Contents

1 Introduction

An increasing number of neural networks have been proposed lately for music generation. However, the task remains challenging due to the following reasons. First, music is an art of time, necessitating a temporal model. Second, music is usually composed of multiple instruments/tracks, with close interaction with one another. Each track has its own temporal dynamics, but they unfold over time interdependently. Lastly, for symbolic domain music generation, the targeted output is sequences of discrete musical events, not continuous values.

Our models aim to generate both single-track monophonic music, a musical texture with just one voice, and single-track polyphonic music, a type of musical texture consisting of two or more simultaneous lines of independent melody. Generating multi-track music would require a very complex model with a lot of inter-dependencies and it would be unfeasible with the computing resources at our disposal.

We aim to train two types of neural networks, namely LSMTs and Transformers, in order to generate new music from a dataset of single track MIDI songs.

- **Long Short-Term Memory (LSTM)** networks are a type of recurrent neural network that are particularly well-suited for processing sequential data, such as music. LSTMs are able to "remember" past events in a sequence and use this information to make predictions about future events. This makes them particularly effective at generating music, as they are able to capture the patterns and structure of a piece of music and use this information to generate new compositions.
- **Tranformers** are a type of neural network architecture that have become popular due to their ability to process sequences of data effectively. In the context of music generation, transformers can be used to generate new pieces of music. The main advantage of using transformers for music generation is their ability to handle long-term dependencies. In music, long-term dependencies refer to the relationship between musical events that occur over an extended period of time, such as the relationship between different phrases or sections of a piece. Transformers are able to capture these long-term dependencies by attending to different parts of the input sequence, allowing them to generate music that is coherent and structured.

2 Related Work

Automatic music generation has a long history. Its development started more than 60 years ago. During this period, a large number of researchers utilized different methods, such as grammar rules, probability models, evolutionary computation, neural networks, and so on. There are numerous researches on various datasets aiming at different generation tasks.

With the advent of deep learning, using deep learning technologies to automatically generate various contents (images, text, etc.) has become a hot topic. One of the most universal art forms in human society is music, which has attracted plenty of researchers. So far, dozens of music generation subtasks have been researched, such as melody generation, chord generation, style transfer, performance generation, audio synthesis and singing voice synthesis.

Currently, deep learning algorithms have become the mainstream method in the field of music generation research. RNNs were the first neural network architecture to be used for music generation. However, due to the gradient vanishing problem, it becomes difficult for RNNs to store long historical information about sequences. To solve this problem, a special RNN architecture has been designed - the LSTM, Long Short Term Memory - in order to assist the network in memorizing and retrieving information in the sequence. In 2002, LSTM was used in music creation for the first time, improvising blues music with good rhythm and reasonable structure based on a short recording.

With the continuous development of deep learning technologies, powerful deep generative models such as VAEs, GANs, and Transformers have gradually emerged. GANs are very powerful but notoriously difficult to train and are usually not applied to sequential data. However, Yang and Dong demonstrated the ability of CNN-based GANs in music generation. Specifically, Yang proposed a GAN-based MidiNet [9] to generate melody one bar after another, and proposed a novel conditional mechanism to generate current bar conditioned on chords. The MuseGAN [2] model proposed by Dong is considered to be the first model that can generate multi-track polyphonic music. Yu successfully applied RNN-based GAN [10] to music generation for the first time by combining reinforcement learning technology.

Recently, Transformer models have shown its great potential in music generation. Donahue proposed using Transformers to generate multi-instrument music [1]. Huang proposed a new music representation method named REMI and exploited the language model Transformer XL [5] as sequence model in order to generate popular piano music.

3 Methods and Experiments

3.1 Datasets

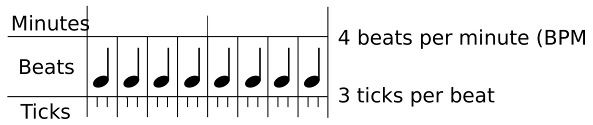
For the scope of our project we used two different datasets: (i) *Jazz music*, a database of jazz solos retrieved from music database Weimar; and (ii) *piano*, a piano MIDI dataset that contains 775 MIDI files of piano performances. We used the Jazz Dataset for monophonic music generation and the Piano Dataset for polyphonic music generation.
Our datasets are composed of MIDI files, one for each song. MIDI does not record analog or digital sound waves. It encodes keyboard functions, which includes the start of a note, its pitch, length and velocity, namely the intensity of the note. As a result, MIDI files take up considerably less space than digitized sound files.
A MIDI file is composed of events. These events can have different types, but the ones that we are interested in are the *"note_on"* events, which means that a note has been played. In order to decode our MIDI files we used the MidiTok [8] Python library. Here's an example of MIDI encoding:

```
0: pitch=56, note_name=G#3, duration=0.0352
1: pitch=44, note_name=G#2, duration=0.0417
2: pitch=68, note_name=G#4, duration=0.0651
3: pitch=80, note_name=G#5, duration=0.1693
4: pitch=78, note_name=F#5, duration=0.1523
5: pitch=76, note_name=E5, duration=0.1120
6: pitch=75, note_name=D#5, duration=0.0612
7: pitch=49, note_name=C#3, duration=0.0378
8: pitch=85, note_name=C#6, duration=0.0352
```

3.2 Encoding

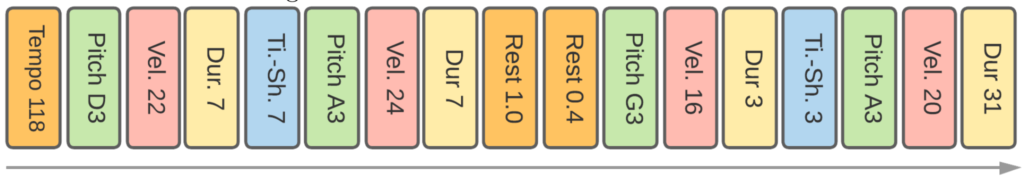
Before delving into the encoding it's better to make a step back and explain how time is represented in MIDI tracks. We can assume that all of our songs are in 4/4, which means that a music bar is composed of 4 quarter notes, also called beats, a unit of measure typically used to define the tempo of a song in BPM (Beats per minute). Timing in MIDI files is centered around beats and ticks. Beats are divided into ticks, the smallest unit of time in MIDI. Our files have a precision of 384 ticks per beat.

Figure 1: MIDI Tempo and Beat Resolution



We encoded our files using MidiTok, a Python library that converts MIDI music files into sequences of tokens, i.e. integers, ready to be fed to sequential neural networks like Transformers or RNNs. MidiTok features most known MIDI tokenization strategies, like *MIDI-Like*, *TimeShift Duration* and *REMI*. The base tokenizer, MIDI-Like, represents MIDI messages (Note On, Note Off, Velocity and Time Shift) as tokens. For our project we decided to use **TimeShift Duration (TSD)**, a strategy similar to MIDI-Like that uses explicit `Duration` tokens to represent note duration.

Figure 2: TimeShift Duration Tokenization



3.2.1 Jazz Dataset

We encoded the Jazz Dataset using the following tokens:

- Pitch, a perceptual property of sounds that allows their ordering on a frequency-related scale. In MIDI files the pitch ranges from 0, the lowest musical note, to 127, the highest one. Nonetheless the GM2 specification recommends ranges from 21 to 108 for piano, which covers the recommended pitch values for all MIDI program, so we discarded the notes that were outside of that range.
- Velocity, the force with which a note is played, vitally important in making MIDI performances sound human. Velocity ranges from 0, silent note, to 127, loudest sound, but MidiTok allows to define a number of velocity values to represent. For instance with 32, the velocities of the notes will be quantized into 32 velocity values from 0 to 127.
- TimeShift, the time between two note/rest events.
- Rest, a segment of time where the MIDI is silent, i.e. no note is played within. This token type is decoded as a TimeShift event.
- Duration, the length of a note.

These last three tokens depend on the `Beat resolution` parameter, namely the number of samples within a beat. We set the parameter to 16, due to the fact that these Jazz solos are live recordings of improvisations, which tend to be not as precise as studio recorded performances. By setting this parameter we quantize our MIDI files. Quantization is the studio-software process of transforming performed musical notes, which may have some imprecision due to expressive performance, to an underlying musical representation that eliminates the imprecision. The process results in notes being set on beats and on exact fractions of beats. We quantized the notes using a precision of 16 ticks per beat, which corresponds to a sixty-fourth note.

3.2.2 Piano Dataset

For our polyphonic music generation we used the same tokens we used when encoding the Jazz Dataset but with a few modifications.
Firstly we added two new token types:

- Chord, a token that indicates the presence of a chord at a certain time step. By using this we enable our model to generate polyphonic music.
- Tempo, a token that specifies the current tempo, the speed or pace of a given piece. This allows to train a model to predict tempo changes, which are present in our Piano dataset. Tempo is defined in Beats per Minute (BPM). We used the default parameters, with 32 tempo bins and a tempo range from 40 tom 250.

Then we changed the `Beat resolution` to 8, due to the fact that the Piano MIDIs are not recorded from live performances.

3.2.3 Byte Pair Encoding (BPE)

BPE is a compression technique that originally combines the most recurrent byte pairs in a corpus. In the context of tokens, it allows to combine the most recurrent token successions by replacing them with a new created symbol (token). This naturally increases the size of the vocabulary, while reducing the overall sequence length. Today BPE is used to build almost all tokenizations of natural language, as it allows to encode rare words and segmenting unknown or composed words as sequences of sub-word units. In the case of symbolic music, it has been shown to improve the performance of Transformers models while helping them to learn more isotropic embedding representations [3].
MidiTok implements BPE in its tokenizers with the `learn_bpe()` and `apply_bpe_to_dataset()` functions.

3.3 Dataset creation

In order to feed the tokens to the Neural Network we had to perform some pre-processing. We split the dataset into three different parts:

- Training set, used to train the network
- Validation set, used to stop the training before the network starts to overfit
- Test set, used to test the trained model on never seen before data

All three sets underwent the same processing.
In order to make the model generate new music we need to train it to predict the next note in a sequence. We transformed each song into a series of sequences of size $seq\_length + 1$, each sequence being obtained by shifting the window by one. For each input sequence, the corresponding targets contain the same length of tokens shifted one token to the right. We used a sequence length of 64. Then we applied the following functions to the generated sequences:

- `shuffle()`, used to randomly shuffle the sequences. This is a best practice to prevent the model from learning sequences based on their ordering
- `batch()`, used to combine consecutive elements of the dataset into batches. Each batch corresponds to a training step of the neural network
- `cache()`, used to cache the elements of the dataset in memory
- `prefetch()`, used in order to allow later elements to be prepared while the current element is being processed. This often improves latency and throughput

3.4 Recurrent Neural Network model

Recurrent Neural Networks (RNNs) are well suited for music generation for several reasons:

- Capability of processing sequences of data, which is a key requirement for music generation. Music is typically represented as a sequence of notes or events over time, and an RNN can be trained to predict the next note or event in the sequence based on the previous ones.
- Memory mechanism that allows them to maintain information about the sequence they have processed so far, allowing them to generate longer pieces of music that are coherent and have a clear structure.
- Can be trained on large datasets of existing music, allowing them to learn the statistical patterns and relationships that are present in real-world music. This makes it possible for an RNN to generate new music that is similar in style and structure to existing music.

Overall, RNNs' ability to process sequences of data, maintain information about the sequence, and learn from large datasets make them a powerful tool for music generation.

3.4.1 Architecture

The RNN takes a sequence of inputs and processes them one at a time. The input is fed into the network at each time step, and the hidden state of the network is updated based on the input and the previous hidden state. The hidden state is then used to generate the output for that time step. The key characteristic of an RNN is the use of a hidden state that is passed from one time step to the next. This allows the network to retain information from the previous time steps, which is important for processing sequences of data. There are different types of RNNs, such as Vanilla RNNs, LSTMs (Long Short-Term Memory), and GRUs (Gated Recurrent Units). When training a vanilla RNN using back-propagation, the long-term gradients which are back-propagated can "vanish" (that is, they can tend to zero) or "explode" (that is, they can tend to infinity), because of the computations involved in the process, which use finite-precision numbers. RNNs using LSTM units partially solve the vanishing gradient problem, because LSTM units allow gradients to also flow unchanged. For this reason we decided to use LSTMs in our project.

3.4.2 Model

The following operations were performed on the jazz dataset without Byte Pair Encoding. The exact same steps can be also applied to Jazz dataset with Byte Pair Encoding or to the piano dataset.
Our base model is composed of three layers:

- `Embedding`: a trainable lookup table that will map each token to a vector with `embedding_dim` dimensions;
- `LSTM`: a LSTM with size `rnn_units`
- `Dense`: the output layer, with `vocab_size` outputs. It outputs one logit for each token in the vocabulary. Each logit is the log-likelihood of each token according to the model.

We use `SparseCategoricalCrossentropy` as loss function, which computes the cross entropy loss between the labels and predictions using the logits.
We use the `Adam` optimizer. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. According to Kingma et al., 2014 [6] the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".

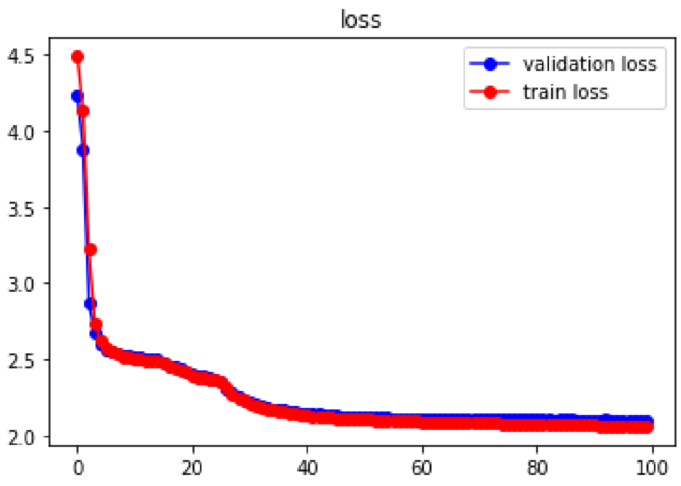
3.4.3 Improving model fit

Let us define a simple model with the following hyperparameters:

- `embedding_dimension = 64`
- `lstm_units = 64`
- `lstm_layers = 1`
- `learning_rate = 0.001`

After training the model for 100 epochs we get the the loss curve shown in Figure 3

Figure 3: Underfitting loss curve



The model is underfitting, which means that it is not complex enough to capture the patterns in the training data. To achieve the perfect fit, we must first overfit. Therefore our goal is to achieve a model that shows some generalization power and that is able to overfit. Once we have such a model, we will focus on refining generalization by fighting overfitting.

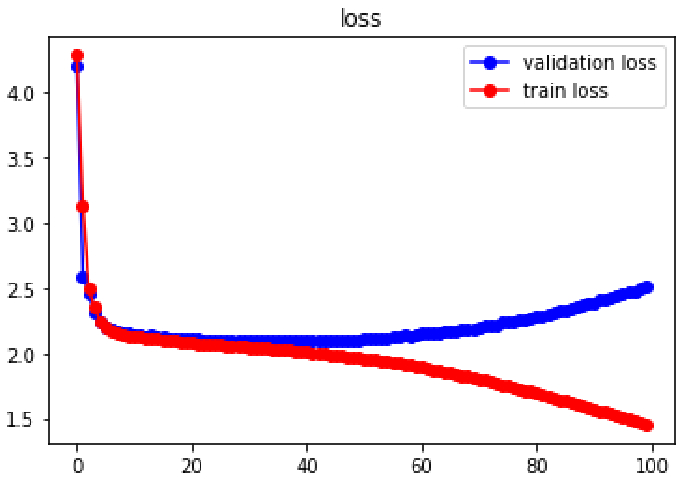
3.4.4 Improving generalization

Overfitting happens when a very complex statistical model fits the observed data because it has too many parameters relative to the number of observations.
Let us define a more complex model with the following hyperparameters:

- `embedding_dimension = 64`
- `lstm_units = 512`
- `lstm_layers = 2`
- `learning_rate = 0.001`

We trained the model for 100 epochs, obtaining the loss curve shown in Figure 4
After 30 epochs the model starts to overfit, as the validation error increases (positive slope) while the training error steadily decreases (negative slope)

Figure 4: Overfitting loss curve



3.4.5 Early stopping

In deep learning, we always use models that are vastly overparameterized. This is not an issue, because we will always interrupt training long before we have reached the minimum possible training loss. Finding the exact point during training where we have reached the most generalizable fit is one of the most effective things we can do to improve generalization. We typically do this with an `EarlyStopping` callback, which will interrupt training as soon as validation metrics have stopped improving, while remembering the best known model state. We implemented EarlyStopping with a *patience* of 5, which means that the callback will stop the training when there is no improvement in the validation loss for five consecutive epochs.

3.4.6 Dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training. The *dropout rate* is the fraction of the features that are zeroed out. The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that are not significant, which the model will start memorizing if no noise is present.
In order to implement Dropout in our model we add a `Dropout` layer between the last LSTM layer and the Dense layer.

3.4.7 Recurrent Dropout

It has long been known that applying dropout before a recurrent layer hinders learning rather than helping with regularization. Yarin Gal [4] determined the proper way to use dropout with a recurrent network: the same dropout mask (the same pattern of dropped units) should be applied at every timestep, instead of using a dropout mask that varies randomly from timestep to timestep. Furthermore, in order to regularize the representations formed by the recurrent gates of layers such as GRU and LSTM, a temporally constant dropout mask should be applied to the inner recurrent activations of the layer (a recurrent dropout mask). Using the same dropout mask at every timestep allows the network to properly propagate its learning error through time; a temporally random dropout mask would disrupt this error signal and be harmful to the learning process.
In order to add Recurrent Dropout to our recurrent layers we used the `recurrent_dropout` argument, specifying the dropout rate of the recurrent units.

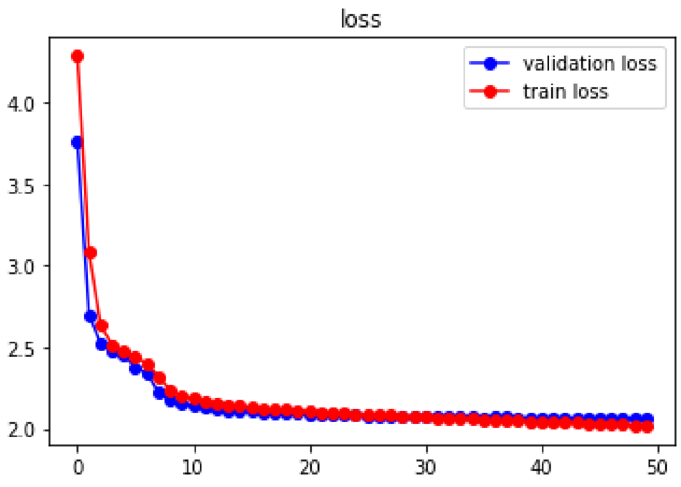
After applying these three techniques we trained the model for 100 epochs, obtaining the loss curve shown in figure 5. The model starts overfitting later and the training is stopped at epoch 50 using `EarlyStopping`.

We trained the model on the Jazz dataset with Byte Pair Encoding. We then evaluated the trained model on the test dataset, obtaining a loss of 3.62.

3.4.8 Hyperparameters search

Having dealt with overfitting we can perform the hyperparameters search. The following operations were performed on the Jazz dataset with Byte Pair Encoding. We only applied hyperparameter search on the jazz dataset due to the Google Colab constraints on computing resources and computing time. The procedure for

Figure 5: Loss curve



the Piano dataset would be the same.

Our RNN model has the following hyperparameters:

- `embedding_dimension`, the output dimension of the Embedding layer. Possible values are 64 or 128.
- `lstm_units`, the number of units of our recurrent layers. Possible values are 256, 512 or 1024.
- `lstm_layers`, the number of recurrent layers in the model. Possible values range from 1 to 3.
- `learning_rate`, the learning rate of the Adam optimizer. Possible values are 0.01 or 0.001.
- `dropout`, the dropout rate of the Dropout layer. Possible values are 0.2, 0.5 or 0.8.
- `recurrent_dropout`, the dropout rate of the recurrent layers. Possible values are 0.2, 0.5 or 0.8.

We used the `Hyperband Tuner`, a tuning algorithm developed by Li, Lisha, and Kevin Jamieson [7] that uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model. The algorithm trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round. Hyperband determines the number of models to train in a bracket by computing $1 + \log_{factor} max\_epochs$ and rounding it up to the nearest integer. We set `max_epochs` to 100 and `factor` to 3.
The search lasted 8 hours, and gave the following best hyperparameters for the Jazz dataset:

- `embedding_dimension = 64`
- `lstm_units = 512`
- `lstm_layers = 1`
- `learning_rate = 0.01`
- `dropout = 0.8`
- `recurrent_dropout = 0.5`

The loss on the test data for the tuned model was 3.57, lower than the loss computed on the non-tuned model.

3.5 Transformers

Transformers are well-suited for music generation due to several factors:

- Ability to handle long-term dependencies: Transformers have the ability to capture patterns in sequential data, making them ideal for modeling musical sequences.
- Ability to generate new content: Transformers can generate new content based on their training data, allowing for the creation of original music.

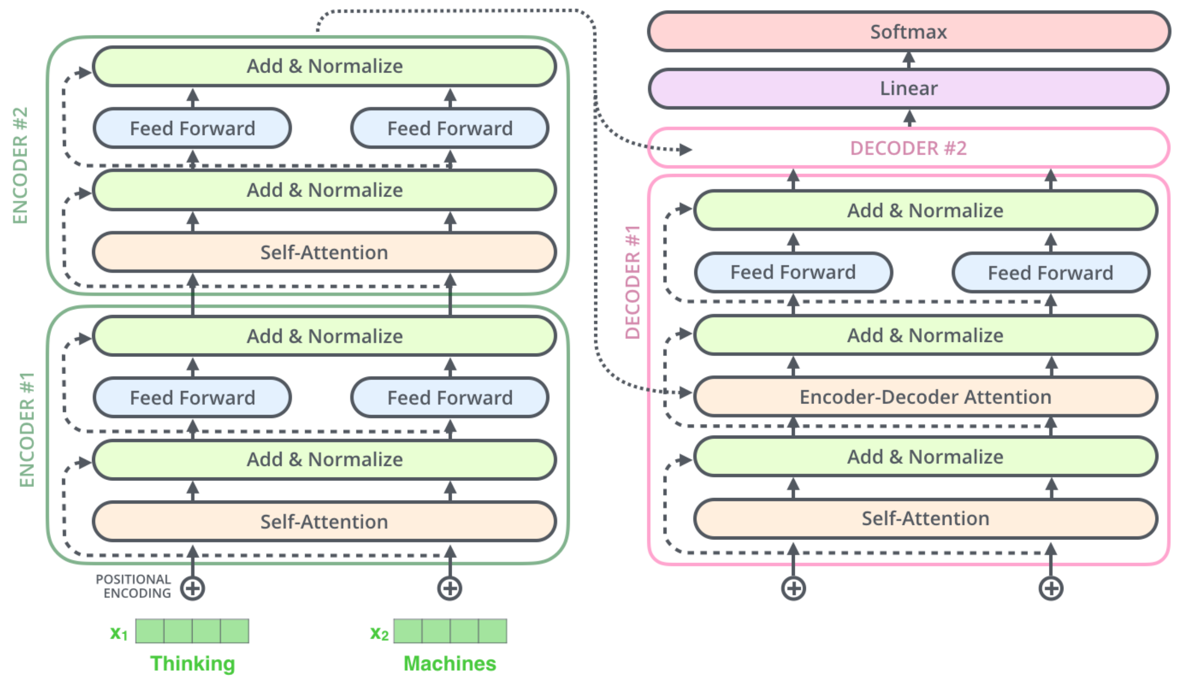


Figure 6: Transformer architecture.

- Efficient parallelization: Transformers can be trained efficiently in parallel, which speeds up the training process for music generation models.
- State-of-the-art performance: Transformer-based models have achieved state-of-the-art performance in various sequential data generation tasks, including music generation.

3.5.1 Architecture

The general architecture of a Transformer consists of an encoder and a decoder. The **encoder** processes the input sequence and generates a representation that summarizes the information in the sequence. The **decoder** then uses this representation to generate the output sequence. In music generation, there is no source sequence: you are just trying to predict the next tokens in the target sequence given past tokens, which we can do using a decoder-only architecture. Therefore, the encoder is not necessary. The decoder takes the input representation and generates new music by sampling from the learned distribution over possible musical sequences.

3.5.2 Model

Our Transformer is composed by two main component:

- **Positional Encoding**: Transformer uses Positional Encoding, which provides information about the relative position of each element in a sequence. In music generation, the relative position of each note in a musical sequence is crucial in order to capture the structure and flow of the music. Without it, the Transformer would not have any information about the order of the elements in the sequence, which could lead to the generation of musically inconsistent or repetitive outputs.
- **Decoder layer**: just like the RNN decoder, it reads tokens 0...N in the target sequence and tries to predict token N+1. Decoder layers are all identical in structure, but they do not share weights, composed by the following layers:
  - *Self Attention*: The purpose of self-attention is to modulate the representation of a token by using the representations of related tokens in the sequence. This produces context-aware token representations. In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions before the softmax step in the self-attention calculation.
  - *Layer Normalization*: thanks to the normalization layers present in each decoder layer we are able to normalize the output of each decoder layer. This helps stabilize the training process and prevent the outputs from becoming too large or too small. In music generation, the use of layer normalization is

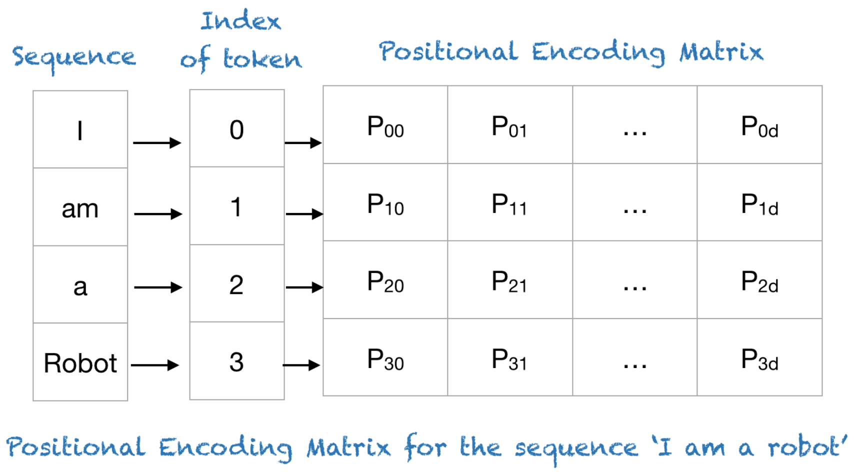


Figure 7: Example of position encoding applied to text processing.

important to ensure that the model is able to learn meaningful patterns in the input data without being overly influenced by the scale of the outputs. Layer normalization helps prevent the model from overfitting the training data and can lead to more robust and generalizable music generation performance.

– *Feed Forward*: the exact same feed-forward network is independently applied to each word position.

Note: In a transformer-based model, there is also the encoder-decoder attention layer, which is used to connect the decoder to the encoder and help the decoder generate more accurate outputs by attending to relevant information in the input sequence. However, in an decoder-only transformer, there is no encoder, so there is no need for an encoder-decoder attention layer. The decoder in this case attends only to the previously generated tokens in its own sequence, which is known as self-attention.

In order to improve the model's ability to generate more complex outputs we can use multiple decoder layers stacked on top of each other. Each decoder layer adds a level of abstraction to the representation learned by the model, allowing it to capture higher-level patterns in the input data, leading to better music generation performance. Additionally, having multiple decoder layers also helps to mitigate the vanishing gradients problem in deep networks by allowing the gradients to flow more easily through the network during backpropagation. This leads to a faster and more stable training.

3.5.3 Improving model fit

Let us define a simple Transformer with the following hyperparameters:

- `embedding_dimension = 64`
- `latent_dim = 32`
- `num_heads = 1`
- `num_layers = 1`
- `learning_rate = 0.001`

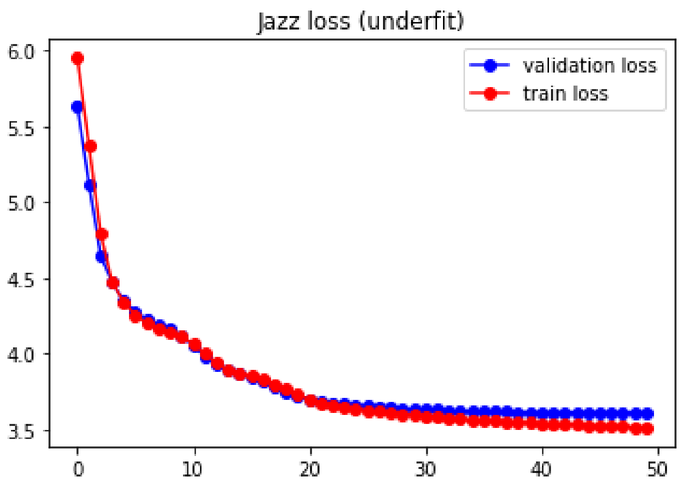
After training the model for 50 epochs we get the the loss curve shown in Figure 8. The model is underfitting, which means that the number of trainable parameters is not enough high to learn the patterns in the training data. To achieve a better fit, we must first overfit. Therefore our goal is to achieve a model that shows some generalization power and that is able to overfit. Once we have such a model, we will focus on refining generalization by fighting overfitting.

3.5.4 Improving generalization

Similar to what we have recently done with RNN model, we increase the architecture of the model by adopting the following hyperparameters:

- `embedding_dimension = 512`

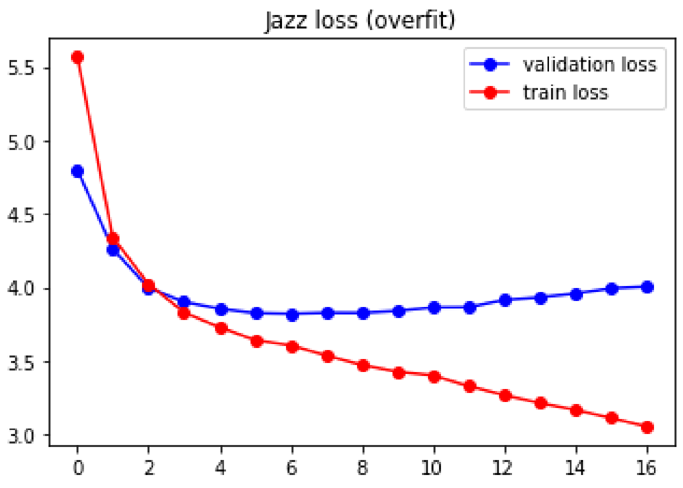
Figure 8: Underfitting loss curve



- `latent_dim = 512`
- `num_heads = 2`
- `num_layers = 2`
- `learning_rate = 0.001`

We trained the model for 50 epochs, obtaining the loss curves shown in Figure 9. After few epochs the model starts to overfit, as the validation error increases (positive slope) while the training error steadily decreases (negative slope).

Figure 9: Overfitting loss curve



3.5.5 Hyperparameters search

We applied the **Hyperband Tuner** in order to perform the hyperparameter search. We only defined a limited set of possible hyperparameters due to the fact that the process is computationally expensive.

- `embedding_dimension`, the output dimension of the Embedding layer. Possible values are 32, 64, 128 or 256.
- `latent_dim`, the number of units of our dense layers. Possible values are 32, 64, 128 or 256.
- `num_heads`, the number of attention heads. Possible values range from 1 to 6.
- `num_layers`, the number of decoder layer. Possible values range from 1 to 4.

- `learning_rate`, the learning rate of the Adam optimizer. Possible values are 0.01 or 0.001.

This operation lasted several hours and gave the following best hyperparameters for the Jazz dataset:

- `embedding_dimension = 32`
- `latent_dim = 32`
- `num_heads = 4`
- `num_layers = 1`
- `learning_rate = 0.01`

The loss on the test data for the tuned model was 3.66, lower than the loss computed on the non-tuned model.

3.6 Music generation

Both the models behave similarly during the inference phase. In this phase we use the learned models with tuned hyperparameters to generate new music based on a short prompt that has never been seen by the model. In order to do this we take a short sequence from our test dataset.
At each step the model outputs one logit (the log-likelihood of the token) for each token in the vocabulary. A naive approach is *greedy sampling*, consisting of always choosing the most likely next character. But such an approach results in repetitive and predictable outputs. A better way to do this would be with *stochastic sampling*, where we introduce randomness in the sampling process by sampling from the probability distribution for the next token. This allows even unlikely tokens to be sampled some of the time, generating more interesting sequences and sometimes showing creativity by coming up with new, realistic-sounding melodies that did not occur in the training data. In order to control the amount of stochasticity in the sampling process, we used a parameter called the **softmax temperature**, which characterizes the entropy of the probability distribution used for sampling. A new probability distribution is computed dividing the original distribution by the temperature value. Higher temperatures result in sampling distributions of higher entropy that will generate more surprising and unstructured generated data, whereas a lower temperature will result in less randomness and much more predictable generated data.

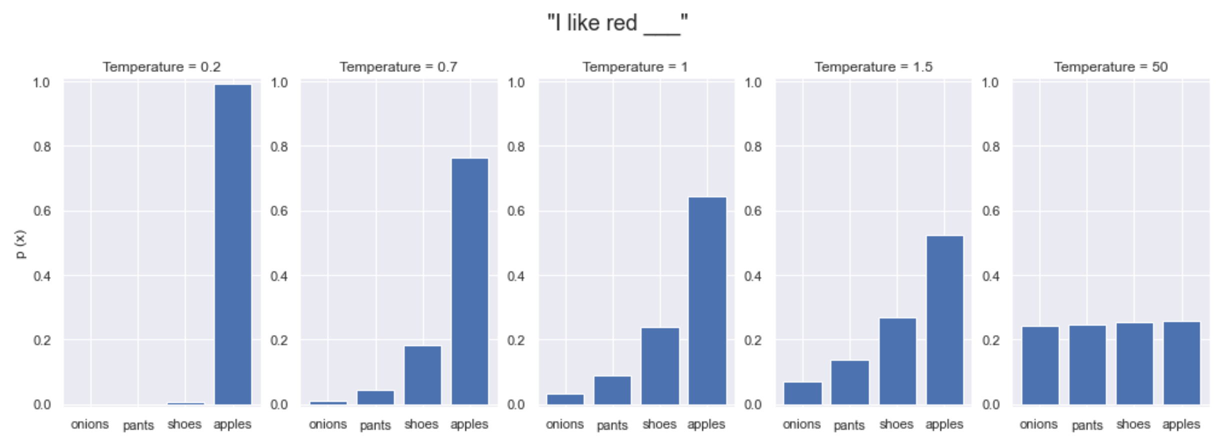


Figure 10: Effect of temperature on the probability distribution

4 Conclusion

Our project is composed of two main parts: monophonic and polyphonic music generation. For the monophonic part we were able to perform hyperparameter optimization for both the LSTM model and the Transformer model. This was possible due to several reasons: *(i)* the Jazz dataset contained a small number of songs, *(ii)* the songs only contained the melodic part, *(iii)* the songs were very short. This allowed us to perform the tuning using the limited resources at our disposal. The loss value obtained by the two tuned models is comparable as we used the same tokenization and the same loss function. The LSTM model obtained a loss of 3.57, while the Tranformer 3.68. This is probably due to the fact that a Transformer model typically requires more training data in order for the model to perform better. As for the polyphonic part, it was not possible to perform hyperparameter tuning for the Piano dataset due to the limited GPU resources offered by Google Colab. After 8 hours of tuning Colab would automatically interrupt our notebooks. Nonetheless we still wanted to try polyphonic music generation, so we only trained two non-tuned models.

Here are some aspects of our project that need further attention:

- the models were trained on a little data. In order to overcome this limitation we could use some data augmentation techniques on the original MIDI files. For example we could shift all the pitch values up or down, obtaining the same songs in different keys. MidiTok provided a way to agment the data, but we could not use it due to resource constraints.
- the training process is very slow. Even if we had a higher number of GPUs, recurrent dropout isn't supported by the LSTM and GRU cuDNN kernels, so adding it to our layers forces the runtime to fall back to the regular TensorFlow implementation, which is generally two to five times slower on GPU.
- the hyperparameter optimization was performed on a limited number of hyperparameter values. This is also due to the limited resources at our disposal, as adding even a single hyperparameter value causes the number of possible combinations to grows exponentially.
- we did not compare the different tokenizers offered by MidiTok. We chose TSD as it was the tokenizer that contained all we needed, but some other tokenizer could have performed better on the training data.
- the length of the training sequence is a hyperparameter, but we did not include it in the hyperparameter optimization, due to the resource contraints.
- no objective metrics exist for evaluating generated music. Evaluating music generated by a neural network is a subjective process, so it is typically done using blind tests. A group of listeners hear a playlist that includes both music generated by the neural network and music created by human musicians in the same genre or style. They rate each song based on various factors, such as melody, rhythm, harmony, instrumentation, and overall emotional impact. Then the data is analyzed and the test is repeated with multiple groups of listeners. It is important to note that *quantitative metrics* for evaluating music generation are still largely experimental and subject to ongoing refinement and debate. While some metrics have been widely used and validated in the field, there is still much discussion around which metrics are most appropriate for evaluating different aspects of music generation, and how best to interpret and compare results across different studies. Additionally, some critics argue that focusing too heavily on quantitative metrics can lead to a narrow definition of musical quality and creativity, and may not fully capture the richness and complexity of human musical experience.

Overall the project has met the expectations discussed in the proposal phase, as we were able to generate pieces with some musically coherent moments. Unfortunately we were not able to implement a SeqGAN model due to the fact that GANs are well known to be difficult to train, especially with low computational resources. Futhermore, the paper we used as reference combines standard Deep Learning technologies with reinforcement learning technologies, which were out of the scope of this course.

References

[1] Chris Donahue, Huanru Henry Mao, Yiting Ethan Li, Garrison W. Cottrell, and Julian McAuley. Lakhnes: Improving multi-instrumental music generation with cross-domain pre-training, 2019.

[2] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment, 2017.

[3] Nathan Fradet, Jean-Pierre Briot, Fabien Chhel, Amal El Fallah Seghrouchni, and Nicolas Gutowski. Byte pair encoding for symbolic music, 2023.

[4] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks, 2015.

[5] Yu-Siang Huang and Yi-Hsuan Yang. Pop music transformer: Beat-based modeling and generation of expressive pop piano compositions, 2020.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[7] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. 2016.

[8] Jean-Pierre Briot Nathan Fradet. 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.

[9] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation, 2017.

[10] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017.