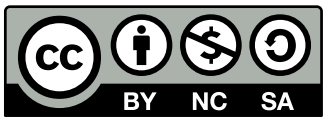
Aalto University
School of Science

Master's programme in Mathematics and Operations Research

Generating Complementary Instrument Tracks with a Transformer Model

Mikko Murhu

Master's Thesis
2024



Aalto University
School of Science

Author Mikko Murhu

Title Generating Complementary Instrument Tracks with a Transformer Model

Degree programme Mathematics and Operations Research

Major Systems and Operations Research

Supervisor Prof. Ahti Salo

Advisor PhD Shamsiiat Abdurakhmanova

Date 23.5.2024 **Number of pages** 49+8 **Language** English

Abstract

The recent increase in the accessibility of machine learning resources has led many to explore machine learning methods in music generation. A subfield of machine learning that has seen a particularly significant rise in popularity is deep learning. However, many model-application combinations in the field still remain unexplored.

In this thesis a sequence to sequence transformer model's ability to generate complementary instrument tracks for existing instrument tracks was investigated. The model was implemented and analyzed in order to assess whether it is able to generate instrument tracks that together with the original piece make coherent music. Although the results that the implemented model generated were not adequate for any real-life applications, indications for the model's viability for such applications were found, if a larger amount of resources is used and more research is conducted.

Keywords Machine Learning, Deep Learning, Optimisation, Music, Generative model

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Mikko Murhu

Työn nimi Täydentävien instrumenttiraitojen generointi transformer-mallilla

Koulutusohjelma Mathematics and Operations Research

Pääaine Systems and Operations Research

Työn valvoja Prof. Ahti Salo

Työn ohjaaja FT Shamsiiat Abdurakhmanova

Päivämäärä 23.5.2024 **Sivumäärä** 49+8 **Kieli** englanti

Tiivistelmä

Koneoppimiseen tarvittavien resurssien saatavuuden kasvu on lisännyt koneoppimismetodien suosiota myös musiikin generoinnissa. Erityisen paljon suosion kasvua on saanut koneoppimisen alakategorioista syväoppiminen. Siitä huolimatta moni mallin ja käyttötarkoituksen yhdistelmä on vielä tutkimatta.

Tässä opinnäytetyössä tutkittiin sekvenssiparitransformerin kykyä tuottaa täydentäviä instrumenttiraitoja olemassaolevalle instrumenttiraidalle. Työssä toteutettiin malli, jonka generoituja tuotoksia analysoitiin etsien vastausta kysymykseen siitä, kykeneekö malli tuottamaan instrumenttiraitoja, jotka aiempien instrumenttiraitojen kanssa ovat yhdessä koherentti kokonaisuus. Vaikka tulosten perusteella malli ei ollut riittävä käyttötarkoituksiin todellisissa sovelluksissa, merkkejä mallin kelpoisuudesta tällaisiin sovelluksiin oli nähtävissä, mikäli kehitystyöhön panostetaan lisää.

Avainsanat koneoppiminen, syväoppiminen, optimointi, musiikki, generoiva malli

Contents

Abbreviations

ML	Machine Learning
DL	Deep Learning
NLP	Natural Language Processing
(A)NN	(Artificial) Neural network
FNN	Feedforward Neural Network
RNN	Recurrent Neural Network
ReLU	Rectified Linear Unit
SDG	Stochastic Gradient Descent
(B)BPE	(Byte-level) Byte Pair Encoding
MIDI	Musical Instrument Digital Interface
REMI	Revamped MIDI-derived events

1 Introduction

Music generation is a relatively under-advertised area in machine learning, compared to other domains such as text and image generation, translation, speech-recognition and more. However, even music generation has been affected by the increasing popularity of deep learning applications in all machine learning [1]. State of the art music generation models are massive deep learning models that are able to generate whole coherent human-like songs purely in audio domain, either from nothing or from a conditioning such as text-prompt [1] [2] [3].

The Deep Learning methods in music generation include many varying types of models. Feed-forward networks, both fully connected and convolutional, have been used as parts of models in music generation. Recurrent neural networks such as long-short-term memory networks, as well as generative adversarial networks, variational autoencoders, transformers and diffusion models all have been shown to work in specific music generation tasks.

Other machine learning methods besides Deep Learning have been widely used as well for generating music. A Markov chain model can be used to calculate the conditional probability of the next musical element, given previous elements, and is therefore suitable to add continuation or additional layers to an existing piece of music. A genetic algorithm can be used to modify an existing music piece to another style, by defining the chromosomes as pieces of music and the genes as musical elements and creating a suitable fitness function that the algorithm uses to converge towards the new style. To add complexity, these kinds of models can be combined with different types of models, e.g. neural networks, to create increasingly sophisticated models. [1]

Transformer models have recently been one of the most popular and successful machine learning model types for their unparalleled parallelization applications and the ability to find long-term dependencies in sequences. They have thrived in many fields of machine learning, including natural language processing, computer vision, speech recognition and more. [4] Notably, currently some of the most well-known models in the field such as GPT [5] and BERT [6] models are versions of transformer models. However, transformer models have been utilized relatively little in music generation compared to many other methods. A review of music generation methods, which includes peer-reviewed papers concerning music generation from the years 2017 to 2021, listed only ten studies that utilized transformer models. The most common generation model type was a traditional RNN model, that amounted to 52 papers out of the total of 136 [1].

The objective of this thesis is to explore the capabilities of a sequence to sequence transformer model in generating an additional music track to an existing set of tracks in symbolic domain. The task is analogous to a traditional translation task, but the language to translate from is "music with a set of instruments" and the language to translate to is "complementary music with an other instrument". These kinds of sequence to sequence translation tasks are traditionally performed by an encoder-decoder transformer model, as in the original transformer paper [7], and the model implemented in this thesis mimics that type of a model architecture.

This project has some similarities to previous work. The Music Transformer

[8] uses a decoder-only transformer to generate piano music from scratch and an encoder-decoder version to condition the music with a melody. Both the LakhNES [9] and the Multi-Track Music Machine [10] generate music that is conditioned by whole instrument tracks, but the model architectures are decoder-only architectures. Additionally, there are a number of other studies that use either an encoder-decoder or decoder-only transformer models with varying objectives [1]. However, none are an exact match to the setting proposed in this thesis.

To evaluate the performance of the seq2seq-transformer implemented in this thesis, a dataset that consists of songs in MIDI-format is prepared into a form that the transformer model can process. The transformer is trained on the prepared dataset, whereafter its music generating capabilities are evaluated.

2 Music generation and machine learning

Machine learning methods have been used for music generation in multiple domains, ranging from event-based discrete music generation to working with raw waveform data. Lately, deep learning methods have become increasingly popular, due to accessibility of the required resources. [1] This thesis focuses on music generation in MIDI domain, using transformers. The most relevant research articles to this thesis to date are the Music Transformer [8], LakhNES [9] and the Multi-Track Music Machine [10] which all utilize transformer models to generate music in MIDI domain. The methods and resources that have been used in this thesis have taken inspiration, not solely, but largely from these three research papers.

The Music Transformer was trained on choral pieces as well as a large piano dataset to successfully produce human-like novel continuations to snippets of music similar to the dataset. Whereas the focus of generation in this thesis is a conditioned setting using an encoder-decoder model in which the generation is wholly conditioned on an existing music track, the Music Transformer was mainly used as a decoder-only model that was only primed with the beginning snippet in order to generate the continuation. The researchers performed small experiments with conditioning on a melody as well by utilizing an encoder-decoder architecture, and found that their model performed better than the baseline models of the time. The Music Transformer's novel contribution is that it implements an efficient calculation for an attention mechanism which includes additional relative positional data in the attention, described further in section 3.2. [8]

LakhNES is a transformer model that was pre-trained on the Lakh MIDI dataset [11] and fine-tuned on the Nintendo Entertainment System Music Database (NES-MDB) [12], which consists of 46 hours worth of 8-bit chiptunes. The researchers utilize the massive Lakh dataset to improve the quality of the model as opposed to training just on the smaller chiptune dataset. The model can produce human-like chiptunes, both in an unconditioned settings as well as settings, where the generation is conditioned with a rhythm. [9]

The Multi-Track Music Machine focuses specifically on conditioned user-controlled generation. The model can generate small segments of complementary instrument tracks with adjustable parameters, such as the preferred instrument and note-density. While the objective of the paper is very similar to that of this thesis, the researchers use a non-time-ordered sequence representation for the music and again utilize a decoder-only model, whereas the model implemented in this thesis is an encoder-decoder model and the representation for the music is time-ordered.

The following sections give the background to understand the model that was implemented in the thesis, as well as the results that the model generated.

2.1 Machine learning

Machine learning is a broad umbrella term for many different methods that share a similar strategy for problem solving. Methods of machine learning range from simple linear regression to more complex deep learning methods. In its core, machine learning consists of three components: a model, data and a loss function. The model

is trained by fitting it to the training data by minimizing some loss function. When tackling a problem with machine learning methods, the goal is to implement and train a model, which can predict useful output even when encountering unseen data-points. [13]

Machine learning can be divided into three main paradigms: supervised learning, unsupervised learning and reinforcement learning. Supervised learning involves processing human-labeled data. The model encounters input-output pairs, where the preferred output is labeled beforehand and the model is fitted by minimizing the loss between its predictions and the preferred "true labels". Unsupervised learning involves unlabeled data and is used to find patterns in the data without any human intervention. Typical unsupervised learning methods are clustering and dimension reduction methods such as principal component analysis. Reinforcement learning involves an agent (the ML model) in a game-like environment making decisions and taking actions. The outcome of the actions are evaluated and good outcome is rewarded, reinforcing the behavior that led to that outcome. Through trial and error the model then converges to some optimal decision-making behavior in the context of its environment. [13]

In this thesis, the model that is implemented and trained is given labeled music data and therefore falls under the supervised learning category.

2.2 Deep learning

Deep learning is a subcategory of machine learning that involves artificial neural networks (ANNs or NNs for short). Neural networks consist of a set of connected nodes, that each implement a small function as a part of the larger system. Deep neural networks typically have a very large amount of nodes with many layers of connections, allowing the collection of these small functions as a whole to succeed in nearly arbitrarily complex tasks.

While potentially highly effective, deep learning models are very black-box in nature, which leads to a number of disadvantages. Finding the best parameters for training and using deep learning models needs to be done by trial and error, which could require a lot of effort and computing power. Additionally, black-box models are not a good choice for high-risk settings or settings that require accountability and explainability. Combined with the sudden increase in popularity for deep learning models and especially transformers, there has been a surge of research on transformer explainability, the most popular methods being based on transformer's attention mechanism. [14]

2.2.1 Neural networks

A neural network can either be a feedforward neural network (FNN), in which information can only flow from input to the output, or a recurrent neural network (RNN), which include feedback loops allowing information to flow backwards as well.

Figure 1 shows an illustration of a feedforward neural network. In a simple, fully-connected FNN the nodes are arranged in layers. The input layer is followed by

a number of hidden layers, which are followed by an output layer. Starting with the input layer, the input gets multiplied by weights, each weight being unique to each connection between two nodes. The nodes in the hidden layer receive input from all the nodes in the previous layer, apply a simple activation function to the value, then feed the output to all the nodes in the next layer, again multiplying the value with the weights. This process is repeated until the output at the output layer is received.

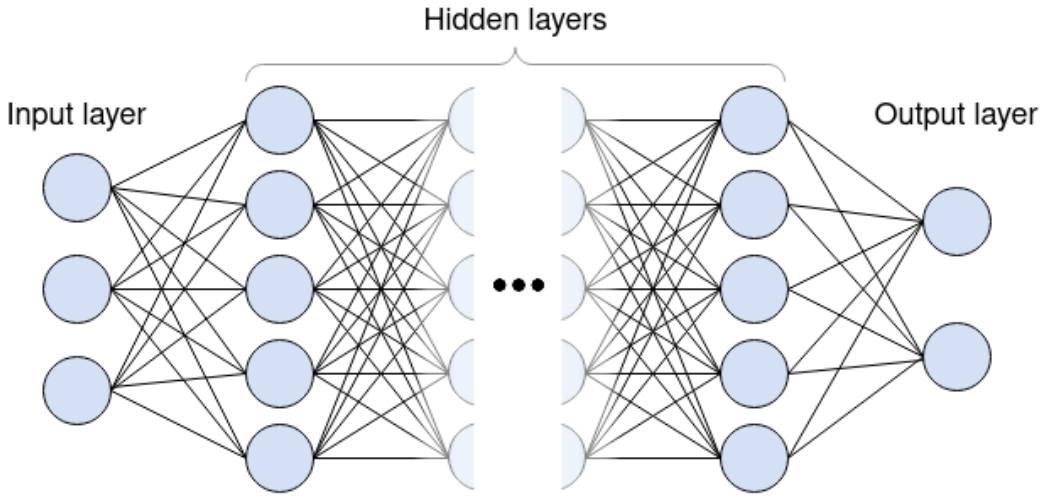


Figure 1: Fully connected neural network.

A typical RNN works similarly to the FNN, but unlike in an FNN, some of the inputs of the model are derived from past outputs of some of the RNN's layers. That is, an RNN receives two types of inputs: the present input and the recent past input. As a result, RNNs are sequential in nature, as the network is run multiple times to receive the final output. The normal use-case for RNNs is to process sequential data, since the output of the network can be dependent on previous elements of a sequence.

2.2.2 Activation functions

Activation functions are simple functions that are applied to values after each node. They are required in a neural network so that non-linearity is included in the model. Without the use of activation functions, a neural network's output signal would be just a polynomial with degree one, and would be unable to learn any non-linear mappings. Conversely, using activation functions makes it possible to learn arbitrarily complex functions. In order to apply gradient descent to update the model's weights, activation functions need to be differentiable. Many different types of activation functions exist, but the most commonly used one is ReLU, which stands for rectified linear unit. Figure 2 shows the ReLU function. ReLU, which can be defined as $\mathrm{ReLU}(x) = \max(0, x)$, is differentiable everywhere except at the single point $x = 0$. Although in theory it should be extremely rare for the input of ReLU to be zero, in practice it tends to occur frequently in a large deep learning model due to limited numerical-precision. The

most prevalent deep learning libraries such as PyTorch and TensorFlow calculate the derivative as

$$\text{ReLU}'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0, \end{cases} \tag{1}$$

effectively disregarding the issue. However, their choice of $\text{ReLU}'(0) = 0$ has been shown to be generally the most robust option. [15] [16]

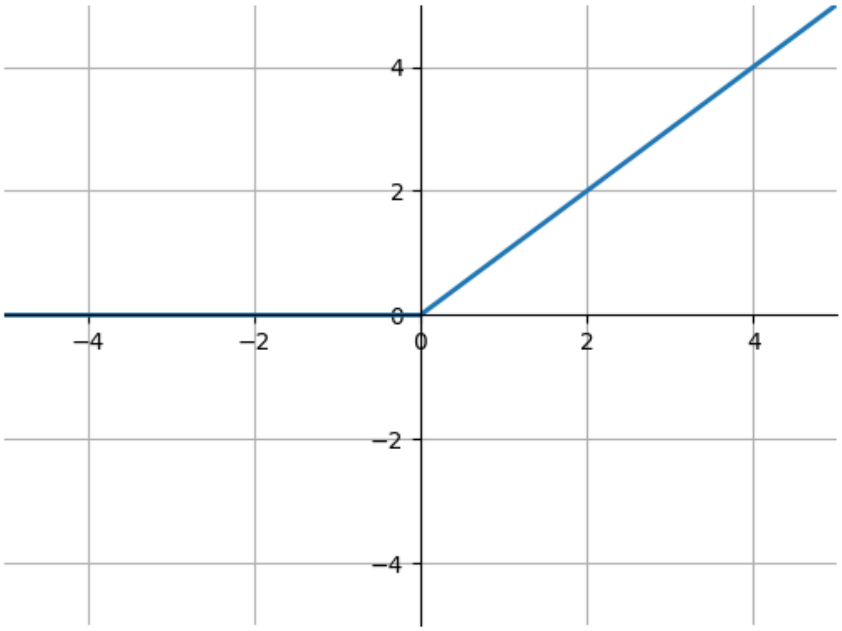


Figure 2: ReLU activation function.

2.2.3 Loss functions

Training a deep learning model includes making adjustments to the model's weights such that the output of some loss function gets minimized. A plethora of different loss functions are used in the field, the choice of the function depending on the model and the task. For a multi-class classification task such as the one in this thesis, the most common choice is the cross-entropy loss

$$\text{CrossEntropy}(y, \hat{y}) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{C-1} y_{i,j} \log(\hat{y}_{i,j}), \tag{2}$$

where y are the true labels (one-hot-encoded vectors, in practice), $\hat{y}$ are the predicted probabilities of labels, N is the amount of samples and C is the number of different

classes. Essentially, the cross-entropy is the averaged negative sum of those log-probabilities that correspond to the true labels. [17]

In this thesis, the implemented deep learning model is a version of a transformer model. The raw output of a transformer model $Z = (z_0 ... z_{N-1})$ is a sequence of logit vectors with values that act as likelihoods for the different classes. Before calculating the loss, the vectors are mapped into probability distributions with the softmax function

$$\hat{y}_{i,j} = \text{Softmax}(z_i)_j = \frac{e^{z_j}}{\sum_{k=0}^{C-1} e^{z_k}}, \quad \forall i \in (0, ..., N-1), \forall j \in (0, ..., C-1), \tag{3}$$

such that the predicted probability distribution's $\hat{y}_i$ elements become bound to [0 1] and sum up to one for all $i \in (0, ..., N-1)$.

2.2.4 Gradient descent and backpropagation

Teaching a deep learning model is done by gradually minimizing the perceived loss between the true labels and the predicted outputs of samples by updating the weights of the model as training samples are encountered. The weights are updated via some form of gradient descent and the backpropagation algorithm [18], which is an efficient implementation to calculate gradients of the weights of the neural network model with relation to the loss function.

Let L be the number of layers in the network. Let $x \in R^n$ be the input and $y \in R^m$ be the expected output of a training sample, also known as the true labels. Let W be the set of weights and $W^{[l]}$ be the weights between layers $l-1$ and l. Let $g(W, x) \in R^m$ be the predicted output and $C(y, g(W, x))$ be the cost, or the loss function, between the expected and predicted outputs. Let $f^{[l]}$ be the activation functions at layer l and $a^{[l]}$, be the activation function outputs at layer l.

The backpropagation algorithm starts with a forward-pass, where the input x traverses through the network and produces the prediction

$$g(W, x) = f^{[L]}(W^{[L]}(f^{[L-1]}(W^{[L-1]} ... f^{[1]}(W^{[1]}x))), \tag{4}$$

which is used to calculate the loss $C(y, g(W, x))$ against the true labels y, such as in equation (2) where $y = y$ and $\hat{y} = g(W, x)$. The gradient of the loss is then calculated with respect to the weights. For a network of fully-connected layers, the gradient is calculated with the following equations

$$\frac{\partial C}{\partial z^{[l-1]}} = (W^{[l]})^\top \frac{\partial C}{\partial z^{[l]}} f'^{[l-1]}(z^{[l-1]}) \tag{5}$$

$$\frac{\partial C}{\partial W^{[l]}} = \frac{\partial C}{\partial z^{[l]}} (a^{[l-1]})^\top, \tag{6}$$

where $z^{[l]}$ is the weighted input of layer l, which is the output of the previous layer multiplied by the weights between the previous layer and the current layer. $\partial C / \partial z^{[l]}$ is commonly denoted as the error of layer l. Calculating the gradient for the loss function starts with a straight-forward calculation of $\partial C / \partial z^{[L]}$ starting from the final

layer. Then, equation (5) is used to compute errors for all the layers in the network, moving from the last layer towards the first layer. Finally, equation (6) is used to calculate the partial derivatives for the weights, ultimately gaining the whole gradient of the loss-function with respect to the weights of the model. All the values (excluding the errors of the layers) required for these calculations can be cached during the forward-pass. [18][19]

In practice, backpropagation is parallellized for mini-batches of data. A mini-batch, which is a set of samples, is run through the model in parallel, calculating the gradients in the way described above. The gradients are then averaged over the samples in the batch before updating the models' weights.

Once the gradient has been calculated, the weights are updated. The simplest method is stochastic gradient descent (SGD), which updates the model weights, such that

$$W_{t+1} = W_t - \alpha \frac{\partial C(y, g(W_t, x))}{\partial W_t}, \tag{7}$$

where W_{t+1} are the weights at time-step $t+1$, W_t are the weights at time-step t, α is the learning rate, and $g(W_t, x)$ is the forward-pass function with weights W_t. Usually, more sophisticated methods are used for updating the weights than pure SGD. The most common optimization algorithm for deep learning models is the Adam optimizer, which is an extended version of SGD that utilizes momentum and root mean square propagation. Pseudocode for the Adam optimizer algorithm is provided in 1, where m_0 is the first moment vector, v_0 is the second moment vector, g_t are the weights' gradients w.r.t. the loss function, β_1 and β_2 are decay rates for the moment estimates, $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected moment estimates and ϵ is some small constant value. [20]

Algorithm 1 Adam optimizer algorithm [20]

$m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
while Ending condition has not been reached **do**
  $t \leftarrow t + 1$
  # Compute gradients
  $g_t \leftarrow \partial C / \partial W_{t-1}$
  # Compute the first and second moment vectors
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  # Compute the bias-corrected moment vectors
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
  # Update the weights
  $W_t \leftarrow W_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
end while

2.3 Transformers

Transformers are RNN-like models that replace the usual recurrence structures with a positional encoding and an attention mechanism. The RNN-models' recurrence structure forces the training to be done sequentially per element in a sequence, whereas the attention mechanism together with positional encoding allows transformer models to process a whole sequence parallel during training [7]. As a result, transformers are extremely efficient to train and are a very enticing option for tasks involving sequences. Figure 3 shows the sequence to sequence transformer model from the original transformer paper [7]. The model consists of embedding layers for the model inputs, positional encoding and N encoder layers and decoder layers, which themselves implement a certain structure of attention layers, normalization layers and feedforward layers with residual connections. The parts that make up the transformer are explained in the following sections.

2.3.1 Embedding

Transformers work with sequence inputs, where the elements of the sequence are called *tokens*, which are comparable to classes in a traditional multi-class classification problem. In the world of NLP, tokens represent words, parts of words or letters and the set of all the possible tokens is called the vocabulary. Each token in the vocabulary has an id, and the input of the transformer is usually a sequence of token id:s. The transformer's subsequent layers cannot work with just a sequence of discrete integer token id:s, so the first layer of the transformer maps the token id:s into learnable *embeddings*, which are continuous vectors. Using a sequence of continuous vectors instead of discrete values in the model has plenty of advantages. While the numbers in the vectors do not necessarily have any human-readable meaning, it is possible to calculate similarity-scores for different embeddings, for example by calculating the dot product between two embedding vectors. Deep learning models utilize these kinds of mechanisms to find dependencies and similarities between tokens from a dataset of token-sequences [21]. The length of the embedding vectors is often denoted as model dimension, or model size, and is the output dimension for each sub-model in the transformer.

2.3.2 Positional encoding

Since transformers do not exploit a recurrent structure in training, the inherent positional information that RNNs have is lost. The positional information is important for predicting elements in a sequence, so it is necessary to be implemented in the model in some manner. Positional encoding solves this issue for transformers.

Positional encoding can be applied in multitude of methods, including both fixed and learned methods. The most common fixed method is the one implemented in the original transformer paper [7]. Each element in a sequence is mapped with a unique vector, which forms a positional embedding matrix. The vector is generated with sine

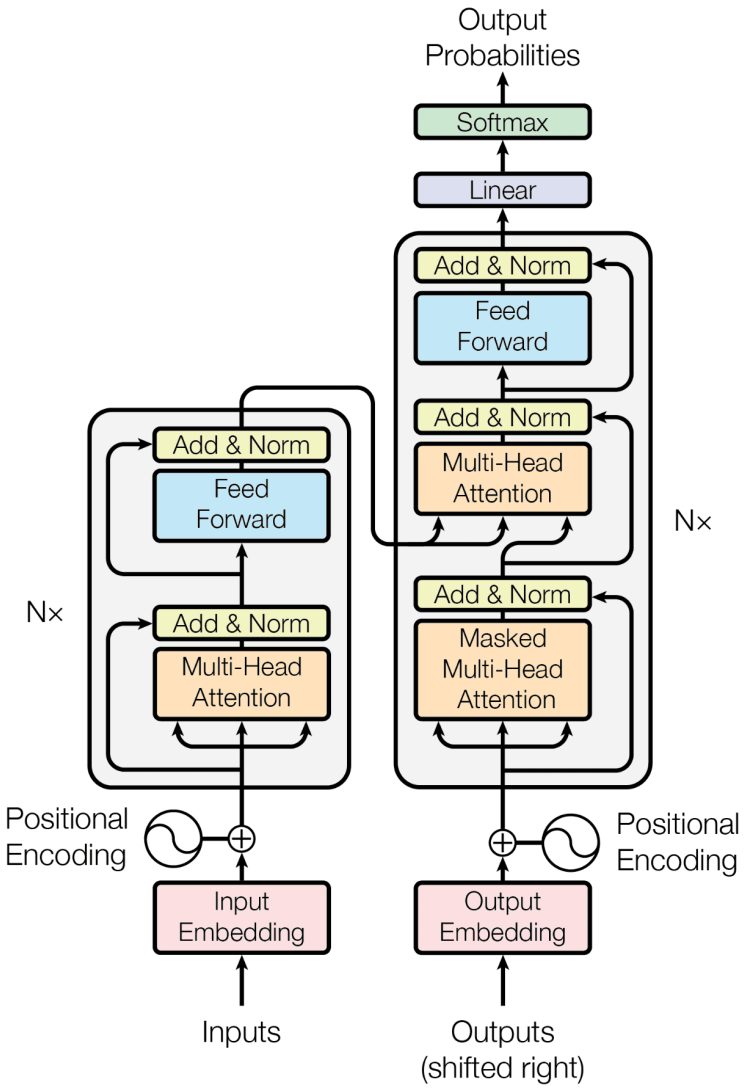


Figure 3: Encoder-decoder transformer model from the original transformer paper Attention Is All You Need [7].

and cosine functions of different frequencies

$$PE(pos, 2i) = \sin(pos/n^{(2i/d)}) \tag{8}$$

$$PE(pos, 2i+1) = \cos(pos/n^{(2i/d)}), \tag{9}$$

where pos is the position of the element in a sequence, i is the dimension of the R^d dimensional embedding vector and n is an arbitrary large number. This positional embedding is added to the word embedding to create the input of a transformer model. The authors of the paper hypothesized that this implementation would allow the model to learn to attend to relative positions easily, since the positional encoding vectors can be represented as linear transformations of each other, for any fixed positional offset. [7][22]

Using a fixed method instead of a learned method has potential perks. One of the risks of a learned method is that learning the representations for positional relations is reliant on data, which means that very long distance relations could be rare enough in the data that the model fails to learn them. However, the tests made for the original paper yielded similar results for both fixed and learned methods, and the authors decided to use the fixed method for its simplicity. [7]

2.3.3 Attention mechanism

Transformers rely on an attention mechanism to learn connections between the elements of a sequence. The attention mechanism calculates attention scores between elements in two sequences, which can be interpreted as a metric for the strength of relationship between the two elements. Figure 4 illustrates an example of what a heatmap of attention scores could look like in a translation task on a trained model. [23]

The attention mechanism implemented in the original transformer paper is the scaled dot-product attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V, \tag{10}$$

where Q is the query matrix, K is the key matrix V and is the value matrix d_k is a scaling factor equal to the dimension of keys. Attention in transformer models are conventionally applied as multi-head attention, where the attention is split into h different heads, each applying different learned linear projections to queries, keys and values. The different heads produce output, that is concatenated before projected for a final time to yield the final output for the multi-head attention. Thus,

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_\text{h})W^O \tag{11}$$

$$\text{head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \tag{12}$$

where W^O is a projection matrix for the concatenated output, and W_i^Q, W_i^K and W_i^V are projection matrices of the i:th head for the input query, key and value respectively. Figure 5 shows the the scaled dot-product attention and multi-head attention layers. [7]

In an encoder-decoder transformer that is trained for a causal sequence generation task, in which the last member of the target sequence is dependent on all of the previous members, the attention mechanism is used in three different contexts. The encoder contains self-attention layers, where Q, K and V all come from the previous layer's output. Self-attention captures the inner connections between the tokens in a single sequence. The decoder contains self-attention layers similar to the encoder, but the values need to be masked so that information from future values cannot flow to the past values. This is done by setting illegal connections' values to $-\infty$ inside the softmax function. The decoder also contains attention layers, where the queries come from the previous layer, but the keys and values come from the encoder's output, which would produce attention scores similar to what is illustrated on Figure 4. [7]

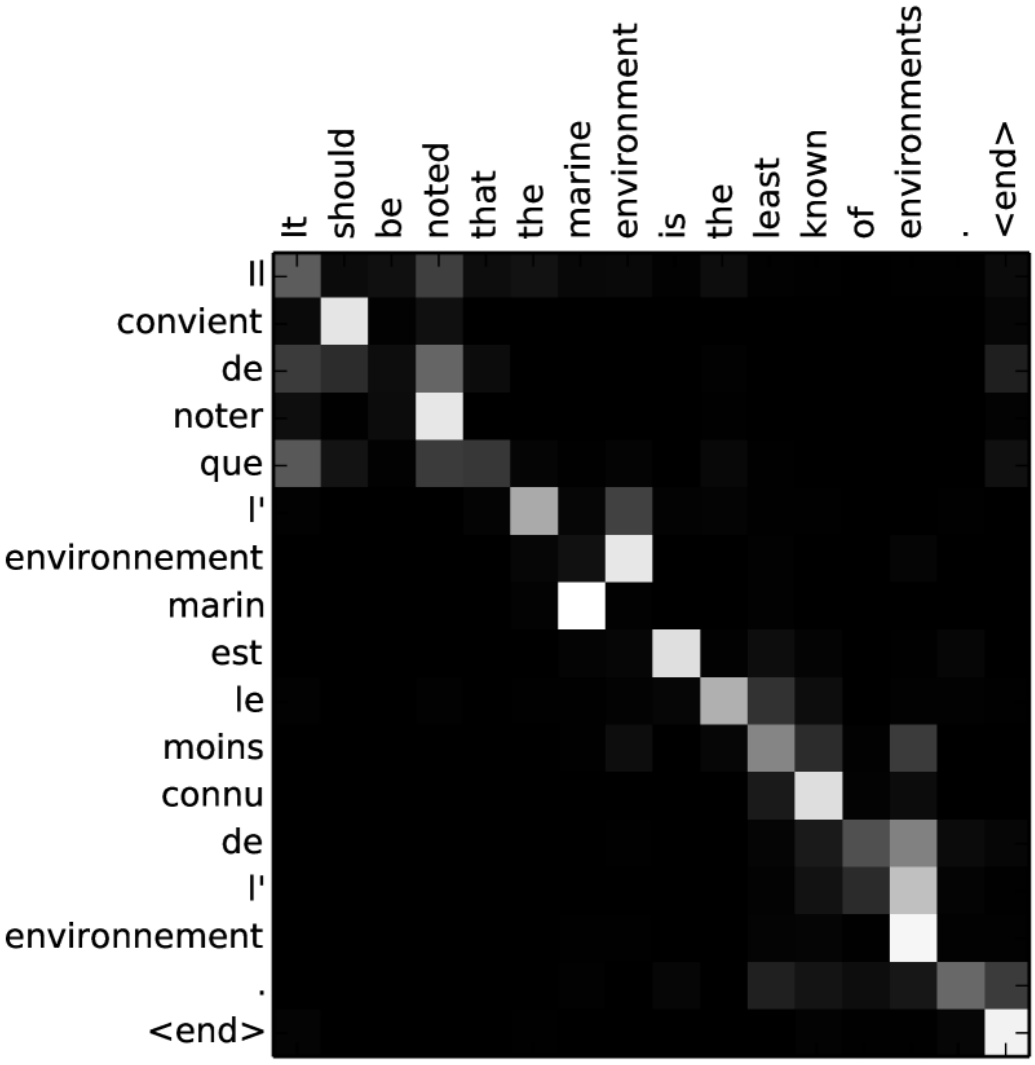


Figure 4: An example of an attention score heatmap. A lighter color shows a stronger connection between the words. [23]

2.3.4 Normalization layers, feedforward layers and residual connections

After each attention layer or feedforward layer, there is a normalization layer. The layer implements the function

$$y = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}}\gamma + \beta, \tag{13}$$

where x is the layer input, y is the layer output, γ is a learnable scaling parameter, β is a learnable shifting parameter and ϵ is a small constant for numerical stability, in case the input's variance becomes zero by chance. Essentially, the normalization layers force the input to have zero mean and unit variance, whereafter the input is scaled and shifted. This combination of a projection and a scaling allows the attention

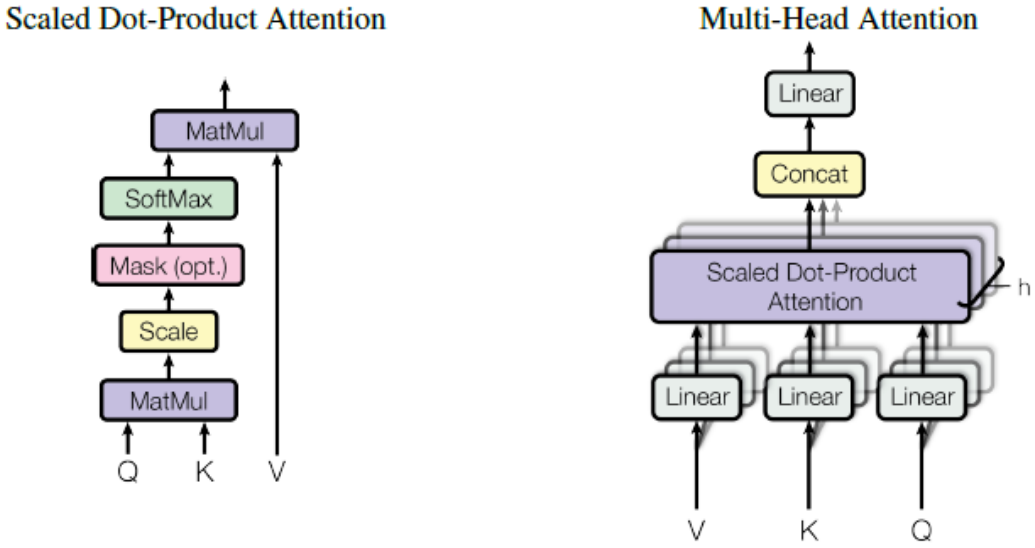


Figure 5: The scaled dot-product attention and multi-head attention.

mechanism to attend to each key equally, and prevents the existence of "unselectable keys". [24] [25]

The feedforward layers in the transformer models are simply fully connected layers that consist of two hidden layers with a ReLU activation function in between.

The model also utilizes residual connections with each normalization layer. The input of the normalization layers are not just the previous layer's outputs. Instead, outputs of one of the model's even earlier layers skip the layer that comes after and are added to the input of the normalization layer. This means that the layers that the skip-connection affects are attempting to learn the residual between the input and the output, instead of a whole transformation from input to output. These types of residual connections have been shown to increase accuracy of very deep neural network models. They also alleviate numerical instability problems that can often arise in very deep models such as vanishing and shattered gradients. [26] [27]

2.3.5 Tokenization

As previously stated in section 2.3.1, the inputs of a transformer models are sequences of tokens. The information that a single token represents can be chosen freely, but the choice has a large effect on the model's performance. If the tokens correspond to different words, the total vocabulary size and consequently the model size becomes huge, increasing resource requirements and slowing down training and inference. If the tokens correspond to single letters of the alphabet, it is difficult for the model to learn meaningful input representations, as a single letter does not have much inherent meaning like a single word would. One solution is to use subword tokenization, which is a middle-ground of the earlier mentioned approaches. [28][29]

Byte pair encoding (BPE) is a simple but effective subword tokenization method. Given a base-vocabulary, e.g. the letters of the alphabet, the words in the dataset

are first split into the base-vocabulary elements. The tokenizer trains on a dataset by calculating the frequency of different pairs of vocabulary members in the dataset. The most frequent pair is added to the vocabulary, along with the merge-rule to modify the dataset so that the previously split pairs are merged. This process is repeated until a chosen vocabulary size is reached. When encoding text with a trained BPE-tokenizer, the text is split into the base-vocabulary, after which all the learned merge rules are applied to get the final tokenization. [28][29]

Defining all of the symbols the BPE could encounter as a base-vocabulary to avoid ever producing unknown-tokens would potentially grow the vocabulary size to massive proportions. Byte-level BPE or BBPE solves this problem and is one of the most language-agnostic subword tokenizers. It uses bytes as base-vocabulary, which forces the base vocabulary to be of size 256, while never encountering an unknown token. [30]

Using BPE or similar general tokenization methods have been shown to to enhance the performance of music generation regardless of the dataset or model used. [31] [32]

2.4 Music

Music is usually written as compositions, which are sets of instructions about how to perform the piece. A composition includes concepts such as melody, harmony, rhythm and timbre and these concepts are formed by the arrangement and the properties of the set of notes in the composition [33].

This thesis focuses on a relatively small subset of concepts that can be attributed to music. The MIDI format, which the model uses as the medium for music generation, implements a western music system that by default excludes musical concepts that are foreign to western music, such as microtonality. Since the model's objective is band instrument generation, the main genre in the dataset is western popular music, as the tracks that do not contain the target instrument are filtered out. Concepts such as timbre and agogic expression are also excluded. Instead, the thesis is focused on all the musical principles that can be derived from just specifying the notes that are played and the points in time in which the notes are played relative to each other. These include the aforementioned melody, harmony, rhythm and more. The following sections explain the necessary background in music and the MIDI-format to understand the intentions of this thesis and the analyses of the results.

2.4.1 Elements of music theory

Although in principle music is an unrestrained art-form, there are plenty of different sets of rules for music. Usually, the rules are loosely defined by the genre of the music in question and are affected by a mixture of cultural and universal interpretations of what sounds stimulating in some way. The rules can either be a set of written guidelines or simply some commonly accepted practices. Oftentimes a composition intentionally breaks these rules, which necessarily does not make poor quality music. In fact deviating from some of the commonly accepted practises is often the key to create something interesting and new, in some cases creating whole new trends of

music in the process. This is one of the reasons that makes interpreting a generated piece of music as "good" or "bad" ambiguous and subjective. However, it is still possible to objectively analyze whether a piece of music follows a certain rule or not, so it is possible to evaluate whether the model manages to learn some of the rules or not.

In the western music system, there are seven named pitches, listed in an ascending order: *C*, *D*, *E*, *F*, *G*, *A* and *H* by the Finnish, Scandinavian and Germanic naming convention. After *H*, the next pitch in sequence would be another *C* again that is one *octave* higher than the earlier *C*. Notes that are an octave apart sound similar which is why they have the same name. Additionally, the pitches can be either raised or lowered *chromatically* by a *semitone*, which is the smallest standard unit for difference in pitch recognized in western music systems. Generally two named pitches are a *whole tone* apart which is equal to two semitones. This means that a raised note is the same as the next note in the sequence lowered, for example a raised *G* (also known as *G*# or *G* sharp) is the same as a lowered *A* (also known as *A*b or *A* flat). In music theory lingo the two notations are then *enharmonically equivalent*. There are two pairs of named pitches that are only one semitone apart, *E* & *F* and *H* & *C*. This means that for example a raised *E* is enharmonically equivalent to *F* and a lowered *F* is enharmonically equivalent to *E*. All together this divides an octave into a total of twelve different unique pitches. [34] Figure 6 shows an illustration of piano keys that are labeled with the pitch names to provide a tangible example for the explanation above.

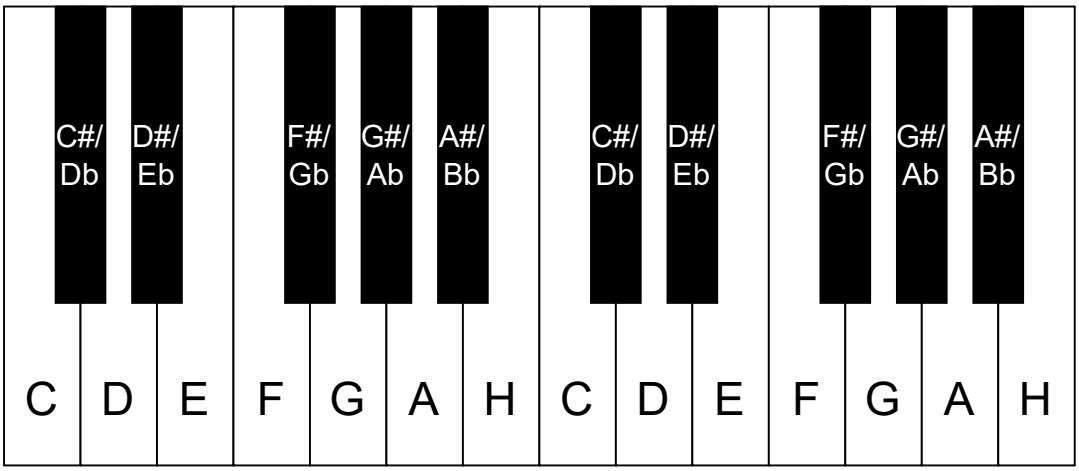


Figure 6: Piano keys labeled with the pitch names.

Sections in music are conventionally based on a certain tonal *key* that defines the tonal context for a section, that is, what different notes' and combinations of notes' functions in the section are. This also means that most of the notes in the section tend to be from a set of notes that is called the *scale* associated with the key. There is a major and a minor key associated with each unique pitch, making a total of twelve different major and minor keys and scales, which form the most common and basic scales called *diatonic scales*. An example of the major *C* scale would be *C, D, E, F,*

G, A, H, the notes in the scale having *degrees* based on their position in the scale. Notes that are outside the scale in a section of music with a clearly definable key and scale are called altered notes. The melodies and chords that make the harmony in the section can lie either inside or outside the key, generally depending on whether they contain altered notes or not. [34]

Whereas harmony is a broader and more abstract concept, a chord or an interval is a realized element of harmony. An interval is defined as two notes sounding at the same time and a chord is defined as more than two notes sounding at the same time. The most commonly used chords in western practices are *triads*, which are a specific pattern of three notes. Specifically, the harmony of popular music tends to consist mostly of triads. Each diatonic scale has seven different triads, one for each degree in the scale. These triads that are tied to a diatonic scale are formed in a specific pattern, by starting from some degree and picking the first, third and fifth degree notes relative to that degree. For example in the key of *C* major, the first degree chord is *C, E, G* and the fourth degree chord is *F, A, C*. [34] Chords tend to have certain progressions in music, where the chords move from one to another as the music flows forwards, some progressions being more common than others. In popular western music, it is common to repeat a certain chord progression until the end of one section in the music, then begin repeating another progression in the next one.

Rhythm is the timing of musical elements in relation to each other. A musical piece is usually divided into *bars*. Notes and the absence of notes called *rests* can have different time values. A *beat* is the "pulse" of the music. A section in a piece has a defined *time signature*, which consists of two numbers, the lower number defining the time value of one beat, and the upper number declaring how many of those beats a bar can have. [34] A typical time signature would be for example 4/4, which means that there are four beats with time value one quarter in a bar. One quarter beat can contain either one quarter note, two eighth notes or four sixteenth notes and so on. A composition then consists of bars with defined beats, that have notes with defined time values.

2.4.2 Sheet music

Some results of this thesis will be shown in a format that is familiar to musicians: sheet music. Figure 7 shows an annotated example of sheet music. Sheet music is read similarly to how text is read, left to right, up to down. Each "row" of written sheet music is called a system. At the beginning of each system, there are clefs, which give context for the pitches of the notes. At the beginning of a piece, or if there is a change to it in the middle of the piece, a time signature is given. Different bars, also known as measures, are separated by barlines. The second system shows a change in key going from C-major into D-major, which is reflected by the key signature. Additionally, figure 7 shows various examples of different length notes, chords and rests, as well as examples of *accidentals*, which can be used to raise or lower the note by a semitone.

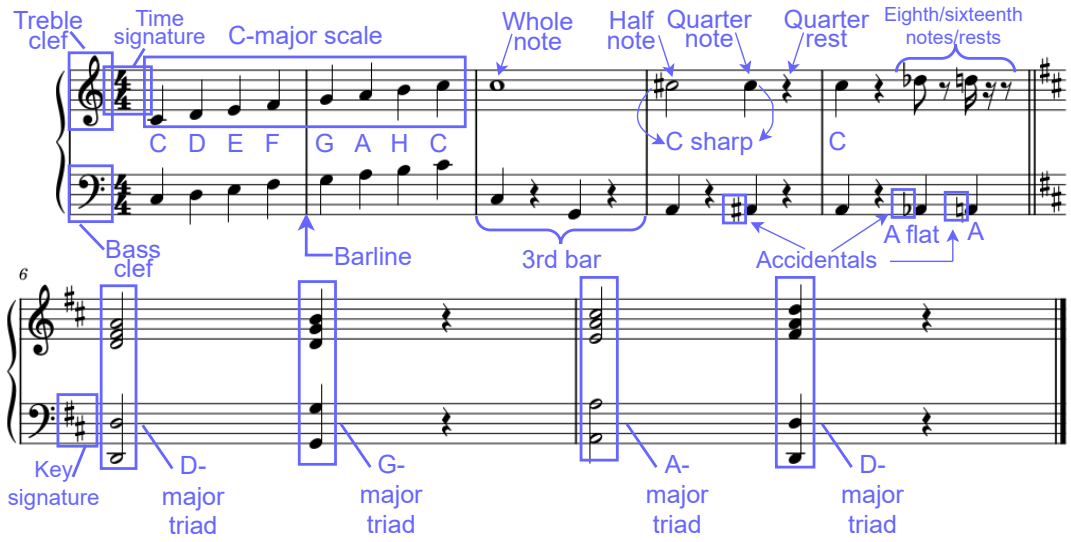


Figure 7: Annotated sheet music.

2.4.3 MIDI-format

Musical instrument digital interface, or MIDI for short, is a standard for storing and communicating information about music playback. A MIDI file stores data as an 8-bit stream, grouped in 32-bit chunks. The chunks are either header chunks or track chunks, header chunks giving information about the whole file, while track chunks define a sequential stream of events for a single track. [35]

The header chunk defines the file's MIDI format, the number of tracks and division, which is a measure for how much time each MIDI clock-tick takes, either in real-world units or abstract per-beat units. The MIDI format has three different options, one for a single multi-channel track, one for multiple separate tracks that play simultaneously and one for multiple separate tracks that play independently of each other. The division defined in the header chunk declares the meaning of time values in the events, either by ticks per quarter-note or a mapping to subdivisions of a second. [35]

Track chunks each declare a stream of events for a track. These include multiple types of meta-events such as defining track name or lyrics, MIDI-system-exclusive messages that might relay information to some specific MIDI-system units and most importantly MIDI-events, which contain timestamped events for things analogous to a musician playing a synthesizer. [35]

The MIDI-events send information to one of sixteen channels, which are parallel communication pathways between different MIDI-devices. In practice, a channel usually contains one instrument's output. Channel 10 is special, designated for percussion sounds and effects instead of sounds with a defined pitch. [35]

For this thesis, the information extracted from the MIDI-files include the division, note on and note off events as well as the program change event. This is enough to define the whole of "when" and "what" each instrument is playing.

3 Research material and methods

3.1 Resources and data

The dataset used in this thesis is the clean subset from Lakh MIDI-dataset [11]. The Lakh MIDI-dataset is a collection of 176581 unique midi-files, and the dataset used in this thesis is a subset of 17233 of those files, with the directory structure and filenames mapping each file to an artist and a song.

Figure 8 shows the pipeline for preparing data as well as training and inferencing the model. To work with the midi-files, csv is used as a medium, by utilizing a python library [36] that turns midi-files into csv and back. The midi-file samples are filtered by instrument so that the necessary instruments are found from every sample in the dataset. After that, the files are split into smaller chunked snippets, so that each snippet contains 96 beats. In other words, with a 4/4 time signature the snippets are 24 bars long. The split snippets are separated into pairs of a midi-file that contains the target instrument track and a source midi-file that contains the other instrument track. The samples are then tokenized into token ids with the MidiTok-tool [37]. (B)BPE-tokenization is trained on the tokens, then applied to the dataset to create a BPE-encoded dataset. These tokens can be used by the transformer model for training and inference, and the resulting tokens can be decoded back into midi with the previously trained tokenizers. The transformer models are implemented in the PyTorch deep learning framework [38].

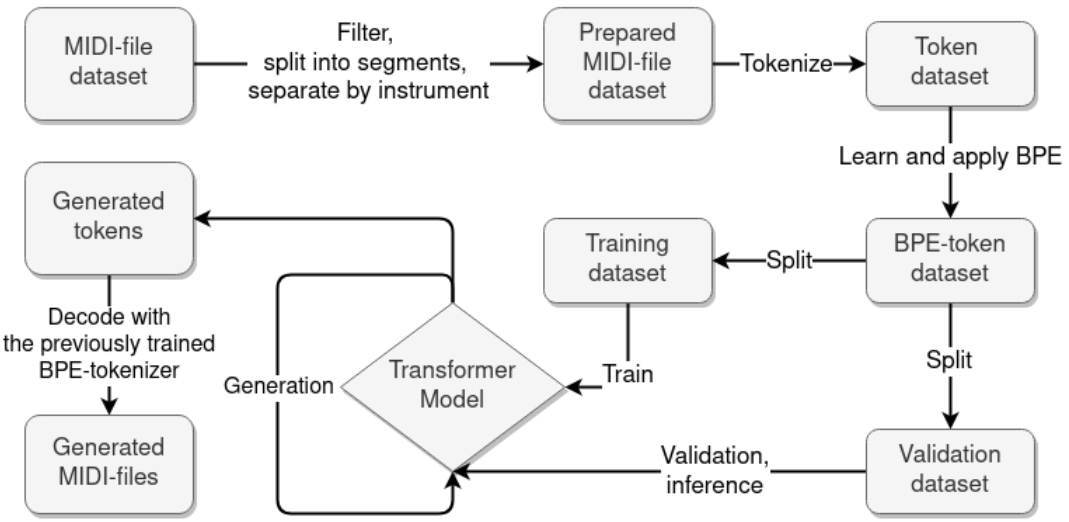


Figure 8: The complete pipeline from a dataset full of MIDI-files into generated MIDI-samples.

The MidiTok library implements multiple tokenization schemes, copied from a handful publications that have had success in music generation [37]. The tokenization scheme used in this project is REMI, which stands for Revamped MIDI [39]. This scheme was chosen, since it was originally made with popular music and beat-based generation in mind, and should suit the used dataset and the objective of the thesis well.

Figure 9 shows an example of REMI-tokenization. The MIDI events are grouped into bars, which is discretized into some number of possible positions. The information of the notes is arranged into different tokens, such that a position token is followed by the pitch token, which is followed by a velocity token, which is followed by a duration token. The token sequence goes as follows: a bar token is followed by a number of the note-related tokens until all the notes in the bar are fully defined by the tokens, then another bar token is inserted. Additional tokens such as tempo-tokens, rest tokens, time signature tokens and program tokens would be possible to use as well, but they are not included in the tokenization in this project. In order to use BPE tokenization on these tokens, each token has some byte-representation mapped to them. [37][39] The tokenization parameters are shown in Table 1.

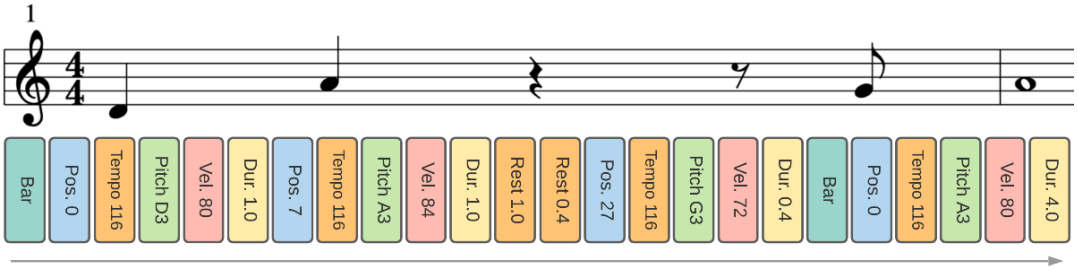


Figure 9: An example of REMI-tokenization [37][39].

3.2 Model

The transformer models used for the experiments are modified versions of the Music Transformer [8]. The Music Transformer is originally a decoder-only transformer that utilizes relative global attention in its attention layers. For the purposes of this thesis it has been modified into a seq2seq encoder-decoder transformer, similar to the one in Figure 3. Relative global attention includes additional embedding matrix E^r in each attention head. E^r contains an embedding for each pairwise distance $r = j_k - i_q$ between query position i_q and key position j_k. These embeddings are multiplied by the queries, and the resulting matrix is rearranged so that we gain S^{rel}, in which each entry (i_q, j_k) is a dot product of the query at position i_q and the embedding for the distance $j_k - i_q$. This matrix is injected to the attention equation (10) such that the attention in the Music Transformer is calculated as

$$\text{RelGlobalAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top + S_{rel}}{\sqrt{d_k}}\right)V. \tag{14}$$

Adding this relative position information to the model on top of the absolute position information has been shown to decrease validation loss and increase the quality of generated samples in music generation models. [40][8] The model parameters are in Table 1.

3.3 Model training

The training inputs are the token ids for the reference instrument track, and the token ids for the target instrument track. Both of the token sequences are padded with a designated pad-token id value, so that the sequences match the sequence length of the model. In order to not pay attention to these padded values, the model is also provided with a padding mask matrix for both sequences each iteration, which tells the model which tokens should not be included in the calculations. For the target sequence, a causal mask is also provided, which prevents the model from "peeking into the future" as described before in section 2.3.3. The target sequence is also shifted to the right by one and prepended with a designated starting-token id.

The models were trained minimizing cross entropy loss, which is described in section 2.2.3. The Adam optimizer algorithm (1) was used to train the models, in which the gradients of the weights calculated in each iteration with the backpropagation algorithm described in section 2.2.4. Dropout was also used in the model before each normalization layer. Dropout is a technique, in which some of the connections between the layers are deactivated randomly during training. Usually, this produces better generalisation, as the model does not learn to rely on information flowing just from few specific pathways. During inference, dropout is deactivated and all the connections are always active. This makes the model deterministic during inference and stochastic when training.

Each 200 training steps, a validation loss is calculated. This was done by picking ten random mini-batches of samples from the validation dataset, calculating the loss for these batches, then dividing the resulting cumulative loss by the amount of mini-batches to make the validation loss comparable with training loss. Also, every 500 training steps, the model weights are saved to disk and 3 different samples are picked at random from the validation set which are used to produce a number of generated samples, in order to quickly see how the model performs at different stages of training.

The calculations presented in the thesis were performed using computer resources within the Aalto University School of Science “Science-IT” project.

3.4 Model inference

Model inference is done recurrently for transformers. The model takes in a target sequence, which is at minimum just a "start-token", then the next token is sampled from the output and concatenated to the target sequence. This is repeated until some token (or arrangement of tokens) is sampled that marks the end of the sequence, or until the maximum sequence length is reached. Inference was done in two different ways. One is inference by only conditioning the generation with the token sequence extracted from the source instruments in the encoder. The other one is inference by additionally conditioning the generation with a start of the target instruments token sequence, with 50 initial tokens for bass generation and 75 initial tokens for piano generation, giving slightly more tokens for piano generation since piano-samples are generally more note-dense per bar.

Since the model outputs logits, which are likelihood values for different possible token outputs from the vocabulary, there are choices for sampling the next token in the sequence. Common choices include top-k and top-p sampling, which have been shown to yield better results in NLP than pure sampling or maximum probability sampling [41]. Top-k sampling picks the top k likely samples while the top-p sampling picks samples until a cumulative probability p is reached. The probabilities are then normalized accordingly so that the total probability sums up to one.

Temperature is another hyperparameter that can be used in inference. In order to increase randomness in sampling, the output of the model are scaled so that the softmax function which connects the output into probabilities is given by

$$\text{Softmax}(x_i) = \frac{e^{x_i/\tau}}{\sum_{j=1}^{n} e^{x_j/\tau}}, \tag{15}$$

where x_i are the weighted likelihoods for each token, n is vocabulary size and τ is temperature. When generating samples in this thesis, only top-k sampling is utilized with $k = 3$ and $\tau = 1$.

Tokenization parameters (REMI)	
Pitch range	[21, 109]
N velocity bins	8
Position resolution	32
Use time signatures	False
Use rests	False
Use tempos	False
Use programs	False
BPE Vocabulary size	500
Model parameters	
Model dimension	256
Vocabulary size	500
Encoder layers	6
Decoder layers	6
N attention heads	4
Feedforward hidden dim.	128
Sequence length	1500
Training parameters	
Learning rate (α)	10^{-4}
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	10^{-8}
Dropout rate	0.1
Mini-batch size	16
Inference parameters	
Top k sampling	3
Top p sampling	-
Temperature (τ)	1
Init. tokens bass/piano	0 and 50/75

Table 1: Tokenization parameters, model parameters, training parameters and inference parameters for both the bass generating and piano generating models.

4 Results

4.1 Tokenization

Figure 10 shows the REMI tokenization token distribution before and after BPE encoding for the bass training dataset. The leftmost spike in the dataset is token id 4, which is the "bar" token. After that the next token ids 5 through 92 are pitch tokens, which are lopsided towards the lower end of pitches for the bass, as one would expect. The largest spike around token id 100 consists of the 8 velocity tokens with ids 93 to 100, and the duration tokens with ids 101 to 164. The final group of tokens is the set of 32 position tokens for token ids 165 through 196, where the downbeat (the first beat in a measure) position is the largest and the third beat position is the second largest spike.

The effects of BPE on the token distribution are clear. Every token with id 197 through 499 is a new token that is a result of a merge in the BPE algorithm. The total number of tokens is roughly half of the original dataset. Since it is more likely for a more common token to get merged into a new token, the spikes in the distribution are flattened significantly.

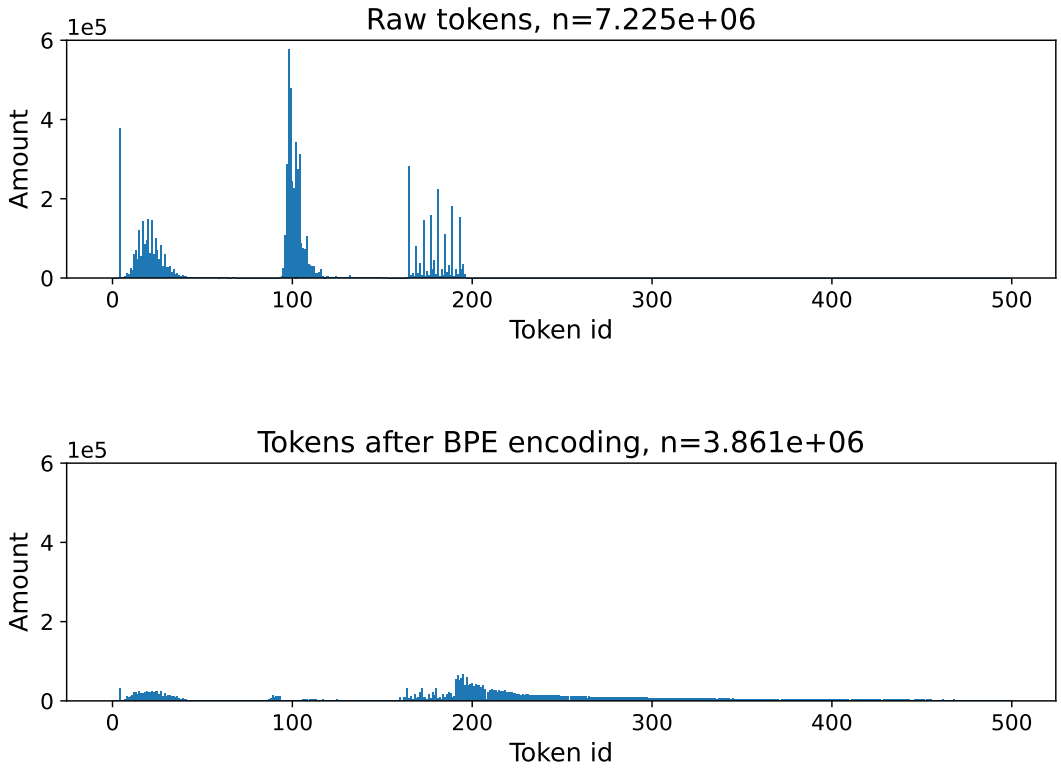


Figure 10: Token distributions for the bass dataset.

Figure 11 shows the token distribution for the piano dataset. The distribution of raw tokens is reasonably close to that of the bass, but with a few differences. In total, there are almost double the tokens, since the piano parts tend to be more note-dense than the bass parts in a track. The pitch tokens are shifted towards the mid-range of

pitches, and while the downbeat is still the leading position, the other positions are more even for the piano than for the bass.

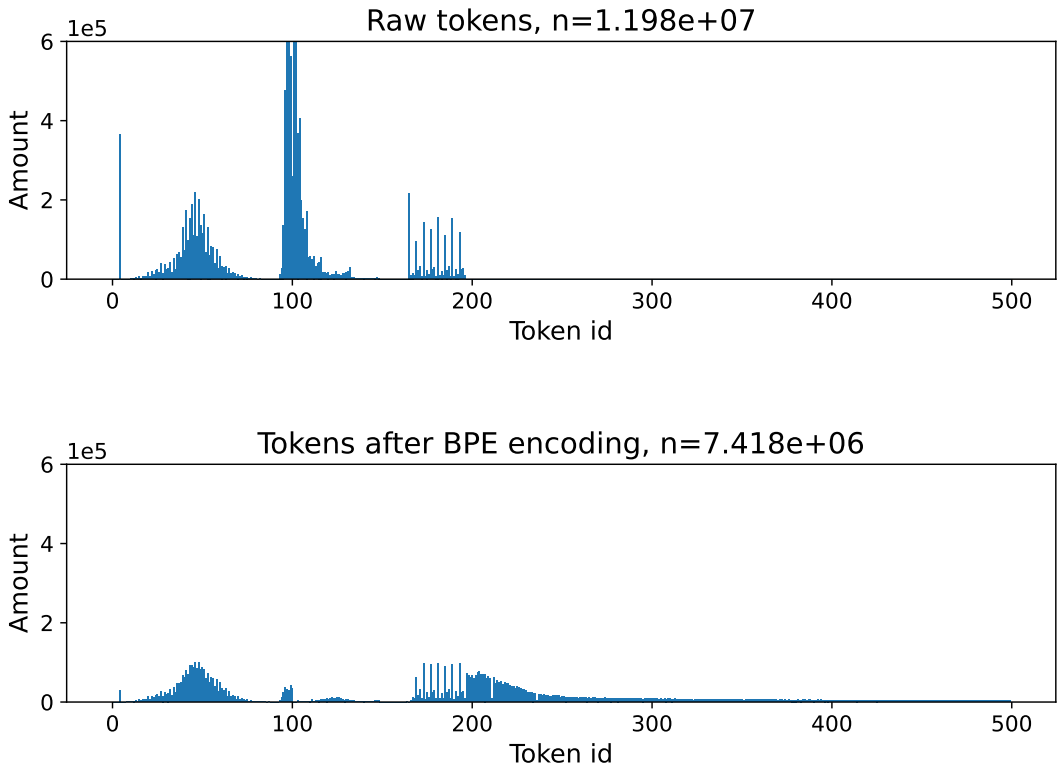


Figure 11: Token distributions for the piano dataset.

4.2 Training

Figures 12 and 13 show the training and validation losses for the bass generating and the piano generating models, as a function of number of training steps. The validation loss is nearly the same as the training loss throughout the training, although slightly higher, for both of the models. At 20 000 steps, the training of both models has converged.

4.3 Generation

For analysing the models' performance, both the piano generating and the bass generating models were instructed to generate 300 samples, using random samples from their respective validation datasets as reference. As a result, 299 piano samples and 294 bass samples were successfully generated, the remaining token sequences being faulty beyond repair so that they could not be subsequently converted into midi-files. Rhythm and tonality analysis was performed on the generated results.

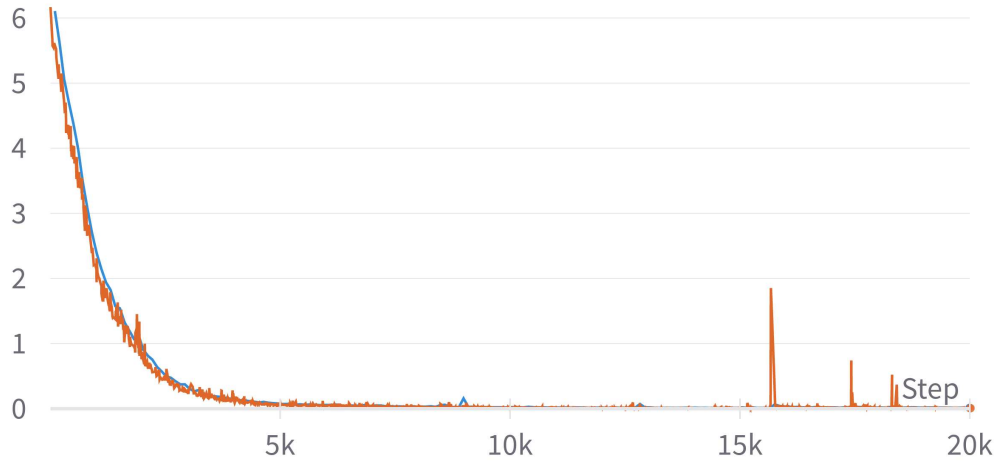


Figure 12: Validation and training losses for the bass generating model.

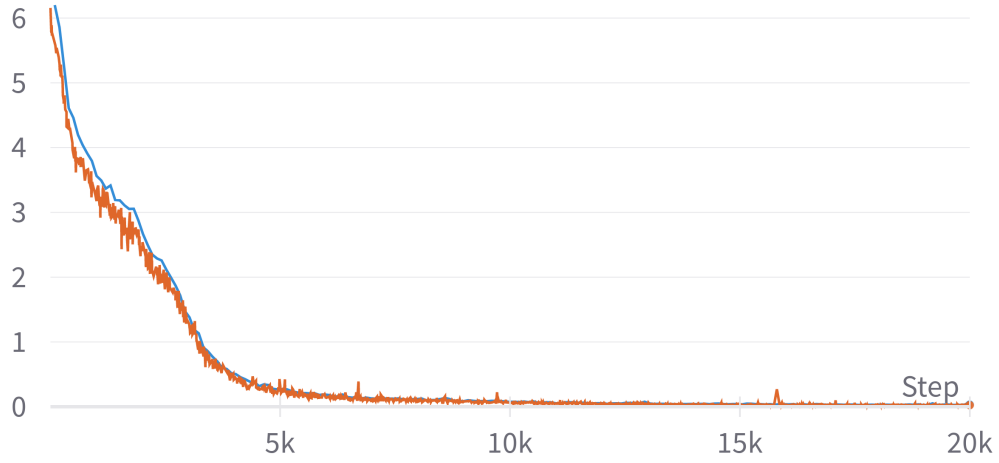


Figure 13: Validation and training losses for the piano generating model.

4.3.1 Rhythm analysis

For rhythm analysis, the token data was reduced into rhythm-sequences, that account only for the position-tokens in the datasets. The data was split into bars, and every bar was split into 32 positions similarly to the tokenization that was performed earlier. The bars were presented as 32 character strings, where 1 means that a tone starts from the position and 0 means that no tone starts from the position: for example a "10000000000000001000000000000000" would represent a rhythm where a tone starts at position 0 (the downbeat) and position 16 (the third beat) and nowhere else.

In addition to turning both the validation dataset and the generated dataset into the rhythm-sequences, a mock-dataset was created for both instrument scenarios. The mock-dataset reflects the generated dataset in that it has same density for tones per bar, but the distribution of starting positions is flat, so each position in the rhythm sequence has equal probability of having a 1. This acts as a baseline, expected behaviour of a

model that has learned nothing about where to place the tones in the bar.

Figure 14 shows the distribution of tone positions in the three bass datasets and Figure 15 shows the similar histograms for piano datasets. All the plots are normalized by the total amount of bars in the respective dataset. The bass model generates mostly tones that start on the downbeat and third beat in a bar. These are accurately the most common positions in the original dataset, but the original dataset does include many positions that the model almost entirely fails to generate. Additionally, the total amount of tones the model generates per bar is vastly smaller than that of the original dataset. The piano model behaves quite similarly. In the original dataset, the distribution is flatter, and this is somewhat reflected in the generated samples. Similarly to the bass variant, the piano model also creates less tones per bar compared to the original model. However, it does create more tones per bar compared to the bass model, which is expected of a working piano and bass model pair.

The generated rhythm sequences as well as the mock sequences were compared to the original sequences using Levenshtein distance as the metric, which is considered a decent approximation for human intuition for differences in two separate rhythm sequences [42]. A random pair of a generated/mock sequence and an original dataset rhythm sequence was picked 10 million times and the average Levenshtein distance was calculated from these pairs. The results are shown in table 2. The average distances do not hold too much inherent meaning and the distances between different instrument datasets cannot be compared directly, but for both instrument scenarios the generated dataset performed better than their mock counterpart. This means that the generated rhythms were on average more similar to the rhythms on dataset than pure noise, which suggests that the models have learned a non-zero amount about where to place the tones in a bar.

	Bass validation	Piano validation
Generated	4.104	4.103
Mock	4.973	4.800

Table 2: Average Levenshtein distances from 10 million randomly picked pairs of samples between validation dataset and the generated dataset or the mock dataset.

4.3.2 Tonality analysis

Analysing the models' understanding of tonality means analysing whether the generated samples reflect the harmonies that are present in the reference counterpart. The analysis was done in a broad manner, which only takes into consideration whether the overall keys of the generated samples match the reference sample.

To evaluate the key that a sample is in, the Krumhansl-Schmuckler key-finding algorithm was utilized. The algorithm is based on empirically constructed key-profiles for major and minor keys, in which a value is assigned for each of the twelve notes. Then the total duration of each note sounding in the sample is measured, and a line is fitted between the profile's assigned values for notes and the time that the

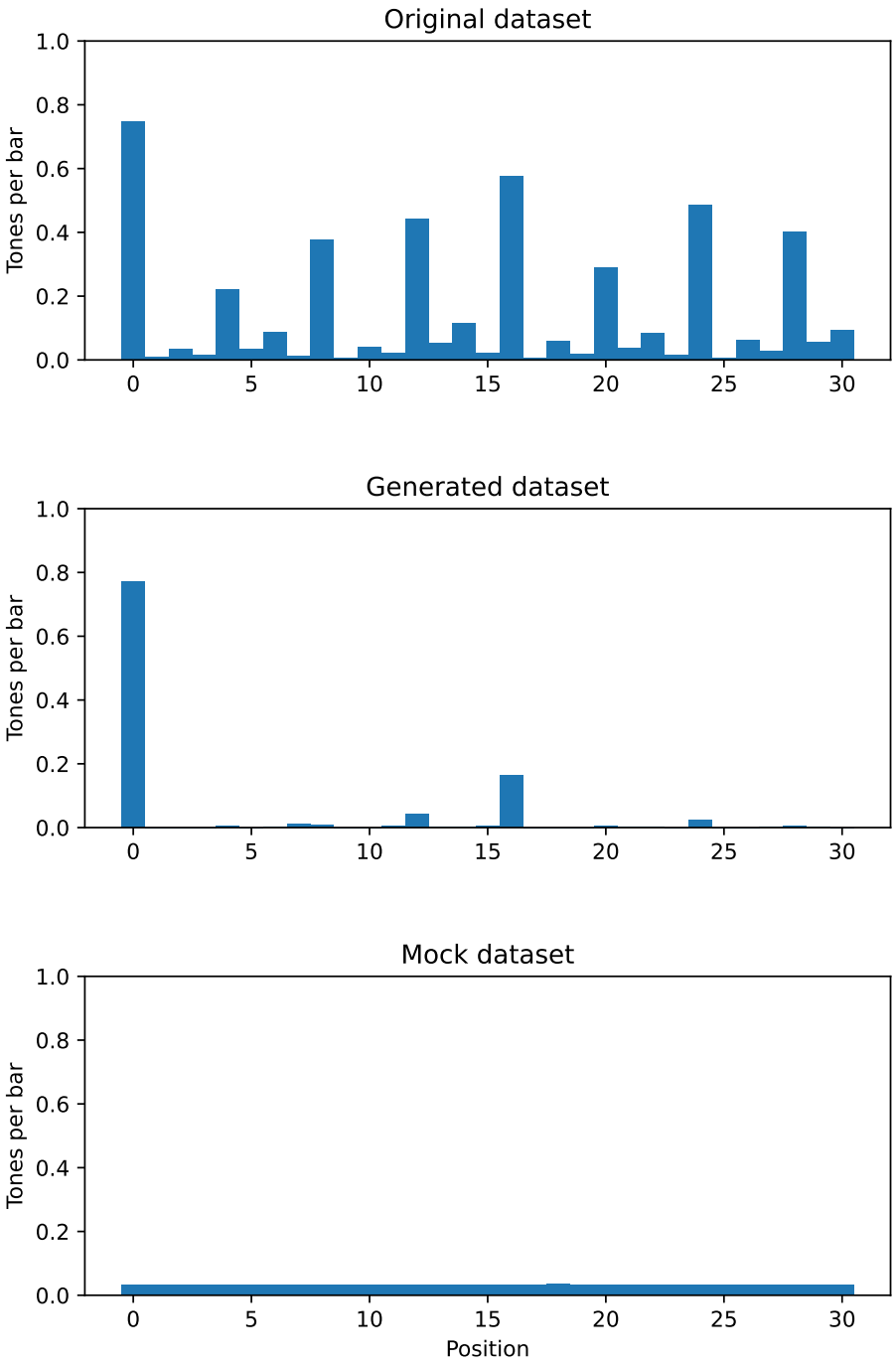


Figure 14: Rhythm distributions for the bass datasets.

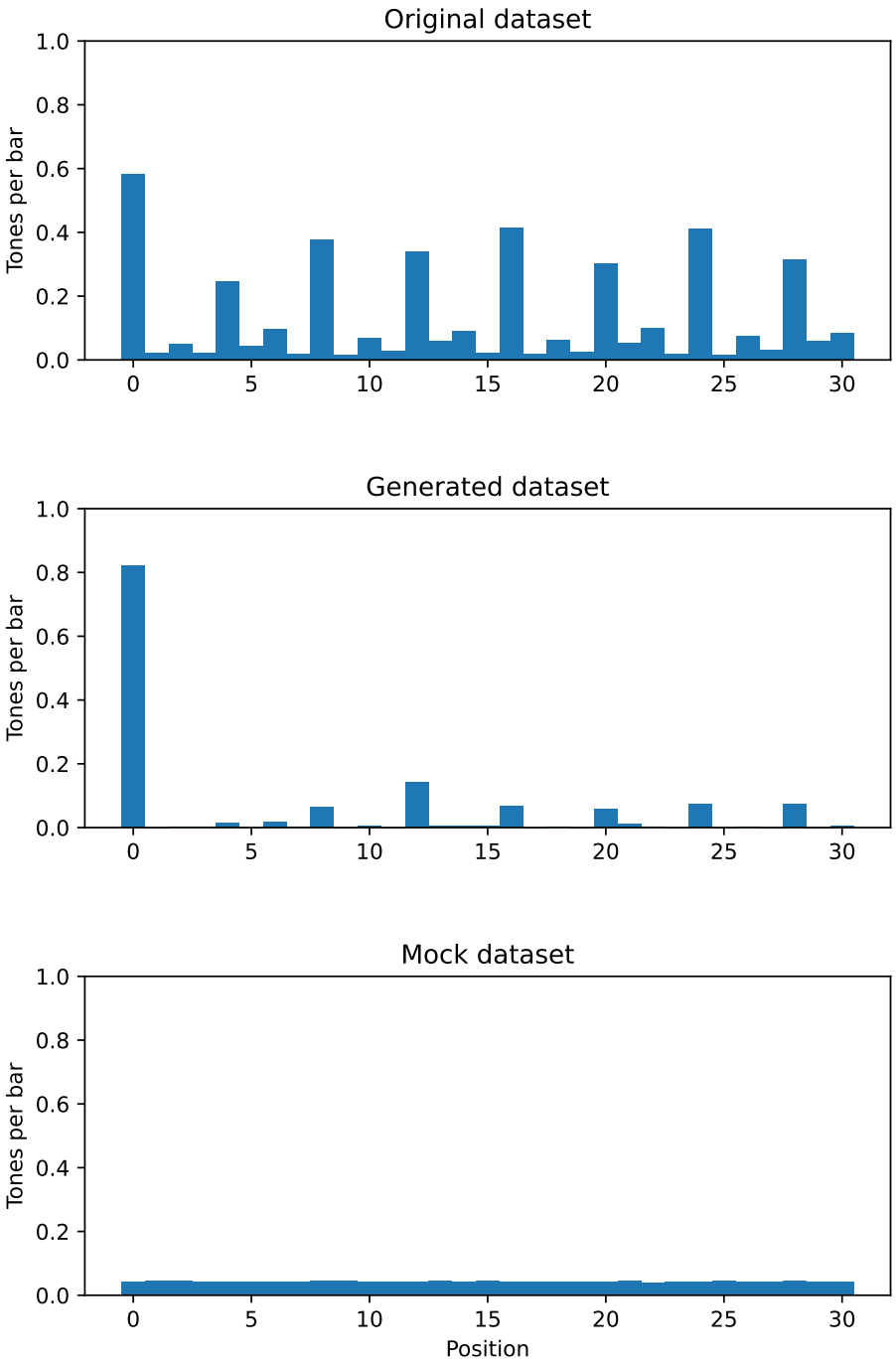


Figure 15: Rhythm distributions for the piano datasets.

corresponding note is sounding. This gives us a Pearson correlation coefficient which can be interpreted as a metric for how likely the sample is from said key. [43] [44]

The analysis was performed by calculating the most likely key from the reference sample and the top k key-candidates from the generated sample, and checking whether the key of the reference sample found from the key-candidates. This was done for all the generated samples.

Figures 16 and 17 show the fraction of successful matches for each value of k for key-candidates from generated samples and shuffled key-candidates. The shuffled candidates result in a triangle shaped distribution, where the number of matches with the original samples' keys increase linearly with the amount of top k candidates, which was expected behavior. For both the bass and the piano scenarios, the generated samples' top k key candidates shared more matches with the original key than the shuffled key-candidates, especially with lower values of k. For bass, in nearly third of the samples the first key candidate was the same as the original samples key. The piano model does not quite get there, with the first candidate being the same as the original roughly one time out of six, however already among the top 3 candidates the piano model catches up to the bass model in this regard. For a perfect model, it would be expected that almost all the samples share the top one match. However, since the models outperform the random mock-case by a mile, it is safe to say that the models have learned a non-zero amount from tonality.

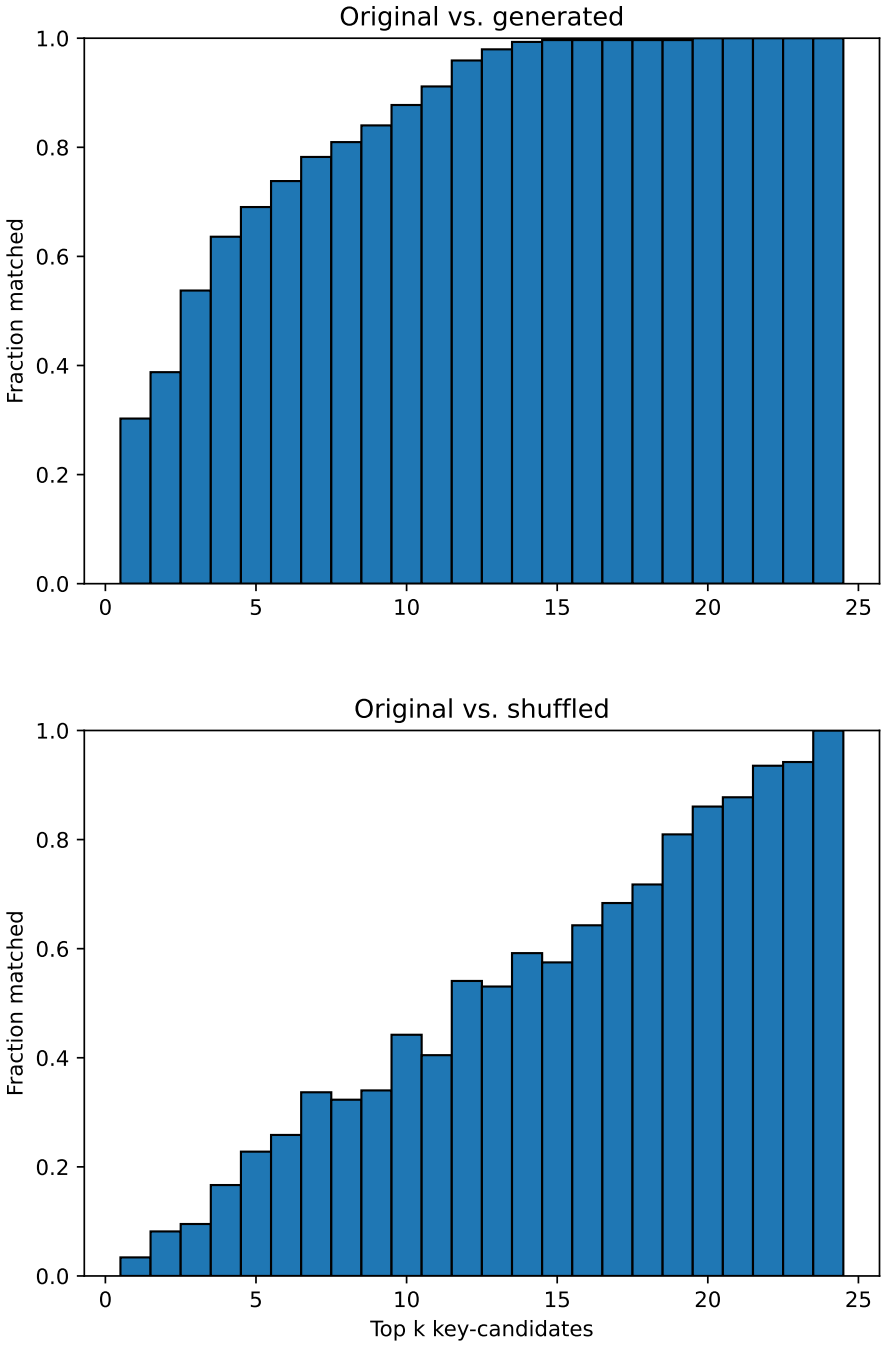


Figure 16: The fraction of bass model generated samples that have a matching key of the reference sample in the top k key-candidates, given by the Krumhansl-Schmuckler key finding algorithm.

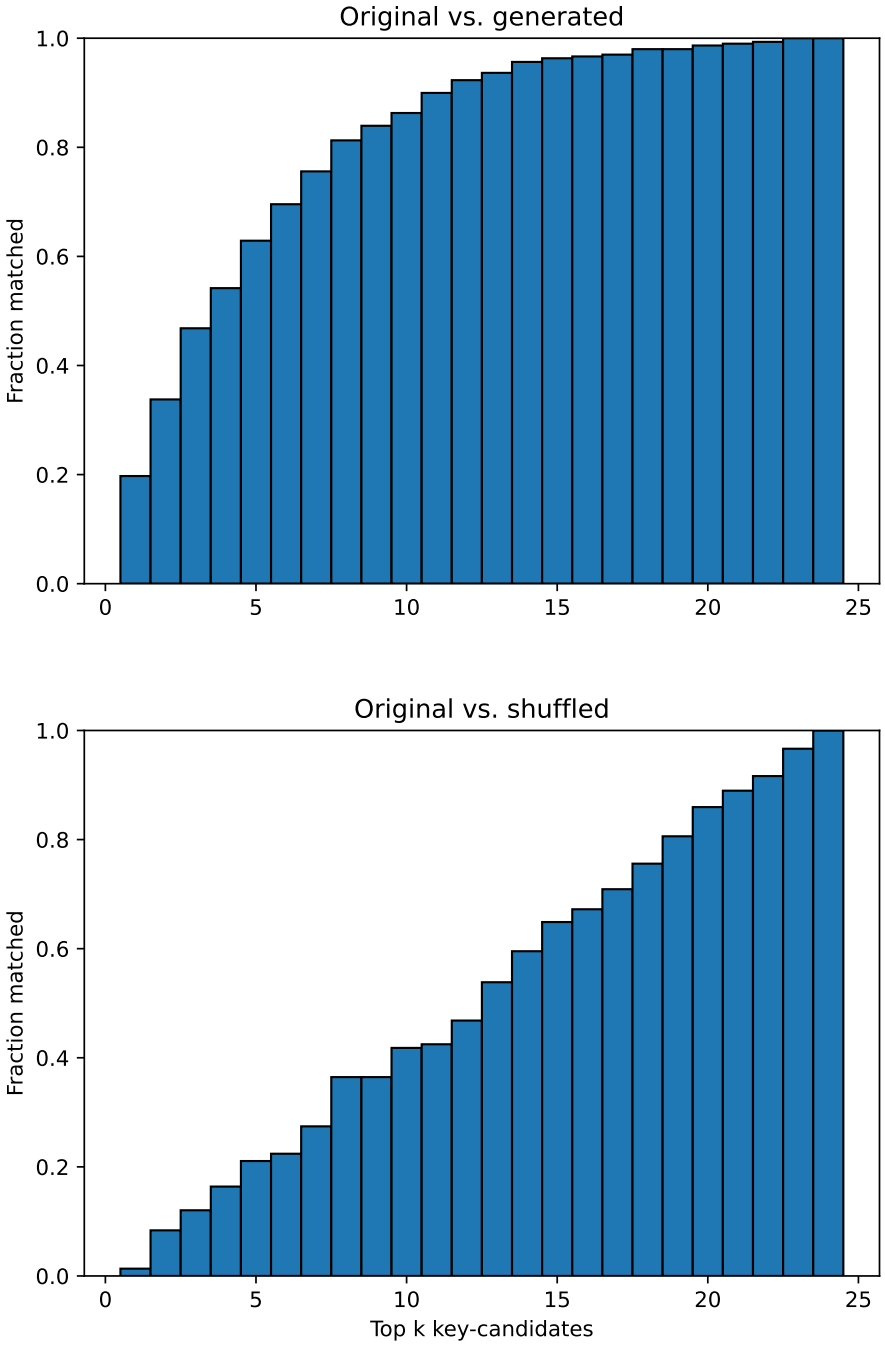


Figure 17: The fraction of piano model generated samples that have a matching key of the reference sample in the top k key-candidates, given by the Krumhansl-Schmuckler key finding algorithm.

4.3.3 Generated examples

For producing generated example samples for a more general analysis, a simple four chord progression snippet of 24 bars was composed, shown in Figure 18. It follows the classic "I V vi IV" chord progression that is present in many popular songs. The key of the snippet is in Bb major, and the bass loops a simple pattern while the piano plays accompaniment style chords with hints of melodic movement. The piece is tokenized, and the piano part of the snippet is used as the reference for bass-generation and vice-versa. A total of 3 generation cases were set up per instrument, one with a model that is trained for a long time and two with a model that is trained for a shorter time, one of which is a transposed version of the original snippet. For each case, two generation examples were produced: one with only the other instrument as reference and one with some of the original clip as well.

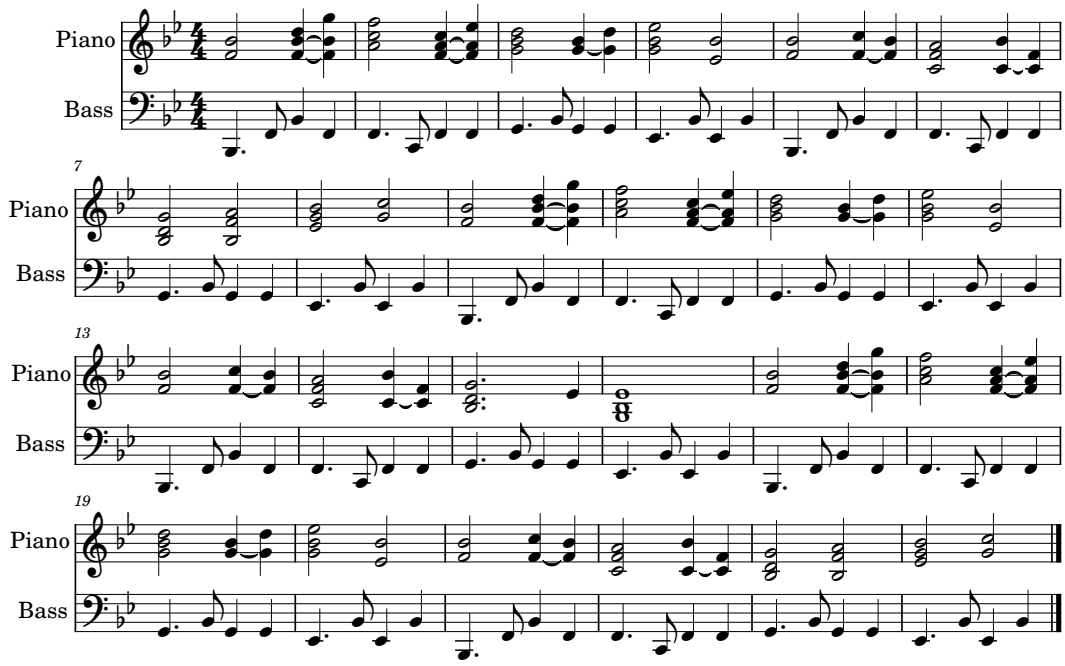


Figure 18: An example 24-bar snippet with a simple four chord progression for piano and bass.

Figures 19 and 20 show two generation cases from the bass generation examples and Figure 21 shows one of the generation cases from the piano generation examples (see Appendix A for the remaining generation cases). The generated examples show that the models manage to extract some information from the reference piece and use that information to generate somewhat instrument-like corresponding samples. However, all the smaller scale details of harmony are ignored by the models. Instead of matching the underlying harmony at the time, the models generate seemingly random notes from the correct key. Furthermore, the generations are very note-sparse, consisting mostly of silence. More detailed explanations of the results are given in the captions of the figures.

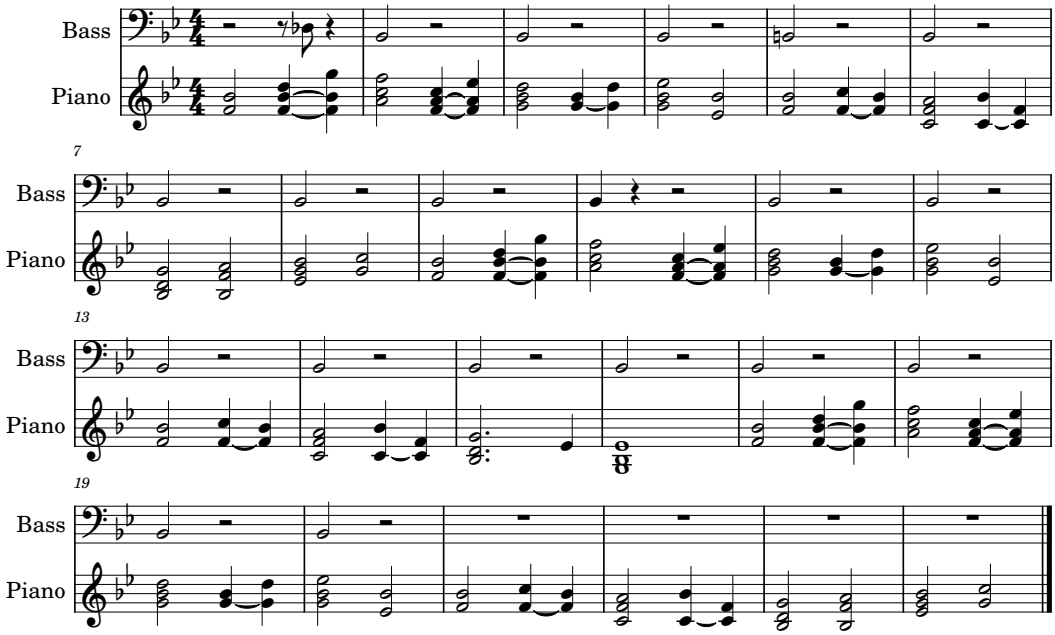


Figure 19: A generation example for a bass generating model that was trained until convergence of the validation loss for 20000 steps. The generated bass does not react to the changes in the piano reference well: changes to chords in the reference are not reflected in the generated bass in any way. The Db in the first bar is out of place, since it is not in the key and even clashes especially severely with the underlying Bb major chord. The abrupt H in the fifth bar is just as out of place. Other than that the generated bass only repeats Bb, which is the first degree note of the key, every downbeat.

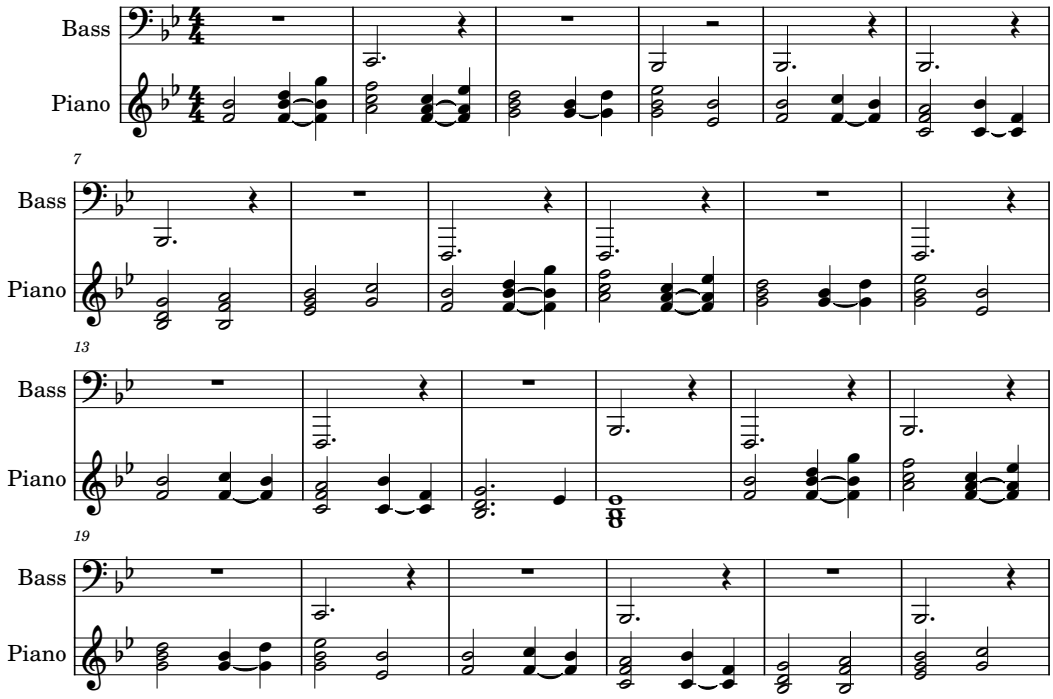


Figure 20: A generation example from a bass generating model that has been trained only for 1000 steps. The model seems to make bass-like decisions on what to generate, but does not seem to react to the chords changing in the reference, instead it appears to output random 3 beats long Bb, C and F notes on the downbeat. Although the generation is not responsive to the reference piano part, the notes are very natural choices in the key of Bb major.

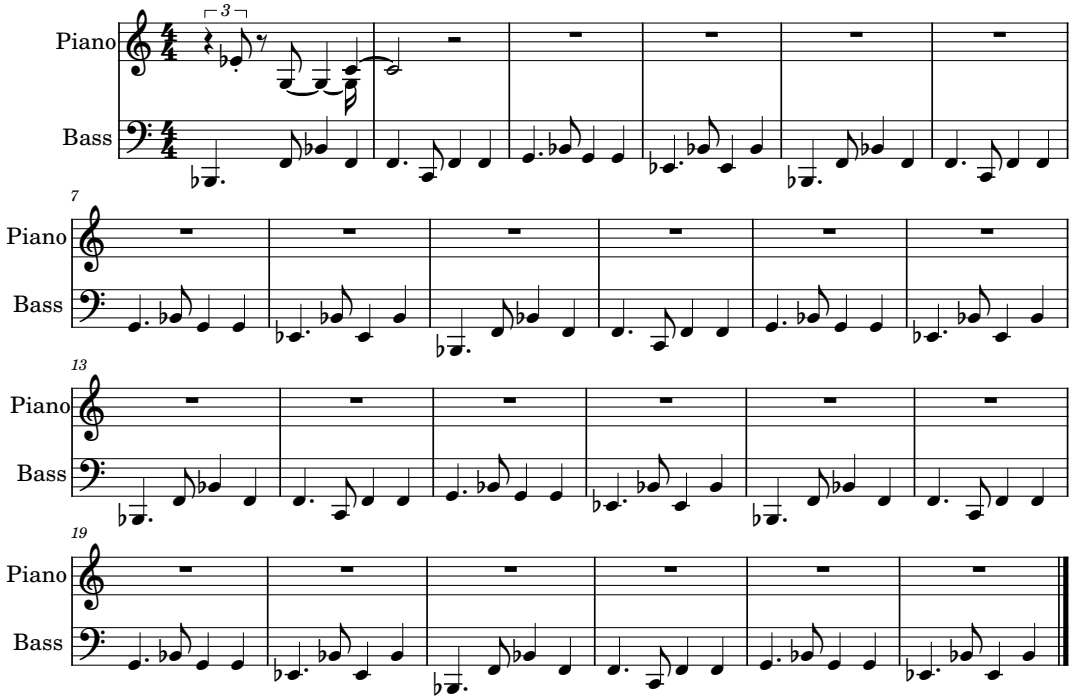


Figure 21: A generation example from a piano generating model with 20000 training steps. The model generates a small sequence of notes that are in key, then just silence. The sequence of notes is more piano-like than the generated bass examples.

5 Conclusions

The neural network models trained in this project managed to learn some of the features present in the music data that they were trained on, such as elements of rhythm and tonality. Ultimately, the end results that the models manage to generate are musically unimpressive. The models do not seem to react to the reference sample in a smaller scale, as in, there seems to be no reaction to something changing bar-to-bar in the reference. Instead the models seem to be able to interpret some features from the sample in a macro scale, such as the key of the piece, and then randomly generate something that could resemble the instrument trained on, based on the larger-scale information that the model extracts from the reference used for conditioning.

There are multiple factors that could be the reason why the models do not perform as well as similar models trained in previous studies. The amount of training the models received is tiny compared to training amounts that deep learning models usually go through. For example for the LakhNES model, the top-performer was first pre-trained on the whole lakh dataset for 400k steps using a mini-batch size of 30 before fine-tuning on the NES-MDB dataset for more training [9]. The training of the models in this project already converged after roughly 20k steps with a smaller batch size, and with the current generation parameters only generated more and more silence. This could potentially be fixed by using a larger dataset, either by performing a large amount of data-augmentation such as transposing the samples into multiple keys, or just having a larger dataset in the first place. Trying different training parameters such as the learning rate could also help the model to converge slower and learn more while avoiding over-fitting.

The quality of the dataset could also be an issue. Not a lot of qualitative sample-by-sample analysis was done on the dataset, and the process that was used to create the reference-target instrument samples does not guarantee that the samples are of good quality. Since the original dataset does not consist strictly of music pieces that are written for piano and bass, there is a high likelihood that outlier samples exist, for example samples where either one of the instrument is silent for almost the entire duration of the sample. The more samples there are where the target instrument is silent, the more there is a risk of a mode collapse towards just producing silence, that is, the model converges to a local minimum where the winning strategy is only generating empty bars instead of trying to guess the correct tones. This could be counteracted by creating metrics for filtering outliers from the dataset, such as note density. Note density could also be used as a conditioning metric for generation, as was done in the Multi-Track Music Machine paper [10], so that the model could be asked to generate "denser" music than silence.

The inference method could also be something to improve on. The sampling parameters could be experimented on to find better candidates, or a whole other sampling strategy could be utilized, such as nucleus sampling, typical sampling or ancestral sampling [45]. Additionally, masking unwanted token candidates during generation could lead to interesting results, but this can be difficult when working with BPE-encoded tokens.

The tokenization or model parameters could also be an issue. In a black-box model

as complex as the one in this project, the pinpointing of the possible issues can be challenging and would require plenty of experimentation beyond the scope of this thesis.

Art generating models can be useful tools for any artist and non-artist alike, either for finding inspiration or producing a template for some project, or using the model to create some finished product. Even so, developing art generating technologies do not come without issues. While the model implemented in this thesis does not pose a threat to any musician's livelihood, the societal, cultural and economical effects of AI-generated art should not be disregarded in any setting that works on the domain. As AI-generated art becomes more and more accessible, the work of skilled artists might get underappreciated, especially by non-professional consumers. [46] Furthermore, using copyrighted pieces as training data for generation models does not require providing any compensation for the original authors of the piece currently, even if the generation model would be used commercially, which is problematic.

In the end, the models that were trained in this thesis managed to generate some nearly instrument-like samples. However, the generated samples were modest at best. If the objective would be to create a model that produces complementing instrument tracks with good-quality, a simple procedural model would likely achieve much better results with less implementation effort and computational resources. While not usable in any real-life applications as is, the models and methods that were created and examined in this thesis could serve as a starting point for building an improved version of the model in the future.

References

[1] Miguel Civit et al. "A systematic review of artificial intelligence-based music generation: Scope, applications, and future trends". *Expert Systems with Applications* 209 (2022), p. 118190. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2022.118190. URL: https://www.sciencedirect.com/science/article/pii/S0957417422013537.

[2] Andrea Agostinelli et al. *MusicLM: Generating Music From Text*. 2023. arXiv: 2301.11325 [cs.SD].

[3] Prafulla Dhariwal et al. *Jukebox: A Generative Model for Music*. 2020. arXiv: 2005.00341 [eess.AS].

[4] Saidul Islam et al. "A comprehensive survey on applications of transformers for deep learning tasks". *Expert Systems with Applications* 241 (2024), p. 122666. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2023.122666. URL: https://www.sciencedirect.com/science/article/pii/S0957417423031688.

[5] Tom Brown et al. "Language models are few-shot learners". *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[6] Jacob Devlin et al. "BERT: Pre-training of deep bidirectional transformers for language understanding". *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[7] Ashish Vaswani et al. "Attention is all you need". *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[8] Cheng-Zhi Anna Huang et al. "Music transformer". *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJe4ShAcF7.

[9] Chris Donahue et al. "LakhNES: Improving multi-instrumental music generation with cross-domain pre-training". *Proceedings of the 20th International Society for Music Information Retrieval Conference* (Delft, The Netherlands). ISMIR, Nov. 2019, pp. 685–692. DOI: 10.5281/zenodo.3527902. URL: https://doi.org/10.5281/zenodo.3527902.

[10] Jeff Ens and Philippe Pasquier. *MMM : Exploring Conditional Multi-Track Music Generation with the Transformer*. 2020. arXiv: 2008.06048 [cs.SD].

[11] Colin Raffel. *Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching*. 2016. URL: https://colinraffel.com/projects/lmd/.

[12] Chris Donahue, Huanru Henry Mao, and Julian McAuley. "The NES music database: A multi-instrumental dataset with expressive performance attributes". *Proceedings of the 19th International Society for Music Information Retrieval Conference* (Paris, France). ISMIR, Nov. 2018, pp. 475–482. DOI: 10.5281/zenodo.1492455. URL: https://doi.org/10.5281/zenodo.1492455.

[13] Alexander Jung. *Machine Learning: The Basics*. Springer Singapore, 2022. ISBN: 978-981-16-8193-6. DOI: https://doi.org/10.1007/978-981-16-8193-6.

[14] Paolo Fantozzi and Maurizio Naldi. "The explainability of transformers: Current status and directions". *Computers* 13.4 (2024). ISSN: 2073-431X. DOI: 10.3390/computers13040092. URL: https://www.mdpi.com/2073-431X/13/4/92.

[15] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". *International Journal of Engineering Applied Sciences and Technology* 04 (May 2020), pp. 310–316. DOI: 10.33564/IJEAST.2020.v04i12.054.

[16] David Bertoin et al. "Numerical influence of ReLU'(0) on backpropagation". *Advances in Neural Information Processing Systems*. Advances in Neural Information Processing Systems 34 (NeurIPS 2021). Paris, France, Dec. 2021. URL: https://hal.science/hal-03265059.

[17] Juan Terven et al. *Loss Functions and Metrics in Deep Learning*. 2023. arXiv: 2307.02694 [cs.LG].

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[19] John McGonagle et al. *Backpropagation. Brilliant.org*. Accessed 6.5.2024. 2024. URL: https://brilliant.org/wiki/backpropagation/.

[20] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[21] Will Koehrsen. *Neural Network Embeddings Explained – How deep learning can represent War and Peace as a vector*. Accessed 26.10.2023. 2018. URL: https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.

[22] Amirhossein Kazemnejad. "Transformer architecture: The positional encoding". *kazemnejad.com* (2019). URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[23] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[24] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[25] Shaked Brody, Uri Alon, and Eran Yahav. "On the expressivity role of LayerNorm in transformers' attention". *Findings of the Association for Computational Linguistics: ACL 2023*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 14211–14221. DOI: 10.18653/v1/2023.findings-acl.895. URL: https://aclanthology.org/2023.findings-acl.895.

[26] Kaiming He et al. "Deep residual learning for image recognition". *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[27] Andreas Veit, Michael J Wilber, and Serge Belongie. "Residual networks behave like ensembles of relatively shallow networks". *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/37bc2f75bf1bcfe8450a1a41c200364c-Paper.pdf.

[28] Thomas Wolf et al. "Transformers: State-of-the-art natural language processing". *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[29] Huggingface. *Summary of the tokenizers*. Accessed 18.10.2023. 2023. URL: https://huggingface.co/docs/transformers/tokenizer_summary.

[30] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. "Neural machine translation with byte-level subwords". *Proceedings of the AAAI Conference on Artificial Intelligence* 34.05 (Apr. 2020), pp. 9154–9160. DOI: 10.1609/aaai.v34i05.6451. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6451.

[31] Adarsh Kumar and Pedro Sarmento. *From Words to Music: A Study of Subword Tokenization Techniques in Symbolic Music Generation*. 2023. arXiv: 2304.08953 [cs.SD].

[32] Nathan Fradet et al. "Byte pair encoding for symbolic music". *The 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 2023)*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. EMNLP 2023. Association for Computational Linguistics. Singapore, Singapore: Association for Computational Linguistics, Dec. 2023, pp. 2001–2020. DOI: 10.18653/v1/2023.emnlp-main.123. URL: https://hal.science/hal-03976252.

[33] Dorien Herremans, Ching-Hua Chuan, and Elaine Chew. "A functional taxonomy of music generation systems". *ACM Computing Surveys* 50.5 (Sept. 2017). ISSN: 0360-0300. DOI: 10.1145/3108242. URL: https://doi-org.libproxy.aalto.fi/10.1145/3108242.

[34] Aarre Joutsenvirta and Jari Perkiömäki. *Music Theory 1*. Accessed on 9.10.2023. 2023. URL: https://sites.uniarts.fi/en/web/musiikin-teoria-1/.

[35] *Standard MIDI-File Format Spec. 1.1, updated*. Accessed on 17.10.2023. URL: http://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html.

[36] *py_midicsv Python library*. URL: https://pypi.org/project/py-midicsv/.

[37] Nathan Fradet et al. "MidiTok: A Python package for MIDI file tokenization". *Extended Abstracts for the Late-Breaking Demo Session of the 22nd International Society for Music Information Retrieval Conference*. 2021. URL: https://archives.ismir.net/ismir2021/latebreaking/000005.pdf.

[38] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". *Advances in neural information processing systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[39] Yu-Siang Huang and Yi-Hsuan Yang. "Pop music transformer: Beat-based modeling and generation of expressive pop piano compositions". *Proceedings of the 28th ACM International Conference on Multimedia*. MM '20. Seattle, WA, USA: Association for Computing Machinery, 2020, pp. 1180–1188. ISBN: 9781450379885. DOI: 10.1145/3394171.3413671. URL: https://doi.org/10.1145/3394171.3413671.

[40] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. *Self-Attention with Relative Position Representations*. 2018. arXiv: 1803.02155 [cs.CL].

[41] Ari Holtzman et al. "The curious case of neural text degeneration". *Proceedings of International Conference on Learning Representations*. 2020. URL: https://pubs.cs.uct.ac.za/id/eprint/1407/.

[42] Juan Felipe Beltran et al. "Measuring musical rhythm similarity - statistical features versus transformation methods". *International Conference on Pattern Recognition Applications and Methods*. 2013. URL: https://api.semanticscholar.org/CorpusID:5175340.

[43] Carol L. Krumhansl. *Cognitive Foundations of Musical Pitch*. Oxford University Press, Nov. 2001. ISBN: 9780195148367. DOI: 10.1093/acprof:oso/9780195148367.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780195148367.001.0001.

[44] Robert Hart. *Krumhansl-Schmukler Key-finding Algorithm*. Accessed on 11.5.2024. 2012. URL: http://rnhart.net/articles/key-finding/.

[45] Mathias Rose Bjare, Stefan Lattner, and Gerhard Widmer. *Exploring Sampling Techniques for Generating Melodies with a Transformer Language Model*. 2023. arXiv: 2308.09454 [cs.SD].

[46] Makhabbat Amanbay. "The ethics of AI-generated art". *Available at SSRN 4551467* (2023). URL: https://ssrn.com/abstract=4551467.

A Additional generation results

Figures A1 to A9 contain the generation results from the cases that were not presented in section 4.3.3, including cases where the generation is primed with an initial set of tokens, and cases where the original sample was transposed by a semitone.

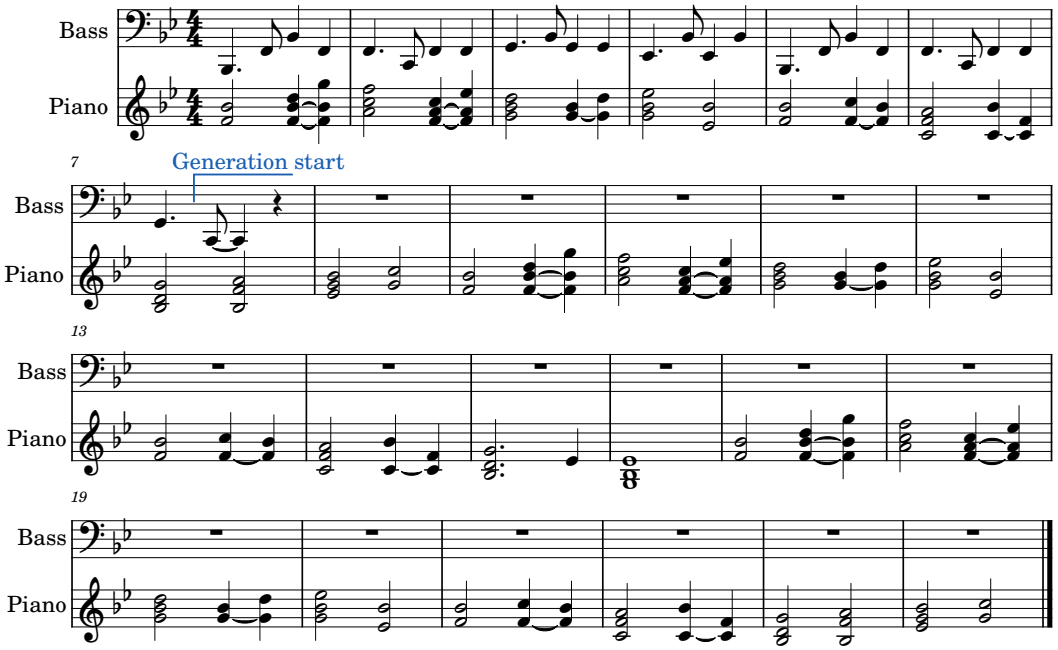


Figure A1: A generation example from a bass generating model that has been trained for 20000 steps, where the generation was primed with 50 initial tokens from the original tokenized piece. Here the model only produces one C, which does follow the earlier rhythmic pattern and is a note that exists in this key, but is definitely not any of the more obvious choices, such as the notes that are in the underlying G minor chord. After generating the C, the rest is just silence. This is an effect that was perceived with the model overall: the more they were trained the more certain they were about producing just silence.

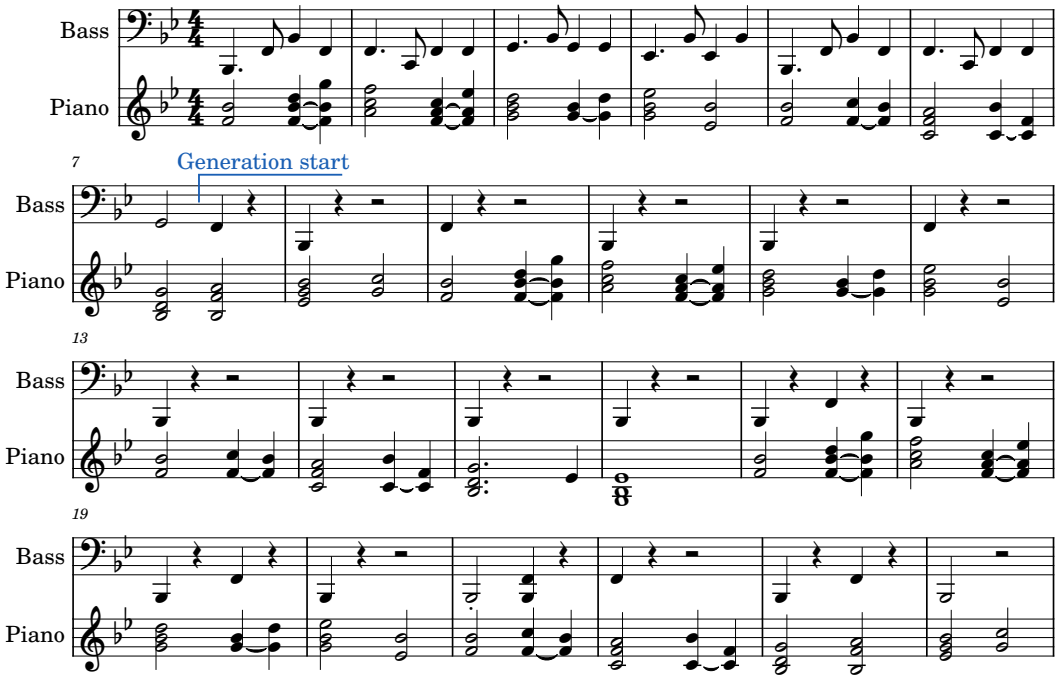


Figure A2: A generation example from a bass generating model that has been trained for 1000 steps, where the generation was primed with 50 initial tokens. The model generates random Bb and an F notes on the downbeat, so no particularly interesting differences to the case where no initial tokens were given.

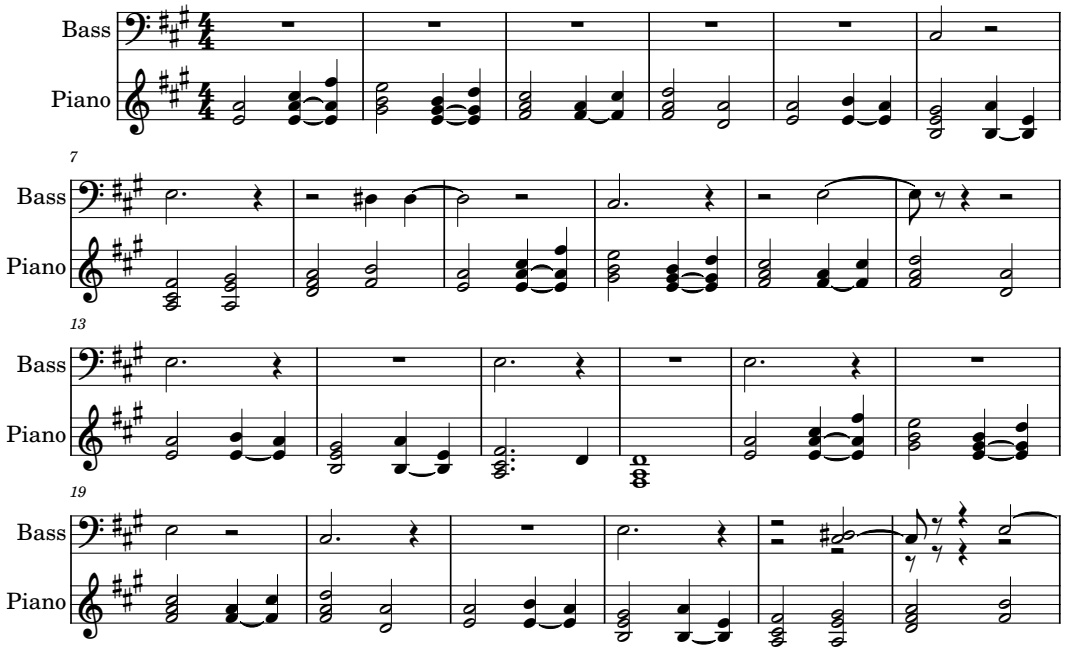


Figure A3: A generation example from a bass generating model trained for 1000 steps, where the original piece has been transposed by a semitone into A major. Here the model generates E, D# and C# tones, where E and C# are quite natural choices for any tune in A major, but the D# is a little bit out of place. The D# could "lead" to the dominant fifth degree E so it might act as a secondary dominant in the key of A-major, but the D# is nevertheless a less natural choice in this context, and would need a bit more justification by surrounding harmony decisions to make sense.

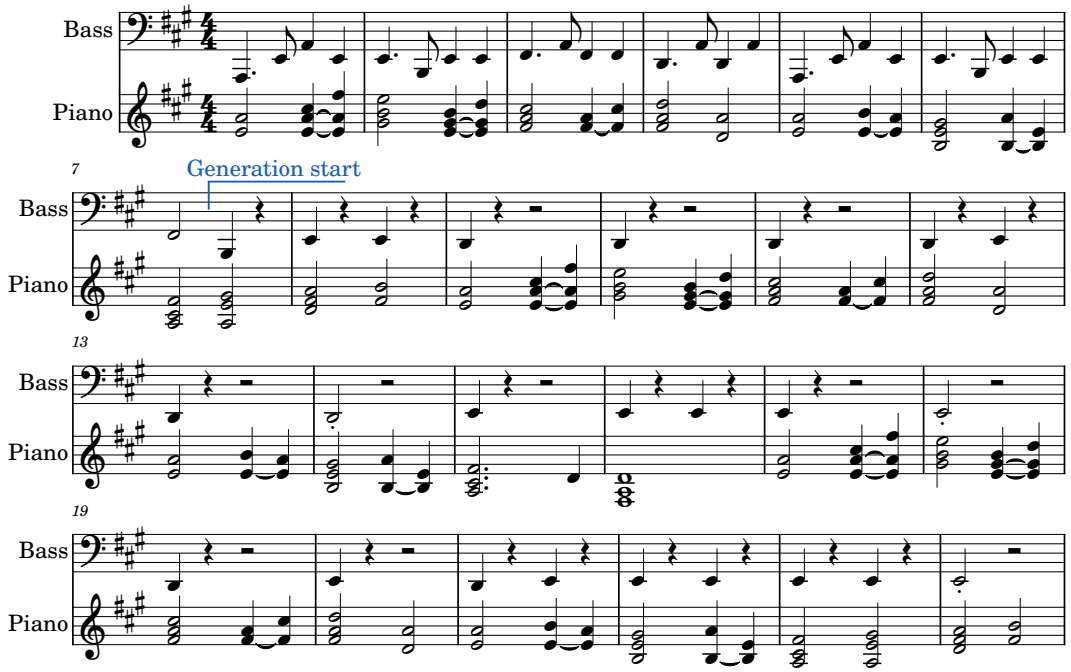


Figure A4: A generation example from a bass generating model trained with 1000 steps, where the key is in A-major and the model is primed with 50 initial tokens. Here the model just outputs a D and a C# throughout the sample, and while both make sense in A-major, they do not themselves convince that the model has understood the exact key that they are in.

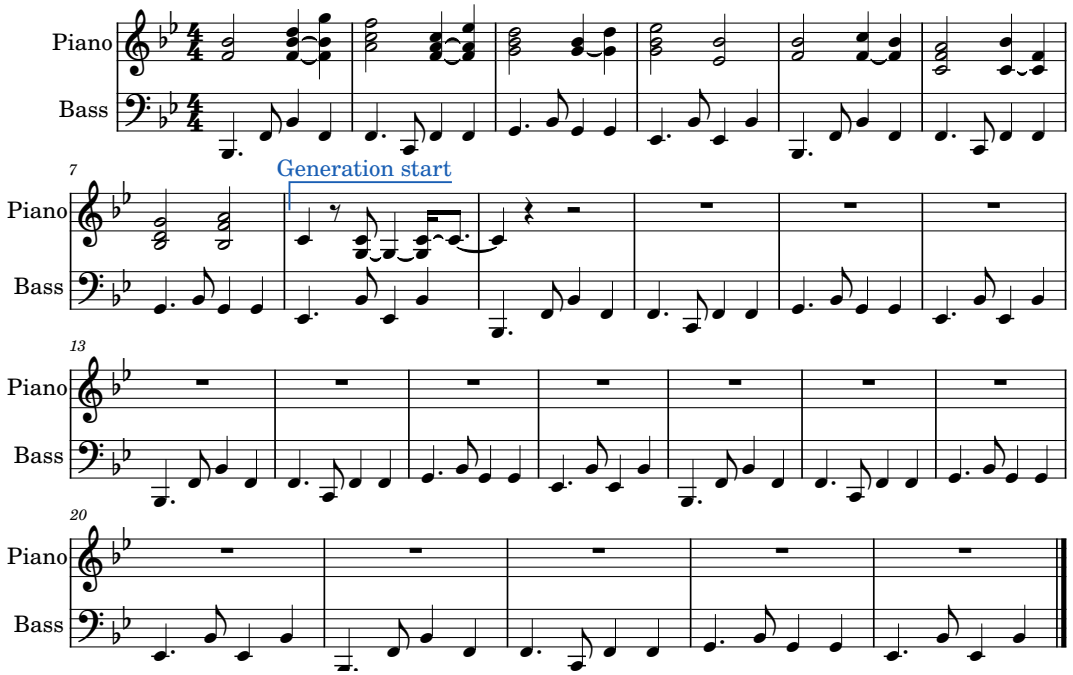


Figure A5: A generation example from a piano generating model with 20000 training steps, primed with 75 initial tokens. The model generates a bar worth of tones in the correct key and the rest is silence.

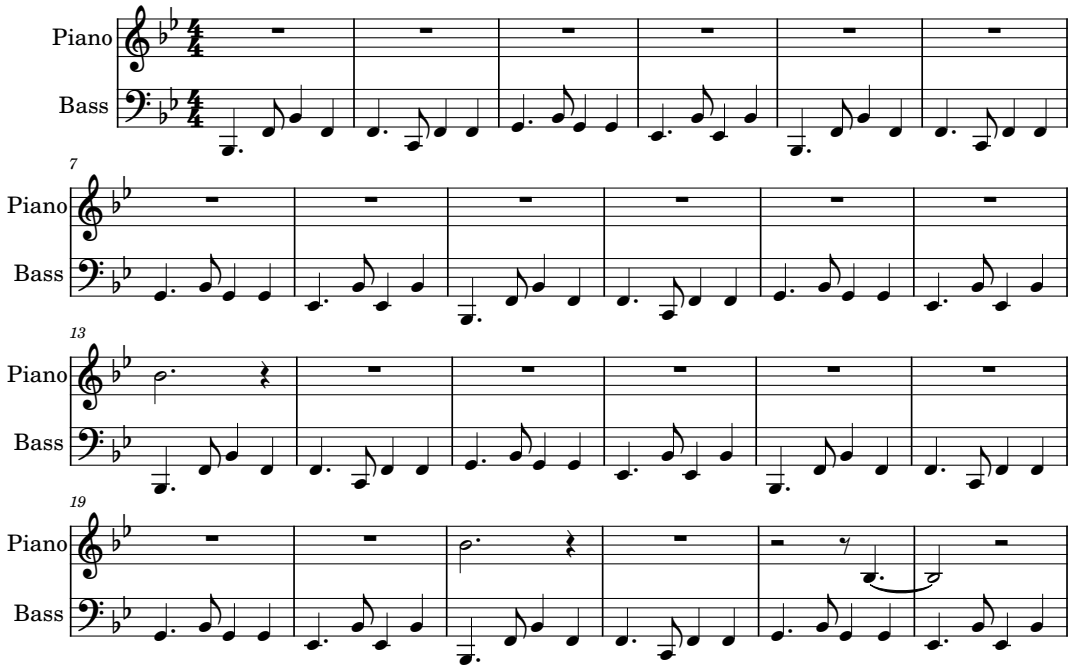


Figure A6: A generation example from a piano generating model with 500 training steps. The model generates almost nothing for the most part, then a single Bb note, later another Bb note and finally one more Bb note except an octave lower.

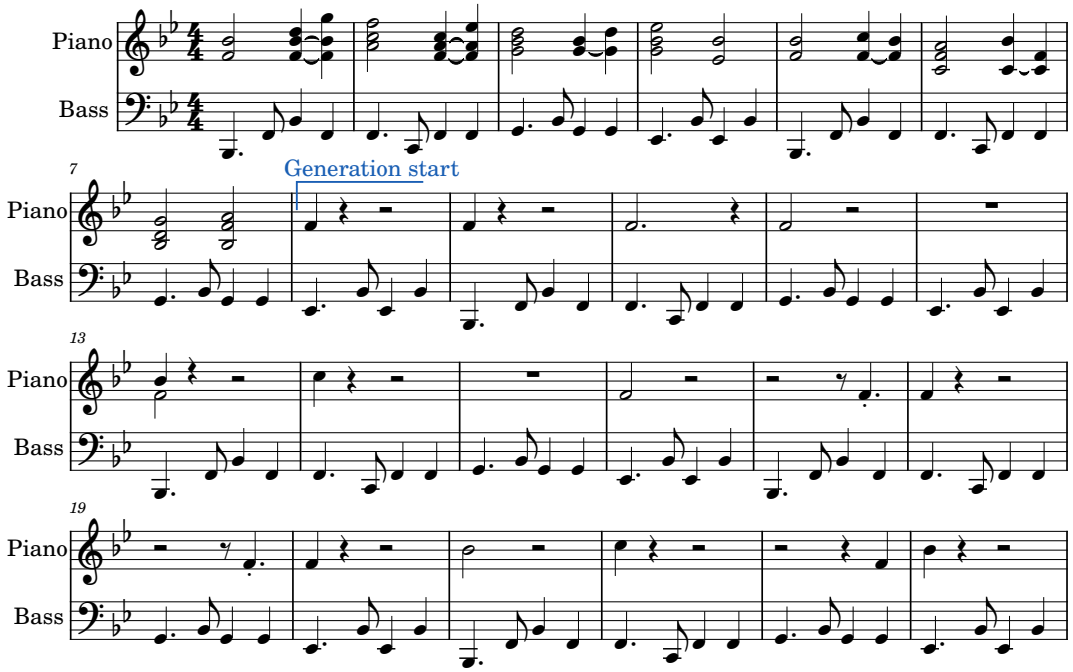


Figure A7: A generation example from a piano generating model with 500 training steps, primed with 75 initial tokens. This time the model generates a bit more music, using F, Bb and C, all natural choices in the key of Bb-major.

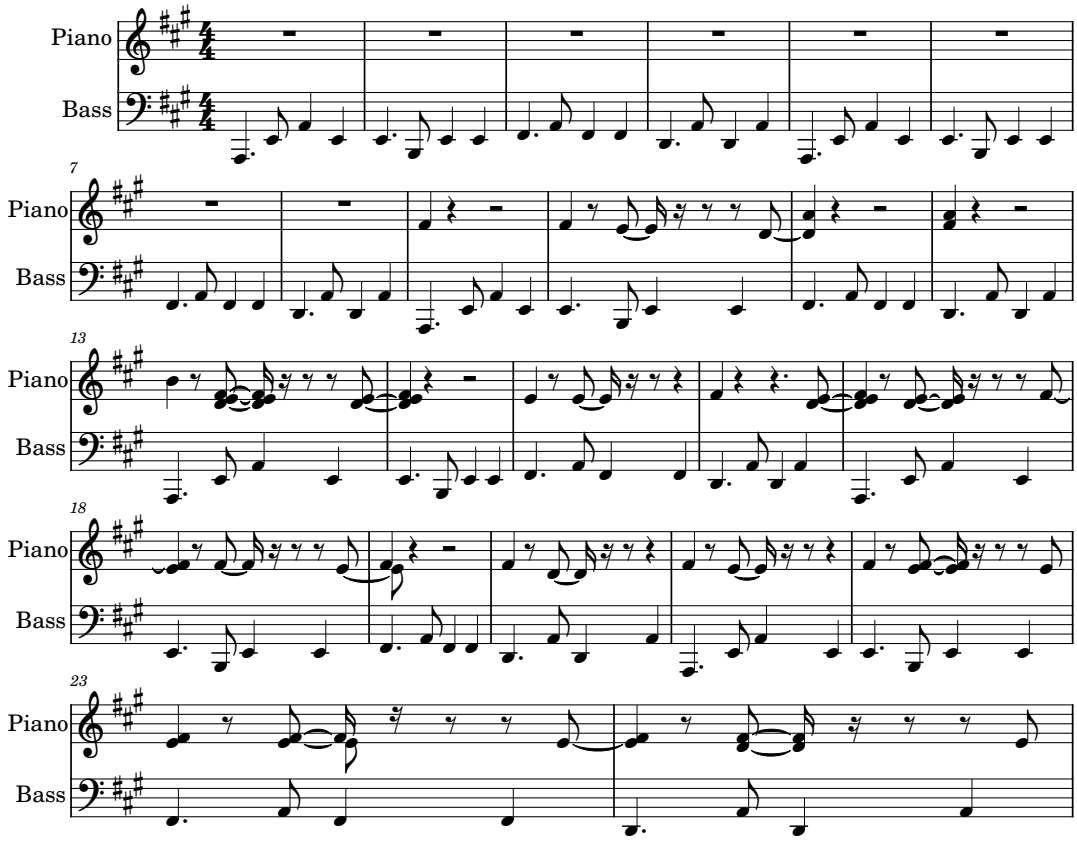


Figure A8: A generation example for a piano generating model with 500 training steps, where the reference piece is transposed to A-major. The model generates a lot more music than the Bb-major counterparts, even playing some non-triad chords every now and then. Once more it seems like the model generates random notes in the correct key, just much more densely than in Bb-major.

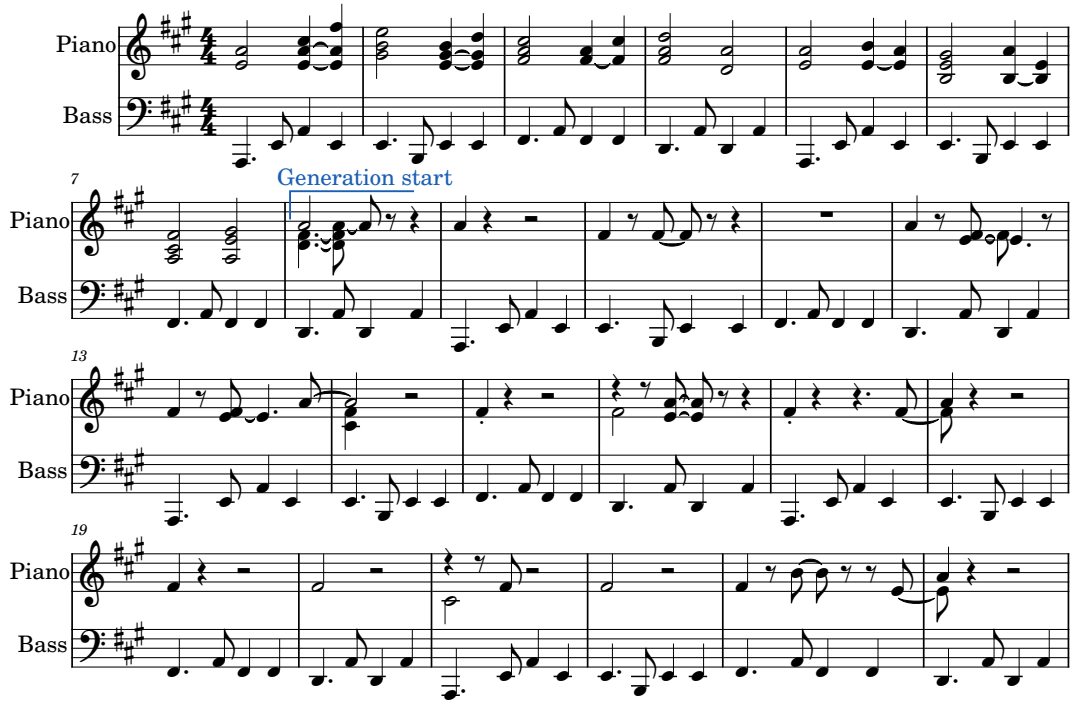


Figure A9: A generation example from a piano generating model with 500 training steps, where the reference piece is transposed to A-major and the model is prime with 75 initial tokens. The generation acts very similarly to the version that does not get any initial tokens, outputting notes in the correct key seemingly randomly.