

M2 Logiciel sûrs - IA

Music Generation

**Réalisé par :
Merzeg Ramzi
Khitous Rania**

lien de notre code source sur git :

https://github.com/ramzimerzeg/Music_Generation

exemples de musiques que nous avons générés :

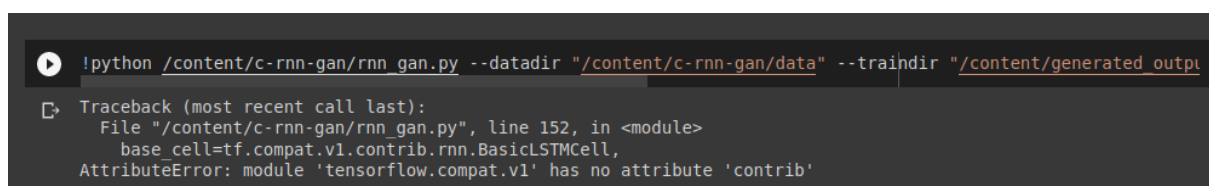
https://github.com/ramzimerzeg/Music_Generation/tree/main/generated_music

Problème rencontrés :

on a essayé d'exécuter le code source git cité dans le fichier Music_generation.pdf :

[GitHub - danieldjohnson/biaxial-rnn-music-composition: A recurrent neural network designed to generate classical music.](https://github.com/danieldjohnson/biaxial-rnn-music-composition)

mais on a rencontré plusieurs problèmes. Comme le code source date de plus de 5 ans, on a essayé de mettre à jours la version python et les versions des librairies mais après plusieurs modifications on s'est arrêté à cause d'une erreur qu'on a malheureusement pas pu corriger sur tensorflow ou le code utilise un version très ancienne de tensorflow.



```
!python /content/c-rnn-gan/rnn_gan.py --datadir "/content/c-rnn-gan/data" --traindir "/content/generated_outp
Traceback (most recent call last):
  File "/content/c-rnn-gan/rnn_gan.py", line 152, in <module>
    base_cell=tf.compat.v1.contrib.rnn.BasicLSTMCell,
AttributeError: module 'tensorflow.compat.v1' has no attribute 'contrib'
```

Donc on a recherché d'autres code qui opte une architecture LSTM pour la création de la musique à l'aide d'un réseau de neurones récurrent en Python, en utilisant la bibliothèque Keras.

Informations sur l'architecture et le code source :

Réseaux neuronaux récurrents (RNN)

Un réseau neuronal récurrent est une classe de réseaux neuronaux artificiels qui utilisent des informations séquentielles. Ils sont dits récurrents car ils exécutent la même fonction pour chaque élément d'une séquence, le résultat dépendant des calculs précédents. Alors que les sorties sont indépendantes des calculs précédents dans les réseaux neuronaux traditionnels.

On a utilisé LSTM (Long Short-Term Memory). Il s'agit d'un type de réseau neuronal récurrent qui peut apprendre efficacement par descente de gradient. Grâce à un mécanisme de déclenchement, les LSTM sont capables de reconnaître et d'encoder des modèles à long terme. Les LSTM sont extrêmement utiles pour résoudre des problèmes où le réseau doit se souvenir d'informations pendant une longue période, comme c'est le cas pour la musique et la génération de textes.

Music21

Music21 est une boîte à outils Python utilisée pour la musicologie assistée par ordinateur. Elle permet d'enseigner les bases de la théorie musicale, de générer des exemples musicaux et d'étudier la musique. La boîte à outils fournit une interface simple pour acquérir la notation musicale des fichiers MIDI. De plus, elle nous permet de créer des objets Note et Accord afin que nous puissions créer facilement nos propres fichiers MIDI.

On va utiliser Music21 pour extraire le contenu de notre ensemble de données et pour prendre la sortie du réseau neuronal et la traduire en notation musicale.

Keras

On va utiliser la bibliothèque Keras pour créer et entraîner le modèle LSTM. Une fois le modèle formé, nous l'utiliserons pour générer la notation musicale de notre musique.

Training

Dans cette section, nous verrons comment nous avons recueilli les données pour notre modèle, comment nous les avons préparées pour qu'elles puissent être utilisées dans un modèle LSTM et l'architecture de notre modèle.

Données

Dans notre dépôt Github, nous avons utilisé de la musique pour piano, principalement tirée des bandes originales de Final Fantasy.

La première étape de l'implémentation du réseau neuronal consiste à examiner les données avec lesquelles nous allons travailler.

On peut voir que pour générer de la musique avec précision, notre réseau neuronal doit être capable de prédire la note ou l'accord suivant. Cela signifie que notre tableau de prédiction devra contenir chaque objet note et accord rencontré dans notre ensemble d'entraînement, un réseau LSTM peut facilement le faire.

Préparation des données

Vu qu'on a déterminé que les caractéristiques que nous voulons utiliser sont les notes et les accords comme entrée et sortie de notre réseau LSTM

Tout d'abord, nous allons charger les données dans un tableau, comme on peut le voir dans l'extrait de code ci-dessous :

```

from music21 import converter, instrument, note, chord
notes = []
for file in glob.glob("midi_songs/*.mid"):
    midi = converter.parse(file)
    notes_to_parse = None

    parts = instrument.partitionByInstrument(midi)

    if parts: # file has instrument parts
        notes_to_parse = parts.parts[0].recurse()
    else: # file has notes in a flat structure
        notes_to_parse = midi.flat.notes

    for element in notes_to_parse:
        if isinstance(element, note.Note):
            notes.append(str(element.pitch))
        elif isinstance(element, chord.Chord):
            notes.append('.'.join(str(n) for n in
element.normalOrder))

```

Le code commence par charger chaque fichier dans un objet de flux Music21 en utilisant la fonction `converter.parse(file)`. En utilisant cet objet de flux, on obtient une liste de toutes les notes et de tous les accords du fichier.

Le code ajoute la hauteur de chaque note à l'aide de sa notation en chaîne.

Ensuite il ajoute pour chaque accord en codant l'identifiant de chaque note de l'accord dans une seule chaîne, chaque note étant séparée par un point.

Ces codages nous permettent de décoder facilement la sortie générée par le réseau en notes et accords corrects.

Après avoir mis toutes les notes et tous les accords dans une liste séquentielle, nous pouvons créer les séquences qui serviront d'entrée à notre réseau.

Tout d'abord, la fonction de mappage permet de passer des données catégorielles basées sur des chaînes de caractères aux données numériques basées sur des nombres entiers.

Après cela, on trouve la création des séquences d'entrée pour le réseau et leurs sorties respectives. La sortie de chaque séquence d'entrée sera la première note ou le premier accord qui suit la séquence de notes de la séquence d'entrée dans notre liste de notes.

Dans notre code, la longueur de chaque séquence est fixée à 100 notes/accords. Cela signifie que pour prédire la note suivante dans la séquence, le réseau dispose des 100 notes précédentes pour l'aider à faire sa prédiction.

Modèle

Dans notre modèle, on a quatre types de couches différents :

Les couches LSTM sont des couches de réseau neuronal récurrent qui prennent une séquence en entrée et peuvent renvoyer des séquences (`return_sequences=True`) ou une matrice.

Les couches de Dropout sont une technique de régularisation qui consiste à fixer une fraction d'unités d'entrée à 0 à chaque mise à jour au cours de la formation afin d'éviter un surajustement. La fraction est déterminée par le paramètre utilisé avec la couche.

Les couches denses ou couches entièrement connectées sont des couches de réseau neuronal entièrement connectées où chaque nœud d'entrée est connecté à chaque nœud de sortie.

La couche d'activation détermine la fonction d'activation que notre réseau neuronal utilisera pour calculer la sortie d'un nœud.

```
model = Sequential()
model.add(LSTM(
    256,
    input_shape=(network_input.shape[1], network_input.shape[2]),
    return_sequences=True
))
model.add(Dropout(0.3))
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.3))
model.add(Dense(n_vocab))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
optimizer='rmsprop')
```

Pour chaque couche LSTM, Dense et Activation, le premier paramètre est le nombre de nœuds que la couche doit avoir. Pour la couche Dropout, le premier paramètre est la fraction des unités d'entrée qui doivent être abandonnées pendant la formation.

Pour la première couche, nous devons fournir un paramètre unique appelé `input_shape`. Le but de ce paramètre est d'informer le réseau de la forme des données qu'il va entraîner.

La dernière couche doit toujours contenir le même nombre de nœuds que le nombre de sorties différentes de notre système. Cela garantit que la sortie du réseau correspondra directement à nos classes.

On a utilisé un réseau simple composé de trois couches LSTM, trois couches de Dropout, deux couches denses et une couche d'activation.

Pour calculer la perte pour chaque itération de la formation, nous utiliserons l'entropie croisée catégorielle puisque chacune des sorties n'appartient qu'à une seule classe et que nous avons plus de deux classes avec lesquelles travailler. Et pour optimiser notre réseau, nous utiliserons un optimiseur RMSprop, qui est généralement un très bon choix pour les réseaux neuronaux récurrents.

```
filepath = "weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(
    filepath, monitor='loss',
    verbose=0,
    save_best_only=True,
    mode='min'
)
callbacks_list = [checkpoint]

model.fit(network_input, network_output, epochs=200, batch_size=64,
callbacks=callbacks_list)
```

La fonction `model.fit()` de Keras est utilisée pour entraîner le réseau. Le premier paramètre est la liste des séquences d'entrée que nous avons préparée précédemment et le second est une liste de leurs sorties respectives. Le réseau sera entraîné pendant 200 époques, chaque lot propagé dans le réseau contenant 64 échantillons.

Pour nous assurer que nous pouvons arrêter l'entraînement à tout moment sans perdre tout notre travail, nous utiliserons des points de contrôle de modèle. Les points de contrôle du modèle permettent d'enregistrer les poids des nœuds du réseau dans un fichier après chaque époque. Cela nous permet d'arrêter l'exécution du réseau neuronal une fois que nous sommes satisfaits de la valeur de perte sans avoir à nous soucier de la perte des poids. Sinon, il faudrait attendre que le réseau ait terminé les **200** époques avant de pouvoir enregistrer les poids dans un fichier.

Générer de la musique

On peut maintenant utiliser le modèle entraîné pour commencer à générer des notes.

Puisque nous avons une liste complète de séquences de notes à notre disposition, nous allons choisir un index aléatoire dans la liste comme point de départ, ce qui nous permet d'exécuter plusieurs fois le code de génération sans rien changer et d'obtenir des résultats différents à chaque fois.

On a également une fonction de mappage pour décoder la sortie du réseau. Cette fonction permettra de passer des données numériques aux données catégorielles (des entiers aux notes).

```

start = numpy.random.randint(0, len(network_input)-1)
int_to_note = dict((number, note) for number, note in
enumerate(pitchnames))

pattern = network_input[start]
prediction_output = []

# generate 500 notes
for note_index in range(500):
    prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
    prediction_input = prediction_input / float(n_vocab)

    prediction = model.predict(prediction_input, verbose=0)

    index = numpy.argmax(prediction)
    result = int_to_note[index]
    prediction_output.append(result)

    pattern.append(index)
    pattern = pattern[1:len(pattern)]

```

Nous avons choisi de générer 500 notes à l'aide du réseau, car cela représente environ deux minutes de musique et donne au réseau beaucoup d'espace pour créer une mélodie. Pour chaque note que nous voulons générer, nous devons soumettre une séquence au réseau. La première séquence que nous soumettons est la séquence de notes à l'index de départ. Pour chaque séquence suivante que nous utilisons comme entrée, nous supprimons la première note de la séquence et insérons la sortie de l'itération précédente à la fin de la séquence.

Vu qu'on a une liste de notes et d'accords générés par le réseau, on va créer un objet Music21 Stream en utilisant la liste comme paramètre. Enfin, pour créer le fichier MIDI contenant la musique générée par le réseau, on a la fonction write de la boîte à outils Music21 pour écrire le flux dans un fichier.

```

midi_stream = stream.Stream(output_notes)
midi_stream.write('midi', fp='test_output.mid')

```

Résultats :

Nous avons fait plusieurs tests de générations de musique avec différents modèles : à 1 epochs - 21 epochs - 59 epochs - 98 epochs - 200 epochs.

On a remarqué que plus on fait entraîner le réseau avec un nombre important d'epochs plus on a de bons résultats. Vous trouverez l'ensemble des musiques générées avec les différentes configurations dans le dossier : "generated" du code source.

On a pas pu mettre le modèle obtenu lors de l'exécution car le fichier est trop volumineux et git ne l'a pas accepté.

Conclusion

On a pu voir comment créer un réseau neuronal LSTM pour générer de la musique. Les résultats sont impressionnants et nous montrent que les réseaux de neurones peuvent créer de la musique et pourraient potentiellement être utilisés pour aider à créer des pièces musicales plus complexes.