JOHANNES KEPLER
UNIVERSITÄT LINZ

Submitted by
Sebastian Sonderegger

Submitted at
Institute of Computational Perception

Supervisor
PhD Francesco Foscarin

07 2024

Autoregressive Jazz Piano Music Generation from Midi Tokens

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Artificial Intelligence

Abstract

The aim of this Bachelor Thesis is to test the generative abilities of GPT-2 style Autoregressive Transformer models, trained on MIDI-playalongs from the Jazz genre. It builds on my previous work in the Practical Project, where I extracted piano sources from the Aebersold dataset and converted them to MIDI. I tested and evaluated three different models with the same model architecture but trained on differently pre-processed data. When tasked to continue from jazz-style inputs, all three models exhibited basic structural understanding, but only limited rhythmic and harmonic coherence, especially when generating sequences longer than those they were trained on.

Contents

1 Introduction

Jazz music with its complex harmonics and improvisational playing styles is a real challenge for computational music generation. In this thesis I will approach this challenge as a sequence-modeling problem, exploring the abilities of conventional Transformer architectures [15] to generate Jazz music.
Adding to the complexity of the problem, the models are trained on human-played Jazz piano music, generated from a dataset of Aebersold playalongs [7], which does not contain any information on tempo or meter.
I will use a pre-processing framework, which I established in previous work [13], to create MIDI files which are then converted into Transformer-ready token sequences.
After training a GPT-2 [10] type Transformer model on three differently pre-processed sets of the data, I will perform a small-scale evaluation of the resulting three models' abilities to continue from jazz-style inputs. This will be done with a testset, partly handcrafted and partly taken from the Aebersold corpus - unseen by the model.
Evaluation criteria include maintaining the established tempo, meter, and style, as well as adhering to the established harmonic context.
With this work I want to investigate, if Transformer architectures are able to learn and replicate the complicated characteristics of Jazz music.
Samples of the generated model outputs can be examined and listened to in my github repository [14].

This thesis is divided into 5 main sections; in Chapter 2 I will introduce the dataset, Chapter 3 outlines the preparation of the data and its limitations, Chapter 4 introduces the model, and in Chapter 5 I will illustrate the experimental setup as well as the training and evaluation strategy. I will then finish with my Conclusions and future outlook.

2 Dataset

In this chapter, I will first give a brief introduction to the MIDI standard and 'Quantization,' an important concept in digital music representation. Then, I will outline how the dataset used for the experiments was generated.

2.1 MIDI

MIDI (Musical Instrument Digital Interface)[1] was established in the 1980s as an industrial standard for connection and communication between electronic musical instruments and computers. It allows different devices to send and receive musical instructions, such as note pitches, note on and off messages, and control changes. MIDI enables musicians to record, edit, and perform complex compositions with multiple instruments synchronized together. It also supports a variety of control messages for modulating sounds and effects in real-time. Nowadays this standard has become essential in music production, live performances, and audio software. MIDI-messages can convey a wide variety of information, the most important, in the scope of this project are illustrated in Figure 2.1

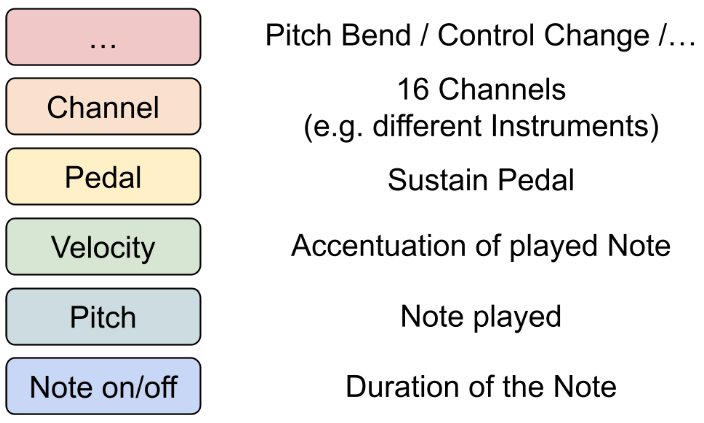

Figure 2.1: Information Encapsulated in a MIDI Event

2.2 Quantization

An important difference between sheet music and recorded live-performances in MIDI format is quantization. Sheet music adheres to a fixed temporal structure with bars, fixed note and pause durations, and an overall tempo. Live-performed music usually does not maintain a consistent tempo unless recorded with a so-called click track, which fixes the overall tempo. However, slight deviations introduced by human musicians can still occur. Note onsets, note durations, and time vary with the style of the music and the interpretation of the artists. To be able to represent these nuances in a recording, MIDI inherently quantizes to an extremely fine-grained grid, with the potential resolution only constrained by hardware limitations. More on this in section (3.1.1).

2.3 Dataset Generation

The dataset used in this project stems from my previous work in the seminar 'Practical Work in AI' [13]. There I constructed a piano transcription pipeline from state-of-the-art audio transcription tools

and applied it to the Aebersold dataset [7], a collection of high quality Jazz play-alongs, played and recorded by professional musicians. These play-alongs are used by Jazz musicians to practice standards, scales and improvisation. Originally they consist of a drum, piano, and bass track, which are separated during transcription. From this I received 1249 piano transcriptions in MIDI format, between 39 and 814 seconds in duration, with an average number of ∼3161 Notes per file. For a distribution of notes per file see Figure 2.2.
Similar to recordings from a live performance, these MIDI files do not contain any information about key or time of the pieces, only a tempo estimated by the transcription system.

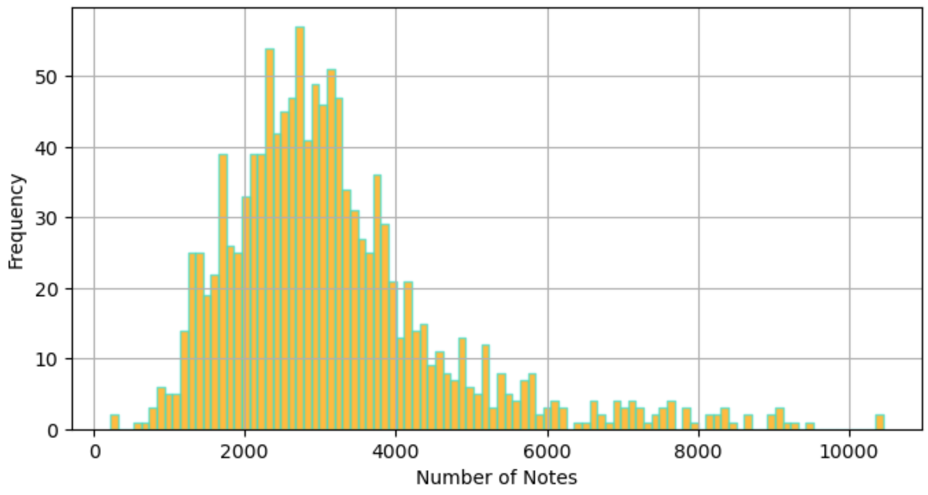

Figure 2.2: Distribution of Number of Notes per Piece

3 Data Preparation

In order for the machine learning models to process input sequences like text, or in our case MIDI, these sequences need to be in a convenient numerical form. Here the tokenizer comes into play. It maps small parts of the input sequence i.e. MIDI-events to specific tokens which are usually just integer numbers, forming a dictionary. This means that first, a vocabulary of possible tokens has to be defined, either manually or, as it is common in the NLP domain, learned from the corpus. To keep this dictionary to a reasonable size for training a model, it is often restricted to a certain number of tokens, depending on the domain. Inputs that are not present in this dictionary, are usually mapped to a special 'unknown' or 'out-of-dictionary'-token. Also special 'begin-of-sequence' as well as 'end-of-sequence' tokens are defined to indicate start and end points to the model during training.
In the MIDI domain, this process is not so straight-forward. As I illustrated in 2.1, one MIDI-event contains many different layers of information. To account for that, the tokenizer has to map one MIDI-event to several tokens in a sequential way, one for each piece of information.

3.1 Tokenizing MIDIs

To handle the special structure and sparse information encompassed in MIDI from recordings of music played by humans, I had to look for a tokenizer that is able to represent a sequence of MIDI notes only based on note duration, pitch and velocity[1].
The Structured tokenizer from the MidiTok [4] package, with the underlying idea first introduced by G. Hadjeres and L. Crestel in their work *The Piano Inpainting Application* (2021) [5], is a suitable candidate for this task. It follows a recurrent token type succession, as illustrated in Figure 3.3. Temporal distance between consecutive notes are indicated with the time-shift token, where simultaneous notes have time-shift value of 0. This allows to represent note sequences of any length, as long as the maximum pause between notes does not exceed the maximum time-shift value (which is typically set to the length of 4 bars). The dictionary of this tokenizer has to be predefined. This is straightforward for discrete numerical spaces such as pitch and velocity where ranges are already predefined by the MIDI-standard but can also be narrowed down to the use cases. In my case, the pitch range is restricted to [21,109], which is the range of the 88 keys of a piano (A0 to C8), and the velocities are binned into 32 bins instead of the original 128 possible values.

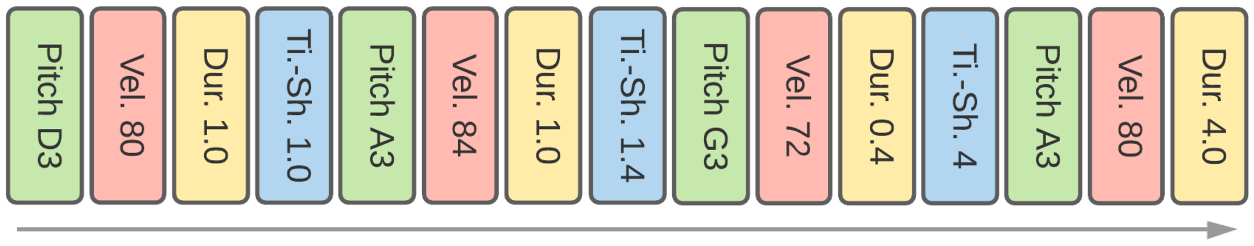

Figure 3.3: Example Token Sequence of the Structured Tokenizer [4]

The not-so-straightforward part is the time resolution for duration and time-shift tokens. The beat resolution parameter of the tokenizer divides into two parts; beat resolution in the first bar and in the second to fourth bar. This means that relative distances to previous notes can be represented more accurately if they are within the duration of 1 bar. Durations between two consecutive notes, that exceed the maximum distance (set to 4 bars in my case), are set to the maximum value. This is a major disadvantage of this tokenizer. Additionally, one has to decide for the optimal trade-off between vocabulary size and time resolution. Larger vocabulary size meaning more computational complexity,

[1]The intensity with which a key was pressed, which equates to volume of output note.

smaller time resolution meaning less fine-grained tokenization. I decided for a resolution of 16 parts per quarter note in the first bar, and 4 parts per quarter note in the second to fourth bar, which results in a total of 112 time shift tokens. This problem is further discussed in Section 3.1.1.

3.1.1 Limitations

A sound wave is a continuous signal that can be represented as a function of change in air pressure over time. To be able to digitally store this signal, measurements at very small intervals have to be made. The time resolution of these measurements per second is called Hertz (Hz), named after the physicist Heinrich Hertz, who was a pioneer in the study of electromagnetism and radio waves.
Music is typically recorded with a resolution of 44100 Hz, or 44.1 KHz. MIDI measures the time resolution in ticks or pulses per quarter note (PPQ), and the MIDI-standard supports up to 960 PPQ [1].
If we wanted to represent this high resolution in our tokenizer dictionary we would have to use a number of tokens four times as high, to only be able to model the first bar, assuming we have a 4/4 time signature. As our tokenizer encodes time differences between consecutive notes on a relative basis, also long pauses between notes have to be accounted for. Thus we need time-shift tokens that span two and more bars, also with reasonable resolutions. In the end we have to decide for the optimal trade-off between dictionary size and time-resolution, keeping in mind that larger dictionaries increase model complexity and training costs.
In the pianoroll representation in Figure 3.4 you can visually compare the effect of different PPQ settings on a MIDI-performance after tokenization. The resolutions illustrated are 8 PPQ (tok-8) and 16 PPQ (tok-16) for the first bar. It is clearly visible, that the onsets, as well as the durations of the notes are affected by the tokenization process. Although getting closer, the tokenizer with the higher PPQ resolution is also not able to represent the fine nuances that are so important for musical performances, especially in Jazz music. In future works, if computational resources permit, improving results could involve implementing finer resolutions in the tokenization step.

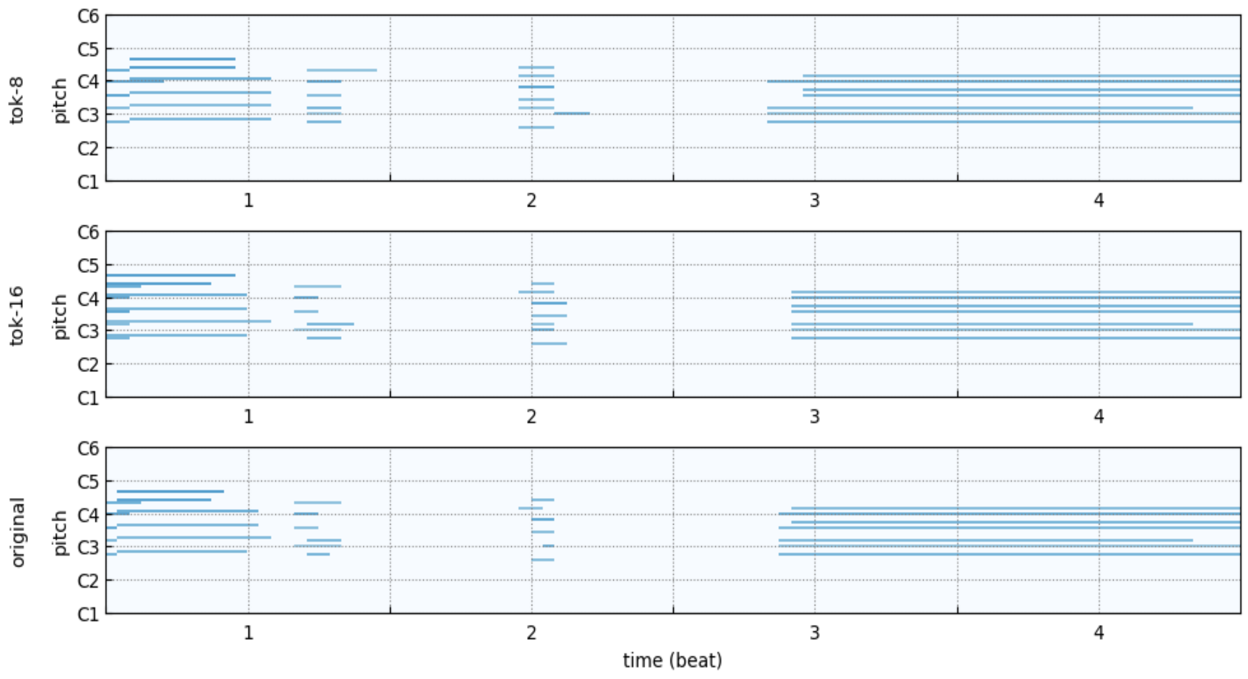

Figure 3.4: Tokenization Resolution Comparison

3.2 Splitting MIDIs into chunks

As training on the entire Midi-files would lead to very long token sequences, infeasible for training a model, I had to decide for a strategy to split the files into smaller chunks. MidiTok comes with a built-in function for this, where the main parameters are 'max_midi_length' and 'overlap'. Whereas max_midi_length is more a guideline as the function tries to split at bars and thus deviates from this value, especially if overlap is zero. With overlap set to 1 bar, the function varies the length of the overlap such that the final length matches max_midi_length. Consequently the resulting sequences are as least as long as the set max value, which seems counter-intuitive at first, but for training generative models the input sequences need to be of same length. If many inputs were shorter than this length they would have to be padded with zeros. This reduces efficiency because the zeros contribute no training information, yet still need to be processed. Chunks at the end of files will still be shorter. See Figure 3.5 for a comparison of the distribution of sequence lengths, depending on the overlap value used for splitting.

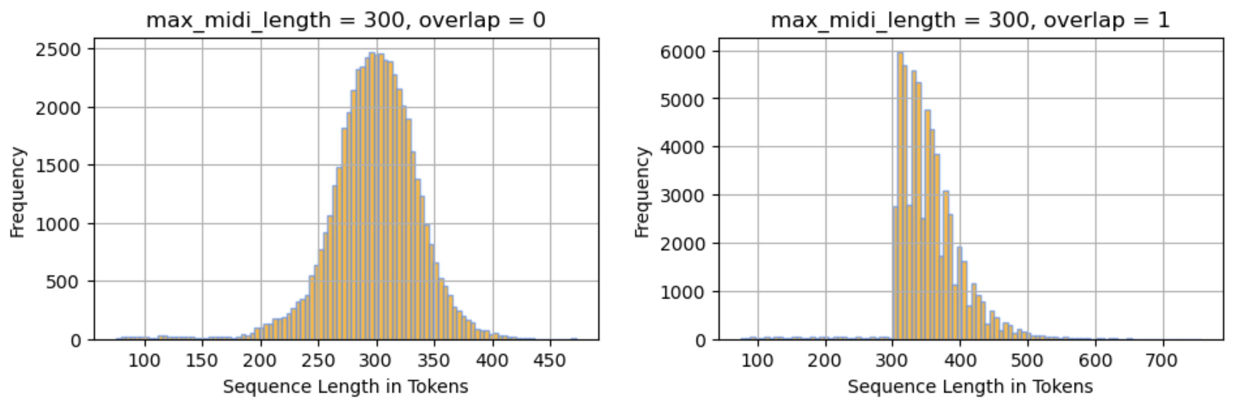

Figure 3.5: Comparison of Distribution of Lengths of MIDI-Chunks with Different Overlap Settings

The final sequence length the models will be trained on, is set in the configuration of the tokenizer. The models need training inputs in a fixed and predefined length, therefore the tokenizer truncates or pads inputs that are longer or shorter than that set value. The maximum feasible sequence length for my computing resources was 300 tokens which results in chunks that are on average longer than 4 bars [2], as you can see in Figure 3.6.

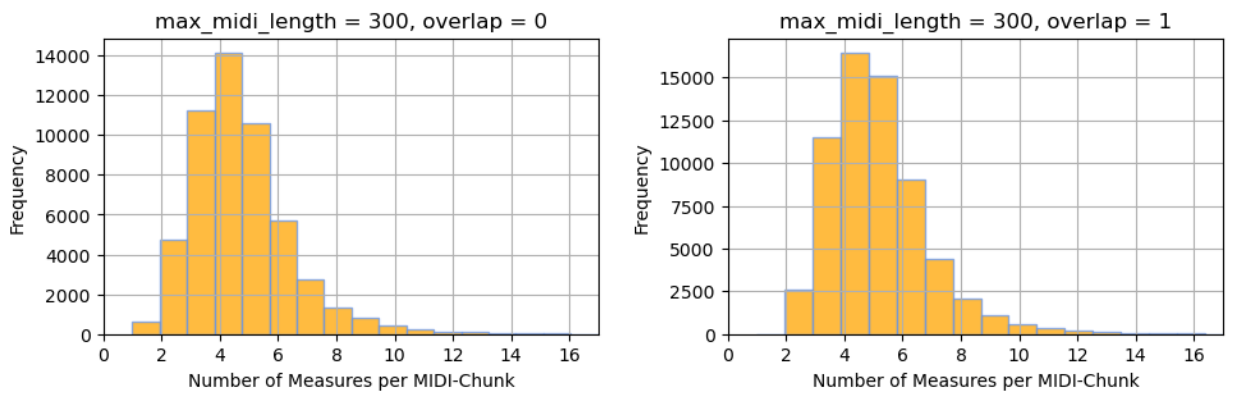

Figure 3.6: Comparison of Distribution of Number of Bars per MIDI-Chunk

[2]Note that this calculation relies on the estimated tempo from the piano transcription framework.[13]

This should give the models enough context during training to learn short chord progressions and melodies even without overlapping inputs.

4 Model

For this project I used the well known Autoregressive Transformer architecture GPT-2 [10], available through the Hugging Face platform[3] via the transformers library, which builds on pytorch lightning [8]. Despite providing numerous pre-trained models mainly from the NLP domain, Hugging Face also offers access to untrained model architectures that are easy to configure and come with ready-to-use internal implementations. These include processing of input sequences, such as masking and embedding, as well as various generation strategies.
Transformers use a technique called self-attention which enables them to not only take the tokens in a sequence into account, but also their position in relation to the other tokens in the sequence. This knowledge is then included when creating representations of tokens in high dimensional space, so-called 'embeddings', from an input. GPT-2 uses absolute position embeddings, this means absolute positions of tokens in a sequence are learned. Opposed to relative embeddings, this requires less computational effort, but consequently the models are often unable to generate coherent output sequences, which are longer than the sequences they were trained on[12].
GPT-2 uses the GELU-activation function[6] for it's hidden layers.
With only basic computing resources available, I used a rather small version of this architecture, see the model configuration in table 4.1 .

Hyperparameter	**Value**	**Description**
model_type	gpt2	Type of model
n_ctx	1024	Size of the context window
n_embd	512	Dimensionality of the embeddings
n_head	16	Number of attention heads
n_layer	16	Number of layers in the model
n_positions	1024	Maximum input/generation length
activation_function	gelu_new	Activation between hidden layers
Total Number of Parameters	51.13 M	

Table 4.1: GPT-2 Configuration Parameters

4.1 Training Loss

Language models apply a so-called next-token prediction loss. The model gets the first token of the sequence as input and tries to predict the next, then it gets the first two tokens as input and tries to predict the third, and so on. At every step the predictions are evaluated with a loss function over all possible tokens and the losses summed up. In our case the Cross-Entropy loss is used which is defined for a single step as follows:

$$\mathcal{L}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{4.1}$$

Where C is the number of classes (different tokens), y_i is the true probability of each class i and $\hat{y}_i$ is the probability predicted by the model for this class. Using the negative logarithm this loss grows exponentially the further the predicted probability for the correct class is away from 1.

4.2 Autoregressive Sequence Generation

The main objective of training these models, is to be able to generate valid and coherent token sequences from an input sequence, or even from empty input. Autoregressive in this sense means, that after every token generated, the model gets as input the whole sequences of input, plus everything generated up to that point (assuming the sequence length does not exceed model capacity). Models like this are also often referred to as 'causal models', hinting at the causality chain, linking inputs and outputs.

4.3 Generation Strategies

There is a wide variety of sampling strategies that can be applied to the model output at inference to generate sequences. At each timestep in the generation process, the model returns its predictions as a probability distribution over all tokens in the tokenizer's dictionary. The simplest way to generate a sequence from this distribution is called 'greedy sampling', where always the token with the highest probability is chosen. This often results in incoherent sequences with very low or no variety. To introduce randomness and thus more variety, the probability distribution of the predicted next token can be sampled. This is called 'Multinomial Sampling', where each token is chosen according to the predicted probability that it is the 'correct' next token.

4.4 Multinomial Sampling

Instead of sampling from the distribution of the whole set of possible tokens, the sampling is often restricted to a subset. There are two prominent varieties, namely 'top-k' and 'top-p' sampling. In top-k sampling, the distribution is restricted to the set of 'k' most probable tokens, in top-p sampling, also called 'nucleus sampling', the subset of tokens is considered, whose cumulative probability exceeds the chosen threshold $p \in [0, 1]$. This top-p approach is thus more adaptive to the confidence of the model in its prediction; when confidence is high, the next token is sampled from a smaller selection of tokens, which has proven beneficial to the overall coherence of a sequence. [11]
Another important parameter for generation is 'temperature' where $temperature \in [0, inf]$. The logits-ouputs of the models are divided by this value, before the probabilities are computed. Values close to 0 increase the peakiness of the probability distribution, thus favoring the more probable tokens. Values closer to and larger 1 dampen the distribution, leading to more exploration and randomness.

5 Approach

To evaluate the abilities of this framework, to capture and reproduce the musical knowledge contained in the data, I trained three different models with similar hyperparameters, but on differently pre-processed data:

Model	Training Data
baseline	Trained on MIDI chunks without overlap
baseline_olap	Trained on MIDI chunks with 1 bar overlap
bos_only	Trained on beginnings of MIDI's only

Table 5.2: Training of the Different Models

With the baseline_olap model, I want to check if overlapping training data can help the model, to better learn causality between chunks and thus generate longer coherent sequences. It will also be interesting to see how the model performs, when generation length exceeds the length of training inputs, which is usually the limitation when using absolute positional embeddings, as outlined in 3.1.1. The purpose of the bos_only model is to check, whether giving only the beginnings of pieces to the model as training data, is already enough for the model, to learn how to continue from the first few bars of a piece, like the 5 pieces in our testing data. This reduced the training set to as few MIDI chunks as there are different MIDI files (1249) as opposed to 52,908 (baseline) and 63,786 (basline_olap) MIDI chunks for the other two models.
All experiments were conducted with top-p sampling, where p as well as temperature is set to 0.95, which after several trial runs proved to be a fair trade-off between exploration and exploitation. With smaller values the generated notes were very clustered together and the pieces very short. With larger values the models produced very long outputs with lots of scattered single notes.
All models were tasked to generate sequences of 500 tokens (equivalent to 125 notes), which exceeds the length of the sequences used for training by 200 tokens. This should be enough to get insights in how the models perform after training length.
Already during training I noticed, that early models which were still performing quite bad, were not even able to generate the tokens in the right order for the tokenizer to transform them back to MIDI notes. I then implemented a simple control function, that counts generated notes and compares them to the number of generated tokens, to check if all tokens end up being valid MIDI notes. All three models in the experiments passed this check.

5.1 Training

After a manual hyperparameter search on the baseline model, all models were trained on the found optimal parameters and with slight adjustments of the learning-rate, if convergence was not as expected.
All models were trained until convergence of the evaluation loss with early-stopping patience of 3 epochs[3]. The optimizer used during training was the AdamW-Optimizer[9] which uses weight decay for regularization and also supports learning-rate scheduling, both features have proven beneficial during hyperparameter-search. All models have been trained on a NVIDIA GeForce GTX 1060 6GB GPU. For logging the weights-and-biases package [2] was used. Training duration varied depending on final dataset size, which differed for each model and ranged from 2.5 (bos_only) to 14 hours (baseline_olap).

[3]This means that after 3 epochs without improvement of the evaluation loss, the training stops.

5.2 Evaluation

Since music, and Jazz music in particular, is a highly subjective field, the evaluation of generated pieces is not so straight forward, as there is often no clear right and wrong. There are still aspects of music though, that can be looked at, to see if the trained models perform as expected. For this, I created a test set consisting of pieces from the original training data, not seen by the models during training, as well as short human-played pieces, with characteristic Jazz-style elements. Samples generated from this test set as inputs are then evaluated under the following aspects, to see how well the models can understand and continue from the concepts presented to them as inputs:

5.2.1 Token Type Succession

To generate MIDIs from the generated token sequences, the models' output sequences have to be in the order expected by the tokenizer: Pitch, Velocity, Duration and Time Shift. Thus if there is any MIDI output at all, we can already conclude that the models learned the correct token type successions.

5.2.2 Tempo and Meter

When given input with a specific tempo and meter, I expect the model to continue in the same tempo and meter, only deviating in artistically common ways (ritardando[4], expressive performing).

5.2.3 Style

As our data is from the Jazz-genre, which again has countless subgenres with varying stereotypical playing styles, rhythms and chord progressions, it will be interesting to see if the models are able to learn these concepts.

5.2.4 Key/Mode

Depending on the subgenre, Jazz music is highly modular, meaning that the tonal center is changing throughout the piece, unlike pop-music where there usually no key changes in a piece [5]. I do not expect the model to exhibit this behavior, but rather that it is able to stay in the key it is presented with in the input.

[4]Intentional slowing down at the end of a piece.

[5]There is a common modulation in pop music where at the end of a song the last chorus is modulated to a higher pitch, to make the last repetition more uplifting and interesting.

5.3 Testset

For testing the model outputs we set aside 5 MIDI files from the corpus, to give the first four bars as input, as well as 3 additional human-played MIDIs containing a single chord, a standard jazz chord progression (I-ii-V[6]) and an arpeggiated chord[7]. Also generation without input is tested. For a complete list of inputs for the experiments, see Table 5.3.

Input	**Description**	**Type**	**Length**
No input	Generation from empty input	No input	0
Single chord	One short C-major chord, all notes simultaneous	Simple	20
Arpeggio	Slow C-major arpeggio	Simple	36
Turnaround	Common chord progression, I-ii-V, F-major	Simple	52
Blues	Standard blues form with walking bass, fast	Complex	136
Turnaround	Standard jazz chord progression, slow	Complex	216
One Note Samba	Latin jazz standard, fast chord progression	Complex	256
In a Sentimental Mood	Jazz standard, slow, melodic	Complex	300
Cantaloupe Island	Jazz standard, iconic piano intro, medium tempo	Complex	476

Table 5.3: Generation Inputs to Evaluate Model Performance, Length is Number of Tokens

[6]The roman numerals denote the relation of the chords to the tonic or root (I), upper case letters for major chords and lower case for minor. F.e. if we were in C-Major this progression would be C-Dm-G.

[7]Chord notes are played one after the other, in a repeating pattern, f.e. low to high and back.

5.4 Experiment 1 - Generation without Input

Generating from no input at all is a very challenging task for generative models, as they can only sample from the overall distribution of starting tokens and then continue from that. This often leads to nonsensical output.

5.4.1 Results

The outputs of all three models exhibit chord like structures which are coherent in style and tempo for the first part of the sequence. Then, after passing the training sequence length, especially for the two baseline models, we can see that the generated notes still adhere to some chord like structure and beat, but notes are much more spread and shorter. Also the parts at the beginning perceptively share a common tonality whereas the latter parts rather sound like random keystrokes. The bos_only model tends to generate slower but longer and more melodic pieces. See a piano-roll representation of generated samples of all three models in Figure 5.7.

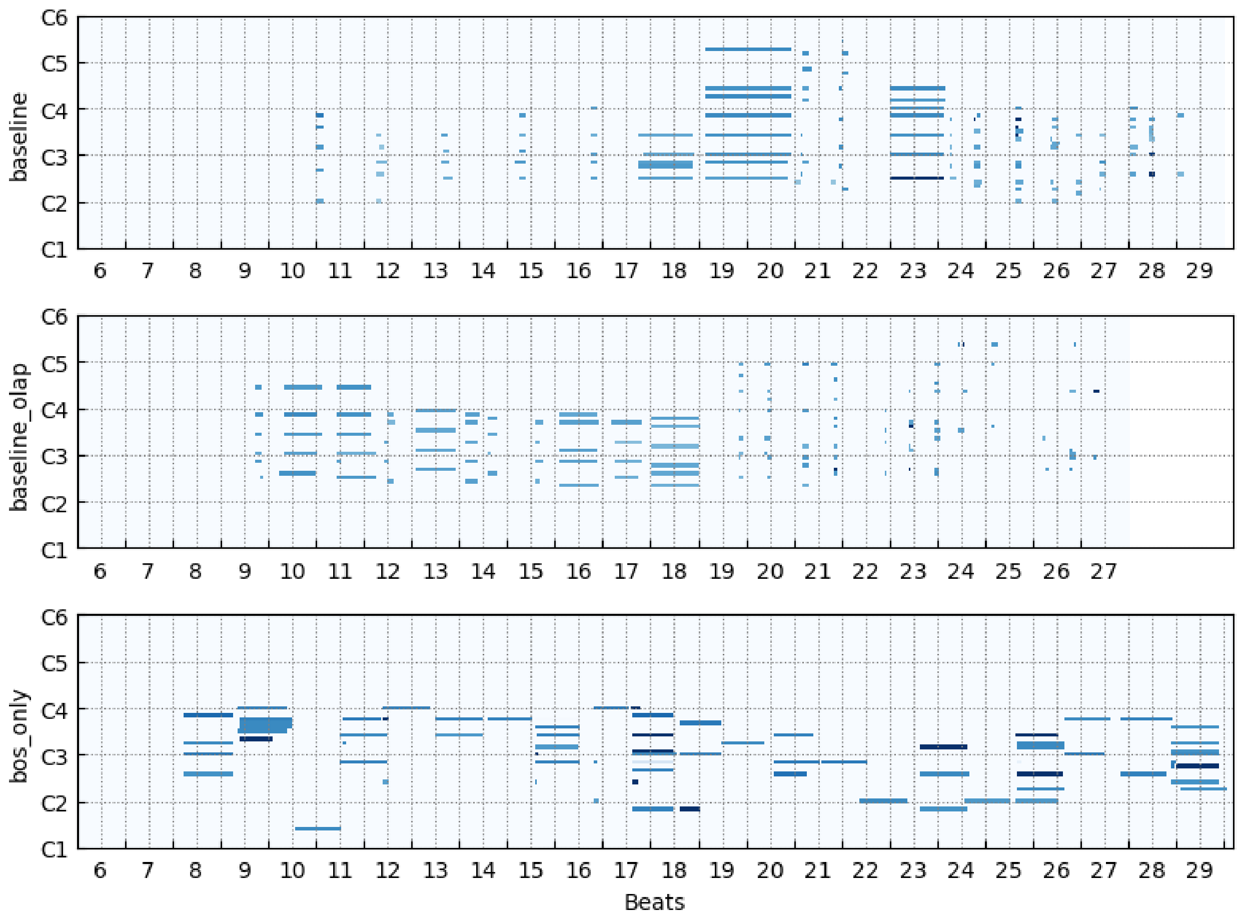

Figure 5.7: Generation from no Input

5.5 Experiment 2 - Generation with Simple Input

The three simple inputs for this experiment consist of a single C-Major chord, a C-Major arpeggio, and a simple chord progression (I-ii-V) in F-Major. With this experiment I want to test the ability of basic musical understanding of the models. Namely if they can stay in the same key, use chords and notes that fit the established harmonic context, and continue in the same tempo and style.

5.5.1 Results

From the single chord input the models generate solid chord sequences up to training sequence length, then they lose coherence quickly and revert to randomized scattered notes. The chords roughly fit into the tonal context and the tempo is mostly constant. Interestingly all three models tend to generate notes already on top of the input instead of appending them. Inspect the generated samples in Figure 5.8

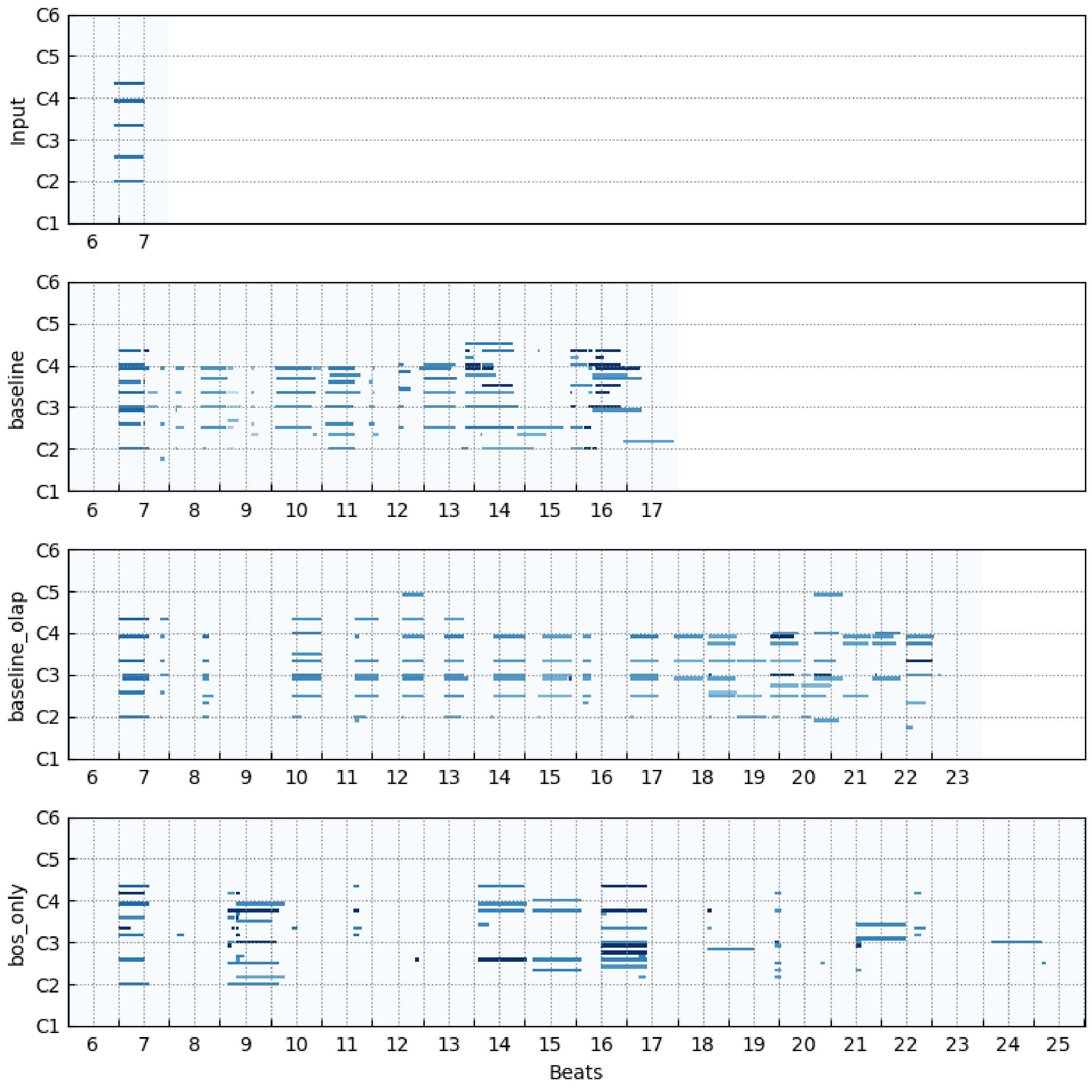

Figure 5.8: Sample Output from Single Chord Input

From the arpeggiated chord input both baseline models quickly revert to chords where notes are played simultaneously, as this playing style is more common in the data. Still they manage to hold the same tempo. Only the bos_only model tries to continue the pattern but quickly looses coherence, generating random notes going up and down, probably lacking the deeper musical understanding behind this pattern. You can inspect generated samples in Figure 5.9.

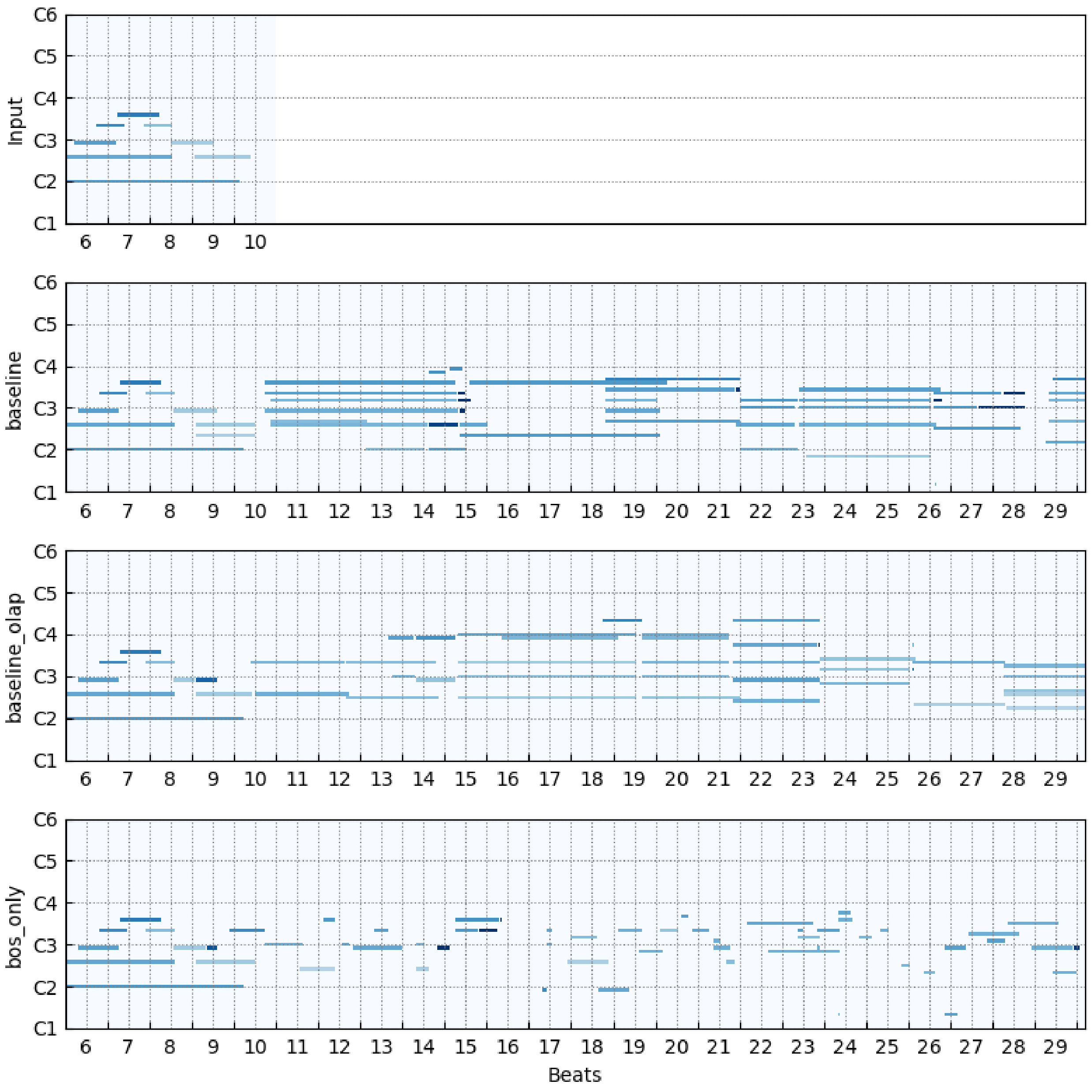

Figure 5.9: Sample Output from Arpeggiated Chord Input

From the I-ii-V chord progression as input, the two baseline models exhibit some understanding of chord progressions, although they rather piece together pairwise related chords and move away from the established tonal center quickly. The bos_only model again looses coherence very easily, generating clusters of notes short in duration at random. See another comparison of sample output in Figure 5.10

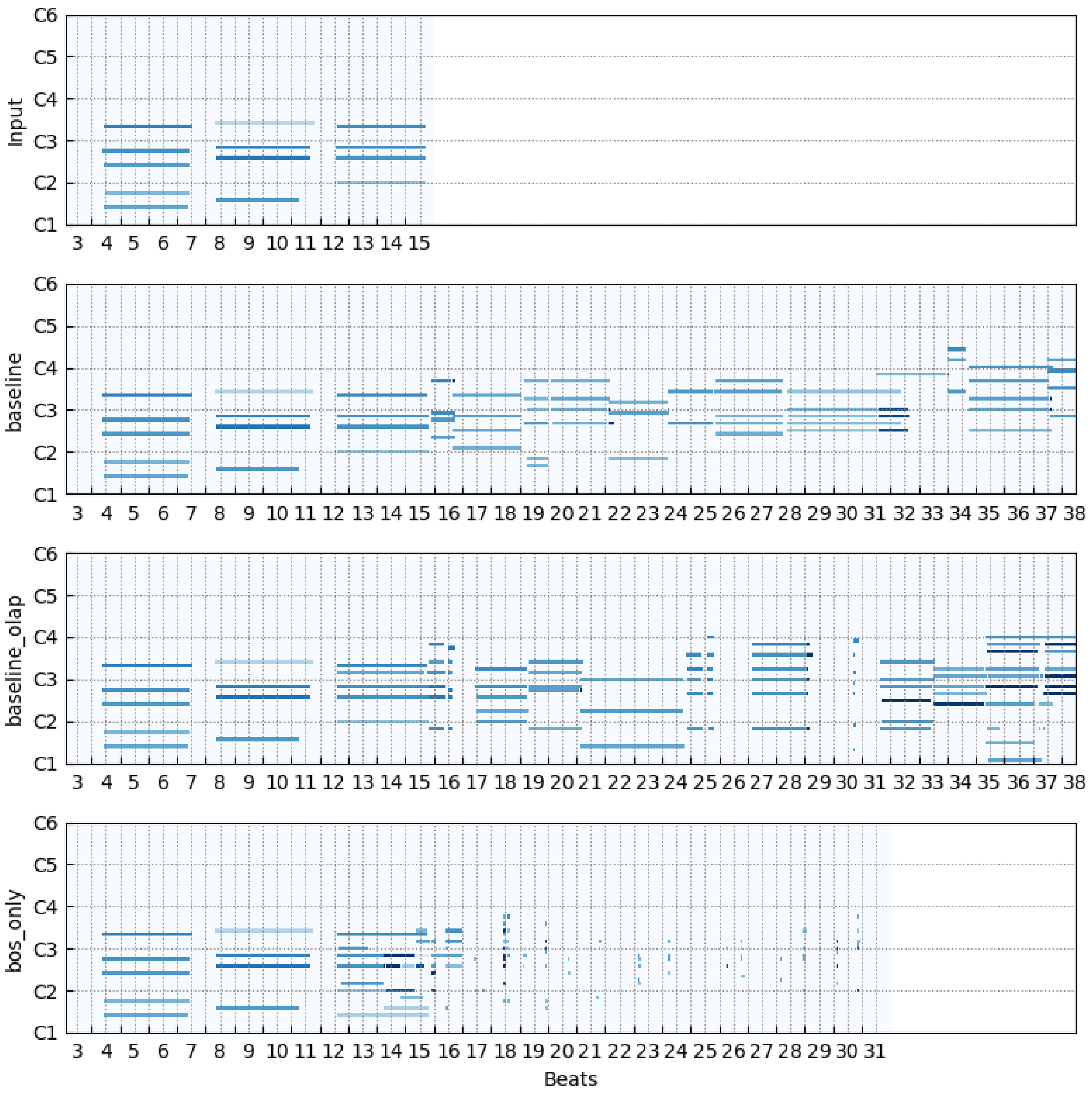

Figure 5.10: Sample Output from I-ii-V Chord Progression Input

5.6 Experiment 3 - Generation with Complex Input

In this experiment I looked at model performance from complex input, taken from the corpus itself, but not seen during training. The model is given the first 4 bars of a typical piece from the corpus as input and the generated output is compared to how the piece would continue in reality. This experiment is especially interesting for the bos_only model, that only got the beginnings of all pieces as training data.

5.6.1 Results

From the five complex inputs I will describe the three that provide the most insight into model performance. Starting with the Blues input, you can see in Figure 5.11 that all three models try to continue the 'walking' bass line and chords on top. Both baseline models are quite successful to keep up the rhythm and style, whereas the bos_only model again looses coherence quickly. Harmonically all three models diverge from the established tonal context and as soon as they reach training sequence length they loose coherence.

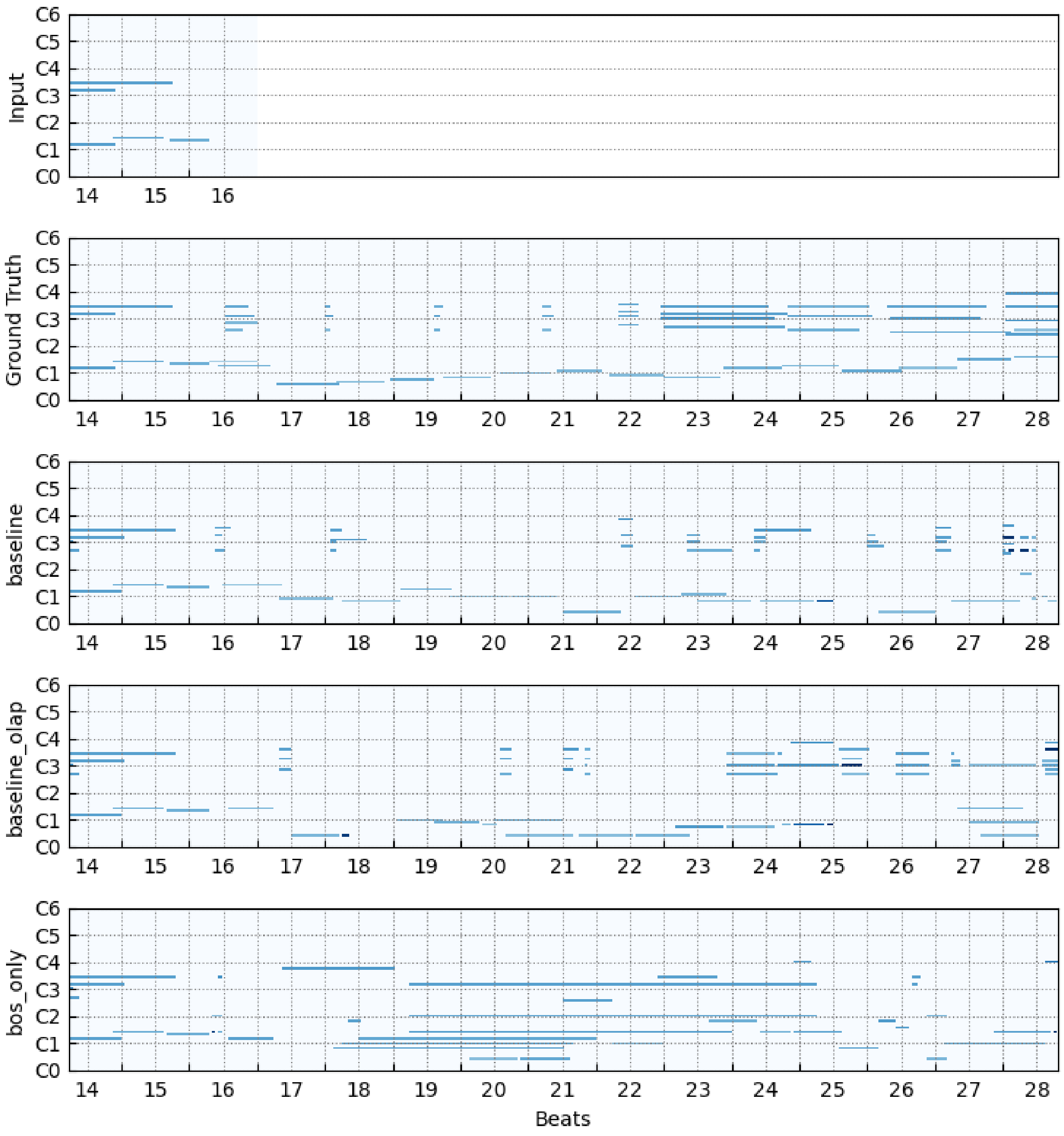

Figure 5.11: Sample Output from Blues Input

The second complex input I will look into is another turnaround with typical jazz chord progressions. The baseline model picks up the slow pace of the piece as well as the playing style and generates a nice and coherent chord sequence. The other two models rather generate random chords and do not adhere to the tempo of the input. What you also can see well in the sample visualization in Figure ?? is, that they use very high velocities at times not present in the input.

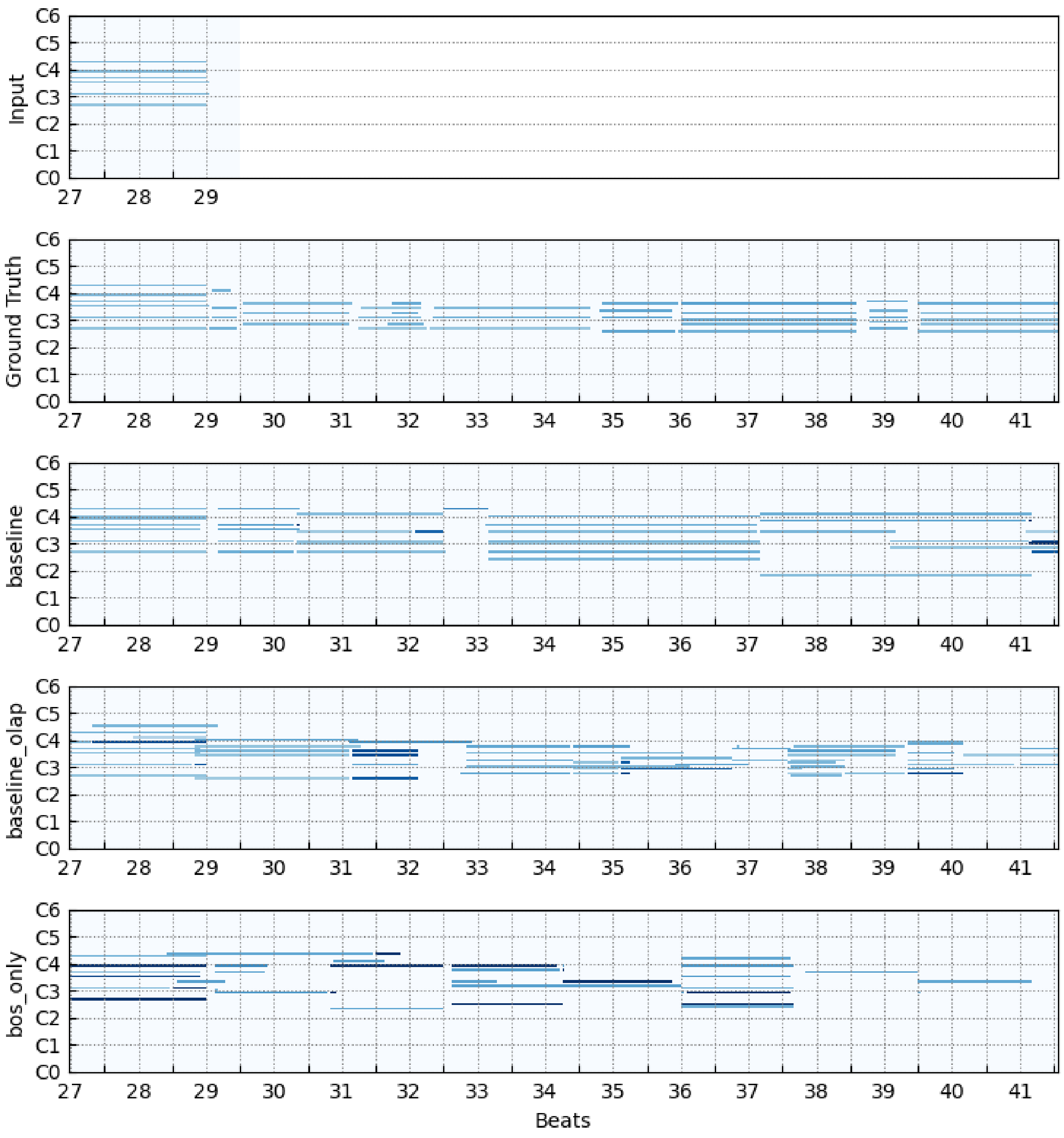

Figure 5.12: Sample Output from Turnaround Input

The third input is an interesting one, as the input alone already surpasses training sequence length. While the baseline model without overlap generates more or less randomly the other two models are kind of able to pick up the rhythm of this jazz-funk style piece and continue it in a rhythmically coherent way. Harmonically they drift off, but not as much as expected.

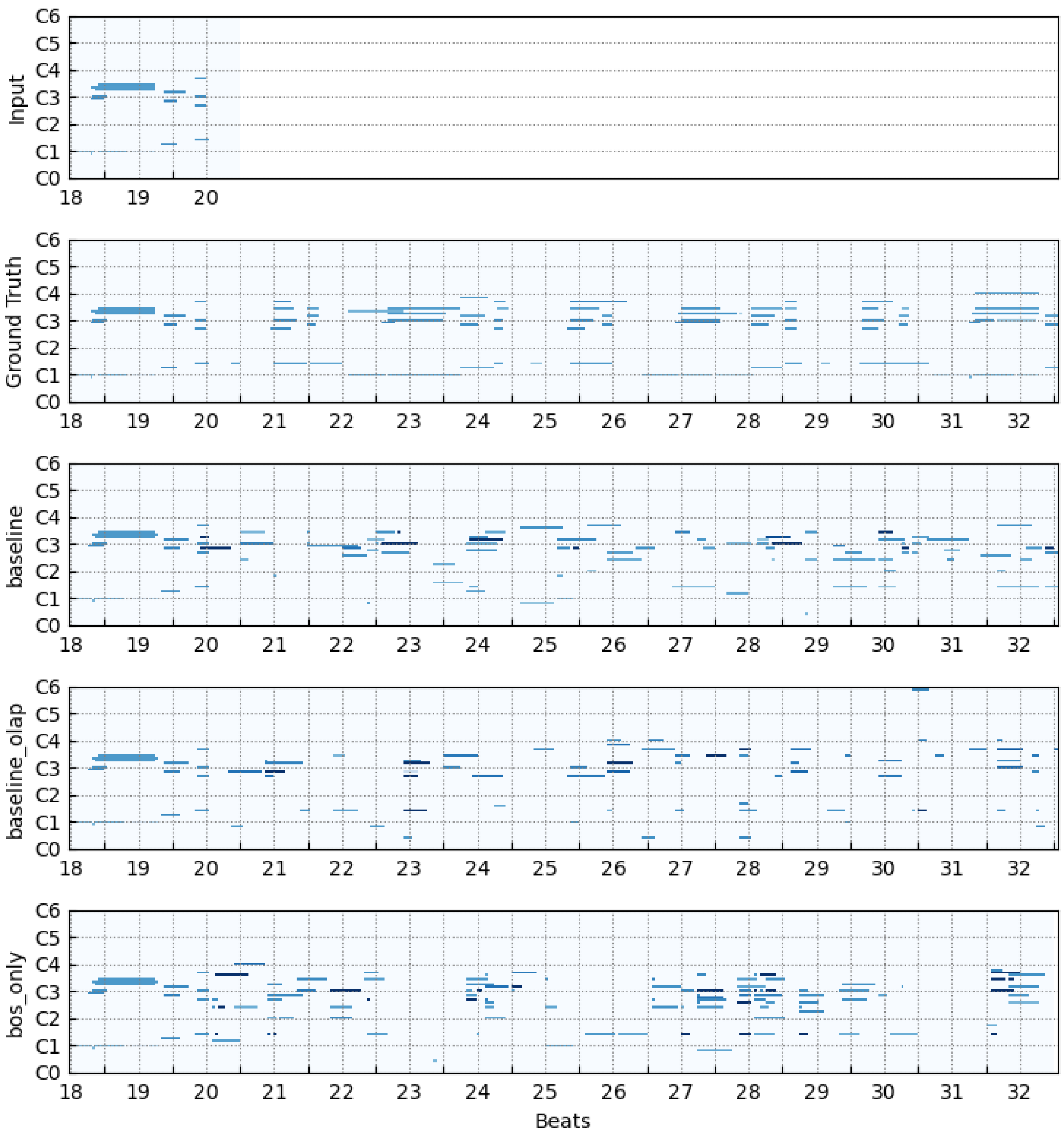

Figure 5.13: Sample Output from Cantaloupe Island Input

5.7 Experiments with Data Augmentation

Data augmentation is often used to generate additional data, which may help the model to generalize better to unseen input. With audio data, this can be achieved by modulating the key of the piece through all 12 semitones. For MIDI, this is straightforward, and a method for this is already implemented in the MidiTok package. There would also be the possibility to augment tempo and velocities, but I did not experiment with these options, as they would add too much complexity to the problem.
I used this approach for experiments with the bos-data, where only the beginning of all the pieces were used, as this dataset was quite small from the beginning. I noticed very strong overfitting behavior during training when using the augmented bos-data, even for very small learning rates. Therefore, I had to cut the training process very early. There was no notable improvement on the generated outputs compared to the bos_only model, thus I left it out of the main experiments.
To evaluate the overall benefit of data augmentation, I would have had to train all three models on augmented data and compare the outcome. As a dataset size of 12-fold very much exceeds my current computational resources, I leave the exploring of this option for future work.

6 Conclusions

In this thesis I used human-played jazz music in MIDI format to train 3 different autoregressive Transformer models. Then, I conducted several experiments to evaluate their capabilities in replicating the musical concepts they were presented with during training.
Overall it can be assumed that the models performed better on the complex inputs, probably due to their similarity to the training data as well as their length, giving the models more context. Also the baseline model trained with overlapping sequences is subjectively more successful in keeping up coherence after surpassing training sequence length.
Still outputs of all three models are far from accurate with respect to either rhythm or harmonic structure, but they show glimpses of basic structural understanding.
With a lot of randomized processes involved, I was not able to identify a clear superior model between the three models evaluated in the experiments. Although the bos_only model mostly performs worse than the two baseline models, we have to keep in mind that those were trained on 50 times as many data. For a few inputs bos_only even matches the performance of the other two. In general it shows better performance on slower, more melodic inputs, maybe because it did not overfit so much on chord progressions.
Only in coherence after reaching training sequence length, the basline_overlap model seems to be on an advantage.
To keep model complexity to a reasonable size for my computational resources, I stuck to the tok16 tokenizer resolution. A higher resolution could have been beneficial, but would have to be combined with a more capable (larger) model to handle the added complexity.
Working with limited computational resources, I did not expect my models to achieve breakthrough performance. Therefore, it is interesting to see that despite the highly complex structure of jazz music, the models have learned and reproduced basic musical concepts such as chord structures and rhythmic patterns. I think, with improved hardware, this framework could potentially yield much more adequate and usable results.

My hope for future applications of models like these, was a framework where the user inputs textual instructions, like desired chord progressions, style, tempo, and meter and it would output a playalong accordingly. Besides that this framework would require a whole pipepline of models, and one would have to train much bigger models on more data, to really get usable outputs. It will be interesting to see the advancements in this field, especially related to Jazz music, as there is just not a lot data for training (yet).

7 Acknowledgments

This thesis was completed as part of the Artificial Intelligence Bachelor program at Johannes Kepler University Linz. I would like to extend special thanks to Professor Gerhard Widmer's team at the Institute of Computational Perception, particularly to my supervisor Francesco Foscarin, for his invaluable guidance and advice throughout this thesis, as well as during the preceding seminar and practical work.

References

[1] MIDI Association. Summary of midi 1.0 messages. https://midi.org/summary-of-midi-1-0-messages, 2024. Accessed: 2024-07-12.

[2] Weights & Biases. Weights & biases documentation. https://docs.wandb.ai/, 2024. Accessed: 2024-07-12.

[3] Hugging Face. Gpt-2 model documentation. https://huggingface.co/docs/transformers/model_doc/gpt2, 2024. Accessed: 2024-07-12.

[4] Nathan Fradet, Jean-Pierre Briot, Fabien Chhel, Amal El Fallah Seghrouchni, and Nicolas Gutowski. MidiTok: A python package for MIDI file tokenization. In *Extended Abstracts for the Late-Breaking Demo Session of the 22nd International Society for Music Information Retrieval Conference*, 2021.

[5] Gaëtan Hadjeres and Léopold Crestel. The piano inpainting application. *CoRR*, abs/2107.05944, 2021.

[6] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[7] Jamey Aebersold Jazz. Aebersold. https://www.jazzbooks.com/, 1967. Accessed: 2024-07-12.

[8] PyTorch Lightning. Pytorch lightning documentation. https://lightning.ai/docs/pytorch/stable/, 2024. Accessed: 2024-07-12.

[9] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[10] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[11] Shauli Ravfogel, Yoav Goldberg, and Jacob Goldberger. Conformal nucleus sampling. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 27–34. Association for Computational Linguistics, 2023.

[12] Koustuv Sinha, Amirhossein Kazemnejad, Siva Reddy, Joelle Pineau, Dieuwke Hupkes, and Adina Williams. The curious case of absolute position embeddings. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 4449–4472. Association for Computational Linguistics, 2022.

[13] Sebastian Sonderegger. Piano Transcription - Project in AI. https://github.com/seb-son/project-ai. Accessed: 2024-07-12.

[14] Sebastian Sonderegger. Bachelor's thesis github repository. https://github.com/seb-son/bac-thesis, 2024. Accessed: 2024-12-07.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum	Unterschrift