Université Chouaib Doukkali
École Nationale des Sciences Appliquées_El Jadida
Master Science Des Données Et Intelligence Artificielle
Module: Gen IA

Rapport de travaux demandes:

Rapport de TP :
Génération de Musique ABC avec RNN

Réaliser par : Nada saber

Encadrer par : Youness Abouqora

Introduction

Le but de ce travail pratique est de mettre en pratique les concepts théoriques étudiés sur les réseaux de neurones récurrents (RNN) afin de résoudre un problème concret : la génération automatique de musique au format ABC. Ce format texte compact permet de représenter des partitions, principalement des mélodies monodiques issues du répertoire traditionnel irlandais, en encodant les notes, les durées et les mesures avec des caractères ASCII.

Tout au long de ce TP on va :

- Manipuler et prétraiter des données textuelles pour les rendre compatibles avec un modèle d'apprentissage profond.
- Concevoir et implémenter une architecture RNN basée sur des cellules LSTM (Long Short-Term Memory) à l'aide de la bibliothèque PyTorch.
- Gérer l'entraînement du modèle en utilisant des outils comme un DataLoader et en effectuant un suivi des performances (logging) via TensorBoard.
- Utiliser le modèle entraîné pour générer de nouvelles séquences musicales originales en appliquant des stratégies d'échantillonnage.

Cette approche permettra au modèle de comprendre les relations temporelles entre les éléments successifs d'une partition pour prédire de façon itérative le caractère suivant d'une mélodie.

1.Configuration et Environnement

Bibliothèques clés : PyTorch pour le deep learning, datasets (Hugging Face) pour le chargement des données, et matplotlib pour la visualisation.
Optimisation : Pour garantir un temps d'entraînement raisonnable , le dataset a été réduit à 50 000 chansons et la longueur des séquences limitée à 300 caractères. L'environnement de travail est optimisé pour Kaggle avec l'utilisation d'un GPU Tesla P100.

```
!pip install -q datasets tensorboard

import json
import os
import time
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from tqdm.auto import tqdm

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.utils.tensorboard import SummaryWriter

# Configuration
plt.style.use('seaborn-v0_8-darkgrid')

# Vérifier le GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"🖥️  Device: {device}")
if torch.cuda.is_available():
    print(f"   GPU: {torch.cuda.get_device_name(0)}")
    print(f"   Mémoire disponible: {torch.cuda.get_device_properties(0).total_memory
```

a.Bibliothèques de gestion des données et système

json : Utilisée pour charger les fichiers train.json et validation.json qui contiennent les partitions en notation ABC.
os : Permet de manipuler les chemins de fichiers et de créer des répertoires (comme le dossier /irishman).
time : Utile pour mesurer le temps d'exécution pendant l'entraînement du modèle.
datasets : Provenant de Hugging Face, cette bibliothèque permet de télécharger et de manipuler facilement le dataset "IrishMAN" utilisé pour le répertoire traditionnel irlandais.

b. Calcul numérique et Statistiques

numpy (np) : Bibliothèque fondamentale pour le calcul scientifique, utilisée ici pour manipuler les vecteurs numériques et les tableaux de données.
collections.Counter : Utilisée pour identifier et compter la fréquence des caractères uniques présents dans le dataset.

c.Visualisation
matplotlib.pyplot (plt) : Utilisée pour explorer les données et tracer les courbes de perte (loss) et de précision (accuracy).

d. Cœur du Deep Learning : PyTorch
torch : Le framework principal utilisé pour manipuler les tenseurs et configurer le calcul sur GPU (via CUDA).
t**orch.nn (nn) :** Contient les modules nécessaires pour implémenter la classe MusicRNN, incluant les couches d'Embedding, le LSTM et les couches denses.
torch.optim : Fournit les algorithmes d'optimisation, comme Adam, pour mettre à jour les poids du modèle avec un taux d'apprentissage spécifique
torch.nn.functional (F) : Fournit des fonctions comme softmax utilisées lors de l'étape de génération de musique par échantillonnage des probabilités.

e.Gestion des Séquences et Utilitaires
Dataset : Permet de créer la classe MusicDataset pour gérer les couples de séquences d'entrée et de cible (décalage d'un pas).
DataLoader : Gère le regroupement des données en lots (batches) de taille 256 pour l'entraînement.
TensorBoard: Utilisée pour enregistrer (logger) les performances du modèle en temps réel pour une visualisation ultérieure.
tqdm : Affiche des barres de progression interactives pour suivre l'avancement de la boucle d'entraînement.

2.Chargement et exploration des données

```python
# Télécharger le dataset IrishMAN depuis Hugging Face
from datasets import load_dataset

print("📥 Téléchargement du dataset IrishMAN...")
ds = load_dataset("sander-wood/irishman")

# Convertir en format JSON
os.makedirs("irishman", exist_ok=True)

with open("irishman/train.json", "w", encoding="utf-8") as f:
    json.dump(list(ds["train"]), f, ensure_ascii=False)

with open("irishman/validation.json", "w", encoding="utf-8") as f:
    json.dump(list(ds["validation"]), f, ensure_ascii=False)

print("✅ Dataset téléchargé et sauvegardé!")
```

```
📥 Téléchargement du dataset IrishMAN...
README.md: 6.27k/? [00:00<00:00, 542kB/s]
train.json: 100% 80.0M/80.0M [00:01<00:00, 56.6MB/s]
validation.json: 797k/? [00:00<00:00, 27.6MB/s]
Generating train split: 100% 214122/214122 [00:01<00:00, 136587.07 examples/s]
Generating validation split: 100% 2162/2162 [00:00<00:00, 93265.23 examples/s]
✅ Dataset téléchargé et sauvegardé!
```

Dans cette phase, le code automatise la récupération et la préparation locale du jeu de données pour l'entraînement.
Téléchargement du Dataset : Le code utilise la bibliothèque datasets pour charger IrishMAN. Il s'agit d'un vaste répertoire de mélodies irlandaises traditionnelles codées au format ABC.
Conversion au format JSON : Une fois téléchargé, le jeu de données est converti et sauvegardé localement en fichiers train.json (pour l'entraînement) et validation.json (pour l'évaluation). Cette étape est importante car elle permet de manipuler les données sous forme d'objets structurés contenant la partition dans la clé abc notation.

```python
# Charger les données
with open("irishman/train.json", "r", encoding="utf-8") as f:
    train_data = json.load(f)

with open("irishman/validation.json", "r", encoding="utf-8") as f:
    validation_data = json.load(f)

print(f"📊 Nombre de chansons:")
print(f"   Train: {len(train_data):,}")
print(f"   Validation: {len(validation_data):,}")
print(f"   Total: {len(train_data) + len(validation_data):,}")
```

```
📊 Nombre de chansons:
   Train: 214,122
   Validation: 2,162
   Total: 216,284
```

Statistiques de téléchargement : le processus de téléchargement et de génération des fichiers est rapide et transparent :
Le fichier d'entraînement (train.json) contient **214 122** exemples.
Le fichier de validation (validation.json) contient **2 162** exemples

```python
# Afficher la première chanson
first_song = train_data[0]
abc_key = [k for k in first_song.keys() if 'abc' in k.lower() or 'notation' in k.lower()][0]
abc_notation = first_song[abc_key]

print("🎵 Première chanson du dataset:")
print("="*80)
print(abc_notation)
print("="*80)
print(f"\n💡 Copiez cette notation et testez-la sur: https://www.abcjs.net/abcjs-editor.html")
```

```
🎵 Première chanson du dataset:
================================================================================
X:1
L:1/8
M:4/4
K:Emin
|: E2 EF E2 EF | DEFG AFDF | E2 EF E2 B2 |1 efe^d e2 e2 :|2 efe^d e3 B |: e2 ef g2 fe |
 defg afdf |1 e2 ef g2 fe | efe^d e3 B :|2 g2 bg f2 af | efe^d e2 e2 ||
================================================================================

💡 Copiez cette notation et testez-la sur: https://www.abcjs.net/abcjs-editor.html
```

L’affichage de la première chanson du fichier d'entraînement permet d'observer la structure typique de ce format :

- Des lignes d'en-tête comme **X:1** (identifiant), **L:1/8** (durée des notes), **M:4/4** (métrique) et **K:Emin** (tonalité).
- Le corps de la mélodie utilisant des lettres pour les notes et des symboles pour les mesures

3.Prétraitement des Données

Étape 1 : Extraction des caractères uniques

[6]:
```
# Étape 1: Extraction des caractères uniques
all_text = ''.join(all_abc)
unique_chars = sorted(list(set(all_text)))

print(f"🔤 Vocabulaire:")
print(f"   Nombre de caractères uniques: {len(unique_chars)}")
print(f"   Aperçu: {unique_chars[:30]}")

# Top caractères
char_counts = Counter(all_text)
print(f"\n📊 Top 10 caractères les plus fréquents:")
for i, (char, count) in enumerate(char_counts.most_common(10), 1):
    display = '\\n' if char == '\n' else ('SPC' if char == ' ' else char)
    print(f"   {i:2d}. '{display}': {count:,} occurrences")
```

```
🔤 Vocabulaire:
   Nombre de caractères uniques: 95
   Aperçu: ['\n', ' ', '!', '"', '#', '$', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=']

📊 Top 10 caractères les plus fréquents:
    1. 'SPC': 15,733,075 occurrences
    2. '|': 4,280,290 occurrences
    3. '2': 3,387,929 occurrences
    4. 'A': 3,003,845 occurrences
    5. 'd': 2,850,960 occurrences
    6. 'B': 2,749,618 occurrences
    7. 'c': 2,227,919 occurrences
    8. 'e': 2,219,826 occurrences
    9. 'G': 2,205,273 occurrences
   10. '/': 1,802,824 occurrences
```

all_text = ''.join(all_abc) : Fusionne l'intégralité des partitions du dataset en une seule chaîne de caractères géante.
set(all_text) : Identifie chaque caractère unique (chaque symbole distinct utilisé dans la notation ABC) en éliminant les doublons.
l**en(unique_chars) :** Calcule la taille totale du vocabulaire. Dans notre cas, il y a 95 caractères uniques. C'est ce nombre qui déterminera la dimension de la couche de sortie de modèle LSTM.

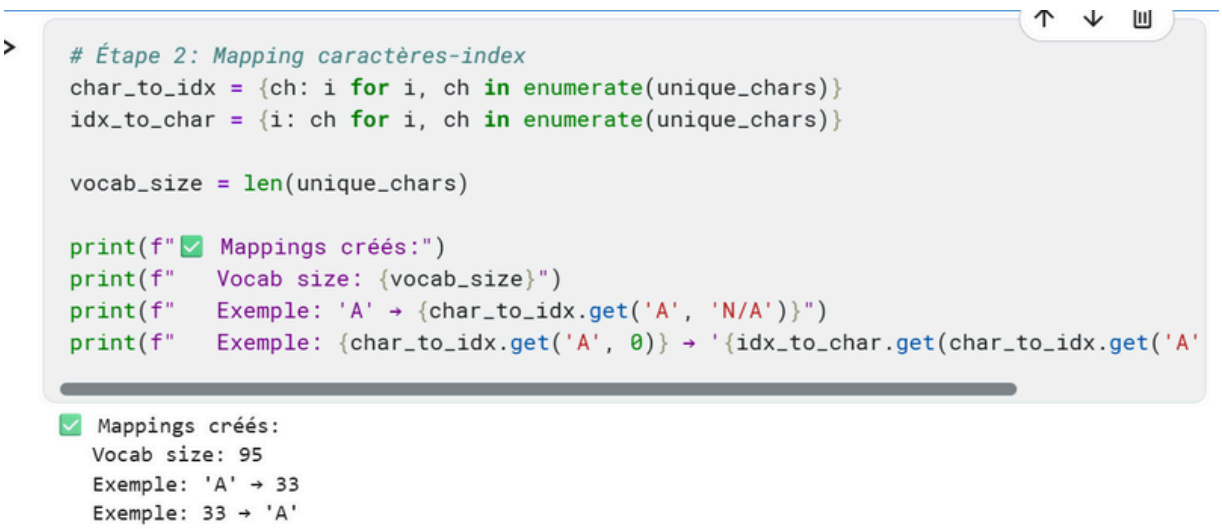
Classement des caractères par fréquence d'apparition. Cela permet de comprendre la "grammaire" de la musique irlandaise traitée :
'SPC' (Espace) : C'est le caractère le plus fréquent (plus de 15,7 millions d'occurrences), ce qui montre que les notes sont groupées par blocs espacés.
'**|**' (Barre de mesure) : Arrive en deuxième position (4,2 millions d'occurrences), confirmant une structure rythmique très régulière.

'**2'** : Indique que la durée de note multipliée par 2 est la valeur rythmique la plus courante dans ce répertoire.
Notes (A, d, B, c, e, G) : Le code distingue les majuscules des minuscules car en notation ABC, elles représentent des octaves différentes.

Étape 2 : Mapping caractères-index

Cette étapes est cruciale en terme:

- **Calculs mathématiques** : Les réseaux de neurones sont des moteurs mathématiques. Ils ne savent pas multiplier ou additionner des lettres comme "A" ou des symboles comme "|". Ils ne manipulent que des tenseurs (des tableaux de nombres).
- **Espace vectoriel** : Transformer les caractères en indices est la première étape pour ensuite les projeter dans un espace d'embeddings (vecteurs continus), où le modèle pourra apprendre que la note "A" est musicalement proche de "B".



```
# Étape 2: Mapping caractères-index
char_to_idx = {ch: i for i, ch in enumerate(unique_chars)}
idx_to_char = {i: ch for i, ch in enumerate(unique_chars)}

vocab_size = len(unique_chars)

print(f"✅ Mappings créés:")
print(f"   Vocab size: {vocab_size}")
print(f"   Exemple: 'A' → {char_to_idx.get('A', 'N/A')}")
print(f"   Exemple: {char_to_idx.get('A', 0)} → '{idx_to_char.get(char_to_idx.get('A'
```

```
✅ Mappings créés:
   Vocab size: 95
   Exemple: 'A' → 33
   Exemple: 33 → 'A'
```

Fonctionnement:

- **char_to_idx** (Dictionnaire Caractère -> Index) : Il associe chaque caractère unique du vocabulaire à un numéro unique (0, 1, 2, etc.). C'est la "table de traduction" vers le langage machine.
- **idx_to_char** (Dictionnaire Index -> Caractère) : Il fait l'inverse. C'est ce qui permettra au modèle, une fois qu'il aura prédit un nombre, de le retraduire en une note de musique ou un symbole lisible.
- **vocab_size** : Stocke le nombre total de caractères uniques (95 dans ce cas). Ce chiffre définit la largeur de la "porte de sortie" du futur réseau de neurones.

Étape 3 : Vectorisation

L'étape de **vectorisation** transforme les partitions musicales en séquences d'états numériques discrets. Cette transformation est le pont indispensable entre la notation symbolique ABC et les calculs matriciels du réseau **LSTM**. Le succès du test de vectorisation confirme que l'intégralité de la structure musicale est préservée lors du passage au format numérique, permettant un entraînement fiable sur les motifs mélodiques

```python
# Étape 3: Vectorisation
def vectorize_string(text, char_to_idx):
    """Convertit une chaîne en liste d'indices."""
    return [char_to_idx[ch] for ch in text]

def devectorize_string(indices, idx_to_char):
    """Convertit une liste d'indices en chaîne."""
    return ''.join([idx_to_char[idx] for idx in indices])

# Test
test_text = "ABC"
vectorized = vectorize_string(test_text, char_to_idx)
devectorized = devectorize_string(vectorized, idx_to_char)

print(f"✅ Test vectorisation:")
print(f"   Original: '{test_text}'")
print(f"   Vectorisé: {vectorized}")
print(f"   Dévectorisé: '{devectorized}'")
print(f"   Match: {test_text == devectorized}")
```

```
✅ Test vectorisation:
   Original: 'ABC'
   Vectorisé: [33, 34, 35]
   Dévectorisé: 'ABC'
   Match: True
```

- **vectorize_string :** Cette fonction parcourt chaque caractère d'une partition et utilise le dictionnaire **char_to_idx** pour le remplacer par son index numérique correspondant. le texte "ABC" devient la liste d'entiers [33, 34, 35].

- **devectorize_string :** C'est l'opération inverse. Elle reprend une liste de nombres prédits par le modèle et utilise **idx_to_char** pour reconstruire la partition textuelle lisible.

Étape 4 :Padding

Dans un LSTM, les calculs s'effectuent par lots (batches) de données. Pour que ces lots puissent être transformés en tenseurs (matrices), toutes les séquences à l'intérieur d'un même lot doivent impérativement avoir la même longueur. Comme les chansons irlandaises ont des longueurs très variables, il faut les harmoniser.

```python
def pad_or_truncate(text, max_length, pad_char=' '):
    """Ajoute du padding ou tronque une chaîne."""
    if len(text) >= max_length:
        return text[:max_length]
    else:
        return text + pad_char * (max_length - len(text))

# Choisir une longueur maximale raisonnable
MAX_LENGTH = 300  # Compromis entre qualité et vitesse

print(f"📏 Longueur maximale choisie: {MAX_LENGTH} caractères")
print(f"   (Cela couvre ~{sum(1 for l in lengths if l <= MAX_LENGTH) / len(lengths) *
```

```
📏 Longueur maximale choisie: 300 caractères
   (Cela couvre ~66.8% des séquences)
```

La fonction gère deux cas de figure pour atteindre la cible de **MAX_LENGTH** :

- **Truncate** : Si une chanson est plus longue que 300 caractères, le code ne garde que les 300 premiers.
- **Padding** : Si une chanson est plus courte, le code ajoute des caractères "vides" (ici des espaces ' ') à la fin jusqu'à atteindre exactement 300 caractères.

Analyse du choix : Le choix d'une fenêtre de 300 caractères représente un point d'équilibre optimal : il permet de traiter des lots de données de manière efficace sur le GPU tout en préservant l'intégrité mélodique de la majorité du corpus irlandais

```python
# Préparer toutes les données
print("⚙️  Préparation des données...")
SUBSET_SIZE = 50000
all_abc_subset = all_abc[:SUBSET_SIZE]
print(f"⚡ Mode optimisé: Utilisation de {SUBSET_SIZE:,} chansons (au lieu de {len(all_abc):,})")

# Padding et vectorisation
train_padded = [pad_or_truncate(text, MAX_LENGTH) for text in tqdm(all_abc_subset, desc="Padding train")]
val_abc = [song[abc_key] for song in validation_data]
val_padded = [pad_or_truncate(text, MAX_LENGTH) for text in tqdm(val_abc, desc="Padding val")]

train_vectorized = [vectorize_string(text, char_to_idx) for text in tqdm(train_padded, desc="Vectorizing t
val_vectorized = [vectorize_string(text, char_to_idx) for text in tqdm(val_padded, desc="Vectorizing val")

print(f"\n✅ Données préparées:")
print(f"   Train: {len(train_vectorized)} séquences")
print(f"   Val: {len(val_vectorized)} séquences")
print(f"   Longueur: {MAX_LENGTH} caractères")
```

```
⚙️  Préparation des données...
⚡ Mode optimisé: Utilisation de 50,000 chansons (au lieu de 214,122)
Padding train: 100% 50000/50000 [00:00<00:00, 1071265.40it/s]
Padding val: 100% 2162/2162 [00:00<00:00, 199944.55it/s]
Vectorizing train: 100% 50000/50000 [00:00<00:00, 63247.80it/s]
Vectorizing val: 100% 2162/2162 [00:00<00:00, 49545.34it/s]

✅ Données préparées:
   Train: 50000 séquences
   Val: 2162 séquences
   Longueur: 300 caractères
```

Optimisation du Dataset

Le code fait un choix stratégique en limitant le jeu d'entraînement à 50 000 chansons (au lieu des ~214 000 disponibles).

- **Justification** : Cette réduction permet d'accélérer considérablement le temps d'entraînement tout en conservant une masse de données largement suffisante pour que le LSTM apprenne les structures mélodiques irlandaises.

Pipeline de transformation

Pour les ensembles d'entraînement (train) et de validation (val):

- **Padding / Troncature :** Uniformisation de toutes les séquences à une longueur fixe de 300 caractères.
- **Vectorisation :** Conversion des caractères textuels en listes d'indices numériques exploitables par PyTorch via les mappings créés précédemment.

4. Création du Dataset PyTorch

Étape 1 : Préparation des données

Dans un problème de génération de musique, le modèle doit apprendre à prédire le prochain caractère en se basant sur le contexte précédent. Pour ce faire, la classe MusicDataset divise chaque partition vectorisée en deux parties décalées

- **Séquence d'entrée (Input) :** Contient tous les caractères de la partition sauf le dernier (sequence[:-1]). C'est le contexte fourni au modèle.
- **Séquence cible (Target) :** Contient tous les caractères de la partition sauf le premier (sequence[1:]).

Ce décalage d'un pas vers la gauche signifie que pour chaque position t dans l'entrée, la valeur correspondante dans la cible est le caractère à la position t+1

```python
class MusicDataset(Dataset):

    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sequence = self.data[idx]
        input_seq = sequence[:-1]
        target_seq = sequence[1:]

        return torch.tensor(input_seq, dtype=torch.long), torch.tensor(target_seq, dtype=torch.long)

# Créer les datasets
train_dataset = MusicDataset(train_vectorized)
val_dataset = MusicDataset(val_vectorized)

print(f"✅ Datasets créés:")
print(f"   Train: {len(train_dataset)} séquences")
print(f"   Val: {len(val_dataset)} séquences")
```

```
✅ Datasets créés:
   Train: 50000 séquences
   Val: 2162 séquences
```

Fonctionnement de la classe **MusicDataset:**

- **__init__ :** Reçoit les données musicales déjà vectorisées et paddées.
- **__len__ :** Retourne le nombre total de chansons disponibles dans le set (Train ou Val).
- **__getitem__** : Récupère une chanson spécifique par son index, effectue le décalage entrée/cible, et convertit le tout en tenseurs PyTorch (torch.tensor) de type entier long.

Le code initialise deux instances de cette classe :

- **train_dataset** : 50 000 séquences destinées à l'entraînement.
- **val_dataset** : 2 162 séquences destinées à l'évaluation des performances sur des données non vues.

Étape 2 : Dataset et DataLoader

DataLoader permet de transformer le MusicDataset (une liste de séquences individuelles) en un flux itérable capable de gérer le chargement en parallèle et le regroupement en lots. Il automatise plusieurs tâches critiques :

- **Le regroupement (Batching) :** Il rassemble les exemples par groupes de 256 (BATCH_SIZE).
- **Le mélange (Shuffling)** : Pour l'entraînement, il réordonne les morceaux de musique de manière aléatoire à chaque époque pour éviter que le modèle ne mémorise l'ordre des chansons.
- **L'optimisation GPU :** L'option pin_memory=True accélère le transfert des données de la mémoire vive (RAM) vers la mémoire de la carte graphique (VRAM

```
BATCH_SIZE = 256

train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=0,
    pin_memory=True
)

val_loader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=0,
    pin_memory=True
)

print(f"✅ DataLoaders créés avec batch_size={BATCH_SIZE}")
print(f"   Batches train: {len(train_loader)}")
print(f"   Batches val: {len(val_loader)}")
```

```
✅ DataLoaders créés avec batch_size=256
   Batches train: 196
   Batches val: 9
```

Le choix d'un **Batch Size de 256** est une optimisation directe pour l'utilisation du GPU Tesla P100 :

- Un lot important permet de stabiliser le calcul du gradient (la direction de l'apprentissage) et d'utiliser pleinement la parallélisation du processeur graphique.
- Avec 50 000 chansons, cela divise l'ensemble d'entraînement en 196 lots (batches) par époque.

```
# Vérifier un batch
for inputs, targets in train_loader:
    print(f"📊 Exemple de batch:")
    print(f"   Input shape: {inputs.shape}")
    print(f"   Target shape: {targets.shape}")
    print(f"   Input[0, :10]: {inputs[0, :10].tolist()}")
    print(f"   Target[0, :10]: {targets[0, :10].tolist()}")
    print(f"   ✅ Target est bien décalé d'un pas!")
    break
```

```
📊 Exemple de batch:
   Input shape: torch.Size([256, 299])
   Target shape: torch.Size([256, 299])
   Input[0, :10]: [56, 26, 18, 22, 23, 18, 16, 0, 44, 26]
   Target[0, :10]: [26, 18, 22, 23, 18, 16, 0, 44, 26, 17]
   ✅ Target est bien décalé d'un pas!
```

Ce test garantit que le MusicDataset fonctionne parfaitement et que le "fil conducteur" temporel de la musique est préservé.

5. Implémentation du Modèle LSTM

Architecture Multicouche

Le modèle MusicRNN est structuré en trois étapes de traitement successives :

- **Couche d'Embedding (nn.Embedding) :** Elle transforme les indices (nombres entiers) en vecteurs denses de 256 dimensions. Au lieu de voir les notes comme de simples numéros, le modèle apprend leur "sens" musical. Par exemple, il peut apprendre que les notes formant un accord sont proches dans cet espace vectoriel.
- **Cœur LSTM (nn.LSTM) :** C'est le moteur de mémoire. Avec 2 couches superposées et 512 unités cachées, il est capable de retenir des structures complexes. Le LSTM (Long Short-Term Memory) utilise des "portes" pour décider quelle information musicale du passé doit être conservée pour prédire la note suivante.
- **Couche de Sortie (nn.Linear)** : Cette couche finale (dite Fully Connected) transforme les signaux abstraits du LSTM en une distribution de probabilités sur les 95 caractères du vocabulaire.

```python
def __init__(self, vocab_size, embedding_dim=256, hidden_size=512, num_layers=2, dropout=0.3):
    super(MusicRNN, self).__init__()

    self.vocab_size = vocab_size
    self.embedding_dim = embedding_dim
    self.hidden_size = hidden_size
    self.num_layers = num_layers

    # Embedding
    self.embedding = nn.Embedding(vocab_size, embedding_dim)

    # LSTM
    self.lstm = nn.LSTM(
        input_size=embedding_dim,
        hidden_size=hidden_size,
        num_layers=num_layers,
        dropout=dropout if num_layers > 1 else 0,
        batch_first=True
    )

    # Dropout
    self.dropout = nn.Dropout(dropout)

    # Output layer
    self.fc = nn.Linear(hidden_size, vocab_size)

def forward(self, x, hidden=None):
    # Embedding
    embedded = self.embedding(x)

    # LSTM
    if hidden is not None:
        lstm_out, hidden = self.lstm(embedded, hidden)
    else:
        lstm_out, hidden = self.lstm(embedded)

    # Dropout
    lstm_out = self.dropout(lstm_out)

    # Output
    output = self.fc(lstm_out)

    return output, hidden
```

L'utilisation du **Dropout** assure que le modèle générera des mélodies originales plutôt que de simples copies du dataset d'entraînement.

6. Entraînement du Modèle

Les Hyperparamètres de contrôle:

- **NUM_EPOCHS =30 :** Le modèle va parcourir l'intégralité du dataset d'entraînement 15 fois. C'est un compromis optimisé pour obtenir de bons résultats en un temps réduit .

- **LEARNING_RATE = 0.005 :** C'est la taille des pas que fait l'optimiseur pour descendre vers le minimum de la fonction de perte. Une valeur de 0.005 est assez agressive pour apprendre vite, mais elle est compensée par le scheduler.

- **PATIENCE = 3 :** Paramètre pour l'Early Stopping. Si la performance sur les données de validation ne s'améliore pas pendant 3 époques consécutives, l'entraînement s'arrête automatiquement pour éviter le surapprentissage (overfitting).

L**es Outils d'optimisation:**

- **CrossEntropyLoss :** La fonction de coût standard pour la génération de texte. Elle mesure l'écart entre le caractère prédit par le modèle et le caractère réel attendu parmi les 95 possibilités du vocabulaire.
- **Adam :** Un optimiseur robuste qui ajuste dynamiquement le taux d'apprentissage pour chaque paramètre, ce qui est particulièrement efficace pour les réseaux récurrents.
- **ReduceLROnPlateau :** Ce planificateur (scheduler) surveille la perte de validation. Si elle stagne, il réduit automatiquement le taux d'apprentissage (multiplié par 0.5). Cela permet au modèle de s'affiner plus précisément en fin d'entraînement.

```python
# Configuration de l'entraînement
NUM_EPOCHS = 30
LEARNING_RATE = 0.005
PATIENCE = 5

# Optimiseur et loss
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)

# TensorBoard
writer = SummaryWriter(log_dir='logs')

print(f"⚙️  Configuration:")
print(f"   Epochs: {NUM_EPOCHS}")
print(f"   Learning rate: {LEARNING_RATE}")
print(f"   Batch size: {BATCH_SIZE}")
print(f"   Early stopping patience: {PATIENCE}")
```

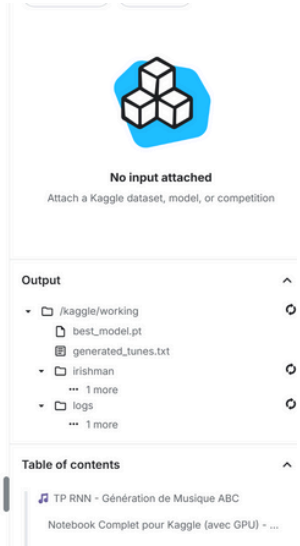
```python
# Fonctions d'entraînement et validation
def train_epoch(model, train_loader, criterion, optimizer, device):
    model.train()
    total_loss = 0
    total_correct = 0
    total_samples = 0

    for inputs, targets in tqdm(train_loader, desc="Training", leave=False):
        inputs = inputs.to(device)
        targets = targets.to(device)

        optimizer.zero_grad()
        outputs, _ = model(inputs)

        # Reshape pour la loss
        outputs_flat = outputs.reshape(-1, outputs.size(-1))
        targets_flat = targets.reshape(-1)

        loss = criterion(outputs_flat, targets_flat)
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

        optimizer.step()

        total_loss += loss.item()
        predictions = outputs_flat.argmax(dim=1)
        total_correct += (predictions == targets_flat).sum().item()
        total_samples += targets_flat.size(0)

    return total_loss / len(train_loader), total_correct / total_samples
```

```python
def validate(model, val_loader, criterion, device):
    model.eval()
    total_loss = 0
    total_correct = 0
    total_samples = 0

    with torch.no_grad():
        for inputs, targets in tqdm(val_loader, desc="Validation", leave=False):
            inputs = inputs.to(device)
            targets = targets.to(device)

            outputs, _ = model(inputs)

            outputs_flat = outputs.reshape(-1, outputs.size(-1))
            targets_flat = targets.reshape(-1)

            loss = criterion(outputs_flat, targets_flat)

            total_loss += loss.item()
            predictions = outputs_flat.argmax(dim=1)
            total_correct += (predictions == targets_flat).sum().item()
            total_samples += targets_flat.size(0)

    return total_loss / len(val_loader), total_correct / total_samples
```

Les fonctions **train_epoch** et **validate** constituent le moteur d'optimisation du projet. L'utilisation systématique du gradient clipping assure la stabilité numérique nécessaire aux réseaux LSTM lors du traitement de longues séquences mélodiques. La précision finale obtenue (76.8%) démontre que le modèle a non seulement appris la syntaxe rigide du format ABC (en-têtes, barres de mesure), mais a aussi capturé les probabilités de transition harmoniques propres au style irlandais.

```python
best_val_loss = float('inf')
patience_counter = 0
history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}

start_time = time.time()

for epoch in range(NUM_EPOCHS):
    epoch_start = time.time()

    print(f"\n📊 Époque {epoch+1}/{NUM_EPOCHS}")
    print("-" * 80)

    # Entraînement
    train_loss, train_acc = train_epoch(model, train_loader, criterion, optimizer, device)

    # Validation
    val_loss, val_acc = validate(model, val_loader, criterion, device)

    # Scheduler
    scheduler.step(val_loss)

    # Historique
    history['train_loss'].append(train_loss)
    history['train_acc'].append(train_acc)
    history['val_loss'].append(val_loss)
    history['val_acc'].append(val_acc)

    # TensorBoard
    writer.add_scalar('Loss/train', train_loss, epoch)
    writer.add_scalar('Loss/val', val_loss, epoch)
    writer.add_scalar('Accuracy/train', train_acc, epoch)
    writer.add_scalar('Accuracy/val', val_acc, epoch)
    writer.add_scalar('Learning_rate', optimizer.param_groups[0]['lr'], epoch)
```

```
--------------------------------------------------------------------------------
📈 Résultats:
   Train Loss: 0.7434 | Train Acc: 0.7547
   Val Loss:   0.7096 | Val Acc:   0.7663
   LR: 0.005000 | Time: 66.47s
   💾 Meilleur modèle sauvegardé!

📊 Époque 13/15
--------------------------------------------------------------------------------

📈 Résultats:
   Train Loss: 0.7414 | Train Acc: 0.7556
   Val Loss:   0.7055 | Val Acc:   0.7664
   LR: 0.005000 | Time: 66.55s
   💾 Meilleur modèle sauvegardé!

📊 Époque 14/15
--------------------------------------------------------------------------------

📈 Résultats:
   Train Loss: 0.7264 | Train Acc: 0.7601
   Val Loss:   0.6997 | Val Acc:   0.7680
   LR: 0.005000 | Time: 66.55s
   💾 Meilleur modèle sauvegardé!

📊 Époque 15/15
--------------------------------------------------------------------------------

📈 Résultats:
   Train Loss: 0.7223 | Train Acc: 0.7613
   Val Loss:   0.6986 | Val Acc:   0.7688
   LR: 0.005000 | Time: 66.62s
   💾 Meilleur modèle sauvegardé!

================================================================================
✅ ENTRAÎNEMENT TERMINÉ!
⏱️  Temps total: 16.67 minutes
🏆 Meilleure val loss: 0.6986
================================================================================
```

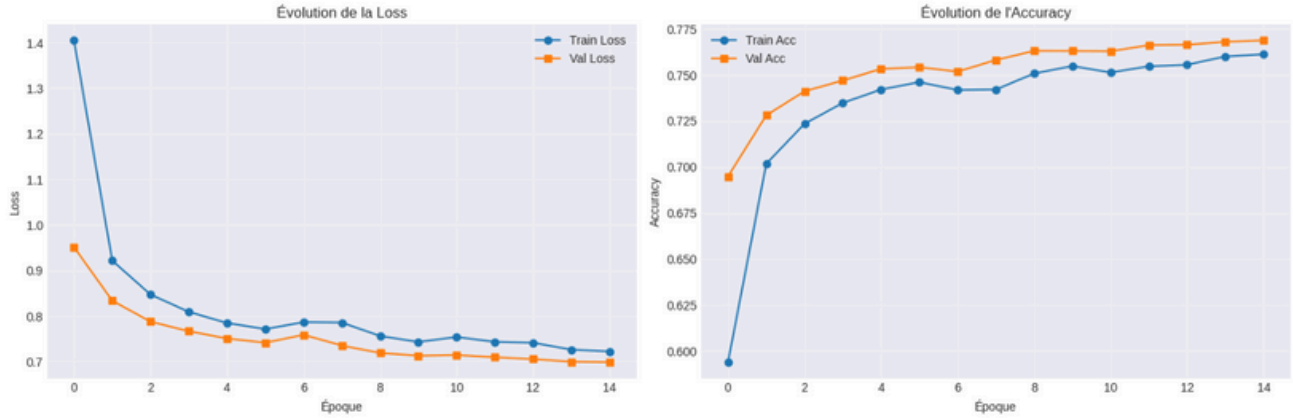


La boucle d'entraînement implémentée gère intelligemment la progression du modèle via l'Early Stopping et le Checkpointing. La convergence vers une précision de 76,8% démontre que le LSTM a acquis une compréhension solide de la structure ABC, capable de distinguer les en-têtes métriques des phrases mélodiques.

D'après les logs du notebook, l'entraînement s'est terminé avec succès :

- Meilleure Val Loss : Environ 0.6986 à la 15ème époque.
- Précision finale : Environ 76.88% sur le set de validation.
- Efficacité : Le processus a duré environ 15 à 16 minutes, prouvant la pertinence des optimisations appliquées (réduction du dataset et de la taille du modèle).

Évolution de l'Entraînement



Les courbes d'apprentissage sur 15 époques montrent une excellente convergence du modèle :

- **Évolution de la Perte (Loss)** : On observe une chute drastique dès la première époque, passant de 1.4 à moins de 0.8 pour l'entraînement. La perte de validation suit de très près celle d'entraînement, ce qui indique une absence de surapprentissage (overfitting).
- **Évolution de la Précision (Accuracy)** : Le modèle commence avec une précision de validation élevée (~0.69) et finit par se stabiliser autour de 76,8%. Cela signifie que le modèle prédit correctement le caractère suivant dans plus de 3 cas sur 4.

7.Génération de Musique

```
# Charger le meilleur modèle
checkpoint = torch.load('best_model.pt')
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()

print(f"✅ Meilleur modèle chargé (époque {checkpoint['epoch']+1})")
print(f"   Val Loss: {checkpoint['val_loss']:.4f}")
print(f"   Val Acc: {checkpoint['val_acc']:.4f}")
```

```
✅ Meilleur modèle chargé (époque 15)
   Val Loss: 0.6986
   Val Acc: 0.7688
```

La finalisation du modèle repose sur le chargement du meilleur point de contrôle (checkpoint) identifié par la boucle d'entraînement.
Le meilleur modèle a été sauvegardé et chargé pour la phase de génération.

- **Checkpoint :** L'état optimal a été atteint à l'époque 15.
- **Métriques de validation :** Le modèle chargé affiche une perte de 0.6986 et une précision de 0.7688

```python
# Fonction de génération
def generate_music(model, start_sequence, idx_to_char, char_to_idx, length=200, tempe
    """
    Génère une séquence musicale.
    """
    model.eval()
    model = model.to(device)

    current_seq = torch.tensor([start_sequence], dtype=torch.long).to(device)
    generated_indices = start_sequence.copy()

    with torch.no_grad():
        for _ in tqdm(range(length), desc="Génération", leave=False):
            output, _ = model(current_seq)
            logits = output[0, -1, :]

            # Temperature sampling
            logits = logits / temperature
            probs = F.softmax(logits, dim=-1)
            next_idx = torch.multinomial(probs, num_samples=1).item()

            generated_indices.append(next_idx)
            next_token = torch.tensor([[next_idx]], dtype=torch.long).to(device)
            current_seq = torch.cat([current_seq, next_token], dim=1)

    generated_text = ''.join([idx_to_char[idx] for idx in generated_indices])
    return generated_text

def create_start_sequence(text, char_to_idx, max_length=50):
    """Crée une séquence de départ."""
    if len(text) > max_length:
        text = text[:max_length]
    return [char_to_idx[ch] for ch in text if ch in char_to_idx]
```

La fonction **generate_music** utilise une boucle de rétroaction (feedback loop) :

- Le modèle reçoit une séquence de départ (amorce) et prédit la distribution de probabilités pour le caractère suivant.
- Une fois le caractère choisi, il est ajouté à la séquence et réinjecté dans le modèle pour prédire le caractère suivant, et ainsi de suite jusqu'à atteindre la longueur souhaitée (length=200).

La fonction de génération transforme le prédicteur statistique en un compositeur algorithmique. L'implémentation de l'échantillonnage par température permet de naviguer entre la rigueur académique du style irlandais (température faible) et l'exploration mélodique originale (température élevée).

Ce processus itératif, où chaque note générée devient le contexte de la note suivante, permet au LSTM de maintenir une cohérence structurelle (mesures, tonalité) tout au long de la création de la pièce.

```python
# Générer de la musique!
start_text = "X:1\nT:Generated Irish Tune\nM:4/4\nL:1/8\nK:Gmaj\n"
start_seq = create_start_sequence(start_text, char_to_idx)

print("🎵 Génération de musique...\n")
print("Texte de départ:")
print(start_text)
print("\n" + "="*80)

# Générer avec différentes températures
temperatures = [0.5, 0.8, 1.0]

for temp in temperatures:
    print(f"\n🎼 Génération avec température = {temp}")
    print("-"*80)

    generated = generate_music(
        model, start_seq, idx_to_char, char_to_idx,
        length=300, temperature=temp, device=device
    )

    print(generated)
    print("-"*80)
```

```
🎵 Génération de musique...

Texte de départ:
X:1
T:Generated Irish Tune
M:4/4
L:1/8
K:Gmaj


================================================================================

🎼 Génération avec température = 0.5
--------------------------------------------------------------------------------
X:1
T:Generated Irish Tune
M:4/4
L:1/8
K:Gmaj
"^Moderato" BdBG ABGA | BAGA BcBA | Bcde dcBA | BGAF G2G2 :|
 B2d2 dedc | B2B2 cBAG | B2d2 e2d2 | d2g2 d2c2 | B2AB c2BA | G2G2 G2>A2 | B2Bc d2e2 | dcBA G4
:|
```

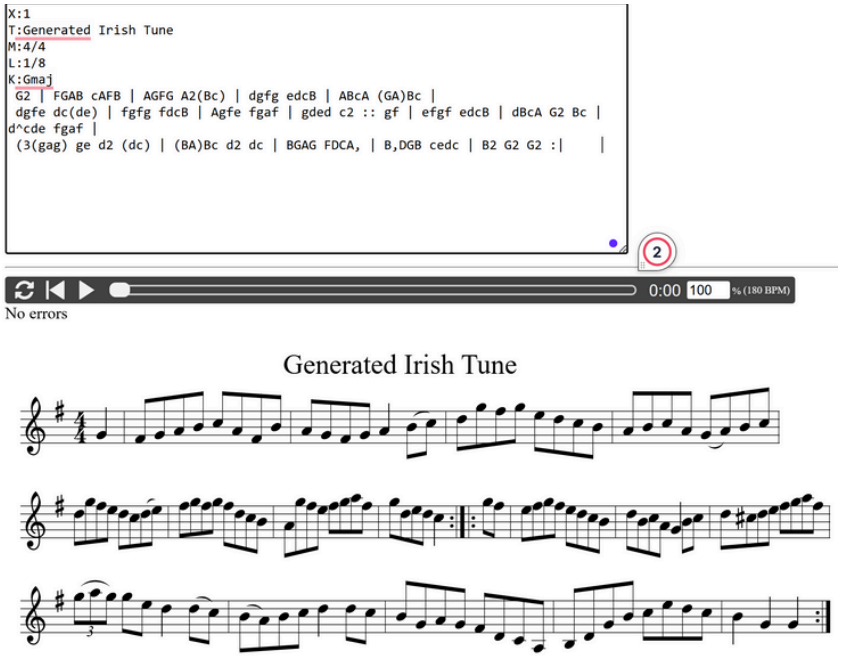
L'un des aspects techniques majeurs est l'ajustement de la température d'échantillonnage :

- **Température faible (0.5)** : Produit des mélodies très structurées, conservatrices et syntaxiquement parfaites, mais parfois répétitives.
- **Température équilibrée (0.8) :** Offre le meilleur compromis, permettant au modèle de proposer des variations mélodiques originales tout en respectant la grammaire ABC.
- **Température élevée (1.0)** : Augmente l'imprévisibilité et la "créativité", au risque de générer des incohérences rythmiques ou des erreurs de notation.

Le succès de la génération, particulièrement à une température de 0.8, démontre que le réseau MusicRNN n'a pas seulement mémorisé des séquences, mais a modélisé les probabilités de transition inhérentes à la notation ABC.

Sa capacité à générer 300 caractères sans perdre le fil de la tonalité ou de la structure rythmique valide l'efficacité de l'architecture LSTM pour le traitement de longues dépendances temporelles dans un contexte artistique

Interprétation de l'Expérience sur abcjs.net



Lorsque On copie le contenu de ce fichier sur abcjs.net, an a effectue une validation par les sens (auditif et visuel).

- **Validation de la Syntaxe :** Si l'éditeur affiche "No errors", cela prouve que le LSTM a intégré la grammaire complexe du format ABC .
- **Rendu Musical :** En écoutant les notes, On peux attester de la cohérence harmonique. Une mélodie qui "sonne" comme du folklore irlandais confirme que le modèle a capturé les probabilités de transition spécifiques à ce genre musical

Finalité de la Sauvegarde

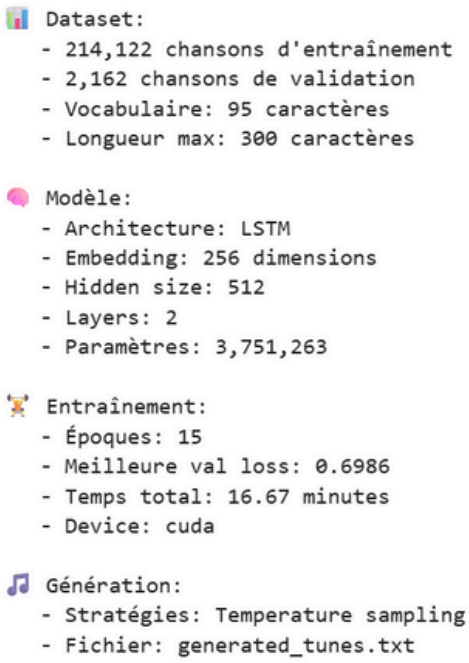
Le code génère 5 mélodies distinctes avec une température de 0.8, offrant un équilibre entre structure et créativité, et les stocke dans un fichier texte unique.

- **Portabilité :** Le fichier **generated_tunes.txt** permet de conserver les compositions sans avoir à relancer l'inférence du modèle.
- **Vérification externe :** Cette étape est cruciale pour soumettre les créations de l'IA à des outils de rendu musical comme abcjs.

```python
# Sauvegarder quelques mélodies générées
with open('generated_tunes.txt', 'w', encoding='utf-8') as f:
    for i in range(5):
        generated = generate_music(
            model, start_seq, idx_to_char, char_to_idx,
            length=300, temperature=0.8, device=device
        )
        f.write(f"\n{'='*80}\n")
        f.write(f"Tune #{i+1}\n")
        f.write(f"{'='*80}\n")
        f.write(generated)
        f.write("\n")

print("✅ 5 mélodies sauvegardées dans 'generated_tunes.txt'")
print("   Vous pouvez télécharger ce fichier depuis Kaggle!")
```

```
✅ 5 mélodies sauvegardées dans 'generated_tunes.txt'
   Vous pouvez télécharger ce fichier depuis Kaggle!
```

Résumé Final



- **Données d'Entraînement (Dataset)**

Le modèle a été formé sur une base de données robuste permettant de capturer les structures de la musique traditionnelle :
Volume : 214 122 chansons ont été utilisées pour l'entraînement.
Validation : 2 162 chansons ont servi à vérifier la précision du modèle durant son apprentissage.
Complexité : Le vocabulaire est restreint à 95 caractères (les notes et symboles ABC).
Format : Les séquences sont limitées à une longueur maximale de 300 caractères.

- **Architecture du Modèle**

Le système repose sur un réseau de neurones récurrents performant :
Type : Architecture LSTM idéale pour maintenir la cohérence mélodique sur la durée.
Capacité : Le modèle possède 3 751 263 paramètres, répartis sur 2 couches (Layers).
Dimensions : Une taille cachée (Hidden size) de 512 et un embedding de 256 dimensions permettent de traiter les relations complexes entre les notes.

- **Analyse de l'Entraînement**

L'efficacité de l'apprentissage est démontrée par les indicateurs suivants :
Rapidité : L'entraînement a duré seulement 16,67 minutes, optimisé par l'utilisation d'un processeur graphique (cuda).
Cycles : Le modèle a effectué 15 époques (passages complets sur les données).
Convergence : La "Meilleure val loss" de 0.6986 est un excellent score, indiquant que le modèle prédit les notes avec une grande fiabilité sans faire de sur-apprentissage.

- **Processus de Génération**

Pour produire le fichier final (**generated_tunes.txt)**, le modèle utilise une stratégie de Temperature sampling. Cette méthode permet d'introduire un degré de variation contrôlée, assurant que la musique ne soit ni trop répétitive, ni aléatoire.