L'Université Chouaib Doukkali (UCD)
Ecole Nationale des Sciences Appliquées
El Jadida.

IA GENERATIVE ET INGENIERIE DES PROMPTS

Science De Données Et Intelligence Artificielle

Compte Rendu du TP1 RNN

Préparé par :
M.ARIRI Abdelaziz

Professeur :
Pr.Youness Abouqora

Année Académique 2025/2026

1 Introduction

Le présent TP a pour objectif de mettre en pratique les concepts théoriques étudiés sur les réseaux de neurones récurrents (RNN) afin de résoudre un problème concret : la génération de musique au format ABC. Ce travail permet de se familiariser avec la manipulation de données textuelles, la conception et l'entraînement de modèles RNN avec `PyTorch`, ainsi que la génération de nouvelles séquences musicales à partir d'un modèle entraîné.

Les objectifs principaux de ce TP sont les suivants :

- Prétraiter les données pour les rendre compatibles avec un modèle RNN.
- Créer un dataset `PyTorch` et gérer les séquences avec un `DataLoader`.
- Implémenter un modèle RNN avec une architecture LSTM pour la prédiction de notes.
- Configurer et exécuter une boucle d'entraînement avec suivi des métriques via `TensorBoard`.
- Appliquer des techniques avancées telles que l'*early stopping* et la sauvegarde du meilleur modèle.
- Générer de nouvelles séquences musicales à l'aide du modèle entraîné.
- (Bonus) Explorer des stratégies d'augmentation de données musicales.

Le TP se déroule en plusieurs étapes : le chargement et l'exploration des données, le prétraitement des séquences musicales, la création d'un dataset `PyTorch`, l'implémentation et l'entraînement du modèle LSTM, ainsi que la génération de musique.

Les données utilisées proviennent du répertoire `/irishman`, contenant les fichiers `train.json` et `validation.json` au format JSON. Chaque fichier comprend une liste d'objets représentant des partitions musicales en notation ABC, un format texte compact permettant de représenter les notes, leurs durées, les mesures et les métadonnées (titre, métrique, tonalité).

La notation ABC étant principalement utilisée pour la musique monodique traditionnelle (comme les mélodies irlandaises), chaque partition contient des lignes d'en-tête telles que `X:` (identifiant), `T:` (titre), `M:` (métrique) et `K:` (tonalité), suivies du corps de la mélodie. Le TP consiste notamment à vectoriser ces partitions pour les rendre exploitables par un RNN et à entraîner un modèle capable de prédire le prochain caractère d'une séquence, ce qui constitue la base de la génération musicale automatisée.

2 Prétraitement de données

2.1 Extraction des caractères uniques et vectorisation

La première étape du prétraitement consiste à transformer le corpus de partitions musicales, initialement au format texte (notation ABC), en une représentation numérique compréhensible par le modèle. Comme les réseaux de neurones manipulent exclusivement des tenseurs mathématiques, il est impératif de convertir chaque symbole textuel en un identifiant numérique unique.

Analyse du vocabulaire

En fusionnant l'intégralité des chaînes de caractères du jeu de données d'entraînement, nous avons identifié l'ensemble des symboles distincts utilisés (lettres, chiffres, symboles de ponctuation musicale, retours à la ligne). Cette analyse a révélé un vocabulaire composé de **95 caractères uniques**.

Justification de la numérisation

Le passage des caractères aux indices entiers est motivé par plusieurs besoins techniques :

- **Compatibilité logicielle :** Les couches de type `Embedding` de PyTorch, ainsi que les fonctions de perte telles que la `CrossEntropyLoss`, requièrent des indices entiers en entrée.
- **Efficacité computationnelle :** La représentation par indices permet de manipuler les données de manière compacte tout en facilitant l'apprentissage des relations statistiques entre les caractères.
- **Calcul matriciel :** Cette forme numérique permet d'effectuer les opérations d'algèbre linéaire nécessaires au fonctionnement interne des cellules LSTM.

Pour assurer la réversibilité du processus lors de la phase de génération, deux tables de correspondance (mappings) ont été créées : `char2idx` pour la phase d'encodage et `idx2char` pour la conversion des prédictions du modèle en texte lisible.

2.2 Création du mapping entre caractères et indices

Une fois le vocabulaire identifié, l'étape suivante consiste à établir une correspondance bidirectionnelle entre les caractères textuels et leur représentation numérique. Cette double indexation est indispensable pour assurer la fluidité entre l'entrée des données dans le modèle et l'interprétation des résultats en sortie.

Dictionnaire Caractère vers Index (`char_to_idx`)

Nous avons généré un dictionnaire de hachage associant chaque caractère unique à un entier allant de 0 à 94. Cette structure permet une transformation instantanée des séquences textuelles en vecteurs d'entiers. Cette conversion est nécessaire car :

- Les réseaux de neurones ne peuvent traiter que des données numériques (tenseurs).
- Les couches d'*Embedding* et les fonctions de perte (ex: `CrossEntropyLoss`) de PyTorch exigent des indices numériques en entrée pour effectuer les calculs de probabilités sur les classes.

Dictionnaire Index vers Caractère (`idx_to_char`)

En parallèle, une liste ordonnée (ou dictionnaire inverse) a été créée pour convertir les indices prédits par le modèle en caractères lisibles. Cette étape est cruciale lors de la phase de **génération musicale** : le modèle prédit l'indice du caractère suivant ayant la plus haute probabilité, et nous utilisons `idx_to_char` pour reconstruire la partition au format ABC.

Synthèse mathématique

Soit $\mathcal{V}$ notre vocabulaire de taille $N = 95$. Le mapping définit une fonction bijective $f : \mathcal{C} \to \mathcal{I}$ telle que :

$$\forall c \in \mathcal{V}, \quad f(c) = i \in \{0, 1, \ldots, 94\} \tag{1}$$

Cette représentation facilite non seulement les calculs matriciels mais permet également au modèle d'apprendre plus efficacement les relations statistiques entre les différents symboles musicaux.

2.3 Vectorisation des séquences musicales

La dernière étape du prétraitement avant la structuration en batchs consiste en la vectorisation effective des données. Cette opération transforme chaque partition musicale complète en une liste d'entiers, rendant le corpus prêt pour l'injection dans les couches d'*embedding* du modèle.

Mise en œuvre de la fonction de vectorisation

Nous avons implémenté une fonction utilitaire nommée `vectorize_string` qui parcourt chaque caractère d'une partition et retourne l'indice correspondant à l'aide du dictionnaire `char_to_idx`. Cette approche par compréhension de liste permet de traiter efficacement l'ensemble du texte :

- **Entrée :** Une chaîne de caractères brute (ex: `"X:1\nL:1/8..."`).
- **Sortie :** Un vecteur numérique (ex: `[56, 26, 17, 0, 44, ...]`).

Validation sur le jeu de données

Pour valider cette étape, un test a été réalisé sur la première partition du dataset d'entraînement. Comme illustré dans les résultats du TP, la structure de la partition (identifiants `X:`, `L:`, `M:` et les notes de musique) est fidèlement conservée sous forme de séquence d'indices.

Cette représentation numérique est essentielle car elle permet :

- Le calcul mathématique des gradients lors de la rétropropagation.
- L'apprentissage des relations temporelles entre les notes et les éléments de structure (mesures, armures).
- Une manipulation optimisée des données sous forme de tenseurs `PyTorch`.

2.4 Padding et troncature des séquences

Afin de permettre un entraînement efficace par mini-batchs sur GPU, il est nécessaire que toutes les séquences d'entrée au sein d'un même tenseur possèdent une longueur identique. Or, les partitions musicales au format ABC présentent des longueurs très variables selon la complexité du morceau.

Détermination de la longueur de référence

Nous avons d'abord analysé l'ensemble du jeu d'entraînement pour identifier la séquence la plus longue. Cette analyse a révélé une **longueur maximale de 2968 caractères**. Cette valeur sert de base pour l'uniformisation de l'ensemble du corpus.

Mise en œuvre du Padding

Pour égaliser les longueurs, nous avons implémenté la fonction `pad_or_truncate`. La stratégie adoptée est la suivante :

- **Le Padding :** Pour toute partition plus courte que la longueur maximale (par exemple, la première chanson qui ne compte que 183 caractères), des caractères d'espacement (`" "`) sont ajoutés à la fin de la séquence jusqu'à atteindre le seuil de 2968.
- **La Troncature :** Bien que théoriquement inutile ici puisque nous utilisons la longueur maximale observée, la fonction prévoit de couper les séquences qui dépasseraient cette limite pour garantir l'intégrité des dimensions des tenseurs.

Impact sur l'architecture

Cette étape est cruciale pour la construction du `DataLoader`. Elle garantit que chaque batch de données sera un tenseur de dimension constante $(Batch\_Size, Max\_Length)$, permettant ainsi au modèle LSTM de traiter les données de manière parallélisée tout en gérant les zones de "vide" (le padding) durant l'apprentissage.

3 Création du Dataset PyTorch

Après les étapes préliminaires de nettoyage, nous structurons les données sous forme d'objets `PyTorch` afin de faciliter l'alimentation du modèle durant l'entraînement. Cette phase repose sur une tokenisation plus fine et la gestion des dimensions de tenseurs.

3.1 Préparation des données et Tokenisation

Contrairement à l'approche par caractères simples, nous avons implémenté la fonction `prepare_data_tokens` qui utilise des expressions régulières (*regex*) pour extraire des entités musicales signifiantes.

- **Extraction sémantique :** Le motif `r"[A-Ga-gz][0-9]"` permet d'isoler les notes (avec leurs durées éventuelles) ainsi que les barres de mesure.
- **Gestion du vocabulaire :** Un dictionnaire `token_to_idx` est construit à partir de ces tokens uniques, complété par un jeton spécial `<PAD>` pour le remplissage (*padding*).

Pour permettre le calcul parallèle sur GPU via des batchs, toutes les séquences doivent avoir la même dimension.

- **Calcul de la longueur maximale :** Nous identifions la séquence de tokens la plus longue dans le corpus (`max_length`).
- **Application du jeton `<PAD>` :** Chaque séquence plus courte est complétée par le token de padding jusqu'à atteindre cette longueur maximale. Cela permet de conserver une structure de données rectangulaire au sein des tenseurs `PyTorch` sans altérer le contenu musical original.

Enfin, la fonction retourne les données sous forme de listes d'indices numériques (`vectorized_data`). Cette représentation est directement prête à être encapsulée dans une classe `Dataset` héritant de `torch.utils.data.Dataset`, où chaque échantillon sera découpé pour définir les couples entrées-cibles (x, y) nécessaires à l'apprentissage supervisé du RNN.

3.2 Dataset et Data Loader

Une fois les données vectorisées et normalisées par *padding*, nous les organisons selon une structure compatible avec l'entraînement de modèles `PyTorch` en utilisant les classes `Dataset` et `DataLoader`.

3.2.1 Implémentation de la classe `MusicDataset`

Nous avons conçu une classe personnalisée `MusicDataset` héritant de `torch.utils.data.Dataset`. Cette classe permet de structurer les données pour un apprentissage supervisé de type *next-token prediction* :

- `__len__` : Retourne le nombre total de séquences musicales présentes dans le jeu de données.
- `__getitem__` : Pour un indice donné, cette méthode extrait une séquence et génère le couple (x, y) nécessaire à l'entraînement :
  - **Entrée (x)** : La séquence amputée de son dernier élément (`seq[:-1]`), représentant le contexte actuel.
  - **Cible (y)** : La séquence décalée d'un rang vers la droite (`seq[1:]`), représentant les tokens que le modèle doit apprendre à prédire.

3.2.2 Configuration des `DataLoaders`

Pour optimiser l'entraînement, des objets `DataLoader` ont été instanciés pour les ensembles d'entraînement et de validation :

- **Taille des batchs** : Fixée à **8** pour équilibrer la vitesse de calcul et la stabilité de la convergence.
- **Mélange (*Shuffle*)** : Activé (`True`) pour l'entraînement afin de garantir que le modèle n'apprenne pas l'ordre des morceaux, favorisant ainsi une meilleure généralisation.

3.2.3 Validation des dimensions

La vérification d'un batch d'entraînement confirme la cohérence de notre pipeline. Comme illustré par les tests, un batch présente une dimension de `torch.Size([8, 1730])`. Le décalage systématique entre $X[0]$ et $y[0]$ (par exemple, le premier élément de y étant le deuxième de X) valide la préparation des données pour la tâche de génération séquentielle. Une fois les données vectorisées en tokens, l'étape suivante consiste à les organiser dans une structure compatible avec l'entraînement par mini-batchs de PyTorch.

4 Implémentation du modèle

4.1 Architecture du modèle MusicRNN

Pour la génération de musique au format ABC, nous avons conçu un modèle récurrent baptisé `MusicRNN`. Cette architecture est optimisée pour traiter des séquences temporelles et apprendre les dépendances entre les notes, les mesures et les structures harmoniques.

Le modèle repose sur une structure séquentielle composée de trois couches fondamentales, totalisant **1 744 307 paramètres** entièrement entraînables.

1. **Couche d'Embedding (28 500 paramètres) :** Cette couche projette chaque token du vocabulaire (95 caractères uniques) dans un espace vectoriel dense de dimension 300. Elle permet au modèle de capturer les relations sémantiques entre les symboles musicaux plutôt que de les traiter comme de simples entiers.
2. **Couche LSTM (1 667 072 paramètres) :** Il s'agit du composant central du réseau. Nous utilisons une cellule *Long Short-Term Memory* avec une dimension cachée (*hidden size*) de 512. Cette architecture permet de pallier le problème de disparition du gradient et de maintenir une mémoire à long terme, essentielle pour la cohérence d'une mélodie.
3. **Couche Linéaire de sortie (48 735 paramètres) :** Enfin, une couche dense effectue une projection de l'espace caché (512) vers l'espace du vocabulaire (95). Elle produit les *logits* qui seront utilisés pour calculer la probabilité du prochain token via la fonction de perte.

Résumé des dimensions et flux de données

Le tableau ci-dessous, extrait de l'analyse `torchsummary`, détaille le passage d'une séquence de test de longueur 50 à travers le réseau :

Couche	Type	Forme de sortie (Output Shape)	Paramètres
1-1	`Embedding`	$[1, 50, 300]$	28 500
1-2	`LSTM`	$[1, 50, 512]$	1 667 072
1-3	`Linear`	$[1, 50, 95]$	48 735
Total	—		**1 744 307**

Table 1: Détail des couches et des paramètres du modèle `MusicRNN`.

L'empreinte mémoire estimée pour le modèle et ses calculs est de **7,34 MB**, ce qui en fait un modèle léger et rapide à entraîner sur les infrastructures standard.

4.2 Boucle d'entraînement et stratégie d'optimisation

L'entraînement du modèle est orchestré par une fonction dédiée, `train_model`, qui gère l'itération sur les données, la mise à jour des poids et le suivi des performances. Cette étape est cruciale pour assurer la convergence du modèle vers une génération cohérente.

Configuration de l'optimisation

Pour l'apprentissage, les paramètres suivants ont été sélectionnés :

- **Fonction de perte :** Nous utilisons la `CrossEntropyLoss`. Comme les sorties du modèle (*logits*) sont de forme `(batch, seq_len, vocab_size)`, un *reshape* est appliqué pour aplatir les dimensions temporelles et les batchs afin de calculer la perte sur chaque prédiction de token individuellement.
- **Optimiseur :** L'algorithme `Adam` est utilisé avec un taux d'apprentissage (*learning rate*) de 5×10^{-3}. Ce choix se justifie par sa capacité à adapter le pas d'apprentissage pour chaque paramètre, accélérant ainsi la convergence.

Mécanismes de contrôle et de suivi

Afin de garantir un entraînement robuste, plusieurs techniques avancées ont été intégrées :

- **Early Stopping (Arrêt précoce) :** Le modèle surveille la perte sur le jeu de validation (*val_loss*). Si celle-ci ne s'améliore pas pendant une période de `patience` fixée à 10 époques, l'entraînement s'arrête préventivement. Cela permet d'éviter le *overfitting* (sur-apprentissage).
- **Sauvegarde du meilleur modèle :** Seuls les poids offrant la *loss* de validation la plus basse sont sauvegardés dans le fichier `best_model.pth`. En fin d'entraînement, le modèle est restauré dans cet état optimal.
- **Monitoring avec TensorBoard :** Les métriques de perte (*train* et *val*) sont enregistrées en temps réel via `SummaryWriter`, permettant une visualisation graphique de la courbe d'apprentissage.

Déroulement d'une itération

Pour chaque batch, le flux suit le cycle classique du *Deep Learning* : 1. Passage à zéro des gradients (`zero_grad`). 2. Propagation avant (*forward pass*) pour obtenir les prédictions. 3. Calcul de la perte et rétropropagation du gradient (`backward`). 4. Mise à jour des poids de l'architecture LSTM via l'optimiseur (`step`).

4.3 Configuration et exécution de l'entraînement

Une fois l'architecture définie, l'entraînement du modèle est réalisé via une procédure structurée incluant un échantillonnage des données et une gestion optimisée de la fonction de perte.

Échantillonnage et préparation des données

Compte tenu de la taille du corpus, nous avons opté pour un entraînement sur un sous-ensemble aléatoire (*subset*) afin d'accélérer les itérations expérimentales :

- **Ensemble d'entraînement :** 1 000 séquences choisies aléatoirement.
- **Ensemble de validation :** 250 séquences choisies aléatoirement.

Ces données sont encapsulées dans des `DataLoader` avec une taille de batch de 64 (définie dans la fonction de train) pour équilibrer la vitesse de calcul et la stabilité de la descente de gradient.

Définition des hyperparamètres

Le modèle `MusicRNN` est instancié avec les dimensions suivantes :

- **Taille du vocabulaire :** Basée sur le nombre de tokens uniques extraits précédemment.
- **Dimension d'embedding :** 128.
- **Dimension cachée (Hidden Size) :** 512.

Logique de perte et optimisation

La fonction d'entraînement `train_model` intègre des spécificités essentielles pour le traitement de séquences musicales :

- **Ignorance du Padding :** La fonction `CrossEntropyLoss` est configurée avec le paramètre `ignore_index=train_dataset.pad_idx`. Cela garantit que les jetons de remplissage (`<PAD>`) n'influent pas sur le calcul du gradient, forçant le modèle à se concentrer uniquement sur les éléments musicaux réels.
- **Optimiseur Adam :** Utilisé avec un *learning rate* de 5×10^{-3} pour une convergence efficace.
- **Early Stopping :** Un mécanisme de patience de 10 époques surveille la perte de validation pour interrompre l'apprentissage avant l'apparition du sur-apprentissage (*overfitting*), tout en sauvegardant l'état optimal dans `best_music_rnn.pth`.

Déroulement du cycle de calcul

À chaque itération, les *logits* prédits par le modèle sont redimensionnés de `[batch, seq_len, vocab_size]` vers une forme bidimensionnelle pour être comparés aux cibles. Ce processus permet de calculer l'erreur de prédiction pour chaque token de la séquence de manière parallélisée.

Résultats et suivi de l'entraînement

L'entraînement a été réalisé sur un cycle court de **10 époques**. Ce choix permet d'observer les premières phases de convergence du modèle tout en limitant le temps de calcul. Comme l'illustre la capture d'écran ci-dessous (Figure 1), la perte sur le jeu de validation montre une tendance à la baisse, signe que le modèle commence à capturer les régularités statistiques de la notation ABC.

Malgré un nombre d'époques réduit, le modèle parvient à :

- Diminuer de manière constante la fonction de perte (*Cross-Entropy*).
- Sauvegarder les meilleurs poids via le fichier `best_music_rnn.pth` pour la phase de génération.
- Apprendre la structure globale des fichiers (en-têtes et alternance notes/barres de mesure).

5 Génération de nouvelles séquences musicales

La phase finale du TP consiste à utiliser le modèle entraîné pour générer de nouvelles partitions au format ABC. Cette étape repose sur une approche de prédiction itérative caractère par caractère.

Figure 1: Logs de la console montrant l'évolution de la Val Loss sur les 10 époques.

5.1 Algorithme de génération et post-traitement

La fonction `generate_abc_music` implémente un processus de génération stochastique enrichi par des règles de filtrage pour assurer la cohérence musicale :

- **Échantillonnage avec température :** Nous utilisons une *température* de 0,7 appliquée aux *logits* avant la fonction `softmax`. Cela permet de balancer le déterminisme et la créativité du modèle. Un échantillonnage multinomial est ensuite effectué pour choisir le token suivant.
- **Filtrage syntaxique :** Pour garantir que la sortie soit exploitable, nous avons restreint les caractères générés aux notes valides (`C, D, E, F, G, A, B, z`). Tout caractère non reconnu est automatiquement converti en silence (`z`).
- **Contrôle de la structure :** Nous avons intégré une logique de comptage de notes. Après chaque groupe de 4 notes (représentant une mesure en 4/4), le script force l'insertion d'une barre de mesure (`|`). De plus, le nombre de barres consécutives est limité pour éviter les répétitions inutiles.

5.2 Résultats de la génération

Le modèle commence la génération à partir d'une séquence d'amorce (*seed*) contenant les métadonnées standards (`X:1, T:MySong, M:4/4, K:C`). Le résultat obtenu montre que le modèle a bien assimilé la structure répétitive de la notation ABC.

```
X:1 T:MySong M:4/4 K:C zzz|||||zz||||z|||||zz||||z||||z||||z||C||||Cz||||z|||||z
||||zz|||||zz||||z|||||zz||||C||||Czz|||||Cz||||z||||z||||z||||z|||||z|z|||z||||
|G||||Cz|||||z||||z||||C|||||C||||C||||z||||CzC||||zz||||z|||||z||
```

Figure 2: Exemple de partition ABC générée par le modèle LSTM après 10 époques.

Comme illustré dans la Figure 2, la séquence générée respecte les contraintes imposées par le post-traitement et présente une structure visuellement proche d'une partition traditionnelle, avec une alternance cohérente entre notes et barres de mesure.

6 Bonus : Augmentation de données musicales

Afin d'améliorer la capacité de généralisation du modèle et de pallier la taille limitée du dataset, nous avons exploré des techniques d'augmentation de données (*Data Augmentation*). L'objectif est de générer des variantes réalistes des partitions existantes par des transformations mathématiques simples.

6.1 Transposition tonale

La fonction `transpose_abc` permet de décaler chaque note de la partition d'un certain nombre d'intervalles (demi-tons théoriques).

- **Principe :** Pour chaque caractère appartenant à l'ensemble `CDEFGAB`, nous calculons son nouvel index dans la gamme par une opération de modulo : $new\_idx = (old\_idx + steps) \pmod 7$.
- **Intérêt :** Cela apprend au modèle que la structure mélodique (les intervalles entre les notes) est plus importante que la note absolue, enrichissant ainsi le vocabulaire dans différentes tonalités.

6.2 Modification du rythme (Time Stretching)

La fonction `change_rhythm` utilise des expressions régulières pour identifier les valeurs numériques associées aux durées des notes dans le format ABC.

- **Principe :** Chaque multiplicateur de durée trouvé est multiplié par un facteur k. Par exemple, une note `C2` devient `C4` avec un facteur 2.
- **Intérêt :** Cette technique permet de varier le tempo des morceaux originaux, aidant le LSTM à reconnaître des motifs rythmiques identiques même s'ils sont joués plus lentement ou plus rapidement.

6.3 Impact sur l'apprentissage

En combinant ces méthodes, la taille effective du dataset peut être multipliée par le nombre de transpositions effectuées. Cela réduit considérablement le risque d'*overfitting* et permet d'obtenir un modèle plus robuste, capable de générer des mélodies avec une structure rythmique plus variée.

7 Conclusion

Ce TP nous a permis de mettre en pratique l'intégralité d'une chaîne de traitement de données séquentielles appliquée à la génération de musique au format ABC. À travers la manipulation de réseaux de neurones récurrents (RNN) et plus particulièrement des cellules **LSTM**, nous avons pu aborder les défis liés à la modélisation du langage musical.

Les points essentiels retenus de ce travail sont les suivants :

- **Importance du prétraitement :** La tokenisation par expressions régulières et la vectorisation sont des étapes critiques. La gestion du *padding* et des dictionnaires de correspondance est indispensable pour transformer un texte brut en tenseurs exploitables par `PyTorch`.
- **Efficacité de l'architecture LSTM :** Malgré un entraînement court de seulement 10 époques, le modèle a démontré une capacité réelle à apprendre la syntaxe complexe du format ABC (entêtes, alternance notes/mesures), validant ainsi le choix des LSTM pour capturer des dépendances temporelles.
- **Nécessité du post-traitement :** La phase de génération a montré que l'utilisation d'une température d'échantillonnage et de règles de filtrage (correction des caractères, gestion automatique des barres de mesure) est cruciale pour garantir la validité musicale de la sortie.
- **Robustesse via l'augmentation :** Les techniques de transposition et de modification rythmique explorées en bonus constituent des pistes sérieuses pour améliorer la diversité du modèle sans nécessiter de nouvelles données sources.

En conclusion, bien que les mélodies générées puissent encore être affinées par un entraînement plus long sur un GPU performant, ce projet démontre la puissance des modèles récurrents pour la créativité artificielle. Ce TP constitue une base solide pour explorer des architectures plus complexes comme les *Transformers* ou les mécanismes d'attention appliqués à la musique.