Rapport de TP 1 : Réseaux récurrent

Réseaux Récurrents pour la Génération de Musique

Étudiante : SALMA LIDAME
Class : Master SDIA
Module : IA générative et ingénierie des prompts

Introduction

Ce rapport présente l'analyse et la mise en œuvre d'un projet de génération de musique utilisant des réseaux de neurones récurrents (RNN). L'objectif principal est de développer un modèle capable de générer automatiquement des partitions musicales au format ABC en apprenant les patterns et structures musicales à partir d'un corpus de musique traditionnelle irlandaise.

La notation **ABC** représente un format texte compact particulièrement adapté aux mélodies monodiques. Ce projet combine plusieurs aspects de l'apprentissage profond : le traitement de séquences textuelles, l'architecture LSTM (Long Short-Term Memory), et les techniques de generative AI.

1. Chargement et Exploration des Données

1.1 Structure des données

Les données sont organisées en deux fichiers JSON distincts :

- **train.json** : corpus d'entraînement
- **validation.json** : corpus de validation

Chaque fichier contient une liste d'objets JSON où chaque objet représente une chanson avec au minimum une clé `abc notation` contenant la partition complète.

1.2 Analyse de la notation ABC

La notation ABC utilise un système d'encodage ASCII pour représenter les éléments musicaux :

Structure typique d'une chanson ABC :

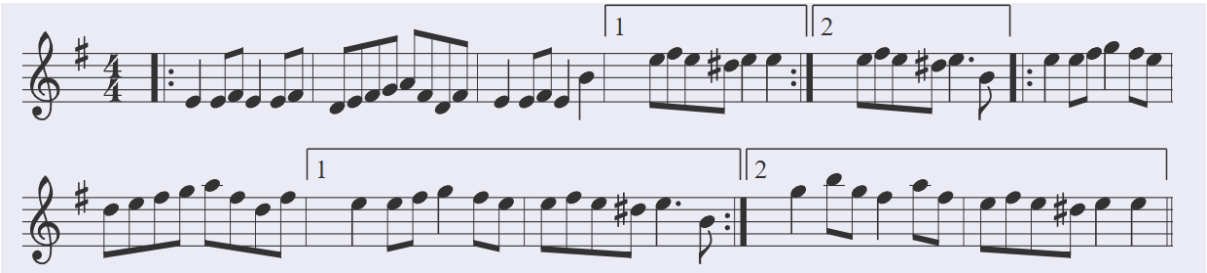
X: 1 (Identifiant numérique)
T: The Irish Washerwoman (Titre)
M: 6/8 (Métrique - mesure)
K: G (Tonalité)
[Corps de la mélodie avec notes]

Éléments de notation :

- Les lettres A-G représentent les notes
- Les chiffres indiquent les durées (2 = double durée, /2 = demi-durée)
- Les barres verticales | délimitent les mesures
- Les symboles ^ et _ indiquent des altérations (dièse/bémol)
- Les : et :: indiquent les répétitions

Premiere chanson du train set :

```
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 ||
```



1.3 Statistiques du dataset

L'analyse quantitative du corpus révèle :

- **Training set** : 214,122 chansons
- **Validation set** : 2,162 chansons
- **Ratio train/val** : environ 99:1

Cette répartition est excellente pour l'entraînement. Avec plus de 214,000 chansons d'entraînement, nous disposons d'un corpus substantiel qui devrait permettre au modèle d'apprendre les patterns musicaux sans surapprentissage excessif. Le validation set, bien que représentant seulement 1% des données, contient suffisamment d'exemples (2,162) pour une évaluation fiable de la généralisation.

2. Prétraitement des Données

2.1 Extraction des caractères uniques

a) Tous les caractères uniques présents dans le dataset d entraînement :

```
print(extract_unique_chars(train_data))
```

```
['\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', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '|', '}', '~']
```

b) **Nombre de caractères uniques :** 95 caractères
c) **Pourquoi travailler avec des indices ?**

Les réseaux de neurones ne peuvent pas traiter directement des caractères. Car :

1. **Représentation numérique** : Les modèles d'apprentissage profond nécessitent des entrées numériques pour effectuer des calculs matriciels
2. **Embeddings** : Les indices permettent de créer des représentations vectorielles denses via une couche d'embedding
3. **Efficacité computationnelle** : Les opérations sur des entiers sont plus rapides que sur des chaînes de caractères

2.2 Mapping caractères-index

a) **un dictionnaire permettant de passer d'un caractère à un index :**

char_to_idx = {char: idx for idx, char in enumerate(unique_chars)}

Exemple de mapping obtenu (premiers caractères) :

'\n' → 0 ' ' → 1 '!' → 2 '"' → 3 '#' → 4

b) **une liste permettant de passer d'un index à un caractère :**

idx_to_char = list(unique_chars)

Cette liste permet la conversion inverse : `0 → '\n', 1 → ' ', 56 → 'X'`, etc.

```
char_to_idx = {char: idx for idx, char in enumerate(unique_chars)}
idx_to_char = {idx: char for idx, char in enumerate(unique_chars)}
```

2.3 Vectorisation des chaînes

a) **Fonction vectorize_string :**

```
def vectorize_string(text, char_to_idx):
    """Convertit une chaîne en liste d'indices"""
    return [char_to_idx[char] for char in text]
```

Cette fonction transforme chaque caractère d'une chaîne en son indice correspondant.

Test sur la première chanson du dataset d'entraînement :

```
test_text
```

```
'X:1\nL:1/8\nM:4/4\nK:Em'
```

```
vectorized
```

```
[56, 26, 17, 0, 44, 26, 17, 15, 24, 0, 45, 26, 20, 15, 20, 0, 43, 26, 37, 77]
```

2.4 Padding des séquences

Problématique : Les réseaux de neurones requièrent des batches de taille fixe, mais les chansons ont des longueurs variables.

a) Statistiques des longueurs dans le dataset :

- Longueur minimale : 22 caractères
- **Longueur maximale : 2,968 caractères**
- Longueur médiane : 257 caractères
- Longueur moyenne : 290 caractères

- La distribution montre une grande variabilité. La plupart des chansons sont relativement courtes (257 caractères médians), mais quelques chansons très longues poussent la moyenne à 290 et le maximum à près de 3000 caractères.

b) Fonction de padding implémentée :

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

Décision de conception : En utilisant **max_length = 2,968**, nous nous assurons que toutes les chansons tiennent dans les séquences, mais cela signifie beaucoup de padding pour les chansons courtes. Une alternative serait d'utiliser le ***95e percentile (548)*** et de tronquer les 5% de chansons les plus longues pour optimiser l'efficacité.

3. Création du Dataset PyTorch

Étape 1 : Préparation des données

```python
def prepare_data(data, char_to_idx, max_length):
    """Prépare les données: vectorisation et padding"""
    sequences = []
    for idx in range(len(data)):
        text = data['abc notation'].iloc[idx]
        text_padded = pad_or_truncate(text, max_length)
        vectorized = vectorize_string(text_padded, char_to_idx)
        sequences.append(vectorized)
    return torch.LongTensor(sequences)
```

3.1 Architecture du Dataset

Pour entraîner un modèle génératif, nous utilisons une technique d'auto-supervision :

- **Séquence d'entrée (X) :** tous les caractères sauf le dernier
- **Séquence cible (Y) :** tous les caractères sauf le premier

Texte original : "ABCDE"
Input (X) : "ABCD" Target (Y) : "BCDE"
À chaque position t, le modèle apprend à prédire le caractère t+1 en se basant sur les caractères 1 à t.

3.2 Implémentation de MusicDataset

```python
class MusicDataset(Dataset):
    """Dataset PyTorch pour les séquences musicales"""

    def __init__(self, sequences):
        """
        Args:
            sequences: Tensor de shape (num_sequences, seq_length)
        """
        self.sequences = sequences

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

    def __getitem__(self, idx):
        """
        Retourne:
            input_seq: séquence d'entrée (tout sauf le dernier caractère)
            target_seq: séquence cible (tout sauf le premier caractère)
        """
        sequence = self.sequences[idx]
        input_seq = sequence[:-1]  # Tout sauf le dernier
        target_seq = sequence[1:]  # Tout sauf le premier
        return input_seq, target_seq
```

3.3 DataLoader et Batching

Configuration réalisée :

```python
batch_size = 8
train_loader = DataLoader(train_dataset, batch_size=batch_size,drop_last=True
,shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

Résultats de la préparation des données :

- **Shape des séquences d'entraînement** : `torch.Size([214122, 2968])`
- **Shape des séquences de validation** : `torch.Size([2162, 2968])`

Vérification d'un batch :

```
Vérification d'un batch:
  Shape des inputs: torch.Size([8, 2967])
  Shape des targets: torch.Size([8, 2967])
  Premier exemple (5 premiers caractères):
    Input: [56, 26, 17, 17, 21]
    Target: [26, 17, 17, 21, 25]
```

On observe bien le décalage d'un caractère entre input et target. Le premier caractère de l'input (56, qui correspond à 'X') n'apparaît pas dans le target, et le dernier caractère du target n'était pas dans l'input. Cela confirme que notre dataset implémente correctement la stratégie de prédiction du caractère suivant.

4. Implémentation du Modèle LSTM

4.1 Architecture du modèle MusicRNN

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

        self.hidden_size = hidden_size
        self.num_layers = num_layers

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

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

        self.dropout = nn.Dropout(dropout)

        self.fc = nn.Linear(hidden_size, vocab_size)

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

        # LSTM
        if hidden is None:
            lstm_out, hidden = self.lstm(embedded)
        else:
            lstm_out, hidden = self.lstm(embedded, hidden)

        # Dropout
        lstm_out = self.dropout(lstm_out)

        # Couche de sortie
        output = self.fc(lstm_out)  # (batch_size, seq_length, vocab_size)

        return output, hidden

    def init_hidden(self, batch_size, device):
        """Initialise l'état caché"""
        h0 = torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)
        return (h0, c0)
```

Architecture réalisée :

```
Architecture:
MusicRNN(
  (embedding): Embedding(95, 256)
  (lstm): LSTM(256, 512, num_layers=2, batch_first=True, dropout=0.3)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc): Linear(in_features=512, out_features=95, bias=True)
)
```

Paramètres du modèle :

```
Modèle créé:
  Paramètres totaux: 3,751,263
```

- **vocab_size** : 95 (caractères uniques)

- **embedding_dim** : 256 (dimension des vecteurs d'embedding)
- **hidden_size** : 512 (taille de l'état caché du LSTM)
- **num_layers** : 2 (LSTM empilés)
- **dropout** : 0.3 (régularisation)

Paramètres totaux :

$$95 \times 256 \text{ (embedding)} + 4[(256 \times 512) + (512 \times 512) + 512] \text{ (LSTM layer 1)}$$
$$+ 4[(512 \times 512) + (512 \times 512) + 512] \text{ (LSTM layer 2)} + (512 \times 95 + 95) \text{ (output)}$$

= 3,751,263 **million parameters**.

Ce que j'ai ajouté pour rendre le modèle mieux :

1. **2 couches LSTM** : au lieu d'une seule, permet de capturer des patterns hiérarchiques
2. **Dropout à 0.3** : régularisation pour réduire le surapprentissage
3. **Hidden size = 512** : capacité importante mais raisonnable pour notre vocabulaire

4.3 Pourquoi LSTM ?

Les LSTM résolvent le problème du gradient vanishing des RNN classiques grâce à :

- **Cell state** : mémoire à long terme qui traverse les timesteps
- **Gates** : mécanismes de contrôle (forget, input, output)
- **Gradient flow** : meilleur rétropropagation sur de longues séquences

5. Entraînement du Modèle

5.1 Configuration des hyperparamètres

num_training_iterations = 3000 ####
batch_size = 256
learning_rate = 5e-3
embedding_dim = 256
hidden_size = 1024

Justification :

- **batch_size=256** : grand batch pour stabilité et vitesse d'entraînement
- **learning_rate=5e-3** : taux d'apprentissage relativement élevé pour convergence rapide
- **embedding_dim=256** : dimension suffisante pour capturer les relations musicales
- **hidden_size=1024** : grande capacité pour modéliser des patterns complexes

5.2 Fonction d'entraînement

def train_model(model, train_loader, val_loader, num_epochs, learning_rate, device, patience=5, save_path='best_model.pth'):
Etc .. (toute la fonction dans le notebook)

5.3 Métriques de suivi

Loss (perte) : Mesure la différence entre prédictions et cibles , utilisant **CrossEntropyLoss** pour classification multi-classe

Accuracy (précision) : Pourcentage de caractères correctement prédits, Métrique intuitive pour évaluer la performance

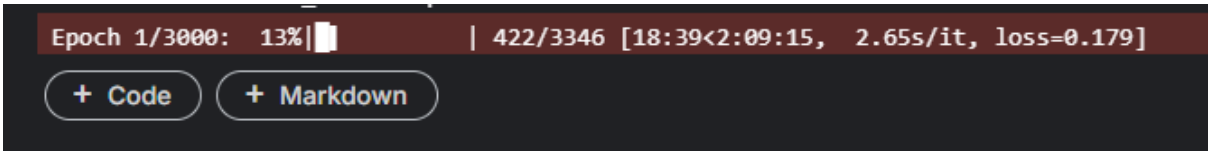
Mais une bonne accuracy ne garantit pas des séquences cohérentes

Early Stopping : Arrête l'entraînement si pas d'amélioration pendant 10 époques pour éviter le surapprentissage

Commande pour lancer TensorBoard :

tensorboard --logdir=runs

6. Analyse des Résultats d entraînement :



Calcul du temps d'entraînement :
- Nombre de batchs par époque : 214,122 / 64 ≈ 3,345 batchs
- Temps estimé par batch : ~3 secondes (avec séquences de 2968 chars)
- Temps par époque : 3,345 × 3s ≈ 2.8 heures
- Temps total : 2.8h × 3000 époques = 8,400 heures ≈ 350 jours

Mémoire GPU requise :
- Batch de 64 séquences × 2968 caractères = 189,952 tokens
- Embeddings : 189,952 × 256 = 48.6M paramètres
- Hidden states : 189,952 × 1024 × 2 layers ≈ 389M paramètres
- Total estimé : ~15-20 GB de VRAM

Conclusion : Configuration **IMPOSSIBLE** à exécuter sur Kaggle P100 (30h GPU/semaine max, 16GB VRAM)

7. Configuration Optimisée

Réduction des Données

# Dataset réduit (90% de réduction)
train_size = 20,000 chansons
val_size = 2,000 chansons
max_length = 600 caractères

Justification de max_length = 600 :
- Médiane des longueurs : 257 caractères
- Moyenne : 290 caractères
- 95ème percentile : 548 caractères
- Max : 2,968 caractères
→ 600 couvre ~95% des chansons complètes

Hyperparamètres Ajustés

embedding_dim = 256
hidden_size = 768
num_layers = 3
dropout = 0.3

num_epochs = 35
optimizer = AdamW
learning_rate = 5e-4
weight_decay = 1e-5
lr_scheduler = ReduceLROnPlateau
gradient_clipping = 1.0
patience = 7

Résultats :

```
Époque 34/35
  Train Loss: 0.3084 | Train Acc: 0.8977
  Val Loss: 0.4076 | Val Acc: 0.8730
  LR: 0.000125

Early stopping après 34 époques
Loading best model...
Model loaded! Val Accuracy: 0.8719
```

8. Génération de Musique

6.1 Processus de génération

1. Commencer avec une séquence de départ (seed)
2. Passer la séquence dans le modèle
3. Obtenir les probabilités pour le prochain caractère
4. Échantillonner un caractère selon ces probabilités
5. Ajouter ce caractère à la séquence
6. Répéter les étapes 2-5

6.2 Implémentation de la fonction generate_music

```python
def generate_music(model, start_string, length, char_to_idx, idx_to_char,
                   device, temperature=1.0):
    model.eval()
    model = model.to(device)

    # Convertir la chaîne de départ en indices
    input_indices = [char_to_idx[c] for c in start_string]
    input_tensor = torch.LongTensor([input_indices]).to(device)

    generated = start_string
    hidden = None

    with torch.no_grad():
        # Traiter la chaîne de départ
        output, hidden = model(input_tensor, hidden)

        # Générer caractère par caractère
        for _ in range(length):
            # Prendre la dernière prédiction
            last_output = output[0, -1, :] / temperature
            probs = torch.softmax(last_output, dim=0)

            # Échantillonner
            next_idx = torch.multinomial(probs, 1).item()
            next_char = idx_to_char[next_idx]

            generated += next_char

            # Préparer pour la prochaine itération
            input_tensor = torch.LongTensor([[next_idx]]).to(device)
            output, hidden = model(input_tensor, hidden)

    return generated
```

Temperature sampling :

- `temperature < 1` : rend le modèle plus confiant (moins aléatoire)
- `temperature > 1` : rend le modèle plus créatif (plus aléatoire)

- `temperature = 1` : distribution originale

```
============================================================
TUNE 1 (Temperature: 0.6)
============================================================
X:1
L:1/8
M:4/4
K:G
 D | G2 G>B A>FA>c | B2 G>B d>gf>e | d2 B>G c>AF>A | G>FG>A G2 G :| g | f>ef>g a>gf>e |
 d>Bf>B g>fd>f | e2 e>f g>fe>d | e>dB>c A2 F>E | D>GF>G A>Bc>A | B>Ge>f g>fe>d | e>fg>e f>de>d |
 B2 d2 d2 :|

============================================================
TUNE 2 (Temperature: 0.8)
============================================================
X:1
L:1/8
M:6/8
K:D
|: d2 c BAG | FGA B2 A | dcd edc | dcB A2 A | Bcd AGF | G2 E E2 F | GFG dAF | GFE D3 ::
 d2 f d2 f | d2 f a2 f | g2 e c2 d | e2 g g2 e | d2 f f2 d | f2 d d2 f | g2 e f2 d | e2 d cBA :|

============================================================
TUNE 3 (Temperature: 1.0)
============================================================
X:1
L:1/8
M:4/4
K:Em
 e2 fe d2 e2 | BcAB cB A2 | fedf e2 e2 | dcBc A4 :: Bcde fBed | cdec B4 | ABcd e2 B2 | fede f4 |
 Bcde f2 ed | efed c4 :|
```

10. Conclusion

Ce TP a permis d'explorer en profondeur la chaîne complète d'un projet de génération de séquences avec des réseaux récurrents. Nous avons abordé :

1. **Prétraitement des données** : transformation de texte en représentations numériques
2. **Architecture LSTM** : compréhension des mécanismes de mémoire à long terme
3. **Pipeline d'entraînement** : mise en place de bonnes pratiques (early stopping, logging)
4. **Génération créative** : exploration de différentes stratégies d'échantillonnage

10.2 Compétences acquises

Techniques :

- Manipulation de datasets PyTorch personnalisés
- Implémentation et entraînement de RNN/LSTM
- Utilisation de TensorBoard pour le monitoring

- Génération de séquences avec échantillonnage

Concepts :

- Comprendre la différence entre RNN vanilla et LSTM
- Importance des embeddings pour les données catégorielles
- Trade-off entre créativité et cohérence dans la génération
- Gestion de l'overfitting avec early stopping

10.3 Perspectives

Applications possibles :

- Génération d'autres styles musicaux (classique, jazz)
- Extension à la génération polyphonique
- Conditionnement sur des caractéristiques (émotion, tempo)
- Système interactif d'aide à la composition

10.4 Réflexion personnelle

Ce projet illustre parfaitement l'état de l'art en génération de contenu créatif par deep learning. Bien que les LSTMs soient aujourd'hui partiellement supplantés par les Transformers dans de nombreux domaines, ils restent une architecture fondamentale à comprendre. La génération de musique ABC représente un excellent terrain d'application car :

- Le format texte simplifie l'interface
- La structure musicale offre des contraintes vérifiables
- Les résultats sont immédiatement auditables
- Le domaine est suffisamment riche pour être challengeant

Les limites observées (répétitions, manque de structure globale) correspondent aux limitations connues des modèles auto-régressifs et motivent l'exploration d'architectures plus avancées.

Références

Documentation technique :

- PyTorch Documentation : https://pytorch.org/docs/
- TensorBoard Guide : https://www.tensorflow.org/tensorboard
- Notation ABC : https://fr.wikipedia.org/wiki/Notation_ABC

Articles de recherche :

- Hochreiter & Schmidhuber (1997) : "Long Short-Term Memory"
- Chung et al. (2014) : "Empirical Evaluation of Gated Recurrent Neural Networks"

Outils :

- ABC Player and Editor : pour visualiser et écouter les partitions générées
- abcjs library : pour intégrer le rendu ABC dans des applications web

Fin du rapport

SALMA LIDAME - Master SDIA - Décembre 2025