

# Projet - Génération de Mélodies

Rapport - 29/04/25

Hélène Barbillon - BARH30530200  
Amandine Lapique - LAPA07570200

<b>Introduction</b>	<b>2</b>
<b>1. Contexte</b>	<b>2</b>
a. Organisation du travail	2
b. Dataset	2
c. Outils	2
<b>2. Génération aléatoire</b>	<b>2</b>
<b>3. VAE</b>	<b>3</b>
a. Première approche	3
Implémentation	3
Traitement des données	3
Résultats	3
b. Implémentation finale	4
Traitement des données	5
Implémentation	5
Résultats	6
<b>4. LSTM</b>	<b>7</b>
a. Pré-traitement des données	7
b. Modèle	7
c. Compléter une mélodie	9
<b>5. GAN</b>	<b>9</b>
a. Gan simple	9
b. GAN convolutif	10
c. GAN complexe	11
<b>Conclusion</b>	<b>12</b>

# Introduction

Le but de ce projet est d'explorer différentes méthodes de génération de mélodies, en utilisant des méthodes d'apprentissage profond.

Nous allons dans un premier temps vous présenter le contexte de notre projet, puis nous présenterons les différents modèles que nous avons utilisés.

Nous avons regroupé les résultats de mélodies générées pour chaque partie dans le fichier `generated_melodies/Listen.ipynb`, si vous souhaitez écouter ce que nous avons produit.

## 1. Contexte

### a. Organisation du travail

Nous avons travaillé en groupe tous les dimanches après-midi (ou un autre jour si il y avait besoin de décaler).

### b. Dataset

Après plusieurs recherches, nous avons utilisé le dataset [ESAC](#), qui contient des musiques traditionnelles de différentes régions du monde. Les données sont des fichiers `.krn` (que l'on peut convertir en `.midi`), et sont des mélodies courtes, simples et mono instrument. Plus particulièrement, nous avons utilisé des musiques provenant de Chine (dossier `han`) et d'Allemagne (dossier `deutsch`) pour nos entraînements, étant donné que ces deux sous-dossiers comprenaient un nombre important de données (plus de 1500 mélodies chacun).

### c. Outils

Le projet est en python, et est principalement sous forme de notebooks jupyter. Nous avons utilisé la librairie `music21` pour le traitement des données musicales, ainsi que le logiciel `muscore` pour pouvoir les écouter et visualiser les partitions. Nous avons utilisé `pytorch` pour le VAE et le GAN, et `tensorflow` pour le LSTM.

Nous utilisons git pour le partage de code, voici le lien du dépôt :

<https://github.com/Hel88/MelodyGenerator>

## 2. Génération aléatoire

`Aleatoire/random_gen.ipynb`

Avant de se lancer dans des modèles complexes, et afin de nous familiariser avec les outils et les données, nous avons commencé par générer des musiques aléatoires. Nous avons essayé en sélectionnant des notes d'une gamme classique, mais également avec toutes les notes possibles pour voir ce qu'on pourrait obtenir. Les résultats sont disponibles dans le fichier `generated_melodies/Listen.ipynb`, et sont plutôt étranges. On constate bien

que la musique n'est pas vraiment une séquence de notes et de rythmes aléatoires, il y a une structure et des règles.

## 3. VAE

Un variational autoencoder est une architecture de réseau de neurones composée d'un encodeur et d'un decodeur. L'encodeur permet d'extraire une représentation latente probabiliste des données d'entrée et le décodeur tente de reconstruire les données d'origine à partir de cette représentation latente ce qui permet au modèle de générer de nouvelles données. Afin de nous familiariser avec cette architecture nous avons d'abord tenté une approche simple qui s'est soldée par un échec. Nous avons ensuite amélioré le modèle, notamment en transformant le traitement des données et nous avons obtenu des résultats plus intéressants.

### a. Première approche

AutoEncodeurs/VAE\_gen1.ipynb

#### Implémentation

Pour cette première approche, nous avons utilisé un modèle simple, inspiré d'un modèle d'autoencoder vanille pour mnist de Geeksforgeeks :

<https://www.geeksforgeeks.org/implementing-an-autoencoder-in-pytorch/>

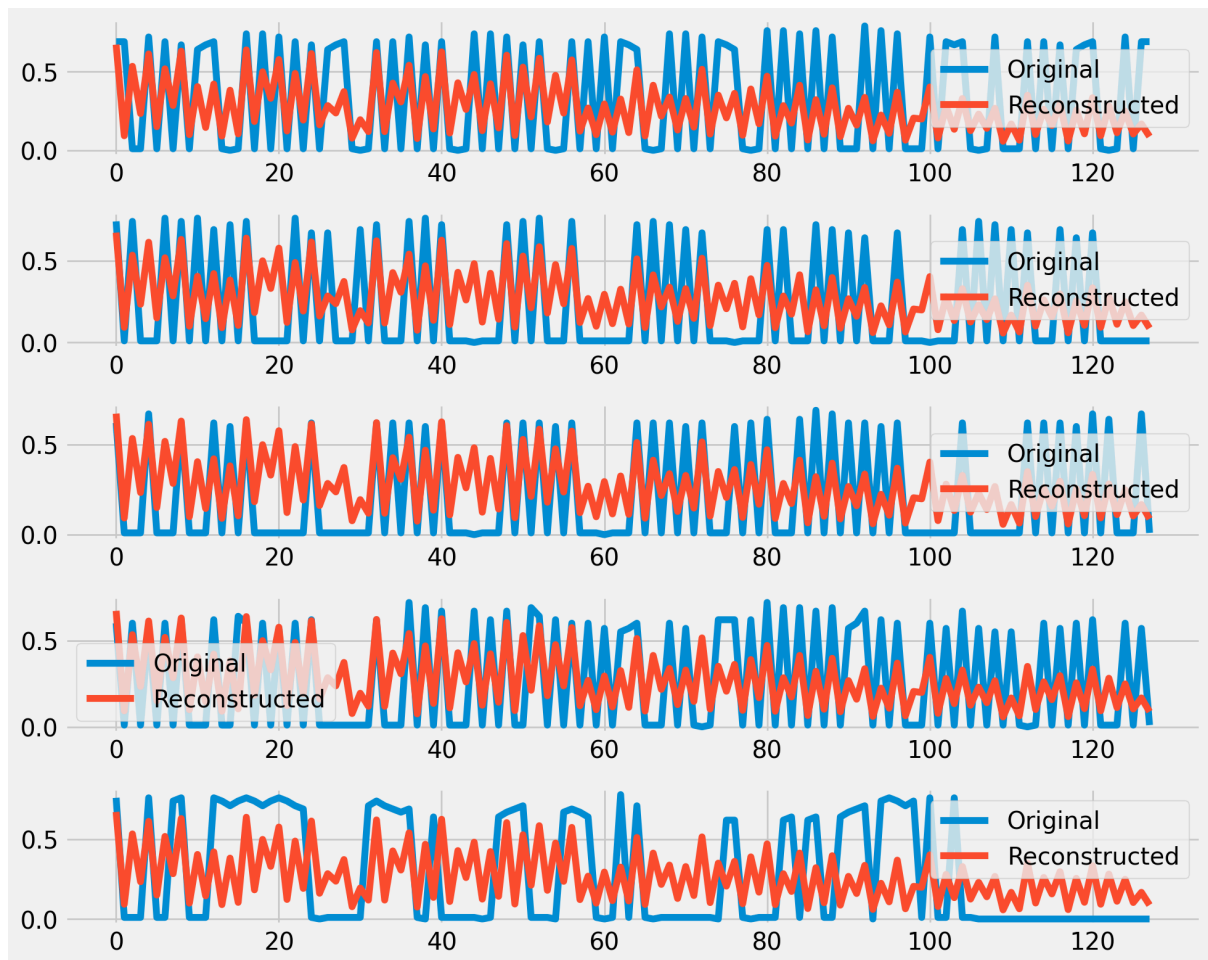
Nous n'avons pas cherché à améliorer le modèle car le but de cette première implémentation était de nous familiariser avec les autoencoder.

#### Traitement des données

Nous avons utilisé un pré-traitement des données basique qui consiste à transformer les notes en une liste de nombres. Si la note est tenue, la durée est représentée par un ensemble de 1. De plus, les silences sont représentés par des 0. La liste de nombres est ensuite normalisée et ramenée entre 0 et 1. Les modèles sont entraînés à partir de cette liste.

#### Résultats

Pour observer les résultats il est possible de comparer la séquence en entrée du VAE avec celle en sortie. Ainsi, les résultats de cette première approche sont les suivants :



Un exemple de mélodie obtenue est également disponible dans le dossier `generated_melodies`.

Les résultats obtenus ne sont pas très satisfaisants. On remarque que les mélodies reconstruites sont très différentes des mélodies originales et on observe également qu'elles sont très similaires entre elles. Cela provient certainement du traitement des données. Le fait de représenter les notes tenues par des 1 pose un problème lors du décodage. En effet, lors de la reconstruction, des notes très basses apparaissent. Cela est dû au fait que le modèle n'apprend pas, que le 0.1 est une valeur spécifique correspondant à une note tenue et donc beaucoup de notes basses mais non égales à 1 apparaissent. Pour tenter de régler ce problème, lors du décodage nous avons indiqué que toutes les notes en dessous de 0.5 étaient des 0.1. Cela est problématique car ça ne respecte pas vraiment la logique de l'encodage.

## b. Implémentation finale

`AutoEncodeurs/VAE_gen2.ipynb`

## Traitement des données

Afin d'améliorer le modèle nous avons d'abord changé le traitement des données. Nous avons choisi de représenter les données musicales sous la forme de deux séquences distinctes : l'une pour les notes et l'autre pour le rythme. Chaque séquence est ensuite paddée afin que toutes les séquences fassent la même taille. Puis elles sont normalisées avant d'être utilisées pour entraîner deux autoencodeurs séparés : un dédié à la modélisation des notes et un autre au rythme. Cette approche permet de contourner les problèmes rencontrés précédemment avec la représentation du rythme.


Initialement, nous avons envisagé de concaténer les deux séquences et d'entraîner un seul modèle. Toutefois, après réflexion, nous avons opté pour la création de deux modèles distincts, jugeant cette approche plus pertinente étant donné que le rythme et le choix des notes sont deux actions musicales distinctes.

## Implémentation

Nous avons amélioré notre modèle de VAE en nous inspirant du modèle de Jackson Kang disponible à l'adresse suivante :

[https://github.com/Jackson-Kang/Pytorch-VAE-tutorial/blob/master/01\\_Variational\\_AutoEncoder.ipynb](https://github.com/Jackson-Kang/Pytorch-VAE-tutorial/blob/master/01_Variational_AutoEncoder.ipynb)

Nous avons réalisé plusieurs entraînements avec divers hyperparamètres. Nous avons alors observé que les modèles obtenaient les meilleurs résultats avec des paramètres différents. L'image suivante présente le modèle ainsi que les hyperparamètres utilisés.



### MODÈLE - VAE

**Encodeur**


- 1e couche linéaire de `input_dim` → `hidden_dim`
- `LeakyReLU(0.2)`
- 2e couche linéaire de `hidden_dim` → `hidden_dim`
- `LeakyReLU(0.2)`
- Deux sorties parallèles :
  - `FC_mean` : moyenne de la distribution latente
  - `FC_var` : log-variance de la distribution latente

**Reparamétrisation**

$\epsilon \sim N(0, I)$  : bruit gaussien  
 $z = \text{mean} + \log\text{var} * \epsilon$  : vecteur latent échantillonné

**Décodeur**

- 1e couche linéaire de `latent_dim` → `hidden_dim`
- `LeakyReLU(0.2)`
- 2e couche linéaire de `hidden_dim` → `hidden_dim`
- `LeakyReLU(0.2)`
- `FC_output` : couche linéaire de `hidden_dim` → `output_dim`
- `Sigmoid` : activation finale (entre 0 et 1)




### NOTES

```
hidden_dim = 500
latent_dim = 250
epochs = 200
batch_size = 32
```

**Optimiseur Adam :**

- Taux d'apprentissage : `lr = 1e-5`
- Régularisation L2 : `weight_decay = 1e-6`



### RYTHMES

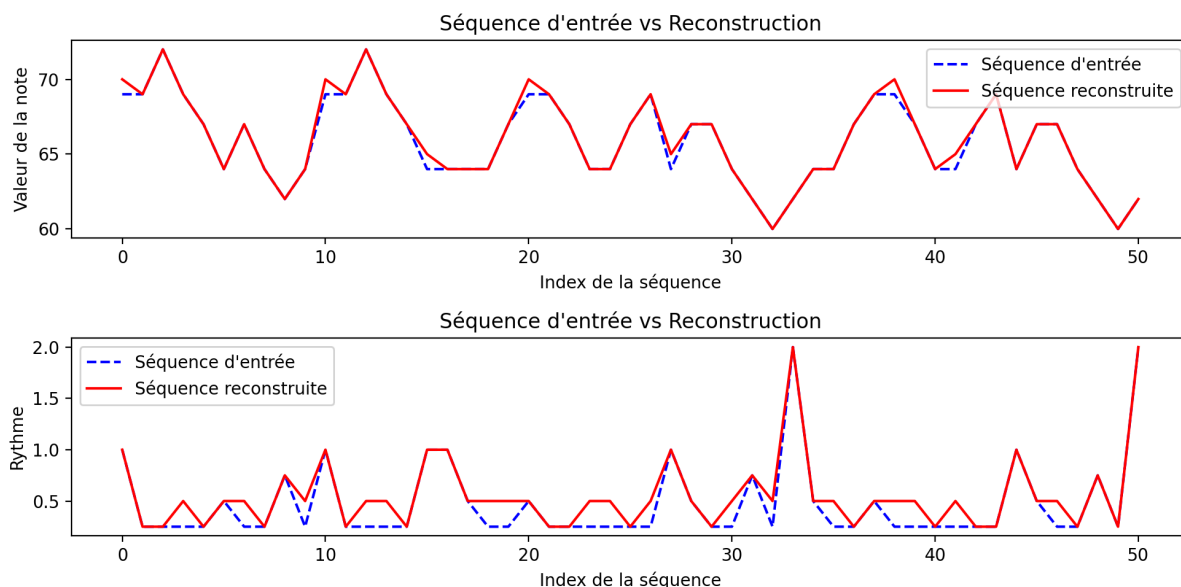
```
hidden_dim = 500
latent_dim = 300
epochs = 8
batch_size = 64
```

**Optimiseur Adam :**

- Taux d'apprentissage : `lr = 1e-3`

## Résultats

Les résultats obtenus avec ce modèle sont bien meilleurs que les précédents. L'image suivante montre le résultat obtenu sur une mélodie pour le rythme et les notes en comparant la mélodie initiale à la mélodie reconstruite.



Pour les notes, on observe de petites variations et pour le rythme des variations un peu plus importantes mais dans l'ensemble l'autoencodeur agit comme on l'attend. En observant les courbes on peut penser que les résultats sont satisfaisants. Malheureusement si on regarde attentivement la partition suivante qui en découle ou si on écoute le morceau on remarque des petites incohérences. Des exemples de mélodies reconstruites sont d'ailleurs disponibles dans le dossier `generated_melodies`.



Il est possible d'entendre parfois des notes qui sonnent faux. Cela est dû aux petites variations créées par le VAE. Ce n'est pas très agréable mais c'est inhérent au modèle. Il est difficile voire presque impossible d'améliorer la situation. En effet, le VAE cherche à reconstruire la mélodie à partir d'un espace latent probabiliste. Le but étant qu'il crée des variations pour simuler de la génération. Les petites variations vont avoir tendance à transformer les notes d'un ton ou d'un demi ton et cela va créer des fausses notes en fonction de la tonalité de la mélodie initiale. Malgré cela, les mélodies générées restent intéressantes. Il faut aussi garder en tête que pour faire de la génération on pourrait utiliser seulement le VAE entraîné sur les rythmes et mixer les résultats obtenus avec d'autres mélodies. Cela permettrait d'éviter les fausses notes tout en faisant de la génération.

## 4. LSTM

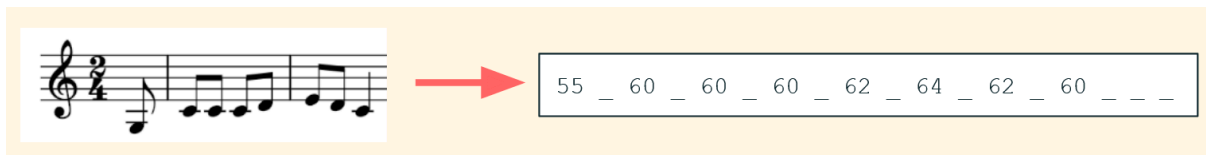
`LSTM/RNN-LSTM-han.ipynb`, `LSTM/RNN-LSTM-deutsch.ipynb`

Un LSTM (Long Short Term Memory network) est un réseau de neurones récurrent (RNN) capable de capturer des dépendances à long terme. Un RNN est adapté pour les données séquentielles, ce qui est le cas de nos données musicales. On espère ici que le modèle va apprendre des règles musicales afin de générer une mélodie agréable à écouter. Nous nous sommes inspirées du dépôt git et des vidéos youtube de Valerio Vedardo.

### a. Pré-traitement des données

`LSTM/preprocessing.py`

Comme pour le VAE, nous avons transposé les musiques en Do majeur, et supprimé les durées non acceptables, c'est-à-dire plus courtes qu'une double croche. Les musiques ont ensuite été encodées dans un format texte, où chaque note correspond à un nombre (ou la lettre "r" pour un silence), et où leur durée est exprimée en nombre de tirets du bas "\_".



Toutes les chansons encodées ont été mises dans un seul fichier texte, en étant séparées par des "/" (le nombre de "/" dépend de la taille des inputs). Un fichier de mapping permet de convertir ces symboles en nombres entiers. Les séquences d'input et d'output pour l'entraînement sont générées à partir de ces nombres.

Pour générer les inputs et outputs pour l'entraînement, on utilise une fenêtre de taille `SEQUENCE_LENGTH` (64 correspondant à 2 mesures à 4 temps, ou 4 mesures à 2 temps) qui correspond à un input, et la note suivant cette séquence sera l'output (la note que l'on cherche à prédire). Cette fenêtre est ensuite décalée d'une valeur et ainsi de suite sur tout le dataset.



Les séquences sont ensuite encodées en vecteur one-hot avant d'être données au modèle.

Les fonctions de pré-traitement sont dans le fichier `LSTM/preprocessing.py`. Pour chaque dataset (han et deutsch), le fichier texte contenant toutes les musiques du dataset s'appelle `file_dataset.txt`, et le fichier contenant les mappings `mapping.json`.

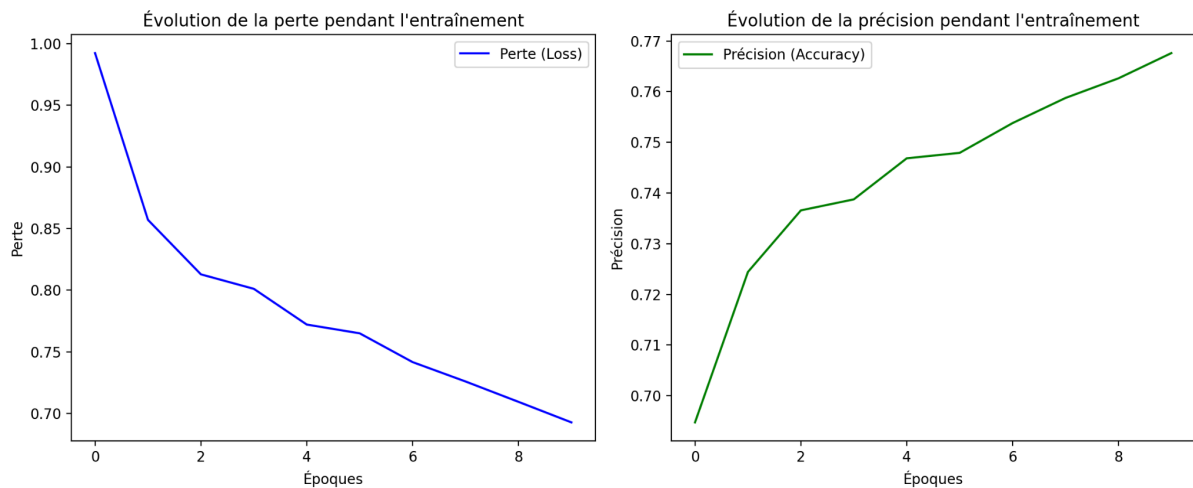
### b. Modèle

`LSTM/training.py`

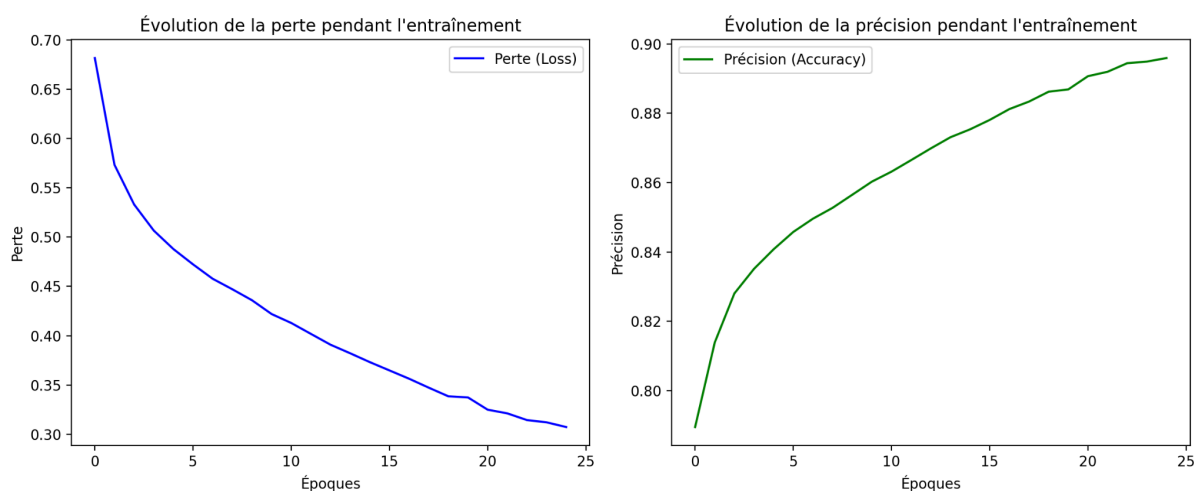
Le modèle est composé d'une couche d'inputs, d'une couche LSTM (256 unités cachées), d'une couche de dropout (20%), et d'une couche dense (activation softmax). Le modèle est optimisé avec Adam (learning\_rate = 0.001).

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, None, 38)	0
lstm_1 (LSTM)	(None, 256)	302,080
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 38)	9,766

Nous avons entraîné ce modèle sur 10 époques pour chaque dataset. Voici les courbes de loss pour le dataset han :



On voit que la perte diminue et que la précision augmente, mais qu'elle n'atteint pas de palier. Nous avons des résultats similaires pour le dataset deutsch et nous avons donc voulu pousser l'entraînement à 25 époques pour ce dataset :



Cet entraînement a pris plusieurs heures, on voit que les résultats se sont améliorés d'un point de vue de la perte et de la précision. Cependant en écoutant les musiques générées nous n'avons pas entendu de différences notables. Nous pensons qu'on peut pousser



l'entraînement plus loin, mais peut-être que nous risquons d'avoir un peu de surapprentissage. Nous nous sommes donc arrêtées à 25 époques.

Des musiques générées avec ce modèle sont visibles et écoutables dans `generated_melodies/Listen.ipynb`. Les résultats sont assez satisfaisants, les mélodies sont cohérentes, de durées différentes, et on entend la différence de style musical entre les deux datasets. En revanche, on entend très peu la différence entre le modèle entraîné sur 10 époques et celui sur 25 époques.

### c. Compléter une mélodie

`LSTM/complete_famous_melodies.ipynb`

Dans les générations que nous avons faites, on donne une note de départ (un sol), mais nous avons voulu compléter une mélodie connue que nous avons trouvé dans le dataset français : "ah vous dirais-je maman".

Les résultats sont différents à chaque fois, on en a sélectionné quelques-uns dans `generated_melodies/Listen.ipynb`, et on reconnaît encore bien la différence de style musical entre le dataset han et deutsch.

## 5. GAN

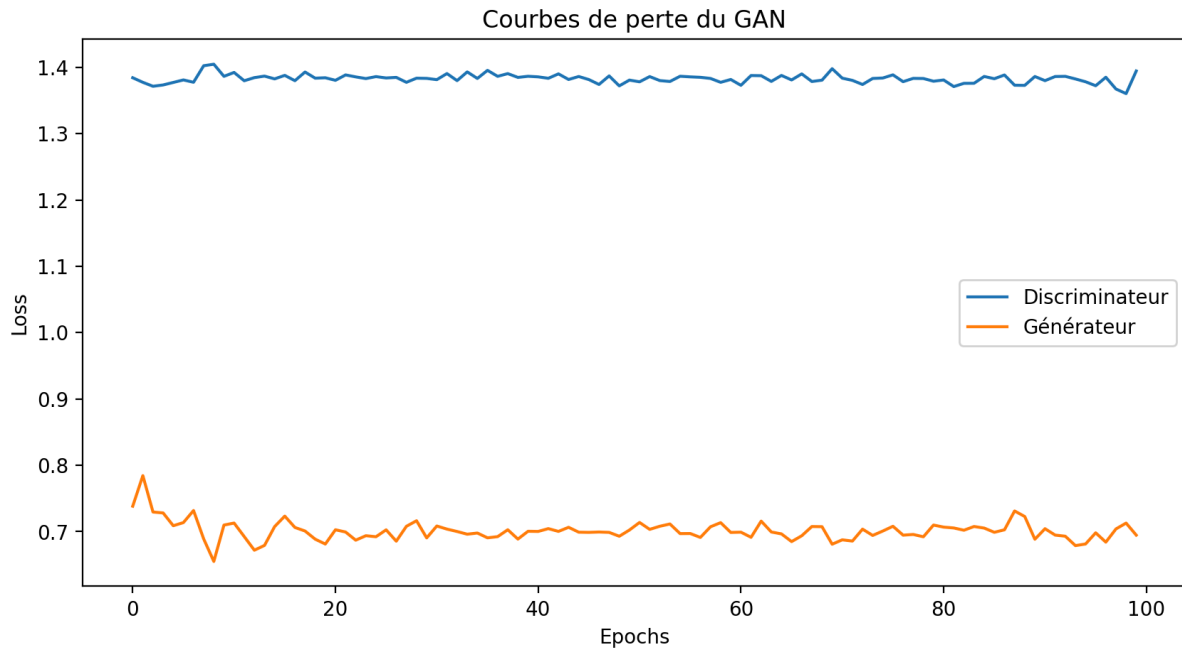
Un Generative Adversarial Network (GAN) est une architecture de réseau de neurones composée de deux parties principales : un générateur et un discriminateur. Le générateur tente de créer des données réalistes à partir d'un bruit aléatoire (ou d'une entrée aléatoire), tandis que le discriminateur cherche à distinguer les données générées des données réelles. L'objectif du générateur est de tromper le discriminateur, en produisant des données de plus en plus réalistes à chaque itération.

Pour entraîner nos modèles de GAN nous avons utilisé le pré-traitement des données que nous avons réalisé pour les autoencodeurs.

### a. Gan simple

`GAN/gan.ipynb`

Afin de nous familiariser avec cette architecture nous avons d'abord mis en place un modèle de GAN très simple composé uniquement de couches denses. Nous avons décidé d'entraîner le modèle avec deux dataset, le dataset han et le dataset deutsch. L'entraînement de ce modèle est stable comme on peut le voir sur la figure suivante :



On observe que le discriminateur est moins bon que le générateur sur ce modèle. Cela est problématique car ça signifie que le générateur ne sera pas suffisamment contraint pour produire des données réalistes, ce qui pourrait entraîner la génération de données de mauvaise qualité ou trop similaires à l'entrée aléatoire, réduisant ainsi l'efficacité globale du modèle.

Les résultats obtenus sont disponibles dans le dossier `generated_melodies`. Les résultats obtenus sont surprenants, ni particulièrement bons ni particulièrement mauvais. En revanche, lors de la génération des mélodies, on observe que le vecteur aléatoire d'entrée a un impact important sur les données générées. Cela est une bonne chose, car ça signifie que le modèle réussit à exploiter la variabilité de l'entrée aléatoire pour produire des sorties diversifiées et non redondantes, ce qui est un signe de flexibilité et de potentiel créatif du générateur.

## b. GAN convolutif

GAN/`gan2.ipynb`

Nous avons mis en place un modèle de GAN avec des couches convolutives afin d'observer si cela améliorerait la génération des données. Comme précédemment, les données générées sont difficiles à évaluer et ne semblent pas meilleures. Cela peut s'expliquer par le fait que les données musicales, étant des séquences temporelles complexes, nécessitent peut-être une approche plus spécifique que les convolutions classiques, qui sont généralement plus adaptées à des données spatiales comme les images. Il est possible que le modèle doive être ajusté pour mieux capturer les structures temporelles et rythmiques des mélodies.

### c. GAN complexe

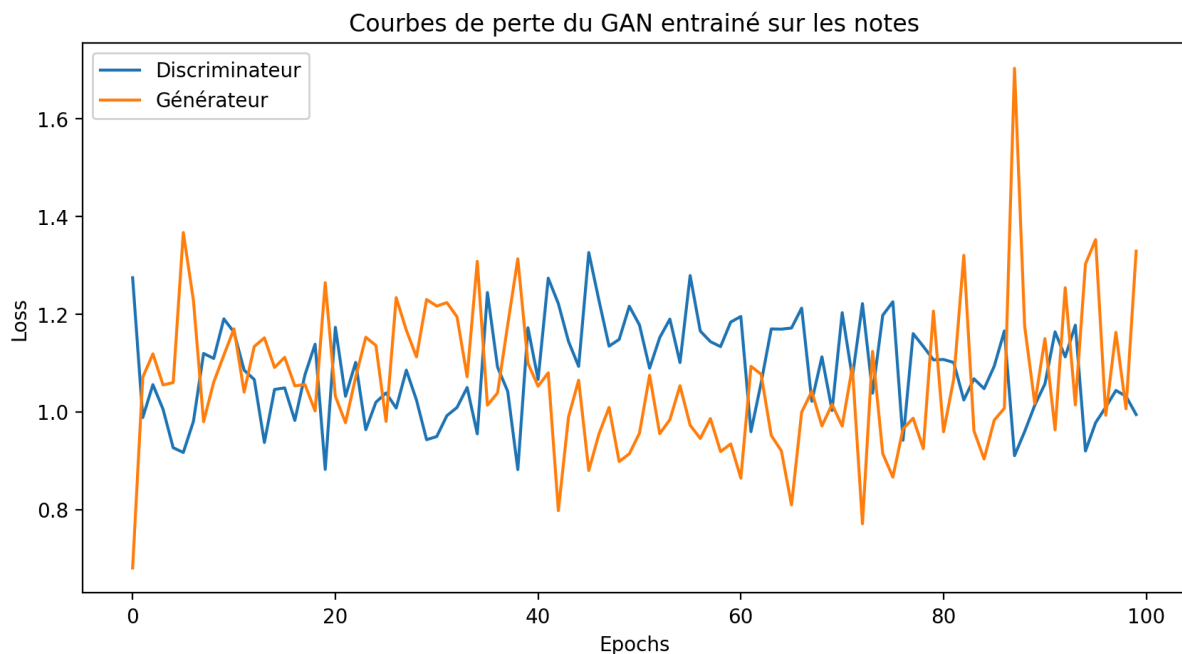
GAN/gan2.ipynb

Enfin, nous avons mis en place un modèle de GAN plus complexe dont l'implémentation est disponible ci-dessous.

```
# Générateur avec plus de couches et LeakyReLU
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_size, 128),
            nn.BatchNorm1d(128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, sequence_size),
            nn.Tanh()
        )
```

```
# Discriminateur avec Dropout et plus de couches
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(sequence_size, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.1),
            nn.Linear(256, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 64),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.4),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
```

Pour obtenir ce modèle, nous avons ajusté le nombre de couches, les hyperparamètres, ainsi que la stratégie d'entraînement (en entraînant le discriminateur deux fois pour chaque itération du générateur) afin de nous rapprocher d'un jeu à somme nulle. Après ces modifications, les courbes d'entraînement obtenues sont les suivantes.



On observe que les deux modèles semblent être en compétition, ce qui témoigne d'un véritable processus d'entraînement, où le générateur et le discriminateur améliorent progressivement leurs performances respectives.

Les résultats obtenus sont améliorés, bien qu'ils ne soient pas encore parfaits. Il est cependant difficile de quantifier ces résultats, car l'évaluation repose principalement sur l'écoute des mélodies générées. De plus, les données générées sont fortement dépendantes des vecteurs de bruit, et certains résultats peuvent être nettement meilleurs que d'autres.

L'implémentation de différents modèles de GAN nous a permis de mieux comprendre le fonctionnement de cette architecture et de modifier les paramètres pour obtenir des architectures proches d'un jeu à somme nulle. Cela a également mis en évidence la fragilité des GAN, car de petites modifications dans les paramètres peuvent entraîner une chute significative de la qualité des mélodies générées. À un moment, nous avons même rencontré un cas où le GAN ne produisait qu'une seule donnée à chaque itération.

Nous sommes satisfaites du modèle que nous avons obtenu à la suite de ces expérimentations, mais nous pensons qu'il est perfectible.

## Conclusion

Nous avons ainsi pu explorer différentes méthodes de génération de musique par apprentissage profond, et avons réussi à générer des musiques. Nos meilleurs résultats ont été obtenus avec le LSTM, qui a peut-être pu mieux capturer les dépendances à long terme que les autres. Nous notons qu'il est difficile d'évaluer la qualité de musiques générées au-delà d'une vague impression "ça sonne bien" ou "c'est bizarre". Il aurait été intéressant d'avoir des mesures pour évaluer la qualité d'une musique générée. On pourrait poursuivre ce projet avec un dataset plus complexe avec plusieurs instruments, ou en essayant d'implémenter un Transformer.