Fine-Tuning de Modèles de Langage pour la Génération Musicale

Groupe MAP07

Avril 2024

Abstract

La génération de musique assistée par ordinateur a connu récemment des avancées significatives grâce aux progrès de l'intelligence artificielle. Le perfectionnement des *Large Language Models*(LLM) permet d'envisager s'en servir pour générer de la musique à travers une représentation textuelle. Cet article présente une méthode de génération par *fine-tuning* de *Llama-2-7b* et de *Mistral-7B-v0.1* passant par la notation *Simplified MIDI*. En partant des bases de données *GiantMIDIPiano* et *DadaGP* converties dans notre syntaxe, nous obtenons deux modèles spécifiquement conçus pour la génération de morceaux de piano et de guitare. Les évaluations subjectives démontrent que les modèles parviennent à générer des réponses structurées et spécifiques à l'amorce fournie, pouvant aider les musiciens à trouver de l'inspiration pour leurs compositions. L'ensemble du code du projet est disponible sur GitHub à l'url : https://github.com/fegounna/LLM-Fine-Tuning-for-Music-Generation

Mots clés: Génération de Musique, Grand modèle de langage, Fine-tuning.

1 Introduction

1.1 État de l'art

Depuis les premières expérimentations dans les années 1980 jusqu'aux développements les plus récents, la génération de musique assistée par ordinateur a connu une évolution remarquable, propulsée par les progrès rapides dans le domaine de l'intelligence artificielle et de l'apprentissage automatique. Au fil des décennies, des chercheurs et des ingénieurs ont exploité un large éventail de techniques, allant des réseaux de neurones aux modèles de transformation de séquences, pour créer des systèmes capables de composer de la musique dans une variété de styles et de genres.

Dans les années 1980 et 1990, les premières incursions dans l'automatisation de la composition musicale ont été marquées par des projets pionniers tels que HARMONET[18], qui utilisait des réseaux neuronaux pour harmoniser des chorales dans le style de Johann Sebastian Bach. Les années suivantes ont vu l'émergence de techniques basées sur les réseaux de neurones, telles que la composition musicale par prédiction[28].

Le début des années 2000 a été caractérisé par une exploration plus approfondie de la structure temporelle de la musique, avec des travaux comme celui d'Eck et Schmidhuber sur l'improvisation blues avec des réseaux récurrents LSTM (Long Short-Term Memory)[12]. Puis, dans les années 2010, des modèles comme DeepBach [16] et MorpheuS[17] ont ouvert la voie à la génération de musique structurée avec des motifs contraints et une attention particulière à la polyphonie.

L'avènement des GAN (réseaux antagonistes génératifs) a également marqué un tournant dans la génération de musique, avec des projets comme Polyphonic GAN[14] ou la composition musicale avec LSTM, permettant une exploration plus riche des possibilités créatives.

Enfin, les avancées les plus récentes, telles que le Bar Transformer[29] et FIGARO[34], démontrent une capacité croissante à contrôler et à modeler le processus de génération de musique, ouvrant ainsi la voie à de nouvelles possibilités dans la création musicale assistée par ordinateur. Le compositeur peut maintenant influer sur la génération de musique et ainsi l'œuvre devient une coécriture du compositeur et de l'intelligence artificielle.

En outre, des projets récents comme SingSong [11] ont ouvert de nouvelles perspectives passionnantes en matière de génération musicale, en se concentrant spécifiquement sur la création d'accompagnements musicaux à partir de performances vocales. De nouveaux projets ne cessent de se créé comme MusicLM[3] qui permet de générer de la musique à partir de paroles déjà écrites.

1.2 L'originalité de notre approche

Les progrès récents et phénoménaux des modèles de langage permettent désormais d'envisager faire du fine-tuning sur des modèles de fondation, en adaptant en format texte la notation musicale.

Or, c'est le format *MIDI* qui a été l'option la plus récurrente en termes de formats de notation musicale, en ce qui concerne les ensembles de données publiés au sein de la communauté MIR (Music Information Retrieval), que ce soit dans le cadre de la génération musicale, qui a récemment explosé en utilisant des approches d'apprentissage profond, ou dans le but de l'analyse musicale, de la musicologie ou de la simple récupération d'informations. Les produits commerciaux les plus perfectionnés du marché, comme MuseNet de OpenAI et Google Magenta Studio sont d'ailleurs tous centrés sur cette notation.

Pour passer du format *MIDI* au texte, nous utilisons, sous les conseils de notre tuteur, qui a publié sur le sujet[15] une notation appelée *Simplified MIDI* sur laquelle nous reviendrons.

Cependant, nous voulions générer des solos de guitare, ce qui nécessite une plus grande variété de sons que pour le piano. Il nous a donc fallu légèrement complexifier la notation *Simplified MIDI* afin avoir un bon rendu.

Notre base de donnée est issue d'un format de tablature appelé *GuitarPro*, très largement utilisé par les guitaristes et les bassistes, mais aussi par les groupes musicaux. Il procède d'une tout autre approche de la notation musicale : par tablature (notation prescriptive), indiquant directement l'action à faire sur l'instrument par opposition aux partitions sur portées (notation descriptive). Des outils de transcription existent avec le format *MIDI*, qui malgré leurs imperfections nous ont suffit pour prendre en compte un certain nombre d'effets sonores spécifiques de la guitare dans notre format *Simplified MIDI*.

Cette approche n'a jamais été explorée d'après notre connaissance, la plupart des chercheurs travaillant avec des notations en tablature préférant générer directement dans leur format. Nous pensons que la perte d'information pendant les différentes transcriptions est compensée au moins en partie par la simplicité et la régularité de notre notation ce qui est très important puisque nous ne choisissons pas la *tokenisation* du modèle de fondation. Si nous ne partions pas d'un modèle très puissant et pré-existant, ces transcriptions seraient en effet purement néfastes et nous gagnerions à bien choisir la *tokenisation* pour qu'elle soit directement adaptée au format *GuitarPro*.

2 Constitution de la base de Donnée en MIDI

Notre approche pour constituer le dataset de *MIDI* a complètement évolué au cours du projet. Si nous avons commencé par envisager faire du scraping, la prise en compte de paramètres spécifiques à la guitare et l'accès à des bases de données suffisamment étendues ont fait que nous nous sommes finalement rabbatus sur une approche visant plutôt à convertir le plus proprement possible en format *MIDI* des bases existantes (*GuitarSet* puis *DadaGP*)

2.1 Approche initiale

Pour construire notre base de donnée initiale en format *MIDI*, nous voulions nous inspirer de la méthode de `GiantMIDIPiano`[23], un projet qui a réussi à constituer une base de donnée de solos de piano très conséquente. Il s'agit donc de faire du web scraping à partir de fichiers audios de YouTube. Pour cette méthode, deux étapes étaient nécessaires: récupérer l'audio sur Youtube et le convertir en format MP3, puis dans un deuxième temps, de le convertir en format *MIDI*. Nous utilisions pour cela `FFmpeg`, une collection de logiciels libres permettant de traîter les flux audios.

Si la conversion en format MP3 s'avère simple, il en est tout autrement pour celle de MP3 à *MIDI*. Au passage, s'il y avait une méthode universelle pour passer de MP3 à *MIDI* en conservant toute l'information nécessaire, cela rendrait le commerce des partitions numériques complètement obsolète. Parmi les problèmes qui se présentent, l'un des principaux est la régularité rythmique : les enregistrements s'écartent toujours quelque peu de la régularité parfaite, et il peut s'avérer très difficile pour un logiciel de reconstituer la vraie rythmique à partir de l'enregistrement (la note dure-elle 1/4 de temps ou 5/16 de temps quand elle est jouée pendant une durée de 5/16 ?). Ce problème ne nous concerne pas véritablement, car nous n'avons pas tant l'ambition de créer des partitions parfaites que d'aider le compositeur, qui peut s'accomoder de ce genre de variation, dans son travail créatif (surtout s'il utilise la génération sous sa forme audio). Le problème qui lui nous concerne beaucoup plus est celui de la précision du signal converti au niveau de la hauteur des notes. En effet, aucun instrument n'émet de sinusoïdes parfaites, et chacun a son profil d'harmoniques. Dans ce cas, se pose le problème de distinguer un Do3 seul d'un accord Do3 + Do4 par exemple. Dans les faits, les outils de conversions en *MIDI* sont très mauvais à cela, et donnent un signal complètement brouillé. Les convertisseurs non entraînés ont également beaucoup de difficulté à distinguer des notes répétées d'une note maintenue c'est-à-dire par exemple à distinguer deux noires consécutives et identiques d'une blanche.

Il faut donc aider à la conversion, avec des moyens spécifiques pour chaque instrument. `GiantMIDIPiano`, qui comme son nom l'indique se concentre sur du piano seul, utilise une librairie python qui s'intitule `Piano Transcription Inference`[24]. Cette librairie utilise de l'intelligence artificielle pour inférer, à

partir du son, quel est véritablement la (ou les) note(s) jouée(s), **en présumant que l'instrument est du piano.** Avant de réfléchir à la conception d'un tel outil pour notre usage spécifique (c'est-à-dire des solos de guitare, de Blues de préférence), nous avons voulu nous assurer de sa nécessité, en appliquant le modèle spécifique au piano sur un enregistrement de guitare, et une conversion en *MIDI* classique. Voilà la comparaison des résultats :

Nous représentons les audios sous leur forme dite "piano-roll".

Figure 1: Audio de **piano traité par un convertisseur naïf** : Il rajoute beaucoup de notes, qui sont des harmoniques de la note jouée, et fragmente également les notes qui ont une longue durée.

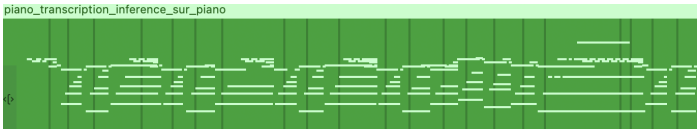


Figure 2: Audio de **piano traité par un convertisseur spécifiquement entraîné sur le piano** : Nous ne constatons aucun des deux effets signalés pour le convertisseur naïf, la conversion est de qualité.

Au vu des figures 1,2,3 et 4, la **nécessité d'un outil de conversion spécifique à la guitare** saute donc aux yeux pour espérer faire du scraping. Alerté par l'absence de librairies faisant ce genre de scraping sur de la guitare, nous avons sondé plus profondément la littérature de la *Music Information Retrieval*(MIR) lié à cet instrument. L'approche y est différente parce qu'un morceau de guitare, contrairement au piano, ne peut pas se réduire à un alignement de note discrètes : en effet, malgré la présence de *fret*, les cordes peuvent être tirées orthogonalement à l'axe du manche pour faire ce que l'on appelle un *bend*. Et cela contrarie très fortement la capacité à passer d'un audio à un *piano roll* (Comment différencier un Do augmenté d'un demi-ton d'un Ré diminué d'un demi-ton ? Comment rendre compte de la continuité).

Aussi, nous avons décidé de rejoindre cette approche consensuelle dans la génération de musique de guitare et d'abandonner l'idée de créer notre propre scraper.

Temporairement, pour que la partie training puisse avancer indépendamment, nous avons décidé de tester notre modèle de génération sur la base de donnée de GiantMIDIPiano directement, Une fois celle-ci consistante et performante,

Figure 3: Audio de **guitare traité par un convertisseur naïf** (l'audio original est un thème à une seule voix) : Les mêmes effets que pour le piano sont présents, avec une démultiplication des notes, tant par l'ajout des harmoniques que par leur fragmentation temporelle.

Figure 4: Audio de **guitare traité par un convertisseur entraîné sur le piano** : L'entraînement diminue beaucoup la tendance à démultiplier les harmoniques, cependant, on ne la supprime pas totalement (rappelons que l'audio consiste en une seule voie). Au niveau de la fragmentation temporelle cependant, on ne fait que très peu de progrès. Le convertisseur marche moins bien que sur du piano.

nous pourrons essayer d'introduire nos propres données et voir si la génération reste de qualité.

2.2 *GuitarSet* puis *DadaGP*

Nous avons commencé par utiliser les bases de données en libre accès. La base de donnée *GuitarSet* nous semblait prometteuse par sa précision et sa facilité à la convertir en *MIDI*. Malgré sa taille réduite (360*30 secondes d'enregistrement), nous avons d'abord pensé que cela suffirait, quitte à faire un premier training sans *bend* et un deuxième très spécifique sur le bend. Le format utilisé dans l'encodage utilise la bibliothèque python *JAMS*[21](JSON Annotated Music Specification), disposant d'un convertisseur en *MIDI* plutôt transparent même s'il nous a dans un premier temps posé beaucoup de problèmes d'implémentation.

Après discussion avec notre tuteur de stage au sujet du caractère un peu limité de ce dataset, il nous a orienté vers Alexander D'Hoodge, un doctorant un génération musicale, qui s'intéresse de près aux tablatures de guitare[10]. Celui-ci nous a réorienté vers la base de donnée *DadaGP* [], dans le format *GuitarPro*, extrèmement répandu dans le milieu des guitaristes. Celle-ci n'est accessible que sur demande, mais après avoir contacté son auteur principal, Pedro Sarmento, nous avons pu y accéder.

Cette base est véritablement massive, avec plus de 26 000 morceaux. Aussi

nous avons eu la possibilité de nous restreindre à un type de musique spécifique : le *hard metal*, ce qui la réduisait considérablement. Pour sélectionner les morceaux de *hard metal*, nous avons parcouru à la main le Dataset en sélectionnant le plus souvent au niveau du nom des groupes, réputés pour leur répertoire de métal. Tous les dossiers dont nous n'arrivions pas à déterminer le style musical ont été écartés par prudence.

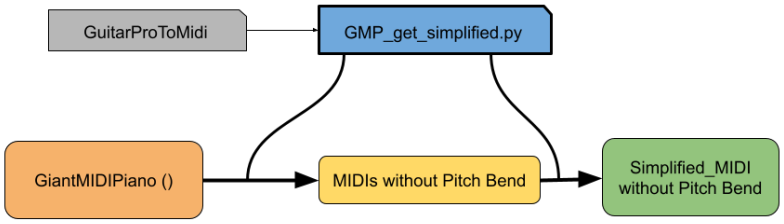


Figure 5: Solution temporaire pendant la confection du dataset de guitare. Les encadrés bleus représentent notre code

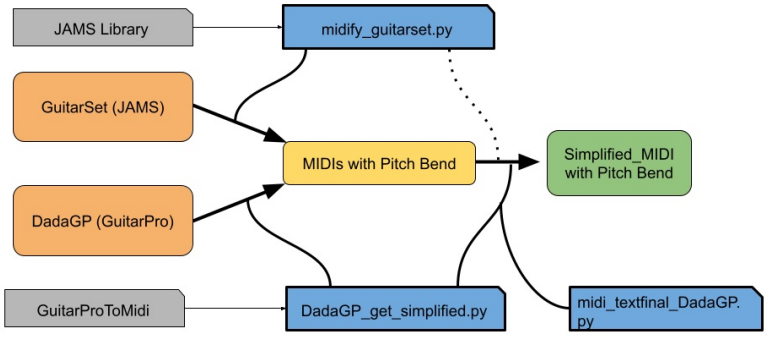


Figure 6: Graphe synthétisant la démarche de constitution du dataset du guitare en Simplified *MIDI*. (Le lien en pointillé n'a finalement pas été réalisé, puisque *DadaGP* est devenu notre source prioritaire.)

Cependant, *GuitarPro* est un format beaucoup plus complet que *MIDI*, aussi certaines notations ne connaissaient pas de conversion et nous avons dû nous passer de ces fichiers (nous aurions pu les supprimer, mais puisque nous en avions assez, nous avons préféré ne pas toucher aux morceaux initiaux, ignorant véritablement l'impact de nos modifications).

Pour convertir en *MIDI*, nous avons utilisé *GuitarProToMidi*[1] un programme open source de transcription en *MIDI*. Nous obtenons ainsi une base en *MIDI* de 2692 fichiers, de plusieurs minutes chacun.

DadaGP fournit des morceaux complets (bien souvent plusieurs guitares, une basse, une batterie...), qui sont bien séparables dans des pistes différentes. Nous souhaitions initialement récupérer toutes les pistes de guitare qui avaient du *bend* , et uniquement celle-ci. Cependant, beaucoup de piste n'ont pas de référence directe à la guitare, et sont intitulées d'innombrables façon différentes, comme par exemple par le nom du guitariste. Aussi, nous avons préféré récupérer toutes les pistes qui possèdent du *bend* (dans le fichier *DadaGP_bended_track_names.txt*), récupérant ainsi toutes les voies de guitare basse, maisi aussi les quelques pistes de violon et de voix. En effet, l'alternative, qui était de ne sélectionner que les pistes qui contenaient la chaine de caractère "Guitar" ou "guitar", nous conduisait à abandonner plus des deux tiers des pistes de guitare. Nous avons donc fait le pari que l'incorporation des autres instruments (la référence à l'instrument est supprimée dans la constitution de la base de donnée d'entraînement) ne modifierait pas trop le comportement du modèle.

3 Data Augmentation

Dans le cadre de la création d'une base de données d'une taille conséquente afin de mieux pouvoir fine-tuner notre modèle, nous avons dû procéder à de la data augmentation sur les fichiers *MIDI* existants. Nous allons ici détailler le processus de DATA Augmentation pour des fichiers *MIDI*, mais également justifier la pertinence des méthodes de Data Augmentation d'un point de vue musical. Afin de dupliquer nos fichiers, nous disposons ici de 4 grands axes de modifications :

- Modification de la hauteur (pitch en *Simplified MIDI*) de 2 manières que nous allons détailler plus loin (Transposition et modulation)
- Modification du tempo
- Modification des nuances (Velocity en *Simplified MIDI*)

Avant d'expliquer le processus permettant ces modifications, nous allons justifier d'un point de vue musical le choix de ces axes de travail.

Tout d'abord, la modification de tempo (vitesse du morceau) semble être la modification la plus simple à comprendre : à partir d'un morceau de base, nous créons simplement le même morceau mais plus rapide ou plus lent, en fonction du facteur multiplicatif choisi. Ainsi, on peut, sans trop affecter la dynamique du morceau, créer une dizaine de partitions à partir d'une seule. D'un point de vue musical, une augmentation ou diminution de tempo peut changer l'ambiance, l'esprit du morceau et en particulier la perception que l'auditeur s'en fait. Il est donc intéressant d'avoir dans la base donnée deux mêmes morceaux à des tempos différents.

La variation de nuances se justifie également assez aisément. En effet, tout morceau de musique est composé de nuances, qui représentent la force avec laquelle les notes sont jouées. Ceci peut donner des effets de variations de l'intensité du morceau, notamment par l'utilisation d'augmentation progressive de l'intensité (crescendo) ou inversement (decrescendo). Ainsi, en modifiant de manière aléatoire et dans un intervalle modéré l'intensité à laquelle chaque note de notre morceau est jouée, on va pouvoir créer de nombreux fichiers différents à partir d'un seul. Cette pratique de "randomisation" de la vélocité des notes est assez courante dans la production musicale, en particulier dans le domaine de la production électronique et de la musique assistée par ordinateur (MAO) pour apporter des phrasés différents, voire novateurs. Cela permet aussi d'humaniser l'interprétation des notes et d'ajouter du réalisme aux instruments virtuels.

La modification de la valeur "pitch" nécessite un peu plus d'explications. Dans la musique occidentale depuis le XVIIème siècle et aujourd'hui dans la grande majorité des oeuvres musicales, il existe 12 notes :

$$(Do, Do\#, Ré, Ré\#, Mi, Fa, Fa\#, Sol, Sol\#, La, La\#, Si)$$

Le format *MIDI* reprend naturellement ce système dodécaphonique. Ces 12 notes se succèdent (et se répètent) dans cet ordre, à chaque fois que l'on augmente la hauteur (ou "pitch") de notre fichier *MIDI* de 1 (ainsi, par exemple, la

modification nécessaire pour passer d'un Mi à un Sol est +3, ou -9). La présence de 12 notes permet donc d'obtenir 12 partitions différentes à partir d'une seule.

On notera, de plus, que changer la tonalité implique de changer uniformément la hauteur (ou pitch) de chaque note, ce qui constitue un processus assez simple, dans le sens où il suffit, pour changer de tonalité, d'augmenter la valeur "pitch" de chaque note d'un entier compris entre 1 et 11 (Voire plus, car, même si on retombe sur nos pas en terme de nom de note, on aura tout de même des notes plus aigus et donc potentiellement une source intéressante d'entraînement du modèle). Changer uniformément le "pitch" de chaque note a pour nom la "transposition" de la partition. Ainsi on obtient facilement 12 partitions a priori différentes à partir d'une seule.

De plus, comme nous l'avons déjà mentionné, il existe un second axe de travail sur la hauteur (ou pitch) des notes. En effet, si il est possible, et courant en musique, d'avoir recours à de la transposition, qui consiste à changer totalement la tonalité d'un morceau, on peut également changer la tonalité d'une partie seulement du morceau, on parle alors de modulation. Le processus est le même, il suffit simplement de déterminer quand débuter la modulation et quand la terminer.

En pratique, nous avons décidé d'augmenter nos datasets en utilisant des facteurs de modifications aléatoires dans une plage de valeurs pré-déterminée. Ce choix se justifie par le fait que, d'une part, cela créera un maximum de diversité au sein des datasets ainsi créés (et pas juste une modification uniforme de celui de base). De plus, la pertinence de ce choix, d'un point de vue musical, se justifie par le fait que la modulation d'un morceau de musique, le changement de rythme (tempo) et la variation des nuances sont assez fréquemment observés.

Ci-dessous nous pouvons observer une portion de partition d'un même morceau avant et après lui avoir fait subir des facteurs de modifications aléatoires. En l'occurence on peut observer dans l'exemple que la hauteur, le "pitch" des notes a été augmenté de 11 demi-tons, et que le tempo a été drastiquement réduit.

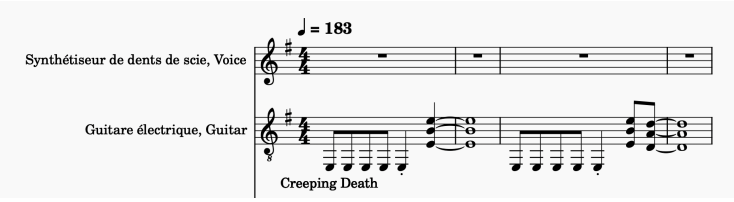


Figure 7: Début de la partition du morceau "Creeping Death" de Metallica

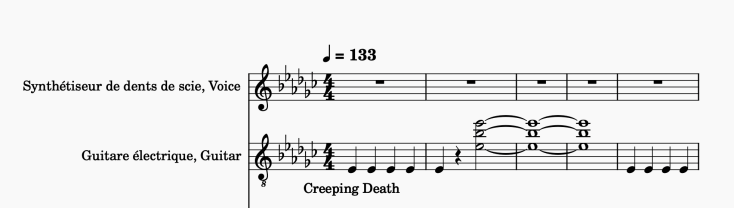


Figure 8: Partition du morceau modifié

4 Formalisme et convertisseurs

4.1 Généralités

Afin d'entraîner notre modèle, il nous faut lui donner un langage qu'elle puisse comprendre. Pour cela, le format le plus répandu est le format *MIDI*. Un fichier *MIDI* (Musical Instrument Digital Interface) est composé d'une séquence de données qui représente des instructions musicales plutôt que des sons réels. Il contient des messages qui indiquent comment jouer une musique, tels que des notes, des changements de volume, des modulations, etc. Les informations dans un fichier *MIDI* sont structurées en événements temporels, décrivant le début, la durée et l'intensité des notes. Ces événements peuvent également inclure des données de contrôle pour gérer des aspects comme la pédale, la modulation et d'autres paramètres. Contrairement aux fichiers audio, les fichiers *MIDI* ne contiennent pas de sons eux-mêmes, mais servent de guide pour des instruments électroniques ou logiciels qui interprètent ces instructions pour produire la musique.

Plus précisément, les messages qui décrivent les instructions musicales sont les suivants :

Note On et Note Off : Ces messages indiquent le début et la fin d'une note, respectivement. Ils comprennent des informations telles que la hauteur de la note (le numéro de la note MIDI), la vélocité (la force avec laquelle la note est jouée), et le canal *MIDI* auquel la note est assignée.

Control Change : Ces messages gèrent divers paramètres de contrôle tels que le changement de volume, la modulation, la pédale d'expression, etc.

Program Change : Indique le changement d'instrument ou de timbre.

Pitch Bend : Permet de moduler la hauteur des notes de manière continue, simulant des variations subtiles de ton.

System Exclusive : Messages spécifiques au fabricant qui peuvent inclure des instructions propres à un équipement ou logiciel particulier.

L'assemblage de ces messages dans une séquence chronologique forme le fichier *MIDI*, permettant ainsi à des dispositifs compatibles de reproduire la musique selon les indications fournies.

Une note dans un fichier *MIDI* est définie par plusieurs paramètres :

Hauteur de la note (Note Number) : Représentée par un numéro de 0 à 127, chaque valeur correspond à une note spécifique sur le clavier, où le numéro 60 est généralement associé à la note "Middle C".

Vélocité (Velocity) : C'est la mesure de la force avec laquelle une note est jouée. La vélocité est exprimée en valeurs de 0 à 127, où 0 représente une note "relâchée" et 127 une note jouée avec la plus grande force.

Durée de la note : Indique la durée pendant laquelle la note est maintenue. Ce paramètre est défini par le moment où le message "Note On" est suivi du message "Note Off" correspondant.

Canal MIDI : Un numéro de canal *MIDI* (généralement de 1 à 16) est attribué à chaque note et est utilisé pour spécifier le canal sur lequel la note est jouée. Cela permet de contrôler plusieurs instruments ou voix simultanément.

L'ensemble de ces paramètres configure une note spécifique dans un fichier *MIDI*, permettant une représentation précise des éléments musicaux dans la séquence.

Pour notre projet, nous pouvons nous affranchir d'un certain nombre de données. D'abord celle du canal *MIDI*, puisque nous n'allons donner en entrée à notre programme que des musiques qui ne comportent qu'une piste, un seul canal *MIDI*. Ensuite, nous avons décider de ne pas prendre en compte les messages comme programm change ou encore control change, car nous cherchons en premier lieu la simplicité dans ce que le programme va générer.

Un fichier *MIDI* contient donc un nombre d'informations assez conséquent, qui ne se résume pas seulement à une succession de notes. Pour rendre la lecture en entrée de la musique plus facile pour notre programme, le format d'entrée ne sera pas un dossier de fichiers *MIDI*, mais *Simplified MIDI*, une version allégée du format précédent, qui va se débarrasser de notions dont nous n'aurons pas besoin comme les messages d'instructions musicales, qui donnent des détails trop précis sur la manière de jouer un morceau, et ne sont pas pertinents pour ce qui nous intéresse in fine, rendre notre modèle capable de générer lui même des compositions en sortie.

4.2 Simplified MIDI

Le format *Simplified MIDI*, proposée dans [15], est une succession de quadruplés de la forme $(px : vy : dz : tw)$ où x, y, z, w sont des nombres et p, v, d, t sont les abbréviations de pitch, velocity, duration et time. Les champs pitch, velocity et duration correspondent aux valeurs dans le format *MIDI* original, le champ time lui représente le temps avant qu'une autre note ne soit jouée. C'est là la principale différence entre les deux formats :

- En *MIDI*, le temps est représenté de manière absolue : chaque évènement est daté par une durée en tick qui correspond directement à un temps en secondes par l'intermédiaire du BPM.
- En *Simplified MIDI*, le temps est représenté de manière relative : la date de début d'un évènement est défini par rapport au début de l'évènement précédent.

Le principal intérêt du format *Simplified MIDI* est qu'il permet de passer d'une représentation du morceau imbriquée et complexe (`NoteOFF` faisant référence à des `NoteON` bien plus haut dans le morceau) à une représentation linéaire que nous pensons le modèle parviendra mieux à comprendre.

Pour parvenir à donner à notre programme en entrée des fichiers de type Simplified *MIDI*, nous avons dû écrire un convertisseur du format *MIDI* vers SimplifiedMIDI et dans l'autre sens.

4.3 Pour la guitare : intégrer le Pitch Bend au *Simplified MIDI*

La façon la plus logique d'incorporer le Pitch Bend dans ce format est de rajouter une cinquième donnée à chaque quadruplet. Nous redéfinissons donc le Pitch Bend au niveau de toute les notes (ce qui n'est pas le cas en *MIDI* où il est traité complètement indépendamment par un évènement différent : le passage de l'un à l'autre est simple à réaliser)

Aussi, le format passe de $(px_1 : vx_2 : dx_3 : tx_3)$ à $(px_1 : vx_2 : dx_3 : tx_3 : bx_4)$ avec le caratère b pour abbréger *bend*. La plage de valeurs pour le *bend* en format *MIDI* $\{0, 1, 2, \ldots, 16383\}$. Une valeur de 8192 signifie que la hauteur de la note n'est pas modifiée, une valeur de 0 (resp. 16383 que la note est diminuée (resp. augmentée) de 2 tons. Cela fournit une incroyable précision pour la prise en compte de la distorsion du son (et qui est nécessaire, rappelons-le car les sons à la guitare peuvent être continus), cependant est plutôt problématique pour notre modèle, qui doit choisir dans une plage immense de valeur. Aussi, sous les conseils de notre tuteur, nous avons procédé à une quantification du Pitch Bend, qui nous fait perdre en précision, mais réduit la plage de valeur.

Cependant, certaines *features* liées au Pitch Bend, comme le vibrato, jouent sur des plages de *bend* relativement faibles, et plutôt proches de 8192. Aussi, il ne faut pas quantifier avec une précision égale les valeurs proches du "zéro-distorsion" que les valeurs qui en sont loin.

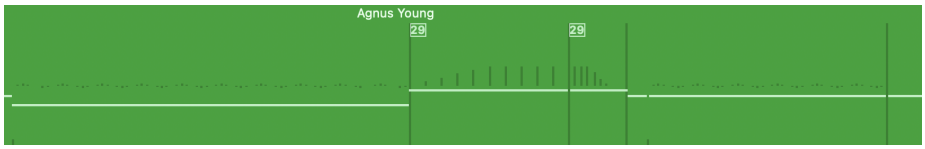
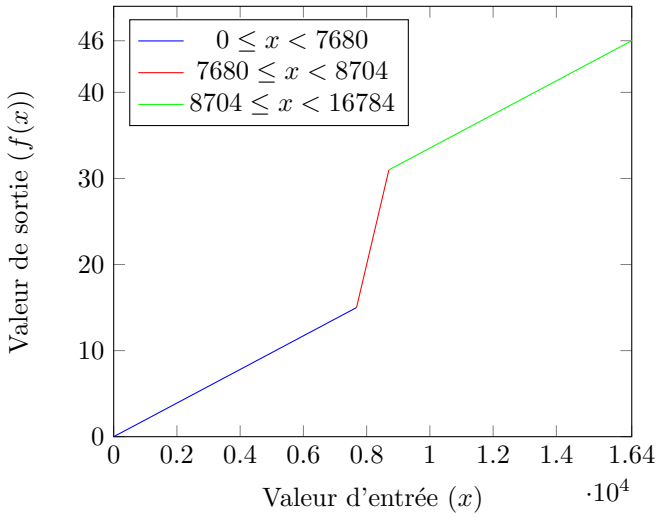


Figure 9: Music roll d'une piste de guitare. Les barres verticales grises représentent le *bend* à un instant donné. On remarque que deux notes tenues avec un vibrato encadrent un passage avec des distorsions beaucoup plus fortes.

Nous proposons donc, pour pallier à la nécessité d'avoir une plus grande précision d'encodage proche de 8192, une fonction linéaire par morceau, choisie pour sa simplicité.

La fonction $f : \mathbb{N} \to \mathbb{N}$ définie par :

$$f(x) = \begin{cases} \lfloor \frac{x}{512} \rfloor, & \text{si } x < 7680 \\ 15 + \lfloor \frac{x-7680}{64} \rfloor, & \text{si } 7680 \leq x < 8704 \\ 31 + \lfloor \frac{x-8704}{512} \rfloor, & \text{si } 8704 \leq x < 16784 \end{cases}$$



Pour inverser cette quantification après la génération du modèle, nous utilisons l'inverse (approximative):

$g : \mathbb{N} \to \mathbb{N}$ définie par :

$$g(x) = \begin{cases} 512x, & \text{si } x < 15 \\ 7680 + 64(x - 15), & \text{si } 15 \leq x < 31 \\ 8704 + 512(x - 31), & \text{si } 31 \leq x < 46 \end{cases}$$

Ainsi, nous avons approximativement un tiers de la plage de valeurs consacrée aux *bends* très faibles, un tiers aux *bends* très forts, et un tiers pour les *bends* proche du zéro-distorsion.

5 LLM

5.1 Objectif

Nous avons choisi d'utiliser des large langage models (LLMs) comme cadre fondamental, en raison de leur performance exceptionnelle dans diverses tâches de traitement du langage naturel. Notre modèle est entraîné sur une combinaison de datasets d'instructions. Chaque exemple dans le dataset consiste en une instruction et un output. Le modèle est prompté avec l'instruction et entraîné à générer l'output de manière autoregressive. [5]

La fonction de perte est uniquement calculée sur la partie output et peut être exprimée comme suit :

$$\mathcal{L}_{\mathrm{SFT}}(\Theta) = \mathbb{E}[-\sum_{i \in \{output\}} \log p(x_i | x_0, x_1, \ldots, x_{i-1}; \Theta)] \tag{1}$$

Où , Θ représente les paramètres du modèle, et $x = (x_0, x_1, \ldots)$ représente la séquence d'input tokenisée.

5.2 Structure d'un LLM

La structure des LLMs suit celle qui a été originellement proposée dans [33] mais n'est composée que d'un décodeur dans la plupart des cas. Nous revenons rapidement sur la structure de ces modèles.

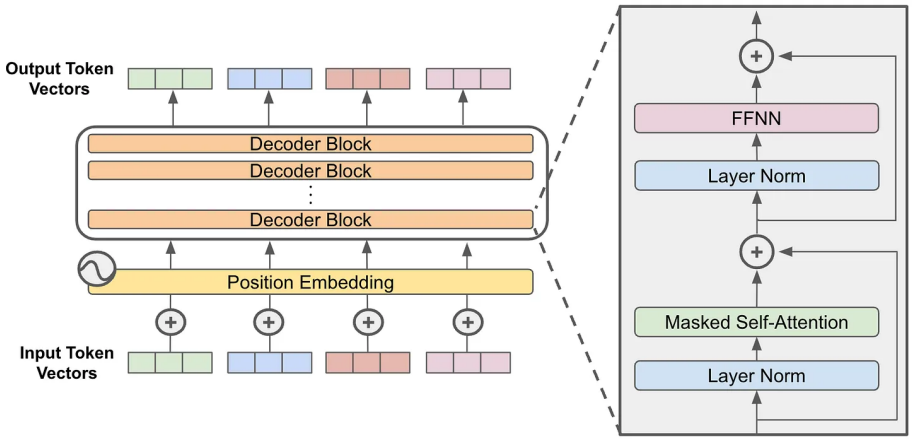


Figure 10: Structure d'un LLM, Source : https://cameronrwolfe.substack.com/

Le texte en entrée du modèle est tokenisé et on y ajoute des "positionnal embeddings" afin que le modèle dispose d'information sur l'ordre des tokens en entrée. Ces tokens sont ensuite projetés sur 3 sous-espaces via les matrices apprises Q, K et V.

L'attention peut ensuite être calculée :

$$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^{\text{T}}}{d_k})V$$

L'attention calculée est légèrement modifiée en fixant les valeurs de QK^{T} au-dessus de la diagonale à $-\infty$ ce qui empêche les tokens à une position donnée d'être influencés par les tokens suivants. Par ailleurs, en pratique ce calcul est décomposée en plusieurs "Attention heads" : après avoir appliqué plusieurs projections apprises aux matrices Q, K et V les différents scores d'attention sont calculés puis concaténés.

Les tokens obtenus sont enfin passés dans un réseau de neurones à propagation avant puis décodés pour obtenir le résultat généré. Le token nouvellement généré est alors passé en entrée et on itère le procédé.

5.3 LoRA

Le paradigme d'entraînement traditionnel qui met à jour tous les paramètres des LLMs est extrêmement coûteux et souvent pas réalisable pour beaucoup d'experiences, ni en termes de temps ni de coûts. Low-Rank Adaptation (LoRA) [20] offre une méthode d'entraînement efficiente en paramètres en conservant les poids du modèle pré-entraîné tout en introduisant des matrices de décomposition de rang faible entraînables. LoRA fige les poids du modèle pré-entraîné et insère des matrices entraînables de bas rang dans chaque couche. Cette méthode réduit considérablement le nombre total de paramètres entraînables, rendant l'entraînement des LLMs possible avec beaucoup moins de ressources de calcul.

Pour une couche linéaire avec une matrice de poids $W_0 \in \mathbb{R}^{d\times k}$, où k est la dimension d'entrée et d la dimension de sortie, LoRA ajoute deux matrices entraînables décomposées de bas rang $B \in \mathbb{R}^{d\times r}$ et $A \in \mathbb{R}^{r\times k}$, avec r étant le rang prédéterminé. Le passage en avant avec une entrée x est donné par l'équation suivante :

$$h = W_0x + \Delta Wx = W_0x + BAx$$

Durant l'entraînement, W_0 est gelé et ne reçoit pas de mises à jour de gradient, tandis que B et A sont mis à jour. En choisissant le rang $r \ll \min(d,k)$, la consommation de mémoire est réduite car il n'est pas nécessaire de stocker les états de l'optimiseur pour la grande matrice gelée W_0.

5.4 Quantification des LLM

Malgré l'utilisation de techniques comme LoRA pour diminuer les ressources de calculs nécessaires à l'entraînement du modèle, il reste indispensable de charger le modèle en mémoire ce qui peut impliquer un coût important, et ce particulièrement pour des LLMs qui se caractérisent par leur grand nombre de paramètres (7B dans notre cas).

Une idée pour pallier cette difficulté est de "quantifier" le modèle, c'est à dire diminuer la précision avec laquelle on stocke les poids de celui-ci.

5.4.1 Techniques de quantifications "naïves"

De manière générale, les poids des modèles sont des `FP16` ou `FP32`. Les techniques de quantifications consistent à convertir ces poids vers des types de données plus léger en mémoire. Nous décrivons ci-après les deux principales méthodes élémentaires de quantification.

1. Absmax quantization Une première méthode de quantification consiste à simplement renormaliser les poids d'une matrice d'un type de données dt_1 afin qu'il corresponde à l'étendue d'un type de données dt_2 puis à arrondir les valeurs obtenues vers l'élément de type dt_2 le plus proche :

$$\mathbf{X}_{dt_2} = \left\lfloor \frac{||dt_1||_\infty \mathbf{X}_{dt_1}}{||\mathbf{X}_{dt_1}||_\infty} \right\rceil_{dt_2}$$

2. Zeropoint quantization La quantification *Absmax* ne permet pas nécessairement d'utiliser la totalité des valeurs du type de données vers lequel on quantifie. La quantification *Zeropoint* apporte une réponse à ce point. En reprenant les notations introduites ci-dessus, on définit :

$$r_{dt_1} = \frac{\max_{dt_2} - \min_{dt_2}}{\max(\mathbf{X}_{dt_2}) - \min(\mathbf{X}_{dt_2})}$$

$$z_{dt_1} = r_{dt_1} \min(\mathbf{X}_{dt_2})$$

$$\mathbf{X}_{dt_1} = \lfloor r_{dt_1} \mathbf{X}_{dt_2} \rceil$$

La multiplication de deux nombres quantifiés A et B de zeropoint z_A et z_B est alors définie par :

$$A_{dt_1} \times_{dt_1} B_{dt_1} = (A_{dt_1} + z_A)(B_{dt_1} + z_B)$$

Pour déquantifier, il suffit alors de diviser par les deux facteurs de normalisation r_A et r_B. On remarque que dans les deux cas, on n'a pas de perte de précision pour la valeur maximale de la matrice quantifiée.

5.4.2 Quantification par blocs et par vecteurs

Le principal problème de ces méthodes de quantifications reposent dans leur sensibilité aux valeurs aberrantes. En effet, la présence d'une valeur de grande magnitude dans les poids de la matrice entraîne une grande perte de précision après la quantification. Ce problème est probablement une des principales raisons qui rendent la quantification inefficace pour les modèles de plus de 6 milliards de paramètres. En effet, il a été observé ([7]) que la précision des modèles quantifiés chutent brutalement pour plus de 6.7B de paramètres et que cette chute correspond à l'émergence de poids de grande magnitude.

1. Quantification par blocs

Une première façon de s'attaquer à ce problème est de ne plus quantifier la matrice en une seule fois mais plutôt de la décomposer en plusieurs blocs puis de quantifier ces blocs. L'erreur introduite par une valeur aberrante se répercute alors uniquement sur un seul bloc plutôt que sur la matrice entière.

Formellement, la méthode consiste à considérer une matrice $\mathbf{X} \in \mathbb{R}^{s \times h}$ comme un élément de $\mathbb{R}^{sh}$ puis à la découper en n blocs successifs que l'on quantifie à l'aide d'une des deux méthodes élémentaires ci-dessus. On obtient donc n constantes de quantifications c_i^1. Afin de diminuer encore l'espace mémoire nécessaire, il est possible de quantifier les constantes de quantification pour obtenir c_i^2

2. Quantification par vecteurs
Les poids de grandes magnitudes qui apparaissent dans les modèles de plus de 6.7B de paramètres se concentrent dans un nombre limité (moins de 7) de colonnes/"feature dimensions" des états cachés Une méthode de quantification adaptée à cette observation est la quantification par vecteurs. L'idée est de ne pas quantifier les colonnes des états cachés et les lignes correspondantes dans les matrices de poids afin de conserver une précision maximale sur ces valeurs.

Formellement, étant donné les états cachés $\mathbf{X}_{dt_1} \in \mathbb{R}^{s \times h}$ et la matrice des poids $\mathbf{W}_{dt_1} \in \mathbb{R}^{h \times o}$ et en notant $\mathcal{O}$ l'ensemble des colonnes de grande magnitudes, on quantifie les lignes de X_{dt_1} et les colonnes de $\mathbf{W}_{dt_1}$ n'apparaissant pas dans $\mathcal{O}$ pour obtenir :

$$\mathbf{X}^h_{dt_2} = \mathcal{Q}(\mathbf{X}^h_{dt_1}) \text{ et } \mathbf{W}^h_{dt_2} = \mathcal{Q}(\mathbf{W}^h_{dt_1}) \text{ pour } h \notin \mathcal{O}$$

Et les constantes de quantifications associées :

$$\mathbf{c}_{\mathbf{x}dt_1} \in \mathbb{R}^s \text{ et } \mathbf{c}_{\mathbf{w}} \in \mathbb{R}^o$$

Où $\mathcal{Q}$ représente la quantification *Absmax* ou *Zeropoint*.

Le produit $\mathbf{WX}$ se calcule alors ainsi :

$$\mathbf{X}_{dt_1}\mathbf{W}_{dt_2} \approx \sum_{h \in \mathcal{O}} \mathbf{X}^h_{dt_1}\mathbf{W}^h_{dt_1} + [\mathbf{c}_{\mathbf{x}} \otimes \mathbf{c}_{\mathbf{w}}]^{-1} \sum_{h \notin \mathcal{O}} \mathbf{X}^h_{dt_2}\mathbf{W}^h_{dt_2}$$

Les features de grande magnitude ne représentant qu'environ 0.1% des features ([8]), on garde ainsi un fort gain en mémoire tout en ayant une précision maximale sur les poids de grande magnitude.

5.5 Un nouveau type de données : `NormalFloat4`

Les méthodes de quantifications décrites plus haut sont utilisables pour de nombreux types de données mais les types de données les plus courants restent `FP16`, `32` et `BF16` pour le modèle non quantifié et `int8` ou `int4` pour le modèle quantifié.

Une autre approche pour améliorer la précision du modèle quantifié serait de définir un nouveau type de donnée ad-hoc adapté à la distribution des poids du modèle considéré. C'est ce qui est fait par Tim Dettmers et al. ([9]) qui partent de l'observation que la distribution des poids des modèles de langages les plus répandus est bien approximée par une gaussienne. Ils définissent alors un nouveau type de donnée `NormalFloat4` (ou `NF4`) à 4 bits qui offre une précision optimale pour quantifier des valeurs distribuées selon une gausienne centrée réduite.

5.6 QLoRA

QLoRA (Tim Dettmers et al., [9]) est une technique de PEFT optimisée pour le fine-tuning de LLM quantifiée. Elle utilise un type de donnée de stockage (`NF4`) pour les paramètres du modèle quantifié et un type de donnée pour les calculs (`BF16`) ce qui permet de combiner à la fois le gain en mémoire obtenu par la quantification du modèle de base et un fine-tuning précis des couches linéaires ajoutées.

Formellement on a, pour une seule couche linéaire :

$$\mathbf{Y}_{BF16} = \mathbf{X}_{BF16}\text{doubleDequant}(c^1_{FP32}, c^2_{FP8}, \mathbf{W}_{NF4}) + \mathbf{X}_{BF16}\mathbf{L}^1_{BF16}\mathbf{L}^2_{BF16}$$

Où $\mathbf{L^1}$ et $\mathbf{L^2}$ correspondent à la couche LoRA ajoutée et où :

$$\text{doubleDequant}(c^1_{FP32}, c^2_{FP8}, \mathbf{W}_{NF4}) = \text{dequant}(\text{dequant}(c^1_{FP32}, c^2_{FP8}), \mathbf{W}^{NF4}$$

La méthode de quantification utilisée est la quantification par bloc avec des blocs de taille 64 pour la quantification de $\mathbf{W}_{BF16}$ puis des blocs de taille 256 pour la quantification de c^1_{FP32}.

Les poids du modèle sont donc stockés au format `NF4` et déquantifiés vers `BF16` lorsque nécessaire (que ce soit lors de l'inférence ou pendant la mise à jour des paramètres de $\mathbf{L^1}$ et $\mathbf{L^2}$).

5.7 Gradient checkpointing

Typiquement, lors de chaque passage en avant, toutes les activations intermédiaires sont conservées en mémoire, car elles sont nécessaires pour calculer le passage en arrière. Les valeurs des activations intermédiaires restent stockées longtemps en mémoire avant d'être utilisées pour la rétropropagation Le checkpointing des gradients [4] est une technique permettant de réduire la consommation de mémoire en ne conservant qu'un sous-ensemble des activations intermédiaires et en recalculant les autres selon les besoins. Le compromis réside dans le temps de recalcul supplémentaire. Selon `huggingface.co`, empiriquement on constate que le checkpointing des gradients ralentit l'entraînement de 20 %. Le besoin en mémoire pour les activations, avec N couches de modèle, diminue de $O(n)$ à $O(log(n))$.

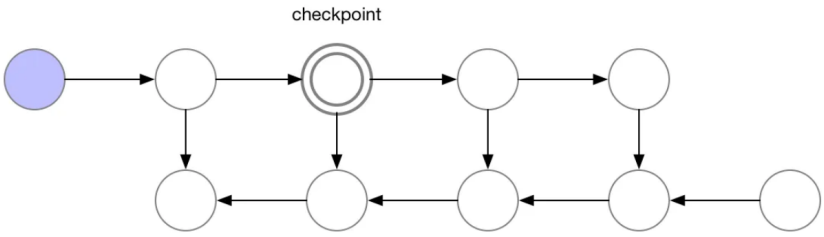


Figure 11: Source: https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9

5.8 Unsloth

L'utilisation de Unsloth [2] nous a permis passer à une longueur de contexte de 4096 en réduisant de 70% la consommation de VRAM et en divisant le temps de calcul par 2. Cette librairie repose sur les optimisations suivantes :

5.8.1 Autograd Manuel et parenthésage optimal

Lors de calcul de l'attention, on a besoin de calculer :

$$
\begin{aligned}
Q &= X\tilde{W}_q = X(W_q + A_q B_q) \\
K &= X\tilde{W}_k = X(W_k + A_k B_k) \\
V &= X\tilde{W}_v = X(W_v + A_v B_v) \\
A(X) &= \sigma\left(\frac{1}{d} Q K^T + M\right) V
\end{aligned}
$$

Unsloth exploite le fait que $r \ll \min(d, k)$ pour déterminer le meilleur ordre de parenthésage des multiplications matricielles enchaînées. Cela permet selon eux

d'économiser 6% de temps de calcul.

5.8.2 Flash Attention 2

Unsloth Utilise Flash Attention 2 [6] intégré dans Xformers [25]. L'idée de cette méthode est d'effectuer le calcul des scores d'attention par blocs. Cela permet de ne jamais avoir à écrire entièrement les matrices intermédiaires en mémoire et ainsi de diminuer fortement la consommation de VRAM. Formellement avec 2 blocs, posons $S = [S^{(1)}, S^{(2)}] = QK^T$, $P = \text{softmax}(S)$ et $O = PV$ avec $S^{(1)}, S^{(2)} \in \mathbb{R}^{B_r \times B_c}$. On a alors :

$$m^{(1)} = \text{rowmax}(S^{(1)}) \in \mathbb{R}^{B_r}$$

$$\ell^{(1)} = \text{rowsum}(e^{S^{(1)} - m^{(1)}}) \in \mathbb{R}^{B_r}$$

$$O^{(1)} = e^{S^{(1)} - m^{(1)}} V^{(1)} \in \mathbb{R}^{\mathbb{B}_\diagdown \times}$$

$$m^{(2)} = \max(m^{(1)}, \text{rowmax}(S^{(2)}) = m$$

$$\ell^{(2)} = e^{m^{(1)} - m^{(2)}} \ell^{(1)}) + \text{rowsum}(e^{S^{(2)} - m^{(2)}}) = \text{rowsum}(e^{S^{(1)} - m}) + \text{rowsum}(e^{S^{(2)} - m})$$

$$O^{(2)} = \text{diag}(e^{m^{(1)} - m^{(2)}})^{-1} O^{(1)} + e^{S^{(2)} - m^{(2)}} V^{(2)} = e^{S^{(1)} - m} V^{(1)} + e^{S^{(2)} - m} V^{(2)}$$

$$O = diag(\ell^{(2)})^{-1} O^{(2)}$$

On a ainsi un calcul efficace de O et de la logsumexp (nécessaire pour la rétropropagation du gradient) $m + \log(\ell)$. Par ailleurs, Flash Attention 2 présente d'autres optimisations pour l'application du masque dans le calcul de l'attention pour un LLM decoder-only : il n'est pas nécessaire de modifier les blocs situés entièrement sous la diagonale de la matrice S et il n'est pas non plus nécessaire de calculer l'attention pour les blocs situés au-dessus.

Enfin, Flash Attention 2 propose encore des optimisations pour faciliter la parallélisation des calculs décrits précédemment.

Toutes ces optimisations conduisent à une économie de 8% de temps de calcul et de 39% de VRAM.

5.8.3 Conversion de Data

Pour le fine-tuning avec QLoRA, les poids de certaines couches du modèle sont convertis en float32 lors des calculs. Cette conversion est réalisée car le fine-tuning avec uniquement des poids en float16 peut être instable.

Cependant, la librairie transformers de Hugging Face effectue une conversion naïve pour rester compatible avec la plupart des architectures de modèles. Unsloth optimise cette conversion des poids spécifiquement pour les modèles pris en charge. Cela permet selon eux de diminuer le temps de calcul de 20% et la consommation de VRAM de 7%.

5.8.4 Triton Implementation

Unsloth réécrit des parties clés du code d'entraînement en utilisant un langage spécial appelé Triton [32], développé par OpenAI, qui est conçu pour l'informatique haute performance. Cela permet d'économiser selon eux 11% de temps de calcul.

5.9 Augmentation du contexte : Rope Scaling

Pour encoder un texte, on utilise généralement une matrice d'embeddings $\mathbf{T}$ qui a la taille du vocabulaire et une matrice embedding de position $\mathbf{P}$ qui encode la position du token dans la séquence d'entrée. La taille d'embedding de position définit la taille du contexte et est défini de sorte que le même mot soit codé différemment aux positions i et j.
Llama 2 utilise des Rotary Positional Embeddings (RoPE) [30]. La technique de RoPE encode la position d'un token à partir d'une rotation dépendant de la position de son vecteur d'embedding. Cela ne change pas la position relative entre deux tokens. Le calcul de l'attention devient alors : $\hat{q}_i^T \hat{x}_j = q_i^T \mathbf{R}_{\theta,i-j} x_j$.
Où $\mathbf{R}_{\theta,i-j}$ est une matrice de rotation et θ dépend de la longueur maximale du contexte.
On peut modifier la longueur de contexte maximale en modifiant la période des rotations.

5.10 L'entraînement parallèle distribué

Les paramètres du modèle sont copiés sur toutes les machines, et les données sont partagées entre elles. Chaque machine effectue la propagation avant et arrière. On agrège les gradients. Avec ces informations, nous pouvons utiliser un optimiseur pour mettre à jour les paramètres du modèle sur toutes les machines. Les paramètres mis à jour seront ensuite utilisés dans le prochain cycle d'itérations de formation du modèle.

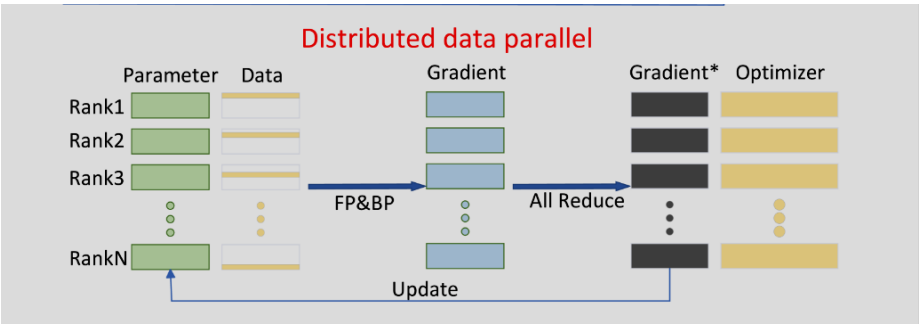


Figure 12: source : https://arxiv.org/html/2401.02038v2

5.11 Inference

Un des défis de la génération par LLM est leur tendance à la répétition excessive [13]. Habituellement, ces modèles génèrent du texte en choisissant le mot suivant parmi les plus probables, ce qui peut mener à des réponses très répétitives. Nucleus Sampling [19] offre une solution en sélectionnant les mots d'un sous-ensemble restreint, ou "noyau", dont la somme des probabilités dépasse un seuil défini. Cette méthode permet de diversifier les résultats tout en maintenant leur cohérence.

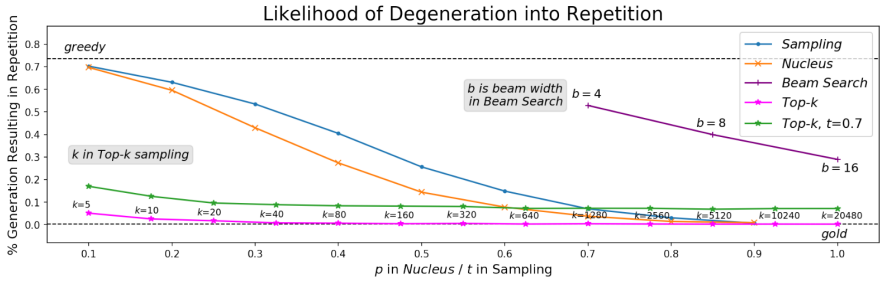


Figure 13: Source : https://arxiv.org/pdf/1904.09751.pdf

Il existe d'autres méthodes comme la pénalisation de la répétition lors de la génération [31] mais cela n'a pas donné de résultats convaincants pour notre modèle.

6 Configurations expérimentales

6.1 Data

Nous avons entraîné nos modèles sur deux datasets principaux :
1. Pour entraîner le modèle pour la génération de piano, nous avons utilisé le dataset GiantMIDIPiano [23], comprenant plus de 10 000 morceaux de musique classique en format *MIDI*. incluant des œuvres de 2 786 compositeurs.

2. pour le modèle spécialisé sur la génération de morceaux de guitare nous avons utilisé le dataset *DadaGP* trié préalablement pour ne garder que les morceaux de *metal*.

Dans tous les cas nous partons des fichiers *.txt* en format *Simplified MIDI*, avec ou sans PitchBend en fonction des entraînements.

Au début de l'entraînement, avant chaque exemple, nous ajoutons un *header* qui explique la signification de notre syntaxe. Le prompt suit des principes de base du *prompt engineering* et est structuré de la façon suivante :

- Nous commençons par donner un *persona* au modèle [22] : **'You are a metal guitarist composer.'**

- Puis, nous donnons le long contexte qui explique la notation : **'In this context, each music note in a musical sequence is described using five parameters: pitch(p) from 0 to 127(highest pitch), volume (v) ...'**
- Ensuite, nous donnons des instructions plus spécifiques : **'Your composition should demonstrate a clear progression and development, appropriate pauses, including thoughtful variations in melody, harmony, rhythm. Your Task is to complete the generation of ... '**
- Finalement, nous lui donnons un morceau du dataset en *Simplified MIDI*, tronqué plus ou moins loin en fonction du *context*. La taille du contexte dépend du modèle et de nos optimisations.

Le dataset est ainsi constitué d'une longue liste de prompt et réponse du type de la Figure 14.

Pour la génération, une fois le modèle entraîné, il est extrêmement important que nous lui redonnions exactement le même *header*. Nous lui donnons également quelques premières notes pour amorcer la séquence.

```
<s>[INST] <<SYS>> You are a metal guitarist composer. In this context, each music note in a musical sequence is described using
five parameters: pitch (p) from 0 to 127 (highest pitch), volume (v) from 0 to 127 (loudest), duration of the note (d) in ticks,
the length of the pause (t) in ticks before the next note begins regardless of the previous note's duration (a tick is
approximately 5.21 milliseconds) and the pitch bend of the note (b) from 0 (2 dial tones under the current pitch) to 46 (2 dial
tones over), with 23 meaning no distortion. Each parameter is followed by its value and separated by colons (e.g.
p52:v57:d195:t212:b18). Your composition should demonstrate a clear progression and development, appropriate pauses, including
thoughtful variations in melody, harmony, rhythm. Your Task is to complete the generation of : <</SYS>> p49:v71:d196:t1174:b23
p44:v77:d196:t0:b23 p49:v61:d196:t1175:b23 [/INST] p44:v81:d195:t0:b23 p49:v71:d195:t1174:b23 p44:v63:d196:t0:b23
p49:v80:d196:t1174:b23 p51:v82:d1175:t0:b23 p58:v81:d1175:t0:b23 p63:v79:d1175:t1175:b23 p51:v83:d587:t0:b23 p58:v89:d587:t0:b23
p63:v93:d587:t587:b23 p53:v89:d587:t0:b23 p58:v85:d587:t587:b23 p51:v78:d587:t0:b23 p56:v72:d587:t0:b23 p61:v69:d587:t587:b23
```

Figure 14: Un exemple tiré du dataset `metal_with_bend`

6.2 Paramètres d'entraînement et d'inférence

Nous avons effectué de nombreux entraînements, avec 4 à 8 GPUs Nvidia A5000 24Go VRAM, afin de tester les différentes combinaisons de paramètres envisageables puis de comparer les modèles que nous obtenions. Les paramètres que nous avons modifié sont les suivants :

- `learning_rate`: $\mathbf{2e-4}$ mesure la vitesse de mise à jour des poids au cours de l'entraînement
- `batch-size` et `num_epochs`: **4 à 16** par machine et 3 epoches, correspond au nombre de samples traitées par itération et au nombre de passage complet du dataset.
- `Optimiseur`: **AdamW** [27], est l'algorithme d'optimisation pour ajuster les paramètres du modèle.

- `Learning rate scheduler`: **Cosine** [26] , un outil qui ajuste le taux d'apprentissage au cours de l'entraînement d'un modèle.
- `top_p`: **0.95** mesure la largeur du noyau utilisé par le *Nucleus Sampling* lors de l'inférence.
- `temperature`: **1** une basse température permet "d'aplatir" la distribution des tokens sélectionnés pour la génération et donc d'augmenter le caractère aléatoire de cette dernière.

Nous avons gardé les mêmes paramêtres de QLoRA au cours de l'entraînement. Comme indiqué dans [9], nous entraînons des adapteurs LoRA sur toutes les couches linéaires du modèles i.e `target_modules = ["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]` et les autres paramètres (`lora_r`, `lora_alpha` ...) influent peu sur la qualité du modèle obtenu.

6.3 Améliorations successives

Nous avons procédé à différentes améliorations au cours de l'entraînement que nous retraçons chronologiquement ici.

1 . Premier modèle obtenu, entraîné sur `Giant-Midi-Piano` avec une longueur de contexte de 1024. Nous avons choisi cette valeur faible en raison de problèmes de mémoire. Ce modèle générait des séquences musicales trop courtes ce que nous avons attribué à la longueur de contexte trop faible utilisée lors de l'entraînement.

2 . Suite à cette observation, nous avons utilisé la libraire `unsloth` ce qui nous a permis de diminuer la VRAM utilisée et ainsi d'augmenenter la longueur de contexte et le `batch-size`. Nous obtenons ainsi un modèle entraîné plus rapidement et générant des séquences plus longues. Cependant, le modèle obtenu a tendance à générer des séquences très répétitves.

3 . Pensant que le problème de répétition dont souffre le précédent modèle était dû à de l'overfitting, nous avons réduit le nombre d'epoch. Cela n'a cependant rien fait pour résoudre le problème.

4 . Le problème de répétition n'étant manifestement pas dû à de l'overfitting, nous avons augmenté le `learning_rate` lors de l'entraînement. Bien que LoRA soit une technique sensible au `learning_rate` et qu'une valeur faible de ce paramètre est généralement conseillée, nous avons pensé qu'étant donné la grande différence entre le langage naturel et notre syntaxe l'augmentation de ce paramètre était une possibilité envisageable. Nous obtenons avec ce nouveau modèle des résultats plus convaincants mais néanmoins toujours assez répétitifs.

5 . Nous avons alors modifié les paramètres d'inférence du modèle, en prenant `top_p = 0.95` et `temperature = 1` en suivant les résultats obtenus

dans [19]. Nous obtenons alors de bons résultats.

6 . Finalement nous avons progressivement augmenté la longueur de contexte des modèles que nous entraînions de 4000 à 16000. Par ailleurs avons également entraîné des modèles sur le dataset *DadaGP* avec et sans PitchBend.

7 Performance des différents modèles obtenus

Nous ne sommes pas parvenus à trouver de métrique intéressante pour évaluer la performance de nos modèles (d'après notre tuteur, il n'existe par ailleurs pas vraiment de méthodes de *benchmarking* reconnues pour la génération musicale). Nous nous contentons donc dans cette section de commenter les séquences musicales générées par nos modèles.

Modèle entraîné sur `Giant-Midi-Piano` avec un contexte de 16K : ce modèle est capable de générer des séquences avec une structure et des mélodies complexes tout en restant dans la même gamme. Il semble également avoir compris le concept de tension-résolution dans la génération de morceaux de piano mais a parfois tendance à générer des morceaux "trop lourds" en superposant trop de notes.

Modèle entraîné sur `Giant-Midi-Piano` puis sur `DadaGP` avec un contexte de 4K : ce modèle génère des riffs de guitare intéressants et est capable de suivre une grille d'accord. Cependant, il a tendance à générer des séquences encore trop répétitives ce qui est sûrement dû à la présence de morceaux de basse dans le dataset d'entraînement.

Modèle entraîné sur `DadaGP` avec un contexte de 8K : ce modèle génère également des séquences de guitare intéressante mais parfois avec des notes tenues trop longtemps. Cela est probablement dû au fait qu'il n'a pas été entraîné avec du PitchBend qui rendrait ces longues notes tenues bien plus intéressantes. Le modèle permettant de la génération avec PitchBend est à ce jour encore en cours d'entraînement.

8 Limitations de notre travail

8.1 Limitations liées au dataset

Notre dataset final a évidemment de fortes limitations, que nous n'avons pas eu le temps de corriger.

- Nous assimilons trop fortement les basses et les guitares. Notre dataset est donc trop peu spécifique et la structure répétitives des basses peut parfois prendre le dessus sur des parties solos. Réduire un peu le dataset

pour le spécialiser sur les solos uniquement pourrait être bénéfique. Nous n'avons pas trouvé d'alternative à le faire à la main, mais des applications comme *GarageBand* parviennent à mettre par défaut le bon instrument sur les pistes. Nous ne savons pas comment elles arrivent à tirer cette information des *MIDI*, mais si nous le trouvions, cela permettrait un tri facilement automatisable du Dataset.

- Nous avons également des améliorations potentielles au niveau de la qualité de la *data augmentation*. Notre choix de moduler les morceaux sur un intervalle aléatoire entraîne, dans certains cas, une légère perte de cohérence, du fait de modulations intervenant de manière assez abrupte dans le morceau, ou même dans de plus rares cas, rentrant en conflit avec les modulations originales du morceau. La solution, afin de limiter ces risques, serait de commencer la modulation à un instant précis, au début d'une mesure ou d'une phrase musicale par exemple, or il est complexe d'avoir accès à ces informations sur des fichiers MIDI. De plus, pour éviter les chevauchements de modulations incohérents, une solution serait de traiter différemment les morceaux contenant déjà des modulations. Cette opération est difficile, car les fichiers MIDI ne contiennent pas cette information qui est déduite, parfois par les logiciels tels que MuseScore, mais souvent uniquement par le musicien lui-même. De plus, comme mentionné précédemment, les nuances étant très monotones dans DadaGP, l'augmentation de celle-ci n'a qu'un effet très subtil sur les morceaux ainsi générés. Le caractère aléatoire de cette modification pourrait également rentrer en conflit avec la dynamique initiale du morceau, on peut, par exemple, imaginer un crescendo (augmentation progressive du volume) perturbé par cette méthode d'augmentation.

- Empiriquement, 3 *epochs* était un bon équilibre sur notre Dataset, donc nous avons décider de réaliser une augmentation par 3 pour le rendu final. Mais notre exploration de l'intensité de l'augmentation, comme c'est également le cas pour le choix des autres hyper-paramètres, est loin d'être exhaustive.

- Nos convertisseurs entre *MIDI* et *Simplified MIDI* étaient également imparfaits (certaines notes sont fusionnées au retour notamment). C'est une perte d'information qui peut probablement être réduite à 0, mais nous n'avons pas réussi à le faire jusqu'ici.

- En l'occurence, dans *DadaGP* l'usage des nuances est extrêmement limité (tout au plus a-t-on 3-4 valeurs différentes par morceaux).À tel point que l'on aurait pu essayer d'assumer le volume des notes constant, et de ne pas prédire ce token, pour reconstituer le MIDI avec une valeur donnée. Cela permettrait de prédire des séquences plus longues, et de ne pas donner l'occasion d'un *overfitting* sur les nuances qui n'aurait aucun sens.

- Au lieu de demander de prédire chaque quintuplet (p,v,d,t,b), il aurait été possible de décomposer la construction du morceau : par exemple prédire

l'agencement des notes (d,t), puis prédire leur hauteur (p), puis prédire les nuances(v) et enfin prédire les bends (b). Conseillée par notre tuteur, nous n'avons pas mis en oeuvre cette méthode par manque de temps et parce que tout prédire simultanément marchait tout même bien. Mais si cela se trouve, la prédiction qui en résulte est encore bien meilleure.

Pour élargir un peu le champs des possibles, disons aussi que notre façon d'aborder le problème a été exclusivement centrée sur le format *MIDI*. D'autres formats, dans lesquelles d'importantes bases de données sont directement disponibles depuis peu comme *GuitarPro*, prennent de plus en plus une place de choix dans la génération musicale de certains types de musique (rock, blues, jazz,etc.). Ils contiennent beaucoup d'éléments portant sur la synchronisation(**encodage temporel**) entre instruments que nous ignorons complètement dans notre travail, et que *MIDI* n'est pas le mieux fait pour encoder(de par son **encodage piste par piste**). C'est l'approche que nous conseillait l'auteur principal de *DadaGP*, mais que nous avons décidé de ne pas approfondir de par tout le travail préliminaire fourni sur le format *MIDI*.

8.2 Limitations liées au Fine-Tuning

Il reste à expérimenter la possibilité d'entraîner un nouveau modèle de langage avec notre nouveau vocabulaire, en utilisant les mêmes ressources computationnelles, et de comparer les résultats pour justifier l'efficacité d'utiliser des LLMs pré-entraînés.

Par ailleurs, nous ne sommes pas parvenus à obtenir un modèle capable à la fois de générer de la guitare et du piano. Cela est peut-être possible mais nécessiterait d'ajuster plus finement les différents paramètres d'entraînement.

9 Remerciements

Nous tenons à remercier chaleureusement Gaëtan Hadjeres, notre tuteur qui nous a aiguillé tout au long du projet. Nous remercions aussi Kossi Neroma pour ses conseils judicieux qui ont inspiré notre travail.

C'est également Alexandre D'Hoodge et Pedro Sarmento qu'il nous faut remercier pour leur expertise au niveau des spécificités de la génération des morceaux de guitare et pour l'accès au dataset *DadaGP*.

Nous sommes également redevable à l'École polytechnique pour la puissance de calcul qu'elle a sû mettre à la disposition de notre projet.

Finalement, nous remercions notre coordinateur El Mahdi El Mahmdi pour sa disponibilité.

References

[1] https://github.com/rageagainsthepc/guitarpro-to-midi.

[2] https://github.com/unslothai/unsloth.

[3] Andrea Agostinelli, Timo I. Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, Matt Sharifi, Neil Zeghidour, and Christian Frank. Musiclm: Generating music from text, 2023.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[5] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models, 2022.

[6] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[8] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization, 2022.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[10] Alexandre D'Hooge, Louis Bigo, and Ken Déguernel. Modeling bends in popular music guitar tablatures. In *Proceedings of the 24th International Society for Music Information Retrieval Conference*, November 2023.

[11] Chris Donahue, Antoine Caillon, Adam Roberts, Ethan Manilow, Philippe Esling, Andrea Agostinelli, Mauro Verzetti, Ian Simon, Olivier Pietquin, Neil Zeghidour, and Jesse Engel. Singsong: Generating musical accompaniments from singing, 01 2023.

[12] Douglas Eck and Jürgen Schmidhuber. Finding temporal structure in music: Blues improvisation with lstm recurrent networks. volume 12, pages 747 – 756, 02 2002.

[13] Zihao Fu, Wai Lam, Anthony Man-Cho So, and Bei Shi. A theoretical analysis of the repetition problem in text generation. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021.

[14] Sang gil Lee, Uiwon Hwang, Seonwoo Min, and Sungroh Yoon. Polyphonic music generation with sequence generative adversarial networks, 2018.

[15] Gaëtan Hadjeres and Léopold Crestel. The piano inpainting application, 2021.

[16] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: a steerable model for bach chorales generation, 2017.

[17] Dorien Herremans and Elaine Chew. Morpheus: Generating structured music with constrained patterns and tension. *IEEE Transactions on Affective Computing*, 10(4):510–523, October 2019.

[18] Hermann Hild, Johannes Feulner, and Wolfram Menzel. Harmonet: A neural net for harmonizing chorales in the style of j. s. bach. In J. Moody, S. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4. Morgan-Kaufmann, 1991.

[19] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020.

[20] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[21] Eric Humphrey, Justin Salamon, Oriol Nieto, Jon Forsyth, Rachel Bittner, and Juan Bello. Jams: A json annotated music specification for reproducible mir research. 10 2014.

[22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2023.

[23] Qiuqiang Kong, Bochen Li, Jitong Chen, and Yuxuan Wang. Giantmidi-piano: A large-scale midi dataset for classical piano music, 2022.

[24] Qiuqiang Kong, Bochen Li, Xuchen Song, Yuan Wan, and Yuxuan Wang. High-resolution piano transcription with pedals by regressing onset and offset times, 2021.

[25] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. `https://github.com/facebookresearch/xformers`, 2022.

[26] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[27] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[28] Michael Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science - CONNECTION*, 6:247–280, 01 1994.

[29] Yang Qin, Huiming Xie, Shuxue Ding, Benying Tan, Yujie Li, Bin Zhao, and Mao Ye. Bar transformer: a hierarchical model for learning long-term structure and generating impressive pop music. *Applied Intelligence*, 53, 08 2022.

[30] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[31] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation, 2022.

[32] Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023.

[34] Dimitri von Rütte, Luca Biggio, Yannic Kilcher, and Thomas Hofmann. Figaro: Generating symbolic music with fine-grained artistic control, 2024.