# IANNwTF Project Report

Bogdan Nikolaychuk, Benjamin Fricke

April 1, 2023

**Abstract**

While the advancements in deep image generation have echoed throughout most of society, advancements in deep music generation has received much less attention. The improvements in both fields were, for the most part, a consequence of applying the transformer architecture to these longstanding problems. It seems, however, as though the transformer architecture might be less successful for music than it is for image generation. For our project we were interested in both learning about this architecture, as well as in seeing for ourselves where the challenges for deep music generation lie.

## 1 Introduction

The transformer architecture was originally introduced in [7], and has since revolutionized the machine learning world. What was originally intended only as a tool for machine translation, has proven superior for almost all modeling tasks where sequential data is involved.
This, of course, also includes musical data, where major advancements have been made, as well. OpenAI's MuseNet [4] and more recently Google's MusicLM [1] are examples of transformer based text-to-music AI's that have heralded a new state of the art.
These advancements have been a major milestone in the long lasting challenge of developing artistically creative AI systems and perhaps let us glimpse the potential of AI-generated art awaiting us in the future.
For our project, we have chosen to build a simpler version of these systems: a straightforward generator, that does not respond to text prompts, but instead predicts the next tokens in a sequence of MIDI-based tokens.

## 2 Motivation

As mentioned, the development of the transformer architecture has been absolutely revolutionary to the field of artificial intelligence as a whole. This alone is reason enough to want to implement a transformer based network. As for the question of application: we wanted to pick a subject that was both interesting to us and the machine learning community and at the same time not overdone. Text generation has been the primary example for the power of transformers ever since their dawn and similarly so for image generation, which has been done very successfully in many different places. We also considered neural theorem proving, but chose music generation in the end, as the necessary data is much more readily available and as it is dearer to our hearts.

## 3 Related Literature

This section introduces the relevant literature for our project. We have worked with

1. Attention is All you Need
   which has first introduced the idea of the transformer architecture and as such is the main source we drew upon for our implementation.

2. MusicLM: Generating Music From Text
   which documents the most successful application of transformers to music generation, or perhaps

of artificial music generation in general. We have been inspired and motivated by their work and base a large part of our discussion on it.

3. LLaMA: Open and Efficient Foundation Language Models
that has introduced highly scaled down, yet in many cases more effective NLP transformers. Here we have been inspired to deviate in our hyperparameters and architecture from [7] and try to implement the new recommendations.

4. MIDITOK: A PYTHON PACKAGE FOR MIDI FILE TOKENIZATION
which presents a python library that is capable of tokenizing MIDI files into a format that is highly effective for transformer-based applications. We have therefore aptly used it for tokenization in our project.

5. MuseNet (Not a research paper, but an official article by OpenAI)
that has been another important milestone in artificial music generation and has provided useful information on data handling and implementation.

## 4 Methods

In essence, we have processed publicly available MIDI-data of several hundred classical music pieces into a format optimized for the transformer architecture i.e. into sequences of byte pair encoded (BPE-) tokens. The architecture itself was then optimized for the processing of MIDI data based on [1, 4] and also took some general inspiration from [6].
The first section will outline the details of the data preparation: from download to BPE-tokenization, while the second discusses how our transformer architecture learns to create classical music, based on our data. We will proceed without delving deep into details of the architecture itself, but rather focus on our deviations from what was originally proposed in [7].

### 4.1 Data Processing

We downloaded our data from: https://www.kaggle.com/datasets/soumikrakshit/classical-music-midi and https://github.com/bytedance/GiantMIDI-Piano They are publicly available data sets, the first one of which (from now on referred to as the 'small data set') contains 295 classical music pieces by most of the great historic composers in MIDI format and spans roughly 3 MB. The second one (the 'large data set') is over 100 times larger (!) spanning over 500 MB.
MIDI (.mid) files deviate from the usual music and audio formats (.mp3, .wav, etc.), in that they are not collections of wave functions, but rather simple instructions for a sound generating system (such as speakers, synthesizers, etc.). The main parameters for the generation of a particular sound are its pitch (i.e. frequency), velocity (i.e. intensity) and duration, as well as (temporal) position.
Why use this data format? One of the main issues in artificial music composition was that wave based audio data is too large. Data processing, as well as network training will take very long as compared to image and text generation, which is one of the reasons that there have been fewer breakthroughs in this field. MIDI data, on the other hand, uses very little memory, while still preserving most of the information about the sound of a piece. It can be thought of as a very efficient and handy abstraction of raw musical data. Moreover, it lends itself quite effectively to tokenization, as we shall see.
After having downloaded the files, the tokenization process began. MidiTok [2] is equipped with some very handy functions for the tokenization of MIDI files. We have first processed the .mid files into a JSON format, that already encoded all the contained information into tokens, using the standard tokenizer.tokenize_midi_dataset function, which we have set to be 'REMI' (this procedure is detailed in [3]), as it is the default and works well for our purposes. As parameters we have given the path to our MIDI collection, as well as the destination of the tokenized data, abstaining from any 'data augmentation offsets' (such as projecting all pitches into a certain range, or neutralizing velocity, etc.) that could have been used for further processing. We have also added start- and end-tokens.
Important to note is the structure of the newly processed .json files: they are built like a python dictionary with two keys: "tokens" and "programs". The token entry contains a sequence of 2500-5000 tokens on average, per piece. Each token was taken from a vocabulary of 512 entries (some of which are empty), and encodes a particular aspect of a MIDI file. To illustrate: there are 88 pitches that a

piano can produce, and therefore each pitch is encoded by a unique respective token. Similarly so for all other properties of a given MIDI file. For viewing and changing the encoding, there is a designated file named 'config.txt'.
On the other hand, "programs" contains information about the instrument that was used, which is important for the MIDI-player, but not for training the network, which is why we removed them later on. Almost all files were piano pieces, therefore little information was lost in any case.
To finish the processing stage we have used byte pair encoding, which is known to improve network performance across most domains. The respective functions were 'tokenizer.learn_bpe' with a vocabulary size of 512 and tokenizer.apply_bpe_to_dataset. The former learns the BPE, while the latter applies it to a data set. The last task left to do to make the data ready for tensorflow, was converting the JSON files into numpy arrays.
In this format, we have proceeded in a pretty standard way: we have taken the first 90% of the data for training and the remaining 10% for validation. We further subdivided both sets into sequences with a length of 4096, which we proceeded to shuffle. It is worth mentioning, that we did not cut the pieces in a way that there is no overlap between different songs. Sequences may begin in the middle of a certain song and end in the middle of another. Perhaps this is one reason for poor network performance.

## 4.2 Implementation of the Transformer

As mentioned before, the basic blueprint for the transformer architecture is spelled out in [7]. Here, we will focus on our deviations from it, staying general with our descriptions of the basic functionality of the transformer, rather than detailing what has already been explained multiple times elsewhere.

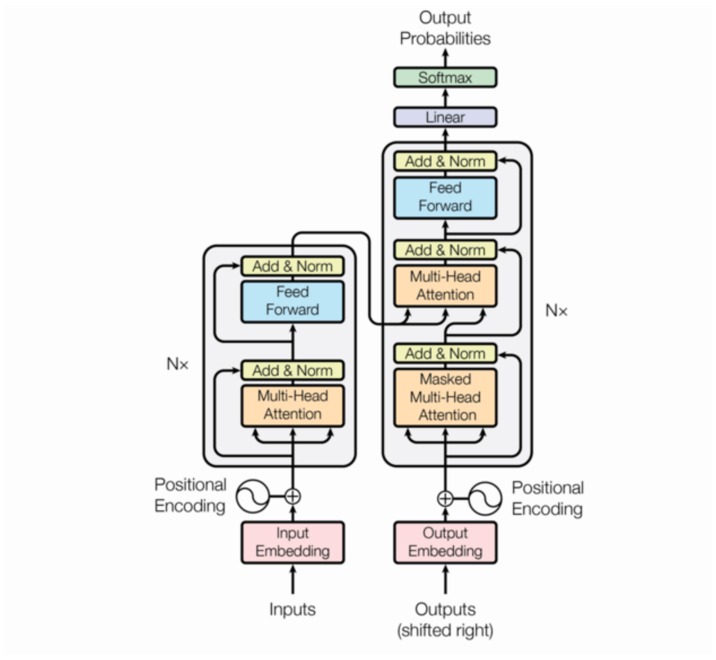

Figure 1: A schematic representation of the basic transformer architecture taken from [7]

### 4.2.1 General Training Procedure

The main idea behind how the transformer learns to create music (or model almost any sequential data, for that matter) is next-token-prediction. The transformer is fed thousands of examples of token-sequences and is evaluated on how well it predicts the next token in a given sequence. There is no need to have source and target data, as any given input can easily be manipulated into a target output. One simply duplicates a given sequence of tokens (whose length is a hyperparameter called 'seq_len') and shifts all the tokens one step to the left, removing the first element. This modified sequence functions as the target, whereas in the input sequence one simply removes the last element, for the two sequences to have matching indices for input-output-pairs. Now, for almost any practical use case it is not enough to simply predict the next token given the token immediately before it (as one would in a bi-gram model). Thus for each target with index i one considers not just the input at index i, but the sequence of all inputs with index $\leq$ i, all the way to the first example with index 0. Notice, that the length of these subsequences ranges from 1 all the way to 'seq_len'. So a single training sequence contains as many examples as it is long.

For our implementation we have chosen either a sequence length of 2048 or 4096. There is a trade-off between computational expenses and information density for a given training example. To have training proceed in reasonable timescales we have invested in more computational resources by purchasing 200 compute units for google colab.
The output of the network is a probability distribution over all possible tokens. The loss is computed using categorical cross-entropy, while gradient descent is then performed over the decoder block and final dense layer.

### 4.2.2 Generation of Music

After a successful training, the model is capable of generating piano music. The procedure is as follows: Generation begins by the model receiving a (possibly empty) input sequence of tokens and predicting the most likely next token, which it then outputs. Or rather: it predicts a probability distribution over all tokens, given the previous sequence of tokens. And instead of always (greedily) choosing the token with the highest probability, it samples from this distribution. This is very important, as the output is not just more diverse in principle, but also contains more than just a single note being repeated identically over and over again (as was the case in our first song "sample_song_greedy.mid").

### 4.2.3 Layer Normalization

We have deviated substantially from [7] and instead implemented what was proposed in [6]: namely to add normalization layers *before* feeding the inputs into the multi-headed attention blocks, as opposed to applying normalization after.

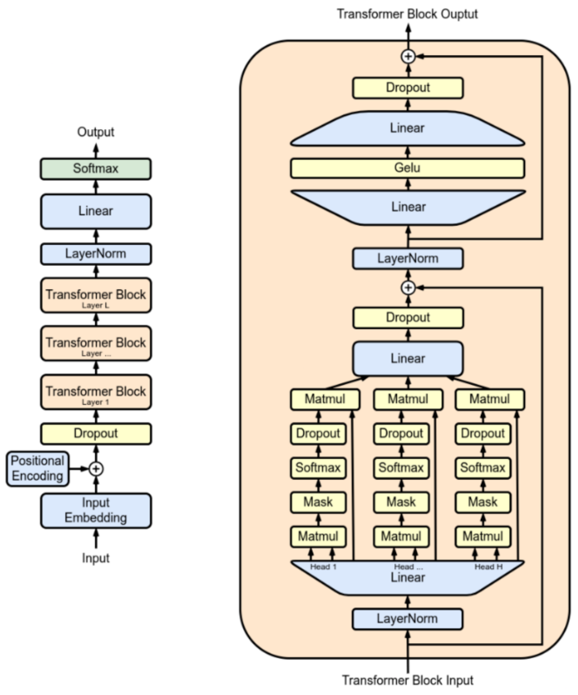

Figure 2: A schematic representation of our implementation. Gelu should be replaced by SwiGlu. Graphic taken from Wikipedia.

### 4.2.4 Activation Functions

As for the activation function used in the feed-forward-networks of the decoder layer, we have chosen SwiGlu instead of ReLU, which was detailed and proven to be superior for transformers in [5].

## 5 Results

We present the following results for our first model, which was trained on the small data set (spanning roughly 3 MB):

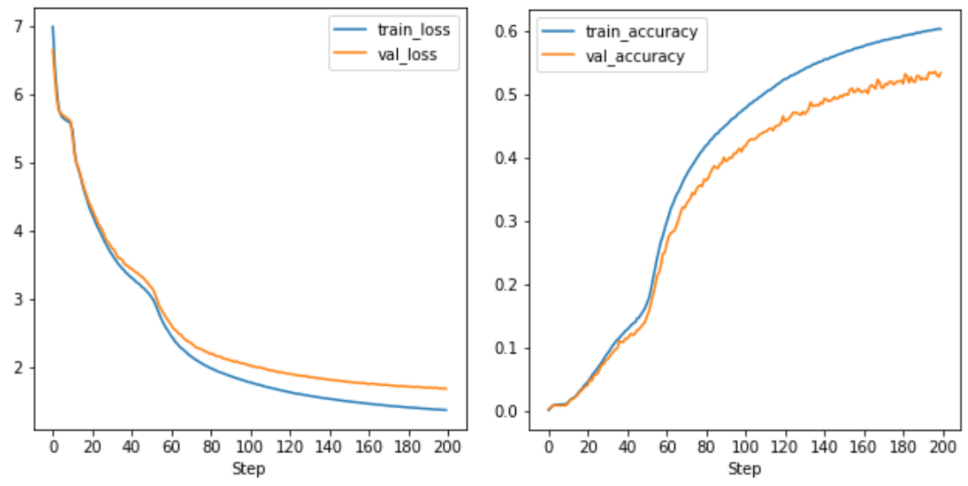

Figure 3: Loss and Accuracy for our first model on the small data set.

We have trained this model for 200 epochs with 66 steps each. The hyperparameters are:
d_model = 96 (which is also the embedding size)
dff = 384 (the dimensionality of the inner feedforward layer)
n_heads = 6 (number of heads in the multihead attention layer)
n_layers = 6 (number of layers)
dropout_rate = 0.1 (dropout rate)
epochs = 200 (number of epochs)
seq_length = 4096 (length of the sequence)
batch_size = 6 (batch size)

As can be seen from the figures, there has been notable overfitting. The validation loss has barely gotten under 2, while the training loss is under 1.5. Similarly for the accuracy, which on the training set has reached 60% and for validation has only gotten a little over 50%.
The second model was trained for 100 epochs with 200 steps each.
The hyperparameters are:
d_model = 128
dff = 512
n_heads = 8
n_layers = 6
dropout_rate = 0.1
epochs = 100
seq_length = 4096
batch_size = 4

This model, which was larger in most respects (d_model = 128 instead of 96, dff = 512 instead of 384, n_heads = 8 instead of 6) has overfitted even more than the previous model. This is in part due to the model's size, which allows the model to memorize large parts of the training set, and also because it has had 20000 training steps instead of (roughly) 13200. The figures indicate quite clearly that there

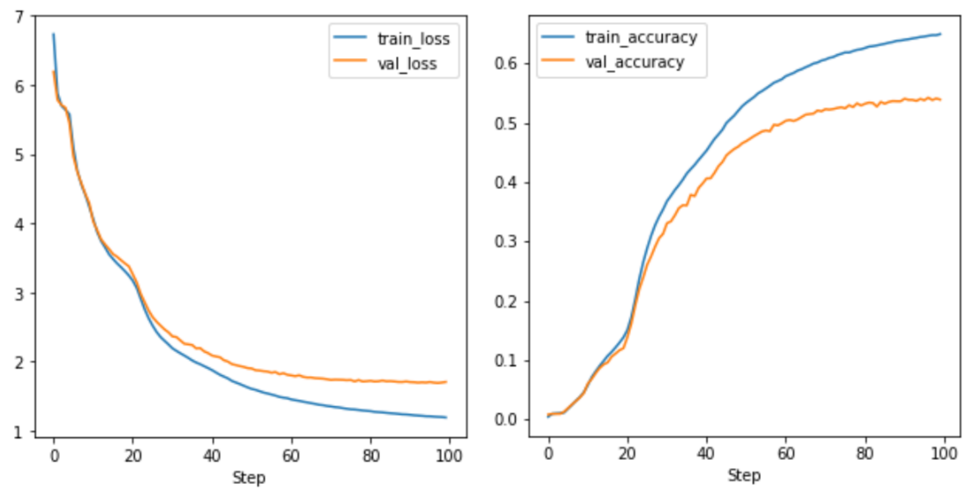

Figure 4: Loss and Accuracy for our second model on the small data set.

has been barely any improvement on the validation set in the last 30 epochs, while the training set was still merrily being improved upon.
In fact, what is even more sobering, is that the second network has ended training with a slightly higher loss than the first (1.7 vs. 1.68), although it did reach a minutely higher accuracy (0.54 vs 0.53).

The following plots detail the loss and accuracy of our final model, which was the best performing.

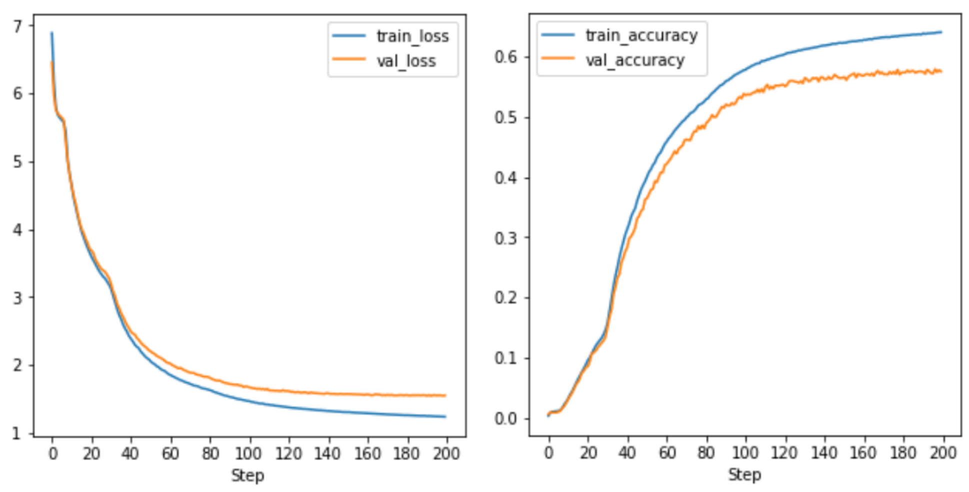

Figure 5: Loss and Accuracy for our third model on the small data set.

There were 100 steps per epoch. The hyperparameters are:
d_model = 96
dff = 384
n_heads = 6
n_layers = 6
dropout_rate = 0.1
epochs = 200
seq_length = 2048
batch_size = 6

On all accounts this model performs best: there is visibly less overfitting, the final accuracy is well

above the other models at 0.57, while the loss is also by far the lowest with 1.55. What's more is that this model has had the shortest total training duration of merely 10000 steps.
On all hyperparameters this network is identical to the first. The two key differences are that it has a halved sequence length and a slightly higher learning rate.

Finally we present our one attempt at the large data set, which spans roughly 500 MB:

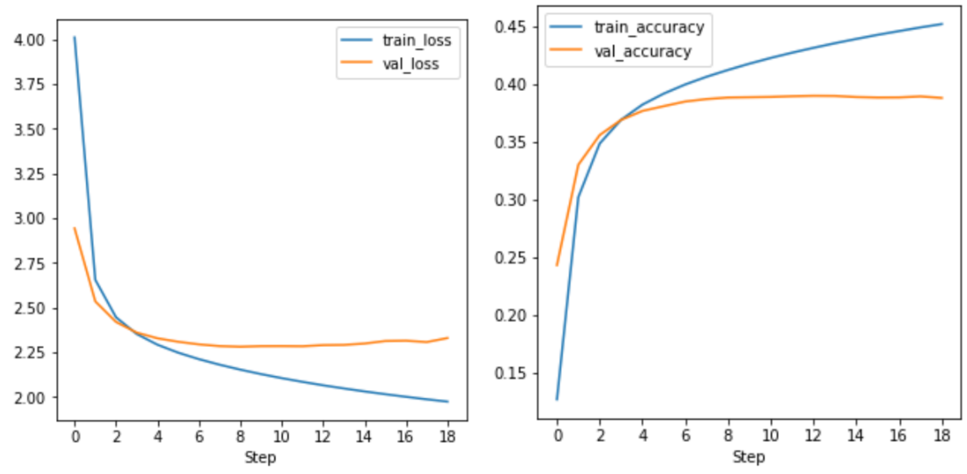

Figure 6: Loss and Accuracy for our first and only model on the large data set.

The hyperparameters of this last model are:
d_model = 768
dff = 3072
n_heads = 12
n_layers = 12
dropout_rate = 0.1
epochs = 20
seq_length = 2048
batch_size = 4

It is by far the largest model in every dimension (and thus training it also ate up most of our compute units).
As is obvious from the graphs, the network performance was quite poor. After the first 5 iterations the loss stopped decreasing and instead even rose a little, whereas the accuracy stayed nearly fixed in place. And also in relation to the other networks the loss was quite high and the accuracy quite low. This is despite the network having the same exact hyperparameters as the previous one (which performed best). This indicates that for larger and more diverse music sets a much larger network is needed. This was sadly not realizable with the computational capacities that we had.

## 6 Discussion

We shall conclude this report with a general discussion of artificial music generation, as well as our personal experiences in doing this project.

### 6.1 Challenges for Artificial Music Generation

One of the core issues in artificial music generation is that sound data is either abstracted (such as in the .mid format) or very large, as opposed to text or images. Even more so for high quality audio data. This poses some obvious but nonetheless serious computational issues. Moreover, there is comparatively little data available. From our personal experience we can attest to the rarity of midi data sets. And while there is a lot of freely available music on the internet, there are still vastly more images and

text, and crucially: there is very little labeled music. This was not a problem for our project, since we did not intend a connection to text-prompts, but it is indeed a serious issue if one does attempt this. Google has publicly released the data set, which was used for training MusicLM [1] - the state of the art model for music generation - and the data set spans merely 5,5k labeled songs. This is tiny, compared to the vast databases of labeled images. And this is not just a problem of little work having been put in: labeling of music data must be performed by trained musicians, lest one wants to allow poor labels to seep into one's models. This stands in stark contrast to image labeling, which entirely untrained people can do.
Aside from the several data-based issues, neural production of high quality/resolution sound has been and is still an issue even in systems that are trained on very high quality, non-abstracted data. Regardless of whether the sounds are music or speech or anything else. It is possible to overcome this issue to a sufficient extent, but that, again, takes more work and resources.
Lastly, the temporal aspect of musical data makes labels less efficient. In one and the same labeled song (or labeled sequence) there may be many aspects that do not fit the label, or are not captured by the label. While this is one more point that was not directly relevant to us, it poses yet another issue to artificial music generation.

### 6.2 Personal Conclusions

This project was quite challenging and rewarding. It took us 4 days to even understand how a basic transformer works and to decide on a subject. The implementation took another week and a half or so and was at many times quite frustrating. Our first attempts, which were largely based on code that others had written, failed miserably. And building it from scratch, we saw ourselves confronted with many ugly error messages from tensorflow that probably even the most sophisticated ML engineers would have had trouble understanding.
In the end, however, we produced something that worked! Even though the first piece was hardly a Mozart sonata, it was still pretty epic to listen to the sounds of the network that we have developed. It was at this stage that we discovered a much larger data set (20x larger, roughly), and trained a new model on it. We expected some appropriate improvements, but got very little, which was again a bitter pill to swallow.
Ultimately, though, with a lot of experimentation and finetuning (and google compute units) we managed to produce a model that we are fairly proud of! (Even though we prefer to listen to human music for the time being.)

## References

[1] Andrea Agostinelli, Timo I Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, et al. Musiclm: Generating music from text. *arXiv preprint arXiv:2301.11325*, 2023.

[2] Nathan Fradet, Jean-Pierre Briot, Fabien Chhel, Amal El Fallah-Seghrouchni, and Nicolas Gutowski. Miditok: A python package for midi file tokenization. In *22nd International Society for Music Information Retrieval Conference*, 2021.

[3] Yu-Siang Huang and Yi-Hsuan Yang. Pop music transformer: Beat-based modeling and generation of expressive pop piano compositions. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1180–1188, 2020.

[4] Christine Payne. Musenet. *OpenAI, openai.com/blog/musenet*, 2019.

[5] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[6] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.