DIMINISHING RETURNS OBSERVED FROM AI MUSIC MODELS

BY CLIFFORD NJOROGE

FORWARD

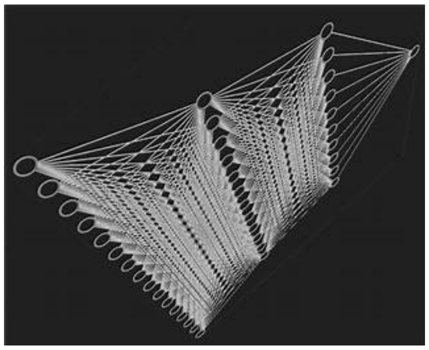
This report is dedicated to examining the factors that contribute to the substandard quality observed in music generated by various AI models. The acknowledgment of this issue suggests that the quality of AI-generated music currently falls short of established standards. The purpose of this report is to conduct a thorough investigation into the underlying reasons behind this phenomenon, offering valuable insights for readers interested in the intersection of artificial intelligence and music composition.

The focus of our analysis is on the music produced by a range of AI models, indicating that the problem is not confined to a specific system but extends across different implementations of artificial intelligence. By undertaking this examination, we aim to provide a comprehensive understanding of the factors influencing the perceived substandard quality.

Our report will delve into the intricacies of AI-generated music, exploring elements such as the algorithms utilized, the datasets employed during training, the complexity of the models, and potential shortcomings in capturing the nuanced aspects of musical expression. The term "phenomenon" is used to highlight that the substandard quality is not a random occurrence but a complex and multifaceted issue that requires a deeper understanding.

By unraveling the contributing factors, this report aims to equip readers with a nuanced perspective on the challenges associated with AI-generated music. The findings presented herein could have implications not only for the developers and researchers in the field of artificial intelligence but also for musicians, music enthusiasts, and anyone intrigued by the evolving landscape of creative technologies. Ultimately, this examination seeks to contribute to the ongoing dialogue surrounding the integration of AI in music composition and inspire potential improvements in the quality of AI-generated musical output.

Generative Adversarial Networks



GANs are a type of machine learning framework that can generate new data from a given dataset. They consist of two neural networks: a generator and a discriminator. The generator tries to create realistic data, while the discriminator tries to distinguish between real and fake data. The two networks compete with each other until the generator can fool the discriminator. GANs can be used for various applications, such as generating images, music, text, or 3D models.

MUSEGAN

For our first generative adversarial model we used the Musegan generative model which uses three models for symbolic multi-track music generation under the framework of generative adversarial networks (GANs). The three models, which differ in the underlying assumptions and accordingly the network architectures, are referred to as the jamming model, the composer model and the hybrid model. We trained the proposed models on a dataset of about 3000 bars of jazz midi.

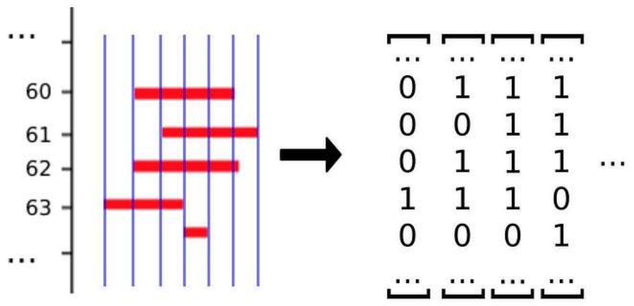
The diminishing returns observed from this type of generative model occur when the predictive results are converted into midi, it is not clear what is the appropriate way to write the predicted results back into midi as the paper mentions using the `pypianoroll.Multitrack` module which requires both an array of stacked Boolean values from the predictions and a dictionary containing the frequency sampling rate (fs) which cannot be obtained from the model predictions. Another issue is when using the generative class of neural networks, we are dealing with converting random noise back into real world images/audio. For our input we generate normalized random noise with the class `torch.randn` which returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution)

i.e., $\text{out}_i \sim \text{N}(0,1)$

This similar procedure is used in the generator training phase to generate/emulate real piano roll vectors from random distributed noise. In the end we settled to use `mido`, which amalgamates all form of midi music processing to write our predictions into music

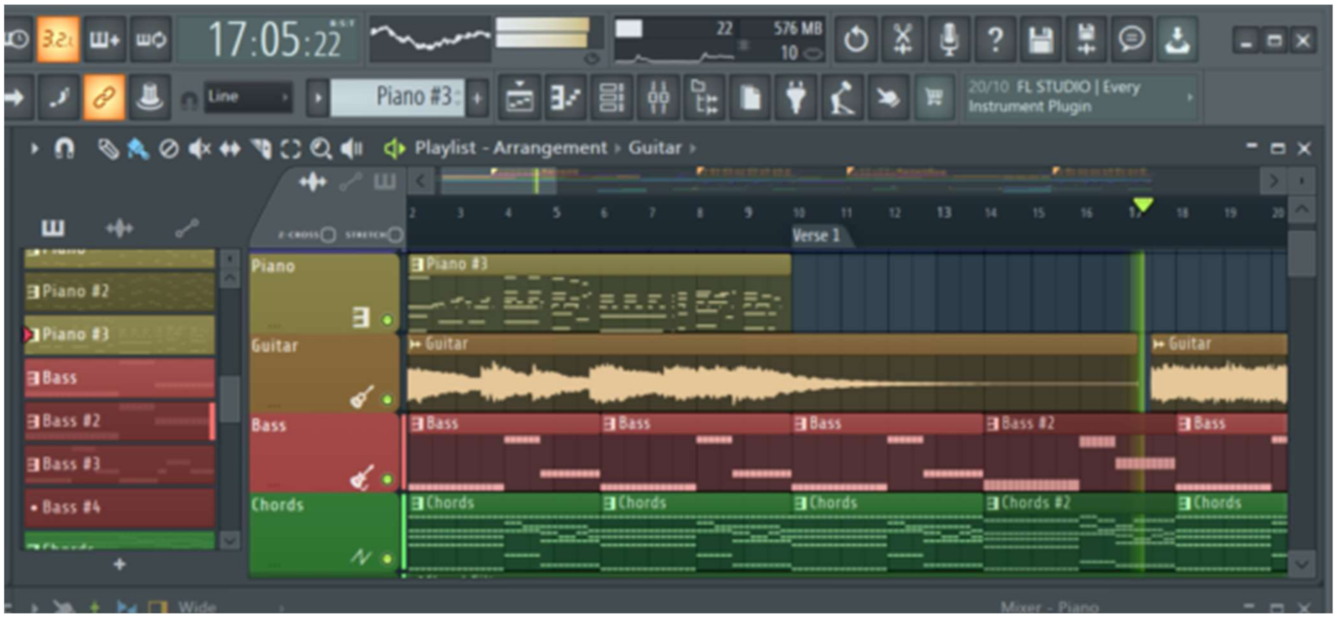
Investigating diminishing returns from music generation with MuseGan

In music theory each note or is generated within a digital audio workstation (DAW) with a unique time signature, and a specific frequency sampling rate. Each note represents a midi key from range 0 to 127 whereas a chord which is a fundamental concept in music theory and composition refers to a group of three or more notes played or heard simultaneously. Chords are the building blocks of harmony and provide the harmonic framework for melodies and musical compositions. Each chord in the DAW contains an octave which can be used to represent the type of instrument being played and each chord can be broken down into multiple constituent notes with their unique time signature and pitch. Musegan has a proper representation of this by using the package pypianoroll which breaks each midi note into machine representation of an array of Boolean: on/off, 1/0, True/False.



$$\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \cdots$$

Where this break occurs is in the complexity of analyzing different octaves; instrument representations when generating chord progressions. Chord progression within a DAW can represent musical annotations in their respective octaves relatively easily as shown below:



OCTAVE REPRESENTATION IN FL STUDIO

In pypianoroll or any other midi processing utility this octave representation does not exist and whatever octave used when parsing the predictions into midi is inherently an infinite value that cannot be set. In jazz music, the most common octave for chord voicings and melodic lines typically ranges between C3 and C5. Here's a breakdown of the common octave ranges used in jazz:

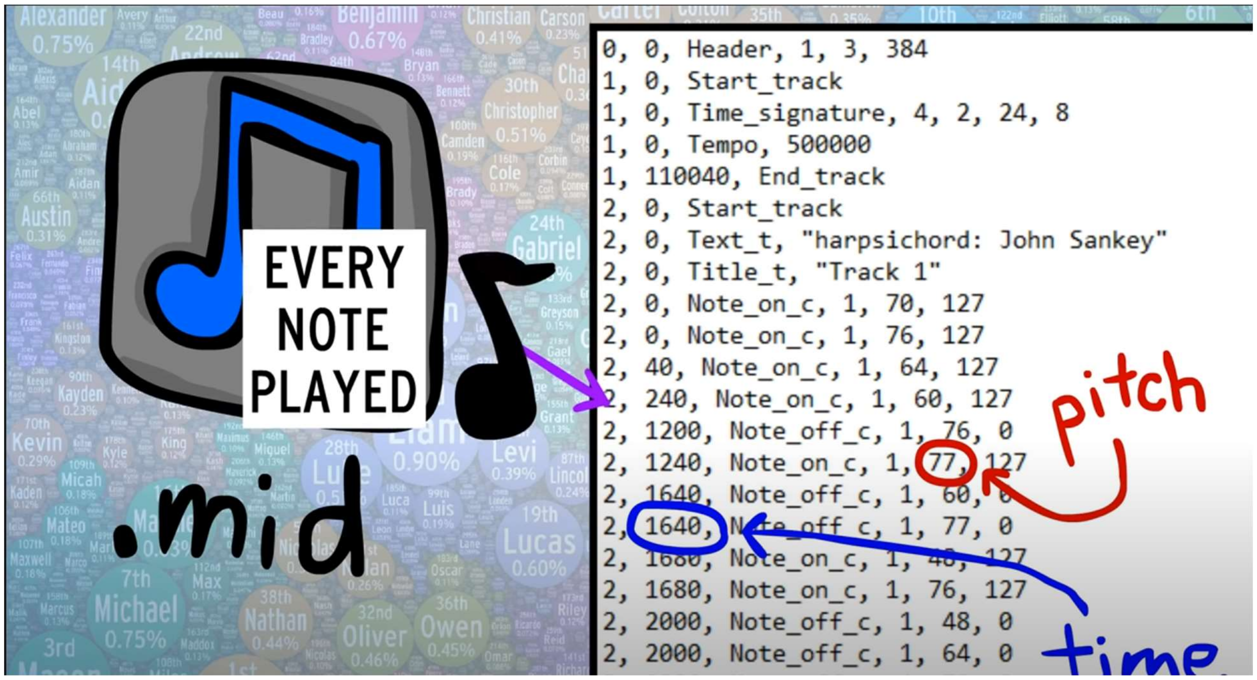
1. Bass (C2 - C3): The bass guitar or double bass often plays in this range, providing the foundational low end of the harmony.
2. Chords and Comping (C3 - C5): Piano, guitar, and other chordal instruments often play in this range. The mid-range is suitable for comping (accompanying) and playing chord voicings.
3. Melody (C4 - C6): Melodic instruments like saxophones, trumpets, and vocals usually perform in this range. It allows for expressiveness and facilitates smooth melodic phrases.

4. Upper Extensions (C6 - C7): Jazz musicians might use this higher range for adding upper chord extensions, such as the 9th, 11th, or 13th, when soloing or comping.

Although, these octave ranges are not strict rules, and jazz musicians often explore different registers to create unique sounds and effects. The choice of octave range may also depend on the specific instrument, musical context, and personal style of the performer. If we observe the pypianoroll representation of notes into midi, we are only looking at 3 aspects of the tune generated from the predictions being the; note, timestep, and harmonics. We do not look at the chord progression, octave roots and DAW representation between chords and notes. Chords are typically created by stacking specific intervals (distances between notes) on top of each other. The most basic type of chord is a triad, which consists of three notes: the root, the third, and the fifth.

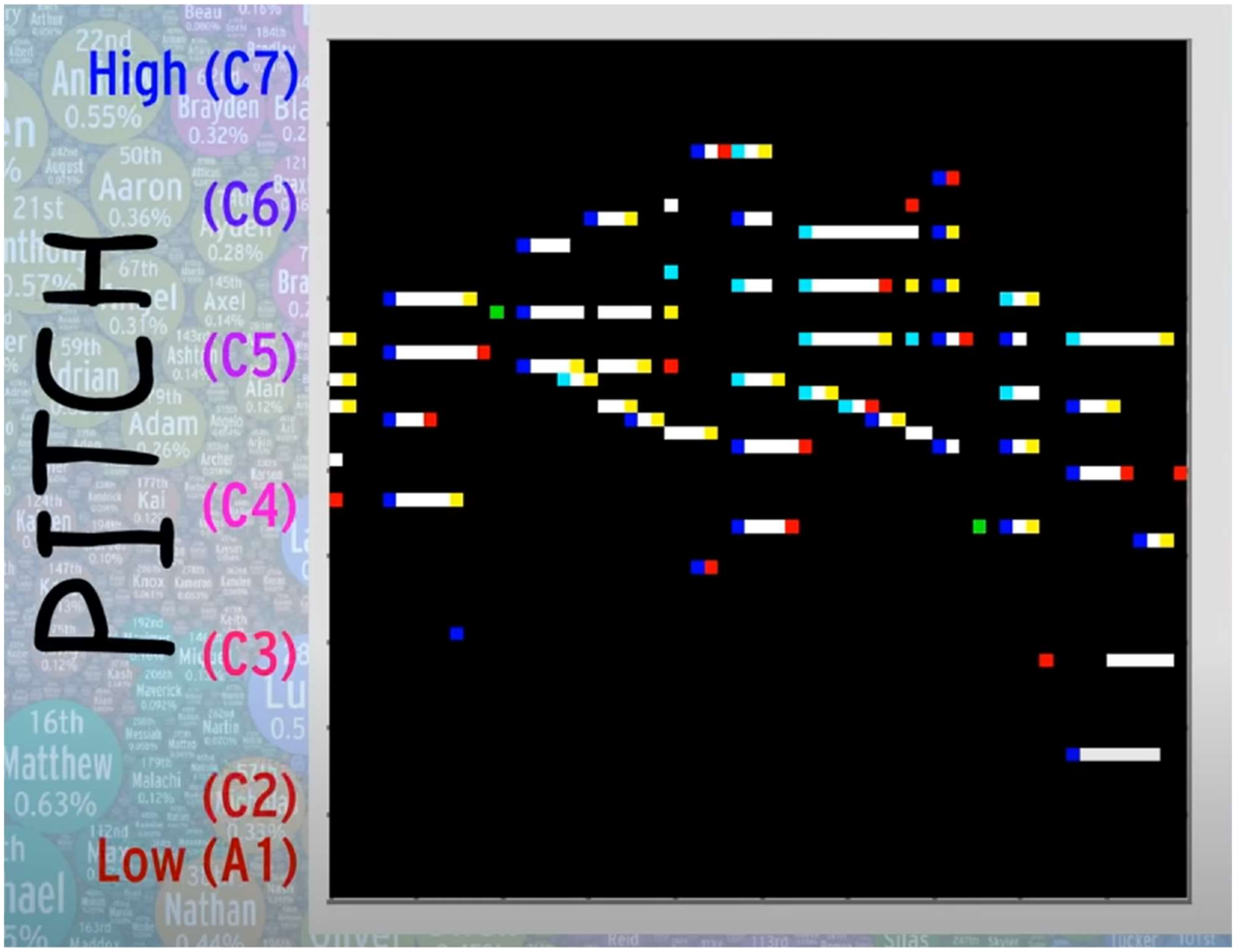
Wave GAN-IMAGE

This representation seeks to streamline the method of generating the data from midi and then taking the observed prediction directly back into midi. Midi files explicitly describe the pitch and time of every note played:

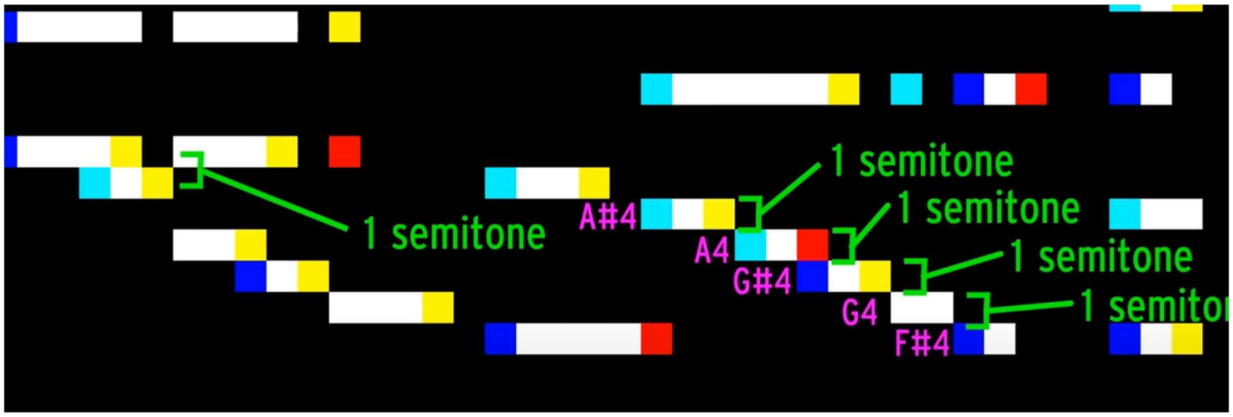


```
0, 0, Header, 1, 3, 384
1, 0, Start_track
1, 0, Time_signature, 4, 2, 24, 8
1, 0, Tempo, 500000
1, 110040, End_track
2, 0, Start_track
2, 0, Text_t, "harpsichord: John Sankey"
2, 0, Title_t, "Track 1"
2, 0, Note_on_c, 1, 70, 127
2, 0, Note_on_c, 1, 76, 127
2, 40, Note_on_c, 1, 64, 127
2, 240, Note_on_c, 1, 60, 127
2, 1200, Note_off_c, 1, 76, 0
2, 1240, Note_on_c, 1, 77, 127
2, 1640, Note_off_c, 1, 60, 0
2, 1640, Note_off_c, 1, 77, 0
2, 1680, Note_on_c, 1, 48, 127
2, 1680, Note_on_c, 1, 76, 127
2, 2000, Note_off_c, 1, 48, 0
2, 2000, Note_off_c, 1, 64, 0
```

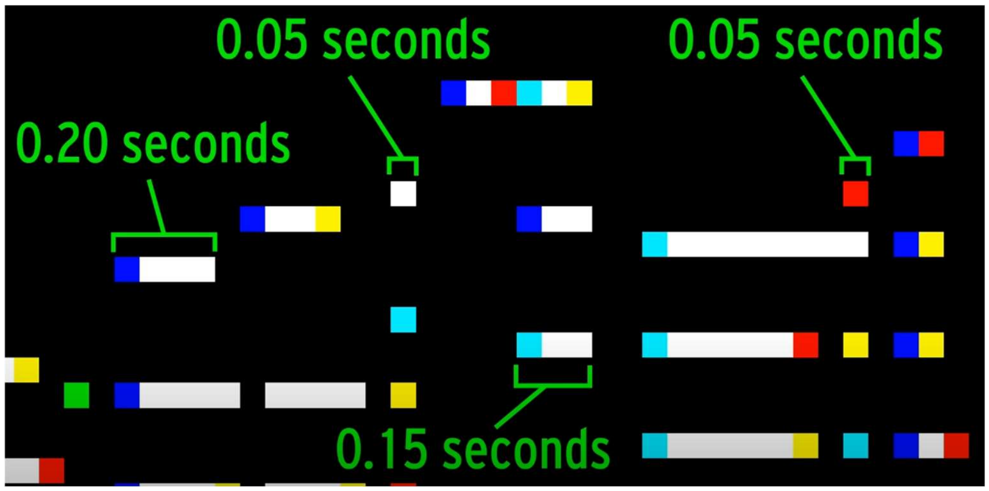
We can use this Idelogy to graph a 2d image representation of our midi file i.e.:



Where up and down represents higher and lower in pitch(note), and in semitone intervals as shown below:



The right and left of the midi 2d image represent forward and backward in time in 20th of a second intervals:



A white pixel represents a note being played and a black pixel means no note being played. But therein lies an issue; when looking at the image we can see the color scheme red blue and white. Musegan-image works faster on smaller images of within 100X50 resolution therefore we needed to resize the images to fit a constant resolution of 96x64 which corresponds to 3.2 seconds. Since this is too short, we introduce a new layer where we encode more information per pixel by taking apart the red green and blue channels of each pixel, treat each channel as a separate pixel placed horizontally. We can then turn each of these Mini pixels on or off independently of the others. In short can encode 27 bits of pixel information into just 9 bits of information using this method, conversely, we can extend;

$$64RGB\ pixels * \frac{0.05secs}{1pixel} = 3.2secs$$

To;

$$192onoff\ pixels * \frac{0.05secs}{1pixel} = 9.2secs$$

The image version of this GAN can seamlessly encode, train and decode music back into midi files which addresses the prior issue of how the predictions can be restored back into midi.

It also addresses the issue of octave/ instrument representation whereby constituent notes are deciphered from their chord progression into pitches that are represented in a 2D RGB color format

Diminishing returns of musegan image

- The caveat of this method is that it fails to use the intuitive the spatial nature of convolutional filters.

- All directions are similar so there's no sense of before and after, therefore it is nearly impossible to learn "if then" concepts like After Note "C5" will move to "C1", but not "C4".

- Pixel color values exist on a spectrum of 0 to 1. This is not discrete and thus leads to blurry ideas of where notes should be played.

- As compared to the piano roll version of musegan the multitrack function has a key word argument that allows us to turn of the drums which fail to follow the harmonics observed from chord progression. The image method however fails to do this and thus leads to overall harmonic representation when training the model. Trying to turn off the drums in rgb format ends up chopping the training data and thus one is restricted to only using midi files with no drum instruments

- GAN overshooting where the unpredictable nature of the discriminator does not always measure quality well, allowing the generator to outsmart the discriminator quickly before any reasonable images have been generated

TRANSFORMERS

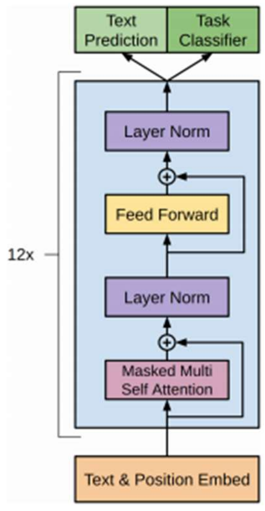
GPT3-MUSIC

The Generative Pre-Trained Transformer or GPT model has achieved astonishing results when dealing with Natural Language Processing (NLP) tasks. However, the model architecture is not exclusive to NLP and has been utilized to solves other problem such as time-series prediction or music generation. In this approach we convert the MIDI into piano roll format with a sampling interval of every 16th note. As such the piano roll is a 2D array of size (song_len, 128) where song_len is the total number of 16th notes in the song and 128 is the number of possible pitches in a MIDI song.

This data encoding approach represents the music note for every constant time interval, thus, allowing us to represent the whole song into a compact 2D array. From here, we can carry out a similar approach to word encoding which is index-based encoding every combination of pitches then feed them into an embedding layer. We decided to not include the velocity features as this will cause our pitch combination vocabulary to explode. And the 16th note was the optimal interval as it can represent the music details accurately enough while also keeping our piano roll array from getting too stretched out.

We used Python `pretty-midi` and `music21` to aid the data parsing and processing steps. To extract the piano part, we filter out the streams that contain the greatest number of notes (as this is often the case for piano streams).

For the architecture GPT utilizes solely the decoder block of the transformer architecture and it stacks these decoder block on top of one another to increase the complexity of the network.

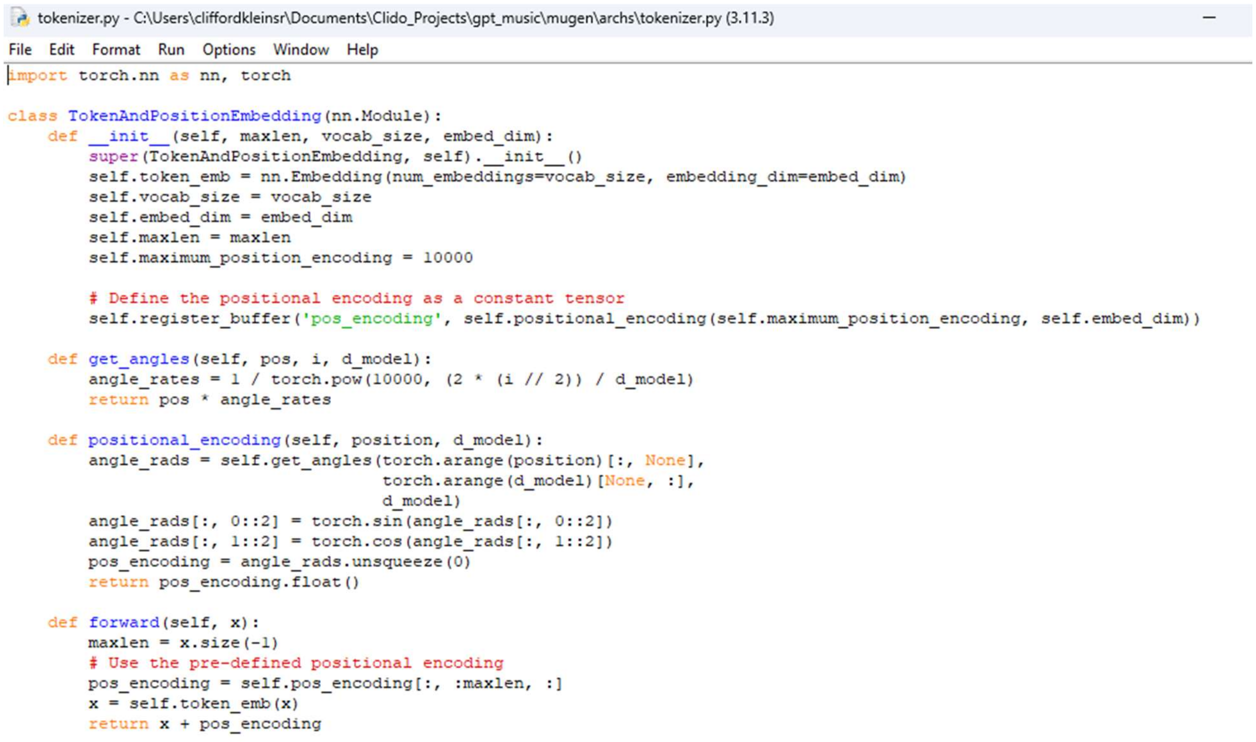


For the embedding of token and position, we use the sine and cosine function.

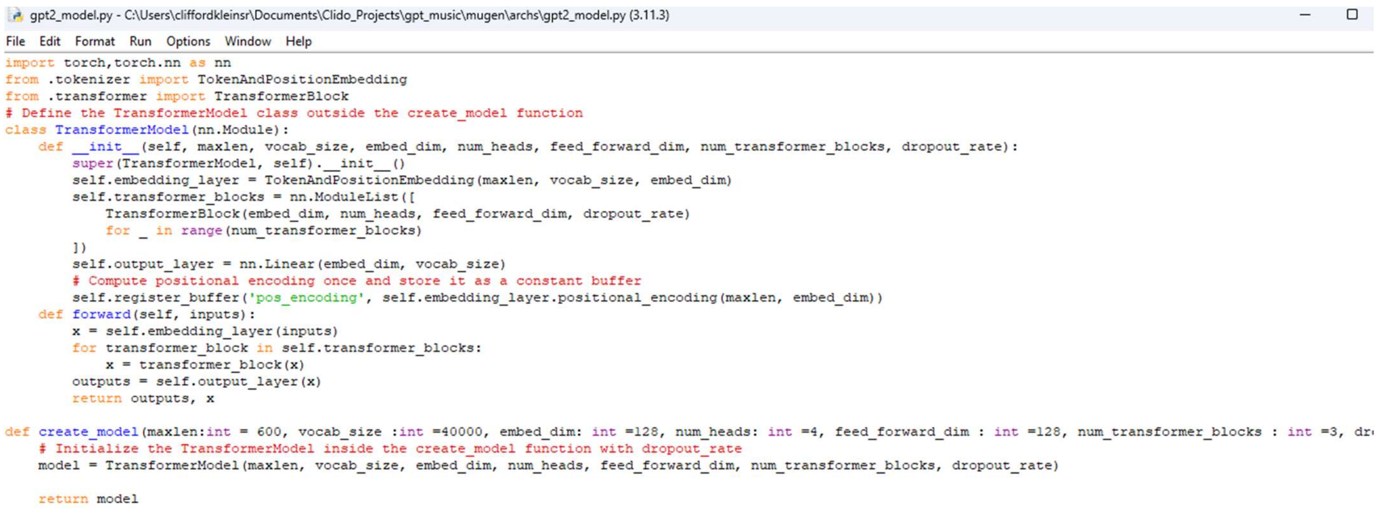
$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

As can be seen in the class below:


tokenizer.py - C:\Users\cliffordkleinsr\Documents\Clido_Projects\gpt_music\mugen\archs\tokenizer.py (3.11.3)
File Edit Format Run Options Window Help

```
import torch.nn as nn, torch

class TokenAndPositionEmbedding(nn.Module):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.token_emb = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embed_dim)
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.maxlen = maxlen
        self.maximum_position_encoding = 10000

        # Define the positional encoding as a constant tensor
        self.register_buffer('pos_encoding', self.positional_encoding(self.maximum_position_encoding, self.embed_dim))

    def get_angles(self, pos, i, d_model):
        angle_rates = 1 / torch.pow(10000, (2 * (i // 2)) / d_model)
        return pos * angle_rates

    def positional_encoding(self, position, d_model):
        angle_rads = self.get_angles(torch.arange(position)[:, None],
                                     torch.arange(d_model)[None, :],
                                     d_model)
        angle_rads[:, 0::2] = torch.sin(angle_rads[:, 0::2])
        angle_rads[:, 1::2] = torch.cos(angle_rads[:, 1::2])
        pos_encoding = angle_rads.unsqueeze(0)
        return pos_encoding.float()

    def forward(self, x):
        maxlen = x.size(-1)
        # Use the pre-defined positional encoding
        pos_encoding = self.pos_encoding[:, :maxlen, :]
        x = self.token_emb(x)
        return x + pos_encoding
```



gpt2_model.py - C:\Users\cliffordkleinsr\Documents\Clido_Projects\gpt_music\mugen\archs\gpt2_model.py (3.11.3)

```python
import torch,torch.nn as nn
from .tokenizer import TokenAndPositionEmbedding
from .transformer import TransformerBlock
# Define the TransformerModel class outside the create_model function
class TransformerModel(nn.Module):
    def __init__(self, maxlen, vocab_size, embed_dim, num_heads, feed_forward_dim, num_transformer_blocks, dropout_rate):
        super(TransformerModel, self).__init__()
        self.embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
        self.transformer_blocks = nn.ModuleList([
            TransformerBlock(embed_dim, num_heads, feed_forward_dim, dropout_rate)
            for _ in range(num_transformer_blocks)
        ])
        self.output_layer = nn.Linear(embed_dim, vocab_size)
        # Compute positional encoding once and store it as a constant buffer
        self.register_buffer('pos_encoding', self.embedding_layer.positional_encoding(maxlen, embed_dim))
    def forward(self, inputs):
        x = self.embedding_layer(inputs)
        for transformer_block in self.transformer_blocks:
            x = transformer_block(x)
        outputs = self.output_layer(x)
        return outputs, x

def create_model(maxlen:int = 600, vocab_size :int =40000, embed_dim: int =128, num_heads: int =4, feed_forward_dim : int =128, num_transformer_blocks : int =3, dr
    # Initialize the TransformerModel inside the create_model function with dropout_rate
    model = TransformerModel(maxlen, vocab_size, embed_dim, num_heads, feed_forward_dim, num_transformer_blocks, dropout_rate)

    return model
```

The final model consists of this tokenizer with 3 blocks of our transformer model with Self-attention and casual masking

The final model can be trained on random information and used to generate midis using the encoding function used in the tokenizer block, which in our case utilizes the sci-kit learn multi label binarizer.

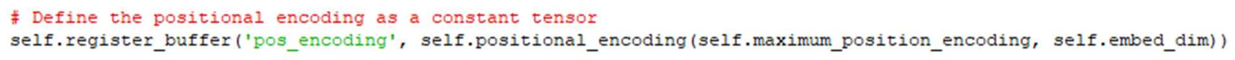
Diminishing returns of gpt3-music

There are signs of overfitting during the evaluation process although we tried to increase the dropout rate, we believe this problem will subdue even when given a larger dataset.

Given the fact that we translated this model from TensorFlow to PyTorch it was observed each framework methodology, including neuron activation and layer definition i.e., `keras.Layers.Dense` and `Pytorch.nn.Sequential` works inherently differently during the model generation. There are also differences when using the `keras.preprocessing` backend within the TensorFlow framework and classical PyTorch training loop in terms of how each handles its gradient accumulation, loss functions and output view comparisons with ground truth.

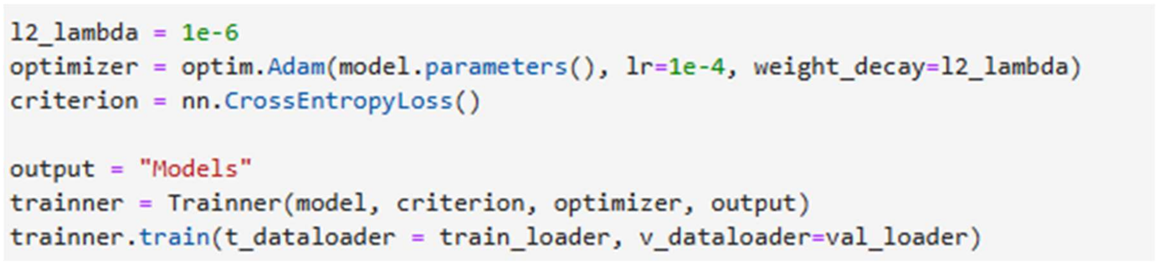
Finally, TensorFlow usage of the gradient tape does not implement L2 penalties which is a technique used in machine learning and optimization to prevent overfitting and improve the generalization performance of a model as observed with the PyTorch optimizers` framework.

It can be implemented on the Keras model compilation stage but since we have token embedding and fixed buffers for positional encoding this is not possible for our case as can be seen in the pytorch implementation: see below



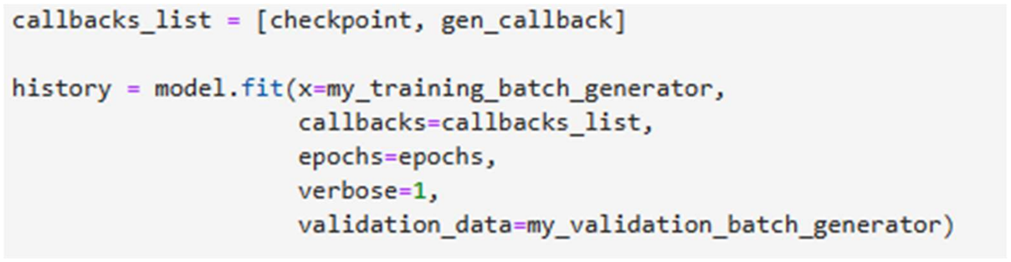
```python
# Define the positional encoding as a constant tensor
self.register_buffer('pos_encoding', self.positional_encoding(self.maximum_position_encoding, self.embed_dim))
```

pt positional encoding registry buffer not implemented in keras version



```
l2_lambda = 1e-6
optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=l2_lambda)
criterion = nn.CrossEntropyLoss()

output = "Models"
trainner = Trainner(model, criterion, optimizer, output)
trainner.train(t_dataloader = train_loader, v_dataloader=val_loader)
```

l2 penalty in present PyTorch trainer



```
callbacks_list = [checkpoint, gen_callback]

history = model.fit(x=my_training_batch_generator,
                    callbacks=callbacks_list,
                    epochs=epochs,
                    verbose=1,
                    validation_data=my_validation_batch_generator)
```

L2 penalty absent in tf.keras trainer

Further steps

In the next Attempt we will seek to resolve all these diminishing returns observed with prior architectures through intuitive music representation, the process of generation will be starlight forward and thus the backlogs observed from harmonic loss, Gan overshooting and chord progression with constituent octave implementation

REFERENCES

MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment. Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, Yi-Hsuan Yang. (2018). In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)arXiv:1709.06298[1].

Wave GAN: Frequency-aware GAN for High-Fidelity Few-shot Image Generation. Zhenyu Zhang, Xiangyu He, Yifan Zhang, Bo Zhang, Jian Sun. (2021). In Proceedings of the 2021 IEEE/CVF International Conference on Computer Vision (ICCV 2021)arXiv:2207.07288[2].

Towards the Generation of Musical Explanations with GPT-3. David R. Winer, Gil Weinberg. (2022). In Proceedings of the 11th International Conference on Computational Creativity (ICCC 2020)Springer Link[3].

GPT-2: Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI blog, 1(8), 9[1]

Pypianoroll: Dong, H. W., Hsiao, W. Y., & Yang, Y. H. (2018). Pypianoroll: Open source python package for handling multitrack pianorolls. In Late-breaking demos of the 19th International Society for Music Information Retrieval Conference (ISMIR)[2]

Pretty MIDI: Cuthbert, M. S., & Ariza, C. (2010). music21: A toolkit for computer-aided musicology and symbolic music data. In Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR)[3]

Music21: Cuthbert, M. S., & Ariza, C. (2010). music21: A toolkit for computer-aided musicology and symbolic music data. In Proceedings of the 11th International Society for Music Information Retrieval Conference (ISMIR)[4]