TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Siim Pender 213110 IAIB

Generating Maps for Rhythm Games Using Machine Learning

Bachelor's Thesis

Supervisor: Gert Kanter
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Siim Pender 213110 IAIB

Rütmimängudele tasemete automaatne loomine masinõppemudeliga

Bakalaureusetöö

Juhendaja: Gert Kanter
PhD

Tallinn 2025

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis and that this thesis has not been presented for examination or submitted for defense anywhere else. All used materials, references to the literature, and work of others have been cited.

Author: Siim Pender

06.06.2025

Abstract

Rhythm games are video games in which a significant part of the gameplay is based on perceiving music and rhythm. In most rhythm games players must press buttons or perform other actions in response to visual or audio cues.

Songs are added to these games by manually timing the audio and placing notes to create a corresponding map. These maps are created entirely by hand with minimal tooling, making the process long and tedious.

This work proposes a machine learning approach to automatically generate such maps for any user-selected song. Over 100,000 unique beatmaps were collected from the game osu! to train a transformer-based model with 240 million parameters. The resulting model is capable of generating fun and engaging maps for osu!, allowing players to enjoy their favorite songs in the game.

The thesis is in English and contains 30 pages of text, 5 chapters, 23 figures, 2 tables.

Annotatsioon
Rütmimängudele tasemete automaatne loomine masinõppemudeliga

Rütmimängud on videomängud, milles oluline osa mängust põhineb muusika ja rütmi tajumisel. Rütmimängus peab mängija vajutama nuppe või sooritama muid tegevusi vastavalt visuaalsetele või helilistele vihjetele.

Uute lugude lisamine rütmimängudesse on pikk ja oskustnõudev protsess. Loo lisamiseks tuleb sellele luua kaardistus (tase). Kaardistuse jaoks peab käsitsi millisekundi täpsusega määrama loo alguse, tuvastama tempo ja paigutama mänguelemendid. Selle abistamiseks on saadaval väga vähe tööriistu.

Käesolevas töös pakutakse välja masinõppel põhinev lähenemine, mis võimaldab kasutaja valitud loo põhjal automaatselt selliseid kaardistusi luua. Mängust osu! koguti üle 100 000 unikaalse kaardistuse ning nende põhjal treeniti 240 miljoni parameetriga transformer-arhitektuuriga mudelit. Valminud mudel suudab luua lõbusaid ja kaasahaaravaid kaardistusi osu! jaoks, võimaldades mängijatel selles mängus oma lemmiklaule mängida.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 30 leheküljel, 5 peatükki, 23 joonist, 2 tabelit.

List of abbreviations and terms

GPU	Graphics Processing Unit
VRAM	Video Random Access Memory
LLM	Large Language Model
RoPE	Rotary Positional Embedding
SiLU	Sigmoid Linear Unit
ReLU	Rectified Linear Unit
RMS	Root Mean Squared
BPM	Beats Per Minute

Table of contents

List of figures

List of tables

1 Introduction

Rhythm games are video games where a significant part of the gameplay is based on perceiving music and rhythm. In most rhythm games, players must press buttons or perform other actions in response to visual or audio cues. These games test the player's sense of timing, reaction speed, and hand-eye coordination.

In most rhythm games, multiple maps with different difficulty levels or styles are created for a song. Creating these maps is a very long, tedious, and skill-demanding process. The creator of the map must manually time the song, determining its tempo and beat timings with millisecond accuracy. Next, the creator assigns the timing, placement, and often additional attributes for each individual game element, depending on the game. There are very few tools available to assist with this process because map creation is highly creative.

This work proposes a machine learning model based solution to this problem. The model takes a song as input and generates a map of user-selected difficulty and style as output.

Some models have already been created that try to do exactly this [1]. However, these models fail to meet the bare minimum of quality expectations. This is mostly because these models are very small and are trained on very small datasets. For example the BeatLearning [1] model only has around 200k parameters. In this work a 240M parameter model will be trained on over 100,000 unique maps. The model trained in this work will be able to overcome the shortcomings of those smaller models.

This work focuses on the rhythm game osu!. osu! was originally published in 2007 by its lead developer ppy. osu!'s gameplay primarily involves clicking circles to a song or beat. osu! is the most popular rhythm game in the world, is open source and has an extensive list of freely available beatmaps.

2 Background

2.1 osu! Gameplay

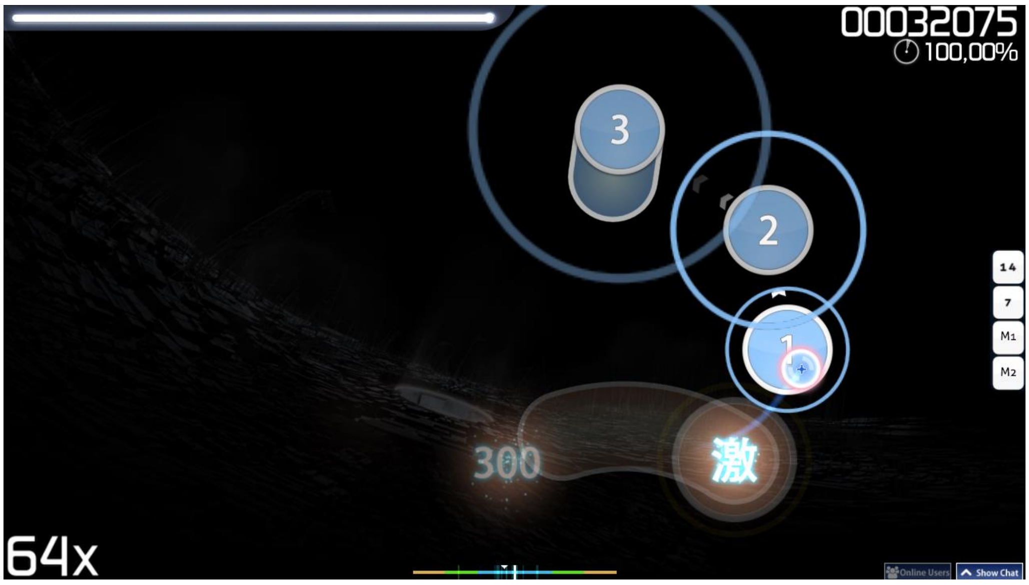

Figure 1. osu! Gameplay [2]

osu! has several game modes (osu!mania, osu!taiko and others) but the most popular is osu!standard (or just osu!). The players have to place their cursor on top of the circles and click them to the beat. While the rules are very simple this game can be very difficult and expressive. Mappers have come up with very creative ways to express different songs and rhythms in the game.

2.1.1 Hit Circle

The hit circle is the simplest hit object in osu!standard. Hit circles are numbered to indicate their order in a combo. Each hit circle is surrounded by an approach circle that shrinks around it. Once the approach circle overlaps the hit circle, the player must click or tap the hit circle to earn score [3].

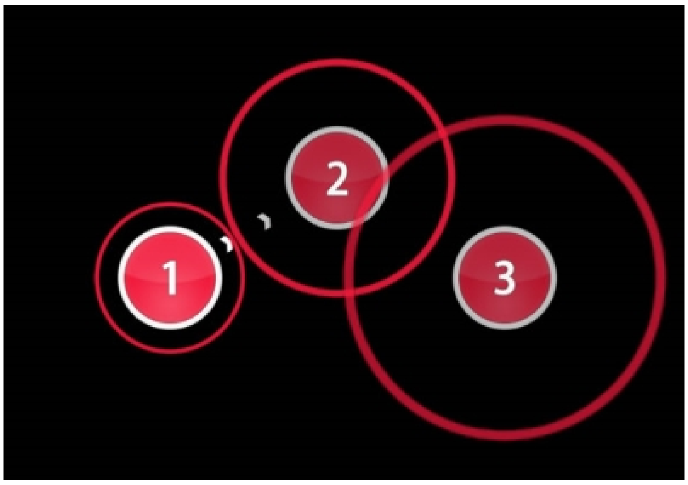

Figure 2. Hit circle [3]

2.1.2 Slider

A slider is a hit object in osu!, which consists of a slider head, a slider body, and a slider tail [4]. Similarly to hit circles, once the approach circle reaches the slider head's border the player must tap or click the beginning of the slider and then follow a moving target along the track until the slider tail is reached.

Sliders may also repeat multiple times, in which case the user has to follow the slider back along the same path.

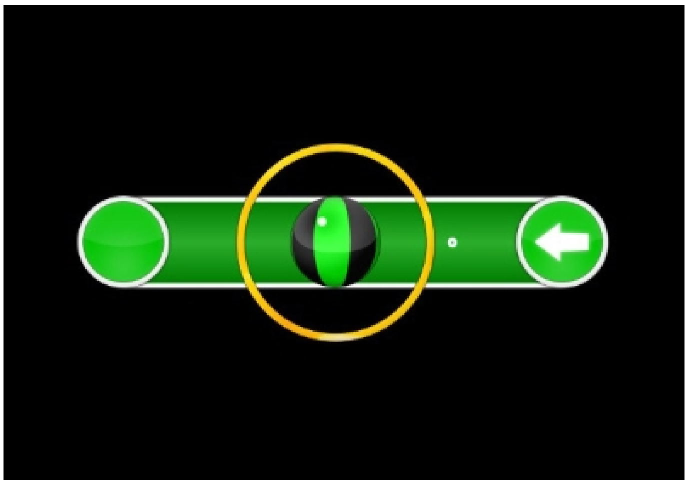

Figure 3. Slider [4]

Curve Types

Sliders curves are defined with (invisible to the player) points in the play grid. Different types of sliders exist that form different paths between these points. Linear sliders draw a straight line between two points in the play grid. Perfect sliders draw a perfect 3 point arc between points in the play grid. Bézier sliders draw a Bézier curve between any number of points in the play grid.

Slider Velocity

Sliders move with a constant velocity from the start of the slider to the end. Each slider can have its own individual velocity. There is no explicit indicator for the velocity of an upcoming slider and players must guess, based on the song, mapping style and other factors, the approximate velocity of a slider to hit it correctly.

2.1.3 Spinner

A spinner is an object in osu! that takes up the whole play field. Players must simply hold down a button and move their cursor in a circular motion until a certain threshold is reached. Spinners are often used as a way to help the player relax at the end of a song.

2.2 Mapping

Mapping refers to the process of creating new difficulties, levels or "mappings" for songs. The creator of a map, or a "mapper", chooses a song they would like to map, they time it, and place hit objects at the appropriate locations and timings to create an interesting and fun-to-play beatmap.

2.2.1 Hit Sounds

Hit sounds are sounds that are played when a player successfully clicks a hit object or performs other actions. Hit sounds are used to give auditory feedback to players to help them judge their accuracy in relation to the song.

Hit sounds consists of a default sample called a "hit normal", and any combination of whistle, finish, or clap sample additions. Different combinations and sequences of hit sounds are often used to better fit the rhythm of the song.

2.2.2 Combos

Hit objects are often grouped together into combos. Each combo set has a unique color and each hit object in a combo has its own combo number that starts from 1 and counts up.

Combos are used to give better visual separation between different patterns. Typically, they consist of 4 to 8 hit objects, but they can be as short as 1 or 2 hit objects or longer than 100 hit objects.

Mappers may add up to 8 unique combo colors. Usually, the colors are cycled one after another, but mappers may also add special combo offsets when starting new combos. This is sometimes done to emphasize different sections of the song. For example, slow parts of the song could use more muted colors, while intense sections of songs could use bright and intense colors.

Players may choose to keep the colors determined by the mapper or override them to their own default color set provided by their skin of the game.

2.2.3 Timing Songs

Technically, hit objects can be arbitrarily placed with millisecond accuracy, but in practice an overarching tempo is determined for the whole song or some sections of the song. The mapper uses their intuition and some basic tools to guess the correct tempo (in beats per minute or "bpm") and an offset of the first downbeat in milliseconds from the start of the song file. A beat length is the length of time between beats. For example, if the tempo of the song is 150 BPM then the beat length is $\frac{1000 \cdot 60}{150} = 400$ milliseconds. Hit objects are typically placed on whole number fractions of the beat length. For example, the most common time signature for popular songs is $\frac{4}{4}$, where each hit object would fall on $\frac{0}{4}$, $\frac{1}{4}$, $\frac{2}{4}$, $\frac{3}{4}$ multiples of the beat length. Many popular songs in rhythm games may have more complicated or changing timing signatures, where, for example, $\frac{1}{4}$ beat timed notes may be mixed with $\frac{1}{3}$ beat timed notes.

2.2.4 Object Placement

Mappers must place each hit object on the play grid by hand. The play grid is set up as a grid with a width of 512 and a height of 384. Some simple tools exist, such as placing objects with constant relative distance or converting a slider into a stream of hit circles. But in general it is very difficult to place all the objects in a way that the map is not repetitive, fun to play, and nice to look at.

2.2.5 Mapping Style

Maps come in many different difficulty levels and styles. Some beatmaps are aimed at beginners and may only feature a hit object once every second or so. While other beatmaps that are meant for seasoned veterans may feature 20 hit objects per second or more.

Maps often come in different mapping styles. For example, some common categories include aim/jump maps that focus on fast movement and the player's ability to accurately position their cursor. Stream maps often feature a lot of hit circles placed closely in a row, or "streams" of hit circles. These types of maps require a lot of tapping stamina to play. Other styles include technical maps, reading maps and others that focus on different gimmicks or aspects of the game.

2.2.6 Difficulty Rating

The difficulty of a beatmap is measured by star rating. The star rating of a map is calculated with an algorithm that considers hit object density, spacing, speed, and many other aspects of the map. The algorithm is often updated by developers to more accurately reflect the difficulty of maps. Star rating is expressed on a scale from 0 - 10, with an exponentially increasing level of difficulty. Some of the easiest maps meant for beginners have a star rating of 2 or lower, while most maps fall within the 2 to 5 star rating range, and out of the millions of players that have played osu!, fewer than 10 players have completed a 10 star map without missing a single circle.

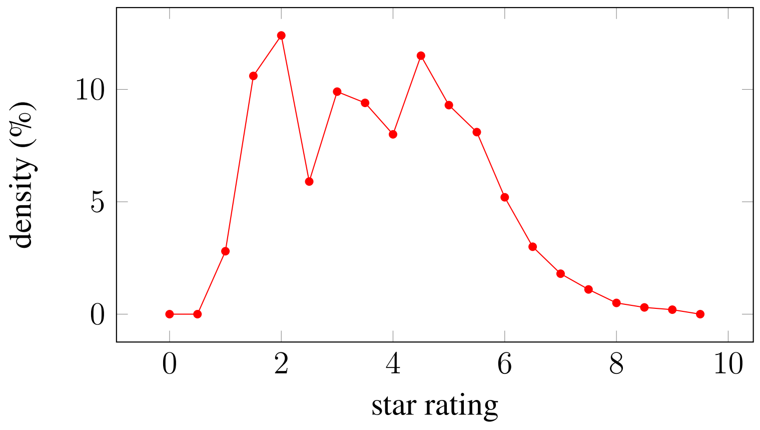

Figure 4. Star rating spread

2.2.7 Map Submission Process

Mappers may freely upload their maps to the osu! beatmap listing on the osu! website. While anyone can submit a beatmap, most maps end up on the unranked list or in the "Graveyard". This means that players cannot see these maps unless they specifically search for them by disabling a couple of filters. The quality of Graveyard maps varies a lot, but usually the average graveyard map is of low quality. If players want to get their map ranked and officially listed on the main beatmap list, they must submit their map for review. Two beatmap nominators must review their map, ask for changes or improvements and then approve the map. Once the map is approved, there is a 1 week waiting list where players can try out the soon-to-be ranked map and if any problems are found and fixed, the week long timer is restarted. Once the map is finally ranked, the state of the map is frozen, no further updates are allowed, and players can earn official score and performance points from that map. This process ensures that a ranked map is generally of high quality, well timed, has proper metadata, and was well liked enough to be approved by 2 professionals. As of 2025 there are 48,000 ranked beatmap sets and over 1 million "Graveyard" beatmap sets.

3 Implementation

3.1 Model Architecture

The model is based on the transformer architecture originally proposed in 2017 in [5].

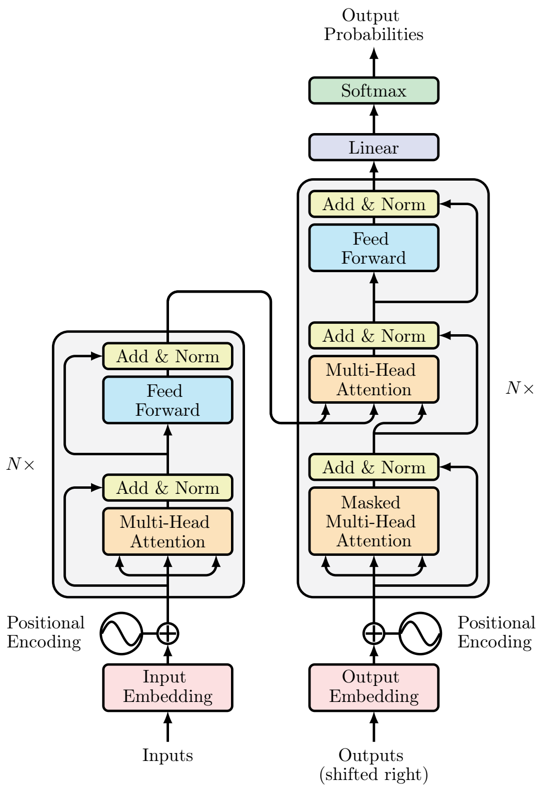

Figure 5. Original transformer architecture

The transformer model is a type of deep learning model designed to process sequential data, like text. The inputs to the model need to be tokenized, or converted into a list of numerical vectors. The model applies several matrix operations and non-linear activation functions to these inputs until a probability distribution is outputted.

Several modifications have been made to this architecture to greatly improve the performance and output quality of the model.

3.1.1 Rotary Positional Encoding

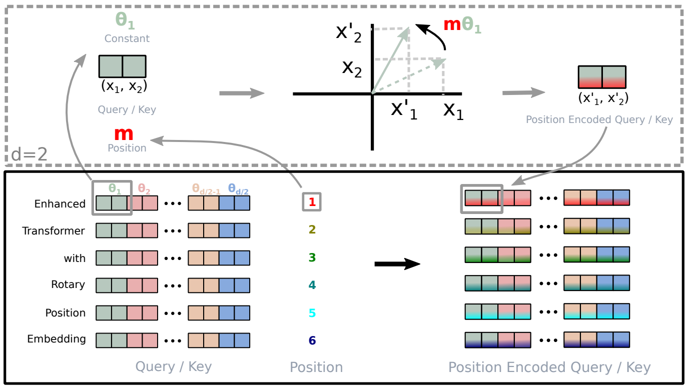

Figure 6. ROPE

The sinusoidal positional encoding used in the 2017 transformer model is replaced with Rotary Positional Encoding (RoPE) originally proposed in [6]. While sinusoidal positional encoding is applied to the embedding, RoPE is applied just before multi-head attention on each of the layers of the transformer. RoPE has proven to encode the relative distance of tokens much better and has been scaled to hundreds of thousands of tokens by models like Llama3 [7]. Although this model does not scale well to very long songs due to how timing information is tokenized, RoPE still helps to generalize to different mapping densities.

3.1.2 RMSNorm

All LayerNorm modules are replaced with RMSNorm which was originally proposed in [8]. RMSNorm offers a more computationally efficient drop-in replacement for LayerNorm while providing similar normalizing properties.

3.1.3 Activation Functions

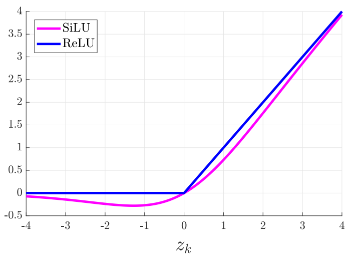

Figure 7. ReLU vs SiLU activation [9]

The activation functions used throughout the model have been replaced by SiLU (Sigmoid-Weighted Linear Units) originally proposed in [9]. This SiLU function is a popular choice in modern LLMs, having been successfully used in models like Llama3 [7].

3.1.4 Conformer Modules

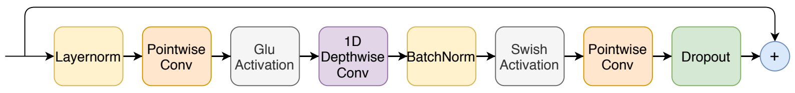

Figure 8. Conformer module [10]

Conformer modules were added to the audio encoder [10]. This greatly increased the model's ability to comprehend the input audio and produced a significant drop in the loss for timing tokens.

Similarly to [10], convolutional modules were introduced as an audio down-sampling layer. The spectrogram of the audio that is one of the inputs of the model has a stride of 10 ms and amplitudes for 80 different frequencies. To represent 20.48 seconds of audio, a total of 2048 audio tokens would be needed. This would be computationally expensive compared to the rest of the model, so a down-sampling layer was introduced to reduce the number of needed audio tokens. Multiple experiments were run on a smaller 50M parameter size model.

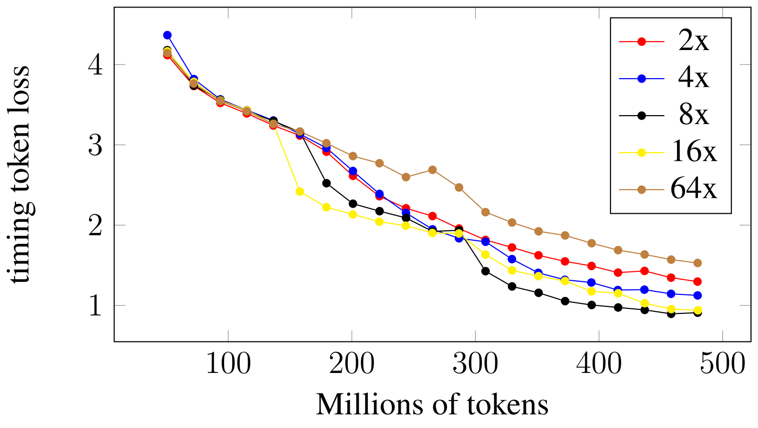

Figure 9. Down-sampling loss

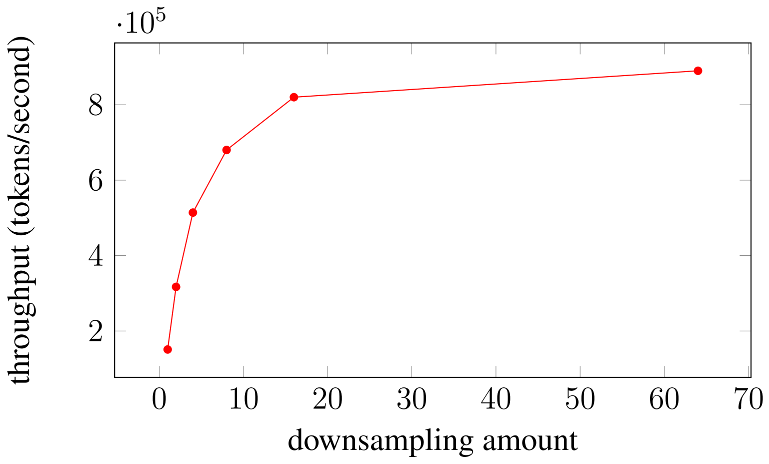

Figure 10. Down-sampling throughput

A down-sampling rate of 8x was chosen for the combination of good loss and throughput. With this setup, each audio token represents 80ms of audio and 256 tokens represent 20.48 seconds of audio.

3.1.5 Flash Attention

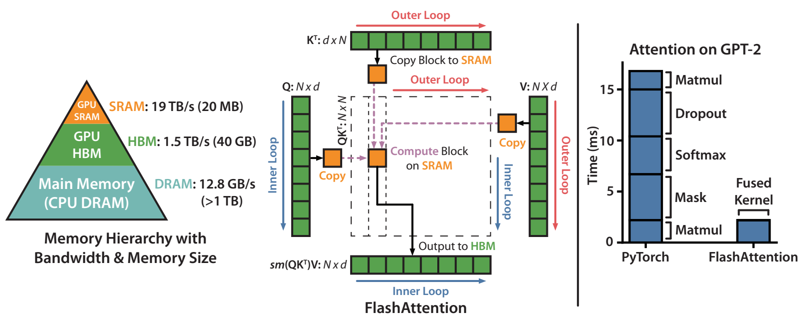

Figure 11. Flash attention [11]

The classic implementation of the attention mechanism was replaced with flash attention originally introduced in [11]. The attention mechanism scales with the square of the context length because every token needs to be able to communicate with all other tokens. Flash attention makes this attention step so efficient that the model scales nearly linearly at smaller (<4000) context sizes, by making the attention step take only a small portion of the computation time needed for a pass through the model.

3.1.6 Mixed Precision Training

Instead of representing all of the model weights as 32-bit floating point (fp32) numbers, most of the model weights were represented as 16-bit bfloat16 (bf16). Bf16 is a modern format found on modern Nvidia GPUs. The Bf16 format keeps the same number of exponent bits as the fp32 numbers, giving it the same dynamic range as fp32, just at a reduced precision. Keeping the same number of exponent bits avoids scaling issues that using fp16 can introduce.

Mixed precision training greatly reduces memory usage and improves throughput, while barely affecting the quality of the model.

3.1.7 Weight Initialization

Model weights were initialized by sampling weights randomly from a normal distribution with a mean of 0 and a standard deviation of $\frac{1}{\sqrt{d}}$ where d is the dimension of the model.

The weights of the residual layers were initialized with a standard deviation of $\frac{1}{\sqrt{N \cdot d}}$where d is the dimension of the model and N is the number of residual layers, similarly to the GPT-2 technical paper [12].

This initialization strategy produced a model that had an initial cross entropy loss really close to $\log(vocabularySize) \approx 8.36$, this means that the model was guessing each token with an equal probability and this served as a good baseline to start training. If the weights are initialized sub-optimally, the initial loss may be several times larger and the model may fail to train entirely.

3.2 Beatmap Tokenization

To feed the beatmap into the model, the beatmap needs to be tokenized, or converted into a long list of tokens.

3.2.1 Hit Circle

Hit circles are encoded with 6 tokens

<|time|> <|hit circle|> <|x coordinate|> <|y coordinate|> <|combo|> <|hit sound|>

These tokens are explained in greater detail later in this section.

3.2.2 Slider

Encoding sliders can take many more tokens

<|time|> <|slider type|> <|x coordinate|> <|y coordinate|> ... <|combo|> <|hit sound|> <|time|> <|repeat|> <|time|> <|slider end|>

The time between the first timing token and the last token determines the total duration of the slider. If the slider repeats, a repeat timing token is encoded. The slider is repeated until the final end timing point is reached. This way, it is possible to encode every possible slider duration and any possible number of repeats.

The <|slider type|> token determines if the slider is a linear slider (2 points), perfect slider (3 points), or a Bézier slider (3+ points).

3.2.3 Timing

A point in time is encoded as a single token denoting the time. A unique token is created to represent each possible time point with a 10 ms accuracy. A spacing of 10 ms was chosen as a trade off between accuracy and vocabulary size. Even in the fastest beatmaps with a tempo of 300 bpm, a 1/12 note lasts 16.66 ms and songs of that speed typically only contain 1/4 or 1/3 notes, so 10 ms should be enough accuracy to accurately represent the vast majority of notes. Veteran osu! players may feel the difference of 10 ms or less when hitting a note, so a more accurate timing for a map can be provided and all hit objects can be snapped to the correct timing with 1 ms accuracy in the decoding process. To represent 20 seconds of a song, approximately 2,000 tokens are required. Unfortunately, this method does not scale very well to very long songs, because as each token is unique, the model must learn the meaning of each token individually and thus may fail to generalize to songs several minutes long.

3.2.4 Coordinates

Multiple strategies for tokenizing coordinates of hit objects were explored.

The simplest strategy is to encode the coordinates as simple x and y tokens. For example, a hit object at the coordinates (423,56) will get encoded as <|x=423|> <|y=56|>. It is also possible to create a token for only the even-numbered coordinates. This would help reduce the total number of tokens, but does in turn lose some fine-grained accuracy.

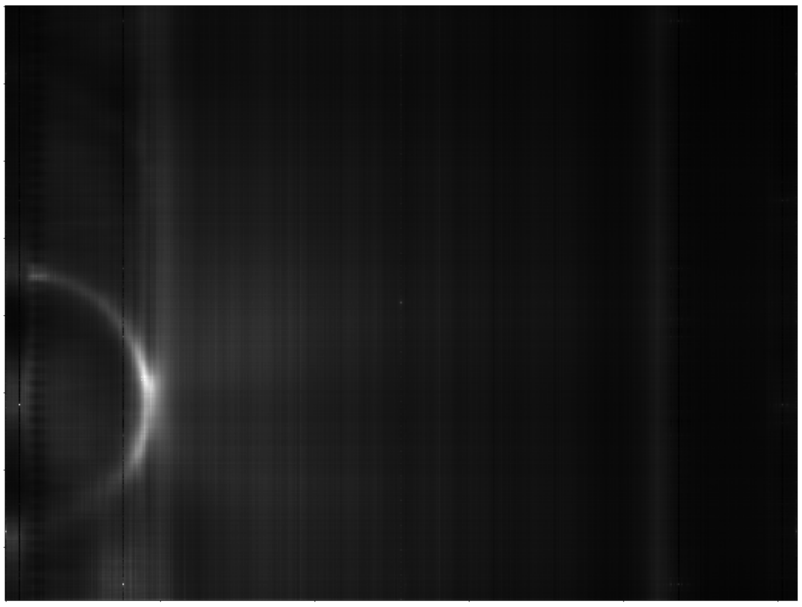

Figure 12. Probability heatmap with x,y strategy

Another strategy is splitting the coordinate grid into squares and creating a token for each square. For example, the grid could be split into 8x8 squares, which would create a total of $\frac{512 \cdot 384}{8 \cdot 8} + \frac{512}{8} + \frac{384}{8} = 3,184$ unique tokens. Choosing too high of an accuracy generates too many tokens, for example if we choose squares of size 1x1 almost 200k tokens are generated. At a model dimension of 1,024 just the embedding table for 200k tokens would take over 200M parameters making it extremely costly in terms of the total model size. Having too many unique tokens also greatly worsens the problem with cross entropy loss for coordinate encoding. In practice, if trained from a randomized starting point the model fails to generalize even a little if the tokens are too sparse.

A strategy to avoid this pitfall would be to keep the accuracy quite low, but include another token specifying a more accurate location inside the square. This way the first token would determine the approximate global position of the hit object and the next token would determine the exact location inside that square.

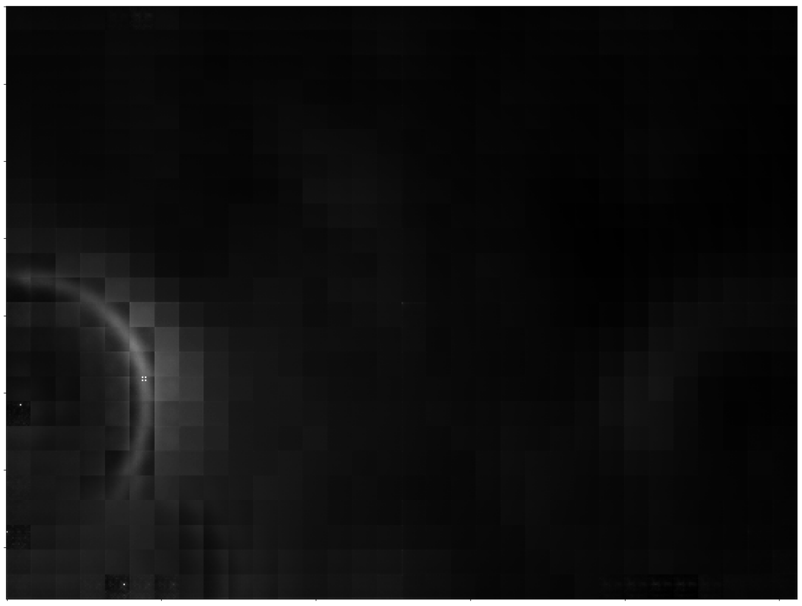

Figure 13. Probability heatmap with grid strategy

Another way to help the model learn with just a single token that determines an hit objects's global position is to first start with super low accuracy, e.g. 64x64. This way, the model first learns the rough global placement of hit objects. During training, the the grid size can then be progressively decreased, helping the model to generalize better. In practice, this strategy yields far better results than just training with high accuracy from the start, but still falls short of the aforementioned strategies.

In conclusion, the simple x and y encoding was chosen for its simplicity and good loss in test training runs.

3.2.5 Hit Sounds

Hit sounds are encoded as a single token representing all the possible combinations of sound sets and hit sounds. In total, there are 3 sets (soft, normal, and drums) and 3 hit sounds for each set (clap, whistle, and finish) making a total of $3 \cdot 2^3 = 24$ possible hit sound tokens.

3.2.6 Combo

Combo colors are encoded as combo tokens <|newcombo,x|> where x (1-8) represents the next combo color to be used.

The colors themselves are encoded in the metadata section of a beatmap. While osu! supports 8-bits per channel for colors, they are down-sampled to 5-bit per channel for a smaller vocabulary size.

3.2.7 Compression

A common strategy for reducing the total length of a tokenized sequence is increasing the vocabulary size by adding several merged tokens that can encode two or more tokens as a single new token. In LLMs, these merged tokens are most commonly found with an algorithm called byte pair encoding, where the most common pairs of tokens are greedily merged together until a target vocabulary size is reached.

This strategy does not produce good results for tokenized beatmaps, because encoded beatmaps have different kinds of patterns that can be taken advantage of, which do not exist in natural languages [13]. Tokens can be rearranged or even omitted without losing any information about the beatmap if done in a smart way.

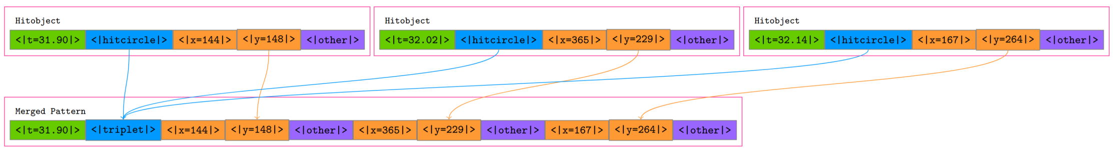
Hitobject
<|t=31.90|> <|hitcircle|> <|x=144|> <|y=148|> <|other|>
Hitobject
<|t=32.02|> <|hitcircle|> <|x=365|> <|y=229|> <|other|>
Hitobject
<|t=32.14|> <|hitcircle|> <|x=167|> <|y=264|> <|other|>
Merged Pattern
<|t=31.90|> <|triplet|> <|x=144|> <|y=148|> <|other|> <|x=365|> <|y=229|> <|other|> <|x=167|> <|y=264|> <|other|>

Figure 14. Merging 3 hit circles into one

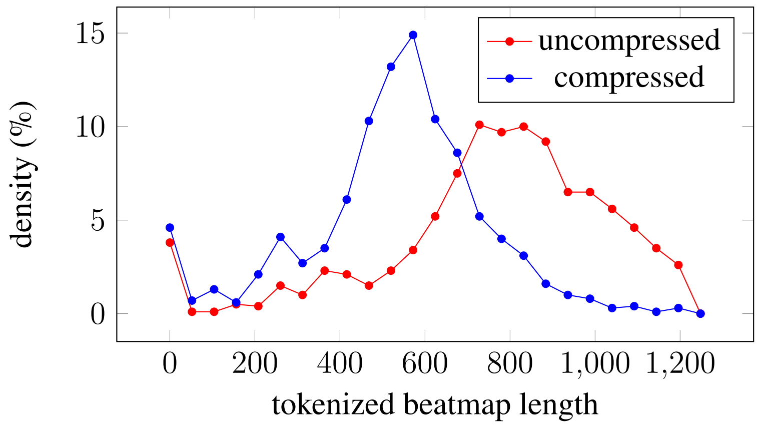

Figure 15. Token compression

A full example of a 20 second tokenized beatmap can be found in Appendix 5.

3.3 Optimizer

For optimizing the model, the adamW optimizer is used with $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 1 \cdot 10^{-6}$ $weight\_decay = 0.1$. Weight decay is selectively applied to weights with the number of dimensions>2 [14]. The global norm of the gradient is clipped at 1.0 during training.

The training rate is linearly increased from 0 to $6 \cdot 10^{-4}$ over the first 1,000 steps of training as a warm up. Then it is kept high until the final 1,000 steps where cosine decay is used for the learning rate until it drops down to 10% of the maximum value similarly to how GPT3 was trained [15].

A batch size of 512 was used, but due to memory limitations, mini-batching was also used, where the final 512 batches of gradients were accumulated over many steps, before doing an optimization step. The exact size of the mini-batch was chosen as the biggest size that would fit on VRAM.

Loss Function

A loss function is used to measure how well the model is predicting the next token. Generally, the lower the loss the better the model is at predicting the next token. When the loss is 0 the model is predicting every token perfectly. For the loss function, cross entropy loss was chosen. This is the most popular choice for modern LLMs. Cross entropy loss can be thought of as measuring the distance between two probability distributions.

While cross entropy loss works great for a large number of mostly independent tokens like in LLMs, in the tokenized beatmap syntax many tokens are closely related to one another. For example when the correct token is <|t=1.00|> but the model guesses <|1.01|>, this is, in the view of the loss function, just as wrong as guessing <|spinner|> or some other completely unrelated token, because this loss considers each token as its own independent class.

Some losses do exist that can consider the relative "distance" to the right guess, for example the Mean Squared Error (MSE) loss. With a loss function like MSE we can calculate how far off the guess was, but a big disadvantage of this is that the model must output just one guess for the correct token instead of a probability distribution for every token. In practice, using MSE for coordinate tokens and cross entropy for everything else did not produce good results, because the placement of objects is extremely hard to predict and inconsistent. Since there is no right way to map in osu! there are usually multiple perfectly reasonable possible coordinates for each hit object. For example, let us consider a hit object that is placed on <|x=100|> in 50% of the difficulties in the dataset and on <|x=200|> for the other half. Cross entropy loss, in a perfect world, would learn to predict exactly 50% for <|x=100|> and 50% for <|x=200|>. But when using a loss like MSE the model would learn to predict the token that gives the least error, which for this case is <|x=150|>, because it is in the exact middle of these tokens, even when the probability in the dataset for this token is 0% and a placement on the coordinate x=150 for that hit object may be completely ridiculous.

For this reason, cross entropy loss was used for every token, even though it is hugely inefficient for coordinates.

3.4 Dataset

For the dataset, all the ranked maps from 2012-present were included. The ranked maps from 2011 and earlier were excluded, because at that time the ranking criteria were not as strict and many maps from that era are mistimed or poorly mapped. From other categories of maps (including Graveyard and Loved maps), all the maps that had at least 100 favorites were included. This filtering excludes maps that have never been played by anyone other than the original mapper and helps to increase the quality of the dataset.

3.4.1 Further Filtering and Cleaning

Once all the maps were downloaded, each one was parsed and additional cleanup processes were employed. Maps with multiple objects appearing at the same time, extremely low or high tempos (less than 0.001 bpm or more than 10000 bpm) were excluded and other indicators of quality were used. Maps with 70% of the song not having any hit objects in a 15 second window were also excluded, because these are often practice difficulties where a small part of the map is copied to a separate difficulty.

All difficulties from all the remaining maps were extracted and grouped by the hash of their audio file.

3.4.2 Final Dataset Statistics

After all the filtering, 26,577 unique audio files remained with 108,043 unique difficulties in total.

The total length of all the unique audio files is around 523 hours, and the total length of all unique beatmaps is 2,129 hours.

When encoded, this amounts to over 350 million tokens of unique high-quality beatmaps.

After converting all audio files to spectrograms, the dataset takes around 200GB of disk space.

For the evaluation dataset, about 2% (total of 512) of the songs were put aside for testing and evaluation.

3.5 Training the Model

For the final model, parameters defined in Table 1 were chosen.

Table 1. Model parameters.

Hyperparameter	**Value**
Layers	12
Model Dimension	768
Attention Heads	12
Head Dimension	64
Batch Size	512
Audio context	1024
Beatmap context	3072
Max learning rate	$6 * 10^{-4}$

The final model has a size of 240 million parameters. It was trained for 48 hours on an Nvidia RTX 4090 running at an average of 420 Watts. The initial training was done with a song context length of 512 which is equivalent to 20.48 seconds of audio and 768 tokens of the encoded beatmap.

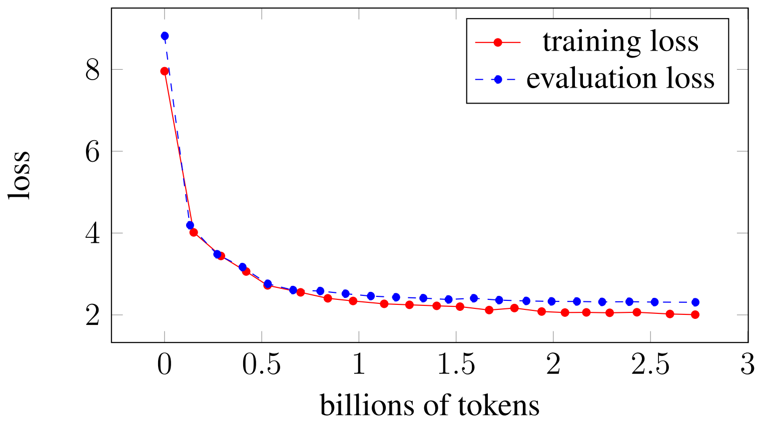

Figure 16. Training run

3.5.1 Context Extending

When the model was sufficiently trained with the initial context size, the context size was doubled and the model was trained until the loss returned to original levels. This process was performed once more so that the final model could produce maps for 80 seconds of audio at a time. This process is similar to one used in the training of Llama3 models [7]. The model mostly just needs to learn representations for the new timing tokens introduced with the longer context size. The model does this very fast because it has already learned most of the patterns from the smaller context sizes.

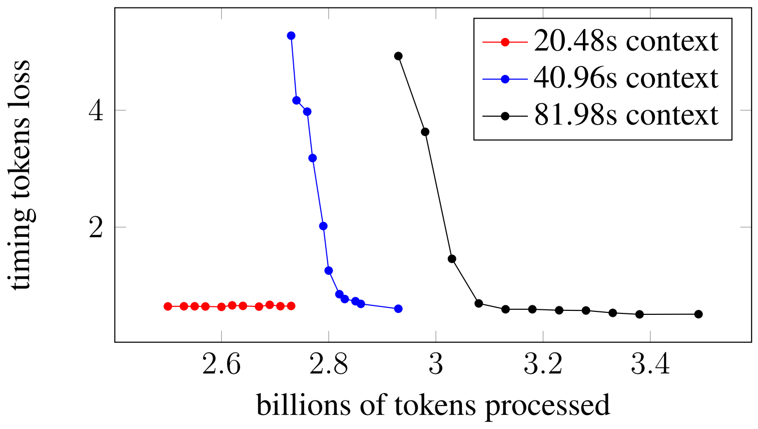

Figure 17. Context extension

4 Results

4.1 Statistical Analysis

To get an idea of how well the model is predicting certain aspects of beatmaps, it is possible to calculate losses and prediction accuracies on just certain types of tokens by masking out all other kinds of tokens.

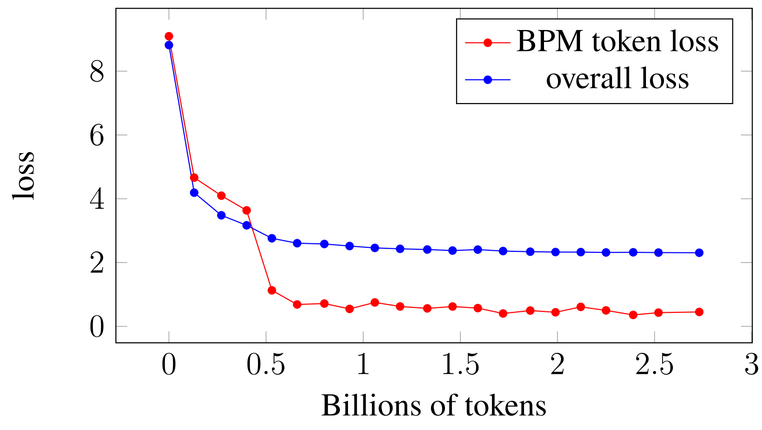

Figure 18. Total vs bpm loss

Figure 18 shows loss over all tokens compared to the loss from only tokens in the form <|bpm=xxx|>. It is really interesting to note that until 500 million tokens the model suddenly "figures out" how to correctly guess the tempo of a song. It is likely that before that moment the model was trying to guess the bpm token based on the distribution found in the training dataset and not actually analyzing the song.

4.1.1 Prediction Accuracy

Accuracy can also be a very interesting statistic to look at, when evaluating a model. The accuracy of a model is measured as the probability that the correct token was the top 1 most probable guess outputted by the model. For example, let's say the correct next token is <|x=56|> denoting that the next x coordinate should be 56. If the model outputs the <|x=56|> as the most probable, it is counted towards being accurate, even if the probability distribution looks something like this:

- <|x=56|> - 10.2%
- <|x=55|> - 8.3%
- <|x=54|> - 5.9%
- ...
- <|hitcircle|> - 0.1%
- ...

In this case the model had very low "confidence" in the guess, but it will still get counted as being "accurate" in this instance, because accuracy measurement does not take into consideration the "confidence" of a model like cross entropy loss does.

4.1.2 Accuracy for Different Token Classes

Table 2. Statistics per token class

class	**loss**	**accuracy**
Timing tokens	0.652	86.9 %
Coordinates	3.745	20.45 %
Slider velocity	2.149	50.45 %
Hit object type	1.683	66.7 %

In Table 2, some different classes of tokens are analyzed separately for accuracy and loss. For example, timing tokens of the form <|t=xx|> have an accuracy 86.9%. This statistic is very high considering that, to accurately predict a time token, a lot of factors need to be taken into account, other than the simple tempo of the map. For example, time tokens encode a lot of the rhythmic and pattern choices made by the mapper. There is no ground truth in mapping, every song can be represented in a number of different ways, but still the model manages to output the correct timing token as it's top 1 guess 86.9% of the time.

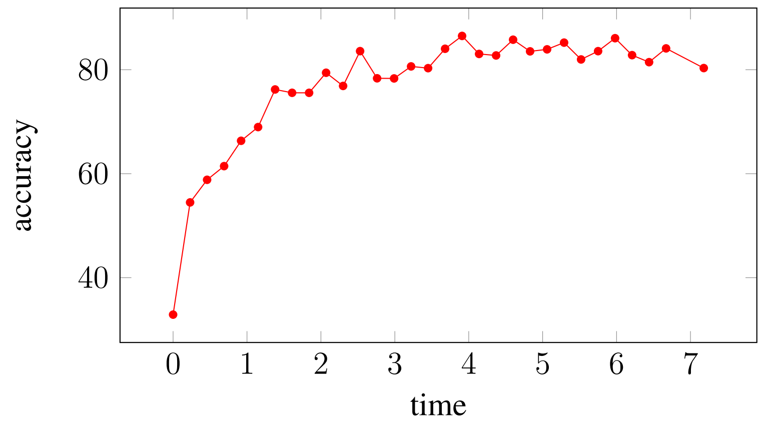

Figure 19. Time token accuracy

On Figure 19, accuracy for timing tokens from <|t=0.00|> to <|t=7.00|> is graphed. The model's accuracy significantly improves after a couple of seconds from the start of the song, where the rhythmic choices of the mappers are more concrete and the model can more confidently predict based on previous tokens and not just the song.

4.2 Attention Maps

4.2.1 Attention for Beatmap Tokens

While most of the weights of the model are really hard to analyze, thanks to the attention mechanism it is possible to see what tokens the model was taking into consideration when predicting the next token.

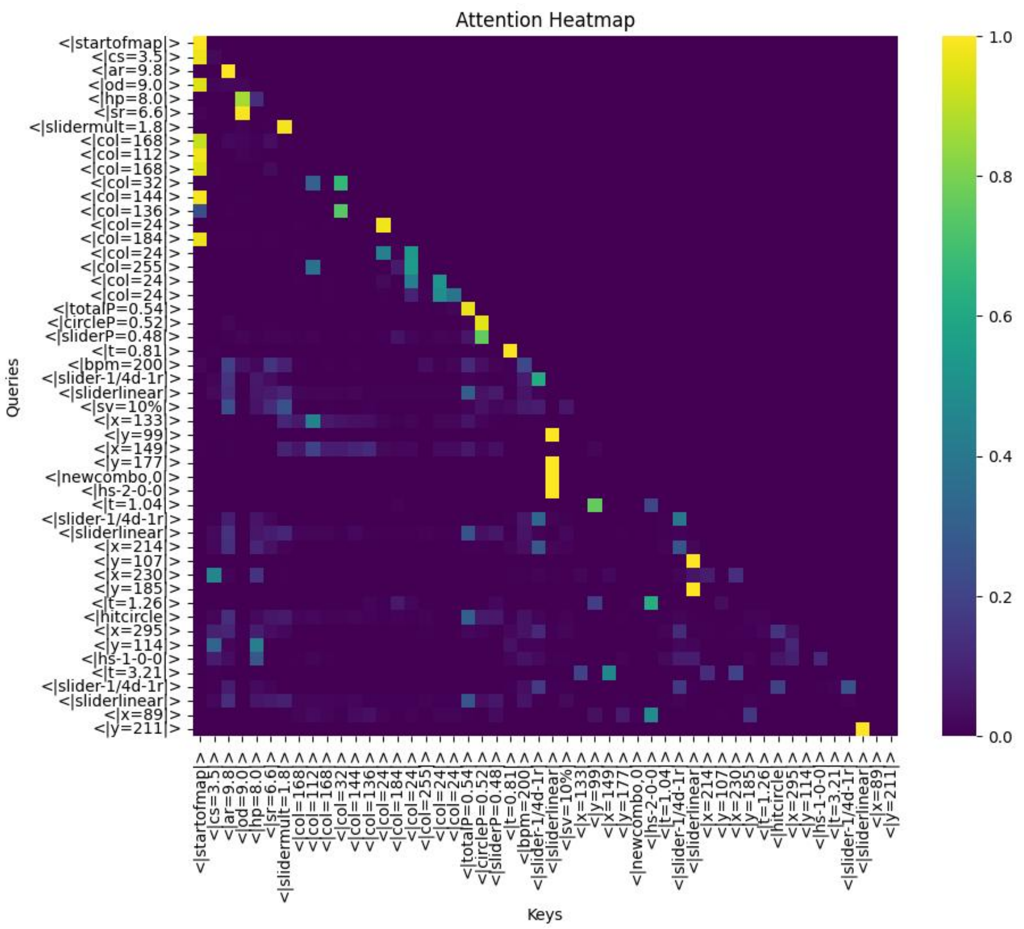

Figure 20. Beatmap attention

Figure 20 depicts the attention mechanism on only one of the heads of the first layer of the model. While most of it is hard to understand, in the exact middle a vertical yellow bar is visible. This bar corresponds to the tokens <|y=99|>, <|y=177|>,<|newcombo|>,<|hs-|> taking attention of the token <|sliderlinear|>. This is exactly what one would expect. To predict the tokens of a slider, the model needs to know what type of slider it is trying to predict. While this one may be looking at the type of the slider, other attention heads may be looking at other hit objects, different timings, metadata and other tokens.

On this attention heatmap, the top right triangle is blank because the model is not allowed to consider future tokens.

4.2.2 Attention for Audio

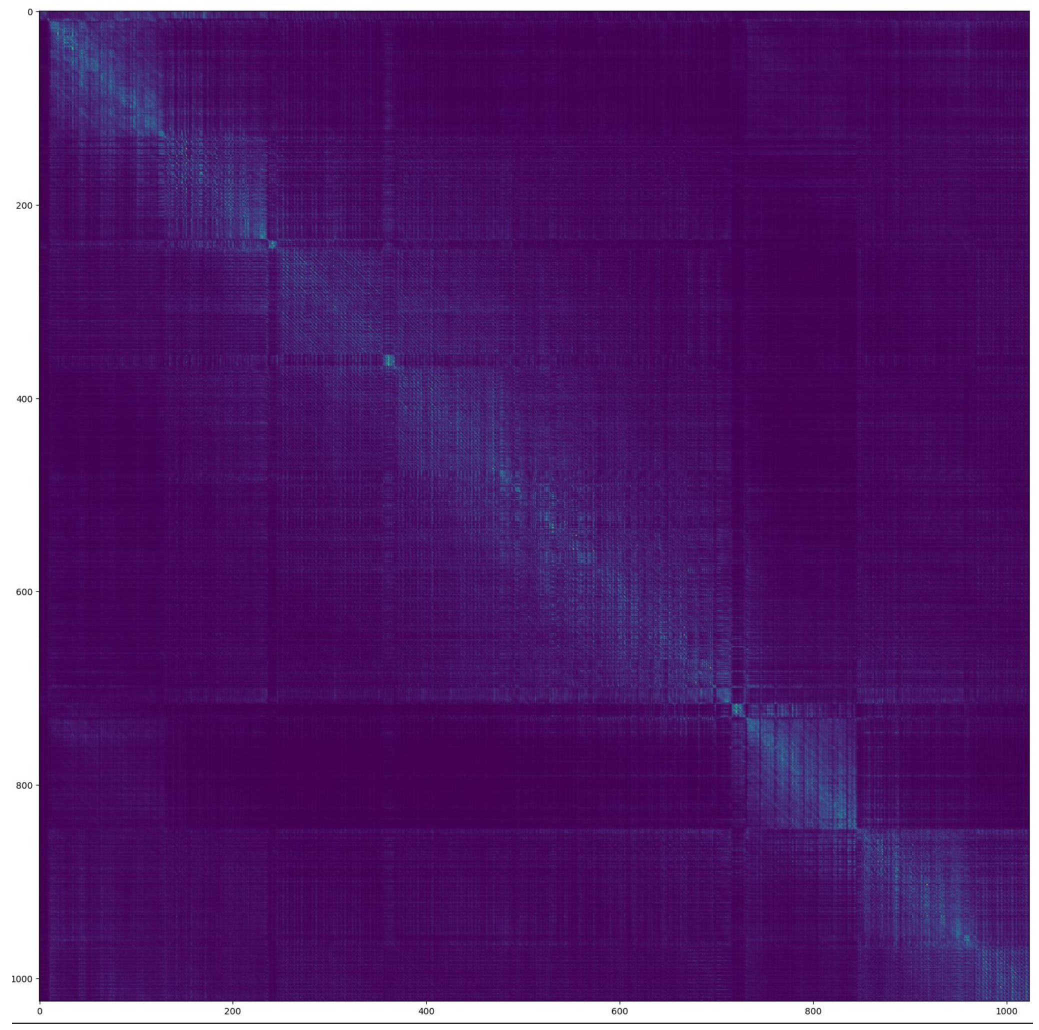

Figure 21. Audio attention

It is much harder to exactly understand what the attention mechanism is doing when analyzing the song in Figure 21. The model seems to be finding different patterns in the audio.

4.2.3 Cross Attention

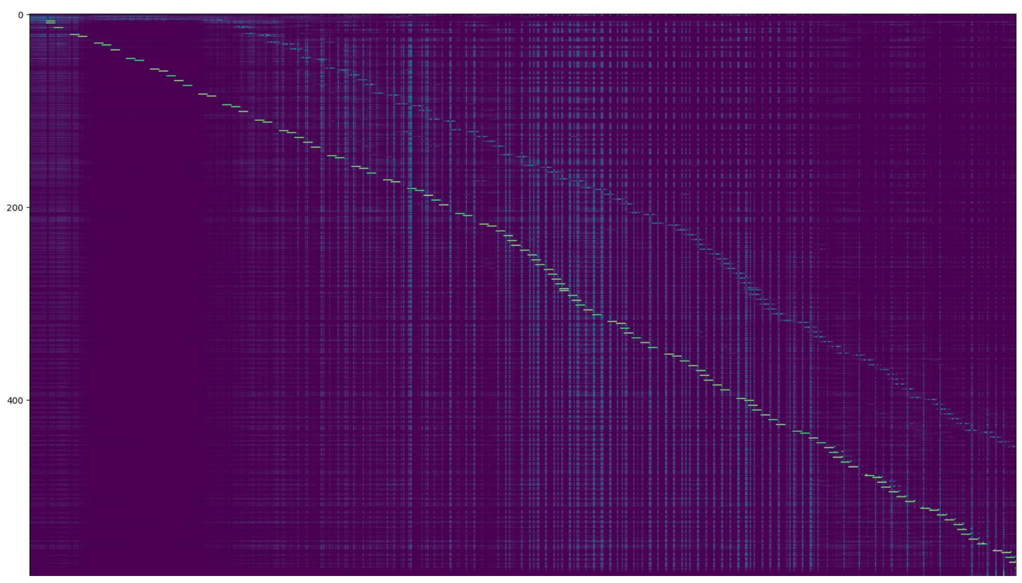

Figure 22. Cross attention

Figure 22 shows what audio tokens were considered when trying to predict a token for the beatmap. The attention heatmap is very noisy but it is possible to make out a very particular curve on the right. The brighter yellow/green line was drawn manually where the model "should" have been attending to if the timing tokens of a beatmap corresponded to the exact right parts of the song. <|t=2.32|> attending to the audio token representing the slice of the song starting at 2,320 ms. The curve on the right and the manually drawn curve on the left are a near match and give confirmation that the model has learned to associate the correct timing token with the correct part of a song without ever being told to do so.

4.3 Generated Beatmaps

4.3.1 Inference

To use the model to generate a beatmap, the user must select a song and set their preferred metadata parameters like circle size and star rating. The song is transformed into a spectrogram and the initial metadata is tokenized and then fed into the model. The model predicts one token at a time. When given the song and the current tokens, the model outputs a probability distribution over all the possible tokens. Multiple strategies were tested for sampling tokens from this probability distribution, some of which are discussed below. When a token is picked from the probability distribution, it is appended to the end of the current tokens and fed through the model again. This process is repeated until an <|endofmap|> token is reached at which point the tokens can be converted back into an .osu file format and played in osu!.

During inference, it is also possible to apply certain algorithms to the outputted tokens to help the model in the generation process. For example, it would be possible to mask out invalid tokens to completely prevent the model from outputting invalid syntax. For example, two timing tokens should never be next to one another, or if a <|spinner|> token is outputted, a <|spinnerend|> token should closely. It is also possible to force the model to output hit objects with correct timing this way - for example if we want the whole beatmap to be in 1/4 timing it is possible to mask out all the other timing tokens to aid the model. Furthermore, due to the encoding process rounding timing points to a 10 ms accuracy it is sometimes impossible for the model to know the exact timing token that is correct.

In this example two beatmaps are encoded with slightly different first beat offsets. From the perspective of the model the timing information given is the same, but the timing of the 2nd beat is inexplicably different.

180bpm offset=0.498 -> <|t=0.50|> <|bpm=180|> <|t=1.17|> <|some hitobject...

180bpm offset=0.504 -> <|t=0.50|> <|bpm=180|> <|t=1.16|> <|some hitobject...

During the inference process, if the user provides timing information with better than 10 ms accuracy, subsequent timing points are corrected to exact millisecond accuracy timings.

Sampling

The simplest strategy for sampling is to pick the token with the highest probability, but this leads to very predictable and boring beatmaps and often leads to model collapse, where for example every hit object could end up on the exact same coordinates. Another strategy is to randomly sample the exact probability distribution outputted from the model. While this leads to better-quality generated beatmaps, this strategy can lead to very unlikely tokens being outputted, because every token has a non-zero probability in the distribution due to the Softmax step used when calculating the probability distribution. This means there is a non-zero chance for the generation process to randomly end because of the <|endofmap|> token being outputted. A strategy to stop very unlikely tokens from being outputted is top-k sampling, where the k top probable tokens are kept and the probability of other tokens is set to zero. k is often chosen to be a number like 50 in general LLMs, but in this use case it is very hard to pick one number to be applied across the sampling process. This is because different parts of the beatmap syntax have wildly different amounts of possible options for each token. For example, when trying to sample a coordinate for an hit object the probability distribution can be very spread out over hundreds of possible coordinate tokens, but when choosing which type of slider should be outputted there are only about 9 reasonable options. A better approach for this use case is to use top-p sampling or "nucleus" sampling [16]. Top-p sampling calculates the cumulative sum of the most probable tokens and keeps the least amount of tokens to exceed the threshold p. For this model the threshold is set to p=0.95. This fits the beatmap syntax much better by dynamically filtering out unlikely tokens and ensures diversity in outputs. This type of sampling also removes the need to use some other masking to prevent the model from outputting incorrect tokens. When using top-p=0.95 the model predicts well enough that it just never outputs an syntactically incorrect token.

Running on a 4090 the 240M size model achieves a generation rate of 20 tokens/s averaging a generation speed 2x faster than real time on the most densely populated beatmaps, and several times faster than real time on less difficult and less dense beatmaps.

4.3.2 Quality of Generated Beatmaps

The maps generated by the model feel natural and fun to play. The model is good at choosing rhythms and patterns that fit the song well. Thanks to it's relatively long context window songs also feel consistent - when the song gets more intense the maps get harder and repeating parts of a song are often mapped similarly, just like in human mapped maps. While the maps are perfectly playable they can feel a bit generic and will not be replacing maps made by professional human mappers.

4.3.3 Weak Points of the Model

Changing Tempo

The model often struggles to correctly time songs with changing tempo.

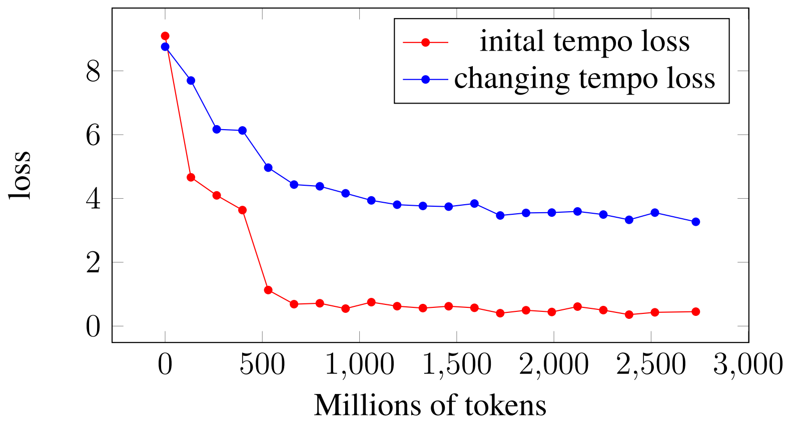

Figure 23. Initial vs changing bpm loss

As seen on Figure 23 the model fails to generalize and predict changing tempo, like it does with guessing the initial tempo. This is likely due to the relatively low density of songs with changing tempo and could probably be fixed with curated or synthetic datasets containing more of these examples.

Coordinates

The weakest part of the model is still predicting the coordinates of hit objects and especially the slider bodies. Due to this the sliders can take on unnatural shapes not often found in human mapped maps. To fix this, a different approach is likely needed that can consider the coordinates as a continuous space and not as a massive list of independent classes like this model does. A post-processing step using something like a diffusion model found in modern image generators is likely to greatly improve the placement of objects and the shapes of sliders.

Mapping Style Selection

Although simple parameters, such as difficulty, are available to guide the generation process, it is not possible to select the exact mapping style the model should output. This aspect of the model is much more difficult to improve upon because of the lack of easily available data on the mapping style of individual beatmaps.

5 Summary

In this work, a model was implemented that accomplishes the initial goal established in the thesis. A user can select a map of their choosing and generate a map for it with very little effort. The model was optimized to run well on modern GPUs and was thoroughly analyzed. The model achieves state-of-the-art performance, but is still not sufficient to completely replace human mappers.

References

[1] Richard Nagyfi. *BeatLearning: Open Source Generative AI Models for Automatic Rhythm Game Beatmap Generation*. https://github.com/sedthh/BeatLearning. Accessed: 2025-05-19. 2025.

[2] osu! community. *Slider*. https://osu.ppy.sh/wiki/en/Game_mode/osu!. Accessed: 2025-05-19. 2025.

[3] osu! community. *Hit circle*. https://osu.ppy.sh/wiki/en/Gameplay/Hit_object/Hit_circle. Accessed: 2025-05-19. 2025.

[4] osu! community. *Slider*. https://osu.ppy.sh/wiki/en/Gameplay/Hit_object/Slider. Accessed: 2025-05-19. 2025.

[5] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[6] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

[7] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[8] Biao Zhang and Rico Sennrich. *Root Mean Square Layer Normalization*. 2019. arXiv: 1910.07467 [cs.LG]. URL: https://arxiv.org/abs/1910.07467.

[9] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. *Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning*. 2017. arXiv: 1702.03118 [cs.LG]. URL: https://arxiv.org/abs/1702.03118.

[10] Anmol Gulati et al. *Conformer: Convolution-augmented Transformer for Speech Recognition*. 2020. arXiv: 2005.08100 [eess.AS]. URL: https://arxiv.org/abs/2005.08100.

[11] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG]. URL: https://arxiv.org/abs/2205.14135.

[12] Alec Radford et al. „Language Models are Unsupervised Multitask Learners“. In: *OpenAI* (2019). URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[13] László Kozma and Johannes Voderholzer. *Theoretical Analysis of Byte-Pair Encoding*. 2024. arXiv: 2411.08671 [cs.DS]. URL: https://arxiv.org/abs/2411.08671.

[14] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101.

[15] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[16] Ari Holtzman et al. *The Curious Case of Neural Text Degeneration*. 2020. arXiv: 1904.09751 [cs.CL]. URL: https://arxiv.org/abs/1904.09751.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Siim Pender

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Generating Maps for Rhythm Games Using Machine Learning”, supervised by Gert Kanter
    1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
    1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright
2. I am aware that the author also retains the rights specified in clause 1 of the nonexclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

06.06.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive licence shall not be valid for the period.

Appendix 2 – Tokenized Beatmap

```
<|startofmap|> <|cs=3.5|> <|ar=9.8|> <|od=9.0|> <|hp=8.0|> <|sr=6.6|> <|slidermult=1.8|> <|col=168|> <|
    col=112|> <|col=168|> <|col=32|> <|col=144|> <|col=136|> <|col=24|> <|col=184|> <|col=24|> <|col
    =255|> <|col=24|> <|col=24|> <|totalP=0.54|> <|circleP=0.52|> <|sliderP=0.48|>
<|t=0.81|> <|bpm=200|> <|slider—1/4d—1r|> <|sliderlinear|> <|sv=10%|> <|x=133|> <|y=99|> <|x=149|>
    <|y=177|> <|newcombo,0|> <|hs—2—0—0|>
<|t=1.04|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=214|> <|y=107|> <|x=230|> <|y=185|>
<|t=1.26|> <|hitcircle|> <|x=295|> <|y=114|> <|hs—1—0—0|>
<|t=3.21|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=89|> <|y=211|> <|x=105|> <|y=289|> <|newcombo,0|>
    <|hs—2—0—0|>
<|t=3.44|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=170|> <|y=219|> <|x=186|> <|y=297|>
<|t=3.66|> <|hitcircle|> <|x=251|> <|y=226|> <|hs—1—0—0|>
<|t=5.01|> <|sliderlinear|> <|sv=20%|> <|x=34|> <|y=311|> <|x=50|> <|y=384|> <|newcombo,2|> <|hs
    —2—0—0|>
<|t=5.46|> <|sliderend|>
<|t=5.61|> <|slider—1/4d—1r|> <|sliderlinear|> <|sv=100%|> <|x=376|> <|y=106|> <|x=360|> <|y=184|> <|
    newcombo,0|>
<|t=5.84|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=295|> <|y=114|> <|x=279|> <|y=192|>
<|t=6.06|> <|hitcircle|> <|x=214|> <|y=121|> <|hs—1—0—0|>
<|t=8.01|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=420|> <|y=218|> <|x=404|> <|y=296|> <|newcombo,2|>
     <|hs—2—0—0|>
<|t=8.24|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=339|> <|y=226|> <|x=323|> <|y=304|>
<|t=8.46|> <|hitcircle|> <|x=258|> <|y=233|> <|hs—1—0—0|>
<|t=9.81|> <|sliderlinear|> <|sv=20%|> <|x=475|> <|y=318|> <|x=459|> <|y=384|> <|newcombo,0|> <|hs
    —2—0—0|>
<|t=10.26|> <|sliderend|>
<|t=10.41|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|1/4|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|
    sliderperfect|> <|sv=180%|> <|x=503|> <|y=233|> <|x=465|> <|y=231|> <|x=379|> <|y=286|> <|
    newcombo,2|> <|sliderperfect|> <|x=346|> <|y=292|> <|x=308|> <|y=303|> <|x=269|> <|y=301|> <|
    sliderlinear|> <|sv=50%|> <|x=158|> <|y=250|> <|x=166|> <|y=227|> <|newcombo,0|> <|sliderlinear|>
     <|x=269|> <|y=301|> <|x=261|> <|y=324|>
<|t=11.01|> <|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|hitcircle|> <|x=363|> <|y=182|> <|newcombo,2|> <|x
    =57|> <|y=245|> <|hs—1—0—0|> <|x=384|> <|y=174|> <|hs—2—2—8|> <|x=181|> <|y=134|> <|hs
    —2—0—0|>
```

```
<|t=11.61|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderlinear|> <|sv=120%|> <|x=447|> <|y=63|> <|x
    =464|> <|y=168|> <|newcombo,0|> <|sliderlinear|> <|x=359|> <|y=35|> <|x=376|> <|y=140|>
<|t=11.91|> <|hitcircle|> <|x=512|> <|y=295|> <|hs—2—2—8|>
<|t=12.06|> <|hitcircle|1/4|hitcircle|1/4|hitcircle|> <|x=257|> <|y=368|> <|newcombo,2|> <|hs—2—0—0|> <|
    x=313|> <|y=375|> <|hs—2—2—8|> <|x=369|> <|y=382|> <|hs—2—0—0|>
<|t=12.36|> <|hitcircle|> <|x=512|> <|y=295|> <|hs—1—0—0|>
<|t=12.51|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|1/4|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderbezier
    —4|> <|sv=180%|> <|x=275|> <|y=172|> <|x=236|> <|y=167|> <|sharp|> <|x=150|> <|y=195|> <|
    newcombo,0|> <|hs—2—0—0|> <|sliderbezier—4|> <|x=136|> <|y=276|> <|x=175|> <|y=281|> <|sharp
    |> <|x=261|> <|y=253|> <|sliderlinear|> <|sv=120%|> <|x=127|> <|y=117|> <|x=110|> <|y=222|> <|
    newcombo,2|> <|sliderlinear|> <|x=43|> <|y=112|> <|x=26|> <|y=217|>
<|t=13.11|> <|slider—3/4d—1r|> <|sliderperfect|> <|sv=180%|> <|x=213|> <|y=93|> <|x=224|> <|y=50|> <|
    x=89|> <|y=70|> <|newcombo,1|>
<|t=13.41|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|1/4|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderlinear
    |> <|sv=80%|> <|x=67|> <|y=274|> <|x=72|> <|y=240|> <|newcombo,2|> <|sliderlinear|> <|x=142|> <|
    y=324|> <|x=152|> <|y=267|> <|newcombo,2|> <|sliderlinear|> <|sv=100%|> <|x=247|> <|y=192|> <|
    x=238|> <|y=271|> <|newcombo,0|> <|sliderbezier—4|> <|sv=180%|> <|x=313|> <|y=384|> <|x
    =337|> <|y=353|> <|sharp|> <|x=434|> <|y=377|> <|newcombo,2|>
<|t=14.01|> <|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|hitcircle|> <|x=449|> <|y=309|> <|newcombo,0|> <|hs
    —1—0—0|> <|x=228|> <|y=319|> <|hs—2—0—0|> <|x=449|> <|y=309|> <|newcombo,2|> <|hs
    —2—2—8|> <|x=228|> <|y=319|> <|hs—1—0—0|>
<|t=14.61|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderperfect|> <|sv=120%|> <|x=422|> <|y=158|>
    <|x=439|> <|y=207|> <|x=416|> <|y=259|> <|newcombo,0|> <|hs—2—0—0|> <|sliderperfect|> <|x
    =375|> <|y=362|> <|x=358|> <|y=313|> <|x=381|> <|y=261|><|t=14.91|> <|slider—1/2d—1r|1/2|slider
    —1/2d—1r|> <|sliderbezier—6|> <|sv=180%|> <|x=247|> <|y=192|> <|x=201|> <|y=185|> <|sharp|> <|
    x=147|> <|y=214|> <|sharp|> <|x=79|> <|y=201|> <|newcombo,2|> <|sliderperfect|> <|x=499|> <|y
    =117|> <|x=442|> <|y=72|> <|x=340|> <|y=127|> <|newcombo,1|>
<|t=15.51|> <|hitcircle|1/2|hitcircle|1/2|hitcircle|> <|x=281|> <|y=272|> <|newcombo,1|> <|hs—1—0—0|> <|
    x=212|> <|y=82|> <|hs—2—2—2|> <|x=159|> <|y=337|> <|hs—1—0—0|>
<|t=15.96|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderlinear|> <|sv=120%|> <|x=212|> <|y=82|> <|x
    =193|> <|y=183|> <|newcombo,0|> <|hs—2—0—0|> <|sliderlinear|> <|x=270|> <|y=216|> <|x=289|>
    <|y=317|>
<|t=16.26|> <|slider—1/4d—1r|> <|sliderlinear|> <|x=75|> <|y=322|> <|x=56|> <|y=384|>
<|t=16.41|> <|slider—1/2d—1r|> <|sliderbezier—x|> <|sv=180%|> <|x=0|> <|y=137|> <|x=24|> <|y=108|>
    <|x=41|> <|y=97|> <|x=73|> <|y=96|> <|sharp|> <|x=94|> <|y=125|> <|sharp|> <|x=118|> <|y=102|>
    <|x=147|> <|y=97|> <|x=159|> <|y=126|> <|newcombo,0|>
<|t=16.71|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderbezier—4|> <|sv=120%|> <|x=323|> <|y
    =217|> <|x=295|> <|y=225|> <|sharp|> <|x=251|> <|y=208|> <|newcombo,1|> <|sliderbezier—4|> <|x
    =444|> <|y=164|> <|x=418|> <|y=154|> <|sharp|> <|x=390|>
```

```
<|y=162|>
<|t=17.01|> <|hitcircle|> <|x=496|> <|y=304|> <|newcombo,0|> <|hs—1—1—2|>
<|t=17.16|> <|hitcircle|1/4|hitcircle|1/4|hitcircle|> <|x=338|> <|y=60|> <|newcombo,2|> <|hs—1—0—0|> <|x
    =287|> <|y=55|> <|x=236|> <|y=51|> <|hs—1—1—2|>
<|t=17.46|> <|hitcircle|1/4|hitcircle|1/4|slider—1/4d—1r|> <|x=82|> <|y=305|> <|newcombo,0|> <|hs
    —1—0—0|> <|x=133|> <|y=300|> <|sliderperfect|> <|sv=100%|> <|x=184|> <|y=296|> <|x=196|> <|y
    =270|> <|x=190|> <|y=243|> <|newcombo,2|> <|hs—2—0—0|>
<|t=17.76|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderperfect|> <|x=289|> <|y=218|> <|x=280|> <|y
    =190|> <|x=258|> <|y=173|> <|sliderbezier—4|> <|sv=120%|> <|x=372|> <|y=331|> <|x=384|> <|y
    =359|> <|sharp|> <|x=369|> <|y=384|> <|newcombo,0|>
<|t=18.06|> <|slider—1/4d—1r|> <|sliderbezier—4|> <|x=461|> <|y=253|> <|x=449|> <|y=281|> <|sharp|>
    <|x=464|> <|y=310|>
<|t=18.21|> <|hitcircle|1/4|hitcircle|1/4|hitcircle|1/4|hitcircle|1/4|hitcircle|1/4|hitcircle|1/4|hitcircle|1/4|
    hitcircle|> <|x=380|> <|y=210|> <|newcombo,0|> <|hs—1—0—0|> <|x=388|> <|y=202|> <|x=396|> <|y
    =195|> <|x=405|> <|y=188|> <|x=414|> <|y=181|> <|x=424|> <|y=176|> <|x=434|> <|y=171|> <|x
    =444|> <|y=167|>
<|t=18.81|> <|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|hitcircle|1/2|
    hitcircle|> <|x=454|> <|y=164|> <|hs—1—2—2|> <|x=79|> <|y=82|> <|newcombo,0|> <|hs—2—2—2|>
    <|x=148|> <|y=272|> <|x=187|> <|y=154|> <|x=43|> <|y=227|> <|newcombo,2|> <|hs—2—3—8|> <|x
    =380|> <|y=210|> <|hs—2—3—4|> <|x=43|> <|y=227|> <|newcombo,0|> <|hs—1—0—0|> <|x=380|> <|
    y=210|>
<|t=20.01|> <|slider—3/4d—1r|> <|sliderperfect|> <|sv=100%|> <|x=207|> <|y=100|> <|x=185|> <|y=155|>
    <|x=140|> <|y=115|> <|newcombo,2|> <|hs—2—0—0|>
<|t=20.31|> <|slider—1/4d—1r|1/4|slider—1/4d—1r|> <|sliderperfect|> <|sv=120%|> <|x=155|> <|y=45|> <|
    x=124|> <|y=45|> <|x=78|> <|y=73|> <|newcombo,0|> <|sliderperfect|> <|x=43|> <|y=227|> <|x=74|>
    <|y=227|> <|x=120|> <|y=199|> <|endofmap|>
```