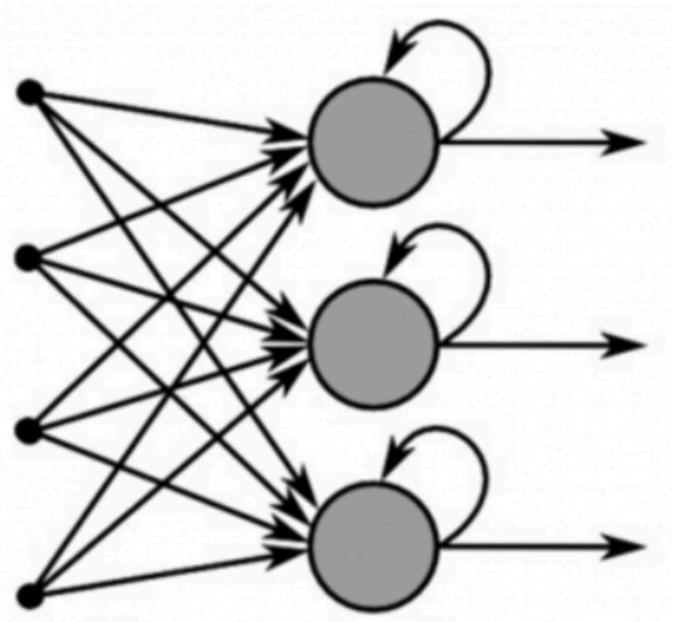
Département Télécommunications, Réseaux et Informatique

Master SDIA

Master Artificial Intelligence
&
Data Science

Rapport TP1_RNN



RNN

fait par RIM GOURRAM

Prof Youness Abouqora

année universitaire 2025/2026

Chapitre 1 : Introduction

1. Introduction:

Les réseaux de neurones récurrents (RNN) constituent une architecture particulièrement adaptée à la modélisation de données séquentielles, telles que le texte, la parole ou la musique. Contrairement aux réseaux neuronaux classiques, les RNN disposent d'une mémoire interne leur permettant de capturer les dépendances temporelles entre les éléments d'une séquence, ce qui est essentiel pour des tâches de prédiction ou de génération séquentielle.

Dans ce TP, nous avons exploré l'application des RNN à la génération de musique au format ABC, un format textuel compact pour représenter des partitions musicales, particulièrement adapté aux mélodies monodiques. L'objectif principal était de concevoir un modèle capable de prédire la note suivante à partir d'un contexte musical donné, puis d'utiliser ce modèle pour générer de nouvelles séquences musicales cohérentes.

Le TP s'articule autour de plusieurs étapes : le prétraitement des données pour les rendre compatibles avec un RNN, la création d'un dataset PyTorch et la gestion des séquences via un DataLoader, l'implémentation d'un modèle LSTM avec couche d'embedding et couche dense de sortie, ainsi que la mise en place d'une boucle d'entraînement incluant le logging, l'early stopping et la sauvegarde du meilleur modèle. Enfin, le modèle entraîné est utilisé pour générer de nouvelles partitions musicales en utilisant des stratégies de prédiction telles que l'approche greedy et l'échantillonnage probabiliste.

Chapitre 2 : Présentation des données et du format ABC

Introduction:

Dans ce chapitre, nous présentons les données utilisées pour entraîner notre modèle de génération musicale, ainsi que le format ABC qui les caractérise. Les données proviennent de partitions musicales au format texte, regroupées en ensembles d'entraînement et de validation. Chaque partition est représentée sous forme de chaîne de caractères, permettant ainsi un traitement séquentiel adapté aux réseaux récurrents.

1. Définition du format ABC

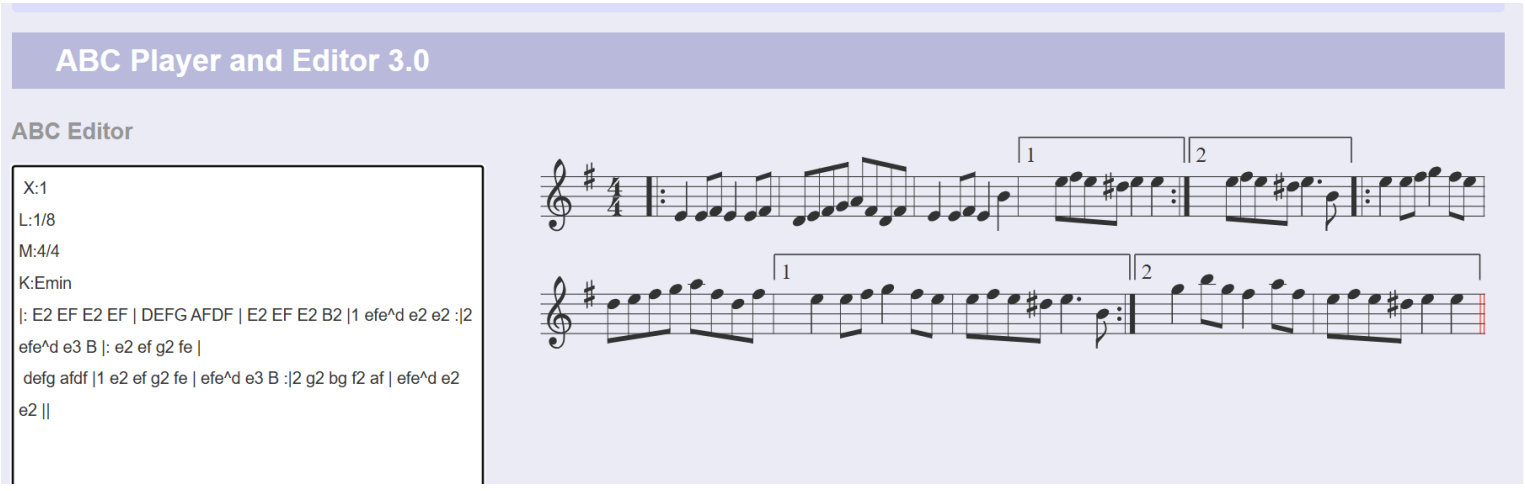
Le format ABC est un standard textuel pour représenter des partitions musicales sous forme de caractères ASCII. Il permet de coder les notes, leurs hauteurs et durées, les mesures, ainsi que certaines informations musicales comme le titre, la métrique et la tonalité. Ce format est particulièrement utilisé pour les mélodies monodiques, comme celles du répertoire traditionnel irlandais, et permet une manipulation simple des

partitions par des programmes informatiques, tout en restant lisible par l'homme.

```
X: 0
T: My Tune
M: 4/4
L: 1/4
K: C
C,D,E,F, | G,A,B,C | DEFG | ABcd | efga |bc'd'e' | f'g'a'b' |
```

2. Exemple d'utilisation

Pour mieux comprendre le format ABC, nous pouvons visualiser une partition réelle. Nous prenons la **première chanson** de notre dataset d'entraînement et la copions dans un lecteur/éditeur ABC en ligne, tel que [ABC Player and Editor]. Cela permet de jouer la mélodie et de voir concrètement comment les caractères ASCII se traduisent en notes musicales, en rythme et en mesure.



3. Présentation de la Dataset

Pour entraîner et évaluer notre modèle RNN, nous avons utilisé le dataset Irishman, disponible sur Hugging Face. Ce dataset contient des partitions musicales au format ABC et est divisé en deux parties :

- train.json : données d'entraînement
- validation.json : données de validation

`train_data`

..

	control code	abc notation
0	S:2\nB:5\nE:5\nB:6\n	X:1\nL:1/8\nM:4/4\nK:Emin\n|: E2 EF E2 EF | DE...
1	S:1\nB:25\n	X:2\nL:1/4\nM:3/4\nK:C\n G | E3/2 E/ E | G2 G ...
2	S:2\nB:9\nE:7\nB:9\n	X:3\nL:1/8\nQ:1/4=100\nM:2/4\nK:Emin\n E | ABc...
3	S:4\nB:1\nE:1\nB:8\nE:3\nE:1\nB:1\nE:1\nE:4\nE...	X:4\nL:1/8\nM:6/8\nK:A\n|: EFG ||"A" A2 A A2 A...
4	S:2\nB:8\nE:5\nB:8\n	X:5\nL:1/8\nM:6/8\nK:G\n (G/A/B).D D>ED | DcB ...
...	...	...
214117	S:1\nB:17\n	X:214118\nL:1/8\nM:2/4\nK:F\n C | FFFF | AFFF ...
214118	S:2\nB:9\nE:6\nB:9\n	X:214119\nL:1/8\nM:4/4\nK:G\n (g/>a/) | (b<g) ...
214119	S:3\nB:8\nE:3\nB:4\nE:5\nE:4\nB:6\n	X:214120\nL:1/8\nM:3/4\nK:A\n|: [Eca]3 c/a/ B3...
214120	S:3\nB:8\nE:5\nB:8\nE:2\nE:3\nB:16\n	X:214121\nL:1/8\nQ:1/4=110\nM:2/4\nK:D\n"_See ...
214121	S:4\nB:9\nE:4\nB:9\nE:5\nE:4\nB:9\nE:4\nE:6\nE...	X:214122\nL:1/8\nQ:1/8=140\nM:3/4\nK:F\n"^slow...

214122 rows × 2 columns

Le dataset utilisé pour entraîner notre modèle contient 214 123 partitions musicales au format ABC. Chaque partition est représentée par deux colonnes principales :

1. control code : informations techniques sur la partition (par exemple, numéro de mesure, durée des notes, tempo).

2. abc notation : la partition musicale en notation ABC, un format texte compact qui encode les notes, leur durée, les mesures, la clé et

éventuellement d'autres métadonnées (titre, tonalité, métrique).

val_data

	control code	abc notation
0	S:2\nB:9\nE:3\nB:10\n	X:1\nL:1/8\nM:6/8\nK:Bb\n F | B2 d c2 f | edc ...
1	S:2\nB:8\nE:3\nB:8\n	X:2\nL:1/8\nM:4/4\nK:G\n GBAB G2 ge | dBGB AE ...
2	S:5\nB:1\nE:1\nB:5\nE:1\nE:5\nB:5\nE:0\nE:4\nE...	X:3\nL:1/8\nM:4/4\nK:A\n d |: cABG A2 ED | CEA...
3	S:2\nB:8\nE:6\nB:8\n	X:4\nL:1/8\nM:2/4\nK:Amin\n|: ce/d/ cB | Aa^ga...
4	S:2\nB:8\nE:6\nB:8\n	X:5\nL:1/8\nM:4/4\nK:Amix\n cA A2 EA A2 | BGdG...
...	...	...
2157	S:2\nB:9\nE:5\nB:9\n	X:2158\nL:1/8\nM:6/8\nK:G\n f | edB BAB | GEF ...
2158	S:1\nB:16\n	X:2159\nL:1/8\nQ:1/8=232\nM:4/4\nK:D\n|: A2 AB...
2159	S:2\nB:9\nE:5\nB:9\n	X:2160\nL:1/8\nM:6/8\nK:Amix\n F2 D D=CA, | =C...
2160	S:1\nB:16\n	X:2161\nL:1/8\nM:4/4\nK:G\n g3 d BGAB | cBAG F...
2161	S:2\nB:9\nE:8\nB:9\n	X:2162\nL:1/8\nM:3/4\nK:G\n DG | B2 BA GB | d2...

2162 rows × 2 columns

Le dataset de validation contient 2 163 partitions musicales, également au format ABC, organisées en deux colonnes :

1. control code : informations techniques sur la partition (numéro de mesure, durée des notes, tempo, etc.).

2. abc notation : partition musicale au format ABC, encodant les notes, leur durée, les mesures, la tonalité et d'autres métadonnées.

Chapitre 3 :Prétraitement et exploration des données

Introduction:

Dans ce chapitre, nous présentons la mise en œuvre pratique du modèle RNN pour la génération musicale. Nous détaillerons l'ensemble du pipeline, depuis le prétraitement des données et la préparation des datasets, jusqu'à l'entraînement du modèle, l'évaluation sur les données de validation, et enfin la génération de nouvelles séquences musicales. L'objectif est de montrer comment les concepts théoriques vus précédemment se traduisent en étapes concrètes permettant de créer et tester un modèle capable de prédire et générer de la musique au format ABC.

Étape 1 : Chargement et exploration des données

```
print("Nombre de chansons dans le train :", len(train_data))
print("Nombre de chansons dans la validation :", len(val_data))
```

```
Nombre de chansons dans le train : 214122
Nombre de chansons dans la validation : 2162
```

Première chanson

```
first_song = train_data.loc[0, 'abc notation']
print("Première chanson :\n", first_song)
```

```
Première chanson :
 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 ||
```

Chaque partition est un texte : on peut la considérer comme une séquence de caractères.

RNN / LSTM va apprendre à prédire le caractère suivant, donc il va apprendre les notes, rythmes et motifs. Les caractères uniques seront : lettres (A-G, a-g), chiffres (1-3…), symboles (|, :, ^, etc.). Les lignes d'en-tête (X:, L:, M:, K:) fournissent du contexte musical.

Étape 2 : Extraction des caractères uniques

```
# Concaténer toutes les partitions en une seule grande chaîne
all_text = "".join(train_data['abc notation'].tolist())

# Trouver les caractères uniques
unique_chars = sorted(list(set(all_text)))

print("Caractères uniques :", unique_chars)
print("Nombre de caractères uniques :", len(unique_chars))
```

```
Caractères uniques : ['\n', ' ', '!', '"', '#', '$', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/',
Nombre de caractères uniques : 95
```

Raisons principales d'utiliser des indices:

- Compatibilité avec les tenseurs : PyTorch et TensorFlow ne traitent que des nombres.

- Facilité de batch processing : on peut créer des matrices (batch_size × sequence_length) pour entraîner le RNN.
- Embeddings : les indices permettent de projeter chaque caractère dans un espace vectoriel dense.
- Génération : le RNN prédit des probabilités sur les indices, et on reconvertit les indices en caractères pour lire la partition.

Étape 3 : Mapping caractères-index

Écrivez un dictionnaire permettant de passer d'un caractère à un index.

```
# char2idx : chaque caractère unique a un index unique
char2idx = {ch: idx for idx, ch in enumerate(unique_chars)}

# Exemple d'utilisation
print("Index du caractère 'E' :", char2idx['E'])
```

```
Index du caractère 'E' : 37
```

Dans cette étape, nous créons un mapping caractère → index (char2idx) qui permet de convertir chaque caractère unique du dataset en un entier unique. Cette transformation est essentielle car les modèles de deep learning, comme les RNN, ne peuvent pas traiter directement des caractères ou des chaînes de texte : ils travaillent uniquement avec des représentations numériques.

Écrivez une liste permettant de passer d'un index à un caractère.

```
# Option 1 : avec un dictionnaire
idx2char = {idx: ch for idx, ch in enumerate(unique_chars)}

# Option 2 : plus simple avec une liste
idx2char_list = list(unique_chars)

# Tester
print("Caractère pour l'index 30 (dictionnaire) :", idx2char[37])
print("Caractère pour l'index 30 (liste) :", idx2char_list[37])
```

```
Caractère pour l'index 30 (dictionnaire) : E
Caractère pour l'index 30 (liste) : E
```

Ici, nous effectuons l'opération inverse du mapping précédent : nous créons un mapping index → caractère (idx2char). Cela permet de convertir les prédictions numériques du modèle en caractères lisibles.

Étape 4 : Vectorisation des chaînes

Maintenant, nous allons écrire une fonction qui transforme une chaîne de caractères en une liste d'indices correspondants.

```
def vectorize_string(text, char2idx):

    return [char2idx[ch] for ch in text]

# Test avec la première partition du dataset
first_song = train_data['abc notation'].iloc[0]
vectorized_first_song = vectorize_string(first_song, char2idx)

# Afficher un petit extrait
print("Premiers indices :", vectorized_first_song[:20])
print("Longueur de la partition :", len(vectorized_first_song))
```

```
Premiers indices : [56, 26, 17, 0, 44, 26, 17, 15, 24, 0, 45, 26, 20, 15, 20, 0, 43, 26, 37, 77]
Longueur de la partition : 183
```

Ici, nous convertissons chaque partition musicale en une séquence d'indices numériques grâce au dictionnaire char2idx. Cette étape est appelée vectorisation et permet de transformer les chaînes de caractères en données exploitables par le modèle LSTM.

Étape 5 : Padding des séquences

Pour former des batches, toutes les séquences doivent avoir la même longueur. Nous allons donc

ajouter artificiellement des espaces pour atteindre la longueur maximale : c'est du padding.

```
#Trouver la longueur maximale des partitions
# Longueurs de toutes les partitions
lengths = train_data['abc notation'].apply(len)

# Longueur maximale
max_length = lengths.max()
print("Longueur maximale des partitions :", max_length)
```

```
Longueur maximale des partitions : 2968
```

Fonction pour padding ou troncature :

```
def pad_or_truncate(text, max_length):

    if len(text) < max_length:
        # Ajouter des espaces
        return text + ' ' * (max_length - len(text))
    else:
        # Couper la fin
        return text[:max_length]

# Test avec la première partition
padded_first_song = pad_or_truncate(first_song, max_length)
print("Longueur après padding :", len(padded_first_song))
```

```
Longueur après padding : 2968
```

len(text) → longueur actuelle de la partition.

max_length - len(text) → nombre d'espaces à ajouter si la partition est trop courte.

Si la partition est trop longue, on la tronque pour ne garder que max_length caractères.

Conclusion

Dans ce chapitre, nous avons préparé les partitions musicales au format ABC pour qu'elles puissent être utilisées par un modèle de réseau de neurones récurrent. Nous avons d'abord identifié tous les caractères uniques présents dans le dataset, ce qui nous a permis de créer des correspondances entre caractères et indices (`char2idx` et `idx2char`). Ensuite, nous avons vectorisé les partitions en transformant chaque caractère en son indice correspondant. Enfin, pour garantir que toutes les séquences puissent être traitées en batchs, nous avons appliqué un padding afin d'uniformiser leur longueur.

Grâce à ces étapes, les données textuelles initiales sont désormais converties en représentations numériques normalisées, prêtes à être exploitées pour l'entraînement d'un modèle LSTM capable de prédire la prochaine note dans une partition.

Chapitre 4: Création du dataset PyTorch

Introduction:

Dans ce chapitre, nous passons de la préparation des données à leur exploitation dans un modèle de réseau de neurones récurrent. Nous allons d'abord créer un dataset PyTorch capable de gérer les séquences musicales, en séparant chaque partition en séquences d'entrée et cibles décalées, ce qui permet au modèle d'apprendre à prédire le prochain caractère à partir du contexte précédent.

Étape 1 : Préparation des données

Nous voulons rassembler tout le prétraitement en une fonction qui retourne les mappings et les données vectorisées prêtes à l'emploi.

```python
def preprocess_dataset(dataset):

    # Concaténer toutes les partitions pour trouver les caractères uniques
    all_text = ''.join(dataset['abc notation'].tolist())
    unique_chars = sorted(list(set(all_text)))

    # Mappings
    char2idx = {ch: idx for idx, ch in enumerate(unique_chars)}
    idx2char = {idx: ch for idx, ch in enumerate(unique_chars)}

    # Longueur maximale
    max_length = max(dataset['abc notation'].apply(len))

    # Vectorisation + padding
    vectorized_data = []
    for song in dataset['abc notation']:
        # Padding ou troncature
        if len(song) < max_length:
            song = song + ' ' * (max_length - len(song))
        else:
            song = song[:max_length]
        # Convertir en indices
        vectorized_data.append([char2idx[ch] for ch in song])

    return char2idx, idx2char, vectorized_data, max_length

# Exemple d'utilisation
char2idx, idx2char, train_vectorized, max_len = preprocess_dataset(train_data)

print("Nombre de caractères uniques :", len(char2idx))
print("Premiers indices de la première partition :", train_vectorized[0][:20])
print("Longueur maximale :", max_len)
```

```
Nombre de caractères uniques : 95
Premiers indices de la première partition : [56, 26, 17, 0, 44, 26, 17, 15, 24, 0, 45, 26, 20, 15, 20, 0, 43, 26, 37, 77]
Longueur maximale : 2968
```

Étape 2 : Dataset et DataLoader

On va maintenant créer la classe MusicDataset et préparer les DataLoaders pour le RNN.

Pourquoi créer un Dataset PyTorch ?

PyTorch fonctionne avec torch.utils.data.Dataset et DataLoader pour gérer :

Les batches (groupes d'exemples)

Le shuffle (mélange des données à chaque epoch)

Le prétraitement à la volée (par exemple vectorisation ou padding si nécessaire)

Donc même si tu as déjà tes partitions vectorisées, il faut mettre ces données dans un format que PyTorch comprend : c'est exactement ce que fait MusicDataset.

```python
#Classe MusicDataset
class MusicDataset(Dataset):
    def __init__(self, vectorized_data):
        """
        Args:
            vectorized_data (list of list of int): partitions vectorisées avec padding
        """
        self.data = vectorized_data

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

    def __getitem__(self, idx):
        seq = self.data[idx]
        # Séquence d'entrée : tout sauf le dernier caractère
        x = torch.tensor(seq[:-1], dtype=torch.long)
        # Séquence cible : tout sauf le premier caractère (décalée d'un pas)
        y = torch.tensor(seq[1:], dtype=torch.long)
        return x, y
```

C’est quoi un DataLoader ?

Un DataLoader est un outil PyTorch qui sert à :

donner les données au modèle pendant l’entraînement,

par petits paquets (batches), dans le bon format, automatiquement.

Tu peux le voir comme un serveur de données pour ton modèle.

```python
#Initialiser les DataLoaders
from torch.utils.data import DataLoader

batch_size = 8

# Dataset
train_dataset = MusicDataset(train_vectorized)
# Pour validation, si tu as déjà vectorisé val_data :
# val_char2idx, val_idx2char, val_vectorized, _ = preprocess_dataset(val_data)
# val_dataset = MusicDataset(val_vectorized)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
# val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

# Vérification d'un batch
x_batch, y_batch = next(iter(train_loader))
print("Shape x_batch :", x_batch.shape)  # (batch_size, sequence_length)
print("Shape y_batch :", y_batch.shape)
print("Premier batch x :", x_batch[0][:20])
print("Premier batch y :", y_batch[0][:20])
```

```
Shape x_batch : torch.Size([8, 2967])
Shape y_batch : torch.Size([8, 2967])
Premier batch x : tensor([56, 26, 17, 21, 20, 16, 24, 17,  0, 44, 26, 17, 15, 24,  0, 49, 26, 17,
        15, 20])
Premier batch y : tensor([26, 17, 21, 20, 16, 24, 17,  0, 44, 26, 17, 15, 24,  0, 49, 26, 17, 15,
        20, 29])
```

Class MusicRNN:

```python
class MusicRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(MusicRNN, self).__init__()

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

        # LSTM
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_size,
            batch_first=True
        )

        #  Couche dense pour la prédiction
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x):
        """
        x : (batch_size, sequence_length)
        """
        # Embedding
        x = self.embedding(x)
        # x : (batch_size, sequence_length, embedding_dim)

        # LSTM
        out, _ = self.lstm(x)
        # out : (batch_size, sequence_length, hidden_size)

        # Projection vers le vocabulaire
        out = self.fc(out)
        # out : (batch_size, sequence_length, vocab_size)

        return out
```

Conclusion

En conclusion, cette étape a permis de transformer le prétraitement en un format directement exploitable par PyTorch. La création de la classe `MusicDataset` assure que chaque séquence d'entrée est correctement alignée avec sa séquence cible, ce qui est essentiel pour l'apprentissage des relations temporelles dans les architectures RNN comme les LSTM. L'utilisation des `DataLoader` pour les ensembles d'entraînement et de validation permet non seulement de gérer efficacement les batches, mais aussi de simplifier l'itération pendant l'entraînement. Cette organisation des données constitue une base solide pour l'étape suivante, à savoir l'implémentation et l'entraînement du modèle LSTM pour la prédiction de notes musicales.

Chapitre 5: Implémentation du modèle

Introduction:

Dans ce chapitre, nous passons de la préparation des données à la construction et à l'entraînement du modèle de génération musicale. L'objectif est de concevoir un réseau de neurones récurrent capable de prédire la prochaine note d'une partition à partir des notes précédentes, en capturant les dépendances temporelles et structurelles propres à la musique.

Étape 1: Architecture du modèle

Le modèle que nous avons développé pour générer de la musique repose sur une architecture RNN de type LSTM. Cette architecture se compose de trois blocs principaux :

1. Couche d'Embedding

   Chaque caractère de la partition musicale est représenté par un index unique. La couche d'Embedding transforme ces indices discrets en vecteurs continus de dimension fixe. Ces vecteurs permettent au modèle de capturer des relations sémantiques et musicales entre les différents caractères (notes, rythmes, symboles).

2. Couche LSTM

   Les vecteurs issus de l'Embedding sont ensuite passés dans un LSTM (Long Short-Term Memory), qui est capable de mémoriser des informations sur de longues séquences et de modéliser les dépendances temporelles. Cette couche permet au modèle de prédire la note suivante en tenant compte de tout le contexte musical

précédent.

3. Couche Dense de sortie

   La sortie du LSTM est ensuite projetée via une couche dense (fully connected) qui transforme le vecteur en une distribution de probabilités sur tous les caractères possibles. Le caractère ayant la plus haute probabilité peut être sélectionné pour générer la suite de la partition (approche greedy) ou un échantillon peut être tiré selon une distribution probabiliste pour plus de diversité.

Étape 2: Réduction du dataset pour l'entraînement

Lors de l'expérimentation avec les hyperparamètres recommandés dans le TP (taille de batch de 256, embedding_dim = 256, hidden_size = 1024), nous avons rencontré une limitation mémoire qui empêchait l'entraînement sur l'ensemble complet des données.

Pour pallier ce problème, nous avons réduit la taille du dataset :

- Dataset d'entraînement : 10 000 partitions
- Dataset de validation : 1 000 partitions

Cette réduction permet de maintenir la faisabilité de l'entraînement tout en conservant une représentation significative des partitions pour l'apprentissage du modèle.

```
# DEBUG MODE — ONLY A FEW SONGS
train_data = train_data.iloc[:10000]
val_data   = val_data.iloc[:1000]
```

Étape 3: Boucle d'entraînement

```
num_training_iterations = 25
embedding_dim = 16
hidden_size = 64
learning_rate = 1e-3
```

[105]

```
def train_model(
    model,
    train_loader,
    val_loader,
    num_iterations,
    learning_rate
):
    print("Training started")
    model.to(device)

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

    train_losses = []
    val_losses = []

    for epoch in range(num_iterations):
        # ===== TRAIN =====
        model.train()
        train_loss = 0

        for x, y in train_loader:
            x, y = x.to(device), y.to(device)

            optimizer.zero_grad()
            outputs = model(x)

            loss = criterion(
                outputs.view(-1, outputs.size(-1)),
                y.view(-1)
            )

            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_loss /= len(train_loader)
        train_losses.append(train_loss)
```

```
        train_losses.append(train_loss)

        # ===== VALIDATION =====
        model.eval()
        val_loss = 0

        with torch.no_grad():
            for x, y in val_loader:
                x, y = x.to(device), y.to(device)

                outputs = model(x)
                loss = criterion(
                    outputs.view(-1, outputs.size(-1)),
                    y.view(-1)
                )

                val_loss += loss.item()

        val_loss /= len(val_loader)
        val_losses.append(val_loss)

        print(
            f"Epoch {epoch+1}/{num_iterations} | "
            f"Train Loss: {train_loss:.4f} | Val Loss: {val_loss:.4f}"
        )

    return train_losses, val_losses
```

```
model = MusicRNN(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim,
    hidden_size=hidden_size
).to(device)

train_model(
    model=model,
    train_loader=train_loader,
    val_loader=val_loader,
    num_iterations=num_training_iterations,
    learning_rate=learning_rate
)
```

Pour entraîner notre modèle `MusicRNN`, nous avons utilisé une boucle d'entraînement par epochs dans laquelle le modèle apprend à prédire le prochain caractère d'une séquence musicale à partir des caractères précédents. À chaque étape, les données d'entraînement sont envoyées au modèle, la loss est calculée, puis les gradients sont rétropropagés pour mettre à jour les poids grâce à l'optimiseur Adam. Ensuite, le modèle est

évalué sur les données de validation pour suivre la performance sans modifier ses paramètres. Cette approche permet de mesurer l'évolution de l'apprentissage et de détecter d'éventuels problèmes de surapprentissage. Pour des raisons de mémoire, nous avons réduit la taille du dataset, en utilisant 10 000 partitions pour l'entraînement et 1 000 pour la validation, tout en conservant la représentativité des données.

```
🚀 Training started
Epoch 1/25 | Train Loss: 2.0219 | Val Loss: 1.6045
Epoch 2/25 | Train Loss: 1.4919 | Val Loss: 1.4409
Epoch 3/25 | Train Loss: 1.3751 | Val Loss: 1.3490
Epoch 4/25 | Train Loss: 1.3060 | Val Loss: 1.3057
Epoch 5/25 | Train Loss: 1.2618 | Val Loss: 1.2606
Epoch 6/25 | Train Loss: 1.2302 | Val Loss: 1.2436
Epoch 7/25 | Train Loss: 1.2072 | Val Loss: 1.2195
Epoch 8/25 | Train Loss: 1.1899 | Val Loss: 1.2067
Epoch 9/25 | Train Loss: 1.1760 | Val Loss: 1.1876
Epoch 10/25 | Train Loss: 1.1645 | Val Loss: 1.1838
Epoch 11/25 | Train Loss: 1.1545 | Val Loss: 1.1748
Epoch 12/25 | Train Loss: 1.1462 | Val Loss: 1.1716
Epoch 13/25 | Train Loss: 1.1394 | Val Loss: 1.1653
Epoch 14/25 | Train Loss: 1.1327 | Val Loss: 1.1641
Epoch 15/25 | Train Loss: 1.1271 | Val Loss: 1.1546
Epoch 16/25 | Train Loss: 1.1231 | Val Loss: 1.1538
Epoch 17/25 | Train Loss: 1.1174 | Val Loss: 1.1511
Epoch 18/25 | Train Loss: 1.1130 | Val Loss: 1.1438
Epoch 19/25 | Train Loss: 1.1087 | Val Loss: 1.1394
Epoch 20/25 | Train Loss: 1.1051 | Val Loss: 1.1376
Epoch 21/25 | Train Loss: 1.1016 | Val Loss: 1.1367
Epoch 22/25 | Train Loss: 1.0983 | Val Loss: 1.1294
Epoch 23/25 | Train Loss: 1.0955 | Val Loss: 1.1247
Epoch 24/25 | Train Loss: 1.0919 | Val Loss: 1.1285
Epoch 25/25 | Train Loss: 1.0898 | Val Loss: 1.1259
```

L'entraînement du modèle montre une diminution progressive des pertes (loss) sur les ensembles d'entraînement et de validation au fil des 25 epochs. La loss de l'entraînement commence à 2.02 et descend jusqu'à 1.0898, tandis que la loss de validation passe de 1.60 à 1.1259. Cette tendance indique que le modèle apprend efficacement à prédire le prochain caractère dans les séquences musicales. On remarque également que la loss de validation suit globalement la loss d'entraînement, ce qui suggère que le

modèle ne surapprend pas de manière excessive et généralise correctement aux données de validation.

Conclusion

Dans ce chapitre, nous avons conçu et entraîné un modèle LSTM capable de prédire le prochain caractère dans une séquence musicale. L'utilisation d'une couche d'embedding a permis de transformer les indices de caractères en vecteurs continus, fournissant au LSTM une représentation riche et dense des partitions. Le LSTM, suivi d'une couche dense, a ainsi appris les dépendances temporelles et structurelles des séquences musicales. La boucle d'entraînement avec suivi de la loss sur les ensembles d'entraînement et de validation a permis d'optimiser les paramètres du modèle tout en évitant le surapprentissage grâce à des techniques comme l'early stopping. Ce travail constitue la base nécessaire pour la génération de nouvelles partitions musicales à partir d'un modèle entraîné.

Chapitre 6: Génération de musique

Introduction

Dans ce chapitre, nous utilisons le modèle LSTM entraîné pour générer de nouvelles partitions musicales. La génération repose sur le principe de prédiction séquentielle : à partir d'une séquence initiale donnée, le modèle prédit le caractère suivant, qui est ensuite réutilisé comme entrée pour l'étape suivante. Ce processus itératif permet de créer des séquences musicales de longueur arbitraire, reproduisant la structure et les motifs appris durant l'entraînement

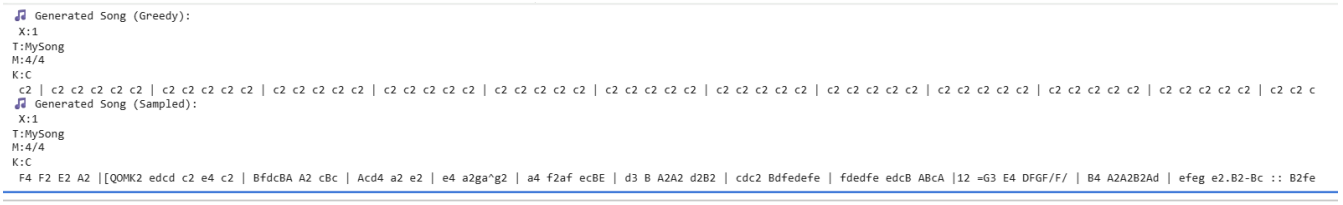
Étape 1: Generate_music

```python
def generate_music(model, start_sequence, char2idx, idx2char, length=200, device='cpu', sample=False, temperature=1.0):
    model.eval()
    input_seq = [char2idx[ch] for ch in start_sequence]
    input_tensor = torch.tensor(input_seq, dtype=torch.long).unsqueeze(0).to(device)

    generated = start_sequence

    hidden = None  # pour LSTM, on peut passer None pour commencer

    for _ in range(length):
        outputs = model(input_tensor)  # shape: [1, seq_len, vocab_size]
        last_logits = outputs[0, -1]  # on prend le dernier caractère

        if sample:
            probs = torch.softmax(last_logits / temperature, dim=-1)
            next_idx = torch.multinomial(probs, num_samples=1).item()
        else:
            next_idx = torch.argmax(last_logits).item()

        next_char = idx2char[next_idx]
        generated += next_char

        # mise à jour de input_tensor pour le prochain pas
        next_input = torch.tensor([[next_idx]], dtype=torch.long).to(device)
        input_tensor = torch.cat([input_tensor, next_input], dim=1)

    return generated
```

La fonction de génération permet de créer de nouvelles partitions au format ABC en utilisant le modèle entraîné. En mode greedy, le modèle

prédit à chaque étape le caractère le plus probable, produisant des séquences cohérentes mais parfois peu variées. L'échantillonnage avec température introduit de la diversité tout en respectant la structure musicale apprise.

Étape 2: Test



Pour générer de nouvelles partitions musicales, nous avons utilisé le modèle entraîné avec une séquence de départ contenant les métadonnées de la chanson (X, T, M, K). Deux stratégies de génération ont été testées : l'approche greedy, où le modèle choisit à chaque étape le caractère le plus probable, produisant des séquences cohérentes mais parfois peu variées, et l'approche échantillonnée avec température, qui sélectionne les caractères en fonction de leur probabilité, permettant ainsi d'obtenir des séquences plus créatives et diversifiées. Ces méthodes montrent comment le modèle peut prolonger une séquence initiale et générer de nouvelles mélodies au format ABC.

Conclusion

En résumé, le modèle entraîné est capable de générer de nouvelles partitions en prolongeant une séquence initiale, avec des variations selon la méthode de génération utilisée, démontrant ainsi son aptitude à modéliser les séquences musicales en notation ABC.