Projet Deep Learning (partie A)

Pierre Falez

10 mai 2021

1 Objectif

Dans ce TP nous allons créer un modèle capable de générer de la musique. Pour cela, nous n'utiliserons pas directement des fichiers audio, mais un format qui représente directement les notes jouées. Utiliser des données plus abstraites (de plus hauts niveaux) apporte généralement des avantages. Dans notre cas, cela permet d'une part d'éviter de s'occuper de toute la partie de traitement du signal audio, et d'autre part de réduire le poids des données (les jeux de données peuvent peser plusieurs téra-octects) mais aussi la complexité des modèles.

Le format utilisé est le MIDI *(Musical Instrument Digital Interface)* qui permet de jouer directement les notes de chaque instrument. Pour écoutez un fichier MIDI, vous pouvez utiliser la fonctionnalité fluidsynth de VLC, ou directement fluidsynth : `https://www.fluidsynth.org/`. Des outils de visualisation de fichier MIDI existe également : `https://github.com/kosua20/MIDIVisualizer`

Afin d'arriver à notre objectif, nous allons dans un premier temps mettre en place le pré-traitement des données, afin de transformer les fichiers MIDI vers une représentation compatible avec nos modèles. Ensuite, nous entraînerons des réseaux de neurones sur ces données afin de les rendre capables de prédire la note suivant une séquence. Finalement, nous utiliserons le modèle afin de générer des séquences de notes à partir d'une séquence aléatoire.

2 Pré-requis

Il est fortement conseillé d'utiliser une version compatible GPU de Pytorch afin d'entraîner les modèles dans des temps raisonnables. Cependant, vous ne serez en aucun cas pénalisés si vous ne disposez pas du matériel nécessaire. La notation de ce TP se fera sur la qualité et la lisibilité du code rendu.

Pour obtenir la version CUDA (pour les cartes Nvidia) ou RoCM (pour les cartes AMD), rendez-vous à l'adresse suivante : `https://pytorch.org/get-started/locally/`

Les bibliothèques python suivantes sont requises pour le bon fonctionnement du TP (vous pouvez utiliser *Pip* pour les installer) :

— torch
— numpy
— pretty_midi
— tensorboard

Enfin, nous aurons besoin d'un jeu de données MIDI afin d'entraîner les modèles. Nous utiliserons Maestro 3.0.0, qui regroupe un nombre conséquent de pièces de piano classique. Ce jeu de données est disponible à l'adresse suivante (téléchargez seulement la version midi) : `https://magenta.tensorflow.org/datasets/maestro#v300`

3 Rendu

Vous rendrez une archive zip ou tar.gz contenant l'ensemble des fichiers python, mais également les graphiques de *tensorboard* des métriques obtenues lors de l'entraînement.

4 Manipulation des données MIDI

Les données sont la première chose dont il faut s'occuper avant de pouvoir entraîner un réseau de neurones. Nous allons donc commencer par traiter nos fichiers MIDI afin de les rendre exploitables par nos modèles. Dans le fichier *midi.py*, vous disposez de deux fonctions fournies :

— **midi_2_piano_roll(path, frequency)**, qui lit le fichier MIDI *path* et retourne une représentation matricielle (appellée *piano roll*, voir figure1) de la première piste d'instruments). Celle-ci est échantillonnée à la fréquence spécifiée par le paramètre *frequency* (F_s). Cette matrice est de taille $[n, t]$, avec $n = 128$ la dimension des notes (voir Annexe A) et t la dimension temporelle ($t = s * F_s$, avec s la durée en secondes du fichier MIDI)
— **piano_roll_2_midi(data, frequency)**, qui effectue l'opération inverse, en transformant une matrice vers un objet MIDI.

Vous disposez également du code permettant de tester vos fonctions en appelant le script directement : *python midi.py <maestro-root>*

Nous allons utiliser des méthodes similaires au traitement du langage naturel (*NLP*) pour travailler sur les partitions. Au lieu d'utiliser des mots, nous aurons un vocabulaire constitué de notes ou d'accords (plusieurs notes jouées simultanément). Une première étape consiste à transformer les matrices *piano roll* vers une séquence de mots. À chaque temps t, il faut donc récupérer la liste des notes jouées afin de constituer l'étiquette (voir figure 1). Lorsque aucune note n'est joué, nous utilisons une chaine de caractères vide. Lorsqu'une note est jouée, nous utilisons l'index de la note. Enfin, lorsque

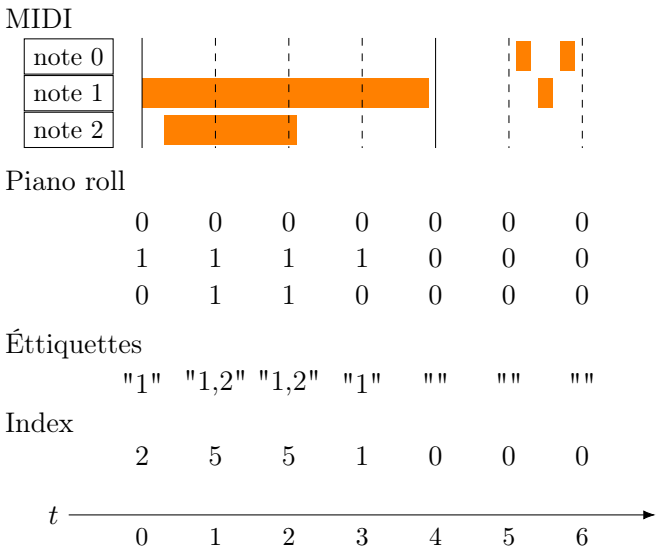


FIGURE 1 – Étapes de pré-traitements. Dans cet exemple nous disposons d'un fichier MIDI avec 3 notes d'une durée de 1,5 secondes. Si nous utilisons une fréquence d'échantillonnage $f_s = 4$, nous obtenons une matrice $[3, 7]$. Cette matrice est transformée en liste d'étiquette puis d'index de taille 7

plusieurs notes sont jouées en même temps, nous concaténons les index des notes jouées avec une virgule, en triant les index par ordre croissant. Par exemple :

- L'étiquette "28" signifie que seul la note 28 est jouée au temps t
- L'étiquette "12, 42, 113" signifie que les notes 12, 42 et 113 sont jouées au temps t
- L'étiquette "" signifie qu'aucune note n'est jouée au temps t

Question 1/ Complétez la fonction **notes_2_label(notes)** qui, à partir de la liste des notes jouées au temps t, retourne l'étiquette correspondante. La valeur retournée doit être une chaîne de caractéres.

Question 2/ Complétez la fonction **label_2_notes(label)**, qui effectue l'opération inverse, en retournant la liste des notes jouées à partir d'une étiquette. La valeur retournée doit être une liste d'entiers.

Question 3/ Complétez la fonction **piano_roll_2_notes(piano_roll)** qui, à partir d'une matrice *piano roll* (au format numpy), retourne la liste des étiquettes de notes jouées.

Question 4/ Complétez la fonction **notes_2_piano_roll(notes)**, qui effectue l'opération inverse en retournant une matrice numpy.

5 Pré-traitement des données

Maintenant que nous avons les outils nécessaires pour transformer les données vers notre vocabulaire, nous devons appliquer la transformation sur le jeu de données. Pour ceci, nous allons écrire un script *generate_ data.py* qui va se charger de faire les traitements nécessaires avant l'entraînement. Utilisez-le de la manière suivante : *python generate_ data.py <dataset-path>*. Vous pouvez utiliser l'option *–limit n* afin de tester le script sur les n premiere entrées pour des questions de temps d'exécution. Ce script va générer les fichiers *train.npy*, *validation.npy*, *test.npy* contenant les données transformées.

Maestro dispose d'un fichier CSV à la racine contenant les méta-informations sur les fichiers MIDI. Vous disposez d'une fonction **load_data** qui charge ce fichier CSV et retourne la liste des morceaux. La liste d'objets retournés contient les champs suivants :

— *notes* contient la séquence de notes du morceau.
— *split* permet de savoir à quelle partie du jeu de données appartient le morceaux (*train*, *validation* ou *test*)

Le nombre de combinaisons de notes (et donc la taille potentielle du vocabulaire) étant énorme, nous allons utiliser qu'une sous-partie des étiquettes présentes dans le jeu d'entrainement. Pour cela, nous ne garderons que les étiquettes présentes au minimum m fois dans le jeu d'entraînement. Pour les étiquettes non retenue, une heuristique basique va nous permettre de lui attribuer une autre étiquette : nous retirons la note la plus basse jusqu'a trouver une étiquette valide (dans le cas contraire, nous utilisons l'étiquette vide). Par exemple, si l'étiquette "23,45,56" n'est pas validé, nous testerons l'étiquette "45,56". Si cette dernière n'est toujours pas valide, nous testerons avec l'étiquette "56". Finalement, nous utiliserons l'étiquette "" si aucune des étiquettes précédentes n'est valide.

Question 5/ Complétez la fonction **get_all_labels(data)** afin de lister l'ensemble des étiquettes du jeu de données d'entrainement. Veillez à ne pas supprimez les doublons lorsqu'un même mot apparait plusieurs fois dans le jeu de données.

Question 6/ Complétez la fonction **make_label_list(labels, threshold)** qui va compter les occurrences des étiquettes et va garder uniquement ceux utilisé plus de *threshold* fois. Vous pouvez vous servir de la classe *Counter* pour cela. Cette fonction retourne la liste des étiquettes restantes sans doublons.

La position de l'étiquette dans cette liste déterminera son indice. Cette liste est sauvegardé dans le fichier *labels* pour retrouver l'étiquette à partir de sont indice. les fonctions **make_label_2_index_dict(labels)** et **make_index_2_label_dictt(labels)** permettent de créer les dictionnaires capable de transformer une étiquette vers son indice, et un indice vers sont étiquette respectivement.

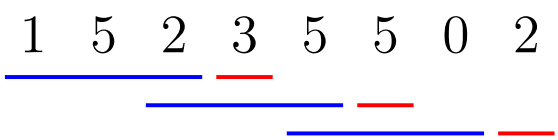


FIGURE 2 – Exemple d'une entrée généré à partir d'une séquence de taille 8. Ici, 3 entrées sont générés avec un enjambement *stride* de 2 et une longeur de séquence *sequence_length* de 3. Pour chaque entrée, la séquence utilisée pour X est souligné en bleu et le nombre à prédire est souligné en rouge.

Question 7/ Complétez la fonction **note_2_index(note, labels_2_index)** qui à partir d'une étiquette et du dictionnaire de conversion étiquette-indice, retourne l'indice correspondant. Cette fonction se charge d'appliquer l'heuristique si l'étiquette n'est pas valide (et donc n'appairait pas dans le dictionnaire).

Question 8/ Complétez la fonction **note_2_index(note, labels_2_index)** qui à partir d'une étiquette et du dictionnaire de conversion étiquette-indice, retourne l'indice correspondant. Cette fonction se charge d'appliquer l'heuristique si l'étiquette n'est pas valide (et donc n'appairait pas dans le dictionnaire).

Question 9/ Complétez les fonctions **notes_2_index(sequence, labels_2_index)** et **index_2_notes(sequence, index_2_labels** qui respectivement transforme une liste d'étiquettes vers une liste d'indices et une liste d'indices vers une liste d'étiquettes à partir des dictionnaires.

6 Générateur de données

Il nous reste encore à écrire une classe capable de charger les données et de les mettre en forme pour notre tâche d'entraînement. Pour ceci, nous allons implémenter la classe **MaestroDataset**. Celui-ci va lire la liste des séquences contenues dans un des fichiers précédemment crées. Dans chaque fichier, plusieurs entrées (X, y) vont être utilisé en fonction du paramètre d'enjambement *stride* et de la longeur de la séquence X *sequence_length* (voir figure 2). Une entrée va donc être constituée de X, de taille *[sequence_length]* mots et de y contenant le mot suivant.

Question 10/ Complétez le constructeur de la classe qui va charger le jeu de données *data_file* grâce à la fonction *generate_data.read_data*. Construisez une liste qui va contenir la liste des entrées de ce fichier. Chaque element de la liste contient un tuple avec l'indice du fichier et la position de départ de la séquence X.

Question 11/ Complétez la méthode **__len__(self)** qui retourne le nombre d'entrée du jeu de données.

Question 12/ Complétez la méthode **__getitem__(self, idx)** qui

retourne l'entrée numéro *idx*. La valeur retournée est un tuple contenant X (un tenseur d'entier de taille [*sequence_length*]) et y (un tenseur d'entier contenant un scalaire).

7 Création du modèle

Dans le fichier *lstm.py*, nous allons créer notre modèle. Comme son nom l'indique nous utilisons une LSTM pour prédire une note à partir d'une séquence d'entrée. Ce réseau prendra en entrée une matrice d'indice de taille *[batch_size, sequence_length]* Nous utiliserons l'architecture suivante :

- Un *Embedding (nn.Embedding)*, qui transforme les indices d'entrées vers une représentation dense de taille *embeding_size*.
- Une *LSTM (nn.LSTM)*. Cette aura *num_layers* couches cachées de taille *hidden_size*.
- Une couche de *dropout (nn.Dropout)* avec une probabilité *dropout*.
- Une couche dense avec *class_number* neurones de sortie.

Question 13/ Completez la classe **LSTMClassifier**, afin de déclarer notre modèle. Vous pouvez tester votre modèle en appelant directement le fichier *lstm.py*

8 Entraînement et validation du modèle

Avant de nous lancer dans l'écriture de la procédure d'entrainement, nous allons écrire les fonctions nécessaires pour valider et tester notre modèle.

Question 14/ Dans le fichier *eval.py*, complétez la fonction **run(model, criterion, dataset, device)** qui va se charger de calculer le coût moyen (*loss*), mais également les taux top1 et top5 de prédiction correcte sur le jeu de données *dataset*. Pour cela utilisez la fonction **top_k_error**, avec *topk=(1, 5)*. Vous retournerez ces valeurs dans une tuple.

Maintenant que tout est en place, nous pouvons enfin attaquer l'entraînement ! Vous disposez du fichier *train.py* qui contient la structure du code.

Question 15/ Complétez la fonction **train_epoch(model, criterion, optimizer, dataset, device)** qui effectue l'entraînement pour une epoch. Cette fonction retourne le coût moyen (*loss*) durant cette epoch.

Nous allons utiliser *tensorboard*, afin de visualiser l'évolution de l'entraînement. Vous pouvez le lancer en utilisant la commande suivante dans le répertoire où se situent les fichiers python : *tensorboard –logdir=runs*

Une fois le script lancé, vous devriez avoir accès au modèle utilisé (onglet *Graph*), mais aussi aux différentes métriques (onglet *Scalar*). **Question 16**/ Lancez l'entraînement, et sauvegarder les graphs. Vous les ajouterez à l'archive rendu. Vous pouvez ajuster les paramètres (stride du jeu de données, nombre d'epoch, taille et nombre de couches, etc...) si votre ordinateur prend trop de temps à compléter l'entraînement.

Question 17/ Complétez le fichier *test.py* afin de calculer les taux de prédiction top1 et top5 sur le jeu de données de test.

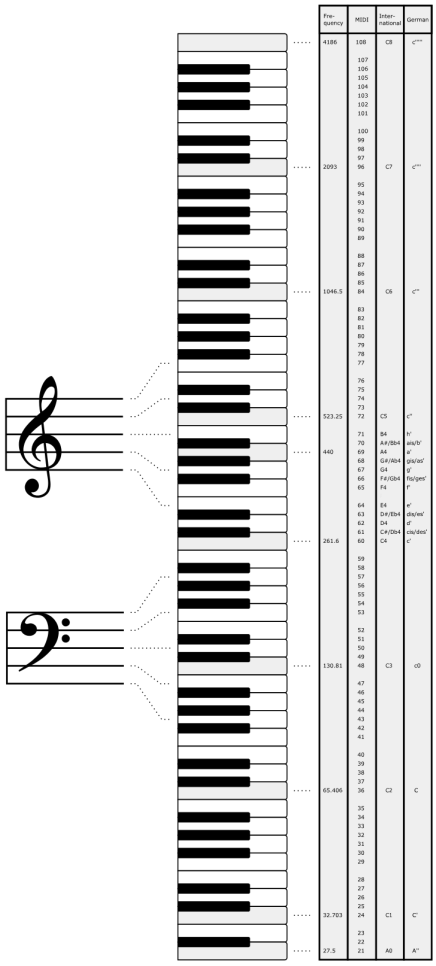
9 Génération de séquence

Finalement, nous pouvons maintenant générer de nouveau morceau grâce a notre modèle. Pour ce faire, nous allons générer une séquence initiale aléatoirement, puis itérativement, nous utiliserons le modèle pour prédire les notes suivantes.

Question 18/ Complétez la fonction **generate_rand_full(seq_len, labels)** qui génère une séquence de taille *seq_len* avec des notes aléatoire.

Question 19/ Complétez la fonction **generate_rand_one** qui génère une séquence avec seulement la dernière note aléatoire. Les autres notes sont inialisées avec la note vide ("").

Question 20/ Complétez le script afin de générer les notes. Pour ce faire, il faut calculer la distribution de probabilité des notes en appliquant la fonction softmax *nn.Softmax* à la sortie du modèle. Vous utiliserez la fonction *random.choices* pour choisir une note aléatoirement en fonction de la distribution.



A Correspondances des notes dans le format MIDI