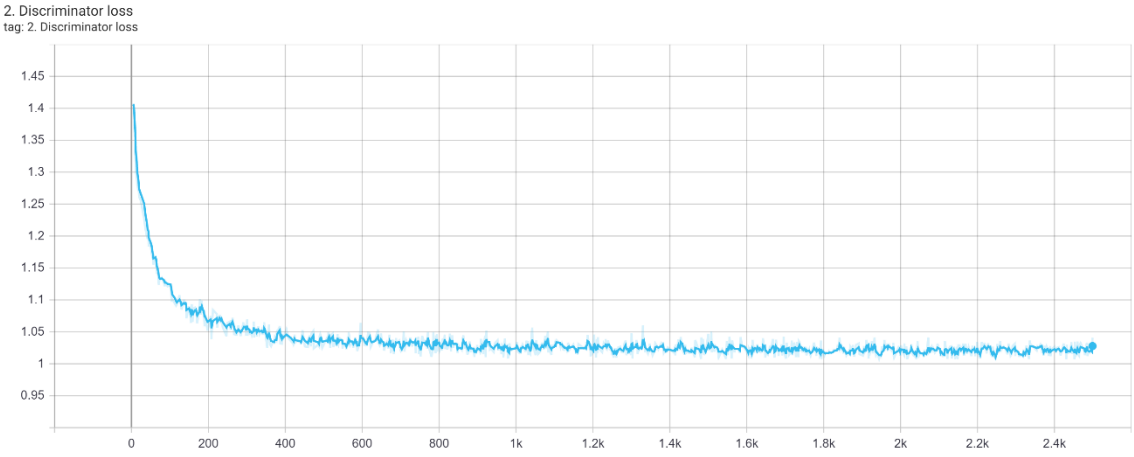
Voici la courbe représentant les variations de la perte du générateur :



Fait à l'aide de l'outil : TensorBoard

On peut constater que la perte du générateur augmente très rapidement les 400 premières époques pour se stabiliser autour de 0.68. On peut supposer que la valeur de perte ne diminue à aucun moment de façon significative parce que le générateur ne progresse pas. En effet, si la mise à jour des poids n'était pas volontairement désactivée, le générateur chercherait à minimiser cette valeur par son algorithme d'optimisation en adaptant ses poids.

Voici la courbe représentant les variations de la perte du discriminateur :



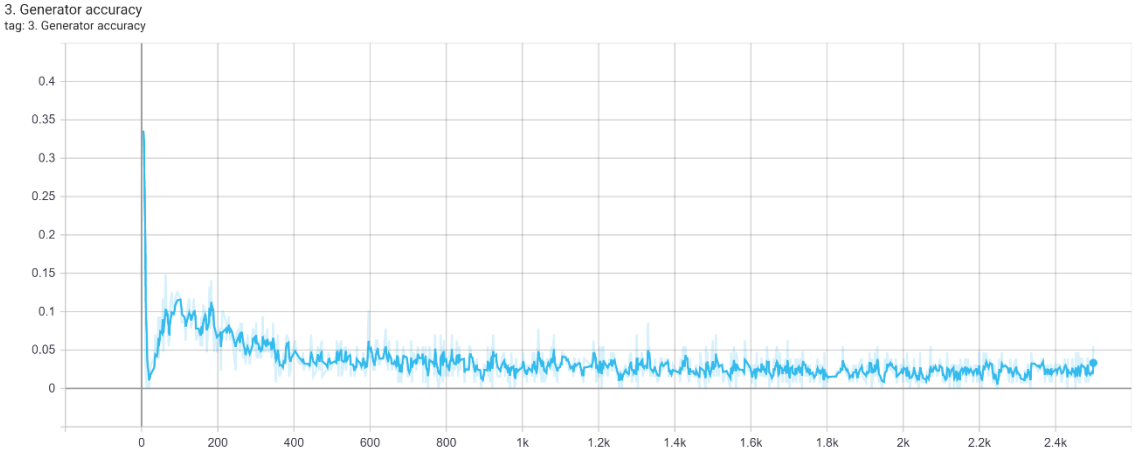
Fait à l'aide de l'outil : TensorBoard

A noter que les échelles sont différentes entre le générateur (sur 1) et le discriminateur (sur 2) mais nous nous concentrerons surtout sur la variation générale de ces valeurs.

Du côté du discriminateur, on constate une situation contraire. En effet, la perte du discriminateur tombe assez rapidement pour se stabiliser autour de 1. On peut supposer que le discriminateur évolue fortement et rapidement durant les 400 premières époques, pour ensuite affiner modérément son apprentissage jusqu'à 2500 époques. Etant donné que la progression du générateur est volontairement désactivée, le discriminateur n'a pas à gérer l'évolution qualitative des faux échantillons. Il peut donc progresser sans perturbation et peut rapidement différencier efficacement les vrais des faux échantillons.

Pour conforter ces observations, regardons maintenant les courbes représentant l'évolution des performances de ces deux réseaux.

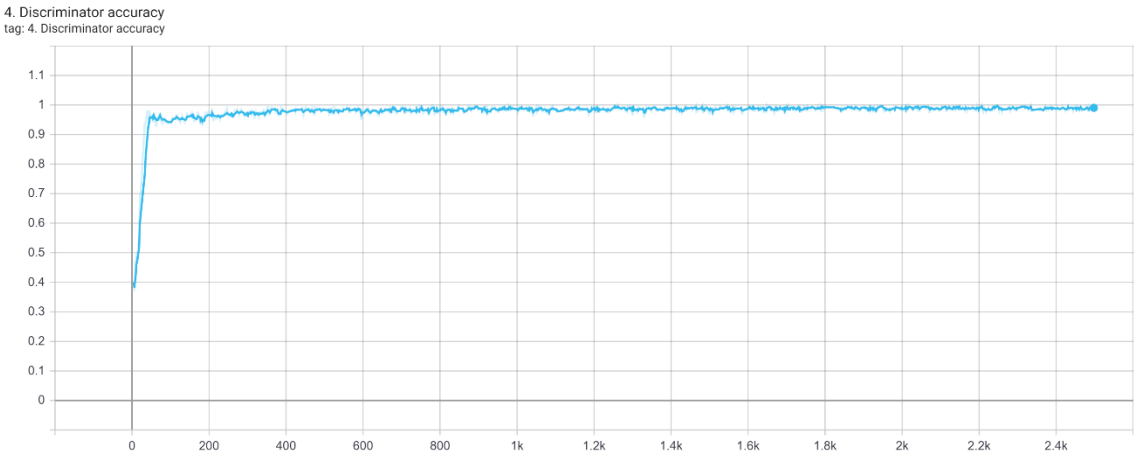
Courbe de variation des performances du générateur :



Fait à l'aide de l'outil : TensorBoard

La performance du générateur tombe dès le début de l'entrainement pour rester très légèrement au-dessus de 0%. Cette très mauvaise performance veut dire que très peu des faux échantillons générés ne sont classifiés comme « vrai » par le discriminateur.

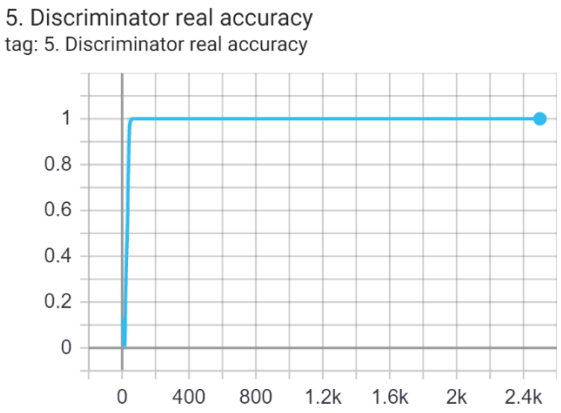
Courbe de variation des performances du discriminateur :

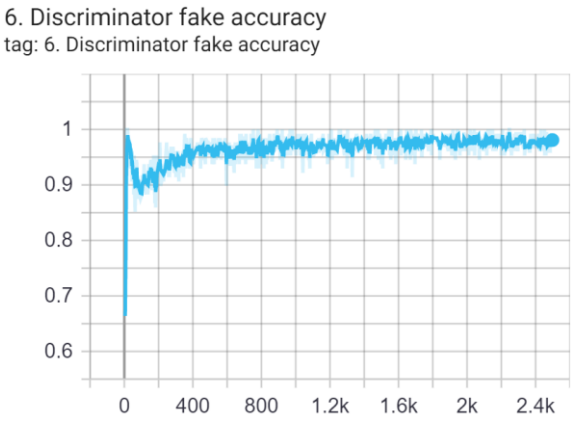


Fait à l'aide de l'outil : TensorBoard

La performance du discriminateur augmente très rapidement pour rester très proche de 100%. Cette courbe illustre la progression très rapide du générateur qui est capable, seulement après quelques époques, de classifier correctement la quasi-totalité des vrais et des faux échantillons.

Voici d'ailleurs les performances du discriminateur par type d'échantillons :

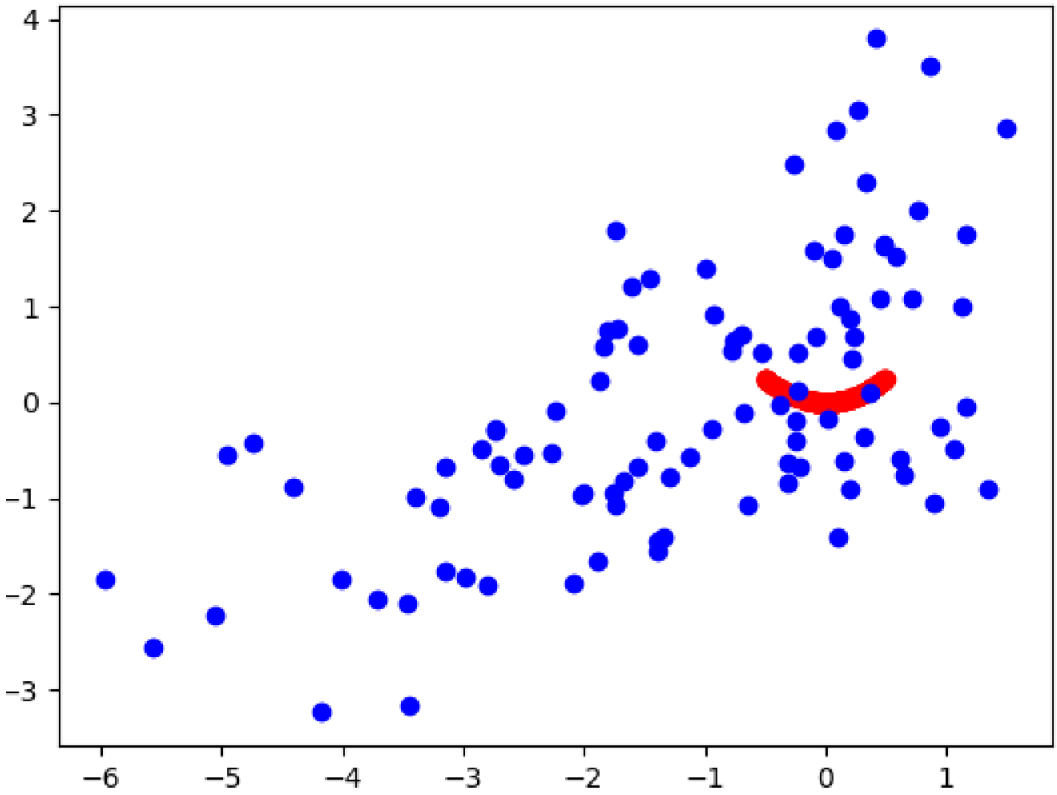




Fait à l'aide de l'outil : TensorBoard

Ces graphiques confirment la capacité du discriminateur à classifier aussi bien les vrais que les faux échantillons.

Enfin, si on visualise les faux échantillons générés par le générateur après 2500 époques, on constate bien que celui-ci génère toujours des faux échantillons totalement aléatoirement (en bleus) et bien loin des vrais échantillons (en rouge).

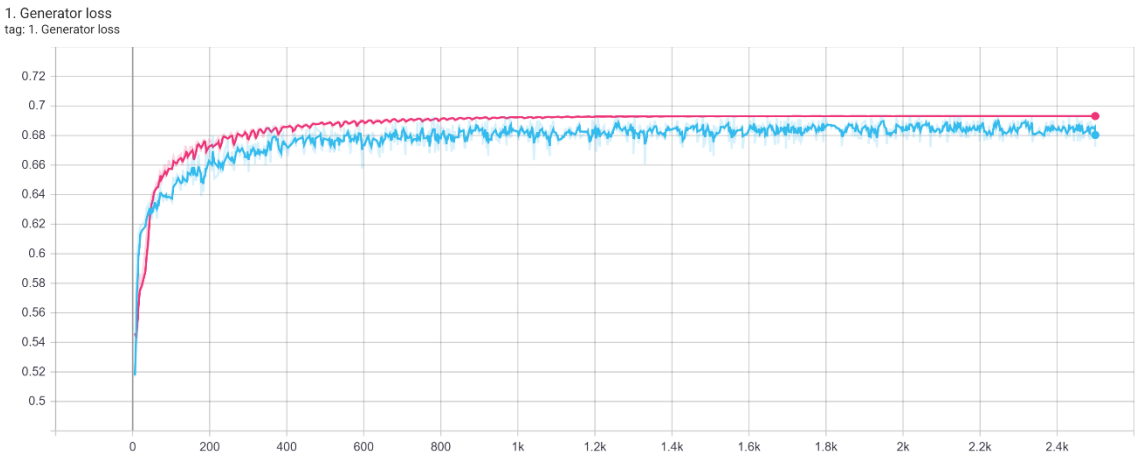


Fait à l'aide de l'outil : TensorBoard

Après avoir observé le comportement d'un GAN totalement dominé par le discriminateur, activons maintenant la mise à jour des poids du générateur et fixons-lui un taux d'apprentissage également à 0.001.

<u>Entrainement concurrentiel des deux réseaux avec un taux d'apprentissage à 0.001</u>

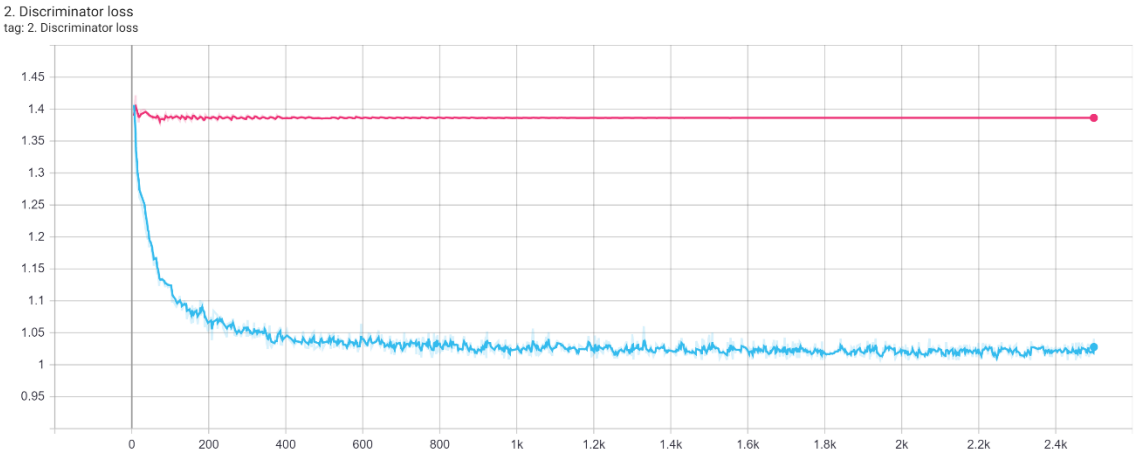
Après avoir recommencé l'entrainement en activant la mise à jour des points des deux réseaux ainsi qu'un taux d'apprentissage à 0.001, voici la courbe représentant la variation de la perte du générateur (en rouge) :



Fait à l'aide de l'outil : TensorBoard

On peut constater que la courbe est très similaire à l'entrainement précédent (en bleu). En effet la perte du générateur augmente très vite pour se stabiliser autour de 0.68, on pourrait suggérer que le générateur n'évolue pas correctement. Avec une perte si élevée, on pourrait supposer que les faux échantillons générés par le générateur sont très vite catégorisés en « faux » par le discriminateur.

Voici la courbe représentant la variation de la perte du générateur (en rouge) :



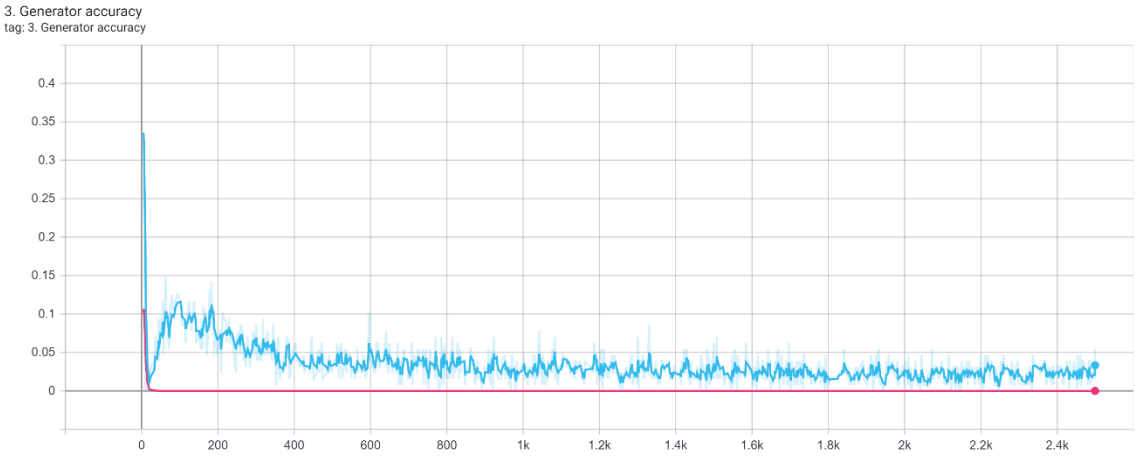
Fait à l'aide de l'outil : TensorBoard

La courbe stagne très rapidement autour de 1.37 alors que lors de l'entrainement précédent (en bleu), la perte du discriminateur était descendue assez rapidement pour se stabiliser autour de 1 et le discriminateur avait été très bien entrainé.

On pourrait donc supposer, en constatant la stagnation de la perte, que le discriminateur n'arrive pas à apprendre suffisamment.

Regardons maintenant les courbes de performances pour essayer de mieux isoler la problématique.

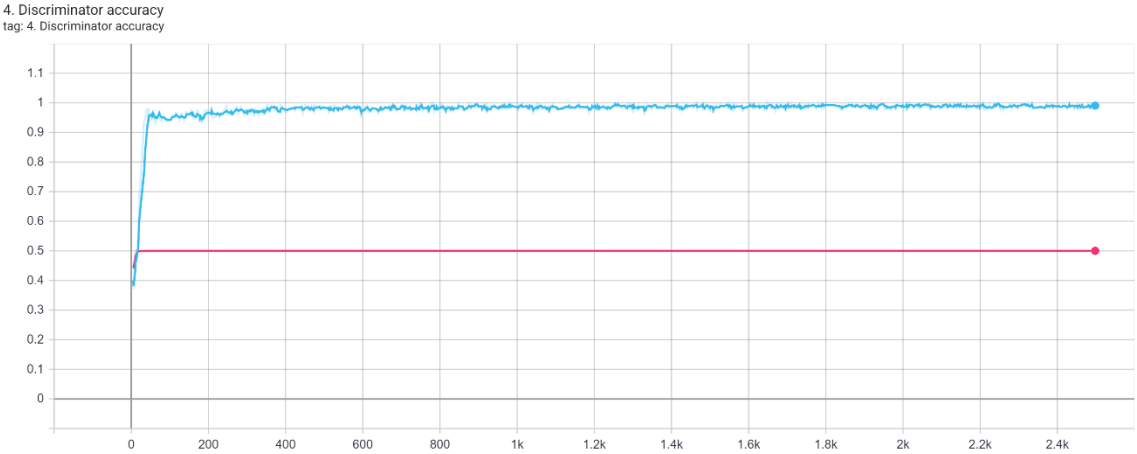
Courbe de variation des performances du générateur (en rouge) :



Fait à l'aide de l'outil : TensorBoard

Ce graphique est très révélateur. En effet, la performance du générateur chute à 0% après seulement quelques époques ce qui est encore moins performant que lors de l'entrainement précédent (en bleu), avec un générateur dont l'évolution était désactivée. Ce qui veut dire qu'après très peu de temps, 100% des faux échantillons du générateur sont détectés comme « faux » par le discriminateur.

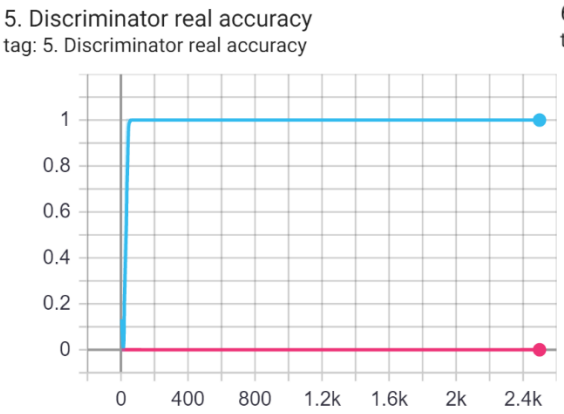
Courbe de variations des performances du discriminateur (en rouge) :

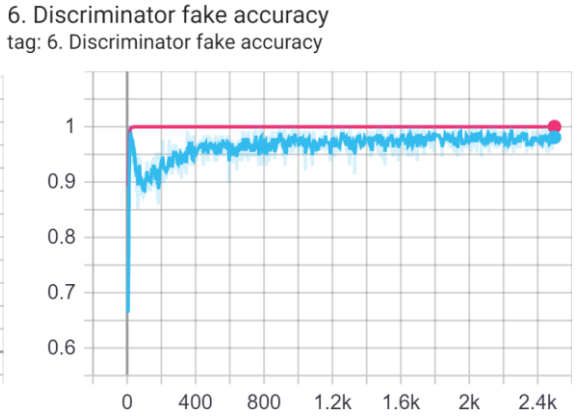


Fait à l'aide de l'outil : TensorBoard

Cette courbe nous indique qu'après seulement quelques époques, le discriminateur classifie correctement 50% des faux et des vrais échantillons. Sachant qu'il classifie correctement 100% des faux échantillons (constaté dans le graphique précédent), cela voudrait dire qu'il classifie correctement 0% des vrais échantillons.

Vérifions cela avec la performance détaillée du discriminateur (en rouge) avec les vrais échantillons puis les faux échantillons :





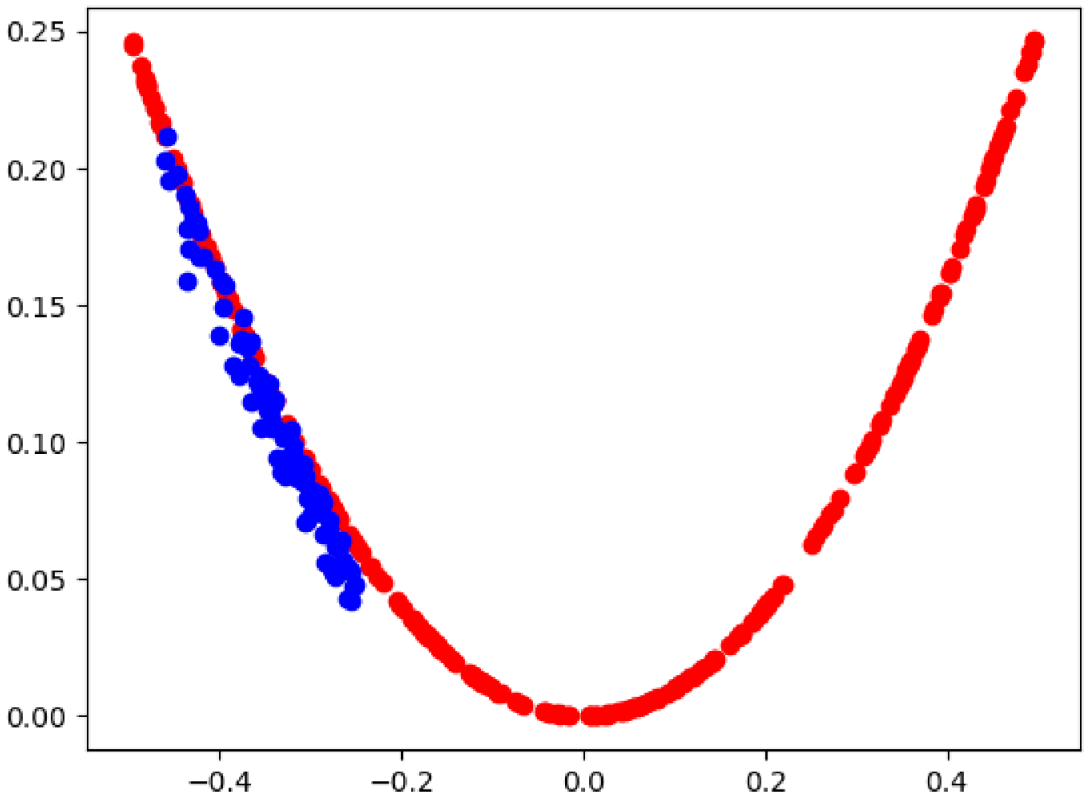
Fait à l'aide de l'outil : TensorBoard

Ces graphiques confirment ce qui a été précédemment constaté, le discriminateur classifie correctement 0% des vrais échantillons et 100% des faux échantillons.

Ce qui veut dire que lors de cet entrainement, le discriminateur s'est mis très rapidement à retourner systématiquement un pourcentage inférieur à 0.5, ce qui revient à classifier chaque échantillon en « faux ». Cependant, durant environ la première moitié de l'entrainement, le pourcentage retourné par le discriminateur a toujours été légèrement supérieur à 0, ce qui a permis quand même au générateur de progresser sensiblement.

Le fait que le discriminateur classifie tous les échantillons en « faux » laisse à penser que celui-ci s'est fait très rapidement dominer par le générateur et donc n'arrive pas à progresser suffisamment pour prédire ne serait-ce qu'une partie des vrais échantillons comme « vrai ». Dans son rapport de dominance, le générateur, qui est dépendant de la progression du discriminateur pour évoluer correctement, progresse lentement et pas idéalement.

Voyons maintenant les faux échantillons générés par le générateur après 2500 époques. On constate que celui-ci génère des faux échantillons (en bleus) plus proche des vrais échantillons (en rouge) que lors de l'entrainement précédent. Cependant, on peut observer que les faux échantillons ne sont pas bien répartis.



Fait à l'aide de l'outil : TensorBoard

Ce graphique nous confirme que le générateur apprend quand même lorsqu'il prend le dessus sur le discriminateur, mais pas suffisamment bien. On peut donc supposer que pour qu'un générateur sache produire des faux échantillons réalistes il faut que le discriminateur puisse suffisamment progresser pour faire évoluer le générateur dans la bonne direction.

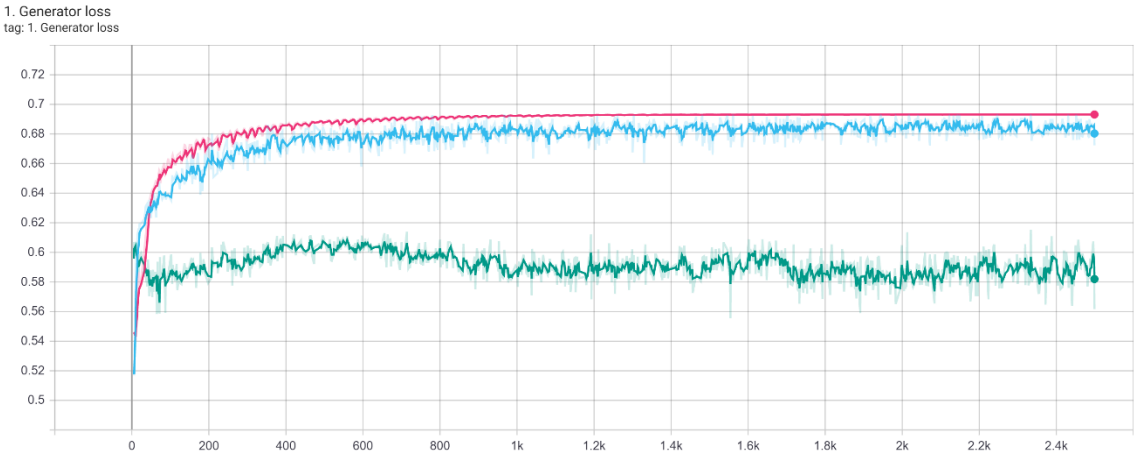
Pour corroborer l'hypothèse que cette situation a été provoquée par la dominance du générateur sur le discriminateur, je vais chercher à contrebalancer cette dominance en diminuant le taux d'apprentissage du générateur et en augmentant le taux d'apprentissage du discriminateur.

L'idéal serait de trouver un équilibre dans lequel le discriminateur arriverait à prédire suffisamment d'échantillons réels afin de pouvoir guider plus efficacement le générateur dans son apprentissage. Le tout en évitant qu'un des deux réseaux prenne l'ascendant sur l'autre.

Après avoir effectué différents tests, je propose de réaliser un nouvel entrainement avec un taux d'apprentissage à 0.0003 pour le générateur et à 0.0036 pour le discriminateur.

Entrainement avec un taux d'apprentissage à 0.0003 pour le générateur et à 0.0036 pour le discriminateur:

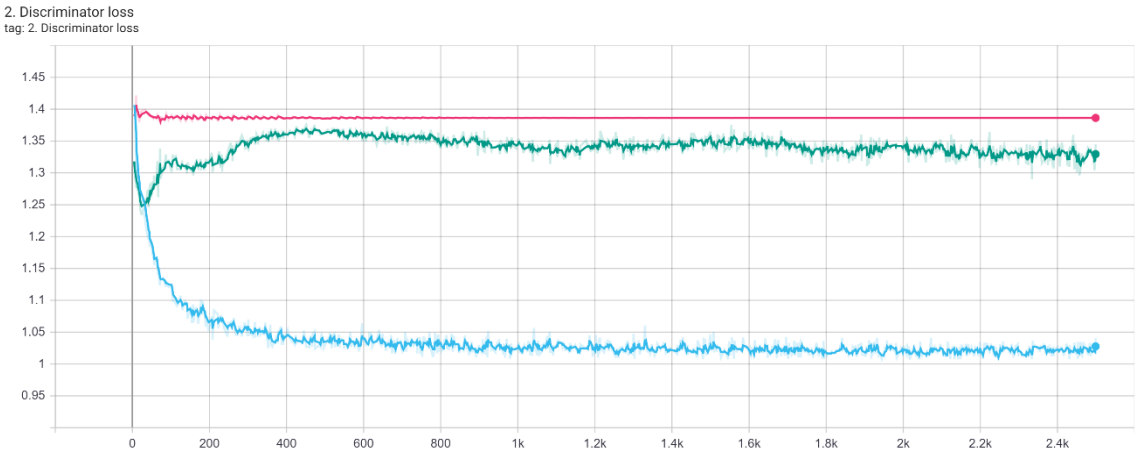
Voici la courbe représentant les variations de la perte du générateur avec le nouveau taux d'apprentissage (en vert) :



Fait à l'aide de l'outil : TensorBoard

On peut constater que la perte du générateur reste en dessous des deux précédents entrainements (en rouge et bleu). On peut suggérer qu'un nombre plus important de faux échantillons sont classifiés comme « vrai » par le discriminateur. De plus, on constate des variations de valeur légèrement plus importantes, ce qui laisse à penser que le générateur réalise des cycles de progression en synchronisation avec le discriminateur. On peut également émettre l'hypothèse qu'il est positif pour l'entrainement du générateur que la perte ne descende pas plus bas. En effet cela voudrait dire que le discriminateur garde tout de même une certaine avance et donc guiderait plus efficacement le générateur.

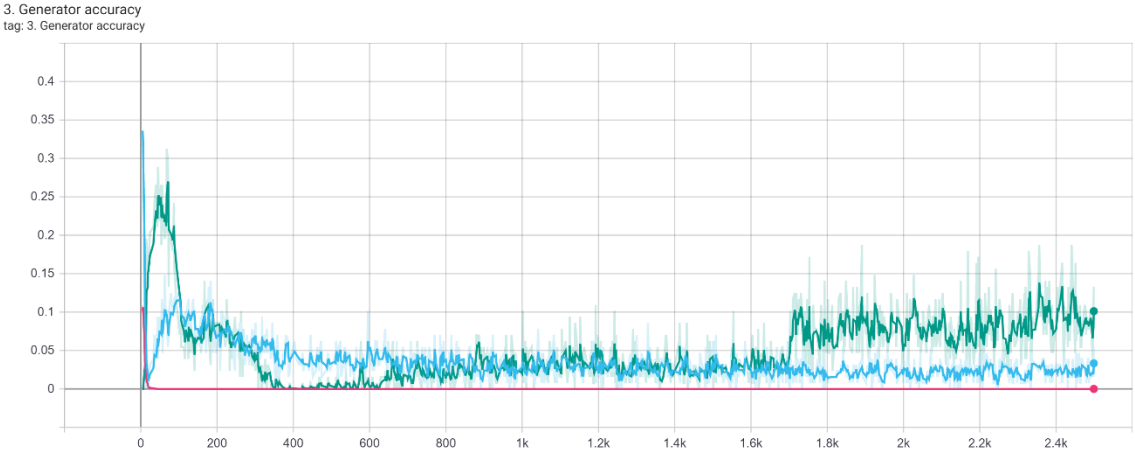
Voici la courbe représentant les variations de la perte du générateur avec le nouveau taux d'apprentissage (en vert) :



Fait à l'aide de l'outil : TensorBoard

On peut observer que la perte du discriminateur reste bien plus élevée que lors du premier entrainement (en bleu) pendant lequel le discriminateur était entrainé sans être concurrencé par le générateur. Cependant, elle reste légèrement plus basse que lors de l'entrainement précédent (en rouge) dans lequel je supposais une domination du générateur sur le discriminateur. On peut également constater les mêmes oscillations de la valeur de perte que lorsque le discriminateur progressait efficacement (en bleu). Je suppose donc que le discriminateur reste en avance sur le générateur mais qu'il classifie tout de même quelques faux échantillons comme « vrai », ce qui permet au générateur de rester dans la course.

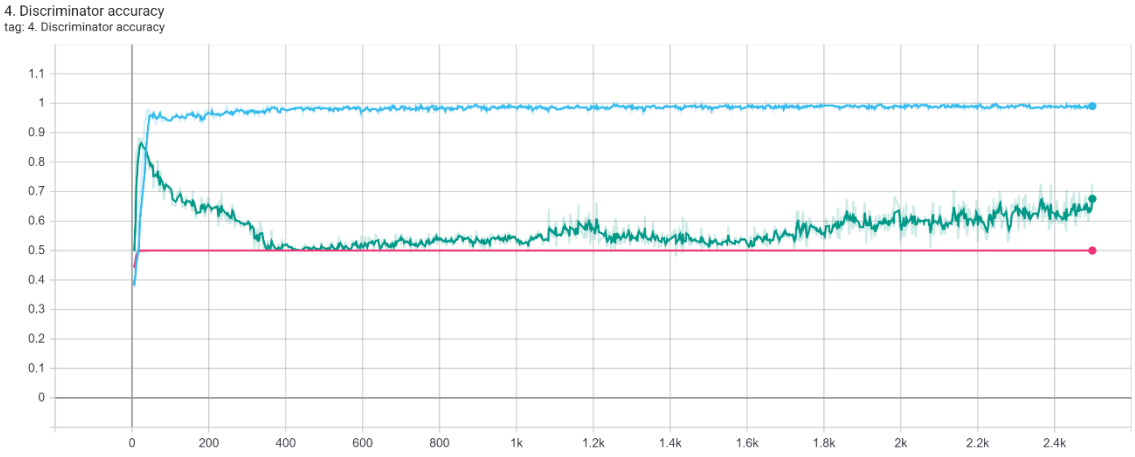
Courbe de variations des performances du générateur (en vert) :



Fait à l'aide de l'outil : TensorBoard

On observe que, contrairement aux précédents entrainement (en rouge et bleu), la performance du générateur est en légère progression. On peut déduire qu'entre 700 et 1700 époques, environ 5% des faux échantillons étaient classifiés comme « vrai » par le discriminateur, et qu'entre 1700 et 2500 époques, c'est 7.5% des faux échantillons qui sont classifiés comme « vrai » par le discriminateur. On peut donc déduire que le discriminateur est plus efficace à classifier les faux échantillons que le générateur à produire des échantillons réalistes. Cependant le générateur progresse et arrive tout de même à faire classifier ses faux échantillons comme « vrai » par le discriminateur.

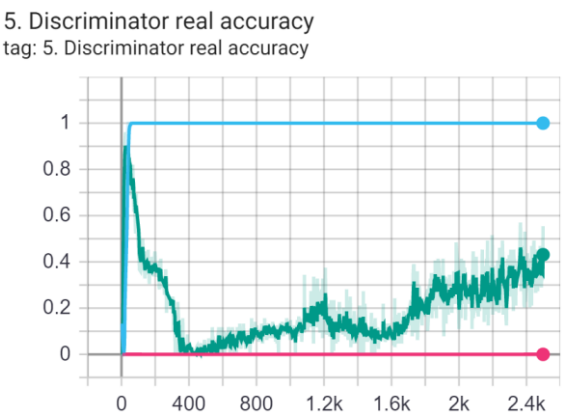
Courbe de variations des performances du discriminateur (en vert) :

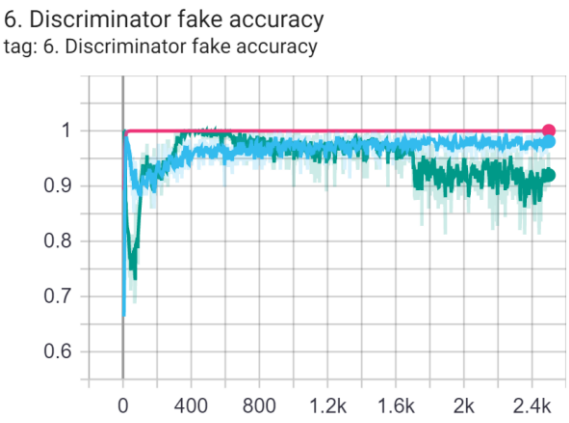


Fait à l'aide de l'outil : TensorBoard

On peut constater que le discriminateur classifie plus de 50% des échantillons correctement ce qui nous permet d'éliminer le comportement rencontré lors de l'entrainement précédent (en rouge), dans lequel le discriminateur classifiait tous les échantillons comme « faux ». De plus, on observe que le discriminateur n'est pas aussi rapidement entrainé que lors du premier entrainement (en bleu). En effet on pourrait supposer que le générateur en concurrence progresse suffisamment vite pour freiner la progression du discriminateur en gardant ses faux échantillons à un niveau de qualité suffisamment élevé.

Observons maintenant les performances détaillées du discriminateur (en rouge) avec les vrais échantillons puis les faux échantillons :





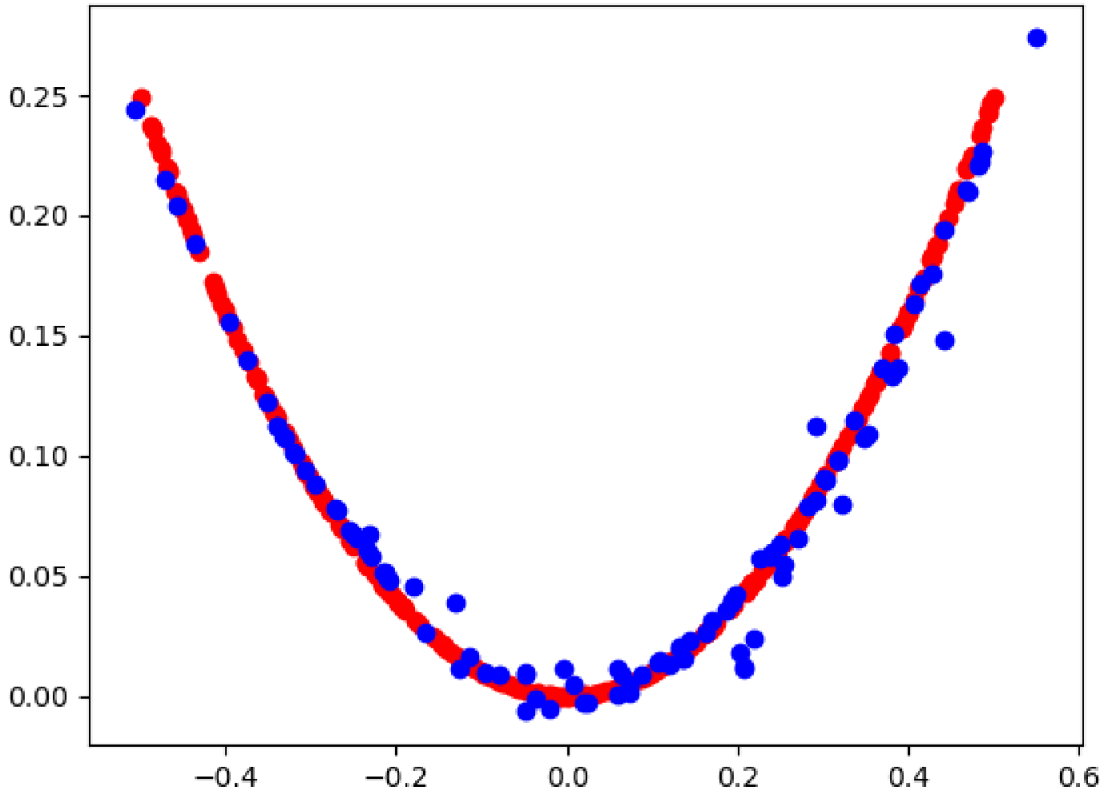
Fait à l'aide de l'outil : TensorBoard

La courbe de performance du discriminateur avec les vrais échantillons est très intéressante. En effet, on peut constater qu'elle évolue entre les deux courbes obtenues lors des précédents entrainements (en rouge et bleu). Ce qui veut dire que le discriminateur ne va ni trop vite ni trop lentement dans sa progression d'apprentissage de classification des vrais échantillons. On pourrait supposer que cette modération

d'évolution permet de laisser le temps au générateur de progresser correctement afin de proposer un certain nombre de faux échantillons assez réalistes pour être confondus avec de vrais échantillons.

La courbe de performance du discriminateur avec les faux échantillons est également révélatrice de la position dominatrice du discriminateur sur le générateur. En effet, le discriminateur est rapidement en capacité à détecter une partie des faux échantillons, mais pas le 100% des faux échantillons. Ce qui laisse la possibilité au générateur de progresser et ainsi se conforter dans sa position de concurrent par rapport au discriminateur.

Voyons maintenant les faux échantillons générés par le générateur après 2500 époques. On peut constater que les faux échantillons ont atteint un niveau de réalisme tout à fait acceptable à vue d'œil :



Fait à l'aide de l'outil : TensorBoard

Les précédentes expériences ont montré l'importance des valeurs utilisées dans les taux d'apprentissage lors de l'entrainement d'un réseau GAN. Cependant, il est important de préciser que le taux d'apprentissage n'est pas le seul paramètre permettant le bon déroulement d'un apprentissage. On trouvera également la composition des réseaux de neurones, la fonction de perte, l'algorithme d'optimisation, la taille des lots, le nombre

d’époques, etc… L’impact de ces différents paramètres ne sera toutefois pas traité dans le cadre de cet annexe .