Score-to-song : Application de Modèles Neuronaux de Synthèse Vocale à la Synthèse Musicale

Valentin Gaillard, Maxime Poret, Rémi Brisset
Client : Pierre Hanna

Mars 2019

Contents

Introduction

La génération automatique d'audio est une tâche complexe qui entre en jeu dans de nombreuses applications actuelles, telles que la transcription texte-parole, la synthèse musicale, ou l'édition de son (transfert de style par exemple).

Les méthodes les plus anciennes consistent à partir d'une base de données de sons basiques pré-enregistrés ou pré-synthétisés (phonèmes pour la paroles, notes pour la musique), et de concaténer les sons voulus les uns aux autres, éventuellement avec une légère superposition, pour former la séquence attendue [1][2]. Le problème de ces méthodes est qu'il en résulte un son clairement synthétique, manquant de fluidité et d'expressivité. Il est aussi nécessaire de disposer d'un très grand dictionnaire de phonèmes, et ce pour chaque locuteur différent.

L'autre grande famille historique de méthodes de synthèse sonore est paramétrique, et consiste à connaître et restituer les phonèmes à partir de leurs caractéristiques spectrales et d'un vocoder. Zen et al.[3] fournissent une analyse de l'état de l'art pour ces méthodes en 2009.

Les approches plus récentes sont paramétriques aussi, mais basées sur l'apprentissage (le plus souvent par réseaux de neurones profonds) et la restitution de descripteurs spectraux. Ces méthodes peuvent être source d'artefacts indésirables lors du retour au domaine temporel. Pour cette raison, nous nous intéressons ici à une méthode de synthèse basée sur un apprentissage neuronal de la construction du son, mais qui reste dans le domaine temporel d'un bout à l'autre (*End-to-end*).

En particulier, nous avons constaté que beaucoup de méthodes de synthèse sonore par réseau de neurones *End-to-end* étaient destinées à des applications de synthèse vocale (*text-to-speech*), et nous nous sommes demandés si ces mêmes méthodes pouvaient être adaptées à la synthèse musicale sous la supervision de partitions, ici en format MIDI (par analogie au *Text-to-speech*, nous appellerons cette tâche le *Score-to-song*).

1 Etat de l'art

Dans cette section, nous décrirons brièvement les réseaux de neurones existants que nous avons étudiés, ainsi que les applications qui en ont été tirées.

Wavenet: un réseau de neurones profond pour générer du signal audio brut[4]. Le modèle est entièrement probabiliste et autorégressif, tel que la distribution prédictive de chaque échantillon audio est conditionnée par chacun des précédents. Il a été prouvé que, appliqué au *text-to-speech*, Wavenet est capable de produire des sons que les auditeurs humains considèrent comme plus naturels que ceux résultant des meilleurs modèles antérieurs. C'est la technologie utilisée par l'assistant vocal de Google [5]

SampleRNN : un réseau de neurones profond développé par Mehri et al.[6], plus récent que Wavenet et visant à approfondir le concept de celui-ci. Ce réseau consiste en une cascade de neurones récursifs sur plusieurs niveaux de détail interdépendants, qui s'entraînent à restituer les dépendances sur le long terme des échantillons numériques qui constituent le son. Par défaut ce modèle est inconditionnel : la synthèse s'effectue automatiquement et sans supervision humaine, donc ne permet pas de synthétiser une musique écrite au préalable.

Dadabots : Il a été montré par le projet "Dadabots" sur Bandcamp[7][8] que l'application du modèle inconditionnel de SampleRNN donne des résultats plutôt convainquants pour certains styles de musique particulièrement expérimentaux et déconstruits (math-rock, métal progressif).

Char2Wav : Le modèle SampleRNN a été adapté à une synthèse sonore conditionnelle par Sotelo et al.[9] pour une application text-to-speech. La génération de son est alors supervisée par la lecture de texte et l'extraction des séquences de phonèmes à produire, avec notamment un mécanisme d'attention gérant la transition entre les sons.

Clarinet : Ce réseau de neurones profond est spécialisé dans le *text-to-speech*, et plus généralement dans la synthèse supervisée *End-to-end* de son. Il passe par une représentation intermédiaire des phonèmes basée sur le Mel-spectrogramme (représentation du spectre sur l'échelle logarithmique, qui est proche de la perception humaine) et s'entraîne à reconstituer lesdits phonèmes dans le signal sonore à l'aide d'un modèle de type Wavenet[10]. Voir la figure 4 pour une description de l'architecture de Clarinet tel qu'il est utilisé en *Text-to-speech*.

2 Problématique

L'état de l'art témoigne d'une expérimentation avec les réseaux de neurones plutôt étendue dans le domaine de la synthèse vocale, mais montrant peu de tentatives d'appliquer les mêmes technologies à la synthèse musicale. Dans les applications du *Deep Learning* à ce domaine, on peut observer d'une part des réseaux qui écrivent des partitions à partir des styles musicaux qu'ils ont assimilés[11][12], et d'autre part des réseaux restituant le timbre de leur musique d'entraînement [4][6][10], mais aucun qui apprenne à jouer une partition de la même façon qu'un codeur vocal apprendrait à prononcer un texte.

3 Objectifs

Le but de notre projet est donc de construire un programme conditionnel de synthèse musicale en partant d'un réseau de neurones apprenant le timbre et l'enchaînement d'échantillons temporels de la musique d'entraînement. Une fois son apprentissage terminé, ce système doit être capable de baser la création du son à la fois sur les textures sonores apprises et sur la lecture d'une partition, afin de produire un pseudo-enregistrement musical retranscrivant la partition selon les sonorités de l'ensemble d'entraînement. Par exemple un réseau entraîné sur un piano enregistré dans certaines conditions acoustiques doit pouvoir simuler le jeu de ce même piano, dans ces mêmes conditions, sur une partition qu'il n'a jamais lue auparavant. On cherchera aussi à déterminer à quel point la synthèse d'une partition par notre programme se rapproche d'une vraie interprétation humaine du morceau.

4 Cas d'Utilisation

4.1 Entraînement du réseau neuronal

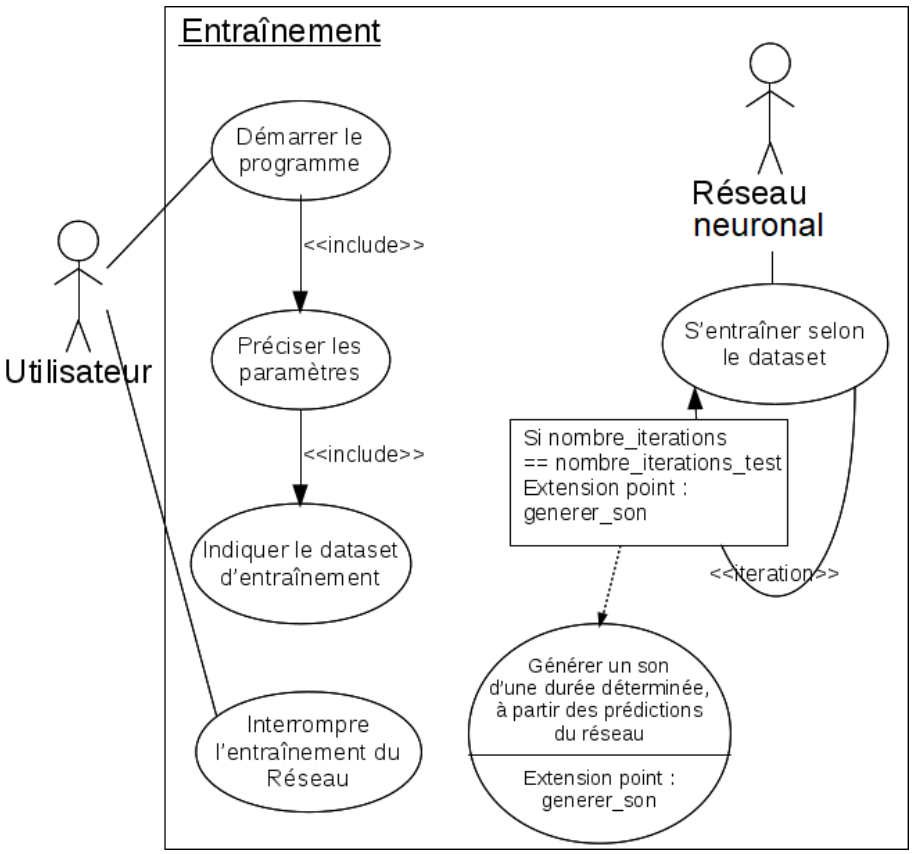


Figure 1: Cas d'Usage 1 : Entraînement du réseau de neurones sur la texture des sons fournis

4.2 Synthèse de musique

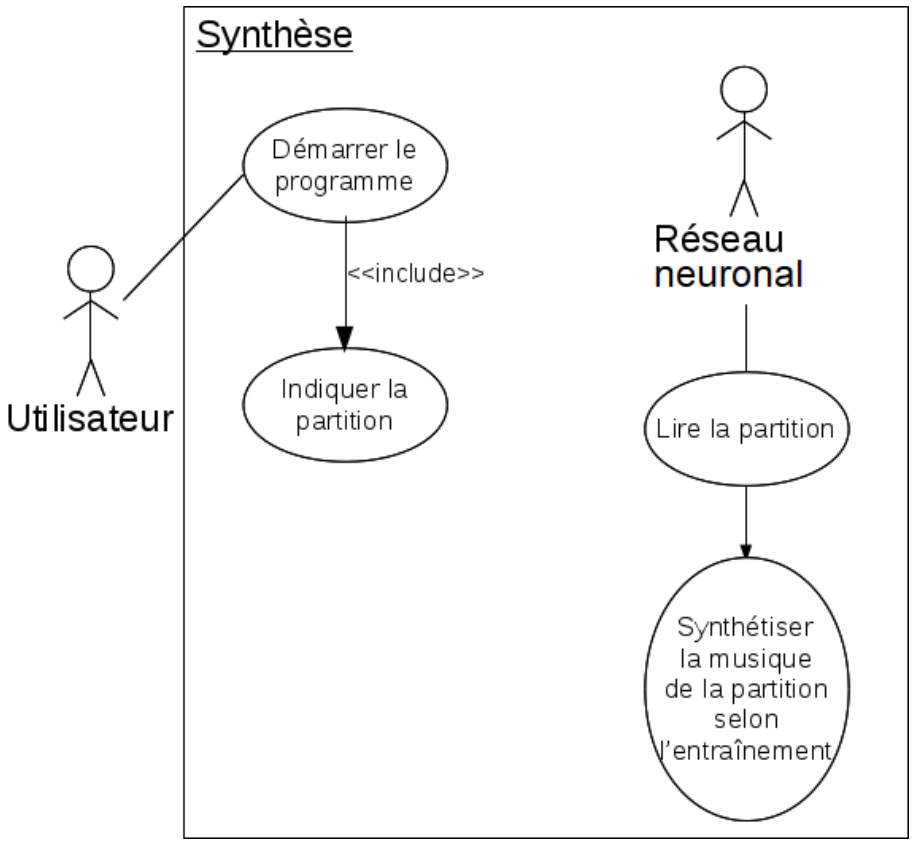


Figure 2: Cas d'Usage 2 : Synthèse musicale à partir de l'apprentissage du réseau de neurones

Notation des priorités: Dans les sections qui suivent, l'ordre de priorité d'un besoin est dénoté par un nombre de 1 (priorité forte) à 3 (priorité faible).

5 Besoins non fonctionnels

Vitesse d'exécution: (Priorité : 2) Un test préalable sur un ordinateur du CREMI (4 coeurs, CPU:E3-1240(v6) 16Go, GPU:GTX1060(6Go)) a établi qu'il fallait environ 7 heures pour effectuer une seule epoch d'entraînement, ainsi que la validation, pour le réseau SampleRNN existant. Dans ces conditions, il est difficile d'exiger d'un programme basé sur ce réseau qu'il aille à une bonne vitesse.
Le besoin non-fonctionnel de rapidité consistera au moins à faire en sorte que les fonctionnalités ajoutées ne multiplient pas le temps d'exécution par plus que 2 sur un matériel équivalent.

Note sur la vitesse d'exécution : Ce besoin a été établi au début du projet, quand nous supposions que notre programme serait basé sur SampleRNN. Sur les mêmes machines, l'autre réseau Clarinet est beaucoup plus rapide et l'entraînement ne prend que quelques minutes par epoch. L'ordre de grandeur établi plus tôt pour le taux d'augmentation de la vitesse d'exécution est toujours celui attendu.

Proposant un son synthétisé de bonne qualité: (Priorité : 2) Le programme doit produire un son, de préférence peu ou pas bruité, qui peut être reconnu comme musical par un auditeur humain. Idéalement, la musique synthétisée doit être indistincte d'un enregistrement naturel.
Test : comparaison subjective de sons sur un certain nombre d'auditeurs:
On présente aux auditeurs des ensembles de X extraits musicaux (dont $Y < X$ sont naturels et $X - Y$ sont synthétiques), et on leur demande de classer, à leur avis, les naturels et les synthétiques.
Ce que l'on veut pour considérer que la synthèse est réaliste, c'est que les auditeurs aient des difficultés à faire la différence entre les Naturels et les Synthétiques, donc on veut un fort ratio de Faux-Naturels FN (Synthétiques classés comme Naturels) et Faux-Synthétiques FS (Naturels classés comme Synthétiques) par rapport à l'ensemble des classifications, de sorte que :

```
indicateur_de_realisme = (FN+FS)/(TN+TS+FN+FS)
```

6 Besoins fonctionnels

6.1 Cas d'utilisation 1

Entraîner le réseau: (Priorité : 1) Le lancement du programme en invite de commande python, avec indication des paramètres et du dataset, doit mettre en route l'entraînement du réseau neuronal choisi.

Permettre l'interruption arbitraire de l'entraînement par l'utilisateur: (Priorité : 2) L'envoi par l'utilisateur d'un signal d'interruption dans le terminal qui fait tourner le programme doit permettre d'arrêter l'entraînement.

Mettre à disposition le réseau entraîné : (Priorité : 2) Une fois l'entraînement interrompu, les paramètres optimisés dans le réseau doivent être sauvegardés dans un fichier, et pouvoir être réutilisés dans des exécutions consécutives du programme.

Produire au cours de l'entraînement des fichiers sons de validation: (Priorité : 2) Ces sons doivent être générés à l'issue de chaque epoch d'entraînement et sont constitués à partir de séries de prédictions d'échantillons par le réseau.

Annoter la musique d'entraînement: (Priorité : 3) Lors de son entraînement, le réseau neuronal doit extraire , soit par lecture d'une annotation, soit par extraction automatique de descripteurs, l'information musicale décrivant les notes qu'il "entend".

Test : on fait lire par le programme un morceau dont on connaît les caractéristiques, et on s'assure que l'enchaînement des descripteurs calculé par le programme est le même que celui de la vérité terrain.

6.2 Cas d'utilisation 2

Lire une partition MIDI: (Priorité : 1) Extraire la suite d'informations de note, octave, durée, d'une séquence MIDI passée en fichier d'entrée du programme, et en faire une séquence de symboles (par analogie avec les phonèmes pour la parole) qui sera interprétée par l'outil neuronal de synthèse. Pour simplifier on considère que le programme ne prend que des MIDI d'une seule piste (piano).

Test : on part d'une partition MIDI simple et brève dont on connaît toutes les notes et qu'on fait lire/traduire en symboles dans des tests unitaires, avant d'effectuer une comparaison systématique de la séquence de symboles calculée par rapport à la séquence connue.

Interpréter une partition lue: (Priorité : 2) Synthétiser la musique selon les prédictions du réseau neuronal choisi, entraîné au préalable.

7 Contraintes

Exécution du programme: Comme indiqué dans la section "Vitesse d'exécution" des Besoins Non-Fonctionnels, il se peut que l'on doive faire s'entraîner le réseau très longtemps pour obtenir des résultats acceptables.

Annotation des morceaux (si on se base sur un réseau neuronal non-conditionnel): Faire tenir compte d'une annotation de la musique par SampleRNN nécessitera probablement de grandes modifications sur un code déjà

complexe, ainsi qu'une annotation musicale des sons d'entraînement, qui risque de prendre longtemps si on ne peut pas l'automatiser. On s'inspirera de la façon dont le projet de synthèse vocale Char2Wav[9], basé sur SampleRNN, extrait l'information de ses sons d'entraînement. On doit en particulier déterminer si les méthodes de ce programme, dédiées à la parole, sont transposables à la musique.

8 Implémentation

8.1 Choix du réseau de neurones

Au cours de ce projet, nous avons étudié deux réseaux de neurones différents, SampleRNN et Clarinet, en tentant d'établir les moyens dont nous disposions pour les mettre à profit dans une synthèse musicale par lecture de partition.

8.1.1 Première piste : Rendre SampleRNN conditionnel

Tout d'abord, la piste d'implémentation suivie a été de rendre la synthèse du réseau SampleRNN conditionnelle en ajoutant à son vecteur d'entrée un biais représentant les caractéristiques mélodiques de la trame à générer. Voir l'architecture décrite par la figure 3.

Cette piste est basée sur la façon dont Sotelo et al. présentent leur travail pour Char2Wav[9], en particulier le traîtement préalable du texte par le système encodeur-décodeur, produisant des trames qui représentent les phonèmes à synthétiser dans l'ordre chronologique. Cependant, le code mis à disposition sur github pour ce projet n'est pas fonctionnel, et nous ne sommes pas parvenus à en extraire les modifications à appliquer au réseau pour le conditionner de la façon décrite. Par ailleurs, le code de SampleRNN fourni par Mehri et al. n'est pas très ouvert aux modifications, et n'avons pas pu déterminer comment ajouter le biais lié aux caractéristiques mélodiques voulues. Pour ces raisons, cette piste a été abandonnée en faveur d'une autre impliquant le réseau Clarinet, déjà conditionnel.

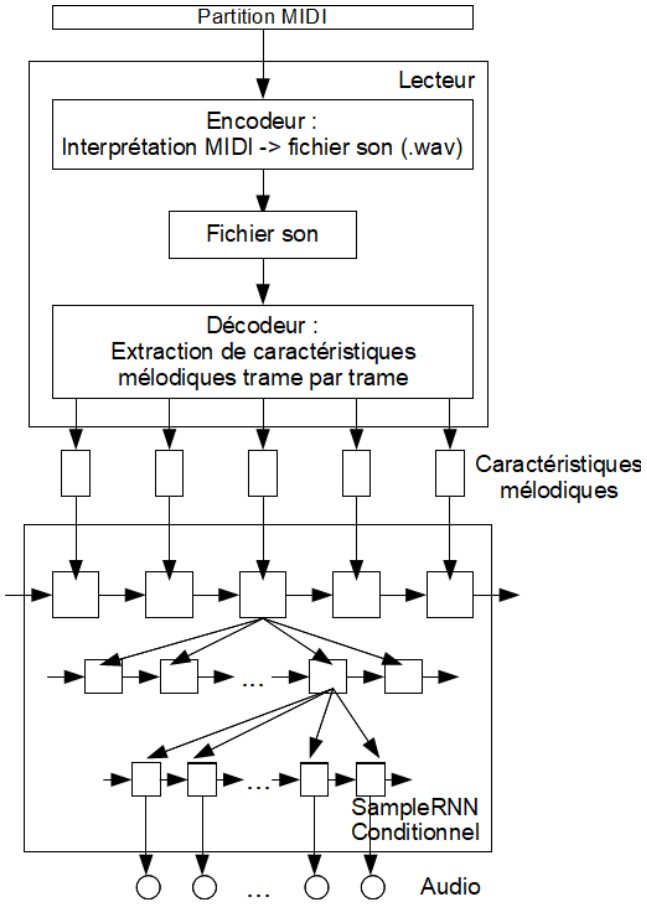


Figure 3: Architecture prévue au début du projet. Dans notre cas, le système encodeur-décodeur produit une première représentation basique de la musique décrite par le fichier MIDI, puis en extrait des trames de descripteurs mélodiques qui sont passées à une version conditionnelle de SampleRNN pour la synthèse.

8.1.2 Deuxième piste : Adapter Clarinet à la musique

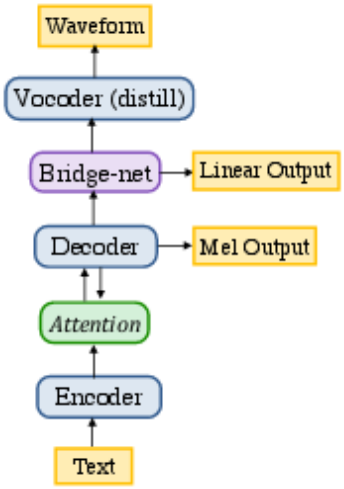


Figure 4: Architecture de Clarinet

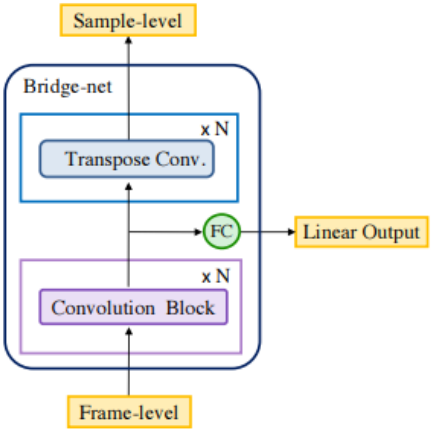


Figure 5: Détail de Bridge-net

L'utilisation de Clarinet en *text-to-speech* repose sur un système *encoder-attention-decoder* (voir figure 4) qui lit le texte lettre par lettre en le transformant en une série de phonèmes (*encoder*), puis applique un mécanisme d'attention pour que l'enchaînement des phonèmes paraisse naturel (*attention*), puis transforme la série chronologique de phonèmes pondérés en une série de trames de Mel-spectrogramme qui supervise l'entraînement et la synthèse du réseau (*decoder*).

Avec *Clarinet*, Ping et al. [10] ont adapté Wavenet [4] en rajoutant de la *knowledge distillation*[13]. L'idée est d'entraîner 2 réseaux, un *teacher* "généraliste" plus volumineux et un *student*, "spécialiste". Cela permet, en entraînant d'abord le *teacher* puis le *student* à partir du *teacher*, de compresser dans un plus petit réseau un modèle complexe, pour par exemple l'inclure dans une application mobile.

8.1.3 Architecture adaptée

L'architecture de notre projet est basée sur les scripts python de Clarinet. Elle reprend une grande partie du code de ce dernier tout en ajoutant des modules et en changeant le dataset utilisé afin de l'adapter à l'apprentissage et à la synthèse de musique. L'essentiel des modifications apportées au code ont trait au prétraitement des données avant leur passage par le réseau. Ci-dessous, la figure 6 résume les modules utilisés dans la chaîne de traitement du projet.

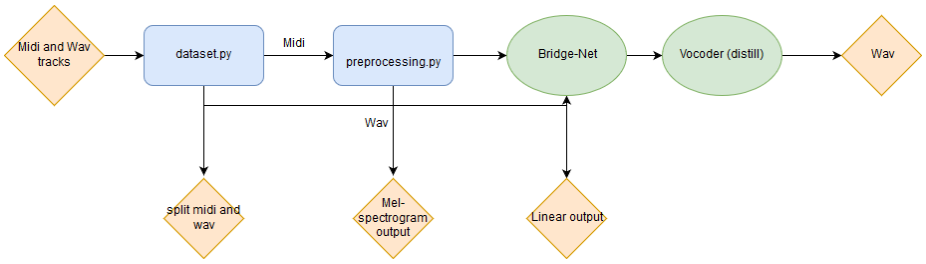


Figure 6: Architecture utilisée pour le projet

dataset.py : Effectue un recoupage du dataset pour obtenir des phrases musicales courtes. Voir section 8.5.

preprocessing : Transforme les fichiers MIDI en leur forme intermédiaire Mel-spectrogramme, qui est passée au réseau pour l'apprentissage.

8.2 Choix de la représentation intermédiaire

Dans le fonctionnement d'origine de Clarinet, le texte est transformé en une représentation intermédiaire consistant en une série de trames de Mel-spectrogramme, pondérées par un mécanisme d'attention afin de donner une fluidité naturelle à l'enchaînement des phonèmes.

Il s'agit alors d'informations basiques de timbre, propres à l'analyse de la parole. Dans notre projet, on veut que l'information conditionnant le réseau soit plutôt associée à la mélodie, puisque le timbre régit plutôt le choix de l'instrument, qui ne sera pas obligatoirement un piano dans les utilisations futures du programme. Une possibilité aurait alors été d'utiliser un chromagramme calculé sur le MIDI comme forme intermédiaire pour le conditionnement du réseau.

Cependant de crainte d'avoir plus de code à adapter, et sachant que nous n'allions traiter qu'un type d'instrument dans nos travaux, nous avons préféré garder dans un premier temps la forme intermédiaire déjà utilisée par le réseau: un Mel-spectrogramme.

8.3 Réseau, entraînement et synthèse sonore

Le but est d'adapter les scripts Clarinet pour passer d'une synthèse vocale à une synthèse musicale. Cela passe notamment par la modification des scripts python Clarinet *preprocessing.py* et *data.py*.

preprocessing.py : Le module *preprocessing* effectue un travail équivalent aux sections "Encoder - Attention - Decoder" de l'architecture de départ. Il consiste à interpréter les fichiers MIDI une première fois de manière basique sous la forme d'un fichier wav. Cette tâche est effectuée par le programme externe *timidity*. À partir de ce fichier wav, on calcule ensuite le *Mel-spectrogramme* associé à la musique qu'on doit retranscrire, à l'aide de la bibliothèque python de traitement du signal *librosa*.

data.py C'est le module central. Il contient toutes les fonctions de création et d'entraînement du réseau. Nos modifications ont principalement porté sur le chargement de nos données et la création des ensembles d'entraînement et de test.

Les scripts *create_csv.py* et *rename_files.py* créent également un fichier csv différent correspondant au nouveau jeu de données, et renomment les fichiers wav et MIDI selon les extraits générés précédemment.

8.4 Choix des datasets

Le choix du dataset a été effectué en fonction de la complexité des morceaux. Il est plus simple pour le réseau d'utiliser de la musique constituée d'une seule piste correspondant à un seul instrument. Nous voulions également un dataset qui mette en évidence les différences d'interprétation (dynamique, changements de tempo) entre la partition et l'enregistrement d'un artiste qui la joue. Le but est d'observer si le réseau est capable de s'inspirer aussi des variations personnelles de l'artiste pour la synthèse.

Nous avons effectué nos entraînements sur deux bases de données différentes:

Essen30 : Le dataset *Essen* nous a été recommandé par M. Pierre Hanna. Il est composé uniquement de pistes MIDI monophoniques, jouant une seule note à la fois. Le dataset étant particulièrement grand et possédant des pistes trop longues pour être utilisables par notre réseau, nous avons choisi de ne sélectionner que celles de durée inférieure à 30 secondes. Afin de compléter ce dataset avec les wavs correspondant aux fichiers MIDI, nous avons utilisé un script faisant appel au programme *timidity* produisant des "interprétations" synthétiques des morceaux.

Ce dataset nous a servi de base d'entraînement car il est une simplification de notre sujet. Il ne présente pas d'interprétation dynamique de la part de l'artiste qui joue (alignement parfait entre le MIDI et l'enregistrement) et les pistes sont beaucoup plus simples que de vrais morceaux de musique.

Maestro_Bach : Le dataset maestro est composé de pistes de piano pour une durée totale de plus de 172 heures. L'enregistrement audio et la création du MIDI sont réalisés en même temps avec une différence d'environ 3 ms entre la note du wav et du celle du MIDI. Afin de réduire la taille du dataset et

d'obtenir une cohérence dans la composition, nous avons trié ce dernier pour ne sélectionner que les enregistrements de morceaux écrits par Johann Sebastian Bach. La qualité de son et la complexité des pistes sont supérieures à celle du dataset *Essen*, cela permet un entraînement sur des exemples plus concrets.
Nous obtenons 14h de wav et les partitions équivalentes en MIDI, toutes monophoniques. Un problème qui subsiste est que les pistes sont beaucoup trop longues pour être utilisées, nous avons donc implémenté un module de découpage de pistes wav et MIDI afin d'obtenir des durées acceptables.

8.5 Recoupage des données annotées

Nous avons programmé un module capable de couper automatiquement des enregistrements musicaux et leurs partitions en séquences d'une longueur de l'ordre de la dizaine de secondes. On effectue aussi par ce procédé une augmentation des données.

Localisation et usage : Le module est codé dans le fichier /clarinet/dataset.py. Pour l'appeler, entrer la commande :

```
python dataset.py <dossier wav> <dossier midi>
```

<dossier wav> donne le chemin vers le dossier contenant les wavs à recouper.
<dossier midi> donne le chemin vers le dossier contenant les MIDIs à recouper, sachant que pour chaque wav à couper, il doit y avoir son équivalent MIDI portant le même nom, mais avec l'extension .mid, dans ce dossier.
Les fichiers d'origine sont supprimés et remplacés par tous les sous-fichiers découpés qui résultent de l'appel au module.

Découpage du MIDI : Le MIDI est découpé à chaque endroit où aucune note n'est en train d'être jouée (donc aux silences), et on impose en plus que chaque partie coupée dure plus qu'un certain seuil de temps. Ce seuil est codé en dur et vaut 4 secondes par défaut.

Découpage du wav : On suppose que le dataset est déjà conçu de sorte que les wavs et MIDIs sont parfaitement synchronisés. Ainsi on peut couper le wav aux mêmes points temporels que là où on a coupé le MIDI.

Tests et remarques : Ce module a été testé sur plusieurs morceaux de longueurs variées, pour lesquels les MIDIs et wavs étaient toujours bien synchronisés, et on a pu vérifier que jusqu'au bout les wavs découpés correspondaient bien aux mêmes notes inscrites dans les MIDIs correspondants. Des problèmes de décalage se présentent en revanche quand le MIDI et le wav ne sont pas synchronisés, par exemple parce que l'interprétation de l'artiste intègre des changements de tempo. De plus, dans ce cas, ce décalage se propage pour chaque découpage qui suit sur le même morceau.

8.6 Entraînement et synthèse par le réseau

Actuellement, l'entraînement du réseau se déroule de la même façon que dans sa version de base. Nous avons surtout modifié le type des données et la façon dont elles sont traitées avant de participer à l'entraînement, le test, et la validation du réseau.

9 Evaluation & Résultats

L'évaluation automatique de la qualité des extraits audio synthétisés est un domaine où beaucoup reste encore à faire. En effet, on observe que les mesures quantitatives usuelles (maximum likelihood, nearest neighbour estimation ...) sont peu corrélées avec le jugement humain [14]. Malgré l'émergence de nouvelles métriques, comme l'inception score [15]pour l'évaluation des *réseaux génératif adversariaux (GANs)*, aucune des solutions actuelles ne se substitue à l'évaluation humaine.

Méthodes d'évaluation: Nous décidons ici d'utiliser comme métrique la loss des ensembles de test et de validation des 2 réseaux *teacher* et *student* ainsi qu'un jugement humain subjectif.

NOTE: Comme décrit dans la partie limitations, l'entraînement du réseau demande beaucoup de temps. C'est pourquoi les résultats présentés dans ce rapport ne sont pas représentatifs des performances qui pourraient être attendues de Clarinet à l'issue d'un entraînement adéquat.
Ces résultats ont été obtenus à partir de l'entraînement du réseau sur l'ensemble *Essen30*. Des résultats supplémentaires sur l'ensemble *maestro_bach* sont prévus pour la soutenance.

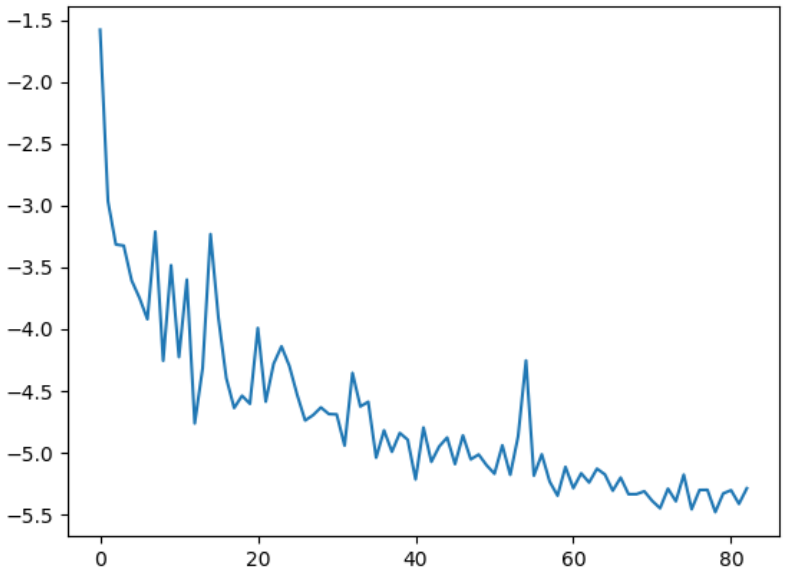


Figure 7: Courbe du calcul de loss d'entraînement en fonction du nombre d'époques pour le réseau teacher avec le dataset Essen30

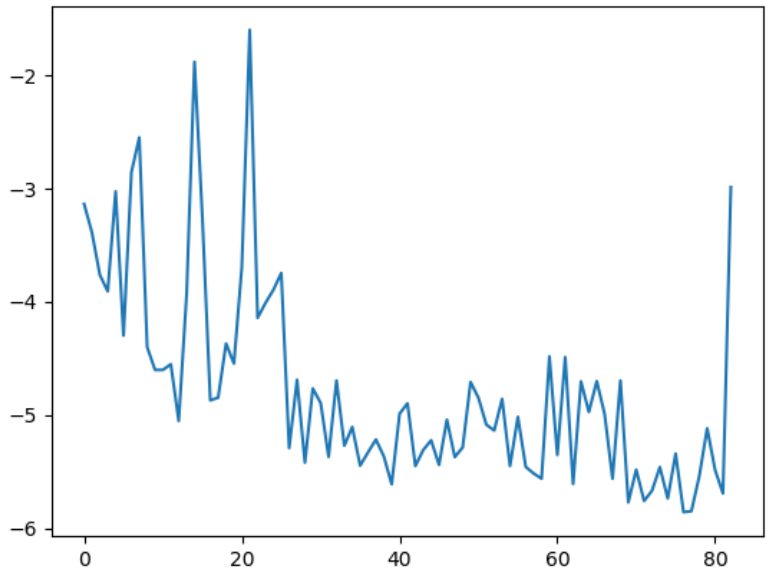


Figure 8: Courbe du calcul de loss de test en fonction du nombre d'époques pour le réseau teacher avec le dataset Essen30

On observe que pour le réseau *teacher*, la loss d'entraînement diminue bien comme on l'espère. Il y a par ailleurs un début d'overfitting qui se présente sur la loss de test des dernières époques.

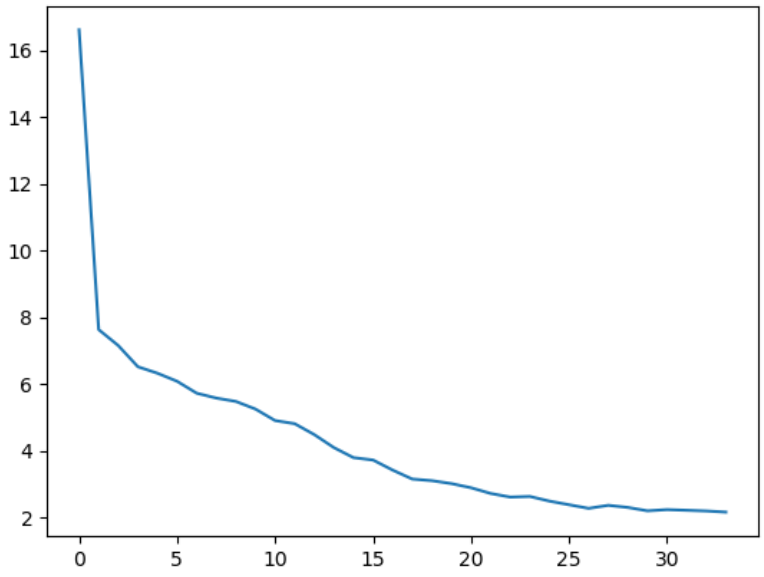


Figure 9: Courbe du calcul de loss d'entraînement en fonction du nombre d'époques pour le réseau student avec le dataset Essen30

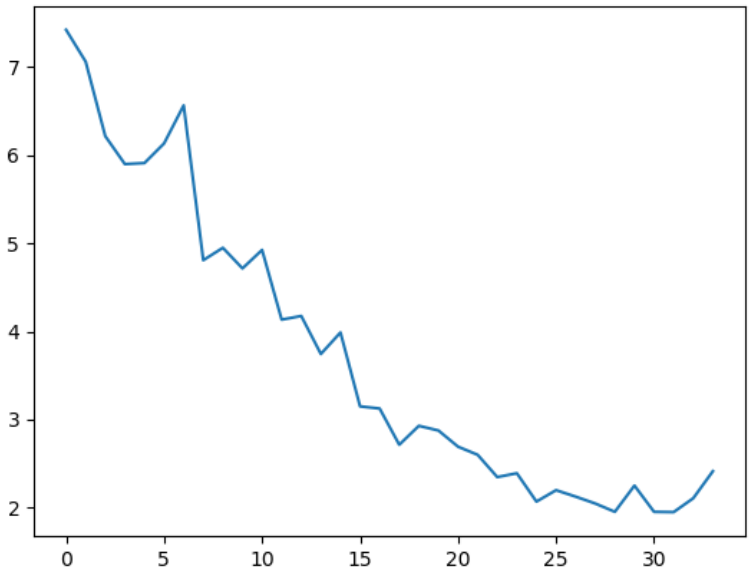


Figure 10: Courbe du calcul de loss de test en fonction du nombre d'époques pour le réseau student avec le dataset Essen30

On observe encore une fois que la loss diminue bien au cours des époques d'entraînement, et que les valeurs de test suivent la même évolution. Le même début d'overfitting est observable dans le test des dernières époques.

10 Bilan

Comparaison Gantt Initial vs Gantt Final (Diagrammes en annexe) Le Gantt prévisionnel montre un projet plus simple et plus fluide que celui réalisé au final, cela est dû notamment au fait d'avoir passé beaucoup de temps à relire et analyser SampleRNN, puis d'avoir été obligés de chercher un autre réseau quand il est devenu évident que nous ne pourrions pas le modifier nous-mêmes. La différence se fait notamment au milieu du projet avec un temps de recherche beaucoup plus important que prévu pour trouver un réseau adaptable. Les problèmes de datasets et de scripts utilitaires n'avaient pas non plus été pris en compte dans le planning.

Limitations Matérielles Nous détaillons ici les problèmes rencontrés au cours de notre projet:

- Puissance de calcul: Tous les scripts d'entraînement et de synthèse du réseau ont été exécutés via ssh sur la machine *Tesla* du *CREMI*, équipé d'une carte graphique *Tesla k20* disposant de 5 Go de RAM. Les calculs étant très coûteux, cela n'a pas toujours été suffisant. En effet, ce manque de puissance nous a limités dans nos expérimentations de par les dépassements de mémoire (erreur de type: *CUDA: Out of Memory*) intempestifs lors de certaines phases de l'entraînement, et tout particulièrement lors de la synthèse où le dépassement était totalement imprévisible.

- Temps de calcul : Découlant du point précédent, le manque de puissance de calcul à notre disposition nous a demandé beaucoup d'attente pour peu de résultats.

- Espace disque : Autre problème lié à notre utilisation des infrastructures de l'université. Nos comptes personnels disposent d'un quota d'espace disque qui est de 70Go pour l'espace de travail individuel. L'utilisation de larges jeux de données combinée, entre autres, à la sauvegarde des paramètres des différentes époques, cause des dépassements de quota qui bloquent toute opération pouvant nécessiter une sauvegarde par la suite.

Conclusion Le projet n'a pas abouti à l'issue du temps imparti. Les problèmes principaux viennent d'objectifs de départ trop ambitieux, d'un important temps de recherche au début du projet, et d'un manque de temps et de matériel qui auraient permis d'effectuer plus efficacement des tests de performance du réseau.

11 Perspectives/Améliorations

Afin d'améliorer notre réseau il faudrait déjà obtenir des résultats satisfaisants permettant d'effectuer le test de confusion entre les morceaux synthétisés et les enregistrements naturels.

Par la suite plusieurs avancées sont encore envisageables :

- Réaliser le cas d'utilisation 2 présenté plus haut en permettant au réseau de lire une partition MIDI d'entrée et de générer son interprétation en wav. Idéalement on veut que les MIDI pour lesquels on synthétise l'interprétation ne soient pas présents dans l'ensemble d'entraînement du réseau.
  On peut construire ce module en effectuant le pré-traitement de la piste MIDI passée en paramètre, puis en communiquant son Mel-spectrogramme à la fonction de synthèse existante de Clarinet.

- Il sera intéressant d'améliorer le réseau pour lui permettre d'apprendre sur des partitions MIDI plus complexes (polyphonie, plusieurs pistes d'instruments, structures complexes).
  Pour accomplir cela, en supposant que le réseau n'est pas déjà capable de produire ce genre de résultat après l'entraînement approprié, on peut

par exemple faire en sorte de synthétiser plusieurs pistes pour chaque instrument ou pour chaque note simultanée, et les combiner par synthèse additive.

- On pourra améliorer la robustesse du module de découpage de dataset pour lui permettre de découper des ensembles dont le MIDI et le wav ne sont pas parfaitement synchronisés.
  Une méthode pour obtenir cette robustesse est de tenir compte du nombre de notes attendues dans le découpage du MIDI, ainsi que d'une liste de leurs hauteurs, et de s'assurer qu'on retrouve bien cet enchaînement dans la portion de wav correspondante. Certaines pistes ont d'ores et déjà été explorées [16].

- Comme expliqué dans les limitations, l'évaluation automatique de sons synthétique est un domaine encore très ouvert, qu'il pourrait être bénéfique d'investiguer.

12 Annexes

Dépôt GitHub

Le code de notre projet est disponible sur GitHub dans le répertoire:
https://github.com/ZuperKuchen/PFE_SampleRNN

References

[1] Eric Moulines and Francis Charpentier. Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones. *Speech Communication*, pages 453–467, 1 Aug 1990.

[2] Andrew J Hunt and Alan W Black. Unit selection in a concatenative speech synthesis system using a large speech database. *International Conference on Acoustics, Speech, and Signal Processing*, pages 373–376, 1996.

[3] Keiichi Tokuda Heiga Zen and Alan W Black. Statistical parametric speech synthesis. *Speech Communication*, pages 1–23, 2009.

[4] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *Computing Research Repository*, abs/1609.03499:pp1–15, 2016.

[5] DeepMind. https://deepmind.com/blog/wavenet-launches-google-assistant/, 4 October 2017.

[6] Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron C. Courville, and Yoshua Bengio. Samplernn: An unconditional end-to-end neural audio generation model. *Computing Research Repository*, abs/1612.07837:pp1–11, 22 Dec 2016.

[7] Zack Zukowski. Generating black metal and math rock: Beyond bach, beethoven, and beatles. *Computing Research Repository*, abs/1811.06639:pp1–3, 16 Nov 2018.

[8] C. Carr and Zack Zukowski. Generating albums with samplernn to imitate metal, rock, and punk bands. *Computing Research Repository*, abs/1811.06633:pp1–4, 16 Nov 2018.

[9] Jose Sotelo, Soroush Mehri, Kundan Kumar, João Felipe Santos, Kyle Kastner, and Aaron Courville. Char2wav: End-to-end speech synthesis. pages 1–6, Mar 2017.

[10] Wei Ping, Kainan Peng, and Jitong Chen. Clarinet: Parallel wave generation in end-to-end text-to-speech. *Computing Research Repository*, abs/1807.07281:pp1–15, 2018.

[11] Flow machines publications. https://www.flow-machines.com/history-cat/publications/. Accessed: 2019-03-21.

[12] Jeroen Vranken and Efstratios Gavves. Generating music in different genres using long short-term memory networks. *Bachelor thesis, University of Amsterdam*, pages pp1–24, 24 June 2016.

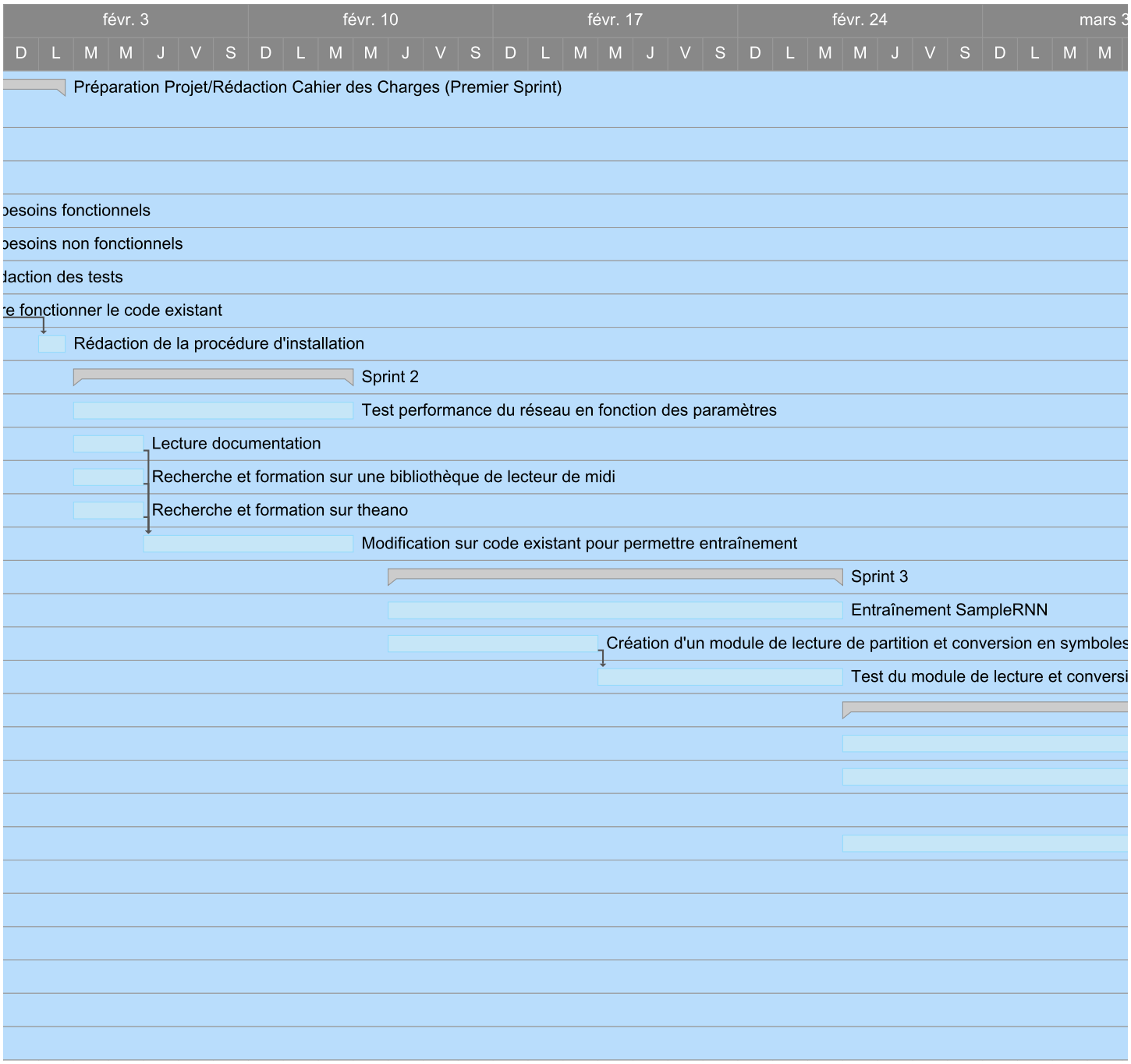
[13] Jeff Dean Geoffrey Hinton, Oriol Vinyals. Distilling the knowledge in a neural network. 9 March 2015.

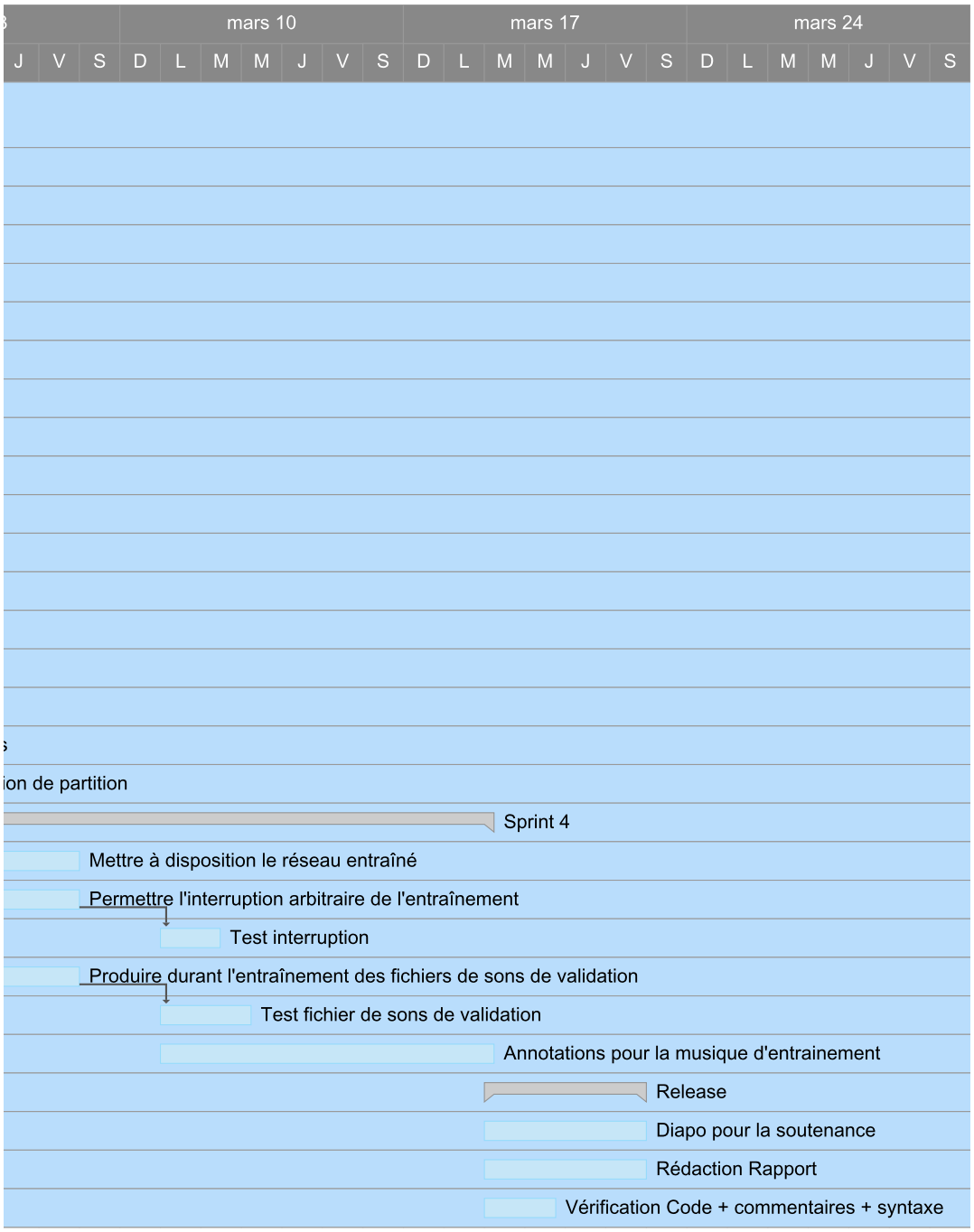
[14] Matthias Bethge Lucas Theis, Aäron van den Oord. A note on the evaluation of generative models. 24 April 2016.

[15] Rishi Sharma Shane Baratt. A note on the inception score. 6 Jan 2018.

[16] Collin Raffel. Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching. 2016.

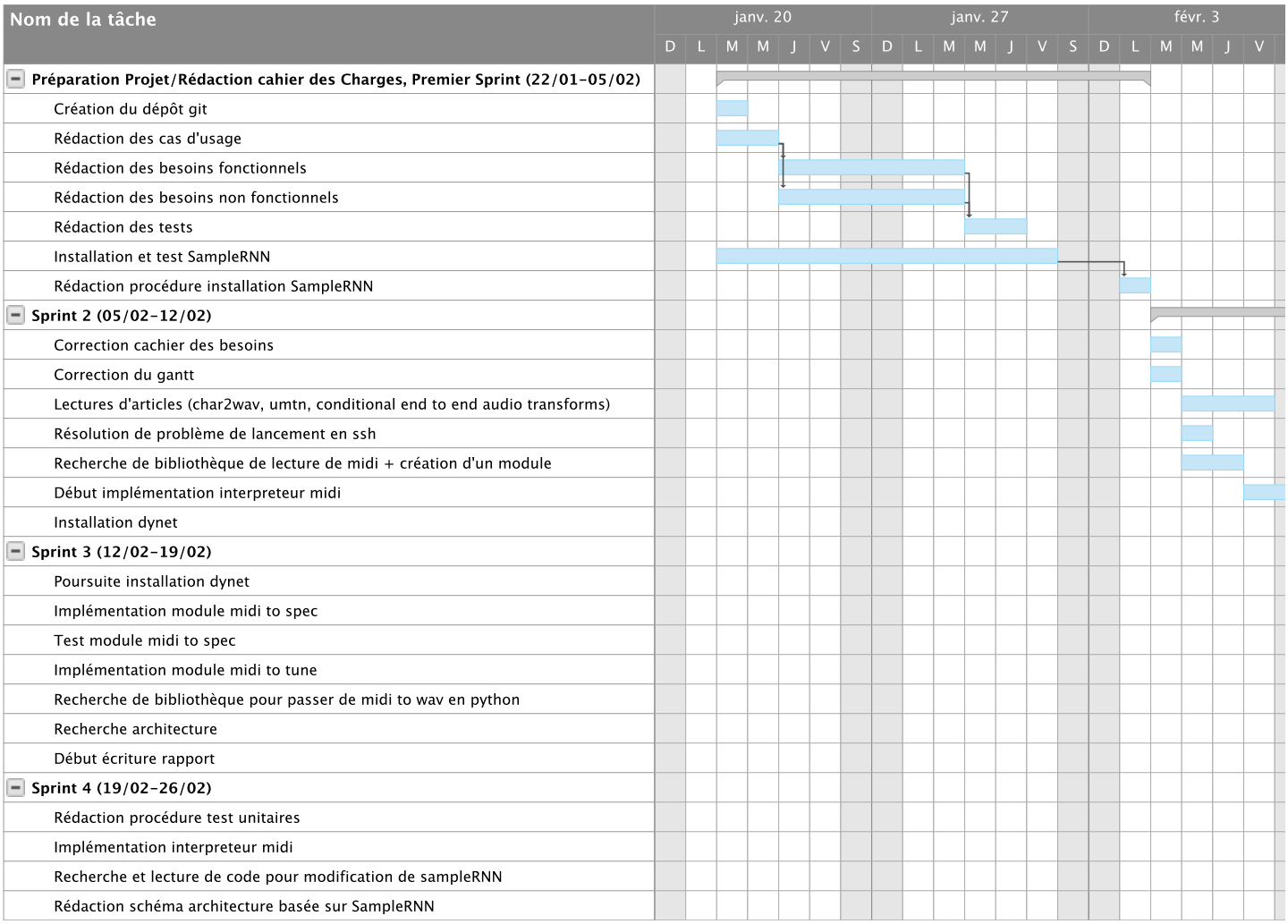
Nom de la tâche	janv. 20							janv. 27						
	D	L	M	M	J	V	S	D	L	M	M	J	V	S
Préparation Projet/Rédaction Cahier des Charges (Premier Sprint)														
Création du dépôt git														
Rédaction des cas d'usage														
Rédaction des besoins fonctionnels														
Rédaction des besoins non fonctionnels														
Rédaction des tests														
Faire fonctionner le code existant														
Rédaction de la procédure d'installation														
Sprint 2														
Test performance du réseau en fonction des paramètres														
Lecture documentation														
Recherche et formation sur une bibliothèque de lecteur de midi														
Recherche et formation sur theano														
Modification sur code existant pour permettre entraînement														
Sprint 3														
Entraînement SampleRNN														
Création d'un module de lecture de partition et conversion en symboles														
Test du module de lecture et conversion de partition														
Sprint 4														
Mettre à disposition le réseau entraîné														
Permettre l'interruption arbitraire de l'entraînement														
Test interruption														
Produire durant l'entraînement des fichiers de sons de validation														
Test fichier de sons de validation														
Annotations pour la musique d'entrainement														
Release														
Diapo pour la soutenance														
Rédaction Rapport														
Vérification Code + commentaires + syntaxe														


févr. 3
févr. 10
févr. 17
févr. 24
mars 3
D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M
Préparation Projet/Rédaction Cahier des Charges (Premier Sprint)
besoins fonctionnels
besoins non fonctionnels
daction des tests
re fonctionner le code existant
Rédaction de la procédure d'installation
Sprint 2
Test performance du réseau en fonction des paramètres
Lecture documentation
Recherche et formation sur une bibliothèque de lecteur de midi
Recherche et formation sur theano
Modification sur code existant pour permettre entraînement
Sprint 3
Entraînement SampleRNN
Création d'un module de lecture de partition et conversion en symboles
Test du module de lecture et conversi

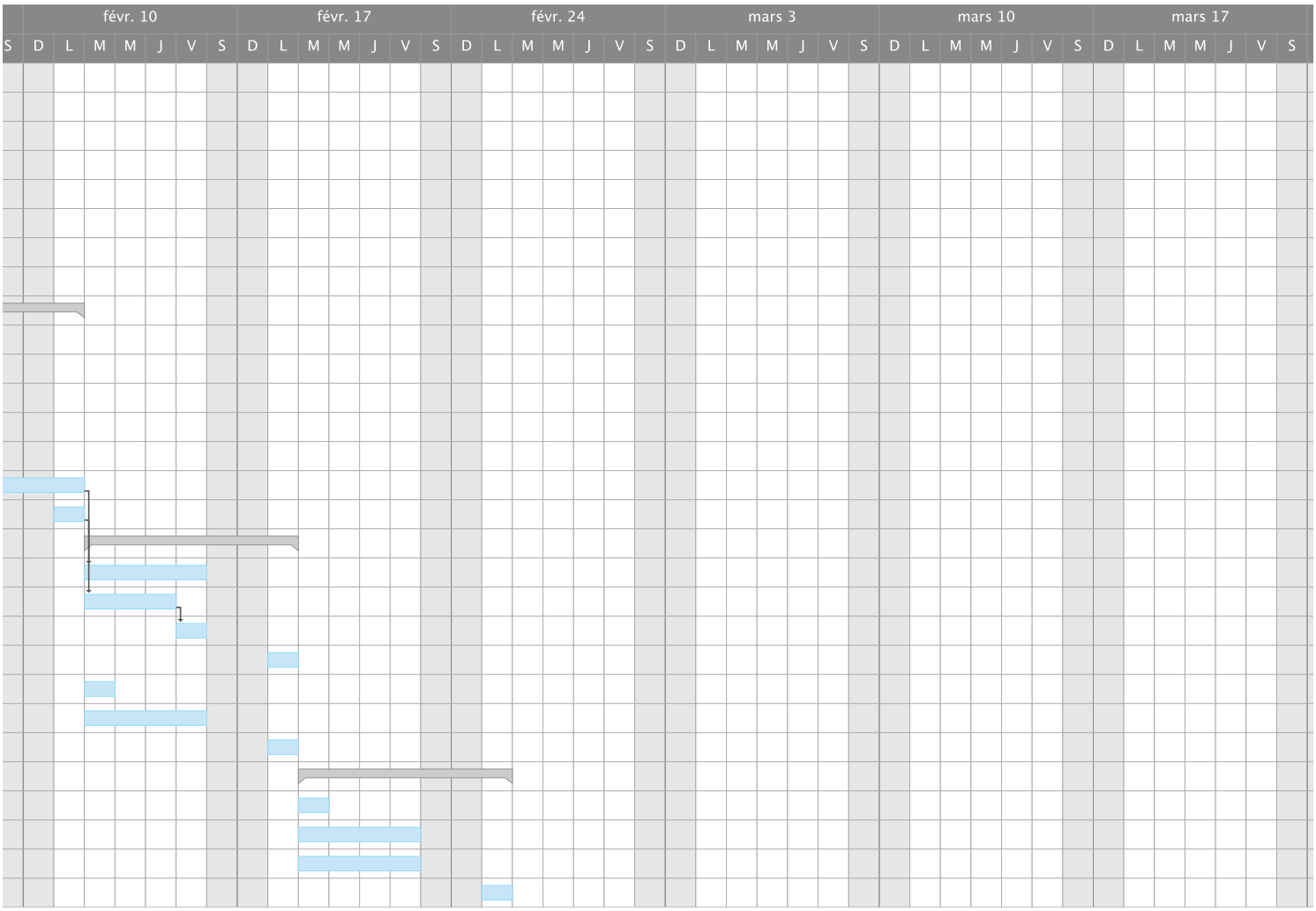

mars 10
mars 17
mars 24
J V S D L M M J V S D L M M J V S D L M M J V S
ion de partition
Sprint 4
Mettre à disposition le réseau entraîné
Permettre l'interruption arbitraire de l'entraînement
Test interruption
Produire durant l'entraînement des fichiers de sons de validation
Test fichier de sons de validation
Annotations pour la musique d'entrainement
Release
Diapo pour la soutenance
Rédaction Rapport
Vérification Code + commentaires + syntaxe

Nouvelle feuille

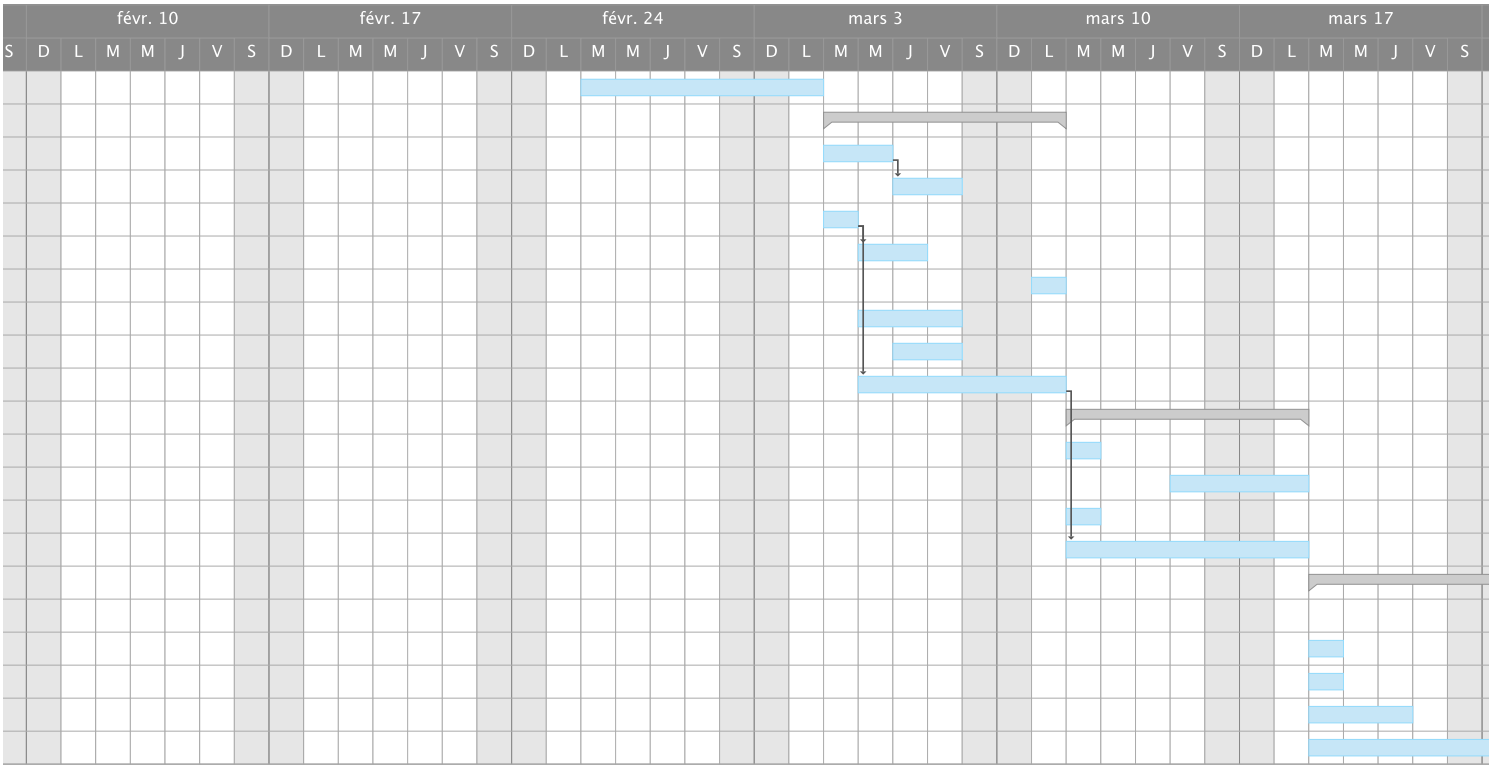
smartsheet

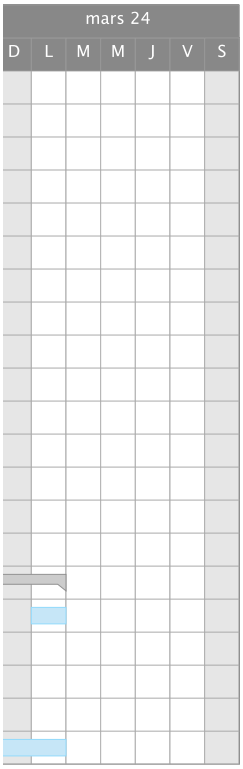


	Nom de la tâche
1	**Préparation Projet/Rédaction cahier des Charges, Premier Sprint (22/01-05/02)**
2	Création du dépôt git
3	Rédaction des cas d'usage
4	Rédaction des besoins fonctionnels
5	Rédaction des besoins non fonctionnels
6	Rédaction des tests
7	Installation et test SampleRNN
8	Rédaction procédure installation SampleRNN
9	**Sprint 2 (05/02-12/02)**
10	Correction cachier des besoins
11	Correction du gantt
12	Lectures d'articles (char2wav, umtn, conditional end to end audio transforms)
13	Résolution de problème de lancement en ssh
14	Recherche de bibliothèque de lecture de midi + création d'un module
15	Début implémentation interpreteur midi
16	Installation dynet
17	**Sprint 3 (12/02-19/02)**
18	Poursuite installation dynet
19	Implémentation module midi to spec
20	Test module midi to spec
21	Implémentation module midi to tune
22	Recherche de bibliothèque pour passer de midi to wav en python
23	Recherche architecture
24	Début écriture rapport
25	**Sprint 4 (19/02-26/02)**
26	Rédaction procédure test unitaires
27	Implémentation interpreteur midi
28	Recherche et lecture de code pour modification de sampleRNN
29	Rédaction schéma architecture basée sur SampleRNN


févr. 10
févr. 17
févr. 24
mars 3
mars 10
mars 17
S D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S

	Nom de la tâche	janv. 20							janv. 27							févr. 3						
		D	L	M	M	J	V	S	D	L	M	M	J	V	S	D	L	M	M	J	V	
30	Vacances (26/02-05/03)																					
31	**Sprint 5 (06/03-12/03)**																					
32	Installation Clarinet																					
33	Test d'entrainement Clarinet																					
34	Recherche Datasets																					
35	Création Datasets																					
36	Modification Code Clarinet (Teacher)																					
37	Commentaires de code pour clarinet																					
38	Création scripts renommage et sélection pour datasets + script pour csv																					
39	Implémentation module dataset (méthode 1 et 2)																					
40	**Sprint 6 (12/03-19/03)**																					
41	Procédure installation (pour clarinet)																					
42	Création diaporama pour soutenance																					
43	Ajout et modification pour le rapport																					
44	Implémentation 3 méthodes pour découpe de datasets																					
45	**Sprint 7 (19/03-26/03)**																					
46	Nettoyage code (verification commentaire, syntaxe...)																					
47	Ajout et modification du rapport																					
48	Modification code Clarinet (Student)																					
49	Correction diaporama pour soutenance																					
50	Tests pour Clarinet																					


févr. 10
févr. 17
févr. 24
mars 3
mars 10
mars 17
S D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S D L M M J V S


mars 24
D
L
M
M
J
V
S