Compte Rendu - TP : Réseaux récurrent

Préparé par : Enihe Nouhaila

Master SDIA

Introduction

Dans le cadre de ce TP sur les réseaux récurrents (RNN), l'objectif est d'appliquer les notions étudiées en cours à un cas concret : la génération de musique en notation ABC. Le travail consiste à charger et explorer un dataset de partitions textuelles, puis à effectuer un prétraitement (vocabulaire de caractères, encodage en indices, padding) afin de préparer les séquences pour l'apprentissage. Ensuite, un modèle LSTM est implémenté avec PyTorch et entraîné pour prédire le caractère suivant d'une partition, avec suivi des performances dans TensorBoard, sauvegarde du meilleur modèle et early stopping. Enfin, le modèle entraîné est utilisé pour générer de nouvelles séquences musicales, permettant d'évaluer la capacité du réseau à apprendre la structure et la logique de la notation ABC.

Chargement et exploration des données

[2]:

```python
from datasets import load_dataset
import json, os

ds = load_dataset("sander-wood/irishman")  # splits: train / validation
os.makedirs("irishman", exist_ok=True)

# Sauvegarde au format proche de ton TP : liste d'objets
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("OK -> irishman/train.json & irishman/validation.json")
```

```
README.md: 6.27k/? [00:00<00:00, 543kB/s]
train.json: 100% 80.0M/80.0M [00:01<00:00, 87.4MB/s]
validation.json: 797k/? [00:00<00:00, 23.3MB/s]
Generating train split: 100% 214122/214122 [00:01<00:00, 136539.79 examples/s]
Generating validation split: 100% 2162/2162 [00:00<00:00, 86315.04 examples/s]
OK -> irishman/train.json & irishman/validation.json
```

Les fichiers **train.json** et **validation.json** ont été chargés depuis le dossier irishman/. Chaque fichier contient une liste d'objets JSON avec la clé **abc notation** (partition en texte ABC).

[3]:
```python
import json

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:
    val_data = json.load(f)

print("Nb chansons train :", len(train_data))
print("Nb chansons val   :", len(val_data))

print("\nClés disponibles dans un exemple :", list(train_data[0].keys()))
```

```
Nb chansons train : 214122
Nb chansons val   : 2162

Clés disponibles dans un exemple : ['abc notation', 'control code']
```

On obtient **214122** chansons pour l'entraînement et **2162** pour la validation.

[5]:
```python
def find_abc_key(example: dict):
    keys = list(example.keys())
    # priorité à une clé qui contient "abc" et "notation"
    for k in keys:
        lk = k.lower()
        if "abc" in lk and "notation" in lk:
            return k
    # sinon une clé qui contient "abc"
    for k in keys:
        if "abc" in k.lower():
            return k
    raise ValueError(f"Aucune clé ABC trouvée. Clés: {keys}")

abc_key = find_abc_key(train_data[0])
print("abc_key =", abc_key)

first_song = train_data[0][abc_key]
print("\n--- Première chanson (texte brut) ---\n")
print(first_song)

print("\n--- Aperçu lignes (structure) ---\n")
for line in first_song.splitlines()[:15]:
    print(line)
```

```
abc_key = abc notation

--- Première chanson (texte brut) ---

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 ||

--- Aperçu lignes (structure) ---

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 ||
```

L'affichage d'un exemple montre une structure typique de la notation ABC : en-têtes (ex. X:, L:, M:, K:) suivis du corps mélodique (notes, barres |, durées, symboles).

Interprétation :

Le dataset est bien adapté à une approche “caractère par caractère” : la musique est représentée comme une **séquence de caractères ASCII** (métadonnées + notes). L’objectif du modèle sera donc d’apprendre la logique de cette séquence pour prédire le prochain caractère et générer de nouvelles partitions.

Étape 1 : Prétraitement : extraction des caractères uniques

```
]:
unique_chars = set()

for obj in train_data:
    s = obj[abc_key]
    unique_chars.update(list(s))

print("Nb caractères uniques (train) :", len(unique_chars))
print("Exemple de caractères :", sorted(list(unique_chars))[:80])
```

```
Nb caractères uniques (train) : 95
Exemple de caractères : ['\n', ' ', '!', '"', '#', '$', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/', '0', '1',
'2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o']
```

a) Caractères uniques (train) : On a parcouru toutes les partitions du jeu d’entraînement et extrait l’ensemble des caractères distincts utilisés (lettres, chiffres, symboles ABC, espaces et retours à la ligne).

b) Nombre de caractères uniques : Le dataset d’entraînement contient **95 caractères uniques.**

c) Pourquoi utiliser des indices ? : Un modèle (*PyTorch*) ne traite que des données numériques ; chaque caractère est donc converti en indice pour pouvoir l’encoder (ex. via une couche *Embedding*) et apprendre à prédire le caractère suivant.

Étape 2 : Mapping caractères-index

```
:
# Étape 2 — Mapping caractères-index

# On part de unique_chars calculé à l'étape 1
# Important : on fixe un ordre stable (tri) pour que les index soient reproductibles
idx2char = sorted(list(unique_chars))          # index -> char
char2idx = {c: i for i, c in enumerate(idx2char)}  # char -> index

print("Taille vocabulaire :", len(idx2char))
print("Exemple mapping char2idx :", {c: char2idx[c] for c in idx2char[:10]})
print("Exemple mapping idx2char :", [idx2char[i] for i in range(10)])
```

```
Taille vocabulaire : 95
Exemple mapping char2idx : {'\n': 0, ' ': 1, '!': 2, '"': 3, '#': 4, '$': 5, '&': 6, "'": 7, '(': 8, ')': 9}
Exemple mapping idx2char : ['\n', ' ', '!', '"', '#', '$', '&', "'", '(', ')']
```

Après avoir identifié l’ensemble des caractères uniques du dataset d’entraînement, nous avons construit deux structures de correspondance afin de convertir les partitions ABC en données numériques.

Le premier mapping, **char2idx**, est un dictionnaire qui associe chaque caractère à un index unique. Le second, **idx2char**, est une liste qui permet l'opération inverse : retrouver le caractère à partir de son index (idx2char[i]).

Un ordre stable (tri des caractères) a été utilisé pour garantir la reproductibilité des index. La taille du vocabulaire obtenue est de 95 caractères, ce qui correspond au nombre de classes possibles à prédire pour le modèle lors de l'apprentissage.

Étape 3 : Vectorisation des chaînes

```python
# Étape 3 — Vectorisation des chaînes

def vectorize_string(s: str, char2idx: dict):
    return [char2idx[c] for c in s]

# Test avec la première chanson du train
first_song = train_data[0][abc_key]

vec = vectorize_string(first_song, char2idx)

print("Longueur texte :", len(first_song))
print("Longueur vect  :", len(vec))
print("Début (indices):", vec[:50])
print("Début (reconstruit):", "".join(idx2char[i] for i in vec[:200]))
```

```
Longueur texte : 183
Longueur vect  : 183
Début (indices): [56, 26, 17, 0, 44, 26, 17, 15, 24, 0, 45, 26, 20, 15, 20, 0, 43, 26, 37, 77, 73, 78, 0, 92, 26, 1,
37, 18, 1, 37, 38, 1, 37, 18, 1, 37, 38, 1, 92, 1, 36, 37, 38, 39, 1, 33, 38, 36, 38, 1]
Début (reconstruit): 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 ||
```

Après avoir construit les correspondances char2idx et idx2char, nous avons implémenté la fonction **vectorize_string** qui transforme une partition en notation ABC (chaîne de caractères) en une **liste d'indices**. Chaque caractère de la chaîne est remplacé par son identifiant numérique via char2idx, ce qui permet d'obtenir une séquence exploitable par PyTorch.

Le test sur la première chanson montre que la **longueur du texte** est égale à la **longueur du vecteur** (ex. 183 caractères → 183 indices), ce qui confirme que la transformation conserve l'information originale. Une reconstruction partielle avec idx2char redonne bien le début de la partition, ce qui valide la cohérence du mapping et de la vectorisation.

Étape 4 — Padding des séquences

```
# Étape 4a — Longueur maximale des séquences (train)
max_len = max(len(obj[abc_key]) for obj in train_data)
print("Longueur maximale (train) :", max_len)
```

```
Longueur maximale (train) : 2968
```

```
# Étape 4b — Padding / Troncature

def pad_or_truncate(s: str, max_len: int, pad_char: str = " "):
    if len(s) < max_len:
        return s + pad_char * (max_len - len(s))
    return s[:max_len]

# Test sur une chanson
s = train_data[0][abc_key]
s2 = pad_or_truncate(s, max_len)

print("Avant :", len(s))
print("Après :", len(s2))
print("Derniers caractères (après) :", repr(s2[-50:]))
```

```
Avant : 183
Après : 2968
Derniers caractères (après) : '                                                  '
```

Afin de pouvoir former des **batches** (toutes les séquences doivent avoir la même taille), nous avons d'abord calculé la longueur maximale des partitions du jeu d'entraînement. Le calcul sur train_data a donné une longueur maximale de **2968 caractères**.

Ensuite, une fonction pad_or_truncate a été implémentée :

- si une partition est plus courte que max_len, on ajoute des **espaces** à la fin (padding) jusqu'à atteindre max_len ;
- si elle est **plus longue**, on **coupe** la fin (troncature) pour conserver exactement max_len caractères.

Le test sur la première chanson confirme le bon fonctionnement : la séquence initiale de **183 caractères** devient une séquence de **2968 caractères**, et l'affichage des derniers caractères montre principalement des **espaces**, ce qui prouve que le padding a bien été ajouté.

Création du dataset PyTorch

Étape 1 : Préparation des données

Nous avons regroupé toutes les étapes de prétraitement dans une fonction prepare_data afin de produire directement des séquences prêtes à l'emploi. Cette fonction applique le **padding/troncature** à une longueur fixe (max_len), puis transforme chaque partition ABC en une **séquence d'indices** grâce au dictionnaire char2idx. Les séquences finales sont stockées sous forme de tenseurs PyTorch de type torch.long, ce qui les rend compatibles avec une couche d'Embedding.

```python
import torch
from torch.utils.data import Dataset, DataLoader

def prepare_data(train_data, val_data, abc_key, char2idx, max_len, pad_char=" "):
    def pad_or_truncate(s: str):
        if len(s) < max_len:
            return s + pad_char * (max_len - len(s))
        return s[:max_len]

    def vectorize(s: str):
        return [char2idx[c] for c in s]

    train_seqs = [torch.tensor(vectorize(pad_or_truncate(obj[abc_key])), dtype=torch.long)
                  for obj in train_data]
    val_seqs   = [torch.tensor(vectorize(pad_or_truncate(obj[abc_key])), dtype=torch.long)
                  for obj in val_data]

    return train_seqs, val_seqs
```

Étape 2 : Dataset et DataLoader

```python
class MusicDataset(Dataset):
    def __init__(self, sequences):
        self.sequences = sequences  # liste de tensors [max_len]

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

    def __getitem__(self, idx):
        seq = self.sequences[idx]     # [L]
        x = seq[:-1]                  # [L-1]
        y = seq[1:]                   # [L-1] (décalée)
        return x, y

# Construire les sequences
train_seqs, val_seqs = prepare_data(train_data, val_data, abc_key, char2idx, max_len)

train_ds = MusicDataset(train_seqs)
val_ds   = MusicDataset(val_seqs)

train_loader_check = DataLoader(train_ds, batch_size=8, shuffle=True)
val_loader_check   = DataLoader(val_ds, batch_size=8, shuffle=False)

xb, yb = next(iter(train_loader_check))
print("x batch shape:", xb.shape)  # [8, L-1]
print("y batch shape:", yb.shape)  # [8, L-1]
print("Exemple x[0][:20]:", xb[0, :20])
print("Exemple y[0][:20]:", yb[0, :20])
```

```
x batch shape: torch.Size([8, 2967])
y batch shape: torch.Size([8, 2967])
Exemple x[0][:20]: tensor([56, 26, 18, 17, 16, 18, 25, 16,  0, 44, 26, 17, 15, 24,  0, 45, 26, 22,
        15, 24])
Exemple y[0][:20]: tensor([26, 18, 17, 16, 18, 25, 16,  0, 44, 26, 17, 15, 24,  0, 45, 26, 22, 15,
        24,  0])
```

Nous avons implémenté la classe MusicDataset héritant de **torch.utils.data.Dataset.** Pour chaque séquence, la méthode __getitem__ renvoie :

- la séquence d'entrée **x = seq[:-1]** (tout sauf le dernier caractère),
- la séquence cible **y = seq[1:]** (tout sauf le premier caractère), donc décalée d'un pas.

Ensuite, un DataLoader a été créé pour le train et la validation avec **batch_size = 8** afin de vérifier le bon fonctionnement. Le batch obtenu a la forme **[8, 2967]** pour x et y, ce qui confirme que le dataset génère bien des couples (entrée, cible) alignés dans le temps.

Implémentation du modèle LSTM

Le modèle MusicRNN est composé de :

- une couche **Embedding** pour transformer les indices en vecteurs continus,
- une couche **LSTM** pour modéliser les dépendances temporelles,
- une couche **linéaire (Fully Connected)** pour produire les logits de prédiction sur tout le vocabulaire.

Le modèle prend en entrée un tenseur x de forme [B, T] et retourne des logits [B, T, V], où V est la taille du vocabulaire (95 caractères).

```
!pip -q uninstall -y tensorboard tensorboard-data-server protobuf
!pip -q install tensorboard==2.15.2 protobuf==3.20.3
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 53.8 MB/s eta 0:00:00a 0:00:01
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 162.1/162.1 kB 13.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.6/6.6 MB 86.7 MB/s eta 0:00:00:00:0100:01
```

```
import torch.nn as nn

class MusicRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden=None):
        # x: [B, T]
        e = self.embed(x)                # [B, T, E]
        out, hidden = self.lstm(e, hidden)# [B, T, H]
        logits = self.fc(out)            # [B, T, V]
        return logits, hidden
```

Boucle d'entraînement (TensorBoard + Early Stopping + Sauvegarde)

Nous avons écrit une fonction train_model qui :

- entraîne le modèle sur un nombre d'itérations défini,
- calcule la **loss** (CrossEntropy) et l'**accuracy**,
- logge les métriques train/val dans **TensorBoard** via SummaryWriter,
- applique un **early stopping** basé sur la loss de validation,
- sauvegarde automatiquement le **meilleur modèle** (torch.save) lorsqu'il améliore la validation.

```python
from torch.utils.tensorboard import SummaryWriter
import torch.optim as optim
import os

def accuracy_from_logits(logits, y):
    # logits: [B,T,V], y: [B,T]
    preds = logits.argmax(dim=-1)
    return (preds == y).float().mean().item()

def train_model(model, train_ds, val_ds, num_training_iterations=3000,
                batch_size=256, learning_rate=5e-3,
                patience=3, log_dir="runs/music_rnn", save_path="best_music_rnn.pt"):

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
    val_loader   = DataLoader(val_ds, batch_size=batch_size, shuffle=False)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    writer = SummaryWriter(log_dir=log_dir)

    steps_per_epoch = len(train_loader)
    max_epochs = (num_training_iterations + steps_per_epoch - 1) // steps_per_epoch

    best_val_loss = float("inf")
    bad_epochs = 0
    global_step = 0
```

```python
    for epoch in range(1, max_epochs + 1):
        # ---- Train ----
        model.train()
        train_loss_sum, train_acc_sum, n_train = 0.0, 0.0, 0

        for xb, yb in train_loader:
            if global_step >= num_training_iterations:
                break

            xb, yb = xb.to(device), yb.to(device)

            optimizer.zero_grad()
            logits, _ = model(xb)

            # CrossEntropy: [B*T, V] vs [B*T]
            loss = criterion(logits.reshape(-1, logits.size(-1)), yb.reshape(-1))
            loss.backward()
            optimizer.step()

            acc = accuracy_from_logits(logits, yb)

            bs = xb.size(0)
            train_loss_sum += loss.item() * bs
            train_acc_sum  += acc * bs
            n_train += bs

            global_step += 1

        train_loss = train_loss_sum / max(1, n_train)
        train_acc  = train_acc_sum  / max(1, n_train)

        # ---- Val ----
        model.eval()
        val_loss_sum, val_acc_sum, n_val = 0.0, 0.0, 0
```

```
        with torch.no_grad():
            for xb, yb in val_loader:
                xb, yb = xb.to(device), yb.to(device)
                logits, _ = model(xb)
                loss = criterion(logits.reshape(-1, logits.size(-1)), yb.reshape(-1))
                acc = accuracy_from_logits(logits, yb)

                bs = xb.size(0)
                val_loss_sum += loss.item() * bs
                val_acc_sum  += acc * bs
                n_val += bs

        val_loss = val_loss_sum / max(1, n_val)
        val_acc  = val_acc_sum  / max(1, n_val)

        # TensorBoard logs
        writer.add_scalar("loss/train", train_loss, epoch)
        writer.add_scalar("loss/val",   val_loss,   epoch)
        writer.add_scalar("acc/train",  train_acc,  epoch)
        writer.add_scalar("acc/val",    val_acc,    epoch)

        print(f"Epoch {epoch}/{max_epochs} | train_loss={train_loss:.4f} val_loss={val_loss:.4f} | train_acc={train_

        # Early stopping + best save
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            bad_epochs = 0
            torch.save(model.state_dict(), save_path)
        else:
            bad_epochs += 1
            if bad_epochs >= patience:
                print("Early stopping déclenché.")
                break

    writer.close()
    print("Best model saved ->", save_path)
    return save_path
```

À cause de contraintes mémoire GPU, les hyperparamètres ont été ajustés pour l'exécution, par exemple :

- **embedding_dim = 128, hidden_size = 512**
- **batch_size = 32**

```
vocab_size = len(idx2char)
embedding_dim = 128
hidden_size = 512

model = MusicRNN(vocab_size, embedding_dim, hidden_size)

best_path = train_model(
    model,
    train_ds, val_ds,
    num_training_iterations=3000,
    batch_size=32,
    learning_rate=5e-3,
    patience=3,
    save_path="best_music_rnn.pt"
)
```

```
Epoch 1/1 | train_loss=0.1122 val_loss=0.0950 | train_acc=0.9645 val_acc=0.9692
Best model saved -> best_music_rnn.pt
```

Résultat observé : **Epoch 1/1, train_loss = 0.1122, val_loss = 0.0950, train_acc = 0.9645, val_acc = 0.9692**

Le meilleur modèle a été sauvegardé dans **best_music_rnn.pt**.

Interprétation :

L'early stopping évite le **surapprentissage** et réduit le temps d'entraînement. Le logging TensorBoard permet de suivre l'évolution des performances. La sauvegarde du meilleur modèle garantit qu'on conserve la version la plus performante sur validation pour la génération de musique.

Génération de musique

Après l'entraînement et la sauvegarde du meilleur modèle, nous avons utilisé le LSTM pour générer une nouvelle partition ABC caractère par caractère. La fonction generate_music prend en entrée le modèle, une **séquence de départ (seed)**, ainsi qu'une longueur de génération (ici **200 caractères**). La seed est d'abord **vectorisée** grâce à char2idx afin d'obtenir une liste d'indices, puis elle est passée au modèle une première fois pour **initialiser l'état caché** du LSTM.

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MusicRNN(vocab_size=len(idx2char), embedding_dim=embedding_dim, hidden_size=hidden_size).to(device)
model.load_state_dict(torch.load("best_music_rnn.pt", map_location=device))
model.eval()
```

```
MusicRNN(
  (embed): Embedding(95, 128)
  (lstm): LSTM(128, 512, batch_first=True)
  (fc): Linear(in_features=512, out_features=95, bias=True)
)
```

Ensuite, la génération se fait de manière itérative : à chaque étape, on fournit au modèle le **dernier caractère généré**, on récupère les **logits** sur le vocabulaire, puis on calcule une distribution de probabilités avec softmax. Contrairement à l'approche *greedy* (qui prend toujours la classe la plus probable), nous utilisons un **échantillonnage** (torch.multinomial) pour choisir le prochain caractère, ce qui permet d'obtenir des séquences plus variées. Le paramètre temperature contrôle cette diversité : une température plus élevée augmente la variété, tandis qu'une température plus faible rend la génération plus "prudente".

```python
import torch.nn.functional as F

def generate_music(model, start_seq, char2idx, idx2char, length=200, temperature=1.0):
    device = next(model.parameters()).device
    model.eval()

    # vectoriser la séquence de départ
    start_ids = [char2idx[c] for c in start_seq]
    generated = start_ids[:]  # liste d'indices

    hidden = None

    with torch.no_grad():
        # passer toute la seed pour initialiser l'état caché
        x = torch.tensor([start_ids], dtype=torch.long, device=device)  # [1, T]
        _, hidden = model(x, hidden)

        last_id = start_ids[-1]

        for _ in range(length):
            inp = torch.tensor([[last_id]], dtype=torch.long, device=device)  # [1,1]
            logits, hidden = model(inp, hidden)  # logits: [1,1,V]
            logits = logits[0, 0]  # [V]

            # temperature + proba
            logits = logits / max(temperature, 1e-6)
            probs = F.softmax(logits, dim=-1)

            # échantillonnage
            next_id = torch.multinomial(probs, num_samples=1).item()

            generated.append(next_id)
            last_id = next_id

    return "".join(idx2char[i] for i in generated)
```

Résultat : la séquence générée conserve une structure ABC cohérente, notamment les en-têtes (X:, L:, M:, K:) suivis d'un corps mélodique composé de notes et séparateurs (|), ce qui montre que le modèle a appris des motifs réguliers de la notation ABC et peut produire de nouvelles partitions plausibles.

```python
seed = "X:1\nL:1/8\nM:4/4\nK:Em\n"
generated_abc = generate_music(model, seed, char2idx, idx2char, length=200, temperature=1.0)

print("----- Generated ABC -----")
print(generated_abc)
```

```
----- Generated ABC -----
X:1
L:1/8
M:4/4
K:Em
 E2 BE G/A/B B2 | A2 A2 G2 E>G | E2 E>D E2 (E/F/E/D/) | E2 E G3 G3 G |
 B2 B>c B>A G2 | E4 E4 | F4 G3 F | E/E/E/E/ G>E E4 ||
```